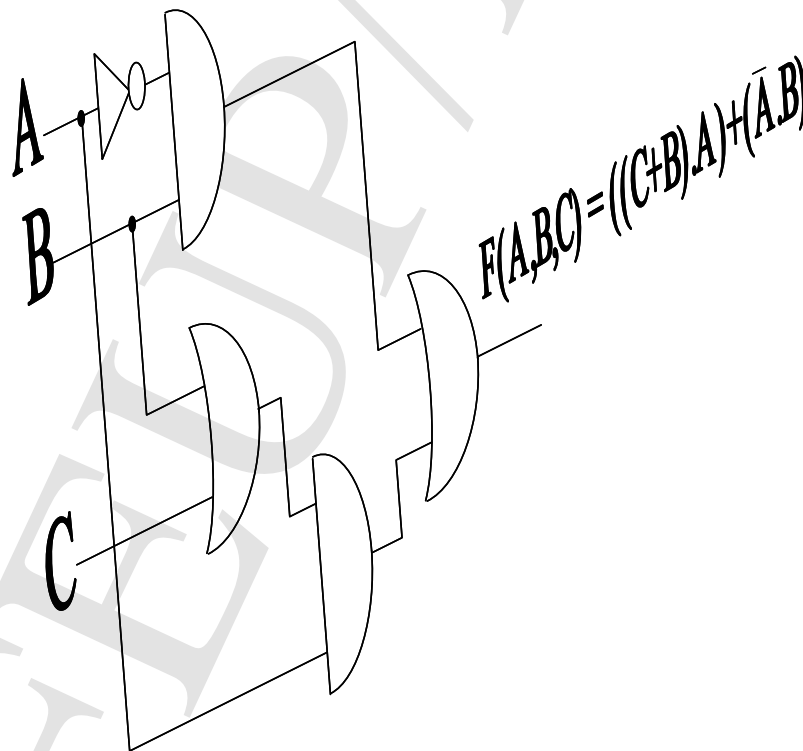


SISTEMAS DIGITAIS



© José Carlos Alves, FEUP 2002/2006

Este texto foi produzido para apoio às aulas teóricas da disciplina Sistemas Digitais do 1º ano da Licenciatura em Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto. A sua utilização fora do âmbito desta disciplina carece de autorização expressa do autor. É autorizada a cópia parcial ou integral deste texto exclusivamente para uso pessoal, contando que se mantenham todos os elementos identificadores da sua origem, incluindo a marca de água e o rodapé presente em cada página.

Conteúdo

1	O que são sistemas digitais?	6
2	Como se representa informação em binário	13
2.1	Introdução	13
2.2	Representando números	19
2.2.1	Números fraccionários	20
2.2.2	O sistema binário	21
2.2.3	Os sistemas octal e hexadecimal	21
2.2.4	Como se representa um número inteiro em base 2?	23
2.2.5	Como se representa um número fraccionário em base 2?	24
2.2.6	Números com parte inteira e parte fraccionária	25
2.3	Adição e subtracção binária	25
2.4	Multiplicação e divisão binária	27
2.5	Dimensão dos resultados e <i>overflow</i>	28
2.6	Representação de números negativos	30
2.6.1	Sinal e grandeza	30
2.6.2	Complemento para a base	31
2.6.3	Complemento para dois	36
2.7	Representação binária de números decimais (<i>BCD</i>)	40
3	Álgebra de Boole	43
3.1	Axiomas e teoremas	44
3.1.1	Axiomas	45
3.1.2	Teoremas	46
3.2	Representação de funções booleanas	48
4	Projecto de circuitos combinacionais	55
4.1	Optimizar o projecto	56

4.1.1	Tecnologias de implementação	57
4.2	Minimização de funções representadas em formas padrão	62
4.2.1	Mapas de Karnaugh	63
4.2.2	Minimização de expressões soma de produtos	64
4.2.3	Minimização de expressões na forma produto de somas	68
4.2.4	Minimização de funções com termos indiferentes	71
4.3	Circuitos lógicos	73
5	Funções combinacionais padrão	78
5.1	Regras para desenho de circuitos	81
5.1.1	Representação de barramentos	82
5.1.2	Nomes dos sinais	83
5.1.3	Entradas e saídas	83
5.1.4	Entradas e saídas negadas	84
5.2	Um exemplo: calculador do máximo e do mínimo	85
5.3	Codificadores e decodificadores	93
5.3.1	Codificador binário	93
5.3.2	Codificadores de prioridade	96
5.3.3	Decodificadores binários	97
5.3.4	Decodificadores para mostradores de 7 segmentos	100
5.4	Selectores (ou <i>multiplexadores</i>)	104
5.4.1	Multiplexadores como geradores de funções	104
5.4.2	Multiplexadores em circuitos da série 74	105
5.5	Funções aritméticas	106
5.5.1	Comparadores	108
5.5.2	Somadores e subtratores	112
5.5.3	Outras funções aritméticas	115
6	Circuitos sequenciais	121
6.1	Circuitos sequenciais elementares	126
6.1.1	Circuitos astáveis	126
6.1.2	Circuitos biestáveis	127
6.2	Síntese de circuitos sequenciais síncronos	140
6.2.1	Especificação	141
6.2.2	Circuito de mínimo custo	151
6.3	Modelo de máquinas de estado de Mealy	157
6.3.1	Diagramas de transição de estados para máquinas de Mealy	160

6.3.2	O detector de sequência projectado como uma máquina de Mealy	163
6.4	Codificação de estados	164
6.4.1	Formas de codificação	167
6.5	Minimização de estados	171
7	Funções sequenciais padrão	173
7.0.1	Registos	174
7.1	Contadores	176
7.1.1	Contador em cascata (assíncrono)	177
7.1.2	Contador síncrono	179
7.1.3	Contador síncrono com carregamento paralelo	179
7.1.4	Algumas aplicações com contadores	180
7.1.5	Contador ascendente e descendente	186
7.2	Registos de deslocamento	186
7.2.1	Algumas aplicações com registos de deslocamento	190

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). Na área de aplicação que nos interessa, sistemas digitais electrónicos são circuitos eléctricos que processam informação que é representada por conjuntos de zeros e uns, correspondentes a certos 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, comandar as várias fases de funcionamento de uma máquina de lavar roupa ou realizar as operações complexas de um computador pessoal actual.

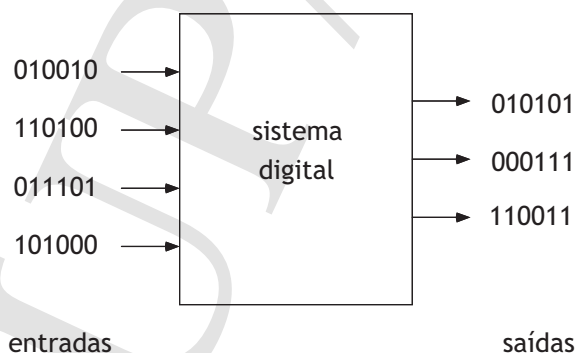


Figura 1.1: Modelo de um sistema digital.

De uma forma geral, podemos considerar qualquer sistema digital como uma caixa negra que tem *entradas* e *saídas* e que realiza uma *função* determinada (figura 1.1). As entradas recebem zeros e uns que 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 nível de temperatura especificado). O sistema digital processa a informação binária colocada nas entradas e apresenta nas suas saídas informação binária (zeros e uns) que correspondem, por exemplo, ao estado de uma lâmpada ou de um LED¹ (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 a funcionalidade desejada (por exemplo, lavar roupa...).

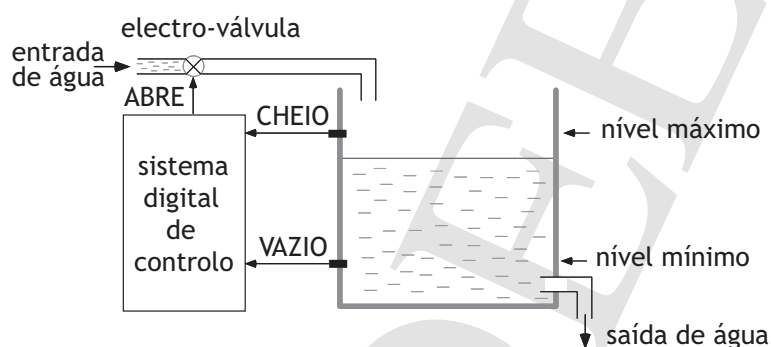


Figura 1.2: Controlo digital do nível de água num depósito.

Consideremos como exemplo um sistema automático para encher um depósito de água (figura 1.2). As entradas desse sistema de controlo são dois sinais eléctricos *digitais* provenientes de 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 passagem 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 passagem de 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 projectar um circuito electrónico com a função que acabamos de exemplificar.

Digital e não digital

Um sistema de controlo que realiza uma função semelhante ao descrito mas que não é digital, é a válvula de entrada de água de um depósito controlada por uma bóia: quando se descarrega o depósito a bóia desce abrindo a válvula que deixa entrar a água; à medida

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

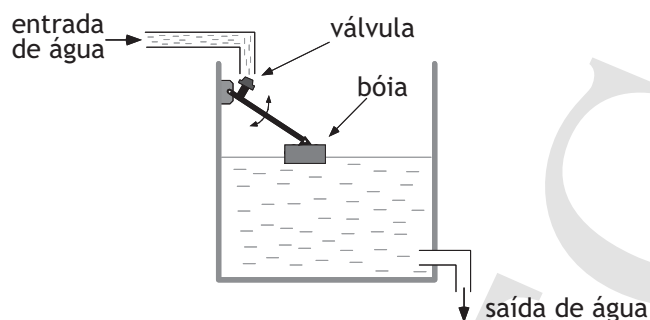


Figura 1.3: Um sistema *analógico* para controlar o nível de água num depósito.

que o depósito enche a bóia vai subindo, fechando progressivamente a entrada de água (figura 1.3). Ao contrário do sistema de controlo digital, em que a electro-válvula *digital* apenas pode estar no estado aberto ou no estado fechado, esta válvula *analógica* pode estar num número infinito de estados, desde toda aberta em que deixa passar o caudal máximo de água até totalmente fechada quando o nível de água atinge o máximo e a bóia fecha a saída de água.

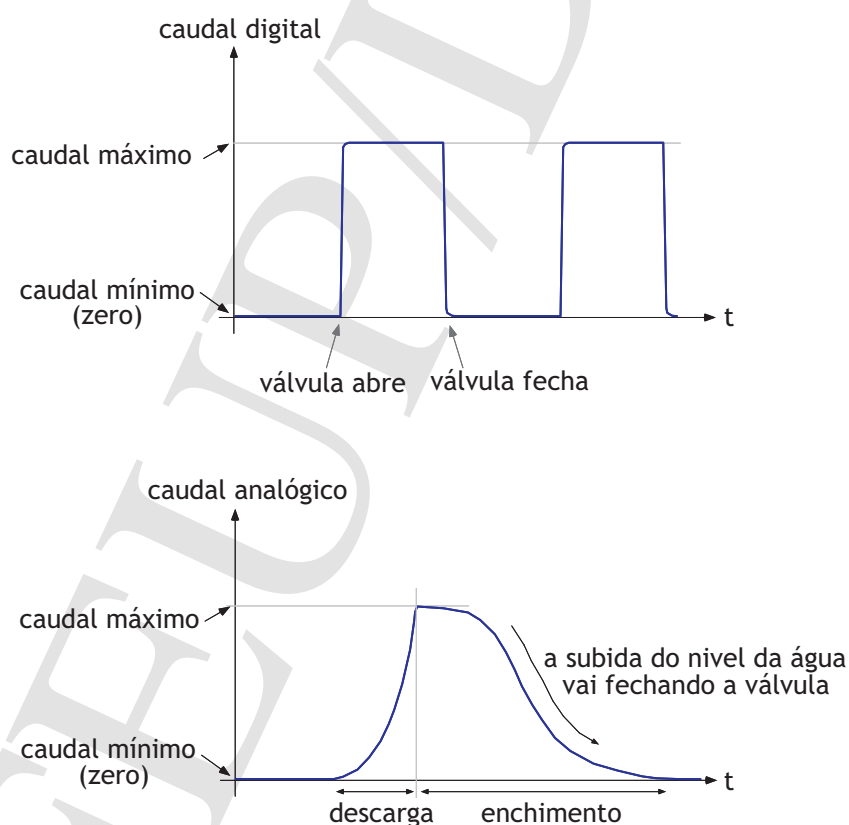


Figura 1.4: Variação temporal do caudal de água: digital vs. analógico.

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 mais “suave” ao longo do tempo (figura 1.4). Uma diferença importante entre grandezas analógicas e digitais é que enquanto que para as primeiras todos os valores que assumem são importantes para o funcionamento do sistema, em grandezas digitais só é importante, para o comportamento do sistema, saber em qual dos dois estados está. Ainda referindo-nos ao exemplo do sistema de controlo do nível de água no tanque, note que a relação entre o caudal de água que entra no tanque e o nível de água pode ser facilmente descrita com exactidão no sistema digital (como?), mas são necessárias expressões matemáticas complexas para fazer o mesmo no sistema analógico.

bits e mais bits...

Como uma grandeza binária (ou um *bit*²) só pode ter dois estados diferentes, não serve para muito, mas agrupando vários zeros e uns podemos aumentar o número de “coisas” diferentes que se podem representar. Por exemplo, com dois *bits* podemos formar 4 grupos (ou *códigos*) 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 com duas lâmpadas 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 nível 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 possível representar 2^N coisas diferentes. Por exemplo, existem apenas 256 combinações diferentes que se podem fazer com 8 *bits*, mas com 32 *bits* este número já 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 os uns e zeros são geralmente representados por tensões eléctricas *altas* e *baixas*: um 1 equivale a tensões acima de um

²do inglês *binary digit*

determinado limiar, e um zero a valores inferiores a um certo limite mínimo.

Imaginemos que o estado, aceso ou apagado, de uma vulgar lâmpada de incandescência igual às que temos em nossas casas é usado para representar um 1 ou um 0, respectivamente. A lâmpada não acende apenas quando a tensão da rede eléctrica atinge exactamente os 230V, 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 230V com uma vulgar pilha de 1.5V...) e a partir de uma tensão eléctrica suficientemente “alta” a lâmpada mantém-se no estado acesa. Naturalmente que para um certo intervalo de valores da tensão eléctrica poderemos considerar o estado da lâmpada como nem bem apagado nem bem aceso, ou seja é um estado *indefinido* em que não se pode representar correctamente um 1 nem um 0. Assim, se ocorrerem pequenas variações³ 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 apague e deixe de representar um 1 (figura 1.5).

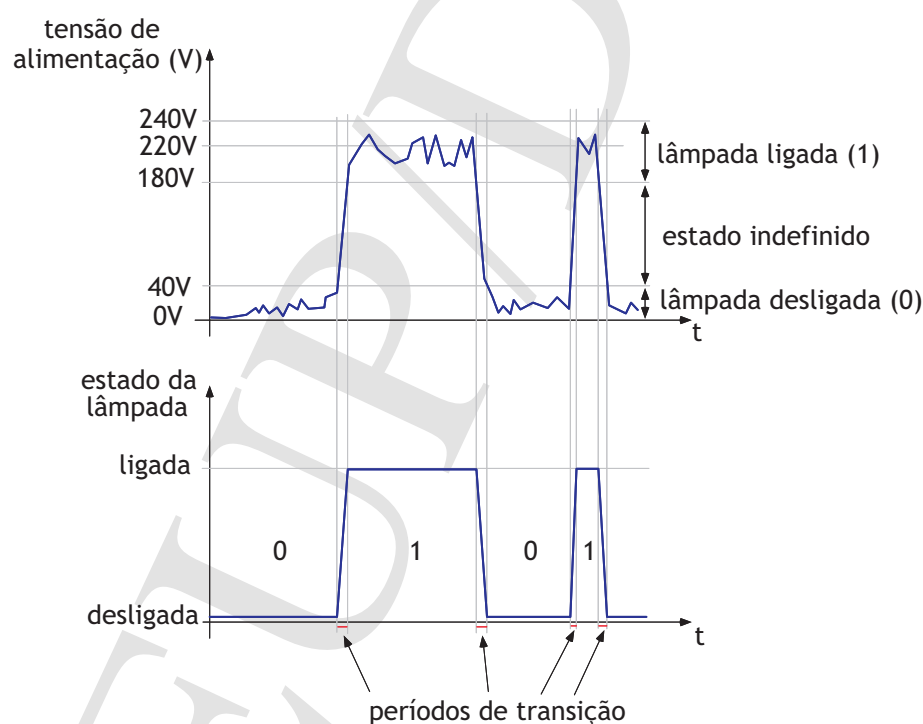


Figura 1.5: Os dois estados possíveis de uma lâmpada (ligada e desligada) em função do valor da tensão de alimentação

Um sistema electrónico digital também “entende” como zeros e uns tensões eléctricas

³neste caso de poucas dezenas de Volt.

altas e baixas que podem ter pequenas variações. Por exemplo, numa tecnologia corrente usada para o fabrico de circuitos digitais⁴ (CMOS—*Complementary Metal-Oxide Semiconductor*) funcionado com uma 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). Outras tecnologias de construção 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 zeros e uns e fazem com que um circuito digital não funcione correctamente (figura 1.6).

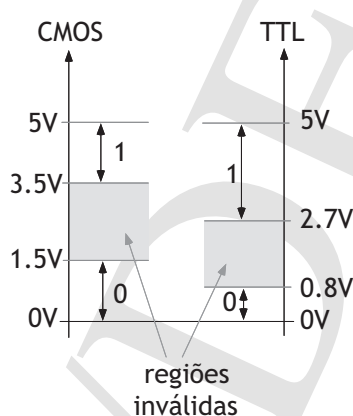


Figura 1.6: Tensões eléctricas e valores lógicos para circuitos digitais CMOS e TTL.

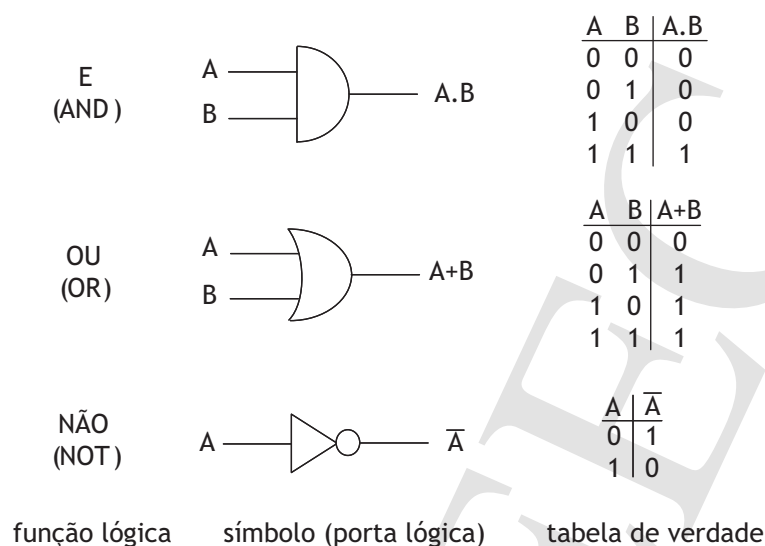
portas lógicas

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

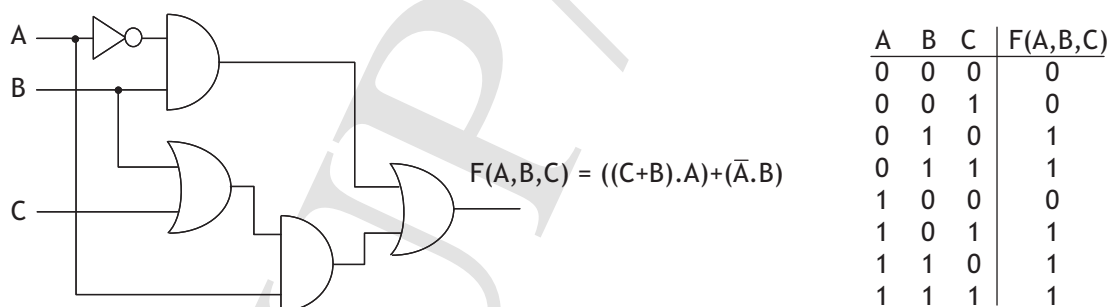
As funções lógicas elementares operam sobre valores lógicos 0 ou 1 e produzem um valor lógico também 0 ou 1 para cada uma das combinações possíveis dos seus operandos. A figura 1.7 mostra as tabelas que descrevem o comportamento dessas funções elementares e os símbolos gráficos geralmente utilizados para as representar no desenho de circuitos lógicos.

⁴Uma 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.

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

**Figura 1.7:** As 3 funções lógicas elementares

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 lógica mais complexa pode ser representada ligando entre si vários desses componentes elementares, como se exemplifica na figura 1.8.

**Figura 1.8:** Uma função lógica complexa construída com portas lógicas

Estas 3 funções elementares são a base em que assenta o funcionamento de qualquer sistema digital e constituem o conjunto básico de funções em que juntamente com um conjunto de regras permitem tratar matematicamente o comportamento de sistemas digitais (Álgebra de Boole a estudar no capítulo 3). No capítulo seguinte será estudada a forma como se pode representar diferentes tipos de informação usando zeros e uns, em particular valores numéricos e as operações aritméticas elementares.

Capítulo 2

Como se representa 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, textos, imagens, sons ou mesmo acções mecânicas (ligar um motor eléctrico, abrir um portão). 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*. Por exemplo, nos computadores actuais os dados são armazenados em quantidades elementares formadas por agrupamentos de 8 *bits*, a que se chama um *byte*. Um *byte* permite representar 256 “coisas” diferentes, correspondendo a todas as combinações possíveis de 8 zeros ou uns. Esta dimensão é conveniente porque permite representar todos os caracteres alfabéticos, numéricos e sinais de pontuação e além disso é uma potência inteira de 2. Actualmente a dimensão dos dispositivos de memória tais como discos, circuitos electrónicos ou CDROMs é expressa em número de bytes já que essa é uma medida aproximada do número de caracteres alfa-numéricos (alfabéticos e numéricos) que podem armazenar. Um *byte* pode também representar os números inteiros positivos entre 0 e 255 e agrupando vários *bytes* podem-se representar valores numéricos inteiros, fraccionários ou de vírgula flutuante em intervalos maiores. Veremos mais tarde como se podem representar números inteiros com o sistema de numeração binário 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 pontuação) num código de 8 *bits*.

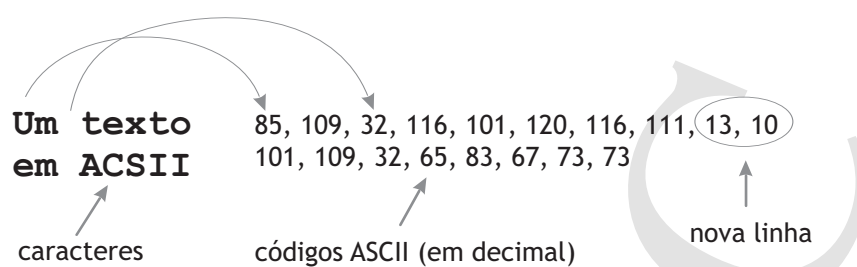
A codificação dos caracteres alfabéticos pelo padrão ASCII atribui a cada carácter um número inteiro. Por exemplo, as letras maiúsculas são representadas, por ordem alfabética, a partir do número 65 (A=65, B=66,...,Z=90), as letras minúsculas representam-se a partir do número 97 (a=97, b=98,..., z=122) e os algarismos correspondem aos números entre 48 (dígito zero) e 57 (dígito nove). A ordem “alfabética” que é entendida por um sistema digital (por exemplo um computador pessoal) que represente texto segundo esta norma não é mais do que a ordem natural dos números que representam os caracteres. Assim, num texto em ASCII a palavra “Zebra” é entendida como sendo alfabeticamente anterior à palavra “animal” e ambas são alfabeticamente superiores à palavra “0000”.

Na figura 2.1 mostra-se a sequência de números inteiros que correspondem aos *bytes* que codificam um pequeno texto com duas linhas e apresenta-se a tabela ASCII para codificação de texto.

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 se permite que cada ponto de imagem 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. Esta codificação é conveniente para imagens a preto-e-branco com 256 níveis de cinzento que apresentam já uma qualidade razoável. Com 32 *bits* por cada *pixel* podem-se representar 4.294.967.296 cores diferentes, o que para os nossos olhos é já 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. No arquivo que contém essa imagem digital, cada *pixel* é representado por um número inteiro que codifica a sua tonalidade entre completamente preto (0) e totalmente branco (255).



código	carácter	código	carácter	código	carácter	código	carácter
000	^@	032		064	@	096	`
001	^A	033	!	065	A	097	a
002	^B	034	"	066	B	098	b
003	^C	035	#	067	C	099	c
004	^D	036	\$	068	D	100	d
005	^E	037	%	069	E	101	e
006	^F	038	&	070	F	102	f
007	^G	039	'	071	G	103	g
008	^H	040	(072	H	104	h
009	^I	041)	073	I	105	i
010	^J	042	*	074	J	106	j
011	^K	043	+	075	K	107	k
012	^L	044	,	076	L	108	l
013	^M	045	-	077	M	109	m
014	^N	046	.	078	N	110	n
015	^O	047	/	079	O	111	o
016	^P	048	0	080	P	112	p
017	^Q	049	1	081	Q	113	q
018	^R	050	2	082	R	114	r
019	^S	051	3	083	S	115	s
020	^T	052	4	084	T	116	t
021	^U	053	5	085	U	117	u
022	^V	054	6	086	V	118	v
023	^W	055	7	087	W	119	w
024	^X	056	8	088	X	120	x
025	^Y	057	9	089	Y	121	y
026	^Z	058	:	090	Z	122	z
027	^[059	;	091	[123	{
028	^\ ^	060	<	092	\	124	
029	^]	061	=	093]	125	}
030	^^	062	>	094	^	126	~
031	^_	063	?	095	_	127	

Figura 2.1: Exemplo de representação de um texto em código ASCII e a tabela de codificação ASCII. Os símbolos precedidos pelo carácter “^” são introduzidos num computador pessoal pressionando simultaneamente a tecla CTRL e a tecla da letra correspondente.

sinais de áudio

Num sistema electrónico, um sinal de áudio (ou outro sinal analógico) 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 altifalante, provoca a vibração mecânica de um cone de cartão ou plástico, produzindo dessa forma as ondas sonoras.

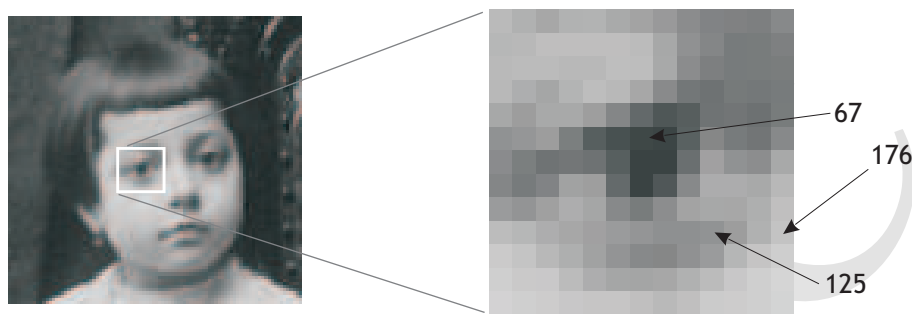


Figura 2.2: Ampliação de uma região de uma imagem digital a preto-e-branco. Numa imagem a preto-e-branco codificada com 1 *byte* por *pixel* existem 256 níveis de cinzento; o branco é representado pelo código 255 e o preto pelo código 0.

Nos antigos discos de vinil (analógicos), o sinal de áudio era gravado como um sulco cravado em espiral sobre o plástico, cujas oscilações representavam o sinal a reproduzir. A reprodução era (e ainda é!) feita por uma agulha que percorre esse sulco e que está associada a um sistema electrónico que traduz essas vibrações mecânicas em sinais eléctricos. Esses sinais, depois de amplificados, são aplicados aos altifalantes que reproduzem o som. Naturalmente que qualquer imperfeição que exista na superfície do disco é traduzida em vibrações indesejáveis da agulha e estas em ruídos que são produzidos pelos altifalantes.

Um sinal de áudio (ou qualquer outro sinal analógico) é representado em binário por um conjunto de números que representam a amplitude do sinal analógico em *amostras* que são lidas em intervalos de tempo regulares definidos por uma *frequência de amostragem*. A este processo chama-se digitalização de um sinal analógico e é caracterizado pela frequência de amostragem e pelo número de *bits* usados para codificar cada amostra. A figura 2.3 mostra um segmento de um sinal analógico e a correspondente representação digital usando 8 *bits* para codificar cada amostra¹.

Apesar de um sinal de analógico variar de forma contínua no tempo, a codificação do valor 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 definição e consequentemente melhor é 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*. Esta codificação é usada em sistemas de comunicação de voz tal como o telefone. No entanto, para áudio de

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

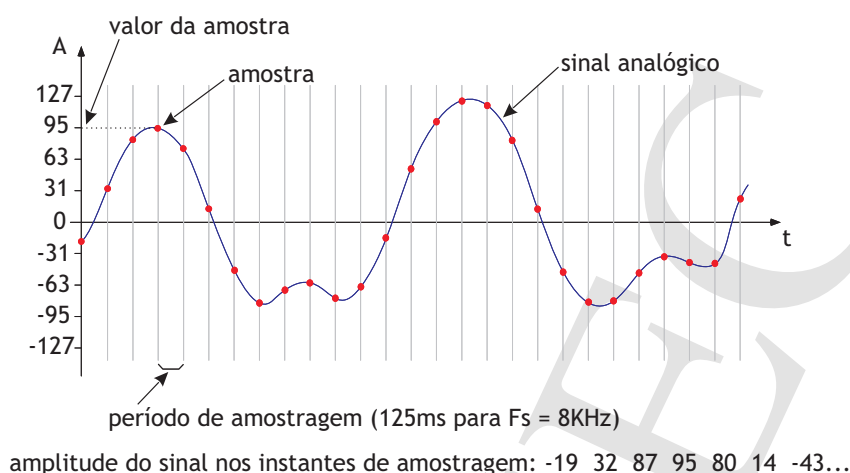


Figura 2.3: Um sinal analógico e a sua representação digital obtida com uma digitalização com $F_s=8\text{KHz}$ e 8 bits por amostra.

alta fidelidade é já necessário amostrar o sinal com uma frequência bastante mais elevada e codificar cada amostra com um maior número de *bits*. Por exemplo, nos CDs de áudio é usada uma frequência de amostragem de 44.1 KHz e cada amostra é representada com 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 produz como resultado um conjunto de *bits* que formam um código representando o valor da grandeza eléctrica medida, geralmente tensão eléctrica.

A função inversa é realizada por um dispositivo chamado conversor digital/analógico ou conversor D/A. Este dispositivo traduz um código binário que representa a amplitude de um sinal, numa tensão eléctrica cujo valor é proporcional ao número representado por esse código. 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 tratado posteriormente por circuitos analógicos que “arredondam” (ou *filtram*) as

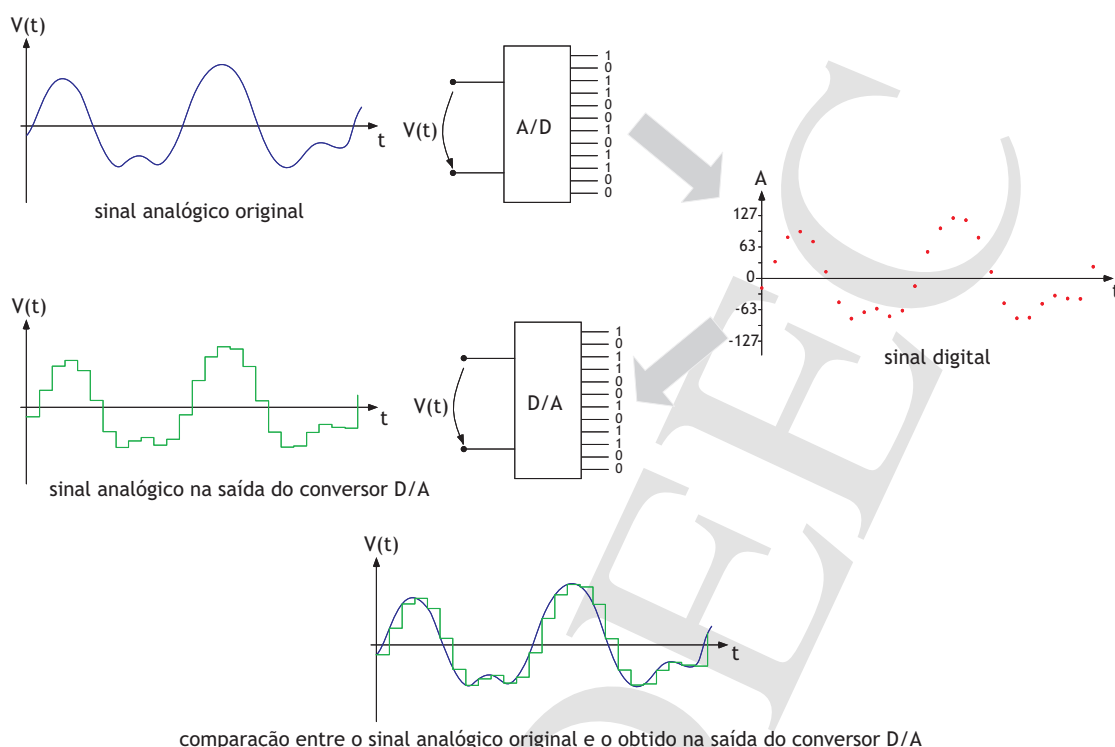


Figura 2.4: Conversores A/D e conversores D/A.

transições abruptas entre amostras e com isso melhoram a qualidade do sinal.

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 inteligível, e de acordo com o tipo de informação representada. Como informação digital não é mais do que colecções de números, a forma como esses números são interpretados deve estar de acordo com o tipo de informação que representa. Por exemplo, não faz sentido imprimir como texto os dados que estão gravados num CD de música, interpretando-os como caracteres ASCII. Do mesmo modo, não seria reproduzido qualquer som inteligível se se tentasse reproduzir um CD de música gravado com a lista de *bytes* que constitui este texto.

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 traduzir informação digital em desenhos que representem caracteres alfabéticos, algarismos ou sinais de pontuação. Existe um tipo particular de mostrador composto por apenas 7 LEDs que são designados vulgarmente por mostradores de 7 segmentos (figura 2.5) e que permitem, de forma económica, afixar os 10 dígitos decimais e alguns

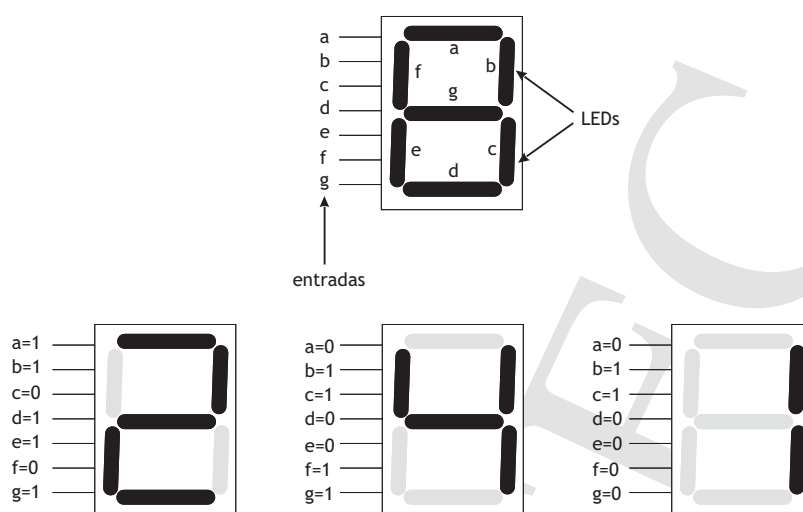


Figura 2.5: Mostradores de 7 segmentos

símbolos adicionais. Este tipo de mostrador é muito comum em variados equipamentos como relógios ou termómetros digitais.

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.

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 números: a cor de um *pixel*, a amplitude de uma amostra de um sinal de áudio ou os códigos atribuídos a caracteres.

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 existem 10 símbolos que representam as quantidades zero a nove: os 10 dígitos decimais. Para representar uma quantidade superior a 9 usamos formamos um conjunto de dígitos onde a posição de cada um define o seu peso no valor do número. Assim, o número representado pela *sequência de algarismos* 345 tem um valor igual a 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 dígitos 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 dúvida 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 utilizando qualquer base $b \geq 2$, quer dizer com qualquer número de “dígitos” superior ou igual a 2^2 . Assim, o valor do número com N dígitos $B_{N-1}B_{N-2}\dots B_2B_1B_0$ representado em base b obtém-se como:

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

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 dígito mais significativo (ou MSD do inglês *Most Significant Digit*) ao dígito da esquerda, 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, que aparece imediatamente à direita do ponto fraccionário (ou ponto decimal), tem um peso igual a 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 calcula-se em base 10 com a expressão:

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

²para já apenas consideraremos a representação de 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: Tabela com as potências inteiras positivas e negativas 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 só é 1 ou zero, as operações a realizar envolvem apenas a adição das potências inteiras de 2 correspondentes aos dígitos que são iguais a 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 formas convenientes para representar informação binária de uma maneira mais compacta do que utilizando directamente o sistema binário (i.e. escrevendo uns e zeros). 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 do ponto fraccionário e para a direita do ponto fraccionário:

$$1101011.11010_2 = 153.64_8$$

Note-se que, neste exemplo, 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 e por isso não modificam o valor representado.

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*, agrupando-os para a direita e para a esquerda a partir do ponto fraccionário:

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

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

O sistema octal teve interesse no início da era dos mini-computadores 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 para leitura, escrita e execução de um ficheiro. 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* iguais a 1 e desligando as que correspondem aos *bits* que são zero (754_8 escreve-se em binário 111101100_2)

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, 1 *byte* (8 *bits*) é representado só por dois dígitos hexadecimais.

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 que se podem representar números em bases numéricas diferentes da base 10 e como se pode obter o valor decimal desses números. Como se pode determinar a representação binária de um número dado em base 10?

Antes de estudarmos como se pode representar em base 2 um número dado em base 10, vejamos como se podem obter os dígitos decimais de um número representado em base 10. Em primeiro lugar, note-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 por b^N um número representado na base b 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 por 10 obtemos à direita do ponto decimal o dígito das unidades, e que é igual ao resto da divisão inteira do número por 10. Se repetirmos este processo aplicado-o sucessivamente aos quocientes obtidos podemos determinar todos os dígitos que compõem o número.

O mesmo procedimento pode ser aplicado para determinar os dígitos binários que representam um número em base dois. 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 igual a zero, determinam-se todos os *bits* que representam o número em binário. A figura 2.7 exemplifica a aplicação deste processo.

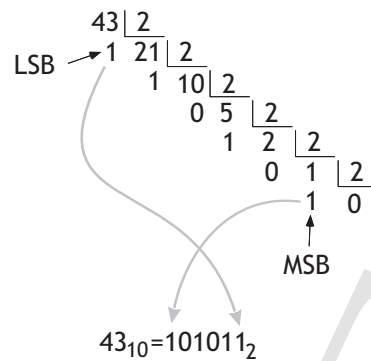


Figura 2.7: Cálculo da representação do número inteiro 43 em base 2.

2.2.5 Como se representa um número fraccionário em base 2?

Em primeiro lugar vejamos como podemos obter os dígitos da parte decimal que compõem um número fraccionário em base 10. Por exemplo, dado o número 0.571, como poderemos “extrair” os 3 dígitos que formam a sua parte decimal? Uma vez que multiplicar um número por 10 equivale a deslocar o ponto fraccionário uma posição para a direita, basta então multiplicar 0.571 por 10 para obter o primeiro dígito do número (o 5) como a parte inteira desse resultado:

$$0.571 \times 10 = 5.71$$

Aplicando repetidamente este processo à parte decimal restante obtêm-se os outros dígitos que formam o número.

Para calcular 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 exemplifica 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 não ser representado exactamente com um número finito de *bits*. Como se mostra na figura 2.9, a representação de 0.8 em binário é formada por uma 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 (por defeito), que é tanto mais exacta quantos mais *bits* forem usados. Por exemplo, se o valor 0.8 for representado pelo número binário com 10 *bits* 0.1100110011_2 , obtém-se uma aproximação por defeito daquele valor igual a $0,7998046875_{10}$.

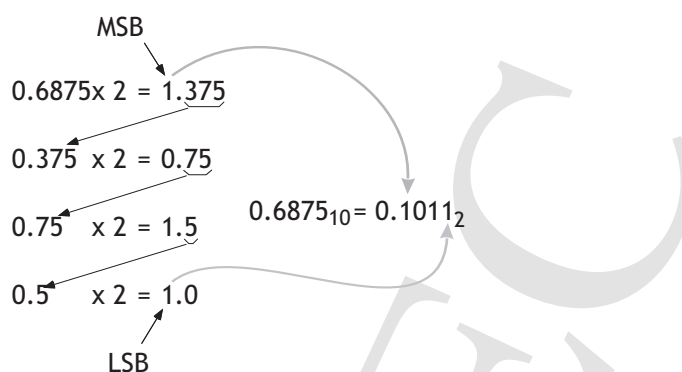


Figura 2.8: Representação do número fraccionário 0.6875 em base 2.

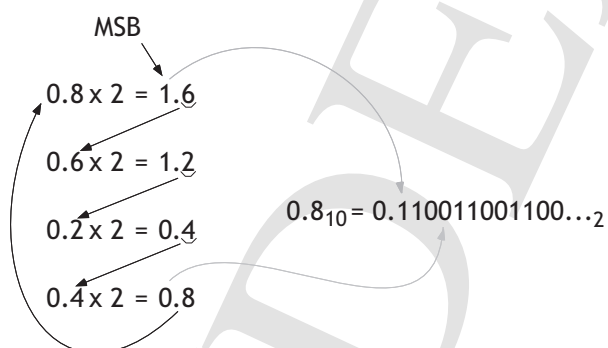


Figura 2.9: Representação em base 2 do número decimal 0.8

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 obtida aplicando separadamente os procedimentos apresentados acima para a parte inteira e para a parte fraccionária. Por exemplo, o número 43.6875_{10} formado pela adição dos números inteiro e decimal usados nos exemplos das 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 números em base 2 (ou em qualquer outra base numérica).

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

$$\begin{array}{r}
 1\ 1\ 0\ 1 \leftarrow \text{transporte} \\
 \hline
 4\ 3\ 4\ 7 \\
 +\ 8\ 9\ 1\ 4 \\
 \hline
 1\ 3\ 2\ 6\ 1
 \end{array}$$

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

$$\begin{array}{r}
 1\ 1\ 0\ 0 \leftarrow \text{transporte} \\
 \hline
 1\ 1\ 0\ 1 \\
 +\ 0\ 1\ 0\ 0 \\
 \hline
 1\ 0\ 0\ 0\ 1
 \end{array}$$

um mais um dá dois (10_2)
 escreve-se 0 e gera-se o transporte 1
 para a soma seguinte

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

em decimal

Figura 2.11: Adição de dois números em base 2.

adição excede 9 (i.e. não pode ser representado por um único dígito do sistema decimal), escrevemos apenas o dígito das unidades como resultado nessa posição e *transportamos* uma unidade para a casa seguinte (figura 2.10).

A adição de 2 números representados em base 2 é feita por um processo idêntico: quando o resultado da adição de dois dígitos não pode ser representado por um único algarismo do sistema binário (i.e. quando não 0 nem 1) transporta-se uma unidade que será somada aos dígitos seguintes. Assim, a operação realizada em cada passo é na realidade a soma de 3 *bits*, podendo dar como resultado as quantidades zero, um, dois ou três, que se representam 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 *bit carry-out* é produzido por um andar da adição e o *bit carry-in* é o que entra no andar seguinte.

A subtração é efectuada em binário da mesma forma que a realizamos no sistema 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 diminuidor, 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. Normalmente essa correcção é feita somando o *bit* de transporte que é assim gerado ao *bit* seguinte do diminuidor; se esta soma der o valor 2 é gerado um novo transporte para o andar seguinte. Na figura 2.12 exemplifica-se este processo.

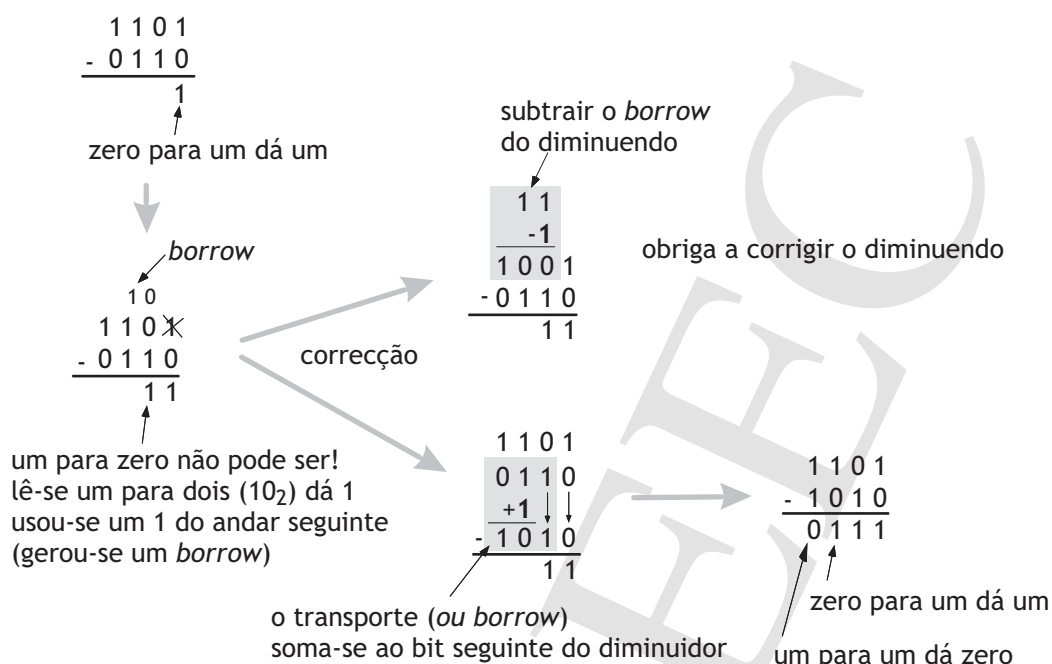


Figura 2.12: Subtração binária $13 - 6 = 7$.

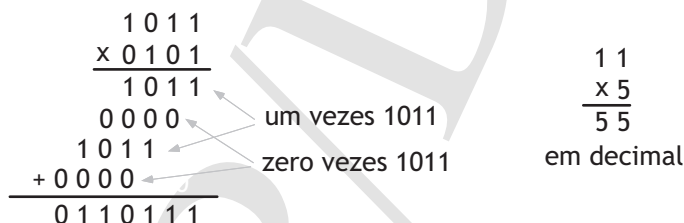


Figura 2.13: Multiplicação binária.

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 parciais de cada dígito do multiplicador pelo multiplicando, “alinhados” pela coluna do dígito do multiplicador. Como os dígitos de um número binário só valem um ou zero, o seu produto pelo multiplicando ou é zero ou é o próprio multiplicando. A figura 2.13 mostra como é realizado o produto de dois números binários.

A divisão binária é um pouco mais complexa de realizar “à mão” mas segue o mesmo algoritmo que usamos para dividir números representados em base 10 (figura 2.14). Note que se o divisor for uma potência inteira de 2 (um número do tipo 2^N , com N inteiro) o

resultado pode calcular-se imediatamente deslocando o ponto fraccionário de N posições para a esquerda.

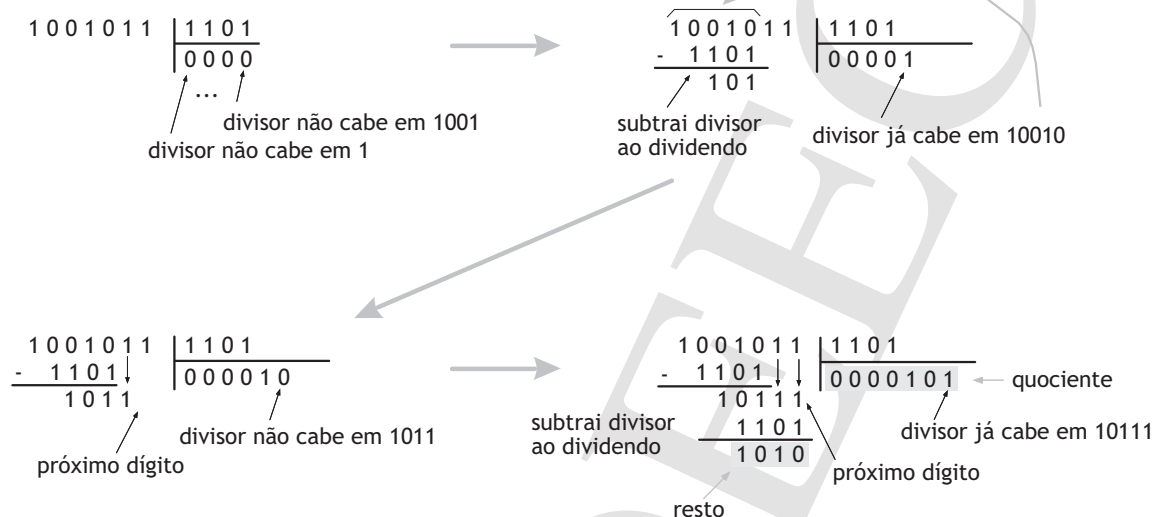


Figura 2.14: Divisão binária.

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 (ou até menos *bits*), o que só permite representar e operar números inteiros positivos no intervalo $[0, 255]^3$.

Quando são realizadas operações aritméticas com números binários, é normalmente necessário conhecer 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, utilizando-se geralmente a palavra inglesa *overflow* para designar esta situação.

Por exemplo, a adição dos números representados 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* (o valor representado pelos 8 *bits* menos significativos é $01100011_2 = 99_{10}$).

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

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

$$\begin{array}{rcl}
 0 \leftarrow \text{transporte} = 0: \text{n\~ao ocorre } \textit{overflow} & & \\
 \begin{array}{r}
 \overset{\curvearrowright}{1} 0 0 1 \quad (9) \\
 + 0 0 1 1 \quad (3) \\
 \hline
 1 1 0 0 \quad (12)
 \end{array} & & \\
 \text{resultado com 4 bits \~e correcto} & & \\
 \\
 1 \leftarrow \text{transporte} = 1: \text{ocorre } \textit{overflow} & & \\
 \begin{array}{r}
 \overset{\curvearrowright}{1} 1 0 0 \quad (12) \\
 + 0 1 1 1 \quad (7) \\
 \hline
 1 0 0 1 1 \quad (\cancel{8})
 \end{array} & & \\
 \text{o resultado s\~o com 4 bits \~e incorrecto} & &
 \end{array}$$

Figura 2.15: Adição binária e *overflow*.

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 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).

$$\begin{array}{rcl}
 0 \leftarrow \text{borrow} = 0: \text{n\~ao ocorre } \textit{overflow} & & \\
 \begin{array}{r}
 \overset{\curvearrowright}{1} 1 0 1 \quad (13) \\
 - 0 1 1 0 \quad (6) \\
 \hline
 0 1 1 1 \quad (7)
 \end{array} & & \\
 \text{resultado com 4 bits \~e correcto} & & \\
 \\
 1 \leftarrow \text{borrow} = 1: \text{ocorre } \textit{overflow} & & \\
 \begin{array}{r}
 \overset{\curvearrowright}{0} 1 1 0 \quad (6) \\
 - 1 1 0 1 \quad (13) \\
 \hline
 1 1 0 0 1 \quad (\cancel{8})
 \end{array} & & \\
 \text{resultado com 4 bits \~e incorrecto} & &
 \end{array}$$

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*), pode-se concluir que $A \geq B$; se ocorrer transporte então

isso significa que o resultado não pode representar correctamente essa diferença e por isso conclui-se que $A < B$.

A operação de multiplicação produz resultados que geralmente, 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, na adição dos produtos parciais, 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 os números negativos são representados precedendo o seu valor de um sinal menos ($-$). Os números positivos não têm sinal ou então são antecidos de um sinal mais ($+$).

A esta forma de representação chama-se *sinal e grandeza* (ou sinal e magnitude) e pode ser usada para representar números com sinal em qualquer base de numeração. Por exemplo, o número negativo -134 pode ser representado como:

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

Embora este seja o processo habitual que é usado pelas pessoas em cálculos manuais, o seu uso obriga a utilizar caracteres adicionais (o sinal menos e o sinal mais) para além dos dígitos que constituem o sistema de representação numérica. Em sistemas de cálculo automático, e em particular em sistemas digitais, é fundamental dispor de processos para representar valores numéricos com sinal que apenas necessitem do conjunto de símbolos usado pelo sistema de numeração. Nesta secção serão estudadas formas para representar números com sinal usando o sistema binário, que podem ser realizados como sistemas electrónicos digitais.

