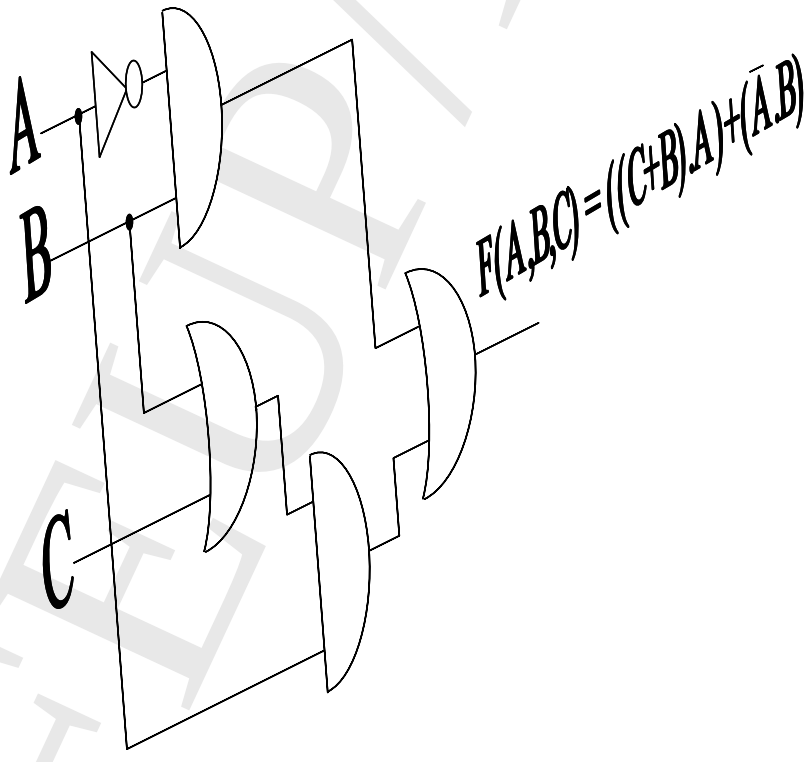
Sistemas Digitais



José Carlos Alves
FEUP/DEEC

V0.51 — 31 de Outubro de 2004

Conteúdo

Capítulo 1

O que são sistemas digitais?

Sistemas digitais são sistemas que processam informação binária, i.e. que pode ser codificada como entidades binárias, normalmente representadas por 1 (um) e 0 (zero). No dominio que nos interessa, sistemas digitais electrónicos são circuitos eléctricos que processam dados representados por zeros e uns, correspondentes a determinados níveis de tensões eléctricas (Volt) ou de intensidades de corrente (Ampère). O tipo de processamento realizado por um sistema digital pode ser muito diverso, desde controlar o funcionamento de um elevador, sequenciar as várias fases de funcionamento de uma máquina de lavar roupa ou realizar as complexas operações que um PC dos nossos dias pode fazer.

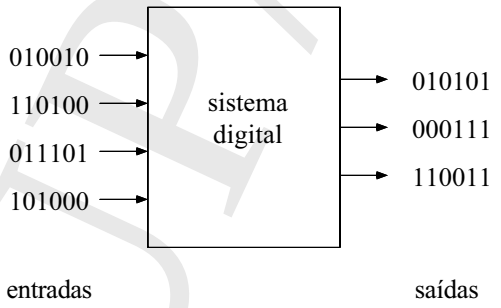


Figura 1.1: Modelo de um sistema digital.

De uma forma geral, podemos considerar um sistema digital como uma caixa negra que tem *entradas* e *saídas* e que realiza uma *função* determinada (figura 1.1). As entradas são representadas simbolicamente por zeros e uns e correspondem, por exemplo, ao estado de um interruptor (ligado ou desligado) ou de um botão de pressão (pressionado ou não pressionado), ao estado de um sensor de nível de água (cheio ou vazio) ou à saída de um termostato (atingido ou não um nivel de temperatura especificado). As saídas são também representadas por zeros

e uns que têm correspondência com, por exemplo, o estado de uma lâmpada ou de um LED[1] (aceso ou apagado), de um motor eléctrico (ligado ou desligado), de uma electro-válvula (aberta ou fechada). A função realizada pelo sistema digital estabelece uma relação entre as entradas e as saídas de forma a que o sistema cumpra uma funcionalidade desejada (por exemplo, lavar roupa...).

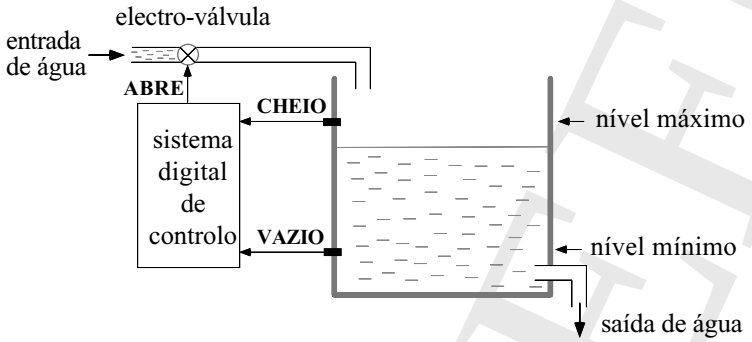


Figura 1.2: Controlo digital do nível de água num depósito.

Consideremos como exemplo um sistema automático para encher um depósito de água (figura 1.2). As entradas desse sistema de controlo são sinais eléctricos *digitais* resultantes de dois sensores de nível de água (CHEIO e VAZIO) e a única saída também digital (ABRE) actua uma electro-válvula que permite abrir ou fechar a entrada de água para o depósito. Podemos descrever facilmente por palavras qual deve ser o funcionamento deste sistema de controlo de forma a que o nível de água seja mantido entre o nível mínimo e máximo: a electro-válvula deve ser ligada (abrindo a água) quando o sensor VAZIO deixar de ser actuado e deve ser desligada quando o sensor CHEIO for atingido. A partir desta descrição "informal" podemos estabelecer relações formais entre as entradas e saídas que nos permitirão ajudar a projectar um circuito como o que acabamos de exemplificar.

Digital e não digital

Um sistema de controlo que apresenta um funcionamento semelhante ao descrito mas que não digital, é a válvula de entrada de água de um autoclismo controlada por uma boia: quando se descarrega o depósito a boia desce abrindo a válvula que deixa entrar a água; à medida que o depósito enche a boia vai subindo, fechando progressivamente a entrada de água (figura 1.3). Ao contrário do sistema de controlo digital, em que a electro-válvula apenas podia estar no estado aberto ou no estado fechado, esta válvula *analógica* pode estar num número infinito de

[1] *Light Emmiting Diode* ou díodo emissor de luz: componente electrónico que quando atravessado por uma corrente eléctrica emite luz com uma libertação de calor muito reduzida

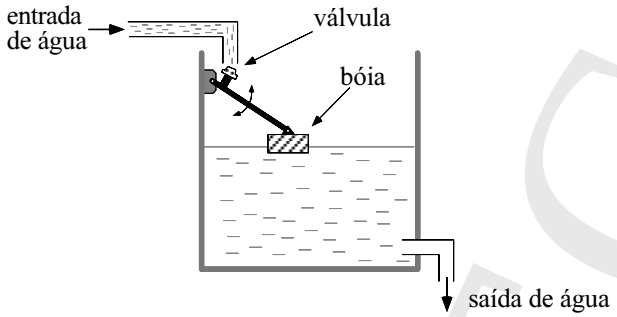


Figura 1.3: Um sistema *analógico* para controlar o nível de água num depósito.

estados, desde toda aberta em que deixa passar o fluxo máximo de água até totalmente fechada quando o nível de água atinge o máximo.

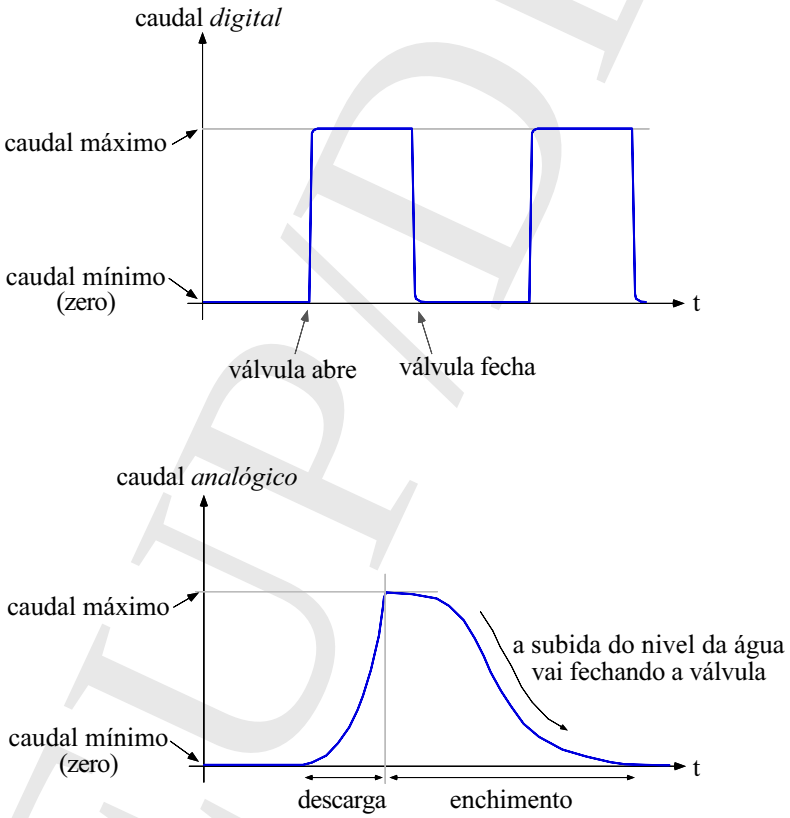


Figura 1.4: Variação temporal do caudal de água: digital vs. analógico.

Se representarmos graficamente a evolução ao longo do tempo do caudal de entrada de água no tanque em ambas as situações, vemos que o caudal "digital" apenas está em dois níveis diferentes (zero ou máximo), enquanto que o caudal "analógico" varia de forma contínua ao

longo do tempo (figura 1.4). Esta é a grande diferença entre grandezas analógicas e digitais: enquanto que as primeiras evoluem de forma contínua no tempo, as segundas apenas podem assumir dois estados discretos bem definidos.

bits e mais *bits*...

Como uma grandeza binária (ou um *bit*[2]) só pode ter dois estados diferentes, não serve para muito. Como podemos representar coisas mais complicadas como números, caracteres, cores, imagens ou sons? Agrupando vários zeros e uns podemos expandir o número de "coisas" diferentes que se podem representar. Por exemplo, com dois *bits* podemos formar 4 grupos diferentes de zeros e uns: 00, 01, 10 e 11. Relembrando o nosso sistema digital para controlo do nível de água no depósito, poderíamos, com dois *bits* detectar 4 níveis diferentes de água e codificá-los como: 00=vazio, 01=quase-vazio, 10=quase-cheio e 11=completamente cheio.

Estes estados adicionais poderiam, por exemplo, servir para afixar num mostrador o estado corrente do nível da água ou emitir um sinal de aviso quando fosse atingido o nível quase-vazio. Acrescentando mais *bits* (e também mais sensores de nivel de água) seria possível codificar e processar um número maior de níveis de água. Como por cada *bit* que se acrescenta é duplicado o número de códigos possíveis, com N *bits* é assim possível representar 2^N entidades diferentes. Ppor exemplo, com 8 *bits* podem-se representar apenas 256 coisas diferentes, mas com 32 *bits* este número chega a 4.294.967.296! Este assunto será abordado com detalhe no capítulo 2

e como funcionam?

Um sistema digital (electrónico) é um circuito eléctrico que estabelece uma relação *lógica* entre valores *binários* (zeros e uns) colocados nas suas entradas e os valores binários que aparecem nas suas saídas. Num circuito electrónico digital, uns e zeros são geralmente representados por tensões eléctricas *altas* e *baixas*: um 1 equivale a tensões acima de um determinado limiar, e um zero para valores inferiores a um certo limite mínimo.

Imaginemos que o estado, aceso ou apagado, de uma vulgar lâmpada de incandescência, das que temos em nossas casas, representa um 1 ou um 0, respectivamente. A lâmpada não acende apenas quando a tensão da rede eléctrica atinge os 220V, nem apaga só quando esse valor chega aos 0V. Existe um intervalo de tensões eléctricas "baixas"para as quais a lâmpada não emite qualquer luz (experimente acender uma lâmpada de 220V com uma vulgar pilha de 1.5V...), e a partir de um tensão eléctrica "alta" a lâmpada fica no estado acesa. Existe, naturalmente um intervalo de valores da tensão eléctrica em que poderemos considerar o estado da lâmpada como nem bem apagado nem bem aceso, ou seja um estado *indefinido* em que não se pode representar

[2] do inglês *binary digit*

bem nem o 1 nem o 0. Desta forma, se ocorrerem pequenas variações[3] da tensão eléctrica "baixa" quando a lâmpada está apagada, esta permanece apagada e continua a representar um zero; também no estado aceso, pequenas flutuações da tensão eléctrica não fazem com que a lâmpada deixa de representar um 1 (figura 1.5).

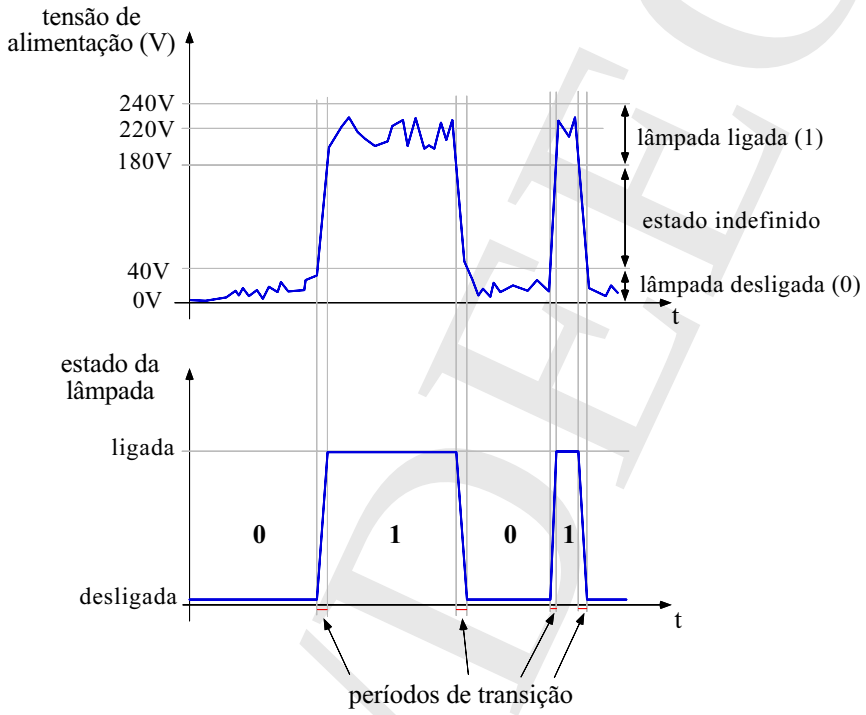


Figura 1.5: Os dois estados possíveis de uma lâmpada (ligada e desligada) em função da tensão de alimentação

Um sistema electrónico digital também "entende" como zeros e uns tensões eléctricas que admitem flutuações. Por exemplo, numa tecnologia corrente usada para o fabrico de circuitos digitais[4] (CMOS–*Complementary Metal-Oxide Semiconductor*) funcionado com um tensão de alimentação de 5V, é interpretado o valor lógico 0 para tensões eléctricas abaixo de 1.5V (30% de 5V), e é entendido o valor lógico 1 quando se apresenta nas entradas uma tensão eléctrica acima de 3.5V (70% de 5V) V. Outras tecnologias de implementação de circuitos digitais apresentam valores diferentes: por exemplo, circuitos digitais TTL (*Transistor-Transistor Logic*) interpretam o valor lógico 0 para tensões inferiores a 0.8V e o valor lógico 1 para tensões acima dos 2.7V. Tensões eléctricas fora destes intervalos não representam correctamente nem zero nem um (figura 1.6).

[3]neste exemplo poderemos considerar pequenas variações como poucas dezenas de Volt.

[4]Uma tecnologia de fabrico de circuitos digitais estabelece a forma como os circuitos electrónicos são construídos e é caracterizada por um conjunto de parâmetros eléctricos e dinâmicos.

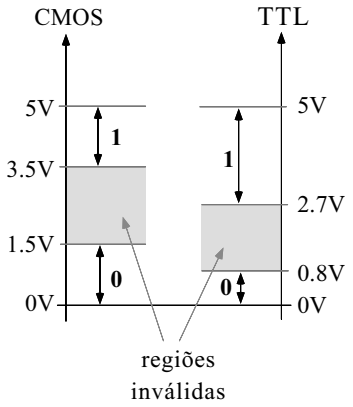


Figura 1.6: Tensões eléctricas e valores lógicos para circuitos digitais CMOS e TTL.

portas lógicas

Qualquer sistema digital pode ser construído usando apenas 3 tipos de *funções lógicas* elementares designadas por *E*, *OU* e *NÃO* [5].

As funções lógicas elementares operam sobre dois valores lógicos 0 ou 1 (ou apenas um para o caso da função NÃO) e produzem um valor lógico para cada uma das combinaçãoes possíveis dos valores lógicos dos operandos. A figura 1.7 mostra as tabelas que descrevem o comportamento dessas funções elementares e os símbolos gráficos geralmente utilizados para as representar.

Aos componentes electrónicos que realizam essas funções lógicas elementares chamam-se *portas lógicas* (em inglês *gates* ou *logic gates*), e estende-se também esta designação aos símbolos gráficos que as representam. Uma função complexa pode ser construída ligando entre si símbolos que representam as portas lógicas elementares, da forma que se exemplifica na figura 1.8.

Estas 3 funções elementares são a base em que assenta o funcionamento de qualquer sistema digital e formam o conjunto bàsico de funções em que se fundamenta a Algebra de Boole, que permite formalizar e tratar matematicamente o comportamento de sistemas digitais. Este assunto será aprofundado no capitulo 3.

[5] Muitas vezes emprega-se em liguagem corrente os termos em Inglês AND, OR e NOT para designar estas funções

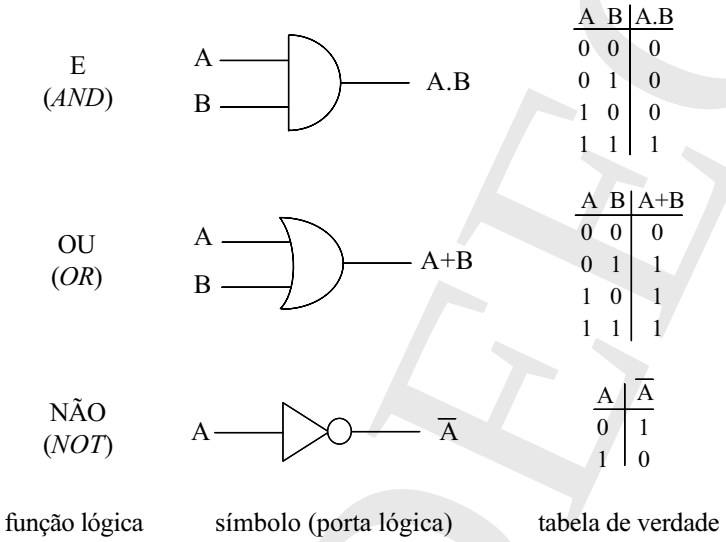


Figura 1.7: As 3 funções lógicas elementares

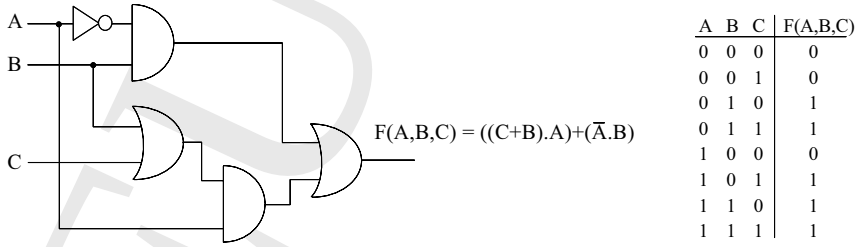


Figura 1.8: Uma função lógica complexa construída com portas lógicas

Capítulo 2

Representação de informação em binário

2.1 Introdução

Sistemas digitais processam informação representada em binário. Essa informação pode ser vista por um humano ou interpretada por uma máquina de formas muito diversas: números, texto, imagens, sons ou mesmo accões mecânicas (ligar um motor eléctrico, por exemplo). Como um único *bit* de informação apenas permite representar dois estados distintos (1 ou 0), qualquer colecção de informação é composta por um conjunto de entidades formadas por agrupamentos de vários *bits*. Actualmente é usual representar informação digital como conjuntos de dados elementares cada um constituído por 8 *bits* (ou um *byte*). Um *byte*permite representar 256 "coisas" diferentes, correspondendo a todas as combinações diferentes de 8 zeros ou uns. Um conjunto conveniente dessas "coisas" são os números inteiros positivos de 0 a 255 ou números com sinal entre -127 e $+127$. Veremos mais tarde como se podem representar números inteiros com um sistema de numeração que apenas usa os dígitos 1 e 0 (secção 2.2).

como se representa um texto?

Com um conjunto de *bytes* é então possível representar um texto, em que cada carácter, dígito ou sinal de pontuação é *codificado* num código de 8 *bits*. Actualmente a grande maioria de fabricantes de computadores (os grandes produtores e consumidores de informação digital) segue um padrão para representação de texto que se chama código ASCII (*American Standard Code for Information Interchange*) e que codifica cada carácter do alfabeto (incluindo as letras maiúsculas, minúsculas, dígitos e sinais de pontuaçãao) num código de 8 *bits*. Na figura 2.1 mostra-se a sequência de *bytes* que codifica um texto com duas linhas.

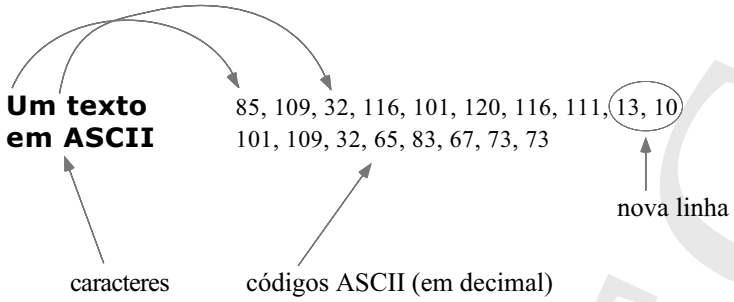


Figura 2.1: Um texto e o código ASCII correspondente.

imagens digitais

Outro tipo de informação digital muito vulgarizada hoje em dia é a imagem. Uma imagem digital é constituída por um conjunto de pontos elementares de cor (*pixel*), cada um representado por um ou mais *bits*. Utilizando um só *bit* por cada pixel apenas permite que cada ponto de imagem apenas possa ter duas cores diferentes, normalmente o preto e o branco. Se cada *pixel* for codificado com um *byte*, então já é possível representar imagens onde cada *pixel* pode apresentar uma de 256 cores diferentes (imagens a preto-e-branco com 256 níveis de cinzento apresentam já uma qualidade razoável). Com 32 *bits* por cada pixel é possível representar 4.294.967.296 cores diferentes, o que para os nossos olhos é um número "infinito" de cores! Na figura 2.2 mostra-se como uma fotografia digital a preto-e-branco é formada por pequenos pontos com diferentes níveis de cinzento; cada *pixel* é representado por um número inteiro que codifica a sua tonalidade.

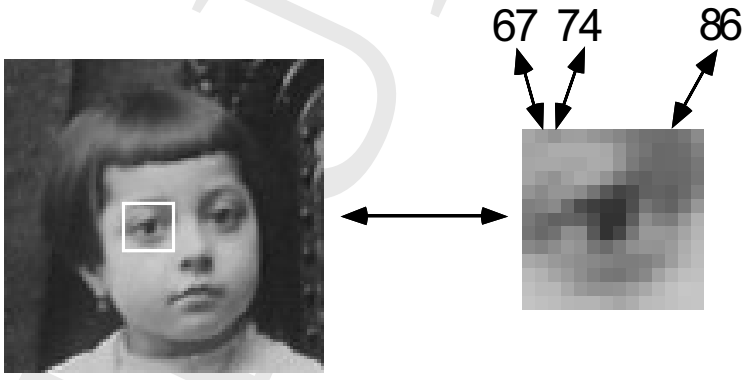


Figura 2.2: Uma imagem digital a preto-e-branco.

sinais de áudio

Num sistema electrónico, um sinal de áudio é representado por uma tensão eléctrica analógica que oscila de forma contínua com a frequência do sinal que reproduz. Quando essa tensão eléctrica é aplicada a um altifalente, provoca a vibração de um cone flexível, produzindo dessa forma as ondas de pressão que nós interpretamos como sons.

Nos antigos discos de vinil (analógicos), o sinal de áudio é gravado como um sulco cravado em espiral sobre o material plástico, cujas oscilações representam o sinal a reproduzir. A reprodução é feita por uma agulha associada a um sistema electrónico que traduz essas oscilações mecânicas em oscilações de uma tensão eléctrica, que depois de amplificada é aplicada aos altifalantes para produzir o som. Naturalmente que qualquer imperfeição que possa surgir na superfície do disco é traduzida em vibrações indesejáveis da agulha e o consequente ruído produzido pelos altifalantes. Aquilo que vulgarmente se chama "disco riscado" não é mais do que um defeito no sulco que faz com que a agulha salte para um sulco adjacente e não siga o seu percurso normal em espiral.

Um sinal de áudio digital é representado por um conjunto de dados binários que codificam a amplitude do sinal analógico em *amostras* tomadas em intervalos de tempo regulares definidos pela *frequência de amostragem*. Na figura 2.3 mostra-se um segmento de um sinal analógico e a correspondente representação digital usando 8 *bits* para codificar cada amostra[1].

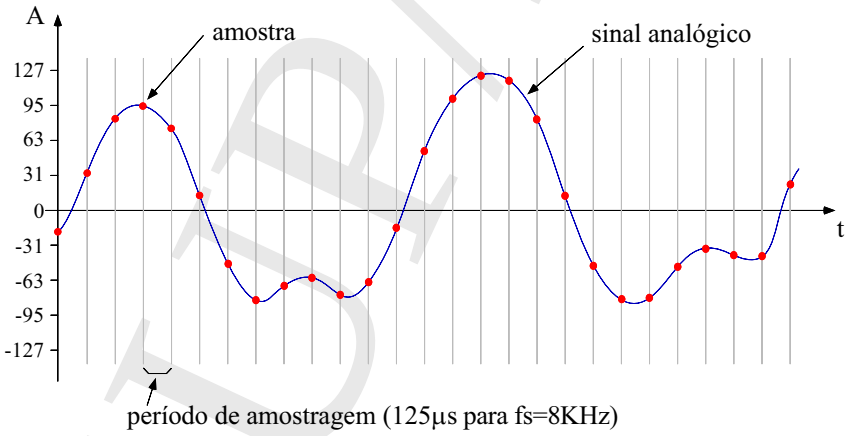


Figura 2.3: Um sinal analógico e a correspondente representação digital.

Apesar de um sinal de analógico variar de forma contínua no tempo, a codificação do valor

[1]considera-se neste exemplo que os valores de amplitude estão representados em sinal e grandeza (ver secção 2.6.1)

das amostras com um número finito de *bits* apenas o permite representar com um conjunto finito de amplitudes do sinal: quantos mais *bits* forem usados para codificar cada amostra, maior é a resolução e a fidelidade obtida.

Um sinal de áudio de baixa qualidade adequado para transmitir voz humana, pode ser correctamente representado por um conjunto de amostras tomadas com uma frequência de "apenas" 8 KHz (8 mil vezes por segundo) e codificadas em 12 *bits*. No entanto, para áudio de alta fidelidade é já necessário amostrar o sinal com uma frequência bastante mais elevada e codificar cada amostra com mais *bits*. Por exemplo, nos CDs de áudio é usada uma frequência de amostragem de 44.1 KHz e cada amostra é codificada em 16 *bits*.

conversores A/D e conversores D/A

Existe um tipo de dispositivo electrónico que realiza a operação de conversão de um sinal analógico para uma representação digital: conversores analógico/digital ou simplesmente conversores A/D. Um conversor A/D traduz para uma representação digital um sinal eléctrico analógico, que, por sua vez, pode representar diferentes grandezas físicas que são "naturalmente" analógicas: temperatura, pressão, deslocamento, velocidade, etc.

A função inversa é realizada por um dispositivo chamado conversor digital/analógico ou conversor D/A. Este dispositivo traduz um código binário que representa o valor da amplitude de um sinal, numa tensão eléctrica cujo valor é proporcional à amplitude que representa. Num leitor de CDs de áudio, a informação digital gravada no disco como sequências de zeros e uns é processada e enviada a um conversor D/A para reconstruir o sinal analógico, que depois de amplificado vai excitar os altifalantes. Como na representação digital de um sinal apenas são conhecidos os valores da sua amplitude nos instantes de amostragem, na sua reconstrução analógica o valor de cada amostra é mantido constante ao longo de um período de amostragem dando origem a um sinal em "escada" como se mostra na figura 2.4. O sinal analógico produzido por um conversor D/A é geralmente tratado posteriormente por circuitos analógicos para filtragem do sinal que permitem "arredondar" as transições abruptas entre amostras.

como se pode *ver* informação binária?

A tradução de informação representada digitalmente para algo que faça sentido para nós (humanos) é feita por dispositivos electrónicos que a traduzem em algo "visível", por exemplo caracteres, números, sons ou imagens. Naturalmente que não fará sentido interpretar a informação gravada num CD de música como um texto ou como uma imagem digital.

Um só *bit* de informação pode ser convenientemente visto como o estado de um LED: ligado representa 1 e desligado representa 0. Associando vários LEDs numa matriz é possível

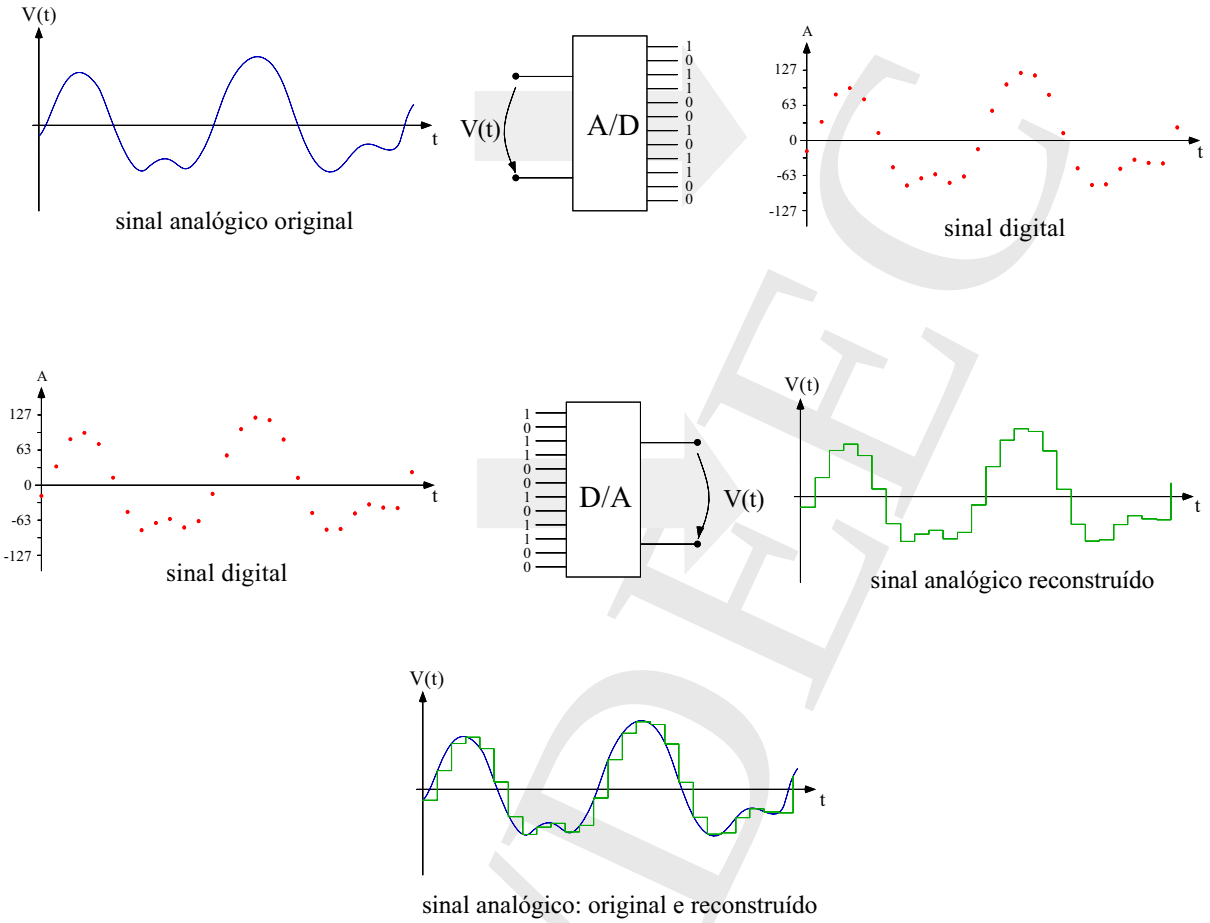


Figura 2.4: Conversores A/D e conversores D/A.

traduzir informação digital em desenhos que representem caracteres alfabéticos, algarismos ou sinais de pontuação. Existe um tipo particular de mostrador composto por 7 LEDs (designados vulgarmente por mostradores de 7 segmentos—figura 2.5) que permitem afixar os 10 dígitos decimais e alguns símbolos adicionais (o sinal menos, por exemplo), vulgarmente utilizados em diversos equipamentos electrónicos (relógios digitais, por exemplo).

De forma semelhante, uma imagem digital pode ser vista num monitor ou impressa em papel, traduzindo a informação binária que representa a cor dos pontos elementares de imagem *pixel* em pontos luminosos apresentados sobre um écran ou pontos de tinta impressos sobre papel.

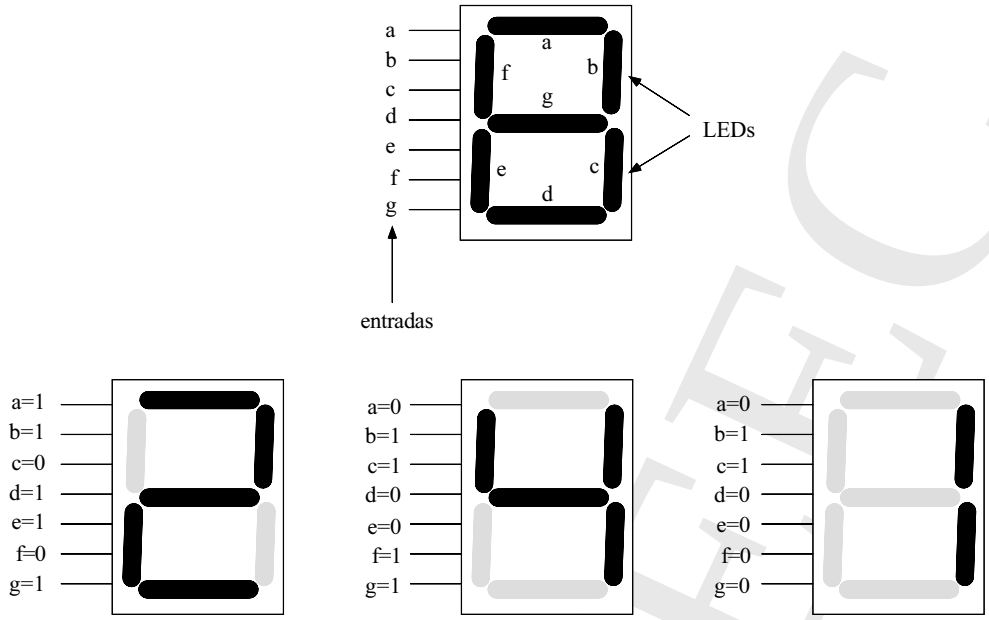


Figura 2.5: Mostradores de 7 segmentos

2.2 Representando números

Outro tipo de informação processada correntemente por sistemas digitais são valores numéricos. Além disso, e como já referimos atrás, muitas outras formas de informação podem ser convenientemente representadas por valores numéricos: a côr de um *pixel*, a amplitude de uma amostra de um sinal de áudio ou os códigos atribuídos a caracteres.

A codificação dos caracteres alfabéticos pelo padrão ASCII atribui aos caracteres números inteiros segundo a ordem alfabética crescente. Por exemplo, as letras maiúsculas são representadas por ordem alfabética a partir do número 65 (A=65, B=66,...,Z=90), as letras minúsculas representam-se a partir do número 97 (a=97, b=98,..., z=122) e os algarismos correspondem aos números entre 48 (dígito zero) e 57 (dígito nove). A ordem "alfabética" entendida por um sistema digital (por exemplo um PC) que represente texto segundo esta norma não é mais do que a ordem natural dos números que representam as letras. Assim, num texto em ASCII a palavra "Zebra" é alfabéticamente anterior à palavra "animal" e ambas são alfabeticamente superiores à palavra "2345".

A representação de quantidades numéricas em binário é feita recorrendo a um sistema de representação de números semelhante ao sistema decimal que usamos correntemente no nosso dia a dia.

No sistema decimal temos 10 símbolos (os 10 dígitos decimais) que representam as quantidades zero a nove. Para representar uma quantidade superior a 9 usamos agrupamos um con-

junto de dígitos cuja posição define o peso de cada um no valor do número. Assim, a *sequência de dígitos* 345 representa o valor trezentos e quarenta e cinco que pode ser calculado como:

$$3 \times 100 + 4 \times 10 + 5 = 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$$

O peso de cada dígito corresponde assim às potências inteiras de 10, onde o número *dez* não é mais do que o número de digitos usados no sistema de numeração. Esse número é também chamado a *base* do sistema de numeração e deve ser escrita como sub-índice a seguir a um número, sempre que houver ambiguidade relativamente à base numérica em que ele está representado (23 em base 10 deve escrever-se 23_{10}).

Generalizando este conceito, podemos representar quantidades numéricas[2] utilizando qualquer base $b >= 2$ (i.e. qualquer número de "dígitos" superior a 2). Assim, o valor do número com N dígitos $B_{N-1}B_{N-2}...B_2B_1B_0$ representado em base b obtém-se como:

$$B_{N-1}.b^{N-1} + B_{N-2}.b^{N-2} + \ldots + B_2.b^2 + B_1.b^1 + B_0.b^0$$

Por exemplo, em base 5 podemos representar números utilizando os 5 dígitos 0, 1, 2, 3 e 4. O valor do número 342_5 obtém-se como:

$$342_5 = 3 \times 5^2 + 4 \times 5^1 + 2 \times 5^0 = 3 \times 25 + 4 \times 5 + 2 = 97_{10}$$

Em sistemas de numeração posicionais como o que descrevemos aqui, chama-se ao dígito da esquerda o dígito mais significativo (ou MSD do inglês *Most Significant Digit*), e chama-se ao dígito mais à direita o dígito menos significativo (LSD do inglês *Least Significant Digit*).

2.2.1 Números fraccionários

O valor da parte fraccionária de um número é obtido por um processo semelhante ao apresentado para a parte inteira, mas onde os pesos dos dígitos à direita do ponto fraccionário são as potências negativas da base de representação. Num número fraccionário representado em base 10, o dígito das décimas imediatamente à direita do ponto fraccionário (ou ponto decimal) tem um peso 0.1 ou 10^{-1}, o seguinte (centésimas) tem um peso 0.01 ou 10^{-2} e assim por diante. Em qualquer outra base numérica podemos calcular o valor da parte fraccionária por um processo semelhante. Por exemplo, o valor do número 0.321_4 escreve-se em base 10:

$$0.321_4 = 3 \times 4^{-1} + 2 \times 4^{-2} + 1 \times 4^{-3} = 0.890625_{10}$$

[2] para já apenas consideraremos números inteiros positivos

X	2^X	2^{-X}
0	1	1.0000000000
1	2	0.5000000000
2	4	0.2500000000
3	8	0.1250000000
4	16	0.0625000000
5	32	0.0312500000
6	64	0.0156250000
7	128	0.0078125000
8	256	0.0039062500
9	512	0.0019531250
10	1024	0.0009765625

Figura 2.6: A "tabuada" das potências inteiras de 2

2.2.2 O sistema binário

O sistema de numeração binário (base 2) permite representar números utilizando apenas os dois dígitos 0 e 1 que valem, respectivamente o valor zero e o valor um. O valor de um número representado em base dois é obtido pelo mesmo processo que se apresentou antes. No entanto, como o valor de cada dígito ou é 1 ou é zero, as operações a realizar envolvem apenas a adição das potências inteiras de dois correspondentes aos dígitos que são 1. Como exemplo, o valor do número inteiro binário 11010_2 é:

$$11010_2 = 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26_{10}$$

e do número fraccionário binário 0.1101_2 é calculado como:

$$0.1101_2 = 1\times 2^{-1} + 1\times 2^{-2} + 0\times 2^{-3} + 1\times 2^{-4} = 2^{-1} + 2^{-2} + 2^{-4} = 0.8125_{10}$$

Como iremos a partir daqui trabalhar com o sistema de numeração binário, é conveniente memorizar a "tabuada" das potências inteiras de 2 que se apresenta na figura 2.6.

