CIRCUITOS DIGITAIS E
Microprocessadores

HERBERT TAUB

McGraw-Hill

Do original

Digital Circuits and Microprocessors

Copyright © 1982 by McGraw-Hill, Inc.

Copyright © 1984 da Editora McGraw-Hill do Brasil, Ltda.

Todos os direitos para a língua portuguesa reservados pela
Editora McGraw-Hill do Brasil, Ltda.

Nenhuma parte desta publicação poderá ser reproduzida, guardada pelo sistema "retrieval" ou transmitida de qualquer modo ou por qualquer outro meio, seja este eletrônico, mecânico, de fotocópia, de gravação, ou outros, sem prévia autorização, por escrito, da Editora.

Capa: Viviane Malhamé

CIP-Brasil. Catalogação-na-Publicação
Câmara Brasileira do Livro, SP

Taub, Herbert, 1918-
T221c Circuitos digitais e microprocessadores / Herbert Taub ; tradução Fernando Fontes Barbosa ; revisão técnica Rodrigo Araês Caldas Farias. São Paulo : McGraw-Hill do Brasil, 1984.
1. Eletrônica digital 2. Microprocessadores I. Título.

17. e 18. CDD-621.38153
17. -621.381958
83-1621 18. -621.38915835

Índices para catálogo sistemático:
1. Circuitos digitais : Engenharia eletrônica
621.38153 (17. e 18.)
2. Microprocessadores : Engenharia eletrônica
621.381958 (17.) 621.38195835 (18.)

SUMÁRIO

Prefácio XIII

Capítulo 1 Álgebra das Variáveis Lógicas 1

1.1 Variáveis e Funções 1
1.2 Variáveis Lógicas 2
1.3 Valores de Uma Variável Lógica 2
1.4 Funções de Uma Variável Lógica 3
1.5 Funções de Duas Variáveis Lógicas 4
1.6 A Função OR 6
1.7 Uma Implementação de Um Sistema Lógico 7
1.8 Representação de Variáveis Lógicas por Tensões Elétricas 8
1.9 Inversão 9
1.10 A Notação 0 e 1 10
1.11 O Sistema de Numeração Binário 11
1.12 Conversões entre Números Binários e Decimais 12
1.13 Os Sistemas de Numeração Octal e Hexadecimal 13
1.14 Números Binários e Variáveis Lógicas 14
1.15 Teoremas da Álgebra de Boole 15
1.16 O Teorema de De Morgan 18
1.17 Diagramas de Venn 18
1.18 As Funções de Duas Variáveis 22
1.19 A Função EXCLUSIVE-OR (OU – EXCLUSIVO) 23
1.20 As Funções NAND e NOR 24
1.21 As Funções Implicação 25

1.22 Relação entre Operações 26
1.23 Suficiência das Operações 27
1.24 Suficiência de NAND; Suficiência de NOR 27
1.25 Exemplos de Aplicação dos Teoremas da Álgebra de Boole 28
1.26 Exemplos Adicionais 30
1.27 Diagramas Lógicos 33
1.28 Códigos Numéricos 34
1.29 Nomenclatura 36
1.30 Códigos de Dados 36

Capítulo 2 Funções Lógicas 37

2.1 Formas Padrão das Funções Lógicas: A Soma Padrão de Produtos . . 37
2.2 O Produto Padrão de Somas 39
2.3 Numeração dos Mintermos e Maxtermos 41
2.4 Especificação de Funções em Termos de Mintermos e Maxtermos . . 42
2.5 Relação entre Mintermos, Maxtermos e a Tabela Verdade 42
2.6 Estruturas de Dois Níveis com Portas 44
2.7 Estruturas Usando um Tipo de Porta 46
2.8 Mapas de Karnaugh 50
2.9 Simplificação de Funções Lógicas com Mapas de Karnaugh 54
2.10 Adjacências Lógicas Adicionais 56
2.11 Agrupamentos Maiores em Um Mapa K 58
2.12 Mapas de Karnaugh para Cinco e Seis Variáveis 61
2.13 Uso dos Mapas de Karnaugh 64
2.14 Mapeamento quando a Função não É Expressa por Mintermos 70
2.15 Funções Incompletamente Especificadas 72

Capítulo 3 Circuitos Combinacionais Básicos 74

3.1 Introdução 74
3.2 Famílias de Circuitos Lógicos 76
3.3 A Série TTL 77
3.4 A Família CMOS 80
3.5 A Família ECL 81
3.6 Encapsulamento 81
3.7 A Chave Operada por Lógica 82
3.8 A Chave Operada por Lógica: Porta OR; a Conexão Wired-AND . . . 83
3.9 A Saída Totem-Pole (Poste de Totem) 86
3.10 A Saída de Três-Estados 87
3.11 Exemplos de Portas Lógicas CI 88
3.12 Símbolos de Terminais de Controle; Lógica Mista 90
3.13 Símbolos de Sinais de Controle 94
3.14 Lógica Mista Aplicada a Estruturas com Portas 95
3.15 Decodificadores 98
3.16 Codificadores 102
3.17 Conversores de Códigos 107

3.18 Multiplexadores 108
3.19 Multiplexação com Saídas de Coletor Aberto e de Três Estados . . . 113
3.20 Demultiplexação 116

Capítulo 4 Flip-Flops, Registradores e Contadores 118

4.1 Introdução 118
4.2 Um Latch com Portas NOR 120
4.3 Um Latch com Portas Nand 121
4.4 A Chave sem Trepidação 123
4.5 Latches Controlados 125
4.6 Sincronismo (*Clocking*) 127
4.7 Uma Limitação do Latch como Elemento Armazenador 128
4.8 O Flip-Flop Mestre-Escravo 130
4.9 Diagrama de Tempos de Um Flip-Flop 134
4.10 Sincronismo de Duas Fases 135
4.11 O Flip-Flop *JK* 136
4.12 A Propriedade de Detecção de 1s do Flip-Flop Mestre-Escravo 137
4.13 Um Flip-Flop *JK* Gatilhado pela Borda 139
4.14 O Flip-Flop Tipo *D* 141
4.15 Tempos de Preparação (Setup), Manutenção (Hold) e Propagação . . 145
4.16 Transferências de Registrador para Registrador 147
4.17 Registradores de Deslocamento 150
4.18 Características Adicionais e Usos dos Registradores de Deslocamento 152
4.19 Contadores 155
4.20 O Contador em Anel 156
4.21 O Contador em Anel Torcido 158
4.22 Outros Contadores Síncronos 159
4.23 Comparação de Velocidade de Contadores Síncronos 161
4.24 Contadores Síncronos de Módulo Arbitrário 162
4.25 O Contador Síncrono Incrementador-Decrementador (Up-Down) . . 166
4.26 Ciclos Inválidos 167
4.27 Contadores por Pulsação (*Ripple Counters*) 168
4.28 Circuitos Integrados Contadores 171