2.6.1 Sinal e grandeza

Números negativos podem ser representados em binário usando um *bit* adicional que apenas representa o sinal do número: 1 significa que o número é negativo e 0 que é positivo. Assim, usando N *bits* é possível representar números positivos e negativos no

intervalo $[-2^{N-1} - 1, 2^{N-1} - 1]$. Por exemplo, com 8 *bits* pode-se representar qualquer número inteiro pertencente ao intervalo $[-127, +127]$, precedendo a representação binária em 7 *bits* do valor absoluto do número de um 0 se o número for positivo ou de um 1 se for negativo:

$$\begin{aligned} +67_{10} &= 01000011_2 & -67_{10} &= 11000011_2 \\ +13_{10} &= 00001101_2 & -13_{10} &= 10001101_2 \end{aligned}$$

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 aritmética que deve ser efectuada entre as grandezas dos operandos, em função do seu sinal; se for feita uma subtracção, é também preciso determinar qual dos 2 números tem 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 realizar todas estas operações, em especial a comparação dos seus valores absolutos e a troca de ordem dos operandos, torna complexa a construção de sistemas digitais que realizem adições entre números com sinal representados neste sistema. Na figura 2.17 exemplifica-se o processo de adição de números representados em sinal e grandeza.

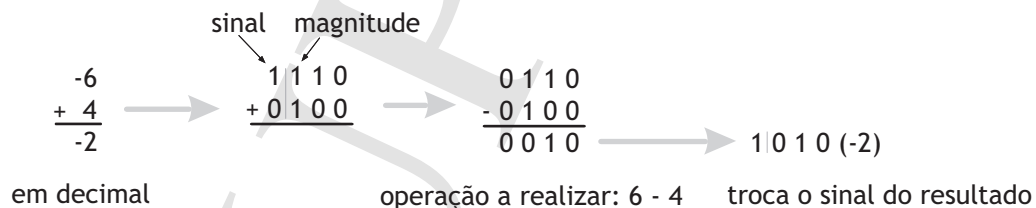


Figura 2.17: Adição binária entre números com sinal representados em sinal e grandeza.

2.6.2 Complemento para a base

Um sistema mais conveniente para a representação de números com sinal é designado complemento para a base (ou complemento para dois no caso da base 2). A representação de números com sinal neste sistema integra a informação do sinal do valor representado no

próprio número e permite simplificar a forma como são realizadas as operações de adição e subtracção. No entanto, as operações de multiplicação e divisão requerem a execução de um processo um pouco mais complexo do que usando representações sinal e grandeza. Antes de estudar a aplicação deste sistema a números representados em base 2, vamos estudar a forma como se podem representar números com sinal no sistema decimal usando complemento para 10.

Complemento para 10

Consideremos um sistema de numeração em que é usado um número fixo de dígitos decimais, por exemplo 4, o que permite representar números inteiros positivos entre 0 e 9999. Em complemento para 10 usando 4 dígitos, representa-se uma quantidade negativa $-X$ por um *número positivo* Y que é obtido pela expressão $Y = 10^4 - X = 10000 - X$. 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. Isto corresponde a dividir o intervalo $[0, 9999]$ sensivelmente a meio onde metade são números positivos e a outra metade corresponde a números negativos.

Experimentemos agora efectuar algumas operações de adição e subtracção com quantidades com sinal representadas em complemento para 10 com 4 dígitos. A realização de operações aritméticas neste sistema deve considerar apenas um número de dígitos do resultado igual ao número de dígitos usado para representar os operandos. Nas operações aritméticas que se exemplificam em seguida, os dígitos que resultam para além dos 4 que iremos considerar são indicados entre parêntesis e não devem ser considerados como fazendo parte do resultado.

Por exemplo, se efectuarmos a subtracção decimal $0 - 17$ aplicando a regra habitual da subtracção decimal e considerando apenas os 4 primeiros dígitos desse resultado, obtemos um número que é igual ao resultado da subtracção $10000 - 17 = 9983$. Pela definição de complemento para 10, este valor representa a quantidade -17 com 4 dígitos:

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

$$10000 - 17 = 9983$$

Vamos agora adicionar à quantidade -17 , que é representada por 9983, o número positivo 23. Se ignorarmos qualquer dígito que resulte dessa operação para além dos 4 dígitos considerados, obtemos o resultado correcto desta operação que é o valor 6:

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

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

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

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

$$-10000 + 9970 = -30$$

Experimentemos agora calcular a diferença $-30 - 9$ e a soma $(-30) + (-9)$, que deverá conduzir ao mesmo resultado -39 :

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

$$-30 + (-9) = 9970 + (10000 - 9) = 9970 + 9991 = (1)9961 \rightarrow -10000 + 9961 = -39$$

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

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

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

Estes resultados mostram que a adição e a subtracção de números positivos e negativos representados neste sistema é feita recorrendo às regras normais da adição e subtracção decimais. Além disso, e ao contrário daquilo que é necessário quando se usa a representação sinal e grandeza, o sinal do número é incorporado no seu valor e não requer qualquer tratamento adicional para decidir a operação a realizar nem o sinal do resultado. Por outro lado, as subtracções podem ser realizadas como adições desde que exista uma forma expedita para trocar o sinal de um dos operandos.

Estes resultados mostram que podemos imaginar que o zero do sistema de numeração que consideramos como exemplo (complemento para 10 com 4 dígitos) é representado pelo número $(1)0000$; os números maiores de $(1)0000$ representam quantidades positivas e os menores de $(1)0000$ representam quantidades negativas. Note que só os 4 dígitos menos significativos são considerados na representação dos valores (figura 2.18).

Neste sistema existe ainda uma indefinição relativamente à forma como se divide o intervalo $[0, 9999]$ entre números positivos e números negativos. Na realidade esse limite pode localizar-se onde pretendermos, desde que se interprete correctamente o “sinal” de um valor numérico. Podemos considerar que apenas são representados números positivos entre 0 e 1000, e que todos os valores entre 1001 e 9999 representam quantidades negativas

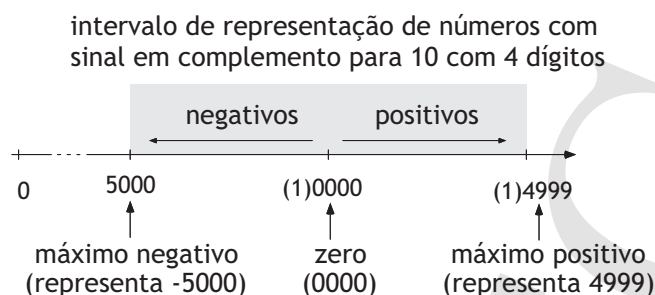


Figura 2.18: Representação de números com sinal em complemento para 10 com 4 dígitos.

de acordo com a definição apresentada antes (o valor mais negativo representável neste sistema seria então $1001 - 10000 = -8999$). No entanto, se se dividir aproximadamente a meio o intervalo $[0, 9999]$ obtém-se uma gama de representação de números com sinal igual a:

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

Esta divisão do intervalo de representação é aproximadamente simétrica e facilita 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, senão é negativo.

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

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

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

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 $9-d$, 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 ao resultado obtido. Aplicando este processo para calcular o simétrico de -3452 , que se representa 6548 em complemento para 10 com 4 dígitos, deveremos complementar os dígitos de 6548 obtendo 3451 e depois adicionar 1 para chegar ao resultado final que é 3452.

Algumas contas mais...

A adição de 2 números positivos ou negativos representados em complemento para 10 com 4 dígitos é realizada normalmente recorrendo ao método de adição de números decimais. No entanto, se essa soma ultrapassar o máximo número positivo que podemos representar neste sistema (que é +4999), então dizemos que foi ultrapassada a capacidade do sistema de representação (ou *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 isso significa que 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, o que é equivalente, subtraindo dois números positivos), nunca é excedida a capacidade do sistema de representação:

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

Adicionando quaisquer dois números negativos também deveremos obter um resultado negativo, caso contrário é excedida a capacidade do sistema de representação. Por exemplo, adicionando -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 (i.e. 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 que 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 subtração.

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 utilizando N *bits* e complemento para dois é dada por:

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

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 em 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 $0B_{N-2}B_{N-3}...B_1B_0$), 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 $1B_{N-2}B_{N-3}...B_1B_0$:

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

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

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

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

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 *bit* mais significativo é 1 e acrescenta a parcela -2^{N-1} ao valor do número.

Consideremos agora alguns exemplos tendo por base o sistema de representação de números com sinal complemento para dois com 4 *bits*, que permite representar 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 $Y = 2^N - X$. Para isso, 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 correspondência com os números binários que os representam.

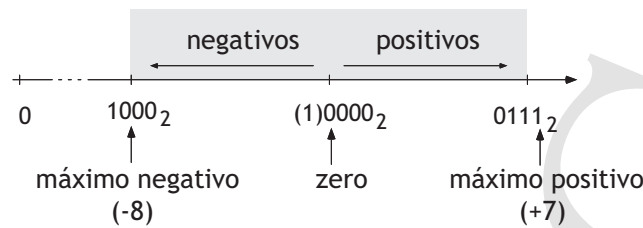


Figura 2.19: Representação de valores com sinal em complemento para 2 com 4 *bits*.

Como 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 complementar 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 adicionando 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 referida antes, 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*. Este processo para detectar esta situação é mais económico do que o 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 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, a subtracção $6 - 4$ pode escrever-se como $6 + (-4)$ e ser realizada como uma operação de adição).

Operando números com diferentes tamanhos

Quando se efectua uma adição binária com números em complemento para dois, os dois operandos e o resultado devem ser representados com 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

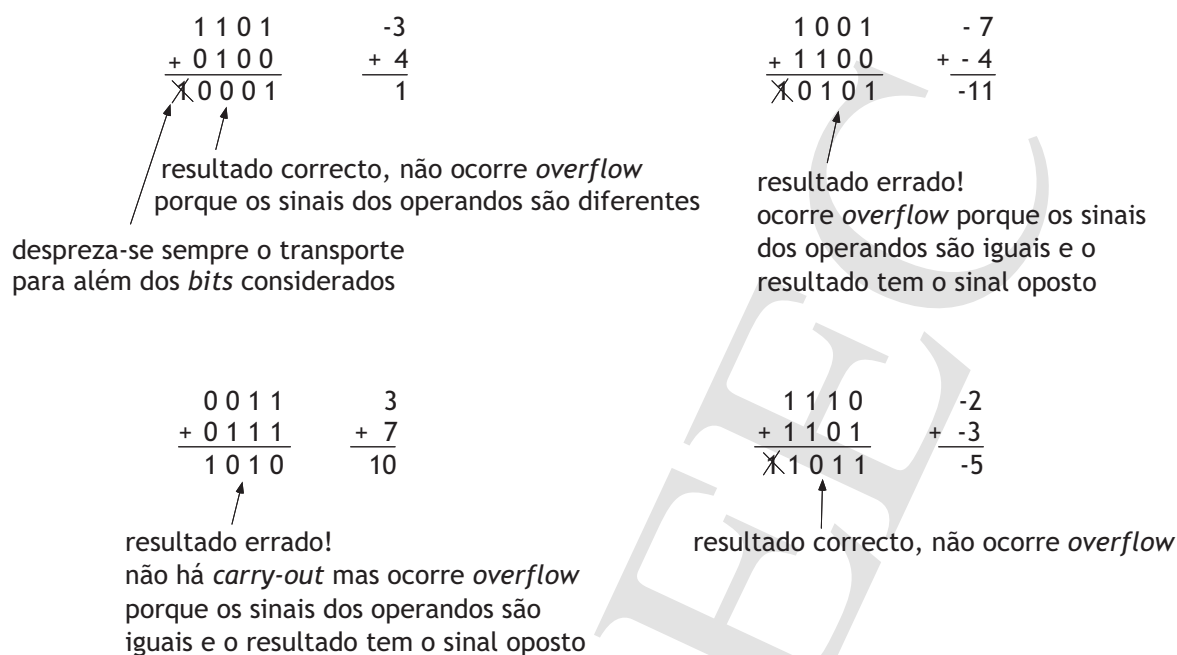


Figura 2.20: Adição de números em complemento para dois com 4 *bits*.

esquerda de forma a preservar o seu sinal. A esta operação chama-se *extensão de sinal* e 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 que, dado o valor -5_{10} representado com 4 *bits* em complemento para dois (1011), se se pretender obter a sua representação com 6 *bits*, deve escrever-se 111011, acrescentando dois *bits* iguais a 1 à esquerda. Se se fosse completado com 2 zeros à esquerda (ficando 001011) passaria a representar o valor positivo $+11_{10}$.

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 que foi apresentado nas secções anteriores. A utilização deste sistema permite utilizar de forma eficiente os diferentes códigos representados por um determinado número de *bits*, e 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 para serem percebidos por um humano (um ecrã, por exemplo), 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...). Para determinar os

adicionar $X=1101$ com $Y=110100$, representados em complemento para 2 com 4 e 6 bits

$$\begin{array}{r}
 1101 \\
 + 110100 \\
 \hline
 \end{array}
 \rightarrow
 \begin{array}{r}
 \text{extensão de sinal de X} \\
 111101 \\
 + 110100 \\
 \hline
 \cancel{1}110001 \\
 \uparrow \\
 \text{resultado com 6 bits: } (-3)+(-12) = -15
 \end{array}$$

adicionar $X=01010$ com $Y=100000$ representados em complemento para 2 com 5 e 6 bits e obter um resultado com 8 bits

$$\begin{array}{r}
 01010 \\
 + 100000 \\
 \hline
 \end{array}
 \rightarrow
 \begin{array}{r}
 \text{extensão de sinal de X} \\
 00001010 \\
 \text{extensão de sinal de Y} \\
 + 11100000 \\
 \hline
 11101010 \\
 \uparrow \\
 \text{resultado com 8 bits: } 10+(-32) = -22
 \end{array}$$

Figura 2.21: Adição entre números binários representados com diferentes números de bits e extensão de sinal.

dígitos decimais que formam um valor numérico é necessário realizar uma série de divisões por 10 que requerem circuitos bastante mais complexos do que os necessários para realizar adições ou subtracções entre números binários.

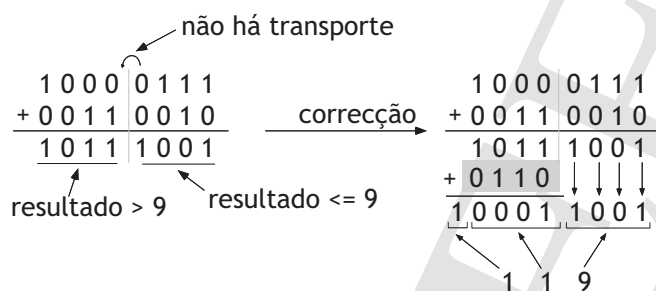
Uma forma alternativa de representar números usando o sistema binário consiste em representá-los como uma série de dígitos decimais onde cada um é representado como um valor de 4 bits entre 0 e 9. Por exemplo o número decimal 3576_{10} pode escrever-se como uma sequência de 4 grupos de 4 bits (num total 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. Por exemplo, 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 maior complexidade dos circuitos digitais que realizam as operações aritméticas 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.

adicionar X = 1000 0111 (87 em BCD) com Y = 0011 0010 (32 em BCD)



adicionar X = 0110 0101 (65 em BCD) com Y = 0111 0110 (76 em BCD)

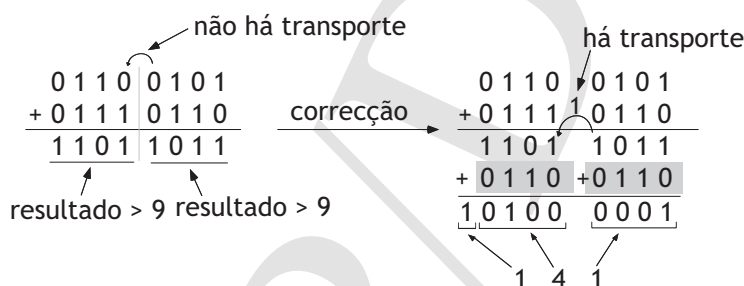


Figura 2.22: Adição de números representados em BCD.

Como esta é na realidade uma representação de números no sistema decimal, a informação do sinal do número 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 para contar o tempo. Usando a codificação BCD não é necessário realizar divisões para obter os dígitos decimais desses valores de forma a afixá-los, por exemplo, num mostrador de LEDs com 7 segmentos.

Capítulo 3

Álgebra de Boole

Como foi visto no capítulo 1, o comportamento de um sistema digital é determinado pela forma como os valores lógicos apresentados pelas saídas do sistema dependem dos valores lógicos colocados nas entradas. Essa função pode ser descrita em linguagem natural (i.e. em Português), com proposições que estabelecem as condições das entradas para as quais as saídas são 1 ou 0. O processo de projecto (ou *síntese*) de um circuito digital consiste em traduzir uma descrição abstracta que define o comportamento pretendido para um sistema, num *circuito lógico* formado por um conjunto elementar de componentes electrónicos que implementam as 3 funções lógicas E, OU e NÃO (ou, em Inglês *AND*, *OR* e *NOT*).

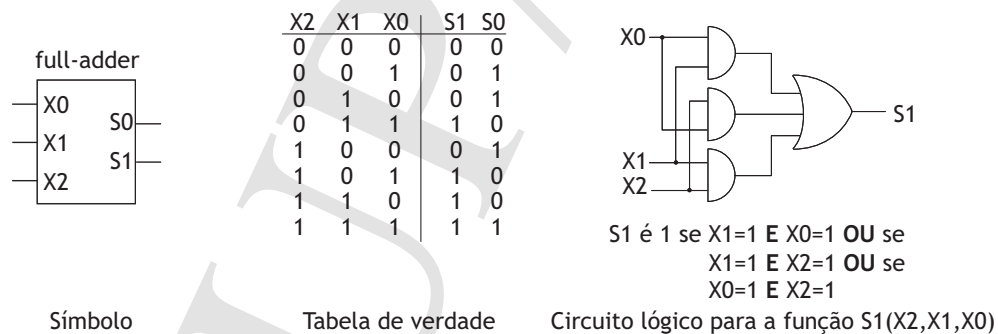


Figura 3.1: Somador completo (*full-adder*): símbolo, tabela de verdade e circuito lógico.

Tomemos como exemplo um circuito lógico que tem por função realizar a adição de 3 números de 1 *bit* ($X0$, $X1$ e $X2$), produzindo um resultado de 2 *bits* ($S1$ e $S0$) que pode ser igual a 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 que se mostra

na figura 3.1. Para construir um circuito lógico que realize essa função, podemos começar por escrever, em Português, as condições que as entradas devem satisfazer para que as saídas sejam iguais a 1. Por exemplo, a função realizada pela saída $S1$, que representa o *bit* mais significativo do resultado, pode ser descrita pela seguinte declaração: *$S1$ é igual a 1 quando duas ou mais entradas forem iguais a 1*. Esta afirmação pode ser reescrita de forma a tornar explícitas as operações lógicas elementares E e OU: *$S1$ 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 formulação conduz ao desenho de um circuito lógico que interliga portas lógicas E e OU, da maneira que se mostra na figura 3.1.

Apesar de neste exemplo ter sido fácil obter um circuito lógico descrevendo a funcionalidade pretendida para o sistema à custa de Es e OUs, não é praticável seguir esta abordagem na generalidade das situações. A álgebra de Boole que se estuda neste capítulo estabelece um conjunto de definições e regras permitindo representar e manipular simbolicamente equações algébricas que traduzem o comportamento de um sistema digital. Estas podem ser transformadas recorrendo a um conjunto reduzido de regras (os axiomas e os teoremas) da álgebra booleana¹ de forma a possibilitar a realização de uma mesma função lógica com circuitos lógicos que apresentam diferentes características e custos variados.

A álgebra de Boole foi inventada em 1854 pelo matemático Inglês George Boole, muito antes da invenção do computador digital. Com esta álgebra, este matemático criou um conjunto universal de regras e operações que permitem combinar proposições verdadeiras ou falsas e avaliar a sua veracidade ou falsidade. Duas ou mais proposições são combinadas usando a conjunção (E), disjunção (OU) e a negação (NÃO), da mesma forma que se mostrou antes para definir o comportamento da função de somador completo.

Em 1938, na altura em que se davam os primeiros passos dos computadores digitais, Claude Shannon adaptou a álgebra de Boole para descrever o funcionamento de circuitos electro-mecânicos construídos com interruptores comandados electricamente (relés). Assim, o estado de um interruptor ou de um circuito eléctrico, que só pode estar ligado ou desligado, é representado pelos estados verdadeiro ou falso de uma variável booleana.

3.1 Axiomas e teoremas

Os axiomas e os teoremas constituem o conjunto de definições que estabelecem as regras de operação da álgebra de Boole. Axiomas representam o conjunto mínimo de regras que

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

se em fundamenta toda a álgebra, e que são assumidos como verdadeiros (i.e. não se podem demonstrar). O conjunto de axiomas da álgebra de Boole definem o que é uma variável *booleana* e as três operações lógicas já referidas antes: E, OU e NÃO. Partindo dos axiomas, pode-se construir um conjunto mais ou menos vasto de teoremas que alargam o conjunto de regras disponíveis para manipular expressões algébricas.

3.1.1 Axiomas

Na figura 3.2 apresenta-se o conjunto de axiomas que definem completamente a álgebra de Boole. O par de axiomas A1 e A1' diz que uma variável booleana (X) apenas pode tomar os valores 0 ou 1 e os restantes definem as três operações elementares da álgebra de Boole: os axiomas A2 e A2' definem a operação NÃO (negação 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:	$X = 0 \text{ se } X \neq 1$	A1':	$X = 1 \text{ se } X \neq 0$
A2:	$\text{se } X = 0 \text{ então } \overline{X} = 1$	A2':	$\text{se } X = 1 \text{ se } \overline{X} = 0$
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 plica após o seu operando (por exemplo, o oposto de X pode-se escrever como \overline{X} ou X').

Pelas semelhanças que apresentam com as operações aritméticas adição e multiplicação, é comum chamar *produto lógico* à função E e *soma lógica* à função OU, representando-as pelos mesmos símbolos que se utilizam para 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), podendo 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'):

$$(X + Y') \cdot (Z \cdot (X + Z)')$$

$$X + Y' \cdot Z \cdot X + Z'$$

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

Os operadores elementares da álgebra de Boole podem também ser representados graficamente utilizando símbolos que têm *entradas* onde são colocados os valores a operar (variáveis independentes) e uma *saída* que representa o valor da função realizada. Uma expressão booleana pode ser representada como um *circuito lógico* formado por vários destes símbolos ligados entre si de forma a compor a função pretendida (figura 3.3).

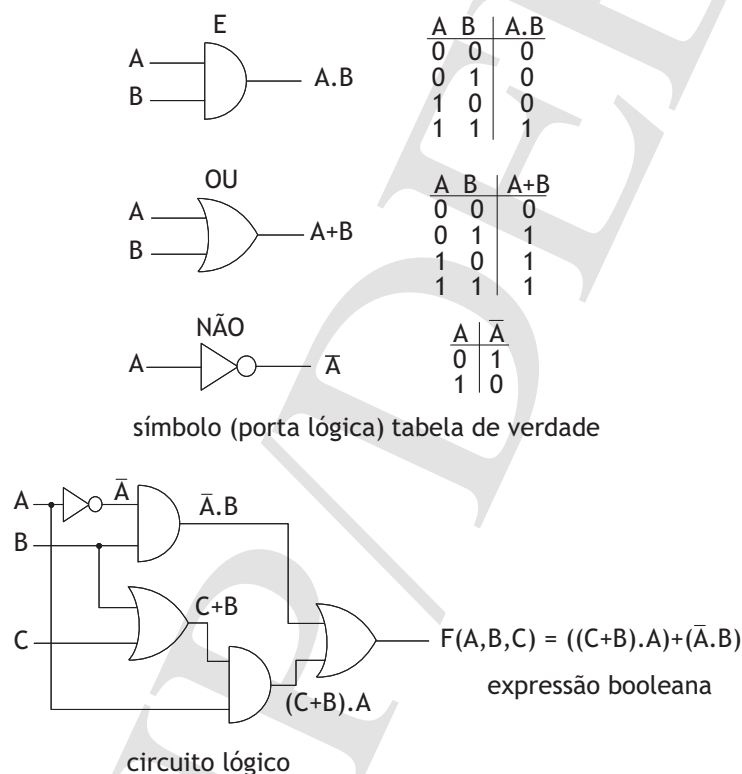


Figura 3.3: As portas lógicas que representam as 3 operações da álgebra booleana e sua aplicação na construção de um circuito lógico.

3.1.2 Teoremas

Os teoremas estabelecem relações de igualdade entre expressões booleanas diferentes que são demonstradas com recurso aos axiomas e a outros teoremas já provados antes. A existência de um conjunto básico de teoremas facilita a manipulação de expressões algébricas, já que não é necessário fundamentar todas as transformações algébricas com recurso às definições mais simples (os axiomas). Na figura 3.4 mostra-se um conjunto de teoremas da

álgebra de Boole envolvendo uma, duas e três variáveis. Estes teoremas são apresentados por ordem crescente de complexidade, o que permite fazer uso de teoremas que já tenham sido provados para fundamentar 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

O princípio da dualidade estabelece que, sendo equivalentes duas expressões booleanas, então também o são aquelas que se obtêm trocando entre si os Es com os OUs, os OUs com os Es, os zeros por uns e uns por zeros (as negações não sofrem alterações). Todos os teoremas “linha” podem ser obtidos à custa dos teoremas “sem linha” trocando entre si os operadores soma lógica e produto lógico e também os valores lógicos 0 e 1. Note-se que esta regra apenas pode ser aplicada a relações de igualdade entre expressões booleanas: 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 que seja mantida a ordem de avaliação das operações.

Leis de DeMorgan

Os teoremas T12 e T12' (figura 3.4) são chamados leis de DeMorgan e permitem transformar operações lógicas E em operações OU e vice-versa:

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

e, aplicando o princípio da dualidade:

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

Este teorema pode ser generalizado para uma função qualquer com N variáveis, permitindo obter a sua negação trocando entre si os operadores E e OU e substituindo cada variável pela sua negação:

$$[f(X_1, X_2, X_3, \dots, X_n, +, \cdot)]' = f(X_1', X_2', X_3', \dots, X_n', \cdot, +)$$

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

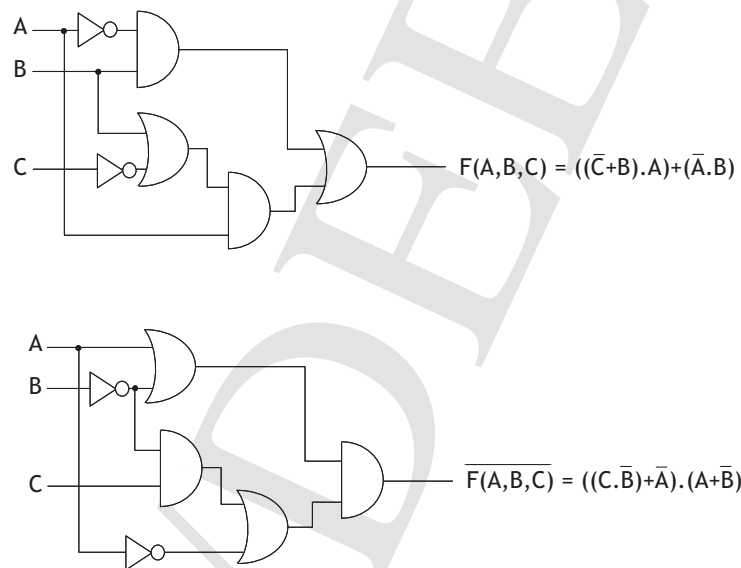
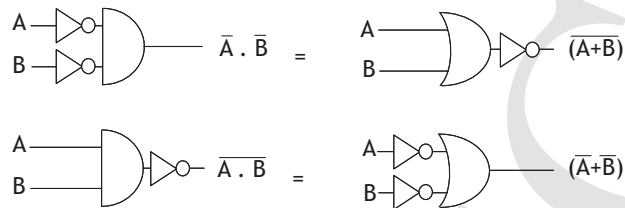
Aplicação dos teoremas

O conjunto de axiomas e teoremas apresentados constituem um conjunto de regras formais que permitem manipular simbolicamente expressões booleanas de maneira a obter realizações variadas para uma mesma função. Na actividade de projecto de sistemas digitais, é importante construir sistemas electrónicos que realizem uma função lógica pretendida, mas que ao mesmo tempo também consigam satisfazer outras metas igualmente importantes: minimizar o custo, o espaço ocupado ou a energia eléctrica consumida. Na figura 3.6 exemplifica-se a aplicação de alguns dos teoremas apresentados para provar a equivalência entre duas expressões booleanas.

3.2 Representação de funções booleanas

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 formado pelas 2^N combinações possíveis dos valores binários das suas N variáveis e o contra-domínio é o conjunto $\{0, 1\}$.

Leis de DeMorgan com duas variáveis

**Figura 3.5:** Aplicação das leis de DeMorgan para obter a negação de uma função.Como provar que $(A \cdot B + (A' + B)') + A \cdot C = A$?

$$\begin{aligned}
 (A \cdot B + (A' + B)') + A \cdot C &= \text{(Teorema T12 - Leis de DeMorgan)} \\
 A \cdot B + A \cdot B' + A \cdot C &= \text{(Teorema T8)} \\
 A \cdot (B + B') + A \cdot C &= \text{(Teorema T5)} \\
 A \cdot 1 + A \cdot C &= \text{(Teorema T1')} \\
 A + A \cdot C &= \text{(Teorema T9)} \\
 A &
 \end{aligned}$$

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

Para além de expressões algébricas, uma função booleana pode também ser representada de várias outras formas:

- **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). Geralmente desenham-se as *entradas* do circuito (as variáveis independentes da função) do lado esquerdo e a saída (que representa o valor da função) do lado direito do esquema.

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

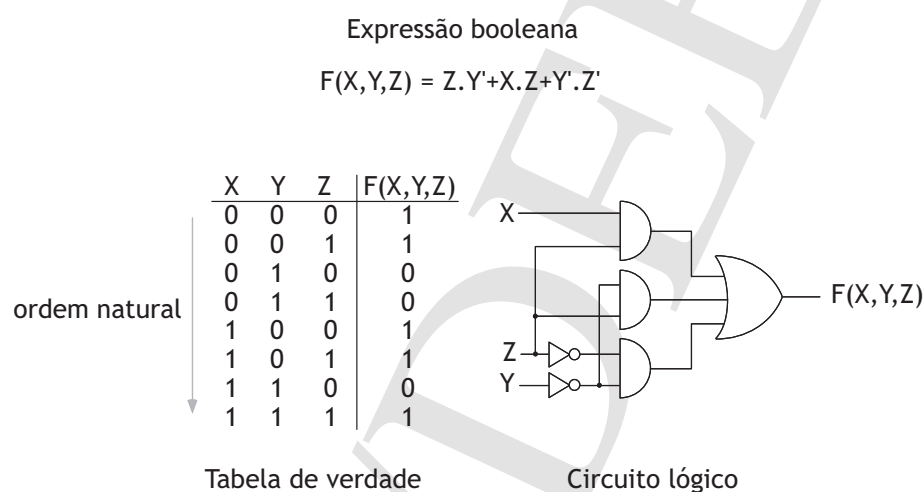


Figura 3.7: Representação de uma função booleana como uma expressão algébrica, tabela de verdade e circuito lógico.

Uma função booleana pode ser representada por uma expressão algébrica arbitrária, i.e. que não segue qualquer estrutura pré-definida. Existem no entanto certas formas padrão usadas para representar funções booleanas, que são mais adequadas para a aplicação das técnicas de minimização lógica que iremos estudar no próximo capítulo. Algumas definições fundamentais necessárias para este estudo são:

- **Literal** é uma variável ou a sua negação. Embora em matemática seja usual representar uma variável por uma só letra, em Sistemas Digitais é recomendável usar nomes para variáveis booleanas que tenham algo a ver com o significado que essa variável tem no sistema digital a que pertence. São exemplos de literais:

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

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

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

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

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

- **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):

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

- **Termo mínimo**² de N variáveis é um termo normal de produto com exactamente N variáveis. Um termo mínimo só vale 1 para uma e uma só combinação dos valores das suas variáveis (corresponde a uma linha da tabela de verdade da função). Exemplos de termos mínimos com 3 variáveis X , Y e Z são:

$$X.Y.Z, X.Y'.Z', X'.Y'.Z'$$

Cada termo mínimo corresponde exactamente a uma linha da tabela de verdade de uma função e é identificado pelo número de ordem dessa linha da tabela, contando a partir de zero. Note que este é igual ao valor do número binário formado pelos valores que as variáveis independentes assumem nessa linha. Assim, o termo mínimo $X.Y'.Z'$ que só é igual a 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 facilmente 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 um ponto de partida para construir um circuito lógico que realize essa função. Por exemplo, a soma canónica da função apresentada na figura 3.8 escreve-se:

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

	X	Y	Z	F(X,Y,Z)	termos mínimos
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	$X.Y.Z$

n° do termo →
 termos mínimos em que F(X,Y,Z) é 1

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

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

Esta expressão representa a mesma função dada na tabela de verdade da figura 3.8 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. Serão estudadas mais à frente outras formas mais compactas para representar funções booleanas e que contêm exactamente a mesma informação do que a expressão canónica ou a tabela de verdade.

Termos de soma, termos máximos e produto canónico

Fazendo uso do princípio da dualidade, pode-se alargar ao operador OU o conjunto de definições dadas antes para o operador E:

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

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

- **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):

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

- **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 de soma):

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

- **Termo máximo**³ de N variáveis é um termo normal de soma com exactamente N variáveis. Um termo máximo só vale 0 para uma e uma só combinação dos valores das suas variáveis, o que corresponde a exactamente uma linha da tabela de verdade. Exemplos de termos máximos com as 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, podendo ser identificado pelo número dessa linha. Assim, o termo máximo $X + Y' + Z'$ só vale 0 quando for $X = 0$, $Y = 1$ e $Z = 1$ e 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	$X'+Y'+Z'$

nº do termo →

termos máximos em que F(X,Y,Z) é 0

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*:

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

Lista de termos mínimos e lista de termos máximos

Para definir completamente uma função booleana, basta identificar as linhas da tabela de verdade em que a função vale 1. Identificando cada linha da tabela de verdade pelo seu número de ordem, tal como é mostrado na figura 3.8, uma função pode ser completamente definida listando apenas os números das linhas da sua tabela de verdade em que assume o valor 1. A esta forma de representação chama-se lista de termos mínimos. Assim, a função apresentada nas figuras 3.8 e 3.9 pode ser representada pela lista de termos

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

mínimos 0,1,4,5,7; fazendo uso do princípio da dualidade, podemos também descrever a mesma função pela sua lista de termos máximos 2,3,6.

Como os números que representam as linhas da tabela de verdade são os valores dos números binários desenhados, a associação entre número da linha e os valores que as variáveis tomam depende da ordem pela qual as variáveis são dispostas na tabela de verdade. Por exemplo, se a mesma função que foi usada no exemplo anterior 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 sempre necessário identificar a ordem pela qual são consideradas as suas variáveis.

ordem das variáveis modificada

	X	Y	Z	F(X,Y,Z)		Y	Z	X	F(X,Y,Z)
0	0	0	0	1	0	0	0	0	1
1	0	0	1	1	1	0	0	1	1
2	0	1	0	0	2	0	1	0	1
3	0	1	1	0	3	0	1	1	1
4	1	0	0	1	4	1	0	0	0
5	1	0	1	1	5	1	0	1	0
6	1	1	0	0	6	1	1	0	0
7	1	1	1	1	7	1	1	1	1

termos mínimos: 0,1,4,5,7 termos mínimos: 0,1,2,3,7

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, produtório *produto* de somas), as variáveis na ordem pela qual são consideradas na tabela de verdade e a lista de números que identificam os termos mínimos (ou máximos). Para a função que tem vindo a ser usada como exemplo, as listas de termos mínimos e máximos são:

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

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

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 regras formais permitindo escrever e transformar expressões que modelam o comportamento de um sistema digital. Mostrou-se como se pode representar uma função booleana como um circuito lógico, interligando símbolos que representam os operadores elementares da álgebra de Boole: E, OU e inversor. 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 traduza o comportamento pretendido para um sistema digital (por exemplo, a expressão canónica soma-de-produtos), pode-se obter um circuito lógico que a realize representando cada operador elementar pela porta lógica respectiva. Com as ferramentas que a álgebra de Boole oferece, é possível modificar uma expressão algébrica de forma a obter um circuito electrónico digital que permita otimizar uma ou mais medidas de qualidade do sistema, como por exemplo reduzir o seu custo ou o consumo de energia. Por exemplo, se as portas lógicas do tipo OU forem mais caras do que as portas E e os inversores, pode ser conveniente manipular as equações que descrevem o comportamento de um circuito de forma a que apenas sejam usadas portas E e inversores e com isso se consiga obter um circuito mais económico.

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 des-

ignados 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. Esta categoria de circuitos caracteriza-se por incluir alguma forma de *memória* que guarda a história anterior dos valores das entradas e requer uma forma de representação, análise e projecto 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 de combustível e o custo de produção. No entanto, no projecto de um motor destinado à competição já será mais importante maximizar a potência ou o binário desenvolvido, relegando para um segundo plano o consumo ou mesmo o custo. Ao projectar um automóvel é necessário estabelecer *compromissos* entre características que não podem ser melhoradas ao mesmo tempo. Por exemplo, o tamanho exterior e a habitabilidade ou a potência do motor e o consumo de combustível: a solução ideal seria um carro pequeno por fora e grande por dentro, potente e ao mesmo tempo económico.

No projecto de circuitos electrónicos digitais podem também ser considerados variados critérios para avaliar a sua qualidade. Os mais importantes e que geralmente são considerados em qualquer projecto, são o custo monetário, a rapidez de funcionamento (muito 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 que iremos medir pelo número de portas lógicas usadas pelo 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 reduzir ao mínimo o número de portas lógicas necessárias para a implementar. 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, 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 com 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 (relés¹), 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 baseado 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 transístor construído 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 transístor² ocuparia um espaço equivalente a um rectângulo com $5mm \times 10mm$, correspondendo a uma área aproximadamente 100×10^6 vezes menor.

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

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

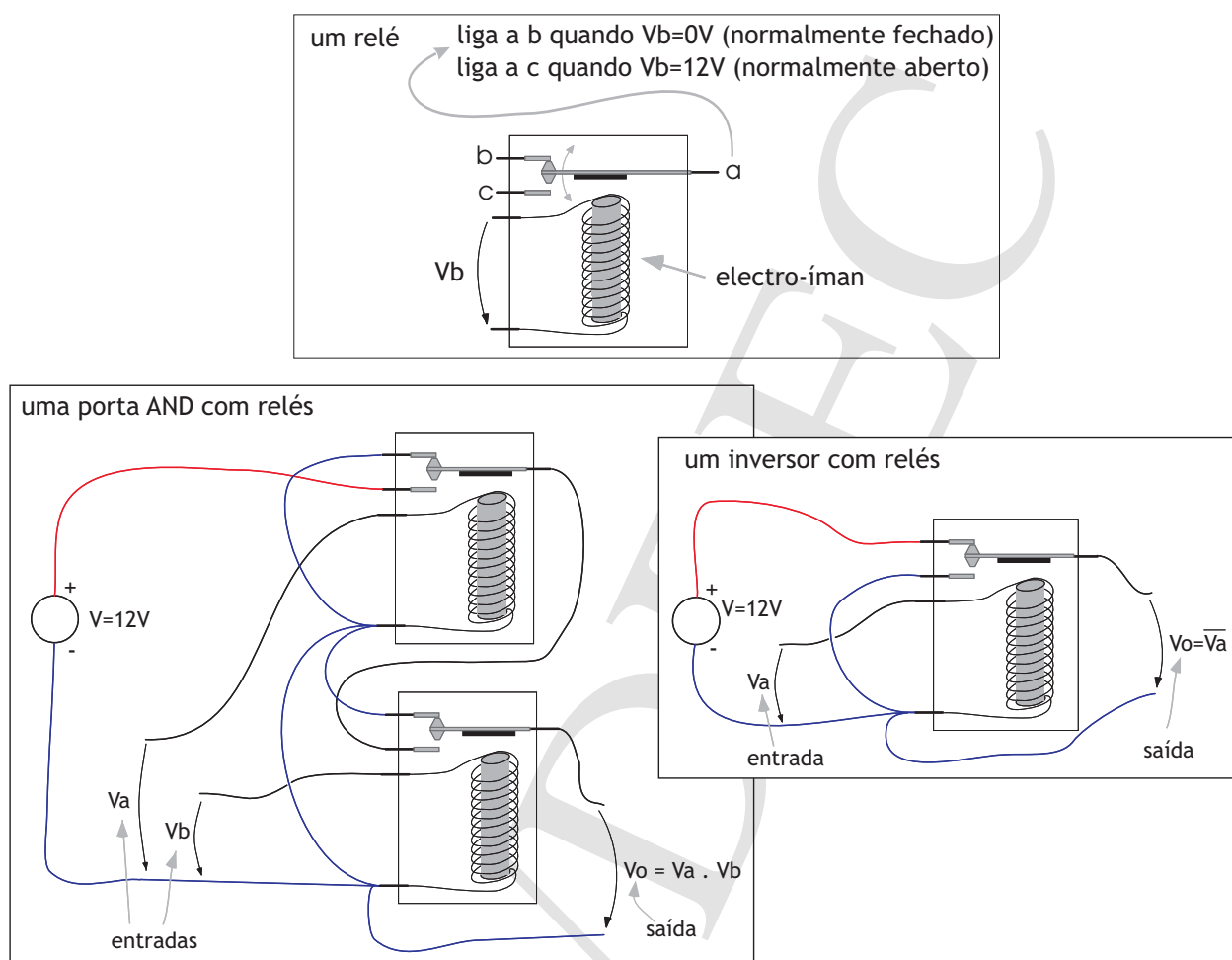


Figura 4.1: Realização das funções lógicas AND e NOT com relés; a função AND é equivalente à associação de dois interruptores em série: a saída só apresenta 12V quando ambas as entradas forem iguais a 12V; a função NOT inverte o nível lógico representado pela tensão eléctrica na sua entrada: quando se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

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.

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 lógica³) 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 complexidade muito reduzida, não ultrapassando 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 (sempre que uma entrada é ligada a 5V, essa ligação deve ser feita através de uma resistência com valor entre 1K Ω e 10K Ω).

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

³Uma 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.

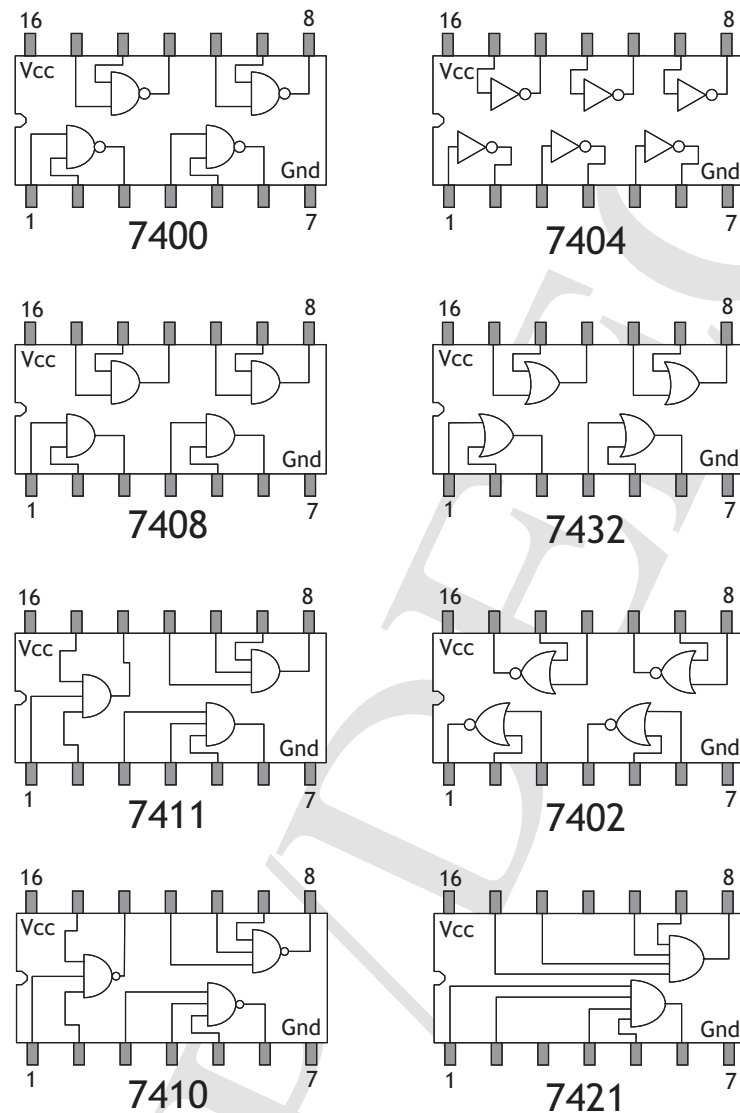


Figura 4.2: Organização interna de alguns dispositivos da série 74XXnnn.

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:

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

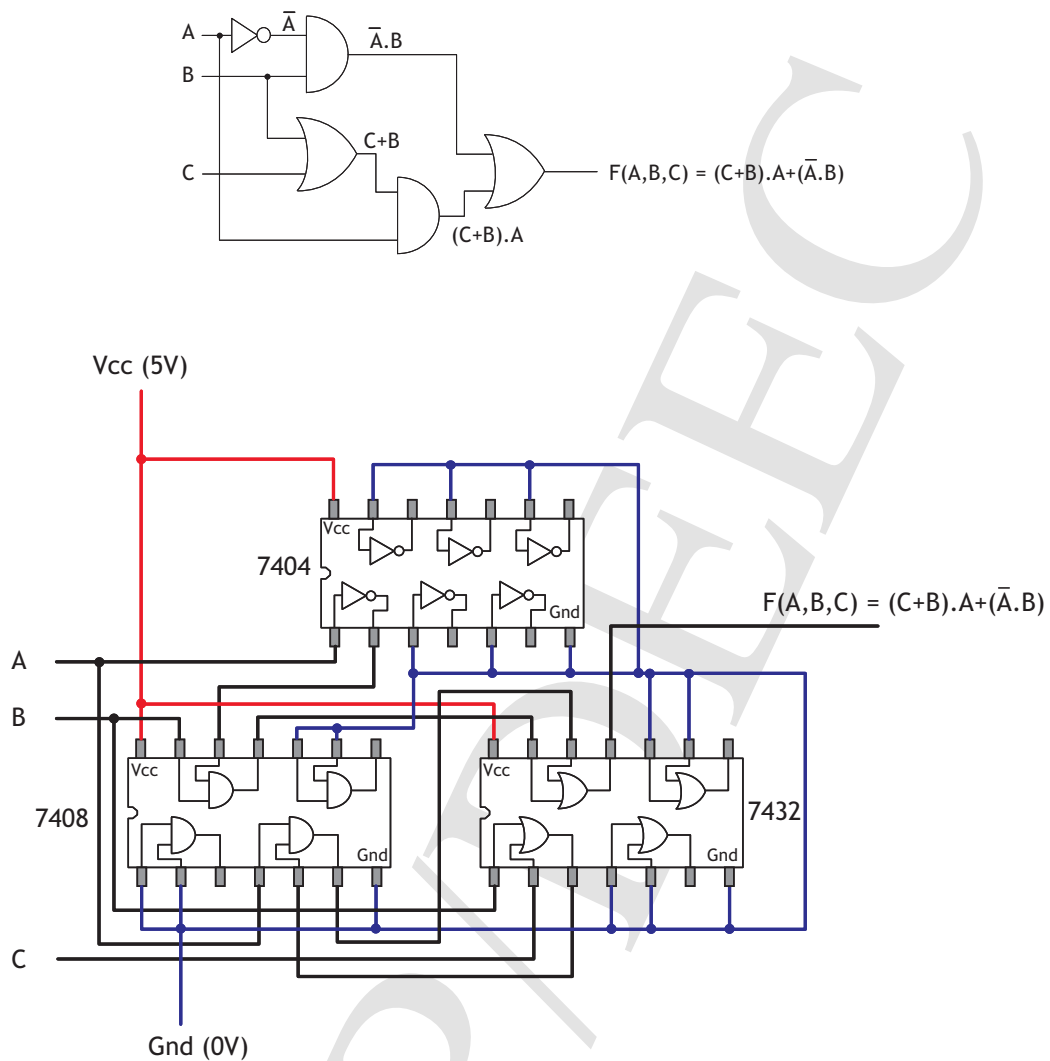


Figura 4.3: Realização de um circuito lógico com dispositivos SSI da série 74.

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).

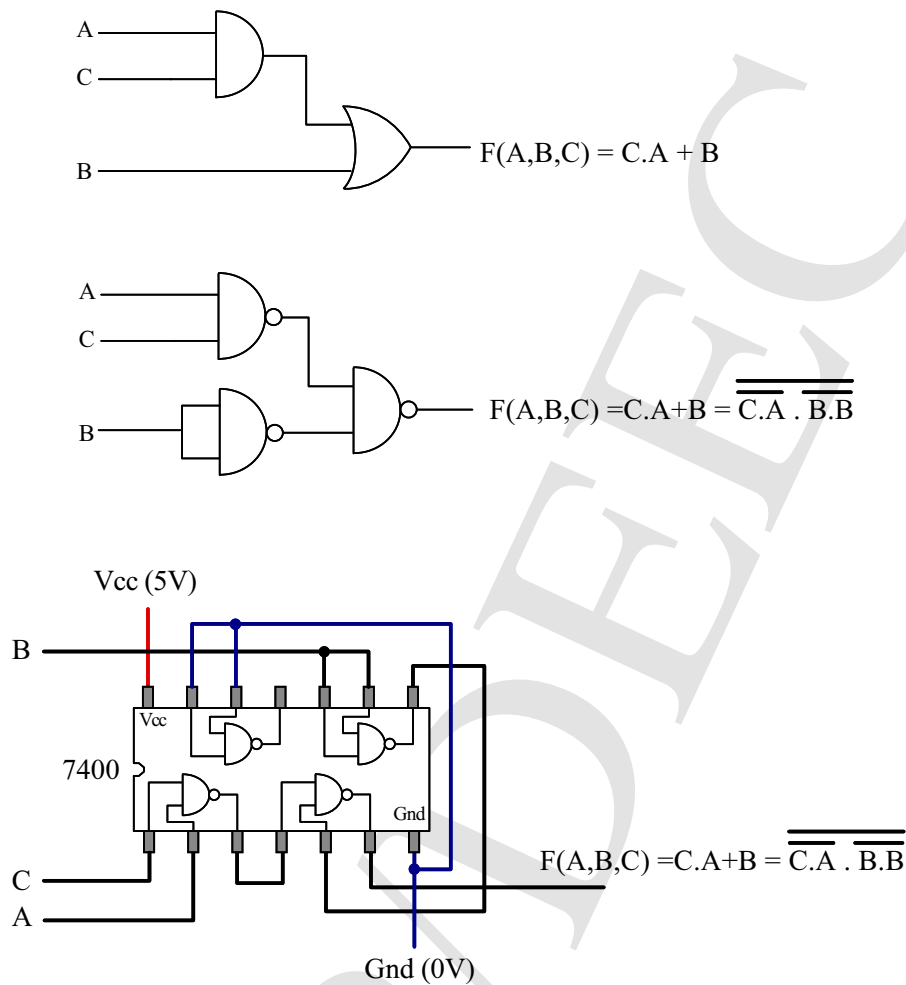


Figura 4.4: Optimização do circuito lógico mostrado na figura 4.3.

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 operadores 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 variáveis⁴ 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.

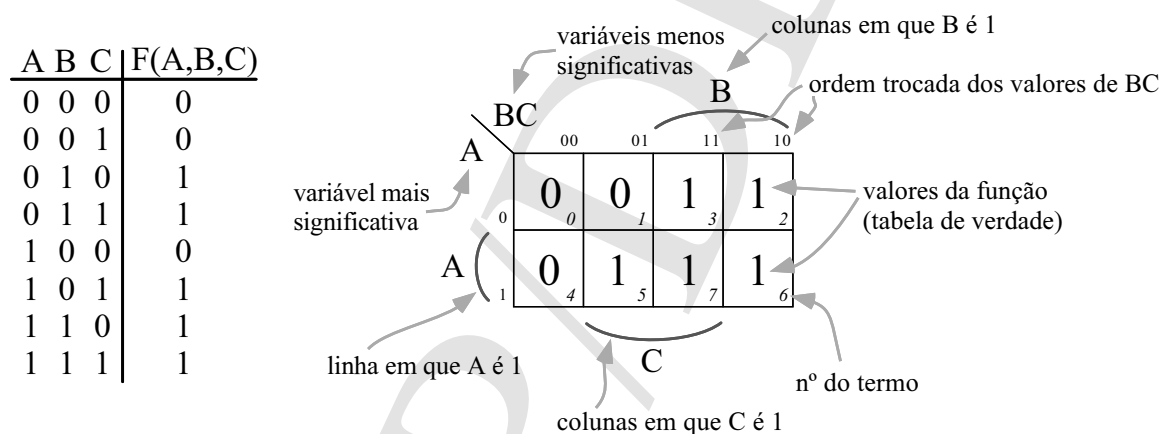


Figura 4.5: Representação de uma função booleana de 3 variáveis num mapa de Karnaugh.