2.2.3 Os sistemas octal e hexadecimal

Para além do sistema binário (base 2), o sistema octal (base 8) e o sistema hexadecimal (base 16) são também formas convenientes para representar informação binária, embora de uma maneira mais compacta do que utilizando o sistema binário.

No sistema octal são usados 8 dígitos (0 a 7). Como 8 é 2^3, estes 8 dígitos são representados pelas 8 combinações possíveis de 3 *bits*, desde $0_{10} = 000_2$ até $7_{10} = 111_2$. Por esta razão, um número em base 2 pode ser representado em base 8 fazendo corresponder a cada grupo de 3 *bits*

um dígito do sistema octal (agrupando-os para a esquerda a partir do ponto fraccionário e para a direita a partir do ponto fraccionário), por exemplo:

$$1101011.11010_2 = 153.64_8$$

Note-se que para obter um número inteiro de grupos de 3 *bits* foi necessário acrescentar 2 zeros à esquerda e 1 zero à direita do número dado, mas que não são significativos.

Para converter para binário um número representado em octal procede-se de forma inversa, substituindo cada dígito octal pela sua representação em binário:

$$7413.15_8 = 111100001011.001101_2$$

O sistema hexadecimal necessita de 16 dígitos para representar as quantidades de zero a 15. Para além dos 10 dígitos decimais (0-9) são usadas as letras de *A* a *F* para representar os "dígitos" 10 a 15, respectivamente. Como 16 é 2^4, cada dígito do sistema hexadecimal é representado por um conjunto de 4 *bits*. De forma semelhante ao apresentado para o sistema octal, a conversão entre base 2 e base 16 pode ser feita fazendo corresponder cada dígito hexadecimal a um grupo de 4 *bits*, contanto para a direita e para a esquerda a partir do ponto fraccionário:

$$01110111101.011111_2 = 3BD.7C_{16}$$

$$1A8D.F8_{16} = 1101010001101.11111_2$$

O sistema octal teve interesse no início da era dos minicomputadores porque a informação com que trabalhavam era convenientemente representada por grupos de 3 *bits*. Hoje em dia isso já não acontece, embora seja por vezes conveniente representar informação binária como dígitos em base 8. Um exemplo é o comando `chmod` do sistema operativo Unix (ou Linux) para modificar as permissões de um ficheiro, onde essas permissões podem ser especificadas como uma cadeia de 9 *bits* representados como 3 dígitos em base 8. Por exemplo, o comando `chmod 754 file` altera as permissões do ficheiro `file` para `rwxr-xr--`, ligando as permissões correspondentes aos *bits* em 1 e desligando as que correspondem a zero.

O sistema hexadecimal tem particular interesse como forma de representação de dados binários porque como 4 é o número de *bits* que representa cada dígito hexadecimal e bastam por isso dois dígitos hexadecimais para representar 1 *byte* (8 *bits*)

Note que a conversão entre a base 8 e base 16 pode ser feita muito facilmente utilizando uma representação intermédia em base 2 e aplicando as regras descritas atrás para converter entre base 2 e as bases 8 e 16.

2.2.4 Como se representa um número inteiro em base 2?

Vimos atrás como se podem representar quantidades numéricas em bases numéricas diferentes da base 10 e como se pode obter o valor decimal dessas representações. Como se pode determinar a representação binária de um número dado em base 10?

Antes de vermos um processo sistemático para representar em base 2 um número dado em base 10, vejamos como se podem obter os dígitos decimais de um número representado em decimal. Em primeiro lugar, repare-se que se tivermos um número representado em base 10, o resultado da divisão desse número por 10 equivale a deslocar o ponto decimal uma posição para a esquerda:

$$9372.65_{10}/10 = 937.265_{10}$$

Na realidade, dividir um número representado em base b por b^N equivale a deslocar o ponto fraccionário N posições para a esquerda. Consequentemente, a multiplicação por b^N corresponde a deslocar o ponto fraccionário N posições para a direita. Por exemplo, com números representados em base 2:

$$45.625_{10} = 101101.101_2$$

$$101101.101_2 \times 4_{10} = 10110110.1_2 = 182.5_{10}$$

$$101101.101_2/8_{10} = 101.101101_2 = 5.328125_{10}$$

Assim, se dividirmos um número inteiro decimal por 10 obtemos à direita do ponto decimal o dígito das unidades, o que não é mais do que o resto da divisão inteira do número por 10 (i.e. o quociente inteiro). Se repetirmos este processo aplicado-o sucessivamente aos quocientes podemos determinar todos os dígitos que compõem o número.

Este procedimento pode aplicar-se para obter os dígitos binários que representam um número em base dois, com a diferença de que a base numérica pela qual o número é dividido é 2. Assim, o resto da divisão inteira de um número por 2 representa o seu *bit* menos significativo (0 ou 1) e o quociente representa o valor dos *bits* restantes; repetindo sucessivamente este procedimento até se obter um quociente nulo determinam-se todos os *bits* que representam o número em binário. A figura 2.7 exemplifica a aplicação deste processo.

2.2.5 Como se representa em base 2 um número fraccionário?

Em primeiro lugar vejamos como podemos determinar os dígitos decimais que compõem um número fraccionário em base 10. Por exemplo, dado o número 0.571, como poderemos "extrair"

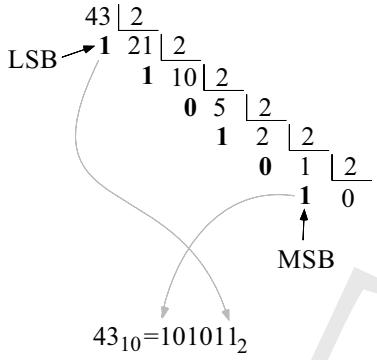


Figura 2.7: Conversão do número inteiro 43 para base 2.

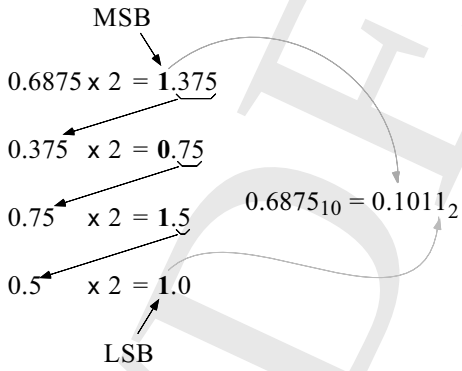


Figura 2.8: Representação em base 2 do número fraccionário 0.6875.

os 3 dígitos que formam a sua parte decimal? Como multiplicar um número por 10 equivale a deslocar o seu ponto fraccionário uma posição para a direita, basta então multiplicar 0.571 por 10 para obter o primeiro dígito do número (5) como a parte inteira do resultado:

$$0.571 \times 10 = 5.71$$

Se aplicarmos repetidamente este processo à parte decimal restante poderemos obter os restantes dígitos que formam o número.

Para obter a representação binária de um número fraccionário dado em base 10 basta multiplicar a parte fraccionária pela base (2) e tomar a parte inteira desse produto como o *bit* mais significativo (MSB) da parte fraccionária do número. Repetindo esse processo para a parte fraccionária resultante obtemos os dígitos binários que representam esse número. A figura 2.8 ilustra a aplicação deste processo.

Enquanto que um número inteiro tem uma representação em base 2 exacta com um número finito de *bits*, um número fraccionário pode requerer um número infinito de *bits* para ser representado com exactidão. Como se mostra na figura 2.9, o número 0.8 é representado por uma

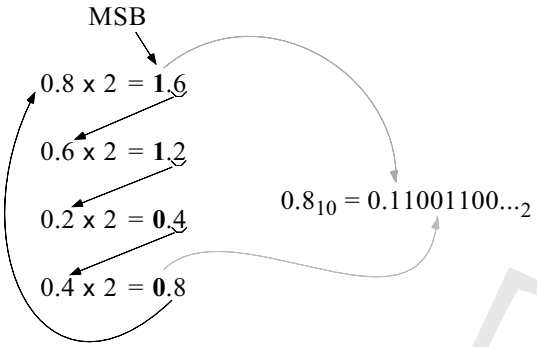


Figura 2.9: Representação em base 2 do número decimal 0.8

sequência de *bits* onde o padrão 1100 se repete indefinidamente. Qualquer outra representação deste valor que utilize um número finito de *bits* será sempre uma aproximação, tanto mais exacta quantos mais *bits* forem usados (por exemplo $0.110011001100_2 = 0,7998046875_{10}$).

2.2.6 Números com parte inteira e parte fraccionária

A representação em base 2 de um número com parte inteira e parte fraccionária é feita aplicando separadamente os procedimentos apresentados acima para a parte inteira e para a parte fraccionária. Por exemplo, o número 43.6875_{10} formado pela parte inteira e parte decimal dos exemplos apresentado nas figuras 2.7 e 2.8 representa-se em binário por 101011.1011_2.

2.3 Adição e subtracção binária

As operações aritméticas elementares que habitualmente realizamos com números representados em base 10 podem ser aplicadas de forma semelhante a número em base 2 (ou em qualquer outra base numérica).

Relembrando, o processo para adicionar dois números em base 10 consiste em alinhar os dois números pelo ponto fraccionário e somar os seus dígitos dois a dois; sempre que o resultado da adição excede 9 (i.e. não pode ser representado por um dígito do sistema decimal), escrevemos apenas o dígito das unidades como resultado e *transportamos* uma unidade para a casa seguinte (figura 2.10).

A adição de números representados em base 2 segue o mesmo processo: quando o resultado da adição de dois dígitos é superior a 10_2, escrevemos zero como resultado dessa casa e transportamos uma unidade para o andar seguinte. Note-se que, como em alguns andares da soma é necessário adicionar o transporte igual a 1 produzido pelo andar anterior, a operação realizada

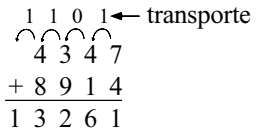


Figura 2.10: Adição de dois números inteiros em base 10.

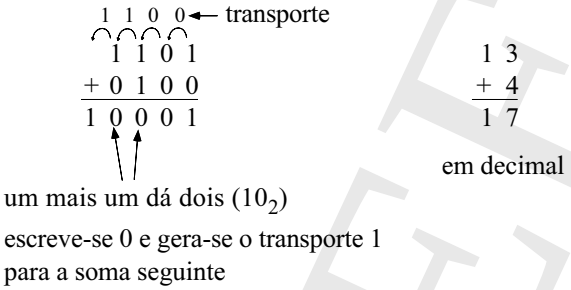


Figura 2.11: Adição de dois números em base 2.

é na realidade a soma de 3 *bits* que pode dar como resultado zero, um dois ou três, representado em binário por 00, 01, 10 ou 11 (figura 2.11). Na literatura em língua inglesa usa-se o termo *carry* para designar o *bit* de transporte durante a realização de uma adição binária: o *bits carry-out* é produzido por um andar da adição e o *bit carry-in* é o que entra no andar seguinte.

A subtracção também é efectuada em binário da mesma forma que a realizamos em decimal, usando sempre o número maior (em valor absoluto) como diminuendo e o número menor como diminuidor. Sempre que o dígito do diminuendo é inferior ao dígito correspondente do diminiudor, é necessário "pedir emprestado" um 1 à casa seguinte e que deve ser retirado dessa posição subtraindo-o ao diminuendo ou somando-o ao diminuidor. Na figura 2.12 mostra-se a sequência de operações realizada para efectuar a subtracção binária $13 - 6 = 7$.

2.4 Multiplicação e divisão binária

O processo de multiplicação conhecido para multiplicar números em base 10 pode aplicar-se também à multiplicação de números representados em binário: o resultado é obtido adicionando os produtos de cada dígito do multiplicador pelo multiplicando, "alinhados" pela coluna do dígito do multiplicador. Como os dígitos de um número binário valem apenas um ou zero, o seu produto pelo multiplicando ou vale zero ou vale o próprio multiplicando. A figura 2.13 ilustra este processo.

A divisão binária é um pouco mais complexa de realizar "à mão" mas segue o mesmo

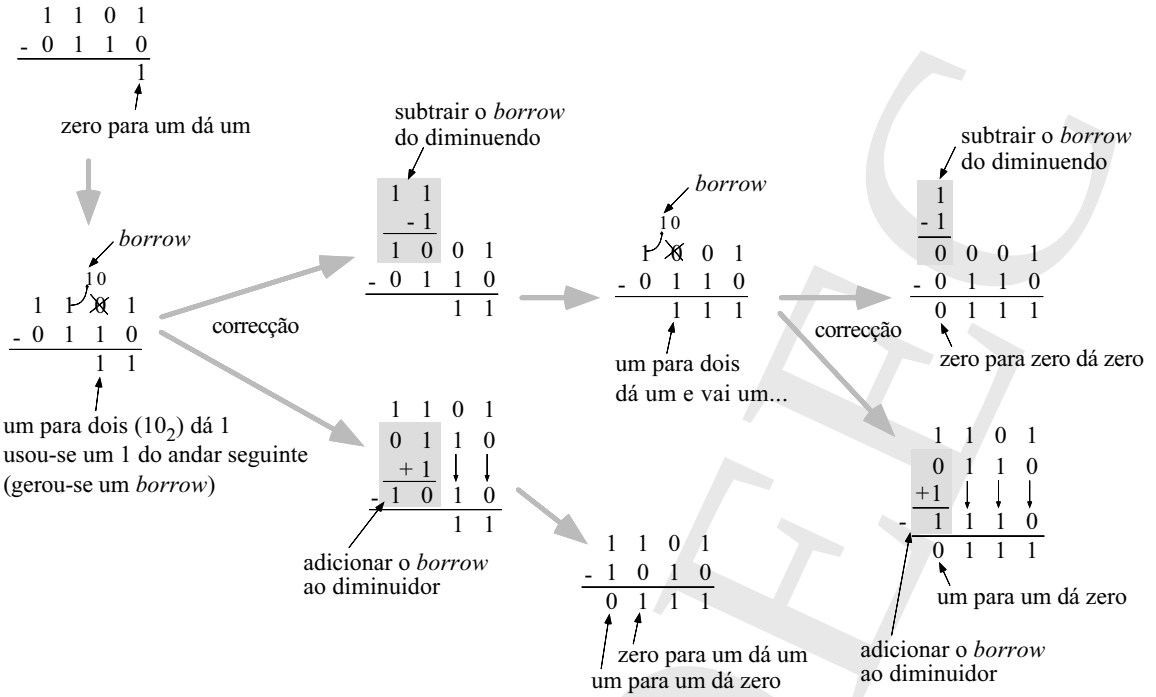


Figura 2.12: Subtracção binária $13 - 6 = 7$.

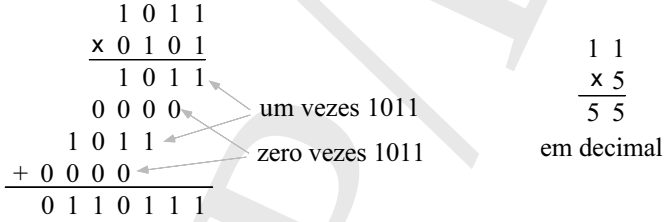


Figura 2.13: Multiplicação de números em base 2.

algoritmo que usamos para dividir números em base 10 (figura 2.14). Note-se que, como foi já referido atrás, a divisão de um número binário por uma potência inteira de 2 (2^N) resume-se a deslocar o ponto fraccionário de N posições para a esquerda.

2.5 Dimensão dos resultados e *overflow*

Sistemas digitais que processam informação numérica representam geralmente grandezas numéricas com um número fixo de *bits*. Por exemplo, os microprocessadores mais simples utilizam *registos* e unidades aritméticas de cálculo com apenas 8 *bits*, o que só permite representar

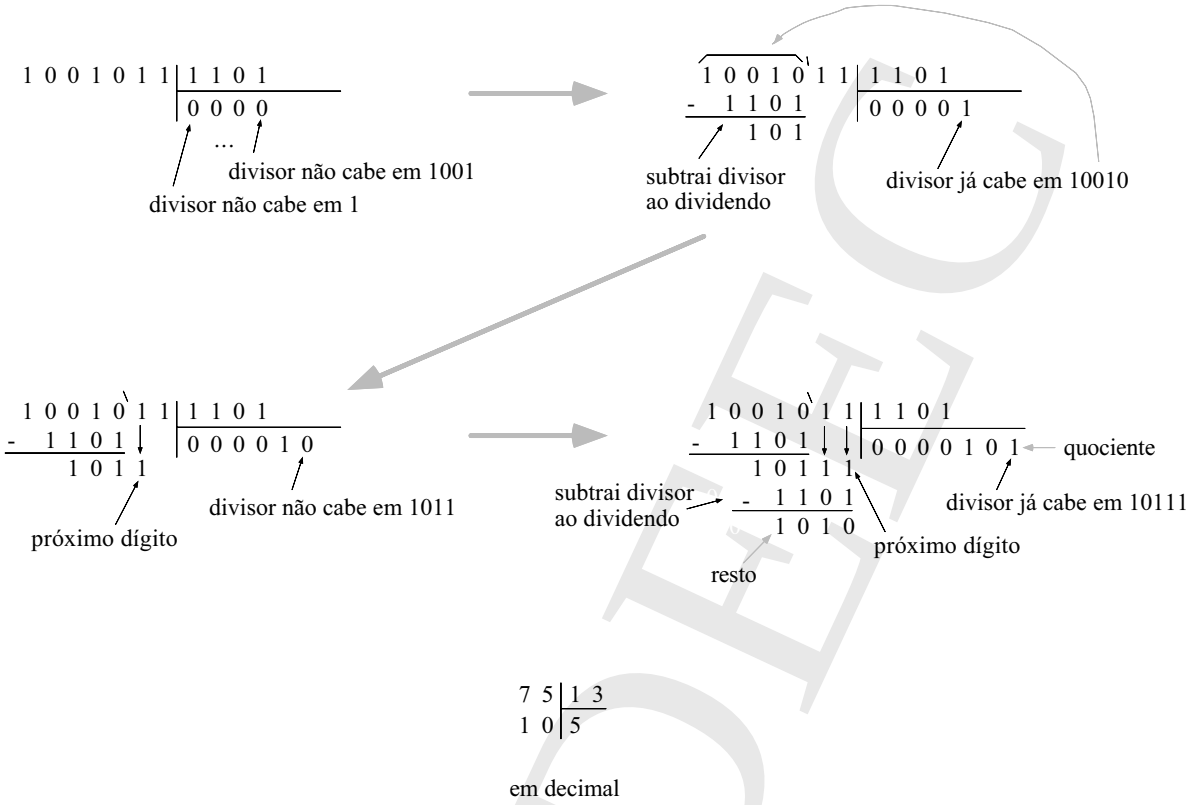


Figura 2.14: Divisão binária.

e operar números inteiros positivos no intervalo $[0, 255]$[3].

Quando são realizadas operações aritméticas com números binários, é normalmente necessário estabelecer o número de *bits* em que o resultado deve ser representado. Se o resultado não "couber" nesse número de *bits* diz-se que é excedida a gama de representação para o número de *bits* considerado e utiliza-se correntemente a palavra inglesa *overflow* para designar esta situação.

Por exemplo, a adição dos números representáveis com 8 *bits* $134_{10} = 10000110_2$ e $221_{10} = 11011101_2$ produz um resultado igual a $355_{10} = 101100011_2$ que não pode ser representado só com 8 *bits*. Naturalmente que se tomarmos apenas os 8 *bits* menos significativos resultantes dessa adição, obtemos $01100011_2 = 99_{10}$ que não representa correctamente o resultado dessa adição.

A situação de *overflow* pode ser identificada na adição binária de números positivos quando ocorre transporte de 1 para além do *bit* mais significativo considerado para o resultado (figura 2.15).

Também na subtracção binária pode ocorrer uma situação idêntica (*overflow*) se o diminuendo for menor do que o diminuidor (experimente efectuar à mão a conta $10 - 13$). Neste caso

[3]O processamento de números maiores requer a programação de funções que tratem números maiores formados por agrupamentos dos números mais pequenos (por exemplo 8 *bits*) que o processador sabe operar

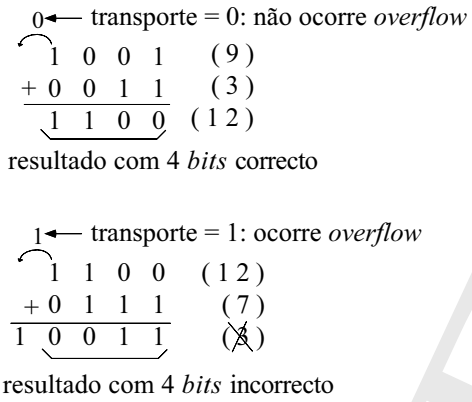


Figura 2.15: Adição binária e *overflow*.

o resultado correcto seria um número negativo mas obrigaria a realizar a subtracção trocando a ordem dos operandos. Efectuando à mão uma subtracção nestas condições verifica-se que ocorre também um "transporte" (ou melhor, um *borrow*) para além do *bit* mais significativo esperado para o resultado (figura 2.16).

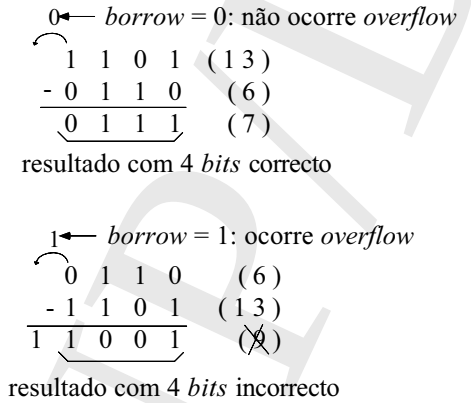


Figura 2.16: Subtracção binária e *overflow*.

Esta situação permite recorrer à subtracção binária para realizar a comparação entre a magnitude de dois números inteiros positivos. Assim, se o resultado de $A-B$ não produzir transporte (*borrow*), então pode-se concluir que $A >= B$; se ocorrer transporte então o resultado não pode representar correctamente essa diferença e conclui-se que $A < B$.

A operação de mutiplicação produz geralmente resultados que necessitam de mais *bits* do que os operandos para serem correctamente representados. De um modo geral, o produto de dois números inteiros positivos com N e M *bits* produz um resultado que pode ocupar até $N+M$ *bits*. Se for considerado para o resultado um número de *bits* L inferior a $N+M$, então ocorrerá

overflow sempre que seja gerado um transporte para além do bit $L-1$.

A divisão binária (inteira) entre um dividendo com *M bits* e um divisor com *N bits* produz como resultado um quociente que nunca excede o número de *bits M* do dividendo, e um resto que é sempre inferior ao divisor e portanto cabe sempre em *N bits*.

2.6 Representação de números negativos

No sistema de numeração decimal que utilizamos correntemente representamos números negativos precedendo o seu valor de um sinal menos ($-$), e números positivos apenas pelo seu valor ou precedendo-o de um sinal mais ($+$).

A esta forma de representação chamamos *sinal e grandeza* (ou sinal e magnitude) e podemos, naturalmente, usá-lo para representar números com sinal em qualquer base de numeração, por exemplo:

$$-134_{10} = -10000110_2 = -86_{16}$$

2.6.1 Sinal e grandeza

No caso particular do sistema binário, o sinal representa-se por um *bit* adicional que é 1 para representar números negativos e 0 para números positivos. Assim, com *N bits* podemos representar números positivos e negativos no intervalo $[-2^{N-1}-1, 2^{N-1}-1]$. Por exemplo, com 8 *bits* podemos representar números com sinal no intervalo $[-127, +127]$. Note-se que, à semelhança do que acontece no sistema decimal, o zero tem também duas representações possíveis (-0 e $+0$). Por exemplo, com 8 *bits*:

$$+67_{10} = 01000011_2 \qquad -67_{10} = 11000011_2$$

$$+13_{10} = 00001101_2 \qquad -13_{10} = 10001101_2$$

As operações aritméticas elementares processam-se da mesma forma que as realizamos no sistema decimal, tratando separadamente o sinal da grandeza. Enquanto que para a multiplicação e divisão a regra é muito simples (se os operandos tiverem o mesmo sinal o resultado é positivo senão é negativo), a realização da operação de adição (ou subtracção) é bastante mais complexa. Em primeiro lugar é necessário decidir qual é realmente a operação a efectuar (adição ou subtracção) entre as grandezas dos operandos, em função do seu sinal; se for feita uma subtracção, é também necessário determinar qual é a maior grandeza para que a subtracção binária produza correctamente o valor do resultado; finalmente é necessário determinar o sinal do resultado que depende também dos sinais dos operandos e das suas grandezas. A necessidade

de realização de todas estas operações, em especial a comparação dos seus valores absolutos e a troca de ordem dos operandos, torna bastante complexa a construção de sistemas digitais que realizem adições com números representados neste sistema. Na figura 2.17 exemplifica-se o processo de adição de números representados em sinal e grandeza.

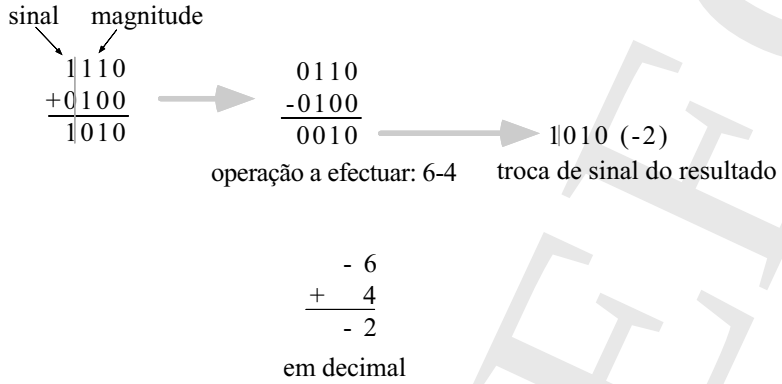


Figura 2.17: Adição binária com números representados em sinal e grandeza.

2.6.2 Complemento para a base

Um sistema mais conveniente para a representação de números com sinal é o designado complemento para a base (ou complemento para dois no caso da base 2). A representação de números com sinal neste sistema permite simplificar a forma como são realizadas as operações de adição e subtracção, sem complicar de forma significativa o processo de multiplicação e divisão. Antes de estudar a aplicação deste sistema a números representados em base 2, vamos estudar como pode ser aplicado ao sistema decimal (chamado complemento para 10).

Complemento para 10

Consideremos um sistema de numeração em que é usado um número fixo de dígitos decimais, por exemplo 4, permitindo representar números inteiros sem sinal entre 0 e 9999. Com o sistema complemento para 10, representa-se uma quantidade negativa $-X$ por um *número positivo* Y obtido como $Y = 10^4 - X = 10000 - X$. Se considerarmos apenas 4 dígitos para representar os resultados, então o resultado de $10000 - X$ coincide com o resultado de $0 - X$. Vamos também considerar que os valores inferiores ou iguais a 4999 representam quantidades positivas, e que os números entre 5000 e 9999 representam valores negativos (note que isto corresponde a dividir o intervalo $[0, 9999]$ sensivelmente a meio, atribuindo metade do intervalo a números positivos e a outra metade a números negativos).

Experimentemos efectuar algumas operações de adição e subtracção com quantidades com sinal representadas em complemento para 10 com 4 dígitos.

Por exemplo, se efectuarmos a subtracção decimal $0-17$ e consideremos para resultado apenas os 4 primeiros dígitos, obtemos um número que é igual ao resultado de $10000-17=9983$:

$$0-17=(9...99)9983$$

$$10000-17=9983$$

Assim, se dissermos que 9983 representa a quantidade -17, como podemos calcular o resultado da soma $(-17)+23=6$? Experimentemos somar a 9983 o número 23, e desprezar qualquer dígito para além dos 4 que estamos a considerar:

$$-17+23=9983+23=(1)0006=6$$

Se adicionarmos agora -17 à quantidade -13 que se representa neste sistema por $10000-13=9987$, devermos obter a quantidade -30:

$$-17+(-13)=9983+9987=(1)9970$$

como o resultado é superior a 4999 podemos concluir que representa uma quantidade negativa cujo valor é:

$$-10000+9970=-30$$

Experimentemos agora calcular $-30-9$ e $-30+(-9)$:

$$-30-9=9970-9=9961=\rightarrow -10000+9961=-39$$

$$-30+(-9)=9970+(10000-9)=9970+9991=9961\rightarrow -10000+9961=-39$$

e finalmente calculemos a diferença $33-47$ e $33+(-47)$:

$$33-47=(9...99)9986=9986=\rightarrow -10000+9986=-14$$

$$33+(-47)=33+(10000-47)=33+9953=9986\rightarrow -10000+9986=-14$$

Estes resultados eram de esperar! Na verdade, podemos imaginar que o "nosso" zero é representado pelo número 10000, e que para além de 10000 representamos quantidades positivas e para baixo representamos quantidades negativas, aproveitando apenas os 4 dígitos menos significativos de cada valor (figura 2.18).

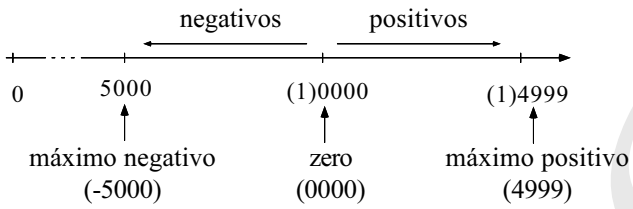


Figura 2.18: Representando números com sinal em complemento para 10 com 4 dígitos.

Neste sistema existe ainda uma indefinição relativamente à forma como se divide o intervalo $[0,9999]$ entre númeroas positivos e números negativos. Na realidade esse limite pode localizer-se onde pretendermos, desde que interpretemos correctamente o "sinal" de um valor numérico representado neste sistema. Podemos, por exemplo, considerar que apenas representamos números positivos entre 0 e 1000, e que os valores entre 1001 e 9999 representam quantidades negativas de acordo com a definição apresentada antes (o valor mais negativo representável neste sistema seria 1001-10000=-8999). No entanto, se se dividir aproximadamente a meio o intervalo $[0,9999]$ obtém-se uma gama de representação de números com sinal igual a:

$$[-10^4/2, 10^4/2-1] = [-5000, +4999]$$

Desta forma o intervalo fica assimétrico, já que podemos representar mais um número negativo do que os positivos: 5000 negativos, de -1 a -5000 e 4999 positivos, de 1 a 4999. No entanto esta divisão é aproxidamente simétrica e permite facilitar a identificação do sinal de um valor representado neste sistema: se o dígito mais significativo for inferior ou igual a 4, então o número é positivo; se for superior ou igual a 5, o número é negativo.

Que vantagens a utilização deste sistema face à representação sinal e grandeza? Em primeiro lugar, adições e subtracções de números com sinal podem ser feitas, como já vimos, sem requerer qualquer tratamento especial do sinal dos operandos para decidir a operação a realizar e o sinal do sresultado. Em segundo lugar, se se tiver disponível um processo para trocar o sinal de um número, as subtracções também podem ser realizadas como adições e não é necessário efectuar qualquer tratamento especial aos sinais dos operandos e resultado. Por outro lado, o tratamento da multiplicação e divisão torna-se mais complexo quando se utiliza uma representação em complemento para a base do que usando a representação sinal e grandeza.

Como se troca o sinal de um número em complemento para 10?

Analisemos agora a operação de subtracção realizada para obter a representação em complemento para 10 de uma quantidade negativa, por exemplo -3452 com 4 dígitos:

$$10^4 - 3452 = 10000 - 3452 = (1 + 9999) - 3452 = (9999 - 3452) + 1 = 6548$$

Note-se que podemos efectuar a diferença $9999 - 3452$ simplesmente subtraindo dígito a dígito, uma vez que nunca ocorre transporte! Se definirmos o *complemento* de um dígito decimal d como $9 - d$ ou $10 - 1 - d$, então podemos estabelecer uma regra prática para "trocar o sinal" de um número representado em complemento para 10: complementam-se todos os dígitos e soma-se 1. Se aplicarmos este processo para calcular o simétrico de -3452, devermos complementar os dígitos de 6548 (que é a representação de -3452 em complemento para 10) obtendo 3451 e adicionando 1 para obter então o resultado final 3452.

Algumas contas mais...

Continuemos a considerar a representação de números com sinal em complemento para 10 com 4 dígitos, e realizemos mais algumas operações aritméticas de adição e subtracção.

A adição de 2 números positivos (ou seja, inferiores a $+4999$) realiza-se normalmente. No entanto, se essa soma ultrapassar o máximo número positivo que podemos representar neste sistema ($+4999$), então dizemos que foi ultrapassada a capacidade do sistema de representação (*overflow* em inglês). Note que um valor superior a $+4999$ deve ser entendido como representando uma quantidade negativa, e naturalmente que a soma de dois números positivos nunca poderá dar um resultado negativo!
Somando dois números positivos obteremos um resultado correcto se este for também positivo:

$$24 + 78 = 102 \quad \text{resultado positivo, correcto}$$

Se a soma de dois números positivos der um valor que representa uma quantidade negativa, então foi excedida a capacidade do sistema de representação:

$$3987 + 2000 = 5987 \quad \text{resultado negativo errado, } \textit{overflow}$$

Somando um número positivo com um negativo (ou subtraindo dois números positivos, o que é equivalente), nunca se excede a capacidade do sistema de representação (note que apenas devem ser considerados para o resultado os 4 dígitos menos significativos)

$$(-2378) + 3690 = (10000 - 2378) + 3690 = 7622 + 3690 = (1)1312 = 1312$$

Adicionando dois números negativos também deveremos obter um resultado negativo, caso contrário é excedida a capacidade do sistema de representação. Por exemplo, se somarmos -2378 com -690

$$(-2378)+(-690)=(10000-2378)+(10000-690)=7622+9310=16932=6932$$

Obtemos um resultado superior a 5000, o que quer dizer que representa uma quantidade negativa igual a $-(10000-6932)=-3068$.

Se o resultado da adição de dois valores negativos der um número positivo (inferior ou igual a 4999), então podemos afirmar que foi excedida a capacidade do sistema de representação (ocorre *overflow*):

$$(-2378)+(-3690)=(10000-2378)+(10000-3690)=7622+6310=13932=3932$$

2.6.3 Complemento para dois

Complemento para dois é um processo para representar números com sinal em base dois, e consiste em aplicar os mesmos princípios que vimos na secção anterior para a representação de quantidades com sinal em base 10.

De forma semelhante ao que vimos para base 10, é também necessário estabelecer o número de dígitos (ou *bits* no caso do sistema binário) em que serão representados e operados números, sendo apenas considerados como válidos esses dígitos como resultado de operações de adição ou subtracção.

Assim, com N *bits* em complemento para dois, representa-se um valor negativo $-X$ por uma quantidade *positiva* Y obtida como $Y=2^N-X$.

A gama de representação de números com sinal utilizendo N *bits* e complemento para dois é (dividindo o intervalo a meio):

$$[-2^N/2,+2^N/2-1]=[-2^{N-1},+2^{N-1}-1]$$

Por exemplo, com 8 *bits* podem-se representar números com sinal em complemento para dois no intervalo $[-128,+127]$ (ou em binário $[10000000,01111111]$).

Note-se que o sinal de um número pode ser facilmente identificado pelo valor do *bit* mais significativo: 0 significa positivo e 1 representa um número negativo. No entanto, ao contrário da representação sinal e grandeza, este *bit* não representa apenas o sinal do número mas contribui também para o seu valor.

O valor de um número com sinal representado em complemento para dois com N *bits* pode ser determinado por um processo semelhante ao que usamos para base 10. Se o número é positivo (então o seu *bit* mais significativo B_{N-1} é zero e o número tem a forma $0B_{N-2}B_{N-3}\ldots B_1B_0$), o seu valor X é o valor representado pelo número binário:

$$X = 2^{N-1}.B_{N-1} + 2^{N-2}.B_{N-2} + ... + 2^2.B_2 + 2^1.B_1 + 2^0.B_0$$

onde a primeira parcela é zero porque B_{N-1} é zero (número positivo).

Se o seu *bit* mais significativo for 1 (representado por $1B_{N-2}B_{N-3}...B_1B_0$) então o número é negativo. Pela definição de complemento para dois, a relação entre o valor binário Y do número $1B_{N-2}B_{N-3}...B_1B_0$ e o valor (negativo) que ele representa em complemento para dois com N *bits* $-X$ é dada por:

$$Y = 2^N - X$$

onde Y representa o valor do número binário $1B_{N-2}B_{N-3}...B_1B_0$:

$$-X = Y - 2^N = 2^{N-1}.B_{N-1} + 2^{N-2}.B_{N-2} + ... + 2^2.B_2 + 2^1.B_1 + 2^0.B_0 - 2^N$$

$$-X = (2^{N-1}.B_{N-1} - 2^N) + (2^{N-2}.B_{N-2} + ... + 2^2.B_2 + 2^1.B_1 + 2^0.B_0)$$

Como $2^{N-1} - 2^N = -2^{N-1}$ podemos escrever:

$$-X = -2^{N-1}.B_{N-1} + 2^{N-2}.B_{N-2} + ... + 2^2.B_2 + 2^1.B_1 + 2^0.B_0$$

A expressão anterior atribui um peso de -2^{N-1} ao *bit* mais significativo e pode ser usada para obter o valor de um número representado em complemento para dois com N *bits*, quer seja positivo quer seja negativo: se é positivo, então o *bit* mais significativo B_{N-1} é zero e o seu valor coincide com o valor representado pelos *bits* restantes; se o número é negativo então o *bits* mais significativo é 1 e contribui com o peso -2^{N-1} no valor do número.

Consideremos agora alguns exemplos tendo por base o sistema de representação de números com sinal complemento para dois com 4 *bits* (representando números no intervalo $[-8, +7]$). O valor do número positivo 0101 pode ser calculado como:

$$0101_2 = -2^3 \times 0 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = +5_{10}$$

e o valor do número negativo 1101:

$$1101_2 = -2^3 \times 1 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = -8 + 5 = -3_{10}$$

Outro modo de calcular o valor representado por 1101 consiste em aplicar a definição de número negativo neste sistema: um valor negativo $-X$ é representado pelo valor positivo $Y = 2^N - X$ primeiro obtemos o valor positivo Y do número binário 1101 e depois subtraímos a esse valor $2^4 = 16$ para calcular o valor (negativo) $-X$ que ele representa:

$$1101_2 = 2^3 \times 1 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 13_{10}$$

$$13 = 2^4 - X \rightarrow -X = 13 - 16 = -3_{10}$$

De forma semelhante ao que apresentamos para o sistema complemento para 10, podemos também considerar que os números positivos crescem a partir de zero (ou de $2^4 = 10000_2$), e os negativos decrescem a partir de $2^4 = 10000_2$ (ou de zero, se desprezarmos o *bit* 1 da esquerda). A figura 2.19 mostra a distribuição dos números positivos e negativos e a sua corresponência com os números binários que os representam.

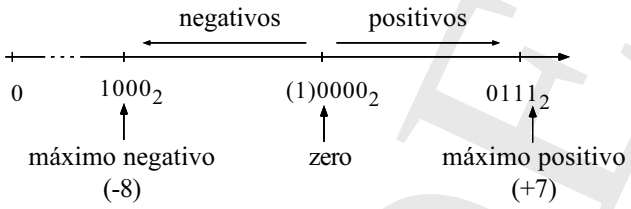


Figura 2.19: Representação de valores com sinal em complemento para 2 com 4 *bits*.

