Última atualização em: 27/06/2023

Notas de aula

MAC 0329
Álgebra booleana e aplicações

Nina S. T. Hirata

Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

São Paulo, 2023

Índice

Capítulo 1

Introdução

Quando interagimos com um computador ou com o celular, utilizamos algum tipo de aplicativo. Por meio dos aplicativos é que conseguimos, por exemplo, ouvir músicas, assistir a um filme, fazer cálculos, conversar no chat com um amigo, e assim por diante. Quanto mais diversificados forem os aplicativos instalados, maior é a variedade de usos possíveis. Não é incrível que um único equipamento, computador ou celular, possa ser usado para fazer tantas coisas ?

Diferentemente de outros equipamentos que em geral são feitos para serem usados de forma específica, os computadores diferenciam-se por serem programáveis. Por meio da programação, podemos alterar ou ampliar o leque de funcionalidades do computador.

No estudo de computadores e computação, em geral fazemos a distinção entre *software* e *hardware*. Este último, *hardware*, refere-se à parte física do computador. Já o *software* refere-se à parte lógica que abarca, por exemplo, os programas de computador e o controle da execução desses programas. Os aplicativos são programas de computador.

Para entendermos como um computador é organizado e como ele funciona, é importante notarmos que eles são essencialmente máquinas para processamento de dados. Em termos de *hardware*, os componentes básicos de armazenamento são dispositivos físicos que conseguem representar e alterar entre dois estados, desligado e ligado. A esses dois estados associa-se os números 0 e 1, respectivamente. Portanto, os dados armazenados e manipulados por um computador são representadas por um bloco de 0s e 1s. Por conta disso, comumente ouvimos o termo "binário" quando o assunto é computação. O *hardware* do computador é projetado para armazenar dados e executar um conjunto de instruções; instruções essas que consistem de operações básicas como comparações e operações aritméticas, ou operação de cópia de dados de um local de armazenamento para outro, denominadas instruções de máquina.

Um programa de computador nada mais é do que uma sequência de instruções. O programador,

aquele que cria programas de computador, poderia especificar a sequência de instruções que o *hardware* irá executar. Porém, escrever um programa usando as "instruções da máquina" – muitas vezes chamada de programação baixo-nível – não é uma tarefa simples (isto ficará claro mais adiante). Na prática, programadores escrevem programas utilizando linguagens de programação de alto-nível tais como C, Java, Python, etc. Essas linguagens de alto-nível são instruções expressas em uma linguagem mais próxima à linguagem natural, diferentemente da instrução de máquina que são códigos. Desta forma, programas escritos em linguagem de alto nível precisam ser "traduzidos" para um pragrama em linguagem de máquina (isto é, uma sequência de instruções de máquina). Esse processo de "tradução" é chamado de compilação ou interpretação. Na compilação, um outro programa denominado programa executável ou programa binário é gerado e é esse programa que é de fato "executado" (i.e., colocado para que a sequência de instruções seja executada pelo *hardware*). Na interpretação, as instruções de máquina são geradas e enviadas para execução uma a uma. Não importa se compiladas ou interpretadas, os programas escritos em linguagem de programação são, portanto, convertidos para uma sequência de instruções de máquina que são as executadas de fato.

O que denominamos organização de computadores neste texto refere-se ao modelo de computação implementado no *hardware*. O estudo de organização de computadores envolve a representação dos dados e instruções de máquina, os componentes que compõem o modelo de computação, e a forma como as instruções são executadas no *hardware*. Um modelo de computação simples que é a base de muitos computadores é a arquitetura Von Neumann (ver figura 1.1). Além de

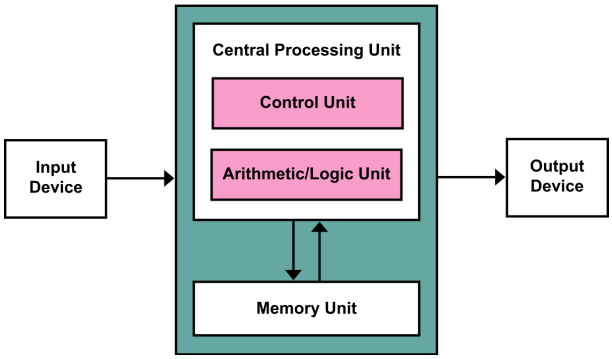


Figura 1.1: Arquitetura Von Neumann (fonte: *Wikipedia*).

dispositivos de entrada e de saída, o modelo de Von Newmann é composto por uma unidade de memória, e uma unidade de processamento central (CPU), que inclui uma unidade lógico-aritmética (ULA) e uma unidade de controle (UC). A memória é usada para o armazenamento de dados ou de instruções. A unidade lógico-aritmética é a parte do *hardware* que realiza as operações lógicas e aritméticas. Os dados operados pela ULA geralmente encontram-se na memória e os resultados da operação precisam ser armazenados também na memória. A UC é a parte do *hardware* que controla a execução das instruções, incluindo o fluxo de dados entre os componentes, por exemplo entre a memória e a ULA. Para um programa ser executado no modelo Von Neumann, suas instruções de máquina devem ser armazenadas na memória e então

executadas uma a uma. A UC "sabe" qual é a próxima instrução a ser executada. Cada ciclo de execução de uma instrução consiste em buscar a instrução, decodificá-la e executá-la. Ao final do curso, espera-se atingir uma boa compreensão sobre como funciona um processador básico.

Além do compilador mencionado acima, outro programa importante é o chamado **sistema operacional** (SO). Ao se ligar um computador, o SO é automaticamente colocado em execução. Quando um usuário inicia o uso de um aplicativo, cabe ao SO colocar as instruções de máquina correspondentes a esse aplicativo na memória do computador e indicar para a CPU onde se encontra a próxima instrução a ser executada. Além disso, o SO gerencia o uso de recursos do computador (tais como espaço em memória e tempo de CPU) de forma que vários aplicativos possam estar em execução "simultânea". Cabe também a ele administrar interrupções (sinais que vêm de fora, tais como o do teclado, o do mouse, ou da rede, etc), ou capturar uma indicação de erro de execução e transmitir ao aplicativo.

Estas notas de aula reúnem conceitos e fundamentos importantes para se entender como ocorre a execução de um programa em um computador, especificamente com respeito à parte relacionada à execução das instruções de máquina. Note porém que não estamos interessados no aspecto físico-elétrico (*hardware*) envolvido nesse processo; estamos interessados apenas no aspecto lógico (ainda que no nível de *hardware*). Note também que entre "escrever" um programa em uma linguagem de programação, gerar a correspondente sequência de instruções de máquina, e colocar essa sequência na memória do computador para que as instruções possam ser executadas, há uma vasta gama de conceitos e conhecimentos envolvidos que estão além do escopo desta disciplina. Vários desses conceitos e conhecimentos serão vistos em outras disciplinas ao longo do curso. Para termos uma visão integrada sobre como os dados e instruções são codificados e como os dados são transferidos e transformados pelo processador, utilizaremos um modelo de processador e um conjunto de instruções de máquina bastante simplificados que estão bem distantes dos processadores comerciais atuais. Mesmo assim, espera-se que eles serão suficientes para a compreensão dos conceitos importantes.

O conteúdo está organizado da seguinte forma [1]

- Começamos com a representação de números em diferentes bases, especialmente a base 2 (binária). Em seguida examinamos a representação de informação no computador, e em especial números inteiros sem e com sinal. A representação leva em conta o fato de computadores serem máquinas "finitas".
- Funções lógicas e circuitos digitais: como computadores são essencialmente processadores de dados, estudaremos a adição e subtração de números inteiros no computador como um exemplo que permitirá entendermos as funções lógicas e respectivos circuitos digitais. Especificamente veremos como construir um circuito somador de *bits* (contração de *binary*

[1] O texto poderá ser atualizado durante o semestre e essa organização poderá sofrer alterações.

digit).

- Álgebra booleana: é a fundamentação matemática das funções lógicas; as que definem o mapeamento dos dados de entrada para os dados de saída (segundo algum processamento desejado). Os circuitos digitais nada mais são do que implementações dessas funções lógicas. Por meio de manipulação algébrica, conseguiremos por exemplo encontrar circuitos "menores".

- Circuitos combinacionais: estudaremos alguns circuitos específicos que podem ser utilizados como componentes, uma peça que está pronta para ser usada, na construção de circuitos complexos.

- Circuitos sequenciais: estes referem-se a circuitos cuja saída depende não apenas da entrada, mas também do "estado atual". Alguns exemplos de circuitos sequenciais são memória, registrador, contador. Associado aos circuitos sequenciais veremos também a noção de *clock* e de máquina de estados.

- Organização de computadores: como os componentes combinacionais (ULA), sequenciais (memória) e de controle interagem; como funciona a execução de uma sequência de instruções de máquina.

Uma sugestão de leitura (leve) para saber um pouco mais sobre vários dos termos mencionados acima (programa, instruções, binário, *hardware*, *software*, circuito lógico, etc) e como esses elementos se entrelaçam é o livro abaixo. O livro apresenta também um panorama histórico da computação, desde a idade da pedra até os computadores de mesa, e é uma leitura interessante e divertida sobre o básico de computação e computadores.

Capítulo 2

Representação de números

Última atualização em 29/04/2021

Computadores, em termos de *hardware*, são capazes de armazenar e manipular dois valores, o 0 e o 1. Na prática, os valores 0 e 1 são níveis de voltagens distintos que podem ser representados e alterados de um para outro de forma fisicamente robusta. Sem entrar em demais detalhes, podemos dizer que a tecnologia atual é capaz de representar eletronicamente apenas dois estados, o desligado e ligado, ou 0 e 1, ou OFF e ON, ou Falso e Verdadeiro. Uma unidade (componente) dessas que armazena os valores 0 ou 1 é denominado *bit*, uma contração de *binary digit*. Qualquer informação a ser armazenada e/ou manipulada no computador precisa ser representada em termos dessas unidades. O termo dado binário, frequentemente utilizado associado a computadores, está relacionado a esse fato. Tanto dados quanto as intruções de máquina são, portanto, codificadas em binário. Desta forma, para entendermos como um programa é executado e como os dados são processados, precisamos entender a representação binária e como dados representados em binário são operados.

Neste capítulo iremos inicialmente entender como os números são representados em binário. Para tanto, recordaremos os sistemas de representação numérica, incluindo a representação na base dez com a qual estamos habituados. Também iremos examinar como converter a representação dada em uma certa base para uma outra base. Uma vez familiarizado com a representação em binário, podemos começar a examinar como os diferentes tipos de dados (tais como números inteiros com e sem sinal, números reais, ou ainda caracteres) são representados em computador. Um ponto importante a ser considerado é o fato de computadores serem máquinas finitas, ou seja, sua capacidade de armazenamento é finita. Examinaremos em maior detalhe a representação de números inteiros e como eles são operados (isto é, adicionados ou subtraídos), considerando-se que o número de *bits* usados para armazenar um número é fixo e limitado.

2.1 Sistemas de representação numérica

Números são conceitos que nos são familiares desde a infância. Temos os números naturais, os números inteiros, os números reais, e outros. No dia a dia, denotamos números usando combinações de dígitos de um conjunto de 10 dígitos: 0, 1, 2, 3, 4, 5, 6, 7, 8, e 9.

Ao escrevermos 9053 temos uma noção da ordem de grandeza desse número. Também sabemos que 9053 é maior que 5651, por exemplo. Essa notação, na qual dígitos são arranjados um ao lado do outro é conhecida como a **notação posicional**. A posição do dígito no arranjo sequencial indica o seu valor no número. Por exemplo, em 32 o dígito 2 vale dois, enquanto em 1207 ele vale duzentos. Essa noção de os valores de um mesmo dígito variarem de acordo com a posição ocupada por ele fica explicitamente indicada na chamada **notação polinomial**. Por exemplo, usando $*$ para denotar a operação de multiplicação, o número 9053 pode ser escrito na notação polinomial como:

$$9 * 10^3 + 0 * 10^2 + 5 * 10^1 + 3 * 10^0 \tag{2.1}$$

Note que os coeficientes 9, 0, 5, e 3 são os dígitos na notação posicional, e o expoente corresponde à posição do respectivo dígito ao enumerarmos as posições da direita para a esquerda, contando a partir de zero. O número 10 que aparece nos termos potência é denominado de **base** do sistema de representação numérica.

2.1.1 Representação em uma base b

Seja $b > 1$, inteiro. Considere também o conjunto de dígitos $D = \{0, 1, 2, \ldots, b-1\}$. Um número qualquer x pode ser expresso na base b, na notação posicional, como

$$d_{n-1}\, d_{n-2} \cdots d_1\, d_0\, .\, d_{-1}\, d_{-2} \cdots d_{-m}$$

tal que n é o número de dígitos em sua parte inteira, m é o número de dígitos em sua parte fracionária, e $d_i \in D, \forall i$. O dígito d_{n-1} é denominado **dígito mais significativo** e o dígito d_{-m} é denominado o **dígito menos significativo**. Quando $m = 0$, o número é inteiro e o dígito menos significativo é o d_0.

Na notação polinomial, omitindo-se o sinal das operações de multiplicação, esse número é expresso como

$$d_{n-1}\, b^{n-1} + d_{n-2}\, b^{n-2} + \cdots + d_1\, b^1 + d_0\, b^0 + d_{-1}\, b^{-1} + d_{-2}\, b^{-2} + \cdots + d_{-m}\, b^{-m}$$

Em boa parte deste texto trataremos apenas números inteiros. No exemplo visto anteriormente (expressão 2.1), foi mostrada a notação polinomial correspondente a 9053, na base $b = 10$. Se considerarmos a notação 101 na base 10, temos $1 * 10^2 + 0 * 10^1 + 1 * 10^0$ (o valor cento e um). Porém, se considerarmos a base $b = 2$, temos $1 * 2^2 + 0 * 2^1 + 1 * 2^0$ que corresponde a $4 + 0 + 1 = 5$ (em notação decimal). Logo, só é possível sabermos o valor do número quando explicitamos a base utilizada em sua notação. Quando há necessidade de se explicitar a base, comumente coloca-se a base em subscrito na notação tais como em $(101)_{10}$ para indicar a base 10, ou em $(101)_2$ para indicar a base 2.

2.1.2 Conversão de uma base qualquer para a base 10

Seja $(1011)_2$. A qual número na base 10 ele corresponde? Para isso, basta considerarmos a notação polinomial e efetuar os cálculos. Seguem alguns exemplos envolvendo diferentes bases:

$$(1011)_2 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 8 + 0 + 2 + 1 = (11)_{10}$$

$$(123)_8 = 1 * 8^2 + 2 * 8^1 + 3 * 8^0 = 64 + 16 + 3 = (83)_{10}$$

$$(1A)_{16} = 1 * 16^1 + 10 * 16^0 = 16 + 10 = (26)_{10}$$

No segundo exemplo temos um número na base 8 e no terceiro exemplo temos um número na base 16. Essas bases são denominadas, respectivamente, **octal** e **hexadecimal**. No caso hexadecimal, os dígitos utilizados são $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ (valores correspondendo de 0 a quinze).

2.1.3 Conversão da base 10 para outra base

Para converter de decimal para uma outra base, podemos usar um algoritmo simples. Por exemplo, suponha que queremos calcular a representação na base 2 ou na base 8 do número $(107)_{10}$. Então procedemos como mostrado na figura 2.1. A representação na nova base consiste em tomar os restos das divisões sucessivas, do último para o primeiro. Note que o dígito mais significativo é o último quociente (um dígito que deve ser necessariamente menor que a base b, momento no qual as divisões sucessivas são interrompidas).

A intuição por trás deste algoritmo é simples. Basta analisarmos o caso da base 10. Podemos "isolar" o dígito menos significativo de um número x na base 10 ao tomarmos o resto da divisão de x por 10. Em seguida, ao se tomar o resto da divisão do quociente por 10, conseguimos isolar o segundo dígito menos significativo de x, e assim por diante. A mecânica desse processo é a mesma para outros valores de base.

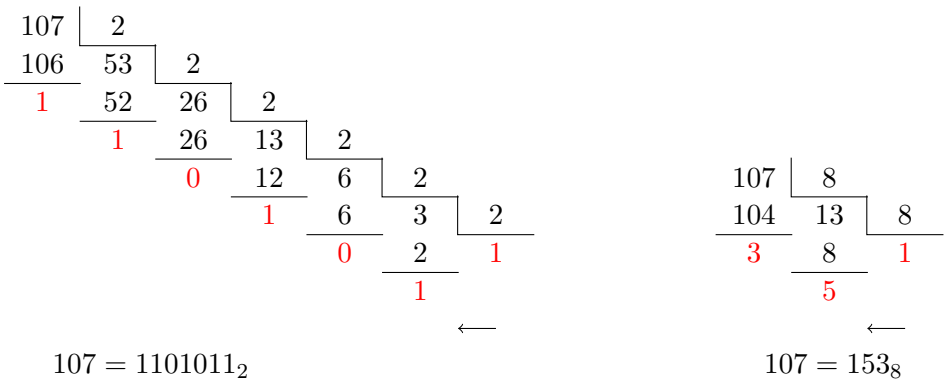


Figura 2.1: Ilustração do método de divisões sucessivas para conversão do número $(107)_{10}$ para a base 2 e 8. (Tipografia gerada baseado no código LaTeX disponível em https://tex.stackexchange.com/questions/107637/repeated-division-converting-from-base-10-to-another-base)

Note também que, uma vez em posse da notação na base 2, pode-se facilmente obter a notação nas bases 8, 16 ou qualquer outra potência de 2. Vimos que $(107)_{10} = (1101011)_2$. Para converter $(107)_{10}$ para a base 8, podemos agrupar os *bits* de $(1101011)_2$ de três em três, da direita para a esquerda, e converter cada grupo de três *bits* para um dígito octal:

$$(107)_{10} \rightsquigarrow (1101011)_2 \rightsquigarrow (\underbrace{1}_{1}\ \underbrace{101}_{5}\ \underbrace{011}_{3})_2 \rightsquigarrow (1\ 5\ 3)_8$$

Confira com o resultado da conversão mostrado na figura 2.1. Para converter $(107)_{10}$ para a base 16, podemos utilizar também o método das divisões sucessivas, ou então fazer o processo similar ao que acabamos de descrever, porém agrupando os *bits* de quatro em quatro:

$$(107)_{10} \rightsquigarrow (1101011)_2 \rightsquigarrow (\underbrace{110}_{6}\ \underbrace{1011}_{B})_2 \rightsquigarrow (6\ B)_{16}$$

Até aqui examinamos como os números são representados em diferentes bases e como pode ser realizada a conversão da representação de uma base para outra. Em geral, as conversões que nos interessam envolvem a base 10 por esta ser a base com a qual estamos acostumados.

2.2 Representação de números no computador

Computadores são máquinas finitas no sentido de serem capazes de armazenar apenas um número finito de *bits*. Por esta razão, não conseguimos representar nos computadores uma quantidade infinita de dígitos. Computadores tipicamente utilizam blocos de *bits* de tamanho fixo n, que chamaremos de **palavras**, para armazenar uma sequência de n dígitos binários.

Assim, qualquer informação a ser processada no computador precisa ser representada em binário, os quais por sua vez são armazenados em palavras. O processador em si não tem conhecimento sobre a natureza da informação representada numericamente: dígitos, caracteres do alfabeto, fotografia, música, um documento, enfim qualquer tipo de dado, no nível do processador é codificado numericamente e representado por um conjunto (potencialmente muito grande) de *bits*.

Neste ponto vale a pena mencionarmos o termo ***byte***, que consiste de unidades formadas por 8 *bits*. *Byte* é a unidade em geral utilizada para expressar a capacidade de armazenamento dos computadores, ou o volume de dados em um arquivo.

Nesta seção examinaremos a representação de números no computador, supondo que as palavras consistem de n *bits*. Do ponto de vista abstrato, n pode ser qualquer inteiro positivo e finito, porém os números tipicamente utilizados pelos computadores são 32 ou 64. Um aspecto de interesse relacionado à representação de números em n *bits* é a representação de números sem e com sinal. Dado que os coputadores conseguem representar essencialmente dois valores, 0 e 1 apenas, é importante entendermos como se faz a distinção entre positivos e negativos e como eles são processados em operações aritméticas como na adição ou na subtração.

Note, antes de mais nada, que em n bits podemos representar 2^n padrões binários (isto é, sequências de n *bits*), começando pelo $000\ldots0$ (todos os *bits* iguais a 0) e terminando em $111\ldots1$ (todos os *bits* iguais a 1). Quando falamos em representação de números inteiros, números reais ou caracteres, o que acontece de fato é que esses padrões binários funcionam como código da informação sendo representada. No caso de representação de números, é natural que a própria representação do número na base 2 seja usada como representação no computador.

2.2.1 Representação de números sem sinal

Supondo que podemos utilizar até n *bits*, quais são os números naturais que podem ser representados? O menor número é aquele que consiste de n zeros, correspondendo ao número 0. O maior número é aquele que consiste de n *bits* 1, que corresponde ao valor

$$\sum_{i=0}^{n-1} 1 * 2^i = 2^0 + 2^1 + \cdots + 2^{n-1}$$

O somatório acima é exatamente $2^n - 1$. Isto pode ser facilmente verificado da seguinte forma:

$$\underbrace{111\ldots11}_{n\ bits} + 1 = 1\underbrace{000\ldots00}_{n\ bits} = 2^n + 0 \Longrightarrow \underbrace{111\ldots11}_{n\ bits} = 2^n - 1$$

Portanto, com n *bits* pode-se representar os números de 0 a $2^n - 1$.

2.2.2 Representação de números com sinal

Para representar números com sinal (isto é, tanto positivos como negativos), lembremos antes de mais nada que o computador consegue representar apenas dois estados em um *bit*, que associamos aos dígitos 0 e 1. Por isso, não temos a possibilidade de representar um outro estado que codifique o sinal − ou +. O que podemos fazer é especificar um *bit* dentre os n *bits* como o *bit* de sinal. Uma escolha natural é escolher o *bit* mais significativo para representar o sinal do número. Também é importante ressaltar novamente que, dados n *bits*, são possíveis a representação de exatamente 2^n padrões binários (do $00\ldots0$ até $11\ldots1$), e isto não muda se os números a serem representados são com ou sem sinal. De fato, o que muda na representação de números com sinal é como interpretamos cada um desses 2^n padrões binários. Temos algumas interpretações conhecidas, descritas a seguir.

Sinal-magnitude: O *bit* mais significativo é utilizado para indicar o sinal, e os demais *bits* representam a magnitude do número. O *bit* de sinal igual a 0 significa que o valor é positivo e o *bit* de sinal igual a 1 significa que o valor é negativo. Assim, se consideramos $n = 4$ *bits*, a representação

0101 corresponde ao número 5 (positivo)
1101 corresponde ao número -5 (negativo)

Um caso a ser notado são as representações

0000 corresponde ao número 0
1000 corresponde ao número -0

Ou seja, nesta representação sinal-magnitude temos duas representações para o número 0.

Complemento de 1: nessa interpertação, o *bit* mais significativo é também utilizado para indicar o sinal como no caso anterior. Porém, a interpretação dos demais *bits* depende do *bit* de sinal:

- se o *bit* de sinal é 0, então o número é positivo e a magnitude é o valor correspondente aos demais *bits*;
- se o *bit* de sinal é 1, então o número é negativo e a magnitude é o valor correspondente ao **complemento** dos demais *bits*;

O complemento de 0 é 1 e o de 1 é 0. O complemento de um dígito d é denotado $\overline{d}$ e o complemento de um número binário $d_{n-1}\, d_{n-2}\, \ldots\, d_1\, d_0$ é definido como o complemento dígito a dígito, isto é, $\overline{d_{n-1}\, d_{n-2}\, \ldots\, d_1\, d_0} = \overline{d}_{n-1}\, \overline{d}_{n-2}\, \ldots\, \overline{d}_1\, \overline{d}_0$.

Na interpretação complemento de 1, por exemplo,

0101 corresponde ao número 5 (positivo)
1101 corresponde ao número -2 (negativo), pois $(\overline{101})_2 = (010)_2 = (2)_{10}$

Neste caso também temos dois 0s. Veja:

0000 corresponde ao número 0
1111 corresponde ao número -0, pois $(\overline{111})_2 = (000)_2 = (0)_{10}$

Complemento de 2: esta é uma interpertação similar à interpertação complemento de 1, porém quando o *bit* de sinal é 1, a magnitude é dada pelo valor correspondente ao complemento dos demais *bits*, mais 1. Por exemplo,

1101 corresponde ao número -3 (negativo) pois $(\overline{101})_2 + 1 = (010)_2 + 1 = (011)_2 = (3)_{10}$

Outra diferença em relação às interpertações anteriores é o fato de esta interpretação ter apenas uma representação para o valor 0.

Observação: As nomenclaturas complemento de 1 e complemento de 2 estão relacionadas com os conceitos gerais ***radix complement*** e ***diminished radix complement***, que aparece no contexto do método dos complementos. Supondo que dois números x e y são limitados por uma potência finita de 10, esse método permite expressar a subtração $x - y$ na forma $x + z$ (na qual z é o *radix complement* de y), Em computadores, essa propriedade é útil pois podemos realizar as operações de subtração por meio de adições, o que implica que não será necessário construir um circuito para adição e outro para subtração, como veremos mais adiante. Embora não seja um tópico a ser investigado neste texto, para mais detalhes sobre *radix complement* veja, por exemplo, https://en.wikipedia.org/wiki/Method_of_complements.

A tabela 2.1 mostra todos os 16 números binários possíveis em 4 *bits* e o valor representado de acordo com as interpretações descritas acima.

Note que, na interpretação complemento de 2, o negativo de um número é calculado como o seu complemento de dois. Por exemplo, 6 é 0110. Seu complemento de 2 é $\overline{0110}+1 = 1001+1 = 1010$. O binário 1010 na interpretação complemento de 2 é -6.

Note também que no caso complemento de 2, o valor de um número representando por $d_3\, d_2\, d_1\, d_0$ pode ser calculado como:

$$-d_3 * 2^3 + \sum_{i=0}^{2} d_i * 2^i = -d_3 * 8 + (d_2 * 4 + d_1 * 2 + d_0)$$

Representação binária	Valor em decimal de acordo com as interpretações			
$d_3\, d_2\, d_1\, d_0$	sem sinal	sinal-magnitude	complemento de 1	complemento de 2
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

Tabela 2.1: Números de 4 *bits* e diferentes interpretações.

Por exemplo, 1110 corresponde a $-1 * 8 + (1 * 4 + 1 * 2 + 0) = -8 + 6 = -2$.

Esta interpretação vale, de forma geral, para qualquer n. Podemos ver que na interpretação complemento de 2, os números que podem ser representados são de -2^{n-1} até $2^{n-1} - 1$. No caso $n = 4$, de -2^3 até $2^3 - 1$ (isto é, de -8 a 7).

2.3 Operações aritméticas com números binários

Agora que já vimos como interpertar os padrões binários como números com ou sem sinal, vamor ver como esses números podem ser operados. Aqui o interesse está nas operações de adição e subtração. Veremos que a adição usual (com a diferença de que estamos lidando com apenas dois dígitos) é suficiente para o cálculo de adições para as interpertações sem e com sinal. Mais ainda, o algoritmo de adição ambém é suficiente para as operações de subtração.

2.3.1 Adição

Para realizar a adição de números binários, basta aplicarmos o mesmo algoritmo de adição que usamos no caso da adição de números na base 10. A única diferença é que no caso dos números binários há apenas dois dígitos possíveis. As possíveis adições de dois números binários de um

dígito são mostrados a seguir. Note que ocorre um vai-um quando calculamos $1+1$.

$$0+0=0 \qquad 0+1=1 \qquad 1+0=1 \qquad 1+1=10$$

De forma similar à adição na base 10 com a qual estamos acostumados, números binários com múltiplos dígitos podem ser adicionados calculando-se a adição coluna a coluna, levando-se em conta um eventual vai-um vindo da coluna anterior. Na adição correspondente a uma coluna qualquer, podemos ter 8 possíveis configurações, mostradas na figura 2.2.

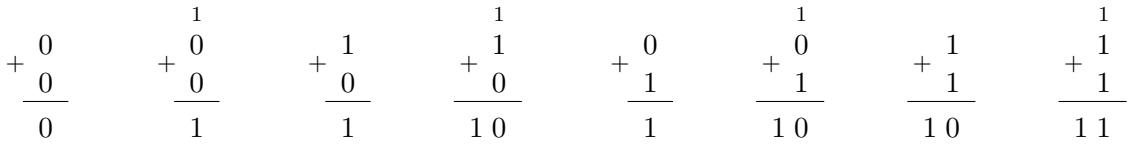
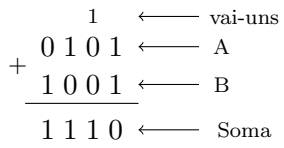


Figura 2.2: Todas as combinações (vai-um, dígito 1, dígito 2) que podem ocorrer em uma coluna qualquer quando realizamos a adição de dois números binários.

A adição dos números $A = 0101$ e $B = 1001$ é mostrada a seguir. **Considerando-se a interpretação sem sinal**, o valor de A é 5 e o de B é 9. A soma, 14, em notação binária é 1110.



Agora consideremos a mesma adição mostrada acima, porém supondo a **interpertação complemento de 2**. Temos que $A = 0101$ é 5 (a mesma de quando interpretamos como número sem sinal), $B = 1001$ é -7, e o resultado 1110 é -2, que corresponde exatamente ao resultado da adição 5+(-7)=-2. Ou seja, **uma mesma adição funciona tanto quando interpretamos os operandos como números sem sinal como quando interpretamos como números com sinal na notação complemento de 2**.

De fato, a ULA (unidade lógico-aritmética responsável por executar as operações lógicas e aritméticas) não precisa ter "ciência" sobre qual interpretação está sendo considerada. Ela simplemente executa a adição de dois números representados em n *bits*, não se importando com a interpretação dos mesmos. Cabe às unidades de controle ou ao software fazer a interpretação correta.

Neste ponto talvez não tenha ficado claro que a adição funciona corretamente independentemente da interpretação em questão. Assim, vamos examinar em seguida todas as possíveis adições de dois números de dois *bits*.

A tabela 2.2 mostra os quatro padrões binários de 2 *bits* (00, 01, 10 e 11) e os respectivos valores quando interpretados sem e com sinal.

Representação binária	Valor em decimal de acordo com as interpretações	
$d_1\ d_0$	sem sinal	complemento de 2
00	0	0
01	1	1
10	2	-2
11	3	-1

Tabela 2.2: Todos os padrões binários de 2 *bits* e seus valores de acordo com as interpretações sem e com sinal.

Se considerarmos todas as diferentes adições de dois números de 2 *bits*, temos então um total de 16 adições possíveis. Estes são mostrados na figura 2.3. Essas 16 adições da forma $A + B$, com $A, B \in \{00, 01, 10, 11\}$, par a par, estão mostradas de forma repetida (as 16 adições no lado esquerdo são exatamente as mesmas que estão no lado direito). Em cada coluna, no topo temos a adição do número binário 00 com todos os 4 números, depois abaixo a adição do número 01 com todos os 4 números, e assim por diante. A diferença entre a parte esquerda e direita é que na parte esquerda são mostrados os valores de acordo com a interpretação sem sinal enquanto na parte direita são mostrados os valores de acordo com a interpretação complemento de 2. Essas interpretações estão abaixo de cada adição. Uma vez que estamos considerando apenas 2 *bits*, nessas interpretações vamos ignorar eventuais vai-uns que "caem para fora". Os casos nos quais o resultado é incorreto (não corresponde à soma esperada) estão anotados em vermelho e são todos devido a *overflow*, quando a quantidade de *bits* (no caso $n = 2$) não é suficiente para armazenar o valor correto da soma.

Estas tabelas mostram que a operação de adição propriamente ocorre da mesma forma, independentemente da interpretação considerada. O que difere é que, dependendo da interpretação, os casos que resultam em *overflow* não são os mesmos. O *overflow* é a situação na qual o resultado da adição não pode ser representado em n *bits*. Se os números binários são interpretados como sem sinal, então ocorrerá *overflow* sempre que a soma for maior ou igual a 2^n. Se os números são interpretados na forma complemento de 2, há *overflow* sempre que o resultado da adição é menor que -2^{n-1} ou maior que $2^{n-1} - 1$. Por exemplo, na segunda tabela no lado esquerdo, $1 + 3 = 0$ é um caso de *overflow* já que a soma correta é 4 e em binário precisaríamos de 3 *bits* para representar o número 4 ($(4)_{10} = (100)_2$).

Detectar *overflow* é importante para que os programas que computam operações aritméticas possam ser informados e tomem as providências adequadas no contexto da aplicação em questão. Observe os casos acima e identifique uma forma simples para detecção de *overflow*. Voltaremos a essa questão mais adiante, após examinarmos como são realizadas as operações de subtração.

00	00	00	00
00	01	10	11
00	01	10	11
0+0=0	0+1=1	0+2=2	0+3=3

00	00	00	00
00	01	10	11
00	01	10	11
0+0=0	0+1=1	0+(-2)=-2	0+(-1)=-1

01	^{1}01	01	^{11}01
00	01	10	11
01	10	11	00
1+0=1	1+1=2	1+2=3	1+3=0

01	^{1}01	01	^{11}01
00	01	10	11
01	10	11	00
1+0=1	1+1=-2	1+(-2)=-1	1+(-1)=0

10	10	^{1}10	^{1}10
00	01	10	11
10	11	00	01
2+0=2	2+1=3	2+2=0	2+3=1

10	10	^{1}10	^{1}10
00	01	10	11
10	11	00	01
(-2)+0=-2	(-2)+1=-1	(-2)+(-2)=0	(-2)+(-1)=1

11	^{11}11	^{1}11	^{11}11
00	01	10	11
11	00	01	10
3+0 =3	3+1=0	3+2=1	3+3=2

11	^{11}11	^{1}11	^{11}11
00	01	10	11
11	00	01	10
(-1)+0=-1	(-1)+1=0	(-1)+(-2)=1	(-1)+(-1)=-2

Figura 2.3: Todas as adições entre números binários de 2 *bits*. À esquerda, a interpretação sem sinal, e à direita a interpretação com sinal (baseado na notação complemento de 2). Em vermelho estão as adições que não são computadas corretamente, devido a *overflow*.

2.3.2 Subtração

Podemos calcular subtrações entre números binários de forma similar ao realizado com números decimais. Por outro lado, note que uma subtração pode ser expressa na forma de uma adição: calcular $A - B$ é equivalente a calcular $A + (-B)$. Isto sugere que se temos uma maquinaria para calcular adições, então talvez não precisemos de outra para a subtração. Para fazer uso da maquinaria de adição existente, precisamos saber como representar $-B$. Na notação complemento de 2, conforme já vimos anteriormente, o negativo de um número binário é calculado complementando-se o número e em seguida adicionando-se 1. Por exemplo, supondo $n = 4$, o número 7 é expresso em binário por 0111. Pela definição de complemento de 2, temos que $\overline{0111} + 1 = 1000 + 1 = 1001$ é a representação do número -7 em binário. De fato, isso pode ser verificado consultando-se a tabela 2.1, ou ainda observando-se que 1001 corresponde ao número $-1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$ que é igual a $-8 + 1 = -7$.

Vamos então calcular $7 - 4$ usando a ideia acima. Como 4 é 0100, temos então que -4 é $\overline{0100} + 1 = 1011 + 1 = 1100$, e portanto $7 - 4$ pode ser calulado efetuando-se $0111 + 1100$:

```
 1 1          ⟵ vai-uns
   0 1 1 1    ⟵ 7
+  1 1 0 0    ⟵ -4
-------------
   0 0 1 1    ⟵ 7+(-4)=3
```

Ignorando-se o último *carry* (vai-um) mais à esquerda, temos o resultado esperado, isto é, 3.

Note, porém, que pode ocorrer um *overflow* quando calculamos $-B$. Por exemplo, a expressão $-2-(-2)$ envolve o número -2 que está no intervalo de números representáveis em 2 *bits* e o resultado da expressão deve ser 0. No entanto, calcular $-B$ neste caso significa calcular $-(-2)$, que corresponderia ao 2, um número que não pode ser representado em 2 *bits*. Assim, a expressão $-2-(-2)$ não será computada corretamente se procedermos como explicado acima.

Uma forma para contornar esse problema é observar que $-B = \overline{B} + 1$. Assim, ao invés de calcular $A-B$ via $A+(-B)$, que exige o cálculo de $-B$, podemos calcular $A-B$ via $A+\overline{B}+1$. Para não trocarmos uma adição por duas adições, podemos usar o termo 1 como o vai-um inicial, na coluna 0 e realizar a adição de A com $\overline{B}$.

A figura 2.4 mostra todas as 16 subtrações, como feito no caso das adições anteriormente, e suas interpretações sem e com sinal. A subtração é efetivamente realizada por meio da adição conforme explicado acima. Note que há um vai-um na coluna mais à direita em todas as adições e que o segundo termo da adição é o complemento do subtraendo B, pelo motivo mencionado no parágrafo anterior. Os casos de *overflow* são indicados em vermelho.

2.3.3 Detecção de *overflow*

Os exemplos acima ilustrando as adições e subtrações de números binários com $n = 2$ *bits* mostram que a notação complemento de dois é versátil pois permite que subtrações sejam realizadas por meio de um algoritmo de adição. Note que essa propriedade pode ser estendida para qualquer tamanho n.

Dado que o número de *bits* para representar os números é fixo, operações aritméticas podem resultar em *overflow*, situações nas quais o valor resultante da operação está além do intervalo de números que podem ser representados naquela quantidade de *bits*. Dependendo do contexto no qual essas operações foram realizadas, é preciso tomar providências para que o resultado incorreto não afete o restante do processamento. Para tanto, a maquinaria que efetua adições precisa, além do resultado, devolver elementos que permitam a identificação da ocorrência de *overflow*.

Ao analisarmos os exemplos de adição e subtração entre números de 2 *bits*, podemos observar alguns padrões. Por exemplo, podemos usar o fato de que a soma de dois positivos não pode ser

$^{111}00$	$^{1}00$	$^{11}00$	$^{1}00$
11	10	01	00
00	11	10	01
0+(-0)=0	0+(-1)=3	0+(-2)=2	0+(-3)=1

$^{111}00$	$^{1}00$	$^{11}00$	$^{1}00$
11	10	01	00
00	11	10	01
0+(-0)=0	0+(-1)=-1	0+(-(-2))=-2	0+(-(-1))=1

$^{111}01$	$^{111}01$	$^{11}01$	$^{11}01$
11	10	01	00
01	00	11	10
1+(-0)=1	1+(-1)=0	1+(-2)=3	1+(-3)=2

$^{111}01$	$^{111}01$	$^{11}01$	$^{11}01$
11	10	01	00
01	00	11	10
1+(-0)=1	1+(-1)=0	1+(-(-2))=-1	1+(-(-1))=-2

$^{111}10$	$^{1\ 1}10$	$^{111}10$	$^{1}10$
11	10	01	00
10	01	00	11
2+(-0)=2	2+(-1)=1	2+(-2)=0	2+(-3)=3

$^{111}10$	$^{1\ 1}10$	$^{111}10$	$^{1}10$
11	10	01	00
10	01	00	11
(-2)+(-0)=-2	(-2)+(-1)=1	(-2)+(-(-2))=0	(-2)+(-(-1))=-1

$^{111}11$	$^{111}11$	$^{111}11$	$^{111}11$
11	10	01	00
11	10	01	00
3+(-0) =3	3+(-1)=2	3+(-2)=1	3+(-3)=0

$^{111}11$	$^{111}11$	$^{111}11$	$^{111}11$
11	10	01	00
11	10	01	00
(-1)+(-0)=-1	(-1)+(-1)=-2	(-1)+(-(-2))=1	(-1)+(-(-1))=0

Figura 2.4: Todas as subtrações, expressas na forma de adições, entre números binários de 2 *bits*. À esquerda, a interpretação sem sinal, e à direita a interpretação com sinal (baseado na notação complemento de 2). Em vermelho estão as operações que não são computadas corretamente, devido a *overflow*.

negativa e, similarmente, que a soma de dois negativos não pode ser positiva. Porém, o *overflow* pode ser caracterizado sem considerarmos o sinal dos operandos. Essa caracterização depende da operação (adição ou subtração) e também da interpretação sem ou com sinal dos números. Por ora, deixaremos a identificação desses padrões como exercício :-) Embora os exemplos mostrados sejam para números de 2 *bits*, não deverá ser difícil perceber que esses padrões se repetem para números com qualquer quantidade de *bits*.

2.4 Comentários finais

Neste capítulo discutimos a representação de números em diferentes bases. No contexto da computação, a base dois é de particular interesse pois os computadores são capazes de representar apenas dois dígitos, o 0 e o 1. Logo, qualquer informação a ser representada e processada pelo computador precisa ser codificada como números binários.

Vimos também que os computadores geralmente usam uma quantidade fixa de *bits* para representar um número, e que essas representações podem ser interpretadas como números sem ou com sinal. Na interpretação com sinal utiliza-se a notação complemento de dois. Com isso, as

operações de adição e de subtração podem ser realizadas utilizando-se uma mesma maquinaria de adição: no caso de uma operação de subtração, o subtraendo deve ser complementado e deve ser fornecido um vai-um para a coluna menos significativa.

No próximo capítulo examinaremos como o algoritmo de adição é implementado em circuitos lógicos.

Exercícios

1. Converta cada um dos seguintes números para a representação na base 10. Escreva explicitamente o cálculo realizado.

   (a) $(1010)_2$

   (b) $(10110010)_2$

   (c) $(111110)_3$

   (d) $(2075)_8$

   (e) $(2A3)_{16}$

2. Escreva os seguintes números decimais nas bases 2, 8 e 16. Mostre o cálculo realizado.

   (a) 9

   (b) 247

   (c) 479

   (d) 1025

3. Escreva todos os números binários de 3 bits em ordem lexicográfica.

4. Supondo que estamos considerando apenas números inteiros sem sinal, qual é o intervalo de números que podemos representar em 7 bits ? Explique.

5. Suponha que precisamos construir um sistema digital binário que seja capaz de armazenar e manipular números de 0 a $(3000)_{10}$. Quantos *bits* (dígitos binários) deve possuir uma "palavra" desse sistema ? Explique.

6. No exercício anterior, se supormos que uma palavra é formada por dígitos ternários (isto é, 0, 1 ou 2), quantos dígitos serão necessários para armazenar os números de 0 a $(5000)_{10}$? Explique.

7. Para cada número abaixo, escreva o valor decimal correspondente segundo as interpretações sem sinal, complemento de 1 e complemento de 2. Calcule também o complemento de 1 e o complemento de 2 e escreva os valores decimais correspondentes (segundo as interpretações complemento de 1 e complemento de 2, respectivamente).

(a) 00000

(b) 00101

(c) 10010

(d) 11111

8. Considere todos os números binários de 3 *bits*. Escreva uma tabela indicando, para cada número binário, o correspondente número na base 10 quando assumimos a interpretação sem sinal, sinal-magnitude, o complemento de 1 e o complemento de dois. Ordene os números binários de acordo com a ordem lexicográfica.

9. No exemplo de 2 *bits* visto no texto, o que acontece se efetuamos a subtração $A-B$ usando $A+(-B)$ (em vez de usar $A+\overline{B}+1$ como foi feito), com $-B$ sendo o complemento de 2 de B ? Discuta o caso no qual $B=-2$.

10. Quando ocorre *overflow* na adição de números sem sinal ? Como detectar a ocorrência de *overflow* ?

11. Quando ocorre *overflow* na adição de números com sinal (complemento de 2)? Como detectar a ocorrência de *overflow* ?

12. Quando ocorre *overflow* na subtração de números sem sinal ? Como detectar a ocorrência de *overflow* ?

13. Quando ocorre *overflow* na subtração de números com sinal (complemento de 2)? Como detectar a ocorrência de *overflow* ?

14. Suponha que dispomos de uma maquinaria capaz de realizar a operação de adição de dois números binários de n *bits* cada. As entradas dessa maquinaria são $2n$ bits, n do primeiro operando e n do segundo operando.

    Como essa maquinaria pode ser usada para efetuar uma operação de subtração entre dois números binários de n *bits* cada ? Qual alteração nos dados ou nas entradas da maquinaria precisam ser efetuadas ?

15. Consulte https://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html, e explique como são representados os números reais em computador (*Floating-Point Number Representation*).

16. Consulte https://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html, e escreva quais são os códigos ASCII das letras de 'a' a 'z'.

17. Escreva um programa em linguagem C que, dado um inteiro positivo n, calcula e imprime o fatorial de todos os inteiros de 1 a n. Teste para $n \geq 15$.

    Em seguida, escreva o mesmo programa em linguagem Python, e faça os mesmos testes.

Experimente executar os dois programs em C a seguir. Observe as saídas dos programas e relacione-as com o que foi discutido neste capítulo.

```
#include <stdio.h>

int main() {
  int i, n ;
  int fat ;

  printf("Número de bytes da variável fat: %ld\n\n", sizeof(fat));
  n = 25 ;
  fat = 1 ;

  for(i=1;i<=n;i++) {
    fat = fat*i;
    printf("%d! = %d\n", i, fat) ;
  }

  return 0;
}
```

```
#include <stdio.h>

int main() {
  int i, n ;
  long int fat ;

  printf("Número de bytes da variável fat: %ld\n\n", sizeof(fat));
  n = 25 ;
  fat = 1 ;

  for(i=1;i<=n;i++) {
    fat = fat*i;
    printf("%d! = %ld\n", i, fat) ;
  }

  return 0;
}
```

Capítulo 3

Funções e circuitos lógicos

Última atualização em 09/03/2020

Comentamos anteriormente que as unidades básicas de armazenamento nos computadores são os *bits*, cada um capaz de representar o valor 0 ou 1. Um grupo de *bits* pode ser interpretado como um número binário. No capítulo anterior, vimos a representação de números na base 2. Vimos também como converter números na base 10 para a base 2 e vice-versa, como as representações binárias em um número finito e fixo de *bits* podem ser interpretadas como números sem e com sinal, e como realizar operações básicas como adição e subtração com números binários. Para entender o funcionamento do computador com respeito a parte referente a sua capacidade de processar/transformar dados, além de entender como os dados são representados, precisamos entender como o mecanismo de processamento é modelado e realizado. Neste capítulo examinaremos este segundo ponto. Introduzimos noções iniciais sobre funções e circuitos lógicos; mais precisamente, veremos que o processamento de informação pode ser modelado por funções lógicas e ser expresso por meio de circuitos lógicos.

3.1 Processamento de dados visto como uma função

De forma grosseira, um computador ou cada um dos aplicativos que rodam nele podem ser vistos como caixas-pretas que recebem dados ou sinais na entrada e produzem novos dados ou efeitos na saída. Por exemplo, um aplicativo do tipo calculadora recebe números, a indicação da operação a ser realizada (adição, subtração, multiplicação, etc), e realiza o cálculo, mostrando em seguida o resultado da operação. Ignorando por um momento considerações relacionadas com a captura do número e outras informações digitadas pelo usuário (seja por meio do teclado

ou por meio do toque na tela) ou com a visualização do resultado na tela, podemos pensar que o cálculo correspondente à operação é uma transformação de dados: supondo que a operação é a de adição, a entrada da transformação são dois números (os números a serem somados) e a saída é um terceiro número que é a soma (o resultado da adição dos dois números de entrada).

Tomemos como exemplo a adição de números binários, vista no capítulo anterior. A operação de adição pode ser realizada calculando-se a soma coluna a coluna, da coluna mais à direita até a última coluna mais à esquerda. A adição referente a uma coluna pode tomar 8 possíveis configurações, como foi mostrado na figura 2.2.

Seja c_{in} (*carry-in*, o vai-um que veio da coluna anterior) e a e b os *bits* dos números binários A e B sendo somados, numa dada coluna. A adição resulta em um *bit* soma s que corresponde à soma binária $c_{in} + a + b$ e poderá gerar *carry-out* c_{out} (vai-um para a próxima coluna). Usando essas notações, podemos escrever o comportamento da soma referente a uma coluna como mostrado na tabela 3.1.

Entrada			Saída	
c_{in}	a	b	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabela 3.1: Tabela que define a soma dos dígitos binários em uma coluna, considerando-se a adição de dois números binários.

Essa tabela define, para todas as possíveis atribuições de valores às variáveis c_{in}, a e b, o valor do *bit* soma s e do *carry-out* c_{out}. Em outras palavras, define uma função binária com três entradas e duas saídas, que descreve o comportamento de um **somador de *bits***.

Observe que se o computador possuir um módulo capaz de realizar a função acima, então tais módulos poderão ser usados para construir um módulo mais geral para realizar a adição de dois números binários de n *bits*. Bastaria que os módulos somadores de *bits* fossem conectados em cascata, para realizar a adição coluna-a-coluna, da coluna menos significativa até a coluna mais significativa. A seguir essa ideia é ilustrada para o caso de números com $n = 4$ *bits*.

Somador (para números de 4 bits): Sejam $A = a_3\, a_2\, a_1\, a_0$ e $B = b_3\, b_2\, b_1\, b_0$ dois números binários de 4 *bits* (onde o subscrito 0 e 3 representam, respectivamente, o *bit* menos e mais

significativo). Suponha que temos componentes somadores de *bits*, que realizam a soma de *bits* de uma coluna, conforme descrito anteriormente. Denotando as entradas dos somadores de *bits* por a_i, b_i e c_i (vai-um vindo da coluna anterior) e as saídas por s_i e c_{i+1}, a ligação em cascata pode ser feita de forma que a saída vai-um de uma coluna i alimente a entrada vai-um da coluna $i+1$, como mostrado na figura 3.1.

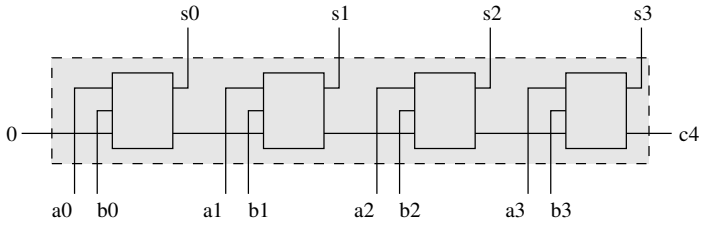


Figura 3.1: Esquema de um somador de 4 *bits*. Cada retângulo corresponde a um componente somador de *bits*, cujos pinos à esquerda representam as entradas a_i, b_i e c_i e os pinos à direita representam as saídas s_i e c_{i+1}. No diagrama $c_0 = 0$, adequado para a realização da operação de adição. O resultado da soma $A + B$ será $S = s_3\, s_2\, s_1\, s_0$.

Essa "transformação", de entrada para saída, da operação de adição de dois números de 4 bits pode também ser descrita em uma tabela como a tabela 3.1. Porém, o tamanho da tabela seria muito maior e não convém escrevê-la explicitamente (quantas linhas terá tal tabela?).

A relação entrada-saída de um somador de *bits* pode ser explicitamente descrita por uma tabela (tabela 3.1). Note que a tabela, por sua vez, define uma função com 3 entradas binárias e duas saídas binárias. Podemos denotá-la por $f : \{0,1\}^3 \to \{0,1\}^2$. Uma entrada dessa função é uma trinca (c_{in}, a, b), com $c_{in}, a, b \in \{0,1\}$, e uma saída é um par (s, c_{out}), com $s, c_{out} \in \{0,1\}$, i.e., $f(c_{in}, a, b) = (s, c_{out})$. Podemos também pensar a função f como um par de funções $f = (f_1, f_2)$ tal que $f_1(c_{in}, a, b) = s$ e $f_2(c_{in}, a, b) = c_{out}$.

Observação: $\{0,1\}^n$ denota o produto cartesiano de conjuntos. Por exemplo, $\{0,1\}^3$ corresponde ao produto cartesiano $\{0,1\} \times \{0,1\} \times \{0,1\}$. Um elemento de $\{0,1\} \times \{0,1\} \times \{0,1\}$ é uma trinca (a, b, c), tal que $a, b, c \in \{0,1\}$. De forma geral, dados por exemplo dois conjuntos A e B, temos que o produto cartesiano de A e B é o conjunto definido por $A \times B = \{(a, b) : a \in A \text{ e } b \in B\}$ (isto é, pares (a, b) tais que o primeiro elemento do par pertence a A e o segundo elemento do par pertence a B).

De forma geral, uma função que mapeia um certo número n de entradas binárias, que denotaremos $x_1, x_2, \ldots, x_n$, para m saídas binárias $y_1, \ldots, y_m$, pode ser denotada por $f : \{0,1\}^n \to \{0,1\}^m$. Podemos também interpretar f como $f = (f_1, f_2, \ldots, f_m)$, com $f_i : \{0,1\}^n \to \{0,1\}$ e $y_i = f_i(x_1, x_2, \ldots, x_n)\ \forall i$.

3.2 Funções binárias (lógicas)

Considerando o somador de bits (tabela 3.1), temos duas funções binárias $s(c_{in}, a, b)$ e $c_{out}(c_{in}, a, b)$, ambas com 3 entradas. Usando a notação anteriormente mencionada, poderíamos também pensar que temos uma função $f(c_{in}, a, b) = \big(s(c_{in}, a, b), c_{out}(c_{in}, a, b) \big)$.

Como podemos calcular eficientemente a saída $\big(s(c_{in}, a, b), c_{out}(c_{in}, a, b) \big)$, dada uma entrada (c_{in}, a, b) ? Consultar a tabela não parece ser uma solução interessante pois a tabela poderá ser muito grande.

Note que a saída s será igual a 1 se e somente se a soma $c_{in} + a + b$ for ímpar. Isto pode ser escrito como:

$$s = 1 \iff (c_{in} + a + b) \,\%\, 2 = 1$$

na qual % denota a operação "resto da divisão". Similarmente, a saída c_{out} será igual a 1 se e somente se a soma $c_{in} + a + b$ for maior ou igual a 2. Isto pode ser escrito como:

$$c_{out} = 1 \iff (c_{in} + a + b) \,//\, 2 = 1$$

na qual // denota a operação "parte inteira da divisão".

Ao considerarmos uma possível implementação da função acima no computador, essas expressões não são muito úteis pois o que estamos tentando implementar é justamente a operação de adição (e as expressões acima envolvem a operação de adição ...) Observe, porém, que podemos escrever a relação entrada-saída da função s de forma mais direta:

$$\begin{aligned} s(c_{in}, a, b) = 1 \iff\ & (c_{in} = 0 \text{ e } a = 0 \text{ e } b = 1) \text{ ou } (c_{in} = 0 \text{ e } a = 1 \text{ e } b = 0) \text{ ou} \\ & (c_{in} = 1 \text{ e } a = 0 \text{ e } b = 0) \text{ ou } (c_{in} = 1 \text{ e } a = 1 \text{ e } b = 1) \end{aligned}$$

Note que no lado direito da equivalência, há uma expressão que inclui os **conectivos lógicos** E e OU, que podem ser "interpretados" da forma que estamos acostumados. A expressão "enumera" as entradas para as quais a função toma valor 1. Não é preciso muito esforço para verificar que para as demais entradas a função s toma valor 0. Essa expressão da função s em termos de conectivos lógicos E e OU sugere que se tivéssemos dispositivos físicos que implementam o comportamento desses conectivos, seria possível implementar fisicamente a função s.