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

⁴será 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

correspondentes às linhas da tabela de verdade em que a função vale 1. Para a função exemplificada na figura 4.5, esta expressão é:

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

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

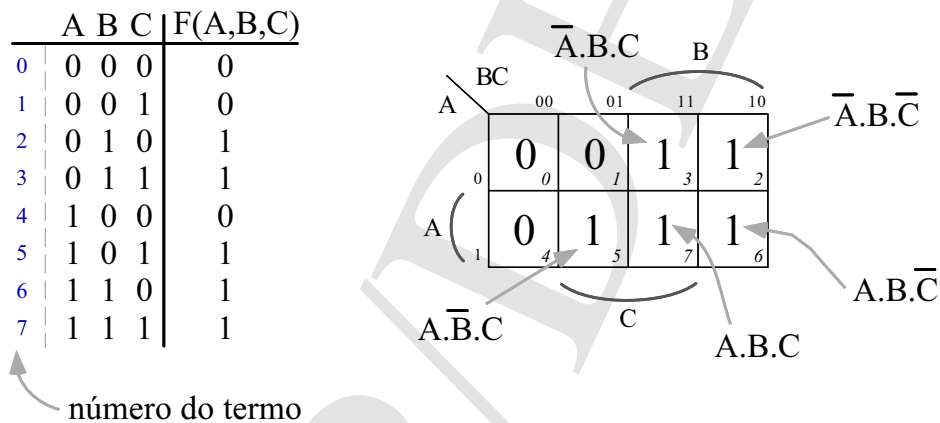


Figura 4.6: Termos de produto para uma função booleana de 3 variáveis representada num mapa de Karnaugh.

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 ($\bar{A}.B.C$), onde a variável A aparece nas formas negada e não negada (aplicação directa do teorema T10):

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

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 :

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

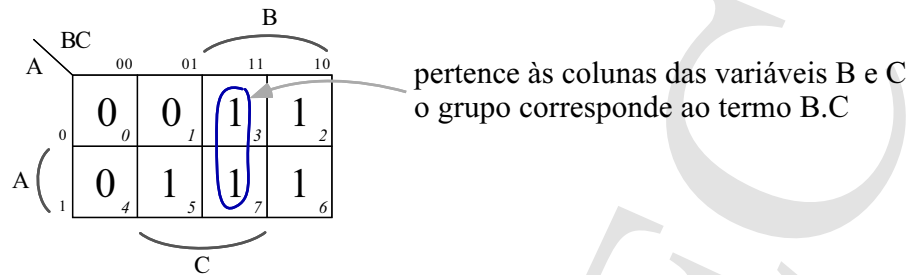
Este passo de combinação pode ser entendida como o agrupamento dos 2 grupos de uns criados anteriormente. O termo de produto correspondente a este grupo de 4 uns pode ser lido directamente do mapa de Karnaugh como B , já que o grupo pertence às colunas da variável B , e as variáveis A e C estão parcialmente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

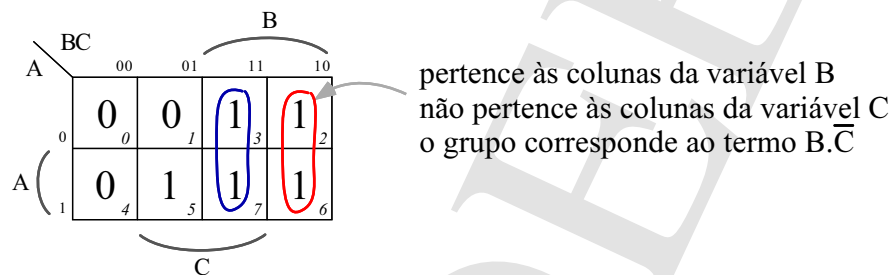
O processo de minimização de expressões booleanas na forma soma de produtos é realizado agrupando os uns no mapa de acordo com as regras seguintes:

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).

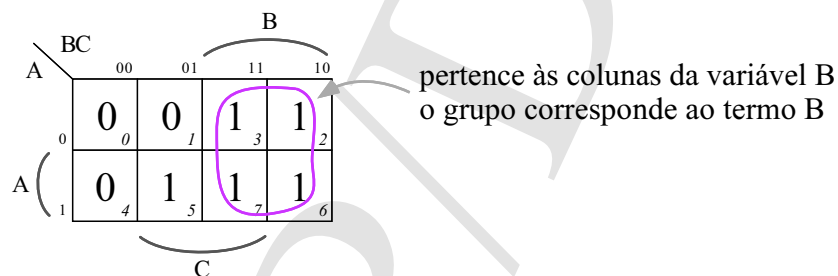
Combinando os termos 3 e 7: $\bar{A}.B.C + A.B.C = B.C.(\bar{A}+A) = B.C$



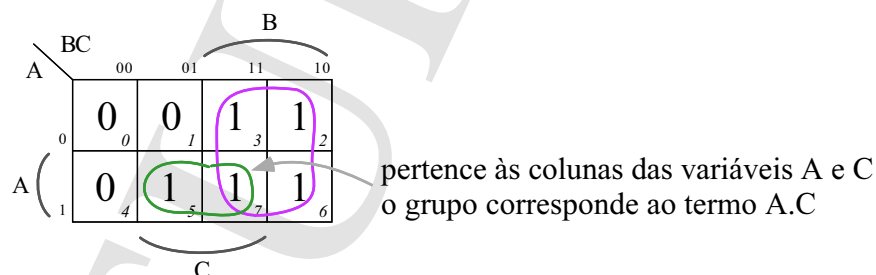
Combinando os termos 2 e 6: $\bar{A}.B.\bar{C} + A.B.\bar{C} = B.\bar{C}(\bar{A}+A) = B.\bar{C}$



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



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



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.

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).

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:

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

diz-se que os termos de produto ($X.Y.\bar{Z}$, $\bar{X}.Z$ e $\bar{X}.\bar{Y}$) 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 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 $A\bar{D}$ 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 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 desenhado 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 48), 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:

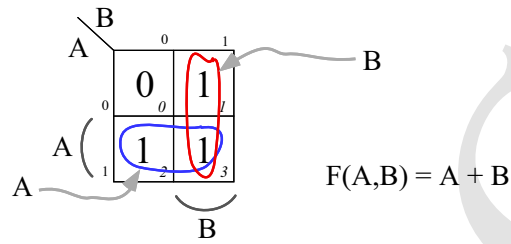
$$F(A, B, C, D) = \sum_{A,B,C,D} (2, 5, 6, 8, 10, 12, 13, 14)$$

ou, na forma lista de termos máximos:

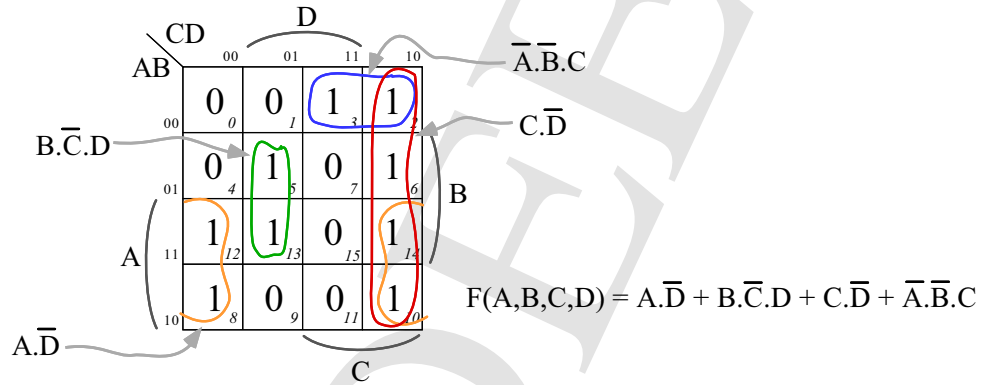
$$F(A, B, C, D) = \prod_{A,B,C,D} (0, 1, 3, 4, 7, 9, 11, 15)$$

então o oposto desta função pode ser representado por:

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



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



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

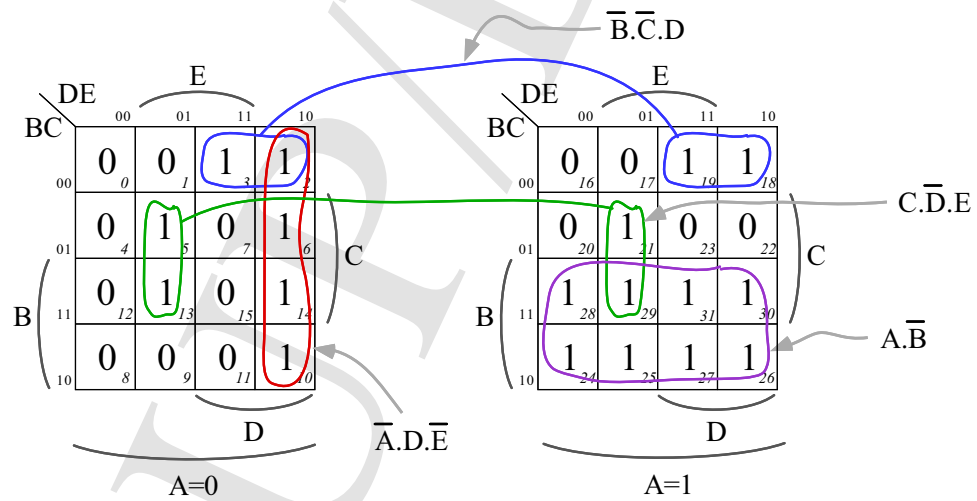


Figura 4.8: Mapas de Karnaugh para funções de 2, 4 e 5 variáveis.

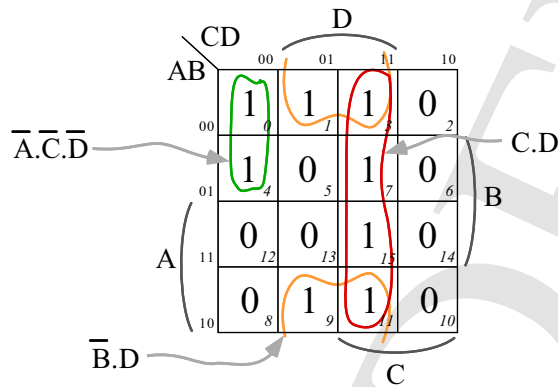
$$\overline{[F(A, B, C, D)]} = \sum_{A,B,C,D}(0, 1, 3, 4, 7, 9, 11, 15)$$

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

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

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

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



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

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

$$F(A, B, C, D) = (B + \overline{D}) . (A + C + D) . (\overline{C} + \overline{D}) \rightarrow \text{Teorema generalizado de DeMorgan} \rightarrow \text{Expressão mínima produto de somas}$$

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

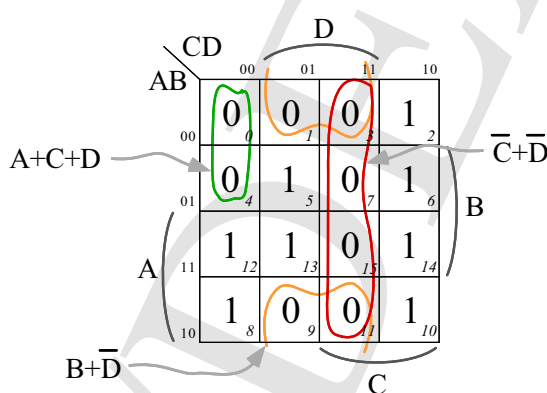
Recorrendo ao teorema generalizado de DeMorgan pode-se obter facilmente a função mínima na forma produto de somas como a negação da expressão anterior, bastando trocar os operadores lógicos OU e E e as variáveis pelas suas negações:

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

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 52).

Pode-se assim relacionar 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.

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



Expressão mínima produto de somas:

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

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

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, 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 decodificador 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.

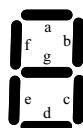
Na figura 4.11 apresenta-se a tabela de verdade para as 7 funções que realizam um decodificador BCD para 7 segmentos, e é exemplificada a aplicação do procedimento de minimização para as funções que implementam os segmentos **g** e **e**. Note que os termos indiferentes apenas devem fazer parte dos grupos 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 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 booleanas 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 $g(b_3, b_2, b_1, b_0)$ 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):

$$g(b_3, b_2, b_1, b_0) = (b_3 + b_2 + b_1).(\overline{b_3} + b_1 + b_0).(\overline{b_2} + \overline{b_1} + \overline{b_0})$$

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 $g(b_3, b_2, b_1, b_0)$, embora sejam funções booleanas distintas entre si.

disposição dos LEDs num mostrador de 7 segmentos



os 10 dígitos decimais desenhados num mostrador de 7 segmentos

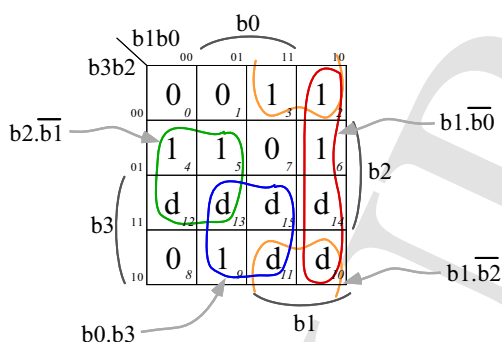


tabela de verdade de um decodificador BCD para 7 segmentos

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
					d	d	d	d	d	d	d
					d	d	d	d	d	d	d
					d	d	d	d	d	d	d
					d	d	d	d	d	d	d
					d	d	d	d	d	d	d
					d	d	d	d	d	d	d
					d	d	d	d	d	d	d
					d	d	d	d	d	d	d

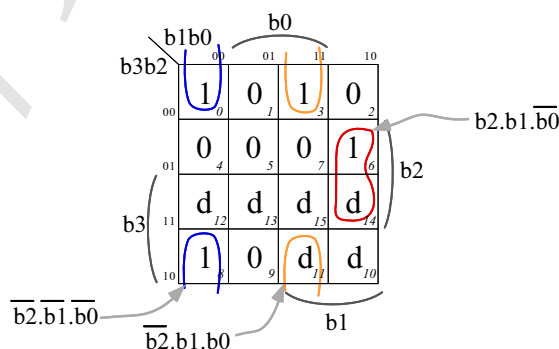
termos *don't care*

minimização da função $g(b_3, b_2, b_1, b_0)$



$$g(b_3, b_2, b_1, b_0) = b_2 \cdot b_1 \cdot b_0 + b_0 \cdot b_3 + b_1 \cdot b_0 + b_1 \cdot b_2$$

minimização da função $e(b_3, b_2, b_1, b_0)$



$$e(b_3, b_2, b_1, b_0) = \overline{b_2} \cdot b_1 \cdot b_0 + \overline{b_2} \cdot \overline{b_1} \cdot \overline{b_0} + b_2 \cdot b_1 \cdot \overline{b_0}$$

Figura 4.11: Minimização de funções booleanas com termos indiferentes: um decodificador BCD para 7 segmentos.

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

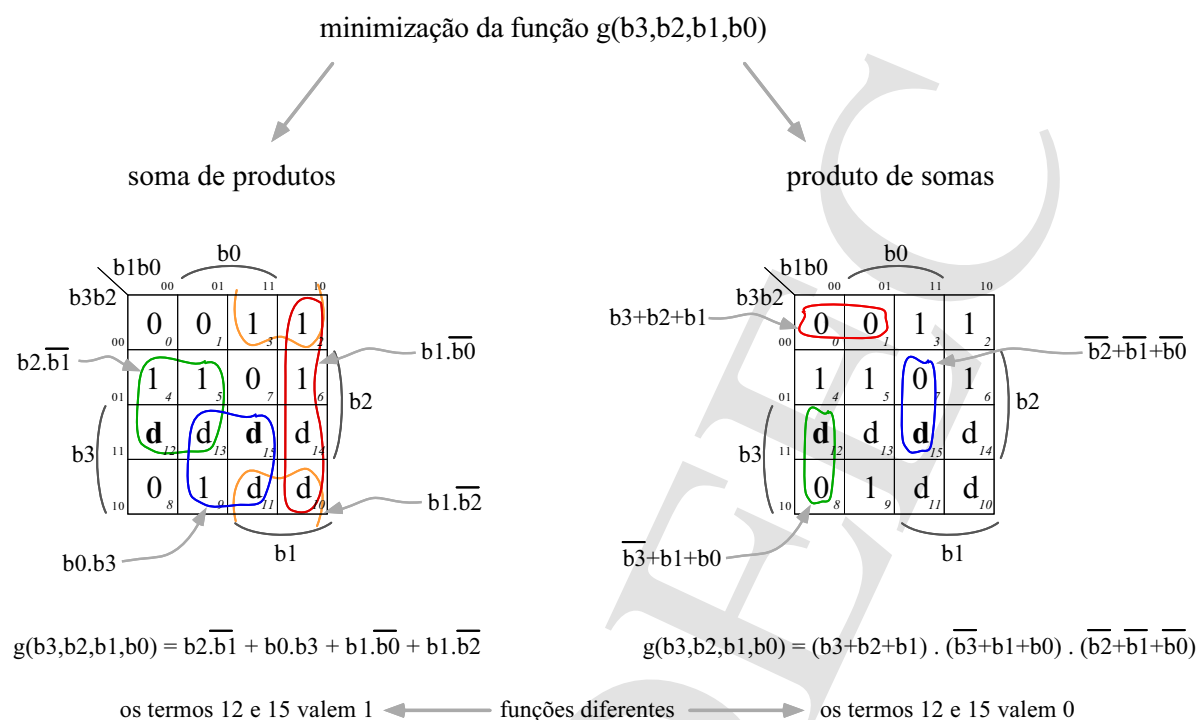


Figura 4.12: Expressões mínimas soma de produtos e produto de somas para a função $g(b_3, b_2, b_1, b_0)$ do decodificador BCD para 7 segmentos.

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.

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 exactamente 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

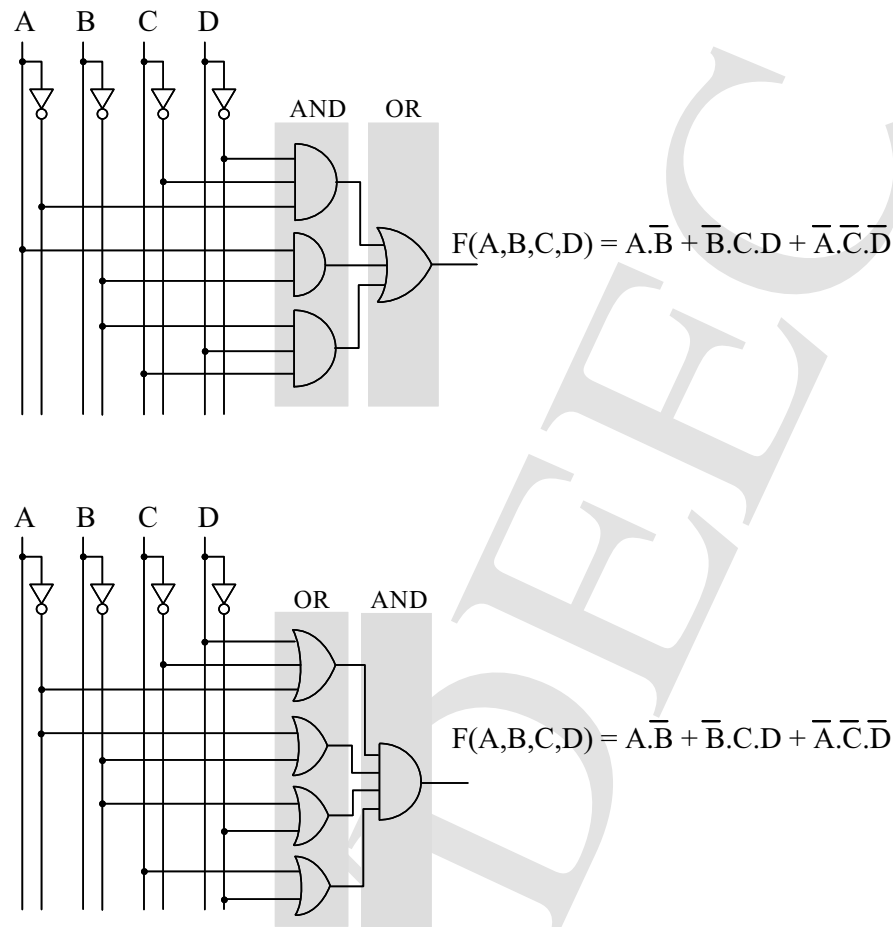


Figura 4.13: Circuitos lógicos AND-OR e OR-AND, correspondentes a expressões do tipo soma de produtos e produto de somas

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).

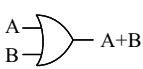
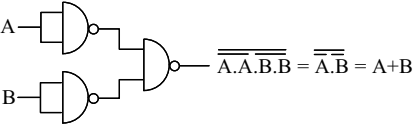
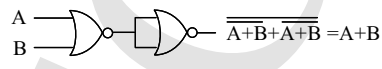
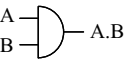
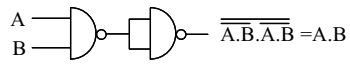
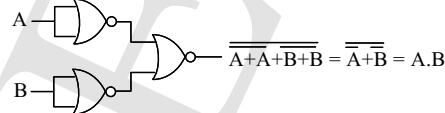
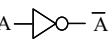
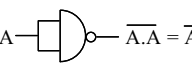
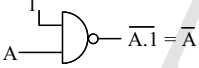
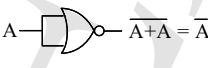
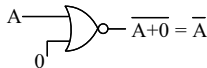
função lógica	realização usando NANDs	realização usando NORs
 $A + B$	 $\overline{\overline{A} \cdot \overline{B}} = \overline{\overline{A+B}} = A+B$	 $\overline{\overline{A+B} + \overline{A+B}} = A+B$
 $A \cdot B$	 $\overline{\overline{A \cdot B}} = A \cdot B$	 $\overline{\overline{A+A} + \overline{B+B}} = \overline{\overline{A+B}} = A \cdot B$
 \overline{A}	 $\overline{A \cdot A} = \overline{A}$  $\overline{A \cdot 1} = \overline{A}$	 $\overline{A+A} = \overline{A}$  $\overline{A+0} = \overline{A}$

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.

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 ($\overline{A \cdot B \cdot C} \neq (\overline{A \cdot B}) \cdot \overline{C}$), 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).

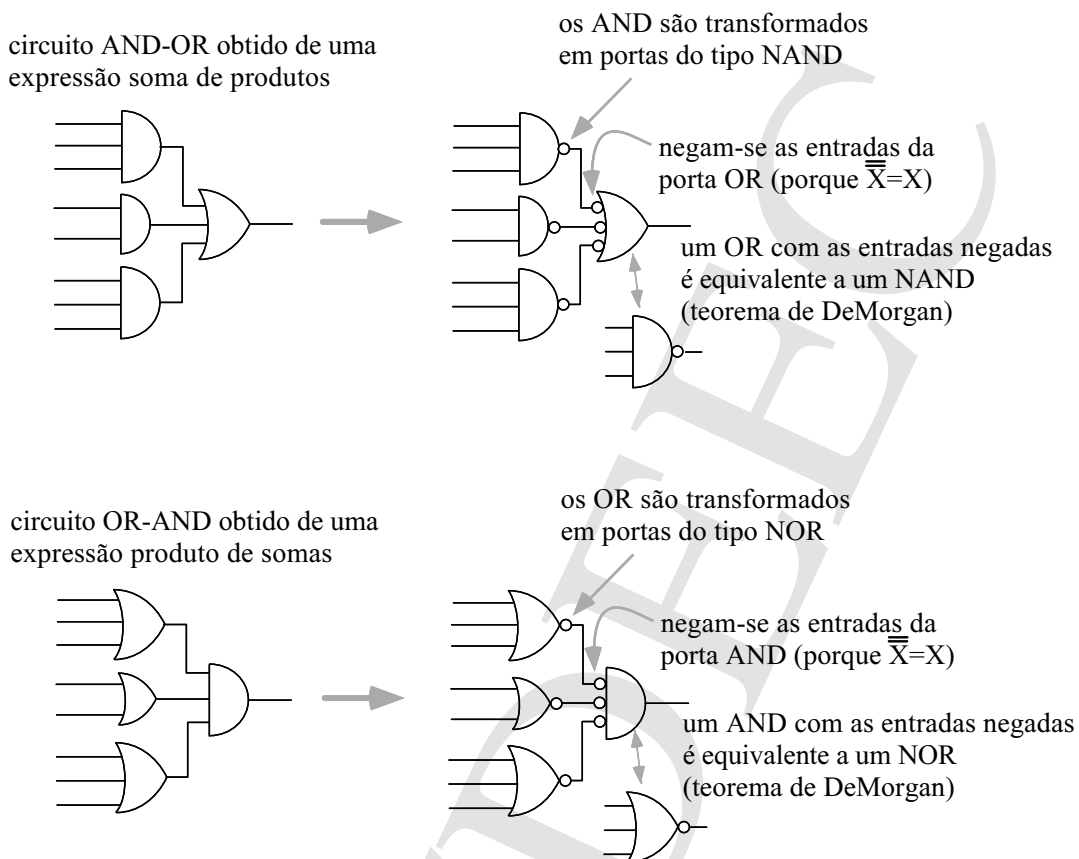


Figura 4.15: Transformação de circuitos AND-OR e OR-AND em circuitos com apenas NANDs e NORs, respectivamente.

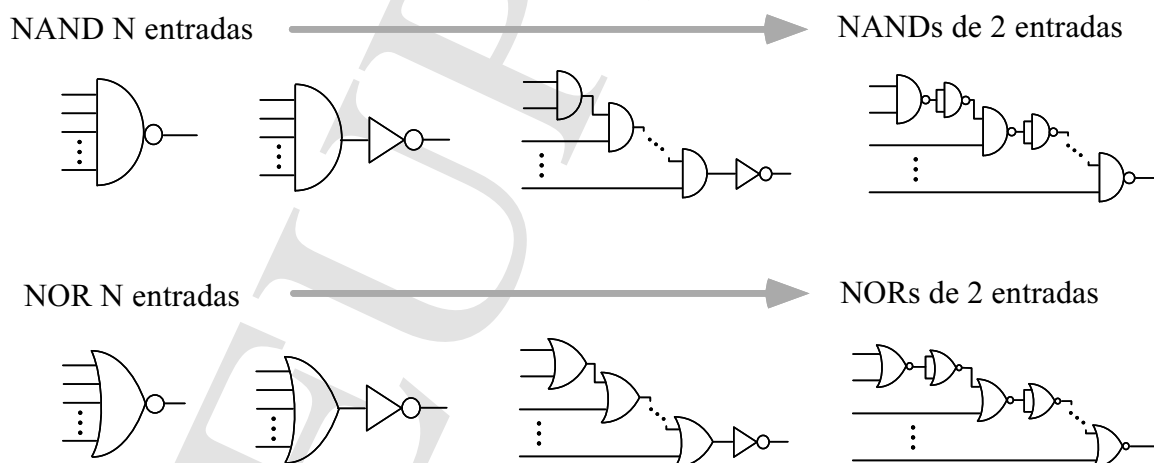


Figura 4.16: Decomposição de portas NAND e NOR com N entradas em portas NAND e NOR com apenas 2 entradas.

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 variáveis¹ 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...

¹Note 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.

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 conquistar. 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 sub-sistemas 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

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.

complexo, por exemplo um microprocessador, é representado à custa da interligação de blocos do tipo “caixa preta” encapsulando funções do nível RTL.

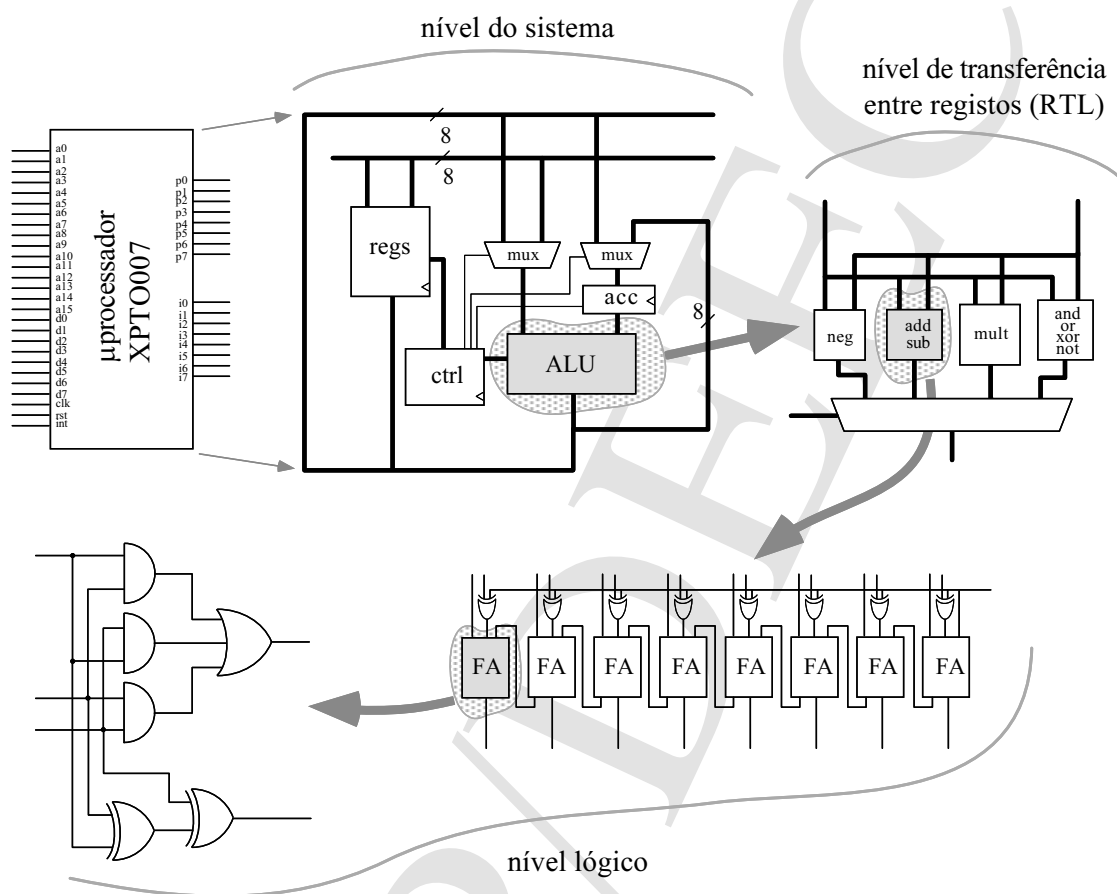


Figura 5.1: Decomposição hierárquica de um sistema digital complexo.

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 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 processador 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 linguagens de descrição de *hardware*.

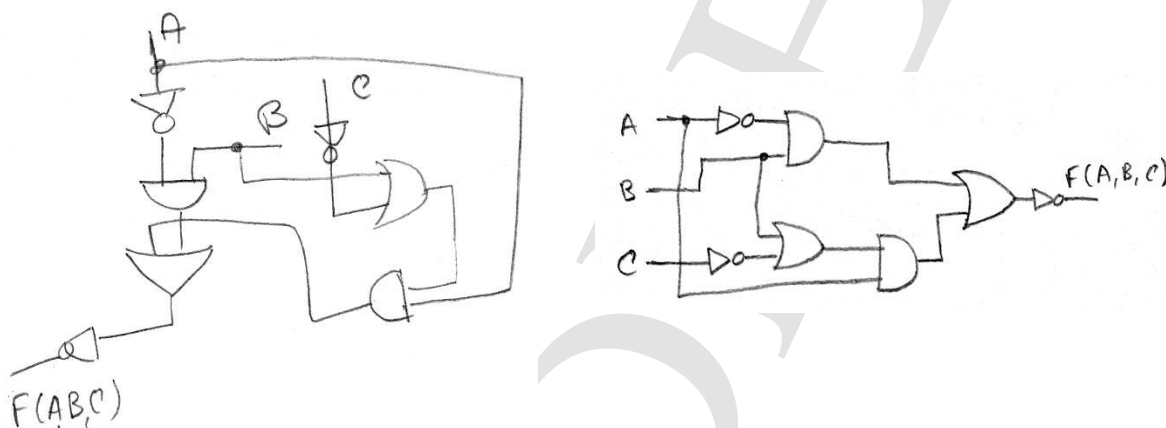


Figura 5.2: Um esquema confuso e um esquema claro.

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 desenhavam 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 carácter deve ser alfabético (por exemplo SEL, OPR_A, I6); em alguns esquemas aparece por vezes o carácter '#' no final de um nome ou o carácter '/' 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 usual³ 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

³Apesar 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

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.

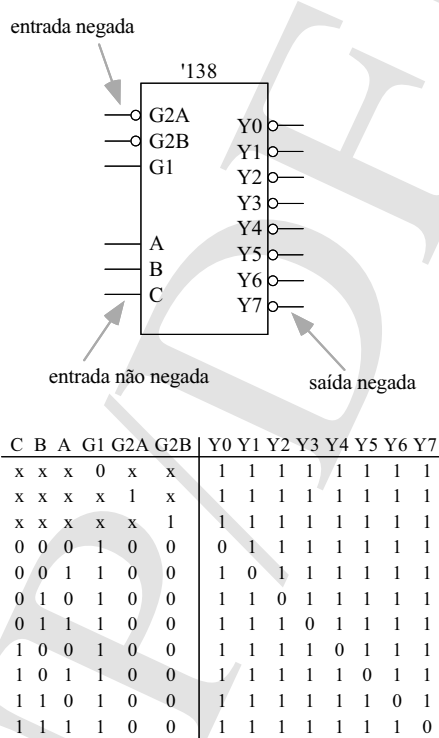
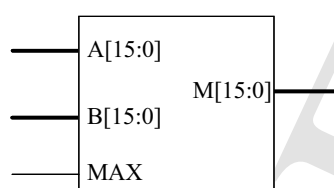


Figura 5.4: Representação do símbolo e da tabela de verdade de um circuito lógico com entradas e saídas negadas (decodificador '138, estudado mais à frente na secção 5.3.3).

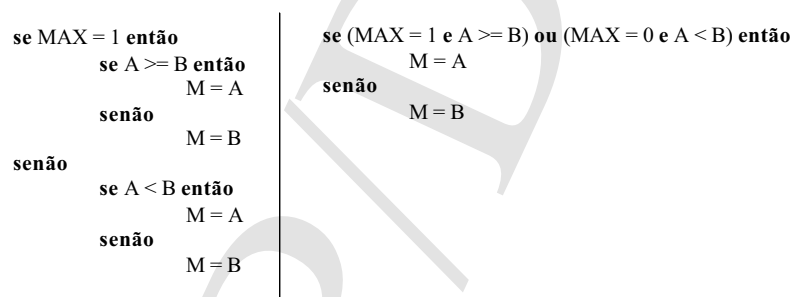
5.2 Um exemplo: calculador do máximo e do mínimo

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.



símbolo lógico do detector de máximo



duas descrições funcionais alternativas

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

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 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 $A \geq B$; se $MAX = 1$ queremos obter na saída o máximo entre A e B: se $A < B$ deve ser escolhido B e se $A \geq B$ 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 $A \geq B$) 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	0	0 (escolhe B)
0	1	1	0	1 (escolhe A)
1	0	1	1	1 (escolhe A)
1	1	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, subtratores 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).

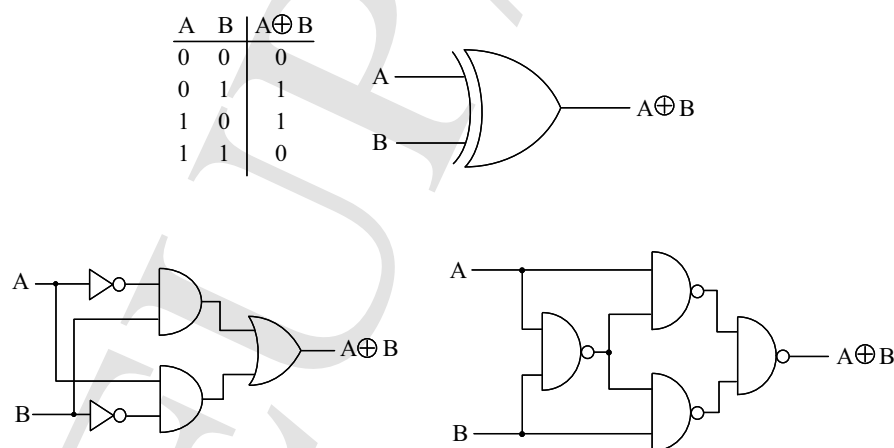


Figura 5.7: Porta lógica OU-exclusivo (XOR): tabela de verdade, símbolo lógico e duas implementações possíveis.

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.

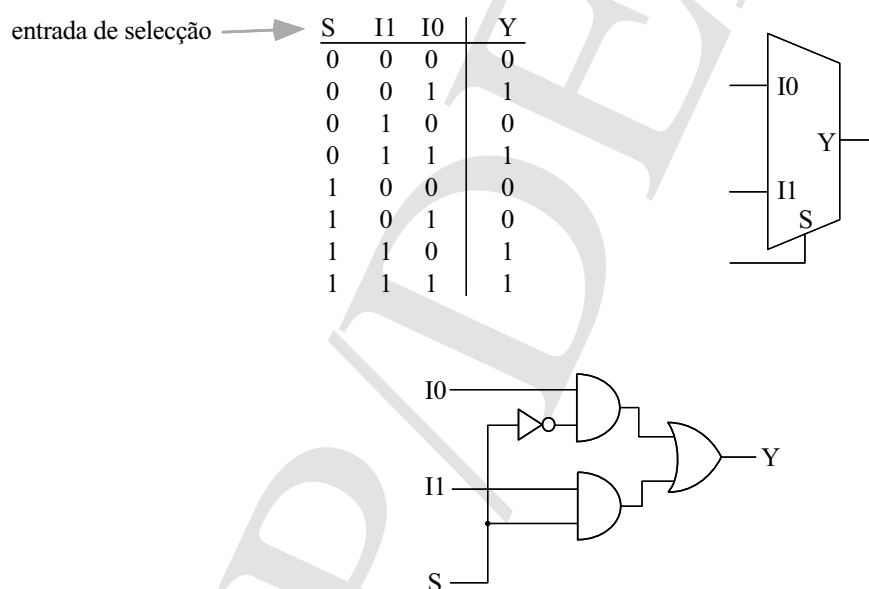


Figura 5.8: Circuito selector de 2 entradas: tabela de verdade, circuito lógico e representação simbólica; a saída Y é igual à entrada $I0$ quando o sinal de selecção é 0, e é igual à entrada $I1$ no caso contrário.

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*

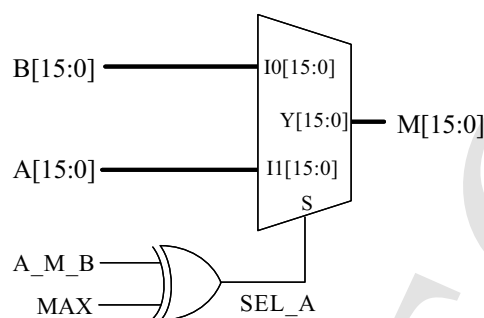


Figura 5.9: Selector do máximo ou mínimo; note que este circuito tem por entrada o sinal A_M_B que representa o resultado da comparação de magnitude entre as entradas A e B e que será estudado a seguir.

magnitude e pode ser 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 dispomos⁴.

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 $A \geq B$ 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

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

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 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,

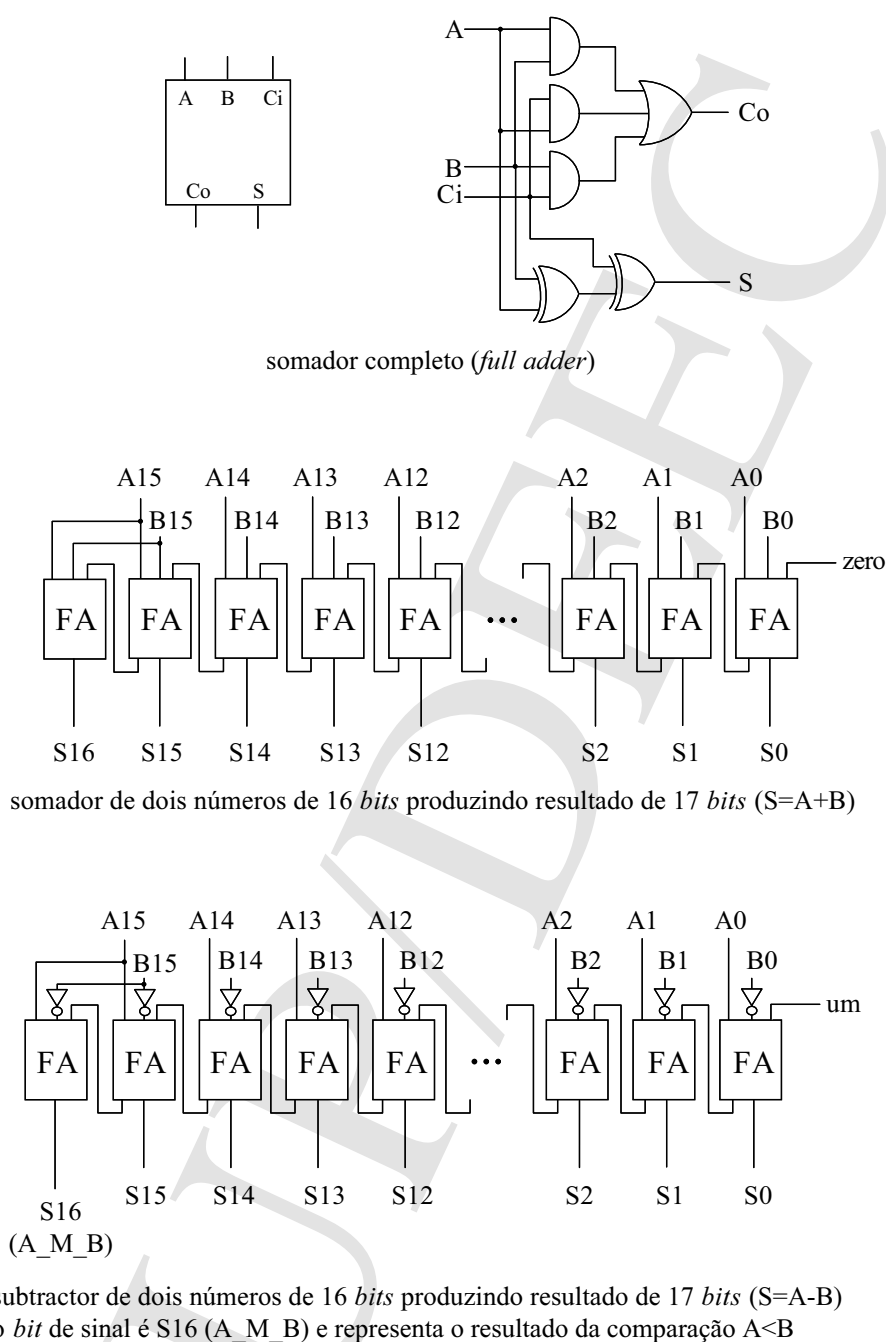


Figura 5.10: Somador completo e circuitos somador e subtrator de 17 bits; note que é feita a extensão de sinal dos operandos A e B copiando os respectivos bits mais significativos, para que se possa obter um resultado de 17 bits em que nunca ocorre *overflow*.

bom como já não se pode chamar *full adder* a cada um dos seus circuitos elementares (figura 5.11).

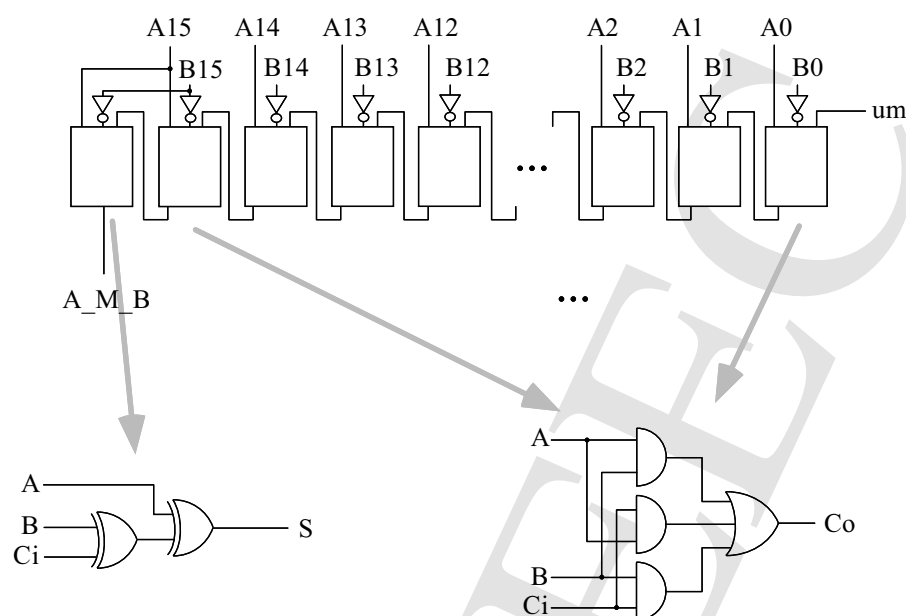


Figura 5.11: Simplificação do subtrator num comparador de magnitude.

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).

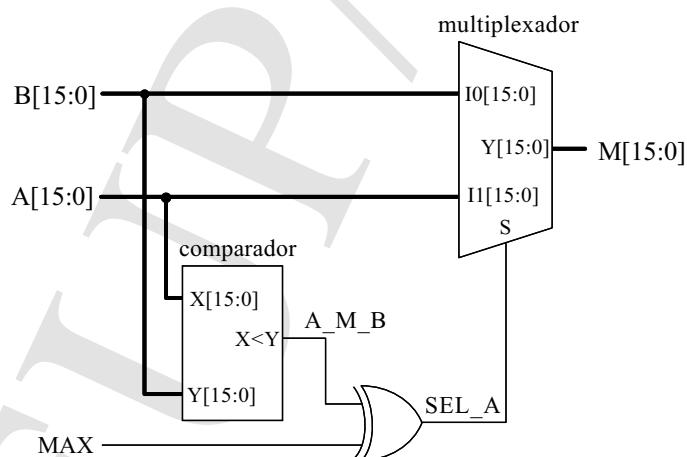


Figura 5.12: Realização do circuito detector de máximo e mínimo recorrendo a funções padrão: um comparador de magnitude, um multiplexador e uma porta XOR.

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. Al-

gumas 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 decodificadores** são circuitos que, de uma forma geral, traduzem um código binário noutra código com um número diferente de *bits*.
- **Multiplexadores:** a função de selecção já foi introduzida, mas serão apresentadas outras variantes e aplicações.
- **Circuitos aritméticos:** para além do somador, subtrator 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 decodificadores

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

Um exemplo de decodificador foi já apresentado no capítulo 4, secção 4.2.4: decodificador de BCD para 7 segmentos. Este circuito transforma um código de 4 *bits*, representado um dígito decimal entre 0 e 9, noutra 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).

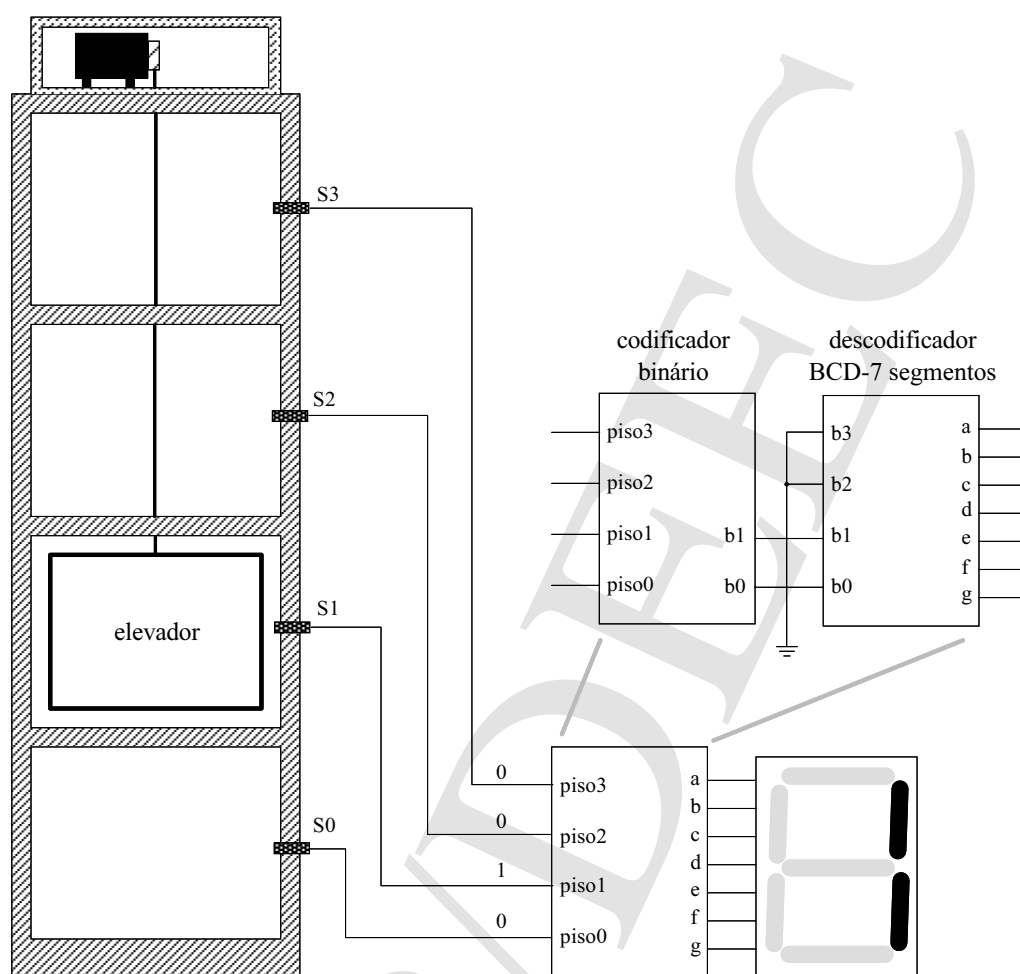


Figura 5.13: Um elevador num edifício com R/C e 3 andares.

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 decodificador 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.

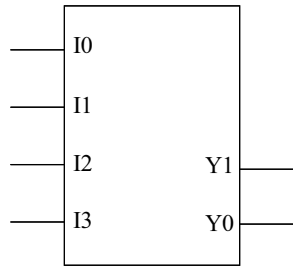
	I3	I2	I1	I0	Y1	Y0	
	0	0	0	0	x	x	$Y1 = I3 + I2$
	0	0	0	1	0	0	
	0	0	1	0	0	1	$Y0 = I3 + I1$
	0	0	1	1	x	x	
	0	1	0	0	1	0	
	0	1	0	1	x	x	
	0	1	1	0	x	x	
	0	1	1	1	x	x	
	1	0	0	0	1	1	
	1	0	0	1	x	x	
	1	0	1	0	x	x	
	1	0	1	1	x	x	
	1	1	0	0	x	x	
	1	1	0	1	x	x	
	1	1	1	0	x	x	
	1	1	1	1	x	x	

Figura 5.14: Codificador binário de 4 para 2: tabela de verdade e implementação lógica otimizada. 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 decodificador 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 activado⁵. Embora só por si esta modificação não resolva o problema, o decodificador 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 decodificador 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 entanto, 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

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

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 *C_i* 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 decodificadores de prioridade com

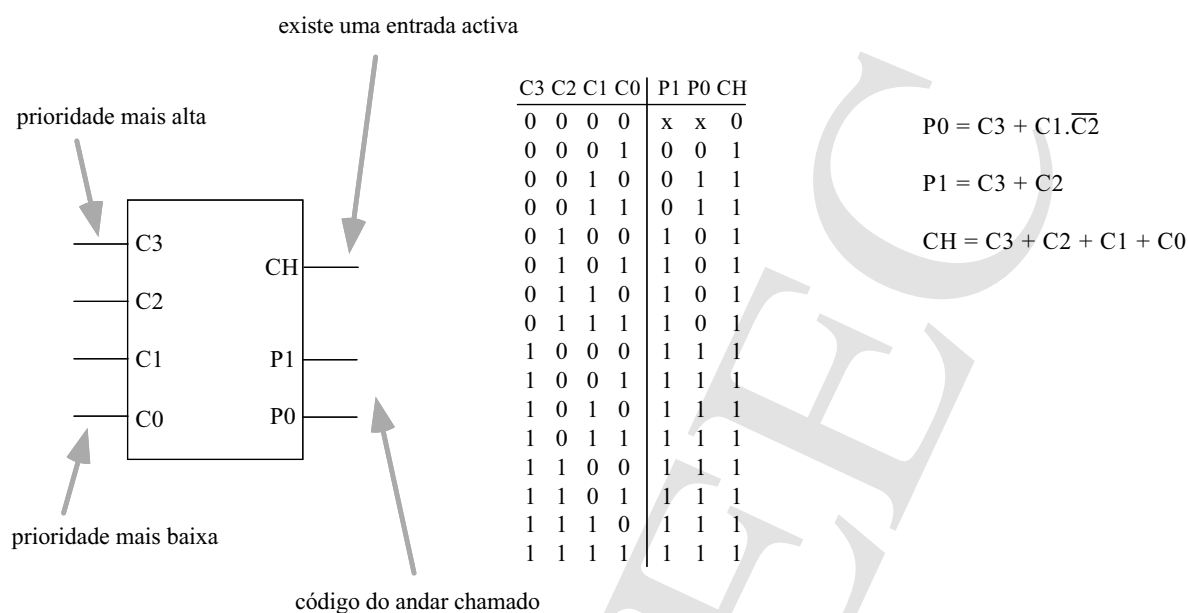


Figura 5.15: Codificação do andar chamado de um elevador com um codificador de prioridade de 4 *bits*. A activação da saída CH significa que alguma das entradas foi activada e só neste caso é que as saídas P1 e P0 fazem sentido e contêm o código da entrada activada.

maior número de entradas usando este tipo de componente.