Trocar o sinal de um número em complemento para dois

O processo para trocar o sinal de um número representado em complemento para dois é semelhante ao que apresentamos antes para complemento para 10.

Tomemos como exemplo o número $X = 0101_2 = 5_{10}$, representado em complemento para dois com 4 *bits*. O seu simétrico (i.e. $-X = -5$) é representado por um número binário com 4 *bits* Y calculado como:

$$Y = 2^4 - X = 2^4 - 0101_2 = 10000_2 - 0101_2 = 1011_2$$

Note que o número binário 10000 pode ser escrito como $1111 + 0001$ e que subtrair um número binário de 1111 corresponde a trocar o valor de todos os seus *bits*. Podemos assim estabelecer uma regra prática para "trocar" o sinal de um número binário representado em complemento para dois: trocam-se os *bits* todos e adiciona-se 1. Assim, a representação do simétrico de $+5$ (em binário 0101_2) pode escrever-se como:

$$10000 - 0101 = (1111 + 0001) - 0101 = (1111 - 0101) + 0001 = 1010 + 0001 = 1011$$

Alguns exemplos (em complemento para dois com 4 *bits*):
O valor -2 representa-se por um número positivo de 4 *bits* obtido por:

$$2^4 - 2 = 10000_2 - 0010_2 = 1110_2 = 14_{10}$$

O seu simétrico ($+2$) pode ser calculado trocando os *bits* (0001) e somando 1:

$$0001_2 + 1 = 0010 = 2_{10}$$

Qual é o valor representado pelo número binário 1001, em complemento para dois com 4 *bits*?

$$-2^3 + 2^0 = -8 + 1 = -7_{10}$$

Como se representa $+7$? Se -7 é representado por 1001, então aplicando a regra prática trocamos os *bits* todos (0110) e somamos 1 para obter 0111, que é a representação binária de $+7$.

Adição e subtracção com números em complemento para dois

Como vimos anteriormente para números com sinal representados em complemento para 10, também em complemento para dois as operações de adição e subtracção são realizadas "normalmente", sem qualquer tratamento especial do sinal dos operandos. É importante relembrar que a representação de valores com sinal em complemento para dois, bem como os resultados obtidos com a realização de operações aritméticas, pressupõe um número de *bits* determinado para a sua representação e deve ser ignorado o transporte que é gerado para além do *bit* mais significativo considerado nesse sistema de representação.

Tal como vimos nos exemplos apresentados em complemento para 10, também podemos detectar situações de *overflow* na realização de adições binárias analisando os sinais dos operandos e do resultado: se os operandos tiverem sinais opostos (*bits* de sinal contrários), então nunca pode ocorrer *overflow*; se os operandos tiverem o mesmo sinal, o resultado excede a gama de representação (ocorre *overflow*) quando o seu sinal for diferente do sinal dos operandos. Outra forma de identificar esta situação consiste em comparar os valores dos *bits* de transporte que saem do penúltimo *bit* e do último *bit*: se forem iguais o resultado está correcto, caso contrário ocorre *overflow*. Esta forma para detectar esta situação é mais económica do que a anterior já que apenas requer a comparação de dois *bits*, enquanto que para analisar os *bits* de sinal é necessário comparar entre si 3 *bits*. Note que quando se adicionam números representados em complemento para dois o transporte gerado para fora do *bit* mais significativo não representa *overflow*.

Na figura 2.20 mostram-se alguns exemplos de adições binárias com números representados em complemento para dois com 4 *bits*. Note-se que utilizando uma representação de números

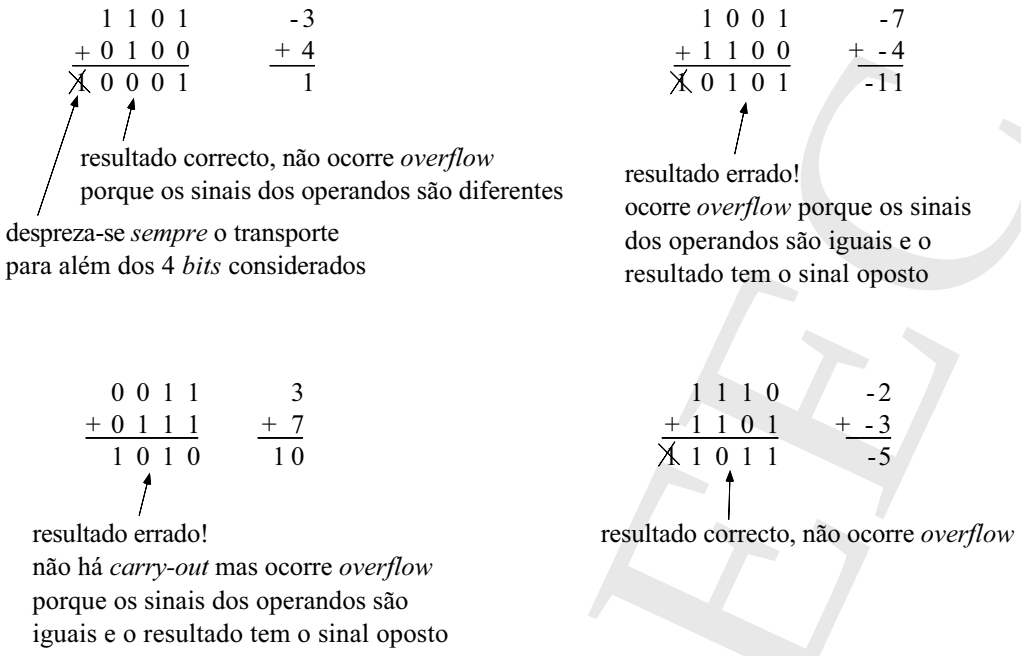


Figura 2.20: Adição de números em complemento para dois com 4 *bits*.

com sinal e dispondo de uma operação para obter o simétrico de um valor (i.e. trocar o seu sinal), apenas é necessário realizar operações de adição já que basta trocar o sinal do diminuidor para efectuar uma subtracção (por exemplo, $6-4$ pode escrever-se como $=6+(-4)$ e ser realizada como uma adição).

Operando números com diferentes tamanhos

Quando se efectua uma adição binária com números representados em complemento para dois, os dois operandos e o resultado devem ter o mesmo número de *bits*. Se os valores a somar tiverem números diferentes de *bits* então o resultado terá, no máximo, o maior número de *bits* de entre os dois operandos. Neste caso é necessário representar o menor operando com o mesmo número de *bits* do maior operando. Se o valor é positivo, então basta acrescentar zeros à esquerda mantendo o valor do seu *bit* de sinal. No entanto, se esse valor é negativo o seu *bit* mais significativo é 1 e é necessário acrescentar uns à esquerda de forma a preservar o seu sinal. A esta operação chama-se *extensão de sinal* já que consiste em estender o *bit* de sinal de um número para completar o número de *bits* requeridos para a realização de uma operação aritmética (figura 2.21).

Note-se que se se representar o valor 1011 (-5_{10} com 4 *bits* em complemento para dois) com 6 *bits*, deve escrever-se 111011 já que se for completado com 2 zeros à esquerda (001011) passa a representar um valor positivo ($+11_{10}$).

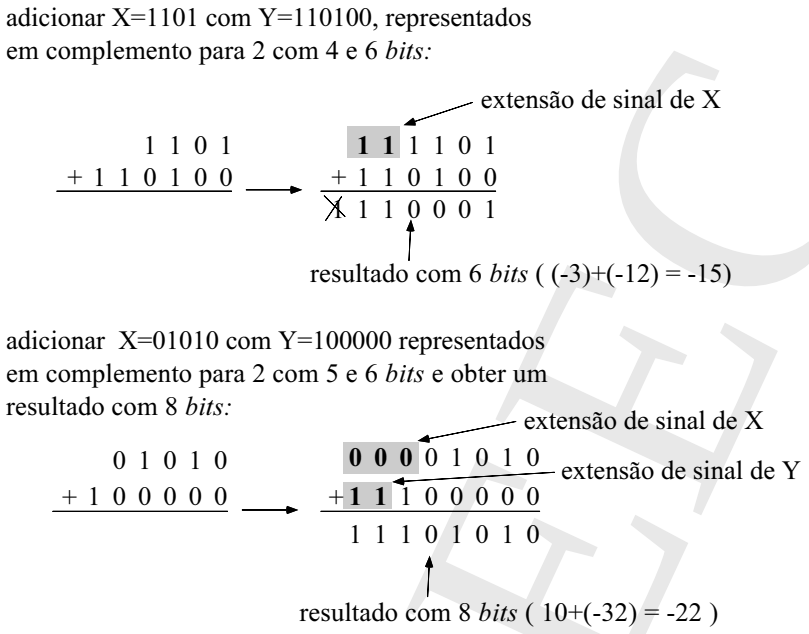


Figura 2.21: Adição com números de diferentes números de *bits* e extensão de sinal.

2.7 Representação binária de números decimais (*BCD*)

A forma mais comum de representar valores numéricos em sistemas digitais consiste na utilização do sistema de numeração binário apresentado nas secções anteriores. A utilização deste sistema é vantajosa porque permite utilizar de forma eficiente os diferentes códigos representados por um determinado número de *bits*, e também porque é fácil construir os circuitos digitais que realizam as operações aritméticas elementares entre estes tipos de dados.

No entanto, quando valores numéricos são apresentados em algum dispositivo de saída (um ecran, por exemplo) para serem percebidos por um humano, devem ser mostrados em formato decimal (imagine uma caixa registadora electrónica de um supermercado a mostrar o preço dos artigos em formato binário ou hexadecimal...). O processo para determinar os dígitos decimais que formam um determinado valor requer a realização de uma série de divisões por 10 que requerem circuitos bastante mais complexos do que somadores ou subtractores binários.

Outra forma de representar números usando o sistema binário consiste em representá-los como uma série de dígitos decimais, cada um representado como um valor de 4 *bits* (entre 0 e 9). Por exemplo o número decimal 3576_{10} pode escrever-se como uma sequência de 4 grupos de 4 *bits* (um total de 16 *bits*), em que cada um representa um dígito decimal: 0011 0101 0111 0110. Chama-se a este formato BCD (do inglês *Binary-Coded Decimal*).

Uma das desvantagens do formato BCD é a má utilização dos *bits* empregues para representar um número. Usando o sistema de numeração binário podemos representar com 16 *bits*

números sem sinal entre 0 e 65535, mas no formato BCD apenas é possível representar valores entre 0 e 9999, já que nos 4 *bits* que representam cada dígito decimal não são usados os códigos entre 1010 e 1111. Outra desvantagem é a complexidade acrescida dos circuitos digitais que realizam as operações aritméticas elementares com números representados neste formato.

A adição de dois números com 4 *bits* representados em BCD (dois dígitos decimais) efectua-se normalmente como uma adição binária, mas se o resultado for superior a 9 (1001_2) então é necessário somar o valor 6 (0110_2) para corrigir esse resultado. A soma de números com vários dígitos decimais representados em BCD é bastante mais complexa porque obriga à realização, em sequência, das adições e correcções dos dígitos decimais que compõem o número à medida que se adicionam os seus dígitos. A figura 2.22 exemplifica a aplicação deste processo.

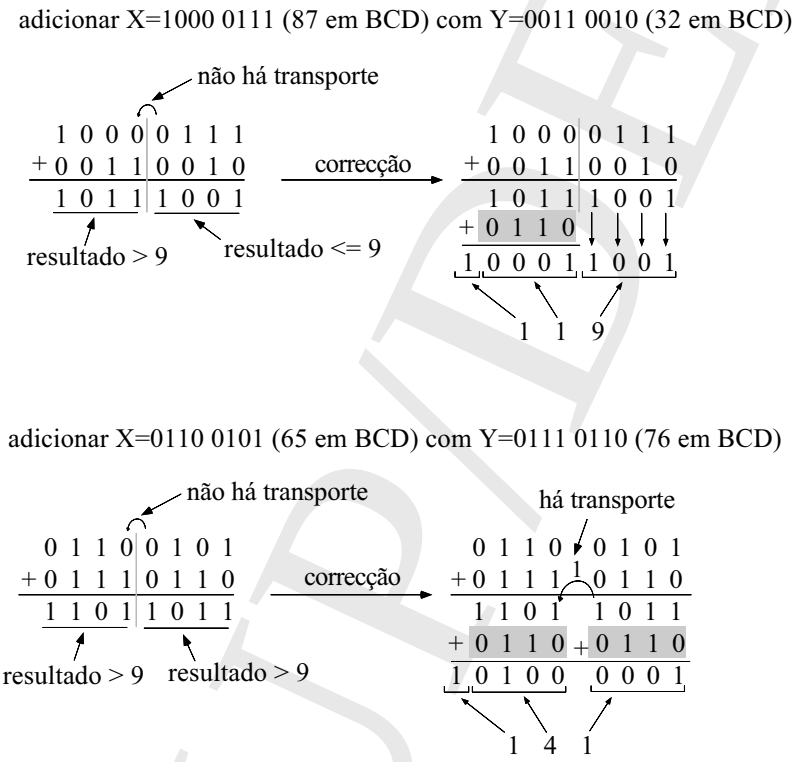


Figura 2.22: Adição de números representados em BCD.

A representação de números com sinal no formato BCD pode ser feita usando uma notação sinal e grandeza ou então o sistema complemento para 10 que se apresentou antes.

A utilização do formato BCD tem interesse quando se pretende evitar as operações de divisão por 10 para obter os dígitos decimais que representam um número, e quando as operações aritméticas a realizar sobre esses valores são simples. Por exemplo, num relógio digital, os valores que representam os segundos, minutos e horas podem ser facilmente representados por dois dígitos decimais e a única operação aritmética que é realizada com esses valores é adicionar-lhes

1 unidade. Usando a codificação BCD não é necessário realizar divisões para obter os dígitos decimais desses valores, o que é necessário para afixar num mostrador de LEDs, por exemplo.

Capítulo 3

Álgebra de Boole

Como foi visto no capítulo 1, o comportamento de um sistema digital é estabelecido pelas relações existentes entre as entradas e as saídas, de forma a realizar uma determimnada tarefa. Essa relação pode ser descrita em linguagem natural (i.e. em Português), com proposições que estabelecem em que condições das entradas as saídas são 1 ou 0. O projecto (ou *síntese*) de um circuito digital consiste em traduzir a descrição que especifica o comportamento do sistema para um *circuito lógico* formado por um conjunto de componentes electrónicos que implementam as 3 funções lógicas elementares: E, OU e NÃO (ou, em Inglês *AND*, *OR* e *NOT*).

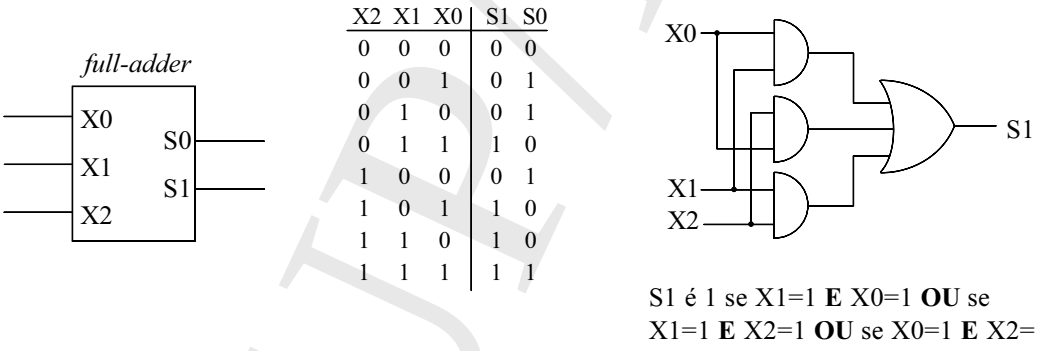


Figura 3.1: Somador completo (*full-adder*): símbolo, tabela de verdade e circuito lógico.

Tomemos como exemplo um circuito lógico que realiza a adição de 3 números de 1 *bit* ($X0$, $X1$ e $X2$), produzindo um resultado de 2 *bits* ($S1$ e $S0$) que pode assumir os valores 0, 1, 2 ou 3. Este circuito chama-se somador completo (em inglês *full-adder*) e é uma peça fundamental usada na construção de circuitos que realizam operações aritméticas. O seu comportamento é representado pela tabela de verdade mostrada na figura 3.1. Para construir um circuito lógico que implemente essa função podemos tentar escrever em Português as condições para as quais

as saídas são 1 e 0. Por exemplo, para a saída $S1$ que representa o *bit* mais significativo do resultado, pode-se escrever: $S1$ é 1 quando duas ou mais entradas forem iguais a 1. Podemos escrever esta relação de uma forma que torne mais explícitas as operações lógicas elementares: $S1$ vale um se $X1 = 1\ E\ X0 = 1\ OU$ se $X2 = 1\ E\ X0 = 1\ OU$ então se $X2 = 1\ E\ X1 = 1$; nos outros casos $S1$ vale zero. Esta expressão pode ser facilmente traduzida num circuito lógico com portas AND e OR, da forma que se mostra na figura 3.1.

Apesar de neste exemplo ser simples descrever a funcionalidade de uma função à custa de Es e OUs, na generalidade das situações é demasiado complexo projectar um circuito digital seguindo uma abordagem semelhante. A álgebra de Boole estabelece um conjunto de definições e regras que permitem relacionar de forma formal as entradas e saídas de um sistema digital e traduzir o seu comportamento para um conjunto de expressões algébricas que podem ser manipuladas de forma rigorosa, de forma semelhante a expressões aritméticas. As regras (os axiomas e teoremas) da ágebra booleana[1] permitem manipular estas expressões de forma a transforma-las para, por exemplo, possibilitar a realização de uma mesma função lógica com um conjunto diferente de circuitos electrónicos que permita reduzir o seu custo.

A álgebra de Boole foi inventada em 1854 pelo matemático Inglês George Boole, muito antes da invenção do computador digital. Com esta álgebra pretendeu estabelecer um conjunto universal de regras para combinar proposições que podem ser verdadeiras ou falsas e avaliar a sua veracidade ou falsidade. Proposições são combinadas usando E, OU e NÃO, da mesma forma que se mostrou acima para especificar o comportamento da função de somador completo.

Em 1938, na altura em que se davam os primeiros passos para o nascimento dos computadores digitais, Claude Shannon adaptou a álgebra de Boole para descrever o funcionamento de circuitos eléctricos construídos com interruptores comandados electricamente (relés). Fazendo corresponder o estado de um interruptor a uma variável booleana, esta só pode assumir dois valores diferentes (ligado ou desligado representado 1 ou 0) que podem ser associados aos valores lógicos verdadeiro ou falso da álgebra de Boole.

3.1 Axiomas e teoremas

Os axiomas e os teoremas constituem o conjunto de definições que estabelecem as regras de operação da álgebra. Axiomas representam o conjunto mínimo de regras que se em fundamenta toda a álgebra, e que são assumidos como verdadeiros (i.e. não se podem demonstrar). Os axiomas da álgebra de Boole começam por estabelecer que, nesta álgebra, uma variável apenas

[1] Boole é o nome do inventor desta álgebra e como tal deve ser escrito com letra maiúscula; é comum utilizar o termo “expressão booleana” ou “variável booleana” para referir uma expressão ou variável que segue as regras desta álgebra

pode assumir os valores 0 ou 1; em seguida definem as três operações lógicas já referidas. Com base nos axiomas pode-se construir um conjunto de teoremas que são relações que, uma vez demonstradas com recurso aos axiomas ou outros teoremas, pode ser aplicados na manipulação de expressões algébricas.

3.1.1 Axiomas

Na figura 3.2 apresenta-se o conjunto de axiomas que definem a álgebra de Boole. O par de axiomas A1 e A1' diz que uma variável booleana (X) apenas pode assumir os valores 0 ou 1 e os axiomas A2 a A5 (A2' a A5') estabelecem as regras de operação para os três operadores elementares da álgebra de Boole: os axiomas A2 e A2' definem a operação negação (NÃO, complemento ou o oposto), a operação E é definida pelos axiomas A3, A4 e A5 e os axiomas A3', A4' e A5' estabelecem as regras de operação da função OU.

A1:	$X = 0$ se $X \neq 1$	**A1':**	$X = 1$ se $X \neq 0$
A2:	se $X = 0$ então $\overline{X} = 1$	**A2':**	se $X = 1$ se $\overline{X} = 0$
A3:	$0.0 = 0$	**A3':**	$1+1 = 1$
A4:	$1.1 = 1$	**A4':**	$0+0 = 0$
A5:	$0.1 = 1.0 = 0$	**A5':**	$1+0 = 0+1 = 1$

Figura 3.2: Axiomas da álgebra de Boole.

A operação NÃO é geralmente representada como uma barra horizontal sobre o seu argumento ou então com uma pelica após o seu operando (por exemplo, o oposto de X pode-se escrevr $\overline{X}$ ou X').

Pelas (algumas) semelhanças que apresentam com as operações aritméticas adição e multiplicação, é comum chamar *produto lógico* à função E e *soma lógica* à função OU, e representá-las por mesmos símbolos que se utilizam para representar a adição (+) e a multiplicação (.). Também à semelhança do que acontece na escrita de expressões aritméticas, considera-se que o produto lógico (E) tem prioridade sobre a soma lógica (OU), e podem ser usados parêntesis para alterar essa prioridade "natural" dos operadores. Apresentam-se a seguir alguns exemplos de expressões booleanas, onde X, Y e Z são variáveis booleanas (i.e. que verificam os axiomas A1 e A1'):

$$(X + Y').(Z.(X + Z)')$$

$$X + Y'.Z.X + Z'$$

$$\overline{(Z.X + Y)} + Z.(Y + \overline{X})$$

Os operadores elementares da álgebra de Boole podem ser representados graficamente utilizando símbolos que têm entradas onde são colocadosos valores a operar (as variáveis independentes) e uma saída que representa o valor da função. Uma expressão booleana complexa pode ser representada como um *circuito lógico* ligando entre si vários destes símbolos de forma a construir a função pretendida (figura 3.3). Os símbolos usados para representar as portas E e OU podem também ser desenhados com um número de entradas superior a 2. Por exemplo, a função $W.X.Y.Z$ pode ser desenhada como uma porta E com 4 entradas.

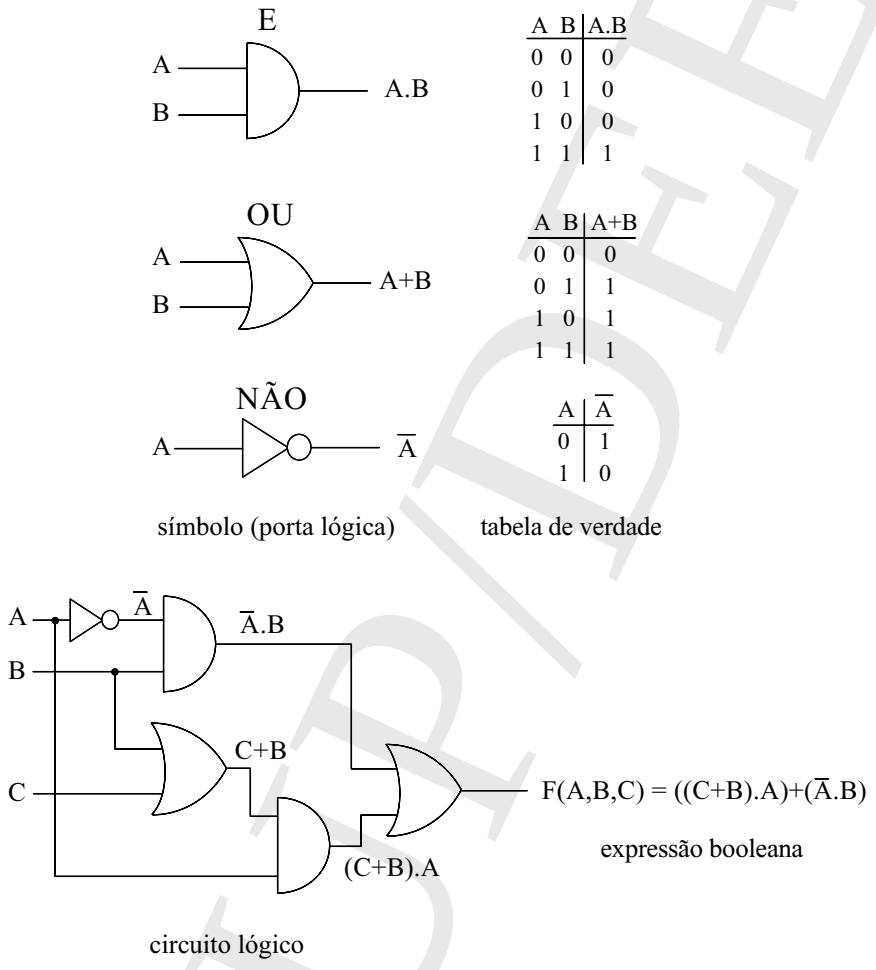


Figura 3.3: As portas lógicas que representam as operações fundamentais da álgebra booleana e sua aplicação na construção de um circuito lógico.

3.1.2 Teoremas

Os teoremas são relações entre expressões booleanas que se considera como verdadeiras e que se podem demonstrar com recurso aos axiomas e a outros teoremas que tenha já sido demonstrados antes. Um conjunto básico de teoremas facilita a manipulação de expressões algébricas,

já que não é necessário basear todas as transformações algébricas nos axiomas. Na figura 3.4 mostram-se os teoremas da álgebra de Boole envolvendo uma, duas e três variáveis. Note-se que os teoremas são apresentados por ordem crescente de complexidade, o que permite fundamentar em teoremas já demonstrados a veracidade de outros teoremas mais complexos.

T1:	$X+0=X$	**T1':**	$X.1=X$
T2:	$X+1=1$	**T2':**	$X.0=0$
T3:	$X+X=X$	**T3':**	$X.X=X$
T4:	$(X')'=X$		
T5:	$X+X'=1$	**T5':**	$X.X'=0$
T6:	$X+Y=Y+X$	**T6':**	$X.Y=Y.X$
T7:	$(X+Y)+Z=X+(Y+Z)$	**T7':**	$(X.Y).Z=X.(Y.Z)$
T8:	$X.Y+X.Z=X.(Y+Z)$	**T8':**	$(X+Y).(X+Z)=X+Y.Z$
T9:	$X+X.Y=X$	**T9':**	$X.(X+Y)=X$
T10:	$X.Y+X.Y'=X$	**T10':**	$(X+Y).(X+Y')=X$
T11:	$X.Y+X'.Z+Y.Z=X.Y+X'.Z$	**T11':**	$(X+Y).(X'+Z).(Y+Z)=(X+Y).(X'+Z)$
T12:	$(X.Y)'=X'+Y'$	**T12':**	$(X+Y)'=X'.Y'$

Figura 3.4: Teoremas fundamentais da álgebra de Boole.

Princípio da dualidade

De forma semelhante com o que foi referido para os axiomas, também os teoremas aparecem aos pares: T1 e T1', T2 e T2', etc. Note que o teorema "linha" pode ser obtido à custa do teoremas "sem linha" trocando entre si os operadores soma lógica (+) e produto lógico (.), e também os valores lógicos 0 e 1 (as negações mantêm-se inalteradas). A esta propriedade chama-se *princípio da dualidade* e pode ser aplicada a qualquer igualdade envolvendo expressões booleanas. À relação resultante chama-se *dual*: por exemplo, os teoremas T1' e T2' são os duais dos teoremas T1 e T2. Note-se que isto apenas é verdade quando aplicado a relações de igualdade entre expressões booleanas e não a expressões isoladas. Assim, se é verdade que:

$$Z.Y+(X+Y')'=(Z+X').Y$$

então, pelo princípio da dualidade, pode-se afirmar que também é verdade:

$$(Z+Y).(X.Y')'=(Z.X')+Y$$

Note-se a necessidade de acrescentar parêntesis na expressão do lado esquerdo para manter a mesma ordem de avaliação das operações.

Leis de DeMorgan

As leis de DeMorgan estabelecem as relações que se apresentaram na figura 3.4 como teoremas T12 e T12':

$$(X+Y)' = X'.Y'$$

e também, pelo princípio da dualidade:

$$(X.Y)' = X'+Y'$$

Esta regra pode ser generalizada a uma função qualquer com N variáveis, permitindo obter a sua negação apenas substituindo cada variável pela sua negação, e trocando entre si os operadores E e OU:

$$[f(X1,X2,X3,...,Xn,+,.)]' = f(X1',X2',X3',...,Xn',.,+)$$

Na figura 3.5 exemplifica-se a aplicação deste teorema a uma expressão booleana e ao circuito lógico correspondente.

Aplicação dos teoremas

O conjunto de axiomas e teoremas apresentados constituem um conjunto de regras que permitem manipular expressões booleanas. Um operação importante, no contexto do projecto de sistemas digitais, consiste na simplificação de expressões de forma a construir circuitos electrónicos que realizem uma função pretendida e que, naturalmente, sejam o mais simples possível. Na figura 3.6 exemplifica-se a aplicação de alguns dos teoremas apresentados para transformar uma expressão booleana "complexa" noutra muito mais simples.

3.2 Representação de funções

Uma expressão booleana envolvendo N variáveis define uma função booleana com N variáveis. Para cada uma das 2^N possíveis combinações para os valores lógicos das N variáveis, o valor da função pode ser apenas 0 ou 1. Ao contrário do que acontece com funções reais de variável real (as que se estudam em Análise Matemática), o domínio de uma função booleana é um conjunto discreto constituído por 2^N *estados* (todas as combinações possíveis das N variáveis), e o contra-domínio é o conjunto $\{0,1\}$.

Como já se viu, uma função booleana pode ser representada de outras formas, para além de expressões algébricas:

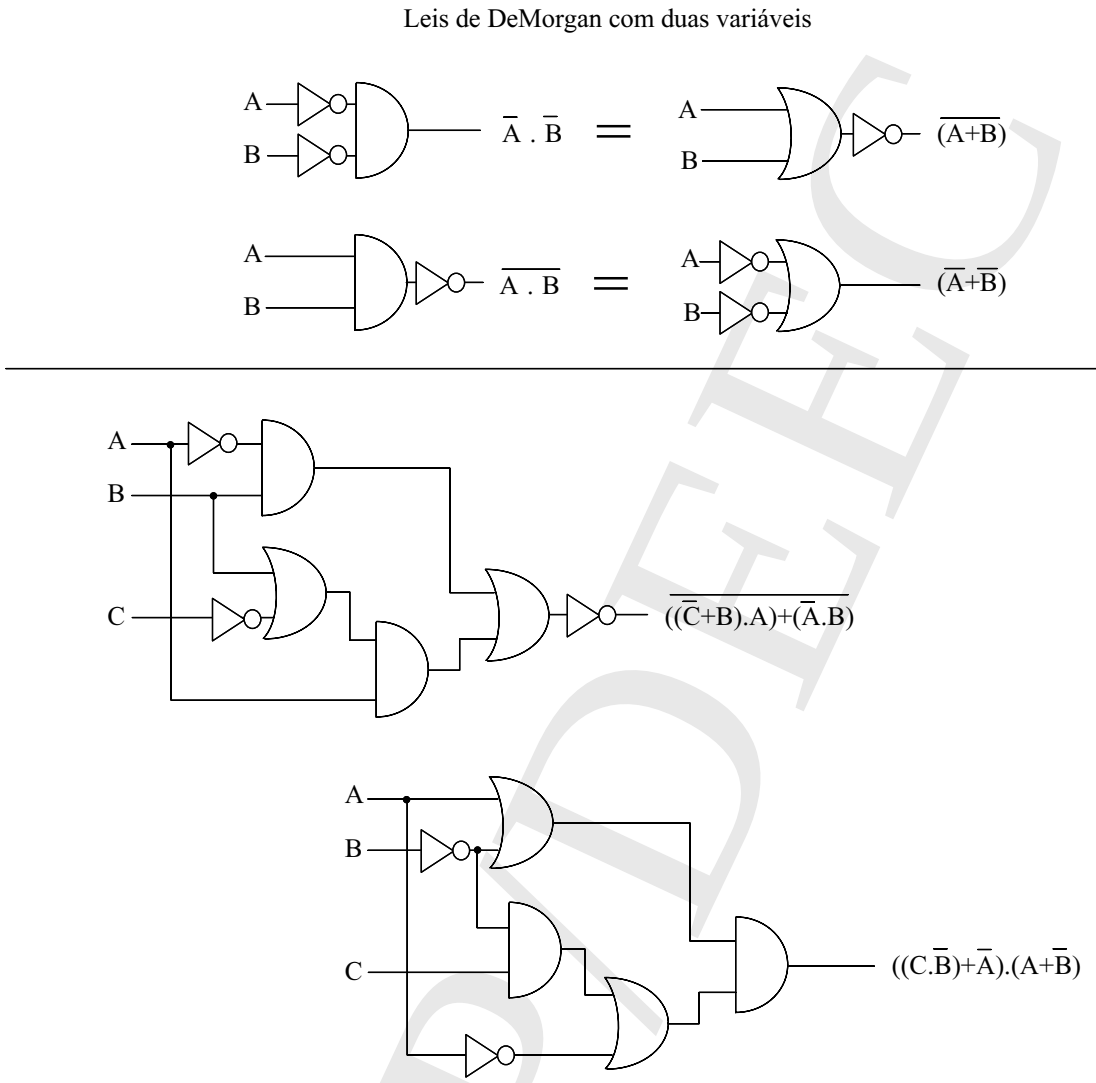


Figura 3.5: Leis de DeMorgan e sua aplicação para obter a negação de uma função.

Provar que (A.B+(A'+B)')+A.C = A

(A.B+(A'+B)')+A.C	=	(Teorema T12 - Leis de DeMorgan)
A.B+A.B'+A.C	=	(Teorema T8)
A.(B+B')+A.C	=	(Teorema T5)
A.1+A.C	=	(Teorema T1')
A+A.C	=	(Teorema T9)
A		

Figura 3.6: Aplicação de teoremas de álgebra de Boole para simplificar uma expressão booleana.

- **Circuito lógico**: "tradução" da expressão algébrica para uma representação gráfica formada pela interligação de símbolos (portas lógicas) que representam os operadores elementares da álgebra de Boole (figura 3.7). É regra comum desenhar as *entradas* do circuito (as variáveis independentes da função) do lado esquerdo e a saída (a função) do lado direito do esquema.

- **Tabela de verdade**: tabela onde se representa o valor da função para cada combinação das variáveis independentes (figura 3.7. As 2^N combinaçãoes das variáveis independentes devem ser escritas segundo a ordem natural dos números binários representados pelos valores das variáveis da função.

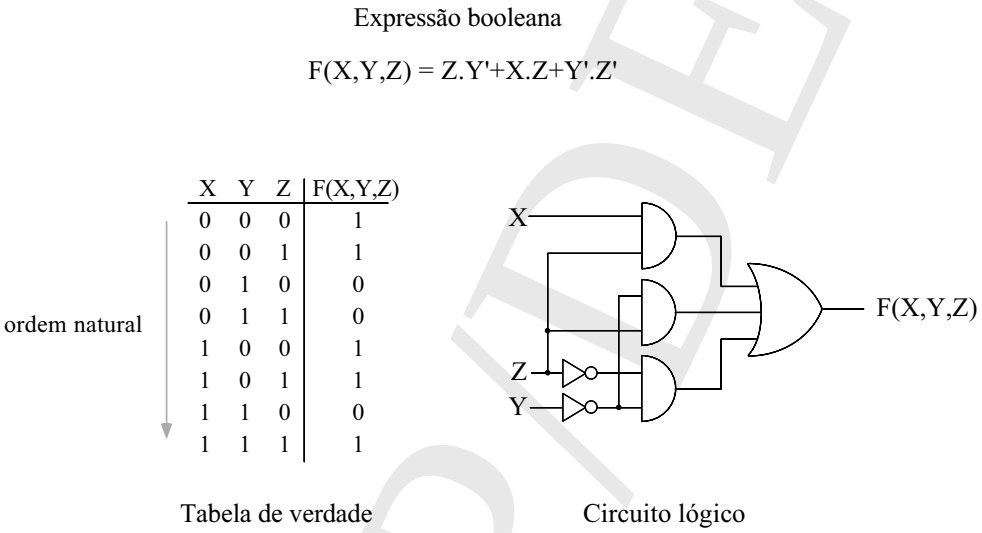


Figura 3.7: Representação de funções booleanas: expressão algébrica, tabela de verdade e circuito lógico.

Uma função booleana pode ser representada por uma expressão algébrica arbitrária, i.e. sem seguir qualquer estrutura pré-definida. Existem no entanto formas padrão de as representar e que têm uma correspondência directa com a sequência de zeros e uns que formam a sua tabela de verdade.

Para auxiliar a sua apresentação é conveniente introduzir um conjunto de definições prévias. Tirando partido do princípio da dualidade, serão apenas apresentadas as relativas ao operador E. As relações correspondentes para o operador OU serão abordadas mais à frente.

- **Literal** é uma variável ou a sua negação. Embora na matemática seja corrente utilizar apenas uma letra para representar uma variável, é aconselhável usar para variáveis booleanas nomes que tenham algo a ver com o seu significado. São exemplos de literais:

$$X, \quad Y', \quad \overline{ACK}, \quad READY$$

- **Termo de produto** é um literal ou o produto lógico de vários literais:

$$X, \quad Y'.W.Y.X, \quad \overline{ACK}.READY, \quad X.X.X$$

- **Soma de produtos** é um termo de produto ou então a soma lógica de mais do que um termos de produto:

$$Y, \quad X+Y', \quad Y'.W.X+Y, \quad \overline{ACK}.X+READY.\overline{Y}$$

- **Termo normal** de produto é um termo de produto em que cada literal não aparece mais de uma vez (por exemplo $X.X.Y$ não é um termo normal):

$$X, \quad Y'.W.X, \quad \overline{ACK}.READY$$

- **Termo mínimo**[2] de N variáveis é um termo normal de produto com N variáveis. Um termo mínimo só vale 1 para uma e uma só combinação das suas variáveis. Exemplos de termos mínimos com 3 variáveis X, Y e Z são:

$$X.Y.Z, \quad X.Y'.Z', \quad X'.Y'.Z'$$

Como cada termo mínimo corresponde exactamente a uma linha da tabela de verdade de uma função, pode ser identificado pelo número de ordem dessa linha da tabela, contando a partir de zero. Este número é igual ao valor do número binário formado pelos valores das variáveis nessa linha. Assim, o termo mínimo $X.Y'.Z'$ que só vale 1 quando for $X=1$ e $Y=0$ e $Z=0$, corresponde à linha 4 da tabela de verdade onde $XYZ = 100_2 = 4_{10}$ (figura 3.8).

Soma canónica

Tendo uma função booleana representada como uma tabela de verdade, pode-se obter imediatamente uma expressão algébrica realizando a soma lógica dos termos mínimos para os quais a função vale 1. A expressão assim obtida chamada *soma canónica* ou *expressão canónica soma-de-produtos* e pode ser tomada como o ponto de partida para construir um circuito lógico

[2] Em Inglês utiliza-se o termo *minterm*.

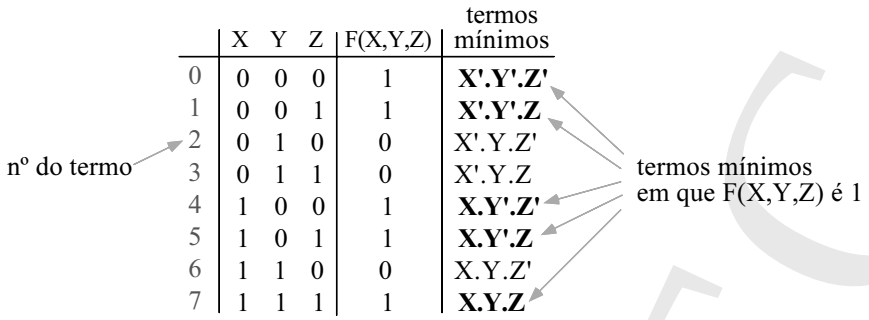


Figura 3.8: Termos mínimos de uma função booleana de 3 variáveis X, Y e Z.

que realize essa função. Por exemplo, a soma canónica da função apresentada na figura 3.8 escreve-se:

$$F(X,Y,Z) = X'.Y'.Z' + X'.Y'.Z + X.Y'.Z' + X.Y'.Z + X.Y.Z$$

Esta expressão representa exactamente o mesmo que a tabela de verdade e tem tantos termos de produto quantos os uns existentes na tabela. Claro que para funções com muitas variáveis (mais do que 5 ou 6) não é praticável escrever e manipular à mão expressões deste tipo. Será visto mais à fente formas alternativas de representação mais compactas mas que contêm exactamente a mesma informação do que a expressão canónica.