Capítulo 5 Aritmética 174

5.1 Representação de Números com Sinal 174
5.2 A Representação de Números com Sinal por Complemento de Dois . 176
5.3 A Representação de Números com Sinal por Complemento de Um . . 179
5.4 Soma de Dois Números Binários 180
5.5 Um Somador Série 184
5.6 Soma em Paralelo 185
5.7 Um Calculador Simples de Soma e Subtração 186
5.8 Subtratores 188
5.9 Somadores Rápidos 189

1.22 Relação entre Operações 26
1.23 Suficiência das Operações 27
1.24 Suficiência de NAND; Suficiência de NOR 27
1.25 Exemplos de Aplicação dos Teoremas da Álgebra de Boole 28
1.26 Exemplos Adicionais 30
1.27 Diagramas Lógicos 33
1.28 Códigos Numéricos 34
1.29 Nomenclatura 36
1.30 Códigos de Dados 36

Capítulo 2 Funções Lógicas 37

2.1 Formas Padrão das Funções Lógicas: A Soma Padrão de Produtos . . 37
2.2 O Produto Padrão de Somas 39
2.3 Numeração dos Mintermos e Maxtermos 41
2.4 Especificação de Funções em Termos de Mintermos e Maxtermos . . 42
2.5 Relação entre Mintermos, Maxtermos e a Tabela Verdade 42
2.6 Estruturas de Dois Níveis com Portas 44
2.7 Estruturas Usando um Tipo de Porta 46
2.8 Mapas de Karnaugh 50
2.9 Simplificação de Funções Lógicas com Mapas de Karnaugh 54
2.10 Adjacências Lógicas Adicionais 56
2.11 Agrupamentos Maiores em Um Mapa K 58
2.12 Mapas de Karnaugh para Cinco e Seis Variáveis 61
2.13 Uso dos Mapas de Karnaugh 64
2.14 Mapeamento quando a Função não É Expressa por Mintermos 70
2.15 Funções Incompletamente Especificadas 72

Capítulo 3 Circuitos Combinacionais Básicos 74

3.1 Introdução 74
3.2 Famílias de Circuitos Lógicos 76
3.3 A Série TTL 77
3.4 A Família CMOS 80
3.5 A Família ECL 81
3.6 Encapsulamento 81
3.7 A Chave Operada por Lógica 82
3.8 A Chave Operada por Lógica: Porta OR; a Conexão Wired-AND . . . 83
3.9 A Saída Totem-Pole (Poste de Totem) 86
3.10 A Saída de Três-Estados 87
3.11 Exemplos de Portas Lógicas CI 88
3.12 Símbolos de Terminais de Controle: Lógica Mista 90
3.13 Símbolos de Sinais de Controle 94
3.14 Lógica Mista Aplicada a Estruturas com Portas 95
3.15 Decodificadores 98
3.16 Codificadores 102
3.17 Conversores de Códigos 107

3.18 Multiplexadores 108
3.19 Multiplexação com Saídas de Coletor Aberto e de Três Estados 113
3.20 Demultiplexação 116

Capítulo 4 Flip-Flops, Registradores e Contadores 118

4.1 Introdução 118
4.2 Um Latch com Portas NOR 120
4.3 Um Latch com Portas Nand 121
4.4 A Chave sem Trepidação 123
4.5 Latches Controlados 125
4.6 Sincronismo (*Clocking*) 127
4.7 Uma Limitação do Latch como Elemento Armazenador 128
4.8 O Flip-Flop Mestre-Escravo 130
4.9 Diagrama de Tempos de Um Flip-Flop 134
4.10 Sincronismo de Duas Fases 135
4.11 O Flip-Flop *JK* 136
4.12 A Propriedade de Detecção de 1s do Flip-Flop Mestre-Escravo 137
4.13 Um Flip-Flop *JK* Gatilhado pela Borda 139
4.14 O Flip-Flop Tipo *D* 141
4.15 Tempos de Preparação (Setup), Manutenção (Hold) e Propagação . . 145
4.16 Transferências de Registrador para Registrador 147
4.17 Registradores de Deslocamento 150
4.18 Características Adicionais e Usos dos Registradores de Deslocamento 152
4.19 Contadores 155
4.20 O Contador em Anel 156
4.21 O Contador em Anel Torcido 158
4.22 Outros Contadores Síncronos 159
4.23 Comparação de Velocidade de Contadores Síncronos 161
4.24 Contadores Síncronos de Módulo Arbitrário 162
4.25 O Contador Síncrono Incrementador-Decrementador (Up-Down) . . 166
4.26 Ciclos Inválidos 167
4.27 Contadores por Pulsação (*Ripple Counters*) 168
4.28 Circuitos Integrados Contadores 171

Capítulo 5 Aritmética 174

5.1 Representação de Números com Sinal 174
5.2 A Representação de Números com Sinal por Complemento de Dois . 176
5.3 A Representação de Números com Sinal por Complemento de Um . . 179
5.4 Soma de Dois Números Binários 180
5.5 Um Somador Série 184
5.6 Soma em Paralelo 185
5.7 Um Calculador Simples de Soma e Subtração 186
5.8 Subtratores 188
5.9 Somadores Rápidos 189

5.10 O Somador com Vai-Um Antecipado 190
5.11 Vai-Um Antecipado Aplicado a Grupos 194
5.12 Uso de Vai-Um Antecipado Adicional 196
5.13 A Unidade Lógica-Aritmética (ULA) 199
5.14 Soma BCD 204
5.15 Multiplicação e Divisão 206

Capítulo 6 Memória 207

6.1 A Memória de Acesso Aleatório (RAM) 207
6.2 Estrutura de uma RAM Semicondutora 209
6.3 Ligação de Circuitos de Memória em Paralelo 209
6.4 Organização Interna de Memórias em Uma e Duas Dimensões 214
6.5 A Memória Só de Leitura (ROM) 214
6.6 Implementação de um Codificador de ROM 217
6.7 ROMs Programáveis e Apagáveis 200
6.8 Volatilidade da Memória 221
6.9 Tempos de Chaveamento das Memórias 221
6.10 O Arranjo Lógico Programável 226
6.11 RAM Dinâmica 228
6.12 Detecção de Dados em Uma Memória Dinâmica 229
6.13 Características de Uma Memória Dinâmica 231
6.14 Memórias Série 231
6.15 Dispositivos Acoplados por Carga; Memórias Série 236
6.16 Pilhas de Memória 239
6.17 Memória de Massa 241
6.18 Simbolismo 245