A tabela 3.2 mostra a definição desses conectivos lógicos. As colunas x_1, x_2 denotam as entradas

e as demais colunas definem funções lógicas correspondentes aos conectivos. Por exemplo, a terceira coluna define o E lógico que toma valor 1 se e somente se $x_1 = 1$ e $x_2 = 1$. Isto pode ser expresso algebricamente por $x_1\, x_2$. A expressão algébrica das demais funções aparece também no cabeçalho de cada coluna. Note que o símbolo +, usado para a operação OR (OU lógico), não é o mesmo da álgebra elementar; na álgebra elementar temos $x + x = 2x$, mas na álgebra booleana temos $x + x = x$. Esses detalhes serão vistos em um próximo capítulo.

		Conectivo lógico		
		E	OU	NÃO
x_1	x_2	$x_1\, x_2$	$x_1 + x_2$	$\overline{x}_1$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Tabela 3.2: Tabela-verdade das operações lógicas básicas. O valor 1 pode ser interpretado como verdadeiro (V) e o valor 0 como falso (F).

Exercícios: Suponha que $x_1 = 0$, $x_2 = 1$ e $x_3 = 1$

1. Qual é o valor da expressão $x_1 + x_2 + x_3$?
2. Qual é o valor da expressão $x_1\, x_2\, x_3$?
3. Qual é o valor da expressão $x_1\, x_2 + x_1\, x_3$?
4. Qual é o valor da expressão $x_1\, x_2 + x_1\, x_3 + x_2\, x_3$?
5. Qual é o valor da expressão $\overline{x_1}$?
6. Qual é o valor da expressão $\overline{x_1 + x_2}$?

Retomando à expressão s, podemos dizer que ela toma valor 1 se ao menos uma das quatro conjunções, ($c_{in} = 0$ e $a = 0$ e $b = 1$), ($c_{in} = 0$ e $a = 1$ e $b = 0$), ($c_{in} = 1$ e $a = 0$ e $b = 0$) ou ($c_{in} = 1$ e $a = 1$ e $b = 1$), for verdade. Em outras palavras, a função que define s é tal que seu valor deve ser 1 para as entradas 001, 010, 100 e 111, e deve ser 0 para as demais entradas.

Observe que, por exemplo, quando $c_{in} = 1$, $a = 1$ e $b = 1$, a expressão $c_{in}\, a\, b$ (produto das três variáveis, ou o E lógico das três variáveis) vale 1. Para qualquer outra atribuição de valor a essas três variáveis, o produto $c_{in}\, a\, b$ toma valor 0. Podemos, portanto, fazer um raciocínio inverso e determinar qual é o produto que toma valor 1 para uma dada entrada específica. Por exemplo, qual é o produto que toma valor 1 quando $c_{in} = 0$, $a = 1$ e $b = 1$? A variável c_{in}

precisa aparecer barrada ($\overline{c}_{in}$) no produto (pois caso contrário, a conjunção (E lógico) de c_{in} com qualquer outro valor seria necessariamente 0). O produto que queremos nesse caso é $\overline{c}_{in}\, a\, b$.

Ao fazermos o OU (disjunção) de todos os termos produtos associados a cada entrada que toma valor 1 na tabela, temos uma forma de expressar a função descrita pela tabela. No caso da função s, temos que:

$$s(c_{in}, a, b) = \overline{c}_{in}\, \overline{a}\, b + \overline{c}_{in}\, a\, \overline{b} + c_{in}\, \overline{a}\, \overline{b} + c_{in}\, a\, b \tag{3.1}$$

Similarmente, temos que a função c_{out} pode ser escrita como:

$$c_{out}(c_{in}, a, b) = \overline{c}_{in}\, a\, b + c_{in}\, \overline{a}\, b + c_{in}\, a\, \overline{b} + c_{in}\, a\, b \tag{3.2}$$

O exemplo acima ilustra que podemos escrever uma função binária na forma algébrica usando apenas as operações lógicas E, OU e NÃO, aplicadas sobre variáveis lógicas que representam as entradas binárias. Resta saber se qualquer função binária pode ser expressa algebricamente. Este assunto será retomado mais adiante.

3.3 Portas lógicas e circuitos lógicos

As operações lógicas são extremamente simples, como pudemos ver acima. Uma vez que a adição de *bits* pode ser expressa por meio dessas operações, parece bastante razoável pensar que se formos capazes de implementar essas operações fisicamente, também seremos capazes de fisicamente implementar um somador de *bits*.

De fato, os sistemas digitais são capazes de realizar as operações lógicas. Os componentes do hardware que implementam essas operações lógicas são chamadas **portas lógicas**. Uma porta E, por exemplo, é um componente que dadas duas entradas (que podem tomar dois valores, 0 ou 1, ou então OFF ou ON) tem valor 1 em sua saída somente quando ambas as entradas também estão em 1. As portas lógicas costumam ser representadas por figuras que seguem padrões da área. O desenho que usaremos neste texto para as **portas lógicas** E e OU e o **inversor** NÃO, que correspondem respectivamente às operações lógicas E, OU e NÃO, são mostrados na figura 3.2. Essas portas recebem sinais de entrada à esquerda e produzem um sinal de saída à direita.

Ao conectarmos a saída de um dispositivo desses nas entradas de outros, podemos construir uma rede interconexa de dispositivos. No caso de sistemas digitais, a interconexão define o que é

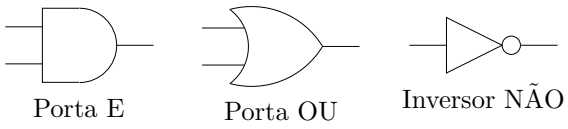


Figura 3.2: Representação gráfica de algumas portas lógicas.

chamado de **circuito digital**. Sem nos atermos à característica física da implementação, podemos usar genericamente o termo **circuito lógico** para nos referirmos a essas redes interconexas. Por exemplo, seja o circuito mostrado na figura 3.3. Ele consiste de três inversores e uma porta E. A saída do circuito, expressa como $f(A, B)$, indica que o valor da saída depende das duas entradas indicadas por A e B, que podem ser vistas como as variáveis de entrada da função.

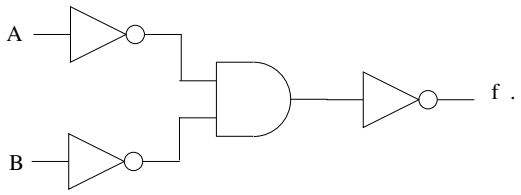


Figura 3.3: Um circuito simples.

Dizemos que **um circuito realiza uma função**. Podemos descrever seu funcionamento por meio de uma tabela-verdade. Cada linha da tabela corresponde a uma das possíveis atribuições de valor às variáveis de entrada do circuito. No caso do circuito da figura 3.3, a tabela-verdade é mostrada na tabela 3.3, juntamente com os valores do circuito em pontos intermediários entre a entrada e a saída. Observe também que uma expressão algébrica que define a função pode ser obtida diretamente do circuito: no exemplo, primeiramente ambas as entradas são negadas e em seguida o resultado do E lógico entre elas é também negado, ou seja, temos $f(A, B) = \overline{\overline{A}\,\overline{B}}$.

A	B	$\overline{A}$	$\overline{B}$	$\overline{A}\,\overline{B}$	$f(A,B) = \overline{\overline{A}\,\overline{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

Tabela 3.3: Tabela-verdade da função $f(A, B) = \overline{\overline{A}\,\overline{B}}$.

Note que $f(A, B) = \overline{\overline{A}\,\overline{B}} = A + B$. Isto mostra que uma mesma função pode ser realizada por diferentes circuitos. Quando dois circuitos realizam uma mesma função, eles são ditos equivalentes. O circuito da figura 3.3 é equivalente à porta OU.

Escrever a tabela-verdade correspondente a um circuito dado é uma tarefa mecânica, embora possa ser trabalhosa caso o circuito envolva muitas portas lógicas. Uma vez que uma tabela-verdade define uma única função lógica, circuitos que resultam em uma mesma tabela-verdade

são equivalentes (isto é realizam uma mesma função). Por outro lado, dada uma tabela-verdade, não sabemos ainda quantas realizações dela na forma de circuitos digitais são possíveis. Qualquer tabela-verdade pode ser realizada por um circuito digital? Dentre os diversos circuitos que possam realizar uma mesma mesma função, existe um que seja melhor que os outros ? Essas questões também serão abordadas mais adiante.

3.4 O circuito somador

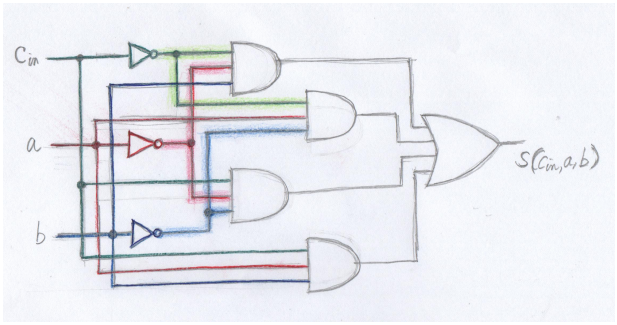
Nesta seção usaremos a adição de dois números binários como exemplo para revisar os conceitos vistos até agora. Especificamente, iremos examinar como pode ser construído um circuito somador de *bits*.

Já vimos que uma função lógica pode ser descrita por meio de uma tabela-verdade (apesar de tal representação só ser praticável quando o número de entradas é pequeno ...). Similarmente, a recíproca é verdadeira (isto é, uma tabela-verdade define uma função). Reforçaremos aqui a ideia de que uma função (ou, equivalentemente, uma tabela-verdade) pode ser expressa por uma expressão lógica, e que expressões lógicas por sua vez podem ser "traduzidas" para um circuito lógico. Buscaremos revisar como esses elementos se relacionam.

Seja a função s, do somador de *bits*. Por conveniência, abaixo reescrevemos a tabela-verdade do somador de *bits*. Vimos que uma expressão da função s é dada por $s(c_{in}, a, b) = \overline{c}_{in}\,\overline{a}\,b + \overline{c}_{in}\,a\,\overline{b} + c_{in}\,\overline{a}\,\overline{b} + c_{in}\,a\,b$.

Entrada			Saída	
c_{in}	a	b	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Admitindo-se portas lógicas com múltiplas entradas, precisamos então de quatro portas E, uma porta OU e três inversores para implementar a função s, como mostrado à esquerda na figura 3.4.



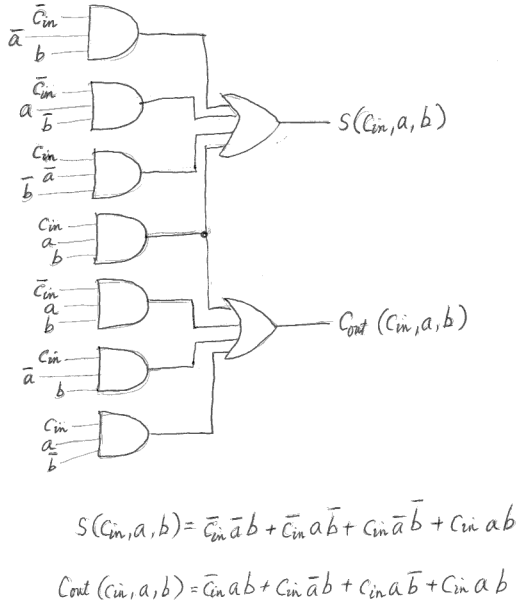


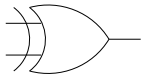
Figura 3.4: À esquerda, circuito correspondente à expressão do *bit* soma $s(c_{in}, a, b) = \overline{c}_{in}\,\overline{a}\,b + \overline{c}_{in}\,a\,\overline{b} + c_{in}\,\overline{a}\,\overline{b} + c_{in}\,a\,b$. À direita, um circuito somador de *bits*, implementando as funções s e c_{out} na forma soma de produtos.

À direita da figura 3.4 é mostrado o circuito de ambas as funções, s e c_{out}. Como o último termo é o mesmo em s e em c_{out}, ele é compartilhado. São utilizados portanto 7 portas E (com três entradas cada) e 2 portas OU (com quatro entradas cada). Além disso, seriam necessários inversores, um para cada entrada. Daqui em diante, porém, em vez de desenhar inversores para as entradas barradas, iremos escrever diretamente a variável barrada na entrada do circuito.

Embora seja claro que um circuito corresponde a uma única função, não é claro quantos circuitos realizam uma função. Na figura 3.5 é mostrado um segundo circuito que também realiza as duas funções s e c_{out}, porém com maior compartilhamento de subcircuitos.

Esse circuito utiliza a porta XOR, que corresponde ao OU EXCLUSIVO, definida e representada graficamente como segue:

x_1	x_2	XOR $(x_1 \oplus x_2)$
0	0	0
0	1	1
1	0	1
1	1	0



A figura 3.6 mostra a tabela-verdade correspondente ao circuito da figura 3.5, incluindo também

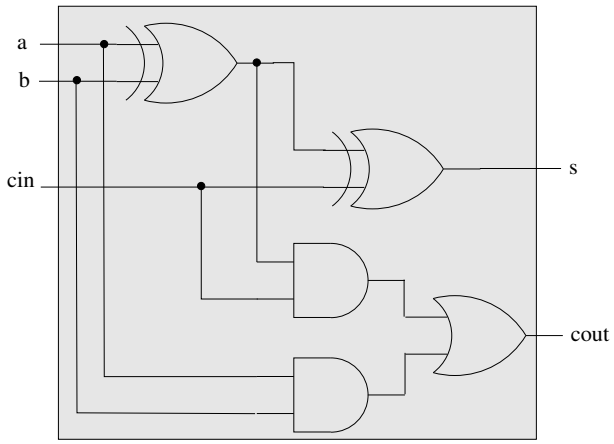


Figura 3.5: Um circuito somador de *bits* que utiliza duas portas XOR, duas porta E, e uma porta OU.

colunas para a saída de cada uma das cinco portas lógicas (não apenas para as portas que geram as saídas do circuito).

c_{in}	a	b	y_1 $a \oplus b$	y_2 $y_1 \cdot c_{in}$	y_3 ab	s	c_{out}
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0
0	1	0	1	0	0	1	0
0	1	1	0	0	1	0	1
1	0	0	0	0	0	1	0
1	0	1	1	1	0	0	1
1	1	0	1	1	0	0	1
1	1	1	0	0	1	1	1

$$s = y_1 \oplus c_{in} = (a \oplus b) \oplus c_{in}$$

$$c_{out} = y_2 + y_3 = y_1 c_{in} + ab = (a \oplus b) c_{in} + ab$$

Figura 3.6: Tabela-verdade do circuito da figura 3.5.

Como podemos ver, obtemos uma tabela igual ao da tabela 3.1. Isso significa que ambos os circuitos realizam a mesma função e portanto são equivalentes. As expressões de s e c_{out} podem ser derivadas diretamente do circuito:

$$s = (a \oplus b) \oplus c \tag{3.3}$$

$$c_{out} = (a \oplus b)\, c_{in} + a\, b \tag{3.4}$$

Ao examinarmos as expressões, não é imediato que elas são equivalentes (i.e., correspondem a uma mesma função). Logo, é interessante sermos capazes de "saber se duas expressões são equivalentes". Será necessário escrever a tabela-verdade delas e compará-las ?

Uma das formas para se mostrar a equivalência de duas expressões é mostrando que uma pode ser derivada a partir da outra aplicando-se manipulações algébricas que não alteram o valor da expressão. Adiantamos aqui um pouco do que veremos no próximo capítulo. Mostraremos como derivar as expressões nas equações 3.3 e 3.4 a partir das expressões nas equações 3.1 e 3.2. Para simplificar a notação, usamos c em lugar de c_{in} e organizamos as variáveis na ordem a, b, c.

$$\begin{aligned}
s(a,b,c) &= \overline{a}\, b\, \overline{c} + a\, \overline{b}\, \overline{c} + \overline{a}\, \overline{b}\, c + a\, b\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, \overline{c} + (\overline{a}\, \overline{b} + a\, b)\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, \overline{c} + (\overline{a}\, \overline{b} + a\, b + 0 + 0)\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, \overline{c} + (\overline{a}\, \overline{b} + a\, b + \overline{a}\, a + b\, \overline{b})\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, \overline{c} + (\overline{a}\, a + \overline{a}\, \overline{b} + b\, a + b\, \overline{b})\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, \overline{c} + (\overline{a}\, (a + \overline{b}) + b\, (a + \overline{b}))\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, \overline{c} + ((\overline{a} + b)\, (a + \overline{b}))\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, \overline{c} + \overline{\overline{((\overline{a} + b)\, (a + \overline{b}))}}\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, \overline{c} + (\overline{a\, \overline{b} + \overline{a}\, b})\, c \\
&= (a \oplus b)\, \overline{c} + (\overline{a \oplus b})\, c \\
&= (a \oplus b) \oplus c
\end{aligned}$$

Similarmente, para a equação do *carry-out* temos:

$$\begin{aligned}
c_{out}(a,b,c) &= a\, b\, \overline{c} + \overline{a}\, b\, c + a\, \overline{b}\, c + a\, b\, c \\
&= \overline{a}\, b\, c + a\, \overline{b}\, c + a\, b\, \overline{c} + a\, b\, c \\
&= (\overline{a}\, b + a\, \overline{b})\, c + a\, b\, (\overline{c} + c) \\
&= (a \oplus b)\, c + a\, b
\end{aligned}$$

As manipulações algébricas acima seguem propriedades da álgebra booleana, que será coberta no próximo capítulo.

Exercícios

1. Sejam A e B dois conjuntos finitos não vazios. Se supormos que A contém n_a elementos e B contém n_b elementos, qual é o número de elementos (pares) no conjunto $A \times B$?

2. Dada a expressão $f(a,b,c) = a\,b + a\,\overline{c}$, escreva a tabela-verdade de f (note que a tabela deve possuir $2^3 = 8$ linhas).

3. Sejam 4 variáveis binárias a, b, c, d. Para quais valores dessas variáveis o produto (E lógico) $\overline{a}\,b\,\overline{c}\,d$ toma valor 1?

4. Sejam 4 variáveis binárias a, b, c, d. Para quais valores dessas variáveis a soma (OU lógico) $a + b + \overline{c} + \overline{d}$ toma valor 0?

5. Sejam 3 variáveis binárias a, b, c. Imagine uma função que toma valor 1 apenas quando $a = 1$, $b = 0$, e $c = 1$, e toma valor 0 para qualquer outra atribuição de valores para essas variáveis. Escreva a expressão lógica (produto) que corresponde a essa função.

6. Quantas funções binárias de $\{0,1\}^2$ em $\{0,1\}$ existem? Escreva todas elas.

7. Desenhe o circuito lógico correspondente à função

$$f(a,b,c) = \overline{a}b + \overline{(a+c)}$$

   Escreva também a tabela-verdade correspondente a f. Inclua na tabela uma coluna para os valores de cada uma das seguintes subexpressões: $\overline{a}b$, $a + c$, $\overline{(a+c)}$.

8. Dada a tabela-verdade abaixo, escreva uma expressão para $f(a,b,c)$ que seja equivalente à tabela.

Entrada			Saída
a	b	c	$f(a,b,c)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

9. Sejam a e b variáveis lógicas. Seu professor pediu para você demonstrar que $\overline{a} + \overline{b}$ é igual a $\overline{a\,b}$. Apresente a sua demonstração e explique.

Capítulo 4

Álgebra booleana

Última atualização em 27/06/2023

O termo "booleana" em álgebra booleana é em homenagem ao matemático inglês George Boole (1815-1864). George Boole foi um dos primeiros a considerar um tratamento sistemático do pensamento [Boole, 1854]. Em outras palavras, ele buscou criar um tratamento do raciocício que fosse similar à representação de cálculos numéricos por meio de processamentos simbólicos. O sistema algébrico resultante é a **álgebra booleana**.

Além da aplicação em lógica, a álgebra booleana tem papel fundamental na modelagem de computadores eletrônicos. A relação entre álgebra booleana e sistemas digitais foi estabelecida por Claude Shannon na década de 1930, quando ele percebeu que as propriedades de circuitos de chaveamentos (redes interconexas de dispositivos do tipo liga-desliga) eram similares aos da álgebra booleana [Shannon, 1938].

Neste capítulo veremos uma definição formal de álgebra booleana, que é baseada em um conjunto de axiomas (ou postulados). Veremos também algumas leis ou propriedades de álgebras booleanas e que todas essas leis podem ser derivadas algebricamente a partir dos postulados da definição. Alguns exemplos de álgebras booleanas são listados.

Referências para esta parte do curso: [Hill and Peterson, 1981], [Garnier and Taylor, 1992], [Whitesitt, 1961] entre outros.

4.1 Definição axiomática de álgebra booleana

Seja uma sêxtupla $\langle A, +, \cdot, \bar{}, 0, 1\rangle$ na qual A é um conjunto, $+$ e $\cdot$ são operações binárias sobre A, $\bar{}$ é uma operação unária em A e 0 e 1 são dois elementos distintos em A. O sistema algébrico $\langle A, +, \cdot, \bar{}, 0, 1\rangle$ é uma **álgebra booleana** se os seguintes axiomas são satisfeitos:

A1. As operações $+$ e $\cdot$ são **comutativas**, ou seja, para todo x e y em A,

$$x + y = y + x \quad \text{e} \quad x \cdot y = y \cdot x$$

A2. Cada operação é **distributiva** sobre a outra, isto é, para todo x, y e z em A,

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z) \quad \text{e} \quad x + (y \cdot z) = (x + y) \cdot (x + z)$$

A3. Os elementos 0 e 1 são os **elementos identidades**, ou seja, para todo $x \in A$,

$$x + 0 = x \quad \text{e} \quad x \cdot 1 = x$$

A4. Todo elemento $x \in A$ possui um **complemento**, ou seja, existe um elemento $\overline{x}$ em A tal que

$$x + \overline{x} = 1 \quad \text{e} \quad x \cdot \overline{x} = 0\,.$$

Observação 1: Na literatura encontramos outras definições para álgebra booleana. Em geral, as definições incorporam um maior número de propriedades. Vale registrar que os postulados acima apresentados, elaborados por Huntington em 1904, correspondem a um conjunto minimal de postulados, isto é, nenhum deles pode ser derivado a partir dos demais. Mais ainda, é um conjunto completo no sentido de que qualquer outra propriedade de uma álgebra booleana pode ser derivada a partir desses postulados. Desta forma, qualquer sistema algébrico que satisfaz os 4 axiomas acima é uma álgebra Booleana. Mais adiante mostraremos como a propriedade associativa (frequentemente incorporada à definição de álgebra booleana) e várias outras podem ser derivadas a partir dos postulados acima.

Observação 2: Pode-se fazer um paralelo com a álgebra elementar dos números. Por exemplo, sobre o conjunto dos números reais, define-se as operações de adição, subtração, etc. Essas operações satisfazem algumas propriedades (por exemplo, a adição é comutativa). Enquanto na álgebra booleana temos a noção de complemento, na álgebra elementar temos a noção de oposto (em relação à adição) e de inverso (em relação à multiplicação).

4.2 Exemplos de álgebra booleana

Exemplo 4.2.1. *O conjunto* $B = \{0, 1\}$ *com as definições*

$$\overline{1} = 0 \quad \overline{0} = 1$$

$$1 \cdot 1 = 1 + 1 = 1 + 0 = 0 + 1 = 1$$

$$0 + 0 = 0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0$$

é uma álgebra booleana.

Os axiomas A1, A3 e A4 são satisfeitos por definição. Para verificar o axioma A2, dados três elementos quaisquer x, y *e* z *em* B, *podemos construir uma tabela verdade para todas as possíveis combinações de valores para* x, y *e* z. *Vejamos, nas colunas indicadas com * na parte inferior da tabela, a validade da distributividade em relação a* $\cdot$, *ou seja, que* $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.

x	y	z	$(y+z)$	$x \cdot (y+z)$	$(x \cdot y)$	$(x \cdot z)$	$(x \cdot y) + (x \cdot z)$
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0	*0*	*1*	*1*	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*0*	*0*	*0*	*0*
0	*1*	*1*	*1*	*0*	*0*	*0*	*0*
1	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	*1*	*1*	*1*	*0*	*1*	*1*
1	*1*	*0*	*1*	*1*	*1*	*0*	*1*
1	*1*	*1*	*1*	*1*	*1*	*1*	*1*
				*			*

Denotamos esta álgebra booleana por $\langle B, +, \cdot, \bar{}, 0, 1\rangle$. *Esta é a álgebra que está por trás dos circuitos lógicos.*

Exemplo 4.2.2. *O cálculo proposicional é um campo da lógica matemática que estuda proposições, ou seja, afirmações que ou são verdadeiras (V) ou são falsas (F), mas não ambas. As proposições podem ser conectadas usando-se os conectivos lógicos E, OU e NÃO, dando origem a novas proposições. Os conectivos lógicos podem ser representados pelos símbolos conforme tabela a seguir.*

Conectivo	*símbolo*
E	$\wedge$
OU	$\vee$
NÃO	$\neg$

Supondo que x e y são duas proposições quaisquer, define-se essas operações conforme a tabela-verdade a seguir:

x	$\neg x$
F	*V*
V	*F*

x	y	$x \wedge y$
F	*F*	*F*
F	*V*	*F*
V	*F*	*F*
V	*V*	*V*

x	y	$x \vee y$
F	*F*	*F*
F	*V*	*V*
V	*F*	*V*
V	*V*	*V*

Qualquer semelhança com as operações lógicas vistas no contexto de circuitos lógicos não é mera coincidência. De fato, a lógica (ou cálculo) proposicional é uma álgebra booleana e ela tem uma correspondência um-para-um com $\langle B, +, \cdot, \bar{}, 0, 1\rangle$, visto acima, conforme apontado a seguir.

Lógica proposicional	*álgebra booleana B*
$\vee$	$+$
$\wedge$	$\cdot$
F	*0*
V	*1*
$\neg x$	$\overline{x}$

Como consequência, temos também a correspondência entre as tabelas-verdade das operações $\neg$, $\vee$, $\wedge$ com as tabelas-verdade das operações : $\bar{}$, $+$ e $\cdot$.

x	y	$\neg x$	$x \vee y$	$x \wedge y$
F	*F*	*V*	*F*	*F*
F	*V*	*V*	*V*	*F*
V	*F*	*F*	*V*	*F*
V	*V*	*F*	*V*	*V*

x	y	$\overline{x}$	$x + y$	$x \cdot y$
0	*0*	*1*	*0*	*0*
0	*1*	*1*	*1*	*0*
1	*0*	*0*	*1*	*0*
1	*1*	*0*	*1*	*1*

Exemplo 4.2.3. *O conjunto $B^n = B \times B \times \ldots \times B$, com as operações $+$, $\cdot$ e $\bar{}$ herdadas de B e definidas, para quaisquer $(x_1, x_2, \ldots, x_n), (y_1, y_2, \ldots, y_n) \in B^n$, da seguinte forma*

$$(x_1, x_2, \ldots, x_n) + (y_1, y_2, \ldots, y_n) = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$$

$$(x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n) = (x_1 \cdot y_1, x_2 \cdot y_2, \ldots, x_n \cdot y_n)$$

$$\overline{(x_1, x_2, \ldots, x_n)} = (\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$$

e com

$$\mathbf{0} = (0, 0, \ldots, 0)$$

$$\mathbf{1} = (1, 1, \ldots, 1)$$

é uma álgebra booleana.

Mostremos que a segunda igualdade $x + (y \cdot z) = (x + y) \cdot (x + z)$ do Axioma 2, distributividade, é satisfeta. Para simplificar a notação, omitiremos a operação $\cdot$ e consideraremos $n = 2$, porém note que o mecanismo aplica-se a qualquer $n > 1$. Nas passagens a seguir, todas as igualdades, exceto a marcada com "A2" (axioma 2), são derivadas aplicando-se a definição das operações $+$ ou $\cdot$ definidas sobre B^2.

$$\begin{aligned}
(x_1, x_2) + (y_1, y_2)(z_1, z_2) &= (x_1, x_2) + (y_1\, z_1, y_2\, z_2) \\
&= (x_1 + y_1\, z_1\,,\; x_2 + y_2\, z_2) \\
&\overset{A2}{=} \Big((x_1 + y_1)\,(x_1 + z_1)\,,\; (x_2 + y_2)\,(x_2 + z_2)\Big) \\
&= \Big((x_1 + y_1)\,,\; (x_2 + y_2)\Big)\,\Big((x_1 + z_1)\,,\; (x_2 + z_2)\Big) \\
&= \Big((x_1, x_2) + (y_1, y_2)\Big)\,\Big((x_1, x_2) + (z_1, z_2)\Big)
\end{aligned}$$

Verifique que os demais axiomas também são satisfeitos.

Exemplo 4.2.4. *Dado um conjunto S, $\mathcal{P}(S)$ denota o conjunto das partes de S (também chamado conjunto potência de S), isto é, $\mathcal{P}(S) = \{X : X \subseteq S\}$. Se S possui k elementos, o conjunto $\mathcal{P}(S)$ possui 2^k elementos (por quê ?)*

Sobre conjuntos temos as operações bem conhecidas de união, interseção e complemento, denotadas respectivamente por $\cup$, $\cap$ e $()^c$. Temos que $\langle \mathcal{P}(S), \cup, \cap, ^c, \emptyset, S\rangle$ é uma álgebra booleana.

Conforme já estamos familiarizados, as propriedades da álgebra de conjuntos equivalentes aos 4 postulados da definição de álgebra booleana são:

A1. $X \cup Y = Y \cup X \quad$ e $\quad X \cap Y = Y \cap X$

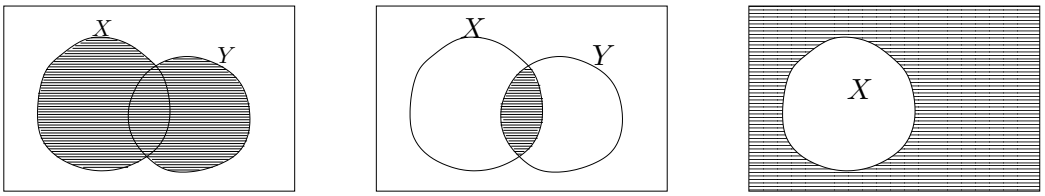
A2. $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z) \quad$ e $\quad X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z)$

A3. $\emptyset \cup X = X \quad$ e $\quad U \cap X = X$

A4. $X \cap X^c = \emptyset \quad$ e $\quad X \cup X^c = U$

Dessas propriedades, a única que pode não ser trivial é a A2. Para se convencer da validade dessas proriedades, pode-se recorrer aos **diagramas de Venn**.

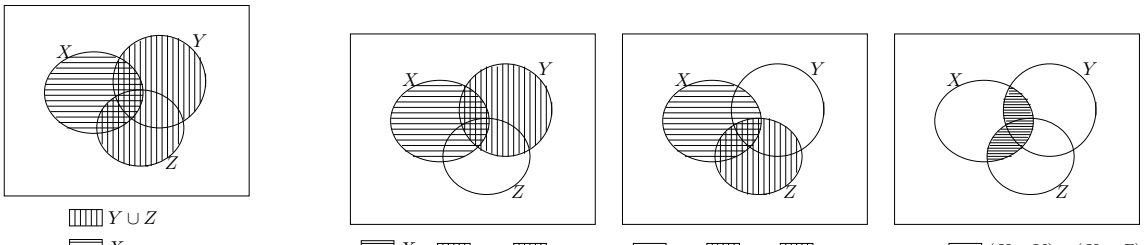
No diagrama de Venn, o conjunto universo é representado por um retângulo, mais precisamente, pelos pontos interiores ao retângulo. Qualquer conjunto é desenhado como sendo uma curva fechada, inteiramente contida no retângulo. Pontos interiores à curva correspondem aos elementos do conjunto. No exemplo da figura 4.1, a união e interseção de dois conjuntos genéricos estão representadas pelas regiões hachuradas das figuras 4.1a e 4.1b, respectivamente. O complemento de um conjunto é representado no diagrama da figura 4.1c.



(a) $X \cup Y$ (b) $X \cap Y$ (c) X^c

Figura 4.1: Diagramas de Venn: (a) união de dois conjuntos, (b) interseção de dois conjuntos, e (c) complemento de um conjunto.

Como exemplo, vamos verificar a propriedade $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$. O conjunto $X \cap (Y \cup Z)$ corresponde à região hachurada pelas linhas verticais e pelas linhas horizontais na figura 4.2a. Esta coincide com a região hachurada no diagrama mais à direita da figura 4.2b, que representa o conjunto $(X \cap Y) \cup (X \cap Z)$.



(a) (b)

Figura 4.2: (a) $X \cap (Y \cup Z)$ e (b) $(X \cap Y) \cup (X \cap Z)$.

Observação: Seja S um conjunto com n elementos. Note que qualquer subconjunto de S pode ser representado por um elemento de B^n. Por exemplo, se $S = \{a, b, c\}$ então o subconjunto

$\{a, c\}$ pode ser representado por 101 e o subconjunto $\{a\}$ por 100 e assim por diante.

4.3 Princípio da dualidade

Vimos que podemos escrever expressões envolvendo elementos de uma álgebra booleana A. Por exemplo, $a \cdot b \cdot c + a \cdot b \cdot \overline{c}$ é uma forma para se expressar um elemento de A (da mesma forma que (2 + 3)*4 expressa o 20 no caso de inteiros). As expressões podem ser manipuladas algebricamente aplicando-se as propriedades da álgebra booleana (por exemplo, $a \cdot b \cdot c + a \cdot b \cdot \overline{c} = a \cdot b \cdot (c + \overline{c}) = a \cdot b \cdot (1) = a \cdot b$). A expressão resultante após a aplicação sucessiva de propriedades, transformando uma expressão em outra, corresponde ao mesmo elemento representado pela expressão inicial.

Desde que sejam sempre aplicadas propriedades válidas em A (tais como as igualdades dadas pelos quatro axiomas da definição de álgebra booleana), a identidade do elemento representado pela expressão inicial se mantém. Desta forma, quando escrevemos $a \cdot b \cdot c + a \cdot b \cdot \overline{c} = a \cdot b \cdot (c + \overline{c}) = a \cdot b \cdot (1) = a \cdot b$, significa que as quatro expressões (a primeira $a \cdot b \cdot c + a \cdot b \cdot \overline{c}$, a segunda $a \cdot b \cdot (c + \overline{c})$, a terceira $a \cdot b \cdot (1)$, e a quarta $a \cdot b$) correspondem a um mesmo elemento em A. Mais ainda, essas identidades algébricas (por exemplo, $a \cdot b \cdot c + a \cdot b \cdot \overline{c} = a \cdot b$, assim como as outras identidades acima) valem para quaisquer três elementos $a, b, c \in A$.

A expressão **dual** de uma expressão qualquer é obtida trocando-se todas as ocorrências de + por ·, todas as ocorrências de · por +, todas as ocorrências de 0 por 1, e todas as ocorrências de 1 por 0.

O **princípio da dualidade** da álgebra booleana afirma que se uma identidade algébrica é válida, então a identidade dual também é válida. Isto é, se conseguimos estabelecer uma identidade algébrica $E_1 = E_2$, automaticamente podemos afirmar que $dual(E_1) = dual(E_2)$.

Por exemplo, seja $E_1 = (0 + a) \cdot b$. Aplicando os axiomas A1 e A3 em sequência, temos $(0 + a) \cdot b = (a + 0) \cdot b = a \cdot b$. Se denotarmos $E_2 = a \cdot b$, o que acabamos de fazer é estabelecer a identidade $E_1 = E_2$. O princípio da dualidade afirma que $dual(E_1) = dual(E_2)$, que neste exemplo corresponderia à identidade $(1 \cdot a) + b = a + b$. De fato, temos $dual(E_1) = (1 \cdot a) + b$, e também que $(1 \cdot a) + b = (a \cdot 1) + b = a + b$. Como $a + b = dual(a \cdot b)$, temos de fato que $dual(E_1) = dual(E_2)$.

A explicação para a validade desse princípio é o fato de que as propriedades definidas pelos quatro axiomas aparecem aos pares, formados por uma identidade e o respectivo dual, conforme pode ser visto a seguir.

Axioma A1	$\begin{array}{ccccccc} x & \cdot & y & = & y & \cdot & x \\ & \downarrow & & & & \downarrow & \\ x & + & y & = & y & + & x \end{array}$
Axioma A2	$\begin{array}{ccccccccccccc} x & \cdot & (y & + & z) & = & (x & \cdot & y) & + & (x & \cdot & z) \\ & \downarrow & & \downarrow & & & & \downarrow & & \downarrow & & \downarrow & \\ x & + & (y & \cdot & z) & = & (x & + & y) & \cdot & (x & + & z) \end{array}$
Axioma A3	$\begin{array}{ccccc} x & + & 0 & = & x \\ & \downarrow & \downarrow & & \\ x & \cdot & 1 & = & x \end{array}$
Axioma A4	$\begin{array}{ccccc} x & + & \overline{x} & = & 1 \\ & \downarrow & & & \downarrow \\ x & \cdot & \overline{x} & = & 0 \end{array}$

Obs.: Uma confusão comum é pensar que uma expressão é equivalente ao seu dual. Isto em geral não é verdade. Por exemplo, o dual de $1 \cdot 1$ é $0 + 0$. Sabemos (pelo axioma A3) que $1 \cdot 1 = 1 \neq 0 = 0 + 0$.

4.4 Leis fundamentais da álgebra booleana

Desta parte em diante omitiremos o símbolo $\cdot$ na maioria das vezes; em vez de $x \cdot y$, escreveremos simplesmente xy. Suponha que $\langle A, +, \cdot, \bar{}, 0, 1\rangle$ é uma álgebra booleana. Então, as seguintes propriedades são válidas.

Unicidade dos elementos identidade 0 e 1: Os elementos 0 e 1 são únicos.

Provamos a unicidade do 0. A unicidade do 1 pode ser provada de forma similar, apoiando-se no princípio da dualidade.

Sejam dois elementos zero, 0_1 e 0_2. Por A3, temos que para quaisquer x_1 e x_2 em A,

$$x_1 + 0_1 = x_1 \quad \text{e} \quad x_2 + 0_2 = x_2$$

Logo, em particular para $x_1 = 0_2$ e $x_2 = 0_1$, temos que

$$0_2 + 0_1 = 0_2 \quad \text{e} \quad 0_1 + 0_2 = 0_1$$

Por A1 temos que $0_2 + 0_1 = 0_1 + 0_2$ e então, pela transitividade de $=$, resulta que $0_1 = 0_2$.

Idempotência: Para todo elemento $x \in A$, $x + x = x$ e $xx = x$.

$$\begin{array}{rcll} x + x & = & (x + x) \cdot 1 & (A3) \\ & = & (x + x)\,(x + \overline{x}) & (A4) \\ & = & x + x\,\overline{x} & (A2) \\ & = & x + 0 & (A4) \\ & = & x & (A3) \end{array} \qquad \begin{array}{rcll} xx & = & x\,x + 0 & (A3) \\ & = & x\,x + x\,\overline{x} & (A4) \\ & = & x\,(x + \overline{x}) & (A2) \\ & = & x \cdot 1 & (A4) \\ & = & x & (A3) \end{array}$$

Elementos anuladores (ou absorventes): Para todo $x \in A$, $x + 1 = 1$ e $x\,0 = 0$.

(Elemento anulador ou absorvente é definido em relação a uma operação binária como sendo aquele que quando operado com qualquer outro elemento resulta nele mesmo)

$$\begin{array}{rcll} x + 1 & = & 1 \cdot (x + 1) & (A3) \\ & = & (x + \overline{x})\,(x + 1) & (A4) \\ & = & x + \overline{x} \cdot 1 & (A2) \\ & = & x + \overline{x} & (A3) \\ & = & 1 & (A4) \end{array}$$

Complemento do um (zero): $\overline{1} = 0$ e $\overline{0} = 1$.

$$\begin{array}{rcll} \overline{1} & = & \overline{1} \cdot 1 & (A3) \\ & = & 0 & (A4) \end{array}$$

Absorção: Para todo $x, y \in A$, $x + x\,y = x$ e $x\,(x + y) = x$.

$$\begin{array}{rcll} x + xy & = & x \cdot 1 + x\,y & (A3) \\ & = & x\,(1 + y) & (A2) \\ & = & x \cdot 1 & (\text{Anulação}) \\ & = & x & (A3) \end{array}$$

Unicidade do complemento $\overline{x}$: O complemento de qualquer elemento $x \in A$ é único.

A definição de complemento é dada pelo axioma A4 (o complemento de um elemento x, denotado $\overline{x}$, é um elemento tal que $x + \overline{x} = 1$ e $x\,\overline{x} = 0$).

Suponha que existem elementos $\overline{x}_1$ e $\overline{x}_2$ em A satisfazendo

$$x + \overline{x}_1 = 1 \quad \text{e} \quad x + \overline{x}_2 = 1 \quad \text{e} \quad x\,\overline{x}_1 = 0 \quad \text{e} \quad x\,\overline{x}_2 = 0$$

Então, temos que

$$\begin{array}{rcll} \overline{x}_2 & = & 1 \cdot \overline{x}_2 & (A3) \\ & = & (x + \overline{x}_1)\,\overline{x}_2 & \text{(hipótese)} \\ & = & x\,\overline{x}_2 + \overline{x}_1\,\overline{x}_2 & (A2) \\ & = & 0 + \overline{x}_1\,\overline{x}_2 & \text{(hipótese)} \\ & = & x\,\overline{x}_1 + \overline{x}_1\,\overline{x}_2 & \text{(hipótese)} \\ & = & (x + \overline{x}_2)\,\overline{x}_1 & (A2) \\ & = & 1 \cdot \overline{x}_1 & \text{(hipótese)} \\ & = & \overline{x}_1 & (A3) \end{array}$$

Involução: Para todo $x \in A$, $\overline{\overline{x}} = x$.

Primeiramente, note que $\overline{x}$ denota o complemento de x. Assim, o elemento $\overline{\overline{x}}$ denota o complemento de $\overline{x}$. Para simplificar a notação, denotemos $\overline{\overline{x}}$ por y. Então, por A4 temos que (i) $\overline{x}\,y = 0$ e $\overline{x} + y = 1$ (i.e., y é um complemento de $\overline{x}$).

Agora, por A4 temos que $x\,\overline{x} = 0$ e por A1 que $x\,\overline{x} = \overline{x}\,x$, e portanto temos que $x\,\overline{x} = \overline{x}\,x = 0$. Similarmente, temos que $x + \overline{x} = \overline{x} + x = 1$. Isto é, (ii) x é um complemento de $\overline{x}$ (pois x satisfaz A3 em relação a $\overline{x}$).

De (i) e (ii) e por causa da unicidade do complemento (e aqui estamos pensando no complemento de $\overline{x}$), podemos concluir que $\overline{\overline{x}} = y = x$.

Associatividade: Para quaisquer $x, y, z \in A$, $x + (y + z) = (x + y) + z$ e $x(y\,z) = (x\,y)\,z$.

Primeiramente, provaremos uma identidade que será utilizada posteriormente.

[Lema] Para quaisquer $x, y, z \in A$, $\quad x\,[(x + y) + z] = [(x + y) + z]\,x = x$.

Por um lado, aplicando A1, temos

$$x\,[(x+y)+z] = [(x+y)+z]\,x$$

Por outro lado, aplicando inicialmente A2, temos

$$\begin{array}{rcll} x\,[(x+y)+z] & = & x\,(x+y)+x\,z & (A2) \\ & = & x+x\,z & (\text{absorção}) \\ & = & x & (\text{absorção}) \end{array}$$

Dessas duas identidades segue o lema acima.

Usando o lema acima, provaremos a propriedade associativa. Para isso definimos inicialmente uma expressão Z, conforme abaixo:

$$Z = [(x+y)+z]\,[x+(y+z)]$$

A partir de Z, podemos fazer duas expansões que nos levarão à identidade desejada:

1. usando A, B, C indicados em $Z = \underbrace{[(x+y)+z]}_{A}\,[\underbrace{x}_{B} + \underbrace{(y+z)}_{C}]$, temos $Z = AB + AC$
2. usando U, V, W indicados em $Z = [\underbrace{(x+y)}_{U} + \underbrace{z}_{V}]\,\underbrace{[x+(y+z)]}_{W}$, temos $Z = UW + VW$

Primeira expansão:

$$\begin{array}{rcll} Z & = & \underbrace{[(x+y)+z]}_{A}\,[\underbrace{x}_{B} + \underbrace{(y+z)}_{C}] & \\ & = & [(x+y)+z]\,x + [(x+y)+z]\,(y+z) & (A2) \\ & = & x + [(x+y)+z]\,(y+z) & (lema) \\ & = & x + \{[(x+y)+z]\,y + [(x+y)+z]\,z\} & (A2) \\ & = & x + \{[(y+x)+z]\,y + z\} & (A1 + \text{Absorção}) \\ & = & x + (y+z) & (lema) \end{array}$$

De forma similar, para a segunda expansão:

$$\begin{array}{rcll} Z & = & [\underbrace{(x+y)}_{U} + \underbrace{z}_{V}]\,\underbrace{[x+(y+z)]}_{W} & \\ & = & (x+y)\,[x+(y+z)] + z\,[x+(y+z)] & (A2) \\ & = & \{x\,[x+(y+z)] + y\,[x+(y+z)]\} + z\,[(y+z)+x] & (A2 + A1) \\ & = & \{x + y\,[x+(y+z)]\} + z\,[(z+y)+x] & (\text{absorção} + A1) \\ & = & \{x + y\,[(y+z)+x]\} + z & (A1 + lema) \\ & = & (x+y)+z & (lema) \end{array}$$

Logo, $x + (y+z) = (x+y) + z$

Teorema de DeMorgan Para quaisquer $x, y \in A$, $\overline{(x+y)} = \overline{x}\,\overline{y}$ e $\overline{x\,y} = \overline{x} + \overline{y}$.

Vamos mostrar que $(x+y) + \overline{x}\,\overline{y} = 1$ e que $(x+y)(\overline{x}\,\overline{y}) = 0$ (isto é, que $\overline{x}\,\overline{y}$ é o complemento de $x+y$).

$$\begin{array}{rcll}
(x+y) + \overline{x}\,\overline{y} & = & [(x+y)+\overline{x}]\,[(x+y)+\overline{y}] & (A2) \\
& = & [\overline{x}+(x+y)]\,[\overline{y}+(x+y)] & (A1) \\
& = & [(\overline{x}+x)+y)]\,[x+(\overline{y}+y)] & (Associativa + A1) \\
& = & 1 \cdot 1 & (A4 + Identidade) \\
& = & 1 & (A3)
\end{array}$$

$$\begin{array}{rcll}
(x+y)\cdot \overline{x}\,\overline{y} & = & x(\overline{x}\,\overline{y}) + y(\overline{y}\,\overline{x}) & (A2 + A1) \\
& = & (x\,\overline{x})\,\overline{y} + (y\,\overline{y})\,\overline{x} & (associativa) \\
& = & 0 + 0 & (A4 + Identidade) \\
& = & 0 & (A3)
\end{array}$$

Portanto, pela unicidade do complemento, podemos concluir que $\overline{(x+y)} = \overline{x}\,\overline{y}$.

A igualdade dual pode ser demonstrada pelo princípio da dualidade, ou usando o fato de que as igualdades acima valem também para $\overline{x}$ e $\overline{y}$ no lugar de x e y. □

Exemplo 4.4.1. *Os axiomas que definem álgebra booleana assim como as propriedades demonstradas acima podem ser utilizadas para provar outras identidades algébricas. Por exemplo, vamos provar que $a + \overline{a}b = a + b$.*

$$\begin{array}{rcl}
a + \overline{a}b & \overset{A2}{=} & (a+\overline{a})(a+b) \\
& \overset{A4}{=} & (1)(a+b) \\
& \overset{A3}{=} & a+b
\end{array}$$

Agora vamos simplificar a expressão $b\,\overline{c}\,(\overline{c}+\overline{c}\,a) + (\overline{a}+\overline{b})(\overline{a}\,b + \overline{a}\,c)$

$$\begin{array}{rcl}
b\,\overline{c}\,\underbrace{(\overline{c}+\overline{c}\,a)}_{absorção} + (\overline{a}+\overline{b})(\underbrace{\overline{a}\,b+\overline{a}\,c}_{A2}) & = & b\,\underbrace{\overline{c}\,\overline{c}}_{idempotência} + \underbrace{(\overline{a}+\overline{b})\,\overline{a}}_{absorção}(b+c) \\
& = & b\,\overline{c} + \overline{a}\,(b+c) \\
& = & \underbrace{b\,\overline{c} + \overline{a}\,b + \overline{a}\,c}_{Teorema\ do\ consenso\ (exercício)} \\
& = & b\,\overline{c} + \overline{a}\,c
\end{array}$$

4.5 Relações de ordem parciais em álgebras booleanas

A álgebra booleana pode ser estudada considerando-se uma outra estrutura algébrica conhecida como reticulado. Reticulados são estruturas matemáticas caracterizadas por um conjunto de elementos, uma relação de ordem parcial definida entre os elementos, e certas propriedades que devem ser satisfeitas por esta relação de ordem parcial. Pode-se mostrar que uma subclasse particular dos reticulados é, na verdade, uma álgebra booleana. Neste texto não iremos abordar reticulados, porém estudaremos ordens parciais pois elas serão úteis para entender alguns conceitos na manipulação de expressões no contexto da álgebra booleana.

4.5.1 Relações de ordem parciais e posets

Uma relação de ordem parcial é um tipo particular de relação binária. Portanto, começamos relembrando relações binárias.

Seja A um conjunto não vazio. Uma **relação binária** R sobre A é um subconjunto de $A \times A$, isto é, $R \subseteq A \times A$. Se $(x, y) \in R$, x está relacionado a y e então denotamos esta relação por xRy (lê-se x-erre-y).

Relação de ordem parcial: Uma relação binária $\leq$ sobre A é uma **ordem parcial** se ela é

1. (reflexiva) $x \leq x$, para todo $x \in A$
2. (anti-simétrica) Se $x \leq y$ e $y \leq x$, então $x = y$, para todo $x, y \in A$
3. (transitiva) Se $x \leq y$ e $y \leq z$ então $x \leq z$, para todo $x, y, z \in A$

Se $\leq$ é uma ordem parcial em A, então a relação $\geq$ definida por, para quaisquer $x, y \in A$, $x \geq y$ se e somente se $y \leq x$, é também uma ordem parcial em A.

Observação: Note que uma relação de equivalência é bem parecida com uma relação de ordem parcial. A diferença está na segunda propriedade: ordens parciais satisfazem anti-simetria, enquanto relações de equivalência satisfazem simetria (i.e., se $x \sim y$ então $y \sim x$, para todo $x, y \in A$).

Conjuntos parcialmente ordenados (posets) Um conjunto A munido de uma relação de ordem parcial $\leq$ é denominado um conjunto parcialmente ordenado (ou *poset*) e denotado por $(A, \leq)$. Se $(A, \leq)$ é um poset, então $(A, \geq)$ também é um poset.

Exemplo 4.5.1. *A relação de ordem $\leq$ usual definida no conjunto dos números reais é uma ordem parcial (na verdade, ela é mais que uma ordem parcial; é uma* **ordem total**, *pois todos os elementos são comparáveis dois a dois). Por outro lado, a relação usual $<$ não é uma ordem parcial pois ela não é reflexiva.*

Exemplo 4.5.2. *A relação de inclusão de conjuntos $\subseteq$ é uma ordem parcial.*

Diagrama de Hasse: Escrevemos $x < y$ quando $x \leq y$ e $x \neq y$. Dado um poset $(A, \leq)$ e $x, y \in A$, dizemos que y cobre x se, e somente se, $x < y$ e não há outro elemento $z \in A$ tal que $x < z < y$. Um diagrama de Hasse do poset $(A, \leq)$ é uma representação gráfica onde vértices representam os elementos de A e dois elementos x e y são ligados por uma aresta se e somente se y cobre x. Em um diagrama de Hasse, os elementos menores (com relação a ordem parcial) são em geral desenhados abaixo dos elementos maiores.

Exemplo 4.5.3. *O diagrama de Hasse do poset $(\mathcal{P}(\{a, b, c\}), \subseteq)$ é mostrado na figura 4.3.*

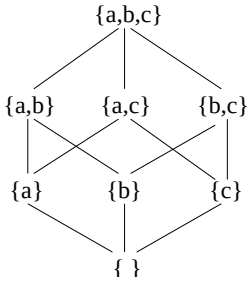


Figura 4.3: Diagrama de Hasse de $(\mathcal{P}(\{a, b, c\}), \subseteq)$.

4.5.2 Uma relação $\leq$ sobre álgebras booleanas

Seja $\langle A, +, \cdot, \bar{}, 0, 1\rangle$ uma álgebra booleana. Seja $\leq$ uma relação binária em A definida da seguinte forma:

$$\forall x, y \in A, \quad x \leq y \ \text{ se e somente se } \ x + y = y \tag{4.1}$$

A relação $\leq$ definida pela equação 4.1 é uma relação de ordem parcial. De fato, a relação $\leq$ é (1) reflexiva pois pela lei de idempotência ($x + x = x$) temos que $x \leq x$ para todo $x \in A$; é (2) anti-simétrica pois se $x \leq y$ e $y \leq x$, então $x+y = y$ e $y+x = x$ e, portanto, pela comutatividade de $+$, segue que $x = y$; e é (3) transitiva pois se $x \leq y$ e $y \leq z$, então

$$\begin{aligned} z &= y + z && \text{(pois } y \leq z\text{)} \\ &= (x + y) + z && \text{(pois } x \leq y\text{)} \\ &= x + (y + z) && \text{(associatividade de } +\text{)} \\ &= x + z && \text{(pois } y \leq z\text{)} \end{aligned}$$

Logo, $x \leq z$.

Note que em uma álgebra booleana, $x + y = y$ se e somente se $x\,y = x$ (a prova é deixada como exercício). Desta forma, podemos definir a relação $\leq$ equivalentemente como

$$\forall x, y \in A, \quad x \leq y \;\text{ se e somente se }\; x\,y = x \tag{4.2}$$

Note também que $x\,y \leq x \leq x + y$ e, em particular, $0 \leq x \leq 1$.

4.6 Átomos e decomposição em termos de átomos

Vimos acima que podemos definir uma relação de ordem parcial $\leq$ sobre qualquer álgebra booleana $\langle A, +, \cdot, \bar{}, 0, 1\rangle$. Também comentamos que vale $0 \leq x \leq 1$, $\forall x \in A$. Ou seja, 0 é o menor elemento em A e 1 é o maior elemento em A (relembre que provamos que os elementos identidade 0 e 1 são únicos).

As álgebras booleanas possuem elementos denominados **átomos**. A importância dos átomos de uma álgebra boolena A está no fato de que qualquer elemento de A pode ser expresso em termos de átomos. Nesta seção definimos o que são átomos e em seguida apresentamos uma demonstração desse fato. O conteúdo desta e da próxima seção é fortemente baseado no livro de Ross e Wright [Ross and Wright, 1992].

4.6.1 Átomo

Um **átomo**[1] de uma álgebra booleana $\langle A, +, \cdot, \bar{}, 0, 1\rangle$ é um elemento não nulo x que não pode ser expresso na forma $x = y + z$ com $y \neq x$ e $z \neq x$.

Exemplo 4.6.1. *Os átomos de* $\langle \mathcal{P}(S), \cup, \cap, ^c, \emptyset, S\rangle$ *são todos os conjuntos unitários.*

[1] Embora aqui estejamos considerando átomos em álgebras booleanas, vale ressaltar que a definição aplica-se, por exemplo, a qualquer reticulado finito.