Termos de soma, termos máximos e produto canónico

Fazendo uso do princípio da dualidade, pode-se estender o conjunto de definições dadas antes para as correspondentes que utilizam o operador OU em vez do E:

- **Termo de soma** é um literal ou a soma lógica de vários literais:

$$X, \quad Y' + W + Y + X, \quad \overline{ACK} + READY, \quad X + X + X$$

- **Produto de somas** é um termo de soma ou o produto lógico de termos de soma (note que, dada a precedência dos operadores E e OU, é necessário utilizar parêntesis para agrupar cada termo de soma):

$$X.Y', \quad (Y' + W + X).Y, \quad (\overline{ACK} + X).(READY + \overline{Y})$$

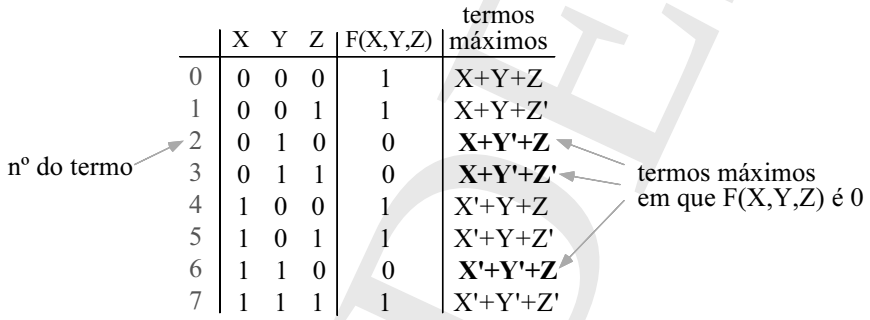
- **Termo normal** de soma é um termo de soma em que cada literal não aparece mais de uma vez (por exemplo $X + X + Y$ não é um termo normal):

$$X, \quad Y' + W + X, \quad \overline{ACK} + READY$$

- **Termo máximo**[3] de N variáveis é um termo normal de soma com N variáveis. Um termo máximo só vale 0 para uma e uma só combinação das suas variáveis. Exemplos de termos máximos com 3 variáveis X, Y e Z são:

$$X+Y+Z, \quad X+Y'+Z', \quad X'+Y'+Z'$$

De forma *dual* ao que foi visto para os termos mínimos, também cada termo máximo corresponde exactamente a uma linha da tabela de verdade de uma função, e é identificado pelo número dessa linha. Assim, o termo máximo $X+Y'+Z'$ só vale 0 quando for $X=0$ e $Y=1$ e $Z=1$, corresponde à linha 3 da tabela de verdade onde $XYZ=011$ (figura 3.9).



	X	Y	Z	F(X,Y,Z)	termos máximos
0	0	0	0	1	X+Y+Z
1	0	0	1	1	X+Y+Z'
2	0	1	0	0	**X+Y'+Z**
3	0	1	1	0	**X+Y'+Z'**
4	1	0	0	1	X'+Y+Z
5	1	0	1	1	X'+Y+Z'
6	1	1	0	0	**X'+Y'+Z**
7	1	1	1	1	X'+Y'+Z'

Figura 3.9: Termos máximos de uma função booleana de 3 variáveis X, Y e Z.

Realizando o produto lógico dos termos máximos para os quais a função vale 0 obtém-se o *produto canónico* ou *expressão canónica produto-de-somas*:

$$F(X,Y,Z) = (X+Y'+Z).(X+Y'+Z').(X'+Y'+Z)$$

Listas de termos mínimos e termos máximos

Uma forma alternativa de representar uma função booleana consiste em listar o número dos termos mínimos (ou máximos) para os quais a função vale 1 (ou zero). Por exemplo, a função apresentada nas figuras 3.8 e 3.9 pode ser representada pela lista de termos mínimos 0,1,4,5,7 ou (pelo princípio da dualidade) pela lista de termos máximos 2,3,6.

Se a mesma função fosse representada numa tabela de verdade com as variáveis dispostas como YZX, então a lista de termos mínimos seria 0,1,2,3,7 e a lista de termos máximos seria 4,5,6 (figura 3.10). Por esta razão, quando se apresenta uma lista de termos mínimos ou de termos máximos para representar uma função é necessário identificar a ordem pela qual são consideradas as suas variáveis.

[3] Em Inglês utiliza-se o termo *maxterm*.

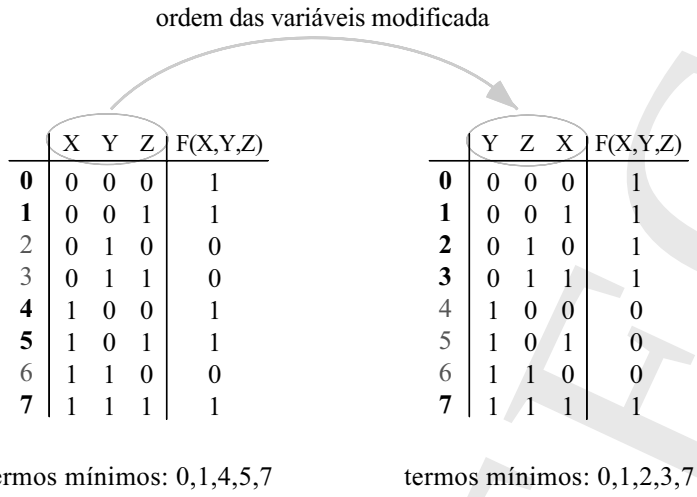


Figura 3.10: Dependência do números dos termos mínimos (ou dos termos máximos) com a ordem das variáveis na tabela de verdade.

A forma convencionada para representar as listas de termos mínimos (máximos) usa o sinal de somatório (produtório) para identificar o tipo de lista (somatório significa soma de produtos), as variáveis na ordem pela qual são consideradas e a lista de números que identificam os termos mínimos (ou máximos). Para a função considerada, as listas de termos mínimos e máximos são:

$$F(X,Y,Z) = \sum_{X,Y,Z}(0,1,4,5,7) \quad \text{(Lista de termos mínimos)}$$

$$F(X,Y,Z) = \prod_{X,Y,Z}(2,3,6) \quad \text{(Lista de termos máximos)}$$

Considerando a tabela de verdade com as variáveis dispostas como mostrado na figura 3.10, a lista de termos mínimos é:

$$F(X,Y,Z) = \sum_{Y,Z,X}(0,1,2,3,7) \quad \text{(Lista de termos mínimos)}$$

Capítulo 4

Projecto de circuitos combinacionais

No capítulo anterior apresentaram-se formas de representar o comportamento pretendido para um sistema digital, recorrendo a um conjunto de formalismos matemáticos a que se chama Álgebra de Boole. Esta álgebra estabelece um conjunto de definições e de regras formais que permitem escrever expressões que modelam o comportamento de um sistema digital, e manipulá-las por forma a representá-las sob diferentes formas. Mostrou-se como se pode representar uma função booleana como um circuito lógico, interligando símbolos que representam os operadores elementares da álgebra de Boole: Es, OUs e inversores. No entanto, não foi tida em conta a *complexidade* ou o *custo* do circuito resultante, sendo apenas traduzido cada operador na expressão booleana pela porta lógica correspondente no circuito (figuras 3.1 e 3.3).

Uma vez obtida uma descrição formal que traduz o comportamento pretendido para um sistema digital (construindo, por exemplo, uma expressão algébrica do tipo soma-de-produtos), o passo seguinte para obter um circuito electrónico digital que a realize consiste em simplificar a função lógica e representá-la com um conjunto de portas lógicas (ou de outras funções mais complexas) que permitam optimizar uma ou mais medidas de qualidade do sistema projectado: por exemplo, custo, rapidez ou consumo de energia. Por exemplo, para construir um certo circuito digital pode ser conveniente não utilizar portas lógicas do tipo OU (porque não estão disponíveis ou porque têm um custo muito elevado), devendo a implementação ser realizada apenas com portas do tipo E e inversores. Como, pelas leis da álgebra de Boole, a função OU pode ser representada à custa de Es e inversores, seria necessário manipular as expressões booleanas de maneira a que fossem usados apenas aqueles tipos de operadores.

Neste capítulo apresenta-se uma técnica para minimizar expressões booleanas nas formas padrão soma de produtos e produto de somas, permitindo obter circuitos digitais *combinacionais* com dois níveis de portas lógicas que necessitam do menor número possível de portas lógicas. Circuitos combinacionais caracterizam-se por a sua saída depender apenas do valor presente nas suas entradas, podendo por isso ser representados por expressões booleanas. Será

estudado mais tarde (capítulo 6) outra classe de circuitos digitais designados por *circuitos sequenciais*, em que as saídas dependem não só dos valores lógicos nas entradas no instante presente, mas também da sequência de valores que ocorreram nas entradas em instantes anteriores, que requerem uma forma de representação mais complexa do que para circuitos combinacionais.

4.1 Optimizar o projecto

No projecto de sistemas digitais, assim como em outras áreas de Engenharia, é fundamental realizar a actividade de projecto *optimizando* uma ou mais medidas de qualidade do sistema projectado. Por exemplo, quando se desenvolve um motor para um pequeno automóvel citadino, dois objectivos importantes a atingir são reduzir ao mínimo o consumo e custo de produção. No entanto, no projecto de um motor de competição já será mais importante maximizar a potência ou o binário desenvolvido do que o consumo ou mesmo o custo. Em ambas as situações existem muitos outros factores importantes a considerar, como por exemplo o peso, tamanho e o nível de ruído produzido, que têm naturalmente de ser tidos em conta durante o projecto e que muitas vezes são contraditórios entre si. Por exemplo, projectar um pequeno automóvel citadino obriga a estabelecer *compromissos* entre, por exemplo, o tamanho exterior e a habitabilidade (pequeno por fora e grande por dentro) ou entre a potência do motor e o consumo (potente e económico).

No projecto de circuitos electrónicos digitais podem também ser considerados variados critérios que devem ser considerados para avaliar a sua qualidade. Os mais importantes e que geralmente são considerados em qualquer projecto são o custo monetário, a rapidez (importante nos computadores), o tamanho físico e o consumo de energia. Destes 4 critérios iremos considerar apenas o factor custo, ou de uma forma geral, a complexidade do circuito.

Para obter uma realização de um circuito lógico com o mínimo custo é necessário manipular a expressão booleana que descreve o seu comportamento, de forma a simplificá-la para reduzir ao mínimo o número de portas lógicas necessárias para a realizar. Embora em alguns casos isso não seja completamente verdade, pode-se considerar que quantas menos portas lógicas forem necessárias para realizar uma certa função lógica, mais barata será a sua implementação física. Além disso, portas lógicas com poucas entradas são mais simples e baratas do que portas que realizam a mesma função mas com muitas entradas. Por exemplo, uma porta AND de 5 entradas tem uma complexidade equivalente a 4 portas AND de duas entradas. Por este motivo, é mais correcto medir a complexidade de um circuito pelo número de portas lógicas de duas entradas (as mais simples) que são necessárias para o construir.

Uma medida padrão que é geralmente utilizada pelos projectistas de sistemas digitais consiste em tomar como unidade de complexidade a porta lógica NAND de 2 entradas, sendo a

complexidade de um circuito expressa pelo número de portas lógicas equivalentes (número de NANDs de duas entradas necessárias para realizar a função lógica) ou então pelo espaço físico ocupado pelo circuito, expressa em unidades do espaço ocupado por uma porta NAND de 2 entradas. Como será estudado mais tarde (capítulo TBD), as portas NAND assumem esta importância porque na tecnologia CMOS em que é fabricada a grande maioria dos circuitos digitais actuais, a porta lógica mais pequena que se pode construir é a porta NAND de 2 entradas.

4.1.1 Tecnologias de implementação

A construção de um circuito electrónico digital que implemente uma dada função lógica requer, em primeiro lugar, a escolha do tipo de dispositivos (electrónicos) que serão usados para realizar as funções lógicas necessárias. Os primeiros sistemas digitais que surgiram nos inícios do século XX realizavam as funções lógicas elementares à custa de sistemas electro-mecânicos (relés[1]), que funcionavam fechando e abrindo contactos eléctricos de forma semelhante a um vulgar interruptor. Na figura 4.1 ilustra-se os elementos constituintes de um relé, mostra-se uma possível realização de uma porta AND e de um inversor usando relés (como exercício, desenhe um circuito com base em relés que implemente a função lógica OR).

Os circuitos digitais actuais são na realidade construídos de forma semelhante ao mostrado na figura 4.1, com a diferença que os interruptores controlados electricamente não são relés mas sim transístores. No contexto de aplicações em circuitos digitais, os transístores funcionam como interruptores tal como os relés, mas ocupando um espaço muito menor, consumindo muito menos energia e trocando entre 0 e 1 de forma muito mais rápida do que o relé. Enquanto que um relé necessita de um espaço de dezenas de mm^2, um transistor usado num circuito integrado digital actual ocupa áreas na ordem da décima de μ^2 (um μ é 10^-6 m). Para se fazer uma ideia do termo de comparação entre estas dimensões, se um relé com dimensões de $5mm \times 10mm$ fosse ampliado para a área de um campo de futebol, um transístor[2] ocuparia um espaço equivalente a um rectângulo com $5mm \times 10mm$, correspondendo a uma área aproximadamente 100×10^6 vezes menor.

Apesar de os transístores serem os dispositivos electrónicos elementares usados na construção de circuitos digitais, quando se projecta um sistema digital é conveniente recorrer a circuitos mais complexos já construídos que realizam as operações lógicas necessárias para compor a função desejada. Um conjunto básico desses circuitos, a que foram chamados portas lógicas, implementam os operadores elementares da álgebra de Boole e constituem as peças fundamentais para realizar circuitos digitais.

[1] Um relé é um interruptor controlado electricamente, onde um contacto mecânico é deslocado por acção de um electroíman

[2] O menor transístor que é possível construir numa certa tecnologia CMOS 0.18μ.

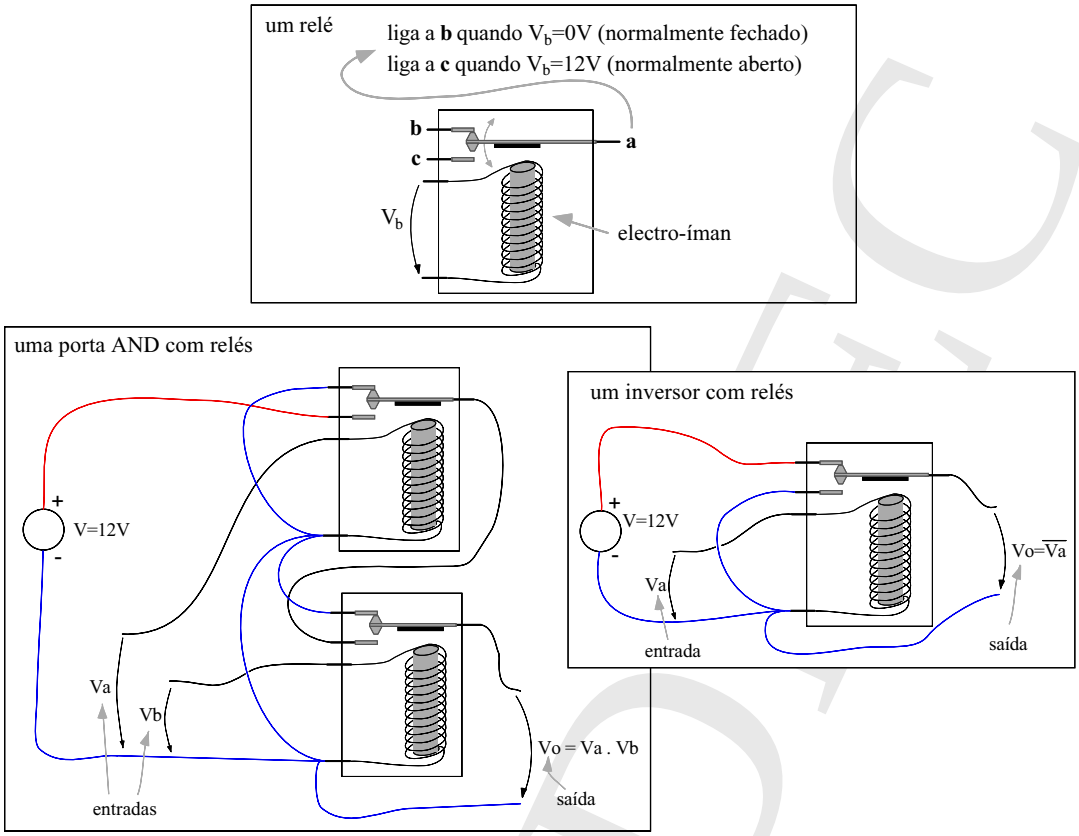


Figura 4.1: Realização das funções lógicas AND e NOT com relés; a função AND é equivalente à associação de dois interruptores em série: a saída só apresenta 12V quando ambas as entradas forem iguais a 12V; a função NOT inverte o nível lógico representado pela tensão eléctrica na sua entrada: quando se colocam 12V na entrada a saída apresenta 0V, e quando se colocam 0V na entrada a saída fica com 12V

No entanto, o custo monetário associado à realização física de um circuito digital pode não ser bem representado pelo número de portas lógicas ou de transístores que o circuito contém. Por razões que se prendem com o custo de produção, não existem disponíveis comercialmente dispositivos electrónicos que realizem a função de uma única porta lógica elementar. Por essa razão, os circuitos electrónicos digitais mais simples que se podem comprar actualmente reúnem num único circuito integrado algumas portas lógicas. Neste caso, é importante medir a complexidade de um circuito digital pelo número de circuitos integrados que são necessários, ou então contabilizar o custo monetário dos componentes utilizados.

Os circuitos integrados digitais da série 74

Nos anos sessenta foi introduzida pela Texas Instruments uma série de dispositivos electrónicos digitais, conhecidos por "série 74". Estes circuitos integrados têm referências do tipo 74Xnnn, onde "X" é um designador que identifica as características eléctricas e temporais do circuito (família lógica[3]) e "nnn" é um número que identifica a função lógica realizada pelo circuito. Por exemplo, o circuito integrado 74X00 contém 4 portas lógicas do tipo NAND de 2 entradas cada uma e o integrado 74X04 tem 6 inversores. Dispositivos electrónicos deste tipo, com complexidades equivalentes a poucas portas lógicas, são classificados como circuitos SSI (*Small Scale Integration* ou de pequena escala de integração). Estes circuitos integrados existem ainda disponíveis comercialmente e oferecem um meio simples e prático para construir sistemas digitais de pequena complexidade (até poucas dezenas de portas lógicas).

Como se usam os integrados 74XXnnn?

Para construir um sistema digital usando dispositivos electrónicos deste tipo, basta ligar os seus terminais de alimentação eléctrica a uma fonte de tensão contínua de 5V, ficando disponíveis nos restantes terminais as entradas e saídas das funções lógicas realizadas pelo circuito. Para garantir o correcto funcionamento, as entradas de portas lógicas que não forem usadas nunca devem ser deixadas desligadas mas devem ser ligadas a 0V (nível lógico zero) ou a 5V que representa o nível lógico 1. Quando uma entrada é ligada a 5V, essa ligação deve ser feita através de uma resistência com valor entre 1KΩ e 10KΩ.

Na figura 4.2 apresenta-se a organização interna de alguns dispositivos desta série e na figura 4.3 exemplifica-se a implementação de um circuito digital sobre uma placa de ligações usando estes circuitos integrados.

Note que para realizar a porta OR de 3 entradas do circuito da figura 4.3 poderia também ser usado um 74X32 que tem 4 portas OR de 2 entradas, ou mesmo um 74X00 que tem 4 portas NAND de 2 entradas (como se pode fazer um OR de 3 entradas com 4 ou menos portas NAND de 2 entradas?). Para minimizar o custo final deste circuito seria necessário escolher a combinação de circuitos integrados desta série que permitisse construir um circuito com a funcionalidade pretendida pelo custo mais baixo possível.

A mesma função lógica realizada pelo circuito da figura 4.3 pode ser optimizada usando o processo que se descreve na próxima secção, conduzindo a uma expressão na forma soma de produtos cuja realização apenas necessita de uma porta lógica AND e outra OR:

[3]Uma família lógica caracteriza os dispositivos electrónicos relativamente aos níveis das tensões eléctricas que representam zeros e uns nas entradas e saídas e que têm de ser compatíveis entre si, e também em termos da rapidez de funcionamento e do consumo de energia.

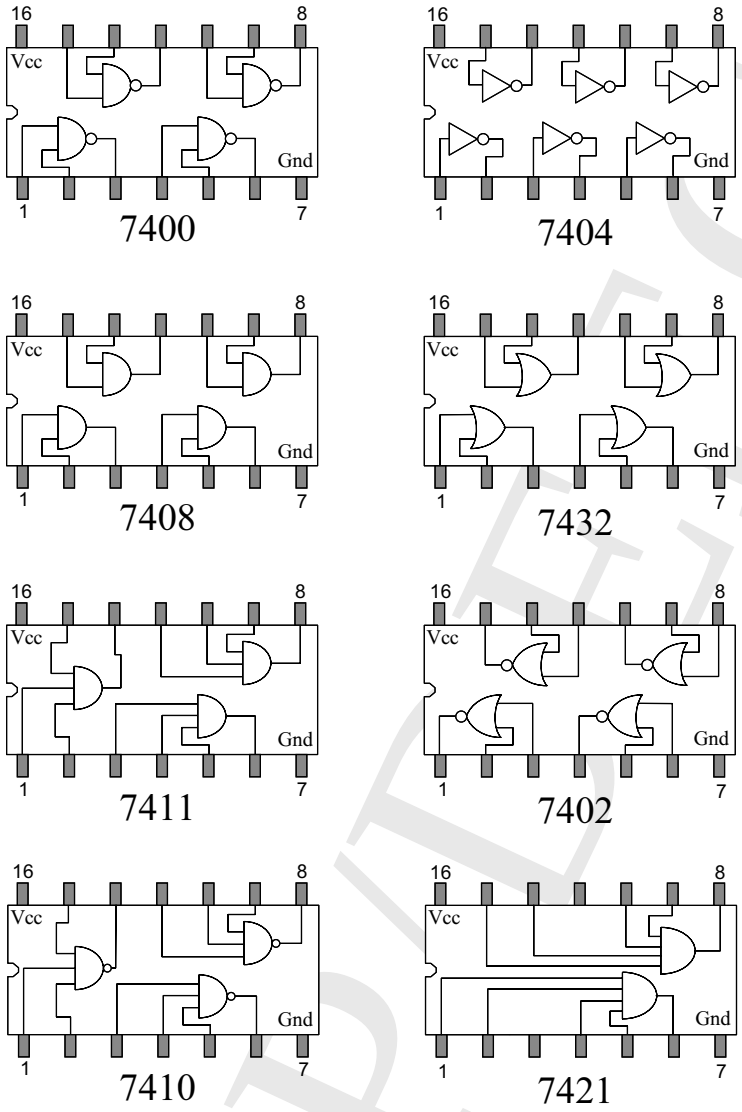


Figura 4.2: Organização interna de alguns dispositivos da série 74XXnnn.

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

No entanto, como não existem circuitos integrados da série 74xxx que contenham simultaneamente portas AND e OR, seria necessário utilizar dois circuitos integrados (um 73x08 e um 74x32) e usar apenas uma das 4 portas lógicas que cada um oferece. Uma transformação posterior, baseada na aplicação do teorema de DeMorgan, permite escrever aquela expressão noutra equivalente mas que apenas necessita de 4 portas do tipo NAND, disponíveis num único circuito integrado do tipo 74x00 (ver figura 4.4).

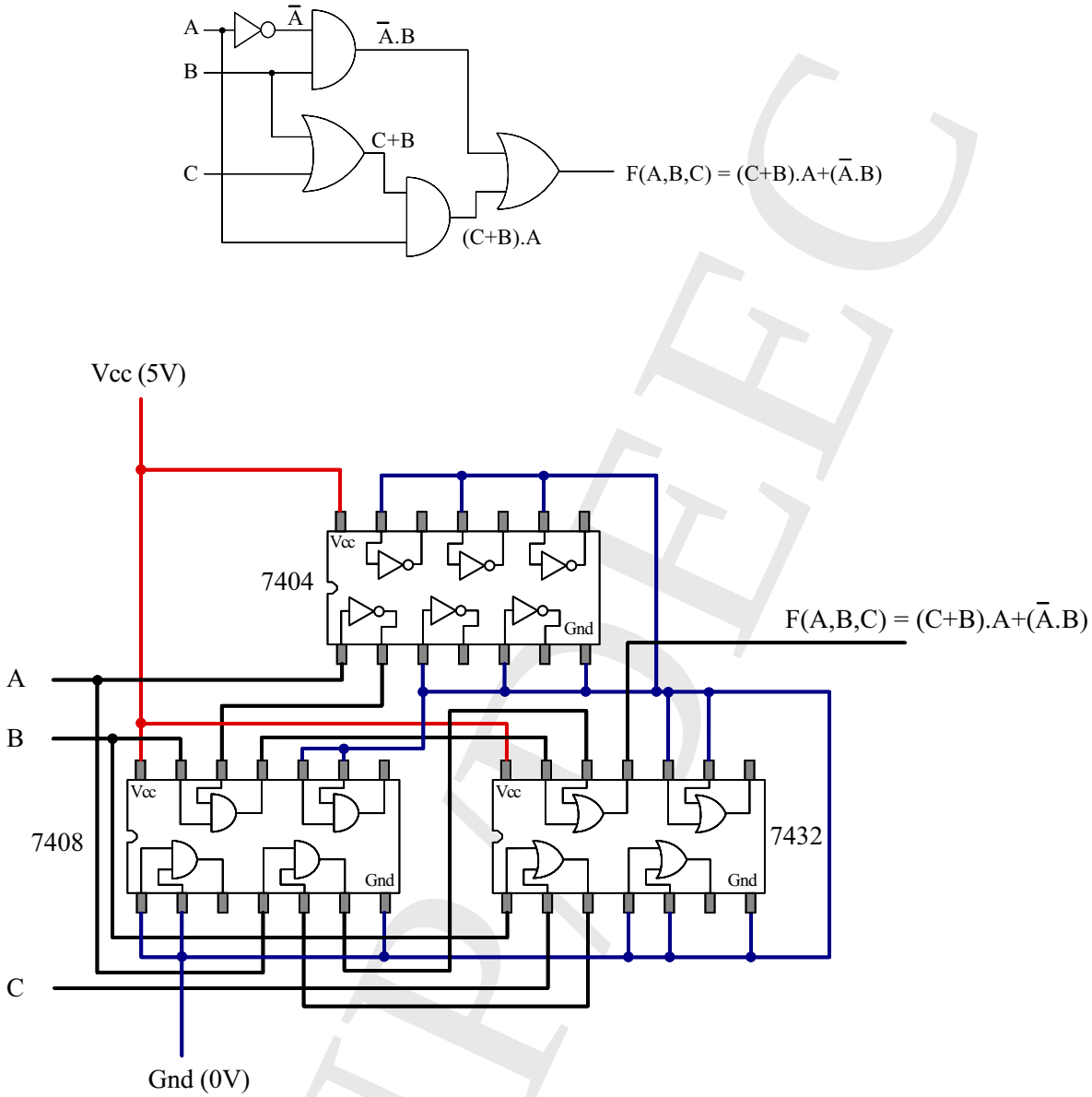


Figura 4.3: Realização de um circuito lógico com dispositivos SSI da série 74.

4.2 Minimização de funções representadas em formas padrão

Embora o processo de minimização de uma função booleana esteja intimamente ligado à forma como o circuito será realizado fisicamente (i.e. a tecnologia de implementação a utilizar), a generalidade dos processos automáticos de simplificação de funções lógicas trabalham sobre funções lógicas representadas nas formas padrão soma de produtos ou produto de somas estudadas no capítulo 3. A aplicação do processo de minimização que se apresenta neste capítulo permite obter expressões booleanas nessas formas padrão que usam o número mínimo de oper-

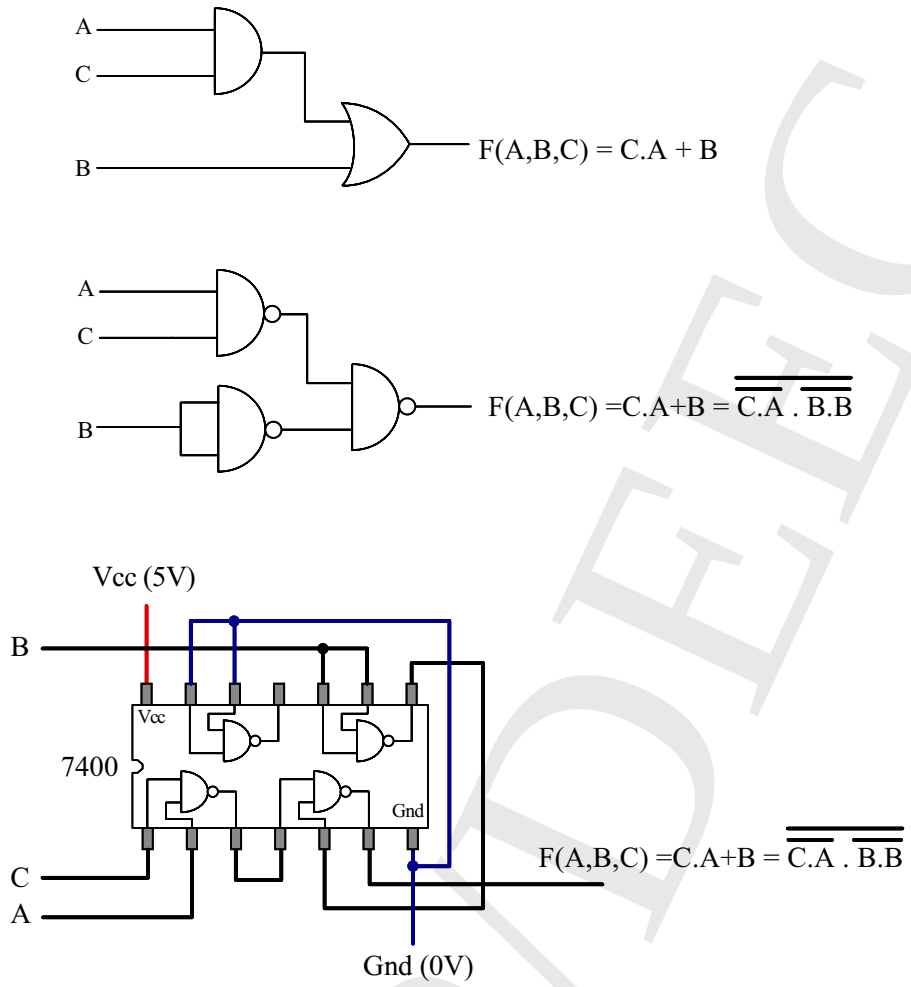


Figura 4.4: Optimização do circuito lógico mostrado na figura 4.3.

adores lógicos E e OU.

Embora a realização física de expressões booleanas representadas nas formas padrão não seja, na generalidade dos casos, a solução que conduz ao menor custo monetário de implementação, as expressões simplificadas nessa forma constituem bons pontos de partida para posteriores melhorias, específicas da tecnologia de implementação que venha a ser utilizada.

4.2.1 Mapas de Karnaugh

Para além das várias formas estudadas no capítulo anterior, uma função booleana pode ser representada usando uma forma tabular a que se chama mapa de Karnaugh. Um mapa de Karnaugh

para um função de 3 variáveis[4] é representada como uma matriz de $2^3 = 8$ elementos, em que se associam às linhas e colunas todos os valores possíveis para as variáveis independentes da função segundo uma ordem bem determinada, sendo a matriz preenchida com os zeros e uns que definem a função. Na figura 4.5 mostra-se a representação de uma função booleana de 3 variáveis num mapa de Karnaugh.

A	B	C	F(A,B,C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

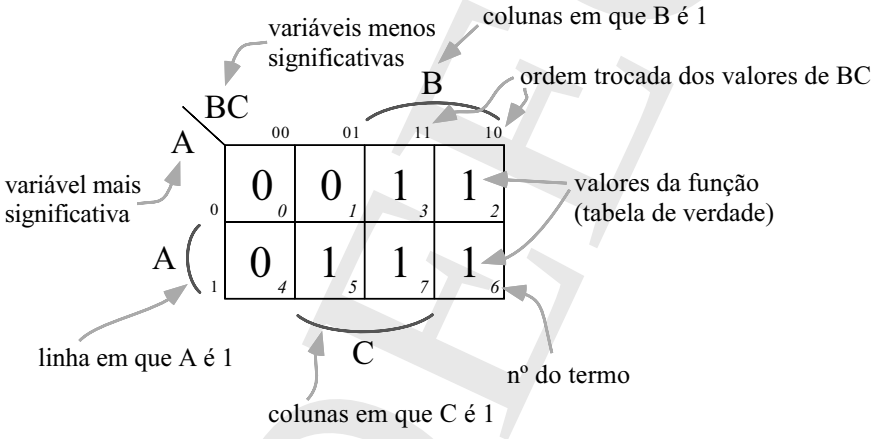


Figura 4.5: Representação de uma função booleana de 3 variáveis num mapa de Karnaugh.

Note-se que o conteúdo do mapa de Karnaugh não é mais do que a tabela de verdade da função, embora seguindo uma organização "não natural" o que vai permitir a aplicação de uma técnica simples de optimização, que garante expressões do tipo soma de produtos ou produto de somas que têm o menor número possível de literais (e consequentemente o menor número de operadores E e OU).

Como se viu no capítulo anterior, pode-se escrever uma expressão booleana (expressão canónica) na forma soma de produtos como uma soma lógica de termos normais de produto correspondentes às linhas da tabela de verdade em que a função vale 1. Para a função exemplificada na figura 4.5, esta expressão é:

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

Na organização apresentada no mapa de Karnaugh, podem-se identificar os termos normais de produto que correspondem a cada 1 da função, determinando a que linhas e colunas pertence cada 1 da tabela: se pertencer a uma linha/coluna em que uma variável é 1, então essa variável aparece no termo de produto na sua forma não negada; se pertencer a uma linha/coluna em que uma variável é zero, então essa variável aparece na sua forma negada. Na figura 4.6 mostram-se

[4]será apresentada mais tarde a forma de mapas de Karnaugh para funções de 2, 4 e 5 variáveis; não são geralmente usados para funções com mais do que 5 variáveis

os termos de produto de uma função e a sua correspondência com a posição que ocupam no mapa de Karnaugh.

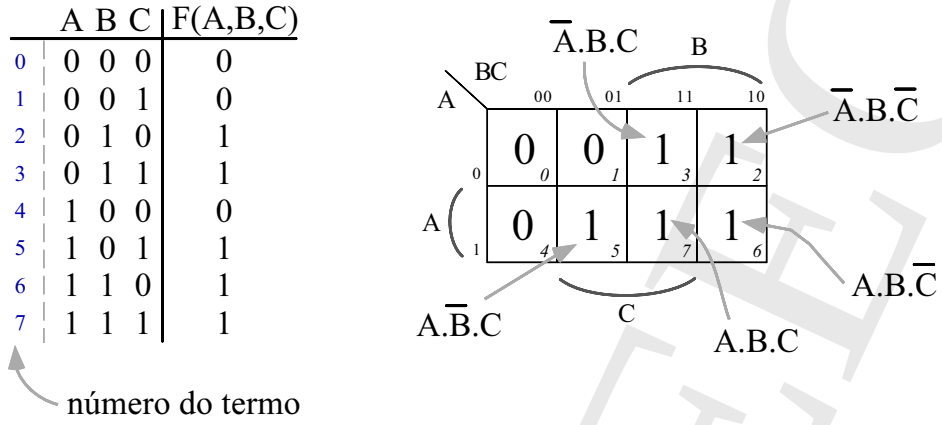


Figura 4.6: Termos de produto para uma função booleana de 3 variáveis representada num mapa de Karnaugh.

4.2.2 Minimização de expressões soma de produtos

Para simplificar a expressão canónica soma de produtos podemos começar por associar o primeiro termo ($A.B.C$) com o segundo ($\overline{A}.B.C$), onde a variável A aparece nas formas negada e não negada (aplicação directa do teorema T10):

$$\begin{aligned} F(A,B,C) &= A.B.C + \overline{A}.B.C + A.\overline{B}.C + \overline{A}.B.\overline{C} + A.B.\overline{C} \\ &B.C.(A+\overline{A}) + A.\overline{B}.C + \overline{A}.B.\overline{C} + A.B.\overline{C} \\ &B.C + A.\overline{B}.C + \overline{A}.B.\overline{C} + A.B.\overline{C} \end{aligned}$$

Podemos interpretar este passo de simplificação como o agrupamento dos dois uns no mapa de Karnaugh correspondentes aos termos 3 e 7 que foram combinados (ver figura 4.7). O termo de produto resultante ($B.C$) pode ser obtido analisando as linhas e colunas a que pertence o grupo de 2 uns: se pertence a linhas/colunas em que uma variável vale 1, então essa variável aparece na forma não negada; se pertence a linhas/colunas em que uma variável vale zero, então essa variável aparece na forma negada; se esse grupo intersecta linhas/colunas em que uma variável é um e as linhas/colunas em que é zero, então essa variável não aparece no termo.

Aplicando o mesmo teorema, podem-se combinar os termos de produto 2 e 6 resultando na simplificação da variável *A*, que pode ser interpretada no mapa de Karnaugh como o agrupamento dos 2 uns correspondentes a esses termos:

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

Os dois termos de produto obtidos pelos dois passos anteriores ($B.C$ e $B.\overline{C}$) podem agora ser combinados entre si eliminando assim a variável C:

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

Este passo de combinação pode ser entendida como o agrupamento dos 2 grupos de uns criados anteriormente. O termo de produto correspondente a este grupo de 4 uns pode ser lido directamente do mapa de Karnaugh como *B*, já que o grupo pertence às colunas da variável *B*, e as variáveis *A* e *C* estão parcialmente dentro e fora desse grupo.

Finalmente, pode ser formado um grupo de 2 uns com os termos 5 e 7 resultando no termo de produto $A.C$. Note que o facto de o termo 7 ($A.B.C$) já ter sido usado no grupo de 4 uns, não impede que seja usado de novo, já que pelo teorema T3 ($X = X + X$) é possível repetir um termo de produto numa expressão soma de produtos um número arbitrário de vezes sem alterar a função representada.

O processo de minimização de expressões boleanas na forma soma de produtos é realizado agrupando os uns no mapa de acordo com as regras seguintes:

1. só podem ser feitos grupos rectangulares de uns geometricamente adjacentes, com um número de uns igual a uma potência inteira de 2 (1, 2, 4, 8,...2^N); como se viu no exemplo apresentado na figura 4.7, esta condições é imposta pelo facto de a dimensão dos grupos que se podem fazer resultar sempre do agrupamento de dois grupos de igual tamanho: dois grupos de 1 dá um grupo de 2, dois de 2 dá um grupo de 4, etc. Como o critério de adjacência corresponde a existir apenas uma troca de variável entre os dois grupos que se combinam, então também são adjacentes os extremos do quadro (os termos 0 e 2 e os termos 4 e 6).

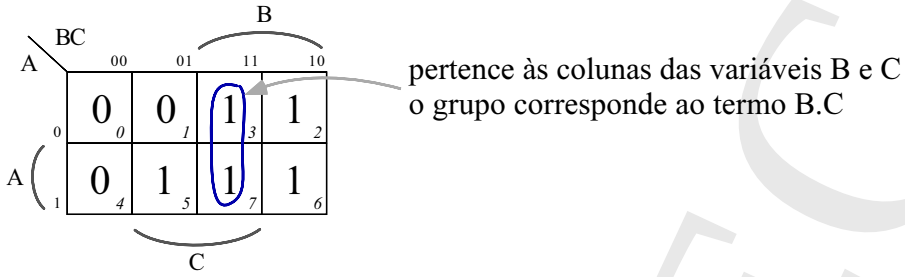
2. todos os uns da função devem pertencer a pelo menos um grupo (ou ser cobertos por todos os termos de produto encontrados).

3. devem ser feitos grupos que sejam o maior possível; note que quanto maior for um grupo menos variáveis terá o termo de produto correspondente.