5.3.3 Descodificadores binários

Um decodificador 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 decodificador 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 decodificadores de um número maior de *bits*.

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

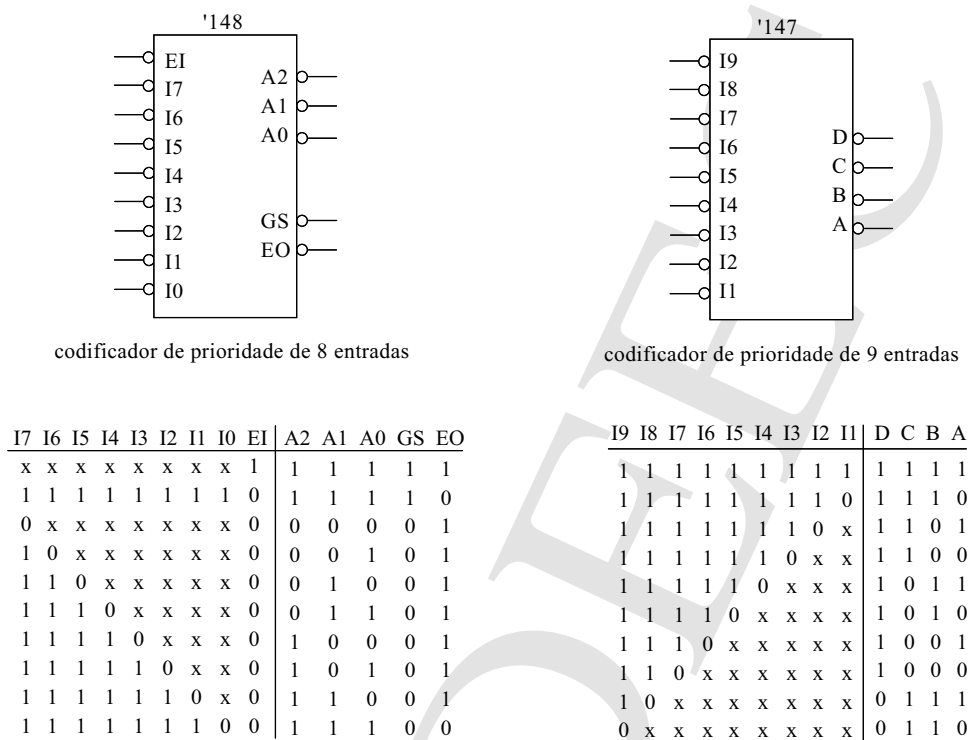


Figura 5.16: Codificadores de prioridade em circuitos integrados da série: '148 e '147.

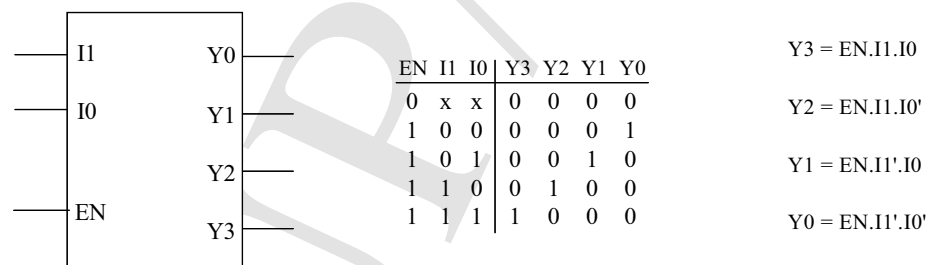


Figura 5.17: Descodificador binário de 2 *bits* para 4 *bits*.

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 decodificador 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 decodificador. Assim, para implementar qualquer função lógica

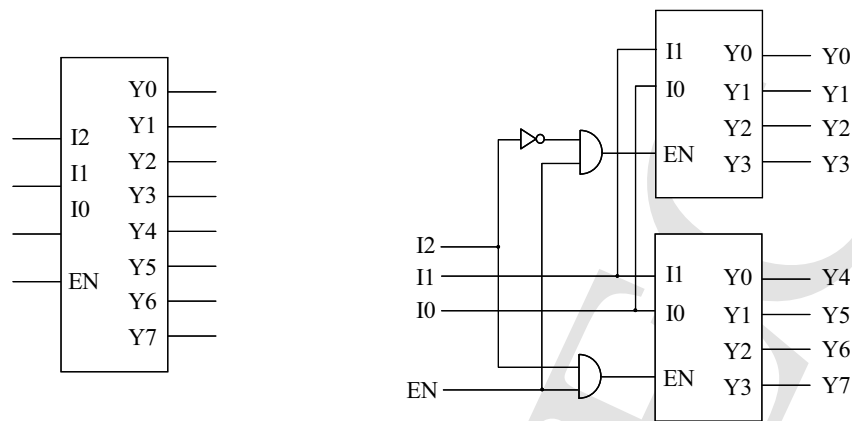


Figura 5.18: Decodificador de 3 para 8 *bits* realizado com dois decodificadores de 2 para 4 *bits*.

de 2 variáveis com base neste decodificador, basta ligar as saídas Y_i 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 Y_0 , Y_2 e Y_3 do decodificador 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 Y_1 do decodificador (figura 5.19).

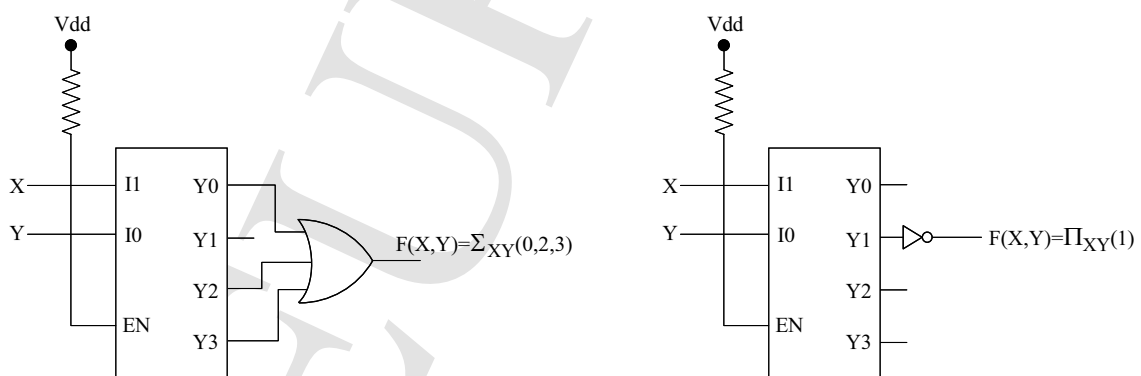


Figura 5.19: Realização de uma função lógica de 2 variáveis com base num decodificador.

Os decodificadores '138 e '139

A função decodificador 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 decodificadores mais complexos seguindo o processo que foi mostrado na figura 5.18, mas sem ser necessário recorrer à utilização de circuitos lógicos adicionais.

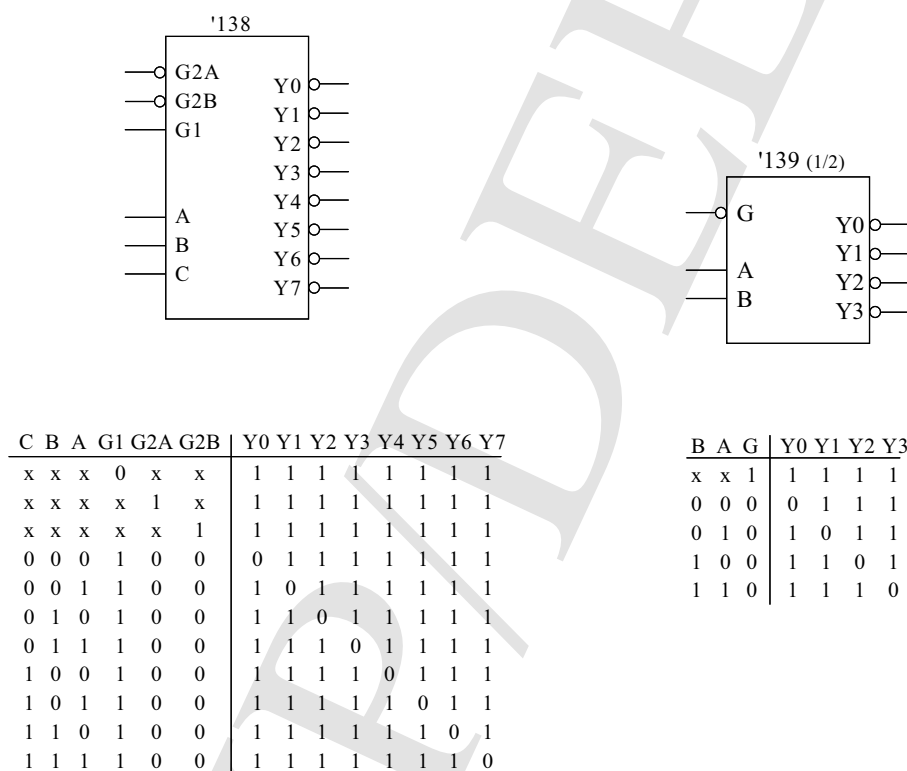


Figura 5.20: Decodificadores '138 e '139. Note que o circuito integrado '139 possui dois decodificadores de 2 para 4 *bits*.

5.3.4 Decodificadores para mostradores de 7 segmentos

Outra categoria de decodificadores são os decodificadores para mostradores de 7 segmentos, tal como o que foi referido atrás na secção 5.3.1. Um decodificador 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 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.

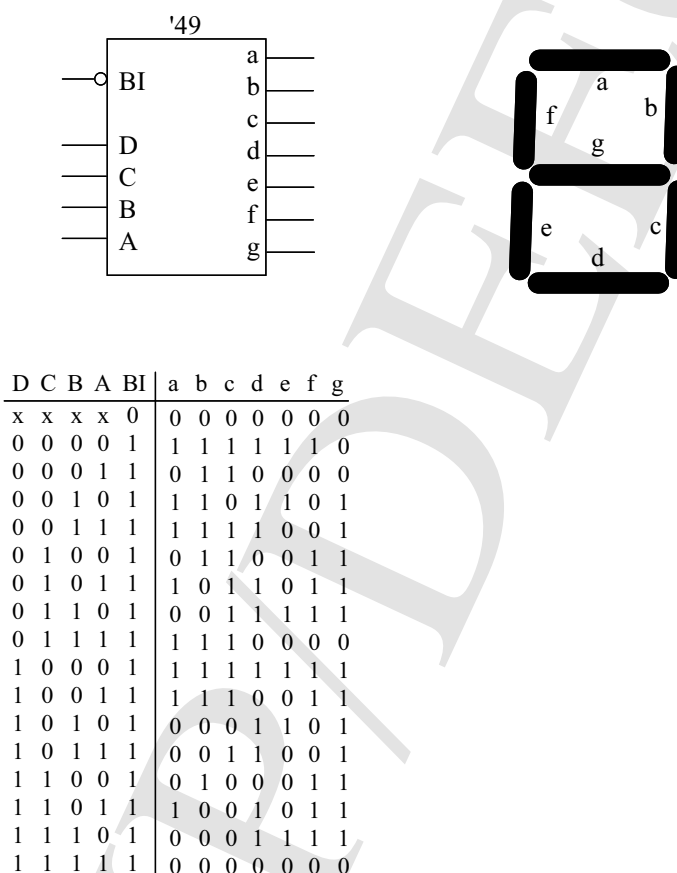


Figura 5.21: Decodificador BCD para 7 segmentos: '49.

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 decodificador.

Decodificadores hexadecimal para 7 segmentos

Um decodificador 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 decodificadores 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 5.22⁶. Este decodificador possui duas saídas que não existem nos decodificadores 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 decodificação 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.

Mostradores de 7 segmentos

Existem várias outras versões deste tipo de decodificador 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 si⁷.

Para ligar um decodificador de 7 segmentos a um mostrador de 7 segmentos deve-se escolher um mostrador de cátodo comum se o decodificador tiver as saídas activas com o nível lógico alto, e um mostrador de ânodo comum quando o decodificador 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

⁶Este 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.

⁷Para 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.

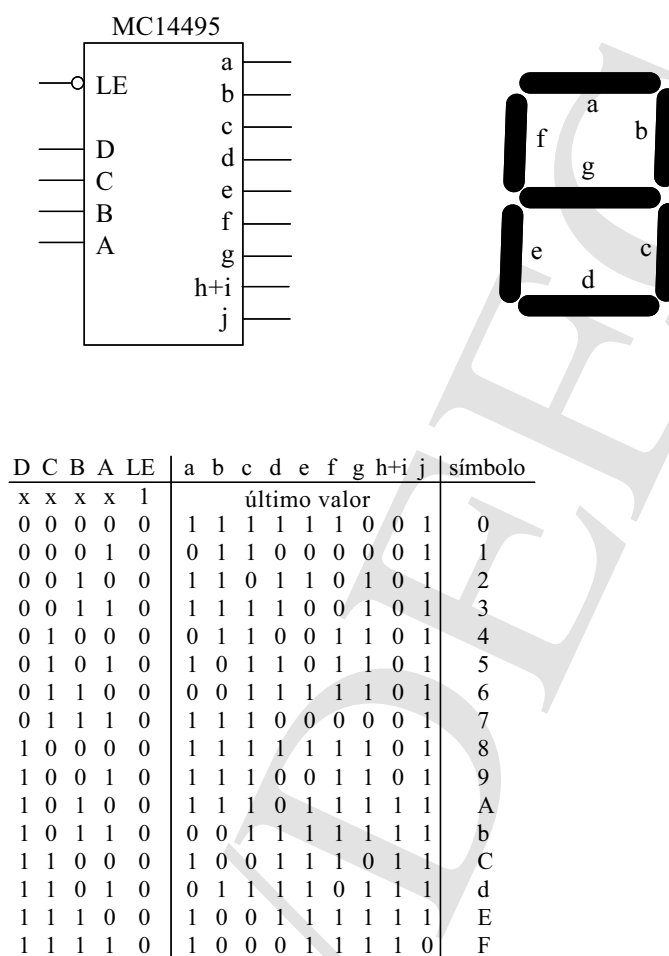


Figura 5.22: Decodificador hexadecimal para 7 segmentos MC14495. Note que a disposição dos LEDs do mostrador obriga a representar os símbolos A, C, E e F como caracteres maiúsculos e os símbolos b e d como caracteres minúsculos.

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 decodificação, as saídas do decodificador 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 decodificadores 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.

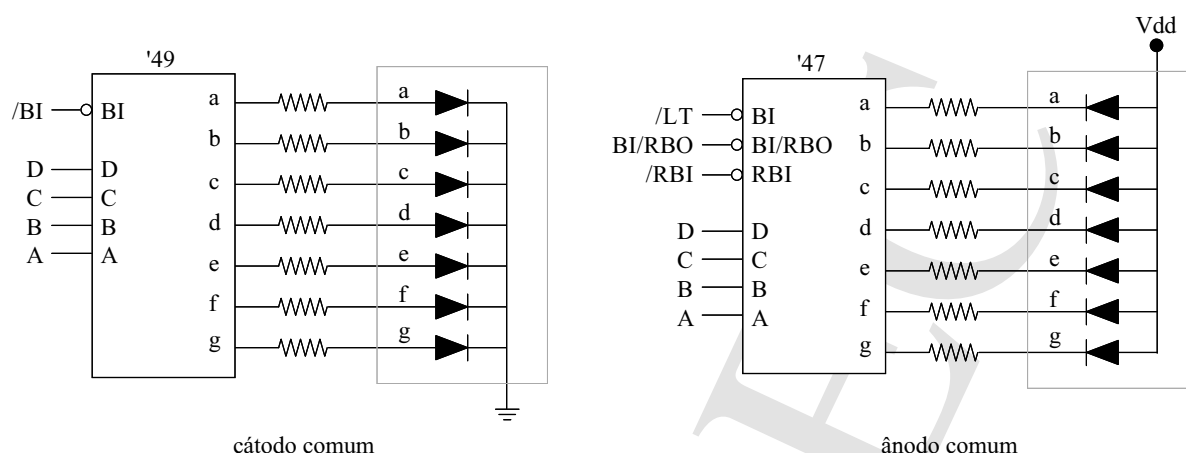


Figura 5.23: Mostradores de 7 segmentos com ânodo comum e cátodo comum, e a sua ligação às saídas de decodificadores. Note que o circuito integrado MC14495 que tem saídas activas no nível lógico alto, possui já internamente as resistências mostradas na figura.

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.

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

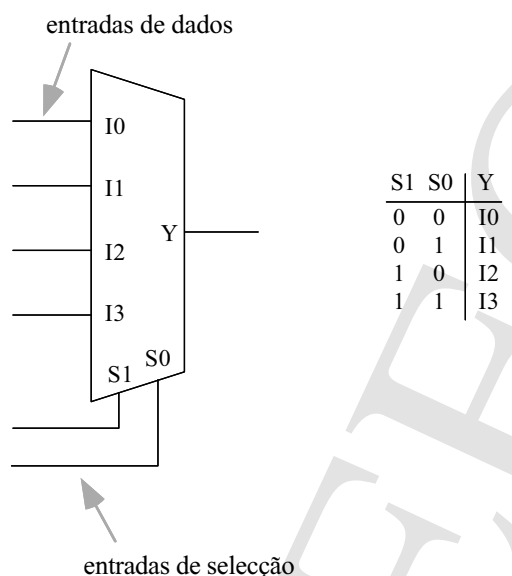


Figura 5.24: Multiplexador com 2 entradas de selecção e 4 entradas de dados: símbolo lógico e tabela de verdade.

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.

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.

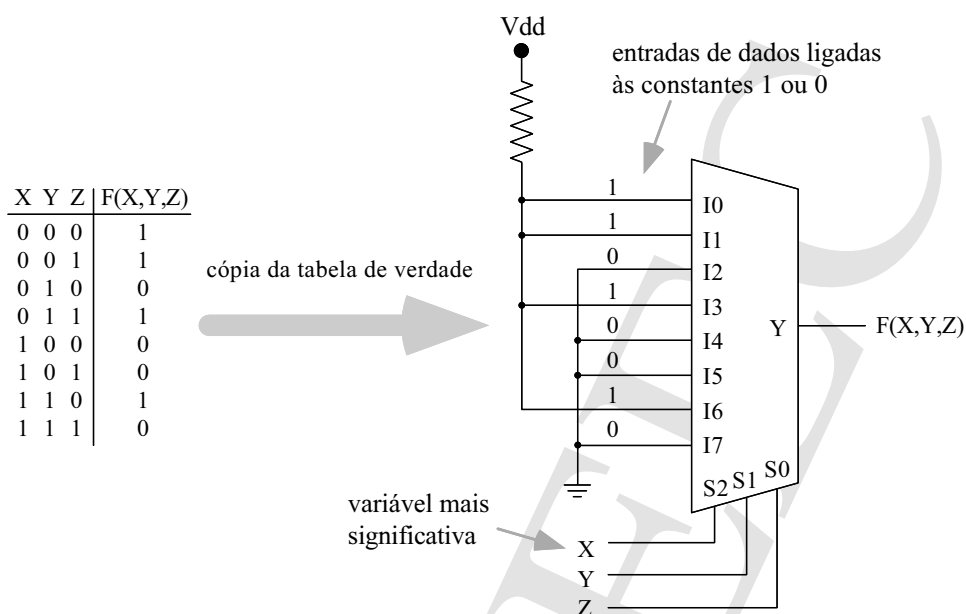


Figura 5.25: Implementação de uma função lógica de 3 variáveis usando um multiplexador com 3 entradas de selecção.

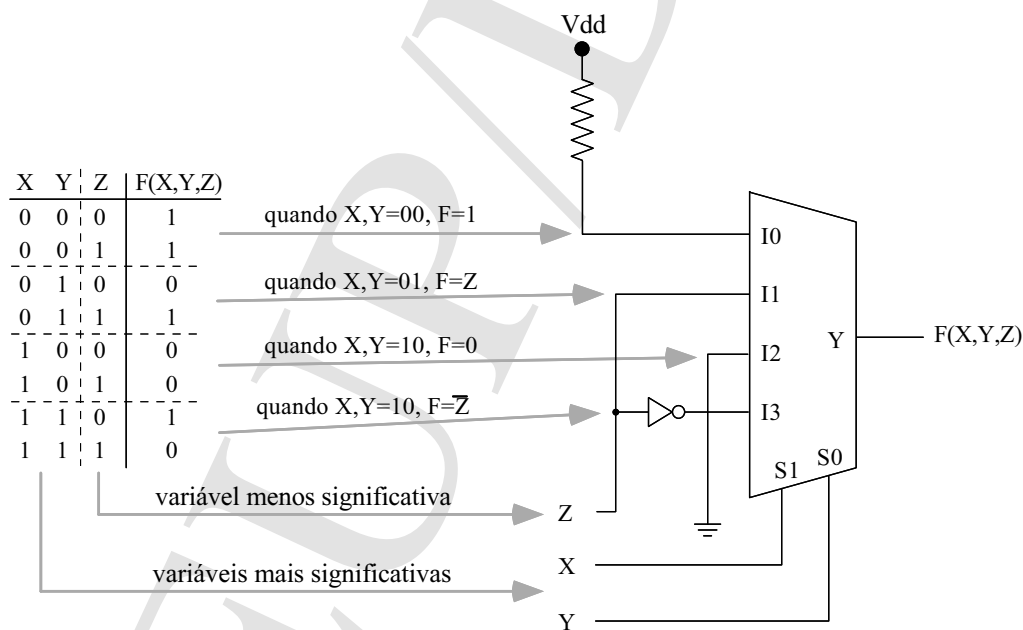


Figura 5.26: Implementação de uma função lógica com 3 variáveis usando um multiplexador com apenas 2 entradas de selecção.

5.5 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

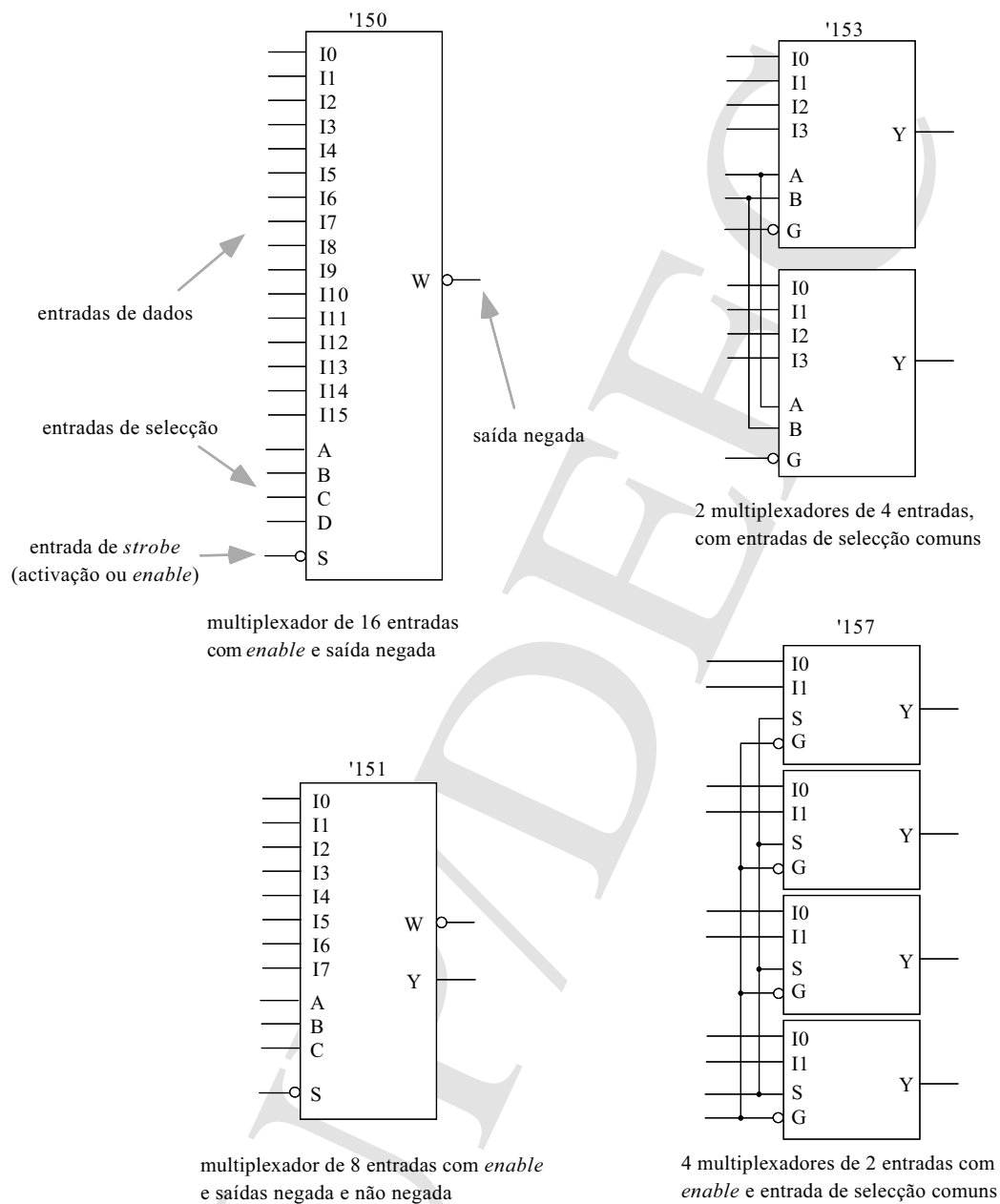


Figura 5.27: Circuitos integrados da série 74 com a função multiplexador.

categoria circuitos muito complexos que realizem operações com números representados em vírgula flutuante ou calculam funções transcendentais (funções trigonométricas, 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 bits⁸. 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 ($A_0 = B_0, A_1 = B_1, \dots, A_{N-1} = B_{N-1}$). O circuito que compara dois bits entre si é realizado pela função lógica XOR apresentada na figura 5.7, página 87, 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).

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

⁸Embora 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.

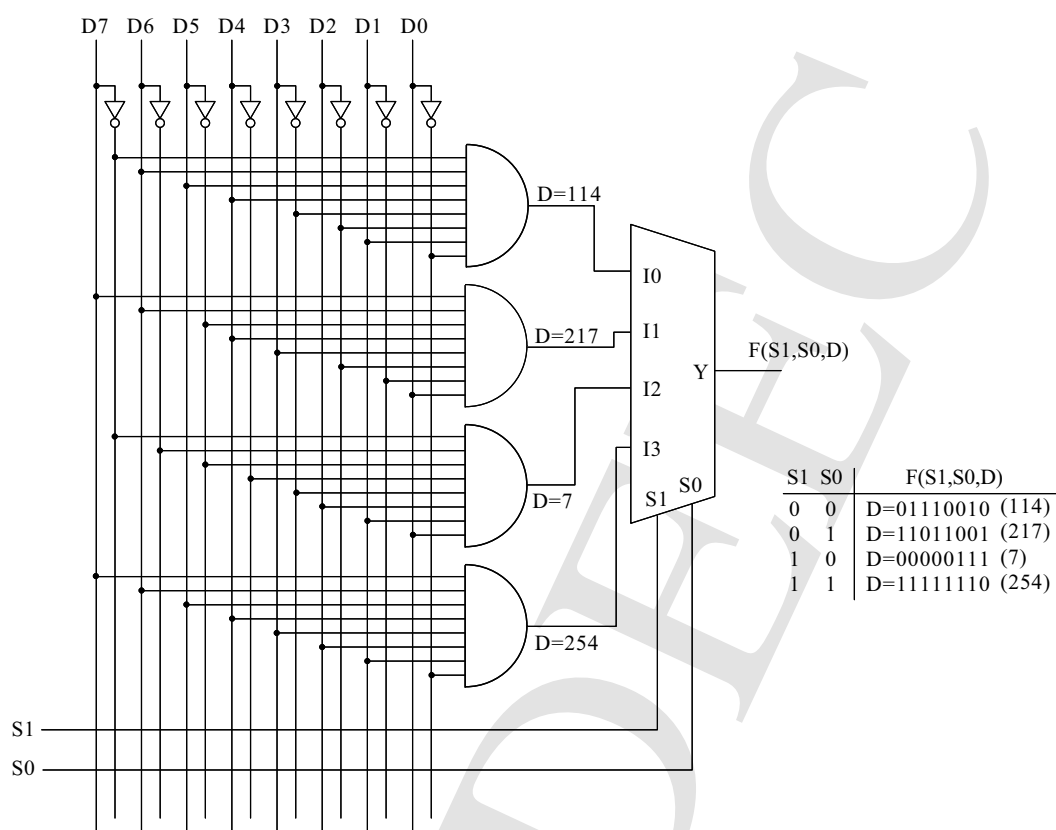


Figura 5.29: Um comparador com 4 constantes diferentes usando um multiplexador.

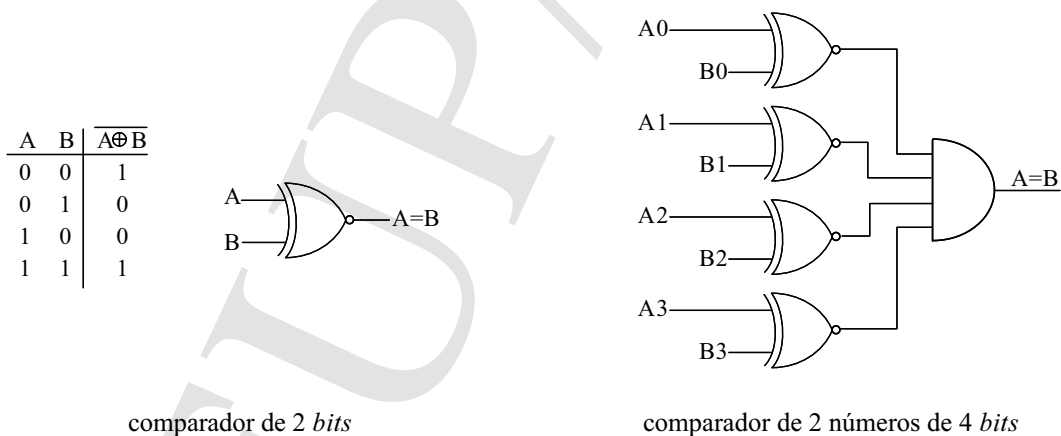


Figura 5.30: Uma porta lógica XNOR (XOR com a saída negada) como um comparador de igualdade de 2 bits e um comparador de igualdade entre dois números com 4 bits.

circuito subtrator. 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 sinal, então esse transporte é igual a 1 e significa que é verdade que $A \geq B$; 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 subtrator 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 subtrator. O circuito lógico deste comparador é muito semelhante ao do comparador de magnitude para números com sinal e é apresentado na figura 5.31.

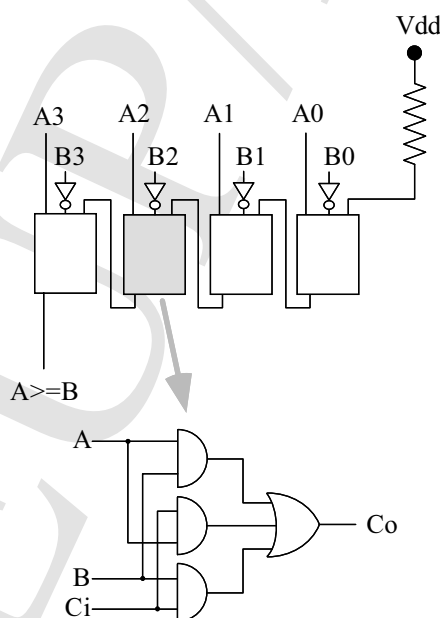


Figura 5.31: Comparador de magnitude para números sem sinal.

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

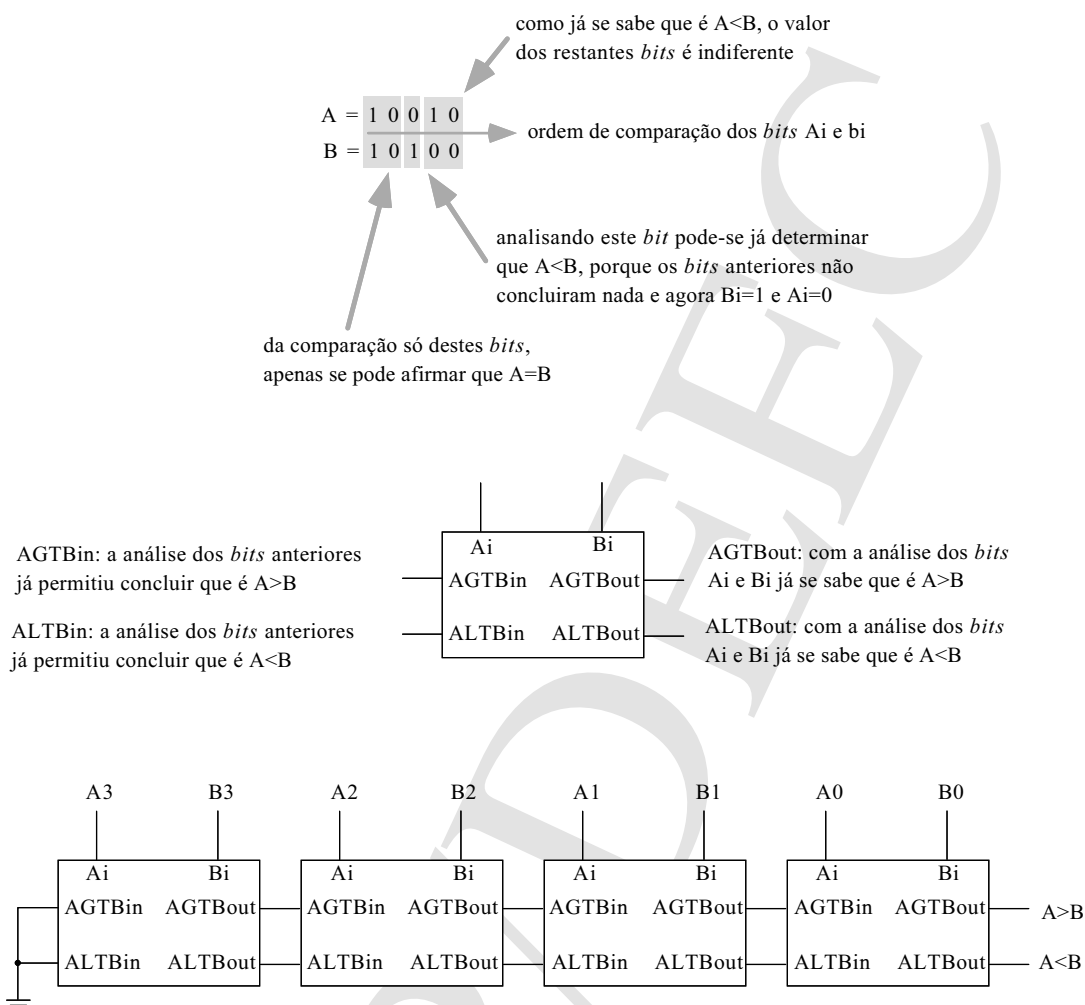


Figura 5.32: Comparador de magnitude para números sem sinal, implementado como um circuito iterativo.

conjunto de entradas e saídas que facilitam a construção de comparadores entre números de qualquer 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 substractores

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

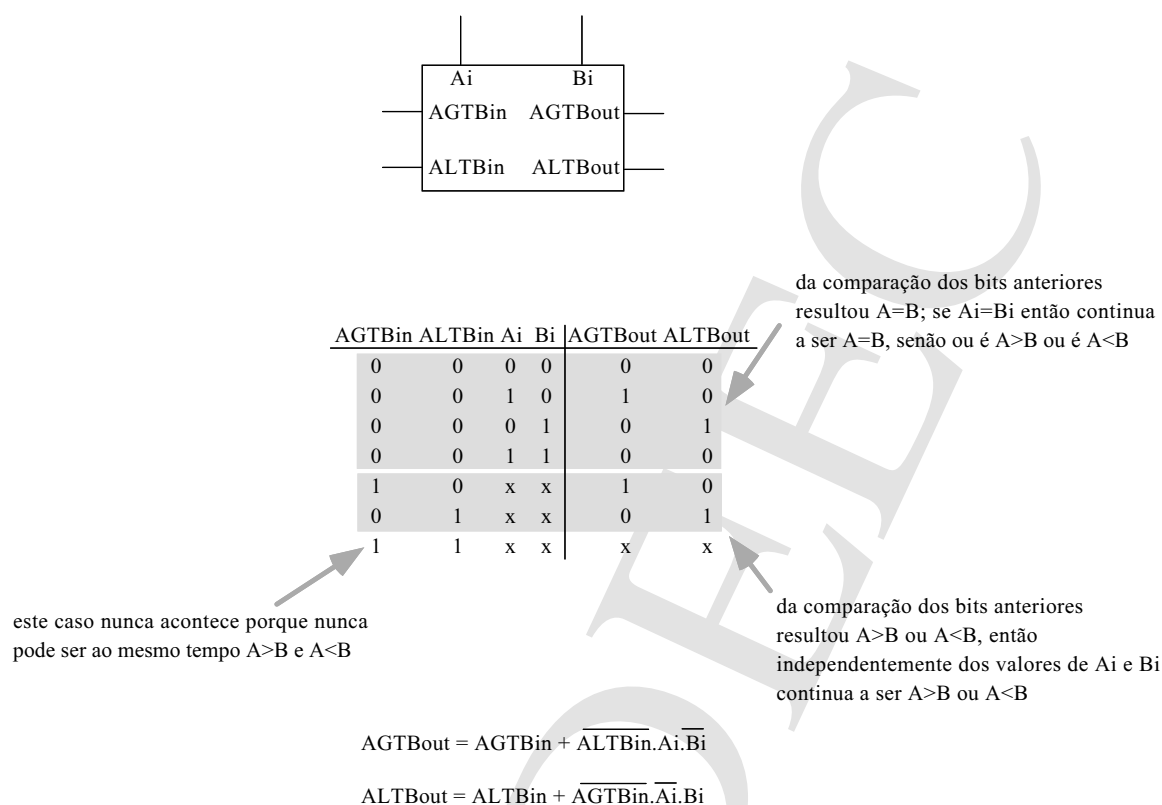


Figura 5.33: Implementação do bloco comparador usado no circuito da figura 5.32

de *bits*⁹.

Circuito somador ou subtrator

Como um somador e um subtrator diferem entre si em muito pouco, torna-se interessante conseguir reunir num 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).

⁹Mas 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 a produzir o resultado.

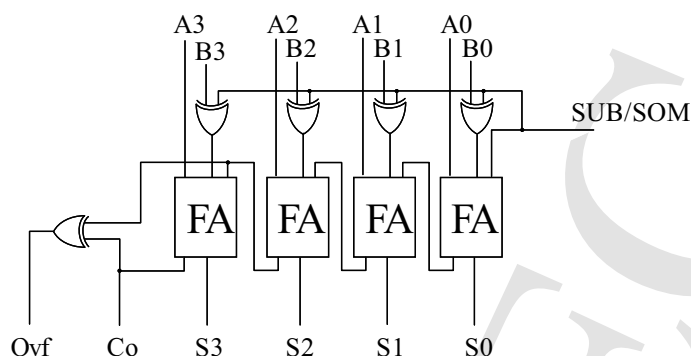


Figura 5.34: Somador/subtrator iterativo (ou *ripple carry*). Note que a detecção de ocorrência de *overflow* em adições com operandos em complemento para 2 pode ser obtida usando apenas uma porta XOR adicional (saída Ovf).

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 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, subtratores e comparadores já estudados, outra função importante é a multiplicação, que é realizada à custa da associação de somadores ou de subtratores. 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 com 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.

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 estendida 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).

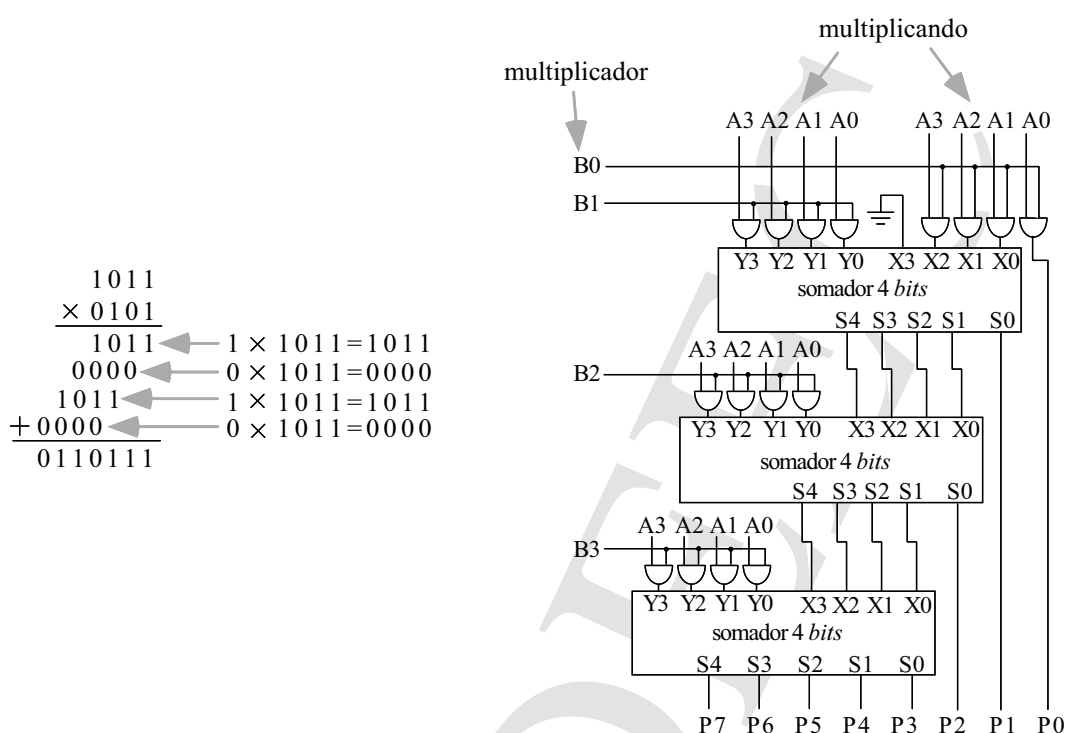


Figura 5.35: Um multiplicador entre 2 números de 4 *bits*.

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.

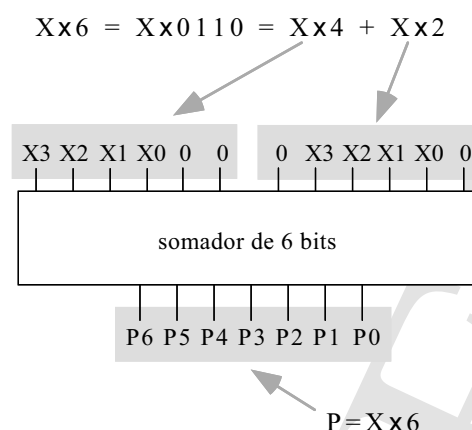


Figura 5.36: Um multiplicador entre um número de 4 *bits* e a constante 6 (0110₂).

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 sinal ou com sinal, e nesse caso se a convenção de representação é sinal 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 sinal em complemento para 2.

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 sinal 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*.

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,

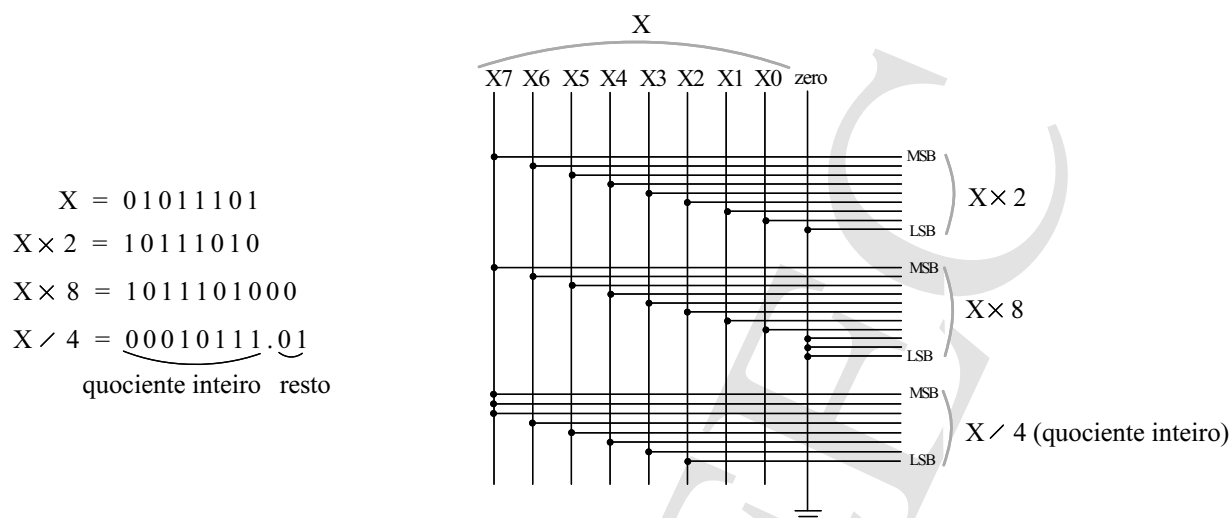


Figura 5.37: Multiplicações e divisões por constantes iguais a potências inteiras de 2 são realizadas apenas à custa da selecção apropriada dos *bits* do operando. Note que o número de *bits* dos produtos é maior do que a dimensão do operando X , para garantir que o resultado pode ser correctamente representado.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

OR ou XOR¹⁰. 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, subtrator, multiplicador, inversores, etc), e usar um multiplexador para escolher um dos vários resultados produzidos. No entanto, como foi já visto para a função somador/subtrator, é mais económico "fundir" num mesmo circuito lógico as várias funções que 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*.

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

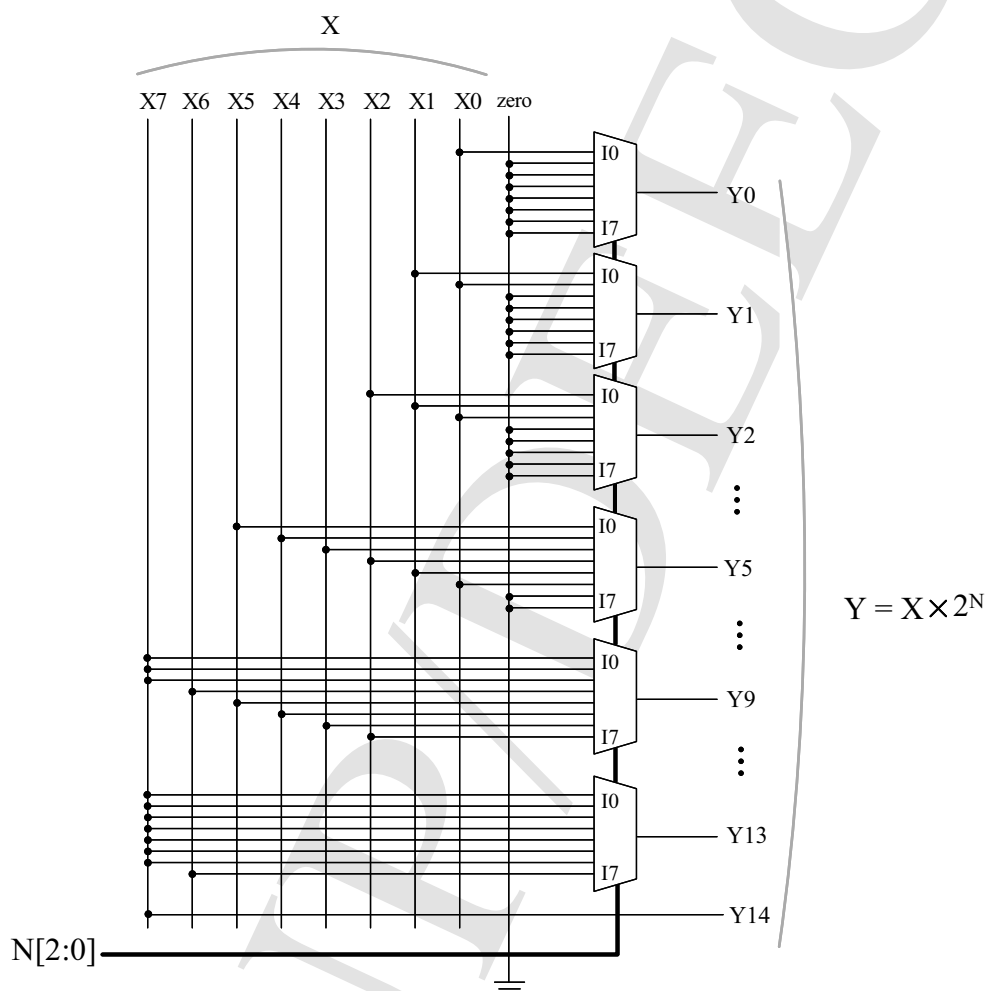


Figura 5.38: Um *barrel shifter* realizado com multiplexadores. Este circuito permite obter os resultados $X \times 2^N$, para qualquer N entre 0 e 7, considerando que o operando X é um valor com sinal em complemento para 2. O circuito pode ser facilmente expandido para qualquer dimensão do operando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

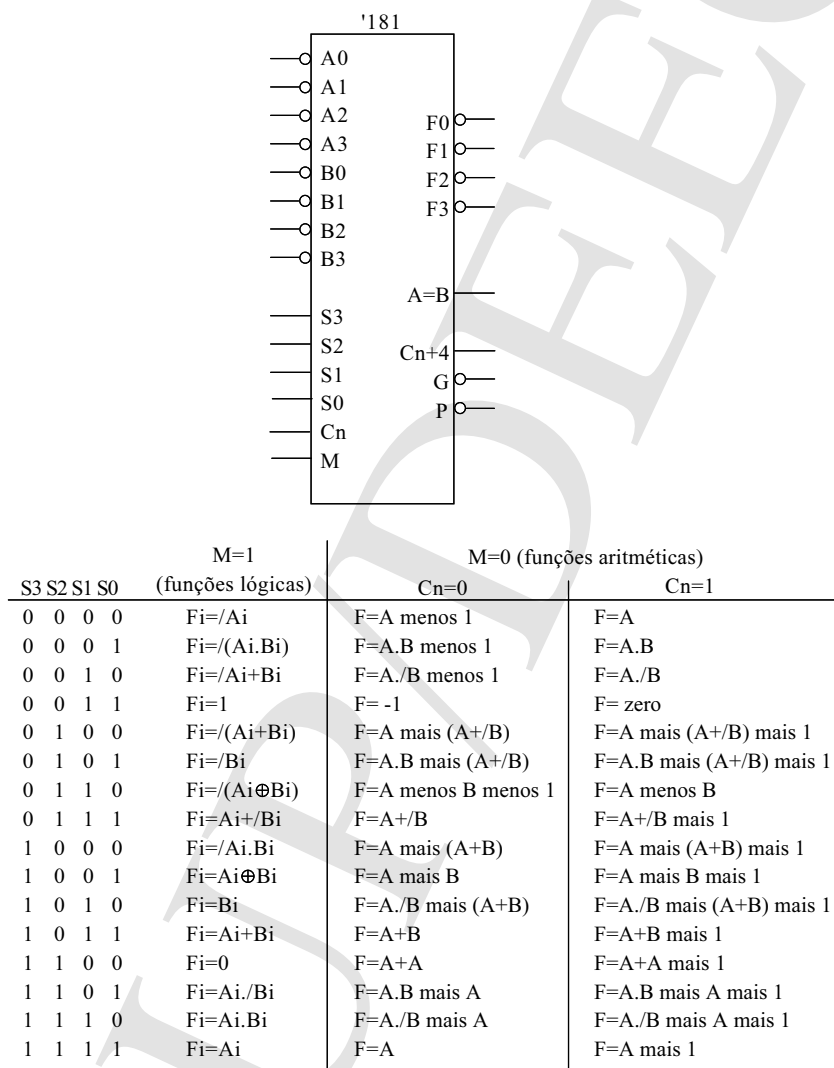


Figura 5.39: Uma unidade lógica e aritmética (ALU)—'181.