Exemplo 4.6.2. *A álgebra booleana B^n de todas as n-uplas binárias tem como átomos as n-uplas com exatamente uma coordenada igual a* 1.

O teorema a seguir mostra uma outra caracterização de átomos, fazendo uso da relação de ordem parcial $\leq$ definida sobre a álgebra booleana.

Teorema 4.6.1. *Um elemento não nulo x de uma álgebra booleana $\langle A, +, \cdot, \bar{}, 0, 1\rangle$ é um* **átomo** *se e somente se não há elemento y em A tal que $0 < y < x$.*

Demonstração. Primeiramente, vamos mostrar que se x é um átomo então não há elemento y em A tal que $0 < y < x$. Suponha que x é um átomo e que $y < x$; então $y \leq x$ e $y \neq x$. De $y \leq x$ segue que $y + x = x$ e $y\,x = y$. Por outro lado, como $x = x\,1$, usando $x = y + x$ e $1 = y + \overline{y}$, temos que $x = x\,1 = (y + x)(y + \overline{y}) = y + (x\,\overline{y})$. Já que estamos supondo que x é um átomo, então ou $y = x$ ou $(x\,\overline{y}) = x$. Como $x \neq y$ por hipótese, só podemos ter $(x\,\overline{y}) = x$. Substituindo esta última igualdade em $y = y\,x$, temos $y = y\,(x\,\overline{y}) = y\,(\overline{y}\,x) = (y\,\overline{y})\,x = 0\,x = 0$. Isto prova que se um elemento é menor que um átomo então ele é o 0.

Por outro lado, vamos mostrar que se não há elemento y em A tal que $0 < y < x$ então x é um átomo. Suponha que não há elemento y em A tal que $0 < y < x$ e (suponha por absurdo) que x não é um átomo. Se x não é atomo, então podemos escrever $x = y + z$ com $y \neq x$ e $z \neq x$. De $y \leq y + z$ e $x = y + z$ segue que $y \leq x$. Porém como $y \neq x$, temos então $y < x$. Além disso, como por hipótese não existe $0 < y < x$, só podemos ter $y = 0$. Logo, teríamos $x = 0 + z = z$, contradizendo a suposição de que $z \neq x$.

Como mostramos ambas as implicações, a prova está completa. ■

4.6.2 Teorema de decomposição como soma de átomos

O teorema a seguir diz respeito à decomposição de qualquer elemento não-nulo de uma álgebra booleana como o supremo[2] de um conjunto de átomos.

Teorema 4.6.2. *Seja $\langle A, +, \cdot, \bar{}, 0, 1\rangle$ uma álgebra booleana finita com conjunto de átomos $AT = \{a_1, a_2, \ldots, a_n\}$. Cada elemento não nulo x de A pode ser escrito como o supremo (ou união[3]) de um subconjunto de átomos $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\} \subseteq AT$, isto é,*

$$x = a_{i_1} + a_{i_2} + \cdots + a_{i_k}$$

Mais ainda, tal expressão é única, a menos da ordem dos átomos.

[2] O termo supremo é utilizado aqui para indicar o resultado da operação + aplicada a um conjunto de elementos.
[3] O termo união aqui denota, similarmente a supremo, o resultado da operação +.

Demonstração. Esta demonstração é dividida em três partes.

Na primeira parte demonstramos que qualquer elemento não nulo de A pode ser expresso como uma união de átomos. Se x é átomo, não há nada a demonstrar. Seja x qualquer, que não é nulo e que não é átomo. Vamos mostrar que x pode ser expresso como união de átomos. A prova será por contradição. Suponha por absurdo que x não pode ser expresso como união de átomos. Como x não é átomo, pela definição de átomos, podemos escrever $x = y + z$ com $y \neq x$ e $z \neq x$. De $x = y + z$, temos que $x \overset{\text{idemp.}}{=} y+y+z = y+x \iff y \leq x$. Logo $y < x$ (pois sabemos que $y \neq x$). Usando o mesmo argumento, temos que $z < x$. Como estamos supondo que x não pode ser expresso como união de átomos, então pelo menos y ou z não é átomo. Supondo, sem perda de generalidade, que y não é átomo e usando o mesmo argumento, temos que y é união de dois elementos menores que ele, dos quais pelo menos um não é átomo. Repetindo esse argumento, teremos uma sequência de elementos não-átomos $y_0 = y > y_1 > y_2 > \ldots$. Porém, como A é finito, haverá índices k e m, com $k < m$ tal que $y_k = y_m$. Pela transitividade de $<$, de $y_k > y_{k+1} > \ldots > y_m$ segue que $y_k > y_m$, contradizendo o fato de que $y_k = y_m$. Portanto, podemos concluir que todos os elementos não-nulos em A podem ser expressos como união de átomos.

Na segunda parte exibimos UMA representação de x como união de átomos. Seja $AT' = \{a \in AT : a \leq x\}$. Vamos mostrar que

$$x = \bigvee \{a \in AT'\} \tag{4.3}$$

isto é, que x pode ser expresso como a união de todos os átomos menores ou iguais a ele. Para mostrar isso, note inicialmente que baseado no que acabamos de provar acima, o 1 pode ser expresso como a união de um conjunto de átomos. Como qualquer átomo a é tal que $a \leq 1$, ao acrescentar os demais átomos, continuamos a ter que a soma é 1. Assim, podemos escrever o 1 como a união de todos os átomos em A. Disso, segue que

$$x = x \cdot 1 = x(a_1 + a_2 + \ldots + a_n) = xa_1 + xa_2 + \ldots + xa_n$$

Pela definição de $\leq$, segue que $x\,a_i \leq a_i$. Os únicos elementos que são menores ou iguais a a_i são o 0 ou o próprio a_i. Então temos $x\,a_i = a_i$ ou $x\,a_i = 0$. Portanto, restarão apenas aqueles tais que $x\,a_i = a_i$, ou seja, (pela definição de $\leq$) aqueles tais que $a_i \leq x$, o que resulta na soma mostrada na equação 4.3.

Na terceira parte, mostramos a unicidade da representação. De acordo com a demonstração feita na primeira parte, podemos afirmar que existe uma expressão de x como união de átomos, que escrevemos $x = a_{i_1} + a_{i_2} + \cdots + a_{i_k}$. Por outro lado, na segunda parte mostramos que $x = \bigvee\{a \in AT'\}$. Para mostrar a unicidade, mostraremos que (a) cada um dos átomos da primeira representação está em AT' e, reciprocamente, que (b) cada átomo em AT' está na primeira representação.

(a) Em $x = a_{i_1} + a_{i_2} + \cdots + a_{i_k}$, devemos ter $a_{i_j} \leq x$ (pois $x\,a_{i_j} = a_{i_1}a_{i_j} + \ldots + a_{i_j}a_{i_j} + \ldots + a_{i_k}a_{i_j} = a_{i_j}$). Logo, $a_{i_j} \in AT'$. (b) Por outro lado, seja $a \in AT'$. Então, como a é átomo e $a \leq x$, usando $x = a_{i_1} + a_{i_2} + \cdots + a_{i_k}$, segue que

$$0 \neq a = a\,x = a\,(a_{i_1} + a_{i_2} + \cdots + a_{i_k}) = a\,a_{i_1} + a\,a_{i_2} + \cdots + a\,a_{i_k}$$

Pelo menos um termo $a\,a_{i_j}$ precisa ser diferente de zero para a soma não ser nula. Sabemos que $a\,a_{i_j} \neq$

$0 \Longrightarrow a = a_{i_j}$ pois ambos, a e a_{i_j}, são átomos. Ou seja, a é um dos átomos que aparecem em $a_{i_1} + a_{i_2} + \cdots + a_{i_k}$.

Isto conclui a demonstração. ■

4.7 Isomorfismos entre álgebras booleanas

Um dos resultados mais importantes sobre álgebras booleanas é o teorema que afirma que qualquer álgebra booleana é isomorfa ao conjunto potência (partes de um conjunto, que por sua vez é também uma álgebra booleana conforme exemplo 4.2.4).

Definição 4.7.1. *Sejam* $\langle A_1, +, \cdot, \bar{}, 0_1, 1_1\rangle$ *e* $\langle A_2, +, \cdot, \bar{}, 0_2, 1_2\rangle$ *duas álgebras booleanas. Uma função* $\phi : A_1 \to A_2$ *que satisfaz, para todo* $x, y \in A_1$,

1. $\phi(x + y) = \phi(x) + \phi(y)$
2. $\phi(x \cdot y) = \phi(x) \cdot \phi(y)$
3. $\phi(\overline{x}) = \overline{\phi(x)}$

é um **isomorfismo** *entre as álgebras booleanas* A_1 *e* A_2. *Dizemos então que duas* **álgebras booleanas são isomorfas** *se existe um isomorfismo entre elas.*

Note que da definição acima, se ϕ é um isomorfismo, então tomando x um elemento qualquer em A_1, temos:

$$\phi(0_1) \overset{Axioma4}{=} \phi(x \cdot \overline{x}) \overset{def.}{=} \phi(x) \cdot \phi(\overline{x}) \overset{def.}{=} \phi(x) \cdot \overline{\phi(x)} \overset{Axioma4}{=} 0_2$$

$$\phi(1_1) = \phi(x + \overline{x}) = \phi(x) + \phi(\overline{x}) = \phi(x) + \overline{\phi(x)} = 1_2$$

Vejamos um exemplo de isomorfismo entre álgebras booleanas.

Exemplo 4.7.1. *Sejam* $\langle \mathcal{P}(S), \cup, \cap, ^c, \emptyset, S\rangle$, *com* $S = \{a, b\}$, *e* $\langle B^2, +, \cdot, \bar{}, \mathbf{0}, \mathbf{1}\rangle$, *com* $B^2 = \{(0,0), (0,1), (1,0), (1,1)\}$, $\mathbf{0} = (0,0)$ *e* $\mathbf{1} = (1,1)$, *duas álgebras booleanas.*

Vamos definir uma função $\phi : \mathcal{P}(S) \to B^2$ *e em seguida mostrar que essa função é um isomorfismo. Antes de mais nada, lembrando que* $S = \{a, b\}$, *vamos definir para cada* $X \in$

$\mathcal{P}(S)$, um par $(x_1, x_2) \in B^2$ da seguinte forma:

$$x_1 = 1 \iff a \in X \qquad e \qquad x_2 = 1 \iff b \in X$$

Assim, por exemplo para $X = \{a\}$, temos $(x_1, x_2) = (1, 0)$.

Agora definimos ϕ, para todo $X \in \mathcal{P}(S)$, da seguinte forma

$$\phi(X) = (x_1, x_2)\,.$$

Vamos verificar se essa função é um isomorfismo. Dados $X, Y \in \mathcal{P}(S)$, precisamos verificar se (i) $\phi(X \cup Y) = \phi(X) + \phi(Y)$, (ii) $\phi(X \cap Y) = \phi(X) \cdot \phi(Y)$, e (iii) $\phi(X^c) = \overline{\phi(X)}$.

Para mostrar (i), seja $Z = X \cup Y$. Por definição, $z_1 = 1 \iff a \in Z$ e portanto $z_1 = 1 \iff a \in X$ ou $a \in Y \iff x_1 = 1$ ou $y_1 = 1 \iff x_1 + y_1 = 1$. Por raciocínio análogo, temos também que $z_2 = 1 \iff x_2 + y_2 = 1$. Disto segue que $(z_1, z_2) = (x_1 + y_1, x_2 + y_2) = (x_1, x_2) + (y_1, y_2)$, ou seja, que $\phi(X \cup Y) = \phi(X) + \phi(Y)$.

De modo similar podemos mostrar (ii). Para mostrar (iii), note que $\phi(X^c) = \overline{\phi(X)} = \overline{(x_1, x_2)} = (\overline{x_1}, \overline{x_2})$ e portanto precisamos mostrar que $\overline{x_1} = 1 \iff a \in X^c$ e $\overline{x_2} = 1 \iff b \in X^c$. Como $x_1 = 1 \iff a \in X$, obviamente $x_1 = 0 \iff a \in X^c$, e como $x_1 = 0 \iff \overline{x_1} = 1$ temos que $\overline{x_1} = 1 \iff a \in X^c$. O mesmo raciocínio pode ser usado para mostrar que $\overline{x_2} = 1 \iff b \in X^c$. Com isso mostramos que a função ϕ é um isomorfismo.

No exemplo acima mostramos que as duas álgebras booleanas em questão são isomorfas. Para isso, exibimos um isomorfismo entre elas. O resultado a seguir mostra como construir isomorfismos entre duas álgebras booleanas que possuem o mesmo número de átomos.

Teorema 4.7.1. *Sejam duas álgebras booleanas finitas A_1 e A_2 com o mesmo número de átomos e sejam $AT_1 = \{a_1, a_2, \ldots, a_n\}$ e $AT_2 = \{b_1, b_2, \ldots, b_n\}$, respectivamente, os seus conjuntos de átomos. Então, existe um isomorfismo ϕ entre A_1 e A_2 tal que $\phi(a_i) = b_i$, $i \in \{1, 2, \ldots, n\}$.*

Demonstração. Vamos definir uma função $\phi : A_1 \to A_2$ que satisfaz a condição acima. Primeiramente, definimos $\phi(a_i) = b_i$, $i = 1, 2, \ldots, n$. Para os demais elementos x, vamos usar o fato de que x pode ser expresso de forma única como união de átomos (Teorema 4.6.2), isto é, que $x = a_{i_1} + a_{i_2} + \cdots + a_{i_k}$, e então definimos

$$\phi(x) = \phi(a_{i_1}) + \phi(a_{i_2}) + \cdots + \phi(a_{i_k}) \tag{4.4}$$

(isto está bem definido pois cada a_{i_j}, $j = 1, \ldots, k$, é um átomo em AT_1 e cada $\phi(a_{i_j})$, $j = 1, \ldots, k$, é um átomo em AT_2).

Agora precisamos mostrar que essa função ϕ que acabamos de definir é um isomorfismo entre A_1 e A_2.

Vamos mostrar que um átomo $a \in AT_1$ está na decomposição de x se e somente se $\phi(a)$ está na decomposição de $\phi(x)$, ou seja,

$$a \leq x \iff \phi(a) \leq \phi(x) \tag{4.5}$$

Pela teorema da decomposição como união de átomos (Teorema 4.6.2), sabemos que $x = \bigvee \{a \in AT_1 \,|\, a \leq x\}$ (isto é, x pode ser escrito como a união de todos os átomos em AT_1 que são menores ou iguais a ele). Logo, a partir da eq. 4.4, podemos escrever:

$$\phi(x) = \bigvee \{\phi(a) \,|\, a \in AT_1 \text{ e } a \leq x\} \tag{4.6}$$

Além disso, em A_2 temos que

$$\phi(x) = \bigvee \{b \,|\, b \in AT_2 \text{ e } b \leq \phi(x)\} \tag{4.7}$$

Como a expressão de $\phi(x)$ na forma união de átomos é única, o conjunto de átomos presente na decomposição da equação 4.6 deve ser exatamente o mesmo da equação 4.7. Ou seja, cada um dos átomos b na segunda decomposição corresponde necessariamente a um dos $\phi(a)$ na primeira (e vice-versa). Isto significa que um átomo $a \in AT_1$ está na decomposição de x se e somente se $\phi(a)$ está na decomposição de $\phi(x)$, que pode ser expresso conforme a equivalência 4.5.

Para mostrar que $\phi(x + y) = \phi(x) + \phi(y)$, primeiramente note que para qualquer $a \in AT_1$ e $x, y \in A_1$ temos que

$$\begin{aligned} \phi(a) \leq \phi(x + y) &\overset{\text{eq. 4.5}}{\iff} a \leq x + y \\ &\overset{*}{\iff} a \leq x \text{ ou } a \leq y \\ &\overset{\text{eq. 4.5}}{\iff} \phi(a) \leq \phi(x) \text{ ou } \phi(a) \leq \phi(y) \end{aligned}$$

[Vamos aceitar $*$ por enquanto]

Na equivalência acima, $\phi(a)$ é um átomo em AT_2. Denotando por b um átomo em AT_2 temos então que $b \leq \phi(x + y) \iff b \leq \phi(x)$ ou $b \leq \phi(y) \overset{*}{\iff} b \leq \phi(x) + \phi(y)$.

Ou seja, o conjunto dos átomos menores ou iguais a $\phi(x + y)$ é o mesmo dos menores ou iguais $\phi(x) + \phi(y)$. Logo, pelo teorema da decomposição como soma de átomos (Teorema 4.6.2) segue que $\phi(x + y) = \phi(x) + \phi(y)$.

Para provar a igualdade $\phi(x \cdot y) = \phi(x) \cdot \phi(y)$ podemos proceder de forma análoga.

Finalmente, para mostrar que $\phi(\overline{x}) = \overline{\phi(x)}$, note que $\phi(x) + \phi(\overline{x}) = \phi(x + \overline{x}) = \phi(1) = 1$ e $\phi(x) \cdot \phi(\overline{x}) = \phi(x \cdot \overline{x}) = \phi(0) = 0$. Logo, da unicidade do complemento segue que $\overline{\phi(x)} = \phi(\overline{x})$.

■

A prova de que $a \leq x + y \iff a \leq x$ ou $a \leq y$ fica como exercício.

A partir do teorema anterior podemos relacionar qualquer álgebra booleana com a álgebra booleana $\langle \mathcal{P}(S), \cup, \cap, ^c, \emptyset, S \rangle$, S não vazio. A relação $\subseteq$ é uma relação de ordem parcial sobre $\mathcal{P}(S)$, e se S contém n elementos, então $\mathcal{P}(S)$ contém exatamente 2^n elementos e os átomos são os n conjuntos unitários em $(\mathcal{P}(S), \subseteq)$.

Corolário 4.7.1.1. *Qualquer álgebra booleana finita com n átomos é isomorfa à álgebra booleana $\langle \mathcal{P}(S), \cup, \cap, \ ^c, \emptyset, S \rangle$ na qual S é um conjunto com n elementos.*

Exercícios

1. Mostre que o conjunto B^n mais as operações definidas no exemplo 4 da página 42 é uma álgebra booleana.

2. Considere o conjunto dos números reais $\mathbb{R}$, juntamente com as operações usuais de adição e multiplicação. Quais dos axiomas A1, A2, A3 não são satisfeitos ? É possível definir uma operação unária em $\mathbb{R}$ tal que o axioma A4 seja satisfeito ?

3. Explique o que é o princípio da dualidade.

4. Seja uma álgebra booleana $\langle A, +, \cdot, \bar{}, 0, 1 \rangle$ qualquer. Prove ou mostre um contra-exemplo: $(a + \overline{b})(\overline{a} + b)(\overline{a} + \overline{b}) = \overline{a} + \overline{b}$

5. Seja uma álgebra booleana $\langle A, +, \cdot, \bar{}, 0, 1 \rangle$ qualquer. Sejam x, y, z três elementos de A. Prove, algebricamente, as seguintes igualdades. Justifique as passagens.

   a) $x + \overline{x}\, y = x + y$ (e sua dual $x\,(\overline{x} + y) = x\, y$)

   b) $x + y = \overline{\overline{x}\,\overline{y}}$ (e sua dual $x\, y = \overline{\overline{x} + \overline{y}}$)

   c) $(x + y)\,(x + \overline{y}) = x$ (e sua dual $x\, y + x\,\overline{y} = x$)

   d) (Teorema do consenso) $x\, y + y\, z + \overline{x}\, z = x\, y + \overline{x}\, z$ (ou o dual $(x + y)\,(y + z)\,(\overline{x} + z) = (x + y)\,(\overline{x} + z)$)

e) $y\,x = z\,x$ e $y\,\overline{x} = z\,\overline{x}$ implica que $y = z$

f) $(x + y + z)\,(x + y) = x + y$

6. Sejam a, b, c três elementos de uma álgebra booleana. A seguinte implicação está correta? Explique.

$$a + b = a + c \Longrightarrow b = c$$

7. Simplifique as seguintes expressões

a) $y\,\overline{z}(\overline{z} + \overline{z}\,x) + (\overline{x} + \overline{y})(\overline{x}\,y + \overline{x}\,z)$

b) $x + x\,y\,z + y\,z\,\overline{x} + w\,x + \overline{w}\,x + \overline{x}\,y$

8. Mostre que a relação $\leq$ definida por

$$\forall x, y \in A, \quad x \leq y \;\text{ se e somente se }\; x\,y = x \tag{4.8}$$

é uma relação de ordem parcial.

9. Mostre que em qualquer álgebra booleana $\langle A, +, \cdot, \bar{}, 0, 1\rangle$,

(a) $x + y = y$ se, e somente se, $x\,y = x$

(b) $x\,\overline{y} = 0$ se, e somente se, $x\,y = x$

10. Seja uma álgebra booleana com a relação de ordem parcial $\leq$ usual. Mostre que para quaisquer elementos x e y:

(a) $x\,y \leq x \leq x + y$

(b) $0 \leq x \leq 1$

11. Seja a um átomo e sejam x e y dois elementso quaisquer de uma álgebra booleana. Mostre que

(a) $a \leq x + y \Longleftrightarrow a \leq x$ ou $a \leq y$

(b) $a \leq xy \Longleftrightarrow a \leq x$ e $a \leq y$

(c) ou $a \leq x$ ou $a \leq \overline{x}$, mas não ambos.

12. Existe álgebra booleana com 6 elementos ?

13. Seja $A = \{1, 2, 3, 5, 6, 10, 15, 30\}$, ou seja, o conjunto de divisores de 30. Defina operações binárias $+$ e $\cdot$ e uma operação unária $\bar{}$ da seguinte forma: para quaiquer $a_1, a_2 \in A$,

$$a_1 + a_2 = \text{o mínimo múltiplo comum entre } a_1 \text{ e } a_2$$

$$a_1 \cdot a_2 = \text{o máximo divisor comum entre } a_1 \text{ e } \ a_2$$

$$\overline{a}_1 = 30/a_1$$

Quais são os elementos identidade (neutro) com respeito a + e · ? Mostre que A, com as três operações acima, é uma álgebra booleana.

Dica: considere a decomposição dos elementos de A em fatores primos.

Capítulo 5

Funções Booleanas e formas canônicas

Última atualização em 25/05/2021

No capítulo anterior mostramos algumas propriedades da álgebra booleana. As letras x, y, z, ou ainda, a, b, c, foram usadas para denotar elementos da álgebra booleana. A expressão $a+b$, por exemplo, foi usada para indicar o elemento resultante ao se operar o elemento a com o elemento b por meio da operação $+$. Observe que a e b representam elementos quaisquer da álgebra booleana. Portanto, tanto a como b podem ser vistos como variáveis cujos valores podem ser quaisquer elementos da álgebra booleana. Assim, podemos considerar expressões que envolvem várias variáveis e, ao se atribuir valores (que sejam elementos da álgebra booleana) a essas variáveis, podemos efetuar o "cálculo" do valor da expressão (que também será um elemento da álgebra booleana).

Neste capítulo definimos inicialmente o que são **variáveis** e **expressões booleanas** e em seguida definimos **funções booleanas**. Veremos que funções booleanas podem ser representadas por diferentes expressões e que, dentre as expressões que definem uma mesma função, existem as que são canônicas.

Do ponto de vista desta disciplina, o interesse em expressões e funções booleanas pode ser resumido da seguinte forma: estamos partindo do pressuposto de que computadores são capazes de representar apenas dois valores, que denotamos 0 e 1. Desta forma, para ser manipulado pelo computador, qualquer dado ou informação precisa ser codificado como uma palavra formada apenas por esses valores. Para dados do tipo numérico, a representação dos mesmos

na base binária é a escolha natural. Todos os demais tipos de dados precisam ser codificados numericamente. Estando em formato binário, esses dados podem ser manipulados e processados. Um processamento de dados pode ser pensado como uma transformação de um bloco com uma certa quantidade de *bits* em um outro bloco com uma quantidade (não necessariamente igual) de *bits*. Formalmente, essa transformação pode ser pensada como uma função da forma $f : \{0,1\}^n \to \{0,1\}^m$, ou então uma coleção de m funções binárias $f_i : \{0,1\}^n \to \{0,1\}$.

Assim, dada uma descrição formal da transformação (ou processamento) desejado, uma importante pergunta é se podemos implementá-la em um computador. Mais especificamente, podemos garantir que todas as funções desse tipo podem ser realizadas por circuitos lógicos usando os componentes lógicos cujos equivalentes físicos existem e são viáveis ? Os componentes físicos a que estamos referindo aqui são as portas lógicas. Vimos que as portas lógicas OU e E e o inversor NÃO correspondem respectivamente às operações $+$, $\cdot$ e $^-$ da álgebra booleana $B = \{0,1\}$. Logo, qualquer expressão booleana (que por sua vez define uma função) pode ser realizada por um circuito lógico. Por outro lado, quais são as funções que podem ser definidas por expressões booleanas (ou seja, podem ser realizados por circuitos lógicos?)

Outra pergunta importante relaciona-se com custos. Supondo que uma função possa ser definida por alguma expressão booleana e dado que podem existir várias expressões que definem essa mesma função, qual é a expressão que resultará em uma realização mais econômica?

Este capítulo reúne alguns conceitos e resultados relacionados à representação de funções booleanas e que são úteis para responder as questões acima.

Neste capítulo iremos nos restringir à algebra booleana $\langle B, +, \cdot, ^-, 0, 1\rangle$.

5.1 Expressões e funções booleanas

5.1.1 Expressões booleanas

Variáveis e literais: uma **variável booleana** é uma variável que toma valores em B.

O **complemento** de uma variável booleana x é denotado $\overline{x}$. Para qualquer $a \in B$, se $x = a$ então $\overline{x} = \overline{a}$.

Um **literal** é uma variável booleana x ou o seu complemento $\overline{x}$.

Expressões booleanas: uma expressão booleana em n váriáveis $x_1, x_2, \ldots, x_n$ é qualquer expressão definida de acordo com as seguintes regras:

- os elementos em B são expressões booleanas;
- as variáveis $x_1, x_2, \ldots, x_n$ são expressões booleanas;
- se x e y são expressões booleanas, então são também as expressões $(x) + (y)$, $(x) \cdot (y)$ e $\overline{(x)}$;
- uma expressão é booleana se e somente se pode ser obtida aplicando-se quaisquer das três regras acima um número finito de vezes.

Observe que uma expressão booleana em n váriáveis $x_1, x_2, \ldots, x_n$ não necessariamente precisa conter todas as n variáveis. Parênteses podem ser removidos da expressão desde que não introduzam ambiguidades. Por exemplo, a expressão $(x_1) + (x_2)$ pode ser escrita $x_1 + x_2$.

Se uma expressão pode ser derivada a partir de outra aplicando-se um número finito de vezes as propriedades da álgebra booleana, então elas são ditas **equivalentes**.

5.1.2 Funções booleanas

Dada uma expressão booleana em n variáveis x_1, x_2, ..., x_n, e uma atribuição de valores de B a essas variáveis, obtém-se o valor da expressão substituindo-se cada ocorrência de cada variável na expressão pelo correspondente valor e efetuando-se as operações indicadas na expressão. Isto é, o valor da expressão para um elemento $a = (a_1, a_2, \ldots, a_n) \in B^n$ é calculado substituindo-se cada ocorrência de x_i na expressão por a_i, $i = 1, 2, \ldots, n$, e calculando-se as operações. Ao se calcular o valor da expressão para todas as possíveis e distintas atribuições de valores a essas n variáveis, temos a "tabela" de uma função de B^n em B.

Não deve ser difícil notar que o valor de expressões equivalentes, para cada atribuição de valores às variáveis, é o mesmo. Isto significa que distintas expressões, equivalentes entre si, definem uma mesma função. Um exemplo é mostrado a seguir:

Exemplo 5.1.1. *A função $f : B^2 \to B$, definida pela expressão $f(x_1, x_2) = x_1 + x_2$ pode ser representada pela tabela-verdade a seguir, do lado esquerdo. Note que ela é igual a tabela-verdade da expressão $x_1 + \overline{x}_1 x_2$, a sua direita. Logo, as expressões $x_1 + x_2$ e $x_1 + \overline{x}_1 x_2$ são equivalentes (ou seja, definem uma mesma função).*

x_1	x_2	$x_1 + x_2$
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*1*

x_1	x_2	$\overline{x}_1$	$\overline{x}_1\, x_2$	$x_1 + \overline{x}_1\, x_2$
0	*0*	*1*	*0*	*0*
0	*1*	*1*	*1*	*1*
1	*0*	*0*	*0*	*1*
1	*1*	*0*	*0*	*1*

Esta equivalência pode também ser verificada algebricamente. Temos $x_1 + \overline{x}_1\, x_2 = (x_1 + \overline{x}_1)(x_1 + x_2) = 1\,(x_1 + x_2) = x_1 + x_2$*. Aqui cabe observar que a verificação algébrica de expressões é geral, válida para qualquer álgebra booleana, enquanto a verificação por tabelas acima é válida apenas em relação à álgebra booleana* $B = \{0, 1\}$*.*

Definição 5.1.1 (Função booleana)**.** *Uma função* $f : B^n \to B$ *é denominada* **função booleana** *se ela pode ser definida por uma expressão booleana em* n *variáveis.*

Veremos que as funções do tipo $f : B^n \to B$ (as que são de interesse no contexto de circuitos digitais) são todas booleanas.

5.1.3 Produtos e somas

Produto: Um **produto** em n variáveis $x_1, x_2, \ldots, x_n$ é uma expressão booleana que é ou um literal, ou uma conjunção[1] de dois ou mais literais, desde que uma variável não apareça mais de uma vez (pode aparecer apenas na forma não barrada ou apenas na forma barrada). Por exemplo, para $n = 4$, alguns exemplos de expressões que são produto:

- $x_1 x_3$
- $x_2 \overline{x}_3 \overline{x}_4$
- x_2

Exemplos de expressões que não são produtos:

- $x_1\, x_2\, \overline{x}_1$
- $x_1\, x_1$
- $\overline{x_1\, x_2}$
- $(x_1 + x_2)\, x_3$

[1] Conjunção é outro nome para a operação $\cdot$ (E lógico).

Produto canônico ou mintermo: Um **produto canônico** em n variáveis $x_1, x_2, \ldots, x_n$ é uma expressão booleana do tipo produto contendo cada uma das n variáveis ou os respectivos complementos (mas não ambas). Ou seja, consiste do produto de n literais, cada um correspondendo a uma variável (se x_i está presente no produto, então $\overline{x}_i$ não está, e vice-versa). Produtos canônicos são também denominados **mintermos**.

Denotemos x por x^1 e $\overline{x}$ por x^0. Assim, qualquer mintermo pode ser expresso por $m_{e_1,e_2,\ldots,e_n} = x_1^{e_1}\, x_2^{e_2}\, \ldots\, x_n^{e_n}$, com $e_i \in \{0,1\}$. Por exemplo, se consideramos $n = 3$, então $m_{001} = x_1^0\, x_2^0\, x_3^1 = \overline{x}_1\, \overline{x}_2\, x_3$. Uma vez que o conjunto de todas as sequências de n *bits* corresponde à representação binária dos números entre 0 e $2^n - 1$, um mintermo pode ser identificado por um índice decimal. A tabela 5.1 apresenta todos os mintermos em três variáveis e a notação com um índice decimal associado a cada um deles.

$e_1\, e_2\, e_3$	mintermo	notação
0 0 0	$\overline{x}_1\overline{x}_2\overline{x}_3$	m_0
0 0 1	$\overline{x}_1\overline{x}_2 x_3$	m_1
0 1 0	$\overline{x}_1 x_2\overline{x}_3$	m_2
0 1 1	$\overline{x}_1 x_2 x_3$	m_3
1 0 0	$x_1\overline{x}_2\overline{x}_3$	m_4
1 0 1	$x_1\overline{x}_2 x_3$	m_5
1 1 0	$x_1 x_2\overline{x}_3$	m_6
1 1 1	$x_1 x_2 x_3$	m_7

Tabela 5.1: Tabela de mintermos em 3 variáveis.

Teorema 5.1.1. *Há 2^n mintermos em n variáveis e não há dois mintermos equivalentes.*

Demonstração. Como um mintermo consiste de n literais, cada um podendo ser uma variável x ou o seu complemento $\overline{x}$, há no total 2^n possíveis formas de se combinar os literais.

Para mostrar que não há dois mintermos equivalentes, seja $m_{e_1,e_2,\ldots,e_n} = x_1^{e_1}\, x_2^{e_2}\, \ldots\, x_n^{e_n}$ um mintermo e considere

$$x_i = \begin{cases} 1, & \text{se } e_i = 1, \\ 0, & \text{se } e_i = 0. \end{cases}$$

Então, $m_{e_1,e_2,\ldots,e_n}(x_1, x_2, \ldots, x_n) = 1$, pois pela forma como $x_i^{e_i}, i = 1, \ldots, n$, foram definidas, todos os literais no mintermo m tem valor 1. Isto mostra que há uma atribuição de valores às variáveis x_1, x_2, ..., x_n que torna 1 o valor de m.

Qualquer outro mintermo m' tem pelo menos um literal x^{e_j} que é complemento do correspondente literal em m. Portanto, substituindo os valores acima das n variáveis em m', haverá pelo

menos um elemento zero no produto (devido ao literal x^{e_j}). Isto quer dizer que m' vale zero, em particular, para esses valores de x_1, x_2, ..., x_n. Logo, para quaisquer dois mintermos, há sempre uma atribuição de valores às variáveis x_1, x_2, ..., x_n que torna um deles 1 e o outro 0 e portanto não há mintermos equivalentes. ■

Soma: Uma **soma** é definida de forma análoga a produtos. Uma **soma** é ou um literal ou a disjunção de dois ou mais literais, desde que uma variável não apareça mais de uma vez (pode aparecer apenas na forma não barrada ou apenas na forma barrada).

Soma canônica ou maxtermo: Uma **soma canônica** em n variáveis $x_1, x_2, \ldots, x_n$, também denominada **maxtermo**, é definida de forma análoga ao mintermo; em vez de produto, consiste de soma de n literais, cada um correspondendo a uma variável. As expressões $\overline{x}_1 + \overline{x}_2 + \overline{x}_3$ e $\overline{x}_1 + x_2 + x_3$ são exemplos de maxtermos em três variáveis. A tabela 5.2 lista todos os maxtermos de 3 variáveis. Por exemplo, para $e_1e_2e_3 = 001$, temos o maxtermo $\overline{x_1^{e_1}x_2^{e_2}x_3^{e_3}} = \overline{\overline{x}_1\,\overline{x}_2\,x_3} = x_1 + x_2 + \overline{x}_3$.

$e_1e_2e_3$	maxtermo	notação
0 0 0	$x_1 + x_2 + x_3$	M_0
0 0 1	$x_1 + x_2 + \overline{x}_3$	M_1
0 1 0	$x_1 + \overline{x}_2 + x_3$	M_2
0 1 1	$x_1 + \overline{x}_2 + \overline{x}_3$	M_3
1 0 0	$\overline{x}_1 + x_2 + x_3$	M_4
1 0 1	$\overline{x}_1 + x_2 + \overline{x}_3$	M_5
1 1 0	$\overline{x}_1 + \overline{x}_2 + x_3$	M_6
1 1 1	$\overline{x}_1 + \overline{x}_2 + \overline{x}_3$	M_7

Tabela 5.2: Tabela de maxtermos com 3 variáveis.

Pode-se mostrar, de forma similar ao caso dos mintermos, que há exatamente 2^n maxtermos, e que eles são dois a dois distintos.

5.2 A álgebra booleana das funções booleanas

Denotamos por $B(n)$ o conjunto de todas as funções booleanas sobre B em n variáveis, isto é, $B(n) = \{f : B^n \to B \mid f \text{ é uma função booleana}\}$. Para quaisquer $f, g \in B(n)$ e para todo $\mathbf{a} \in B^n$, definimos

- $(f+g)(\mathbf{a}) = f(\mathbf{a}) + g(\mathbf{a})$

- $(f \cdot g)(\mathbf{a}) = f(\mathbf{a}) \cdot g(\mathbf{a})$
- $\overline{f(\mathbf{a})} = \overline{f(\mathbf{a})}$

A função $\mathbf{0}(\mathbf{a}) = 0, \forall \mathbf{a} \in B^n$, é a função nula e a função $\mathbf{1}(\mathbf{a}) = 1, \forall \mathbf{a} \in B^n$, é a função constante $\mathbf{1}$. Pode-se mostrar que $\langle B(n), +, \cdot, \bar{}, \mathbf{0}, \mathbf{1} \rangle$ é uma álgebra booleana. A demonstração é deixada como exercício.

Além disso, podemos também definir uma relação de ordem parcial $\preceq$ sobre $B(n)$ da seguinte forma:

$$f \preceq g \iff f(\mathbf{a}) \leq g(\mathbf{a}),\ \forall \mathbf{a} \in B^n \tag{5.1}$$

na qual $\leq$ é a relação de ordem parcial definida sobre a álgebra booleana B. A demonstração de que $\preceq$ é uma relação de ordem parcial também é deixada como exercício.

Há $2^{(2^2)} = 16$ funções de B^2 para B (ver abaixo).

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

As expressões correspondentes a essas 16 funções são mostradas na figura 5.1, juntamente com o diagrama de Hasse.

Quais são os átomos de $B(2)$? Como pode ser visto na figura 5.1, são as funções definidas pelas expressões $\overline{x}_1\,\overline{x}_2$, $\overline{x}_1\,x_2$, $x_1\,\overline{x}_2$ e $x_1\,x_2$. Portanto, pelo teorema 4.6.2 (decomposição de um elemento como soma de átomos) visto no capítulo anterior, decorre que qualquer função em $B(2)$ pode ser escrito na forma soma de um subconjunto desses átomos, de forma única. Na próxima seção mostramos o mesmo resultado de outra forma.

5.3 Formas canônicas

Dentre as expressões booleanas, existem as que são da forma soma de produtos ou então produto de somas, e elas recebem atenção especial pois incluem as chamadas formas canônicas.

$f_0(x_1, x_2) = 0$
$f_1(x_1, x_2) = x_1\, x_2$
$f_2(x_1, x_2) = x_1\, \overline{x}_2$
$f_3(x_1, x_2) = x_1$
$f_4(x_1, x_2) = \overline{x}_1\, x_2$
$f_5(x_1, x_2) = x_2$
$f_6(x_1, x_2) = \overline{x}_1\, x_2 + x_1\, \overline{x}_2$
$f_7(x_1, x_2) = x_1 + x_2$
$f_8(x_1, x_2) = \overline{x}_1\, \overline{x}_2$
$f_9(x_1, x_2) = \overline{x}_1\, \overline{x}_2 + x_1\, x_2$
$f_{10}(x_1, x_2) = \overline{x}_2$
$f_{11}(x_1, x_2) = x_1 + \overline{x}_2$
$f_{12}(x_1, x_2) = \overline{x}_1$
$f_{13}(x_1, x_2) = \overline{x}_1 + x_2$
$f_{14}(x_1, x_2) = \overline{x}_1 + \overline{x}_2$
$f_{15}(x_1, x_2) = 1$

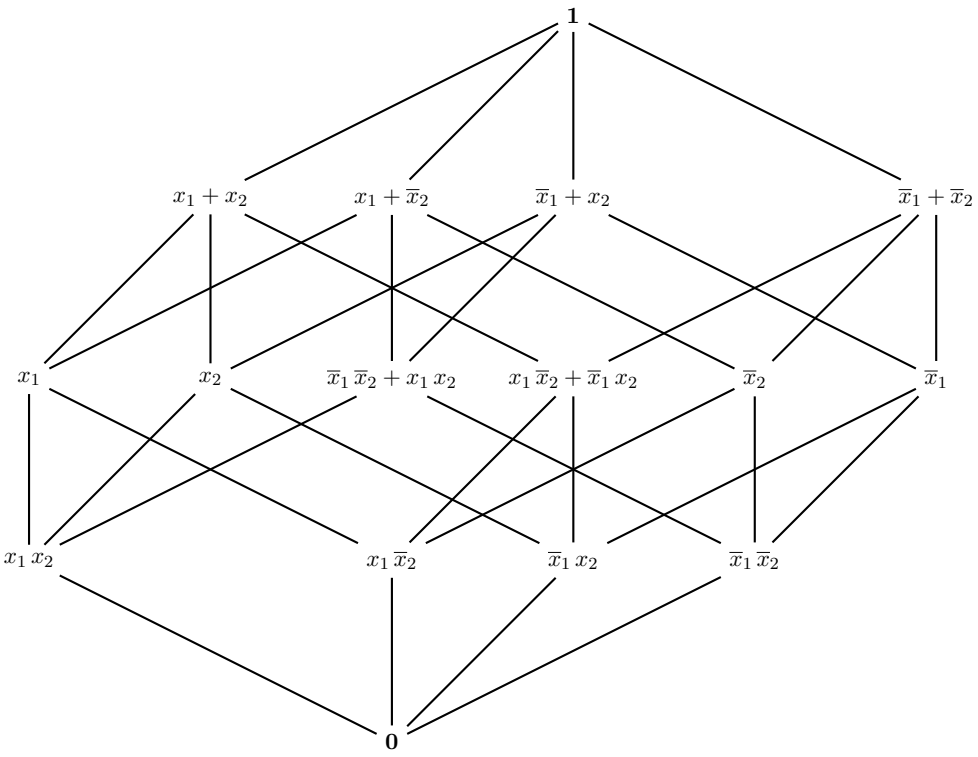


Figura 5.1: Diagrama de Hasse da álgebra booleana $\langle B(2), +, \cdot, \bar{}, \mathbf{0}, \mathbf{1}\rangle$ (i.e., todas as funções booleanas $f : B^2 \to B$).

5.3.1 Soma de produtos e produto de somas

Dizemos que uma expressão está na forma **soma de produtos** (SOP) se ela é um produto ou se é uma soma (disjunção) de dois ou mais produtos e se nenhum par de produtos p e p' é tal que $p \preceq p'$ (a relação $\preceq$ é a definida pela equivalência 5.1).

Analogamente, dizemos que uma expressão booleana está na forma **produto de somas** (POS) se ela é uma soma ou é uma conjunção de duas ou mais somas.

As expressões $x\, y$, $x + yz$ e $xyw + \overline{x}z + yz$ estão na forma SOP. A expressão $x(y + z)$ não está na forma SOP (por causa dos parênteses). A expressão $x\, y + y\, z\, x$ não está na forma SOP (pois $yzx \preceq xy$). De fato,

$$x\, y + x\, y\, z = x(y + y\, z) = x(y \cdot 1 + y\, z) = x(y(1 + z)) = x(y \cdot 1) = x(y) = x\, y$$

Exemplo 5.3.1. *Para escrever a expressão $f(x, y, z, w) = (xz + y)(zw + \overline{w})$ na forma SOP, aplicamos a propriedade distributiva para eliminar os parênteses, como nos dois exemplos abaixo.*

Se houver um termo absorvido por outro, esse deverá ser removido.

$$\begin{aligned} f(x,y,z,w) &= (xz+y)(zw+\overline{w}) \\ &= (xz+y)zw + (xz+y)\overline{w} \quad (distributiva) \\ &= xzw + yzw + xz\overline{w} + y\overline{w} \quad (distributiva) \end{aligned}$$

$$\begin{aligned} g(x,y,z) &= [(x+\overline{y})+z](x+\overline{\overline{y}})\overline{x} \\ &= [(x+\overline{y})+z](x+y)\overline{x} \\ &= [(x+\overline{y})+z](x\overline{x}+y\overline{x}) \\ &= [(x+\overline{y})+z]y\overline{x} \\ &= xy\overline{x}+\overline{y}y\overline{x}+zy\overline{x} \\ &= 0+0+zy\overline{x} \\ &= \overline{x}yz \end{aligned}$$

5.3.2 Soma canônica de produtos (SOP canônica)

Soma canônica de produtos consiste de uma soma de produtos canônicos. O termo "canônica" em soma canônica refere-se ao fato de, para cada função booleana, haver apenas uma forma de escrever uma expressão como soma de produtos canônicos. Esta soma é equivalente à representação da função como união (soma) de átomos.

Os seguintes teoremas mostram a expressão de qualquer função booleana na forma soma canônica de produtos.

Teorema 5.3.1 (Teorema de expansão de Boole)**.** *Seja $f : B^n \to B$ uma função booleana em n variáveis $x_1, x_2, \ldots, x_n$. Então,*

$$f(x_1, x_2, \ldots, x_n) = \overline{x}_1 \cdot f(0, x_2, \ldots, x_n) + x_1 \cdot f(1, x_2, \ldots, x_n)\,. \tag{5.2}$$

Demonstração. Veja, por exemplo, página 408 de R. Garnier and J. Taylor, *Discrete Mathematics for New Technology*, Adam Hilger, 1992. ■

Teorema 5.3.2 (Teorema de expansão de Boole dual)**.** *Seja $f : B^n \to B$ uma função booleana em n variáveis $x_1, x_2, \ldots, x_n$. Então,*

$$f(x_1, x_2, \ldots, x_n) = [\overline{x}_1 + f(1, x_2, \ldots, x_n)] \cdot [x_1 + f(0, x_2, \ldots, x_n)]\,. \tag{5.3}$$

Teorema 5.3.3. *Qualquer função booleana não nula pode ser expressa unicamente na forma* **soma canônica de produtos**. *Mais precisamente, se f é uma função booleana em n variáveis então a sua forma soma canônica de produtos é dada por*

$$f(x_1, x_2, \ldots, x_n) = \bigvee_{(e_1,e_2,\ldots,e_n)\in\{0,1\}^n} f(e_1, e_2, \ldots, e_n)\, x_1^{e_1}\, x_2^{e_2}\, \ldots\, x_n^{e_n}$$

Demonstração. Esta equivalência pode ser mostrada aplicando-se o teorema de expansão de Boole sucessivamente, e a unicidade de representação está relacionada ao fato de mintermos serem distintos dois a dois. Uma demonstração completa pode ser encontrada em R. Garnier and J. Taylor, *Discrete Mathematics for New Technology*, Adam Hilger, 1992, página 408. ■

Os dois exemplos a seguir mostram, respectivamente, a expansão sucessiva de uma função, e a aplicação do teorema para o cálculo da forma soma canônica de produtos.

Exemplo 5.3.2. *Considere uma função booleana $f : B^3 \to B$ e denotemos as três variáveis por x_1, x_2 e x_3. Ao aplicarmos o teorema de expansão de Boole recursivamente, obtemos a seguinte expansão:*

$$\begin{aligned} f(x_1, x_2, x_3) &= \overline{x}_1\, f(0, x_2, x_3) + x\, f(1, x_2, x_3) \\ &= \overline{x}_1\, [\overline{x}_2\, f(0, 0, x_3) + x_2\, f(0, 1, x_3)] + x_1\, [\overline{x}_2\, f(1, 0, x_3) + x_2\, f(1, 1, x_3)] \\ &= \overline{x}_1\overline{x}_2\overline{x}_3\, f(0,0,0) + \overline{x}_1\overline{x}_2 x_3\, f(0,0,1) + \overline{x}_1 x_2\overline{x}_3\, f(0,1,0) + \overline{x}_1 x_2 x_3\, f(0,1,1) + \\ &\quad x_1\overline{x}_2\overline{x}_3\, f(1,0,0) + x_1\overline{x}_2 x_3\, f(1,0,1) + x_1 x_2\overline{x}_3\, f(1,1,0) + x_1 x_2 x_3\, f(1,1,1) \end{aligned}$$

Após substituírmos os valores de f nos pontos $\{0,1\}^3$ na expressão acima, restarão apenas os produtos canônicos (mintermos) m_i tais que $m_i \preceq f$.

Exemplo 5.3.3. *Seguindo o exemplo anterior, para escrever a função $f(x_1, x_2) = x_1 + x_2$ na forma SOP canônica, podemos aplicar o teorema de expansão de Boole:*

$$f(x_1, x_2) = f(0,0)\overline{x}_1\,\overline{x}_2 + f(0,1)\overline{x}_1\, x_2 + f(1,0) x_1\,\overline{x}_2 + f(1,1) x_1\, x_2$$

Agora, podemos calcular o valor de f para todos os elementos $\mathbf{e} \in \{0,1\}^2$, que são: $f(0,0) = 0$ e $f(0,1) = f(1,0) = f(1,1) = 1$. Substituindo esses valores na expressão expandida acima, temos

que a forma soma canônica de produtos de f é:

$$f(x_1, x_2) = 0 \cdot \overline{x}_1\,\overline{x}_2 + 1 \cdot \overline{x}_1\,x_2 + 1 \cdot x_1\,\overline{x}_2 + 1 \cdot x_1\,x_2 = \overline{x}_1\,x_2 + x_1\,\overline{x}_2 + x_1\,x_2 \tag{5.4}$$

Teorema 5.3.4. *Uma função $f : B^n \to B$ é booleana se e somente se ela pode ser expressa na forma soma canônica de produtos em n variáveis $x_1, x_2, \ldots, x_n$,*

$$f(x_1, x_2, \ldots, x_n) = \bigvee_{(e_1,e_2,\ldots,e_n)\in\{0,1\}^n} f(e_1, e_2, \ldots, e_n)\, x_1^{e_1}\, x_2^{e_2}\, \ldots\, x_n^{e_n} \tag{5.5}$$

Demonstração. Se f é uma função booleana, então ela pode ser expressa na forma soma canônica de produtos conforme teorema anterior. Por outro lado, suponha que f pode ser expressa na forma da equação 5.5. Claramente, essa expressão é uma expressão booleana e portanto f é uma função booleana. ■

Conforme vimos acima, pode-se escrever a expressão SOP canônica de uma função booleana aplicando-se a expansão de Boole recursivamente e em seguida substituindo-se os valores de $f(e_1, e_2, \ldots, e_n)$ para cada $(e_1, e_2, \ldots, e_n) \in B^n$.

Esse resultado apenas confirma o que já vínhamos fazendo a partir da tabela-verdade de uma função. Por exemplo, na forma de uma tabela-verdade a seguir, já vimos no capítulo 3 como gerar uma expressão booleana que "concorda" com a atvela.

Entrada			Saída
x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Especificamente, tomamos a entradas para as quais a função toma valor 1. No exemplo da tabela elas são 000, 001, 010, 011 e 111. Para cada entrada, conforme já vimos, há apenas um produto canônico que toma valor 1. Por exemplo, para a entrada 010 o produto canônico que toma valor 1 é o produto $\overline{x_1}\,x_2\,\overline{x_3}$ (pois, $\overline{x_1}\,x_2\,\overline{x_3} = \overline{0}\,1\,\overline{0} = 1 \cdot 1 \cdot 1 = 1$). Para qualquer outra entrada, esse produto toma valor 0 e é por isso que podemos associar um único produto canônico

para cada entrada. Portanto, uma função pode ser escrita como a soma dos produtos associados às entradas para as quais ela toma valor 1. Em particular, a função definida pela tabela acima pode ser escrita como:

$$f(x_1, x_2, x_3) = \overline{x}_1\,\overline{x}_2\,\overline{x}_3 + \overline{x}_1\,\overline{x}_2\,x_3 + \overline{x}_1\,x_2\,\overline{x}_3 + \overline{x}_1\,x_2\,x_3 + x_1\,x_2\,x_3 \tag{5.6}$$

Usando-se o índice decimal associado a cada mintermo, podemos escrever a expressão da função f de forma mais compacta como:

$$f(x_1, x_2, x_3) = m_0 + m_1 + m_2 + m_3 + m_7 = \sum m(0, 1, 2, 3, 7)$$

Pode-se também obter a forma canônica de uma expressão por meio de manipulações algébricas (eliminar os parênteses e complementos de subexpressões e em seguida "introduzir", em cada produto, as variáveis que estão ausentes). Por exemplo, no caso da expressão $f(x, y, z, w) = (xz + y)(zw + \overline{w})$ podemos fazer:

$$\begin{aligned}
f(x,y,z,w) &= (xz+y)zw + (xz+y)\overline{w} \\
&= xzw + yzw + xz\overline{w} + y\overline{w} \\
&= xzw(y+\overline{y}) + (x+\overline{x})yzw + x(y+\overline{y})z\overline{w} + (x+\overline{x})y(z+\overline{z})\overline{w} \\
&= xyzw + x\overline{y}zw + xyzw + \overline{x}yzw + xyz\overline{w} + x\overline{y}z\overline{w} + xy(z+\overline{z})\overline{w} + \overline{x}y(z+\overline{z})\overline{w} \\
&= xyzw + x\overline{y}zw + \overline{x}yzw + xyz\overline{w} + x\overline{y}z\overline{w} + xyz\overline{w} + xy\overline{z}\overline{w} + \overline{x}yz\overline{w} + \overline{x}y\overline{z}\,\overline{w} \\
&= xyzw + xyz\overline{w} + xy\overline{z}\,\overline{w} + x\overline{y}zw + x\overline{y}z\overline{w} + \overline{x}yzw + \overline{x}yz\overline{w} + \overline{x}y\overline{z}\,\overline{w}
\end{aligned}$$

Aplicando-se essa técnica à expressão $f(x_1, x_2) = x_1 + x_2$, temos

$$\begin{aligned}
f(x_1, x_2) &= x_1 + x_2 = x_1(x_2 + \overline{x}_2) + (x_1 + \overline{x}_1)x_2 \\
&= x_1\,x_2 + x_1\,\overline{x}_2 + x_1\,x_2 + \overline{x}_1\,x_2 \\
&= \overline{x}_1\,x_2 + x_1\,\overline{x}_2 + x_1\,x_2
\end{aligned}$$

5.3.3 Produto canônico de somas (POS canônica)

Todos os conceitos e resultados acima com respeito a expressões do tipo soma de produtos podem também ser definidos com respeito a expressões do tipo produto de somas.

Teorema 5.3.5. *Qualquer função booleana que não seja identicamente* 1 *pode ser expressa unicamente na forma* **produto canônico de somas**.

Demonstração. A demonstração desse teorema é dual à demonstração do teorema anterior para a soma canônica de produtos. Em particular, deve-se considerar o teorema de expansão de Boole

dual, visto anteriormente. ■

Exemplo 5.3.4. *A expressão* $x + z + \overline{y}\,\overline{w}$ *na forma POS canônica pode ser calculada como segue:*

$$\begin{aligned} f(x,y,z,w) &= x + z + \overline{y}\,\overline{w} \\ &= x + (z + \overline{y}\,\overline{w}) \\ &= x + (z + \overline{y})(z + \overline{w}) \\ &= (x + z + \overline{y})(x + z + \overline{w}) \\ &= (x + \overline{y} + z + w\overline{w})(x + y\overline{y} + z + \overline{w}) \\ &= (x + \overline{y} + z + w)(x + \overline{y} + z + \overline{w})(x + y + z + \overline{w})(x + \overline{y} + z + \overline{w}) \\ &= (x + \overline{y} + z + w)(x + \overline{y} + z + \overline{w})(x + y + z + \overline{w}) \end{aligned}$$

Exemplo 5.3.5. *Seja a função que toma valor zero para as entradas* 100*,* 101 *e* 110 *e 1 para as demais (esta função foi apresentada acima). A soma* $\overline{x}_1 + x_2 + x_3$ *toma valor 0 para a entrada* 100 *e valor 1 para as demais entradas. Como um produto de somas toma valor 0 se ao menos um dos termos soma tomar valor 0, podemos escrever a função como o produto de somas canônicas associadas às entradas para as quais a função toma valor 0. No caso do exemplo em questão,*

$$f(x_1, x_2, x_3) = (\overline{x}_1 + x_2 + x_3)\,(\overline{x}_1 + x_2 + \overline{x}_3)\,(\overline{x}_1 + \overline{x}_2 + x_3) \tag{5.7}$$

Essa expressão está na forma POS canônica, que em sua forma compacta, é escrita como:

$$f(x_1, x_2, x_3) = M_4 \cdot M_5 \cdot M_6 = \prod M(4, 5, 6)$$

5.4 Observações

Nos textos da literatura relacionados a funções booleanas as funções do tipo $f : B^n \to B$ são comumente denominadas **funções lógicas** ou **funções de chaveamento**.