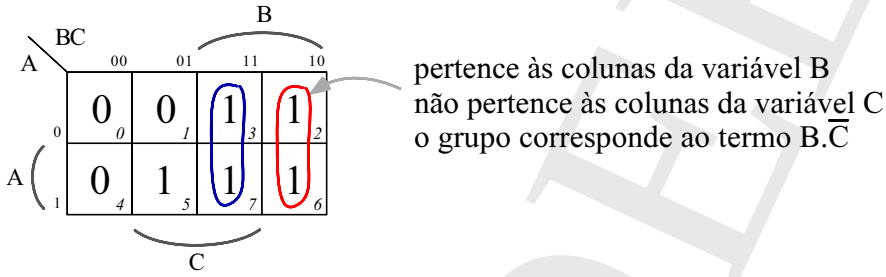
4. todos os uns devem ser cobertos pelo menor número possível de grupos: note que cada grupo de uns corresponderá a um termo de produto na expressão soma de produtos.

A expressão na forma soma de produtos que se obtém seguindo estas regras apresenta o menor número possível de literais (variáveis ou as suas negações).

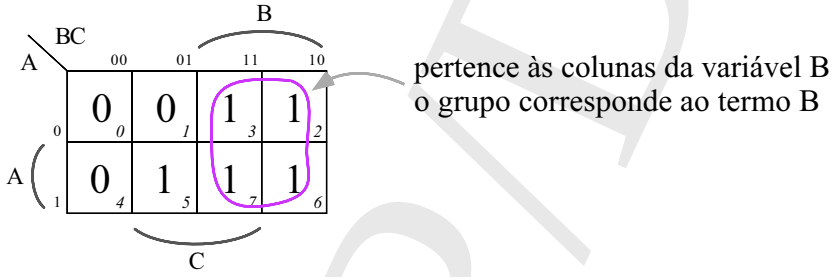
Combinando os termos 3 e 7: $\overline{A}.B.C + A.B.C = B.C.(\overline{A}+A) = B.C$



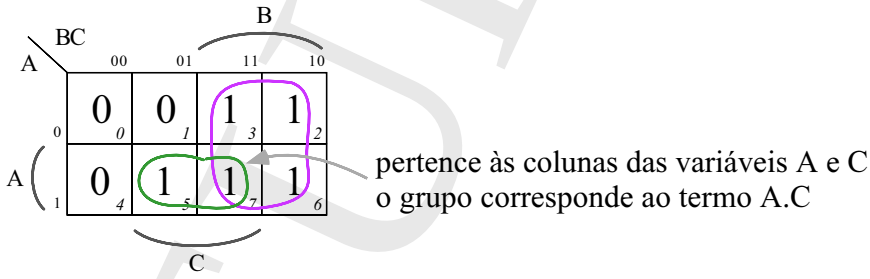
Combinando os termos 2 e 6: $\overline{A}.B.\overline{C} + A.B.\overline{C} = B.\overline{C}.(\overline{A}+A) = B.\overline{C}$



Combinando os termos B.C e B.$\overline{C}$ obtidos: $B.C + B.\overline{C} = B.(C+\overline{C}) = B$



Combinando os termos 5 e 7: $A.\overline{B}.C + A.B.C = A.C.(\overline{B}+B) = A.C$



Expressão mínima soma de produtos: $F(A,B,C) = B + A.C$

Figura 4.7: Minimização de uma função booleana na forma soma de produtos usando mapas de Karnaugh.

Para definir formalmente este processo de minimização à luz da álgebra de Boole, é necessário definir *implicante* de uma função booleana: uma função G é implicante de outra função F se quando G for 1 então isso implicar que F também seja 1. Por exemplo, na função F(X,Y,Z) representada na forma soma de produtos:

$$F(X,Y,Z) = X.Y.\overline{Z} + \overline{X}.Z + \overline{X}.\overline{Y}$$

diz-se que os termos de produto ($X.Y.\overline{Z}$, $\overline{X}.Z$ e $\overline{X}.\overline{Y}$) implicam a função $F(X,Y,Z)$ já que quando são 1 também a função vale 1.

Num mapa de Karnaugh, os grupos de uns (ou os uns isolados) são implicantes da função representada, já que correspondem a termos de produto na expressão soma de produtos. Define-se *implicante primo* de uma função F como um termo normal de produto que implica a função F, mas que deixa de a implicar se for removida qualquer variável desse termo. Como o retirar uma variável de um termo de produto significa duplicar o tamanho do grupo de uns representado no mapa de Karnaugh, então um implicante primo corresponde a um grupo de uns que seja o maior possível. Por exemplo, na função representada na figura 4.7, o termo de produto $A.C$ (correspondente ao agrupamento dos termos 5 e 7) é um implicante primo da função, porque se lhe for retirada a variável A (formando um grupo de 4 uns com os termos 4, 5, 6 e 7) ou se for retirada a variável C (formando um grupo de 4 uns com os termos 1, 3, 5 e 7) o termo de produto resultante deixa de implicar a função F.

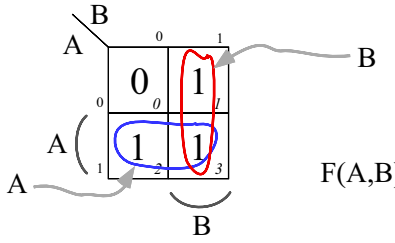
Pode-se assim dizer que a expressão mínima soma de produtos é uma soma lógica de implicantes primos — teorema dos implicantes primos. Para que a função soma de produtos obtida seja mínima, então os implicantes primos devem ser *essenciais*, o que significa que, se pelo menos um desses termos for retirado da expressão soma de produtos, então ela deixa de representar a função dada.

Mapas de Karnaugh de 2, 4 e 5 variáveis

O processo de minimização apresentado antes pode ser igualmente aplicado a funções booleanas com outro número de variáveis, embora a sua resolução "manual" seja geralmente restrita a funções que tenham 5 ou menos variáveis. Na figura 4.8 mostra-se a organização dos mapas de Karnaugh para funções de 2, 4 e 5 variáveis, e exemplifica-se com a minimização de expressões soma de produtos para as funções apresentadas. Note-se que no mapa para uma função de 4 variáveis, são adjacentes entre si (e podem por isso ser agrupadas) as posições nos extremos do mapa (linhas ou colunas), tal como já se tinha visto para os mapas de funções com 3 variáveis (o termo $A.\overline{D}$ na função de 4 variáveis mostrada na figura 4.8).

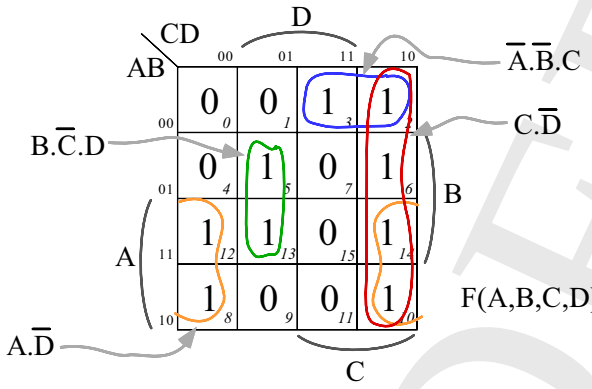
Um mapa de Karnaugh para funções de 5 variáveis já não pode ser representado no plano como uma única matriz, por forma a manter a relação de adjacência geométrica que permite

$F(A,B) = \Sigma_{AB}(1,2,3)$



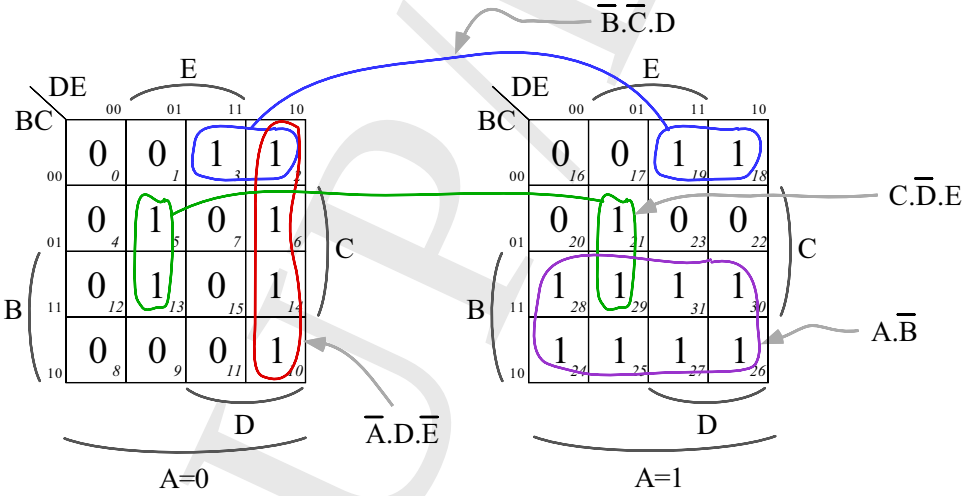
$F(A,B) = A + B$

$F(A,B,C,D) = \Sigma_{ABCD}(2,3,5,6,8,10,12,13,14)$



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

$F(A,B,C,D,E) = \Sigma_{ABCD}(2,3,5,6,10,13,14,18,19,21,24,25,26,27,28,29,30,31)$



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

Figura 4.8: Mapas de Karnaugh para funções de 2, 4 e 5 variáveis.

fazer os agrupamentos que correspondem a combinações dos termos mínimos da função. Tal como é mostrado na figura 4.8, uma representação possível consiste em "dividir" a função em duas metades, e representar cada metade num mapa de Karnaugh de 4 variáveis: o mapa desen-

hado à esquerda representa a função quando a variável mais significativa vale zero, e o mapa da direita quando essa variável vale 1. Assim, para além de se poderem fazer agrupamentos de termos em cada mapa, tal como se faz numa função de 4 variáveis, também é possível combinar grupos iguais entre os dois mapas eliminando com isso a variável mais significativa (a variável A no exemplo da figura).

4.2.3 Minimização de expressões na forma produto de somas

Recorrendo ao teorema generalizado de DeMorgan (ver página 50), pode-se obter facilmente a expressão mínima produto de somas. Assim, dada uma função $F(A,B,C,D)$, pode-se obter a expressão mínima na forma de produtos de $\overline{[F(A,B,C,D)]}$, aplicando em seguida o teorema de DeMorgan para obter $F(A,B,C,D)$ na forma produto de somas.

Como exemplo, considere-se a função $F(A,B,C,D)$ dada como uma lista de termos mínimos:

$$F(A,B,C,D) = \sum_{A,B,C,D}(2,5,6,8,10,12,13,14)$$

ou, na forma lista de termos máximos:

$$F(A,B,C,D) = \prod_{A,B,C,D}(0,1,3,4,7,9,11,15)$$

então o oposto desta função pode ser representado por:

$$\overline{[F(A,B,C,D)]} = \sum_{A,B,C,D}(0,1,3,4,7,9,11,15)$$

Aplicando o procedimento de minimização com mapas de Karnaugh (figura 4.9), obtém-se a expressão mínima soma de produtos da função oposta de $F(A,B,C,D)$:

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

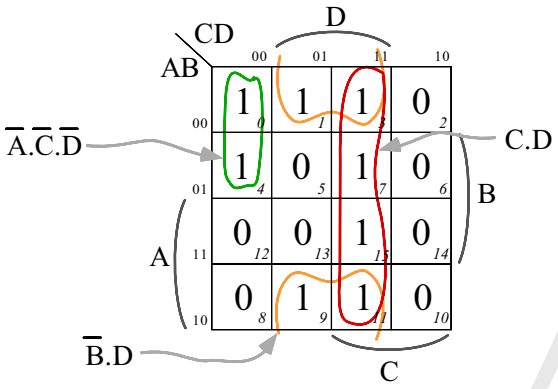
Recorrendo ao teorema generalizado de DeMorgan pode-se obter facilmente a função mínima na forma produto de somas como a negação da expressão anterior, bastando trocar os operadores lógicos OU e E e as variáveis pelas suas negações:

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

A expressão mínima produto de somas pode também ser obtida directamente do mapa de Karnaugh agrupando os zeros segundo regras semelhantes às usadas para agrupar os uns, e associando a cada grupo de zeros um termo de soma que fará parte da expressão produto de somas. Assim, um zero isolado corresponde a um termo máximo, que é um termo de soma em que são negadas as variáveis cujo valor é 1 (ver secção 3.2 na página 54). Pode-se assim relacionar

$$F(A,B,C,D) = \Sigma_{ABCD}(2,5,6,8,10,12,13,14) = \Pi_{ABCD}(0,1,3,4,7,9,11,15)$$

$$\overline{[\ F(A,B,C,D)\ ]} = \Sigma_{ABCD}(0,1,3,4,7,9,11,15)$$



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

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

Teorema generalizado de DeMorgan

$$F(A,B,C,D) = (B + \overline{D}) \,.\, (A + C + D) \,.\, (\overline{C} + \overline{D})$$ → Expressão mínima produto de somas

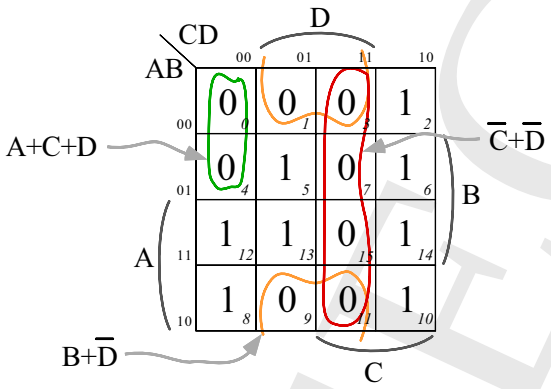
Figura 4.9: Minimização de expressões na forma produto de somas aplicando o teorema generalizado de DeMorgan.

um termo de soma com um grupo de zeros (ou um zero isolado) no mapa de Karnaugh, sendo negadas as variáveis a que um grupo pertence, e não negadas aquelas a que o grupo não pertence na totalidade. De forma semelhante com o que se viu para a minimização de expressões soma de produtos, também se um grupo apenas pertence em parte às linhas ou colunas de uma variável, então isso significa que essa variável não faz parte do termo de soma correspondente. Na figura 4.10 exemplifica-se o processo de minimização da expressão produto de somas, para a mesma função representada na figura 4.9.

4.2.4 Minimização de funções com termos indiferentes

A representação de uma função booleana com N variáveis numa tabela de verdade, deve apresentar todos os seus valores para as 2^N combinações das variáveis independentes. No entanto, existem situações em que a função pretendida apenas deve satisfazer um subconjunto de todos os valores possíveis para as variáveis, sendo indiferente o valor que assume para os restantes casos. Esta situação pode acontecer porque, dado o contexto em que o circuito será utilizado,

$$F(A,B,C,D) = \prod_{ABCD}(0,1,3,4,7,9,11,15)$$



Expressão mínima produto de somas:

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

Figura 4.10: Minimização de expressões na forma produto de somas agrupando os zeros no mapa de Karnaugh.

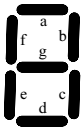
nem todas as combinações das entradas podem acontecer, ou então porque para certos casos não interessa o valor da função.

Um exemplo desta situação ocorre com um circuito digital normalmente utilizado para afixar dígitos decimais em mostradores de 7 segmentos. A este circuito chama-se descodificador BCD para 7 segmentos e permite traduzir (ou *descodificar*) um conjunto de 4 *bits* representando um dígito decimal (entre 0 e 9) num conjunto de 7 valores lógicos que, quando ligados de forma adequada a um mostrador de 7 segmentos, permitem acender os LEDs que desenham o dígito correspondente. Como o número binário esperado na entrada deste circuito apenas deverá representar um número entre 0 e 9, então o valor da função não é especificado para as entradas de 1010_2 a 1111_2. Como, do ponto de vista de função a realizar, tanto faz que nestes casos a função valha 1 ou 0, pode-se representar como *d* (do inglês *don't care*), usando-os como 1 ou como 0, dependendo daquilo que mais permite simplificar a função.

Na figura 4.11 apresenta-se a tabela de verdade para as 7 funções que realizam um descodificador BCD para 7 segmentos, e é exemplificada a aplicação do procedimento de minimização para as funções que implementam os segmentos **g** e **e**. Note que os termos indiferentes apenas devem fazer parte dos grupos de uns (ou de zeros, se o objectivo for minimizar a expressão produto de somas) se com isso se conseguir reduzir a complexidade da expressão (i.e. aumentar a dimensão dos grupos ou reduzir o número de grupos para cobrir todos os uns da função).

Apesar de na especificação de uma função booleana os termos indiferentes não terem um

disposição dos *LEDs* num mostrador de 7 segmentos



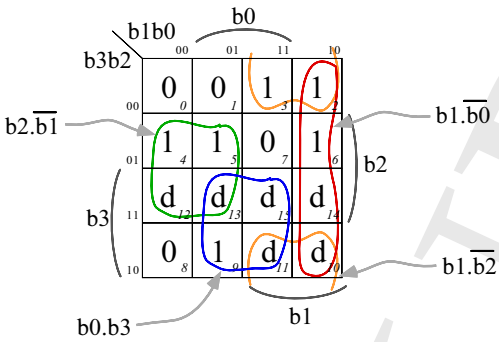
os 10 dígitos decimais desenhados num mostrador de 7 segmentos

tabela de verdade de um descodificador BCD para 7 segmentos

dígito decimal	b3	b2	b1	b0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	0
9	1	0	0	1	1	1	1	1	0	1	1
	1	0	1	0	d	d	d	d	d	d	d
	1	0	1	1	d	d	d	d	d	d	d
	1	1	0	0	d	d	d	d	d	d	d
	1	1	0	1	d	d	d	d	d	d	d
	1	1	1	0	d	d	d	d	d	d	d
	1	1	1	1	d	d	d	d	d	d	d

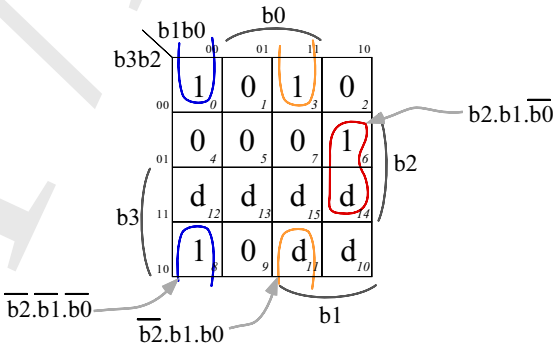
termos *don't care*

minimização da função g(b3,b2,b1,b0)



$g(b3,b2,b1,b0) = b2.\overline{b1} + b0.b3 + b1.\overline{b0} + b1.\overline{b2}$

minimização da função e(b3,b2,b1,b0)



$e(b3,b2,b1,b0) = \overline{b2}.b1.b0 + \overline{b2}.\overline{b1}.\overline{b0} + b2.b1.\overline{b0}$

Figura 4.11: Minimização de funções boleanas com termos indiferentes: um descodificador BCD para 7 segmentos.

valor definido, após a construção de uma expressão booleana (minimizada ou não) que realize essa função, obtêm-se naturalmente valores lógicos (zero ou um) para qualquer uma das combinações das suas variáveis. Como apenas são importantes os valores da função para os

termos não indiferentes, podem existir diferentes expressões boleanas que realizem a mesma função (os seus termos que não são indiferentes), mas que diferem entre si para os casos em que os valores da função não são especificados. Por exemplo, a função $g(b3,b2,b1,b0)$ pode ser realizada pela expressão mínima soma de produtos apresentada na figura 4.11, ou pela expressão mínima na forma produto de somas (ver figura 4.12):

$$g(b3,b2,b1,b0) = (b3+b2+b1).(\overline{b3}+b1+b0).(\overline{b2}+\overline{b1}+\overline{b0})$$

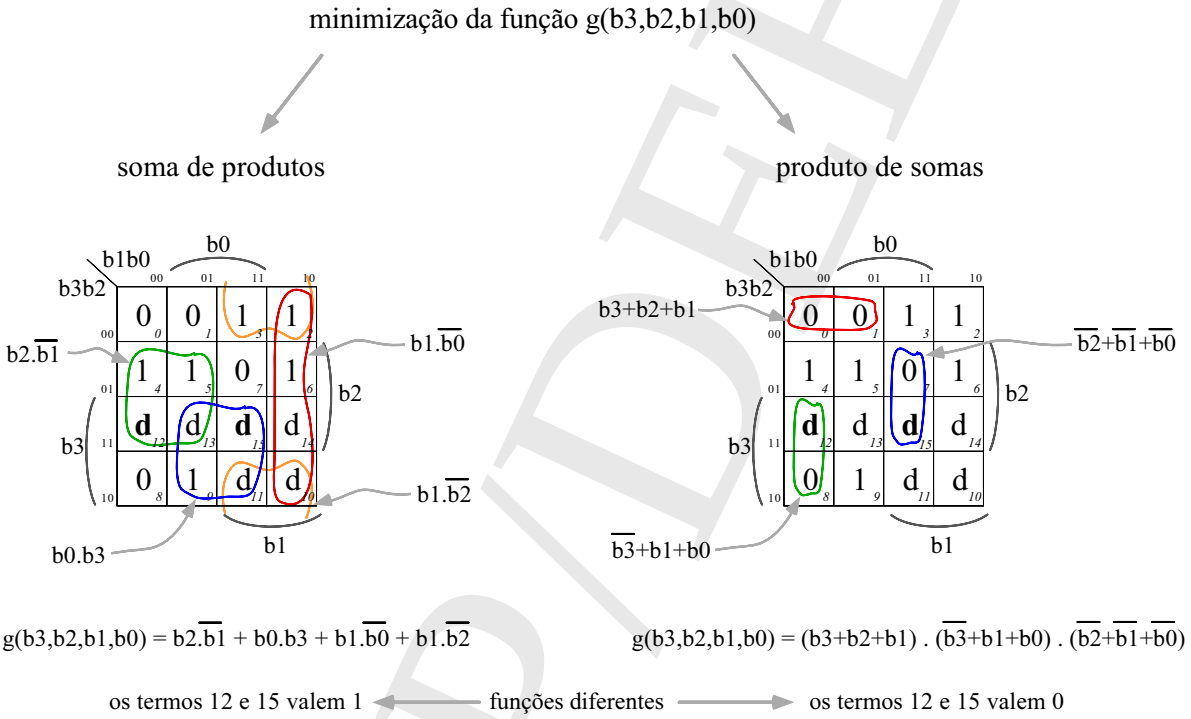


Figura 4.12: Expressões mínimas soma de produtos e produto de somas para a função $g(b3,b2,b1,b0)$ do descodificador BCD para 7 segmentos.

Nesta expressão, os termos 12 e 15 estão incluídos em termos de soma (grupos de zeros), mas na expressão mínima soma de produtos encontrada na figura 4.11 faziam parte de termos de produto (grupos de uns). Por este motivo, estas duas expressões realizam igualmente os termos não indiferentes da função $g(b3,b2,b1,b0)$, embora sejam funções booleanas distintas entre si.

4.3 Circuitos lógicos

As expressões mínimas obtidas pelo método descrito nas secções anteriores podem ser representadas como circuitos lógicos com a estrutura que se mostra na figura 4.13. As expressões

soma de produtos são traduzidas em circuitos que têm um primeiro nível de portas lógicas AND que realizam os termos de produto, e um segundo nível com uma porta lógica OR que faz a soma lógica dos termos de produto. De forma semelhante, as expressões do tipo produto de somas representam-se por circuitos com um primeiro nível de portas lógicas OR (termos de soma) e um segundo nível com uma porta lógica AND que realiza o produto lógico dos termos de soma. É usual designar estes tipos de circuitos lógicos como circuitos AND-OR e OR-AND, representando respectivamente as expressões soma de produtos e produto de somas.

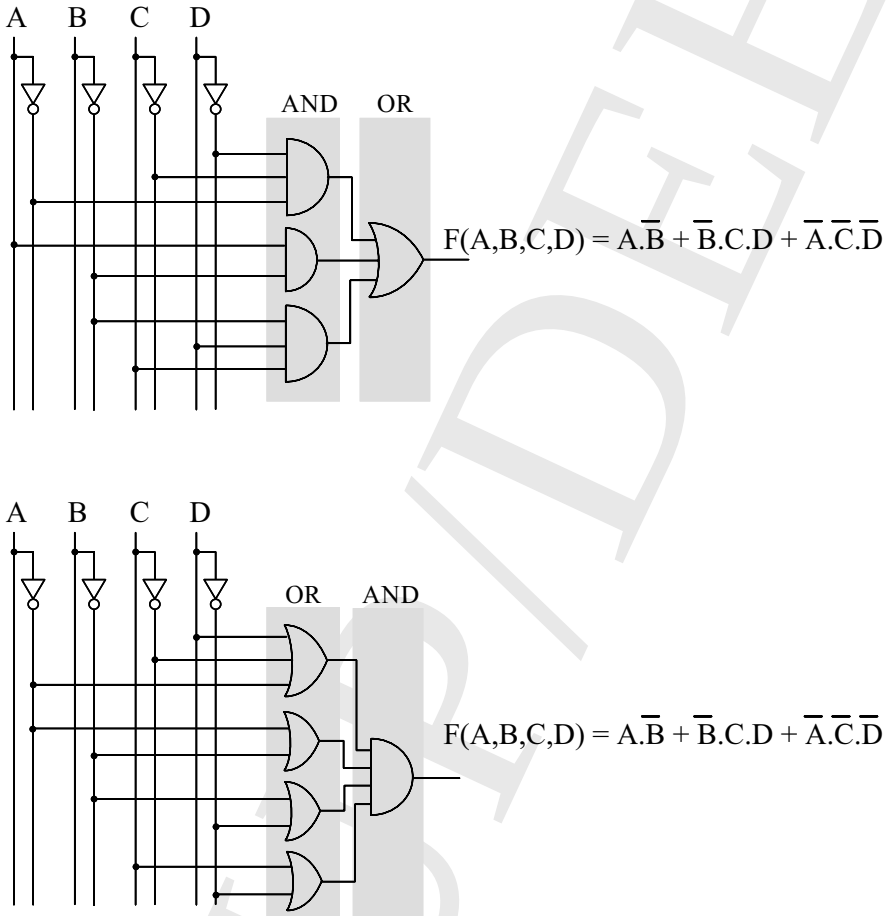


Figura 4.13: Circuitos lógicos AND-OR e OR-AND, correspondentes a expressões do tipo soma de produtos e produto de somas

Como foi já referido anteriormente, o processo de minimização estudado permite obter expressões lógicas nas formas padrão que têm o menor número de literais. Isso não significa que a sua realização, na forma dos circuitos AND-OR ou OR-AND correspondentes, seja a mais simples ou a mais barata possível. Dependendo do tipo de dispositivos electrónicos que se tem disponível para construir um determinado circuito digital, pode ser vantajoso implementar ex-

actamente a expressão mínima obtida com o mapa de Karnaugh, ou então submeter o circuito a transformações posteriores por forma a encontrar o conjunto de componentes que o permite realizar da forma mais económica possível.

Esta operação de optimização é uma tarefa complexa, já que depende de factores tão variados como o tipo de portas lógicas disponíveis, o seu custo individual, a forma como são associadas em circuitos integrados ou o espaço físico que ocupa o conjunto de dispositivos electrónicos necessários para realizar o circuito.

Em várias tecnologias de realização de circuitos electrónicos digitais, e em particular na tecnologia CMOS (*Complementary Metal-Oxide Semiconductor*) em que é hoje em dia fabricada a grande maioria dos circuitos electrónicos digitais, os dispositivos mais simples que se podem fabricar não são portas AND nem OR mas sim as portas inversoras correspondentes NAND e NOR. Como exemplo, uma porta AND de duas entradas realizada em tecnologia CMOS apresenta um "custo" (traduzido no espaço físico ocupado) que é cerca de 1.5 vezes o custo de uma porta NAND também de duas entradas.

Por esta razão, é por vezes importante representar um circuito lógico com apenas portas lógicas dos tipos NAND ou NOR. Na verdade, qualquer circuito lógico pode ser expresso em termos de apenas portas lógicas NAND ou NOR, já que com qualquer uma delas é possível realizar as 3 funções lógicas elementares (figura 4.14).

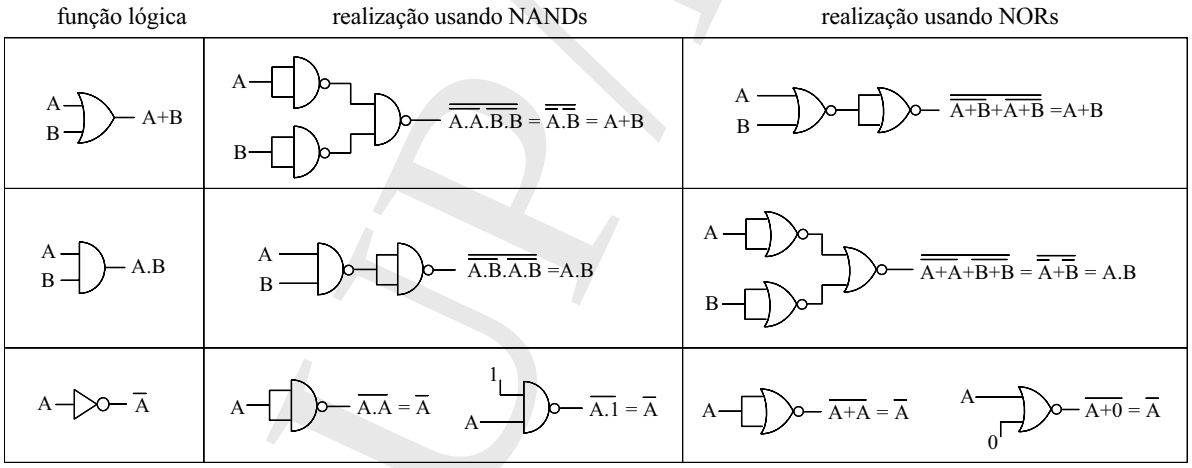


Figura 4.14: Realização das funções lógicas elementares com portas lógicas NAND e NOR.

O processo de minimização de um circuito lógico por forma a que contenha o menor número de NANDs ou NORs é um problema complexo e que ultrapassa o âmbito desta disciplina. No entanto, a representação de circuitos lógicos do tipo AND-OR (soma de produtos) ou OR-AND (produtos de somas) em circuitos equivalentes que apenas contenham portas NAND ou NOR, respectivamente, pode ser feita mediante um processo muito simples que se exemplifica na

figura 4.15.

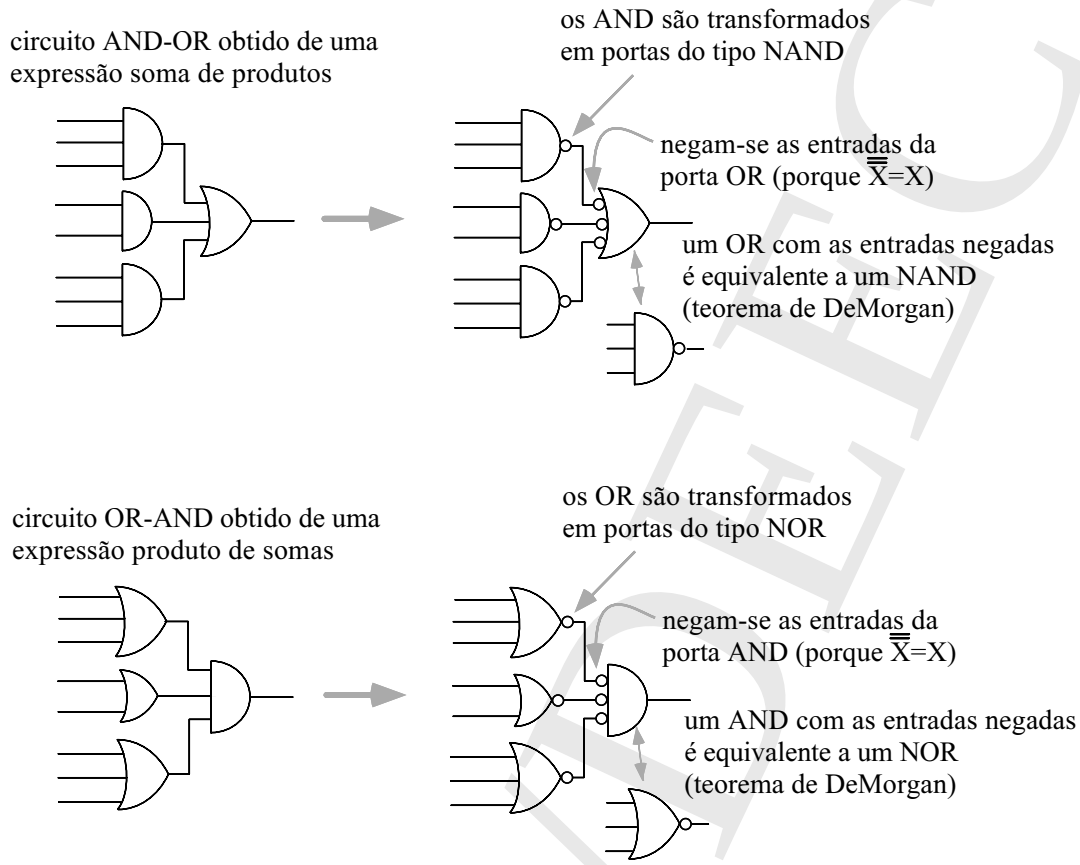


Figura 4.15: Tranformação de circuitos AND-OR e OR-AND em circuitos com apenas NANDs e NORs, respectivamente.

Um problema adicional que se pode colocar quando se pretende representar um circuito só com portas destes tipos, consiste em usar apenas portas lógicas com um número limitado de entradas, por exemplo 2. Como a função lógica NAND não possui a propriedade associativa como o AND ou o OR ($\overline{A.B.C} \neq (\overline{A.B}).\overline{C}$), não se pode decompor um NAND de N entradas em $N-1$ NANDs de 2 entradas, tal como se pode fazer com as portas lógicas AND e OR. Uma porta lógica NAND com N entradas pode ser decomposta em portas NAND com 2 entradas da forma que se mostra na figura 4.16 (um procedimento semelhante pode ser aplicado à transformação de portas lógicas NOR com N entradas em portas lógicas NOR com apenas duas entradas).

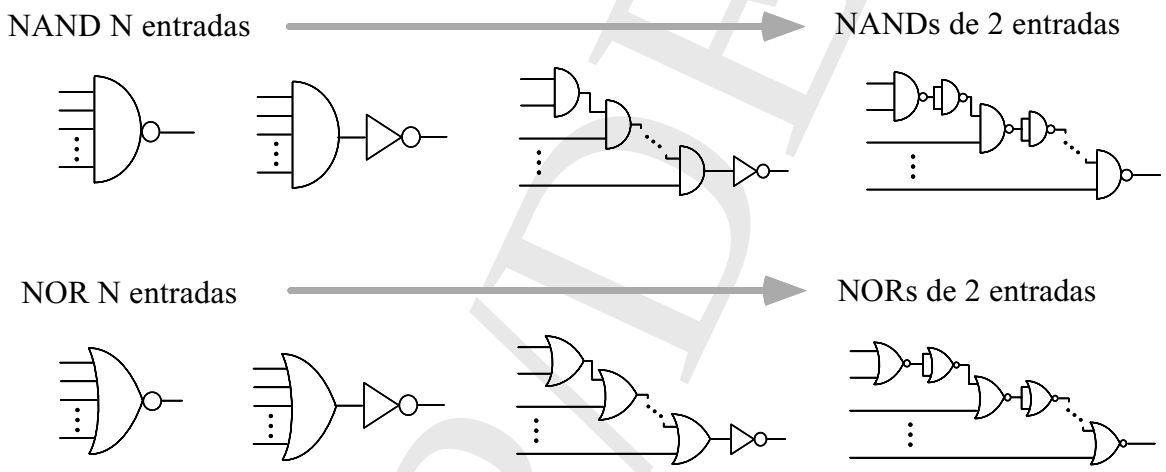


Figura 4.16: Decomposição de portas NAND e NOR com N entradas em portas NAND e NOR com apenas 2 entradas.

Capítulo 5

Funções combinacionais padrão

No capítulo anterior foram apresentadas técnicas que permitem descrever o comportamento de um circuito digital combinacional e obter uma sua realização interligando componentes que implementam as funções lógicas elementares e a que chamamos portas lógicas. Estas funções não são mais do que os operadores primitivos da álgebra booleana, embora com algumas variantes tais como portas lógicas com mais de duas entradas, entradas negadas ou saídas negadas.

Seguindo a metodologia proposta, é fácil projectar um circuito lógico *minimizado* que cumpre uma dada função, especificada, por exemplo, sob a forma de uma tabela de verdade ou uma expressão algébrica . Apesar desse processo nos ter permitido entender os aspectos básicos associados ao processo de projecto de circuitos digitais combinacionais, não é suficiente para resolver problemas complexos que se colocam em situações reais. Por exemplo, o processo de minimização de funções usando mapas de Karnaugh apenas é praticável nos termos em que foi estudado para circuitos de dimensão reduzida, até um máximo de 6 variáveis. A partir daí a utilização manual deste método torna-se difícil, embora existam aplicações computacionais que permitem efectuar processos de simplificação booleana similares.

Imagine-se, por exemplo, o simples problema de projectar um circuito que efectue a adição entre dois números de 10 *bits*, segundo as regras da adição binária estudadas no capítulo 2. Este circuito teria "apenas" 20 entradas (10 *bits* para cada um dos dois operandos), 10 saídas e obrigaria a projectar 10 funções com entre 2 e 20 variáveis[1] e uma função de 20 variáveis é representada por uma tabela de verdade com 1048576 linhas. E se os operandos passassem de 10 para 16 *bits* aquele número crescia para 4294967296...

E como se projectam então circuitos "a sério"? A resposta está num princípio fundamental que é geralmente aplicado a qualquer problema complexo em engenharia: dividir para conquis-

[1]Note que só a função que produz o *bit* mais significativo do resultado é que depende de todos os 20 *bits* dos dois operandos; no outro extremo, o *bit* menos significativo do resultado é apenas uma função de 2 duas variáveis que são os LSBs de cada um dos operandos.

tar. Assim, se o problema que se pretende resolver é demasiado complexo para ser tratado pelos métodos, ferramentas e demais recursos disponíveis, então divide-se em sub-problemas sucessivamente mais simples até se chegar a um nível que possa ser facilmente resolvido, ou então que para o qual já existam soluções prontas a usar.

Imagine, por exemplo, o problema de conceber e construir um automóvel que conta com muitas centenas de componentes, materiais e processos construtivos diferentes. Naturalmente que o construtor não tenta criar o carro de uma só vez, mas divide-o nas suas partes principais para serem tratadas por equipas especializadas: carroçaria, motor, chassis, interiores. Por sua vez, cada uma dessas equipas sub-divide novamente cada um dos seus problemas, recorrendo muitas vezes à reutilização de componentes já criados e testados noutros projectos. Imagine o que seria para os fabricantes de automóveis ter de conceber e construir um motor completamente novo sempre que fosse produzido um novo modelo!

Também no projecto de circuitos digitais, a divisão de um circuito complexo em subsistemas sucessivamente mais simples permite estabelecer uma relação *hierárquica* entre os diversos níveis de complexidade a que um sistema digital pode ser visto (figura 5.1). Como os métodos que estudámos no capítulo anterior conseguimos construir circuitos digitais usando apenas as funções elementares da álgebra de Boole (portas lógicas). A este nível de representação chama-se geralmente *nível lógico* e pode ser concretizado como um esquema de um circuito lógico (nível lógico no domínio estrutural), uma equação algébrica ou tabela de verdade (nível lógico no domínio funcional) ou um esquema da interligação dos circuitos electrónicos que realizam esse sistema digital (nível lógico no domínio físico). A construção de circuitos mais complexos permite "subir" um nível a que geralmente se chama nível de transferência entre registos ou abreviado para RTL (do Inglês *Register Transfer Level*—este termo fará mais sentido quando forem estudados os circuitos sequenciais, embora seja também correcto aplicá-lo a funções estritamente combinacionais). Este nível de representação caracteriza-se por ser formado por sub-circuitos encapsulados numa *caixa preta*[2] que, normalmente, operam sobre dados formados por vários *bits* e que realizam funções aritméticas ou lógicas elementares (por exemplo, adições ou multiplicações). Subindo mais um degrau para cima na hierarquia, chega-se àquilo a que normalmente se designa "nível de sistema", em que um sistema digital complexo, por exemplo um microprocessador, é representado à custa da interligação de blocos do tipo "caixa preta" encapsulando funções do nível RTL.