Capítulo 6

Circuitos sequenciais

Nos capítulos anteriores foi estudada uma categoria de sistemas digitais a que se chamou combinacional, e que apresenta a particularidade de as suas saídas dependerem apenas dos valores presentes nas suas entradas em cada instante. Esta característica permite que a funcionalidade de um circuito deste tipo possa ser formalmente representada como uma função booleana, nas mais variadas formas que foram estudadas. Por esta razão, não interessa quais foram os valores colocados anteriormente nas entradas e quais foram os valores que ocorreram antes nas saídas, porque o resultado que aparece nas saídas para uma determinada *combinação* de zeros e uns presentes nas entradas é sempre o mesmo. Do ponto de vista do circuito lógico, esta propriedade implica que não exista qualquer dependência entre uma saída e ela própria, ou seja que não haja *realimentação*.

Em circuitos sequenciais o valor das saídas não num determinado instante depende não só das entradas presentes nesse instante mas também da sequência de valores que lá foram colocados em instantes anteriores de tempo. Esta característica resulta do facto de que um circuito sequencial tem alguma forma de memória que lhe permite manter informação relevante das entradas anteriores para determinar os valores das saídas.

Para ilustrar o conceito de circuito sequencial, relembremos o exemplo do sistema digital para controlar o nível de água num depósito, apresentado no capítulo 1 (figura 6.1). Este sistema tem duas entradas provenientes dos sensores que detectam o nível de água e uma saída que comanda a electro-válvula. Considere agora que, para permitir o controlo desse sistema por uma pessoa, essas entradas estão ligadas a duas lâmpadas num painel de controlo, e que a electro-válvula é comandada por meio de um vulgar interruptor. Assim, quando a lâmpada CHEIO estiver acesa (e a lâmpada VAZIO tem de estar também acesa), isso significa que a água ultrapassou o nível máximo e quando ambas as lâmpadas estiverem apagadas o nível de água encontra-se abaixo do mínimo. No primeiro caso deve ser desligado o interruptor que liga a válvula, e no segundo caso a válvula deve ser

ligada. Imagine que este sistema é operado por uma pessoa que decide quando deve ligar e desligar a válvula, “olhando” para os dois *bits* de informação transmitidos pelas lâmpadas que estão ligadas aos sensores.

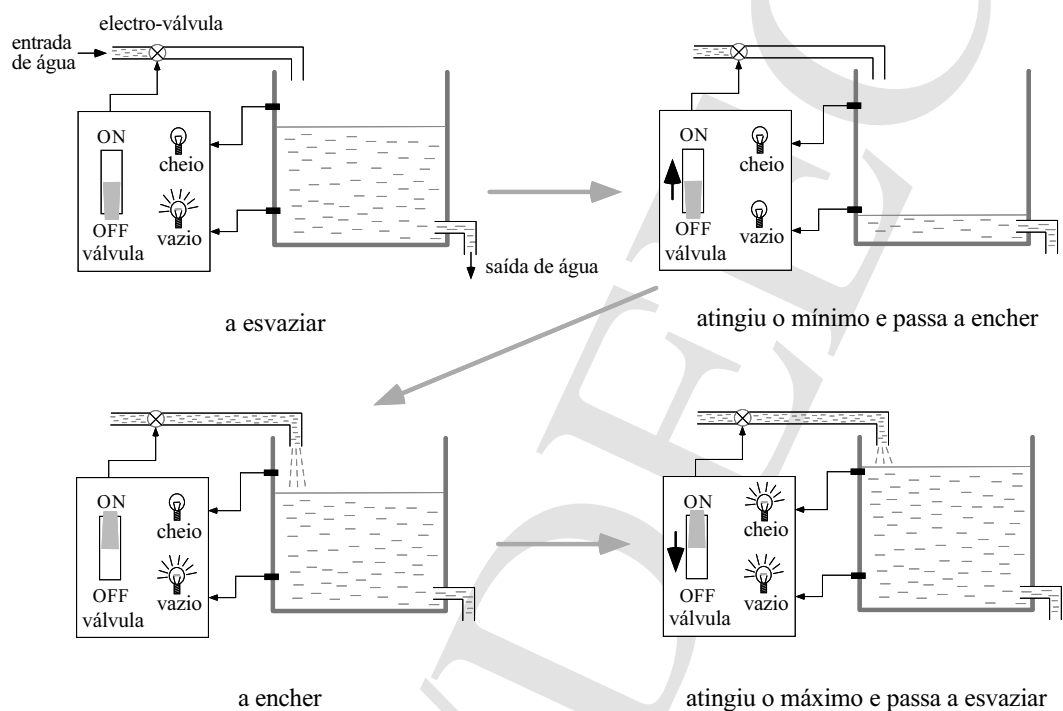


Figura 6.1: Um sistema digital para controlo do nível de água num depósito.

Quando ambas as lâmpadas estão acesas ou apagadas, não há dúvida que a válvula deve ser desligada ou ligada, respectivamente. Mas qual deve ser a decisão se o estado das lâmpadas indicar que o nível de água está entre o mínimo e o máximo? De acordo com o processo de controlo que foi especificado, a posição da válvula deve ser definida em função daquilo que aconteceu *antes*: se anteriormente o nível de água desceu abaixo do valor mínimo, então o tanque deve encher e a válvula deve ser aberta; pelo contrário, se antes o nível de água passou acima do máximo, então a válvula deve permanecer fechada porque o tanque está a esvaziar. Assim, o operador tem de ser capaz de *memorizar* o que aconteceu antes para decidir o que deve fazer *agora*, não bastando por isso conhecer o estado das entradas neste instante. Note que neste caso, a informação a memorizar, fundamental para determinar o valor da saída, pode ser representada por apenas um *bit* que nos diz se o tanque está a encher ou a esvaziar, e que não é mais do que o estado actual da electro-válvula¹.

¹Se imaginarmos que o interruptor que comanda a válvula é um vulgar interruptor de 2 posições,

Esta dependência da saída (a electro-válvula) do valor anterior ocorrido nas entradas (os sensores de nível) representa uma característica fundamental de um circuito sequencial: existe memória para guardar informação sobre o valor anterior das entradas, e as saídas dependem não só das entradas actuais como também do conteúdo dessa memória. Os dados mantidos nessa memória constituem aquilo a que se chama *estado*, e representa a informação relativa aos valores anteriores das entradas que seja relevante para o funcionamento do sistema. No nosso exemplo, o estado é formado por apenas um *bit*, já que é suficiente para codificar o que aconteceu antes nas entradas e que se resume em saber se o tanque está a encher ou a esvaziar.

Em circuitos sequenciais, o estado é representado por um ou mais *bits* que são armazenados em elementos de memória que iremos estudar mais à frente. Os *bits* de estado são chamados *variáveis de estado* e geralmente são internos a um circuito sequencial (i.e. o seu valor não é “visto” nas saídas do circuito). As saídas são produzidas por funções lógicas que dependem das entradas e do estado, e por sua vez o estado que ocorrerá a seguir é uma função do próprio estado e das entradas (figura 6.2).

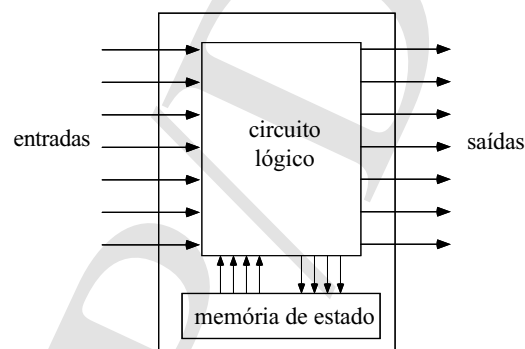


Figura 6.2: Modelo abstracto de um circuito sequencial: entradas, saídas e memória de estado.

Como num circuito sequencial o estado codifica a história dos valores ocorridos nas entradas, deve ser actualizado sempre que ocorram modificações nas entradas que provoquem uma alteração do estado. Existem duas formas principais de efectuar essa alteração e que se implementam através de duas categorias de circuitos sequenciais: assíncronos e síncronos.

Num circuito assíncrono, as mudanças nas entradas conduzem imediatamente à alteração do estado. Assim, sempre que numa entrada ocorre uma transição de 0 para 1 ou então a posição do botão pode ser entendida como a “memória” do sistema: conhecendo o seu estado e o dos sensores o operador já pode realizar correctamente o processo de controlo do nível de água.

de 1 para 0, isso pode provocar a alteração imediata do estado e, conseqüentemente, das saídas que dele dependem. Circuitos sequenciais assíncronos são relativamente difíceis de projectar e o seu funcionamento depende fortemente dos tempos que os dispositivos electrónicos demoram a responder². Iremos estudar apenas alguns circuitos muito simples deste tipo, que implementam funções sequenciais elementares que são fundamentais para a construção de circuitos sequenciais síncronos.

Nos circuitos sequenciais síncronos as entradas são apenas “vistas” em instantes determinados no tempo e o estado apenas é actualizado nesses instantes. Esse modo de operação obriga a que um circuito deste tipo tenha uma entrada de controlo especial onde é aplicado um sinal digital que sincroniza as mudanças de estado. Normalmente esse sinal tem uma frequência constante sendo por isso chamado sinal de *relógio*, e todas as mudanças de estado ocorrem quando esse sinal apresenta as transições de 1 para 0 (ou de 0 para 1). Por exemplo, num circuito sequencial síncrono que opere com um relógio de 1MHz, as suas entradas são avaliadas 1 milhão de vezes por segundo e o estado é actualizado também a esse ritmo.

Um sinal de relógio é caracterizado pela sua frequência f ou período T e também pelo chamado “ciclo útil” (em Inglês *duty-cycle*) que mede, em percentagem do período, o tempo em que o sinal vale 1. Embora numa grande parte dos circuitos sequenciais síncronos que mudam de estado apenas quando o sinal de relógio transita de 1 para 0 ou de 0 para 1 esse parâmetro não seja muito importante, também existem circuitos que usam os dois flancos (os instantes de tempo em que ocorrem as transições) do sinal de relógio e por isso requerem valores determinados para o *duty-cycle*. Na figura 6.3 mostra-se um sinal de relógio e identificam-se os parâmetros que o caracterizam.

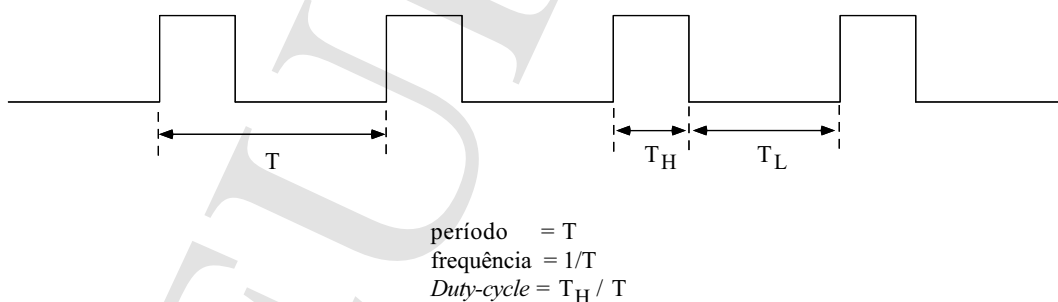


Figura 6.3: Um sinal de relógio e os parâmetros que o caracterizam.

Um circuito sequencial síncrono pode ser decomposto em 3 partes fundamentais, como se mostra na figura 6.4: a memória de estado, um circuito *combinacional* que produz as

²Normalmente chamados tempos de propagação e analisados em pormenor mais à frente na secção 6.1

saídas em função das entradas e do estado actual e outro circuito, igualmente combinacional, que define qual será o próximo estado, igualmente em função do estado actual e das entradas. Um circuito deste tipo é geralmente designado por máquina de estados finitos (ou FSM de *Finite State Machine*) porque naturalmente tem um número finito de estados, determinado pelo número de *bits* da memória de estado.

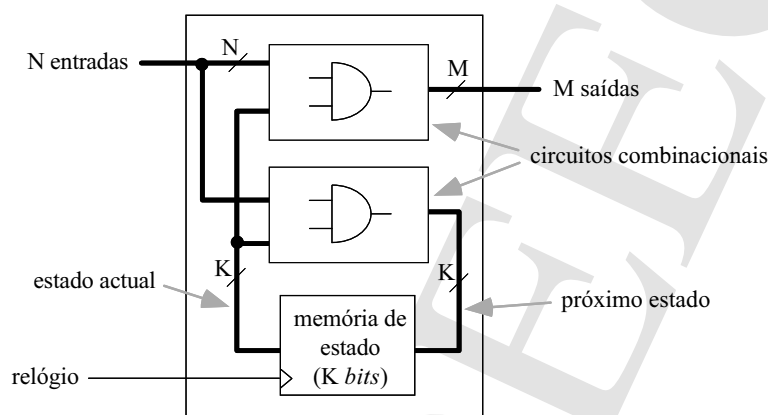


Figura 6.4: Modelo genérico de um circuito sequencial síncrono com N entradas, M saídas e K *bits* de estado (2^K estados).

A memória de estado é realizada por dispositivos electrónicos, geralmente designados por *flip-flops*³, que memorizam e apresentam na saída o valor lógico que estava presente na sua entrada de dados quando na entrada de relógio ocorreu uma transição de 0 para 1 (ou de 1 para 0). Estes dispositivos são as peças fundamentais para a construção de qualquer circuito sequencial síncrono. Como veremos na secção seguinte, podem ser construídos à custa de portas lógicas “normais”, apresentando a diferença, em relação aos circuitos combinacionais, que as saídas são uma função lógica das próprias saídas.

Os dois circuitos combinacionais identificados na figura 6.4 (cálculo das saídas e do próximo estado) podem ser projectados recorrendo às técnicas que foram já estudadas nos capítulos anteriores. Assim, o projecto de um circuito sequencial síncrono que cumpra uma tarefa determinada, consiste fundamentalmente em especificar o número de *bits* necessários para a memória de estado e definir a funcionalidade requerida para esses dois circuitos combinacionais.

Neste capítulo iremos estudar técnicas para projecto de circuitos sequenciais síncronos, começando por um breve estudo sobre os dispositivos sequenciais elementares que são

³O termo *flip-flop* é normalmente empregue para designar elementos de memória que apenas reagem à transição de uma entrada de controlo (sinal de relógio); um outro tipo de dispositivo que iremos estudar chama-se *latch* e o seu funcionamento depende do nível (e não da transição) de um sinal de controlo.

necessários para a realização da memória de estado. Tal como aconteceu para os circuitos combinacionais, as técnicas aqui apresentadas apenas permitem construir máquinas de estados de reduzida complexidade, sobretudo se se recorrer exclusivamente ao projecto manual. Veremos mais tarde no capítulo 7 que também para os circuitos sequenciais existem funções padrão já bem conhecidas e estudadas que podem naturalmente ser usadas para criar circuitos mais complexos.

6.1 Circuitos sequenciais elementares

Os dispositivos sequenciais que implementam os elementos de memória necessários para a construção de circuitos sequenciais síncronos podem ser construídos à custa das portas lógicas estudadas. No entanto, ao contrário dos circuitos combinacionais, num circuito sequencial as saídas são também função das próprias saídas, existindo assim aquilo que se chama *realimentação* entre as saídas e as entradas. É esta diferença em relação aos circuitos combinacionais que permite obter circuitos digitais que implementam funções de memória e que são a peça fundamental necessária para construir circuitos sequenciais síncronos com a estrutura apresentada na figura 6.4.

6.1.1 Circuitos astáveis

O primeiro circuito sequencial que vamos estudar não tem qualquer entrada e consiste num único inversor com a saída ligada à entrada (figura 6.5). Se tentarmos escrever a equação lógica deste circuito obtemos $X = \overline{X}$, o que não está de acordo com as leis da álgebra de Boole porque não existe nenhum valor de X que satisfaça aquela equação lógica. Assim, à luz do que foi estudado sobre álgebra de Boole, pode-se dizer que este circuito é impossível porque não apresenta um comportamento que possa ser descrito pelas regras da álgebra booleana.

Admitamos agora que o inversor apresenta um tempo de propagação de 5ns, o que significa que quando a entrada troca de estado (i.e. muda de 0 para 1 ou de 1 para 0), a saída só é alterada passado 5ns⁴. Imagine-se agora que quando o circuito começa a funcionar, a saída do inversor apresenta o valor lógico 0, que é imediatamente aplicado na entrada do mesmo inversor. Como o inversor demora 5ns a colocar na saída o resultado

⁴Até agora nunca foi referido o factor tempo associado ao funcionamento de um circuito digital; embora este aspecto não fosse relevante para o estudo que foi feito sobre circuitos combinacionais, é fundamental para que se perceba como trabalham os circuitos sequenciais. Veremos mais à frente uma análise mais detalhada do funcionamento de um dispositivo sequencial considerando os tempos de propagação associados às portas lógicas.

da negação desse zero ($\bar{0} = 1$), então a sua saída ainda é mantida com 0 durante 5ns, mudando ao fim desse tempo para 1; quando isso acontece, esse 1 é aplicado imediatamente na entrada do inversor, mas a saída só passa novamente para 0 ao fim de mais 5ns, repetindo-se o processo indefinidamente.

Apesar deste circuito não ter qualquer entrada, a sua saída troca continuamente entre 0 e 1 com uma cadência que é determinada pelo tempo de propagação do inversor (neste caso, o período é igual a 10ns, a que corresponde uma frequência de 100MHz). A este tipo de circuito chama-se *astável* porque a sua saída oscila entre os dois níveis lógicos 0 e 1, sem que para isso exista qualquer acção nas suas entradas (que neste exemplo nem sequer existem!). Note que este modo de funcionamento pressupõe que o circuito “sabe” por algum meio qual era o estado em que estava antes e por essa razão pode-se dizer que apresenta alguma forma de memória.

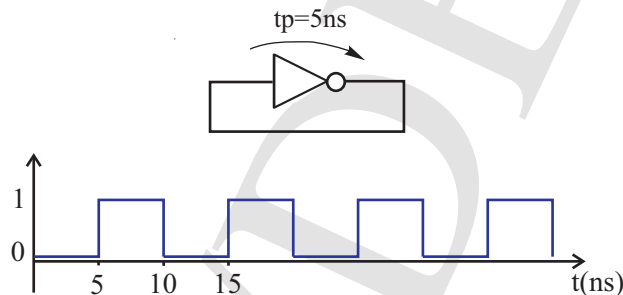


Figura 6.5: Um circuito sequencial astável.

Um circuito deste tipo pode ser usado para gerar um sinal de relógio, embora com uma frequência e *duty-cycle* que são determinados pelas características eléctricas e dinâmicas do inversor, podendo apresentar variações significativas entre dispositivos da mesma família.

6.1.2 Circuitos biestáveis

Ao contrário de um circuito astável, um circuito biestável apresenta a característica de as suas saídas poderem apresentar os dois valores lógicos de forma estável (i.e. que se mantém indefinidamente se não ocorrerem interações externas). Os circuitos biestáveis constituem a peça fundamental na construção dos elementos de memória usados em circuitos sequenciais síncronos.

latch set-reset

Vamos agora modificar o circuito anterior, colocando outro inversor na ligação de realimentação do primeiro inversor (figura 6.6). O comportamento deste novo circuito pode ser descrito pelas 2 equações lógicas $Y = \bar{X}$ e $X = \bar{Y}$ que, ao contrário do circuito anterior já satisfazem as leis da álgebra de Boole ($Y = \bar{\bar{Y}}$). No entanto, estas equações são igualmente satisfeitas se $X = 1$ ou se $X = 0$ e por isso este circuito é designado de bi-estável porque pode ter as suas saídas em dois estados distintos e estáveis, que se mantêm permanentemente se não ocorrerem interações externas (como, por exemplo, desligar a energia eléctrica do circuito).

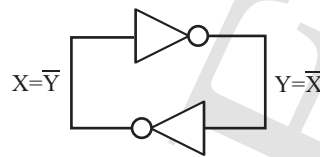


Figura 6.6: Um circuito bi-estável.

Como este circuito não tem entradas e é um circuito estável (i.e. as saídas não se alteram sozinhas), não tem qualquer utilidade prática porque não há meio de alterar o valor apresentado nas saídas.

Vamos agora modificar o circuito colocando uma porta lógica OU na entrada de cada inversor e acrescentando duas entradas S e R , como se mostra na figura 6.7. Como zero é o elemento neutro da soma lógica, quando $S = R = 0$ o circuito mantém o comportamento anterior e por isso a saída Q tanto pode ser 0 como 1.

Vamos admitir que a saída Q apresenta o valor lógico 0 e que as entradas S e R são igualmente zero. Se a entrada S mudar de 0 para 1 (instante 1 no diagrama temporal), a saída do inversor A troca de 1 para 0, mudando a saída do inversor B (saída Q do circuito) de 0 para 1; no entanto, quando a saída Q estiver com o valor lógico 1, então a entrada S pode voltar a ser 0 porque isso não altera o valor 1 presente em Q (instante 2). Assim, colocando temporariamente a entrada S com 1 faz com que a saída Q troque de 0 para 1.

Seguindo uma análise semelhante, pode-se concluir que se seguidamente a entrada R for colocada temporariamente com 1 (instantes 3 e 4), a saída Q passa novamente a ser 0. Note que o valor lógico presente na saída Q é quase sempre igual à negação de Q , e daí a sua designação usual. Isto acontece sempre que o circuito trabalha em condições ditas “normais”, que são aquelas em que nunca acontece $S = R = 1$. Quando ocorre esta

situação “ilegal” (instante 5) ambas as saídas ficam com zero, seja perfeitamente legal do ponto de vista do funcionamento lógico do circuito.

Este circuito constitui assim um elemento de memória que permite armazenar um *bit* de informação, determinado pela entrada que foi activada: *S* para 1 e *R* para 0. A este circuito chama-se *latch set-reset*⁵ e constitui a base para outros circuitos sequenciais mais complexos que iremos estudar de seguida.

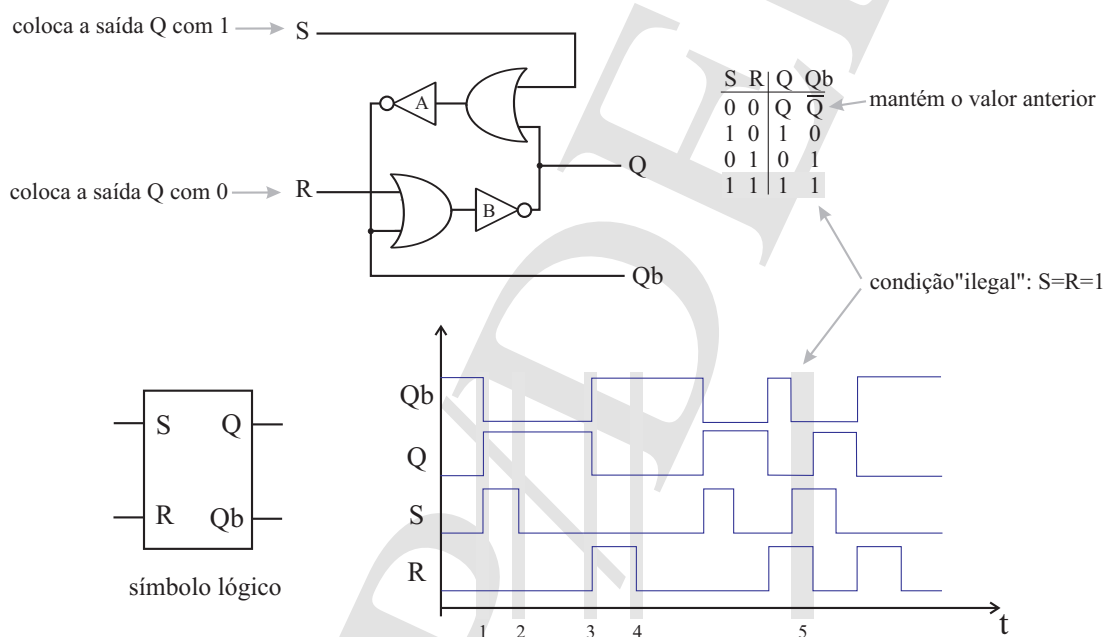


Figura 6.7: Uma unidade de memória elementar: *Latch Set-Reset*.

Latch set-reset com entrada de *enable*

Tal como foi visto quando se estudaram os circuitos combinacionais, também nos circuitos sequenciais é muitas vezes desejável dispor de entradas que permitam ligar ou desligar o funcionamento normal do circuito. Num circuito sequencial, o desactivar o funcionamento normal do circuito significa geralmente manter o estado actual, independentemente dos valores lógicos presentes nas outras entradas de controlo. Aos sinais desse tipo é comum usar a denominação “entrada de habilitação” (ou o termo inglês *enable*).

⁵O termo Inglês *latch* é geralmente empregue para designar circuitos que têm a capacidade de “segurar” um valor lógico.

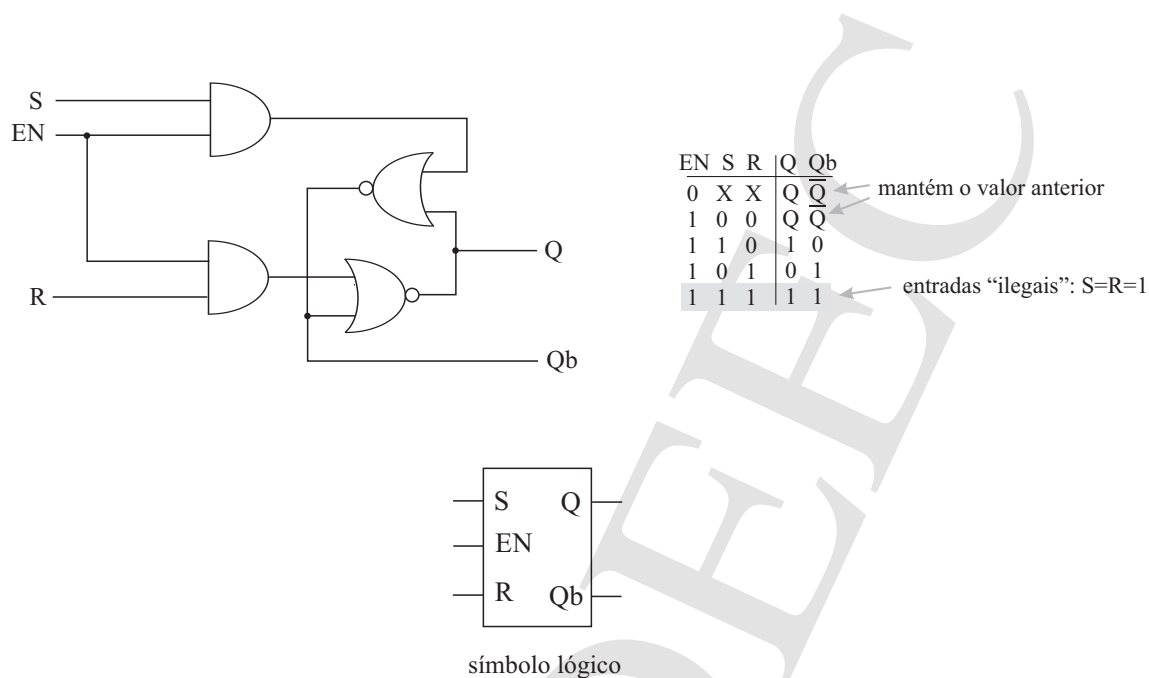


Figura 6.8: *Latch Set-Reset* com entrada de habilitação (*enable*).

A figura 6.8 mostra o circuito lógico de uma *latch set-reset* com entrada *EN* de habilitação. Quando *EN* é zero, as saídas *Q* e *Qb* mantêm o seu estado independentemente dos valores lógicos colocados nas entradas *S* e *R*; quando *EN* é 1, a *latch* mantém o seu comportamento normal porque os valores presentes nas entradas *S* e *R* são aplicados nas entradas das portas lógicas NOR.

Controlo do nível de água do tanque com uma *latch set-reset*

O sistema de controlo de nível de água, estudado no início deste capítulo, pode ser construído usando apenas uma *latch set-reset* (figura 6.9). Como foi mostrado na figura 6.1, pretende-se que a electro-válvula seja ligada quando o sensor VAZIO deixa de ser actuado, e seja ligada de novo logo que o sensor CHEIO seja actuado. Uma solução imediata consiste em ligar a saída *Q* da *latch* à electro-válvula (quando *Q* é 1 a válvula abre), a entrada *R* ao sensor CHEIO (quando o nível de água ultrapassa o máximo a válvula deve fechar) e a entrada *S* ao sensor VAZIO negado (quando o nível de água deixa de actuar o sensor vazio a válvula deve abrir).

O uso de uma *latch set-reset* com entrada de *enable*, permitira acrescentar um sinal de comando adicional para activar ou desactivar o funcionamento do sistema.

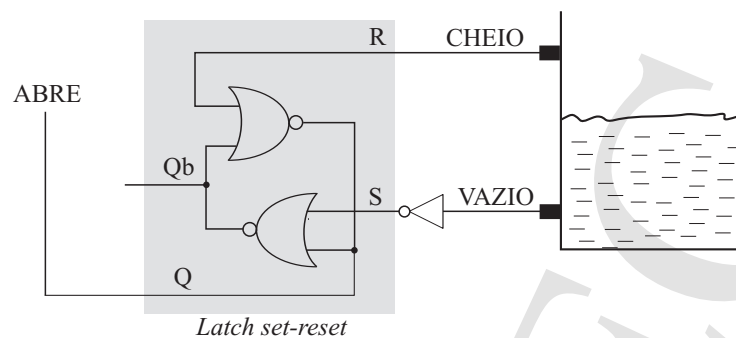


Figura 6.9: Sistema de controlo do nível de água num tanque usando apenas uma *latch set-reset* e um inversor.

latch tipo D

A *latch set-reset* tem as duas entradas de controlo S e R separadas para colocar a saída com 1 ou com 0. Acrescentando nas suas entradas o circuito mostrado na figura 6.10, obtém-se um outro tipo de *latch* chamada *latch* tipo D ou *latch* transparente. Quando a entrada EN (*enable*) vale 1, a saída Q apresenta o mesmo valor presente na entrada D , dizendo-se por isso que é “transparente”; quando a entrada EN muda de 1 para 0 (instantes 1, 2 e 3 no diagrama temporal da figura 6.10), a saída Q memoriza o valor lógico que lá estava presente nesse instante, que se mantém até que EN seja de novo 1. Note que como $S = \overline{R}$, então o estado “ilegal” que foi referido quando se apresentou a *latch set-reset* nunca ocorre, e por isso será sempre $QN = \overline{Q}$.

Note que este circuito constitui uma memória elementar de 1 *bit*, em que a activação da entrada EN permite “escrever” na memória um *bit* de dados colocado na entrada D . Reunindo 8 destes circuitos forma-se uma memória de 1 *byte*, e juntando 1024 destes últimos têm-se uma memória de 1 *Kbyte*⁶.

Flip-flops

A *latch* tipo D apresenta um comportamento “transparente” quando a entrada de controlo EN é activa, dizendo-se por isso que esse circuito é sensível ao *nível* desse sinal de controlo. Também a *latch set-reset* apresenta um funcionamento sensível ao nível das suas entradas: a saída Q muda de estado quando nas entradas S ou R ocorrem transições de 0 para 1, mas o funcionamento dito “normal” só acontece se as duas entradas nunca forem 1 ao mesmo tempo.

⁶Na realidade as memórias electrónicas ditas estáticas (SRAM de *Static Random Access Memory*) são construídas à custa de circuitos elementares que têm uma estrutura semelhante à de uma *latch* tipo D

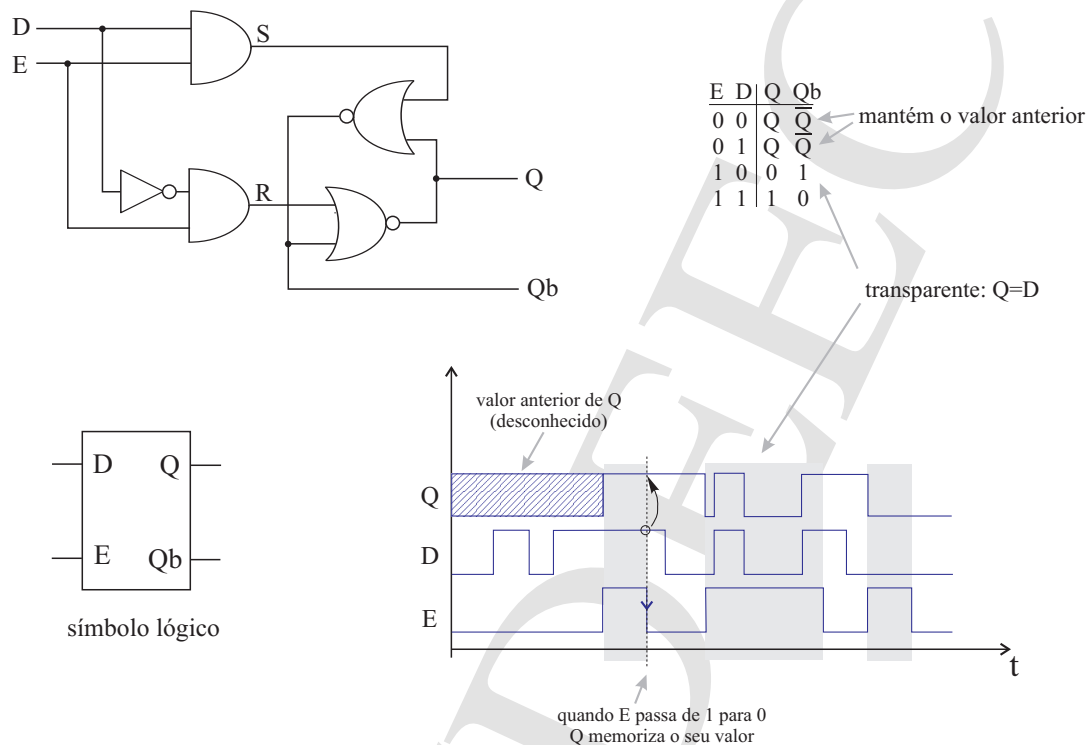


Figura 6.10: Latch tipo D ou latch transparente

Existe um outro tipo de circuito elementar sequencial que apresenta a particularidade de apenas reagir às transições de um sinal de controlo, não sendo sensível ao seu nível lógico. A este tipo de circuito chama-se geralmente *flip-flop* à entrada de controlo cuja transição provoca a mudança de estado é chamada entrada de relógio (em Inglês *clock*)⁷ A designação “relógio” prende-se com o facto de que os *flip-flops* são os dispositivos elementares usados na construção de circuitos sequenciais síncronos, comandados por um sinal periódico cujas transições provocam as mudanças de estado do circuito. Este tipo de dispositivo é também por vezes designado por *latch edge-triggered*, pelo facto de a sua mudança de estado ou “disparo” (*trigger*) ocorrer como consequência de uma transição (*edge*) do sinal de relógio.

Na figura 6.11 apresenta-se o símbolo lógico de um *flip-flop* do tipo D e um diagrama

⁷Não havendo termos em Português que traduzam correctamente o tipo de funcionamento associado a estas duas categorias de dispositivos sequenciais, iremos manter as designações consagradas na língua inglesa *latch* e *flip-flop* para designar os dispositivos sequenciais elementares sensíveis ao nível e às transições das respectivas entradas de controlo.

temporal que exemplifica o seu funcionamento. Sempre que na entrada CLK ocorrer uma transição de 0 para 1, a saída Q passa a conter o valor lógico que estava presente nesse instante na entrada D . Como o *flip-flop* apenas reage à transição ascendente do sinal de relógio, o valor adquirido para a saída Q é mantido até que ocorra uma nova transição na entrada CLK .

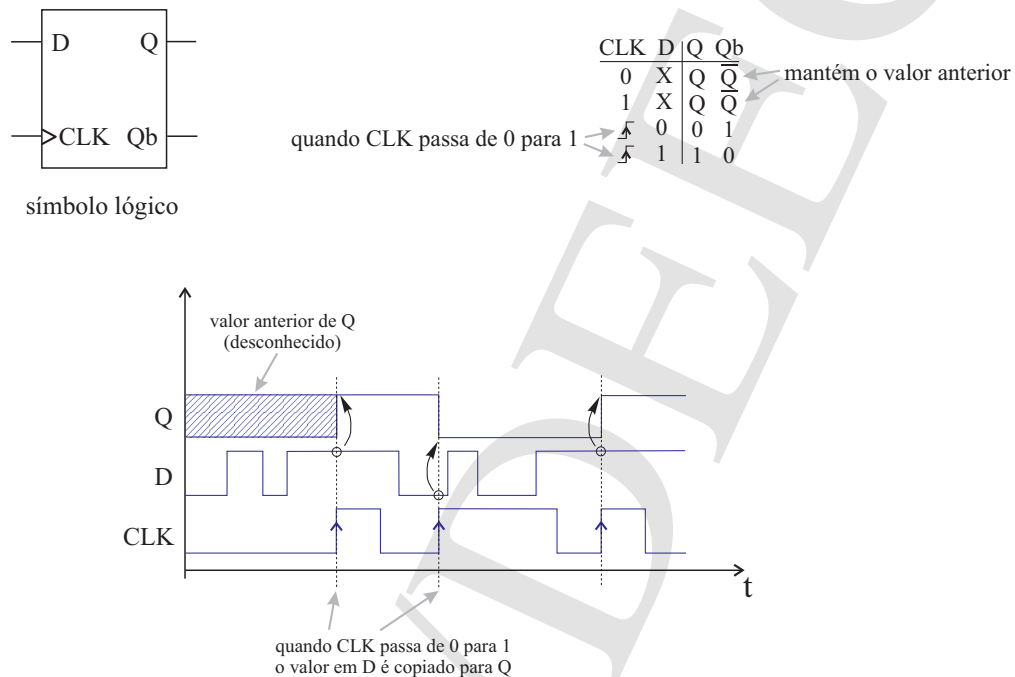


Figura 6.11: Circuito sequencial elementar sensível à transição do sinal de controlo: *flip-flop*. O pequeno triângulo desenhado junto à entrada CLK indica que esta é uma entrada activa na transição ascendente (de 0 para 1) desse sinal.

Um *flip-flop* tipo D pode ser construído com duas *latch* tipo D, da forma que se mostra na figura 6.12, embora existam outras formas de realização mais económicas mas cujo funcionamento é mais complexo de analisar.

O funcionamento deste circuito baseia-se em ter as entradas EN das duas *latch* actuaadas por sinais de controlo opostos. Assim, quando a *latch* 1 está em modo transparente ($CLK = 0$) a *latch* 2 está bloqueada, e quando a *latch* 1 está bloqueada ($CLK = 1$), a *latch* 2 é transparente e coloca na saída Q do circuito o valor presente na saída da *latch* 1. Para perceber o funcionamento deste circuito, vamos analisar passo a passo o diagrama temporal da figura 6.12):

- instante **T1**: Inicialmente CLK vale zero, Q é 0 e D também vale 0.
- período **T2**: Como a *latch* 1 é controlada por \overline{CLK} , então a sua saída $Q1$ terá o mesmo

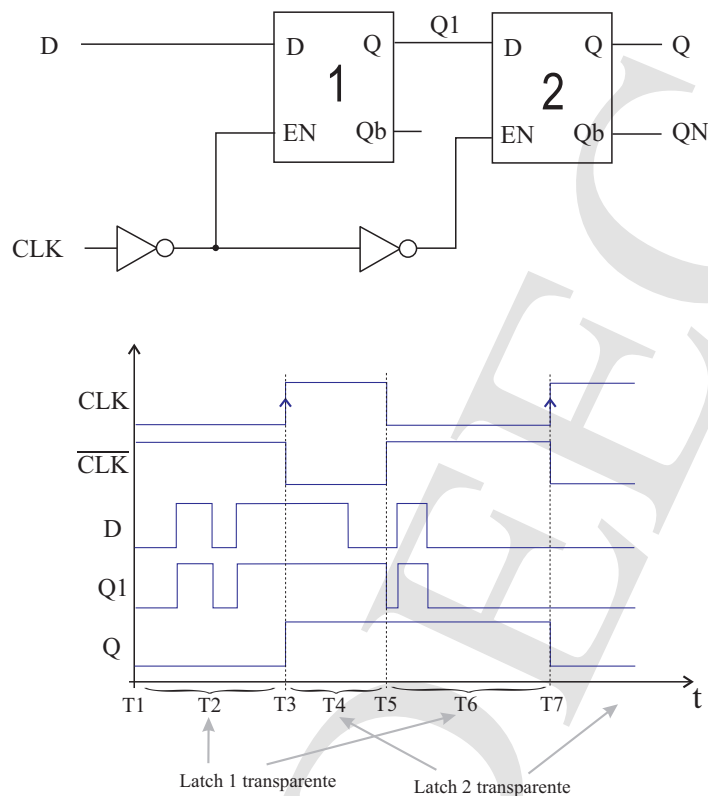


Figura 6.12: Realização de um *flip-flop* tipo D com duas *latch* tipo D.

valor lógico presente na entrada D . Assim, se CLK se mantiver com zero, o sinal $Q1$ acompanha o valor lógico presente na entrada D porque a *latch* 1 é transparente, mas como a *latch* 2 é comandada por CLK que vale 0, a sua saída permanece com o valor lógico 0.

- instante **T3**: Neste instante o sinal CLK passa de 0 para um, e por isso as duas *latch* trocam de papéis: a *latch* 1 passa a estar bloqueada e a *latch* 2 passa a estar em modo transparente. Neste instante, a *latch* 1 memoriza o valor lógico presente na sua saída, que é igual ao estado presente na entrada D .
- período **T4**: Enquanto CLK vale 1, a *latch* 1 está bloqueada, mantendo na sua saída $Q1$ o valor lógico *capturado* da entrada D quando ocorreu a transição de 0 para 1 em CLK ; como a *latch* 2 é transparente, então a saída Q do circuito mantém-se estável, independentemente do valor lógico presente na entrada D .
- instante **T5**: Quando CLK passa de 1 para 0, a *latch* 2 fixa na sua saída o valor lógico que tinha nesse instante.
- período **T6**: Quando CLK é 0, a *latch* 2 está bloqueada e a *latch* 1 é transparente

propagando para a sua saída Q o valor lógico que for colocado na entrada D .

- instante **T7**: Quando CLK passa novamente de 0 para 1, a saída Q passa a ter o valor presente nesse momento na entrada D .

Um *flip-flop* é assim um elemento de memória que, tal como a *latch* tipo D, permite memorizar o *bit* presente na sua entrada D , mas apenas num instante de tempo bem determinado que é quando a entrada de controlo CLK transita de 0 para 1. Como veremos mais à frente, esta característica é fundamental para construir circuitos sequenciais síncronos em que as mudanças de estado são efectuadas em instante de tempo determinados por transições de um sinal periódico.

Características temporais de um *flip-flop* tipo D

A análise de funcionamento do *flip-flop* tipo D apresentada antes não entrou em conta com uma característica que está presente em qualquer circuito electrónico digital e que é o tempo necessário para que as saídas mostrem o resultado da função lógica realizada (tempo de propagação). O diagrama temporal da figura 6.12 mostra a saída Q a passar de 0 para 1 (no instante T3) ao mesmo tempo que o sinal de relógio passa de 0 para 1. Na realidade isso não se passa assim, já que qualquer circuito digital precisa sempre de algum tempo para que a saída mude em consequência de uma alteração nas entradas. No caso de um *flip-flop* tipo D, o tempo de propagação é medido desde o instante em que acontece a transição activa na entrada de relógio até que a saída Q é alterada.

Para além do tempo de propagação, um *flip-flop* é também caracterizado por outros dois parâmetros temporais: tempo de *setup* e tempo de *hold*. Estes valores definem um período de tempo para trás e para a frente da transição de CLK , respectivamente, durante o qual a entrada D deve permanecer estável (figura 6.13). Se essa condição não for garantida, então o *flip-flop* pode não capturar de forma correcta o valor presente em D no instante em que ocorre a transição de relógio e até entrar num estado dito *meta-estável* em que a saída apresenta uma tensão eléctrica que não corresponde a nenhum dos níveis lógicos válidos.

Embora estes tempos sejam muito pequenos (tipicamente inferiores a um nano-segundo), podem não ser verificados se o sinal aplicado à D de um *flip-flop* não “conhecer” o instante em que aparece a transição de relógio. Imagine, por exemplo, que um *flip-flop* a trabalhar com um sinal de relógio de 100MHz é caracterizado por um tempo de *setup* de 1ns e tempo de *hold* igual a 0.5ns. A entrada D deste *flip-flop* é ligada a um botão de pressão que é actuado por uma pessoa, e que naturalmente não está a “ver” quando acontecem as transições de relógio. Assim, durante os 10ns que dura um período de relógio,

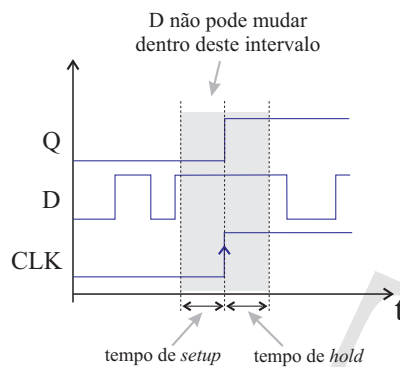


Figura 6.13: Parâmetros temporais de um *flip-flop*: tempo de *setup* e tempo de *hold*.

existe um intervalo de 1.5ns durante o qual a entrada *D* não pode mudar sob pena de o *flip-flop* não funcionar correctamente, e um período de 8.5ns em que a entrada *D* pode trocar de estado. Como é constante a probabilidade de a entrada *D* mudar em qualquer instante ao longo do período de relógio, então isto quer dizer que, em média, 15% das vezes que o botão de pressão for activado a entrada *D* vai mudar demasiado próximo da transição do sinal de relógio (dentro dos 1.5ns), não verificando os tempos de *setup* e *hold* do *flip-flop*. A consequência prática deste tipo de situação é que o circuito efectivamente não funcione às vezes devido ao aparecimento de estados meta-estáveis.

Num circuito sequencial síncrono, as entradas ditas *assíncronas* (i.e. que podem mudar de valor lógico num instante de tempo qualquer não relacionado com o sinal de relógio) devem ser sempre “sincronizadas” com o sinal de relógio antes de serem usadas. Isso pode ser feito usando apenas um *flip-flop* activo com a transição descendente do sinal de relógio, garantindo dessa forma que os circuitos síncronos seguintes, admitindo que são activos no flanco ascendente do relógio, recebem essa entrada estável perto da transição de relógio (figura 6.14).

Meta-estabilidade

A análise feita antes considerou que os componentes do *flip-flop* exibem um comportamento digital “ideal” em que as saídas mudam de estado *instantaneamente* ao fim do respectivo tempo de propagação. Na verdade, num circuito electrónico digital os sinais lógicos são representados por tensões eléctricas que variam de forma contínua no tempo e que necessitam sempre de algum tempo para trocar entre dois estados distintos. Quando num *flip-flop* não são satisfeitos os tempos de *setup* e *hold*, a saída pode mudar de um estado lógico legal (i.e. um nível de tensão eléctrica alto ou baixo representando cor-

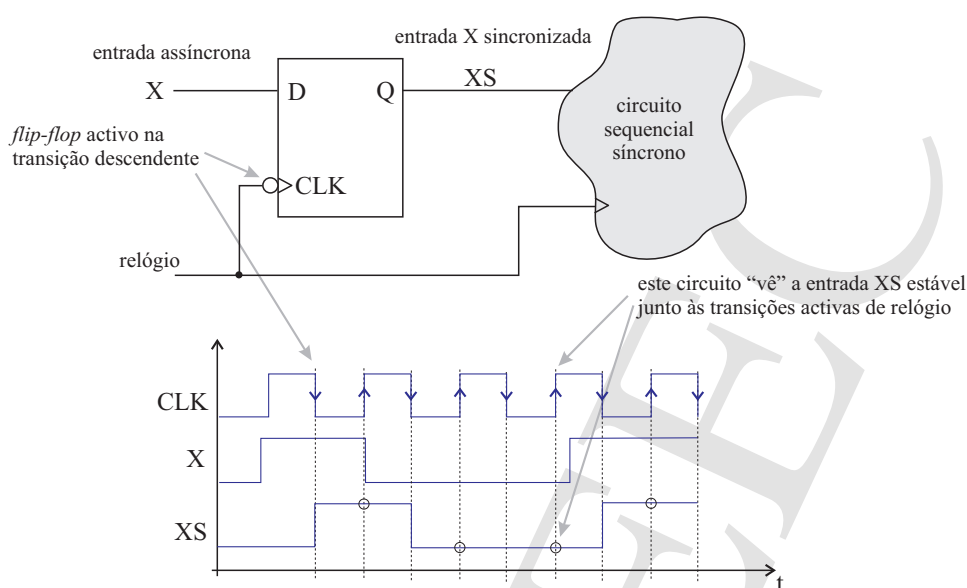


Figura 6.14: Sincronizando uma entrada assíncrona usando um *flip-flop* activo com a transição descendente do relógio.

rectamente os níveis lógicos 1 ou um 0) para outro que se situa entre os níveis legais de tensão eléctrica e que não representa nem 1 nem 0. Esse estado é chamado de *meta-estável* porque, ao contrário dos estados "normais" que representam 1 e 0, não é um estado estável (i.e. é instável) e por isso qualquer pequena "perturbação" faz com que transite para um dos dois estados estáveis.

Podemos comparar esta situação com o comportamento de uma moeda que se tenta equilibrar sobre a borda: apesar de a moeda permanecer estável tombada sobre uma das faces (estado 1 e 0), é difícil colocá-la equilibrada sobre a borda (estado instável); no entanto, quando esse equilíbrio é conseguido, qualquer pequena perturbação (um abanão na mesa, por exemplo) faz com que a moeda tombe lateralmente para um dos seus estados estáveis.

Por outro lado, quando a moeda está estável tombada sobre uma das suas faces, é necessário aplicar-lhe uma determinada quantidade de energia durante um certo tempo para se conseguir virá-la para a outra face. Além disso, durante esta operação ultrapassa-se um estado instável onde a moeda poderia estacionar se a energia mecânica aplicada não fosse suficiente para a virar completamente.

O que se passa num *flip-flop* é semelhante embora com algumas diferenças muito importantes. Em primeiro lugar, tudo se passa em escalas de tempos muito menores: para trocar de estado um *flip-flop* típico necessita de tempos que se medem em nano-segundos (10^{-9} s) ou até menos (por exemplo, num processador que opere com uma frequência

de relógio de 4GHz existem *flip-flops* que trocam de estado em menos de $1/4 \times 10^9$ s ou seja 250×10^{-12} s); além disso, o tempo durante o qual a saída pode apresentar um estado instável é também muito reduzido. Em segundo lugar, a “energia” que é necessário fornecer a um *flip-flop* para que ele troque de estado pode ser entendida como a conjugação da transição do sinal de relógio e do nível lógico apresentado na entrada *D*: o sinal de relógio tem de transitar “rapidamente” e o nível lógico presente na entrada *D* durante esse período tem que permanecer estável. Se essas condições não se verificarem, o *flip-flop* pode apresentar temporariamente um nível lógico não definido que não é correctamente interpretado pelos circuitos que existem a seguir. No entanto, a pior consequência desta situação é que depois disso, o valor na saída pode passar “aleatoriamente” para 1 ou para 0 e não traduzir um valor correcto para o tipo de funcionamento pretendido.

Outros tipos de *flip-flops*

O *flip-flop* tipo D estudado na secção anterior é o componente básico utilizado na construção de circuitos sequenciais síncronos. Existem outros tipos de *flip-flops* que diferem do tipo D no tipo de entradas que definem o qual próximo valor na saída *Q*, e que podem tornar mais simples a implementação de certas funções sequenciais.

O *flip-flop* tipo T (T de *toggle* ou de troca) tem uma entrada de controlo *T*. Quando ocorre a transição activa do sinal de relógio e $T = 1$, o próximo estado na saída *Q* passa a ser o oposto do que lá estava antes; se $T = 0$ é mantido o estado anterior. Na figura 6.15 mostra-se o símbolo de um *flip-flop* tipo T e uma possível implementação desta função recorrendo a um *flip-flop* do tipo D.

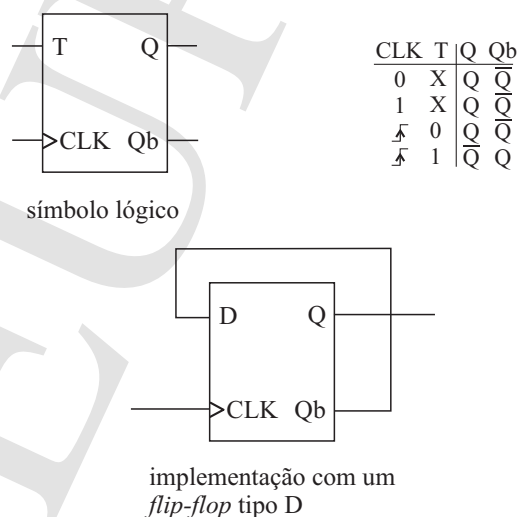


Figura 6.15: *Flip-flop* tipo T.