Cabe notar também que a terminologia utilizada no contexto de circuitos lógicos (tais como produtos, somas e as formas canônicas) varia bastante de autor para autor.

- Em vez de **produto**, alguns autores utilizam também os nomes **termo produto**, **produto fundamental**, **conjunção fundamental** ou **produto normal**.
- Em vez de **soma de produtos**, utiliza-se também os nomes **soma de produtos normais** ou **forma normal disjuntiva**.

- Em vez de **soma canônica de produtos** (SOP canônica), utiliza-se também os nomes **soma padrão de produtos**, **forma normal disjuntiva completa** ou **forma mintermo**. Note, porém, que alguns autores usam o nome **forma normal disjuntiva** em vez de **forma normal disjuntiva completa**.
- Neste texto usaremos **soma de produtos** e **soma canônica de produtos** (ou **soma de mintermos**).

Essas mesmas observações aplicam-se também para o soma e produto de somas.

Exercícios

1. O que expressões booleanas equivalentes? Como mostrar se duas expressões são equivalentes ou não ?
2. Seja $B = \{0, 1\}$ e a álgebra booleana $\langle B, +, \cdot, \bar{}, 0, 1\rangle$ e seja n um inteiro finito. Quantas funções da forma $f : B^n \to B$ existem?
3. O que são funções booleanas?
4. Seja $f(x_1, x_2) = x_1\, x_2$ e seja $g(x_1, x_2) = \overline{x}_1$. Calcule a expressão de $f + g$.
5. Considere três variáveis booleanas a, b e c. Assinale quais das expressões abaixo são produtos.
   a) ab
   b) $a\overline{a}$
   c) a
   d) $a\, a\, b$
   e) $a(b + c)$
   f) $\overline{a}\,\overline{b}\,\overline{c}$
6. Considere três variáveis booleanas a, b e c. Assinale quais das expressões abaixo são somas.
   a) $a + b$
   b) $a + \overline{a}$
   c) a
   d) $a + a + b$
   e) $a + bc$
   f) $\overline{a} + \overline{b} + \overline{c}$
7. Assinale quais das expressões abaixo são somas de produtos.
   a) a
   b) $ab + ac$
   c) $a + ab$
   d) $ab + \overline{a\, b}$
   e) $ab + ab$
   f) abc
8. Escreva todos os produtos canônicos em três variáveis a, b e c.
9. Escreva a expressão das seguintes expressões na forma algébrica.
   a) $f(a, b, c) = m_1 + m_3 + m4 + m_7$
   b) $f(a, b, c) = M_0\, M_6$
10. Mostre que a relação $\preceq$ definida conforme a equação 5.1 é uma relação de ordem parcial.

11. Mostre que $a\,b\,c \preceq b\,c$ e que $x + yz\overline{w} \preceq x + yz$.

12. Escreva as seguintes funções na forma SOP (soma de produtos). Não precisam ficar na forma canônica.

    a) $f(a,b,c) = \overline{a\,b\,c}$

    b) $f(a,b,c) = a + b + c$

    c) $f(a,b,c,d) = (a+b)(b+c+d)$

    d) $f(a,b,c) = \overline{(\overline{a+b}) + c}$

    e) $f(a,b,c,d,e) = (\overline{a}\,\overline{c} + \overline{d})(\overline{b + c\,e})$

13. Escreva as seguintes funções na forma POS (produto de somas). Não precisam ficar na forma canônica.

    a) $f(a,b,c) = \overline{a + b + c}$

    b) $f(a,b,c) = a\,b\,c$

14. Escreva $f(a,b,c,d) = (a+b)c\,\overline{d} + (a+b)\overline{c}\,d$ na forma SOP canônica.

15. Escreva $f(x,y,z,w) = x + z + \overline{y}\,\overline{w}$ na forma SOP canônica. Existe relação entre a forma SOP canônica e a forma POS canônica? Qual?

16. Ache a expressão na forma SOP canônica que define a função dada pela tabela-verdade abaixo. Você consegue simplificar esta expressão e obter uma outra equivalente e mais curta ?

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

17. Prove que qualquer função booleana que não seja identicamente 0 (nulo) pode ser expressa na forma SOP canônica.

18. Prove que a forma SOP canônica de qualquer função booleana que não seja identicamente 0 (nulo) é única, a menos da ordem dos produtos canônicos.

19. (Extra – Ver apêndice A.1) Seja $\langle A, +, \cdot, \bar{}, 0, 1\rangle$ uma álgebra booleana com $A = \{0, a, \overline{a}, 1\}$. Construa a tabela-verdade da função correspondente à expressão $f(x, y) = \overline{a}x + a\overline{y}$. Quantas linhas há nessa tabela ?

20. (Extra – Ver apêndice A.1) Seja $A = \{0, a, \overline{a}, 1\}$. Liste todas as funções booleanas da forma $A \to A$. Dica: use o teorema de expansão de Boole.

Capítulo 6

Minimização de funções booleanas

Última atualização em 12/04/2018

Nos capítulos anteriores vimos que uma mesma função pode ser expressa por mais de uma expressão. Dependendo do contexto no qual essas expressões são consideradas, pode-se desejar encontrar, dentre todas as expressões que representam uma mesma função booleana, aquela que satisfaz algum critério. Por exemplo, pode ser do interesse obter uma expressão "mais curta", ou então, uma expressão que não envolve um termo produto com mais de um determinado número de literais.

Uma das simplificações bastante estudadas no contexto de circuitos lógicos é a **minimização lógica dois-níveis**, na qual o objetivo é encontrar uma expressão na forma SOP (ou POS), envolvendo o menor número possível de termos produto (ou soma) e, em cada produto (ou soma), o menor número possível de literais. Isto reflete diretamente no circuito lógico: cada termo produto corresponde a uma porta E e o número de literais em um produto corresponde ao número de entradas na correspondente porta E.

Neste capítulo abordaremos o problema de minimização lógica dois-níveis.

6.1 Produtos, Intervalos, Cubos

A relação de ordem $\preceq$ entre funções booleanas foi estabelecida pela equação 5.1. Da definição, decorre que dados $f, g : B^n \to B$, $f \preceq g$ se e somente se $f(\mathbf{x}) = 1 \implies g(\mathbf{x}) = 1, \forall \mathbf{x} \in B^n$. Por exemplo, se considerarmos $n = 3$ e as variáveis a, b, c, a função definida pela expressão $a\,b\,c$ toma valor 1 apenas quando $a = b = c = 1$, enquanto a função definida por $a\,b$ toma valor 1 quando $a = b = c = 1$ e quando $a = b = 1$ e $c = 0$. Assim, claramente $a\,b\,c \preceq a\,b$. De forma similar,

temos que $a\,b \preceq a$. Se definirmos $f(a,b,c) = a\,b$, temos $a\,b \preceq f$, assim como $a\,b\,c \preceq f$ e $a\,b\,\overline{c} \preceq f$.

As expressões produto tem uma característica interessante: se o produto contém k dos n literais, então o número de entradas para as quais ele toma valor 1 é exatamente igual a 2^{n-k}. Assim, os produtos canônicos que possuem $k = n$ literais tomam valor 1 em apenas uma única entrada. No exemplo acima, o produto $a\,b\,c$ toma valor 1 em apenas uma única entrada, o produto $a\,b$ toma valor 1 em $2^{3-2} = 2$ entradas e o produto a toma valor 1 em $2^{3-1} = 4$ entradas. Essa característica pode ser vista na figura 6.1. No diagrama mais à esquerda, está destacado o único elemento em B^3 para o qual o produto $a\,b\,c$ tomar valor 1. No diagrama do centro estão destacados os dois elementos para os quais o produto $a\,b$ toma valor 1. Finalmente, no diagrama mais à direita estão destacados os quatros pontos para os quais o produto a toma valor 1. Pode-se notar que $a\,b\,c \preceq a\,b \preceq a$. De forma geral, dadas duas funções f e g, se $f \preceq g$, dizemos que g **cobre** f. Logo a cobre $a\,b$ que cobre $a\,b\,c$. Podemos também dizer que o produto $a\,b\,c$ cobre 111, que $a\,b$ cobre 110 e 111, e que a cobre 100, 101, 110 e 111, e assim por diante.

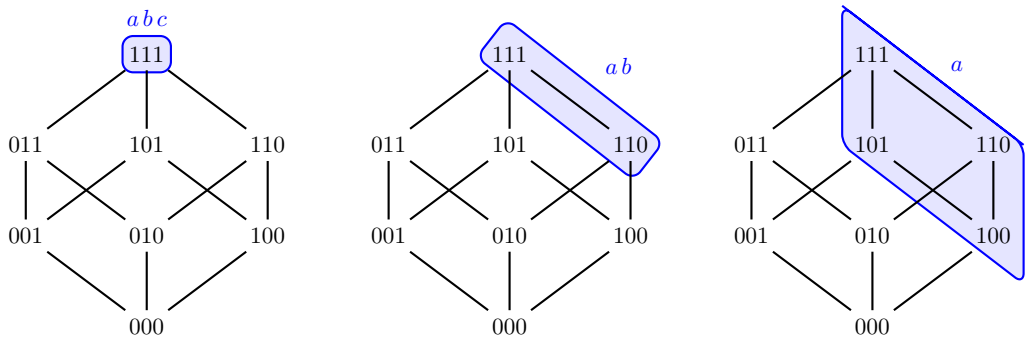


Figura 6.1: Diagrama de Hasse da álgebra booleana $\langle B^3, +, \cdot, \bar{}, \mathbf{0}, \mathbf{1}\rangle$.

Intervalo: Em $B = \{0,1\}$ temos $0 \leq 0$, $0 \leq 1$ e $1 \leq 1$. Essa relação pode ser estendida para B^n fazendo-se $a_1\,a_2 \cdots a_n \leq b_1\,b_2 \cdots b_n \iff a_i \leq b_i, \forall i = 1, 2, \cdots, n$, para quaisquer dois elementos $a_1\,a_2 \cdots a_n$ e $b_1\,b_2 \cdots b_n$ em B^n. Por exemplo, em B^3, $010 \leq 011$ mas $010 \not\leq 101$. Com isso podemos definir um **intervalo** $[a_1\,a_2 \cdots a_n\,,\, b_1\,b_2 \cdots b_n] = \{x_1\,x_2 \cdots x_n | a_1\,a_2 \cdots a_n \leq x_1\,x_2 \cdots x_n \leq b_1\,b_2 \cdots b_n\}$, com extremo inferior $a_1\,a_2 \cdots a_n$ e extremo superior $b_1\,b_2 \cdots b_n$. Por exemplo, $[010, 111] = \{010, 011, 110, 111\}$.

Voltando aos exemplos mostrados nos diagramas da figura 6.1, os vértices cobertos por um produto formam um **intervalo**: note que os pontos cobertos por $a\,b\,c$ formam o intervalo $[111, 111] = \{111\}$ (intervalo degenerado ou 0-cubo); os pontos cobertos por $a\,b$ formam o intervalo $[110, 111] = \{110, 111\}$ (1-cubo); e os pontos cobertos por a formam o intervalo $[100, 111] = \{100, 101, 110, 111\}$ (2-cubo). O conjunto B^3 é um 3-cubo.

Um intervalo pode ser representado compactamente trocando-se as coordenadas não fixas por `X`. Assim `X11` corresponde ao intervalo $[011, 111]$. No caso de três variáveis a, b e c, o produto

a que toma valor 1 para as entradas $\{100, 101, 110, 111\}$ e define o intervalo $[100, 111]$, em notação compacta pode ser denotado por `1XX`. A recíproca é verdade também: qualquer intervalo corresponde a um produto. Por exemplo, o intervalo $[000, 100] = \{000, 100\}$ (ou simplesmente `X00`) correponde ao produto $\overline{b}\,\overline{c}$. O intervalo `X0X` $= [000, 101] = \{000, 001, 100, 101\}$ corresponde ao produto $\overline{b}$. Um intervalo contém necessariamente 2^k elementos, onde $0 \leq k \leq n$.

Esses conceitos/terminologias estão resumidos no quadro a seguir:

Produto	elementos cobertos	intervalo	notação compacta (cubo/intervalo)	dimensão (k)	tamanho (2^k)
ab	$\{110, 111\}$	$[110, 111]$	`11X`	1	$2^1 = 2$
c	$\{001, 011, 101, 111\}$	$[001, 111]$	`XX1`	2	$2^2 = 4$

Observação: Daqui em diante utilizaremos equivalentemente os termos **produto**, **cubo** ou **intervalo** quando nos referirmos a um produto.

Exemplo 6.1.1. *Considere três variáveis a, b e c e a expressão booleana $f(a, b, c) = a\,\overline{c} + a\,b + b\,c$ cuja tabela-verdade é a mostrada a seguir:*

$a\,b\,c$	$f(a, b, c) = a\,\overline{c} + a\,b + b\,c$
0 0 0	*0*
0 0 1	*0*
0 1 0	*0*
0 1 1	*1*
1 0 0	*1*
1 0 1	*0*
1 1 0	*1*
1 1 1	*1*

- *$a\,\overline{c}$, $a\,b$, $b\,c$ são produtos de f*
- *$a\,\overline{c} \preceq f$, $a\,b \preceq f$, e $b\,c \preceq f$*
- *f cobre $a\,\overline{c}$, $a\,b$, e $b\,c$*
- *pontos cobertos por f: $\{011, 100, 110, 111\}$, também chamados de "os 1s de f"*
- *A expressão de f na forma SOP canônica pode ser construída somando-se os produtos canônicos (mintermos) para os quais a função f toma valor 1. Isto é, como f vale 1 para*

as entradas 011, 100, 110 *e* 111, *e os respectivos mintermos são* $\overline{a}\,b\,c$, $a\,\overline{b}\,\overline{c}$, $a\,b\,\overline{c}$, *e* $a\,b\,c$, *na forma SOP canônica* f *é expressa por:*

$$f(a,b,c) = \overline{a}\,b\,c + a\,\overline{b}\,\overline{c} + a\,b\,\overline{c} + a\,b\,c$$

- *cada um dos produtos canônicos (ou mintermos) define uma função que toma valor 1 em apenas um elemento de* B^3. *Dizemos que um mintermo* m_i *é mintermo ou produto canônico de* f *se* $m_i \preceq f$. *Por exemplo, temos* $\overline{a}\,b\,c \preceq f$ *pois* $\overline{a}\,b\,c = 1 \Longrightarrow f(a,b,c) = 1$.

- *Os produtos de* f *podem ser desenhados sobre o diagrama de Hasse como mostrado na figura 6.2. No primeiro diagrama mais à esquerda estão assinalados os quatro mintermos de* f*; no diagrama do centro estão assinalados os produtos de* f *que cobrem exatamente dois elementos (os produtos da expressão inicial); e no diagrama à direita estão os produtos essencias de* f *(aqueles que, se removidos fazem a expressão não mais ser equivalente à* f*). O produto* $a\,b$ *é redundante pois os mintermos que ele cobre já são cobertos pelos outros dois produtos.*

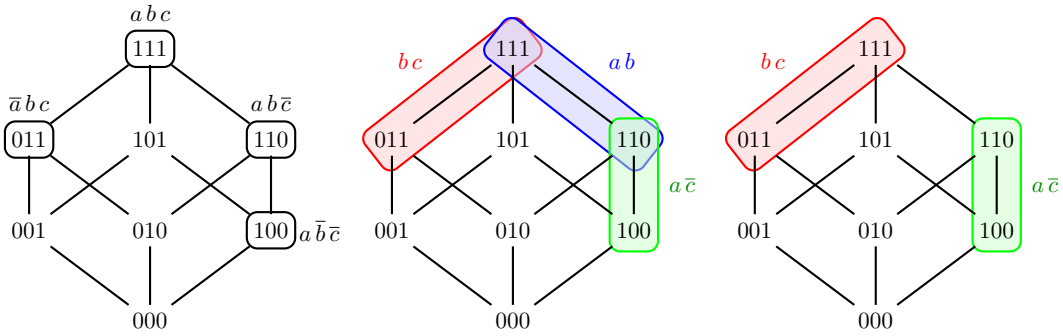


Figura 6.2: Mintermos da função f (esquerda), todos os seus produtos maximais (centro), e os produtos essenciais (direita).

- *No diagrama percebe-se facilmente que* $a\,\overline{c} + a\,b + b\,c = a\,\overline{c} + b\,c$. *Logo,* $f(a,b,c) = a\,\overline{c} + b\,c$, *uma expressão mais "curta" que a inicial (Este tipo de equivalência é conhecida como Teorema do consenso).*

6.2 Formas minimais

Definição: Uma expressão lógica escrita na forma soma de produtos é **minimal** se

1. não existe nenhuma outra expressão equivalente[1] na forma soma de produtos com um número menor de termos, e

[1] Duas expressões são equivalentes se definem uma mesma função.

2. não existe nenhuma outra expressão equivalente na forma soma de produtos com igual número de termos mas com menor número de literais.

Denominaremos tal expressão de **SOP minimal**[2]. Note que ao se remover, de uma forma SOP minimal, um produto ou um literal de qualquer um dos produtos, a expressão resultante não mais representa a mesma função.

O problema pode ser analogamente definido para a forma POS. O problema de encontrar a forma SOP ou POS minimal de uma função é denominada **minimização lógica dois-níveis** devido ao fato de funções na forma SOP ou POS poderem ser realizadas em circuitos de dois níveis (isto é, no caso da forma SOP, o primeiro nível é formado pelas portas E, uma para cada produto, e o segundo nível consiste de uma porta OU que recebe como entradas as saídas das portas E do nível anterior).

Algoritmos de minimização lógica dois-níveis assumem, em geral, que a expressão está na forma SOP. As técnicas mais clássicas como Mapas de Karnaugh e minimização de Quine-McCluskey assumem que a expressão inicial está na forma SOP canônica.

6.2.1 Simplificação algébrica

Usando regras algébricas, como por exemplo em $a\,b\,c + \overline{a}\,b\,c = (a+\overline{a})\,b\,c = b\,c$, podemos realizar a simplificação de uma expressão. A função $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$ pode ser simplificada como segue:

$$\begin{aligned}
f(x_1, x_2, x_3) &= \overline{x}_1\,\overline{x}_2\,\overline{x}_3 + \overline{x}_1\,\overline{x}_2\,x_3 + \overline{x}_1\,x_2\,\overline{x}_3 + \overline{x}_1\,x_2\,x_3 + x_1\,x_2\,x_3 \\
&= \overline{x}_1\,\overline{x}_2\,(\overline{x}_3 + x_3) + \overline{x}_1\,x_2\,(\overline{x}_3 + x_3) + x_1\,x_2\,x_3 \\
&= \overline{x}_1\,\overline{x}_2 + \overline{x}_1\,x_2 + x_1\,x_2\,x_3 \\
&= \overline{x}_1\,(\overline{x}_2 + x_2) + x_1\,x_2\,x_3 \\
&= \overline{x}_1 + x_1\,x_2\,x_3 \\
&= \overline{x}_1 + x_2\,x_3
\end{aligned}$$

A simplificação algébrica, porém, pode ser complicada pois requer o conhecimento das regras e, além disso, dependendo da ordem na qual as regras são aplicadas pode ser necessário um grande número de passos para se chegar à simplificação (sem contar que às vezes podemos, depois de vários passos, voltar à situação inicial). Se a expressão envolver muitas variáveis, fazer a simplificação na mão traz adicionalmente uma grande chance de se cometer erros bobos (trocar o nome de uma variável sem querer, esquecer o barra sobre uma variável de um passo para o

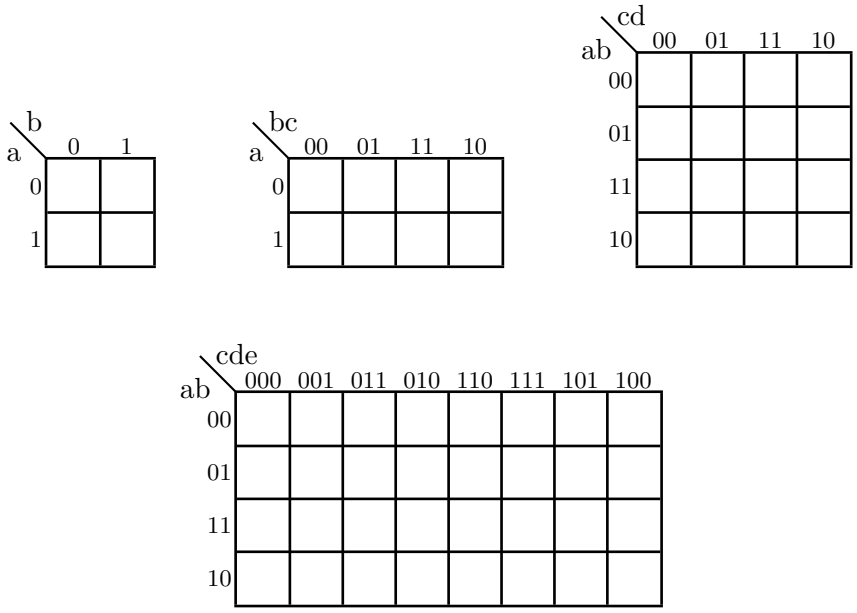
[2] SOP é contração de *Sum of products*

outro, etc). Além disso, pode não ser trivial verificar que a forma mais simplificada possível foi obtida (relembre o teorema do consenso, que afirma que, $x\,y + y\,z + \overline{x}\,z = x\,y + \overline{x}\,z$; não é trivial perceber que a expressão da esquerda é igual à expressão da direita).

6.3 Mapas de Karnaugh

Mapas de Karnaugh são diagramas que são utilizados para auxiliar o processo de minimização lógica dois-níveis, explorando uma representação gráfica adequada. No entanto, seu uso restringe-se à minimização de funções com não mais que 6 variáveis. Apesar disso, é um recurso pedagógico interessante e será explorado a seguir.

A estrutura de mapas de Karnaugh de 2 a 5 variáveis é exibida a seguir:



Cada célula de um mapa de n variáveis corresponde a um elemento de $\{0,1\}^n$ (portanto o mapa contém 2^n células). A concatenação do rótulo da linha com o rótulo da coluna de uma célula dá o elemento correspondente àquela célula, conforme mostrado no mapa à esquerda na figura 6.3. O mapa à direita mostra em cada célula o valor decimal das respectivas entradas.

Observe que o rótulo está disposto em uma sequência não-usual. Por exemplo, para duas variáveis, a sequência natural seria 00, 01, 10, 11. Porém, a sequência utilizada é 00, 01, 11, 10, que possui a característica de dois elementos adjacentes (na sequência) diferirem em apenas 1 bit (note que essa propriedade vale também em relação aos extremos esquerdo e direito do mapa e, similarmente, aos extremos superior e inferior). Esse tipo de sequência é conhecido como *gray code* e pode ser generalizado para n bits.

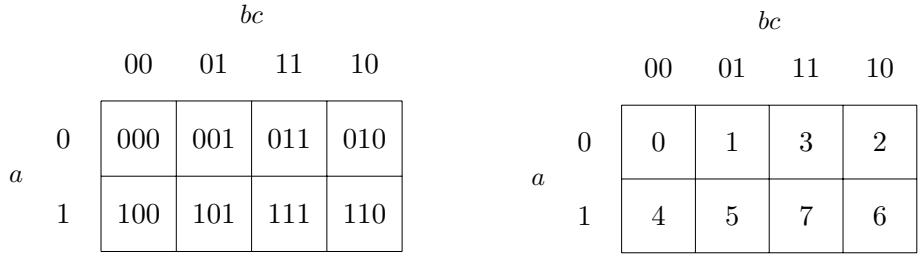


Figura 6.3: As entradas correspondentes a cada célula do mapa de Karnaugh de 3 variáveis em notação binária e decimal, respectivamente.

6.3.1 Minimização usando mapas de Karnaugh

Mostraremos a seguir como realizar a minimização de uma função de três variáveis, utilizando o mapa de Karnaugh. Utilizaremos a mesma função f acima que foi simplificada algebricamente, a qual reescrevemos aqui:

$$f(x_1, x_2, x_3) = \overline{x}_1\,\overline{x}_2\,\overline{x}_3 + \overline{x}_1\,\overline{x}_2\,x_3 + \overline{x}_1\,x_2\,\overline{x}_3 + \overline{x}_1\,x_2\,x_3 + x_1\,x_2\,x_3 = \overline{x}_1 + x_2\,x_3$$

Aparentemente a expressão simplificada acima é minimal. Para utilizarmos o mapa de Karnaugh, precisamos primeiramente transformar os mintermos da função para a notação cúbica. Assim,

Mintermo	notação cúbica
$\overline{x}_1\,\overline{x}_2\,\overline{x}_3$	000
$\overline{x}_1\,\overline{x}_2\,x_3$	001
$\overline{x}_1\,x_2\,\overline{x}_3$	010
$\overline{x}_1\,x_2\,x_3$	011
$x_1\,x_2\,x_3$	111

Lembre-se que os mintermos da função são os produtos canônicos associados às entradas para as quais a função toma valor 1.

Em seguida, as células do mapa correspondentes a esses mintermos devem ser marcadas com 1, conforme mostrado no mapa da esquerda na figura 6.4.

O processo de minimização consiste, então, em procurar, para cada 1 no mapa, o maior agrupamento retangular formado por 1's adjacentes ao 1 em questão. Os agrupamentos devem sempre conter 2^k elementos ($k >= 1$) e eles são chamados **cubos maximais**. No exemplo da figura 6.4, o maior agrupamento com tais características que cobre 000 é o cubo `0XX`. De fato, o cubo `0XX` cobre os mintermos 000, 001, 010 e 011. Algebricamente, esse cubo é equivalente a $\overline{x}_1\,\overline{x}_2\,\overline{x}_3 + \overline{x}_1\,\overline{x}_2\,x_3 + \overline{x}_1\,x_2\,\overline{x}_3 + \overline{x}_1\,x_2\,x_3$ que por sua vez é equivalente a $\overline{x}_1\,\overline{x}_2\,(\overline{x}_3 + x_3) + \overline{x}_1\,x_2\,(\overline{x}_3 +$

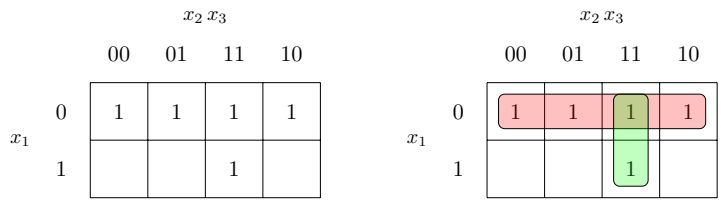


Figura 6.4: Exemplo do uso do mapa de Karnaugh: minimização da função $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$. À esquerda, as células correspondentes às entradas nas quais f toma valor 1 são preenchidas com 1. À direita, o resultado da minimização: em verde o intervalo `X11` que corresponde ao produto $x_2\,x_3$ e em vermelho o intervalo `0XX` que corresponde ao produto $\overline{x}_1$.

$x_3) = \overline{x}_1\,\overline{x}_2 + \overline{x}_1\,x_2 = \overline{x}_1\,(\overline{x}_2 + x_2) = \overline{x}_1$. Isto é, o processo de agrupar 1s, do ponto de vista algébrico, corresponde ao processo de eliminar variáveis substituindo dois ou mais produtos por um. Observe que a expressão $\overline{x}_1$ corresponde ao cubo `0XX`.

Voltando ao mapa da figura 6.4, o cubo `0XX` não cobre o elemento 111. Assim, tomamos também o maior cubo que cobre 111, que no caso é o cubo `X11`. Depois desse procedimento, todos os mintermos da função encontram-se cobertos por algum cubo. Assim, podemos dizer que uma solução SOP minimal corresponde aos cubos `0XX` e `X11`. O produto correspondente ao cubo `0XX` é $\overline{x}_1$ e o correspondente a `X11` é $x_2\,x_3$. Portanto, uma forma SOP minimal é $f(x_1, x_2, x_3) = \overline{x}_1 + x_2\,x_3$.

Exemplo: Minimize a função $f(a, b, c, d) = \sum m(0, 2, 3, 5, 6, 7, 8, 10, 11, 14, 15)$. A resposta é $f(a, b, c, d) = c + \overline{a}\,b\,d + \overline{b}\,\overline{d}$. Veja o mapa da figura 6.5. Note que os elementos dos quatro cantos formam um cubo.

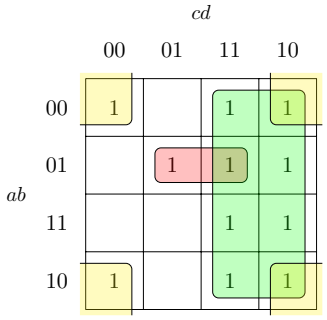


Figura 6.5: Minimização da função $f(a, b, c, d) = \sum m(0, 2, 3, 5, 6, 7, 8, 10, 11, 14, 15)$. Em verde temos `XX1X` (c), em vermelho `01X1` ($\overline{a}\,b\,d$) e em amarelo `X0X0` ($\overline{b}\,\overline{d}$).

Exemplo: Minimize a função $f(a, b, c, d) = \sum m(0, 4, 5, 6, 7, 8, 9, 10, 11, 14, 15)$. Neste caso, há mais de uma solução. O mapa mais à esquerda da figura 6.6 mostra todos os cubos maximais de f. As possíveis soluções são listadas a seguir, duas das quais são ilustradas na figura 6.6 (os

dois mapas da direita).

$$f(a,b,c,d) = \overline{a}\,b + a\,\overline{b} + \overline{a}\,\overline{c}\,\overline{d} + b\,c \qquad f(a,b,c,d) = \overline{a}\,b + a\,\overline{b} + \overline{b}\,\overline{c}\,\overline{d} + b\,c$$
$$f(a,b,c,d) = \overline{a}\,b + a\,\overline{b} + \overline{a}\,\overline{c}\,\overline{d} + a\,c \qquad f(a,b,c,d) = \overline{a}\,b + a\,\overline{b} + \overline{b}\,\overline{c}\,\overline{d} + a\,c$$

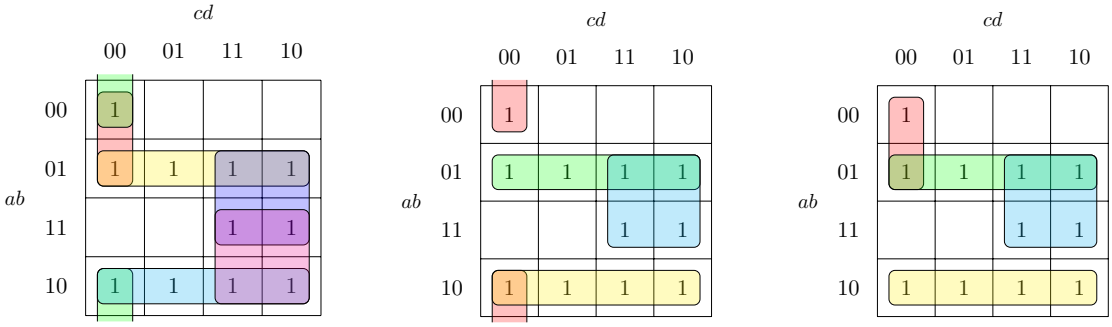


Figura 6.6: Minimização da função $f(a,b,c) = \sum m(0,4,5,6,7,8,9,10,11,14,15)$. Esquerda: todos os cubos maximais (verde=`X000`, salmão=`0X00`, amarelo=`01XX`, azul=`10XX`, lilás=`X11X`, rosa=`1X1X`). Centro: uma solução. Direita: outra solução. Note que há mais duas (e também que as cores não se mantém do primeiro mapa para os demais ...)

Este exemplo mostra que a solução não é única (ou seja, podem existir mais de uma forma SOP minimal de mesmo custo) e que nem todos os cubos maximais[3] fazem parte da forma SOP minimal.

Mapa de Karnaugh para encontrar a forma POS minimal: o mapa de Karnaugh pode ser utilizado também para encontrar a forma POS (produto de somas) minimal de uma função booleana. Considere a função $f(a,b,c) = \sum m(0,4,5,7)$. A minimização SOP de f por mapa de Karnaugh é mostrada na figura 6.7, à esquerda. À direita é mostrada a minimização de $\overline{f}(a,b,c) = \sum m(1,2,3,6)$

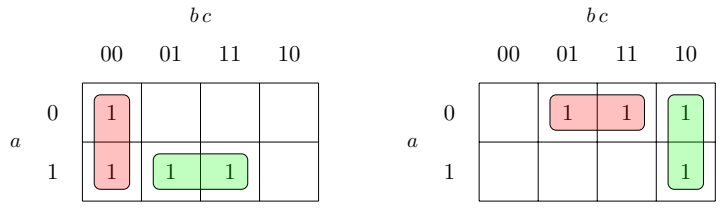


Figura 6.7: À esquerda, minimização da função $f(a,b,c) = \sum m(0,4,5,7)$ (em vermelho `X00` correspondente a $\overline{b}\,\overline{c}$ e, em verde, `1X1` correspondente a $a\,c$). À direita, minimização da função $\overline{f}(a,b,c) = \sum m(1,2,3,6)$

[3]Cubo maximal é o equivalente a um produto que não pode mais ser simplificado, isto é, se eliminarmos mais uma variável do produto, a função resultante passaria a tomar valor 1 em uma entrada na qual ela vale 0. No caso do cubo ou intervalo no mapa de Karnaugh, maximal significa que se estendermos o cubo ele passaria a cobrir um zero, algo que não queremos.

A partir do mapa da direita, temos que $\overline{f}(a,b,c) = b\,\overline{c} + \overline{a}\,c$. Usando o fato de que $f = \overline{\overline{f}}$, segue que $f = \overline{b\,\overline{c} + \overline{a}\,c} = (\overline{b\,\overline{c}})(\overline{\overline{a}\,c}) = (\overline{b} + c)(a + \overline{c})$.

Tudo isto pode ser diretamente realizado no mapa de Karnaugh conforme mostrado na figura 6.8. Em vez de marcar os mintermos de f no mapa, marcamos os maxtermos de f (ou seja, os mintermos de $\overline{f}$ ou, ainda, os zeros da função f). Aplica-se o processo de encontrar os cubos maximais em relação aos zeros, da mesma forma que fizemos com os 1s no caso da forma SOP. Para escrever a função na forma POS minimal, precisamos escrever, para cada cubo maximal encontrado, a soma que toma valor zero para todos os elementos do cubo. Por exemplo, no caso do cubo `0X1` queremos a soma que toma valor 0 para as entradas 001 e 011 (e apenas para essas duas entradas). Sabemos que o termo produto correspondente ao cubo `0X1` é $\overline{a}\,c$ e ele toma valor 1 para as entradas 001 e 011. Logo, o que queremos é $\overline{\overline{a}\,c}$ que é igual à soma $a + \overline{c}$. Fazendo o mesmo para o segundo cubo, temos então que a forma POS minimal é $f(a,b,c) = (a + \overline{c})(\overline{b} + c)$.

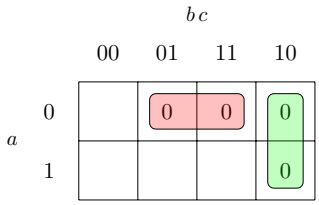


Figura 6.8: Minimização na forma POS da função $f(a,b,c) = \prod M(1,2,3,6)$ (em vermelho o cubo `0X1` que corresponde à soma $a + \overline{c}$ e, em verde, o cubo `X10` que corresponde à soma $\overline{b} + c$.

6.3.2 Minimização na presença de don't cares

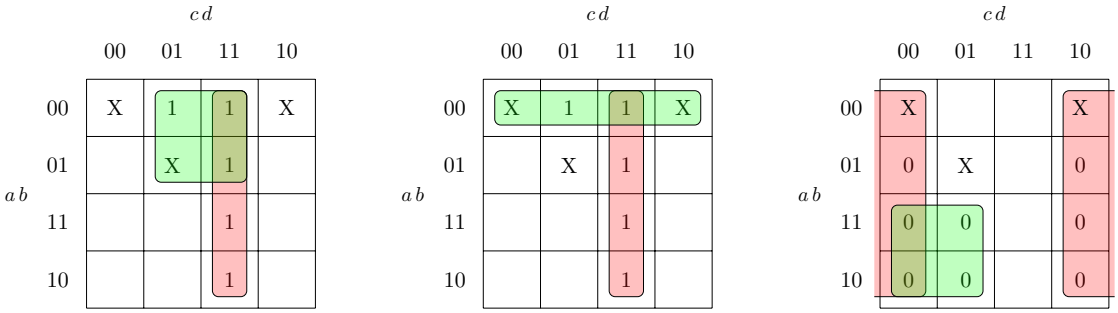
Em algumas situações, o valor de uma função para algumas entradas não é relevante (tipicamente porque tais entradas nunca ocorrerão na prática). Em tais situações, tanto faz se a função toma valor 0 ou 1 nessas entradas, que serão denominadas de **don't cares**.

Para minimizar uma função incompletamente especificada, os *don't cares* podem ser utilizados da forma mais conveniente possível. Isto é, se a minimização é na forma SOP, os *don't cares* podem ser usados para formar cubos maximais junto com os 1's da função. Analogamente, se a minimização é na forma POS, eles podem ser usados para formar cubos maximais junto com os 0's da função. Podem também ser completamente ignorados.

Por exemplo, seja $f(a,b,c,d) = \sum m(1,3,7,11,15) + d(0,2,5)$. Esta função f vale 1 para 0001, 0011, 0111, 1011, e 1111; vale 0 para 0100, 0110, 1000, 1001, 1010, 1100, 1101 e 1110; para 0000, 0010 e 0101 não importa o valor de f. No mapa, preenchemos *don't cares* com X (veja abaixo).

ab \ cd	00	01	11	10
00	X	1	1	X
01	0	X	1	0
11	0	0	1	0
10	0	0	1	0

Para minimizar na forma SOP, consideramos os 1 e os X, conforme os dois mapas da esquerda abaixo. Esses dois mapas ilustram duas possíveis soluções de mesmo custo. O primeiro corresponde a $\overline{a}d + cd$ e o segundo a $\overline{a}\,\overline{b} + cd$. Já para minimizar na forma POS, consideramos os 0 e X (mapa mais a direita). O resultado na forma POS é $d\,(\overline{a} + c)$. Note que em ambos os casos os *don't cares* ajudam na minimização. Ao compararmos a forma SOP e POS minimais, concluímos que a POS minimal é vantajosa pois requer uma porta OU e uma porta E, enquanto a forma SOP requer duas portas E e uma porta OU.



6.4 Minimização de múltiplas funções

Sejam as funções f_1 e f_2 dadas pelas seguintes tabelas:

$x_1\,x_2\,x_3$	$f_1(x_1,x_2,x_3)$
000	1
001	1
010	0
011	0
100	0
101	1
110	0
111	0

$x_1\,x_2\,x_3$	$f_2(x_1,x_2,x_3)$
000	0
001	0
010	0
011	0
100	0
101	1
110	1
111	1

Minimizando-se individualmente estas funções, temos

$$f_1(x_1,x_2,x_3) = \overline{x}_1\,\overline{x}_2 + \overline{x}_2\,x_3$$

$$f_2(x_1,x_2,x_3) = x_1\,x_3 + x_1\,x_2$$

Para implementá-las, são necessárias 4 portas E.

Note, porém, que podemos escrever

$$f_1(x_1\,x_2\,x_3) = \overline{x}_1\,\overline{x}_2 + \overline{x}_2\,x_3 = \overline{x}_1\,\overline{x}_2 + x_1\,\overline{x}_2\,x_3$$

$$f_2(x_1\,x_2\,x_3) = x_1\,x_3 + x_1\,x_2 = x_1\,x_2 + x_1\,\overline{x}_2\,x_3$$

Neste caso, nenhuma das expressões está na forma minimal, mas há um produto comum às duas funções. Logo para implementá-las são necessárias 3 portas E, pois uma das portas E pode ser compartilhada pelas duas funcões.

Este exemplo mostra que minimizar individualmente as funções não necessariamente representa a melhor solução para a minimização de múltiplas funções. Na minimização de múltiplas funções deve-se buscar também a maximização de compartilhamento.

6.5 PLA

A minimização lógica dois-níveis ganhou impulso na década de 1980 devido aos dispositivos conhecidos como **PLA** (*Programmable Logic Arrays*). Eles consistem de um conjunto de entradas, com uma malha programável de conexões para um conjunto de portas E, e uma malha programável de conexões entre as saídas das portas E para um conjunto de portas OU. Por malha programável entende-se que os cruzamentos podem ser conectados (programados) para conduzir o sinal. No estado inicial, nenhum cruzamento está conectado nas malhas de um PLA.

A figura 6.9 mostra um modelo lógico básico de um PLA típico, com 3 variáveis de entrada e três saídas. Os círculos (pontos pretos) sobre o cruzamento das linhas indicam onde há conexão. No exemplo, as portas lógicas E realizam, respectivamente de cima para baixo, as funções (produtos) $a\,\overline{b}$, $\overline{a}\,b\,\overline{c}$, $\overline{b}\,c$ e $a\,\overline{c}$; as portas lógicas OU realizam, respectivamente da esquerda para a direita, as funções $f_1(a,b,c) = \overline{a}\,b\,\overline{c} + a\,\overline{b}$, $f_2(a,b,c) = \overline{a}\,b\,\overline{c} + \overline{b}\,c$ e $f_3(a,b,c) = \overline{a}\,b\,\overline{c} + a\,\overline{c}$.

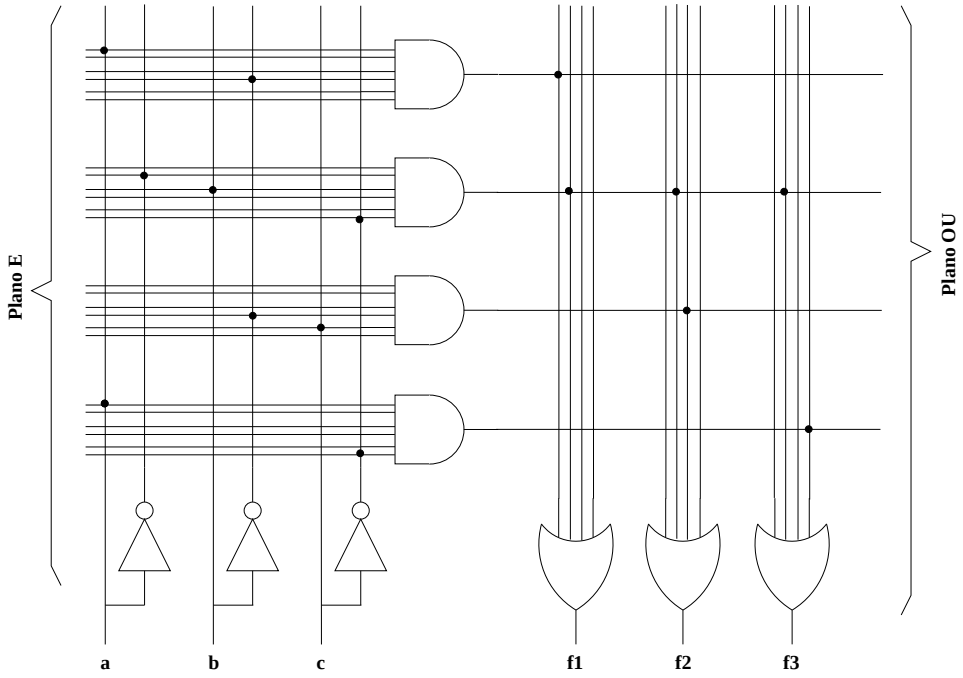


Figura 6.9: Esquema lógico de um PLA.

Para não sobrecarregar o diagrama, em geral desenha-se de forma simplificada como o mostrado na figura 6.10.

PLAs comerciais têm tipicamente[4] entre 10 e 20 entradas, entre 30 e 60 portas E (produtos) e entre 10 e 20 portas OU (saídas). Em um PLA com 16 entradas, 48 produtos e 8 saídas, existem $2 \times 16 \times 48 = 1536$ cruzamentos na malha E e $8 \times 48 = 384$ cruzamentos na malha OU. Um número considerável de funções relativamente complexas podem ser realizadas via um PLA. Claramente, quanto menor o número de variáveis e termos produtos utilizados na expressão de uma função, menor será o "tamanho" do PLA necessário para a realização da função.

Sejam as seguintes funções (já minimizadas individualmente) :

$f_0 = a$

$f_1 = \overline{b}$

[4]Informação colhida em torno de 2010 ...

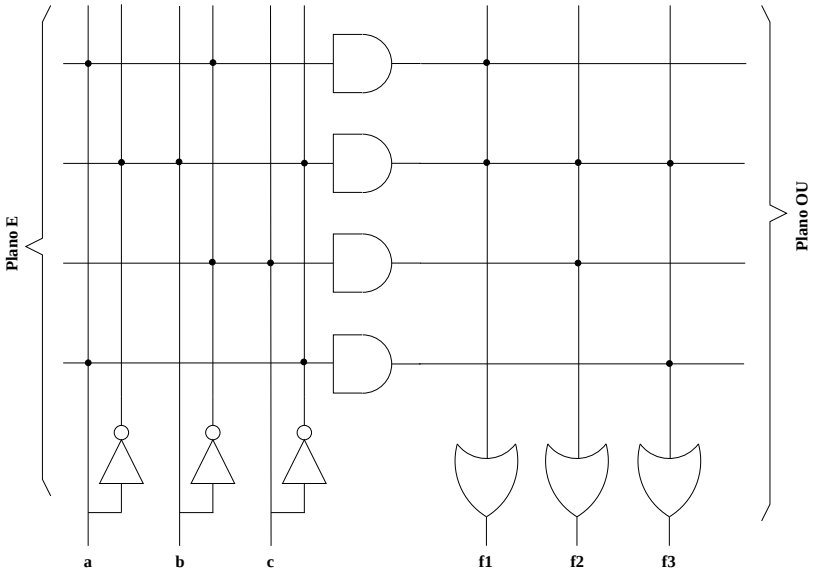


Figura 6.10: Esquema simplificado de um PLA.

$f_2 = bc + ab$

$f_3 = \overline{a}\,\overline{b} + \overline{a}\,c$

Se expressas desta forma, para implementá-las, serão necessárias 6 portas E. No entanto, note que elas podem ser reescritas como:

$f_0 = a\overline{b} + ab$

$f_1 = \overline{a}\overline{b} + a\overline{b}$

$f_2 = \overline{a}bc + ab$

$f_3 = \overline{a}\,\overline{b} + \overline{a}\,b\,c$

e neste caso são necessárias apenas 4 portas E.

Este é outro exemplo no qual mostra-se que o compartilhamento de portas entre múltiplas funções pode levar à redução do número total de portas necessárias para a realização das funções, mesmo que individualmente as expressões não estejam na forma minimal.

Observe, porém, que no exemplo acima, o número total de produtos foi reduzido às custas de aumento no número total de somas. Se o objetivo é a realização em um PLA (*Programmable logic array*) não é relevante se uma função é formada, por exemplo, por 2 ou 3 termos produto (desde que esse número de produtos não ultrapasse o número de portas E no PLA). Porém, se o objetivo for minimizar também o número de entradas das portas, então é preciso tomar cuidado para não aumentar demasidamente o número de termos produto (pois isso implicaria

em aumento no número de entradas da porta OU).

6.6 Método de Quine-McCluskey (*)

Mapas de Karnaugh representam uma maneira visual e intuitiva de se minimizar funções booleanas. No entanto, eles só se aplicam a funções com até 6 variáveis e não são sistemáticos (adequados para programação). O algoritmo tabular de Quine-McCluskey para minimização de funções Booleanas é um método clássico que sistematiza este processo de minimização para um número arbitrário de variáveis.

Tanto os mapas de Karnaugh como o algoritmo de Quine-McCluskey (QM) requerem que a função booleana a ser minimizada esteja na forma SOP canônica. A idéia básica do algoritmo QM consiste em encarar os mintermos da SOP canônica como pontos no n-espaço, ou seja, como vértices de um n-cubo. A partir do conjunto destes vértices (ou 0-cubos) procura-se gerar todos os 1-cubos possíveis combinando-se dois deles (equivale a gerar as arestas do cubo que ligam dois 0-cubos da função). Na figura 6.11 temos um exemplo no qual os 0-cubos 111 e 110 são agrupados para gerar o 1-cubo $11X$. Note, que o 3-cubo (cubo como você conhece) da figura é apenas uma outra representação do mapa de Karnaugh. Este processo de combinar

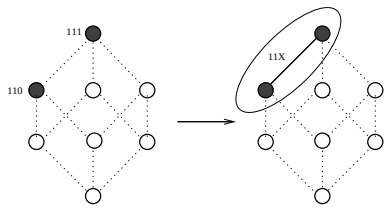


Figura 6.11: Passo elementar do algoritmo de Quine-McCluskey.

dois cubos corresponde, como já vimos no caso do mapa de Karnaugh, ao processo algébrico de simplificação. Os mintermos da expressão na forma canônica inicial correspondem a o que chamamos de 0-cubo. Combinar dois 0-cubos para gerar um 1-cubo corresponde a combinar dois mintermos para eliminar uma variável e gerar um termo com menos literais para substituí-los. No exemplo da figura 6.11, a simplificação algébrica correspondddnte é a seguinte:

$$x_1x_2x_3 + x_1x_2\overline{x}_3 = x_1x_2(x_3 + \overline{x}_3) = x_1x_2 \cdot 1 = x_1x_2$$

Enquanto no mapa de Karnaugh pode-se visualmente detectar o maior agrupamento válido de 0-cubos que inclui um 0-cubo específico, no método QM isso é realizado incrementalmente. Isto é, após gerados todos os possíveis 1-cubos, procura-se gerar todos os possíveis 2-cubos por meio da combinação de dois 1-cubos e assim por diante, até que nenhum cubo de dimensão

maior possa ser gerado a partir da combinação de dois cubos de dimensão menor. Note que neste processo estão envolvidos apenas os 0-cubos (vértices do cubo) que correspondem aos mintermos da função (i.e., o valor da função nesses vértices deve ser 1 necessariamente). Os cubos resultantes (aqueles que não foram combinados com nenhum outro) ao final de todo o processo são os **implicantes primos** (ou seja, cubos maximais) da função. Cubos maximais são exatamente os grupos que demarcamos no mapa de Karnaugh (inclusive os redundantes). Veja um exemplo a seguir.

Exemplo: Considere a expressão Booleana $f(a,b,c) = \sum m(0,1,3,7)$. Os implicantes primos dessa função são $00X$, $0X1$ e $X11$, conforme marcados no mapa de Karnaugh. Graficamente, estes implicantes primos (ou cubos) correspondem respectivamente aos intervalos $[000, 001]$, $[001, 011]$ e $[011, 111]$ ilustrados na figura 6.12(a).

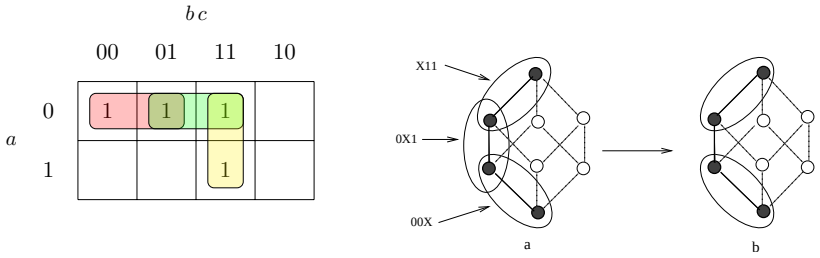


Figura 6.12: No mapa de Karnaugh, em vermelho temos os cubo `00X`, em verde `0X1`, e em amarelo `X11`. Os implicantes primos são mostrados em (a) e uma cobertura mínima em (b). O cubo `0X1` (ou implicante primo `0X1` ou ainda intervalo $[001, 011]$) é redundante.

Nem todos os implicantes primos fazem necessariamente parte da forma SOP minimal, como mostrado no exemplo acima. Portanto, o cálculo da forma SOP minimal de uma função booleana no algorimo QM é dividido em duas etapas:

1. Cálculo de todos os implicantes primos da função
2. Cálculo de uma cobertura (subconjunto dos implicantes primos) mínima da função

O ponto central da segunda etapa é o cálculo de um menor subconjunto do conjunto de implicantes primos, que sejam suficientes para cobrir[5] todos os mintermos da função. Tal conjunto é denominado uma **cobertura mínima**.

No caso de mapas de Karnaugh, estas duas etapas são realizadas conjuntamente de forma um tanto "intuitiva". No caso do algoritmo QM, estas etapas são realizadas explícita e separadamente. As etapas são descritas a seguir, apoiando-se em exemplos.

[5] Um conjunto de implicantes primos (cubos maximais) cobre um mintermo (0-cubo) se este é coberto por pelo menos um dos implicantes primos.

6.6.1 Cálculo de implicantes primos

A primeira etapa do algoritmo QM consiste de um processo para determinação de todos os implicantes primos. A seguir descrevemos os passos que constituem esta etapa, mostrando como exemplo o cálculo dos implicantes primos da função $f(x_1, x_2, x_3) = \sum m(0, 1, 4, 5, 6)$.

- Primeiro passo : converter os mintermos para a notação binária.

  000, 001, 100, 101, 110

- Segundo passo : Separar os mintermos em grupos de acordo com o número de 1's em sua representação binária e ordená-los em ordem crescente, em uma coluna, separando os grupos com uma linha horizontal.

000
001
100
101
110

- Terceiro passo : combinar todos os elementos de um grupo com todos os elementos do grupo adjacente inferior para geração de cubos de dimensão maior. Para cada 2 grupos comparados entre si, gerar um novo grupo na próxima coluna e colocar os novos cubos. Marcar com $\surd$ os cubos que foram usados para gerar novos cubos.

$\surd$	000
$\surd$	001
$\surd$	100
$\surd$	101
$\surd$	110

$\Longrightarrow$

$00X$
$X00$
$X01$
$10X$
$1X0$

  Observação : o novo cubo gerado será inserido no novo conjunto se e somente se ele ainda não estiver contido nele.

  Repetir o processo para cada nova coluna formada, até que nenhuma combinação mais seja possível.

$\surd$	000
$\surd$	001
$\surd$	100
$\surd$	101
$\surd$	110

$\Longrightarrow$

$\surd$	$00X$
$\surd$	$X00$
$\surd$	$X01$
$\surd$	$10X$
	$1X0$

$\Longrightarrow$

$X0X$

- Quarto passo : Listar os implicantes primos. Os implicantes primos são os cubos que não foram combinados com nenhum outro, ou seja, aqueles que não estão com a marca $\surd$.