A figura 5.1 mostra a decomposição de um sistema digital complexo, até chegar ao nível hierárquico mais baixo (nível lógico) que é um circuito com apenas 3 entradas e 2 saídas e por

[2] "caixa preta" (em inglês *black box*) é um termo normalmente utilizado neste contexto para designar um circuito do qual se conhece a sua funcionalidade e o seu interface com o exterior—entradas e saídas—mas em que não se sabe (ou não é importante saber) de que forma é implementado.

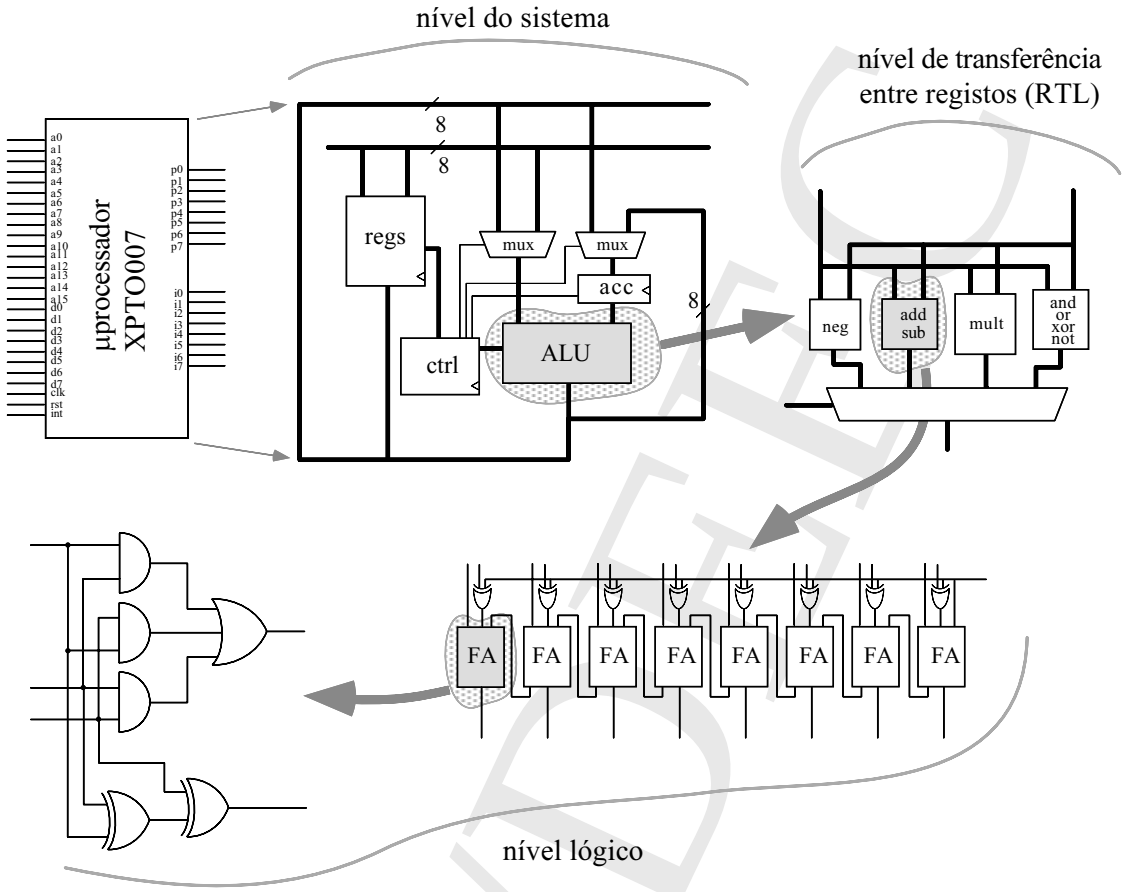


Figura 5.1: Decomposição hierárquica de um sistema digital complexo.

isso muito fácil de projectar com as ferramentas elementares que foram estudadas. Esse circuito simples é depois usado para construir um somador/subtractor que é uma peça fundamental em inúmeros circuitos digitais, desde os processadores dos nossos computadores pessoais até ao vulgar relógio de pulso numérico. Existem variadas formas de construir um circuito digital que realize essa função, e essas soluções já foram estudadas e estão disponíveis para um projectista que delas necessite. Por sua vez, esse bloco é integrado num sistema mais complexo (ALU—*Arithmetic and Logic Unit*), juntamente com outros circuitos do mesmo nível hierárquico como um multiplicador ou um negador.

A actividade de projecto de circuitos digitais complexos recorre assim, sempre que possível, a peças *pré-fabricadas* que realizam funções padrão, tal como a função somador exemplificada acima. Do mesmo modo que as funções lógicas elementares (portas lógicas) existem disponíveis em circuitos integrados da série 74 (capítulo 4), há também circuitos desta família que integram as funções que serão estudadas neste capítulo. Os dispositivos electrónicos desta categoria são normalmente designados por MSI (*Medium Scale Integration*, e que apresentam complexidades equivalentes e várias dezenas de portas lógicas. A generalidade das aplicações

computacionais para desenho de esquemas de circuitos digitais dispõem igualmente de blocos pré-construídos que realizam esses tipos de funções, alguns dos quais imitam os circuitos integrados MSI da série 74.

Neste capítulo serão estudadas as funções lógicas combinacionais padrão mais importantes, de maneira a poderem ser empregues no projecto de sistemas digitais mais complexos. Para cada função será mostrada a sua implementação ao nível lógico, exemplos de aplicação e referidos alguns circuitos integrados da série 74 que as realizam.

5.1 Regras para desenho de circuitos

O desenho de esquemas de circuitos lógicos, quer no papel quer recorrendo a aplicação computacionais, tem vindo a perder importância devido principalmente à dificuldade de construir e manter desenhos com um elevado número de componentes cada vez mais complexos. Imagine-se o que seria desenhar o esquema lógico completo de um processor actual que tem uma complexidade equivalente a alguns milhões de portas lógicas!

Actualmente o projecto de sistemas digitais complexos é feito recorrendo a linguagens textuais parecidas com as linguagens de programação correntes, mas que foram concebidas para descrever funções de circuitos digitais e que são tratadas automaticamente por aplicações computacionais que realizam diversas fases do processo de projecto. No entanto, o desenho de esquemas continua a ser interessante para representar circuitos de pequena complexidade já que permite ver de uma só vez os vários componentes que o compõem e a forma como se encontram ligados entre si.

Ao contrário das funções elementares que são representadas por símbolos padrão cuja forma determina a função realizada, não existem símbolos uniformizados que representam as várias funções padrão mais utilizadas. Assim, é usual representar circuitos que realizam funções complexas como apenas uma caixa (rectângulo) no qual se identificam as entradas e as saídas e de alguma forma se identifica o tipo de função realizada, anotando, por exemplo, o nome da função. Apesar disto só poder ser feito de forma rigorosa à custa de uma representação formal (por exemplo com uma tabela de verdade ou expressão algébrica), a simples indicação do tipo de função padrão realizada, juntamente com o uso de nomes sugestivos para identificar as entradas e saídas, permite criar esquemas de circuitos lógicos claros e fáceis de interpretar.

Nesta secção apresentam-se algumas regras básicas que devem ser seguidas para obter desenhos de circuitos agradáveis de ler e que possam ser claramente interpretados (descubra as semelhanças entre os 2 circuitos da figura 5.2). Embora o objectivo imediato seja a sua aplicação ao desenho de esquemas de circuitos, estes princípios podem ser igualmente aplicados quando são usadas outras formas de “desenho”, nomeadamente representações textuais usando lingua-

gens de descrição de *hardware*.

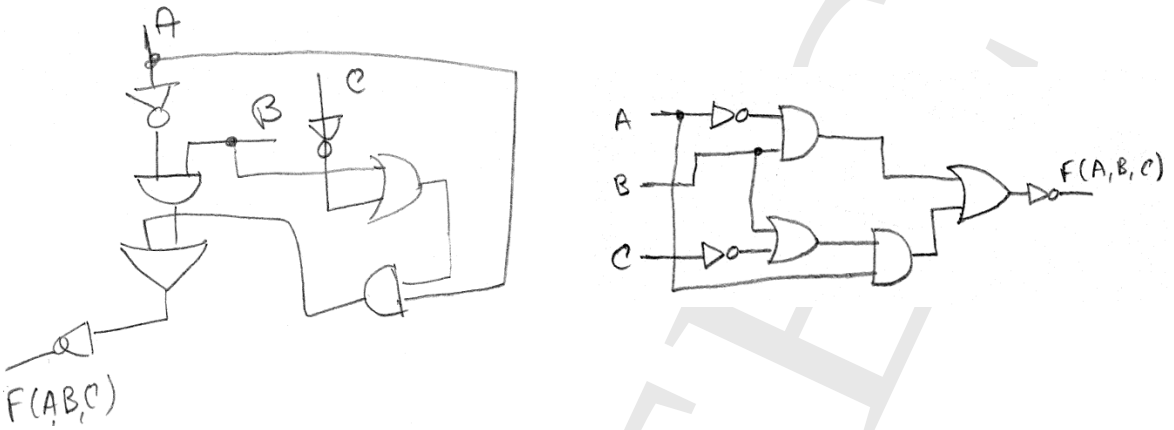


Figura 5.2: Um esquema confuso e um esquema claro.

5.1.1 Representação de barramentos

Nas representações gráficas de circuitos lógicos que foram utilizadas até aqui as interligações entre componentes (portas lógicas) foram desenhadas como traços que "transportam" apenas um *bit* desde uma saída até uma entrada. Quando se desenham circuitos lógicos mais complexos é muitas vezes necessário representar ligações entre componentes que transportem vários *bits* que têm um certo significado como um todo (por exemplo, um número de 16 *bits*). Em vez dum conjunto de 16 linhas é desenhado um *barramento* como uma única linha que representa um agrupamento de vários *bits*, e a identificação do número de *bits* que um barramento transporta pode ser feito de como se mostra nos exemplos da figura 5.3. Alguns programas para desenho de esquemas de circuitos representam também os barramentos com linhas mais grossas do que as usadas para ligações de 1 *bit*, embora em desenhos feitos à mão não seja fácil desenhar linhas finas e grossas.

5.1.2 Nomes dos sinais

Uma regra importante que deve ser seguida no desenho de um circuito lógico é a atribuição de um nome aos vários sinais envolvidos num circuito, em especial às suas entradas e saídas. Os nomes devem ser claros e, na medida do possível, sugerir a função que esse sinal assume no contexto do circuito. Embora não existam estabelecidas convenções universalmente aceites, devem ser usadas as mesmas que são geralmente aceites por qualquer aplicação computacional para desenho de circuitos, e que são basicamente iguais às seguidas para definir variáveis na generalidade das linguagens de programação:

- um nome pode ser composto por caracteres alfabéticos, numéricos ou *underscore*, mas o primeiro caracter deve ser alfabético (por exemplo SEL, OPR_A, I6); em alguns esquemas aparece por vezes o caracter '#' no final de um nome ou o caracter '/' no início (OE# ou /OE) para identificar sinais que são activos no nível lógico baixo, i.e. a acção que realizam é activada com zero em vez de um.

- os nomes atribuídos aos barramentos devem identificar também o número de *bits* que o barramento transporta e a identificação numérica de cada um; uma forma usual[3] consistem em representar, a seguir ao nome e dentro de parêntesis rectos, um par de números que identifica os *bits* desse barramento. Por exemplo, o nome A_BUS[7:0] representa um barramento com 8 *bits* designado por A e constituído pelos *bits* 7, 6, 5, 4, 3, 2, 1 e 0, onde o *bit* 0 é o da direita (menos significativo) e o *bit* 7 é o da esquerda (o mais significativo); A_BUS[4] representa apenas o *bit* 4 e A_BUS[3:0] representa um barramento de 4 *bits* composto pelos *bits* 3, 2, 1 e 0 do barramento A_BUS. Outras representações de nomes de barramentos usam parêntesis curvos ou os sinais '<' e '>' em vez de parêntesis rectos (A_BUS(7:0) ou A_BUS<7:0>).

- para evitar uma grande densidade de desenho de linhas (fios e barramentos) em esquemas complexos, considera-se que 2 sinais que tenham o mesmo nome estão electricamente ligados entre si, sem ser necessário completar essa ligação "no papel", como por exemplo o *bit* menos significativo do barramento A no exemplo da figura 5.3.

5.1.3 Entradas e saídas

Qualquer circuito tem entradas e saídas que devem ser claramente identificadas num esquema. Uma regra que é geralmente seguida no desenho de um esquema de um circuito lógico consiste em desenhar preferencialmente as entradas do lado esquerdo ou em cima e as saídas do lado direito ou em baixo, segundo o fluxo "natural" da informação ao longo do circuito da esquerda para a direita e/ou de cima para baixo. Quando se desenha uma caixa representado um bloco já existente ou um circuito já construído, devem ser representados nomes para as suas entradas e saídas *dentro* dessa caixa, tal como é mostrado no exemplo da figura 5.3. A funcionalidade de um bloco desse tipo deve ser descrita em função dos nomes identificados dentro desse símbolo (por exemplo, a entrada IN[3:0] e a saída SEL do símbolo representado na parte inferior da figura 5.3).

[3] Apesar de existirem várias outras formas de representar barramentos em esquemas de circuitos lógicos, os exemplos apresentados seguirão esta notação que é a adoptada por uma linguagem de descrição de *hardware* muito usada em ambientes de projecto industrial—Verilog HDL

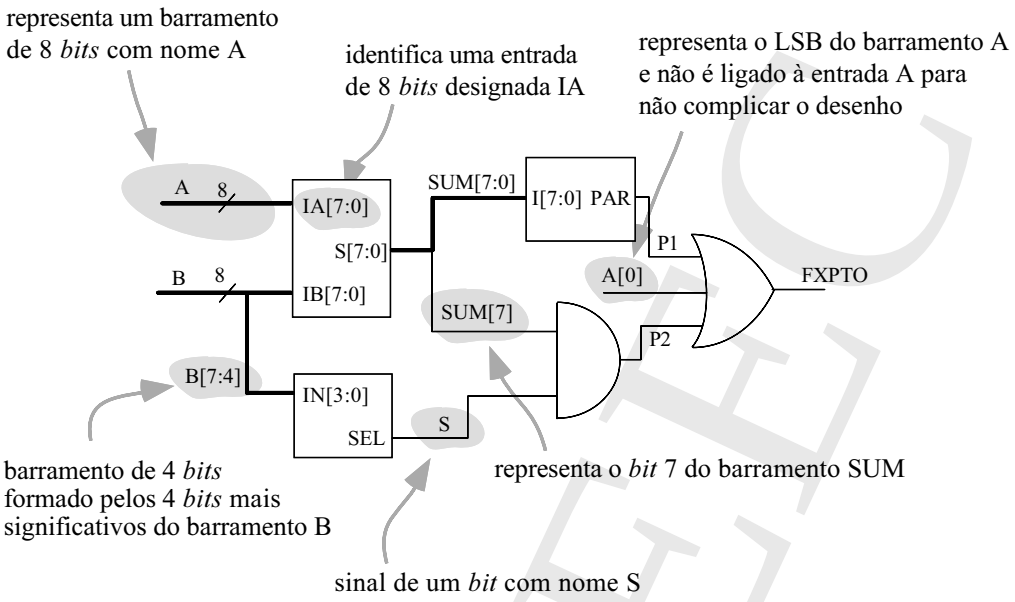


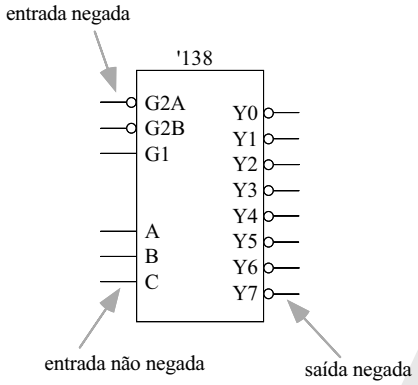
Figura 5.3: Exemplos de nomes a atribuir a barramentos, sinais e entradas ou saídas num circuito lógico.

5.1.4 Entradas e saídas negadas

Certos circuitos apresentam entradas que são activas (i.e. realizam uma certa função) quando o seu nível lógico é zero em vez de 1. Geralmente isso é feito assim porque há vantagens no que diz respeito ao seu funcionamento eléctrico e à integração com outros dispositivos e que será abordado mais tarde.

No desenho do símbolo (caixa preta) de um circuito, as entradas e saídas deste tipo são geralmente identificadas com um pequeno *círculo* desenhado junto ao terminal, significando assim que existe uma negação lógica entre o terminal e o interior do circuito. Outras formas possíveis para identificar entradas ou saídas deste tipo consistem em preceder o nome do terminal por '/' ou então usar os sufixos '#' ou '_L' (por exemplo /SEL, SEL#, SEL_L). Na figura 5.4 mostra-se a representação de um símbolo lógico com entradas e saídas negadas e a descrição da sua funcionalidade numa tabela de verdade.

Este tipo de entradas e saídas conduz por vezes a alguma confusão na interpretação da funcionalidade de circuitos (mesmo em folhas de características de circuitos integrados comerciais!), por não ser claro se a funcionalidade representada se refere às entradas e saídas antes ou depois das negações. Para evitar interpretações incorrectas, as tabelas de verdade que se apresentam ao longo deste capítulo relativas à funcionalidade de circuitos com entradas ou saídas activas com o nível lógico zero, referem-se sempre aos valores lógicos vistos do exterior do circuito.



C	B	A	G1	G2A	G2B	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
x	x	x	0	x	x	1	1	1	1	1	1	1	1
x	x	x	x	1	x	1	1	1	1	1	1	1	1
x	x	x	x	x	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	0	1	0	1	1	1	1	1	1
0	1	0	1	0	0	1	1	0	1	1	1	1	1
0	1	1	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1	0	1	1
1	1	0	1	0	0	1	1	1	1	1	1	0	1
1	1	1	1	0	0	1	1	1	1	1	1	1	0

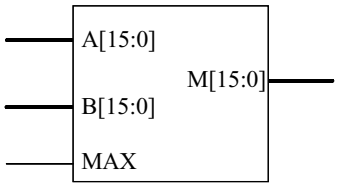
Figura 5.4: Representação do símbolo e da tabela de verdade de um circuito lógico com entradas e saídas negadas (descodificador ’138, estudado mais à frente na secção 5.3.3).

5.2 Um exemplo: calculador do máximo e do mínimo

Para iniciar o estudo de algumas funções padrão iremos analisar um problema que permitirá entender a necessidade de realizar um projecto de forma hierárquica e identificar algumas funções importantes.

O problema consiste em projectar um circuito digital com 33 entradas e 16 saídas e que permita calcular o máximo ou o mínimo entre 2 números de 16 *bits* com sinal segundo a convenção complemento para 2. As entradas são dois números A e B de 16 *bits* cada um e um *bit* adicional MAX que selecciona a função a realizar: calcular o máximo ou calcular o mínimo; as 16 saídas apresentam o número A ou B consoante a relação de grandeza entre eles e o valor da entrada MAX. A figura 5.5 mostra o símbolo do circuito que se pretende projectar e apresenta a funcionalidade pretendida para o circuito. Note que a descrição *funcional* é apresentada numa linguagem próxima da nossa linguagem natural e diz sem ambiguidades de que forma se pretende que o circuito funcione.

Como já foi referido acima, a aplicação da metodologia de projecto estudada no capítulo 4 é impraticável: com 33 entradas temos tabelas de verdade com 8589934592 linhas! Como podemos então construir este circuito? Em primeiro lugar, note que as 16 saídas só podem apresentar o valor presente na entrada A ou o valor presente na entrada *B*. E qual é a condição



símbolo lógico do detector de máximo

```
se MAX = 1 então
        se A >= B então
                M = A
        senão
                M = B
senão
        se A < B então
                M = A
        senão
                M = B
```

```
se (MAX = 1 e A >= B) ou (MAX = 0 e A < B) então
        M = A
senão
        M = B
```

duas descrições funcionais alternativas

Figura 5.5: Detector de máximo e mínimo: símbolo e descrição funcional.

para que seja apresentado o valor de A ou o de B? Se $MAX = 0$ queremos determinar o mínimo entre A e B devendo por isso ser escolhido para a saída o valor A se for verdade que é $A < B$, ou então o valor B se for $A >= B$; se $MAX = 1$ queremos obter na saída o máximo entre A e B: se $A < B$ deve ser escolhido B e se $A >= B$ deve ser escolhido A. Podemos assim estabelecer uma tabela de verdade entre a entrada MAX e a condição $A < B$ (podemos representar esta condição com uma variável A_M_B que vale 1 quando é $A < B$ e 0 quando $A >= B$) e uma saída SEL_A que é 1 se é escolhida a entrada A ou 0 se é escolhida a entrada B (figura 5.6).

MAX	A_M_B	SEL_A	
0	0	0	(escolhe B)
0	1	1	(escolhe A)
1	0	1	(escolhe A)
1	1	0	(escolhe B)

Figura 5.6: Tabela de verdade para o circuito selector de máximo.

Esta função é muito fácil de projectar recorrendo às técnicas já conhecidas, obtendo-se a seguinte expressão minimizada na forma soma de produtos:

$$SEL\_A = \overline{MAX}.A\_M\_B + MAX.\overline{A\_M\_B}$$

Esta função é muito simples e é uma peça elementar usada em circuitos aritméticos, nomeadamente somadores, subtractores e comparadores, como veremos mais à frente neste capítulo. Chama-se a esta função OU-exclusivo (em Inglês *exclusive-OR* ou XOR) e só difere da função

OU quando as duas entradas são iguais entre si—a saída é 1 quando uma entrada é 1 ou a outra entrada é 1 mas é zero quando as duas entradas são 1 ao mesmo tempo. Pela importância que apresenta, esta função é muitas vezes tratada como uma porta lógica adicional para a qual existe um símbolo próprio (figura 5.7).

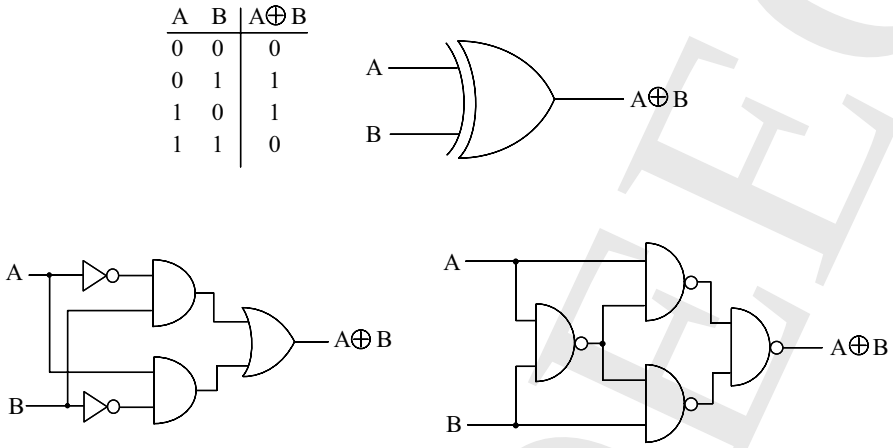


Figura 5.7: Porta lógica OU-exclusivo (XOR): tabela de verdade, símbolo lógico e duas implementações possíveis.

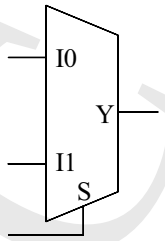
O passo seguinte consiste em construir um circuito que permita escolher para um conjunto de 16 saídas um de 2 conjuntos de 16 entradas (os valores *A* ou *B*) em função do valor do *bit* *SEL_A* gerado pela função criada antes. Podemos constatar, numa análise imediata, que temos de novo um circuito com 32 entradas e que por isso não o conseguiremos projectar de uma só vez. No entanto, uma análise mais atenta permite-nos concluir que podemos decompor este circuito em 16 circuitos iguais e muito mais simples, um para cada *bit* de A, B e do resultado: note que a escolha pode ser feita separadamente para cada um dos 16 *bits* de *A* e *B*, o que se resume assim num conjunto de 16 funções iguais, cada uma com apenas 3 entradas. A tabela de verdade dessa função e o circuito lógico minimizado são apresentados na figura 5.8. Este circuito chama-se selector (ou multiplexador) e de uma forma geral permite escolher para uma saída uma entre várias entradas que são seleccionadas por um conjunto de sinais de selecção.

Até agora resolvemos apenas parte do problema: *sabendo* se é verdade que $A < B$, o circuito selecciona para a saída o máximo ou o mínimo entre *A* e *B* consoante o valor da entrada *MAX* é 1 ou 0, respectivamente. Este circuito é representado pelo esquema da figura 5.9 e contém apenas uma porta lógica XOR e um multiplexador 2-1 de 16 *bits* (que é feito com 16 multiplexadores 2-1 de um *bit*).

Para completar o projecto falta agora construir um circuito com 32 entradas que receba os 2 operandos *A* e *B* e produza o sinal *A_M_B* que, como foi visto, deve valer 1 se é $A < B$ e 0 caso contrário. A um circuito deste tipo chama-se *comparador de magnitude* e pode ser

entrada de selecção →

S	I1	I0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



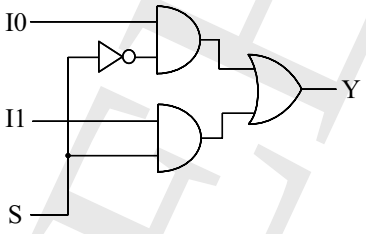


Figura 5.8: Circuito selector de 2 entradas: tabela de verdade, circuito lógico e representação simbólica; a saída Y é igual à entrada I0 quando o sinal de selecção é 0, e é igual à entrada I1 no caso contrário.

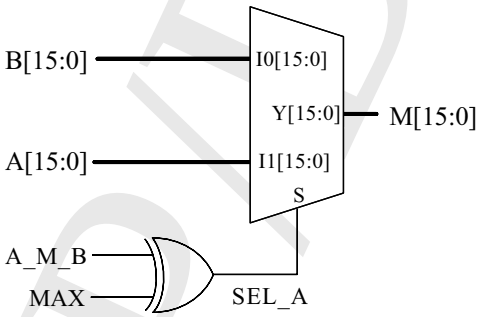


Figura 5.9: Selector do máximo ou mínimo; note que este circuito tem por entrada o sinal *A_M_B* que representa o resultado da comparação de magnitude entre as entradas *A* e *B* e que será estudado a seguir.

construído para operar sobre dados apenas positivos, ou com sinal segundo as diferentes formas de representação que foram estudadas. No nosso exemplo pretendemos um comparador de magnitude para números de 16 *bits* que os "entenda" como números com sinal em complemento para 2. Mais uma vez, não é possível projectar este circuito como um todo e por isso temos de simplificá-lo sucessivamente até chegar a funções conhecidas que já existam construídas (por exemplo portas lógicas XOR ou multiplexadores) ou então funções simples que sejam fáceis de projectar com as ferramentas de que dispomos[4].

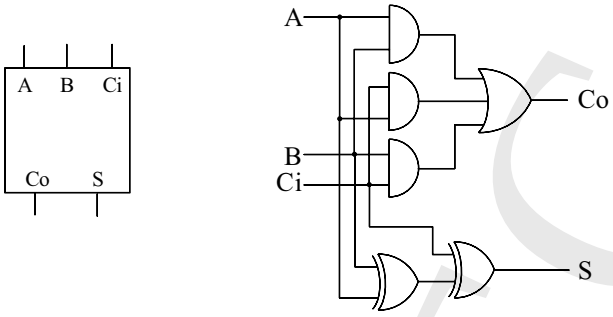
[4]Na verdade, comparadores são funções lógicas muito usadas para as quais já existem realizações optimizadas,

Como se constrói um comparador de magnitude? Relembremos o que foi estudado no capítulo 2 a respeito da representação complemento para 2 e das operações aritméticas de adição e subtracção com operandos representados segundo esta convenção. Como o *bit* mais significativo de um número representa o seu sinal, então podemos usar esse *bit* do resultado da diferença $A-B$ para representar a variável A_M_B que indica se é $A<B$: se $A<B$ então a diferença $A-B$ será um valor negativo e se $A>=B$ esse valor será positivo (note que o valor zero é aqui entendido como positivo). Há ainda um problema para resolver: a possível ocorrência de *overflow* quando se realiza essa operação de subtracção, o que pode fazer com que o resultado não seja correcto (por exemplo, realizando em 8 *bits* a subtracção $(+100)-(-100)$ dá como resultado $+200$ que não é representável em complemento para 2 com 8 *bits*). Isto pode ser facilmente contornado se os nossos operandos A e B forem representados com mais um *bit*: a operação de subtracção realizada com 17 *bits* nunca conduzirá a uma situação de *overflow* porque os valores dos operandos nunca excedem a gama de representação possível com 16 *bits* e complemento para 2.

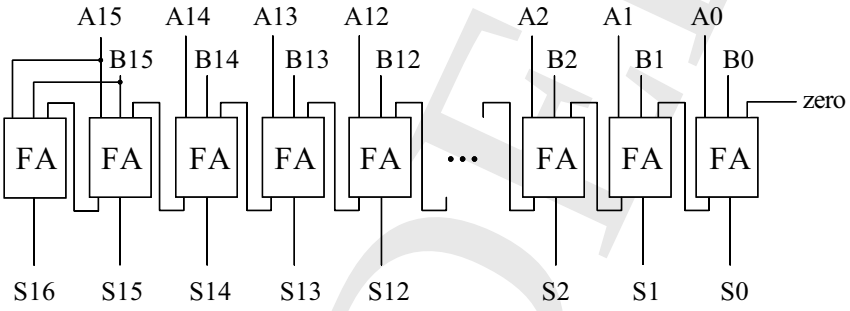
Podemos agora reduzir o problema de projectar um comparador de magnitude a desenhar um circuito que efectue a subtracção binária entre 2 números de 17 *bits*. Com o que já sabemos sobre aritmética binária, podemos concluir que efectuar a subtracção $A-B$ é o mesmo que realizar a adição $A+(-B)$. Sabemos também que trocar o sinal a um número (calcular o simétrico de B, $-B$) pode ser feito negando todos os seus *bits* a adicionar uma unidade. Assim, $A-B$ pode ser escrito como $A+\overline{B}+1$, onde $\overline{B}$ representa o operando B com todos os seus *bits* trocados. Ficamos assim com um novo problema que é mais simples do que o anterior porque realiza uma operação já estudada e que consiste em construir um circuito que realize a adição binária.

A adição binária pode ser implementada recorrendo ao método estudado no capítulo 2 para realizar a operação manualmente. Cada *bit* do resultado é obtido adicionando um *bit* de cada um dos operandos e um *bit* de transporte resultante da adição dos *bits* imediatamente à direita. Um circuito somador para números com N *bits* pode assim ser construído ligando em cascata N circuitos elementares, cada um dos quais adiciona 3 *bits* e produz um resultado de 2 *bits*: o menos significativo é um *bit* do resultado e o mais significativo representa o transporte para o andar seguinte. Cada um destes circuitos elementares chama-se *somador completo* ou em Inglês *full adder*, existindo uma versão simplificada designada *meio somador* (*half adder*) que não tem como entrada o *bit* de transporte anterior. Na figura 5.10 mostra-se a implementação lógica de um *full adder*, um somador de números de 16 *bits* e, finalmente, o subtractor que pode ser usado para realizar a operação de comparação necessária para o nosso circuito. Note que a

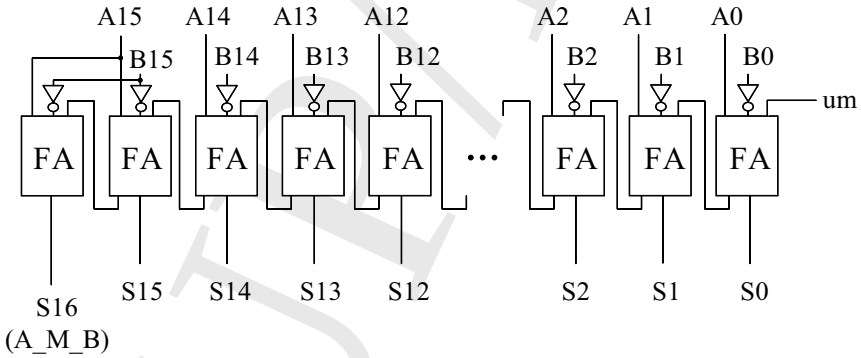
quer em termos de circuito lógico quer como circuitos electrónicos integrados. O nosso projecto poderia parar já neste ponto, já que a solução para a parte que estamos agora a criar é simplesmente um comparador de magnitude.



somador completo (*full adder*)



somador de dois números de 16 *bits* produzindo resultado de 17 *bits* (S=A+B)



subtractor de dois números de 16 *bits* produzindo resultado de 17 *bits* (S=A-B)
o *bit* de sinal é S16 (A_M_B) e representa o resultado da comparação A<B

Figura 5.10: Somador completo e circuitos somador e subtractor de 17 *bits*; note que é feita a extensão de sinal dos operandos *A* e *B* copiando os respectivos *bits* mais significativos, para que se possa obter um resultado de 17 *bits* em que nunca ocorre *overflow*.

adição de uma unidade para obter o simétrico de *B* pode ser conseguida apenas colocando o *bit* de transporte de entrada do somador menos significativo igual a 1.

Antes de reunir todos os circuitos que projectamos até aqui, podemos ainda proceder a uma simplificação importante no nosso comparador. Como já concluímos antes, do resultado da

diferença produzido pelo subtractor apenas nos interessa o *bit* mais significativo que representa o sinal desse resultado e os restantes *bits* podem ser ignorados. Se essas saídas não são usadas então os circuitos lógicos que as produzem podem ser seguramente eliminados. Analisando o circuito lógico de um *full adder* mostrado na figura 5.10, podemos então concluir que o circuito constituído pelas duas portas lógicas XOR não é necessário e pode ser removido, o que reduz significativamente a complexidade de cada *full adder* (note que, como se mostrou na figura 5.7, uma porta lógica XOR pode ser realizada com 4 portas NAND de 2 entradas). Claro que depois desta simplificação o circuito já não faz a subtracção binária realizando apenas a função de comparador de magnitude, bom como já não se pode chamar *full adder* a cada um dos seus circuitos elementares (figura 5.11).

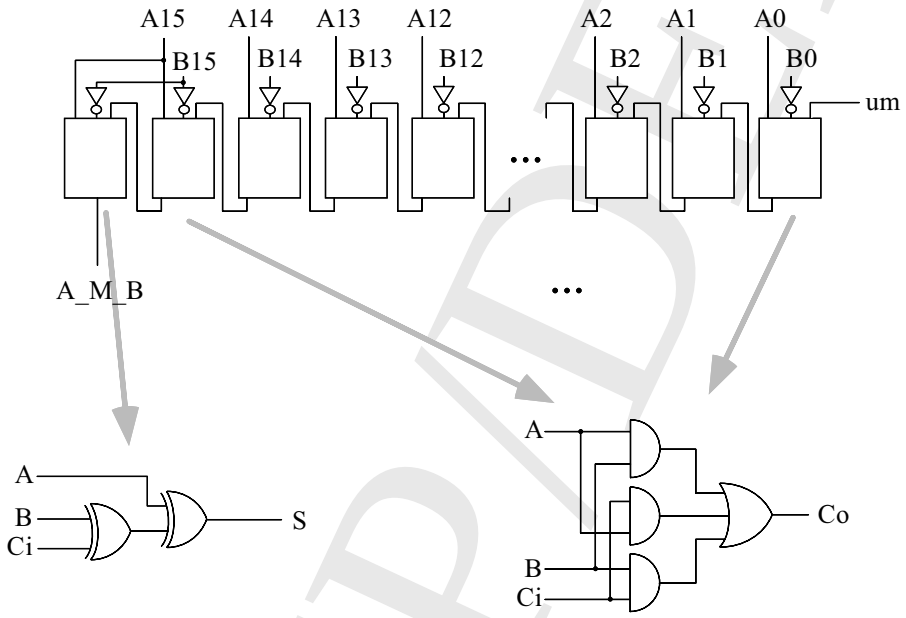


Figura 5.11: Simplificação do subtractor num comparador de magnitude.

Podemos finalmente reunir todos os elementos e desenhar o circuito final recorrendo aos circuitos que acabámos de construir: um comparador de magnitude, uma porta XOR e um multiplexador (figura 5.12).

Estes circuitos, juntamente com outros que serão estudados neste capítulo, constituem um conjunto de funções padrão para as quais já existem implementações disponíveis para o projectista e que podem ser usadas para a construção de circuitos mais complexos. Algumas destas funções são disponibilizadas em circuitos integrados da série 74 e permitem facilmente construir pequenos sistemas digitais com um número reduzido de componentes. As funções padrão que iremos estudar em detalhe são:

- **Codificadores e descodificadores** são circuitos que, de uma forma geral, traduzem um código binário noutro código com um número diferente de *bits*.

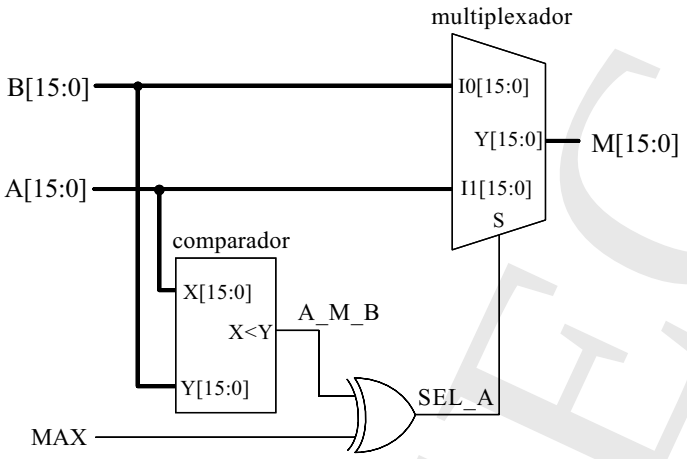


Figura 5.12: Realização do circuito detector de máximo e mínimo recorrendo a funções padrão: um comparador de magnitude, um multiplexador e uma porta XOR.

- **Multiplexadores:** a função de selecção já foi introduzida, mas serão apresentadas outras variantes e aplicações.

- **Circuitos aritméticos:** para além do somador, subtractor e comparador de magnitude que já foram apresentados, existem várias outras funções que se enquadram nesta categoria: outros tipos de comparadores, multiplicadores, multiplicadores por constantes e unidades mais complexas que realizam várias operações aritméticas e lógicas (ALU–*Arithmetic and Logic Unit*).

5.3 Codificadores e descodificadores

Codificadores e descodificadores são funções que permitem traduzir um código binário com N *bits* noutro código com M *bits*, sendo normalmente $N \neq M$. O termo codificador emprega-se geralmente para designar circuitos que traduzem um código noutro com um número menor ou igual de *bits* e descodificador quando o código de saída tem mais *bits* do que o código de entrada.

Um exemplo de descodificador foi já apresentado no capítulo 4, secção 4.2.4: descodificador de BCD para 7 segmentos. Este circuito transforma um código de 4 *bits*, representado um dígito decimal entre 0 e 9, noutro código com 7 *bits*que permite acender os LEDs adequados de um mostrador de 7 segmentos de maneira a visualizar o dígito correspondente.

5.3.1 Codificador binário

Para ilustrar uma aplicação prática deste tipo de circuito considere-se o problema de construir um sistema digital afixe num mostrador de 7 segmentos (ver capítulo 4, secção 4.2.4) o número correspondente ao andar em que se encontra um elevador, num edifício com R/C e 3 andares. O elevador dispõe de um conjunto de sensores que dão a informação do andar em que o elevador se encontra: quando o elevador está estacionado ou a passar no andar i o sensor S_i apresenta o valor lógico 1 e todos os outros têm o valor lógico 0 (figura 5.13).