Outra configuração de *flip-flop* é o chamado *flip-flop* JK. Este tipo de dispositivo tem algum interesse quando se recorre a circuitos integrados de pequena densidade de integração para implementar circuitos sequenciais, pela flexibilidade que oferece face aos dois tipos de *flip-flops* referidos antes. Um *flip-flop* JK tem duas entradas de controlo J e K que definem em conjunto qual o próximo estado a ocorrer na saída Q (figura 6.16). Um *flip-flop* JK pode ser facilmente transformado num *flip-flop* tipo D ou num *flip-flop* tipo D. Como veremos mais tarde, o recurso a *flip-flops* deste tipo conduz, geralmente, ao aparecimento de muitos termos indiferentes nas funções lógicas combinacionais que determinam o próximo estado e as saídas de um circuito sequencial síncrono, o que normalmente conduz a boas oportunidades de simplificação dos respectivos circuitos lógicos. No entanto, como se mostra na figura 6.16, um *flip-flop* JK é mais complexo do que um *flip-flop* tipo D e pode ser obtido acrescentando a este algumas portas lógicas.

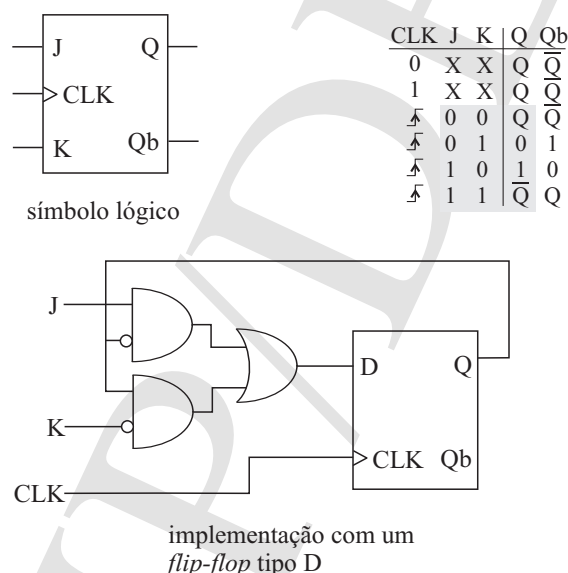


Figura 6.16: *Flip-flop* tipo JK.

Equação característica de um *flip-flop*

Com já vimos antes, o comportamento de um dispositivo sequencial não pode ser descrito da mesma forma do que para um circuito combinacional, porque as saídas podem apresentar valores diferentes para as mesmas combinações de valores nas entradas. Por esta razão é necessário criar uma outra maneira de descrever formalmente o seu comportamento.

No caso particular dos dispositivos estudados na secção anterior a que chamamos *flip-flops*, a saída apenas pode trocar de estado quando no sinal de relógio ocorre uma

transição activa (por exemplo, de 0 para 1). A *equação característica* de um *flip-flop* é uma expressão booleana que determina o valor lógico Q^* a aparecer na saída Q como consequência da transição de relógio e dos valores presentes nas restantes entradas de controlo do *flip-flop* no instante em que ocorre essa transição. No caso de um *flip-flop* tipo D, o *próximo* valor a aparecer na saída Q é o que está presente na entrada D no instante em que ocorre a transição activa do sinal de relógio. Escreve-se por isso:

$$Q^* = D$$

Para um *flip-flop* tipo T, o próximo estado a ocorrer em Q já depende do seu valor actual e também da entrada T : se $T = 0$ Q mantém o estado que tinha antes, e se $T = 1$ Q passa a ser \overline{Q} :

$$Q^* = \overline{T} \cdot Q + T \cdot \overline{Q}$$

Os *flip-flops* JK têm uma equação característica um pouco mais complexa, que traduz o comportamento mostrado na figura 6.16: se $J = K = 0$, Q mantém o seu valor; se $J = 1$ e $K = 0$, Q passará a ser 1, independentemente do seu valor anterior; se $J = 0$ e $K = 1$, Q passará a ser 0, independentemente do seu valor anterior; finalmente, se $J = K = 1$, Q passa a ser o oposto do valor que tinha antes:

$$Q^* = \overline{Q} \cdot J + Q \cdot \overline{K}$$

Note que, como o termo do lado direito das equações características determina o próximo valor na saída Q , qualquer um destes *flip-flops* pode ser construído com um *flip-flop* tipo D cuja entrada D é alimentada por um circuito combinacional que implementa essa expressão.

6.2 Síntese de circuitos sequenciais síncronos

Como vimos antes, um circuito sequencial síncrono caracteriza-se por apresentar mudanças de estado que apenas ocorrem de forma síncrona com certos eventos em determinados sinais de controlo. Usualmente este sincronismo é determinado por um sinal periódico normalmente chamado "relógio", em que as suas transições (de 0 para 1 ou de 1 para 0) provocam o carregamento dos dispositivos da memória de estado e assim a mudança de estado do sistema.

De forma semelhante ao que foi estudado para os circuitos combinacionais, para projectar um circuito sequencial é necessário, em primeiro lugar, definir de forma rigorosa

qual o funcionamento pretendido para o sistema a construir. Enquanto que para circuitos combinacionais isso podia ser feito recorrendo a representações com tabelas de verdade ou equações booleanas, vimos já no início deste capítulo que essa forma de representação não chega para descrever a funcionalidade de circuitos sequenciais. Para obter um circuito digital com a estrutura genérica que se mostrou na figura 6.4, essa descrição formal tem que ser traduzida nas funções booleanas que realizam os dois conjuntos de circuitos combinacionais: cálculo das saídas em função do estado actual e das entradas, e determinação do próximo estado, também em função do valor do estado actual e das entradas.

O processo de síntese de circuitos sequenciais síncronos segue um conjunto de passos que, hoje em dia, podem ser feitos automaticamente por ferramentas computacionais para apoio ao projecto (ferramentas EDA - *Electronic Design Automation*). No entanto, uma fase fundamental deste processo que normalmente não pode ser automatizada e tem que ser feita por "humanos" consiste em interpretar uma descrição informal, em linguagem natural e muitas vezes com algumas ambiguidades, daquilo que se pretende que um determinado sistema faça. Actualmente não há nenhum programa de computador que consiga traduzir num circuito uma especificação do tipo *"... um circuito digital que controle o acesso a um parque de estacionamento com 137 lugares e duas portas de acesso, que tenha um indicador luminoso de parque completo e um contador do número de lugares disponíveis e que..."*. No entanto, tendo essa descrição feita segundo formas padrão, podem ser aplicados processos sistemáticos implementados em aplicações EDA para se obter um circuito digital capaz de realizar essa funcionalidade.

Para se estudar o procedimento de síntese de circuitos sequenciais síncronos (ou máquinas de estados finitos), vamos fazer uso de um exemplo simples que permitirá mostrar todos os passos envolvidos neste processo.

6.2.1 Especificação

Considere um circuito sequencial síncrono com uma entrada X (para além da entrada de relógio presente em qualquer circuito deste tipo que sincroniza as mudanças de estado) e uma saída Y . A saída Y deve ser igual a 1 sempre que em 4 transições ascendentes (i.e. de 0 para 1) consecutivas do sinal de relógio o valor da entrada X apresenta a sequência 1011, sendo 0 no caso contrário. O diagrama temporal da figura 6.17 ilustra o comportamento pretendido para este circuito. Note que o valor lógico presente na entrada X é "visto" apenas quando o sinal de relógio muda de 0 para 1. Para que não ocorram ambiguidades na interpretação do seu valor e para que no circuito real não aconteçam situações meta-estabilidade referidas antes, a entrada X só deve trocar de valor lógico em

instantes de tempo bem afastados das transições de relógio⁸.

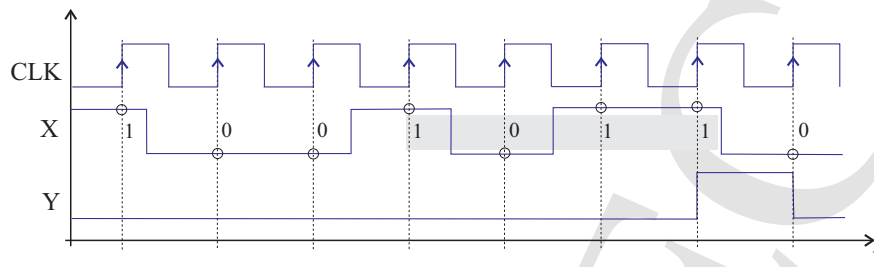


Figura 6.17: Diagrama temporal ilustrando o comportamento de uma máquina de estados que activa a saída Y quando detecta na entrada X a sequência 1011.

Diagrama de transição de estados

O projecto de uma FSM começa pela especificação do seu comportamento recorrendo a uma representação gráfica a que se chama *diagrama de transição de estados* (figura 6.18). Este diagrama descreve de forma rigorosa quais são os estados possíveis do circuito, quais são os valores das saídas em cada estado e de que forma elas dependem das entradas e sob que condições ocorrem as mudanças entre estados.

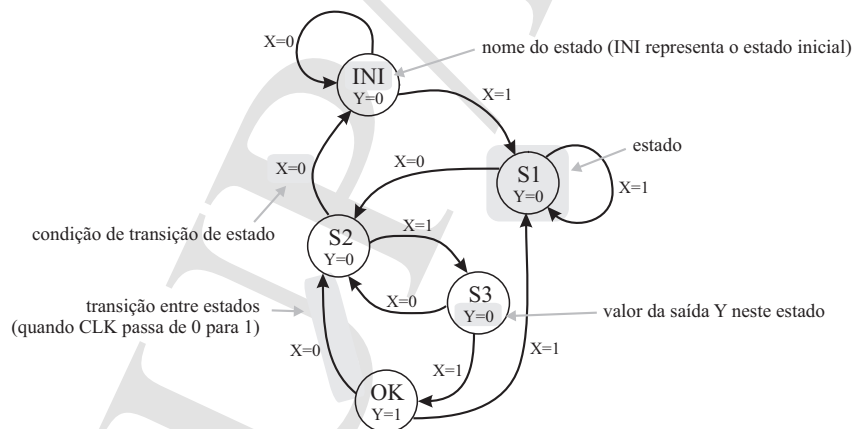


Figura 6.18: Diagrama de transição de estados.

⁸Como vimos antes, a entrada D de um *flip-flop* deve permanecer estável algum tempo antes (tempo de *setup*) e também algum tempo após *tempo de hold* a transição do sinal de relógio; no entanto, como num circuito sequencial síncrono as entradas D dos *flip-flops* de estado são geralmente alimentadas por circuitos combinacionais que operam sobre as entradas da máquina de estados, estas apresentam restrições temporais que não coincidem com os valores característicos dos *flip-flops*.

Nesta fase do projecto é apenas importante descrever de forma não ambígua o comportamento esperado para o sistema. Assim, um estado é representado como uma entidade simbólica que tem associada a “história” daquilo que aconteceu antes, memorizando a informação relevante para o comportamento futuro do sistema. A cada estado deve ser atribuído um nome simbólico que tenha alguma relação com aquilo que esse estado representa. Dentro de cada estado, representam-se também as acções a efectuar nesse estado que no nosso caso se resumem a colocar a saída Y com 1 ou com 0. As setas desenhadas entre os estados traduzem as possíveis mudanças entre estados, sob as condições representadas junto de cada seta. Como no nosso caso apenas existe uma entrada X , todas as condições de transição de estado são expressas apenas em termos de X (por exemplo, se o estado actual é INI e $X = 1$, então o próximo estado será $S1$). Note que para a categoria de circuitos que estamos a estudar, as mudanças de estado acontecem sempre sincronizadas com as transições activas do sinal de relógio.

No nosso exemplo temos 5 estados, cada um dos quais resume o que aconteceu antes na única entrada X do circuito. O estado INI representa o estado em que se assume que é iniciado o funcionamento da máquina de estados. Como tal, não há qualquer memória dos valores anteriores ocorridos na entrada X , ou não é importante saber quais foram esses valores. Em qualquer máquina de estados deverá existir um estado de arranque que seja fácil de impor quando o circuito digital real inicia o seu funcionamento, e que geralmente tem associado o significado “*não se sabe quais foram os valores anteriores nas entradas, ou quais foram os estados anteriores*”.

Para verificar que o diagrama de transição de estados descreve realmente a função pretendida, vamos verificar o seu comportamento para todas as condições que possam surgir na entrada X . Na figura 6.19 mostra-se o diagrama temporal que serviu de exemplo à especificação deste circuito, agora anotado com os nomes dos estados presentes em cada ciclo de relógio. Note que as transições de estado acontecem quando o sinal de relógio passa de 0 para 1, e um estado dura exactamente um período de relógio.

Como queremos detectar a sequência 1011, sempre que no estado INI é detectada a entrada $X = 0$, o próximo estado deverá continuar a ser INI , pois esse valor em X não acrescenta qualquer informação relevante para o estado do sistema. Pelo contrário, se for $X = 1$ então é necessário “memorizar” que se encontrou o primeiro *bit* da sequência a detectar. Isso é traduzido pela mudança para o estado $S1$, que significa assim “*já foi encontrado o primeiro bit da sequência procurada*”.

Se em seguida for detectado novamente $X = 1$ então o próximo estado continua a ser $S1$ porque, para já, apenas é relevante que foi encontrado o primeiro 1 da sequência. Se no estado $S1$ for detectado $X = 0$ então é efectuada uma transição para o estado $S2$,

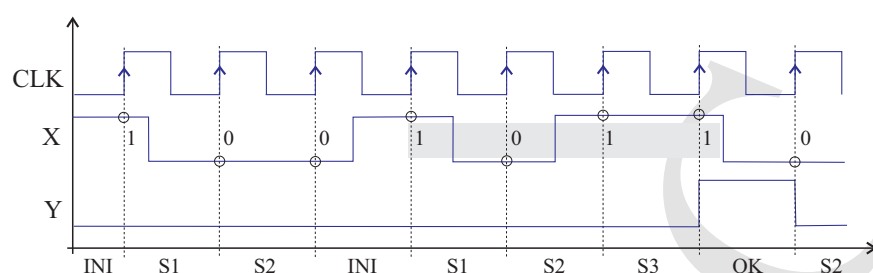


Figura 6.19: Sequência de estados que ocorrem para um exemplo de sequência de valores na entrada X .

significando isso que “foram detectados os 2 primeiros bits da sequência”.

Se no estado $S2$ ocorrer $X = 0$, então o próximo estado deverá ser INI , já que neste caso deve ser reiniciada a procura da sequência desde o início. Esta transição do estado $S2$ para o estado INI quando é lido $X = 0$ deve assim ser entendida como “esquece-se o que aconteceu antes na entrada X e recomeça-se a procura da sequência 1011”. Quando o estado actual é $S2$ e acontece $X = 1$, então transita-se para o estado $S3$ que significa “já foram encontrados 3 bits da sequência procurada”.

Finalmente, quando no estado $S3$ é detectado um 1, atinge-se o estado OK que significa “foi detectada a sequência 1011 procurada” e activa-se a saída Y com 1. Por outro lado, se no estado $S3$ for detectado um zero na entrada X , então regressa-se ao estado $S2$ porque nesse caso (ocorreu em X a sequência ...10) apenas é relevante para o comportamento desejado os 2 últimos *bits* lidos em X , o que é representado pelo estado $S2$.

O comportamento pretendido para a máquina de estados a seguir a ser detectada a sequência 1011 procurada (estado OK) não está completamente especificado na descrição acima, e também não é clarificado pelo diagrama temporal da figura 6.18. A dúvida coloca-se se na entrada X ocorrerem duas sequências iguais à procurada mas *parcialmente sobrepostas*, i.e. com *bits* comuns entre elas. Por exemplo, se em 7 ciclos de relógio consecutivos a entrada X assumir a sequência de valores 1011011, a saída Y deve ou não assinalar a detecção da segunda sequência 1011, já que o primeiro *bit* desta é o último da primeira sequência? A resposta depende naturalmente da aplicação para que se destinaria este circuito, mas é necessário optar por uma das duas hipóteses para se poder completar o diagrama de transição de estados.

O diagrama de transição de estados da figura 6.18 considera que são detectadas sequências parcialmente sobrepostas: do estado OK passa-se para o estado $S2$ se a seguir for detectado $X = 0$ (“aproveita-se” o último 1 da sequência anterior e já se têm 2 *bits* detectados), ou então passa-se para o estado $S1$ se for $X = 1$ (é o primeiro *bit* de uma

nova sequência). A outra situação consiste em considerar que as sequências detectadas não podem ter *bits* comuns (ou seja, não podem ser sobrepostas). Neste caso o estado *OK* deveria “esquecer” que foi encontrada a sequência, mas apenas para efeitos de determinação do próximo estado. Assim, os próximos estados a seguir a *OK* seriam *INI* quando é detectado $X = 0$, ou então *S1* se ocorrer $X = 1$.

Condições de transição de estado

No exemplo apresentado na figura 6.18 apenas existe uma entrada e por isso as condições de transição de estado são expressas em termos de uma só variável (a entrada X do circuito). Estas condições podem ser expressas de variadas formas, embora representem sempre uma função booleana das entradas do circuito. No entanto, quando o número de entradas é elevado, pode não ser prático representar uma expressão booleana complexa num diagrama de transição de estados. A figura 6.20 mostra formas alternativas para anotar um diagrama de transição de estados com as condições de transição entre estados.

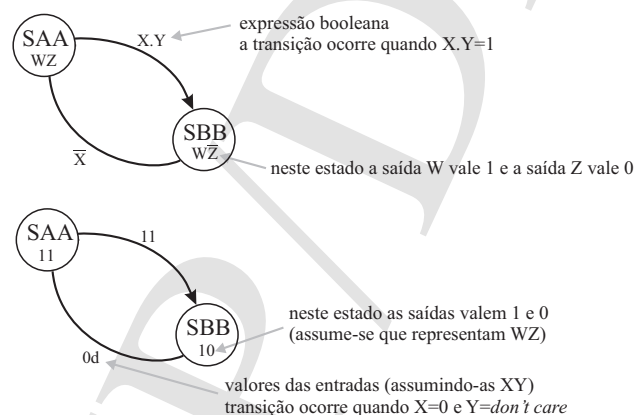


Figura 6.20: Formas de representação das condições de transição de estado.

Tabela de transição de estados

Tendo representado o comportamento do circuito sequencial como um diagrama de transição de estados, o passo seguinte consiste em construir uma *tabela de transição de estados*. Embora esta forma de representação não acrescente qualquer informação à que já consta do diagrama de transição de estados, é um auxiliar conveniente para os passos que vêm a seguir.

A tabela de transição de estados é formada por 3 colunas (figura 6.21): estado presente S , o próximo estado para cada uma das condições possíveis nas entradas S^* e o valor das

saídas em cada estado Y^9 .

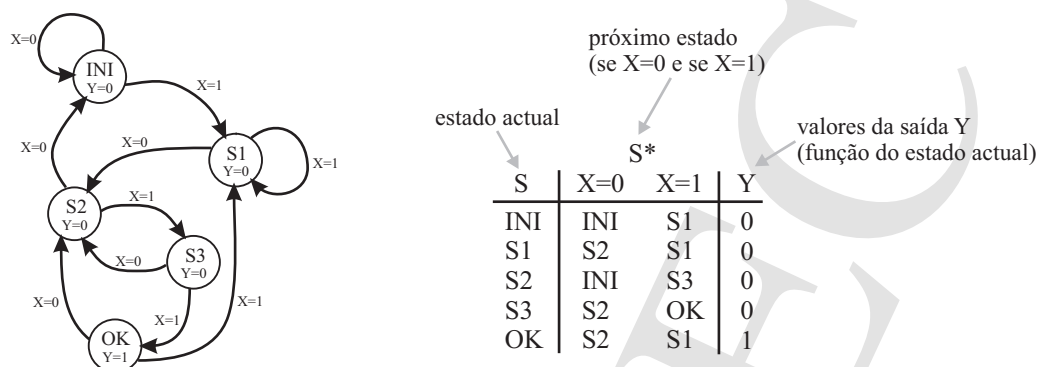


Figura 6.21: Tabela de transição de estados.

A tabela de transição de estados estabelece de forma simbólica as duas funções que são necessárias para construir um circuito sequencial síncrono segundo o modelo geral mostrado na figura 6.4: $S^*(S, X)$ define qual é o próximo estado S^* em função do estado actual S e das entradas X e $Y(S)$ define o valor das saídas as em cada estado.

Codificação dos estados

Para progredir na construção do nosso circuito sequencial, o passo seguinte consiste em transformar a tabela de transição de estados criada antes em tabelas de verdade que descrevam as duas funções lógicas $S^*(S, X)$ e $Y(S)$. Os circuitos combinacionais correspondentes a cada uma destas funções podem depois ser obtidos aplicando as técnicas já conhecidas para projecto de circuitos combinacionais (capítulos 4 e 5).

O primeiro passo para obter essas funções booleanas consiste em representar os estados em binário, por forma a traduzir a tabela de transição de estados numa tabela de verdade. Para codificar N estados são necessários, no mínimo, $\lceil \log_2(N) \rceil$ bits, e um número igual de *flip-flops* para implementar a memória de estado. No nosso caso temos 5 estados e por isso são necessários pelo menos 3 bits para representar esses 5 estados. No entanto, como com 3 bits podem ser formados 8 códigos diferentes, coloca-se o problema de escolher quais os 5 códigos a utilizar, e de que forma os atribuir a cada estado. Como cada conjunto de códigos conduz a diferentes tabelas de verdade, os circuitos digitais resultantes terão complexidades potencialmente diferentes.

⁹O exemplo em estudo segue o modelo de Moore de máquinas de estado, onde as saídas dependem exclusivamente do estado actual; veremos mais tarde um modelo ligeiramente diferente—modelo de Mealy—em que as saídas dependem, em cada estado, também do valor presente das entradas nesse estado.

Este problema é genericamente designado por *codificação de estados* e é um problema muito complexo para o qual não são conhecidos outros meios para obter a melhor solução que não seja experimentar todas as soluções possíveis e escolher a melhor. No entanto, esta abordagem é impraticável, mesmo para problemas muito simples como o que estamos a estudar: tendo uma máquina de estados com N estados e M códigos (com $M \geq N$), o número de conjuntos diferentes de N códigos que se podem fazer é dado por (combinações de M elementos N a N):

$$\frac{M!}{N!(M-N)!}$$

Depois de escolher um destes conjuntos, é ainda necessário atribuir um código a cada um dos N estados, podendo isso ser feito de $N!$ maneiras diferentes (número de permutações de N elementos). No nosso exemplo temos 8 códigos possíveis e 5 estados, donde podemos escolher 56 conjuntos de códigos diferentes. Como cada um desses conjuntos de códigos pode ser atribuído aos 5 estados de $5! = 120$ formas diferentes, então isso significa que existem *apenas* $5 \times 120 = 6720$ formas diferentes de codificar 5 estados com 3 *flip-flops*!

Mais à frente serão apresentadas algumas regras empíricas para codificação de estados que, apesar de não garantirem a obtenção do circuito lógico mais simples, tendem a reduzir a sua complexidade.

A forma mais simples de atribuir códigos aos estados consiste em associar os primeiros N números naturais à lista de estados. Tendo adoptada uma codificação de estados, a tabela de transição de estados pode agora ser reescrita em função desses códigos binários, estabelecendo assim relações booleanas entre um conjunto de 3 *bits* que representa o próximo estado S^* e os 4 *bits* que representam o estado actual S e a entrada X (figura 6.22). Como as variáveis de estado serão realizadas como *flip-flops*, representamos o estado presente por $Q_2Q_1Q_0$ e o próximo estado como $Q_2^*Q_1^*Q_0^*$.

Construção das funções de excitação

Uma vez estabelecidos os códigos binários que representam cada estado, as funções $Y(S)$ e $S^*(S, X)$ passam a poder ser descritas como expressões booleanas já que estabelecem relações entre saídas binárias (Y e S^*) e entradas igualmente binárias (S e X):

$$Y = F(Q_2, Q_1, Q_0)$$

$$Q_2^* = G_2(Q_2, Q_1, Q_0, X)$$

S	$Q_2 Q_1 Q_0$	S*			
		S	$Q_2^* Q_1^* Q_0^*$		
		$Q_2 Q_1 Q_0$	X=0	X=1	Y
INI	0 0 0	0 0 0	0 0 0	0 0 1	0
S1	0 0 1	0 0 1	0 1 0	0 0 1	0
S2	0 1 0	0 1 0	0 0 0	0 1 1	0
S3	0 1 1	0 1 1	0 1 0	1 0 0	0
OK	1 0 0	1 0 0	0 1 0	0 0 1	1

codificação de estados tabela de transição de estados codificada

Figura 6.22: Codificação de estados e tabela de transição de estados codificada.

$$Q_1^* = G1(Q_2, Q_1, Q_0, X)$$

$$Q_0^* = G0(Q_2, Q_1, Q_0, X)$$

Estas funções correspondem aos dois circuitos combinacionais que determinam o próximo estado e as saídas (figura 6.23).

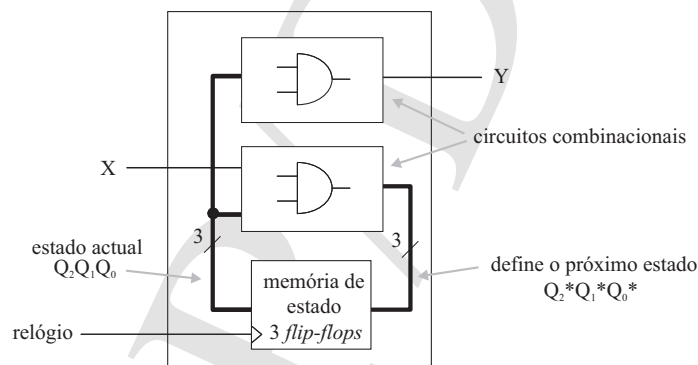


Figura 6.23: Modelo do circuito lógico para a máquina de estados de Moore. Note que as saídas são apenas função do estado presente.

Como foi dito antes, numa máquina de estados deve existir um estado “especial” que é o estado de arranque e que deve ser “forçado” de alguma forma sempre que o circuito electrónico é ligado. Isso pode ser conseguido activando uma entrada de inicialização (*reset*) que normalmente os *flip-flops* possuem e que coloca a sua saída Q em zero independentemente do seu estado anterior e do sinal de relógio (diz-se por isso que é uma entrada de inicialização assíncrona). Por esta razão, um código conveniente para o estado inicial é formado só por zeros

Tendo a tabela de transição de estados representada em função dos códigos binários que foram escolhidos para cada estado, ficam claras as funções lógicas que devem ser

construídas para obter o próximo estado (função do estado presente e das entradas) e as saídas (que neste caso são apenas função do estado presente).

Se representarmos agora estas funções na forma de uma tabela de verdade, podemos verificar que as tabelas não estão completamente especificadas porque, até agora, não se disse nada quanto aos códigos de estado que não foram usados (101, 110 e 111). Esta situação ocorre sempre que o número de estados é inferior ao número de códigos de estado possíveis e por isso há linhas da tabela de verdade que não interferem no comportamento “normal” da máquina de estados mas que devem ser preenchidas para que se possam sintetizar os circuitos lógicos correspondentes (figura 6.24).

	$Q_2 Q_1 Q_0 X$	$Q_2^* Q_1^* Q_0^* Y$
INI	0 0 0 0	0 0 0 0
	0 0 0 1	0 0 1 0
S1	0 0 1 0	0 1 0 0
	0 0 1 1	0 0 1 0
S2	0 1 0 0	0 0 0 0
	0 1 0 1	0 1 1 0
S3	0 1 1 0	0 1 0 0
	0 1 1 1	1 0 0 0
OK	1 0 0 0	0 1 0 1
	1 0 0 1	0 0 1 1
	1 0 1 0	
	...	?
	1 1 1 1	

Figura 6.24: Tabela de verdade incompleta das funções Q_2^* , Q_1^* , Q_0^* e Y .

Uma estratégia que geralmente conduz à minimização do circuito lógico consiste em admitir que nunca são atingidos os estados correspondentes a esses códigos e que por isso não interessa qual é o próximo estado e quais são as saídas. Desta forma, a parte não especificada da tabela de verdade é preenchida com termos indiferentes (*don't cares*) favorecendo assim a simplificação das funções lógicas. Esta é geralmente chamada a estratégia do mínimo custo por conduzir a circuitos lógicos normalmente mais simples do que se for usado outro critério para resolver este problema.

Outra abordagem consiste em considerar que esses estados não especificados podem mesmo ser atingidos, resultando em situações de funcionamento anormal que podem ocorrer devido a interferências externas (por exemplo, uma forte radiação electromagnética ou uma perturbação momentânea da alimentação eléctrica podem fazer trocar o estado de um *flip-flop* atingindo-se assim um estado que não faz parte do funcionamento normal da máquina de estados). Neste caso, a tabela de verdade da função de próximo estado deve ser completada com o código do estado inicial, garantindo assim que o funcionamento normal é retomado se ocorrer uma situação anormal que faça aparecer um estado ilegal.

A tabela que produz as saídas em função do estado actual deve ser sempre completada com valores que sejam “inofensivos” para o funcionamento real do sistema, de forma a que se for atingido um dos estados não usados, as saídas não sejam actuadas indevidamente. Porque este processo garante que a máquina de estados retoma sempre o seu funcionamento normal, chama-se geralmente a este método o critério do mínimo risco.

Adoptando o critério do risco mínimo, vamos preencher o resto da tabela com zeros. Assim, o próximo estado a seguir a um dos estados não usados é sempre o estado inicial da nossa máquina de estados, e se esses estados acontecerem a saída Y não é activada.

A tabela assim construída estabelece uma relação entre o valor do estado presente ($Q_2Q_1Q_0$) e o valor que deverá ser colocado na memória de estado quando o sinal de relógio tiver a transição de 0 para 1 ($Q_2^*Q_1^*Q_0^*$). Uma vez que a memória de estado será realizada com *flip-flops*, devemos agora determinar quais devem ser as entradas de controlo dos *flip-flops* para que se obtenham os próximos valores de Q dados pela tabela de próximo estado. As *funções de excitação dos flip-flops* são obtidas desta tabela, juntamente com as equação característica do tipo de *flip-flop* que se pretende utilizar.

O caso mais simples consiste em usar *flip-flops* tipo D, cuja equação característica é apenas $Q^* = D$. Isto quer dizer que, para cada *flip-flop*, a sua entrada D deverá ser igual ao valor que pretende ter na saída Q , quando acontecer a próxima transição no sinal de relógio.

Com *flip-flops* tipo T, a entrada de controlo T tem de ser igual a 0 se $Q_i = Q_i^*$ e 1 no caso contrário. Usando *flip-flops* JK, devem ser escolhidos valores adequados para as entradas J e K de forma a obter o próximo valor pretendido em Q . Como a equação característica do *flip-flop* JK também depende de Q ($Q^* = \overline{Q}.J + Q.\overline{K}$), apenas um dos valores de J e K tem que ser definido podendo o outro ser qualquer. Na figura 6.25 mostram-se os valores necessários para as entradas de controlo dos 3 tipos de *flip-flops* estudados. Note que apesar de um *flip-flop* JK necessitar de 2 funções lógicas, estas são normalmente muito simples porque são introduzidos muitos termos indiferentes nas suas tabelas de verdade.

Q	Q*	D	T	J	K
0	0	0	0	0	d
0	1	1	1	1	d
1	0	0	1	d	1
1	1	1	0	d	0

← don't care

Figura 6.25: Tabela resumo das equações características para cada tipo de *flip-flop*.

Finalmente, a figura 6.26 apresenta as tabelas de verdade das funções de excitação

para os 3 tipos de *flip-flops* estudados. Para construir o circuito lógico que implementa a máquina de estados, basta agora aplicar as técnicas estudadas para projecto de circuitos combinacionais para obter os dois blocos de circuitos combinacionais referidos na figura 6.23.

estado presente		próximo estado	funções de excitação dos <i>flip-flops</i>																		
			tipo D			tipo T			tipo JK												
Q_2	Q_1	Q_0	X	Q_2^*	Q_1^*	Q_0^*	Y	D_2	D_1	D_0	T_2	T_1	T_0	J_2	K_2	J_1	K_1	J_0	K_0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	d	0	d	0	d	
0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	d	0	d	1	d
0	0	1	0	0	0	1	0	0	0	1	0	0	1	1	0	d	1	d	d	1	
0	0	1	1	0	0	0	1	0	0	0	1	0	0	0	0	d	0	d	0	1	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	d	1	0	d	0
0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	0	d	0	d	0	1	d
0	1	1	0	0	0	1	0	0	0	1	0	0	1	0	0	d	0	d	0	d	1
0	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1	1	d	d	1	d	1
1	0	0	0	0	0	1	0	1	0	0	1	0	1	0	d	1	1	d	0	d	0
1	0	0	1	0	0	0	1	1	0	0	1	1	0	1	d	1	0	d	1	d	0
1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	d	1	0	d	0	d	1
1	0	1	1	0	0	0	0	0	0	0	0	1	0	1	d	1	0	d	0	d	1
1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	d	1	d	1	0	d	0
1	1	0	1	0	0	0	0	0	0	0	0	1	1	0	d	1	d	1	0	d	0
1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	d	1	d	1	d	1	d
1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	d	1	d	1	d	1	d

Figura 6.26: Funções de excitação para a máquina de estados estudada.

Construção do circuito lógico

Recorrendo aos mapas de Karnaugh, obtêm-se expressões minimizadas na forma soma de produtos (figura 6.27) e por fim o circuito lógico completo que realiza a máquina de estados pretendida (figura 6.28).

6.2.2 Circuito de mínimo custo

O circuito da figura 6.28) foi obtido a partir das tabelas de verdade apresentadas na figura 6.26. Recordando, nesta tabela considerou-se que o estado seguinte àqueles que não fazem parte da máquina de estados especificada era o estado inicial (código de estado=000), seguindo assim o critério do mínimo risco.

Adoptando agora o critério do custo mínimo para tratar os códigos de estados não usados, vamos completar a tabela de verdade definindo como termos indiferentes os correspondentes aos estados não usados, para todas as funções de excitação (figura 6.29). Note-se que, à semelhança do que foi feito antes, inclui-se nesta tabela os 3 conjuntos de

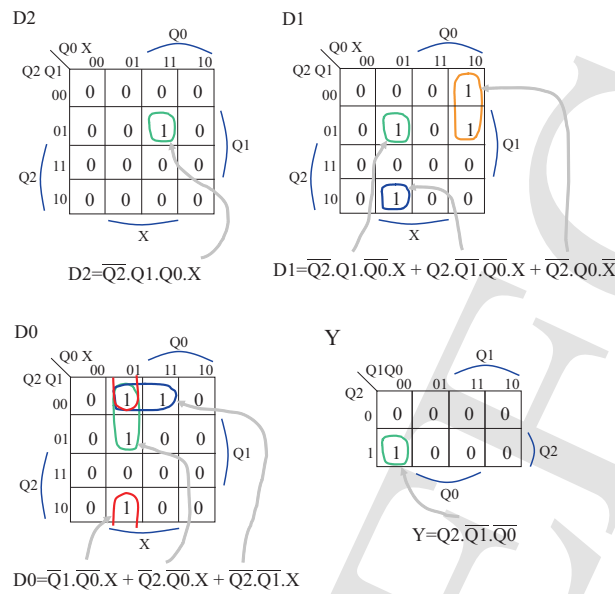


Figura 6.27: Minimização das funções lógicas $D_2(Q_2, Q_1, Q_0, X)$, $D_1(Q_2, Q_1, Q_0, X)$, $D_0(Q_2, Q_1, Q_0, X)$ e $Y(Q_2, Q_1, Q_0)$; note que Y é apenas função do estado presente Q_2, Q_1, Q_0 .

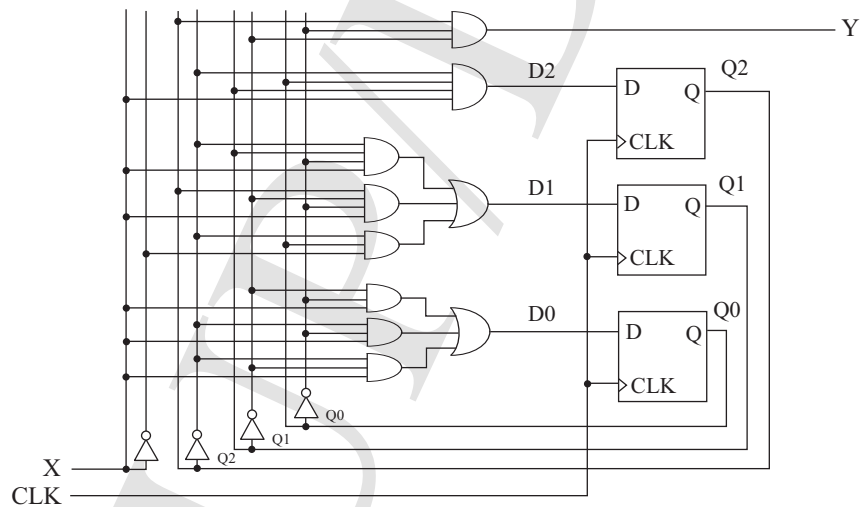


Figura 6.28: Circuito lógico da máquina de estados com *flip-flops* tipo D e adotando o critério do mínimo risco para tratar os estados não especificados.

funções de excitação, para os 3 tipos de *flip-flops* estudados. Naturalmente apenas será necessário definir as tabelas de verdade para as funções de excitação que realmente serão utilizadas.

Aplicando novamente o processo de minimização lógica das expressões soma de produ-

estado presente $Q_2 Q_1 Q_0 X$	próximo estado $Q_2^* Q_1^* Q_0^* Y$	funções de excitação dos <i>flip-flops</i>								
		tipo D			tipo T			tipo JK		
		D_2	D_1	D_0	T_2	T_1	T_0	$J_2 K_2$	$J_1 K_1$	$J_0 K_0$
0 0 0 0	0 0 0 0	0	0	0	0	0	0	0 d	0 d	0 d
0 0 0 1	0 0 1 0	0	0	1	0	0	1	0 d	0 d	1 d
0 0 1 0	0 1 0 0	0	1	0	0	1	1	0 d	1 d	d 1
0 0 1 1	0 0 1 0	0	0	1	0	0	0	0 d	0 d	1 0
0 1 0 0	0 0 0 0	0	0	0	0	1	0	0 d	d 1	0 d
0 1 0 1	0 1 1 0	0	1	1	0	0	1	0 d	d 0	1 d
0 1 1 0	0 1 0 0	0	1	0	0	0	1	0 d	d 0	d 1
0 1 1 1	1 0 0 0	1	0	0	1	1	1	1 d	d 1	d 1
1 0 0 0	0 1 0 1	0	1	0	1	1	0	d 1	1 d	0 d
1 0 0 1	0 0 1 1	0	0	1	1	0	1	d 1	0 d	1 d
1 0 1 0	d d d d	d	d	d	d	d	d	d d	d d	d d
1 0 1 1	d d d d	d	d	d	d	d	d	d d	d d	d d
1 1 0 0	d d d d	d	d	d	d	d	d	d d	d d	d d
1 1 0 1	d d d d	d	d	d	d	d	d	d d	d d	d d
1 1 1 0	d d d d	d	d	d	d	d	d	d d	d d	d d
1 1 1 1	d d d d	d	d	d	d	d	d	d d	d d	d d

Figura 6.29: Funções de excitação adoptando o critério do mínimo custo.

tos, obtemos as funções mínimas (figura 6.30) e finalmente o circuito lógico correspondente (figura 6.31). Comparando este com o obtido anteriormente pelo critério do risco mínimo pode-se verificar que tem menor complexidade (contando, por exemplo, o número de portas lógicas de duas entradas que são necessárias em cada uma das implementações: 23 para o circuito de risco mínimo e apenas 11 para o circuito de custo mínimo).

Implementação com *flip-flops* tipo JK e tipo T

O procedimento para implementar a máquina de estados usando *flip-flops* de outros tipos é semelhante ao que foi visto atrás com *flip-flops* tipo D. Uma vez definida a tabela de verdade das funções de excitação correspondentes a cada tipo de *flip-flop*, basta obter as expressões minimizadas para cada uma dessas funções e reunir os circuitos correspondentes com os *flip-flops* que implementam a memória de estado.

A figura 6.32 mostra as expressões minimizadas (soma de produtos) para as funções de excitação de *flip-flops* JK, e o circuito lógico é mostrado na figura 6.33.

Note-se que o grande número de termos indiferentes que surgem nas funções de excitação para os *flip-flops* JK faz com que o circuito lógico que gera o estado seguinte seja geralmente mais simples do que com *flip-flops* tipo D. Neste exemplo o circuito com *flip-flops* JK necessita de apenas 9 portas lógicas de 2 entradas, enquanto que são precisas 11 para o circuito com *flip-flops* tipo D. Na realidade esta comparação não pode ser feita desta forma, porque o *flip-flop* JK é mais complexo do que um *flip-flop* tipo D.

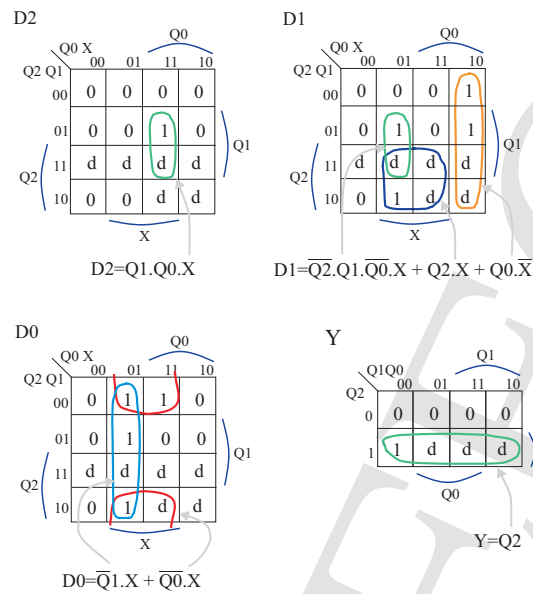


Figura 6.30: Minimização das funções lógicas $D_2(Q_2, Q_1, Q_0, X)$, $D_1(Q_2, Q_1, Q_0, X)$, $D_0(Q_2, Q_1, Q_0, X)$ e $Y(Q_2, Q_1, Q_0)$ adoptando o critério do custo mínimo para os códigos de estado não especificados.

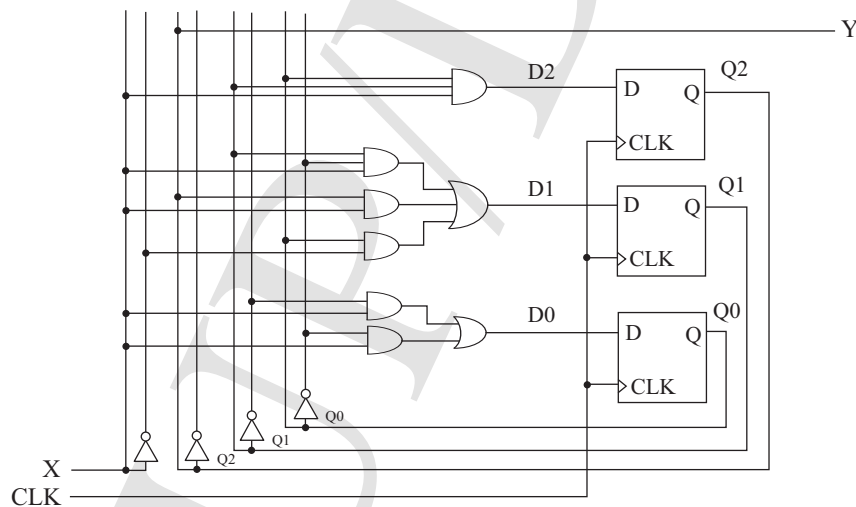


Figura 6.31: Circuito lógico da máquina de estados com *flip-flops* tipo D, seguindo agora o critério do mínimo custo para os estados não especificados.

Considerando a implementação mostrada na figura 6.16, são precisas 3 portas lógicas de 2 entradas e 2 inversores para transformar um *flip-flop* D num *flip-flop* JK. Assim, se contarmos com mais 4 portas lógicas de 2 entradas¹⁰ por cada *flip-flop* JK, este circuito

¹⁰Estabelecendo a nossa medida de comparação o número de portas lógicas de 2 entradas, podemos

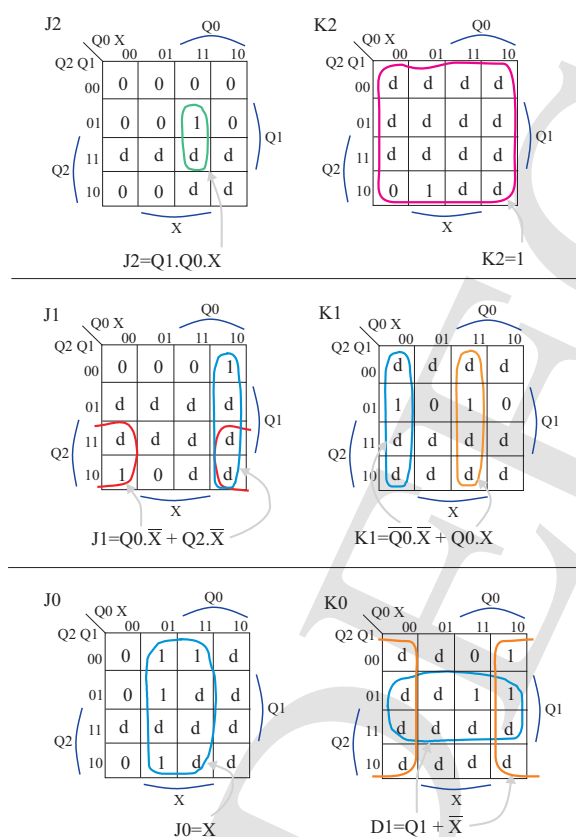


Figura 6.32: Minimização das funções lógicas $J_2(Q_2, Q_1, Q_0, X)$, $K_2(Q_2, Q_1, Q_0, X)$, $J_1(Q_2, Q_1, Q_0, X)$, $K_1(Q_2, Q_1, Q_0, X)$, $J_0(Q_2, Q_1, Q_0, X)$ e $K_0(Q_2, Q_1, Q_0, X)$ adoptando o critério do custo mínimo para os códigos de estado não especificados. Note que a função que produz a saída Y é a mesma que foi construída no circuito com *flip-flops* tipo D

já é claramente pior do que o que usa *flip-flops* tipo D. Os *flip-flops* JK têm especial interesse quando se recorre a circuitos integrados de pequena densidade de integração como os circuitos integrados da série 74', situação em que se torna mais importante reduzir o número de circuitos integrados utilizados do que minimizar o número total de portas lógicas. Nas tecnologias correntes para implementação de sistemas digitais, o dispositivo de memória elementar usado na construção de circuitos sequenciais síncronos é *flip-flop*

considerar que 2 inversores equivalem em termos de complexidade a uma porta lógica de 2 entradas; esta é uma aproximação bastante grosseira, e naturalmente depende da tecnologia de implementação que se considere: por exemplo, em tecnologia CMOS pode-se dizer que 2 inversores são equivalentes a uma porta *inversora* de 2 entradas (NAND ou NOR), enquanto que uma porta não inversora (OR ou AND) já tem uma área equivalente à de 3 inversores porque na verdade é feita com uma porta inversora seguida de um inversor!

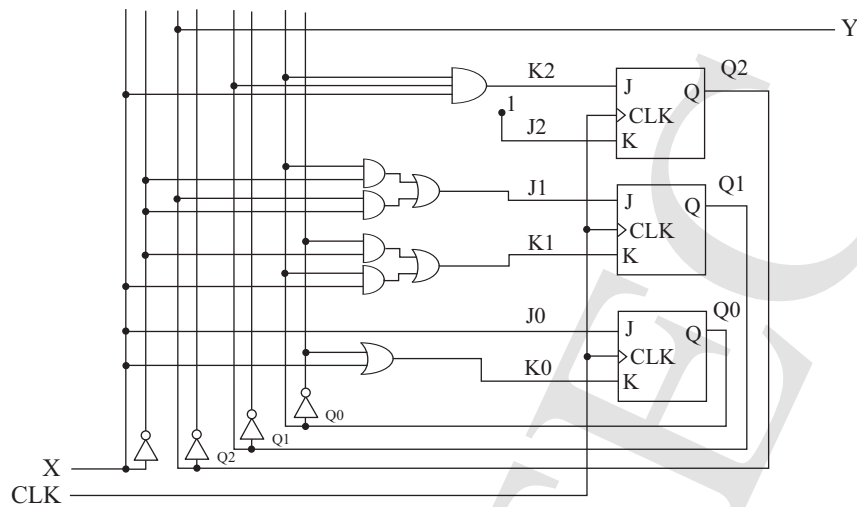


Figura 6.33: Circuito lógico da máquina de estados com *flip-flops* tipo JK, seguindo o critério do mínimo custo para os estados não especificados.

tipo D.

Finalmente, apresenta-se na figura 6.34 as expressões minimizadas das funções de excitação para *flip-flops* tipo T, e na figura 6.35 o circuito lógico correspondente.

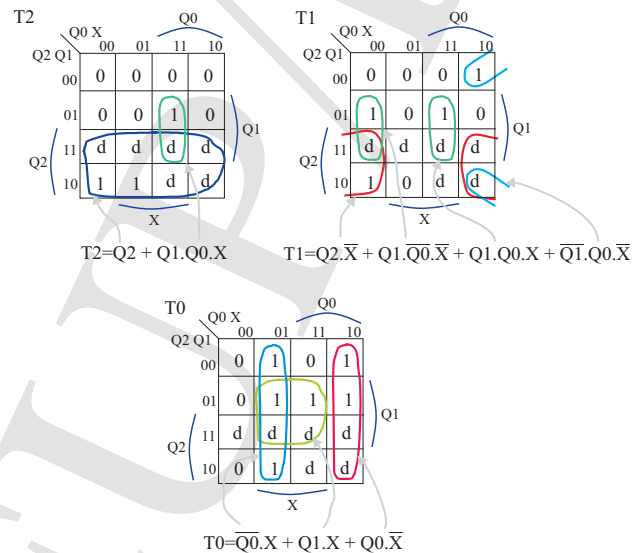


Figura 6.34: Expressões mínimas na forma soma de produtos para as funções de excitação dos *flip-flops* tipo T.

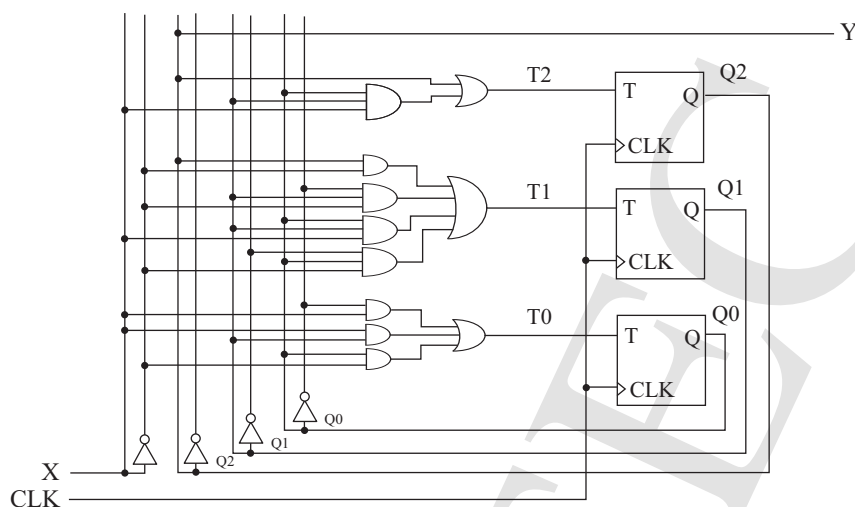


Figura 6.35: Circuito lógico da máquina de estados usando *flip-flops* tipo T.

6.3 Modelo de máquinas de estado de Mealy

O exemplo anterior que serviu de base ao estudo do processo de síntese de máquinas de estados considerou um modelo de circuito sequencial síncrono em que as saídas são produzidas como uma função lógica combinacional do estado actual. Isto quer dizer que, embora exista uma dependência entre a sequência de valores lógicos ocorridos nas entradas e os valores lógicos apresentados pelo circuito nas suas saídas, durante o período de tempo em que dura cada estado (um período de relógio) as saídas permanecem constantes. A este modelo de circuito sequencial síncrono chama-se máquina de estados de Moore e apresenta a particularidade de que apenas é importante o valor lógico presente nas entradas nos instantes de tempo em que acontecem as transições de estado, porque isso apenas é usado para determinar qual vai ser o próximo estado¹¹.

Na figura 6.36 mostram-se 2 diagramas temporais que ilustram o comportamento da máquina de estados de Moore projectada, em que a entrada X exhibe comportamentos muito diferentes. No entanto, como os valores “vistos” nas transições de relógio são os mesmos, também é igual a forma como o circuito responde na sua saída Y .

O modelo de máquina de estados de Mealy caracteriza-se por ter as saídas a depender também directamente das entradas (figura 6.37). Assim, se durante o período de tempo

¹¹Como geralmente as entradas de controlo dos *flip-flops* (D , T , ou J e K) são produzidas por circuitos combinacionais que dependem das entradas da máquina de estados e que têm associado um certo tempo de propagação, então as entradas da máquina de estados devem permanecer estáveis algum tempo em torno do instante em que acontece a transição do sinal de relógio de forma a garantir que são satisfeitos os tempos de *setup* e de *hold* de todos os *flip-flops* do circuito.