Capítulo 7 Circuitos Seqüenciais 248

7.1 Estados 248
7.2 Contadores como Sistemas Seqüenciais 249
7.3 Um Contador de Módulo 4 Incrementador-Decrementador 255
7.4 Um Detector de Seqüências 259
7.5 Circuitos Moore e Mealy 264
7.6 Eliminação de Estados Redundantes 271
7.7 Eliminação de Estados Redundantes por Partição 273
7.8 Um Exemplo 275
7.9 Atribuições de Estados 280
7.10 Projetos Alternativos 280
7.11 Circuitos Seqüenciais no Modo Fundamental 283
7.12 Um Exemplo de Análise 284
7.13 Um Exemplo de Projeto 286
7.14 Corridas 288
7.15 Eliminação de Corridas Críticas 290
7.16 Um Exemplo 292
7.17 Eliminação de Estados Redundantes 294

7.18 Outro Exemplo de Eliminação de Estados Redundantes 297
7.19 Perigo e Circuitos Assíncronos 301

Capítulo 8 Controladores 304

8.1 Transferências de Registradores 304
8.2 Outras Operações 307
8.3 Registrador Sensível a Comandos Múltiplos 310
8.4 Um Controlador Simples 312
8.5 Implementação do Controlador 318
8.6 O Controlador do Registrador de Deslocamentos 321
8.7 Resposta Condicional de Controladores 323
8.8 Seqüência para Subtração 328
8.9 Um Computador Simples 331
8.10 Operação do Computador 335
8.11 Projeto do Controlador do Computador 338
8.12 Interrupção 341
8.13 Confirmação de Conexão (*Handshaking*) 345

Capítulo 9 Computadores 350

9.1 Uma Arquitetura Aprimorada 350
9.2 Instruções 355
9.3 Resumo das Instruções 363
9.4 Adição e Subtração 363
9.5 Uso de JMP e ISZ 366
9.6 Programa da Multiplicação 368
9.7 Programa que Ilustra a Chamada de Sub-Rotina 370
9.8 Microprogramação 370
9.9 Desvio (Branch) do Microprograma 374
9.10 Desvio Condicional 375
9.11 Bombeamento (*Pipelining*) 376
9.12 Um Controlador Microprogramado 378
9.13 Conteúdo da ROM de Controle 380
9.14 Método de Endereçamento 382
9.15 Pilhas (Stacks) de Memória 383

Capítulo 10 Microprocessadores 386

10.1 Introdução 386
10.2 Arquitetura do Programador de um Microprocessador 388
10.3 Instruções de Um, Dois e de Três Bytes no 8080 391
10.4 Instruções para Movimento de Dados 393
10.5 Instruções que Envolvem diretamente o Acumulador 394
10.6 Instruções Aritméticas 397
10.7 Alguns Exemplos 401
10.8 Instruções de Incrementar-Decrementar 402

10.9 Instruções Lógicas 403
10.10 As Instruções de Comparação 404
10.11 As Instruções de Girar (Rotate) 405
10.12 As Instruções de Complementar e Set 405
10.13 Um Exemplo 406
10.14 As Instruções de Jump (Salto) 408
10.15 As Instruções de Chamada e de Retorno 409
10.16 As Instruções "Push and Pop" 413
10.17 Um Exemplo: Multiplicação 417
10.18 O Microprocessador do Tipo 6800 419
10.19 Modos de Endereçamento do Tipo 6800 420
10.20 O Registrador de Código de Condição do 6800 421

Capítulo 11 **Operações de Entrada-Saída (I/0)** 423

11.1 Geração de Sinais de Controle I/0 no 8080 423
11.2 Entrada-Saída Mapeada na Memória e Isolada 427
11.3 Uso de Instruções IN e OUT 432
11.4 Um Teclado Não-Codificado 434
11.5 Controle de Dispositivos Periféricos 437
11.6 Laços de Temporização 437
11.7 Interrupções 439
11.8 Habilitar e Desabilitar Interrupção 443
·11.9 Interrupção com Chamada Seletiva (*Polled*) 445
10.10 Outra Comunicação I/0 448

Apêndices

A Código ASCII (Hexadecimal com Bit zero mais significativo) . . 452
B 454
C Conjunto de Instruções do 8080 456

Problemas 459

Índice Analítico 504

Para
Esther

CAPÍTULO 1

ÁLGEBRA DAS VARIÁVEIS LÓGICAS

1.1 VARIÁVEIS E FUNÇÕES

O conceito de *variável* e de *função* de uma variável nos é familiar. O *campo* de uma variável, isto é, o intervalo de valores que pode ser assumido por uma variável x, pode ser especificado de inúmeras maneiras. Por exemplo, x pode variar sobre o campo dos reais de menos até mais infinito; ou x pode ser restrita ao intervalo de -17 a -4; ou pode ainda ser restrita a valores inteiros de 1 a 10 e assim por diante.

Uma *função* é uma *regra* através da qual se determina o valor de uma segunda variável (dependente) y do valor da variável (independente) x, sendo esta dependência representada por $y = f(x)$. Suponhamos que seja nossa intenção que y seja determinada por x através da regra que x deva ser multiplicada por si mesma, que este produto seja multiplicado por 5 e que 3 seja somado ao resultado. Esta relação funcional entre y e x seria expressa pela equação $y = 5x^2 + 3$. Neste exemplo determinamos y aplicando os processos matemáticos de multiplicação e adição. Quando o número de valores permitidos a x for finito, é possível especificar uma função construindo uma *tabela* em que y é dada para cada valor de x. Quando o número de valores possíveis de x for pequeno, o uso de tal tabela não só é possível mas também conveniente. Se, no exemplo anterior ($y = 5x^2 + 3$), restringirmos x aos valores inteiros $x = 0$, 1, 2 e 3, então a relação funcional entre y e x pode ser expressa em forma tabular como vemos na Fig. 1.1-1.

Através de uma simples extensão destas idéias elementares, torna-se claro que as variáveis dependente e independente, não precisam ser numéricas. Seja, por exemplo, a variável independente x, cujo campo são as cores de um semáforo, e a variável dependente y, que representa o comportamento esperado de um motorista que se aproxima do semáforo. Neste caso a relação funcional entre x e y é a expressa na Fig. 1.1-2.

x	$y = f(x)$
0	3
1	8
2	23
3	48

Fig. 1.1-1 Uma função numérica.

x	$y = f(x)$
Verde	Prossiga
Amarelo	Devagar
Vermelho	Pare

Fig. 1.1-2 Uma relação funcional.

Os valores que podem ser assumidos por x são expressos pelas afirmações "a luz está verde" ou "a luz está amarela" ou "a luz está vermelha". De modo similar, os valores que podem ser assumidos por y são "o motorista pode prosseguir" etc.

1.2 VARIÁVEIS LÓGICAS

Uma variável *lógica* é uma variável que tem três propriedades distintas:

1. A variável lógica só pode assumir um (ou o outro) de *dois* valores possíveis.
2. Os valores são expressos por afirmações declarativas, como no exemplo do semáforo dado acima.
3. Os dois valores possíveis, expressos por afirmações declarativas, devem ser tais que, com base em raciocínio humano, ou seja, com base na lógica, sejam *mutuamente exclusivos*.