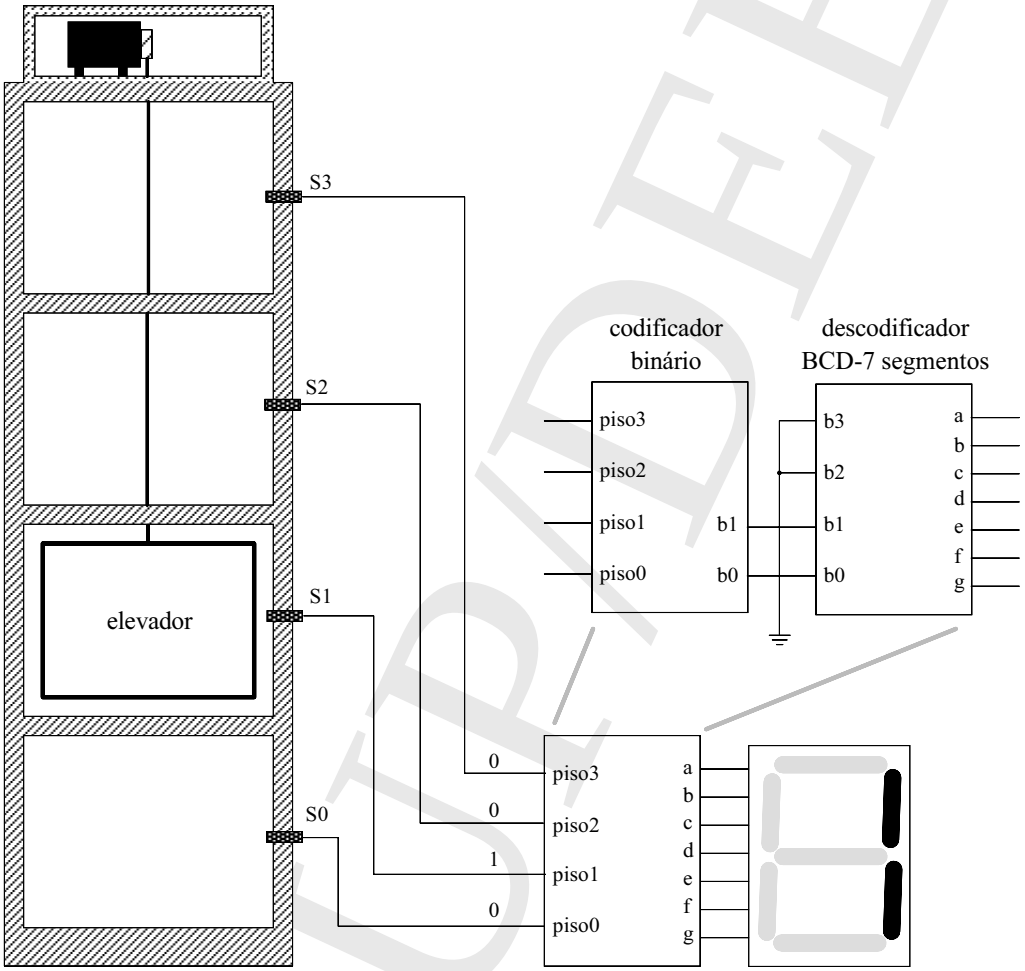
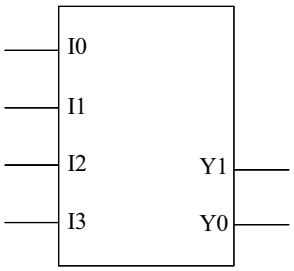


Figura 5.13: Um elevador num edifício com R/C e 3 andares.

O circuito a projectar terá 4 entradas (os 4 sensores de posição do elevador) e as 7 saídas que vão alimentar os 7 LEDs do mostrador. Para se poder reutilizar o descodificador BCD para 7 segmentos, será necessário construir agora um *codificador* que traduza o conjunto de 4 *bits* produzido pelos sensores num número com 2 *bits* que represente o número do andar em que o elevador está (0, 1, 2 ou 3). Embora este circuito tenha 4 entradas e por isso seja representado por uma tabela de verdade com 16 linhas, muitos dos valores dessa tabela serão indiferentes,

atendendo ao contexto em que este circuito será utilizado: como só pode estar uma entrada activa de cada vez (admite-se que o elevador nunca activa 2 sensores ao mesmo tempo), então a saída apenas será definida nessas situações em que os códigos de entrada são 1000, 0100, 0010, 0001. Todos os outros casos nunca ocorrerão e como tal a saída poderá ser considerada indiferente. A figura 5.14 mostra a tabela de verdade e o circuito lógico minimizado para esta função.



I3	I2	I1	I0	Y1	Y0
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	x	x
0	1	0	0	1	0
0	1	0	1	x	x
0	1	1	0	x	x
0	1	1	1	x	x
1	0	0	0	1	1
1	0	0	1	x	x
1	0	1	0	x	x
1	0	1	1	x	x
1	1	0	0	x	x
1	1	0	1	x	x
1	1	1	0	x	x
1	1	1	1	x	x

$Y1 = I3 + I2$

$Y0 = I3 + I1$

Figura 5.14: Codificador binário de 4 para 2: tabela de verdade e implementação lógica optimizada. Note que como a entrada $I0$ não é usada em nenhum termo das funções $Y1$ e $Y0$, então poderia ser removida do circuito, bem como o sensor colocado no piso 0!

Considere agora que, quando o elevador se desloca entre dois pisos ocorre um período de tempo durante o qual nenhum sensor é activado e por isso a entrada do descodificador é 0000. Como com a minimização lógica que foi realizada a saída do codificador será 00 nesse caso, a consequência prática disso será que nesse período o mostrador afixará zero, o que não é desejável. Uma forma de resolver este problema consiste em acrescentar uma saída no codificador que é 1 quando pelos menos uma entrada for 1 (note que esta função não é mais do que o OU lógico das 4 entradas), servindo para distinguir a saída 00 quando o elevador está mesmo no piso zero e as entradas são 0001, da saída 00 que é produzida quando as entradas são 0000 porque nenhum sensor está a ser activado[5]. Embora só por si esta modificação não resolva o problema, o descodificador de 7 segmentos poderia ser modificado acrescentando-lhe uma entrada que, sendo activada, desligasse todas as saídas apagando dessa forma o mostrador. Na realidade, este tipo de descodificador dispõe normalmente de uma entrada de controlo que permite desactivar todas as saídas independentemente do valor presente nas restantes entradas.

Um codificador como o apresentado funciona perfeitamente na situação ilustrada. No en-

[5]Note que este valor poderia ser diferente se o processo de minimização tivesse sido feito de outra forma.

tanto, se ocorrer uma entrada que se assumiu nunca poder acontecer (por exemplo 1010), as saídas continuarão a apresentar um valor entre 0 e 3, não havendo forma de distinguir essa situação de uma saída "normal". Por essa razão, pode ser desejável ter num codificador binário uma saída adicional que permite distinguir entre entradas "legais" e entradas inválidas.

5.3.2 Codificadores de prioridade

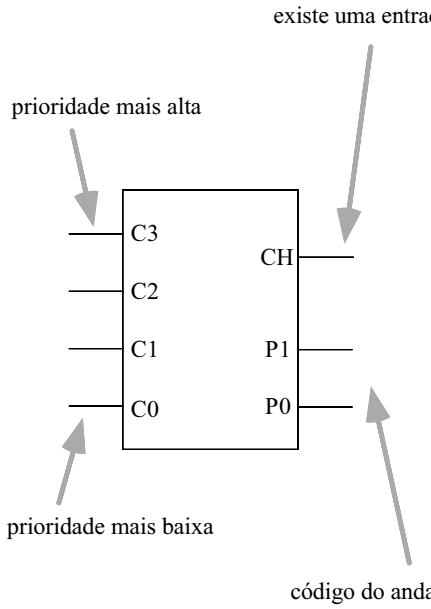
O codificador binário apresentado antes apenas pode ser usado quando as entradas a *codificar* apresentam um dos valores legais. No entanto, existem situações em que é necessário efectuar a operação de codificação como foi ilustrada com o exemplo anterior, mas em que também podem ser activadas mais do que uma entrada em simultâneo, como se mostra no exemplo a seguir.

Considere agora que o sistema de controlo do elevador tem uma entrada onde deve ser colocado um número de 2 *bits* que represente o andar de onde o elevador foi chamado. Como existem 4 botões de chamada, um por piso, é necessário traduzir esse código de 4 *bits* num código de 2 *bits*, de forma semelhante ao realizado pelo codificador anterior. Há, no entanto, uma diferença importante: enquanto que no sistema anterior era garantido que nunca podiam ser activados 2 sensores em simultâneo, agora isso já não acontece porque podem estar duas pessoas a chamar o elevador em pisos diferentes e ao mesmo tempo!

A solução consiste em atribuir *prioridades* às entradas e colocar na saída o código da entrada que está activa e que tem prioridade mais elevada. Esta solução torna legais todas as combinações das entradas, embora continue a ser desejável ter uma saída que distinga o caso em que as entradas são todas zero (não activas) de todos os outros casos em que pelo menos uma entrada está activa. Na figura 5.15 mostra-se a tabela de verdade de um codificador de prioridade de 4 *bits* e sua implementação lógica minimizada. Às entradas C3, C2, C1 e C0 são atribuídas prioridades decrescentes, e na saídas P1 e P0 é apresentado um código binário *i* de dois *bits* que identifica a entrada *Ci* de prioridade mais elevada que está activa.

Codificadores de prioridade em CIs da série 74

A função codificador de prioridade existe disponível nos circuitos integrados '148 e '147, cujos símbolos e tabela de verdade se apresentam na figura 5.16. O '147 realiza a codificação de prioridade de 9 entradas para um código BCD de 4 *bits* em que o código zero significa que não existe nenhuma entrada activa. O '148 é um codificador de prioridade de 8 *bits* possuindo a entrada EI (*enable input*) as saídas EO (*enable output*) e GS (em group select ou *got something*) que facilitam a construção de descodificadores de prioridade com maior número de entradas usando este tipo de componente.



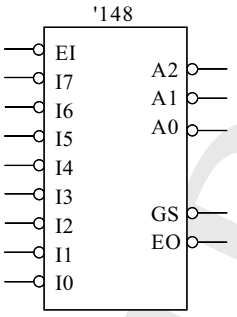
C3	C2	C1	C0	P1	P0	CH
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	0	0	1	1
0	0	1	1	0	1	1
0	1	0	0	1	0	1
0	1	0	1	1	0	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	1	1
1	0	0	1	1	1	1
1	0	1	0	1	1	1
1	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1

$P0 = C3 + C1.\overline{C2}$

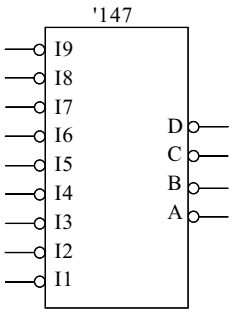
$P1 = C3 + C2$

$CH = C3 + C2 + C1 + C0$

Figura 5.15: Codificação do andar chamado de um elevador com um codificador de prioridade de 4 *bits*. A activação da saída CH significa que alguma das entradas foi activada e só neste caso é que as saídas P1 e P0 fazem sentido e contêm o código da entrada activada.



codificador de prioridade de 8 entradas



codificador de prioridade de 9 entradas

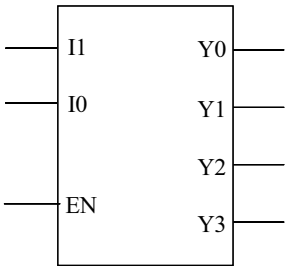
I7	I6	I5	I4	I3	I2	I1	I0	EI	A2	A1	A0	GS	EO
x	x	x	x	x	x	x	x	1	1	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1	1	1	0
0	x	x	x	x	x	x	x	0	0	0	0	0	1
1	0	x	x	x	x	x	x	0	0	0	1	0	1
1	1	0	x	x	x	x	x	0	0	1	0	0	1
1	1	1	0	x	x	x	x	0	0	1	1	0	1
1	1	1	1	0	x	x	x	0	1	0	0	0	1
1	1	1	1	1	0	x	x	0	1	0	1	0	1
1	1	1	1	1	1	0	x	0	1	1	0	0	1
1	1	1	1	1	1	1	0	0	1	1	1	0	0

I9	I8	I7	I6	I5	I4	I3	I2	I1	D	C	B	A
1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	0	x	1	1	0	1
1	1	1	1	1	1	0	x	x	1	1	0	0
1	1	1	1	1	0	x	x	x	1	0	1	1
1	1	1	1	0	x	x	x	x	1	0	1	0
1	1	1	0	x	x	x	x	x	1	0	0	1
1	1	0	x	x	x	x	x	x	1	0	0	0
1	0	x	x	x	x	x	x	x	0	1	1	1
0	x	x	x	x	x	x	x	x	0	1	1	0

Figura 5.16: Codificadores de prioridade em circuitos integrados da série: ’148 e ’147.

5.3.3 Descodificadores binários

Um descodificador binário faz a função inversa de um codificador: dado um código de *N bits* que represente um valor *i*, produz um código *y* com 2^N *bits* em que só o *bit i* está activo. A figura 5.17 mostra a tabela de verdade de um descodificador de 2 *bits* para 4 *bits*, com uma entrada EN de activação global (em Inglês *enable*), que permite "ligar" ou "desligar" a realização da função pelo circuito. Este tipo de entrada é comum em circuitos deste tipo já que muitas vezes facilita a sua integração em sistemas mais complexos, como por exemplo descodificadores de um número maior de *bits*.



EN	I1	I0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Y3 = EN.I1.I0

Y2 = EN.I1.I0'

Y1 = EN.I1'.I0

Y0 = EN.I1'.I0'

Figura 5.17: Descodificador binário de 2 *bits* para 4 *bits*.

A figura 5.18 mostra como se pode construir um descodificador de 3 para 8 *bits* com uma entrada de *enable* usando 2 descodificadores de 2 para 4 *bits* e alguns circuitos adicionais. Note que o mesmo processo pode ser aplicado sucessivamente para criar descodificadores com qualquer número de entradas.

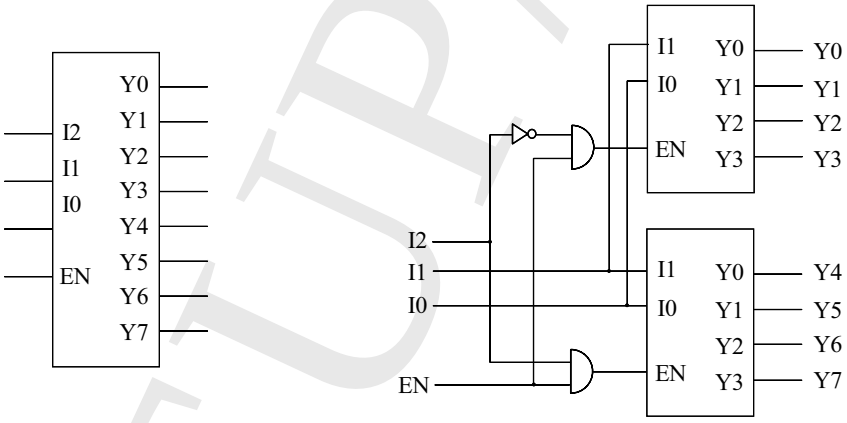


Figura 5.18: Descodificador de 3 para 8 *bits* realizado com dois descodificadores de 2 para 4 *bits*.

Descodificadores como geradores de termos mínimos

Descodificadores binários têm aplicação em variadas situações, e, juntamente com algumas portas lógicas adicionais podem ser usados para implementar funções lógicas. Por exemplo, o

descodificador de 2 para 4 mostrado na figura 5.17 pode ser entendido como um gerador dos *minterms* (termos de produto) de uma função de duas variáveis, ligadas às entradas I1 e I0 do descodificador. Assim, para implementar qualquer função lógica de 2 variáveis com base neste descodificador, basta ligar as saídas Yi correspondentes aos *minterms* i da função às entradas de uma porta lógica OR. Por exemplo, para realizar a função $F(X,Y)$ representada pela sua lista de termos mínimos:

$$F(X,Y) = \sum_{XY}(0,2,3)$$

basta ligar as saídas Y0, Y2 e Y3 do descodificador às entradas de uma porta lógica OR, realizando assim a função na forma soma de produtos. A realização dual, produto de somas, poderia ser facilmente construída ligando apenas um inversor à saída Y1 do descodificador (figura 5.19).

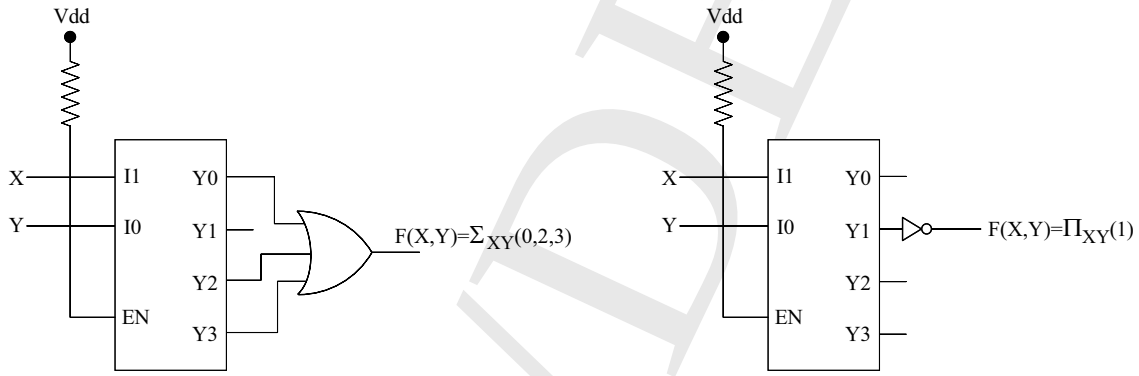


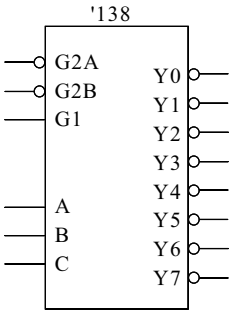
Figura 5.19: Realização de uma função lógica de 2 variáveis com base num descodificador.

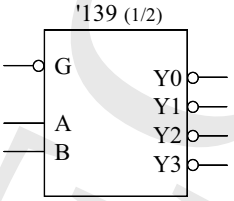
Os descodificadores ’138 ’139

A função descodificador existe disponível nos circuitos integrados ’138 e ’139 da série 74. Na figura 5.20 apresenta-se o símbolo lógico destes circuitos e a tabela que descreve o seu funcionamento. A disponibilidade das 3 entradas de activação no 74x138 (G1, G2A e G2B) facilita a sua expansão em descodificadores mais complexos seguindo o processo que foi mostrado na figura 5.18, mas sem ser necessário recorrer à utilização de circuitos lógicos adicionais.

5.3.4 Descodificadores para mostradores de 7 segmentos

Outra categoria de descodificadores são os descodificadores para mostradores de 7 segmentos, tal como o que foi referido atrás na secção 5.3.1. Um descodificador BCD para 7 segmentos realiza a tradução de um código de 4 *bits* representando dígitos decimais (código BCD) para o código de 7 *bits*que faz acender o dígito correspondente num mostrador de 7 segmentos. Um circuito integrado da série 74 que realiza esta função é o ’49 cujo símbolo e tabela de verdade





C	B	A	G1	G2A	G2B	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
x	x	x	0	x	x	1	1	1	1	1	1	1	1
x	x	x	x	1	x	1	1	1	1	1	1	1	1
x	x	x	x	x	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	0	1	0	1	1	1	1	1	1
0	1	0	1	0	0	1	1	0	1	1	1	1	1
0	1	1	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1	0	1	1
1	1	0	1	0	0	1	1	1	1	1	1	0	1
1	1	1	1	0	0	1	1	1	1	1	1	1	0

B	A	G	Y0	Y1	Y2	Y3
x	x	1	1	1	1	1
0	0	0	0	1	1	1
0	1	0	1	0	1	1
1	0	0	1	1	0	1
1	1	0	1	1	1	0

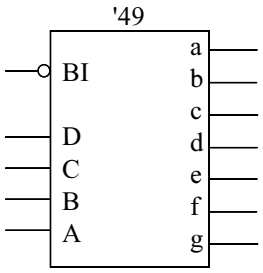
Figura 5.20: Descodificadores ’138 e ’139. Note que o circuito integrado ’139 possui dois descodificadores de 2 para 4 *bits*.

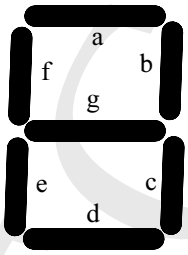
se apresenta na figura 5.21. Este circuito tem uma entrada BI (*blanking input*, activa no nível lógico baixo) que é comum existir neste tipo de circuitos e que permite apagar o mostrador independentemente do valor presente nas suas entradas. Além disso, ligando e desligando rapidamente este sinal com uma temporização adequada é possível modificar a intensidade aparente da luz emitida pelos LEDs.

Existem outras versões deste circuito com as referências ’46, ’47, e ’48 que também possuem sinais de controlo destinados a apagar os zeros à esquerda quando se constroem bancos reunindo vários mostradores deste tipo. Além disso, têm também uma entrada adicional (LT–*lamp test*) que permite ligar todos os segmentos independentemente do valor presente nas restantes entradas do descodificador.

Descodificadores hexadecimal para 7 segmentos

Um descodificador BCD para 7 segmentos destina-se a ser usado apenas para entradas iguais a códigos BCD legais. Se na entrada ocorrer um código não BCD entre 1010_2 e 1111_2 então a saída poderá ser qualquer porque os valores das funções lógicas foram considerados como indiferentes para permitir minimizar o circuito lógico. No entanto existem também descodifi-



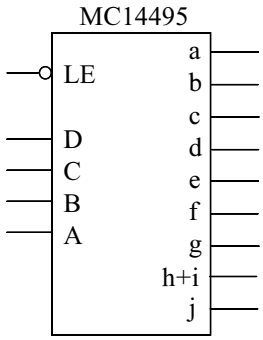


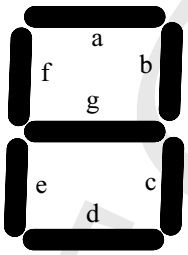
D	C	B	A	BI	a	b	c	d	e	f	g
x	x	x	x	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	0
0	0	0	1	1	0	1	1	0	0	0	0
0	0	1	0	1	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	1	0	0	1
0	1	0	0	1	0	1	1	0	0	1	1
0	1	0	1	1	1	0	1	1	0	1	1
0	1	1	0	1	0	0	1	1	1	1	1
0	1	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	0	1	1
1	0	1	0	1	0	0	0	1	1	0	1
1	0	1	1	1	0	0	1	1	0	0	1
1	1	0	0	1	0	1	0	0	0	1	1
1	1	0	1	1	1	0	0	1	0	1	1
1	1	1	0	1	0	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0	0	0	0

Figura 5.21: Descodificador BCD para 7 segmentos: ’49.

cadores deste tipo que, para além dos 10 dígitos decimais, também traduzem os códigos entre 1010_2 e 1111_2 em símbolos que representam os 6 dígitos hexadecimais de A a F. Um exemplo é o circuito integrado MC14495 mostrado na figura 5.22[6]. Este descodificador possui duas saídas que não existem nos descodificadores de 7 segmentos apresentados anteriormente: a saída h+i é activada quando na entrada está presente um código superior 9 (1001_2) e a saída j é desactivada quando na entrada está presente o código 1111_2. Além disso, este circuito tem uma entrada /LE (em latch enable) que controla elementos de memória internos ao circuito e permite memorizar o valor presente na entrada: quando /LE é zero a saída apresenta o resultado da descodificação do valor presente na entrada, mas quando /LE passa de 0 para 1 é memorizado o valor que existia nas entradas nesse instante, que se mantém até que /LE seja novamente 0.

[6]Este circuito pertence à família lógica CMOS (*Complementary Metal-Oxide Semiconductor*) que é diferente da dos circuitos da série 74x referidos antes. Apesar de ser também um circuito digital que realiza a função lógica referida, apresenta características eléctricas bastante diferentes das dos circuitos integrados da série 74, sendo mesmo incompatível com algumas das famílias dos circuitos dessa série.





D	C	B	A	LE	a	b	c	d	e	f	g	h+i	j	símbolo
x	x	x	x	l	último valor									
0	0	0	0	0	1	1	1	1	1	1	0	0	1	0
0	0	0	1	0	0	1	1	0	0	0	0	0	1	1
0	0	1	0	0	1	1	0	1	1	0	1	0	1	2
0	0	1	1	0	1	1	1	1	0	0	1	0	1	3
0	1	0	0	0	0	1	1	0	0	1	1	0	1	4
0	1	0	1	0	1	0	1	1	0	1	1	0	1	5
0	1	1	0	0	0	0	1	1	1	1	1	0	1	6
0	1	1	1	0	1	1	1	0	0	0	0	0	1	7
1	0	0	0	0	1	1	1	1	1	1	1	0	1	8
1	0	0	1	0	1	1	1	0	0	1	1	0	1	9
1	0	1	0	0	1	1	1	0	1	1	1	1	1	A
1	0	1	1	0	0	0	1	1	1	1	1	1	1	b
1	1	0	0	0	1	0	0	1	1	1	0	1	1	C
1	1	0	1	0	0	1	1	1	1	0	1	1	1	d
1	1	1	0	0	1	0	0	1	1	1	1	1	1	E
1	1	1	1	0	1	0	0	0	1	1	1	1	0	F

Figura 5.22: Descodificador hexadecimal para 7 segmentos MC14495. Note que a disposição dos LEDs do mostrador obriga a representar os símbolos A, C, E e F como caracteres maiúsculos e os símbolos b e d como caracteres minúsculos.

Mostradores de 7 segmentos

Existem várias outras versões deste tipo de descodificador cujas saídas são activas com o nível lógico alto ou com o nível lógico baixo e apresentando diferentes características eléctricas que devem ser compatíveis com o tipo de mostrador a utilizar. Para além de dimensões, formas e cores variadas, existem dois tipos de ligação eléctrica normalmente disponíveis neste tipo de dispositivo: ânodo comum e cátodo comum. No primeiro caso os ânodos (terminais positivos) dos LEDs estão ligados entre si e no outro são os cátodos (terminais negativos) que estão ligados entre si[7].

Para ligar um descodificador de 7 segmentos a um mostrador de 7 segmentos deve-se es-

[7]Para não confundir os termos ânodo e cátodo basta lembrar que no tubo de raios *catódicos* dos nossos televisores e monitores são "disparados" electrões que têm carga eléctrica negativa.

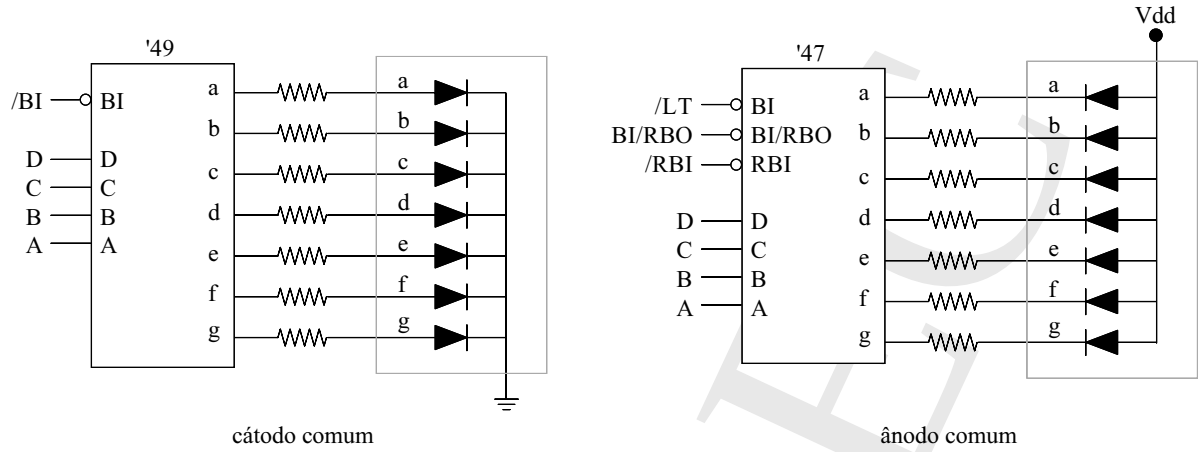


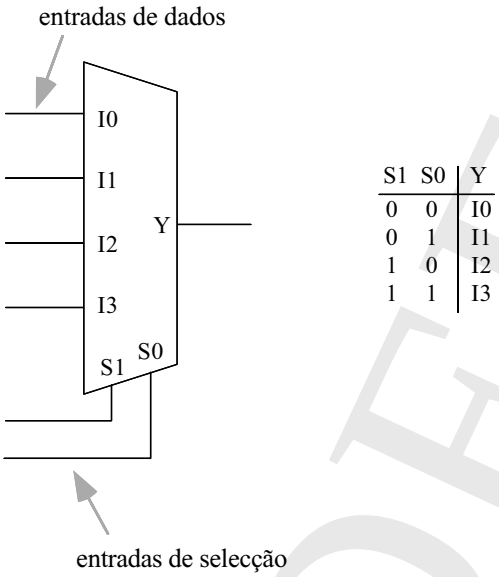
Figura 5.23: Mostradores de 7 segmentos com ânodo comum e cátodo comum, e a sua ligação às saídas de descodificadores. Note que o circuito integrado MC14495 que tem saídas activas no nível lógico alto, possui já internamente as resistências mostradas na figura.

colher um mostrador de cátodo comum se o descodificador tiver as saídas activas com o nível lógico alto, e um mostrador de ânodo comum quando o descodificador tem as saídas activas com o nível lógico baixo. Por exemplo, os circuitos integrados ’46 e ’47 têm saídas activas no nível lógico baixo e por isso só podem ser usados com mostradores de ânodo comum; por outro lado os ’48 e ’49 têm saídas activas no nível lógico alto e ligam-se a mostradores de cátodo comum. Para além de realizar a função lógica de descodificação, as saídas do descodificador devem ser capazes de fornecer (ou absorver) a corrente eléctrica necessária para acender os LEDs, cujos valores típicos se situam entre 3 e 15 mA. Na figura 5.23 mostra-se o esquema eléctrico da ligação de mostradores de ânodo comum e de cátodo comum a descodificadores de 7 segmentos. As resistências eléctricas colocadas em série com os LEDs destinam-se a limitar a intensidade de corrente que passa por eles. Valor típicos situam-se entre 220Ω e 560Ω, mas esse valor deve ser devidamente calculado por forma a garantir a intensidade de corrente correcta nos LEDs.

5.4 Selectores (ou *multiplexadores*)

A função de selector (ou multiplexador, também às vezes abreviado para mux) foi já ilustrada no exemplo mostrado no início deste capítulo. De uma forma geral, um multiplexador é um circuito que tem N entradas de *selecção* S_j, 2^N entradas de *dados* designadas por I_i e uma única saída de dados Y. O valor lógico apresentado na saída é igual ao que está presente na entrada I_k, em que k é o valor colocado nas entradas de selecção S_j. Um multiplexador é caracterizado pelo número de entradas de dados ou pelo número de linhas de selecção (por exemplo, um

multiplexador com 2 linhas de selecção tem 4 entradas de dados), e é geralmente representado pelo símbolo mostrado na figura 5.24.



S1	S0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Figura 5.24: Multiplexador com 2 entradas de selecção e 4 entradas de dados: símbolo lógico e tabela de verdade.

5.4.1 Multiplexadores como geradores de funções

De uma forma geral, um multiplexador tem aplicação sempre que é necessário usar um conjunto de sinais lógicos para escolher entre duas ou mais entradas. Para além disso um multiplexador pode ser usado para implementar funções lógicas recorrendo, eventualmente, a um conjunto reduzido de elementos adicionais.

A realização de uma função lógica de N variáveis com um multiplexador com N entradas de selecção é trivial e não requer qualquer componente lógico adicional: basta ligar as variáveis da função às entradas de selecção do multiplexador e ligar as entradas de dados I_i às constantes 0 ou 1, consoante o valor que a função assume na linha i da tabela de verdade (figura 5.25). Uma solução mais económica pode ser obtida recorrendo a um multiplexador com apenas $N-1$ entradas de selecção e um inversor, como se exemplifica na figura 5.26. Note que a necessidade de um inversor depende do padrão de uns e zeros que constitui a tabela de verdade da função. Como este depende da ordem pela qual as variáveis são representadas (e ligadas às entradas de selecção do multiplexador), podem existir soluções que não necessitem da negação da variável menos significativa, e por isso menos complexas.

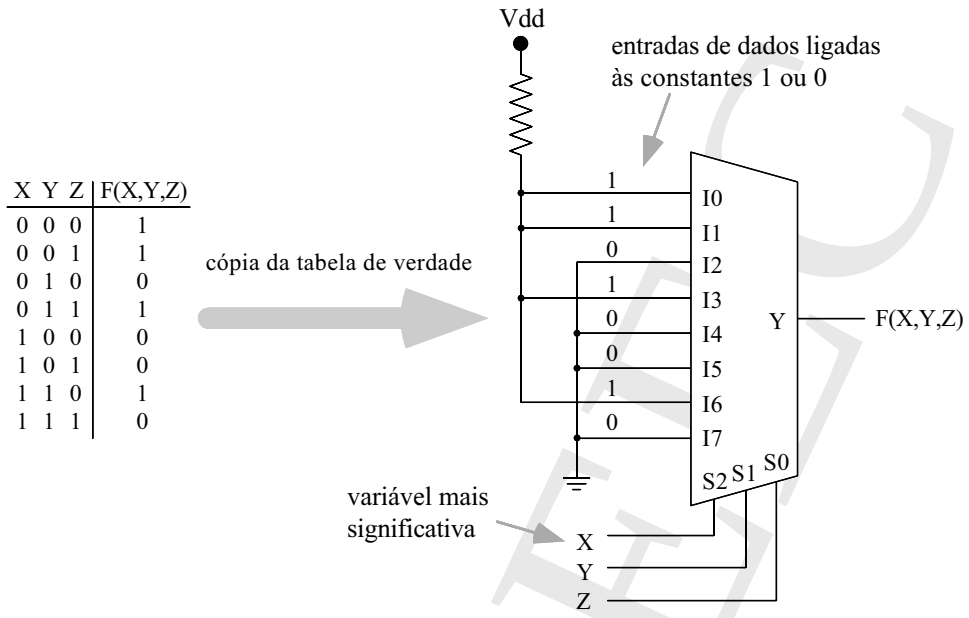


Figura 5.25: Implementação de uma função lógica de 3 variáveis usando um multiplexador com 3 entradas de selecção.

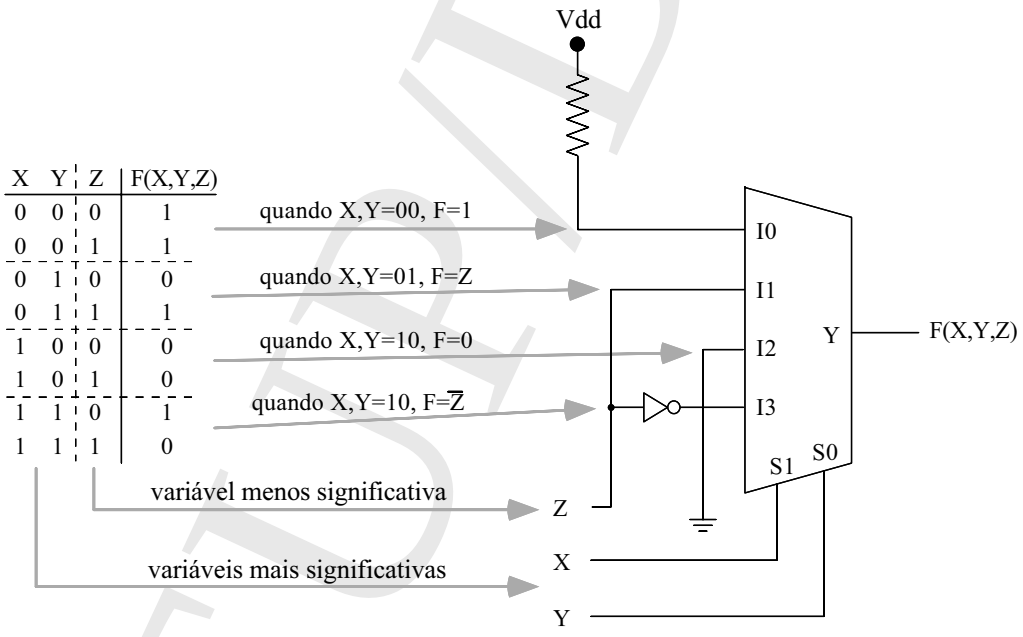


Figura 5.26: Implementação de uma função lógica com 3 variáveis usando um multiplexador com apenas 2 entradas de selecção.

5.4.2 Multiplexadores em circuitos da série 74

A função multiplexador existe disponível em circuitos integrados da série 74 com variadas configurações. A figura 5.27 mostra o símbolo lógico de 4 dispositivos com esta função: ’150

(mux 16-1), ’151 (mux 8-1), ’153 (2 mux 4-1 com entrada de selecção comum) e ’157 (4 mux 2-1 com entrada de selecção comum). Estes circuitos também dispõem de entradas de activação que, entre outras aplicações, facilitam a construção de multiplexadores mais complexos.

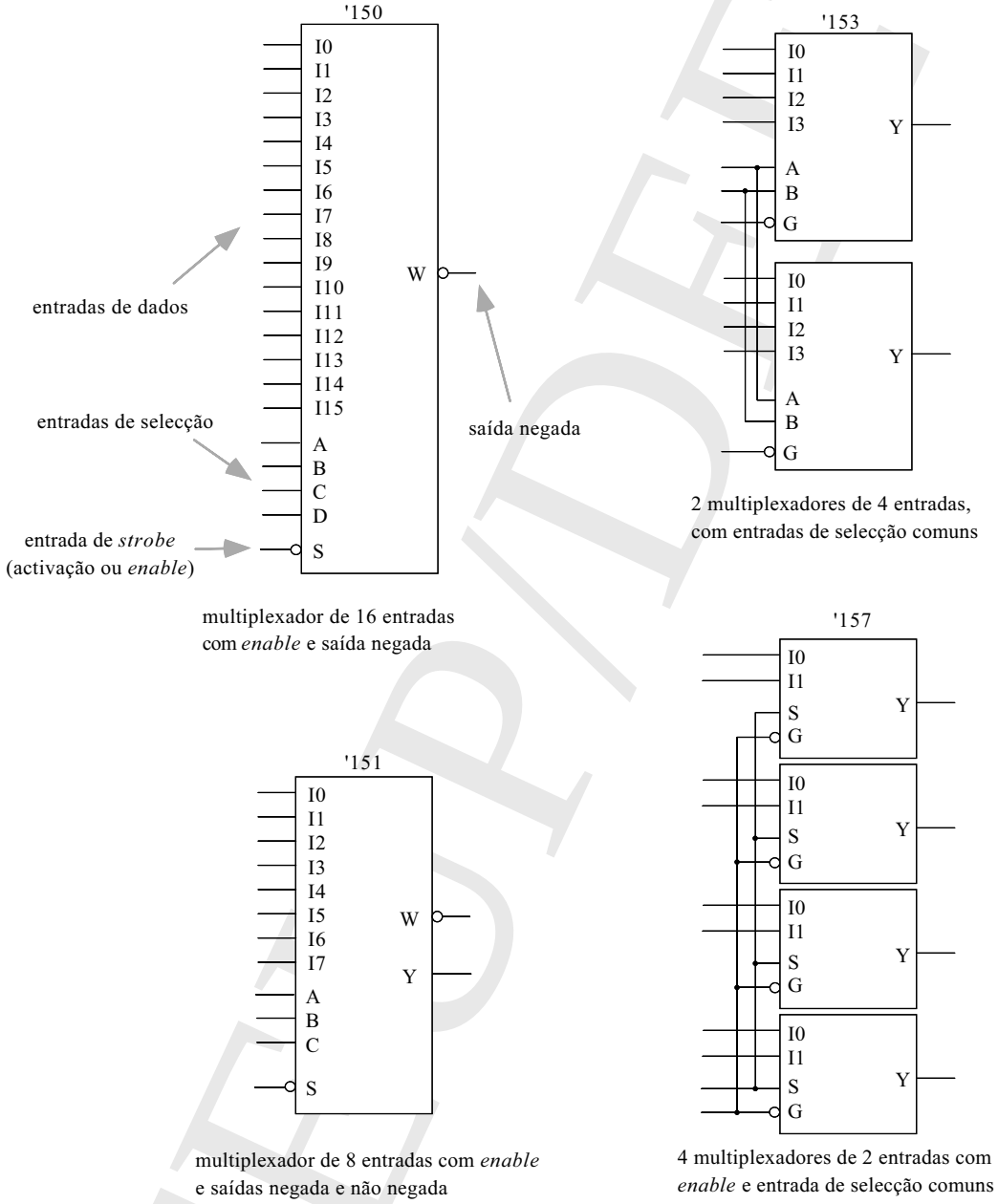


Figura 5.27: Circuitos integrados da série 74 com a função multiplexador.