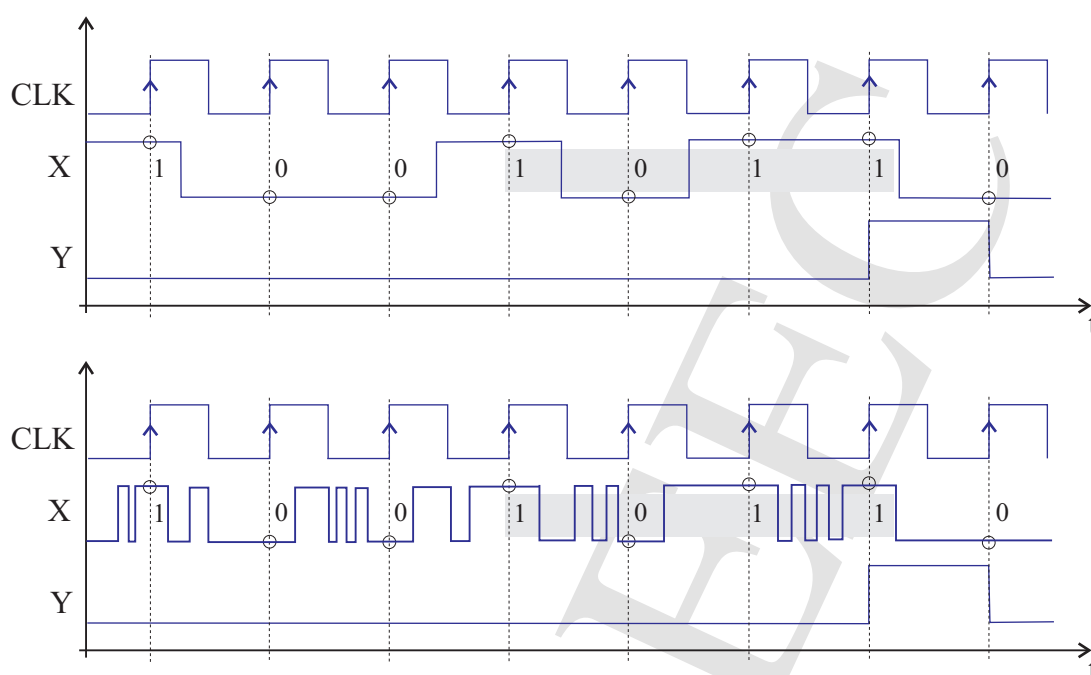


Figura 6.36: Diagramas temporais com duas sequência de entradas diferentes em X, mas que são interpretadas de igual forma na saída Y pela máquina de estados de Moore.

em que dura um estado as entradas mudarem, as saídas podem também trocar de valor. Por esta razão, ao contrário do que acontece numa máquina de estados de Moore em que apenas é importante o valor das entradas *nas transições do relógio*, numa máquina de Mealy o valor das saídas durante o período de tempo que dura um estado depende do valor que está presente nas entradas ao longo de *todo esse intervalo de tempo*.

O projecto de máquinas de estados segundo o modelo de Mealy tem geralmente vantagem face ao modelo de Moore porque normalmente necessita de um menor número de estados conduzindo por isso a um circuito mais simples. No entanto, para que as saídas apresentem correctamente o resultado da função realizada pelo circuito durante o período de tempo que corresponde a um estado, as entradas devem manter-se estáveis durante esse intervalo de tempo. Esta situação não pode ser garantida quando as entradas provêm de outros sistemas que desconhecem o sinal de relógio que determina o sincronismo de mudança de estados. Por exemplo, no exemplo que estudámos inicialmente (controlador de nível de água do tanque), as entradas da máquina de estados podem trocar de valor lógico em qualquer instante de tempo já que apenas dependem da forma como varia o nível de água dentro do tanque. No entanto essa condição já se verifica se as entradas de uma máquina de estados de Mealy forem produzidas como as saídas de uma outra

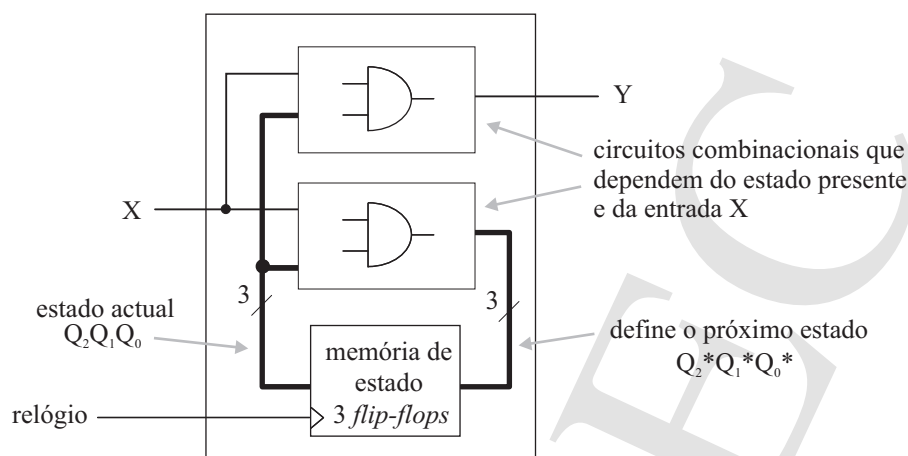


Figura 6.37: Modelo de máquina de estados de Mealy: as saídas são uma função combinacional do estado presente e das entradas.

máquina de estados de Moore que funcione com o mesmo sinal de relógio. Nos exemplos que iremos estudar será considerado que as entradas de máquinas de estados são sempre *síncronas* e as possíveis trocas de valor lógico acontecem sempre logo após as transições activas do sinal de relógio.

Para ilustrar o processo de especificação e síntese de máquinas de estados de Mealy vamos considerar a mesma função que foi usada para o estudo de máquinas de estados de Moore (detector da sequência 1011). No entanto, enquanto que no exemplo anterior a especificação do problema consistia em activar a saída Y quando fosse detectada na entrada X a sequência de valores lógicos 1011 *em 4 transições de relógio consecutivas*, com o modelo de Mealy esta definição fica um pouco ambígua. Podemos dizer, embora de uma forma pouco precisa, que o objectivo é detectar aquela sequência em 4 estados (ou períodos de relógio) consecutivos, mas o que significa isto realmente?

Em primeiro lugar, note que como a mudança de estado ocorre sincronamente com as transições activas do relógio, o valor da entrada X apenas é importante nestes instantes para determinar a sequência de estados ocorridos e por isso *não importa* qual é o valor da entrada X durante o resto do tempo que dura o estado. Esta situação é exactamente igual à que acontece numa máquina de estados de Moore, o que se justifica porque os dois modelos são iguais no que se refere ao circuito lógico que determina o próximo estado. No entanto, como a saída Y é agora uma função combinacional da entrada X , pode-se dizer que, logo que seja detectada em X a sequência de valores 101 em *três* transições de relógio consecutivas (que correspondem aos 3 primeiros *bits* da sequência pesquisada), a saída Y pode ser activada se durante esse estado aparecer em X o quarto *bit* pretendido (um

1). Admitindo que a entrada X só pode trocar de valor lógico logo após cada transição de relógio, pode-se então dizer que o circuito pretendido activa a saída Y logo que “vê” na entrada X a sequência de valores lógicos 1011 em 4 períodos de relógio consecutivos. Concluindo, numa máquina de estados de Mealy as mudanças de estado são determinadas pelo valor das entradas no instante de tempo em que ocorre a transição de relógio, mas o valor lógico apresentado pelas saídas depende do valor colocado nas entradas durante todo o tempo que dura um estado (figura 6.38)

6.3.1 Diagramas de transição de estados para máquinas de Mealy

O processo de projecto de máquinas de estados de Mealy é muito semelhante ao que foi estudado para o modelo de Moore. A única diferença reside na forma como se obtêm os circuitos lógicos que produzem as saídas, e em especial na maneira como se representa a informação associada às saídas no diagrama de transição de estados e na tabela de transição de estados.

Como numa máquina de Mealy o valor lógico apresentado pelas saídas é função do estado e também do valor das entradas, já não é possível representá-lo apenas junto de cada estado, como se fez para as máquinas de Moore. Uma forma de representação exhaustiva deveria permitir especificar, em cada estado, quais os valores para *todas as saídas* em função de *todas as combinações* possíveis das entradas. Isto pode ser feito numa das diversas formas que estudámos antes, como uma tabela de verdade ou uma expressão booleana, que represente em cada um dos estados a relação entre as saídas e as entradas. Embora esta seja a única forma de especificar completamente num diagrama de transição de estados a funcionalidade de qualquer máquina de estados de Mealy, para máquinas de estados pouco complexas a maneira mais comum de o fazer consiste em associar o valor das saídas às condições de transição de estado.

Na figura 6.39 mostra-se parte de um diagrama de transição de estados de uma máquina de estados que tem 2 entradas A e B , e uma saída Z que ainda não está representada nesta figura. Vamos admitir que quando o estado presente é $S1$, a saída Z é definida por $Z = \sum_{A,B} 0, 1, 3$, e quando o estado presente é $S2$ a saída Z é dada por $Z = \sum_{A,B} 0, 3$.

No estado $S1$, as condições de transição de estado são tais que, para cada uma, a saída Z vale 0 ou 1 e podemos assim anotar o diagrama representando o valor de Z junto de cada condição de transição de estado, como se mostra na figura 6.40. No entanto, no estado $S2$ já não podemos associar um único valor de Z à condição de transição de estado $AB = X0$, e por isso é necessário decompô-la na disjunção de duas condições:

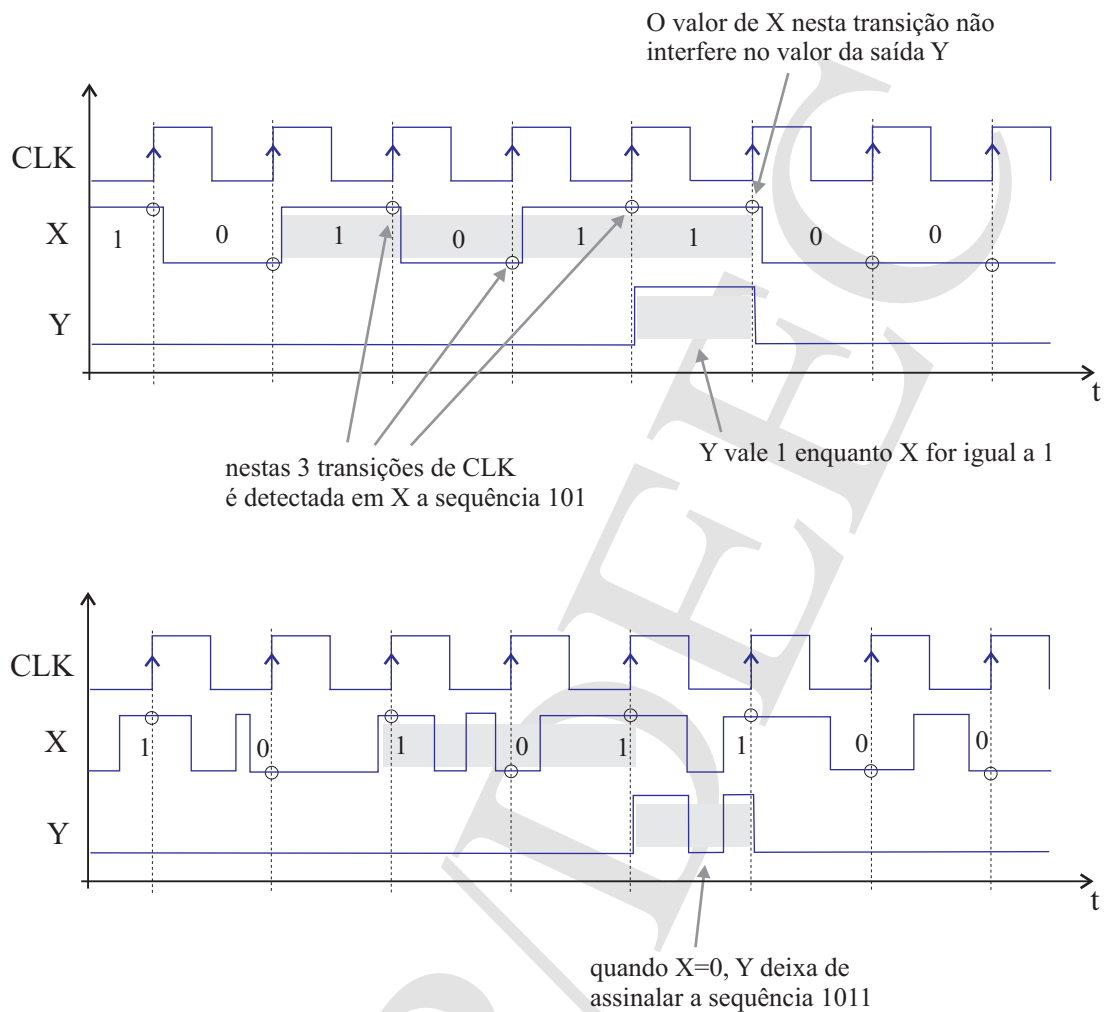


Figura 6.38: Detector da sequência 1011 com uma máquina de estados de Mealy: a) diagrama temporal, admitindo que a entrada só muda de valor imediatamente após as transições do sinal de relógio; b) diagrama temporal considerando agora mudanças arbitrárias de X , embora mantendo os mesmos valores nos instantes das transições de relógio: note-se que após as 3 transições de relógio consecutivas em que se detecta $X = 101$, o valor na saída Y depende de forma combinacional da entrada X : se X é 1 então Y vale 1 assinalando que foi detectada a sequência.

$AB = 00$ para a qual $Z = 1$ e $AB = 10$ a que corresponde $Z = 0$. Note que, apesar de se representar o valor de Z junto de cada condição de transição de estado, isso não significa que aquele valor da saída só aconteça no instante em que ocorre a transição de estado! Na verdade isso quer dizer que aquele valor de Z aparece quando, em cada estado, as entradas apresentam os valores indicados como condição de transição desse estado.

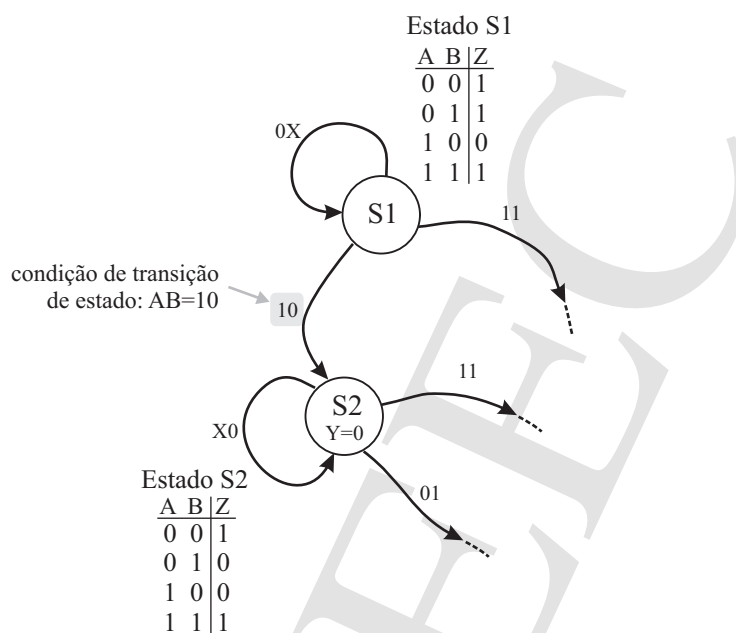


Figura 6.39: Representação parcial do diagrama de transição de estados para uma máquina de estados com 2 entradas A e B , onde o valor que a saída Z assume em cada um dos estados mostrados é representado como tabelas de verdade.

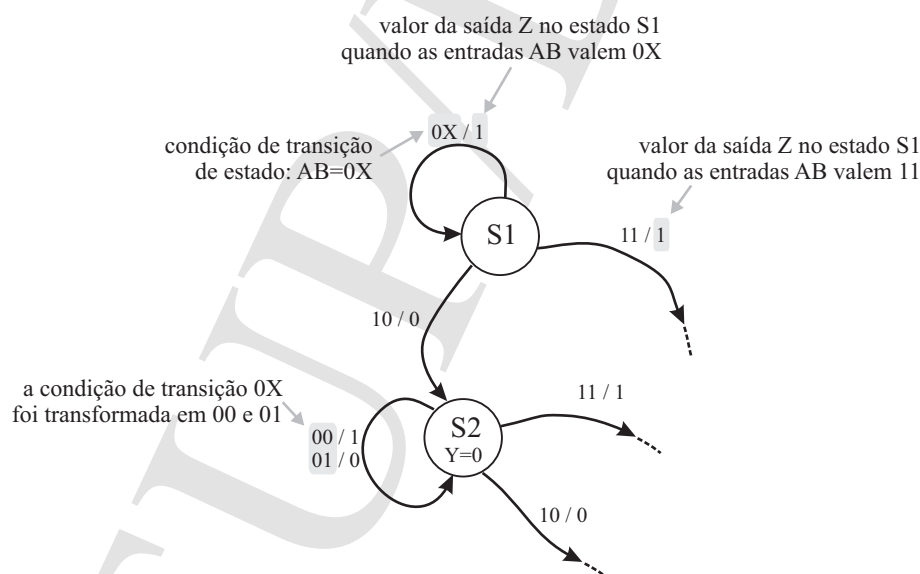


Figura 6.40: Representação do valor da saída Z associado às condições de transição de estado. Note que como no estado $S2$ não há um só valor de Z para para a condição de transição $AB = X0$, é necessário expandir essa condição em $AB = 10$ e $AB = 00$ por forma a associar o valor de Z correcto para cada uma.

6.3.2 O detector de sequência projectado como uma máquina de Mealy

Diagrama de transição de estados

Para construir o diagrama de transição de estados, vamos admitir que o circuito funcionará dentro das condições referidas antes: as entradas só mudam de valor logo após as transições activas de relógio, de forma que são mantidas estáveis durante praticamente a totalidade do tempo que dura um estado (um ciclo de relógio).

A figura 6.41 mostra o diagrama de transição de estados para este circuito. A saída Y assinala a detecção da sequência 1011, quando o estado presente for $S3$ (significa que nas 3 transições de relógio anteriores já foram encontrados os 3 primeiros *bits* da sequência procurada) e a entrada X for igual a 1.

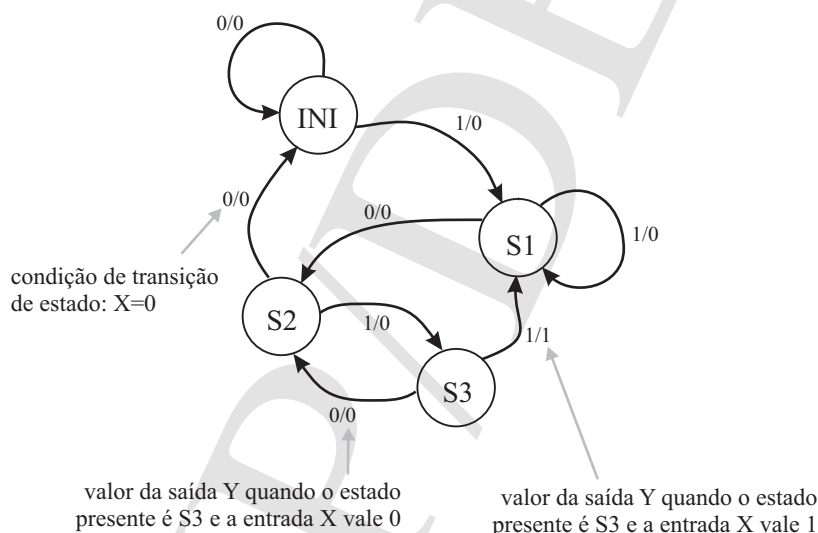


Figura 6.41: Diagrama de transição de estados para a máquina de Mealy detector de sequência 1011.

Tabela de transição de estados

A tabela de transição de estados para uma máquina de Mealy é construída de forma semelhante à que estudámos para máquinas de Moore. Existe contudo uma diferença importante que resulta da dependência directa entre as saídas e as entradas: o valor das saídas tem que ser especificado para cada estado e para cada uma das possíveis combinações das entradas nesse estado. A figura 6.42 mostra a organização da tabela de transição de estados para o exemplo em estudo. Note que agora o circuito lógico que irá

produzir as saídas (de Mealy) terá por entrada as variáveis de estado e todas as entradas das quais as saídas dependem.

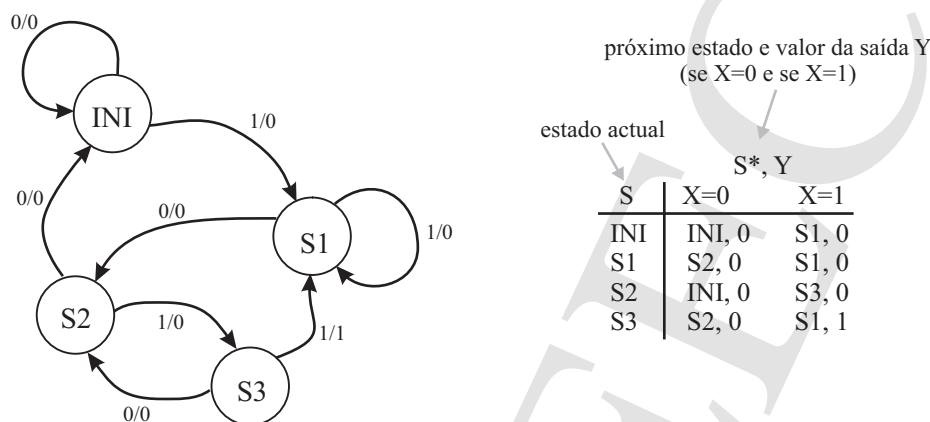


Figura 6.42: Tabela de transição de estados para a máquina de Mealy detector de sequência 1011.

Seguindo um processo análogo ao que foi estudado para máquinas de Moore, o passo seguinte consiste em escolher um conjunto de códigos binários a atribuir aos estados e com isso compor as tabelas de verdade que definem o próximo estado e as saídas (figura 6.43). A partir desse ponto o processo de síntese segue o mesmo caminho estudado antes: escolhe-se um tipo de *flip-flop*, determinam-se as tabelas de verdade necessárias para obter os próximos estados pretendidos, em função das equações de excitação de cada tipo de *flip-flop* e sintetizam-se as funções lógicas nas formas padrão estudadas recorrendo a mapas de Karnaugh para obter realizações mínimas a dois níveis de lógica. Note que a única diferença em relação ao que foi estudado para o processo de projecto de máquinas de estados de Moore é a função que produz as saídas que dependem agora (de forma combinacional) das entradas.

6.4 Codificação de estados

Como foi referido antes, uma fase muito importante do processo de projecto de máquinas de estados finitos consiste na escolha de um conjunto de códigos binários para representar cada estado. Essa escolha pode ser conduzida visando vários objectivos: minimizar o custo ou tamanho do circuito, maximizar a frequência do sinal de relógio ou minimizar o consumo de energia. Satisfazer simultâneamente todos esses objectivos é uma tarefa muito complexa e por vezes é mesmo impossível (por exemplo, poderá ser impossível conseguir que o circuito mais pequeno seja igualmente o mais rápido ou o que consome

Usando a codificação de estados:

Estado	Q1	Q0
INI	0	0
S1	0	1
S2	1	0
S3	1	1

próximo estado e valor da saída Y
(se X=0 e se X=1)

estado actual

Q1 Q0		Q1*, Q0*, Y	
		X=0	X=1
0	0	0 0, 0	0 1, 0
0	1	1 0, 0	0 1, 0
1	0	0 0, 0	1 1, 0
1	1	1 0, 0	0 1, 1

Q1	Q0	X	Q1*	Q0*	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	1	0	0
1	1	1	0	1	1

Tabela de verdade para as funções $Q1^*(Q1, Q0, X)$
 $Q0^*(Q1, Q0, X)$
 $Y(Q1, Q0, X)$

Figura 6.43: Tabela de transição de estados adoptando uma codificação binária natural para os estados, e a reorganização da mesma, na forma de tabela de verdade (máquina de Mealy detector de sequência 1011).

menos energia). O objectivo que geralmente se considera ser o mais importante é a redução da complexidade do circuito ou do seu tamanho físico, que pode ser traduzido, pelo menos de forma aproximada, na redução do número de portas lógicas necessárias para o construir. Como os códigos atribuídos a cada estado determinam as funções lógicas que produzem o próximo estado e as saídas, essa escolha é por isso um factor determinante da complexidade final do circuito que realiza a máquina de estados. O problema de encontrar o melhor conjunto de códigos binários a atribuir aos estados que conduza ao circuito mais simples é um problema classificado como “NP-Completo”. Por um lado, o número de soluções possíveis cresce de forma exponencial com a dimensão do problema, que podemos aqui considerar como o número de estados. Por outro lado, encontrar a solução melhor de todas obriga necessariamente a analisá-las todas para que se consiga saber qual é a melhor. Apesar de isto não ser impossível (na verdade, o número de soluções é sempre um número finito!), veremos que não é praticável essa abordagem porque o número de soluções possíveis cresce de forma extremamente rápida com o número de estados, mesmo que admitíssemos recorrer ao auxílio dos computadores mais rápidos que existem hoje na

Terra!

Para compreender a verdadeira dimensão do problema, recordemos o exemplo que foi usado para ilustrar o processo de projecto de máquinas de estados de Moore (ver diagrama de transição de estados na figura 6.18 na página 142). Esta máquina de estados tem 5 estados e para os codificar são necessários, no mínimo, 3 *flip-flops*. Como com 3 *bits* podemos representar 8 códigos, o problema que se coloca agora é saber qual é o conjunto de 5 códigos e de que forma devem ser atribuídos a cada estado por forma a que o circuito lógico final seja o mais simples possível. Como vimos antes, o número total de grupos diferentes que se podem formar com N elementos (no nosso exemplo, 5) retirados de um conjunto de M elementos (o nosso 8) é dado por uma expressão designada por *combinações* de M elementos N a N que tem a forma:

$$\frac{M!}{N!(M-N)!}$$

Fazendo $M = 8$ e $N = 5$, aquela expressão dá “apenas” 56. Mas, uma vez seleccionado um conjunto de códigos para os estados, surge outro problema que consiste em saber de que maneira deve ser atribuído cada um dos 5 códigos a cada um dos 5 estados. Por exemplo, escolhendo o mesmo conjunto de códigos que foram adoptados no exemplo estudado (000, 001, 010, 011 e 100), teremos possivelmente diferentes soluções para o circuito final conforme atribuamos o código 000 ao estado INI ou ao estado S1 ou ao estado OK. O número de maneiras diferentes de fazer essa atribuição para um conjunto de N elementos é dado por *permutações* de N e calcula-se como $N!$. Assim, com os nossos 5 códigos escolhidos teremos ainda $5! = 120$ maneiras diferentes de os atribuir a cada um dos estados. Como há 56 conjuntos de códigos possíveis para os nossos 5 estados e para cada conjunto ainda há 120 maneiras de os atribuir aos estados, podemos concluir que o nosso pequeno circuito pode ser construído de $56 \times 120 = 6720$ maneiras diferentes, que conduzem a circuitos lógicos potencialmente diferentes. Para saber qual desses circuitos é o melhor, só há uma solução que é construí-los todos e compará-los entre si, o que é impraticável mesmo para sistemas muito pequenos.

Para se perceber a forma como estes números “explodem”, mostra-se na tabela 6.1 o número de codificações possíveis para máquinas de estados que recorram ao número mínimo de *flip-flops* para codificar os seus estados. A coluna “Nº códigos” representa o número de conjuntos diferentes de códigos que é possível ter para o número de estados e *flip-flops* indicado. Por exemplo, uma máquina de estados com 12 estados teria mais de 800 mil milhões de maneiras diferentes de codificar os seus estados! Um computador que demorasse apenas 1 mili-segundo a obter e avaliar cada uma dessas soluções precisaria de mais de 27 anos para poder concluir qual seria a melhor solução.

Tabela 6.1: Número de codificações possíveis para máquinas de estados que recorram ao número mínimo de *flip-flops*.

Nº estados	Nº <i>flip-flops</i>	Nº códigos	Nº codificações
2	1	1	2
3	2	4	24
4	2	1	24
5	3	56	6,720
6	3	28	20,160
7	3	8	40,320
8	3	1	40,320
9	4	11,440	4,151,347,200
10	4	8,008	29,059,430,400
11	4	4,368	174,356,582,400
12	4	1,820	871,782,912,000

Pelas razões apontadas, na prática é impossível determinar qual é a codificação de estados que conduz ao circuito mais simples que implementa uma dada máquina de estados. No entanto, existem certas regras práticas e recomendações resultantes da experiência que, quando seguidas, tendem a conduzir a circuitos mais simples do que se for escolhida uma codificação arbitrária.

6.4.1 Formas de codificação

As formas de codificação de estados que foram seguidas nos exemplos analisados nas secções anteriores foram determinadas seleccionando o número mínimo de *flip-flops* necessários para representar esses estados, atribuindo depois aos vários estados uma sequência de números naturais iniciando com zero. Esta é uma forma simples de atribuir uma codificação de estados, mas também é completamente cega porque não tira qualquer partido do conhecimento que o projectista possa ter da especificação da máquina de estados. Além disso, é por vezes vantajoso codificar os estados usando mais *flip-flops* do que o mínimo necessário: embora não contribua, geralmente, para simplificar os circuitos lógicos que produzem o próximo estado, como será visto mais à frente esta abordagem poderá conduzir a simplificações importantes nos circuitos que produzem as saídas.

Para além de códigos binários, outras codificações habituais são *one-hot* e *quase-one-hot*. Na codificação *one-hot* são usados tantos *flip-flops* quantos os estados da máquina de estados, sendo atribuído a cada estado um código binário em que só um *bit* é igual a

1. Embora esta solução necessite de um número de *flip-flops* igual ao número de estados, conduz, geralmente a simplificações importantes nos circuitos que produzem o próximo estado. Esta codificação é geralmente preferida em implementações para dispositivos FPGA onde abundam *flip-flops* e que se não forem usados não servem para mais nada. Uma variante desta forma de codificação é chamada quase-*one-hot* onde se poupa um *flip-flop* porque é utilizado também o código formado só por zeros (em algumas tecnologias é conveniente usar esse código para estado inicial). Note-se, no entanto, que a complexidade global (ou custo) de uma máquina de estados deve ser medida pelo conjunto de elementos de memória (*flip-flops*), lógica de próximo estado e lógica das saídas. Como se disse antes, a única forma de saber qual é a melhor abordagem para cada caso concreto consiste em implementar e comparar os respectivos circuitos.

Algumas regras práticas

Embora não representem soluções rígidas e que resultem bem em todos os casos, existe um conjunto de regras práticas que deve ser seguido quando é feita a atribuição de códigos binários a estados e que se apresentam nesta secção.

1. Estado inicial: o estado inicial deve ser codificado usando um código que seja fácil de “forçar” nos *flip-flops* usando sinais de inicialização global (*reset*). Geralmente os *flip-flops* têm entradas de *reset* ou de *set* (ou ambas) que quando activadas colocam o *flip-flop* no estado 0 ou 1, respectivamente¹². Uma forma de forçar um estado inicial consiste em ligar entre si todos os sinais de inicialização dos *flip-flops* e activá-lo quando o circuito começa a funcionar. Assim, se os *flip-flops* utilizados dispuserem de entradas de *reset*, é conveniente que o estado inicial seja codificado como 00...00, mas se só tiverem entradas de *set* então o estado inicial deverá ser 11...11.
2. Estados adjacentes (i.e com transições entre si) devem ter códigos que difiram do menor número possível de *bits*. A aplicação desta regra tende a reduzir a complexidade dos circuitos lógicos que produzem o próximo estado. Por exemplo, para o diagrama de transição de estados mostrado na figura 6.44 seria preferível usar a codificação mostrada no diagrama de baixo porque há apenas 2 transições em que trocam 2 *bits* dos códigos dos estados adjacentes, enquanto que no de cima há 7 transições de estado em que trocam 2 *bits* dos códigos respectivos.

¹²As entradas de inicialização dos *flip-flops* podem ser síncronas ou assíncronas; no primeiro caso é necessário que o sinal de relógio tenha uma transição activa para que a saída passe para o estado inicial; uma entrada assíncrona altera imediatamente a saída para o estado inicial independentemente daquilo que aconteça no sinal de relógio.

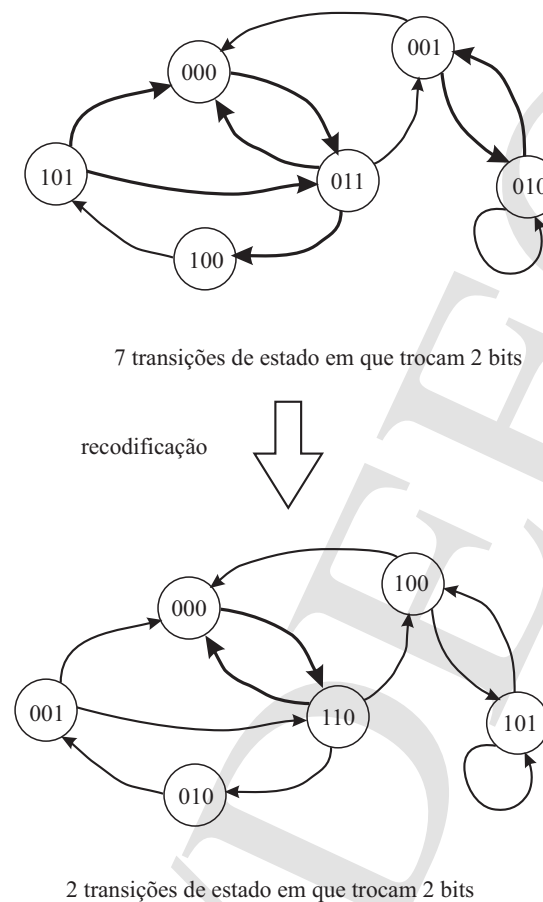


Figura 6.44: Codificando estados procurando minimizar o número de *bits* que trocam entre estados adjacentes.

- Quando o número total de códigos disponíveis por um determinado número de *flip-flops* adoptados para memória de estado é maior do que o número de estados (ou seja, o número de estados não é uma potência inteira de 2), pode-se tentar escolher códigos para os estados que façam com que saídas de Moore se identifiquem com variáveis de estado. Se não for possível aplicar esta regra mantendo o número mínimo de *flip-flops* necessário para representar todos os estados, pode ainda assim ser vantajoso acrescentar uma ou mais variáveis de estado para conseguir aplicar esta regra. A figura 6.45 exemplifica a aplicação desta regra, escolhendo codificações de estados que fazem com que as saídas se identifiquem com variáveis de estado.
- A codificação *one-hot* permite, geralmente, simplificar as equações lógicas que produzem o próximo estado (equações de excitação dos *flip-flops*) mas necessita de tantos *flip-flops* quantos os estados. Com este tipo de codificação de estados, os circuitos lógicos que produzem as saídas de Moore são apenas portas lógicas do tipo

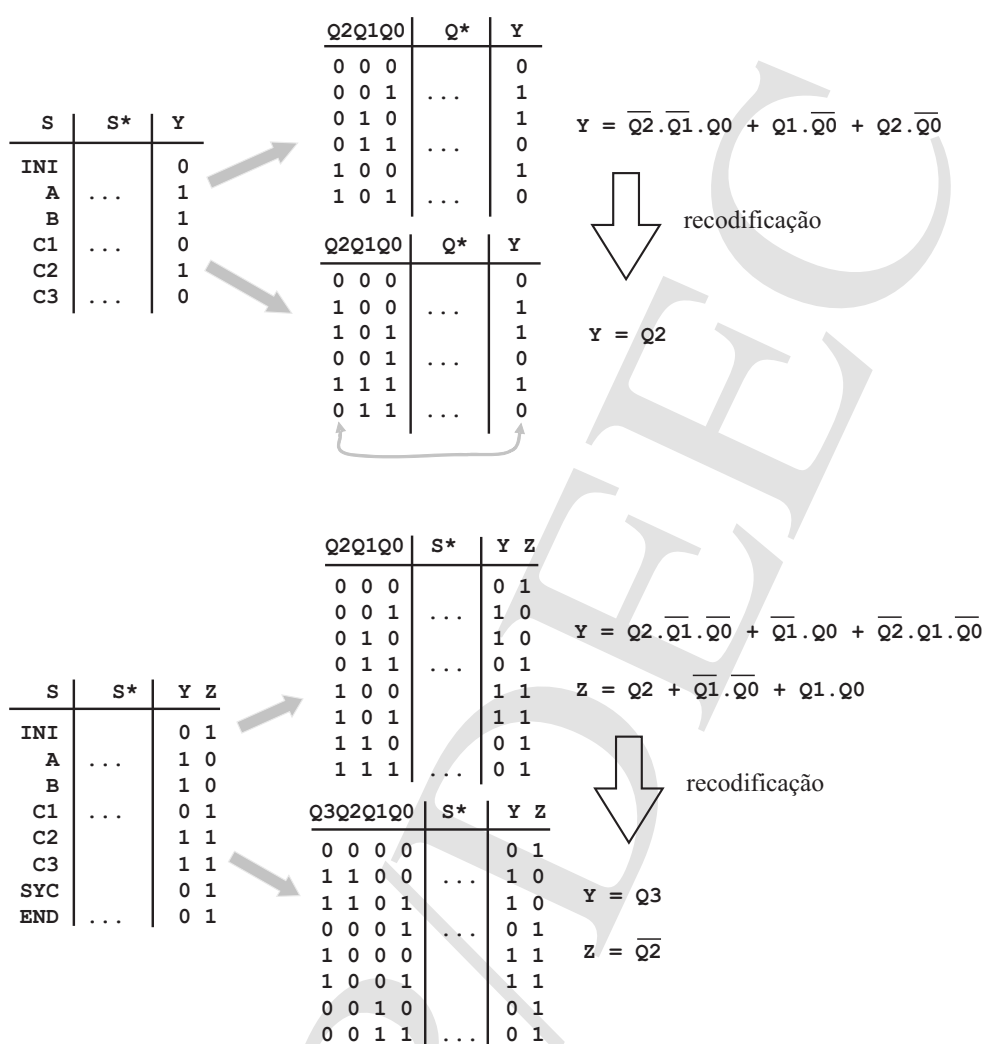


Figura 6.45: Codificando estados de forma a simplificar as funções lógicas que produzem as saídas. No primeiro caso é usado o número mínimo de *flip-flops*, mas no segundo caso é acrescentada uma variável de estado para se conseguir reduzir as funções lógicas que produzem as saídas *Y* e *Z*.

OU com as suas entradas ligadas aos *flip-flops* de estado para os quais cada saída vale 1. A figura 6.46 mostra a forma como numa máquina de estados de Moore implementada com este tipo de codificação podem ser construídos os circuitos lógicos que realizam as saídas.

Para concluir, é importante relembrar que estas regras devem ser entendidas como recomendações que quando seguidas *tendem* a reduzir a complexidade dos circuitos lógicos que realizam uma máquina de estados, mas que não garantem de forma alguma a obtenção de circuitos minimizados.

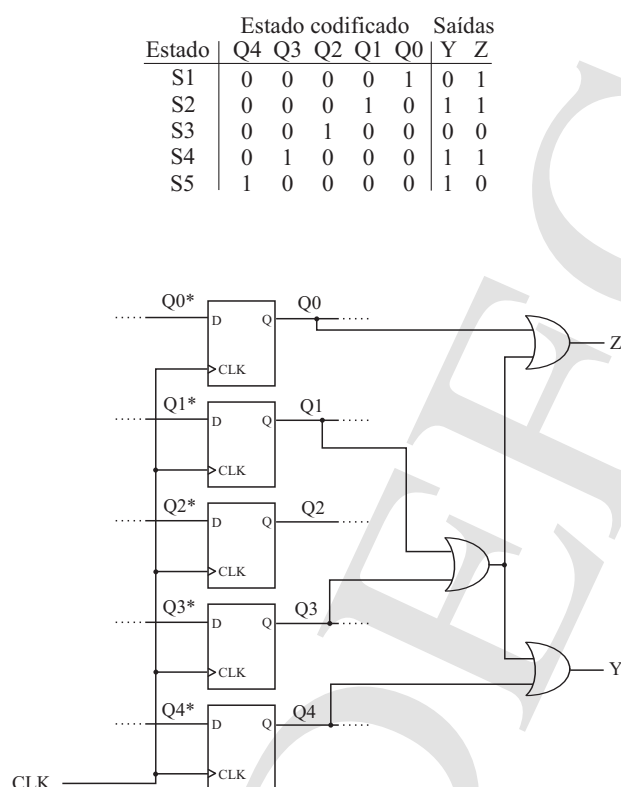


Figura 6.46: Implementação dos circuitos lógicos que realizam as saídas de Moore Z e Y quando é utilizada a codificação *one-hot*.

6.5 Minimização de estados

O primeiro passo no processo de projecto de máquinas de estados finitos consiste na construção de uma especificação formal que identifique claramente os estados, condições de transição entre estados e valores para as saídas e que é geralmente feita com recurso a representações com diagramas de transição de estados. Este é o ponto de partida para o processo de projecto e a escolha correcta de um conjunto mínimo de estados é o primeiro passo para a obtenção de circuitos lógicos eficientes.

Como foi visto anteriormente (secção 6.2), um estado tem implícita uma certa *história* daquilo que no passado ocorreu nas entradas do circuito e que determina um conjunto de acções a realizar nesse estado, bem como os estados que devem ocorrer a seguir. Se dois estados representam exactamente o mesmo, então diz-se que esses estados são equivalentes podendo ser fundidos num só, reduzindo com isso o número total de estados e consequentemente a complexidade do circuito final. Dois estados *S1* e *S2* dizem-se equivalentes se se verificaram simultâneamente as três condições seguintes:

1. As saídas de Moore assumem o mesmo valor em *S1* e em *S2*.

2. As saídas de Mealy têm o mesmo valor em ambos os estados e para iguais condições das entradas.
3. Os estados seguintes a $S1$ são iguais aos estados seguintes a $S2$ e são também iguais as condições que determinam cada transição de estado.

Na figura 6.47 mostra-se um exemplo de um diagrama de transição de estados com saídas de Moore e saídas de Mealy, onde os estados $S1$ e $S2$ podem ser fundidos num único estado $S12$ porque se identificam como sendo equivalentes.

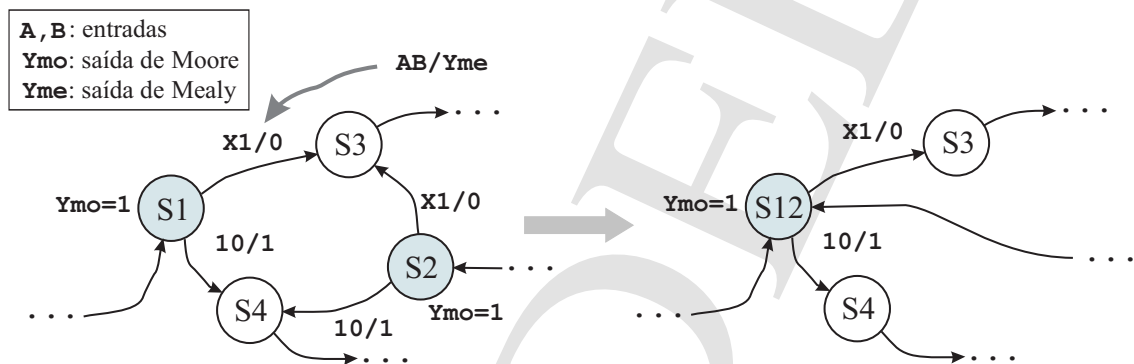


Figura 6.47: Reunião dos estados equivalentes $S1$ e $S2$ num único estado $S12$.

Capítulo 7

Funções sequenciais padrão

A metodologia de projecto de circuitos sequenciais síncronos que foi estudada no capítulo anterior apresenta algumas limitações e só é deve ser aplicada para sintetizar circuitos de pequena complexidade. Não é praticável realizar à mão o projecto de máquinas de estados com muitos estados (para cima de algumas dezenas) porque obrigaria à síntese de circuitos combinacionais com um número elevado de entradas e que não podem ser facilmente minimizados aplicando as técnicas estudadas. Mesmo recorrendo a ferramentas computacionais para sintetizar e otimizar implementações de máquinas de estados, não é desejável tentar construir circuitos demasiadamente complexos como uma única máquina de estados. Como se viu no capítulo anterior, o número de implementações possíveis para uma máquina de estados (o que se traduz na sua complexidade) cresce muito rapidamente com o número de estados. Por exemplo, é muito mais fácil encontrar boas soluções para duas máquinas de estados com 7 estados cada uma, do que para uma única máquina de estados que tenha apenas 9 estados!

Tal como foi estudado para os circuitos combinacionais, o projecto de circuitos sequenciais (síncronos) complexos recorre geralmente à decomposição de um sistema complexo em sub-circuitos mais simples e consequentemente mais fáceis de projectar, e que muitas vezes se identificam com funções padrão para as quais existam já realizações conhecidas e bem estudadas de circuitos que as implementam. Por exemplo, pode ser vantajoso partir uma única máquina de estados em duas ou mais máquinas de estados de menor dimensão e que comuniquem entre si através das suas entradas e saídas.

Neste capítulo será estudado um conjunto de 3 funções sequenciais padrão que podem ser usadas para simplificar o projecto de circuitos sequenciais complexos: registos, contadores e registos de deslocamento. Estas funções existem disponíveis como circuitos electrónicos integrados (por exemplo, dispositivos da série 74), ou então como “componentes” organizados em bibliotecas de ferramentas computacionais para apoio ao projecto

de sistemas digitais.

Um registo não é mais do que um conjunto de *flip-flops* com um sinal de relógio comum, podendo ser entendido como um *flip-flop* de vários *bits*. Tem por função armazenar um conjunto de vários *bits*, representando, por exemplo, um número inteiro com sinal, um carácter alfabético segundo o código ASCII ou a cor de um ponto elementar de imagem (*pixel*).

Um contador é um circuito sequencial síncrono que percorre ciclicamente uma sequência de estados produzindo num conjunto de saídas uma sequência determinada de valores binários. Existem contadores binários que geram a sequência natural de todos os 2^N números binários de N bits, contadores módulo M que contam de 0 a $M - 1$, contadores que geram sequências (pseudo-)aleatórias, contadores que contam para cima ou para baixo, etc. De uma forma geral, o diagrama de transição de estados de um contador apresenta uma sequência de estados dispostos em anel, que pode ser quebrada por activação de entradas de controlo (por exemplo, uma entrada de *reset* pode servir para colocar o contador no estado 0000).

Um registo de deslocamento é um registo que, para além de armazenar uma palavra de N bits, permite deslocar esses bits para a direita ou para a esquerda, sincronizado com as transições activas do sinal de relógio. Uma função muito importante realizada por registos de deslocamento é a conversão de dados em paralelo para dados em série ou vice-versa, necessário em sistemas de comunicações digitais.

7.0.1 Registos

Um registo é formado por um conjunto de *flip-flops* que têm o mesmo sinal de relógio. As saídas Q contêm o valor armazenado no registo e nas entradas D é colocado o conjunto de *bits* que na próxima transição activa do sinal de relógio será carregado para as saídas Q (figura 7.1). Um registo pode ser representado por um símbolo lógico em que o seu número de *bits* é especificado pelos nomes atribuídos às entradas e saídas: a entrada $D[2 : 0]$ representa um conjunto de 3 *bits*, sendo os *bits* individuais identificados por $D[0]$, $D[1]$ e $D[2]$ ¹.

A este circuito podem ser acrescentados vários sinais de controlo que permitem seleccionar para a entrada D outros valores: sinais de inicialização (*reset* para carregar zero ou *set* para carregar 1) ou sinal de habilitação (*enable* ou *load*) que permite activar o carregamento de um novo valor. Todos estas sinais de controlo podem ser implementados

¹Existem várias outras formas de representar sinais lógicos com mais de 1 *bit*; neste texto será adoptada esta por ser a mesma que é usada pelas ferramentas de projecto da Xilinx a utilizar nas aulas práticas laboratoriais.

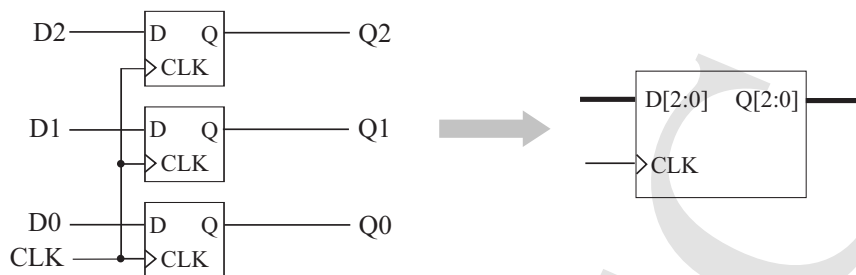


Figura 7.1: Um registo simples de 3 bits construído com *flip-flops* do tipo D.

à custa de um *multiplexer* que escolhe o valor adequado a aplicar na entrada *D*, sendo nesse caso a operação do *flip-flop* completamente síncrona porque a acção definida por esse sinal de controlo só acontece quando ocorre a transição activa do sinal de relógio. A figura 7.2 mostra um registo de 3 bits com entradas de controlo *RESET*, *SET* e *LOAD*. Recorrendo a *flip-flops* que tenham entradas de controlo assíncronas para colocar a saída *Q* a 1 e a 0, podem-se construir registos cuja saída pode ser colocada em zero ou em 1 independentemente do que acontece no sinal de relógio.

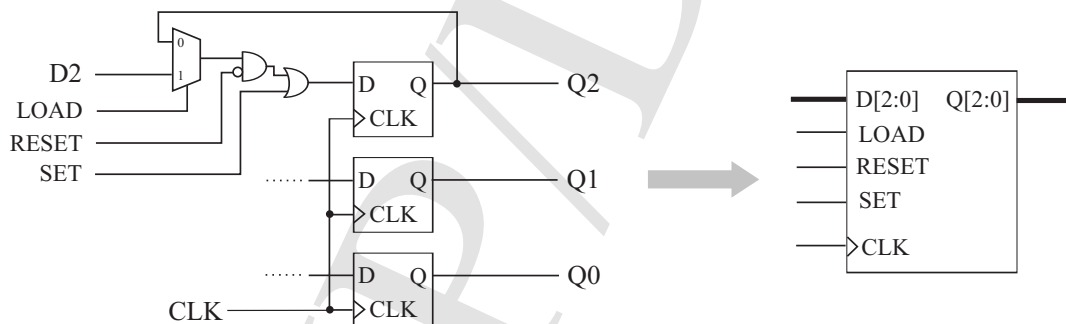


Figura 7.2: Modificação a efectuar no registo da figura 7.1 para implementar as entradas *RESET*, *SET* e *LOAD* (apenas se mostra o circuito para a entrada *D2*; os circuitos nas entradas *D1* e *D0* seriam iguais). Note que a entrada *SET* é que tem maior prioridade, seguida de *RESET* e por fim *LOAD*.

Registos em circuitos integrados da série 74

A figura 7.3 mostra o símbolo lógico de 4 circuitos integrados da série 74 que oferecem a função de registo. O '175 é um registo de 4 bits, com entrada de *CLEAR* assíncrona e activa com o nível lógico 0, dispondo das saídas *Q* e \bar{Q} (note a identificação das entradas e saídas activas com o valor lógico zero). O '174 é um registo de 6 bits com entrada

CLEAR igual à do '175, mas que não dispõe das saídas negadas. O '374 é um registo de 8 bits com entrada de activação da saída (*output enable*) activa com o nível lógico 0. Isto significa que as saídas de cada *flip-flop* são ligadas para o exterior através de um *buffer* de 3 estados, que só é ligado quando a entrada *OE* for activada ($\overline{OE} = 0$). Quando essa entrada estiver com o nível lógico 1, as saídas apresentam um estado de alta impedância que é equivalente a estarem electricamente desligadas do resto do circuito. O '377 é um registo de 8 bits com entrada de permissão de carregamento (*EN* ou *enable*) activa com o valor lógico 0: quando ocorrer a transição activa de relógio e $\overline{EN} = 1$, as saídas *Q* mantêm o estado anterior; se $\overline{EN} = 0$ as saídas *Q* são carregadas com os valores presentes nas entradas *D* correspondentes.

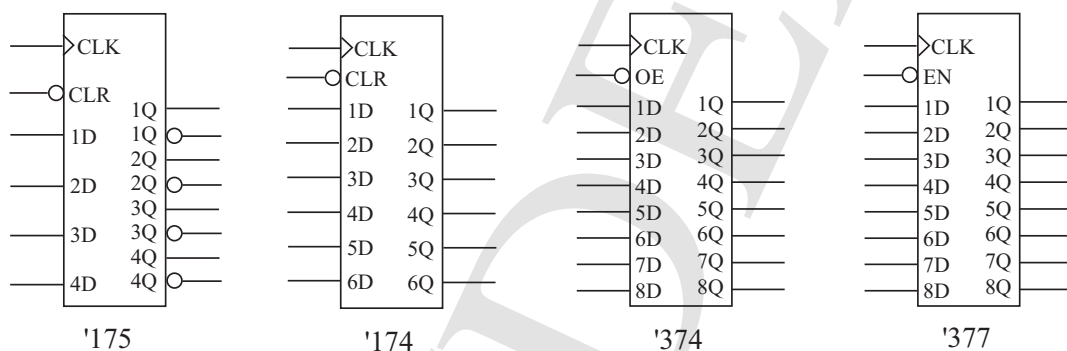


Figura 7.3: Alguns circuitos integrados da série 74 com a função de registo.