Embora como foi visto, não seja necessário que a variável assuma valores numéricos, nada impede que assim seja. Assim, a variável x poderia ter seus dois (e somente dois) valores alternativos e mutuamente exclusivos expressos pelas afirmações "o valor de x é 7" e "o valor de x é 13". Outras propriedades da variável lógica aparecerão na discussão a seguir, onde voltamos ao exemplo do semáforo.

Suponhamos que o semáforo só possa estar verde ou vermelho, excluindo a possibilidade de que esteja amarelo ou de que haja um intervalo durante a troca de cor em que não estaria nem verde nem vermelho. Neste caso a variável x na tabela da Fig. 1.1-2 seria uma variável lógica: as hipóteses possíveis seriam "o semáforo está verde", que seria representada por x = verde, ou então x = vermelho. Devido à mútua exclusividade, se desejarmos representar x = vermelho, podemos fazê-lo desta maneira ou através de x = *não*-verde. De modo mais simples, o "não" é representado colocando uma barra sobre o valor afetado. Assim, x = não-verde seria escrito $x = \overline{\text{verde}}$ e teríamos $x = \overline{\text{verde}}$ = vermelho.

1.3 VALORES DE UMA VARIÁVEL LÓGICA

No caso mais geral de um tipo arbitrário de variável, como o tipo de variável que assume valores numéricos, as variáveis podem representar qualquer coisa. Assim x e y podem representar temperatura, pressão, distância, velocidade ou tempo. Ao considerar a relação funcional entre variáveis do ponto de vista matemático, não nos interessa o que é representado por elas.

Portanto, da equação $y = 5x^2 + 3$ temos que $y = 8$ quando $x = 1$, independentemente do que seja representado por x e y. Os valores que podem ser assumidos pelas variáveis são os mesmos no sentido de que ambos são números.

Do mesmo modo, daremos aos dois valores possíveis de uma variável lógica dois nomes de modo a podermos considerar a variável independentemente do que ela possa representar. Quaisquer dois nomes facilmente distinguíveis serviriam, mas seria interessante que tais nomes sugerissem a idéia de exclusividade mútua, como, por exemplo, "quente e frio", "dentro e fora", "alto e baixo" etc. Um outro conjunto possível de nomes que comentaremos mais tarde usa os valores "verdadeiro" e "falso". Uma variável lógica A teria ou o valor A = verdadeiro (abreviado A = V) ou o valor A = falso (abreviado A = F). Se tivermos A = verdadeiro (A = V), então podemos escrever igualmente $A = \overline{\text{falso}}$ ($A = \overline{\text{F}}$).

Voltemos ao exemplo do semáforo. (Como estamos tratando com variáveis lógicas, seguiremos o costume de usar A e Z, respectivamente, para variáveis independente e dependente, ao invés de x e y.) A relação funcional entre a cor do semáforo e o comportamento do motorista é mostrada na Fig. 1.3-1*a*. Suponhamos que, no caso da variável A, atribuamos *arbitrariamente* o valor A = V à afirmação "o semáforo está vermelho". Automaticamente A = F representará a afirmação "o semáforo está verde". De modo similar e *arbitrariamente* associemos o valor Z = V à afirmação "o motorista prossegue". A relação funcional entre a cor do semáforo e o comportamento do motorista é, então, dada pela Fig. 1.3-1*b*. Se fizéssemos atribuições diferentes a A e Z em relação à cor do semáforo e ao comportamento do motorista, a tabela da Fig. 1.3-1*b* seria diferente, mas a relação funcional entre a cor e o comportamento seria a mesma.

Uma tabela como a da Fig. 1.3-1*b* com valores V e F chama-se *tabela verdade*.

A	$Z = f(A)$
Verde	Prossiga
Vermelho	Pare

(*a*)

A	$Z = f(A)$
F	V
V	F

(*b*)

Fig. 1.3-1 A relação funcional em (*a*) torna-se a tabela verdade em (*b*).

1.4 FUNÇÕES DE UMA VARIÁVEL LÓGICA

Todas as funções possíveis $Z = f(A)$ de uma variável lógica são mostradas nas quatro tabelas verdade da Fig. 1.4-1. Para termos certeza de que nenhuma das funções possíveis foi esquecida, procederemos da seguinte maneira: na coluna A listamos os dois valores possíveis V e F: temos *dois* lugares na coluna Z onde devemos colocar valores. Em cada um destes lugares há *dois* valores possíveis que podem ser colocados; conseqüentemente, o número de colunas Z diferentes que é possível é $2 \times 2 = 4$. Estas quatro colunas Z são mostradas e podemos ter certeza de que não existem outras. Como na Fig. 1.4-1*a* o valor de Z em cada linha é igual ao valor de A, podemos escrever $Z = A$. Na Fig. 1.4-1*b*, $Z = \overline{A}$. Na Fig. 1.4-1*c*, Z = F e na Fig. 1.4-1*d*, Z = V. O leitor decidirá se nas Figs. 1.4-1*c* e *d* não existe função, pois em um caso Z é falsa e no outro caso Z é verdadeira, independentemente do valor lógico de A.

A	$Z = f(A)$
F	F
V	V

(a)

A	Z
F	V
V	F

(b)

A	Z
F	F
V	F

(c)

A	Z
F	V
V	V

(d)

Fig. 1.4-1 As quatro funções de uma variável.

1.5 FUNÇÕES DE DUAS VARIÁVEIS LÓGICAS

Consideremos agora as funções $Z = f(A, B)$ de duas variáveis lógicas A e B. Para formar tais funções, comecemos com uma tabela verdade como a da Fig. 1.5-1, onde existe uma linha para cada combinação possível dos valores lógicos de A e B. Existindo duas variáveis, cada uma com dois valores lógicos possíveis, quatro são as combinações possíveis. Para gerar a função, basta colocar os valores correspondentes na coluna Z; há quatro valores a serem colocados e duas possibilidades para cada valor, conseqüentemente o número total de colunas Z diferentes possível é $2 \times 2 \times 2 \times 2 = 16$ e, em correspondência, há dezesseis funções possíveis de duas variáveis. Como no caso das funções de uma variável, poderíamos pensar que algumas destas "funções" realmente não o são. Iremos eventualmente considerar todas as funções possíveis. No momento examinaremos algumas funções por serem de especial interesse.