5.5 Funções aritméticas

Uma categoria de funções padrão com inúmeras aplicações permitem realizar operações aritméticas entre dois ou mais operandos. Embora também se possam enquadrar nesta categoria circuitos muito complexos que realizem operações com números representados em vírgula flutuante ou calculam funções transcendentes (funções trignométricas, logaritmos, etc.), iremo-nos restringir às operações aritméticas elementares com números inteiros, e em particular às adições e subtracções. Na realidade, os multiplicadores, divisores e todas as outras funções mais complexas são construídas com base somadores ou circuitos derivados de somadores.

Nas secções seguintes serão estudadas outras funções aritméticas e mostrados alguns exemplos de circuitos integrados da série 74x que integram essas funções.

5.5.1 Comparadores

O comparador mais simples que se pode construir permite determinar se um número binário formado por um conjunto de *N bits*é ou não igual a uma constante conhecida, representada no mesmo número de *bits*[8]. O circuito que realiza esta função não é mais do que uma porta lógica do tipo E com *N* entradas, em que são negadas as entradas ligadas aos *bits* que se pretendem comparar com zero (figura 5.28).

Figura 5.28: Uma porta lógica E como comparador de igualdade com as constantes 01110010_2 e 11011001_2.

Com o que já sabemos podemos facilmente construir um circuito que realize a comparação de igualdade com, por exemplo, 4 constantes diferentes escolhidas por um número de 2 *bits*: para isso basta usar um multiplexador de 4 entradas (com 2 entradas de selecção) ligado às saídas dos 4 comparadores, permitindo assim escolher um dos 4 resultados de comparação (figura 5.29).

Para realizar a comparação de igualdade entre dois operandos com *N bits* já é necessário um circuito um pouco mais complexo do que o anterior. Podemos dizer que dois números binários *A* e *B* são iguais se forem iguais os *bits* de *A* e de *B* em posições correspondentes

[8]Embora se fale em comparação de números, naturalmente que um comparador não "sabe" o que representam os *bits* que estão a ser comparados, o que depende do contexto em que o circuito é aplicado.

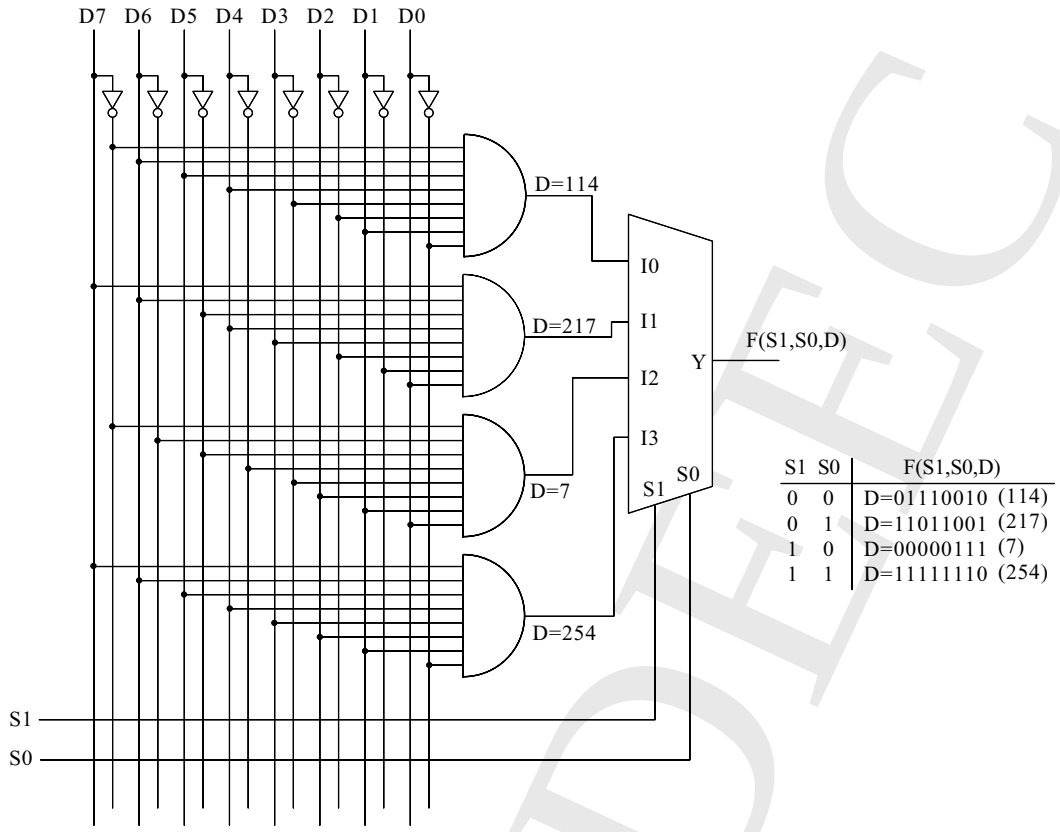


Figura 5.29: Um comparador com 4 constantes diferentes usando um multiplexador.

($A_0 = B_0, A_1 = B_1, ..., A_{N-1} = B_{N-1}$). O circuito que compara dois *bits* entre si é realizado pela função lógica XOR apresentada na figura 5.7, página 90, com um inversor na sua saída. Podemos assim construir um comparador de igualdade com N portas XOR, um inversor na saída de cada porta XOR e uma porta lógica E com N entradas (5.30).

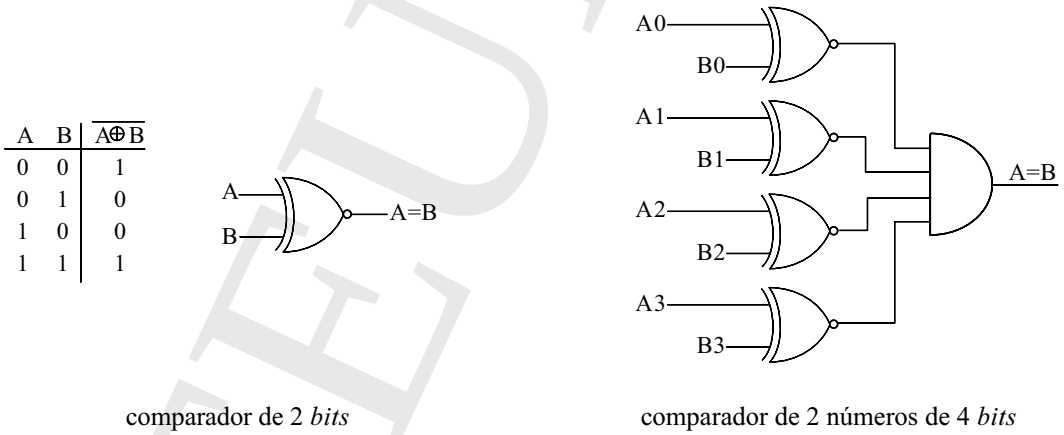


Figura 5.30: Uma porta lógica XNOR (XOR com a saída negada) como um comparador de igualdade de 2 *bits* e um comparador de igualdade entre dois números com 4 *bits*.

Comparador de magnitude

No exemplo apresentado no início do capítulo foi construído um comparador de magnitude para operandos representado números com sinal em complemento para 2, baseado num circuito subtractor. De que forma é necessário modificar este circuito se os operandos representarem agora números inteiros positivos? O processo para determinar se é verdade que $A < B$ pode ser o mesmo: efectua-se a subtracção $A - B$ e avalia-se o sinal do resultado. No entanto, agora não faz sentido falar de sinal do resultado porque estamos a operar com números que *não podem* (ou melhor, não "sabem" como podem) ser negativos. Apesar disso, a operação de subtracção $A - B$ pode ser feita como a adição de A com o complemento para 2 de B, ignorando o transporte que é produzido quando se adiciona o *bit* mais significativo. Na verdade, se o resultado puder ser correctamente representado como um número sem signal, então esse transporte é igual a 1 e significa que é verdade que $A >= B$; por outro lado, se o resultado for um número negativo e não puder ser representado como tal, então esse transporte será zero e significa que é $A < B$ (note que nesta operação, este *bit* de transporte não é mais do que a negação do *bit* de *borrow* gerado quando se aplica o algoritmo da subtracção binária).

Podemos assim concluir que um comparador de magnitude para operandos positivos é constituído por um subtractor do qual apenas se necessita do *bit* de transporte mais significativo. Pela mesma razão que vimos antes, todos os circuitos lógicos que não intervêm na geração desse *bit* podem simplesmente ser removidos do circuito do subtractor. O circuito lógico deste comparador é muito semelhante ao do comparador de magnitude para números com sinal e é apresentado na figura 5.31.

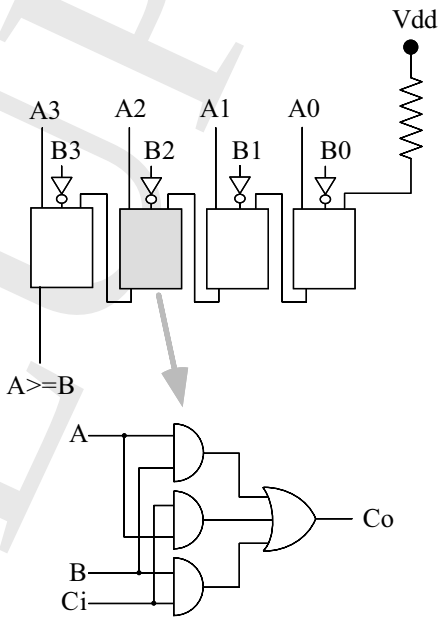


Figura 5.31: Comparador de magnitude para números sem sinal.

Uma abordagem diferente para construir um comparador de magnitude para números sem sinal consiste em aplicar um raciocínio semelhante ao que foi seguido antes para obter o circuito lógico para o somador. O processo consiste em traduzir para circuitos lógicos o processo que efectuamos "à mão" quando pretendemos resolver o problema em papel. Dados, por exemplo, os números $A = 100101$ e $B = 101100$, começamos por comparar os seus *bits* mais significativos: se forem iguais então pode-se concluir que até agora é $A = B$; passando ao *bit* seguinte, é necessário comparar não só esses *bits* entre si, mas também integrar o resultado das comparações feitas anteriormente. Neste exemplo, tínhamos concluído que era $A = B$, então juntando o resultado da comparação dos segundos *bits*, continuamos a concluir que, com 2 *bits* analisados continua a ser verdade que é $A = B$. Prosseguindo para o terceiro *bit* podemos concluir imediatamente que é $A < B$ sem ser necessário olhar para os restantes *bits* à direita.

Com estas conclusões podemos enunciar as regras a seguir na análise de cada par de *bits* A_i e B_i dos números A e B a comparar:

1. Se pela análise dos *bits* anteriores já se concluiu que era $A > B$ então o valor dos *bits* em análise não modificará essa condição.

2. De forma semelhante, se pela análise dos *bits* anteriores já se concluiu que era $A < B$ então continua a ser verdade que é $A < B$ independentemente do valor dos *bits* em análise;

3. Se o resultado da análise dos *bits* anteriores concluiu que é $A = B$, então se é $A_i = 1$ e $B_i = 0$ conclui-se que $A > B$, se $A_i = 0$ e $B_i = 1$ então é $A < B$ e, finalmente, se $A_i = B_i$ então continua a ser verdade que $A = B$.

Esta análise permite-nos construir um circuito comparador para números com N *bits* à custa da interligação em cascata de N circuitos idênticos que implementam as regras apresentadas acima. A este tipo de circuito chamam-se circuitos iterativos porque o resultado final é produzido ao longo de várias iterações realizadas por um conjunto de blocos que recebem recebem informação dos precedentes, processam parte dos operandos e passam o resultado ao seguinte. Note que o resultado final da comparação só pode ser obtido depois de integrar o resultado da comparação dos *bits* menos significativos dos operandos. As figuras 5.32 e 5.33 mostram a implementação deste circuito.

Comparadores em circuitos integrados da série 74

A função de comparação existe disponível nos circuitos integrados '85 e '682. O primeiro é um comparador de magnitude e de igualdade para números de 4 *bits*, dispondo de um conjunto de entradas e saídas que facilitam a construção de comparadores entre números de qualquer

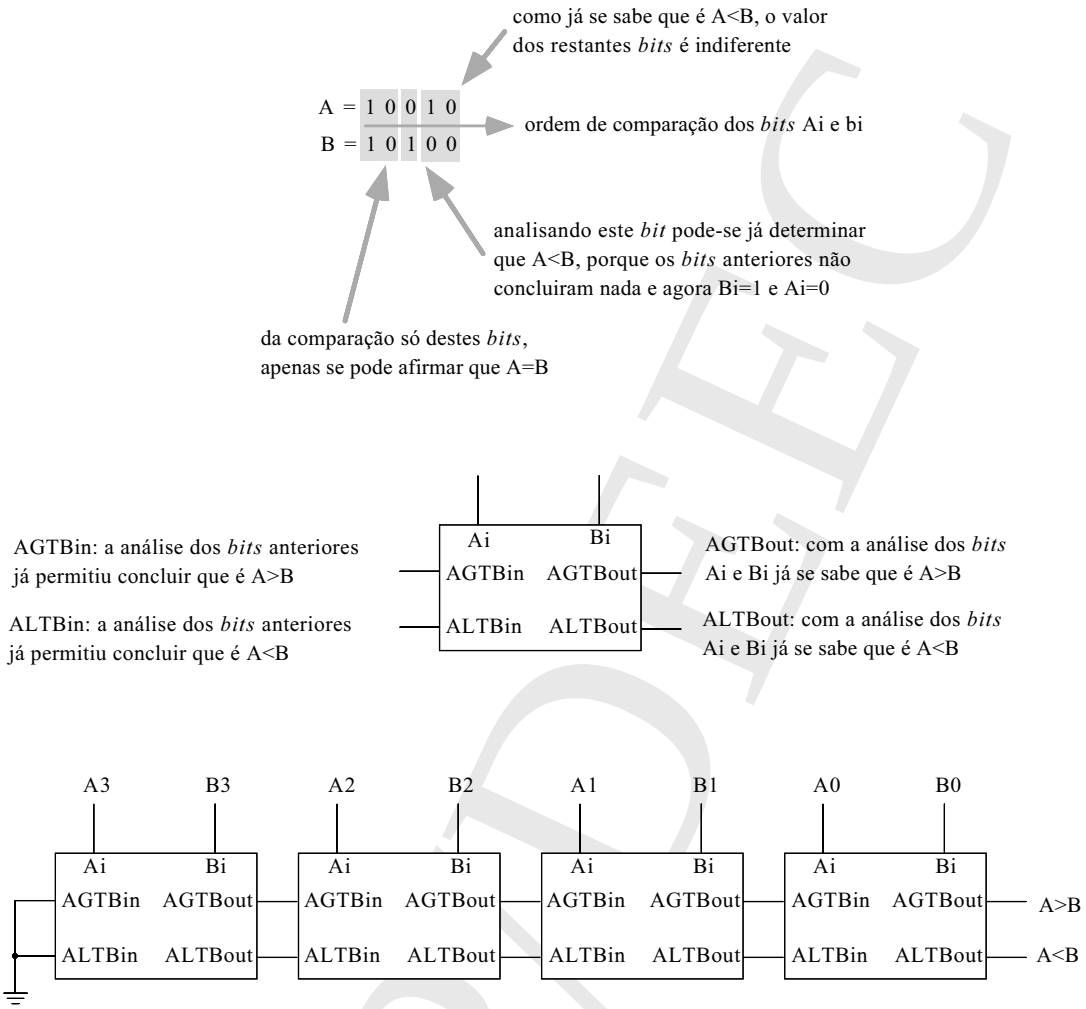


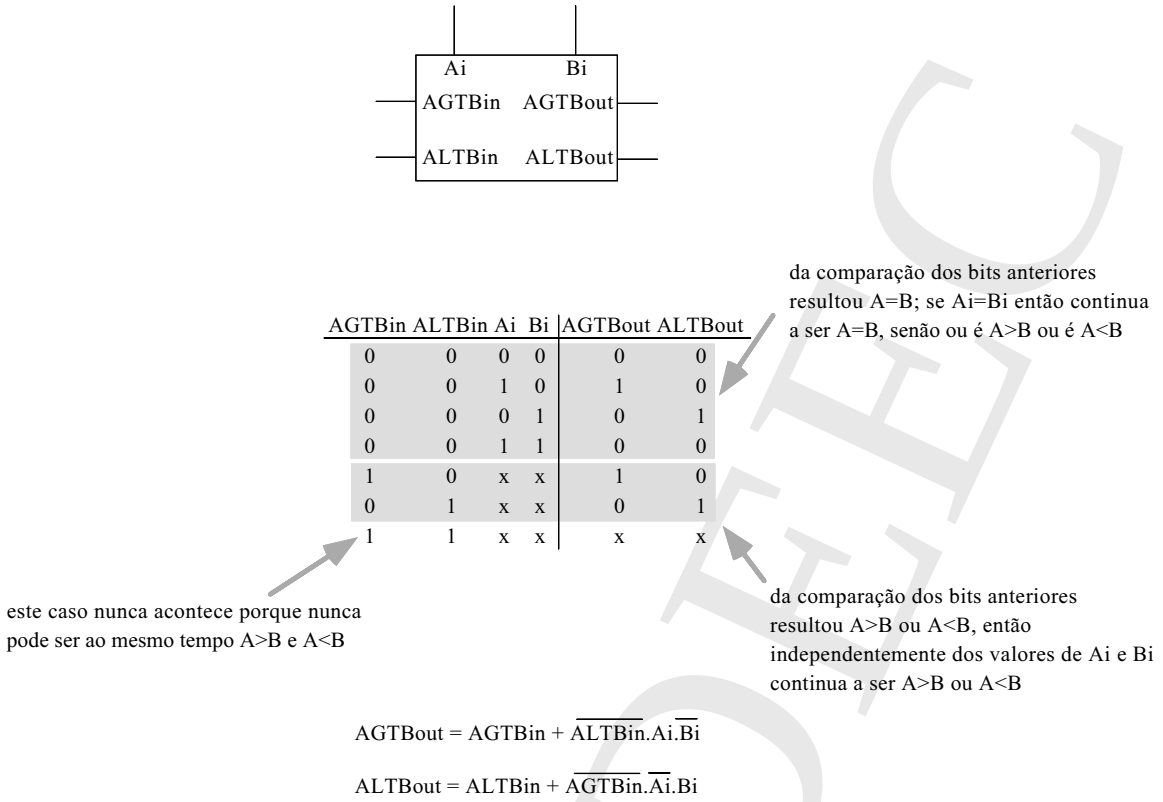
Figura 5.32: Comparador de magnitude para números sem sinal, implementado como um circuito iterativo.

tamanho (desde que o número de *bits* seja múltiplo de 4). O ’682 é um comparador de magnitude para números de 8 *bits*.

5.5.2 Somadores e subtractores

No exemplo que serviu de introdução a este capítulo já foi apresentado um circuito lógico que efectua a adição binária e a sua modificação para realizar a subtracção binária. Embora existam várias outras formas de construir circuitos lógicos que realizam estas operações, esta estrutura em cascata, designada por *ripple carry* tem a vantagem de ser a menos complexa, a mais regular e a que mais facilmente se expande para qualquer número de *bits*[9].

[9]Mas também tem defeitos! Embora as questões relacionadas com a rapidez de funcionamento de circuitos digitais sejam só abordadas no capítulo 6 pode-se já afirmar que este tipo de somador é o que demora mais tempo



AGTBin	ALTBin	Ai	Bi	AGTBout	ALTBout
0	0	0	0	0	0
0	0	1	0	1	0
0	0	0	1	0	1
0	0	1	1	0	0
1	0	x	x	1	0
0	1	x	x	0	1
1	1	x	x	x	x

Figura 5.33: Implementação do bloco comparador usado no circuito da figura 5.32

Circuito somador ou subtractor

Como um somador e um subtractor diferem entre si em muito pouco, torna-se interessante conseguir reunir mum mesmo circuito as duas funções. Como foi já visto, para efectuar a subtracção basta complementar todos os *bits* do diminuidor (o segundo operando) e fazer igual a 1 o transporte que entra no somador do *bit* menos significativo. Atendendo à tabela de verdade da função OU-exclusivo, pode-se concluir que uma porta lógico XOR também pode ser usada como um inversor condicional: sendo $Z = X \oplus Y$, então se $X = 0$ Z é igual a Y e se $X = 1$ Z é igual ao oposto de Y. Assim, podem ser usadas portas lógicas XOR para inverter condicionalmente todos os *bits* do diminuidor, comandadas por um sinal que escolhe a operação subtracção e que também coloca o transporte de entrada do *bit* menos significativo igual a 1 (figura 5.34).

Detecção de *overflow*

No capítulo 2 foi apresentada a forma como é possível detectar a ocorrência de *overflow* na realização da adição binária. A forma mais simples de implementar esta funcionalidade consiste em comparar os 2 últimos transportes produzidos na adição: se forem iguais o resultado

a produzir o resultado.

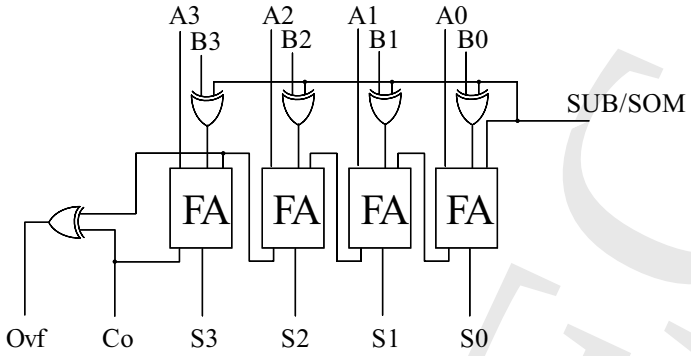


Figura 5.34: Somador/subtractor iterativo (ou *ripple carry*). Note que a detecção de ocorrência de *overflow* em adições com operandos em complemento para 2 pode ser obtida usando apenas uma porta XOR adicional (saída Ovf).

está correcto e se forem diferentes o resultado está errado porque ocorreu *overflow*. Esta função é realizada com apenas uma porta lógica XOR, actuando aqui como um comparador de desigualdade entre dois *bits*.

Somadores em circuitos da série 74

Existem vários circuitos integrados desta série com a função aritmética adição. O ’82 é um somador de 2 números de 2 *bits*, com entrada e saída de transporte e o ’83 (ou ’283) realiza a adição de números de 4 *bits*, tendo igualmente entrada e saída de transporte. A disponibilidade da entrada e saída de transporte possibilitam que possam ser usados para criar somadores para operandos com maior dimensão. Os dois últimos (’83 e ’283) implementam um somador com um circuito lógico diferente do que foi estudado e que permite obter resultados em tempo mais curto. Finalmente, o ’183 contém 2 somadores completos (*full-adders*) independentes, tendo sido concebido com o objectivo de criar circuitos optimizados para a adição de vários operandos, que são necessários para construir multiplicadores.

5.5.3 Outras funções aritméticas

Além dos circuitos somadores, subtractores e comparadores já estudados, outra função importante é a multiplicação, que é realizada à custa da associação de somadores ou de subtractores. A operação de divisão é significativamente mais complexa e será apenas estudado um circuito baseado em multiplexadores que permite efectuar divisões por potências inteiras de 2 (2^N).

Multiplicadores

Recordando o que foi estudado sobre a multiplicação binária (ver secção 2.4 na página 27), podemos concluir que, para obter o resultado do produto de 2 números *A* e *B* con *N bits* é necessário realizar duas operações distintas:

1. obter os produtos de cada *bit* do multiplicador pelo multiplicando;
2. adicionar os produtos anteriores, alinhando cada parcela com a posição do *bit* do multiplicador que lhe deu origem.

A primeira operação é muito fácil de realizar porque, como os *bits* do multiplicador ou são 1 ou são 0, então o produto desse valor pelo multiplicando ou dá o próprio multiplicando ou dá zero. Para construir um circuito que realize essa operação basta usar *N* portas lógicas do tipo E em que em cada uma, uma das entradas liga a um *bit* do multiplicando e a outra liga ao *bit* do multiplicador. A segunda operação consiste em adicionar as *N* parcelas, o que pode ser feito com $N-1$ somadores de *N bits* cada. A figura 5.35 mostra o circuito lógico de um multiplicador de 4 *bits*.

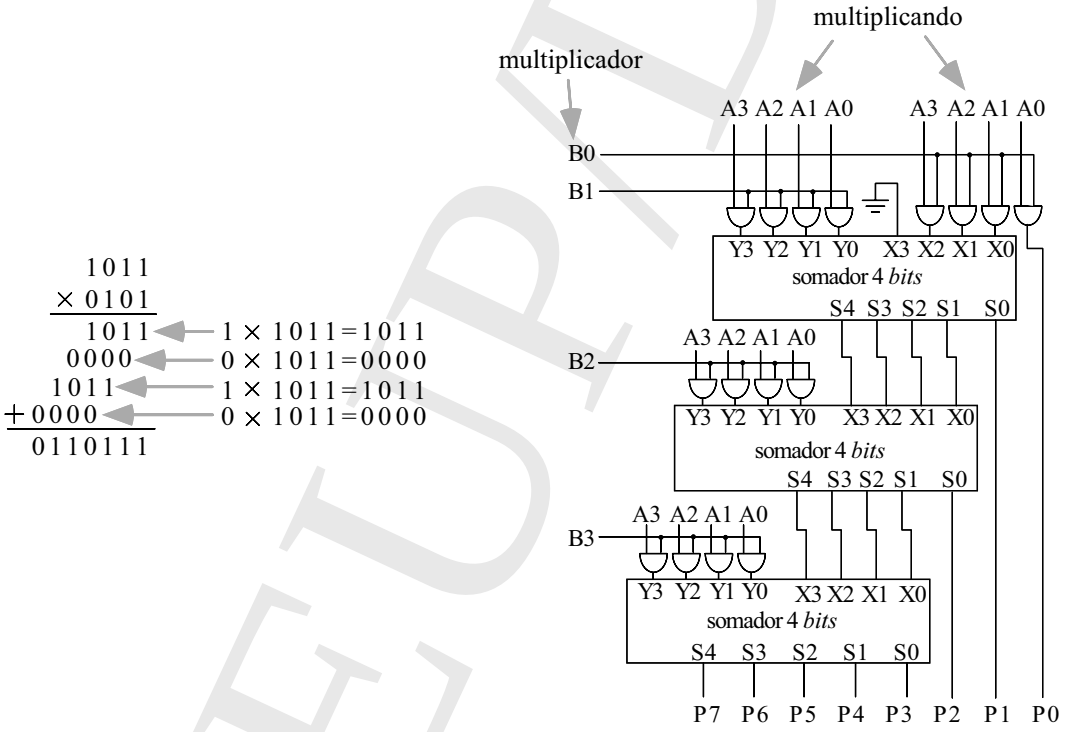


Figura 5.35: Um multiplicador entre 2 números de 4 *bits*.

Embora o circuito mostrado na figura 5.35 apresenta a estrutura mais fácil de perceber, existem várias outras formas de organizar os elementos somadores que conduzem a circuitos mais rápidos e mais fáceis de expandir para qualquer dimensão dos operandos.

O multiplicador apresentado apenas funciona para operandos representado números inteiros sem sinal. A realização do produtos entre número com sinal representados em complemento para 2 é um pouco mais complexa. Pode-se resumir o processo em duas alterações em relação ao que foi visto antes: em primeiro lugar, cada parcela resultante do produto de um *bit* do multiplicador pelo multiplicando deve ser devidamente extendida para o mesmo número de *bits* que irá ter o resultado; em segundo lugar, deverá ser subtraída em vez de adicionada a última parcela que resulta do produto do *bit* mais significativo do multiplicador pelo multiplicando. Esta última operação resulta do facto de que, em complemento para 2, o *bit* mais significativo tem um peso negativo no valor do número (ver secção 2.6.3 na página 36).

O circuito integrado 74x274 implementa a operação de multiplicação mostrada na figura 5.35, produzindo um resultado de 8 *bits* sem sinal. Como exercício, tente construir um multiplicador para 2 números de 8 *bits* usando 4 multiplicadores deste tipo, somadores e circuitos lógicos adicionais.

Multiplicadores por constantes

Se um dos operandos for conhecido, por exemplo o multiplicador, o circuito mostrado na figura 5.35 pode ser simplificado, removendo os somadores que efectuam adições com zero e que não são necessários. Por exemplo, o produto de um número de 4 *bits* pela constante 6 (em binário 0110) apenas necessita de 1 somador de 6 *bits* para adicionar o multiplicando deslocado 1 *bit* para a esquerda com ele deslocado 2 *bits* para a esquerda (figura 5.36). Note também que se o multiplicador for uma constante igual a uma potência inteira de 2, então a sua representação binária tem um único *bit* igual a 1 e não é necessário qualquer somador para obter o resultado.

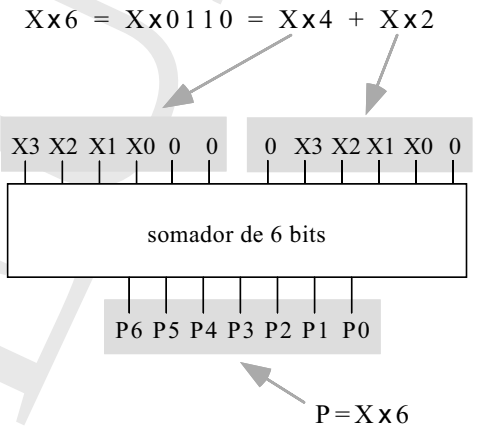


Figura 5.36: Um multiplicador entre um número de 4 *bits* e a constante 6 (0110_2).

Multiplicadores e divisores por 2^N

Como foi estudado no capítulo 2, dividir ou multiplicar um número binário por 2^N (com N inteiro) equivale a deslocar os seus *bits* de N posições para a direita ou esquerda, respectivamente. Essa operação não requer qualquer tipo de recurso lógico e consiste apenas em seleccionar os *bits* apropriados do operando para obter o resultado pretendido. Como as divisões por potências inteiras de 2 correspondem a deslocar os *bits* para a direita, é necessário ter em atenção se o operando representa um número sem signal ou com signal, e nesse caso se a convenção de representação é signal ou grandeza ou complemento para 2. Na figura 5.37 mostra-se como se podem obter facilmente os resultados de algumas multiplicações e divisões por constantes deste tipo, admitindo que o operando representa um número com signal em complemento para 2.

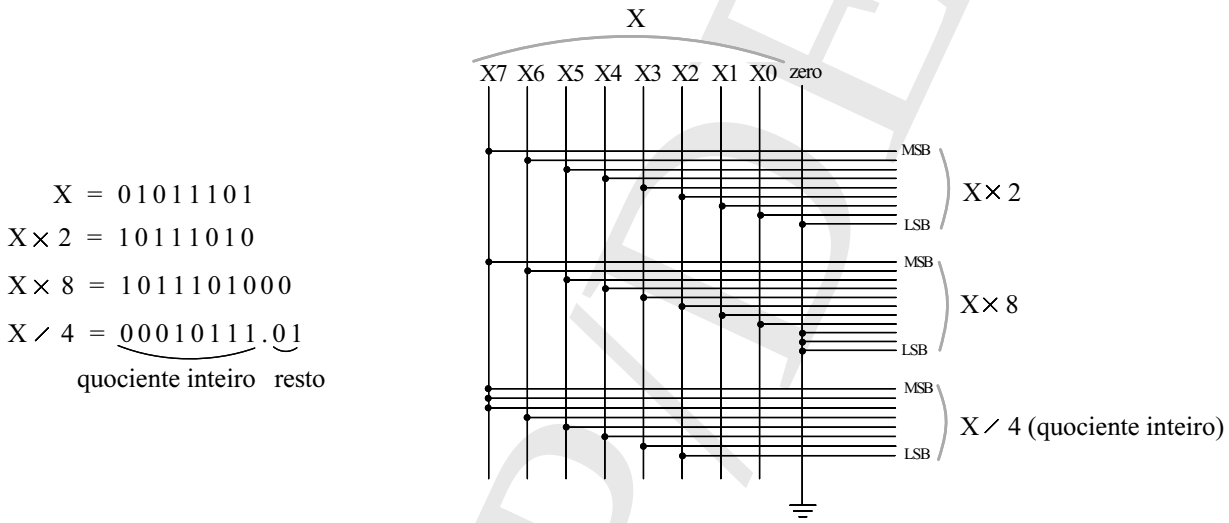


Figura 5.37: Multiplicações e divisões por constantes iguais a potências inteiras de 2 são realizadas apenas à custa da selecção apropriada dos *bits* do operando. Note que o número de *bits* dos produtos é maior do que a dimensão do operando X, para garantir que o resultado pode ser correctamente representado. Na divisão $X/4$ é assumido que o operando X representa valores com signal em complemento para 2, sendo apenas seleccionado o quociente inteiro (o resto é formado pelos 2 *bits* menos significativos do operando X).

O cálculo do produto ou do quociente de um número por uma potência inteira de 2 desconhecida pode ser realizado de forma semelhante à mostrada na figura 5.37. Recorrendo apenas a multiplexadores pode-se construir facilmente um circuito que realiza essa operação e que é normalmente conhecido por *barrel shifter*. Na figura 5.38 mostra-se um circuito deste tipo para operandos de 8 *bits* com signal em complemento para 2, que produz os resultados $X \times 2^N$ para qualquer N entre 0 e 7. Note que para que o resultado possa ser sempre representado correctamente é necessário representá-lo com 15 *bits*.

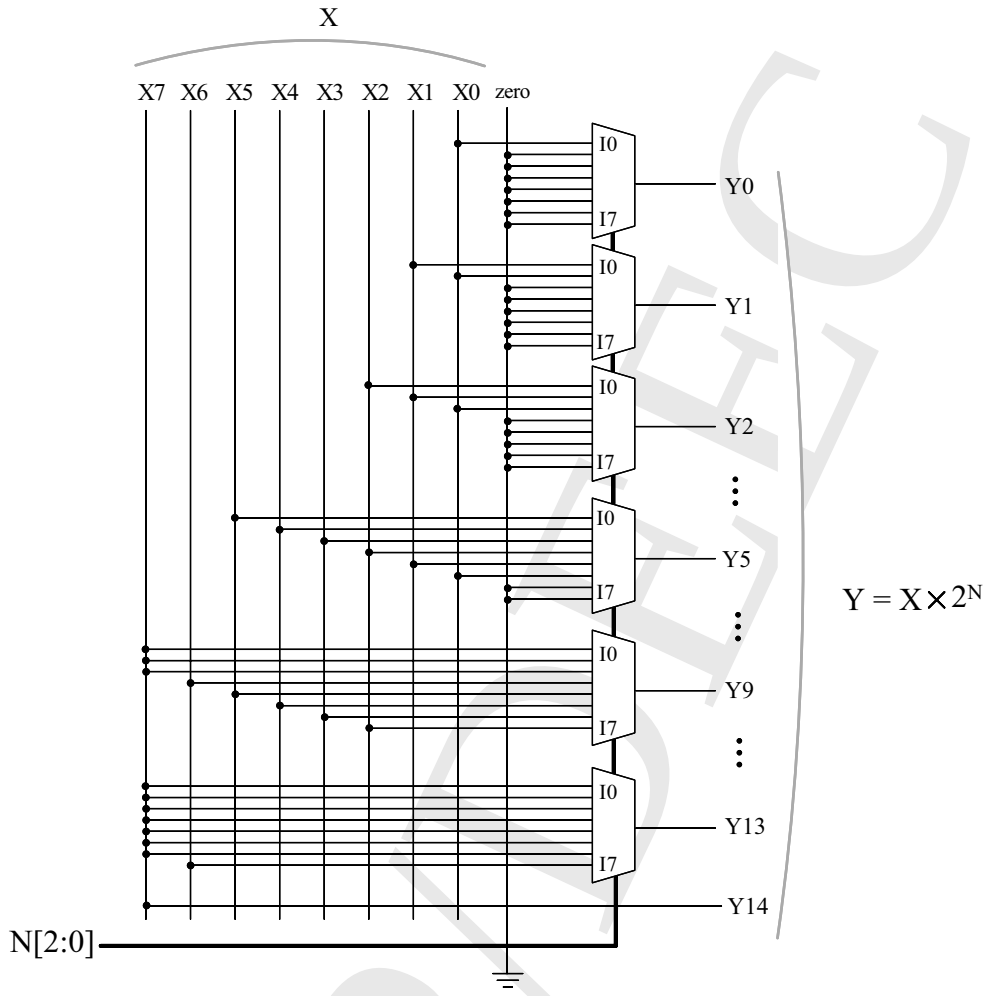


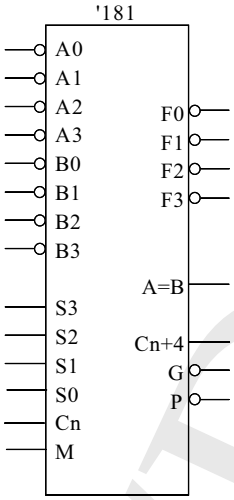
Figura 5.38: Um *barrel shifter* realizado com multiplexadores. Este circuito permite obter os resultados $X \times 2^N$, para qualquer N entre 0 e 7, considerando que o operando X é um valor com sinal em complemento para 2. O circuito pode ser facilmente expandido para qualquer dimensão do operando X e também para realizar divisões do tipo $X/2^N$.

Unidades lógicas e aritméticas—ALUs

Tal como vimos ao longo deste capítulo, as funções aritméticas adição, subtracção e multiplicação partilham muitos elementos comuns, tais como *full-adders*, portas lógicas AND, OR ou XOR[10]. A solução trivial para obter um circuito que permita efectuar diversas operações aritméticas e lógicas, consiste em dispor em paralelo de tantos circuitos quantas as funções a realizar (por exemplo, somador, subtractor, multiplicador, inversores, etc), e usar um multiplexador para escolher um dos vários resultados produzidos. No entanto, como foi já visto para a função somador/subtractor, é mais económico ”fundir”num mesmo circuito lógico as várias funções que

[10]Embora não tenha sido abordada a implementação lógica de divisores, também esta função é construída com base em somadores e subtractores

se pretendem obter. A este tipo de circuito chama-se geralmente unidade lógica e aritmética (é comum usar o acrónimo ALU do Inglês *Arithmetic and Logic Unit*), e como o nome o indica permite efectuar diversas combinações de operações aritméticas e lógicas, que são escolhidas por um conjunto de entradas de selecção. Na figura 5.39 mostra-se o símbolo e a tabela funcional da ALU de 4 *bits* '181. As saídas que não aparecem referidas na tabela permitem acrescentar funcionalidades adicionais às mostradas, como por exemplo a função de comparação de igualdade ou a expansão em ALUs de operandos com mais de 4 *bits*.



S3 S2 S1 S0	M=1 (funções lógicas)	M=0 (funções aritméticas) Cn=0	Cn=1
0 0 0 0	Fi=/Ai	F=A menos 1	F=A
0 0 0 1	Fi=/(Ai.Bi)	F=A.B menos 1	F=A.B
0 0 1 0	Fi=/Ai+Bi	F=A./B menos 1	F=A./B
0 0 1 1	Fi=1	F= -1	F= zero
0 1 0 0	Fi=/(Ai+Bi)	F=A mais (A+/B)	F=A mais (A+/B) mais 1
0 1 0 1	Fi=/Bi	F=A.B mais (A+/B)	F=A.B mais (A+/B) mais 1
0 1 1 0	Fi=/(Ai⊕Bi)	F=A menos B menos 1	F=A menos B
0 1 1 1	Fi=Ai+/Bi	F=A+/B	F=A+/B mais 1
1 0 0 0	Fi=/Ai.Bi	F=A mais (A+B)	F=A mais (A+B) mais 1
1 0 0 1	Fi=Ai⊕Bi	F=A mais B	F=A mais B mais 1
1 0 1 0	Fi=Bi	F=A./B mais (A+B)	F=A./B mais (A+B) mais 1
1 0 1 1	Fi=Ai+Bi	F=A+B	F=A+B mais 1
1 1 0 0	Fi=0	F=A+A	F=A+A mais 1
1 1 0 1	Fi=Ai./Bi	F=A.B mais A	F=A.B mais A mais 1
1 1 1 0	Fi=Ai.Bi	F=A./B mais A	F=A./B mais A mais 1
1 1 1 1	Fi=Ai	F=A	F=A mais 1

Figura 5.39: Uma unidade lógica e aritmética (ALU)—'181.