  $1X0$ e $X0X$

Cálculo de uma cobertura mínima

Uma cobertura mínima pode ser calculada com a ajuda de uma tabela denominada **Tabela de Implicantes Primos**. Este processo é mostrado a seguir, usando como exemplo a minimização da função $f(x_1, x_2, x_3, x_4, x_5) = \sum(1, 2, 3, 5, 9, 10, 11, 18, 19, 20, 21, 23, 25, 26, 27))$.

1. Construir a Tabela de Implicantes Primos: No topo das colunas desta tabela deve-se colocar os mintermos de f e, à esquerda de cada linha, os implicantes primos. Por simplicidade, no cabeçalho das colunas são colocados os índices decimais dos mintermos (isto é, 3 corresponde a m_3 ou ainda à entrada 00011). Os implicantes primos devem ser listados em ordem decrescente de acordo com a sua dimensão, isto é, em ordem crescente de acordo com o número de literais. Deve-se acrescentar uma coluna à esquerda e uma linha na parte inferior da tabela.

   Em cada linha, marcar as colunas com $\surd$ quando o implicante primo da linha cobre o mintermo da coluna.

		1	2	3	5	9	10	11	18	19	20	21	23	25	26	27
	XX01X		$\surd$	$\surd$			$\surd$	$\surd$	$\surd$	$\surd$					$\surd$	$\surd$
	X10X1					$\surd$		$\surd$						$\surd$		$\surd$
	0X0X1	$\surd$		$\surd$		$\surd$		$\surd$								
	00X01	$\surd$			$\surd$											
	X0101				$\surd$							$\surd$				
	1010X										$\surd$	$\surd$				
	10X11									$\surd$			$\surd$			
	101X1											$\surd$	$\surd$			

2. Selecionar os implicantes primos essenciais: deve-se procurar na tabela as colunas que contém apenas uma marca $\surd$. A linha na qual uma dessas colunas contém a marca $\surd$ corresponde a um implicante primo essencial. Em outras palavras, este implicante primo é o único que cobre o mintermo da coluna e, portanto, não pode ser descartado. Então, deve-se marcar com um asterisco (*) esta linha na coluna mais à esquerda, para indicar que este é um implicante primo essencial. A seguir, deve-se marcar, na última linha da tabela, todas as colunas cujo mintermo é coberto pelo implicante primo selecionado.

   No exemplo, o mintermo 2 é coberto apenas pelo implicante primo $XX01X$. Logo $XX01X$ é essencial.

		1	2	3	5	9	10	11	18	19	20	21	23	25	26	27
*	XX01X		√	√			√	√	√	√					√	√
	X10X1					√		√						√		√
	0X0X1	√		√		√		√								
	00X01	√			√											
	X0101				√							√				
	1010X										√	√				
	10X11									√			√			
	101X1											√	√			
			√	√			√	√	√	√					√	√

A linha correspondente a um implicante primo essencial, bem como as colunas cujos mintermos são cobertos por esse implicante primo, devem ser desconsiderados no prosseguimento do processo.

Deve-se repetir o processo enquanto existir, na tabela restante, algum implicante primo essencial.

No exemplo, vemos que o mintermo 25 é coberto apenas pelo implicante primo $X10X1$ e que o mintermo 20 é coberto apenas pelo implicante primo $1010X$. Logo, esses dois implicantes primos também são essenciais.

		1	2	3	5	9	10	11	18	19	20	21	23	25	26	27
*	XX01X		√	√			√	√	√	√					√	√
*	X10X1					√		√						√		√
	0X0X1	√		√		√		√								
	00X01	√			√											
	X0101				√							√				
*	1010X										√	√				
	10X11									√			√			
	101X1											√	√			
			√	√		√	√	√	√	√	√	√		√	√	√

3. Reduzir a tabela: eliminar as colunas cujos mintermos já foram cobertos (ou seja, manter apenas as colunas correspondentes aos mintermos não cobertos pelos implicantes primos essenciais). Eliminar as linhas correspondentes aos implicantes primos essenciais e as linhas que não cobrem nenhum dos mintermos restantes na tabela.

   No exemplo, após a redução, temos a seguinte tabela:

		1	5	23
	0X0X1	√		
	00X01	√	√	
	X0101		√	
	10X11			√
	101X1			√

4. Selecionar os implicantes primos secundariamente essenciais: eliminar as linhas dominadas e as colunas dominantes e, em seguida, selecionar os essenciais.

   Colunas dominantes: Diz-se que uma coluna β na Tabela de Implicantes Primos domina uma coluna α se e somente se todos os implicantes que cobrem o mintermo da coluna α cobrem também o mintermo da coluna β. Se β domina α, então a coluna β pode ser removida da tabela. (Por quê?)

   Linhas dominadas ou equivalentes: Sejam A e B duas linhas na Tabela de Implicantes Primos reduzida. Então dizemos que a linha A domina B se o implicante da linha A cobre, ao menos, todos os mintermos cobertos pelo implicante da linha B. Dizemos que as linhas A e B são equivalentes se os respectivos implicantes primos cobrem exatamente os mesmos mintermos na tabela. Se A domina B, ou se A e B são equivalentes, e **se o custo do implicante da linha A é menor ou igual ao da linha B** (ou seja, o implicante da linha A possui não mais literais que o implicante da linha B), então a linha B pode ser eliminada da tabela (Por quê?).

   Após a eliminação de colunas dominantes e linhas dominadas (ou equivalentes), deve-se repetir o mesmo processo do passo 2, porém os implicantes primos essenciais serão chamados secundariamente essenciais e marcados com dois asteriscos (**).

   No exemplo, a linha do implicante primo $X0101$ pode ser eliminada pois é dominada pela linha do implicante $00X01$. A linha do implicante $101X1$ pode ser eliminada pois é equivalente a do implicante $10X11$. Neste último caso, note que, alternativamente, podemos eliminar a linha do implicante $10X11$ em vez da linha do implicante $101X1$.

		1	5	23
	0X0X1	√		
**	00X01	√	√	
**	10X11			√
		√	√	√

   Deve-se repetir o processo descrito neste passo até que não seja mais possível fazer qualquer eliminação ou até que a tabela fique vazia. Se a tabela não ficar vazia, a tabela restante é chamada **tabela cíclica**.

5. Resolver a tabela cíclica: Para isso, uma possível abordagem é o método de Petrick, um método de busca exaustiva. Ele fornece todas as possíveis combinações dos implicantes primos restantes que são suficientes para cobrir os mintermos ainda não cobertos pelos implicantes primos já selecionados. Deve-se escolher, dentre essas combinações, uma que envolve o menor número de termos. Caso existam mais de uma nestas condições, deve-se escolher a de custo mínimo (aquela que envolve menor número de literais).
   Exemplo: Considere a tabela cíclica a seguir:

		0	4	13	15	10	26	16
a	0X10X		√	√				
b	011XX			√	√			
c	01X1X				√	√		
d	1X0X0						√	√
e	00X00	√	√					
f	X1010					√	√	
g	X0000	√						√

Para que todos os mintermos da tabela cíclica sejam cobertos, a seguinte expressão deve ser verdadeira.

$$(e+g)(a+e)(a+b)(b+c)(c+f)(d+f)(d+g) = 1$$

Uma soma nesta expressão indica quais implicantes primos poderiam ser escolhidos para cobrir um mintermo (uma coluna). O produto de somas expressa, portanto, todas as combinações possíveis de escolha para se cobrir todas as colunas. Transformando esta expressão em soma de produtos, obtemos todas as possíveis soluções (cada produto é uma solução viável). Dentre os produtos, deve-se escolher aquele(s) de menor custo (menor número de implicantes primos e implicantes com menor número de literais). (Se eu não errei nos cálculos, as soluções são $\{a, c, d, e\}$, $\{b, c, d, e\}$ e $\{a, c, d, g\}$ pois os outros tem custo maior).

Outro exemplo: Considere a tabela cíclica a seguir e suponha que o custo de A é menor que o de B e que o custo de C é menor que o de D.

	m_1	m_2	m_3
A	√		
B	√	√	
C			√
D		√	√

Então as possíveis soluções são $(A+B)(B+D)(C+D) = (AB+AD+B+BD)(C+D) = (B+AD)(C+D) = BC+BD+ACD+AD$. Dos que envolvem dois implicantes, certamente o custos de BC e de AD são menores que o custo de BD. Então a escolha final fica entre BC e AD.

Resumo do procedimento para cálculo de cobertura mínima:

1. Montar a tabela de implicantes primos
2. Identificar todos os implicantes primos essenciais e eliminar as linhas correspondentes, bem como as colunas dos mintermos cobertos por esses implicantes.

3. Eliminar colunas dominantes: Se uma coluna β tem $\surd$ em todas as linhas que uma outra coluna α tem $\surd$, a coluna β é dominante e pode ser eliminada (pois se escolhermos um implicante primo que cobre α, β será necessariamente coberto também).

4. Eliminar linhas dominadas ou equivalentes: se uma linha A tem $\surd$ em todas as colunas em que a linha B tem $\surd$, então a linha A domina a linha B. Se elas tem $\surd$ exatamente nas mesmas colunas, então elas são equivalentes. Se, além disso, o número de literais de A é menor que o de B, então a linha B pode ser eliminada (pois se tivéssemos uma cobertura envolvendo B, ao trocarmos B por A na cobertura teríamos uma cobertura de menor custo).

   Observação: Se o objetivo da minimização é encontrar apenas UMA solução minimal (e NÃO TODAS), então podemos eliminar uma linha B se existe uma linha A tal que A domina B, ou A é equivalente a B, e ambos têm um mesmo custo.

5. Identificar os implicantes essenciais secundários e eliminar as linhas correspondentes, bem como as colunas dos mintermos cobertos por esses implicantes.

6. Repetir 3, 4 e 5 enquanto possível

7. Se a tabela não estiver vazia, aplicar o método de Petrick (que lista todas as possíveis soluções para o restante da tabela) e escolher uma solução de custo mínimo.

Exemplo: Considere a função $f(a,b,c) = a\,\overline{b}\,c + \overline{a}\,b\,\overline{c} + a\,b\,\overline{c} + a\,b\,c = \sum m(2,5,6,7)$.

Podemos realizar a simplificação algébrica da seguinte forma:

$$\begin{aligned} f(a,b,c) &= a\,\overline{b}\,c + \overline{a}\,b\,\overline{c} + a\,b\,\overline{c} + a\,b\,c \\ &= a\,\overline{b}\,c + a\,b\,c + \overline{a}\,b\,\overline{c} + a\,b\,\overline{c} + a\,b\,\overline{c} + a\,b\,c \\ &= a\,c + b\,\overline{c} + a\,b \\ &= a\,c + b\,\overline{c} \end{aligned}$$

Por QM temos

$\surd$	010
$\surd$	101
$\surd$	110
$\surd$	111

$\Longrightarrow$

$X10$
$1X1$
$11X$

Os implicantes primos são $X10$ ($b\,\overline{c}$), $1X1$ ($a\,c$) e $11X$ ($a\,b$). Uma cobertura mínima pode ser calculada usando-se a Tabela de Implicantes Primos.

		2	5	6	7
*	X10	√		√	
*	1X1		√		√
	11X			√	√
		√	√	√	√

Os implicantes primos $X10$ e $1X1$ são essencias e cobrem todos os mintermos da função. Logo formam uma cobertura mínima.

6.6.2 Funções incompletamente especificadas

Para minimizar uma função incompletamente especificada pelo algoritmo QM, é interessante considerarmos todos os *don't cares* na etapa de cálculo dos implicantes primos, pois isto aumenta a chance de se obter cubos maiores (portanto produtos com menos literais). Observe que, durante as iterações para a geração dos implicantes primos, um cubo que cobre apenas *don't cares* não deve ser eliminado pois ele pode, eventualmente em iterações futuras, se juntar a outro cubo para formar outro cubo maior.

De forma mais genérica do que a vista anteriormente, podemos caracterizar uma função booleana f através dos seus conjuntos um, zero e dc (de *don't care*), definidos respectivamente por $f\langle 1\rangle = \{\mathbf{x} \in \{0,1\}^n : f(\mathbf{x}) = 1\}$, $f\langle 0\rangle = \{\mathbf{x} \in \{0,1\}^n : f(\mathbf{x}) = 0\}$ e $f\langle *\rangle = \{0,1\}^n \setminus (f\langle 1\rangle \cup f\langle 0\rangle)$.

Na parte de cálculo dos implicantes primos devem ser utilizados todos os elementos de $f\langle 1\rangle \cup f\langle *\rangle$. Na parte de cálculo de uma cobertura mínima devem ser considerados apenas os elementos de $f\langle 1\rangle$.

6.7 Outros algoritmos de minimização lógica 2-níveis

Algoritmos de minimização tabular (como o Quine-McCluskey) têm as seguintes características:

- eles listam explicitamente todos os implicantes primos, cuja quantidade pode ser de ordem exponencial no número de variáveis n
- requerem que a função a ser minimizada esteja na forma SOP canônica. Não é raro que, juntamente com os *don't cares*, o número de produtos canônicos seja da ordem de 2^{n-1}
- a tabela-cíclica pode ser bem grande (e não há algoritmo eficiente para o cálculo da solução ótima)

Do ponto de vista computacional, isto implica em consumo de grande quantidade de memória e tempo de processamento longo.

Muito esforço ocorreu nas décadas de 1980 e início da década de 1990 no sentido de se desenvolver programas para minimização de funções com várias variáveis, motivados pelos PLAs. Os esforços realizados foram no sentido de achar alternativas que não requeressem o cálculo explícito de todos os implicantes primos, formas eficientes para o cálculo de coberturas mínimas e heurísticas eficientes nesses processos.

Um dos resultados do esforço foi a criação do algoritmo ESPRESSO (desenvolvido por um grupo da University of California – Berkeley [Brayton et al., 1984, McGreer et al., 1993]) que foi referência para minimização lógica dois-níveis durante um tempo. Para maiores detalhes veja [Brayton et al., 1984], capítulo 7 de [Micheli, 1994], seção 6.10 de [Hill and Peterson, 1993]. ESPRESSO não requer que a função a ser minimizada esteja na forma SOP canônica, não calcula os implicantes primos explicitamente, e utiliza uma série de heurísticas. Além disso, ele também realiza a minimização conjunta de múltiplas funções e de funções multi-valoradas (lógica multi-valores). Depois do ESPRESSO, foram propostas outras melhorias (como o uso de *BDD — Binary Decision Diagrams*) por Coudert e outros [Coudert, 1994, Coudert, 1995], e o BOOM [Hlavicka and Fiser, 2001, Fišer and Hlavicka, 2003][6]

[6] Não tenho conhecimentos sobre desenvolvimentos posteriores ao BOOM.

Capítulo 7

Circuitos combinacionais

Última atualização em 15/06/2020

O termo **combinacional** no título é utilizado na área de circuitos lógicos em contraste aos circuitos do tipo **sequencial**. Nos circuitos combinacionais, os valores de saída são determinados unicamente apenas a partir dos valores de entrada. Todos os circuitos que vimos até este momento são do tipo combinacional. Já os circuitos sequenciais são aqueles capazes de "armazenar" um valor (manter um certo estado) e nos quais os valores de saída são determinados em função não apenas dos valores de entrada, mas também do estado atual do circuito. Circuitos sequenciais serão estudados em um capítulo mais adiante.

Neste capítulo serão apresentados alguns circuitos combinacionais básicos, úteis como componentes "caixa-preta" na construção de circuitos complexos.

7.1 Somador

O circuito somador já foi discutido no capítulo 3. Ele é parte fundamental de qualquer ULA (unidade lógico-aritmética).

7.2 Comparador

Outro circuito que geralmente está presente em uma ULA é o circuito comparador. Podemos ter circuitos que simplesmente verificam-se dois números são iguais, ou se um número é igual a zero, ou ainda comparam a magnitude e respondem se são iguais, ou se um é maior ou menor que

outro. As comparações de magnitude podem ser realizadas levando-se em conta a interpretação com sinal ou sem sinal dos binários.

Aqui apresentamos apenas um comparador de *bits*, que pode ser composto em cascata, para se construir um comparador de números de n *bits*. O circuito da figura 7.1 possui três entradas de um *bit* cada. O *bit* 1 ao topo funciona como um habilitador do comparador: quando seu valor for zero, nenhuma saída será ativada; quando é 1, uma e somente uma das saídas será ativada, indicando respectivamente se $a = b$, $a > b$ ou $a < b$.

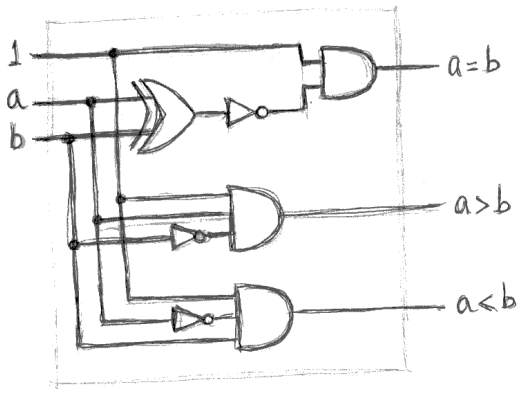


Figura 7.1: Comparador de *bits*. A entrada 1 funciona como um habilitador do comparador; se trocada por zero, todas as saídas irão a zero.

O camparador de *bits* não precisaria dessa entrada 1. No entanto, essa entrada é conveniente se pretendemos construir um comparador de dois números de n *bits*. Supondo dois números sem sinal, $\mathbf{A} = a_{n-1} \ldots a_1 a_0$ e $\mathbf{B} = b_{n-1} \ldots b_1 b_0$, a estratégia natural para a comparação seria começarmos do *bit* mais significativo e avançar em direção ao menos significativo, *bit* por *bit*, enquanto for necessário. Isto é, se ao compararmos o *bit* a_{n-1} com o *bit* b_{n-1} verificarmos que $a_{n-1} > b_{n-1}$ ou que $a_{n-1} < b_{n-1}$, não há necessidade de compararmos o restante dos *bits*. Apenas quando $a_{n-1} = b_{n-1}$ há necessidade de proceder a comparação entre a_{n-2} e b_{n-2}. Assim, pode-se conectar n comparadores de *bits* do tipo acima em cascata. A igualdade só será verificada na saída $a = b$ do último deles. Já as desigualdades podem ser identificadas na saída $a > b$ ou $a < b$ de qualquer um deles (note que, quando em cascata, apenas um dos comparadores de *bits* irá gerar tal sinalização).

Embora essa solução seja simples, uma desvantagem é o fato de que o sinal precisa propagar pelos n módulos na cascata. Para reduzir o tamanho da cascata, pode-se construir por exemplo um comparador de números de 4 *bits* (a partir da tabela-verdade, por exemplo) e então usá-lo em cascata de forma similar ao descrito acima. Para comparar números de 8 *bits*, precisaríamos de dois desses módulos, enquanto que se estivéssemos usando o comparador de *bits* da figura 7.1 precisaríamos de 8 deles.

7.3 Multiplexadores e demultiplexadores

7.3.1 Multiplexadores

Multiplexadores (também conhecidos como seletores de dados ou MUX) são circuitos combinacionais com n entradas de dados, uma saída e entradas de controle denominados seletores. A saída desse circuito é sempre uma cópia de uma das entradas. Dentre as n entradas, a que é enviada para a saída é justamente aquela cujo índice é igual ao valor codificado nos *bits* seletores. A figura 7.2 mostra um multiplexador 4×1, isto é, um multiplexador de 4 entradas.

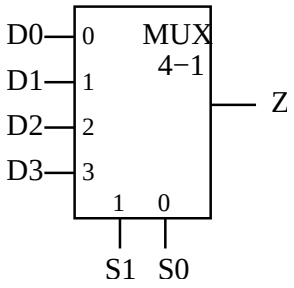


Figura 7.2: Multiplexador de 4 entradas.

O dado a ser enviado para a saída depende do valor dos seletores e está especificado na tabela a seguir:

$s_1\, s_0$	z
00	D_0
01	D_1
10	D_2
11	D_3

Podemos observar que z é uma função dos *bits* seletores s_1, s_0 e dos dados de entrada D_0, D_1, D_2, D_3 (que podem ser pensadas como variáveis lógicas). Assim, podemos escrever z como

$$z(D_0, D_1, D_2, D_3, s_1, s_0) = D_0\, \overline{s}_1\, \overline{s}_0 + D_1\, \overline{s}_1\, s_0 + D_2\, s_1\, \overline{s}_0 + D_3\, s_1\, s_0$$

Note que para cada possível combinação de valores de $s_1\, s_0$, apenas um dos produtos toma valor 1 na equação acima. Por exemplo, se $s_1\, s_0 = 10$, apenas o mintermo $s_1\, \overline{s}_0$ terá valor 1 e teremos então $z = D_2$. O mesmo raciocínio se aplica para os demais valores de $s_1\, s_0$.

Uma realização simples de multiplexadores é na forma de circuitos dois-níveis, consistindo de n portas E no primeiro nível e uma porta OU no segundo nível. A realização do multiplexador de 4 entradas é mostrado na figura 7.3.

De forma geral, se um multiplexador possui k *bits* seletores, pode possuir até 2^k entradas (ou,

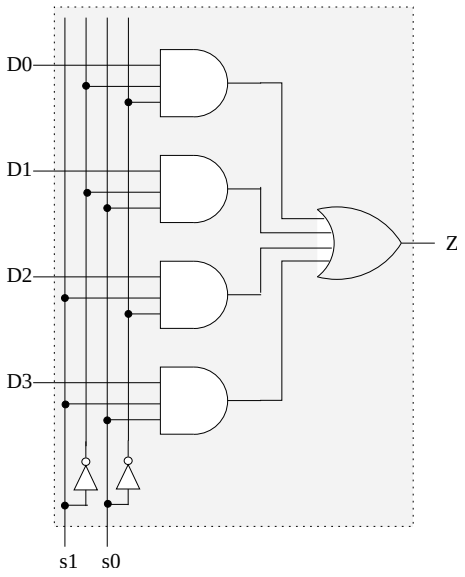


Figura 7.3: Circuito de um multiplexador de 4 entradas.

observado de outro ponto de vista, se possui n entradas de dados, deve possuir $k \geq \log_2 n$ *bits* seletores). No exemplo acima, de 4 entradas, 2 *bits* seletores são necessários e suficientes.

Um MUX pode ser realizado como composição de múltiplos MUX com um menor número de entradas cada. Por exemplo, um MUX 4×1 pode ser realizado usando três MUX 2×1, conforme ilustrado na figura 7.4.

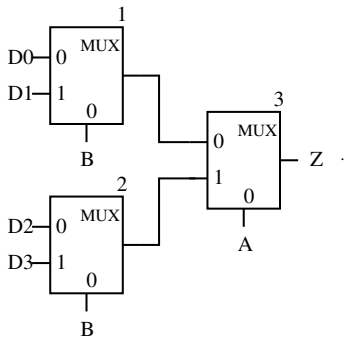


Figura 7.4: Realização de um MUX 4×1 como composição de três MUX 2×1.

Note que se $AB = 00$, a saída do MUX 1 é D_0 e a do MUX 2 é D_2 (já que $B = 0$) e a saída do MUX 3 é D_0 (já que $A = 0$ também). Se $AB = 01$ então as saídas são, respectivamente, D_1, D_3 e D_1. Se $AB = 10$, então as saídas são, respectivamente, D_0, D_2 e D_2. Finalmente, se $AB = 11$, então as saídas são, respectivamente, D_1, D_3 e D_3. Em todos os casos, a saída z da composição (que é a saída do MUX 3) é a mesma de um MUX 4×1 quando o par AB é usado como entrada para os seus *bits* seletores.

Pergunta: E se trocarmos a variável de entrada dos seletores na figura acima? Se colocarmos

A no seletor dos MUX 1 e MUX 2 e B na do MUX 3, ainda é possível realizarmos um MUX 4×1 no qual os bits seletores são alimentados pelo par AB ?

7.3.2 Demultiplexadores

Demultiplexadores (também conhecidos como distribuidores de dados ou DMUX) são circuitos combinacionais inversos aos multiplexadores, isto é, possuem apenas uma entrada que é transmitida a apenas uma das n saídas, determinada pelos *bits* seletores. De forma similar ao MUX, a saída para a qual a entrada é enviada é aquela cujo índice corresponde ao valor definido nos *bits* seletores. Se o demultiplexador possui n saídas, então são necessários k seletores, com $2^k \geq n$.

Uma implementação simples de DMUX consiste de n portas E. Cada porta E gera uma saída z_i, $i = 0, 1, 2, \ldots, n$, que corresponde à função dada por

$$z_i(D, s_{k-1}, \ldots, s_1, s_o) = D\, m_i$$

na qual D é a variável de entrada e m_i é o mintermo correspondente a i. Por exemplo, se $k = 2$ e $i = 1$, temos $z_1 = D\, \overline{s}_1\, s_o$ ($m_1 = \overline{s}_1\, s_o$).

7.3.3 Exemplos de utilização de MUX e DMUX

ULA Adição e subtração são operações realizadas por qualquer ULA. Quando consideramos a notação complemento de dois, vimos que o mesmo circuito somador pode ser usado tanto para realizar a operação de adição ($A + B$) quanto a de subtração ($A - B$). Caso a operação a ser executada seja a de subtração, então B deve ser complementado antes de alimentar o circuito. Ou seja, existe a necessidade de se chavear o tráfego de sinais no circuito de forma que ora B ou ora $\overline{B}$ seja enviado ao circuito somador, de acordo com o cálculo a ser efetuado. Este chaveamento pode ser realizado por meio de um MUX, que recebe nas entradas B e $\overline{B}$ e envia para a saída uma delas, e tendo como entrada no *bit* seletor um *bit* que indica a operação a ser executada.

Transmissão serial Suponha que temos dois subsistemas S e R, sendo que S possui n geradores cujos sinais gerados devem ser transmitidos aos respectivos n receptores em R. A transmissão dos n sinais poderia ser realizada par a par simultaneamente caso existisse um canal de comunicação direta entre os geradores de S e os respectivos receptores em R. No entanto, os sistemas S e R podem estar conectados por apenas um canal de transmissão, o que impossibilitaria a transmissão par a par simultânea. Nestes casos, uma possível solução é uma transmissão serial,

entre um par por vez, e ciclando-se sobre todos os pares. Para tanto, pode-se utilizar um multiplexador acoplado a S antes do canal de transmissão e, na outra ponta, um demultiplexador acoplado entre o canal de transmissão e o sistema R. Nesta solução, os seletores devem estar sincronizados, isto é, os seletores de ambos os sistemas devem estar com o mesmo valor. Se o valor dos seletores do MUX e do DMUX for i, o MUX selecionará o sinal do gerador i para ser transmitido, e na outra ponta, o DMUX irá direcionar o sinal recebido para o receptor i. Alterando-se o valor de i ciclicamente, tem-se a transmissão serial.

7.4 Decodificadores e codificadores

7.4.1 Decodificadores

Decodificadores são circuitos combinacionais com n entradas e 2^n saídas. As n entradas são interpretadas como um número em binário e portanto elas podem codificar os números de 0 a $2^n - 1$. Apenas uma única saída é ativada; justamente aquela cujo índice corresponde ao valor codificado na entrada. A figura 7.5 mostra um esquema genérico de um decodificador $n : 2^n$. As entradas $x_{n-1} \ldots x_1\, x_0$ são interpretadas como um número binário entre 0 e $2^n - 1$ e tem-se $z_i = 1 \Longleftrightarrow \sum_{k=0}^{n-1} x_k\, 2^k = i$.

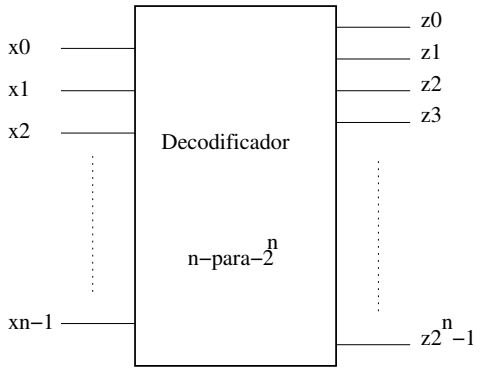


Figura 7.5: Esquema de um decodificador $n : 2^n$.

Exemplo: Seja o decodificador $2 : 4$ mostrado na figura 7.6. Neste caso, o *bit* menos significativo na entrada recebe A e o mais significativo recebe B. Temos $z_0 = \overline{B}\,\overline{A}$, $z_1 = \overline{B}\,A$, $z_2 = B\,\overline{A}$ e $z_3 = BA$. Para valores fixos na entrada, apenas um desses produtos canônicos tomará valor 1 (ou seja, apenas uma saída será ativada).

Conceitualmente, o circuito acima poderia ser estendido para realizar decodificadores $n : 2^n$, para um valor de n arbitrariamente grande. No entanto, na prática existem limitações tecnológicas

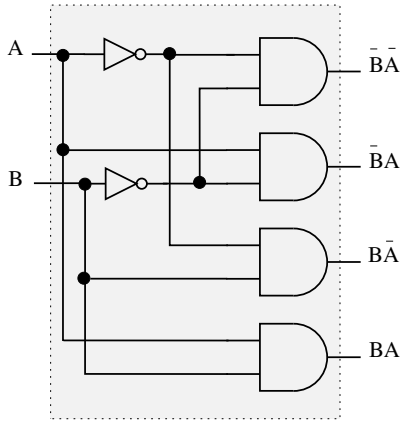


Figura 7.6: Circuito de um decodificador $2:4$.

(físicas) conhecidas como *fan-in* (número máximo de entradas possíveis em uma porta lógica) que restringem este valor n. Para valores grandes de n, decodificadores são realizados por circuitos multi-níveis, conforme veremos a seguir.

7.4.2 Realização multi-níveis de decodificadores*

Vimos que decodificadores possuem n entradas e 2^n saídas e que sua realização trivial utiliza 2^n portas E, com n entradas cada. Para contornar o problema de *fan-in* (número máximo de entradas possíveis em uma porta lógica), decodificadores com grande número de entradas podem ser realizados por circuitos com múltiplos níveis. A figura 7.7 mostra como pode ser realizado um decodificador $12:2^{12}$ em três níveis.

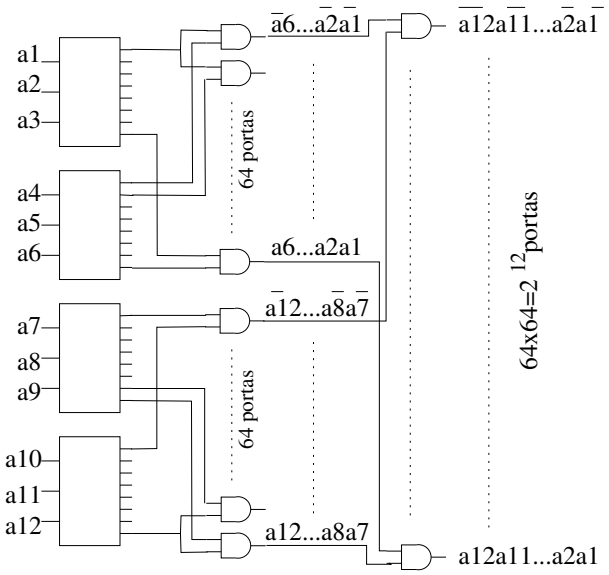


Figura 7.7: Realização três-níveis de um decodificador $12:2^{12}$.

No primeiro nível são usados 4 decodificadores $3:8$. No segundo nível, 64 portas E de duas entradas são usadas para combinar cada uma das 8 saídas do primeiro decodificador com cada uma das 8 saídas do segundo decodificador. A mesma coisa para as saídas do terceiro com as do quarto decodificador. Cada uma das saídas das primeiras 64 portas E do segundo nível são combinadas com cada uma das saídas das últimas 64 portas E no mesmo nível, resultando em um total de $64 \times 64 = 2^{12}$ portas E no terceiro nível. As saídas dessas 2^{12} portas E correspondem aos produtos canônicos de 12 variáveis.

A solução acima utiliza portas E com três entradas no primeiro nível e portas E com duas entradas nos demais níveis. Se o circuito fosse realizado em apenas um nível, as portas E teriam 12 entradas.

Em uma outra possível realização, poderíamos substituir as 128 portas E de duas entradas no segundo nível acima por 2^{12} portas E de quatro entradas e eliminar as portas do terceiro nível. Isto aparentemente reduziria o número total de portas, mas uma vez que 2^{12} domina de longe 128 e uma vez que as portas agora teriam quatro entradas em vez de duas, não se pode dizer que há economia no custo total.

Um outro problema devido às limitações tecnológicas é o conhecido por *fan-out* (número máximo de portas que podem ser alimentadas por uma saída de uma porta lógica). No caso da realização três-níveis do decodificador $12:2^{12}$ visto acima, as saídas das portas no segundo nível alimentam 64 portas no terceiro nível.

Para contornar o *fan-out*, uma possível solução são as realizações em estruturas de árvore. A figura 7.8 mostra a realização de um decodificador $3:8$ em uma estrutura de árvore. Em vez de termos todas as variáveis alimentando portas no primeiro nível, temos variáveis que alimentam portas nos outros níveis. Usando este esquema, pode-se reduzir o número de portas no próximo nível que precisam ser alimentadas pela saída de uma porta no nível anterior. Mesmo assim, o problema de *fan-out* não é totalmente eliminado pois as variáveis introduzidas nos níveis posteriores do circuito precisam alimentar muitas portas. No entanto, é mais fácil controlar o sinal de algumas poucas entradas (variáveis) para que eles sejam capazes de alimentar um maior número de portas do que fazer o mesmo com as saídas das portas lógicas. Esta solução possui um maior número de níveis e um maior número de portas lógicas do que o esquema mostrado na figura 7.7, mas para decodificadores de muitas entradas pode ser a única solução.

Na prática, as realizações de decodificadores para um número grande de entradas é baseada em uma combinação das estruturas da figura 7.7 e da figura 7.8.

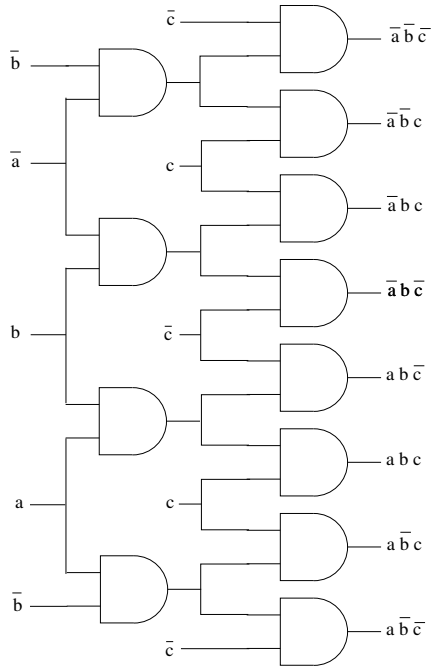


Figura 7.8: Decodificador em estrutura de árvore.

7.4.3 Codificadores

Codificadores são circuitos combinacionais que são o inverso de decodificadores. Um codificador de n entradas deve possuir s saídas satisfazendo

$$2^s \geq n \quad \text{ou } s \geq \log_2 n$$

Usualmente deve-se ter apenas uma entrada ativa e a saída será o código binário correspondente à entrada. Isto é, se a i-ésima entrada estiver ativa, a saída será o código binário de i. A figura 7.9 mostra o esquema de um codificador de n entradas. Codificadores com s saídas

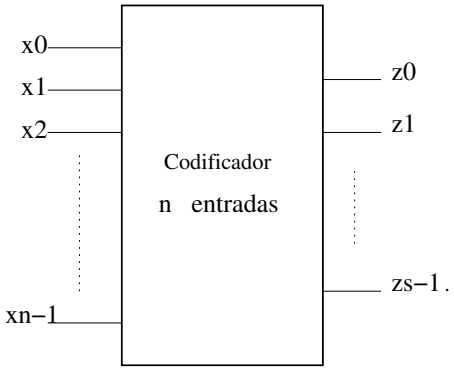


Figura 7.9: Esquema de um codificador.

podem ser realizados com portas OU, uma para cada saída.

Embora usualmente os codificadores sejam definidos como um circuito no qual apenas uma entrada encontra-se ativa, é possível termos codificadores com propósitos específicos que, por exemplo, para certos tipos de combinação de entradas gera um dado código de saída e para outras combinações de entradas gera outro código de saída.

7.4.4 Exemplos de utilização de codificadores e decodificadores

Memórias ROM: Um exemplo de uso de decodificadores são as memórias do tipo ROM (*Read-Only Memory*). Um diagrama que ilustra uma possível realização de uma ROM com 4 posições e cada posição com capacidade para armazenar 4 *bits* é mostrada na figura 7.10. As posições da memória podem ser identificadas por "endereços" númericos; no caso, 0, 1, 2, e 3. Por exemplo, a posição 0 está destacada no diagrama, e nessa posição está armazenado o valor 1110 (na prática, quando há uma bolinha preta, significa que as linhas horizontal e vertical estão conectadas e portanto haverá condução de sinal elétrico). O endereço dessas 4 posições pode ser codificado em dois *bits* $x_1\,x_0$. Por exemplo, se $x_1\,x_0 = 00$, o decodificador irá ativar a saída 0 (a primeira de cima para baixo), o que fará com que o sinal trafegue na linha horizontal e em direção a z_3, z_2 e z_1 (mas não z_0). Teremos, neste caso, $z_3\,z_2\,z_1\,z_0 = 1110$ que é exatamente o "valor" armazenado na posição 0 da ROM.

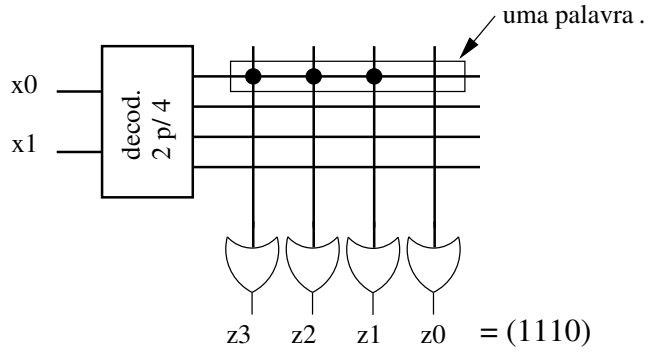


Figura 7.10: Esquema de uma memória ROM com 4 posições e palavras de 4 bits.

De forma geral, a cada endereço $(x_1\,x_0)$ fonecido como entrada do decodificador, apenas uma saída do decodificador ficará ativa. A palavra na posição de memória endereçada (isto é, o dado armazenado na linha de saída ativada) será transmitida para a saída $(z_3\,z_2\,z_1\,z_0)$ (note que o esquema da figura está simplificado; a rigor, cada porta OU possui exatamente 4 entradas que poderão ou não estar conectadas a cada uma das linhas de saída do decodificador).

A estrutura das memórias ROM é semelhante aos PLAs (i.e., um plano de portas E e um plano de portas OU). Na ROM, as portas E estão no decodificador, e há necessariamente 2^n portas E (todos os produtos são canônicos), enquanto num PLA o número de portas E pode variar e elas são programáveis. Na ROM apenas o plano OU pode ser programável. Se o plano OU tem

conexões fixas, trata-se de uma ROM. Se o plano OU puder ser programado, então trata-se de uma PROM (*Programmable ROM*), e se o plano OU puder ser reprogramado trata-se de uma EPROM (*Erasable Programmable ROM*).

Teclado: Decodificadores e codificadores podem também ser usados, por exemplo, em teclados. Suponha por exemplo que um teclado simplificado possui 70 teclas. O computador deve ser capaz de saber qual tecla foi pressionado pelo usuário, para gerar o código (por exemplo ASCII) do caractere correspondente à tecla pressionada para ser enviado para o processador. Ao invés de usarmos 70 linhas conectando cada uma das teclas a um gerador de código ASCII, podemos ter um esquema como o ilustrado na figura 7.11, que aproveita o arranjo "bidimensional" (linhas × colunas) das teclas.

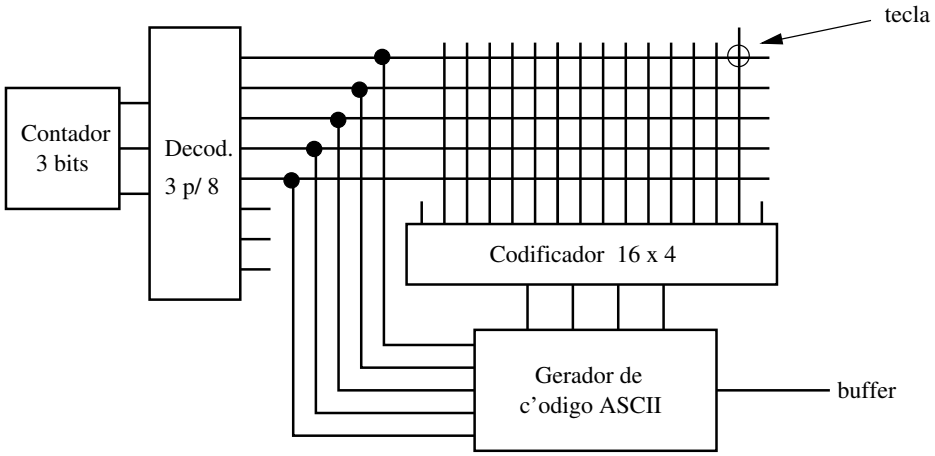


Figura 7.11: Esquema de um teclado. O decodificador identifica a linha e o codificador a coluna da tecla pressionada. É esperado que a cada tecla pressionada, o contador na entrada do decodificador execute um ciclo e, portanto, ative cada uma das 8 saídas do decodificador uma única vez.

Vamos examinar os componentes desse esquema. Ao lado esquerdo está um contador de 3 *bits*. Estudaremos contadores mais adiante e aqui nos limitaremos apenas a descrever seu funcionamento. O contador no diagrama simplesmente gera sucessivamente os números de 0 a 7 e de forma cíclica (ao atingir 7, volta para 0 e continua). Esses números gerados pelo contador alimentam o decodificador. Isto significa que as saídas do decodificador serão ativadas sucessivamente de 0 a 7, uma por vez, e de forma cíclica também. Vamos supor também que o contador completa um ciclo (e apenas um ciclo) durante o período de tempo no qual uma tecla está pressionada. No diagrama, é mostrado em destaque a tecla no canto superior direito. Quando essa tecla estiver pressionada e a saída 0 do decodificador for ativada, o sinal será transmitido pela linha vertical até o codificador. O codificador é usado para detectar em qual das colunas encontra-se a tecla pressionada, gerando o código binário correspondente ao número da coluna (como temos 14 colunas, 4 *bits* são suficientes na saída do codificador). O gerador de código ASCII recebe a indicação de qual é a linha (via entrada lateral) e a coluna (via saída do co-

dificador) da tecla pressionada. Com essas informações, ele sabe qual tecla foi pressionada e poderá então gerar o código correspondente à tecla. Um teclado de verdade é mais complexo do que isto, especialmente com respeito a aspectos relacionados com sincronização (em situações de pressão continuada de uma tecla, como deve ser definido quantas vezes a tecla foi pressionada?, como lidar com combinações tais como CTRL+ALT?, etc).

7.5 Realização de funções arbitrárias

Da mesma forma que funções lógicas representadas por expressões na forma soma de produtos ou produto de somas podem ser facilmente implementados em PLAs, circuitos disponíveis como MUX ou decodificadores podem ser usados para a realização de funções quaisquer.

7.5.1 Realização de funções com MUX

Para realizar funções por meio de MUX, devemos escolher as variáveis que funcionarão como seletores. Baseado nisso, determina-se quais subexpressões devem ser enviadas às entradas de dados do MUX, para que tenhamos a realização da função.

Exemplo: Suponha que desejamos realizar a função $f(a,b,c) = \overline{a}\,\overline{b} + a\,c$ usando um MUX 4×1 e que as variáveis a e b serão utilizadas nos seletores. Para determinar quais subexpressões devem ser enviadas às entradas de dados do MUX, podemos expandir a expressão da função de forma que os literais correspondentes às variáveis a e b apareçam em todos os produtos da expressão resultante. No caso da função dada, temos:

$$\begin{aligned} f(a,b,c) &= \overline{a}\,\overline{b} + a\,c \\ &= \overline{a}\,\overline{b} + a\,(b+\overline{b})\,c \\ &= \overline{a}\,\overline{b} + a\,b\,c + a\,\overline{b}\,c \\ &= (\overline{a}\,\overline{b})\,1 + (\overline{a}\,b)\,0 + (a\,\overline{b})\,c + (a\,b)\,c \end{aligned}$$

Na última linha os produtos envolvendo as variáveis a e b foram destacadas para facilitar o entendimento. A realização em um MUX 4×1 pode ser derivada diretamente dessa última expressão: fazendo-se $s_1 = a$, $s_0 = b$, basta fazermos $D_0 = 1$, $D_1 = 0$, $D_2 = c$ e $D_3 = c$.

7.5.2 Realização multi-níveis de funções com MUX

Considere por exemplo a função $f(a,b,c,d) = \sum m(2,5,8,9,11,12,14,15)$. Se procedermos como feito acima, mas desta vez usando um MUX 8×1, uma possível realização (usando $a\,b\,c$

nos seletores) é mostrada na figura 7.12.

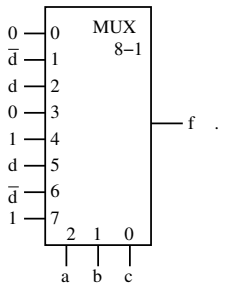


Figura 7.12: Uma realização de $f(a,b,c,d) = \sum m(2,5,8,9,11,12,14,15)$ com um MUX 8×1.

Porém, podemos também utilizar MUX arranjados em múltiplos níveis. Por exemplo, para a mesma função f acima, duas possíveis realizações usando MUX 4×1 e MUX 2×1 são mostradas na figura 7.13.

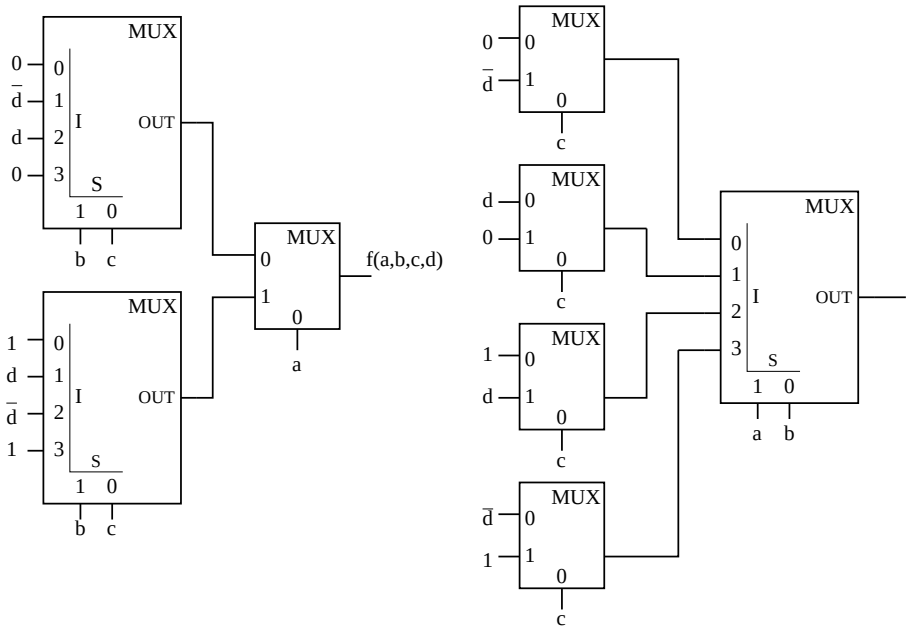


Figura 7.13: Realização de $f(a,b,c,d) = \sum m(2,5,8,9,11,12,14,15)$ com (a) dois MUX 4×1 no primeiro nível e um MUX 2×1 no segundo nível e (b) quatro MUX 2×1 no primeiro nível e um MUX 4×1 no segundo nível.

Estas estruturas podem ser obtidas a partir da análise dos mintermos arranjados em forma tabular, conforme mostrado a seguir. A tabela da esquerda considera o uso das entradas b e c como seletores dos MUX 4×1 no primeiro nível e o uso da variável a como seletor do MUX 2×1 do segundo nível (diagrama à esquerda na figura 7.13). A tabela da direita faz o análogo para a implementação com quatro MUX 2×1 no primeiro nível e um MUX 4×1 no segundo nível (diagrama à direita na figura 7.13).

$abcd$	a	bc	d	input
0010	0	01	0	$\overline{d}$
0101	0	10	1	d
1000	1	00	0	$\overline{d}+d=1$
1001	1	00	1	
1011	1	01	1	d
1100	1	10	0	$\overline{d}$
1110	1	11	0	$\overline{d}+d=1$
1111	1	11	1	

$abcd$	ab	c	d	input
0010	00	1	0	$\overline{d}$
0101	01	0	1	d
1000	10	0	0	$\overline{d}+d=1$
1001	10	0	1	
1011	10	1	1	d
1100	11	0	0	$\overline{d}$
1110	11	1	0	$\overline{d}+d=1$
1111	11	1	1	

Outra abordagem para determinar a estrutura hierárquica dos MUXes na realização de funções com múltiplos níveis de MUXes é baseada na aplicação sucessiva da expansão de Shannon. Um exemplo simples dessa abordagem foi mostrado na seção 7.5.1. Aqui mostramos que dependendo da sequência de variáveis em torno das quais a expansão é aplicada, pode-se chegar a diferentes estruturas multi-níveis.

Conforme já vimos, o teorema de **Expansão de Shannon** afirma que qualquer função f de n variáveis pode ser escrita em termos de funções de $n-1$ variáveis da seguinte forma:

$$f(x_1,\ldots,x_k,\ldots,x_n) = \overline{x}_i f(x_1,\ldots,0,\ldots,x_n) + x_i f(x_1,\ldots,1,\ldots,x_n)$$

As funções $f(x_1,\ldots,0,\ldots,x_n)$ e $f(x_1,\ldots,1,\ldots,x_n)$ são funções de $n-1$ variáveis. O teorema pode ser aplicado recursivamente sobre essas duas funções.

Exemplo: Consideremos novamente a função $f(a,b,c,d) = \sum m(2,5,8,9,11,12,14,15)$. Vamos mostrar agora como reproduzir a realização mostrada no lado direito da figura 7.13. Como no MUX do segundo nível são usadas as variáveis a e b, devemos colocar os produtos que envolvem essas duas variáveis em "evidência". Em seguida, deve-se fazer o mesmo com a variável c.

$$\begin{aligned}
f &= \overline{a}\,\overline{b}\,c\,\overline{d} + \overline{a}\,b\,\overline{c}\,d + a\,\overline{b}\,\overline{c}\,\overline{d} + a\,\overline{b}\,\overline{c}\,d + a\,\overline{b}\,c\,d + a\,b\,\overline{c}\,\overline{d} + a\,b\,c\,\overline{d} + a\,b\,c\,d \\
&= \overline{a}\,\overline{b}\,c\,\overline{d} + \overline{a}\,b\,\overline{c}\,d + a\,\overline{b}\,\overline{c}\,\overline{d} + a\,b\,\overline{c}\,\overline{d} + a\,\overline{b}\,\overline{c}\,d + a\,\overline{b}\,c\,d + a\,b\,c\,\overline{d} + a\,b\,c\,d \quad \text{(rearranjo)} \\
&= \overline{a}\,\overline{b}\,c\,\overline{d} + \overline{a}\,b\,\overline{c}\,d + a\,\overline{c}\,\overline{d} + a\,\overline{b}\,d + a\,b\,c \quad \text{(algumas simplificações)} \\
&= \overline{a}\,(\overline{b}\,c\,\overline{d} + b\,\overline{c}\,d) + a(\overline{c}\,\overline{d} + \overline{b}\,d + b\,c) \quad \text{(expansão em torno de } a) \\
&= \overline{a}\,(\overline{b}(c\,\overline{d}) + b(\overline{c}\,d)) + a\,(\overline{b}(\overline{c}\,\overline{d} + d) + b\,(\overline{c}\,\overline{d} + c)) \quad \text{(expansão em torno de } b) \\
&= \overline{a}\,\overline{b}(c\,\overline{d}) + \overline{a}\,b(\overline{c}\,d) + a\,\overline{b}(\overline{c}\,\overline{d} + d) + a\,b\,(\overline{c}\,\overline{d} + c) \quad \text{(distribuição com respeito a } a) \\
&= \overline{a}\,\overline{b}(\overline{c}(0) + c\,(\overline{d})) + \overline{a}\,b(\overline{c}\,(d) + c(0)) + a\,\overline{b}(\overline{c}\,(1) + c(d)) + a\,b\,(\overline{c}\,(\overline{d}) + c(1)) \quad \text{(expansão em torno de } c)
\end{aligned}$$

A última expressão acima corresponde à realização já mostrada anteriormente (na prática, podíamos ter escrito a penúltima linha diretamente a partir da forma canônica inicial).

Nas equações acima, logo após a expansão em torno de b, poderíamos ter prosseguido da seguinte forma:

$$\begin{aligned} f &= \overline{a}\,(\overline{b}(c\,\overline{d}) + b(\overline{c}\,d)) + a\,(\overline{b}(\overline{c}\,\overline{d} + d) + b\,(\overline{c}\,\overline{d} + c)) \quad \text{(expansão em torno de } b\text{)} \\ &= \overline{a}\,(\overline{b}\,c\,(\overline{d}) + \overline{b}\,\overline{c}(0) + b\,\overline{c}\,(d) + b\,c\,(0)) + a\,(\overline{b}\overline{c}\,(1) + \overline{b}\,c(d) + b\,\overline{c}\,(\overline{d}) + b\,c\,(1))) \end{aligned}$$

Esta última expressão corresponde à realização mostrada no lado esquerdo da figura 7.13.

7.5.3 Realização de funções com decodificadores

Uma vez que um decodificador $n : 2^n$ realiza todos os produtos canônicos de n variáveis, qualquer função com n variáveis pode ser realizada com um decodificador $n : 2^n$ e uma porta OU (com um número de entradas maior ou igual ao número de 1s da função) ou uma porta NÃO-OU (com um número de entradas maior ou igual ao número de 0s da função).

O custo da realização de uma função com decodifcadores é, em termos de portas lógicas, (muito provavelmente) maior que o da realização SOP minimal. No entanto, a simplicidade de projeto torna-o atraente. Além disso, quando múltiplas funções precisam ser realizadas, menor tende a ser a diferença dos custos entre a realização SOP minimal e a realização com decodificadores.

Exemplo: A função $f(a,b,c) = \sum m(0,1,4,6,7) = \prod M(2,3,5)$ pode ser realizada usando decodificadores conforme ilustrado na figura 7.14. No caso da realização com porta NÃO-OU, observe que $f(a,b,c) = \prod M(2,3,5) = \overline{M_2 \cdot M_3 \cdot M_5} = \overline{\overline{M_2} + \overline{M_3} + \overline{M_5}} = \overline{m_2 + m_3 + m_5}$.

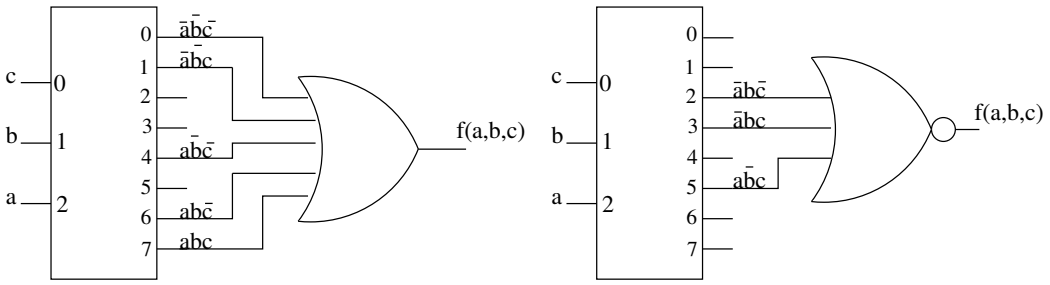


Figura 7.14: Realização de $f(a,b,c) = \sum m(0,1,4,6,7)$ com decodificador $3:8$ e uma porta OU (esquerda) ou uma porta NÃO-OU (direita).