A Função AND

Já vimos que uma função é definida por uma tabela verdade. A função $Z = f(A, B)$, que é definida pela tabela verdade da Fig. 1.5-2, é chamada *função* AND. Expressamos a dependência de Z em relação a A e B escrevendo

$$Z = A \text{ AND } B \tag{1.5-1}$$

A razão para a terminologia adotada reside em que Z = V somente quando A e B são *ambos* verdadeiros, o que pode ser verificado na tabela verdade. Um outro simbolismo para a função AND é

$$Z = A \cdot B \tag{1.5-2}$$

ou, ainda, mais simplesmente:

$$Z = AB \tag{1.5-3}$$

A	B	$Z = f(A, B)$
F	F	
F	V	
V	F	
V	V	

Fig. 1.5-1 Uma tabela verdade incompleta para duas variáveis.

As Eqs. (1.5-2) e (1.5-3) sugerem que Z é o resultado de uma "multiplicação" na qual A e B são os fatores. É claro que A e B não são números e não se imagina multiplicação no sentido aritmético usual; mas, como veremos, a sugestão de multiplicação induzida pelo simbolismo é deliberada e a função A AND B é freqüentemente chamada de *produto lógico* de A e B.

Uma primeira propriedade da função AND é a *comutatividade*, isto é, se A e B tiverem sua ordem trocada (A e B forem comutados), a função Z não se altera, de modo que

$$Z = AB = BA \tag{1.5-4}$$

Na tabela verdade da Fig. 1.5-1 é aparente que isto é verdade. Se tivéssemos construído a tabela de modo que a coluna B fosse a primeira em vez da coluna A, os valores na coluna Z não se alterariam.

Uma segunda propriedade da função AND é a *associatividade*. Suponhamos que existam três variáveis lógicas A, B e C e que formemos, em primeiro lugar, o produto lógico AB. Como este produto é uma variável lógica, podemos formar seu produto lógico com C obtendo $(AB)C$. Se formássemos primeiro o produto BC, poderíamos, a seguir, obter $A(BC)$. Como verificaremos a seguir, o modo como as variáveis estão associadas não faz diferença. Assim:

$$Z = (AB)C = A(BC) \tag{1.5-5}$$

O fato da função AND ser associativa é quase óbvio, mas, para termos a oportunidade de observar a maneira de provar qualquer teorema sobre variáveis lógicas, deduziremos a prova formal. O processo consiste em gerar uma tabela verdade e observar o fato que quaisquer funções que resultem em colunas idênticas, isto é, com valores V e F idênticos em cada linha, são idênticas.

Exemplo 1.5-1 Verificar que a função AND é associativa, isto é, verificar a validade da Eq. (1.5-5).

SOLUÇÃO: Estamos agora tratando com três variáveis A, B e C; como cada variável permite dois valores, existem $2 \times 2 \times 2 = 8$ combinações possíveis de valores das três variáveis. Conseqüentemente, a tabela verdade para três variáveis deve ter oito linhas, como se observa na Fig. 1.5-3. Nas colunas A, B e C listamos todas as combinações diferentes possíveis dos valores das três variáveis. (Mais tarde mostraremos como se pode facilmente fazer tal listagem com a segurança de que nenhuma combinação foi duplicada ou está faltando.)

Na quarta coluna geramos o produto lógico AB, linha por linha, usando a definição da tabela verdade da Fig. 1.5-2. Na quinta coluna geramos $(AB)C$ usando os valores das colunas 3 e 4. Do mesmo modo calculamos $A(BC)$ e colocamos o resultado na coluna 7. Finalmente,

A	B	$Z = A$ AND B
F	F	F
F	V	F
V	F	F
V	V	V

Fig. 1.5-2 Tabela verdade que define a função AND.

Idêntica

1	2	3	4	5	6	7
A	B	C	AB	$(AB)C$	BC	$A(BC)$
F	F	F	F	F	F	F
F	F	V	F	F	F	F
F	V	F	F	F	F	F
F	V	V	F	F	V	F
V	F	F	F	F	F	F
V	F	V	F	F	F	F
V	V	F	V	F	F	F
V	V	V	V	V	V	V

Fig. 1.5-3 Verificação da associatividade da função AND.

notamos que os valores da coluna 5, $(AB)C$, são, *em cada caso*, idênticos aos valores da coluna 7, $A(BC)$; a tabela verdade prova, portanto, que $A(BC) = (AB)C$.

1.6 A FUNÇÃO OR

A função Z, que é definida pela tabela verdade da Fig. 1.6-1, é chamada função OR. A dependência de Z em relação aos valores de A e B é expressa por

$$Z = A \text{ OR } B \tag{1.6-1}$$

A razão para a terminologia adotada reside em que $Z = \text{V}$ se $A = \text{V}$ *ou* $B = \text{V}$, *ou*, ainda, *se ambos* A e $B = \text{V}$, como se verifica na tabela verdade. Um outro simbolismo para a função OR é

$$Z = A + B \tag{1.6-2}$$

O sinal mais na Eq. (1.6-2) não indica soma no sentido aritmético usual, embora a sugestão seja deliberada e a função $A + B$ seja, freqüentemente, chamada de *soma lógica* de A e B.

Como no caso da função AND, verifica-se facilmente que a função OR é associativa e é comutativa. Assim:

$$A + B = B + A \tag{1.6-3}$$

e

$$A + (B + C) = (A + B) + C \tag{1.6-4}$$

A	B	$Z = A$ OR B
F	F	F
F	V	V
V	F	V
V	V	V

Fig. 1.6-1 Tabela verdade que define a função OR.

1.7 UMA IMPLEMENTAÇÃO DE UM SISTEMA LÓGICO

O conceito de variável lógica foi introduzido por volta de 1850 pelo matemático George Boole em conexão com seus estudos sobre o processo do pensamento. A álgebra das variáveis lógicas que consideraremos a seguir é chamada *álgebra booleana*. A adaptação da álgebra booleana para sistemas digitais, nosso interesse presente, foi introduzida em 1938 por Claude Shannon do Bell Laboratories.

Para dar significado mais concreto ao conceito de função lógica e verificar a relevância do termo "lógico", consideremos o seguinte exemplo relacionado com a função AND. Se fôssemos passageiros na cabina de um avião pilotado por dois pilotos e víssemos um piloto deixar seu assento e víssemos também o segundo piloto deixar seu assento antes que o primeiro retornasse, logicamente *deduziríamos* que o avião estaria sem piloto. (Dependendo das circunstâncias, talvez deduzíssemos que o avião estaria em perigo e que medidas de emergência deveriam ser tomadas.)