7.1 Contadores

Um contador é um circuito sequencial que percorre ciclicamente um conjunto de estados, produzindo nas suas saídas uma sequência de valores que caracterizam o tipo de contador. Podemos ter contadores binários que apresentam uma sequência de valores desde 0 a $2^N - 1$ (em que N é o número de *bits* do contador), contadores módulo M que contam de 0 a $M - 1$ (por exemplo, um contador módulo 10 é um contador que conta de 0 até 9), contadores que contam para cima e para baixo ou contadores que produzem uma sequência de valores segundo códigos diferentes do código binário (por exemplo, código Gray que apresenta a característica de trocar apenas um *bit* entre dois estados consecutivos).

Um circuito contador serve naturalmente para contar coisas, como por exemplo os carros que entram e saem de um parque de estacionamento ou então medir tempo contando ciclos do seu sinal de relógio. Para além disso, pode também servir como base para construir certos tipos de máquinas de estados recorrendo a um conjunto reduzido de

circuitos lógicos adicionais.

7.1.1 Contador em cascata (assíncrono)

Um circuito contador muito simples com N bits pode ser construído ligando N *flip-flops* do tipo T da forma que se mostra na figura 7.4: o *flip-flop* que produz o *bit* i troca de estado quando o *bit* $i - 1$ passa de 1 para 0. Como se pode ver pelo diagrama temporal mostrado na figura, o valor apresentado no conjunto de saídas do contador segue uma sequência binária natural crescente desde 0 até 2^{N-1} .

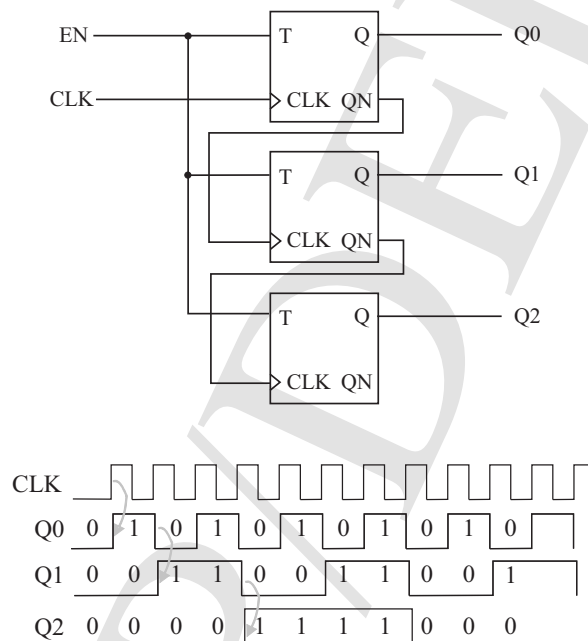


Figura 7.4: Contador assíncrono com *flip-flops* do tipo T .

Este contador apresenta vários problemas que colocam algumas limitações à sua utilização, resultantes do facto de os seus *flip-flops* terem diferentes sinais de relógio (diz-se por isso que é um contador assíncrono). Em primeiro lugar, o facto de cada *flip-flop* ter um sinal de relógio diferente e que depende do estado do *flip-flop* anterior faz com que seja difícil implementar a operação de carregamento do contador com um estado determinado. Como será visto mais tarde, esta função é fundamental para que se possam construir contadores que produzam nas saídas sequências de valores diferentes da sequência binária natural. Para além disto, este tipo de contador apresenta um problema mais grave que não é evidenciado no diagrama temporal mostrado na figura 7.4 e que é devido ao tempo que cada *flip-flop* demora a actualizar a sua saída Q depois do sinal de relógio ter sido actuado. Como a saída de um *flip-flop* é usada como entrada de relógio do *flip-flop* seguinte,

então esse tempo de atraso vai sendo acumulado para os *bits* mais significativos. Dependendo da frequência do sinal de relógio e do número de *bits* do contador, pode acontecer que no conjunto de saídas do contador nunca chegue a aparecer o estado correcto porque cada *bit* é actualizado sempre mais tarde do que o *bit* anterior. A figura 7.5 exemplifica essa situação para um contador de 3 *bits*, considerando que cada *flip-flop* introduz um atraso ligeiramente inferior a metade do período do sinal de relógio.

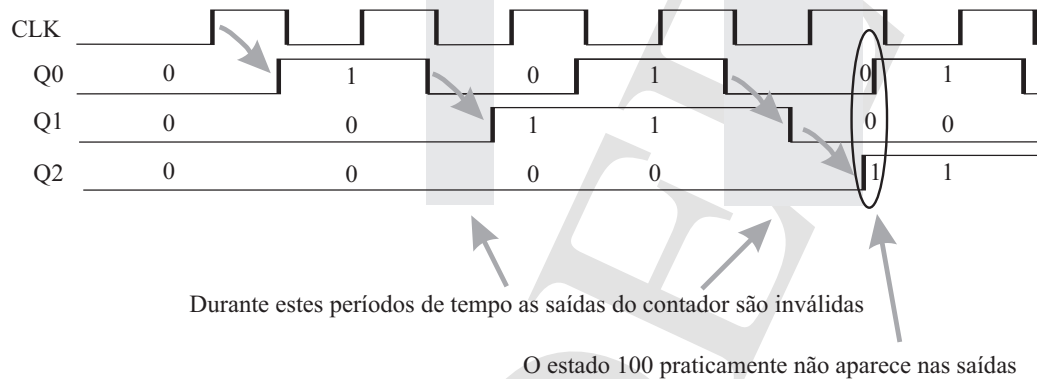


Figura 7.5: Diagrama temporal mostrando o comportamento errado de um contador assíncrono provocado pela acumulação dos tempos de atraso introduzidos pelos *flip-flops*.

Naturalmente que o problema ilustrado na figura 7.5 pode não se colocar se o período do sinal de relógio for muito maior do que os tempos de propagação dos *flip-flops*. Embora continue a ocorrer a acumulação dos tempos de propagação para os *bits* de maior significância, esse tempo pode ser insignificante face ao período do relógio e não perturbar o funcionamento do circuito. Imagine, por exemplo, que um contador de 128 *bits* construído desta forma e com as suas saídas ligadas a um conjunto de 128 LEDs é actuado por um sinal de relógio com uma frequência de 2Hz (período de 0.5s) e que cada *flip-flop* apresenta um tempo máximo de propagação de 10ns. Como o tempo necessário para que o *bit* mais significativo seja actualizado é igual a 128 vezes o atraso de cada *flip-flop*, só ao fim de 0.00128 ms após a transição activa do sinal de relógio é que o contador apresenta um estado correcto. No entanto, apesar deste pequeno atraso não ser perceptível na sequência de estados que se pode visualizar nos LEDs, seria suficiente para que o circuito não trabalhasse correctamente a frequências superiores a 781250 Hz ($1/0.00000128$).

7.1.2 Contador síncrono

O problema identificado no contador anterior é resolvido pelo contador síncrono, que tem a característica de ter todos os seus *flip-flops* actuados pelo mesmo sinal de relógio. Um contador síncrono pode ser visto como uma máquina de estados em que o estado seguinte é obtido adicionando 1 ao valor representado pelo estado presente. Isto pode ser facilmente construído recorrendo a dois blocos já estudados: um registo e um somador (figura 7.6). No entanto, analisando a sequência de valores que se pretende obter nas saídas de um contador binário, pode-se concluir que o próximo estado do *bit* i , Q_i^* , é igual a $\overline{Q_i}$ quando todos os *bits* anteriores a Q_i são iguais a 1, sendo mantido igual a Q_i no caso contrário. Assim, pode-se construir um contador síncrono usando *flip-flops* do tipo T, onde a entrada T do *flip-flop* Q_i é controlada pela função lógica AND de todos os *bits* anteriores a Q_i , (Q_0, \dots, Q_{i-1}). A figura 7.7 mostra o esquema lógico deste circuito.

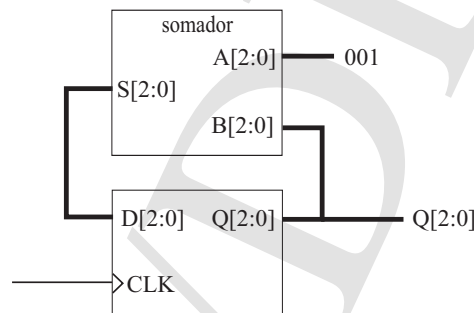


Figura 7.6: Um contador síncrono com um registo e um somador.

7.1.3 Contador síncrono com carregamento paralelo

O circuito contador estudado na secção anterior apresenta um problema motivado pela utilização de *flip-flops* do tipo T: não é fácil modificar aquele circuito de forma a conseguir carregar para os *flip-flops* um estado determinado porque o estado seguinte daquele tipo de *flip-flop* é sempre função do seu estado anterior. Para que se acrescentar uma entrada de controlo que permita carregar os *flip-flops* do contador com valores dados, é conveniente modificá-lo de maneira a usar *flip-flops* do tipo D, da forma que se mostra na figura 7.8.

Para acrescentar um sinal de controlo que permita efectuar um carregamento dos *flip-flops* do contador, basta agora acrescentar *multiplexers* nas entradas D dos *flip-flops* que permitam escolher o próximo estado a carregar para o *flip-flop*, entre o próximo estado natural do contador ou um conjunto de valores lógicos colocados nas entradas (figura 7.9).

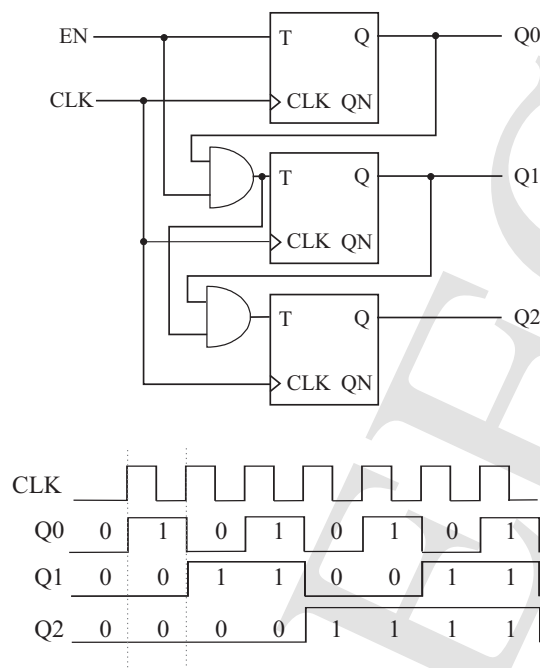


Figura 7.7: Um contador síncrono com *flip-flops* tipo T.

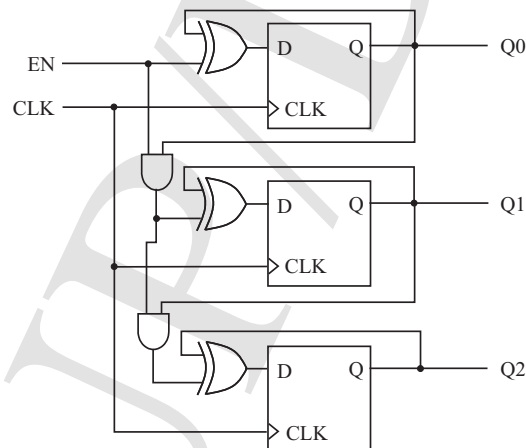


Figura 7.8: Um contador síncrono com *flip-flops* tipo D.

7.1.4 Algumas aplicações com contadores

A função contador existe disponível em vários circuitos integrados da série 74. Um desses circuitos é o '163 que é um contador síncrono de 4 *bits* com entrada para carregamento paralelo e sinal de inicialização síncrono (carregamento com zeros). Tem também duas entradas de habilitação e uma saída que é activada quando o estado apresentado nas saídas é 1111, o que pode ser usado para ligar em cascata várias destas unidades para se

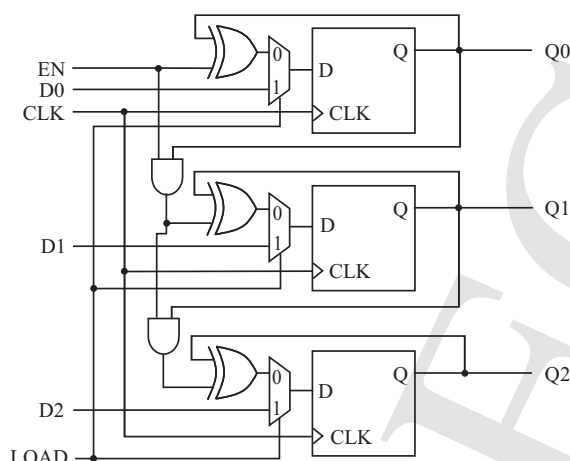


Figura 7.9: Contador síncrono com carregamento paralelo.

obter circuitos semelhantes com um número maior de *bits*. A figura 7.10 mostra o símbolo lógico e a tabela de transição de estados que traduz o funcionamento deste circuito.

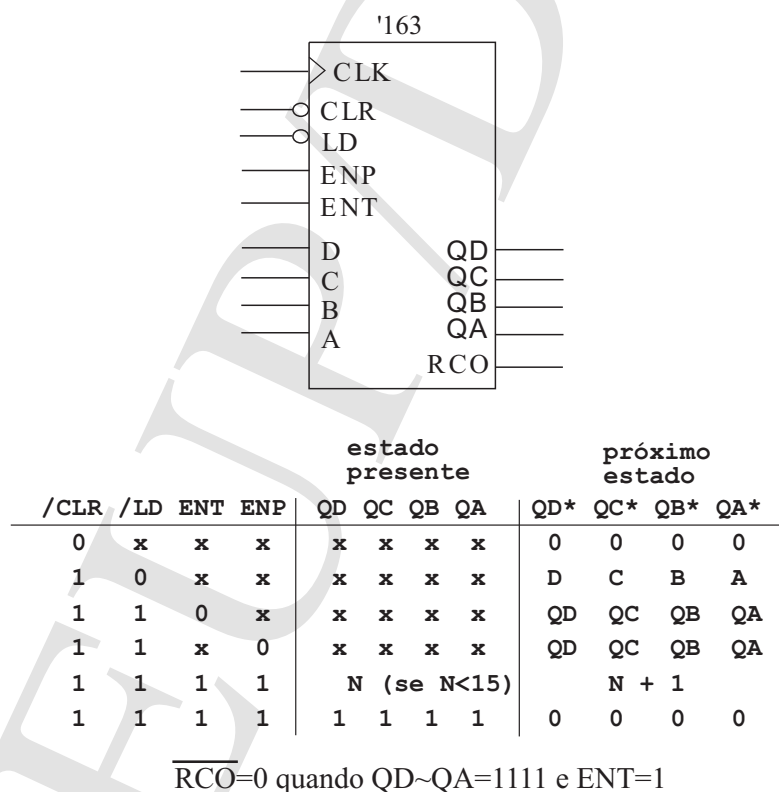
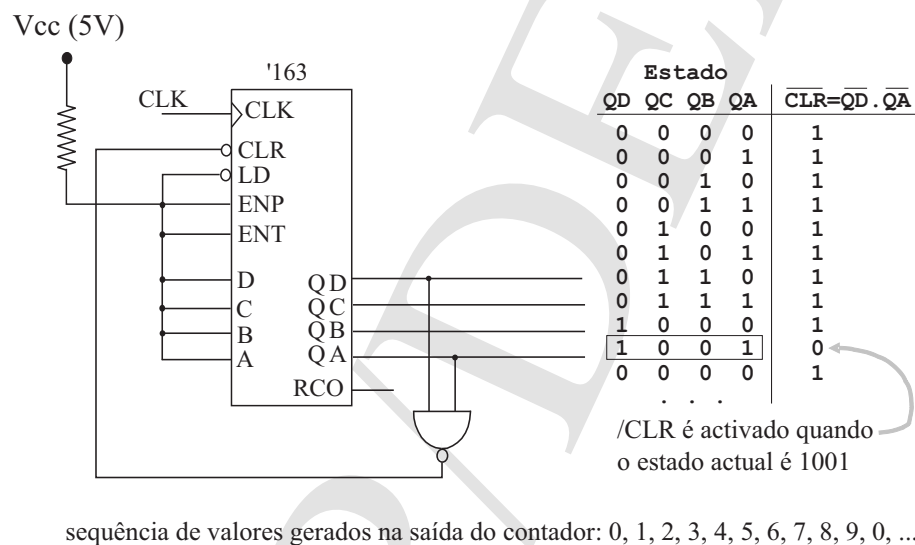


Figura 7.10: O contador de 4 *bits* '163 e a tabela que descreve o seu funcionamento.

Para além da função básica de contagem de números naturais, um contador binário

pode ser modificado facilmente para produzir outras sequências de valores nas suas saídas, ligando as entradas que intervêm na determinação do próximo estado (no caso do '163, todas as entradas excepto CLK) a circuitos lógicos que operam sobre as saídas do contador (ou seja, o estado presente do circuito). Na figura 7.11 mostra-se como se pode modificar um contador binário de 4 *bits* para obter um contador módulo 10 (conta apenas de 0 a 9) necessitando para isso de apenas uma porta lógica NAND ligada à entrada \overline{CLR} , que funciona como detector do estado 1001. Note que neste circuito esta porta lógica NAND não se comporta como um comparador com 9 (1001) porque na verdade os 2 *bits* do meio não são comparados. No entanto, o estado 1001 que se pretende detectar é, na sequência de valores que o contador produz, o primeiro em que os dois *bits* em causa são iguais a 1.



sequência de valores gerados na saída do contador: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, ...

Figura 7.11: Um contador módulo 10: quando o estado presente é 1001 (9 em decimal), a entrada \overline{CLR} é colocada em zero e assim o próximo estado será 0000

Outras sequências de valores podem ser obtidas à custa de circuitos que actuem na entrada \overline{LD} e nas entradas $DCBA$ para carregar os *flip-flops* do contador com um valor dado. O circuito da figura 7.12 mostra uma aplicação em que os dois sinais de controlo \overline{LD} e \overline{CLR} estão ligados a circuitos lógicos cujas saídas são função do estado presente do contador. Consegue-se deste forma criar circuitos do tipo contador que produzem sequências de valores diferentes da sequência binária natural.

Os valores produzidos pelo contador podem ainda ser modificados por circuitos lógicos que transformem os códigos binários gerados pelo contador noutras sequências diferentes. Na figura 7.13 mostra-se como se pode obter um contador que produz uma sequência de códigos *one-hot* ligando apenas um circuito descodificador do tipo '138 directamente às saídas do contador.

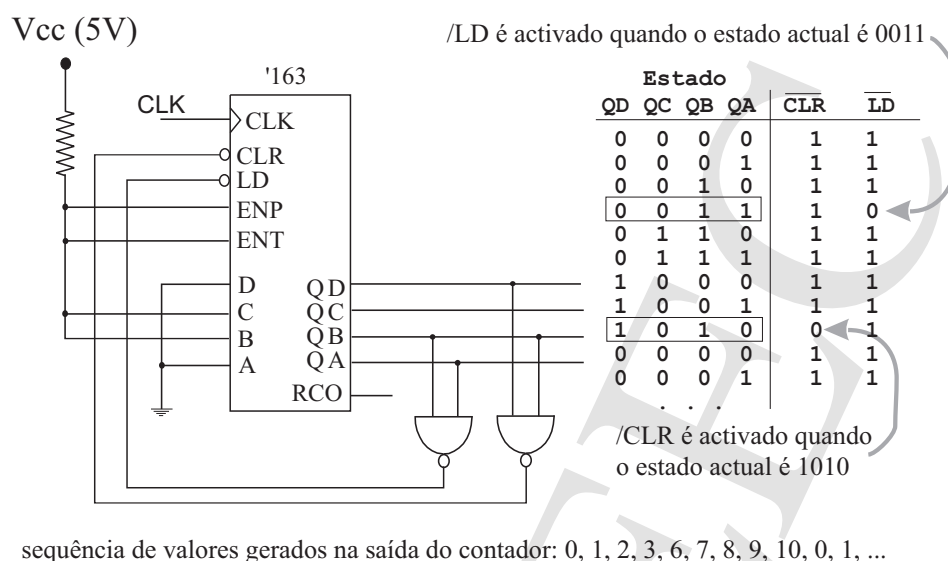


Figura 7.12: Modificando a sequência de contagem por actuação na entrada \overline{LD} e \overline{CLR} .

A saída *RCO* (o nome *RCO* vem de *ripple count output*) deste contador é activada quando o estado presente é igual a 1111. Uma das aplicações imediatas deste sinal consiste em facilitar a ligação em cascata de vários destes circuitos para criar contadores com um número maior de *bits*. O circuito mostrado na figura 7.14 implementa um contador de 8 *bits* usando dois circuitos do tipo '163 e sem necessitar de qualquer circuito lógico adicional.

Uma aplicação corrente de circuitos contadores é como geradores de sinais de relógio, produzindo frequências que são obtidas dividindo a frequência do sinal de relógio por factores inteiros. Como mostra o exemplo da figura 7.15, o *bit* menos significativo de um contador binário troca de estado em cada transição activa do sinal de relógio e assim a sua frequência é igual a metade da do sinal de relógio que alimenta o contador. O *bit* seguinte troca de estado a cada duas transições de relógio e por isso a sua frequência é igual à frequência do sinal de relógio dividida por 4. Por cada *bit* que se avance em direcção ao *bit* mais significativo a frequência de troca de estado vai sendo sucessivamente dividida por 2. Note que para além de dividir a frequência do sinal de relógio, cada *bit* de saída do contador apresenta um ciclo útil de 50%², o que quer dizer que o tempo em que o sinal está no nível lógico alto é o mesmo em que está no nível lógico baixo. Em certas classes de circuitos síncronos os sinais de relógio devem satisfazer esta condição para que o circuito funcione correctamente.

²*duty-cycle* ou a relação entre o tempo em que o sinal está em 1 e a duração do período do relógio

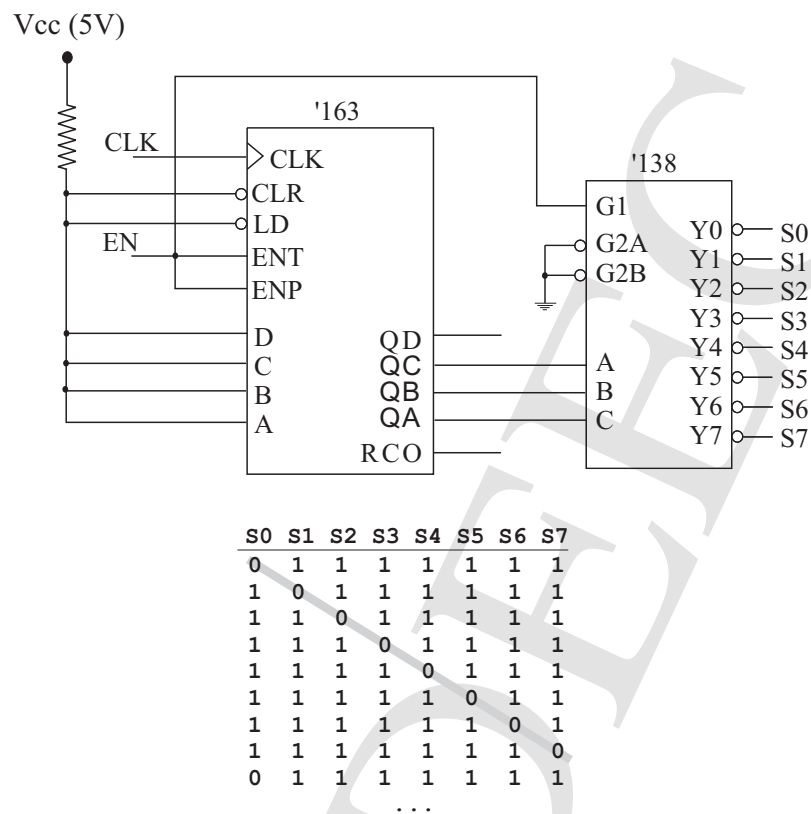


Figura 7.13: Associando um descodificador '138 a um contador binário para criar um gerador de sequência *one-hot*.

A entrada \overline{CLR} do contador '163 é uma entrada de controlo síncrona. Isto quer dizer que a sua acção (carregar zeros para as saídas do contador) apenas acontece na transição de relógio em que esse sinal estiver activo. O contador '161 apresenta um interface igual ao do '163, com a diferença que a sua entrada \overline{CLR} é assíncrona: a sua acção acontece logo que esse sinal é activado, não dependendo daquilo que ocorre no sinal de relógio. Qual é o efeito prático desta diferença no comportamento? Tomemos como exemplo o circuito contador modulo 10 que se apresentou na figura 7.11. Neste circuito, a entrada \overline{CLR} de um '163 é activada quando o estado presente é igual a 1001 (ou 9 em decimal), de forma que o próximo estado será 0000. Se fosse usado nesse circuito um contador do tipo '161, logo que aparecesse na saída o estado 1001 a entrada \overline{CLR} seria activada, colocando na saída o valor 0000 sem que para isso fosse necessário esperar pela próxima transição de relógio. A consequência prática deste comportamento é que o valor 1001 apenas existira durante um tempo muito curto, o suficiente para a porta lógica NAND activar a sua saída e o contador reagir à actuação do seu sinal de inicialização assíncrono (\overline{CLR}). Como estes tempos são muito curtos e geralmente muito inferiores ao período do relógio que comanda

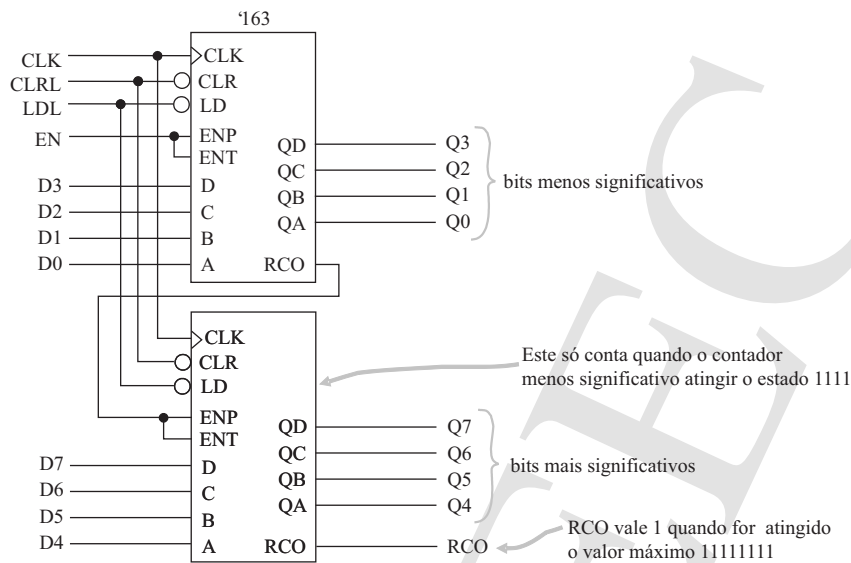


Figura 7.14: Ligação em cascata de 2 contadores de 4 *bits* para criar um contador de 8 *bits*.

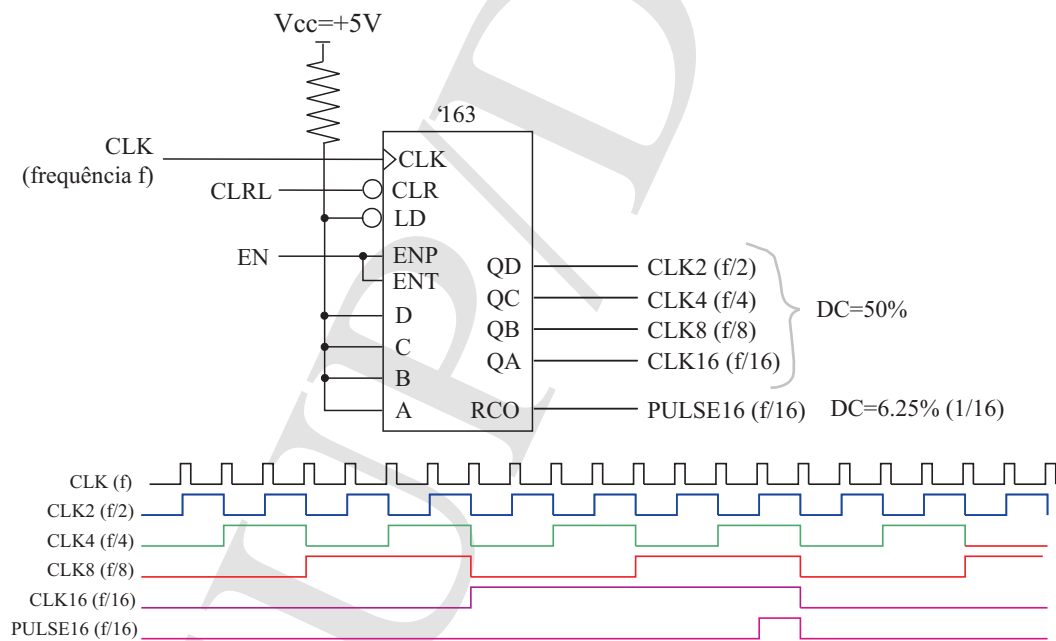


Figura 7.15: Um contador de 4 *bits* como divisor de frequência; note que para além de ser dividida a frequência do sinal de relógio que alimenta o contador, os sinais obtidos nas saídas Q_0 Q_3 apresentam um ciclo útil de 50%, ou seja o tempo em 1 é igual ao tempo em zero.

o contador, pode-se afirmar que na verdade o estado 1001 praticamente não chega a ser

visto nas saídas do contador e que estas passam logo do estado 1000 para o estado 0000.

7.1.5 Contador ascendente e descendente

O contador construído com *flip-flops* do tipo D e com carregamento paralelo que se apresentou na figura 7.9 pode ser facilmente modificado para produzir uma sequência binária decrescente. Seguindo um raciocínio semelhante ao que foi feito para obter esse contador, podemos concluir que para produzir uma sequência binária decrescente, cada *bit* deve trocar de estado sempre que todos os *bits* à sua direita (ou seja, com menor significância) sejam iguais a zero (lembre-se que para obter uma sequência crescente a condição era todos esses *bits* serem iguais a um). Assim, se no circuito na figura 7.9 se inverter a entrada de cada porta AND que liga à saída do *flip-flop* anterior, obtém-se o circuito mostrado na figura 7.16 que produz nas suas saídas uma sequência binária decrescente.

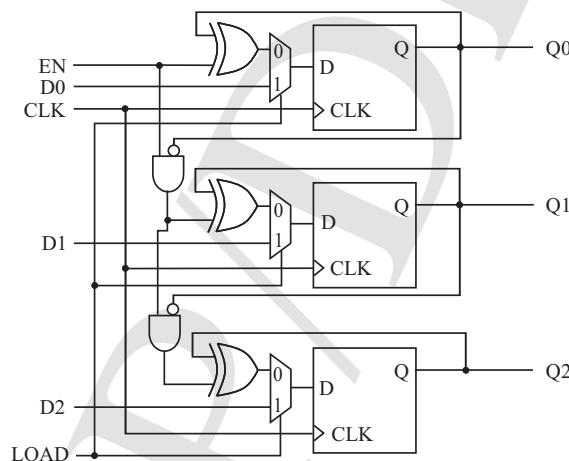


Figura 7.16: Um contador de 3 *bits* decrescente com entrada de carregamento paralelo.

O '169 é um circuito contador que combina as duas funções, contando para cima ou para baixo conforme o estado de uma entrada de controlo que selecciona a direcção de contagem. A figura 7.17 mostra o símbolo lógico deste circuito e a tabela de transição de estados que descreve o seu funcionamento.

7.2 Registos de deslocamento

O segundo tipo de função sequencial padrão que será estudada neste capítulo é o registo de deslocamento (ou *shift-register*). Um registo de deslocamento é um circuito síncrono

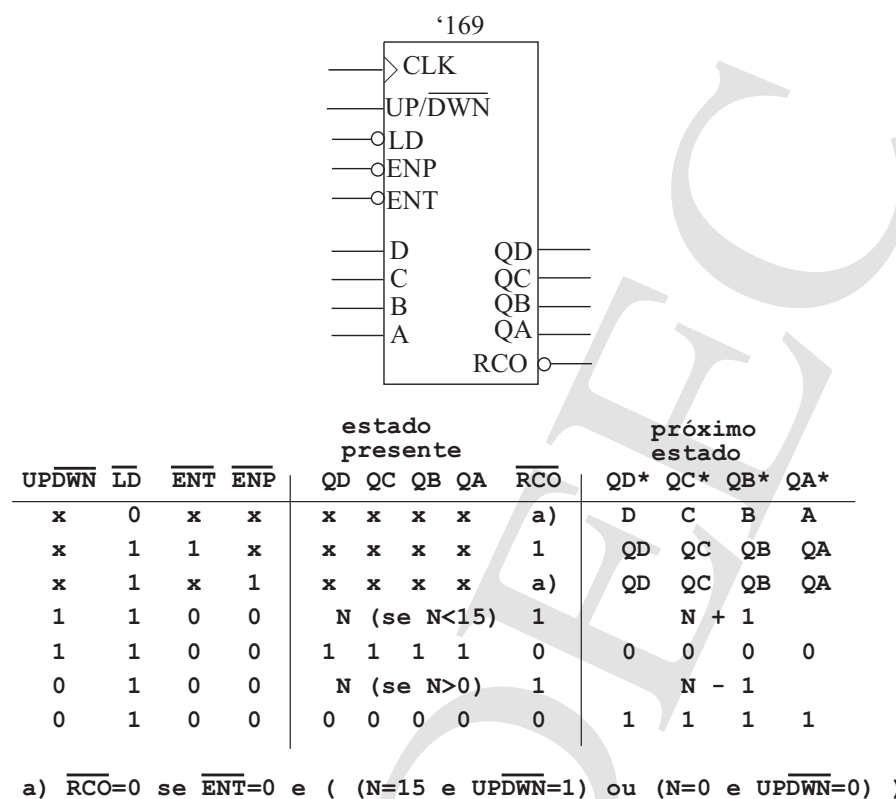


Figura 7.17: O contador *up-down* '169: se *UPDWN* for igual a 1 conta para cima, caso contrário conta para baixo.

formado por um conjunto de *flip-flops* do tipo D ligados de forma que a saída *Q* de um alimenta a entrada *D* do seguinte, como se mostra na figura 7.18. A cada transição do sinal de relógio, o conteúdo de cada *flip-flop* (0 ou 1) é deslocado para o seguinte, e o primeiro *flip-flop* carrega o que lhe for apresentado na sua entrada *D*.

De forma semelhante ao que foi feito para o circuito contador, é também fácil acrescentar ao registo de deslocamento um conjunto de entradas que permitam carregar os seus *flip-flops* com um conjunto de dados. O circuito mostrado na figura 7.19 introduz a entrada LD/\overline{SHIFT} que permite activar o carregamento paralelo ou a função de deslocamento e o conjunto de entradas de dados $D_3D_2D_1D_0$ onde devem ser colocados os dados a carregar para o registo.

O registo de deslocamento é uma peça fundamental em processos de transmissão digital de dados, onde informação presente sob a forma de *bytes* (ou palavras de outras dimensões) deve ser partida de forma a enviar um *bit* de cada vez através de um qualquer meio de transmissão (um par de fios eléctricos ou uma ligação de rádio, por exemplo³). A

³Actualmente os processos de modulação usados pelos sistemas de transmissão digital transmitem

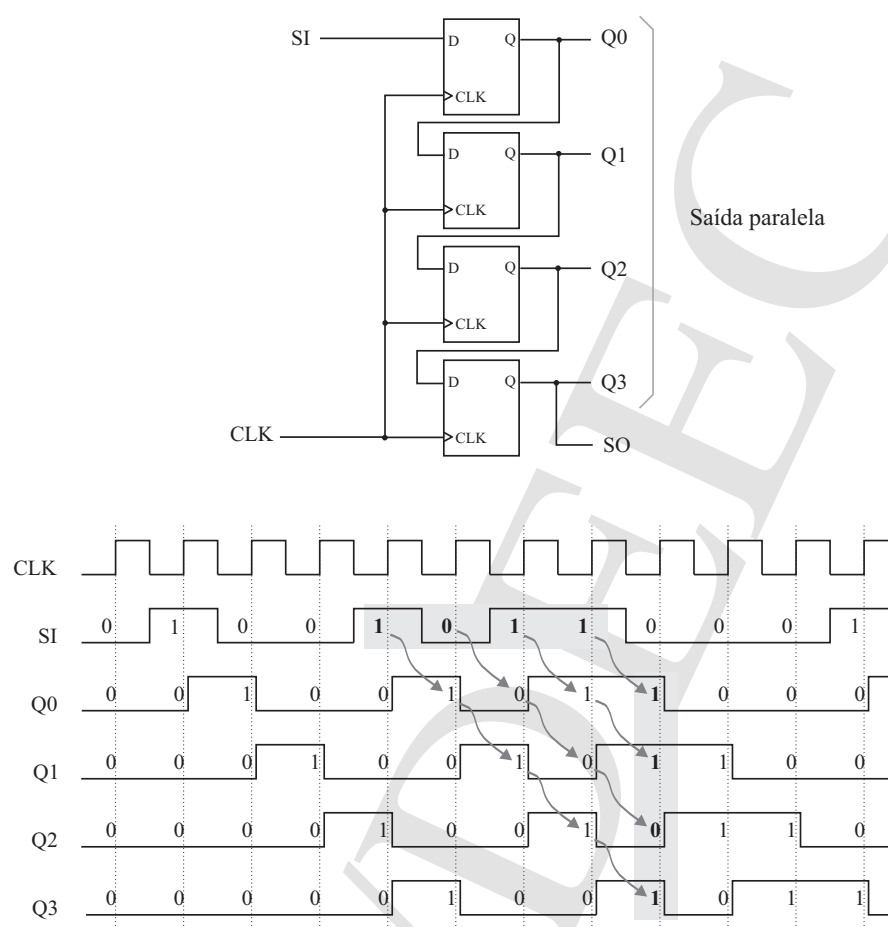


Figura 7.18: Um registo de deslocamento de 4 *bits*: uma sequência de *bits* colocada em série na entrada *SI* é apresentada em paralelo nas saídas paralelas.

figura 7.20 mostra um sistema de transmissão digital muito simples, usando dois registos de deslocamento. O registo de deslocamento do emissor é carregado com o conjunto de 4 *bits* que se pretende transmitir activando a sua entrada LD/\overline{SHIFT} , bastando depois produzir 4 ciclos no sinal de relógio para os 4 *bits* sejam enviados em série e apareçam, em paralelo, na saída do registo de deslocamento do receptor. Note que o registo de deslocamento do receptor é actuado pelo flanco descendente do sinal de relógio, enquanto que o do emissor usa o flanco ascendente. Desta forma, o emissor coloca os seus dados da sua saída série quando o sinal de relógio passa de 0 para 1, e o receptor só captura esse dado e desloca o seu registo de deslocamento após meio período do sinal de relógio, quando este passa de 1 para 0. Desta forma o processo de recepção torna-se menos sujeito a erros que possam resultar de diferentes atrasos no sinal que transporta os dados e no grupos de vários bits de uma só vez, que são codificados em certas características de um sinal como a sua frequência, amplitude, fase, ou combinações entre elas.

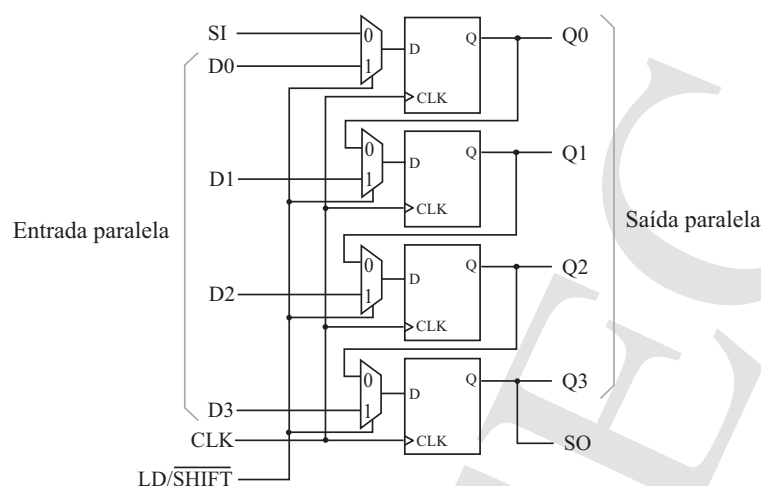


Figura 7.19: Um registo de deslocamento de 4 *bits* com entrada para carregamento paralelo.

señal que transporta o relógio.

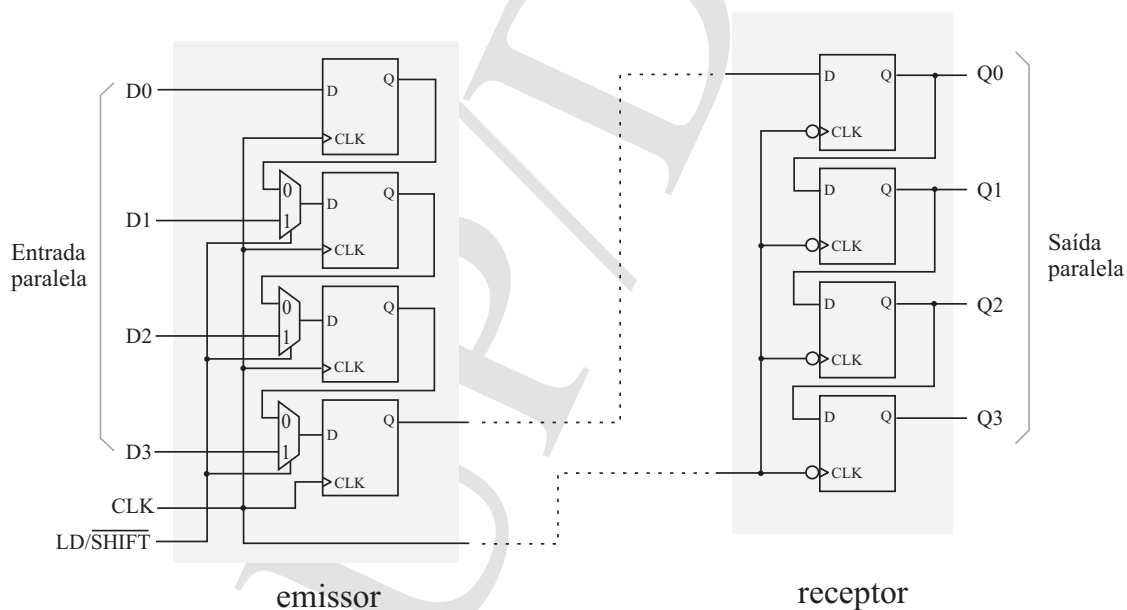


Figura 7.20: Um sistema simples de transmissão digital usando dois registos de deslocamento.

Este método de transmissão de dados em série é chamado de *síncrono* porque juntamente com os dados é também transmitido o sinal de relógio que permite sincronizar o emissor e o receptor. Este processo é geralmente usado para transmitir informação digital sobre ligações curtas construídas sobre condutores eléctricos. Transmissões de dados

digitais que necessitem de ligações longas, feitas sobre a linha telefónica ou através de sinais de rádio, usam normalmente sistemas de comunicação assíncrona em que o sinal de relógio não é transmitido explicitamente mas sim embebido nos dados transmitidos, de onde tem de ser recuperado.

7.2.1 Algumas aplicações com registos de deslocamento

A função registo de deslocamento existe também como circuito integrado da série 74: o '194 é um registo de deslocamento universal de 4 *bits* que, para além da função de carregamento paralelo e inicialização, permite deslocar as suas saídas em ambos os sentidos. A figura 7.21 apresenta o símbolo lógico deste circuito e a tabela de transição de estados que resume o seu funcionamento. As entradas *S1* e *S0* definem o modo de funcionamento do circuito e as entradas *LIN* e *RIN* representam a entrada série para deslocamentos para a esquerda e para a direita, respectivamente. Note que a designação que foi adoptada para as saídas deste circuito ($Q_A Q_B Q_C Q_D$) considera uma ordem de significância das saídas que é diferente da assumida nos circuitos contadores: como o deslocamento para a direita se entende como sendo se Q_A para Q_D , então deve ser entendido que a saída Q_A representa o *bit* mais significativo e Q_D o *bit* menos significativo.

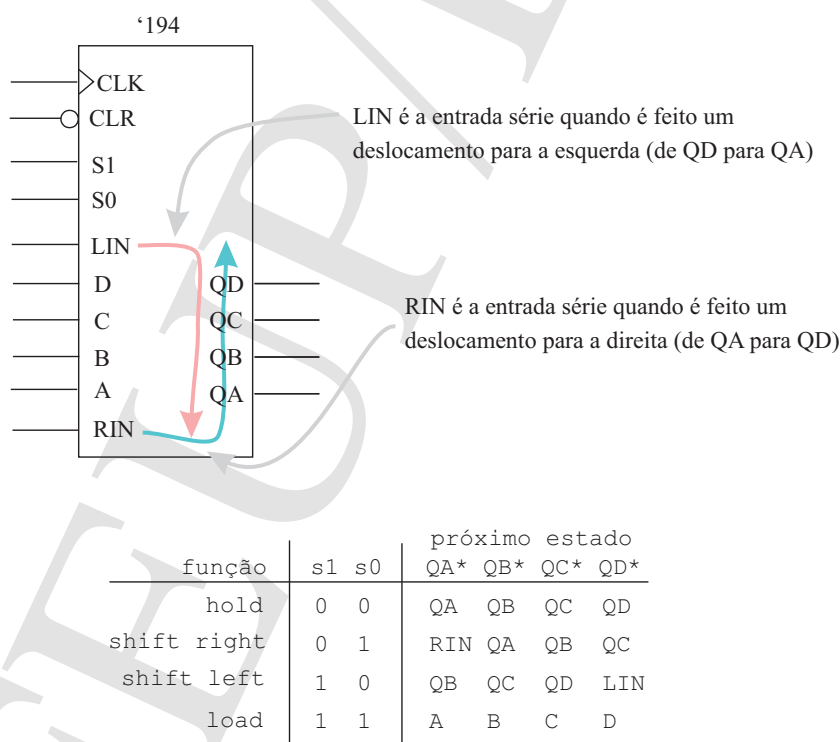


Figura 7.21: Um registo de deslocamento universal de 4 *bits*: '194.

O circuito da figura 7.22 mostra uma aplicação de um registo de deslocamento como um detector de uma determinada sequência de *bits* que são apresentados na sua entrada de forma síncrona com o sinal de relógio. A função realizada por este circuito é a mesma que serviu de introdução ao estudo do processo de projecto de máquinas de estados apresentado na secção 6.2, página 140: detectar a sequência 1011. A saída Z deste circuito é activada logo que o estado presente nas saídas do registo de deslocamento seja igual ao padrão que se pretende detectar. No entanto, a saída será activada sempre que na sequência de *bits* colocados na entrada do registo de deslocamento ocorra o padrão 1011, mesmo que o primeiro 1 de uma seja o último 1 de outra sequência detectada anteriormente (sequências parcialmente sobrepostas).

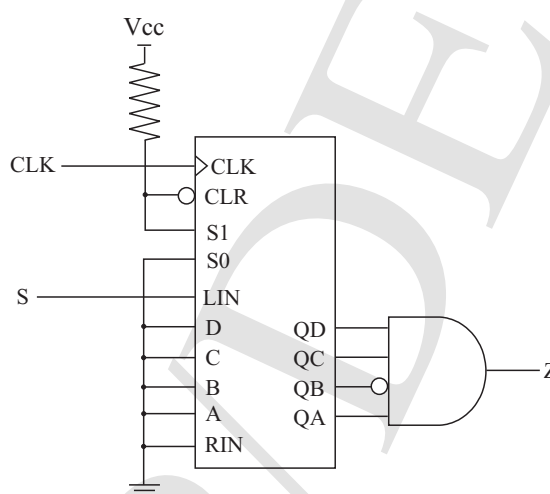


Figura 7.22: Uma máquina de estados construída com base num registo de deslocamento '194 para detectar a sequência 1011.

O circuito anterior pode ser modificado por forma a que apenas sejam detectadas sequências inteiras que não partilhem nenhum *bit* com outras sequências detectadas. Uma forma simples de fazer isso consiste em “estragar” a parte da sequência que foi detectada sempre que é activada a saída Z , carregando zeros para os *flip-flops* do registo de deslocamento. No entanto, o *bit* Q_D deve ser carregado com o valor presente na entrada S , de forma a obter um comportamento equivalente a deslocar para o registo de deslocamento o valor em S e ao mesmo tempo colocar com zeros os restantes 3 *bits* $Q_C Q_B Q_A$. Note que o valor a carregar para desfazer a sequência que se pretende detectar deve ser tal que evite o possível reaproveitamento parcial de uma sequência já detectada. Naturalmente que se o circuito tivesse por objectivo detectar a sequência 0000 já não deveriam ser carregados zeros para limpar a sequência detectada. A figura 7.23 mostra as alterações ao circuito anterior para que não detecte sequências parcialmente sobrepostas.

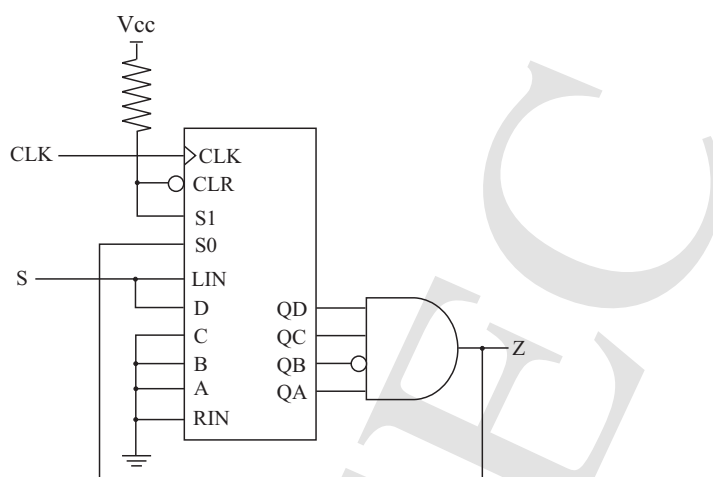


Figura 7.23: Um detector da sequência 1011 que evita a identificação de sequências parcialmente sobrepostas.

Um registo de deslocamento pode também ser utilizado para construir circuitos do tipo contador, que percorrem ciclicamente uma sequência de estados determinada. Isso pode ser feito à custa de circuitos lógicos que operam sobre os valores apresentados nas saídas e actuam nas entradas de dados e de controlo do registo de deslocamento. O exemplo da figura 7.24 produz ciclicamente uma sequência de valores *one-hot*, de forma semelhante à gerada pelo circuito mostrado na figura 7.13 (página 184). Note, no entanto, que o circuito baseado no registo de deslocamento de 4 *bits* tem 4 *flip-flops* e apenas produz 4 valores distintos nas saídas⁴; o circuito baseado no contador binário e no descodificador apenas usa 3 *flip-flops* (note que a saída Q_D não é utilizada) para gerar uma sequência de 8 valores.

⁴Não contando com o estado 0000 que se assume ser o estado inicial e que nunca mais ocorre.

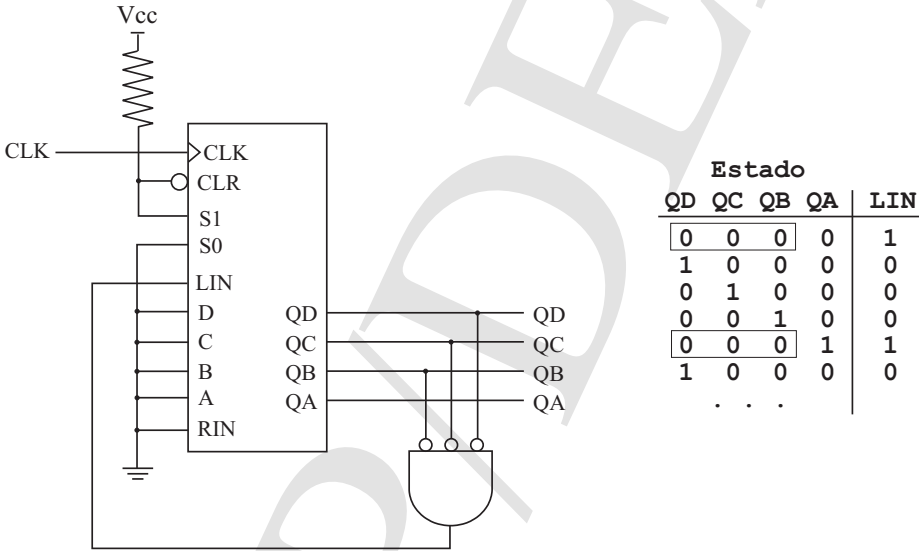


Figura 7.24: Um circuito contador construído em torno de um registo de deslocamento '194.