Exercícios

1. Explique o que é um multiplexador n:1 (n entradas). Quantos *bits* seletores são necessários ? Desenhe o circuito de um multiplexador de 4 entradas (1 saída).

2. Explique o que é um demultiplexador 1:n (n saídas). Quantos *bits* seletores são necessários ? Desenhe o circuito de um demultiplexador de 4 saídas (1 entrada).

3. Explique o que é um decodificador de n bits (entrada corresponde aos *bits* de um número binário de n *bits*). Quantas saídas devemos ter ? Desenhe o circuito de um decodificador de 3 bits.

4. Explique o que é um codificador de n entradas. Quantas saídas devemos ter e o que elas representam ? Devemos supor alguma condição em relação às entradas ? Desenhe o circuito de um codificador de 4 entradas.

5. Como o decodificador é utilizado para acesso a uma posição específica da memória do computador?

6. Descreva um exemplo concreto da utilidade de um multiplexador.

7. Descreva um exemplo concreto da utilidade de um demultiplexador.

8. Como podemos implementar um multiplexador 4-1, usando apenas multiplexadores 2-1? Quantos destes são necessários ?

9. Mostre como realizar a função $f(a,b) = \sum m(0,2,3)$ com um decodificador 2-4 mais uma porta OU.

10. Escreva a realização da função $f(a,b,c) = \overline{a}\,\overline{b} + a\,c$ usando um MUX 8×1, com as variáveis a, b e c como seletores.

11. Escreva a realização da função $f(a,b,c,d) = \sum m(0,1,3,6,7,8,11,12,14)$ usando um MUX 8×1, com as variáveis a, b e c como seletores.

12. Escreva a realização da função $f(a,b,c,d) = \sum m(0,1,3,6,7,8,11,12,14)$ usando um MUX 4×1, com as variáveis a e b (neste caso, as entradas possivelmente dependerão das variáveis c e d e serão necessárias portas adicionais para a realização de f).

13. Seja $f(a,b,c,d) = \sum m(0,3,5,7,11,12,13,15)$. Mostre como realizar a função f usando:

    (a) um multiplexador $16:1$ e eventualmente um mínimo de portas lógicas.

(b) um multiplexador $8:1$ (use a, b, c como entrada para os seletores) e eventualmente um mínimo de portas lógicas.

(c) um multiplexador $4:1$ (use a, b como entrada para os seletores) e eventualmente um mínimo de portas lógicas.

Use a como o bit mais significativo e b como o menos significativo dos *bits* seletores e, analogamente, o inverso (i.e., b como o bit mais significativo e a como o menos significativo). Houve diferença na quantidade de portas lógicas AND e/ou OR adicionais necessárias?

(d) um decodificador 4-para-16 e uma porta OU.

(e) um decodificador 4-para-16 e uma porta NÃO-OU.

Capítulo 8

Lógica sequencial – memória

Última atualização em 28 de junho de 2021

Um programa de computador é formado por uma sequência de instruções que podem ser executadas pelo processador. Em geral, enquanto o programa encontra-se em execução, essa sequência de instruções fica armazenada em uma parte do computador denominada memória. A execução dessas instruções pode gerar dados que deverão ser armazenados para utilização posterior, por outras instruções na sequência. Em muitas situações é conveniente que esses dados fiquem também armazenados na memória do computador, para que o posterior acesso a eles seja eficiente. O processador precisa também manter algumas informações que são úteis para controlar a execução dessas instruções. Portanto, memória é um importante componente dos computadores.

Uma característica importante de dispositivos do tipo memória é a capacidade de manter fixo os valores armazenados, exceto quando são explicitamente instruídos a "trocarem" de valor. Dispositivos do tipo memória podem ser modelados por circuitos lógicos. Eles formam a memória RAM onde são armazenados dados e instruções, mas também são usados no processador. No processador, memória são os elementos que distinguem os circuitos denominados sequenciais daqueles denominados combinacionais.

Neste capítulo fazemos inicialmente um contraste entre circuitos combinacionais e sequenciais. Em seguida, uma vez que aos circuitos sequenciais está associada a noção de tempo, faremos uma breve introdução às ondas digitais, que permitirá termos uma noção de como uma sequência de *bits* é representada fisicamente. Em seguida descrevemos como as unidades de memória são construídas e funcionam (sob o ponto de vista lógico). As unidades de memória, capazes de armazenar um *bit*, são denominadas de *flip-flops*. Exemplos de circuitos que utilizam *flip-flops*, tais como contadores e registradores serão apresnetados nos próximos capítulos.

8.1 Combinacional × sequencial

Circuitos combinacionais são aqueles que possuem um certo número n de entradas $(x_1, x_2, \ldots, x_n)$ e um número m de saídas $(z_1, z_2, \ldots, z_m)$, no qual cada saída depende apenas das entradas. Isto é, cada uma das saídas z_i pode ser expressa por uma função booleana $f_i : \{0,1\}^n \to \{0,1\}$. O diagrama da figura 8.1 mostra a estrutura geral de circuitos combinacionais.

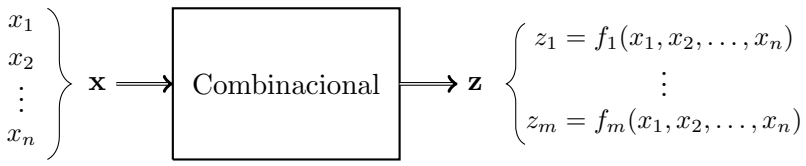


Figura 8.1: Estrutura de um circuito combinacional: n entradas denotadas $x_1, \ldots, x_n$ e m saídas denotadas $z_1, \ldots, z_m$, com $z_i = f_i(x_1, \ldots, x_n)$.

Exemplo 8.1.1. *Um típico exemplo de um circuito combinacional é o circuito somador. Ele recebe na entrada a representação binária de dois números e devolve na saída a representação binária da soma dos dois números da entrada. No capítulo anterior vimos outros exemplos de circuitos combinacionais (multiplexador, codificador, etc).*

Os circuitos sequencias, por sua vez, também possuem entradas $x_1, x_2, \ldots, x_n$ e uma ou mais saídas $z_1, z_2, \ldots, z_m$. A diferença fundamental em relação aos circuitos combinacionais é o fato de eles possuírem uma parte que corresponde à memória do circuito. Os valores armazenados na memória num dado instante de tempo representam o que é chamado de "estado" do circuito. No contexto de circuitos sequenciais, está implícita a noção de tempo, de iteração. A cada nova iteração, há uma transição de estado de $\mathbf{y}$ para $\mathbf{y}^*$. O estado atual, denotado $\mathbf{y}$ é substituído pelo próximo estado, denotado $\mathbf{y}^*$.

A figura 8.2 mostra um diagrama com a estrutura geral de circuitos sequenciais. Tipicamente há uma parte combinacional e uma parte que corresponde à memória do circuito. O estado atual é representado por r variáveis de estado $y_1, \ldots, y_r$. Note que tanto as saídas $\mathbf{z}$ como o próximo estado $\mathbf{y}^*$ dependem das entradas $\mathbf{x}$ e do estado atual $\mathbf{y}$.

Além de funções que definem as saídas do circuito, temos também as funções que definem o "próximo estado" (representado pelas variáveis $y_1^*, y_2^*, \ldots, y_r^*$), ou seja,

$$z_i = g_i(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_r), \quad i = 1, 2, \ldots, m$$

e

$$y_i^* = h_i(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_r), \quad i = 1, 2, \ldots, r$$

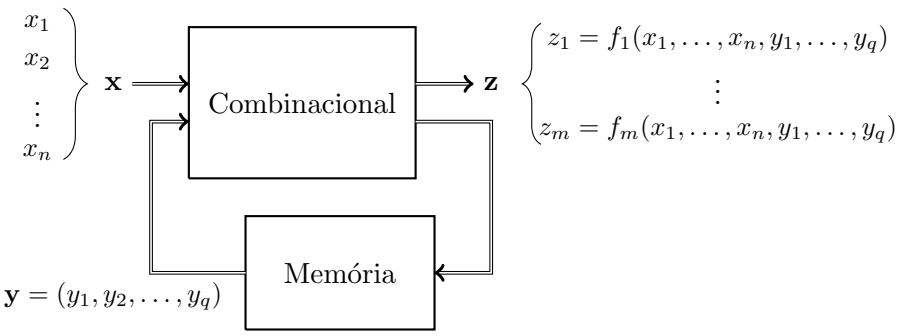


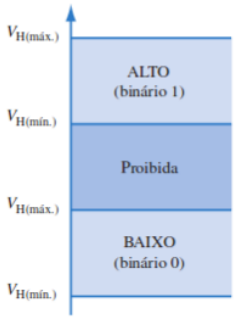
Figura 8.2: Estrutura de um circuito sequencial: n entradas denotadas $x_1, \ldots, x_n$, m saídas denotadas $z_1, \ldots, z_m$ e r estados denotados $y_1, \ldots, y_1$, com $z_i = f_i(x_1, \ldots, x_n, y_1, \ldots, y_r)$.

Exemplo 8.1.2. *Os sinais dos semáforos alternam de verde para amarelo e em seguida para vermelho. Depois voltam para verde e este ciclo se repete continuamente. Este é um dispositivo sequencial; isto é, o próximo estado depende do estado atual. Da mesma forma, um elevador é um dispositivo sequencial no qual as entradas são as chamadas feitas (pelas pessoas em cada andar e no interior do elevador) e o estado atual é o andar no qual o elevador se encontra mais o sentido em que ele está se movendo (parado, subindo, descendo). A posição do elevador nos próximos instantes dependerá das entradas atuais e do estado atual dele. Estes exemplos ilustram a ideia de "estado atual" e "próximo estado".*

8.2 Ondas digitais

Em vez de pensar entradas de um circuito como um valor estático, no contexto de circuitos sequenciais é conveniente pensarmos entradas como um sinal que muda de valor ao longo do tempo. As ondas digitais referem-se a esses sinais. Os conceitos a seguir visam fornecer uma visão geral sobre a representação digital (as figuras que os acompanham foram todas extraídas do livro do Floyd [Floyd, 2007]. Para maiores detalhes veja o capítulo 1 desse livro).

Interpretação do sinal elétrico como *bit*

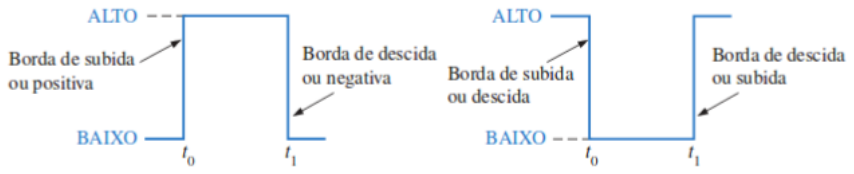


◀ FIGURA 1–5
Faixas de níveis lógicos de tensão para um circuito digital.

Onda digital (idealizado)

Formas de Onda Digitais

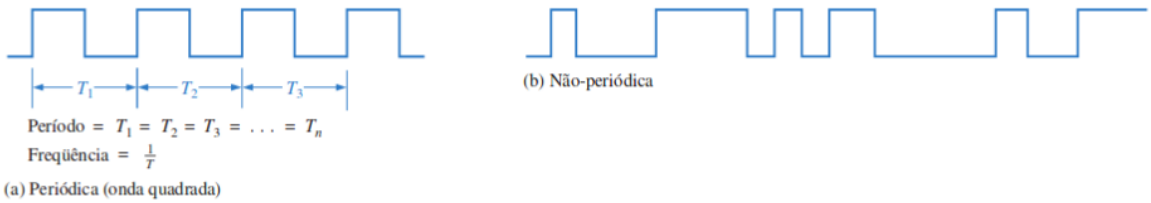
Formas de onda digitais consistem em níveis de tensão que comutam entre os níveis, ou estados, lógicos ALTO e BAIXO. A Figura 1–6(a) mostra que um único **pulso** positivo é gerado quando a tensão (ou corrente) passa do nível BAIXO normal para o nível ALTO e em seguida retorna para o nível BAIXO. O pulso negativo, visto na Figura 1–6(b), é gerado quando a tensão passa do nível ALTO normal para o nível BAIXO e retorna para o nível ALTO. Uma forma de onda digital é constituída de uma série de pulsos.



(a) Pulso positivo (b) Pulso negativo

Na prática pode haver oscilações

Sinal periódico e não periódico



(a) Periódica (onda quadrada)

(b) Não-periódica

Sinal (onda) sincronizado com temporizador pode ser interpretado como uma sequência de bits:

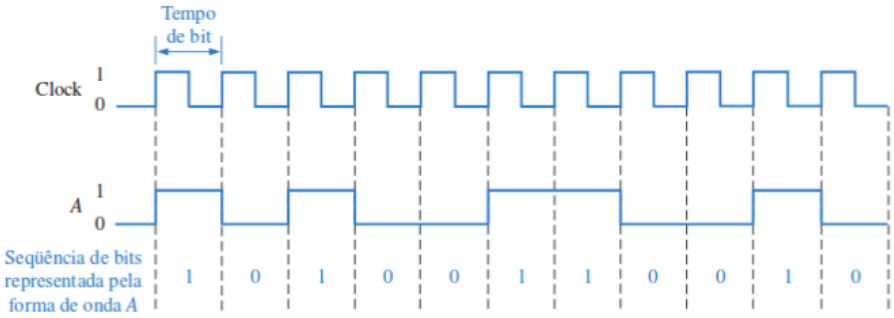
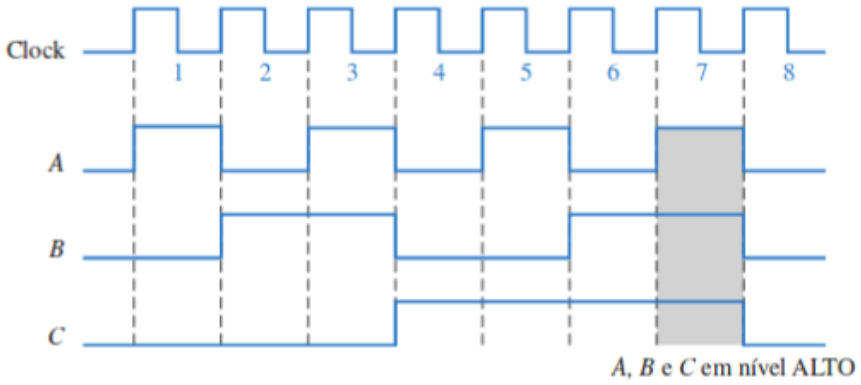


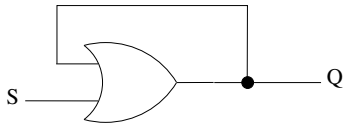
Diagrama temporal: usado para mostrar sinais de forma sicronizada ao temporizador (ou *clock*)



8.3 Latches

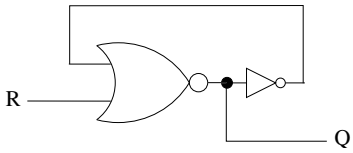
Para entender como construir um circuito com a capacidade de armazenar e manter um certo valor, vamos examinar inicialmente dois exemplos. Eles mostram como a retroalimentação pode ser usada para "congelar" o estado de um circuito. Os circuitos do tipo memória são construídos combinando-se as ideias desses dois exemplos.

Exemplo 8.3.1. *Considere uma porta OU com ambas as entradas inicialmente em* 0. *Nesta situação, a saída é também* 0. *Suponha que de alguma forma conectamos a saída em uma das entradas (ver figura abaixo). O circuito mantém-se em estado estável. Em seguida, suponha que mudamos o valor da outra entrada (S) para* 1. *O que acontece com a saída Q? O que acontece se, em seguida, colocamos o valor de S de volta para* 0*?*



Pode-se perceber que quando o valor da entrada S é alterado para 1, a saída do circuito passa a ser 1. Alterações posteriores na entrada S do circuito não mais afetam a saída. Note também que neste circuito simples o estado e a saída são os mesmos (i.e., $z = Q$).

Exemplo 8.3.2. *Considere uma porta NÃO-OU cuja saída está conectada a um inversor. Este circuito é uma outra forma de representar a função lógica OU. Suponha que inicialmente ambas as entradas da porta NÃO-OU estão em* 0*. Nesta situação, a saída Q da porta NÃO-OU é 1 e a saída do inversor é* 0*. Suponha que a saída do inversor é conectada em uma das entradas da porta NÃO-OU (ver figura abaixo). O circuito mantém-se em estado estável. Suponha que em seguida mudamos o valor da outra entrada (R) da porta NÃO-OU para* 1*. O que acontece com a saída Q? O que acontece se, em seguida, colocamos o valor de R de volta para* 0*?*



De forma similar ao circuito do exemplo anterior, neste caso, quando o valor da entrada R é alterado para 1, o estado muda para 0 e nenhuma alteração posterior no valor de R irá alterar o estado.

Os circuitos acima são chamados de *set latch* e *reset latch*, respectivamente. No primeiro, após o estado Q passar para 1 (set), ele não pode mais ser alterado. No segundo, após Q passar para 0, não mais pode ser alterado. Portanto, esses são dispositivos que armazenam um certo valor lógico, que representa o estado y. O estado é denotado por Q nas figuras uma vez que essa é a notação mais comumente encontrada nas referências bibliográficas. Pelo fato de não podermos mudar o estado desses dispositivos mais de uma vez, eles tem possibilidade de uso muito limitado. Podemos, porém, usar idéia similar para construir dispositivos que permitem a alteração de estado múltiplas vezes. Tais circuitos são conhecidos por *set-reset latches* ou *set-reset flip-flops* e descritos a seguir.

8.4 Latches SR

Na figura 8.3, à esquerda encontra-se o circuito de um *latch* SR que usa portas NÃO-OU (NOR), e à direita o seu comportamento (saídas Q e $\overline{Q}$) em função das entradas (S e R). O nome SR

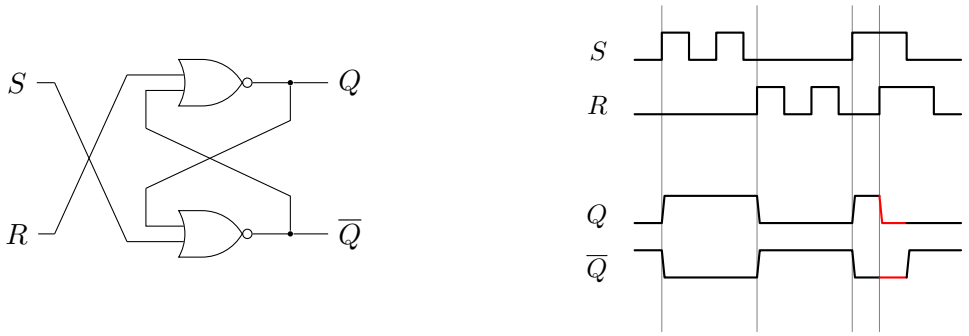


Figura 8.3: Operação de um *latch* SR (S=sinal set; R=sinal reset; Q=saída (estado)), com porta NOR.

vem de *set-reset*. Alguns autores usam a sigla RS em vez de SR.

O diagrama temporal à direita mostra como o estado muda em função de mudanças nas entradas S e/ou R. Inicialmente, ambas as entradas, S e R estão em 0, e a saída Q também está em 0 (consequentemente, a saída $\overline{Q}$ está em 1). Esse estado é consistente. Então temos a seguinte sequência de eventos:

- o sinal S vai a 1 e em consequência disso $\overline{Q}$ vai a 0 e Q vai a 1.
- o sinal S vai a 0, mas isso não afeta o estado.
- o sinal S vai novamente a 1 e em seguida a 0, mas isso também não afeta o estado.
- o sinal R vai a 1 e em consequência disso Q vai a 0 e $\overline{Q}$ vai a 1.
- o sinal R vai a 0 e em seguida a 1 e logo em seguida a 0 novamente, mas essas variações não afetam o estado.
- o sinal S vai a 1 e em consequência disso $\overline{Q}$ vai a 0 e Q vai a 1.
- o sinal R vai a 1 e em consequência disso Q vai a 0, mas $\overline{Q}$ se matém em 0 (trecho destacado em vermelho). Este é um estado inválido, pois é esperado que $\overline{Q}$ seja sempre o complementar de Q.

Podemos verificar que quando a entrada S vai para 1, o estado Q vai para 1 e quando a entrada R vai a 1, o estado Q vai a 0, independentemente do estado atual. Portanto $S = 1$ indica a operação *set* e $R = 1$ indica a operação *reset*. Quando ambos estão em 0, nada acontece. Quando ambos vão a 1, i.e. $S = R = 1$, temos uma situação que não é previsível. Primeiro, se tivermos ambas as entradas em 1, teremos $Q = \overline{Q} = 0$, violando a condição de que uma saída é o complemento da outra. Segundo, ao supormos que $Q = \overline{Q} = 0$ é aceitável, pode ocorrer uma

situação em que ambas as entradas S e R estão em 1 e passam simultaneamente para 0. Em tal situação, a saída das portas NÃO-OU passa a 1 e, em seguida, se as duas portas NÃO-OU funcionarem de forma exatamente iguais, o estado voltará para 0, o que faz com que em seguida passe para 1 e depois novamente para 0 e assim por diante. Isso levaria o estado do circuito a oscilar (ou seja, a saída não se estabilizaria). Se as duas portas não funcionarem de forma exatamente iguais, aquela que responde primeiro ditará o comportamento do circuito. Como na prática é razoável supormos que uma das portas responderá antes da outra e como numa realização física não se tem controle de qual porta responde primeiro, o comportamento será imprevisível.

Pelas razões descritas acima, no *latch SR* a entrada $R = S = 1$ é proibida. O comportamento do *latch* SR pode ser descrito pela seguinte tabela-verdade:

S	R	Q	Q^*	
0	0	0	0	Nenhuma mudança
0	0	1	1	
0	1	0	0	*reset*
0	1	1	0	
1	0	0	1	*set*
1	0	1	1	
1	1	0	?	proibido
1	1	1	?	

Para escrever a expressão que representa a transição de estados desse circuito, considere as seguintes notações:

- S denota o valor do sinal que alimenta a entrada S
- R denota o valor do sinal que alimenta a entrada R
- Q denota o estado atual (e também saída)
- Q^* denota o próximo estado.

A situação $S = R = 1$ é proibida. Portanto podemos tratá-la como *don't care* e então obtemos a expressão que descreve Q^*:

$$Q^* = S + Q\,\overline{R}$$

Essa expressão pode ser facilmente obtida desenhando-se o mapa de Karnaugh correspondente à tabela-verdade dada acima.

O *latch* SR pode também ser realizado com portas NÃO-E (NAND), como mostrado na figura 8.4.

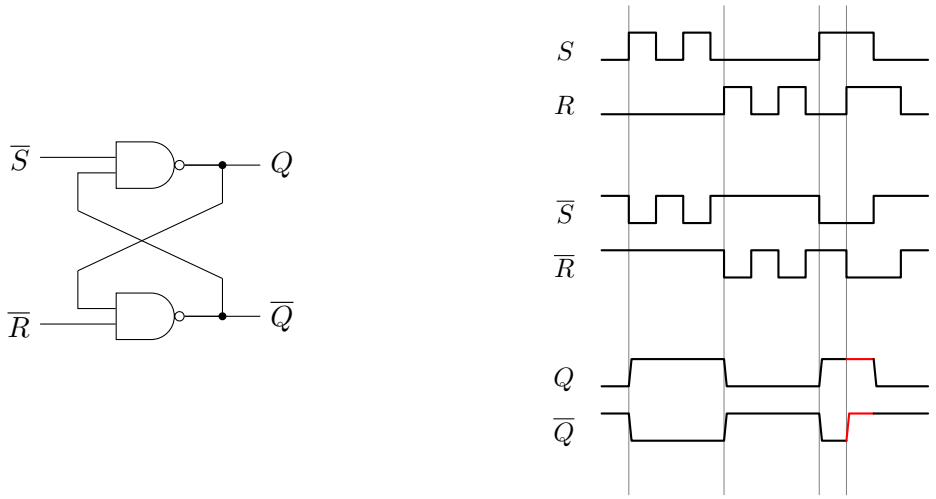


Figura 8.4: *Latch* SR (S=sinal set; R=sinal reset; Q=saída (estado)), com porta NAND.

Neste caso também $S = 1$ faz o estado ir a 1 (Q=1). Similarmente, $R = 1$ faz o estado ir a 0 (Q=0). Temos a situação *set* (com $S = 1$) e *reset* ($R = 1$) como no caso anterior. Porém o circuito é tal que a mudança de estado ocorre com o sinal baixo (no caso anterior, ocorria com o sinal alto) e é por esta razão que a primeira entrada é $\overline{S}$ e a segunda é $\overline{R}$. Note que neste caso também $S = R = 1$ leva o *latch* a um estado inválido.

Cada *latch* pode ser pensado como uma unidade de memória capaz de armazenar um *bit*. Por meio das entradas S e R podemos controlar qual valor armazenar. Em geral, um sistema é composto por várias dessas unidades de memória e pode ser interessante termos a possibilidade de controlar quando elas devem mudar de estado. Para isso, pode-se incluir uma entrada de controle como mostrado na figura 8.5. Quando o sinal de entrada C é 0, a saída de ambas as portas E é 0 e portanto mudanças no valor de R e S não tem efeito nenhum sobre o estado do *latch*. Quando $C = 1$, temos o mesmo comportamento descrito anteriormente. A literatura denomina em geral de *latches* quando não temos uma entrada para o sinal de controle e de *flip-flops* quando há. Mas essa nomenclatura parece não ser universal.

Um sinal de *clock* pode ser usado para sincronizar a mudança de estados de múltiplos *flip-flops*. Note que é importante garantir que os sinais na entrada S e R estejam devidamente preparados no momento em que o sinal C sobe, para que as mudanças esperadas ocorram.

Daqui em diante, sempre que mencionarmos *flip-flop* SR, estaremos considerando a existência de entrada para o sinal de controle. Considerando a entrada C para o sinal de controle, a expressão

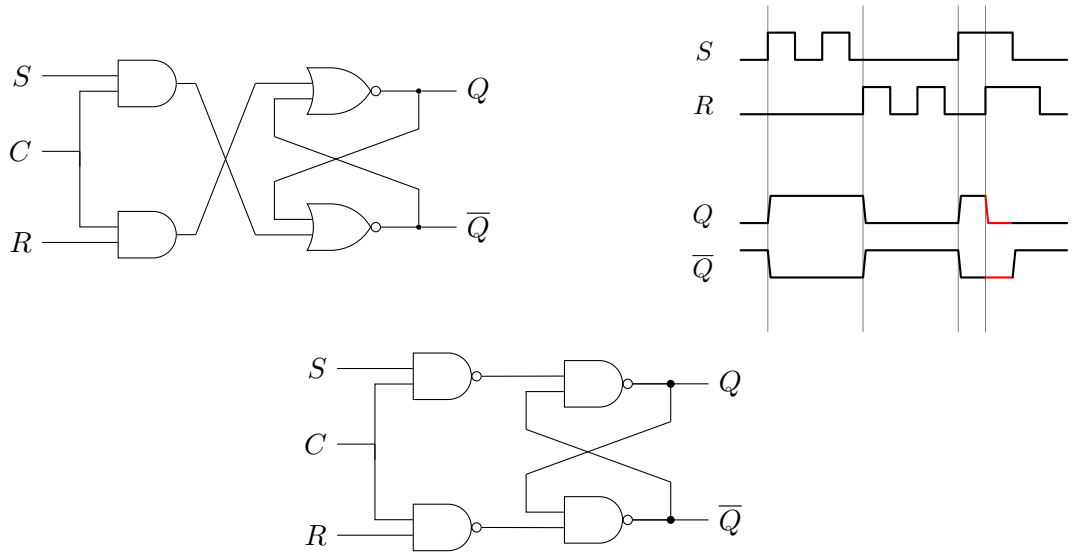


Figura 8.5: Acima: um *flip-flop* SR (há uma entrada para o sinal de controle), com porta NOR. Abaixo: um *flip-flop* SR com porta NAND. Quando $C = 1$, estes *flip-flops* comportam-se da mesma forma que o *latch* SR mostrado na figura 8.3.

do próximo estado de um *flip-flop* SR é dada por

$$Q^* = S\,C + Q\,\overline{R} + Q\,\overline{C}\,.$$

Se $C = 0$, temos $Q^* = Q$; se $C = 1$ então temos $Q^* = S + Q\,\overline{R}$, que é a mesma equação do *latch* SR.

8.5 *Flip-flop* JK

Vimos que nos *flip-flops* SR, se ambas as entradas estiverem em 1 simultaneamente, eles podem apresentar comportamento imprevisível. Isso, do ponto de vista prático, é indesejável pois o projetista do circuito precisaria garantir que as duas entradas nunca ficarão ativas simultaneamente.

Os *flip-flops* JK são uma evolução do SR pois não apresentam esse problema. No JK, quando ambas as entradas ficam em 1 simultaneamente, o estado do circuito se inverte, isto é, se era 1 vai a 0 e vice-versa. A figura 8.6 mostra duas realizações do *flip-flop* JK, uma usando portas NOR e outra usando portas NAND. Ambos comportam-se da mesma forma. Note que quando $J = K = 1$, no mesmo ponto no qual o *flip-flop* SR gerava um estado inválido, ocorre a inversão de estado. É importante notar que o sinal em C deve ser um pulso (valor 1 por um pequeno

instante de tempo) para que a saída das portas no primeiro nível não mude mais de uma vez. Isto é, quando uma eventual alteração em Q ou $\overline{Q}$ chegar a essas portas, C já deve estar em 0. Caso C esteja em 1, quando $J = K = 1$, uma nova rodada de propagação de sinais ocorrerá e o *flip-flop* poderá apresentar comportamento não esperado.

Para simular essa situação de oscilação de estado, vamos considerar o *flip-flop* que usa as portas NAND (mostrado na Fig. 8.6) e supor que ele se encontra em uma situação na qual $C = 0$, $J = K = 0$, $Q = 1$ e $\overline{Q} = 0$ (como o *flip-flop* chegou a esse estado não é relevante). Primeiramente, vamos verificar que este é um estado consitente (o circuito está estabilizado) — a verificação fica como exercício. Agora vamos passar o valor de C de 0 para 1. Isto fará com que as portas NAND no primeiro nível do circuito (aquelas que recebem como entrada os sinais J e K, respectivamente) fiquem habilitados para mudanças. Em seguida, passemos o valor tanto de J como de K de 0 para 1, simultaneamente. A porta NAND no primeiro nível, parte inferior, que recebe como entradas C, K e Q, mudará de estado (a saída dela irá mudar de 0 para 1). Isto fará com que o valor de $\overline{Q}$ mude para 1 e Q mude para 0. O circuito deveria se estabilizar com esses valores, porém os valores novos de Q e $\overline{Q}$ retroalimentarão as postas NAND no primeiro nível, e como C continua com valor 1, o sinal propagará alterando agora Q de 0 para 1, e assim por diante.

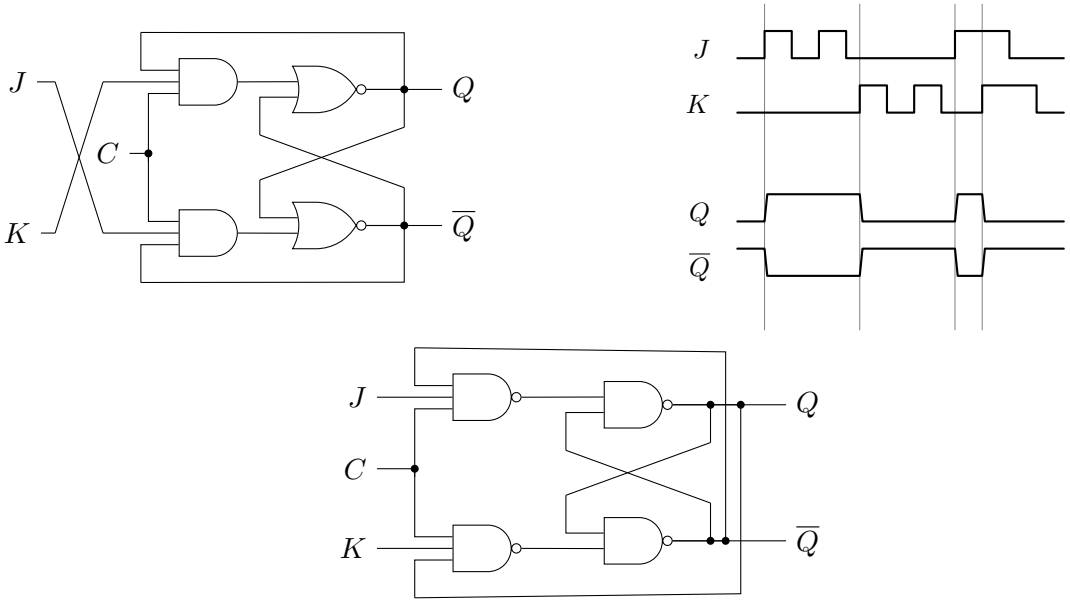


Figura 8.6: Flip-flop J-K, com portas NOR e com portas NAND.

O comportamento do *flip-flop* JK pode ser resumido pela seguinte tabela-verdade:

J	K	C	Q^*	
$\times$	$\times$	0	Q	não muda
0	0	1	Q	mantém
0	1	1	0	*reset*
1	0	1	1	*set*
1	1	1	$\overline{Q}$	inverte

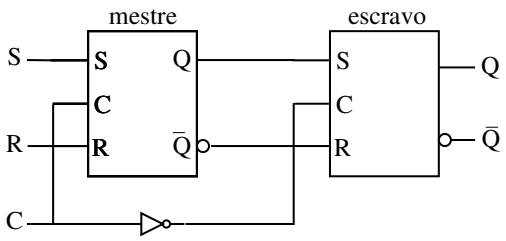
Para $C = 1$ a equação do próximo estado é dada por

$$Q^* = J\overline{Q} + \overline{K}\,Q\,.$$

8.6 *Flip-flops* mestre-escravo

Vimos acima que os *flip-flops* do tipo SR não funcionam propriamente quando $S = R = 1$. Esse problema é contornado com *flip-flops* do tipo JK, que não mais apresentam esse problema. No entanto, no *flip-flop* JK, se $C = 1$ e $J = K = 1$ por um longo período de tempo, o estado pode mudar várias vezes, um comportamento que é indesejável. Em condições ideais, poderíamos controlar C de modo que a duração do pulso seja menor que o tempo necessário para que ocorra mais de uma mudança de estado (isto é, após a inversão dos valores de Q e $\overline{Q}$ e sua propagação de volta para as portas NAND no primeiro nível do circuito, o sinal de controle C deverá estar com valor 0). No entanto, na prática, este tipo de controle é difícil e pode nem ser possível.

Para contornar o problema acima, foram desenvolvidos os chamados ***flip-flops* mestre-escravo**. A característica deles é o fato de mudarem de estado uma única vez a cada pulso de C, independentemente da largura dele. Para entender esse mecanismo, usaremos o esquema de um *flip-flop* SR mestre-escravo, mostrado na figura a seguir.



Cada "caixa" representa um *flip-flop* SR. Quando C está em 0, o primeiro *flip-flop* (mestre) está desabilitado para mudanças. O segundo está habilitado para mudanças, mas como o primeiro não varia, o seu estado é uma cópia do estado do primeiro. Por outro lado, quando C está em 1,

o estado do primeiro varia de acordo com as variações nas entradas S e R, enquanto o segundo está "congelado" sem possibilidade de mudar de estaddo.

Supondo inicialmente $C = 0$, o estado do *flip-flop* é o estado do segundo, que é uma cópia do estado do primeiro. Nesta situação, se C vai a 1, o mestre fica habilitado para mudanças, mas o escravo fica desabilitado, e permanece com o mesmo estado anterior à subida do C. Note que para isso acontecer, deve-se garantir que o escravo será desabilitado antes que ocorra qualquer mudança na saída do mestre; isso pode de certa forma ser garantido pelo "delay" do inversor NÃO ser menor que das portas que estão no interior do mestre. Em seguida, se C for a 0, o mestre será desabilitado (e portanto sua saída será "congelada") e a mudança de estado no escravo será habilitada. Neste momento, a saída do mestre é copiada para a saída do escravo (isto é, ocorre a transição de estado do *flip-flop*).

Esse mecanismo mestre-escravo garante que a cada pulso de C ocorrerá apenas uma mudança de estado no *flip-flop*. Obviamente, se as entradas S e R variarem enquanto C estiver alto, o estado do mestre oscilará correspondentemente. Logo, há necessidade de se garantir que as entradas estão estabilizadas nos valores corretos quando C vai a zero, quando o estado do mestre é congelado e copiado para o escravo.

Uma versão mestre-escravo do *flip-flop* JK é mostrada na figura 8.7. Este circuito evita a oscilação de estado que pode ocorrer na situação em que J e K estão com valor 1 e o sinal C permanece em 1 por um período maior que o ideal, conforme discutido anteriormente.

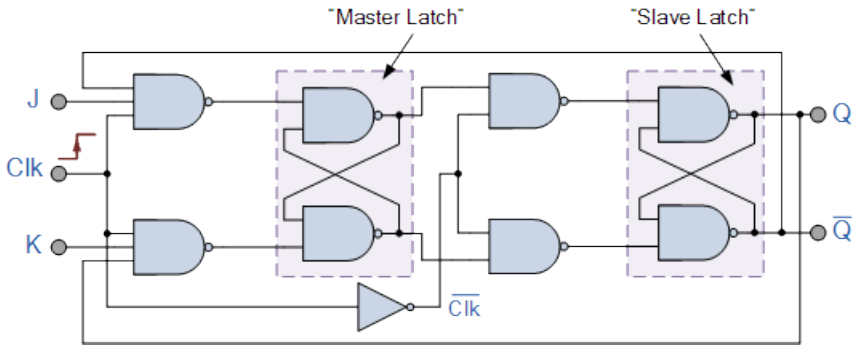


Figura 8.7: Esquema de um flip-flop JK mestre-escravo (imagem extraída de https://www.electronics-tutorials.ws/sequential/seq_2.html)

Na situação $J = K = 1$ e $C = 1$, as portas NAND no primeiro nível garantem que as entradas S e R do *latch* mestre nunca serão tais que $S = R = 1$ (pois uma porta NAND é alimentada pelo Q e o outro pelo $\overline{Q}$). Isto irá evitar que ocorra a oscilação decorrente do fato de as entradas serem $J = K = 1$. Desta forma, esse circuito também comporta-se de forma análoga ao circuito anterior. Para garantir o estado correto, deve-se igualmente garantir que as entradas J e K estão

estáveis e com os valores corretos no momento em que o sinal de *clock* vai a zero (momento em que a mudança de estado do escravo é habilitada e o estado do mestre é copiado ao escravo). Este circuito também não realiza mais que uma mudança de estado por pulso do sinal de *clock*.

Os *flip-flops* do tipo mestre-escravo que mudam de estado não mais que uma vez a cada pulso, são chamados de *flip-flops* disparados **por pulso**.

Atualmente os *flip-flops* mestre-escravo são considerados um tanto obsoletos. O problema da oscilação quando $J = K = 1$ foi contornado por meio de circuitos detectores de borda. A ideia central dos detectores de borda é identificar as bordas de subida ou de descida de um pulso, explorando o *delay* de um inversor conforme mostrados na figura 8.8.

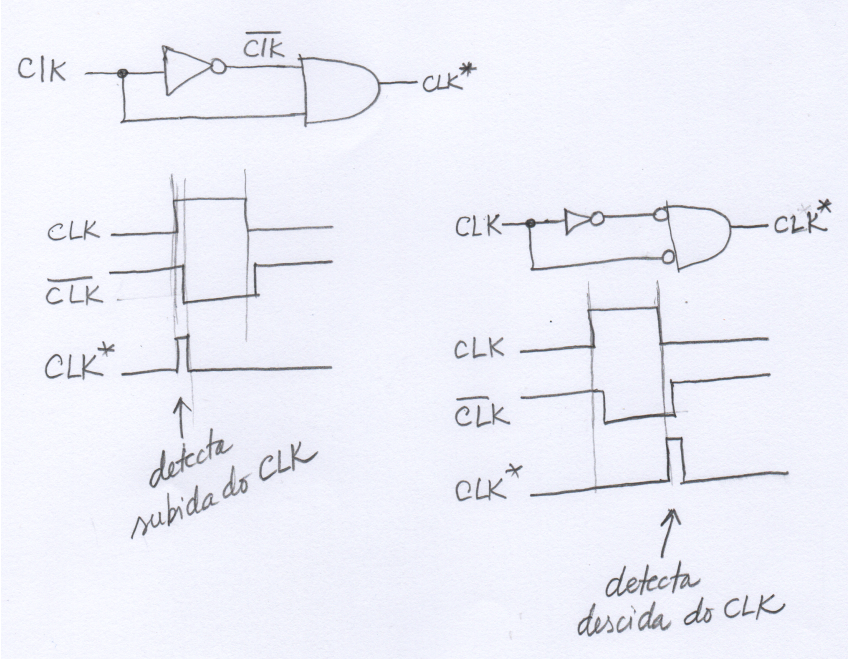


Figura 8.8: Circuitos para detecção de borda de subida e descida, aproveitando o atraso do inversor (NOT).

Desta forma, não importando a largura do pulso do *clock*, os circuitos detectores produzem pulsos de curtíssima duração ou na subida ou na descida do sinal de *clock*. Logo, podemos alimentar a entrada C do *flip-flop* JK com esse pulso de curtíssima duração que será suficiente para disparar e permitir uma, mas não mais que uma, mudança de estado. Desta forma temos garantia de que ocorrerá apenas uma mudança de estado a cada pulso do *clock*. Isto elimina a necessidade de *flip-flops* do tipo mestre-escravo.

Os *flip-flops* que utilizam esse mecanismo de detecção de bordas são chamados de ***flip-flops* disparados por borda** (*edge triggered flip-flops*). A mudança de estados pode ser disparada

ou na subida ou na descida do sinal de *clock*. A figura 8.9 mostra o esquema de um *flip-flop* JK, disparado por borda de subida. Ao lado esquerdo, o circuito completo, e à direita a representação caixa-preta dele. O triangulinho na entrada C indica que o *flip-flop* é *edge-triggered*, e as entradas PRE e CLR são controles assíncronos para carregar o valor 1 e para zerar o valor, a qualquer momento.

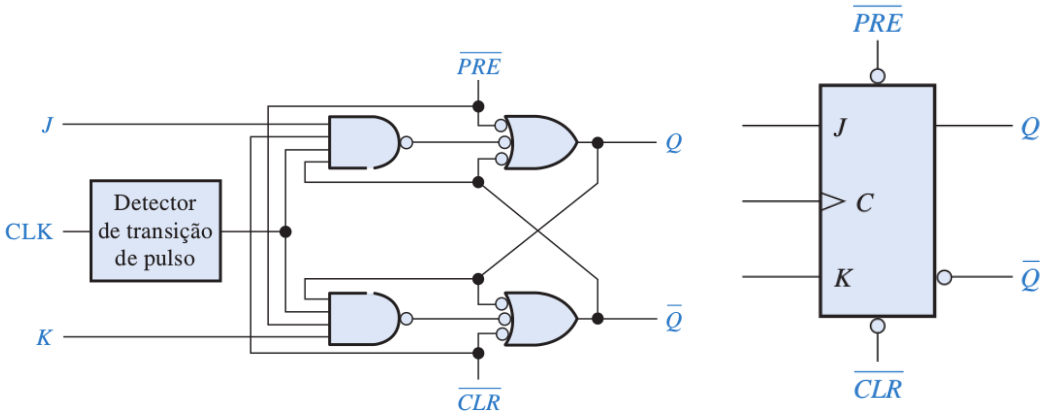


Figura 8.9: Um *flip-flop* JK *edge-triggered*.

PRE=1 faz o estado ir a 1 (i.e., $Q = 1$) e CLR=1 faz o estado ir a 0 (i.e., $Q = 0$). Note que não se pode colocar os dois em 1 simultaneamente. Os seguintes parágrafos explicam como isso acontece.

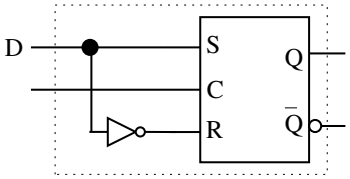
Sempre que fazemos CLR=1, o estado Q irá a zero, independentemente dos demais sinais. Para ver isso, note que se CLR=1, então $\overline{\text{CLR}} = 0$. Assim, já que esse 0 alimenta a terceira entrada da porta OU de baixo e como ele é negado na entrada, torna-se 1. Logo, a saída dessa porta será 1 (isto é, teremos $\overline{Q} = 1$). Na porta OU de cima, as três entradas serão 0: na primeira entrada $\overline{\text{PRE}} = 1$ (já que devemos ter PRE=0 quando CLR=1) que, negado na enrada, é igual a 0; na segunda entrada, a saída da porta E anterior é 1 (pois a segunda entrada dela é $\overline{\text{CLR}} = 0$) que, negado na entrada, também é 0; na terceira entrada temos $\overline{Q} = 1$ que, negado na entrada, fica igual a 0. Assim, a porta OU de cima faz o OU de três 0s, resultando em $Q = 0$. Como isso não mais afeta a saída da porta OU de baixo, o estado estabiliza-se em $Q = 0$.

Por outro lado, quando fazemos PRE=1, teremos $\overline{\text{PRE}} = 0$, que é negado antes na entrada da porta OU de cima. Isso fará com que $Q = 1$. Na porta OU de baixo, as três entradas serão zero: a primeira entrada é Q, que é negado na entrada e, portanto, será igual a 0; a segunda entrada é a saída da porta E anterior, que negado na entrada da porta OU fica igual a 0 (a saída da porta E é 1 pois uma de suas entradas é $\overline{\text{PRE}} = 0$); a terceira entrada é $\overline{\text{CLR}} = 1$ que, negado

na entrada, fica igual a zero. Assim, a saída da porta OU de baixo é 0 e isso não irá afetar a saída da porta OU de cima, e o estado Q estabiliza-se em 1.

8.7 *Flip-flops* D e T

Flip-flop D é um *flip-flop* que possui uma entrada D e tal que quando a mudança de estado é disparada, copia o valor da entrada para o estado. Um *flip-flop* desses pode ser implementado usando um *flip-flop* SR, alimentando a entrada S com o sinal D e a entrada R com o seu complemento (isso garante também que S e R nunca estarão em 1 simultaneamente), como na figura a seguir. Pode-se também usar um *flip-flop* JK em vez de um *flip-flop* SR.

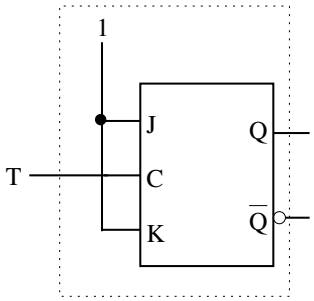


A expressão do próximo estado do *flip-flop* D é dada por

$$Q^* = D\,C + \overline{C}\,Q$$

ou seja, se $C = 0$, $Q^* = Q$ e, se $C = 1$, então $Q^* = D$.

Flip-flop T é um *flip-flop* que inverte o seu estado sempre que há um pulso na entrada T. Um *flip-flop* desses pode ser implementado usando um *flip-flop* JK, com ambas as entradas, J e K, fixas em 1 e com o sinal T alimentando a entrada de controle C, conforme mostrado na figura a seguir.



A expressão do *flip-flop* T é dada por

$$Q^* = \overline{Q}\,.$$

8.8 Comentários finais

Neste capítulo descrevemos alguns tipos de circuitos que funcionam como uma "memória" capaz de armazenar um *bit* (valor 0 ou 1) e cujo valor armazenado permanece constante até que o circuito seja instruído explicitamente, por meio de sinais de entrada, a mudar de valor. Os sinais de entrada que permitem instruir o circuito são o *set* e o *reset*, o primeiro para colocar o estado em 1 e o segundo para colocar o estado em 0. Esses circuitos são em geral conhecidos por *latches* ou *flip-flops*, dependendo de características específicas presentes ou não neles (tais como um sinal de controle que habilita ou desabilita a mudança de estados). Dois *flip-flops* comuns são o SR e o JK, sendo que a diferença é que o JK permite que ambas as entradas, *set* e *reset*, estejam ativas simultaneamente (o efeito, neste caso, é a inversão do estado). Ao longo dos anos esses dispositivos foram sendo aperfeiçoados para dar conta, por exemplo, de comportamento não desejado como a oscilação de estado. Os *flip-flops* mais recentes são do tipo disparados por borda (*edge-triggered*). A entrada de controle (que habilita ou desabilita a mudança de estado) desses *flip-flop* é alimentada por pulsos de curtíssima duração (que são gerados no momento da subida ou então da descida do sinal de controle). Desta forma, garante-se que ocorre apenas uma mudança de estado por pulso do sinal de controle e as mudanças de estado ocorrem ou apenas na descida do pulso ou apenas na subida do pulso.

Nos capítulos subsequentes iremos considerar, portanto, que os *flip-flops* funcionam todos de forma ideal, sem problemas de oscilação, ambiguidade de estados, ou comportamento imprevisível.

Exercícios

1. Escreva a tabela-verdade a expressão (minimizada na forma SOP) do próximo estado Q^* de um *flip-flop* SR, em função dos valores de suas entradas S e R e do valor do seu estado Q. Note que é suposto que a entrada $S = R = 1$ nunca ocorrerá.

2. Repita o exercício anterior, desta vez para o *flip-flop* JK. Note que no JK todas as combinações de valores para as entradas J e K são permitidas.

3. O estado de um *flip-flop* JK passou de 1 para 0. Quais valores nas entradas S e R podem ter provocado essa transição ?

4. Definimos como *flip-flops* os *latches* controlados, isto é, aqueles que possuem uma entrada para um sinal de controle, usado para habilitar ou desabilitar a mudança de estado. Em geral esse controle é feito com o sinal de um *clock* (um sinal periódico com frequência

constante). Qual deve ser a duração de um pulso do *clock* para podermos garantir que um *flip-flop* JK controlado irá funcionar corretamente ?

5. Para contornar problemas de funcionamento que podem ser ocasionados por pulsos de controle de largura fora do esperado, foram desenvolvidos os *flip-flops* disparados por borda. Explique como eles funcionam e qual a importância dessa estratégia de disparo por borda.

6. O que é um *flip-flop* D? Como ele pode ser implementado usando um *flip-flop* JK ?

Capítulo 9

Circuitos sequenciais – contadores, registradores e outros exemplos

Última atualização: 28/06/2021

No capítulo anterior foram mencionados os circuitos combinacionais ou sequenciais. Os sequenciais envolvem unidades de memória (*flip-flops*), enquanto os combinacionais não. Assim, no caso de circuitos combinacionais a saída depende apenas das entradas e qualquer mudança ocorrida nos sinais de entrada reflete-se imediatamente na saída. Já nos circuitos sequenciais a saída depende também do estado (memória) do circuito, que é atualizado na subida ou descida do sinal de controle. Em geral utiliza-se o sinal de *clock* (um sinal periódico) para controlar a mudança de estado dos circuitos sequenciais.

Neste capítulo introduzimos inicialmente a noção de controle síncrono e assíncrono e em seguida apresentamos exemplos de circuitos sequenciais, construídos usando-se *flip-flops*.

9.1 Síncronos × Assíncronos

Em **circuitos síncronos**, a mudança de estado de todos os elementos de memória do circuito (*flip-flops*) ocorre de forma sincronizada ("ao mesmo tempo") e é controlada pelo sinal de um *clock*.

Em **circuitos assíncronos**, a mudança de estado não é sincronizada, ou seja, não se utiliza o sinal de um *clock* para se disparar a mudança de estado de todos os *flip-flops*. Há dois modos

para disparar a mudança de estado em circuitos assíncronos. No modo pulso, o controle do *flip-flop* pode ser alimentado por qualquer outro sinal. No modo fundamental, o retardo intrínseco ou propositalmente colocado no circuito é utilizado para funcionar como "memória". Em ambos os casos há restricões que devem ser satisfeitas para o circuito funcionar propriamente. Um aprofundamento sobre esse tópico está fora do escopo destas notas de aula.

Os exemplos apresentados nas seções subsequentes são todos síncronos, a não ser que seja explicitamente mencionado o contrário.

9.2 Verificador de paridade em transmissão serial

Suponha que, em um sistema de transmissão, os dados são agrupados de 7 em 7 *bits* e que, para cada grupo de 7 *bits*, um oitavo *bit* com a informação de paridade é adicionado. O valor desse oitavo *bit* depende dos valores dos 7 primeiros *bits*. Em paridade par, o valor do oitavo *bit* deve ser tal que o total de *bits* com valor 1 no conjunto de 8 *bits* seja par. Em paridade ímpar, o total de *bits* com valor 1 nos conjunto de 8 *bits* deve ser ímpar. Assim, a cada grupo de oito *bits* transmitidos tem-se necessariamente uma paridade fixa (ímpar ou par). Após a transmissão dos dados, o receptor pode verificar a paridade de cada 8 *bits* e eventualmente detectar erros (não necessariamente todos) que possam ter ocorrido no trajeto da transmissão.

A figura 9.1 mostra um circuito verificador de paridade para 4 *bits*, baseado em portas XOR. A saída desse circuito é 1 se, e somente se, a paridade dos 4 *bits* é impar.

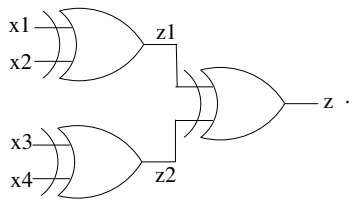


Figura 9.1: Circuito para verificação de paridade para uma entrada de 4 *bits*.

O circuito acima pode ser estendido para verificar a paridade de 8 *bits*, simplesmente conectando-se a saída de dois deles a uma porta XOR, conforme mostrado na figura 9.2.

Caso a transmissão seja paralela (isto é, situação na qual os 8 *bits* são transmitidos simultaneamente por 8 linhas de transmissão), a verificação de paridade poderia ser efetuada imediatamente na chegada do sinal. No entanto, se a transmissão for serial (*bits* transmitidos sequencialmente

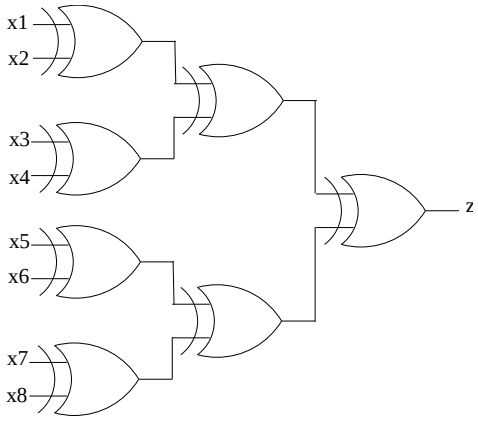


Figura 9.2: Circuito para verificação de paridade para entradas de 8 *bits*.

através de uma única linha de transmissão), a saída do circuito verificador de paridade corresponderá ao resultado da verificação somente após a chegada dos 8 *bits*.