Podemos construir uma máquina simples que executa a operação lógica de deduzir que o avião está sem piloto. Esta máquina é mostrada na Fig. 1.7-1 e consiste nas chaves 1 e 2 colocadas nos assentos dos pilotos P_A e P_B, respectivamente, de tal modo que as chaves fechem quando o piloto deixar sua poltrona. Façamos A representar a variável lógica que é *verdadeira* quando o piloto P_A não estiver em sua poltrona e é *falsa* quando P_A estiver sentado. Façamos B ser a variável correspondente a P_B e Z a variável que é *verdadeira* quando o avião estiver sem

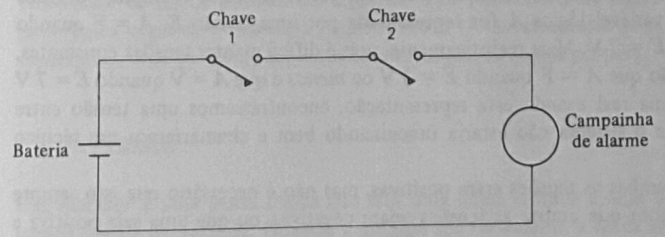


Fig. 1.7-1 Um arranjo de chaves que toca a campainha quando a chave 1 e a chave 2 estiverem fechadas.

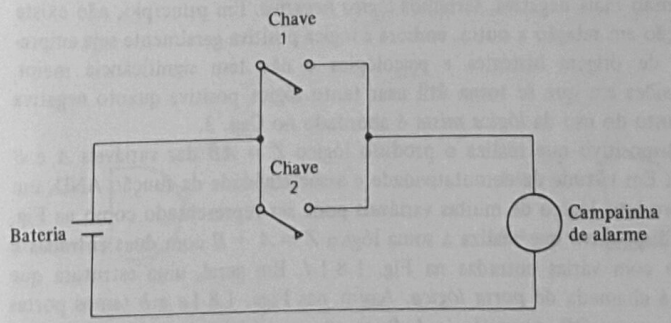


Fig. 1.7-2 Um arranjo de chaves que toca a campainha quando a chave 1 C a chave 2 C ambas estiverem fechadas.

piloto. Quando Z for *verdadeira*, a máquina da Fig. 1.7-1 mostrará este fato tocando a campainha. As variáveis lógicas Z, A e B estão ligadas pela relação funcional $Z = AB$, isto é, quando for verdade que o piloto P_A deixou a cabina e, simultaneamente, for verdade que o piloto P_B deixou a cabina, então será verdade que o avião está sem piloto (e, possivelmente, em perigo). Em qualquer outra situação, exceto quando $A = B = V$, teremos $Z = \text{F}$.

Se julgássemos que o avião requer os serviços simultâneos dos dois pilotos por razões de segurança, então faríamos com que $Z = \text{V}$ se A é *verdade* ou B é *verdade*, ou seja, implementaríamos a lógica $Z = A + B$. Uma máquina que executa esta operação usando chaves é mostrada na Fig. 1.7-2.

1.8 REPRESENTAÇÃO DE VARIÁVEIS LÓGICAS POR TENSÕES ELÉTRICAS

Na seção anterior vimos como o valor de uma variável lógica pode ser indicado pela posição de uma chave. Pareceu-nos conveniente, então, fazer com que uma chave fechada representasse o valor "verdadeiro" e que uma chave aberta representasse o valor "falso". Em outras circunstâncias, talvez fosse mais conveniente inverter a representação de valores fornecidos pela chave, isto é, ter as posições fechada e aberta representando os valores "falso" e "verdadeiro", respectivamente.

Presentemente, em sistemas digitais rápidos, os valores lógicos de uma variável são geralmente representados por tensões elétricas mantidas entre um par de fios. Por exemplo, podemos fazer com que, quando a variável lógica A for representada por uma tensão E, $A = \text{F}$ quando $E = 3\,\text{V}$ e $A = \text{V}$ quando $E = 7\,\text{V}$. Mais realisticamente, pois é difícil manter tensões constantes, adotaríamos a representação que $A = \text{F}$ quando $E = 3\,\text{V}$ ou *menos* e que $A = \text{V}$ quando $E = 7\,\text{V}$ ou *mais*. Se, em um sistema real usando esta representação, encontrássemos uma tensão entre 3 e 7 V, deduziríamos que o sistema não estaria funcionando bem e chamaríamos um técnico para consertá-lo.

No exemplo citado, ambas as tensões eram positivas, mas não é necessário que isto sempre aconteça. Podemos fazer com que ambas as tensões sejam negativas, ou que uma seja positiva e a outra negativa, ou, ainda, que uma ou a outra seja 0 V. Em nosso exemplo escolhemos a tensão mais positiva para representar V e a mais negativa para representar F. Tal representação recebe a designação de *lógica positiva*. Se invertêssemos a representação de modo que o valor *verdadeiro* fosse representado pela tensão mais negativa, teríamos *lógica negativa*. Em princípio, não existe mérito em uma representação em relação a outra, embora a lógica positiva geralmente seja empregada. Esta preferência é de origem histórica e psicológica e não tem significância maior.

Veremos que há ocasiões em que se torna útil usar tanto lógica positiva quanto negativa na mesma discussão; o assunto do uso da *lógica mista* é abordado no Cap. 3.

O símbolo de um dispositivo que realiza o produto lógico $Z = AB$ das variáveis A e B é mostrado na Fig. 1.8-1*a*. Em virtude da comutatividade e associatividade da função AND, um dispositivo que realiza o produto lógico de muitas variáveis pode ser representado como na Fig. 1.8-1*b*. O símbolo de um dispositivo que realiza a soma lógica $Z = A + B$ com duas entradas é mostrado na Fig. 1.8-1*c* e com várias entradas na Fig. 1.8-1*d*. Em geral, uma estrutura que gera uma função lógica Z é chamada de *porta lógica*. Assim, nas Figs. 1.8-1*a* e *b* temos portas AND e nas Figs. 1.8-1*c* e *d*, portas OR. As variáveis A, B etc. são as entradas das portas e a variável Z é a saída da porta. A natureza dos dispositivos eletrônicos que implementam portas AND e

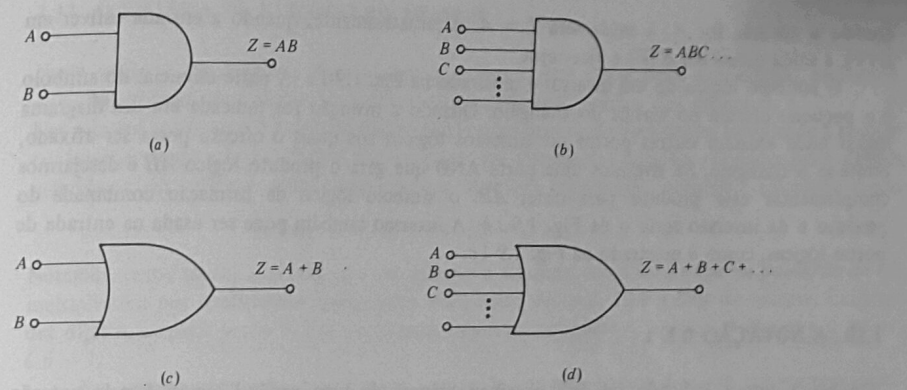


Fig. 1.8-1 (*a*) Porta AND de duas entradas. (*b*) Porta AND com entradas múltiplas. (*c*) Porta OR de duas entradas. (*d*) Porta OR com entradas múltiplas.

OR é tal que entradas adicionais podem ser obtidas facilmente, sendo, pois, comuns portas AND e OR com entradas múltiplas.

A função lógica que um determinado dispositivo físico implementa depende da adoção da convenção de lógica positiva ou negativa. Uma porta OR para lógica positiva é uma porta AND para lógica negativa e vice-versa (veja Prob. 1.8-1). Os fabricantes de portas lógicas uniformemente especificam seus produtos baseados na convenção de lógica positiva.

1.9 INVERSÃO

Um *inversor* é uma porta lógica que tem uma única entrada e uma única saída que é o *complemento* lógico da entrada. Quando a entrada for *verdadeira*, a saída será *falsa* e vice-versa; isto é,

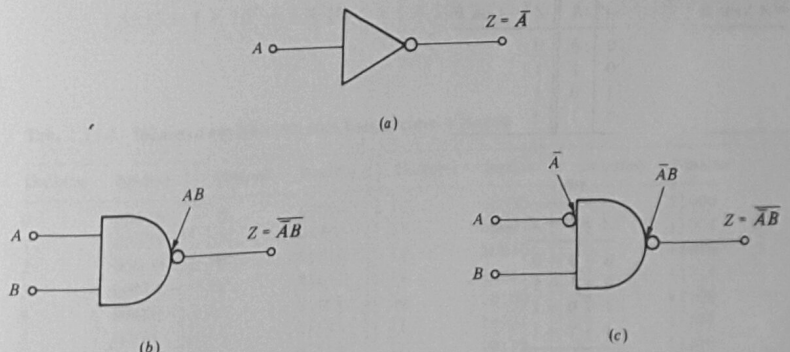


Fig. 1.9-1 (*a*) O símbolo de inversão lógica. (*b*), (*c*) Quando usado em conjunto com outras portas, o símbolo de inversão reduz-se à adição de um círculo aberto.

quando a entrada for A, a saída será $Z = \bar{A}$. Alternativamente, quando a entrada estiver em E (V), a saída estará em E (F) e vice-versa.

O símbolo lógico de um inversor é mostrado na Fig. 1.9-1*a*. A parte essencial do símbolo é o pequeno círculo no vértice do triângulo. Quando a inversão for indicada em um diagrama lógico onde existam outras portas ou símbolos lógicos aos quais o círculo possa ser afixado, omite-se o triângulo. Se tivermos uma porta AND que gera o produto lógico AB e desejarmos complementar este produto para obter $\bar{AB}$, o símbolo lógico da formação combinada do produto e da inversão seria o da Fig. 1.9-1*b*. A inversão também pode ser usada na entrada de portas lógicas, como é mostrado na Fig. 1.9-1*c*.

1.10 A NOTAÇÃO 0 E 1

Até agora temos indicado os dois possíveis valores de uma variável lógica A pela notação A = V (verdadeiro) ou A = F (falso). Introduziremos agora uma notação alternativa que, como veremos, tem diversas vantagens. Usaremos a notação $A = 0$ como alternativa para A = F e $A = 1$ como alternativa para A = V. Devemos enfatizar que o 0 e o 1 usados aqui não são números, mas valores lógicos de uma variável. Em alguns textos os valores lógicos 1 e 0 são impressos em negrito para distingui-los de números. Não o faremos porque sempre ficará claro, do contexto, se os símbolos representam valores numéricos ou valores lógicos.

Vamos reescrever as tabelas verdade das funções AND e OR (Figs. 1.5-2 e 1.6-1) usando a notação 0, 1. Notamos não somente o mérito da notação mas também uma motivação para o uso de "multiplicação" para representar a operação AND e de "soma" para representar a operação OR. As tabelas verdade aparecem nas Figs. 1.10-1*a* e *b*. As linhas das tabelas são escritas na forma de equações nas Figs. 1.10-1*c* e *d*. As notações $\cdot$, +, 0 e 1 constituem um dispositivo mnemônico eficiente. Se imaginássemos que 0 e 1 fossem números e que a soma e a multiplicação fossem aritméticas, as equações estariam em conformidade com as definições, com exceção de $1 + 1 = 1$. Esta correspondência incompleta é vantajosa, pois serve para lembrar-nos de que estamos tratando com valores lógicos e não aritméticos.

A	B	$Z = A$ AND B
0	0	0
0	1	0
1	0	0
1	1	1

(*a*)

A	B	$Z = A$ OR B
0	0	0
0	1	1
1	0	1
1	1	1

(*b*)

AND

$A \cdot B = Z$

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

(*c*)

OR

$A + B = Z$

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 1$

(*d*)

Fig. 1.10-1 Tabelas verdade de AND e OR usando a notação 0, 1.

1.11 O SISTEMA DE NUMERAÇÃO BINÁRIO

Nesta seção e na seguinte abandonaremos a lógica para discutirmos um assunto aritmético.

No sistema de numeração usual, o sistema *decimal*, usamos dez dígitos 0, 1, 2, . . . , 9. Um número maior que 9 é representado usando uma convenção que atribui significado à *posição* ou *lugar* ocupado por um dígito. Por exemplo, em virtude das posições ocupadas pelos dígitos individuais no número 6903, este número tem um significado numérico calculado como

$$6903 = 6 \times 10^3 + 9 \times 10^2 + 0 \times 10^1 + 3 \times 10^0 \qquad (1.11\text{-}1)$$

Notamos, como na Eq. (1.11-1), que um número é expresso como uma soma de *potências de 10* multiplicadas por coeficientes apropriados. No sistema decimal, 10 é a *base* do sistema. Existem dez dígitos, o maior sendo 9. Em um sistema numérico com base n, existem n dígitos e o maior é $n - 1$.

Em relação a sistemas descritos em termos de variáveis lógicas, é muito conveniente usar o sistema numérico de base 2. Esse sistema é chamado sistema *binário* e usa somente os dígitos numéricos 0 e 1. A vantagem do uso do sistema binário é que podemos conseguir uma correspondência biunívoca entre os dois dígitos (números) 0 e 1 e os dois valores lógicos (não numéricos) 0 e 1. Há ocasiões em que a associação se torna tão íntima e conveniente que é vantajoso se esquecer a distinção.

Quando um número é escrito no sistema binário, os dígitos individuais representam os coeficientes de potências de 2 e de potências de 10 como no sistema decimal. Por exemplo, o número *decimal* 19 é escrito em representação *binária* como 10011, pois este arranjo de dígitos binários significa

$$10011 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \qquad (1.11\text{-}2a)$$

$$= 16 + 0 + 0 + 2 + 1 = 19 \qquad (1.11\text{-}2b)$$

Uma pequena lista de equivalências entre números decimais e binários é dada na Tab. 1.11-1. Como usamos cinco dígitos binários, o maior número que podemos representar é 11111 = 31.