Observe que a expressão da saída z do circuito da figura 9.2 (verificador de paridade ímpar de 8 *bits*) é dada por

$$z = [(x_1 \oplus x_2) \oplus (x_3 \oplus x_4)] \oplus [(x_5 \oplus x_6) \oplus (x_7 \oplus x_8)]$$

Porém, ela pode também ser expressa por

$$z = \Big[\Big(\big[\big([(x_1 \oplus x_2) \oplus x_3] \oplus x_4\big) \oplus x_5\big] \oplus x_6\Big) \oplus x_7\Big] \oplus x_8\,.$$

Esta última equação sugere que a verificação de paridade pode ser realizada iterativamente, *bit* a *bit*. O circuito sequencial que realiza tal verificação é mostrado na figura 9.3.

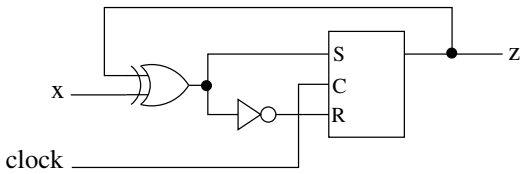


Figura 9.3: Circuito sequencial para verificação de paridade.

Tomemos como exemplo uma sequência de 4 *bits*, 1011 (apenas 4 *bits* para não alongar a explicação). Os *bits* desta sequência irão alimentar a entrada x do circuito, sincronizado com o *clock*. Suponha que o *flip-flop* está inicialmente em 0 e que ele é disparado na descida do sinal de clock (o desenho está desatualizado; deveria haver uma bolinha eum triangulinho na entrada C do *flip-flop*...). O primeiro *bit* da sequência, 1, fará com que a saída da porta XOR seja 1, e isso fará com que a entrada em S seja 1 e em R seja 0. Nestas condições, quando o *clock* baixa,

z passará a 1 (e esta saída é consistente com o número de *bits* 1 vistos até agora). No próximo pulso do *clock*, x vale 0 (segundo *bit* da sequência). Como $z = 1$ e $x = 0$, a saída do XOR continuará em 1. Na descida do *clock*, o *flip-flop* habilita-se para mudar de estado, mas como temos $S = 1$, $R = 0$ e $z = 1$, z continua como 1. Na próxima iteração, temos $x = 1$. Neste caso, o XOR entre $x = 1$ e $z = 1$ resultará em 0. Com isso, na descida do *clock*, o estado do *flip-flop* vai a zero. Note que a saída $z = 0$ neste momento é consistente com os três bits 101 processados até o momento (paridade é par e não ímpar). No próximo pulso do *clock*, x vale 1 e $z = 0$; logo a porta XOR produzirá 1 e isto "setará" a saída z. O resultado $z = 1$ indica corretamente que a paridade é ímpar.

Desta forma, assim que o oitavo *bit* é processado pelo circuito, tem-se a verificação de paridade concluída. Este exemplo, além de ilustrar o uso de *flip-flops*, também ilustra situações nas quais um circuito sequencial faz mais sentido do que um combinacional. Mais do que disso, esse verificador de paridade sequencial pode ser usado independentemente do número de *bits* considerado; já o combinacional tem número de entradas fixa.

9.3 Contadores

Um circuito contador é um circuito sequencial consistindo de n *flip-flops*, capaz de armazenar números binários de 0 a $2^n - 1$, e cujo valor armazenado é incrementado ou decrementado de 1 a cada pulso do sinal de *clock*.

Um **contador incremental módulo** 2^n, com valor inicial 0, apresenta a seguinte sequência de transição de valores:

$$0, 1, 2, 3, \ldots, 2^n - 1, 0, 1, 2, 3, \ldots$$

Por exemplo, o contador incremental módulo 2^3 ($n = 3$) pode ser definido pela tabela a seguir. Nela, $x_2\, x_1\, x_0$ representa o estado atual do circuito (três *bits*) e $x_2^*\, x_1^*\, x_0^*$ representa o próximo estado.

Estado atual $x_2\, x_1\, x_0$	Próximo estado $x_2^*\, x_1^*\, x_0^*$
000	001
001	010
010	011
011	100
100	101
101	110
110	111
111	000

Uma implementação desse contador usando *flip-flops* disparados na borda de subida do *clock* é mostrado na figura 9.4.

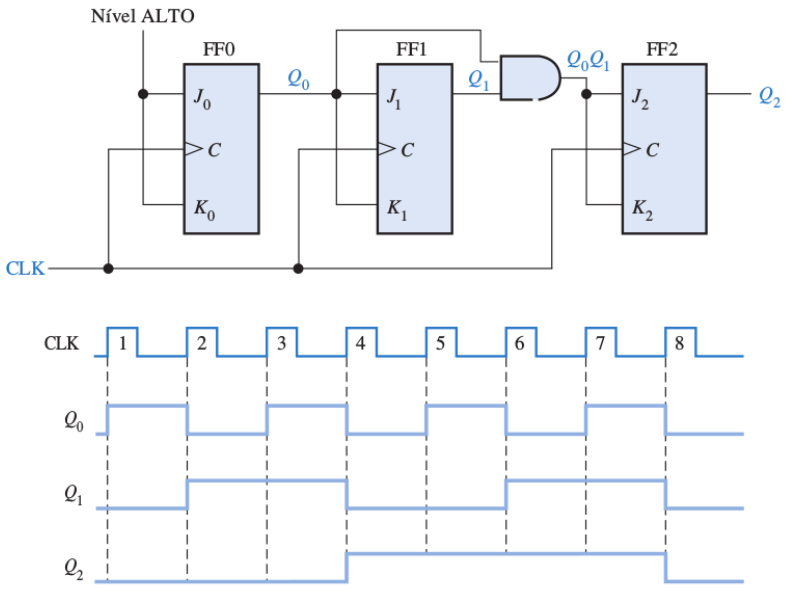


Figura 9.4: Contador módulo 2^3 (Floyd, Sistemas Digitais – Fundamentos e Aplicações, Pág. 454, edição 9).

São usados três *flip-flops*, um para armazenar cada um dos três *bits*. No caso, $x_0 = Q_0$, $x_1 = Q_1$ e $x_2 = Q_2$. O triângulo na entrada C do *flip-flop* indica que ele é disparado por borda. Mais especificamente, na subida do sinal de *clock*. Note que a mudança de estados dos três *flip-flops* é disparado simultaneamente. O primeiro *flip-flop* (FF0) é um *flip-flop* do tipo T que simplesmente inverte o estado a cada pulso do *clock*. FF1 e FF2 são também *flip-flops* do tipo T. Logo abaixo do circuito, encontra-se o diagrama temporal que mostra o comportamento do circuito. Assim, no primeiro pulso do *clock*, Q_0 passa de 0 para 1 (pois $J_0 = K_0 = 1$). Já Q_1 permanece em 0 pois no momento da subida do *clock*, Q_0 ainda estava em 0. No segundo pulso

do *clock*, Q_0 inverte e volta a 0. Já Q_1 vai a 1 pois Q_0 estava em 1 e portanto estávamos com $J_1 = K_1 = 1$ no momento da subida do *clock*. O terceiro *flip-flop* só muda de estado se os dois anteriores estiverem ambos em 1, no momento da subida do *clock*, como podemos observar no pulso 4. Após 8 pulsos do *clock*, o estado do circuito volta para 000.

Podemos construir também contadores incrementais-decrementais, conforme mostrado na figura 9.5. A estrutura do circuito é de certa forma similar ao do contador visto acima, porém ele possui uma entrada adicional (o controle crescente / $\overline{\text{decrescente}}$). Quando crescente=1 o circuito realiza incrementos e quando crescente=0 ele realiza decrementos. De fato, o decremento pode ser implementado de forma similar ao incremento, porém usando a saída $\overline{Q}$ dos *flip-flops*. Outra diferença é que a partir do segundo *flip-flop* há uma porta OU que propaga o sinal de Q caso crescente=1, e propaga o sinal de $\overline{Q}$ caso crescente=0.

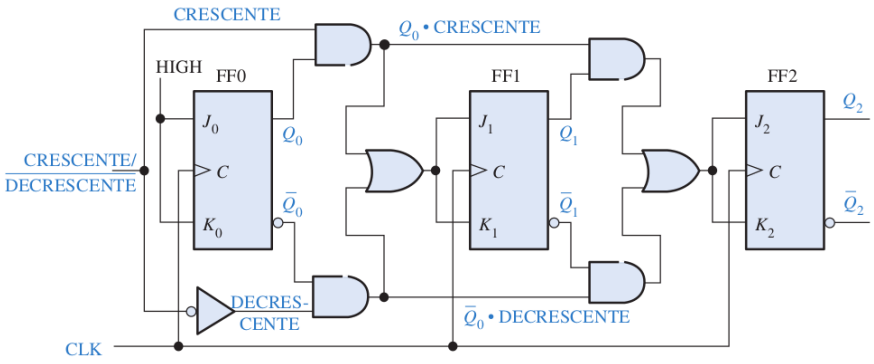


Figura 9.5: Contador incremental/decremental módulo 2^3 (Floyd, Sistemas Digitais – Fundamentos e Aplicações, edição 9).

9.3.1 Contadores assíncronos

Nos contadores incrementais módulo 2^n assíncronos, a mudança de estado dos *flip-flops* não é necessariamente controlada pelo sinal de *clock*. No exemplo da figura 9.6, o primeiro *flip-flop* (FF0, que corresponde ao *bit* menos significativo) é o único que é controlado pelo *clock*. Ele comporta-se da mesma forma ao do primeiro *flip-flop* do contador síncrono. Os *flip-flops* do segundo em diante são controlados pelas saídas dos *flip-flops* imediatamente anteriores. Por exemplo, FF1 muda de estado na borda de subida do sinal que alimenta a sua entrada C, que no caso é $\overline{Q}_0$. A borda de subida de $\overline{Q}_0$ corresponde à borda de descida de Q_0. Assim, Q_1 muda de estado quando Q_0 passa de 1 para 0. O mesmo raciocínio aplica-se para a mudança de estado de FF2. Este muda de estado na borda de subida do $\overline{Q}_1$ (que corresponde à borda de descida do Q_1).

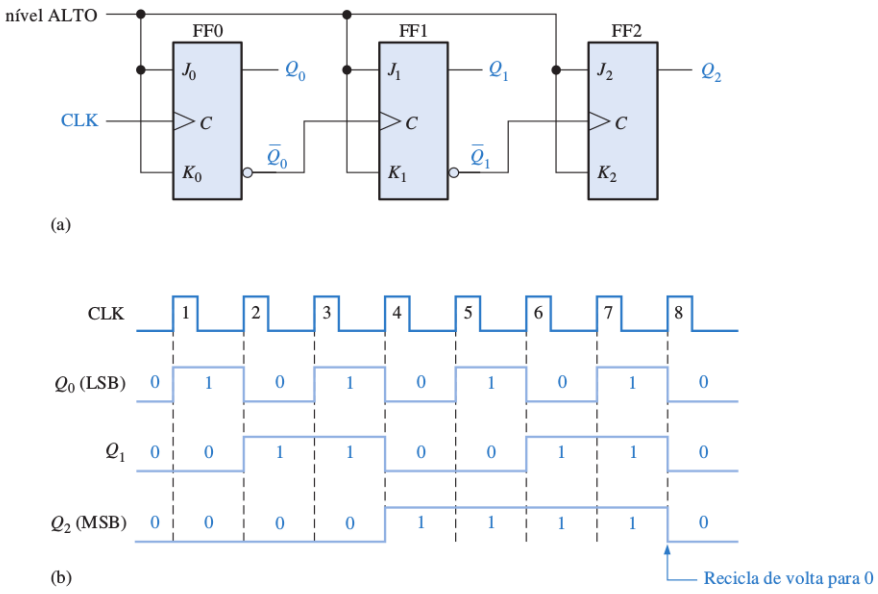


Figura 9.6: Contador assíncrono de 3 *bits* (Floyd, Sistemas Digitais – Fundamentos e Aplicações, Pág. 446, edição 9).

Note que quando os *flip-flops* são conectados em cascata, temos um circuito que funciona como divisor de frequências. Os exemplos da figura 9.7 mostram a divisão da frequência por dois e por quatro.

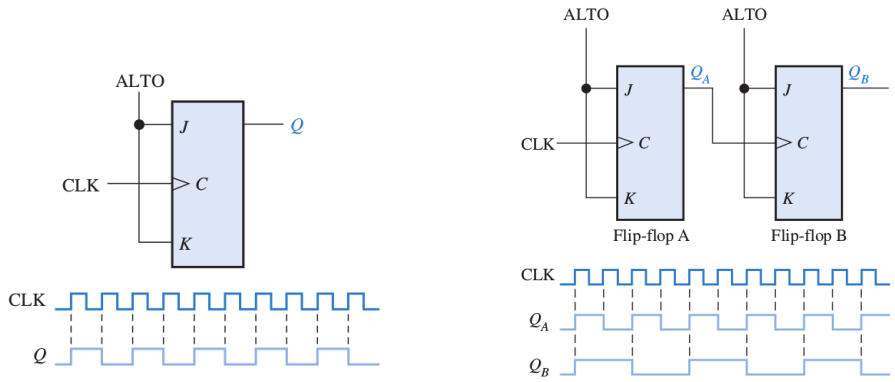


Figura 9.7: Divisão de frequência, usando *flip-flops* em cascata.

9.4 Registradores

Registradores podem ser pensados como uma memória especial com capacidade para armazenar tipicamente uma "palavra" (n *bits*, tais como 32, 64, 128). Essas memórias especiais são usadas pelo processador como memória temporária durante a execução das instruções. Alguns desses usos serão destacados num capítulo adiante (sobre a organização de computadores).

Quando projetamos registradores, devemos considerar como os valores serão armazenados nele e como os valores armazenados nele poderão ser obtidos. As formas de carregamento e leitura de valores de um registrador podem ser seriais ou paralelas. Serial é a situação em que os *bits* da palavra são processados um a um, enquanto paralela é aquela na qual todos os n *bits* são processados de uma única vez. A figura 9.8 ilustra essas formas de operação de registradores.

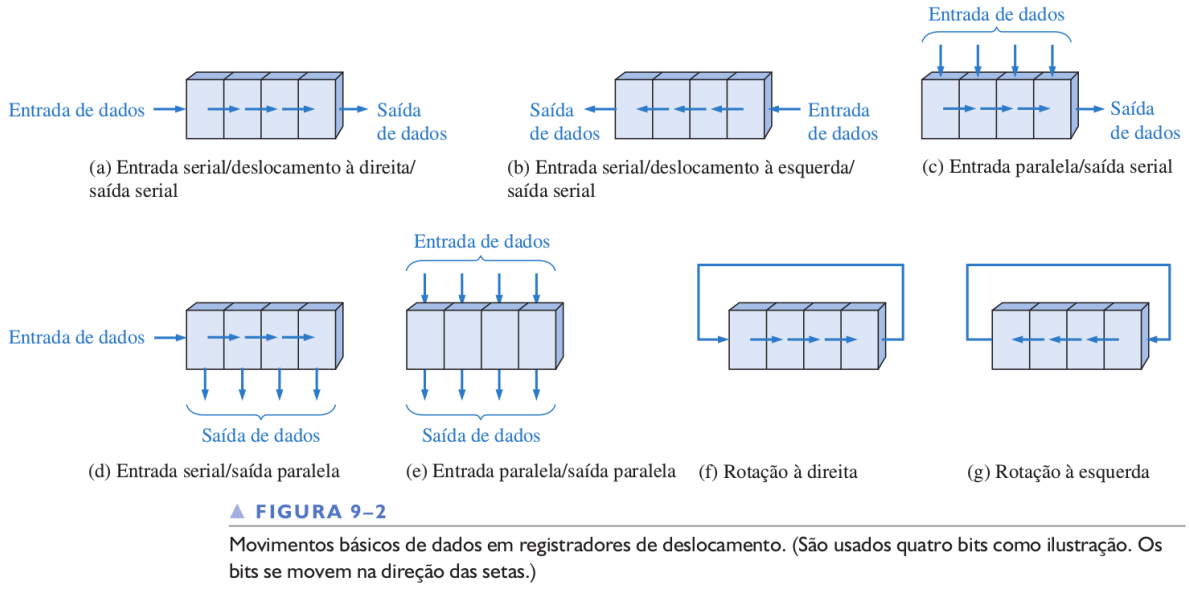
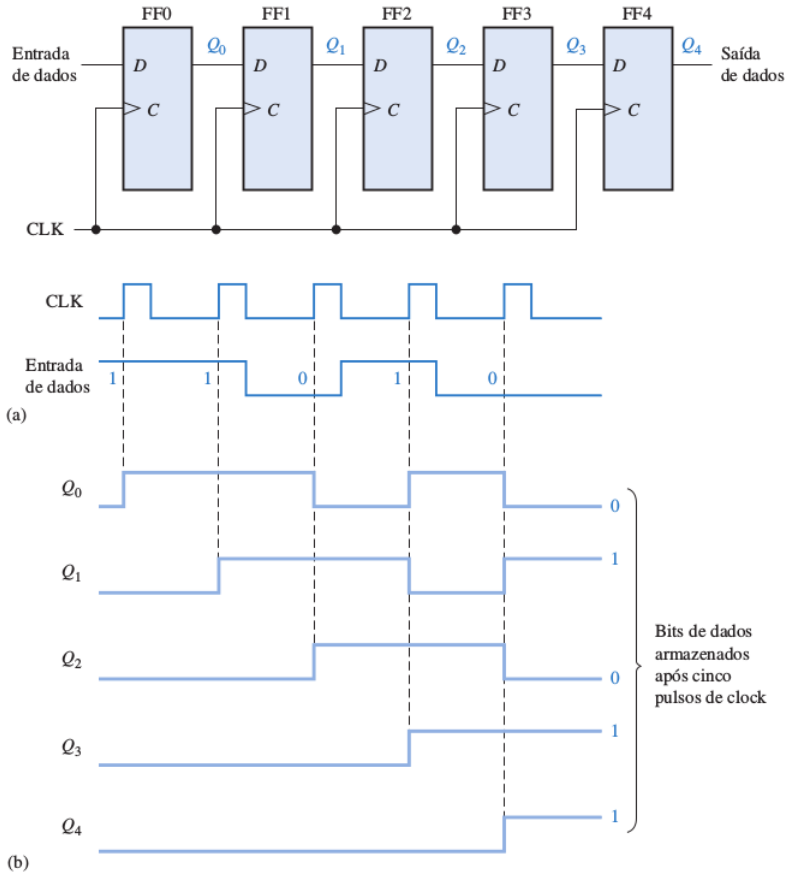
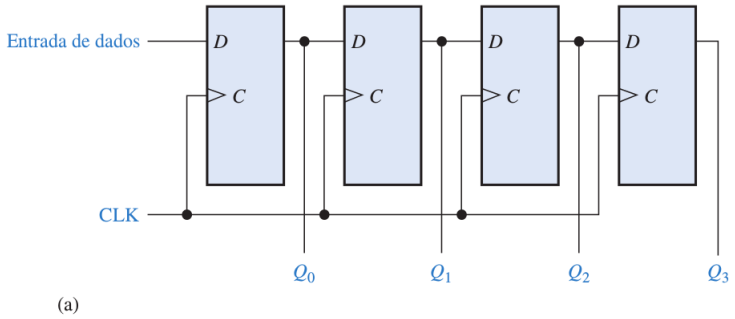


Figura 9.8: Modos de operação de registradores. (fonte: Floyd)

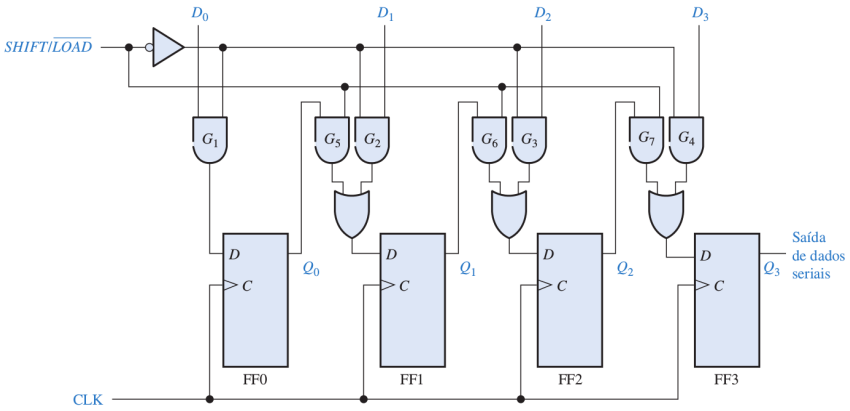
O circuito de alguns desses registradores são apresentados a seguir. Cada *bit* pode ser armazenado em um *flip-flop* do tipo D. No caso de entrada ou saída serial, quando o *clock* sobe, o *bit* armazenado em um *flip-flop* é copiado para o *flip-flop* a sua direita. No caso de entrada pararela, todas as entradas são "carregadas" nos respectivos *flip-flops* quando o *clock* sobe.



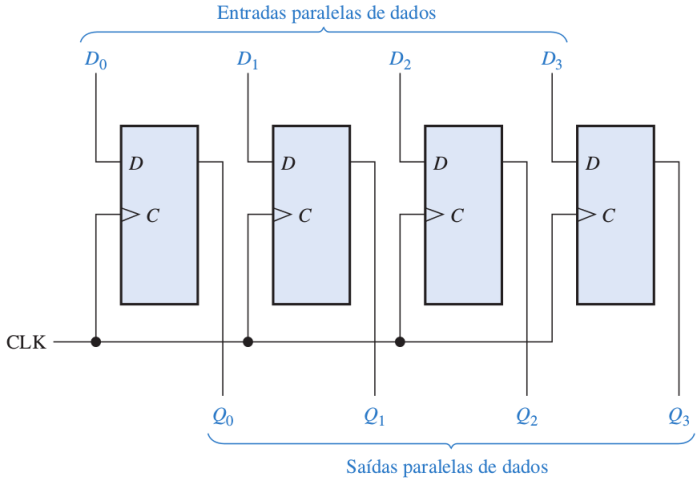
Entrada serial, *shift* à direita



Entrada serial, saída paralela



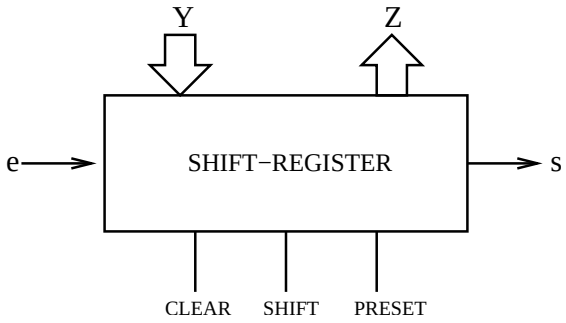
Entrada paralela, saída serial



Entrada paralela, saída paralela

9.4.1 Registradores genéricos

Da mesma forma que *flip-flops* possuem entradas de controle assíncronos, como *preset* e *clear*, registradores também possuem algumas entradas de controle. O diagrama a seguir ilustra um modelo genérico de processador.

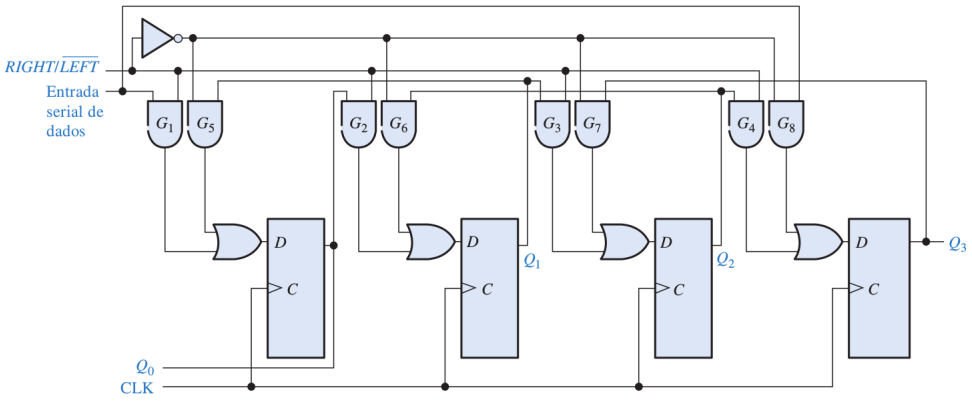


Entrada paralela: $Y = y_{n-1}\, y_{n-2}\, \ldots\, y_1\, y_0$
Saída paralela: $Z = z_{n-1}\, z_{n-2}\, \ldots\, z_1\, z_0$
Entrada serial: e é 1 *bit* de entrada
Saída serial: s é um *bit* de saída

Sinais de controle:

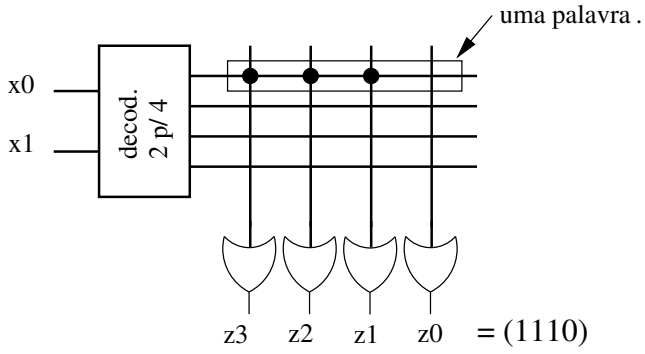
- CLEAR (Limpa): CLEAR=1 "zera" o registrador
- PRESET (Carrega): PRESET=1 "carrega" valor 1 em todos os *flip-flops*
- SHIFT / $\overline{\text{LOAD}}$ (desloca): se $SHIFT = 1$ desloca por uma posição para a direita todos os *bits*; se SHIFT = 0 (e portanto LOAD = 1) realiza o carregamento paralelo de Y.

Deslocadores RIGHT/$\overline{\text{LEFT}}$: neste caso, um sinal de controle pode ser usado para realizar deslocamentos para à esquerda ou para à direita. Note que um deslocamento à direita é equivalente à divisão por dois e para à esquerda é equivalente à multiplicação por dois.



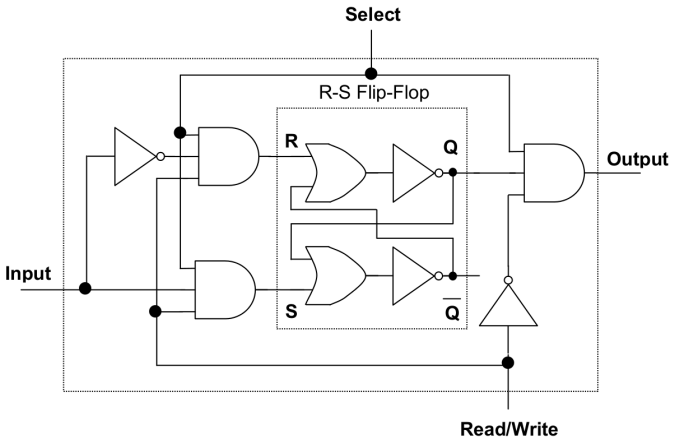
9.5 Memória RAM

Conforme visto anteriormente no capítulo 7, exempo 7.4.4, um esquema simplificado de memória ROM (*Ready only memory*) é ilustrado a seguir. Relembre que o decodificador recebe n *bits* (no exemplo, $n = 2$) correspondendo ao endereço da posição de memória a ser acessada, e ativa uma e apenas uma das suas linhas de saída. Assim, apenas os *bits* armazenados na palavra daquela linha são transferidos para $z_3\, z_2\, z_1\, z_0$. No exemplo, as palavras são de quatro *bits* e a palavra em destaque corresponde ao binário 1110.



A memória RAM (de *Random Access memory*) pode ser pensada como consistindo de uma estrutura (arranjo) similar à memória ROM. A diferença principal é que podemos tanto ler como escrever numa RAM. Cada ponto de cruzamento mostrado no esquema da ROM deve ser substituído na RAM por um *flip-flop*, capaz de armazenar um *bit*.

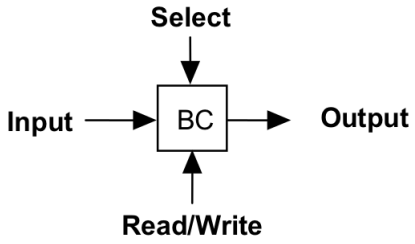
Uma realização de uma célula (*bit*) da RAM é mostrada no diagrama a seguir.



http://watson.latech.edu/book/circuits/circuitsMicrocomputer3.html

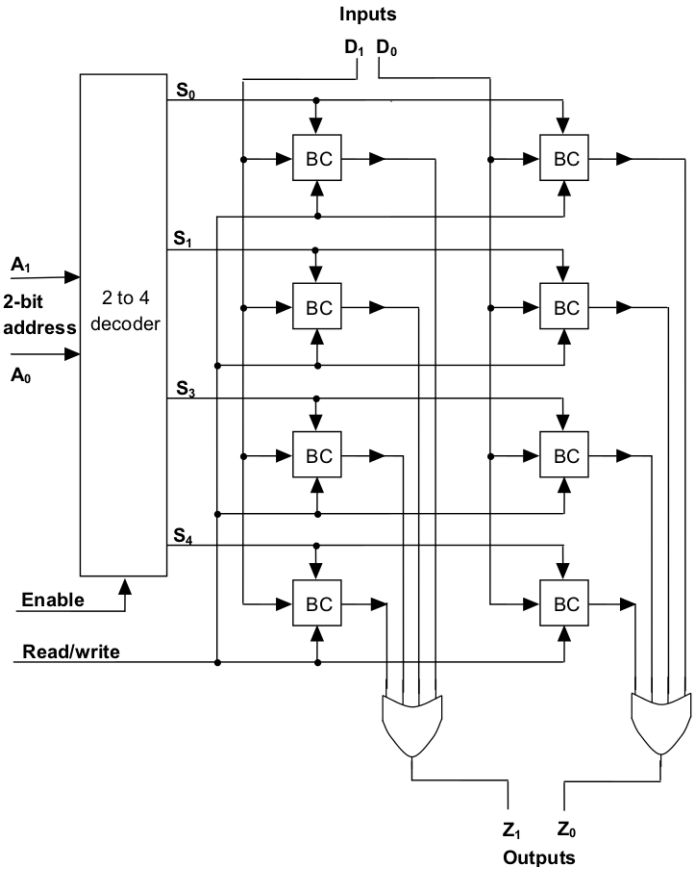
O circuito possui entrada e saída (input e output) e mais duas entradas de controle. A entrada `select` serve para habilitar a célula para leitura/escrita. Quando `select=0` a célula está desabilitada (tanto para mudança de estado como para leitura). A entrada $\overline{\text{read}}$/write indica a operação a ser executada: se $\overline{\text{read}}$/write=1 será realizada escrita na memória e se 0 será realizada a leitura da memória. Quando $\overline{\text{read}}$/write=1, note que a saída output será 0 e o estado Q ficará com o mesmo valor da entrada input. Já quando $\overline{\text{read}}$/write=0, as portas E que conectam-se ao S e R serão 0 e portanto não haverá alteração do estado Q. Por outro lado, a porta E da saída output será exatamente o valor de Q.

A célula acima pode ser esquematizada em sua versão caixa-preta conforme a seguir:



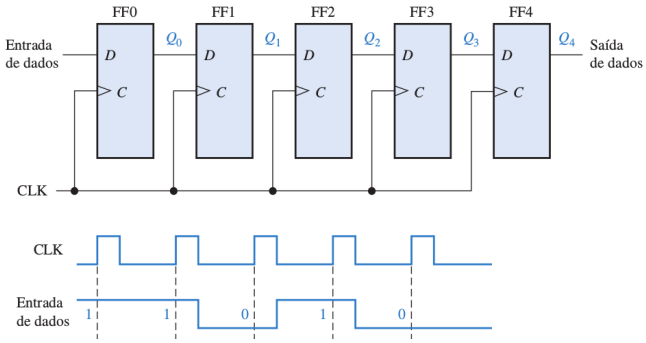
Para executar uma operação de escrita em uma célula, deve-se acertar o valor de *input*, colocar o valor de $\overline{Read}/Write$ em 1 e, em seguida, fazer *select* igual a 1.

A memória RAM pode ser vista como uma coleção de várias dessas células, em um arranjo similar ao da memória ROM já discutido anteriormente. O exemplo a seguir mostra uma memória com 4 palavras de dois *bits* cada. O endereço consiste de $n = 2$ *bits* para indicar a palavra que está sendo acessada. O sinal `enable` serve para habilitar/desabilitar o decodificador. Quando `enable=0`, nenhuma das saídas do decodificador estará ativa e portanto nenhuma operação é efetivada. Quando `enable=1`, então a palavra ativa é a correspondente à saída ativa do decodificador. Esta palavra será enviada para a saída $z_1\, z_0$ caso $\overline{Read}/Write = 0$, ou seu valor será alterado para $D_1\, D_0$ caso $\overline{Read}/Write = 1$.



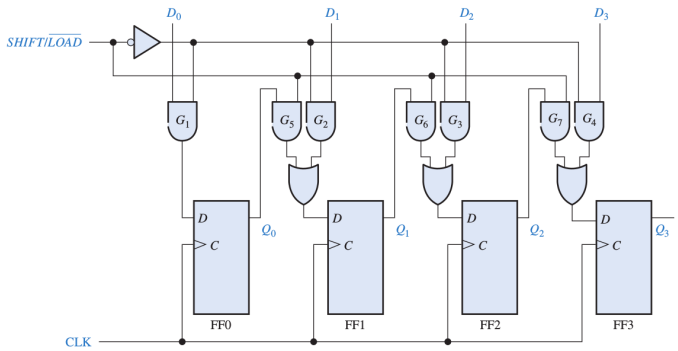
Exercícios

1. O circuito a seguir é um registrador com entrada/saída serial. Considere que inicialmente o registrador está no estado 00000. Complete o diagrama temporal com os estados Q_0 a Q_4. Qual é o estado do registrador após os cinco pulsos do *clock*?



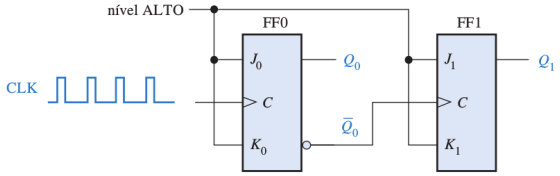
2. Como alterar o registrador do exercício anterior para que ele tenha também uma saída paralela?

3. No registrador a seguir há uma entrada indicada por $SHIFT/\overline{LOAD}$. Suponha a configuração exatamente como a mostrada, i.e., estado $Q_3Q_2Q_1Q_0$ e entrada $D_3D_2D_1D_0$. Mostre qual é o valor de $Q_3^*Q_2^*Q_1^*Q_0^*$, i.e., o estado após um pulso de *clock*, quando a entrada $SHIFT/\overline{LOAD} = 1$ e quando $SHIFT/\overline{LOAD} = 0$.

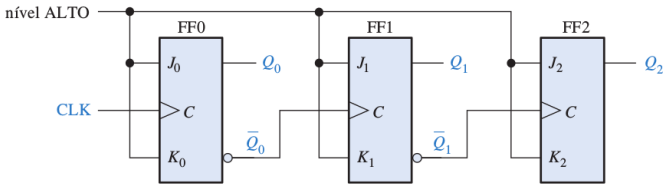


4. Como poderia ser implementado um registrador com entrada/saída paralela (não é necessário que o registrador tenha capacidade de fazer *shifts* / deslocamentos).

5. Simule o circuito a seguir por um período de 5 pulsos de *clock*. O que é esse circuito (o que ele faz)?



6. Seja o circuito a seguir.

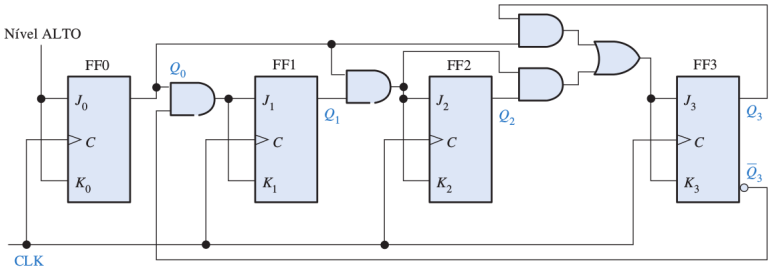


Os *flip-flops* desse circuito são todos disparados na subida do sinal que alimenta a entrada C (que no caso do primeiro *flip-flop* é o sinal de *clock*). Vamos supor que em todos os *flip-flops* há um pequeníssimo atraso (intervalo de tempo) entre o momento da subida do sinal que alimenta C e a estabilização do estado Q. Considerando essa condição, e que o atraso é de apenas uma fração do pulso do *clock*, desenhe o diagrama temporal da

simulação do circuito por 8 períodos do *clock*. Suponha que todos os *flip-flops* estão em zero inicialmente.

7. O circuito do exercício anterior é o de um contador assíncrono. Confira as saídas $Q_2\,Q_1\,Q_0$ e certifique-se de que realmente é um contador incremental. Note, porém, que devido aos atrasos mencionados no exercício anterior, formam-se brevíssimos estados $Q_2\,Q_1\,Q_0$ transitórios. Escreva a sequência dos estados do circuito acima, indicando quais deles são os transitórios.

8. Um circuito contador de décadas é tal que os estados variam de 0000 a 1001. Após o estado 1001, em vez de passar para o estado 1010, ele passa para 0000. Desenhe o diagrama temporal do circuito abaixo para cinco pulsos de *clock*. Suponha que o estado inicial é 0110.



9. Qual é a relação entre o número de *bits* utilizado para representar um endereço e a memória RAM?

10. Duas operações são frequentemente executadas com relação à memória RAM de um computador: dados (ou instruções) são lidos ou escritos em posições específicas. Considerando a RAM como um componente "caixa-preta":

    (a) Como indicamos a operação a ser executada (escrita ou leitura)?

    (b) O que são esperados nas portas de dados e de endereço no caso de uma operação de leitura?

    (c) Idem, no caso de uma operação de escrita?

Capítulo 10

Análise e projeto de circuitos sequenciais

Última atualização em 25 de junho de 2020

(Este capítulo ainda está meio pobre e assim poderá permanecer por enquanto ...)

Neste capítulo o objetivo é a análise e projeto de circuitos sequenciais síncronos, aqueles nos quais todos os elementos de memória – i.e., *flip-flops* – mudam de estado de forma sincronizada com um pulso de *clock*. Inicialmente, na parte de análise, examinaremos como proceder para se entender o funcionamento desses circuitos, por um método que não seja apenas por via de simulações e seus diagramas temporais. Em seguida, na parte de projeto, estudaremos como projetar circuitos desse tipo a partir de uma descrição funcional.

Para recordar as diferenças entre circuitos combinacionais e sequenciais, veja a seção 8.1.

No caso de circuitos combinacionais, a análise pode ser feita escrevendo-se primeiramente a expressão algébrica que corresponde ao desenho do circuito. A partir da expressão booleana, pode-se então gerar a tabela-verdade que define de forma explícita a relação entrada-saída. Por outro lado, o projeto de circuitos pode ser abordado a partir da especificação da relação entrada-saída dada pela tabela-verdade. O procedimento comum é a construção da expressão na forma canônica, seguida de sua minimização, e "tradução" direta da expressão simplificada para o desenho de um circuito.

No caso de circuitos sequenciais, a ideia é a mesma. No processo de análise, inicialmente determinamos as equações que definem a entrada de cada um dos *flip-flops*, e em função delas a equação

do próximo estado. Essas equações são suficientes para descrevermos a transição de estados, que quando mostradas de forma visual por meio de um diagrama permitem um entendimento do funcionamento do circuito. No processo de projeto, segue-se o caminho inverso, isto é, parte-se de uma descrição funcional (uma descrição sobre como deve funcionar o circuito) para se chegar ao circuito propriamente. Uma descrição funcional pode ser dada, por exemplo, por meio de um diagrama de estados e a partir deste pode-se determinar as transições de estados e as entradas em cada um dos *flip-flops* para que tais transições ocorram. Tendo-se essas equações, pode-se desenhar o circuito. Devido ao conceito de estado, o projeto de circuitos sequenciais não é tão direto como o de circuitos combinacionais. Os processos de análise e projeto serão examinados em datelhes a seguir por meio de exemplos.

10.1 Análise de circuitos sequenciais

No processo de análise, dado o desenho de um circuito, desejamos entender o seu funcionamento. O processo envolve os seguintes passos:

1. Determinar a equação que descreve o sinal nas entradas de cada *flip-flop* e também nas saídas do circuito
2. Determinar a equação que descreve o próximo estado de cada *flip-flop*
3. Determinar a tabela de transição de estados
4. Desenhar o diagrama de transição de estados

Exemplo 10.1.1. *Circuito do contador (de 2* bits*) módulo 4: este circuito foi apresentado na seção 9.3.*

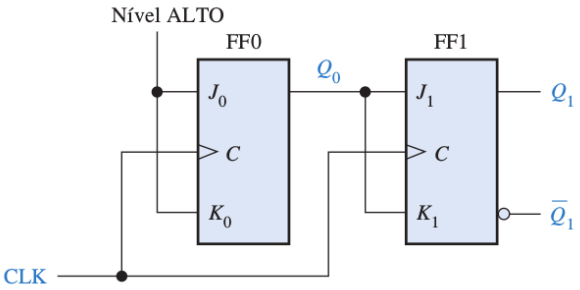


Figura 10.1: Contador síncrono de 2 *bits*.

1. *Equação das entradas dos* flip-flops

   $J_0 = K_0 = 1$
   $J_1 = K_1 = Q_0$

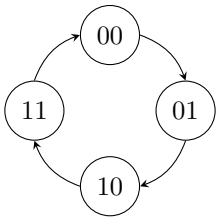
2. *Equação do próximo estado (Lembre que* $Q^* = J\,\overline{Q} + \overline{K}\,Q$*)*

   $Q_0^* = \overline{Q_0}$ *(já que* $J_0 = K_0 = 1$*)*
   $Q_1^* = J_1\,\overline{Q_1} + \overline{K_1}\,Q_1 = Q_0\,\overline{Q_1} + \overline{Q_0}\,Q_1 = Q_0 \oplus Q_1$

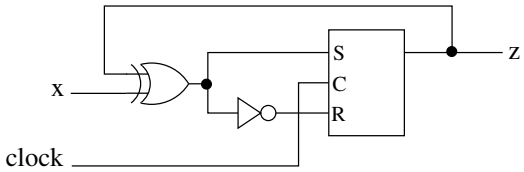
3. *Tabela de transição de estados: baseado nas equações do item anterior,*

Estado atual $(Q_1\,Q_0)$	*Próximo estado* $Q_1^*\,Q_0^*$
00	*01*
01	*10*
10	*11*
11	*00*

4. *Diagrama de transição de estados*



Exemplo 10.1.2. *Circuito verificador de paridade: este circuito foi apresentado na seção 9.2.*



1. *Equação das entradas dos* flip-flops

   $S = Q \oplus x$
   $R = \overline{S}$

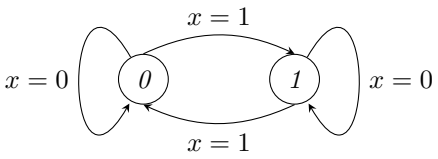
2. *Equação do próximo estado (lembre que* $Q^* = S + Q\overline{R}$*)*

   $Q^* = S + Q\overline{R} = S + Q\,\overline{\overline{S}} = S = Q \oplus x$

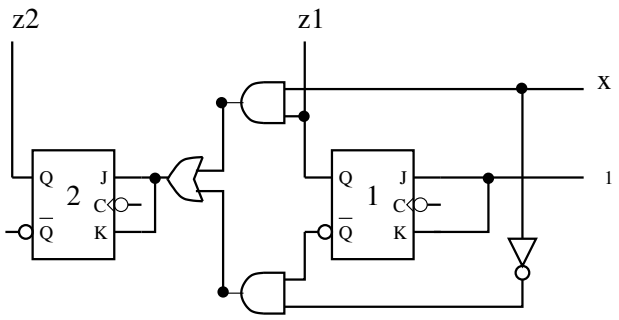
3. Tabela de transição de estados

Estado atual (*Q*)	*Próximo estado* (Q^*)	
	$x = 0$	$x = 1$
0	*0*	*1*
1	*1*	*0*

4. Diagrama de transição de estados



Exemplo 10.1.3. *Circuito contador* up-down



Refazer esta figura

1. Equação das entradas dos flip-flops

$J_1 = K_1 = 1$

$J_2 = K_2 = x\,Q_1 + \overline{x}\,\overline{Q_1}$

2. Equação da saída e do próximo estado

$z_1 = Q_1$

$z_2 = Q_2$

$Q_1^* = 1\,\overline{Q_1} + \overline{1}\,Q_1 = \overline{Q_1}$

$Q_2^* = (x\,Q_1 + \overline{x}\,\overline{Q_1})\,\overline{Q_2} + \overline{(x\,Q_1 + \overline{x}\,\overline{Q_1})}\,Q_2$

$$Q_2^* = \begin{cases} \overline{Q_1}\,\overline{Q_2} + Q_1\,Q_2, & \textit{se} \quad x = 0, \\ Q_1\,\overline{Q_2} + \overline{Q_1}\,Q_2, & \textit{se} \quad x = 1. \end{cases}$$

3. *Tabela de transição de estados: as saídas foram omitidas na tabela já que* $z_1 = Q_1$ *e* $z_2 = Q_2$

Estado atual $(Q_1\,Q_2)$	$Q_1^*\,Q_2^*$	
	$x=0$	$x=1$
00	*11*	*10*
01	*10*	*11*
10	*00*	*01*
11	*01*	*00*

Vamos agora colocar o estado na forma $Q_2\,Q_1$*:*

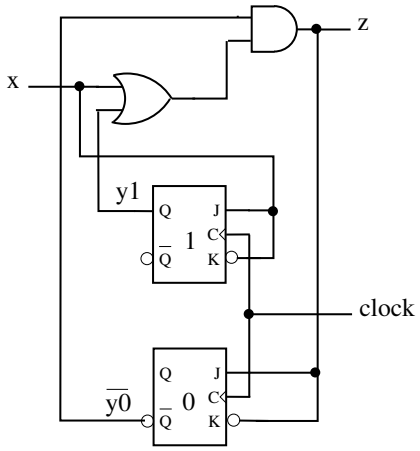
Estado atual $(Q_2\,Q_1)$	$Q_2^*\,Q_1^*$	
	$x=0$	$x=1$
00	*11*	*01*
01	*00*	*10*
10	*01*	*11*
11	*10*	*00*

4. *Diagrama de transição de estados*

Desenhe o diagrama de estados

Exemplo 10.1.4. *1. Equação das entradas dos* flip-flops

$J_1 = x$
$K_1 = \overline{x}$
$J_0 = z = (x + Q_1)\,\overline{Q_0}$
$K_0 = \overline{z} = \overline{(x + Q_1)\,\overline{Q_0}}$



(notação usada aqui: $y_0 = Q_0$ *e* $y_1 = Q_1$*)*

2. *Equação da saída e dos próximos estados: (Lembre que* $Q^* = J\,\overline{Q} + \overline{K}\,Q$*)*

   $z = (x + Q_1)\,\overline{Q_0}$

   $Q_0^* = [(x + Q_1)\,\overline{Q_0}]\,\overline{Q_0} + \overline{\overline{[(x + Q_1)\,\overline{Q_0}]}}\,Q_0 = (x + Q_1)\,\overline{Q_0}$

   $Q_1^* = x\,\overline{Q_1} + \overline{\overline{x}}\,Q_1 = x$

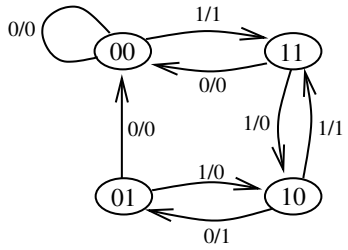
3. *Tabela de transição de estados*

   Cada célula da tabela representa o próximo estado e a saída $((Y_1\,Y_0)/z)$*.*

Estado atual $(Q_1\,Q_0)$	$Q_1^*\,Q_0^*/z$	
	$x = 0$	$x = 1$
00	*00/0*	*11/1*
01	*00/0*	*10/0*
10	*01/1*	*11/1*
11	*00/0*	*10/0*

4. *Diagrama de transição de estados*

   Cada nó representa um estado do sistema. Há uma aresta de um estado para outro se é possível uma transição de um para o outro. O rótulo nas arestas indica entrada x *e saída* z *(leia-se* $x\,/\,z$*). Como há apenas uma variável de entrada, que pode tomar os valores ou* 0 *ou* 1*, então há exatamente* 2 *arestas que saem de cada nó.*



10.2 Projeto de circuitos sequenciais

Projeto de circuitos sequenciais é um processo inverso ao da análise. No entanto, o ponto de partida em geral não é uma tabela ou diagrama de estados, e sim uma descrição funcional do circuito. A partir da descrição funcional pode ser elaborado um diagrama de transição de estados ou uma tabela de transição de estados. Algumas referências para este assunto: [Floyd, 2007] (seção 8.4 Projeto de contadores síncronos), [Hill and Peterson, 1993] (capítulo 10).

As etapas que fazem parte de projeto de circuitos sequenciais são:

- Descrição funcional
- Tabela de estados (que pode ser obtida a partir do diagrama de estados ou não)
- Tabela minimal de estados
- Tabela de transição
- Equação das entradas dos *flip-flops*
- Circuito

Esses conceitos serão abordados por meio de exemplos.

Exemplo 10.2.1. *O primeiro exemplo explorado para ilustrar o processo de análise de circuitos foi o contador síncrono de* 2 bits*. Aqui vamos fazer o processo inverso, i.e., projetar o circuito partindo do diagrama de transição de estados.*

Os estados possíveis no caso são 00, 01, 10 e 11. A transição desses estados deve ser cíclica, isto é,

```
00 --> 01 --> 10 --> 11 --> 00
```

Como temos duas variáveis de estado, vamos utilizar dois flip-flops *JK. Precisamos entender então quais valores de J e K fazem as transições ocorrerem como gostaríamos.*

Primeiro observemos quais são os possíveis valores de J e K que fazem o estado Q de um flip-flop passar para Q^*:

$Q \to Q^*$	J	K
$0 \to 0$	*0*	$\times$
$0 \to 1$	*1*	$\times$
$1 \to 0$	$\times$	*1*
$1 \to 1$	$\times$	*0*

A tabela acima mostra que, por exemplo, para ocorrer a transição $0 \to 0$ (em $Q \to Q^*$), o J deve ser 0 e o K pode ser tanto 0 quanto 1. Por quê ? Se $Q = 0$ e $Q^* = 0$, isso significa que ou o estado permaneceu como estava ($J = K = 0$) ou foi "resetado" ($J = 0$ e $K = 1$).

A sequência de transições `00 --> 01 --> 10 --> 11 --> 00` pode ser descrita conforme a tabela a seguir:

```
Q1 Q0   |  Q1*  Q0*
```

```
----------+-----------
    0 0   |   0 1
    0 1   |   1 0
    1 0   |   1 1
    1 1   |   0 0
```

Agora vamos deduzir a expressão do J e K de cada um dos flip-flops*. Como temos dois* flip-flops*, denotemos por J_1 e K_1 as entradas e por Q_1 o estado do* flip-flop *1, e por J_0 e K_0 as entradas e por Q_0 o estado do* flip-flop *0. Note que as expressões que descrevem essas entradas dependem apenas das variáveis de estado Q_0 e Q_1, já que não temos outras variáveis no circuito.*

Como podemos definir a expressão para as entradas J_1 e K_1? Vamos primeiramente isolar Q_1^* na tabela de transição de estados. Teremos:

```
  Q1 Q0  |  Q1*
---------+-----
    0 0  |   0
    0 1  |   1
    1 0  |   1
    1 1  |   0
```

Podemos ver que quando $Q_1Q_0 = 00$, $Q_1^* = 0$. Neste caso, considerando apenas o flip-flop *1, a transição $Q_1 \to Q_1^*$ que temos é $0 \to 0$. Tal transição pode ocorrer quando $J_1 = 0$ e $K_1 = X$. Podemos aplicar o mesmo raciocínio para as demais linhas da tabela. Por exemplo, quando $Q_1Q_0 = 01$, Q_1 vai de 0 para 1 (e isto ocorre quando $J_1 = 1$ e $K_1 = X$); e assim por diante. Assim, para cada estado Q_1Q_0 podemos definir J_1 e K_1 e portanto podemos pensar ambas como funções dependentes das variáveis Q_1Q_0, i.e., $J_1 = J_1(Q_1Q_0)$ e $K_1 = K_1(Q_1Q_0)$.*

Para obter a expressão de J_1 e de K_1, criamos um mapa de Karnaugh para cada um, no qual as variáveis são Q_1Q_0 e os valores do mapa são os valores de $J_1(Q_1Q_0)$ e $K_1(Q_1Q_0)$, respectivamente.

```
     Mapa do J1                  Mapa do K1
    \ Q0                        \ Q0
Q1 \    0       1           Q1 \    0       1
    +-------+-------           +-------+-------
 0  |   0   |   1            0  |   X   |   X
    +-------+-------           +-------+-------
 1  |   X   |   X            1  |   0   |   1
```

A expressão de J_1 obtida a partir da minimização por mapa de Karnaugh é, portanto, $J_1 = Q_0$. Similarmente, temos também $K_1 = Q_0$.

Repetindo o mesmo processo para J_0 e K_0, temos:

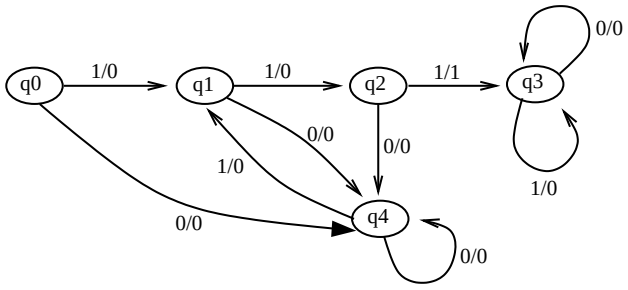
```
    Mapa do J0                  Mapa do K0
   \ Q0                        \ Q0
Q1 \    0       1           Q1 \    0       1
    +-------+-------            +-------+-------
 0  |   1   |   X            0  |   X   |   1
    +-------+-------            +-------+-------
 1  |   1   |   X            1  |   X   |   1
```

Após a minimização, temos $J_0 = K_0 = 1$.

Obtidas as expressões de J e K dos flip-flops*, o circuito pode ser desenhado diretamente a partir das expressões. Compare as expressões obtidas aqui com as obtidas no processo de análise (veja também a figura 10.1. Ao se aplicar o processo de análise sobre o circuito, deveremos chegar ao diagrama de transição de estados que corresponde às transições `00 --> 01 --> 10 --> 11 --> 00` que foi o ponto de partida.*

Exemplo 10.2.2. *Este é um exemplo um pouco mais elaborado. Trata-se de um detector de início de mensagem. Considere uma linha de transmissão de sinal, denotado por x, sincronizada com o* clock*. Uma ocorrência de* 3 bits 1 *consecutivos é considerado início de mensagem. Desejamos projetar um circuito síncrono que detecta um início de mensagem. Suponha que existe algum mecanismo que coloca o sistema detector de início de mensagem em um estado q_0 a cada final de mensagem e suponha que incialmente o sistema encontra-se no estado q_0.*

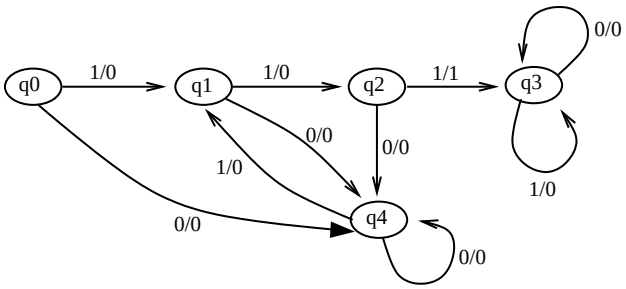
a) **Diagrama de estados***: uma possível forma de se começar o projeto de circuitos sequenciais é a construção de um diagrama de estados correspondente ao comportamento funcional desejado. Por exemplo:*



Nesse diagrama, a detecção de início de mensagem corresponde a atingir o estado q_3.

Neste exemplo os estados são denotados genericamente por q_i, diferentemente do exemplo anterior. No caso anterior, por se tratar de um circuito bem conhecido, sabíamos de antemão quantos estados seriam suficientes.

b) **Tabela de estados***: o diagrama acima (reproduzido abaixo por conveniência) pode ser equivalentemente representado pela tabela de estados abaixo:*



Estado atual	*Próx.estado/saída*	
	$x = 0$	$x = 1$
q_0	$q_4/0$	$q_1/0$
q_1	$q_4/0$	$q_2/0$
q_2	$q_4/0$	$q_3/1$
q_3	$q_3/0$	$q_3/0$
q_4	$q_4/0$	$q_1/0$

Note que a saída é 1 apenas quando o estado q_3 é atingido.

c) **Tabela minimal de estados***: na tabela de estados acima, o estado q_0 é equivalente ao estado q_4. Isso poderia ser percebido até no próprio diagrama de estados. No entanto, em geral nem sempre o diagrama de estados é gerado e, além disso, a equivalência de estados pode não ser tão óbvia. De qualquer forma, nesta etapa reduz-se a tabela de estados a uma tabela minimal, ou seja, eliminam-se os estados equivalentes. Para não gerar confusão na identificação dos estados, na tabela minimal de estados é aconselhável a utilização de outros nomes para os estados. Desta forma, em vez da notação q_i para os estados, passaremos a utilizar as letras a, b, c, d. Fazendo $a = q_4$, $b = q_2$, $c = q_1$ e $d = q_3$ temos:*

Estado	*Prox.estado/saída*	
	$x = 0$	$x = 1$
a	$a/0$	$c/0$
b	$a/0$	$d/1$
c	$a/0$	$b/0$
d	$d/0$	$d/0$

d) **Associação de estados***: se o número de estados na tabela minimal de estados é m, então serão necessários r*flip-flops*, onde r é tal que $2^{r-1} < m \leq 2^r$, para que o sistema seja capaz de representar esses m estados.*

O problema de associação de estados consiste em definir qual das 2^r combinações de valores binários será utilizado para representar cada um dos estados do sistema. Note que não consideramos isso no exemplo anterior, pois naquele caso os valores de estados estavam bem definidos. No exemplo que estamos considerando, como são 4 estados então são necessários $r = 2$ flip-flops *e existem as três seguintes possíveis associações:*

Estados	*Associação*		
	1	*2*	*3*
a	*00*	*00*	*00*
b	*01*	*11*	*10*
c	*11*	*01*	*01*
d	*10*	*10*	*11*

As demais associações são equivalentes a um desses três no sentido de que correspondem a uma rotação vertical ou à complementação de uma ou ambas as variáveis e, portanto, em termos de circuito resultante teriam o mesmo custo.

e) **Tabelas de transição de estados***: para cada uma das associações consideradas, pode-se gerar uma tabela de transição de esatdos. Em cada uma das tabelas de transição abaixo, as atribuições de estado estão listadas seguindo a ordem do gray-code (*$00 - 01 - 11 - 10$*).*

Associação 1			
Estado	$Q_1\,Q_0$	$Q_1^*\,Q_0^*$	
		$x=0$	$x=1$
a	*00*	*00*	*11*
b	*01*	*00*	*10*
c	*11*	*00*	*01*
d	*10*	*10*	*10*

Associação 2			
Estado	$Q_1\,Q_0$	$Q_1^*\,Q_0^*$	
		$x=0$	$x=1$
a	*00*	*00*	*01*
c	*01*	*00*	*11*
b	*11*	*00*	*10*
d	*10*	*10*	*10*

Associação 3			
Estado	$Q_1\,Q_0$	$Q_1^*\,Q_0^*$	
		$x=0$	$x=1$
a	*00*	*00*	*01*
c	*01*	*00*	*10*
d	*11*	*11*	*11*
b	*10*	*00*	*11*

f) **Equação das entradas dos *flip-flops*:** *a equação das entradas dos* flip-flops *pode ser gerada a partir da análise das tabelas de transições, de forma análoga ao feito no exemplo 1.*

Iremos descrever esse processo novamente, considerando a associação 3 do item anterior. Primeiramente, observe que sabemos que do estado $Q_1\,Q_0$ o sistema irá para o estado $Q_1^*\,Q_0^*$ e que cada variável de estado (no caso, Q_1 e Q_0) corresponde a um flip-flop*. Assim, o que queremos descobrir é a expressão que descreve o sinal de entrada desses* flip-flops *para que a transição (mudança de estado) desejada ocorra.*

Vamos analisar inicialmente o estado Q_1. Suponha que usaremos flip-flops *JK neste circuito. Então, qual deve ser o valor de J_1 e K_1 para que a transição $Q_1 \to Q_1^*$ ocorra ? Para isso, recordemos a tabela do* flip-flop *JK. As tabelas abaixo descrevem, novamente, o comportamento do* flip-flop *JK (à esquerda) e as condições nas quais ocorre a transição $Q \to Q^*$ (à direita). O símbolo $\times$ indica uma entrada don't care.*

JK	Q^*	
	$Q=0$	$Q=1$
00	*0*	*1*
01	*0*	*0*
10	*1*	*1*
11	*1*	*0*

$Q \to Q^*$	J	K
$0 \to 0$	*0*	$\times$
$0 \to 1$	*1*	$\times$
$1 \to 0$	$\times$	*1*
$1 \to 1$	$\times$	*0*

Abaixo mostramos o mapa de J_1 e K_1 (em forma tabular), em função de Q_1Q_0 e x:

Associação 3 Q_1Q_0	Q_1	Q_1^*		J_1		K_1	
		$x=0$	$x=1$	$x=0$	$x=1$	$x=0$	$x=1$
a 00	*0*	*0*	*0*	*0*	*0*	$\times$	$\times$
c 01	*0*	*0*	*1*	*0*	*1*	$\times$	$\times$
d 11	*1*	*1*	*1*	$\times$	$\times$	*0*	*0*
b 10	*1*	*0*	*1*	$\times$	$\times$	*1*	*0*

As colunas Q_1 e Q_1^* estão destacadas para evidenciar as transições (e, portanto, definir os possíveis valores de J_1 e K_1 que acarretam essas transições). Na primeira linha é indicada a transição de $Q_1 = 0$ para $Q_1^* = 0$ tanto quando $x = 0$ como quando $x = 1$. Essa transição ocorre quando $J_1 = 0$, independente do valor de K_1. Na segunda linha, quando $x = 0$, $Q_1 = 0$ se mantém (i.e., $Q_1^* = 0$). Mas quando $x = 1$, $Q_1 = 0$ muda para $Q_1^* = 1$ e isso ocorre quando $J_1 = 1$, independente do valor de K_1. E assim por diante.

Podemos agora fazer a minimização (mapa de Karnaugh) como feito no exemplo 1. No caso de J_1, o único 1 na coluna $x = 1$ pode ser agrupado com o don't care $\times$ logo abaixo dele e assim obtemos $J_1 = x\,Q_0$. No mapa de K_1, o também único 1 na coluna $x = 0$ pode ser agrupado com o don't care $\times$ logo abaixo dele e assim obtemos $K_1 = \overline{x}\,\overline{Q_0}$.

De forma análoga, repetimos o processo para Q_0. Restringindo a tabela de transição da associação 3 à variável Q_0, temos:

Q_0	Q_0^*	
	$x=0$	$x=1$
a 0	*0*	*1*
c 1	*0*	*0*
d 1	*1*	*1*
b 0	*0*	*1*

Portanto, as tabelas para J_0 e K_0 serão respectivamente

Q_1Q_0	J_0	
	$x=0$	$x=1$
00	*0*	*1*
01	$\times$	$\times$
11	$\times$	$\times$
10	*0*	*1*

Q_1Q_0	K_0	
	$x=0$	$x=1$
00	$\times$	$\times$
01	*1*	*1*
11	*0*	*0*
10	$\times$	$\times$

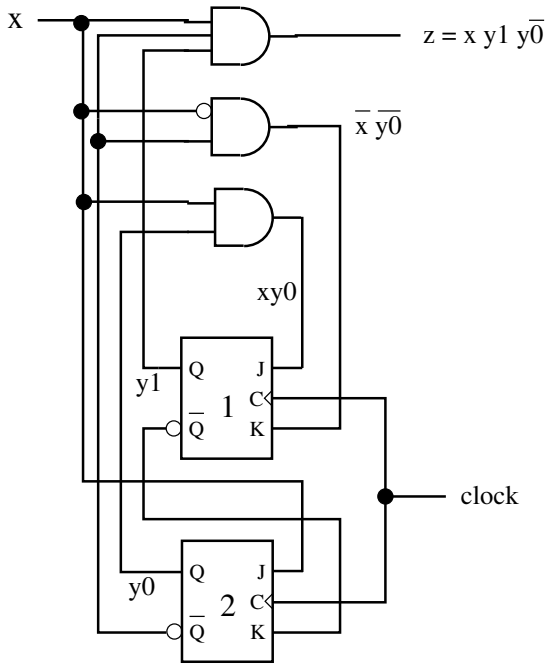
De onde obtemos que $J_0 = x$ e $K_0 = \overline{Q_1}$.

Além disso, a expressão para a saída z é dada por $x\, y_1\, \overline{Q_0}$ (pois existe uma única situação em que a saída do circuito é 1; justamente quando ele se encontra no estado b e a entrada x é $= 1$. A expressão segue do fato de termos associado ao estado b o par $Q_1Q_0 = 10$).

Procedimento similar pode ser aplicado para as associações 1 e 2. A associação 1 resulta em um circuito de custo (em termos de número total de portas lógicas) equivalente ao da associação 3 e a associação 2 resulta em um circuito de custo ligeiramente maior.

g) **O circuito!**

As equações obtidas para a associação 3 correspondem ao seguinte circuito.



(refazer essa figura ...)

10.3 Máquinas de Mealy e Moore

Os circuitos sequenciais podem ser vistos como máquinas de estado finito (*finite state-machine*). Máquinas de estado finito são modelos abstratos usados em computação para modelar circuitos e programas em geral. Dois tipos de máquinas são as máquinas de Mealy e Moore, conforme definidas a seguir.

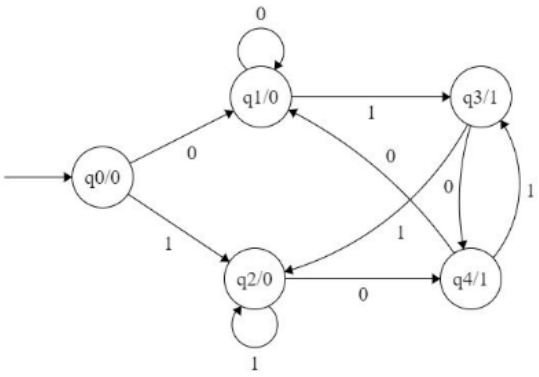


Figura 10.2: Exemplo de um máquina de estados do tipo Moore. (Fonte: https://www.geeksforgeeks.org/mealy-and-moore-machines-in-toc/)

Máquinas de Moore: São máquinas de estado finito sem entradas. As saídas dependem apenas do estado presente. Formalmente podem ser definidas por uma sêxtupla $(Q, q_0, \Sigma, \mathrm{O}, \delta, \lambda)$ na qual:

- Q é um conjunto finito de estados
- q_0 é o estado inicial
- Σ é o alfabeto de entrada
- O é o alfabeto de saída
- δ é a função de transição, i.e., $\delta : Q \times \Sigma \to Q$
- λ é a função de saída, i.e., $\lambda : Q \to O$

Um exemplo de uma máquina de estado do tipo Moore é mostrado na figura 10.2. No diagrama, os estados são representados pelos círculos. As arestas indicam a transição de estados de acordo com a entrada. A saída da máquina é representada junto aos estados, após uma / (barra). O esatdo q_0 é o estado inicial. Os alfabetos de entrada e de saída são binários (isto é, $\Sigma = \mathrm{O} = \{0, 1\}$).

Máquinas de Mealy: são máquinas de estado nas quais a saída depende do estado e entrada correntes. Elas podem também ser definidas por uma sêxtupla $(Q, q_0, \Sigma, \mathrm{O}, \delta, \lambda')$ na qual:

- Q é um conjunto finito de estados
- q_0 é o estado inicial
- Σ é o alfabeto de entrada
- O é o alfabeto de saída

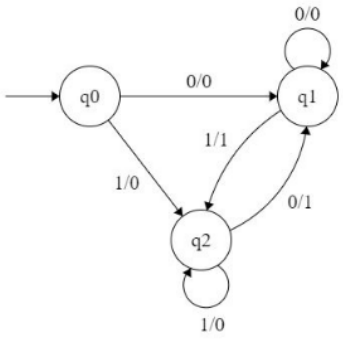


Figura 10.3: Exemplo de um máquina de estados do tipo Mealy. (Fonte: https://www.geeksforgeeks.org/mealy-and-moore-machines-in-toc/)

- δ é a função de transição, i.e., $\delta : Q \times \Sigma \to Q$
- λ' é a função de saída, i.e., $\lambda : Q \times \Sigma \to O$

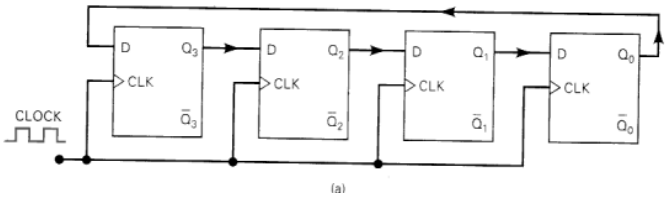
Um exemplo de uma máquina de estado do tipo Mealy é mostrado na figura 10.3. A saída da máquina é representada junto às entradas nas transições, após uma / (barra). Os demais elementos são análogos às máquinas Moore.

Como pode ser visto, a única diferença é que a saída de uma máquina de Mealy depende tanto do estado como entrada correntes, enquanto a saída de uma máquina de Moore depende apenas do estado corrente. Isso em particular significa que uma mudança na saída de uma máquina de Mealy pode acontecer a qualquer instante (sempre que há alguma alteração na entrada), enquanto que na máquina de Moore uma mudança na saída só ocorre quando há um pulso de *clock* (que poderá alterar o estado).

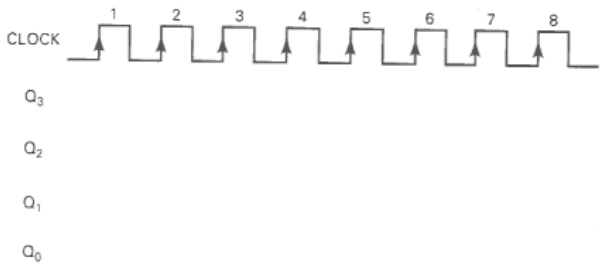
Uma máquina de Mealy pode ser convertida para uma máquina de Moore. Para isso, para cada estado da máquina de Mealy que possui duas saídas associadas deve-se criar dois estados representando um par estado/saída.

Exercícios

1. Seja o circuito a seguir. Note que os *flip-flops* D mudam de estado na subida do sinal de *clock*.



(a) Preencha o diagrama temporal a seguir



(b) Qual é o estado do circuito logo após o primeiro pulso do sinal de *clock* ?

(c) Desenhe o diagrama de transição de estados do circuito.

2. Um circuito sequencial tem 2 *flip-flops* JK, FF1 e FF2, duas entradas (x e y) e uma saída (z). As equações das entradas dos *flip-flops* e da saída do circuito são as seguintes:

$J_1 = Q_2\, x + \overline{Q_2}\, \overline{y}$

$K_1 = \overline{Q_2}\, x\, \overline{y}$

$J_2 = \overline{Q_1}\, x$

$k_2 = Q_1 + x\, \overline{y}$

$z = Q_1\, x\, y + Q_2\, \overline{x}\, \overline{y}$

(a) Desenhe o circuito.

(b) Escreva a tabela de transição de estados.

(c) Desenhe o diagrama de transição de estados.

3. Preencha a tabela-verdade do *flip-flop* JK (tabela da esquerda) e, na tabela da direita, preencha as colunas J e K com os valores (0, 1 ou $\times$) que resultam na transição de estados indicada na primeira coluna.

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

$Q \to Q^*$	J	K
$0 \to 0$		
$0 \to 1$		
$1 \to 0$		
$1 \to 1$		

4. Suponha a seguinte tabela de transição de estados

Estado atual			Próximo estado		
Q_2	Q_1	Q_0	Q_2^*	Q_1^*	Q_0^*
0	0	1	0	1	0
0	1	0	1	0	1
1	0	1	1	1	1
1	1	1	0	0	1

   (a) Desenhe o diagrama de transição de estados.

   (b) Quantas variáveis de estado estão presentes nesse diagrama ? Quantos *flip-flops* são necessários para se implementar um circuito que tenha o comportamento acima?

   (c) O circuito correspondente ao diagrama tem entradas ou saídas ?

   (d) Supondo que serão usados *flip-flops* do tipo JK, escreva a expressão booleana que determina as entradas J e K de cada um deles.

   (e) Desenhe o circuito correspondente.

Capítulo 11

Organização de computadores

Última atualização: 27/06/2023

Neste capítulo descrevemos um pouco sobre a organização de um computador, no nível de circuitos lógicos, com um olhar voltado para a parte relacionada à execução de um programa. É importante observar que o modelo aqui descrito é bastante simplificado e não pretende refletir nenhum modelo comercial específico disponível.

Um programa, antes de ser executado, precisa ser convertido para a linguagem de máquina do computador no qual ele será executado. A linguagem de máquina corresponde ao conjunto de instruções que podem ser executadas pelo computador, denominadas de instruções de máquina. Em geral, cada modelo de processador possui seu conjunto de instruções de máquina. Há dois conceitos relacionados a essa conversão. Um deles é **compilação** e refere-se ao processo de "traduzir" um programa escrito em uma linguagem de alto nível para um programa em linguagem de máquina. Um compilador recebe um programa-fonte (por exemplo, um programa em linguagem C) e gera um arquivo com o programa em linguagem de máquina, comumente chamados de programa binário ou ainda programa executável. Um programa binário é portanto uma sequência de instruções de máquina. A execução do programa consiste na execução sequencial[1] dessas instruções. Já no caso de **interpretadores**, as instruções de máquina são geradas e enviadas ao processador uma a uma durante a execução do programa, não havendo necessariamente a geração de um arquivo com o código binário. Em geral, os interpretadores atuam sobre uma linguagem intermediária (de baixo nível) não dependente de especificidades do *hardware*. Por exemplo, no caso da linguagem Python, ao se executar um programa, ocorre inicialmente um processo de conversão do programa-fonte para uma linguagem intermediária denominada *bytecode*. Em seguida, um interpertador Python converte os *bytecodes* em instruções de máquina,

[1] O termo sequencial aqui é utilizado em contraposição ao termo paralelo, para indicar que uma instrução é executada somente quando a execução da outra já foi concluída.

que são executadas pelo processador. A cada vez que esse mesmo programa é executado ocorre a conversão dos *bytecodes* para as respectivas instruções de máquina.

Quando um computador é ligado, alguns processamentos programados em *hardware* são executados. Entre esses processamentos está a ação de "carregar" e colocar em execução um sistema operacional. O sistema operacional (SO) é o programa que faz o gerenciamento do uso de recursos do computador, ou seja, garante que aplicativos (outros softwares) sejam executados em harmonia. Basicamente, cabe ao SO "carregar" para a memória do computador as instruções correspondentes aos diferentes programas a serem executados e colocar esses programas em uma fila de execução. Após um certo número de instruções de um programa ter sido executado, o SO trata de "interferir" para que a próxima instrução a ser executada seja do próximo programa na fila de execução. Desta forma, os vários programas na fila de execução são executados em esquema de revezamento, várias vezes, até que terminem. O ciclo de revezamento é tão breve que o usuário não percebe que o seu programa pode ter "parado" várias vezes.

Estas descrições são visões simplificadas de como os processamentos ocorrem na prática. Vários detalhes adicionais precisariam sem considerados para um entendimento preciso, uma vez que os computadores modernos possuem múltiplos núcleos de processamento, placas de processamento gráfico, entre outros elementos. Neste sentido, este capítulo é restrito a proporcionar uma visão geral, mostrando como a execução de um programa ocorre em um modelo simples.

Na figura a seguir é mostrado um esquema de hierarquia que ilustra diferentes "camadas" entre um usuário e o *hardware*:

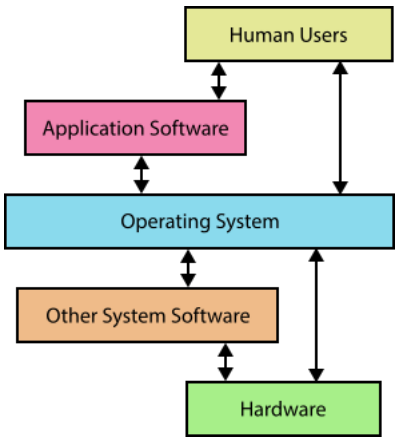


Imagem extraída de https://en.wikibooks.org/wiki/IB/Group_4/Computer_Science/Computer_Organisation

Cada uma dessas camadas tem papel importante no funcionamento/uso de um computador. Neste capítulo, a camada de interesse é a de *hardware*. Portanto, não iremos estudar como um programa escrito em linguagem de alto nível é convertido para uma sequência de instruções de máquina, nem como sua execução é gerenciada pelo SO. Estudaremos os aspectos organizacionais

do processador, relacionadas à execução de instruções de máquina, sob o ponto de vista lógico. Mais especificamente, examinaremos a organização dos circuitos lógicos para a execução de instruções de máquina.

Na literatura da área, utilizam-se os termos "organização de computadores" e "arquitetura de computadores" para se fazer referência aos aspectos relacionados à organização e funcionamento de computadores. De acordo com discussões presentes em alguns fórums, a distinção entre eles não é muito clara. Um dos entendimentos existentes é o seguinte:

- **Arquitetura de computadores**: refere-se a aspectos de desenho de um computador e especifica como é o funcionamento do *hardware* (conjunto de instruções, codificação das instruções, tipo de dados, forma de endereçamento, etc). Estabelece, portanto, um padrão que permite a comunicação entre as camadas superiores (por exemplo, sistemas operacionais) e o *hardware*. Os fabricantes de *hardware* implementam uma dada arquitetura; um sistema (*software*) que adere a essa arquitetura não depende dos detalhes de implementação.
- **Organização de computadores**: refere-se aos aspectos físicos do computador como o desenho dos circuitos, o tipo de memória, etc. São os aspectos que acabam influenciando custo, área do *chip*, eficiência, etc. A implementação de uma dada arquitetura pode considerar diferentes formas de organização de computadores.

Neste texto não nos preocuparemos em definir o que é arquitetura ou o que é organização, nem estudar arquiteturas reais. Estudaremos a organização de computadores usando um modelo de computação simples, com foco no estudo dos principais conceitos que fundamentam qualquer modelo de computação.

11.1 O modelo de computação de von Neumann

O precursor dos modelos de computadores atuais é o modelo proposto por *John von Neumann*. Um ponto notável no modelo de *von Neumann* é a revolucionária ideia de armazenar as instruções na memória do computador em vez de requerer alterações físicas no computador para cada tipo de instrução. Os componentes do modelo de *von Neumann* são mostrados na figura 11.1.

Memória É o módulo que armazena tanto instruções quanto dados. A memória, também conhecida por RAM (*Random Access Memory*) é uma coleção de palavras (múltiplos *bits* cada), identificadas por um endereço único. Podemos imaginar que essas palavras estão organizadas sequencialmente (linearmente), começando no endereço 0.

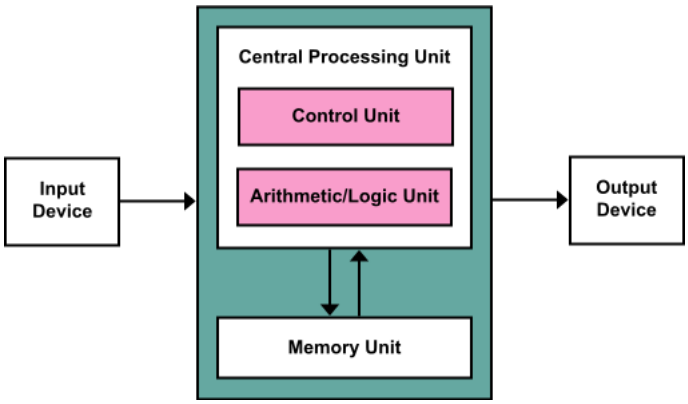


Figura 11.1: Componentes do modelo de von Neumann (Imagem extraída de https://en.wikibooks.org/wiki/IB/Group_4/Computer_Science/Computer_Organisation).

O tamanho da palavra (em termos de número de *bytes*) pode variar de computador para computador. São comuns palavras de 32 *bits* (4 *bytes*) ou de 64 *bits* (8 *bytes*). A magnitude de números inteiros que podem ser representados no computador é definida, portanto, pelo tamanho das palavras. Por exemplo, em um computador de 32 *bits*, pode-se representar 2^{32} padrões binários distintos em uma palavra (portanto, se considerarmos números inteiros sem sinal, podemos representar números de 0 a $2^{32} - 1$).

Os endereços são também representados com uma certa quantidade de *bits*; por exemplo, se consideramos o endereçamento de 8 *bits*, os endereços que podem ser representados variam de 0 a $2^8 - 1$. Muitos computadores atuais permitem endereçamento *byte* a *byte*. Outra possibilidade é o endereçamento palavra a palavra. O endereçamento está diretamente relacionado ao acesso à memória do computador, tanto para leitura como para escrita. Tipicamente, uma operação de leitura de uma posição x da memória permite que se tenha acesso aos k *bytes* da memória a partir da posição x, onde k é o número de *bytes* de uma palavra.

Resumindo, a quantidade de *bits* das palavras define a magnitude dos números que podem ser representados no computador, enquanto o número de *bits* usados para endereçamento define a quantidade máxima de posições de memória que podem ser endereçadas.

Na RAM podem ser armazenados tanto dados como instruções.

ULA Unidade lógico-aritmética: como já vimos, é o módulo responsável pela execução (cálculo) das operações aritméticas e lógicas.

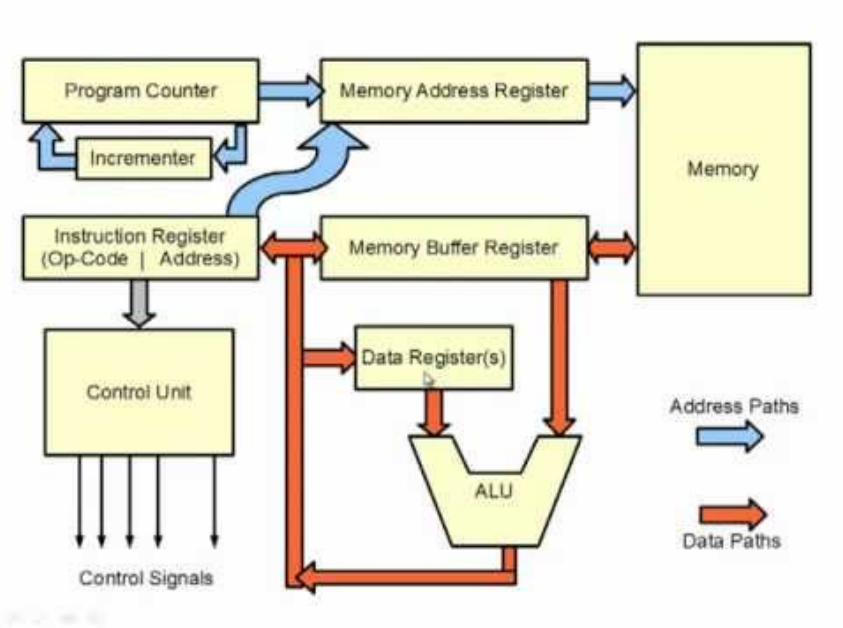
Unidades de entrada e saída São os módulos responsáveis por permitir que os dados "entrem" e "saiam" do computador. Sem esses módulos, não seria possível a um usuário "usar" um computador. Por exemplo, as ações executadas sobre um teclado ou com um *mouse* são apropriadamente tratados por esse módulo, gerando para cada ação uma representação interna adequada para o computador.

Unidade de controle Este é o módulo responsável por executar as instruções de um programa, em um processo conhecido por *Fetch-Decode-Execute Cycle* (isto é, buscar a próxima instrução a ser executada, decodificá-la e executá-la), que será detalhado mais adiante.

A unidade de controle e a ULA estão fortemente acopladas pois a execução de várias das instruções envolve a ULA. Assim, em geral usa-se o termo Unidade de Processamento Central para se referir ao conjunto "ULA + Unidade de controle".

Os módulos acima estão conectados por linhas/canais de comunicação denominados barramentos (*bus*, em inglês). Há basicamente três tipos de barramentos: *data bus* para transporte de dados, *address bus* para transporte de endereços e *control bus* para o transporte de sinais de controle (por exemplo, para controlar se o *data bus* está transportando da direção da memória para o processador ou vice-versa, se a memória está operando em modo leitura ou em modo escrita, etc). O *data bus* liga bidirecionalmente a CPU e a RAM, enquanto o *address bus* liga a CPU à memória. Esses canais de comunicação possuem a largura em *bits* para transportar, por exemplo, uma palavra ou um endereço.

Na figura 11.2 mostramos os principais componentes e os barramentos na organização de um processador simples.



(Fonte: *A more detailed look inside the CPU of a typical computer*, por Daniel Livingstone)

Figura 11.2: Organização de um processador simples, com destaque aos barramentos (de dados, de endereço e de sinais de controle).

Um programa a ser executado deve ser primeiramente carregado na memória RAM. Lá, suas instruções devem ocupar posições contíguas. Existe um registrador, denominado *program coun-*

ter (PC), um contador, que armazena o endereço da próxima instrução a ser executada. A cada instrução, o PC é incrementado em 1 e desta forma passa a "apontar" para a próxima instrução a ser executada. A instrução a ser executada é copiada da memória para um registrador especial denominado *Instruction register* (IR). O código da instrução é então usado para ajustar os sinais de controle de diferentes partes do processador. Por exemplo, se a instrução requer a escrita de algum dado na memória, o pino de controle Leitura/Escrita da memória precisa ser ajustado para Escrita, e a porta de endereço da memória RAM deve receber o endereço da posição na qual o dado será escrito (e não o endereço que está no PC). Outros registradores podem ser necessários dependendo da complexidade da instrução. No diagrama da figura 11.2, podem ser vistos outros três registradores, o MAR (*Memory Address Register*) e o MBR (*Memory Buffer Register*), e o *Data register*. Em nosso modelo, consideraremos um cenário mais simples, em que teremos apenas três registradores: o PC, o IR, e o ACC (este é uma espécie de combinação do MBR e do *Data Register*).

A execução de uma instrução é realizada em um ciclo de instrução (em inglês, *Instruction cycle* ou *Fetch-Decode-Execute Cycle*) do processador. A atualização de conteúdos da memória RAM e dos registradores é realizada de forma sincronizada, pelo sinal de um *clock*. A seguir são apresentados os detalhes do nosso modelo e explicamos em maiores detalhes como um ciclo de instrução funciona.

11.2 Modelo de um processador simples

Descrevemos aqui um modelo simples de processador, mais simples que o mostrado na figura 11.2. O modelo descrito aqui utiliza menos registradores. Ele pode ser implementado usando um simulador de circuitos lógicos como o `Logisim-evolution` (https://github.com/logisim-evolution/logisim-evolution).

Os detalhes a seguir são apresentados visando uma implementação. Iremos supor dados de 8 *bits*, endereço de memória também de 8 *bits*, e posições de memória de tamanho 16 *bits*.

Componentes e características do processador:

- ULA: realiza as operações de adição e subtração, bem como de comparação. A ULA opera com palavras de 8 *bits*.
- Instrução: uma instrução é formada por 16 *bits*. Os 8 *bits* mais significativos correspondem ao código da instrução e os 8 *bits* menos significativos correspondem ao endereço referenciado pela instrução (quando pertinente).

  As instruções que usaremos serão adaptadas do HIPO, um computador hipótetico. Uma

descrição do computador HIPO, incluindo o conjunto de instruções, pode ser encontrada em https://www.ime.usp.br/~jstern/software/hipo/Hipo.pdf.

O HIPO pode também ser explorado por meio do Computador a papel (https://www.ime.usp.br/~vwsetzer/comp-papel.html) para a introdução de conceitos relacionados à computação.

- Contador: o PC (*Program counter*) pode ser um contador. Vamos supor que a primeira instrução estará sempre na posição de endereço zero na RAM. Os endereços serão todos de 8 *bits*.

- Registradores: o acumulador (ACC) e o registrador de instruções (IR) são ambos registradores utilizados pela UC. O ACC é tipicamente utilizado para armazenamento temporário de dados, geralmente em operações que envolvem a ULA. Os dados serão todos de 8 *bits*. O IR (16 *bits*) armazena temporiamente a instrução que está sendo executada no ciclo de instrução atual. O código da instrução é usado para acertar os sinais que controlam diferentes partes do processador para que a instrução seja executada corretamente. Caso a instrução envolva um endereço, esse deve ser eviado para o barramento de endereços.

- RAM: Cada posição da memória RAM consiste de palavras de 16 *bits*, podendo conter um dado ou uma instrução. Se o conteúdo é um dado, iremos supor que estão armazenados nos 8 *bits* menos significativos[2]. Vamos considerar que os endereços são de 8 *bits* e, portanto, nossa memória terá 256 posições.

  O esquema de uma memória RAM, bem como as células binárias que a compõem, foram mostradas na seção 9.5.

11.2.1 Conjunto de instruções

Um conjunto de instruções de máquina é listado na tabela 11.1. O código é mostrado em decimal e em hexadecimal, juntamente com a sua descrição. Note que esses códigos são arbitrários. Quando há uma grande quantidade de instruções, pode-se definir os códigos de forma que o circuito decodificador de instruções fique mais simples (por exemplo de forma que certos *bits* do código possam ser usados diretamente como um sinal de controle). Como temos apenas poucas instruções, os códigos não foram definidos com essa preocupação.

AC refere-se ao registrador acumulador e [AC] ao seu conteúdo. EE é um endereço de 8 *bits* e [EE], de 16 *bits*, é o valor na posição EE da memória. A instrução Leia requer um dado de entrada e esse pode ser simulado utilizando um pino do tipo *input* no simulador. Similarmente a instrução Imprima requer o envio de um dado para uma saída e isso pode ser simulado utilizando

[2]Dado que as palavras na memória possuem 16 *bits*, poderíamos ter definido que todos os dados são representados em 16 *bits*. Porém como construímos a ULA de 8 *bits*, para simplificar suporemos que todos os dados são de 8 *bits*.

Tabela 11.1: Tabela de instruções para o modelo de CPU descrito neste capítulo

Código		Descrição
base 10	base 16	
00	00	NOP (no operation)
01	01	Copie [EE] para o AC
02	02	Copie [AC] para a posição de endereço EE
03	03	Some [EE] com [AC] e guarde o resultado em AC
04	04	Subtraia [EE] de [AC] e guarde o resultado em AC
07	07	Leia um número e guarde-o na posição de endereço EE
08	08	Imprima [EE]
09	09	Pare
10	0A	Desvie para EE (desvio incondicional)
11	0B	Desvie para EE se [AC] > 0
13	0D	Desvie para EE se [AC] $= 0$
15	0F	Desvie para EE se [AC] < 0

um pino do tipo *output* no simulador. Pode-se acrescentar *displays* para a visualização desses dados durante a simulação.

Note que todos os dados são de 8 *bits*; assim, uma vez que uma palavra da RAM é de 16 *bits*, quando um dado é copiado na RAM, deve ser construída uma palavra de 16 *bits*, sendo zero nos 8 *bits* mais significativos e o dado a ser copiado nos 8 *bits* menos significativos.

A instrução 00 (NOP) corresponde a fazer nada no passo 3 do ciclo FDX. A instrução 09 (Pare) deverá fazer com que o processador volte ao estado correspondente ao do início de execução de um programa. Isto é, deve zerar o PC (Program Counter) e voltar à configuração de início de um ciclo de execução.

11.2.2 Exemplo de um programa

Um exemplo de programa é a seguinte sequência de instruções, em notação hexadecimal:

```
0110 0211 0712 0812 0112 0F0a 0111 0312 0211 0a02 0811 0900
```

Vamos supor que essa sequência de instruções é carregada na RAM, a partir do endereço 00. Todos os endereços, assim como as instruções e dados, estão em notação hexadecimal. A seguir está a "tradução" do programa. Note também que a posição de memória 10 deve conter o valor `0000`, para que a simulação funcione corretamente.

Endereço	Instrução	Tradução
00	01 10	Copie [10] para o AC
01	02 11	Copie [AC] para a posição de endereço 11
02	07 12	Leia um número e guarde-o na posição de endereço 12
03	08 12	Imprima [12]
04	01 12	Copie [12] para o AC
05	0f 0a	desvie para 0a se [AC] < 0
06	01 11	Copie [11] para o AC
07	03 12	Some [AC] com [12] e guarde o resultado em AC
08	02 11	Copie [AC] para a posição de endereço 11
09	0a 02	Desvie para 02
0a	08 11	Imprima [11]
0b	09 00	Pare
...	...	...
10	00 00	dado (número 0)
11		espaço para armazenar dado
12		espaço para armazenar dado

Em linguagem um pouco mais alto nível, temos

```
10 ~ Zero
11 ~ Soma
12 ~ Num

     AC <-- Zero
     Soma <-- AC
Ler: Leia Num
     Imprima Num
     AC <-- Num
     Se AC < 0, desvie para Fim
     AC <-- Soma
     AC <-- AC + Num
     Soma <-- AC
     Desvie para Ler
Fim: Imprima Soma
     Pare
```

11.3 Ciclo de instrução (*Fetch-Decode-Execute Cycle*)

Chama-se ciclo de instrução (abreviado por FDX) o processo de execução de uma instrução de máquina no processador. Um ciclo desses, tipicamente consiste de uma sequência de 3 passos que são religiosamente executados, independentemente da instrução sendo executada. De forma geral, os passos são conforme descritos a seguir:

1. *Fetch*: busca da próxima instrução a ser executada (a qual encontra-se na memória RAM)

   - a instrução na posição apontada pelo PC deve ser lida da memória e carregada no IR. Além disso, o valor do PC deve ser incrementado em 1.

2. *Decode*: decodifica-se a instrução (determina-se as ações exigidas pela mesma)

   - o valor dos diversos sinais de controle deve ser ajustado conforme a instrução a ser executada (por exemplo, pode ser necessário definir o modo de operação – leitura/escrita – da memória e dos registradores, os sinais que controlam os pinos seletores dos MUXes, etc).

3. *Execute*: executa-se as ações determinadas pela instrução

   - Além disso, o ciclo deve ser "resetado".

Um ciclo de instrução é tipicamente executado em um número fixo de períodos do *clock*. No nosso modelo simplificado, dos passos acima, apenas o passo 1 (*fetch*) e o passo 3 (*execute*) envolvem atualização de memória ou registrador e portanto dois pulsos de *clock* são suficientes para um ciclo.

A dinâmica de execução é descrita a seguir:

- **Antes do primeiro pulso** do *clock* deve-se garantir que todos os sinais de controle, assim como os endereços pertinentes, estão ajustados adequadamente para que a próxima instrução seja lida da memória e armazenada no IR.
- No primeiro pulso do *clock* o **passo 1** (*fetch*) é executado.
- O **passo 2** do ciclo de instrução (*decode*) depende da instrução a ser executada. Os sinais de controle devem ser ajustados de acordo com a instrução (8 *bits* mais significativos do IR). Essa parte é realizada por um circuito combinacional que ativa/desativa os sinais de controle relevantes de acordo com a instrução sendo decodificada. Note que esse passo não requer um pulso do *clock* pois ele será executado assim que uma instrução for carregada no IR. Basicamente, o que o segundo passo faz é acertar os sinais para a execução do **passo 3**.

- O **passo 3** é executado em um segundo pulso do *clock*, e corresponde à execução propriamente dita da instrução.
- Após a execução do **passo 3**, o circuito deve voltar à configuração do início do ciclo de instrução, pronto para a execução da próxima instrução.

Desta forma, um ciclo pode ser completado em dois pulsos de *clock*. A seguir são apresentados detalhes do ciclo de execução para algumas instruções específicas. Esses exemplos servirão para ilustrar o que acontece em cada um dos passos num ciclo FDX. Vamos supor que em nossa CPU a mudança de estados ocorre na subida do sinal de *clock*.

Exemplo 1: Seja a instrução `01EE`, que significa

```
Copiar [EE] para o AC
```

Aqui `[EE]` significa "conteúdo que está na posição `EE` da memória" e `AC` representa o acumulador (um registrador especial que a UC utiliza para armazenar dados envolvidos na execução de uma instrução).

Suponha que essa instrução será a próxima a ser executada e encontra-se no endereço `00` da RAM.

Isto significa que no início do ciclo, antes do primeiro pulso do *clock*,

- o endereço `00` deve estar armazenado no PC
- a porta de endereços da RAM deve estar recebendo o conteúdo do PC
- o pino `R/W` da RAM deve estar setado em `R`
- os dados da saída da RAM devem estar direcionados para a IR

Fetch: o *fetch* será efetuado no primeiro pulso de *clock*. Isto é, é esperado que a instrução `01EE` seja copiada da RAM para o registrador IR. Neste pulso do *clock* deve ocorrer também o incremento do valor do PC (o que acontece se o incremento for realizado apenas ao final do ciclo, após o passo *Execute*?)

Decode: o passo *decode* é executado tão logo a instrução é copiada para o IR (este passo não requer um pulso de *clock*).

Os 8 *bits* mais significativos (no caso `01` em hexadecimal ou 0000 0001 em binário) serão usados por um decodificador de instruções para preparar os sinais de controle para a execução propriamente dita. Além disso, caso a instrução envolva um endereço (que no caso estaria nos 8 *bits* menos significativos do IR), este deve ser enviado para a porta de endereços da RAM (pois a instrução em questão fará acesso a uma posição da RAM, seja para leitura ou para escrita). No exemplo, o acesso será para leitura. E o dado lido deverá ser direcionado para o AC. Desta forma, o decodificador deve ser projetado para que, após a decodificação da instrução, todas as partes do processador estelam devidamente preparados para a execução da instrução.

Execute: Supondo que o decodificador tenha preparado os sinais de controle e direcionamento de dados de forma correta, no segundo pulso do *clock* ocorrerá a execução propriamente dita da instrução. No exemplo considerado, o número que está no endereço `EE` da RAM será copiado para o AC.

A UC precisa garantir que após essa execução, o ciclo volte ao estado inicial. O PC deverá estar apontando para a posição da memória que contém a próxima instrução a ser executada, a porta de endereço da RAM deve estar recebendo o conteúdo do PC, o pino `R/W` da RAM deve estar setado em `R`.

Exemplo 2: Desvio incondicional `0aEE`

```
Desvie para EE
```

Lição de casa ☺

Exemplo 3: Adição `03EE`

```
Some [EE] com [AC] e guarde o resultado em AC
```

Lição de casa ☺

Exercícios

1. No processador considerado neste capítulo, quais são os registradores usados e para que eles servem ?

2. Quantas posições de memória são possíveis com endereços de 8 *bits* ?

3. Por que o incremento do PC (apontador de instruções) deve ocorrer ao mesmo tempo em que a instrução a ser executada é copiada no IR (registrador de instruções) e não no final do ciclo de instrução ?

4. O que pode acontecer se o conteúdo do PC é alterado inadvertidamente para um endereço de uma posição de memória que contém um dado (e não uma instrução) ?

5. Supondo que cada uma das instruções pode ser executada em um ciclo de instrução consistindo de dois pulsos de *clock*, o que deve ser considerado para se estabelecer a frequência ideal do sinal de *clock* ?

6. Descreva o valor (estado) dos registradores, sinais de controle, e valores nas entradas e saídas dos resgistradores e da RAM antes e depois de casa passo do ciclo de instrução, quando a próxima instrução a ser executada é a instrução `Some [0x0A] com [AC] e guarde o resultado em AC`. Suponha que o valor armazenado no AC é 0x04, a próxima instrução a ser executada está no endereço 0x06 da memória RAM, e o valor na posição 0x0A da memória RAM é 0x02.

Apêndice A

Extras

Neste apêndice reunimos alguns tópicos que estão de alguma forma relacionados com o conteúdo visto nos capítulos deste texto, mas que não são essenciais para a compreensão da organização e funcionamento de um computador.

A.1 Funções do tipo $A^n \to A$

No capítulo 5 discutimos expressões e funções booleanas e em especial as formas (expressões) canônicas de funções booleanas na forma soma de produtos e na forma produto de somas. Para tanto, foi considerada a álgebra booleana B e funções booleanas $f : B^n \to B$.

Os conceitos vistos lá valem também em relação a uma álgebra booleana A qualquer. Esta seção apresenta uma breve descrição de como são as funções booleanas definidas sobre A.

Dada uma expressão booleana sobre A, podemos ver que ela define uma função booleana.

Exemplo A.1.1. *Seja $A = \{0, a, \overline{a}, 1\}$ e seja a expressão $a\,\overline{x} + x$. A função f definida por essa expressão é a mostrada na tabela-verdade a seguir:*

x	$f(x)$
0	a
a	a
$\overline{a}$	*1*
1	*1*

Um primeiro ponto que deve ser observado é que nem todas as funções do tipo $f : A^n \to A$ são

booleanas. Para verificar se uma função qualquer $f : A^n \to A$ é booleana ou não, podemos usar um dos teoremas de expansão de Boole.

Exemplo A.1.2. *Seja $A = \{0, 1, a, \overline{a}\}$. A função f definida pela tabela a seguir é uma função booleana?*

x	$f(x)$
0	a
a	$\overline{a}$
$\overline{a}$	1
1	1

De acordo com o teorema de expansão de Boole, sabemos que se f é uma função booleana podemos escrever $f(x) = \overline{x}f(0) + xf(1) = \overline{x}a + x\,1 = \overline{x}a + x$. Em particular, para $x = a$ teríamos então $f(a) = \overline{a}\,a + a = 0\,a + a = a$. Porém, na definição de f temos $f(a) = \overline{a}$. Logo, f não é uma função booleana.

O teorema 5.3.4 implica que o número total de funções booleanas sobre uma álgebra booleana A com $|A|$ elementos é $|A|^{2^n}$, já que existem exatamente 2^n mintermos e cada um deles pode ser combinado com $|A|$ elementos. Isto significa que, dentre os $|A|^{|A|^n}$ funções do tipo $f : A^n \to A$, apenas $|A|^{2^n}$ são booleanas. No caso da álgebra booleana $B = \{0, 1\}$, como $f(e_1, e_2, \ldots, e_n) \in \{0, 1\}$, cada um dos 2^n mintermos ou aparece ou não na expressão canônica, resultando que todas as $|B|^{2^n} = 2^{2^n}$ funções de B^n em B são booleanas.

Na figura A.1 estão mostradas todas as 16 funções booleanas de $A = \{0, a, \overline{a}, 1\}$ em A. Os átomos são $\overline{a}\,\overline{x}$, $\overline{a}\,x$, $a\,\overline{x}$, e $a\,x$.

O segundo ponto a ser notado é que enquanto no caso das funções booleanas $f : B^n \to B$ os átomos são as funções definidas pelos mintermos, no caso de uma álgebra booleana geral os átomos não são as funções definidas pelos mintermos. Apesar dessa diferença, ambas as formas canônicas são também válidas para as funções booleanas $f : A^n \to A$ – isto é, qualquer função booleana pode ser escrita como união de átomos (veja exemplo na figura A.1 e o exemplo a seguir).

Exemplo A.1.3. *Seja $A = \{0, 1, a, \overline{a}\}$ e a expressão booleana $a + x$. A função booleana definida por essa expressão é $f(x) = a + x$:*

$f_0(x) = 0$
$f_1(x) = a\,x$
$f_2(x) = a\,\overline{x}$
$f_3(x) = a$
$f_4(x) = \overline{a}\,x$
$f_5(x) = x$
$f_6(x) = \overline{a}\,x + a\,\overline{x}$
$f_7(x) = a + x$
$f_8(x) = \overline{a}\,\overline{x}$
$f_9(x) = \overline{a}\,\overline{x} + a\,x$
$f_{10}(x) = \overline{x}$
$f_{11}(x) = a + \overline{x}$
$f_{12}(x) = \overline{x}$
$f_{13}(x) = \overline{a} + x$
$f_{14}(x) = \overline{a} + \overline{x}$
$f_{15}(x) = 1$

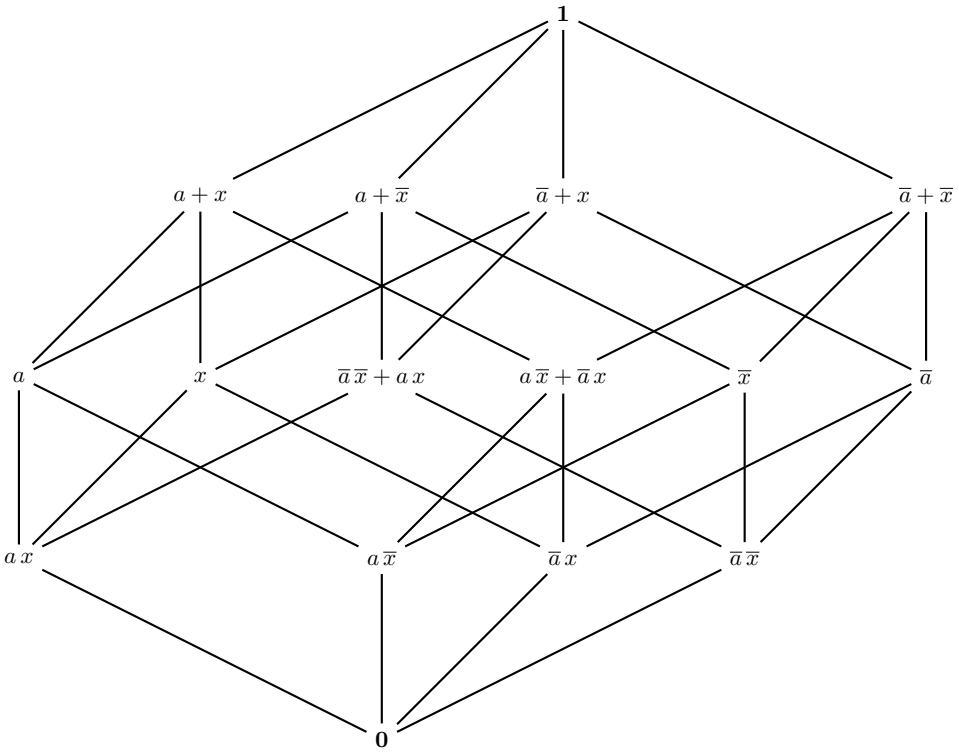


Figura A.1: Diagrama de Hasse da álgebra booleana $\langle A(1), +, \cdot, \bar{}, \mathbf{0}, \mathbf{1}\rangle$, na qual $A = \{0, \overline{a}, a, 1\}$ (i.e., todas as funções booleanas $f : A^1 \to A$).

x	$f(x) = a + x$
0	a
a	a
$\overline{a}$	*1*
1	*1*

Vamos derivar a forma SOP canônica (soma de átomos) dessa função, começando pela expansão de Boole):

$$\begin{aligned} f(x) &= \overline{x}\,f(0) + x\,f(1) \\ &= \overline{x}\,a + x \end{aligned}$$

Note que na expressão expandia aparece um elemento a junto ao mintermo $\overline{x}$, pois este é o valor de $f(0)$. Já se pensarmos na decomposição como soma de átomos (conforme o teorema 4.6.2), note que $x = (\overline{a} + a)x = \overline{a}\,x + a\,x$ e assim temos $f(x) = a\,\overline{x} + \overline{a}\,x + a\,x$, que corresponde à forma "soma de átomos".

Referências Bibliográficas

[Boole, 1854] Boole, G. (1854). *An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities / by George Boole.* London :Walton and Maberly,. http://www.biodiversitylibrary.org/bibliography/29413. 39

[Brayton et al., 1984] Brayton, R. K., Hachtel, G. D., McMullen, C. T., and Sangiovanni-Vincentelli, A. L. (1984). *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers. 101

[Coudert, 1994] Coudert, O. (1994). Two-level Logic Minimization: an Overview. *Integration, the VLSI Journal*, 17(2):97–140. 101

[Coudert, 1995] Coudert, O. (1995). Doing Two-level Minimization 100 Times Faster. In *Proc. of Symposium on Discrete Algorithms (SODA)*, San Francisco CA. 101

[Fišer and Hlavicka, 2003] Fišer, P. and Hlavicka, J. (2003). Boom - a heuristic boolean minimizer. *Computers and Informatics*, 22(1):19–51. 101

[Floyd, 2007] Floyd, T. L. (2007). *Sistemas Digitais - Fundamentos e Aplicações.* Bookman, nona edition. 121, 158

[Garnier and Taylor, 1992] Garnier, R. and Taylor, J. (1992). *Discrete Mathematics for New Technology.* Adam Hilger. 39

[Hill and Peterson, 1981] Hill, F. J. and Peterson, G. R. (1981). *Introduction to Switching Theory and Logical Design.* John Wiley, 3rd edition. 39

[Hill and Peterson, 1993] Hill, F. J. and Peterson, G. R. (1993). *Computer Aided Logical Design with Emphasis on VLSI.* John Wiley & Sons, fourth edition. 101, 158

[Hlavicka and Fiser, 2001] Hlavicka, J. and Fiser, P. (2001). BOOM - A Heuristic Boolean Minimizer. In *Proc. of ICCAD*, pages 439–442. 101

[McGreer et al., 1993] McGreer, P. C., Sanghavi, J., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. (1993). Espresso-Signature : A New Exact Minimizer for Logic Functions. *IEEE trans. on VLSI*, 1(4):432–440. 101

[Micheli, 1994] Micheli, G. D. (1994). *Synthesis and Optimization of Digital Circuits.* McGraw-Hill. 101

[Ross and Wright, 1992] Ross, K. A. and Wright, C. R. B. (1992). *Discrete Mathematics.* Prentice Hall, 3rd edition. 53

[Shannon, 1938] Shannon, C. E. (1938). *A Symbolic Analysis of Relay and Switching Circuits*. 39

[Whitesitt, 1961] Whitesitt, J. E. (1961). *Boolean Algebra and its Applications*. Addison-Wesley. 39