Um número que não é inteiro pode ser expresso usando a notação de *vírgula decimal*. O número 1,8125, por exemplo, tem o significado numérico

$$1{,}8125 = 1 \times 10^0 + 8 \times 10^{-1} + 1 \times 10^{-2} + 2 \times 10^{-3} + 2 \times 10^{-3} + 5 \times 10^{-4} \qquad (1.11\text{-}3)$$

Tab. 1.11-1 Números equivalentes com base decimal e binária

Decimal	Binário	Decimal	Binário	Decimal	Binário	Decimal	Binário
0	00000	8	01000	16	10000	24	11000
1	00001	9	01001	17	10001	25	11001
2	00010	10	01010	18	10010	26	11010
3	00011	11	01011	19	10011	27	11011
4	00100	12	01100	20	10100	28	11100
5	00101	13	01101	21	10101	29	11101
6	00110	14	01110	22	10110	30	11110
7	00111	15	01111	23	10111	31	11111

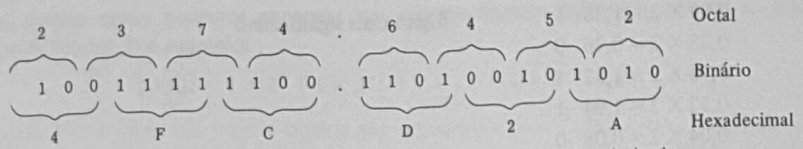


Fig. 1.13-1 Conversões entre os sistemas de numeração octal, binário e hexadecimal.

(ou hexadecimal) para binário, simplesmente substituímos cada dígito octal (ou hexadecimal) pelo binário equivalente de três bits (ou de quatro bits).

1.14 NÚMEROS BINÁRIOS E VARIÁVEIS LÓGICAS

Como se pode imaginar, a utilidade especial do sistema binário de numeração em relação à álgebra das variáveis lógicas deriva-se do fato de as variáveis lógicas somente terem dois valores e o sistema binário só ter dois dígitos. Esta utilidade é reforçada pelo uso da notação 1, 0, introduzida na Seção 1.10, para representar os valores verdade de uma variável lógica.

Suponhamos que se deseje definir uma função Z de três variáveis $Z = f(A, B, C)$ através de sua tabela verdade. Iniciaríamos construindo a *estrutura* da tabela, ou seja, desenharíamos uma estrutura de tabela verdade com quatro colunas. Três destas colunas, possivelmente as mais à esquerda, corresponderiam a A, B e C, enquanto a quarta coluna corresponderia a Z. Nas colunas A, B e C listaríamos, como na Fig. 1.5-3, todas as possíveis combinações de valores verdade das três variáveis. Deveríamos certificarmo-nos de que nenhuma combinação faltasse e também, para evitar redundância, de que nenhuma combinação fosse repetida. Em princípio, a ordem em que as combinações são listadas não interessa. Formada a estrutura da tabela, ela seria usada para definir Z colocando V's e F's nas posições apropriadas na coluna de Z.

Uma tabela verdade de três variáveis apropriada para definir a função Z aparece na Fig. 1.14-1*a* (o leitor pode verificar que a função representada é $Z = A + BC$, embora a função seja irrelevante a esta discussão). Na Fig. 1.14-1*b* repetimos a tabela usando a notação 0, 1 e numerando decimalmente as $(2^3 = 8)$ linhas de 0 a 7. Devemos reconhecer que os valores na

A	B	C	Z
F	F	F	F
F	F	V	F
F	V	F	F
F	V	V	V
V	F	F	V
V	F	V	V
V	V	F	V
V	V	V	V

(*a*)

Linha nº	A	B	C	Z
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

(*b*)

tabela não são números, mas, se ignorarmos este fato e *supusermos* que os valores são números, observaremos que, em cada linha, os valores nas colunas das variáveis independentes A, B e C representam, em aritmética binária, o número daquela linha. Isto foi conseqüência da maneira como a ordem das linhas foi planejada, já que se alterando a posição das linhas não se altera a tabela.

Generalizando, podemos estabelecer o seguinte processo para formar a estrutura de uma tabela verdade de n variáveis: construímos uma tabela contendo $(n + 1)$ colunas, sendo as n primeiras para as variáveis independentes e a última para a função Z. Para assegurar que todas as combinações de valores das variáveis independentes sejam incluídas sem repetição, escrevemos, em ordem e em aritmética binária, os números de 0 a $2^n - 1$. Na tabela assim construída, os 0s e 1s serão interpretados como valores lógicos e não como números.

Independentemente da ordem em que listamos as combinações de possíveis valores das variáveis independentes, temos agora um método para atribuir um *número* a cada uma das combinações ou linhas. Simplesmente supomos que os valores verdade são dígitos binários e lemos o número resultante. Esta característica permite uma maneira adicional de definir uma função. Podemos dizer que (1) $Z = A + B$ ou que (2) a função é definida pela tabela verdade da Fig. 1.14-1*a* ou *b* ou, ainda, que (3) a função é definida como tendo valores 1 nas linhas 4, 5, 6 e 7 da coluna Z de uma tabela verdade de três variáveis (alternativamente podemos dizer que a função tem 0s nas linhas 0, 1, 2 e 3). Note-se que a terceira alternativa permite definir a função sem mostrar explicitamente a tabela verdade.

1.15 TEOREMAS DA ÁLGEBRA DE BOOLE

Desenvolveremos a seguir diversos teoremas envolvendo as operações AND, OR e NOT, que são úteis na simplificação de expressões contendo variáveis lógicas. Usaremos os símbolos de soma e multiplicação para representar as operações OR e AND, e a notação 0, 1 para representar os valores falso e verdadeiro de uma variável lógica, respectivamente.

Existe um princípio especial, descrito pelo termo *dualidade*, que se aplica a teoremas envolvendo as operações AND e OR (ver Prob. 1.15-1). Este princípio já foi sugerido pelas tabelas verdade da Fig. 1.10-1. Examinando qualquer linha de uma destas tabelas, observamos que, se (1) trocarmos os sinais + e · entre si e (2) trocarmos os 0s e 1s entre si, teremos substituído a equação original por outra igualmente válida. Consideremos, por exemplo, a equação $0 \cdot 0 = 0$, que aparece na primeira linha da tabela verdade da função AND. Fazendo as duas trocas indicadas, a equação torna-se $1 + 1 = 1$, que é uma equação válida e aparece na quarta linha da tabela verdade da função OR. Do mesmo modo, verifica-se que fazendo as trocas indicadas em todas as equações da tabela verdade da função AND chega-se, em correspondência biunívoca, – às equações da tabela verdade de OR e vice-versa.

Baseados neste princípio de dualidade, se tivermos um *teorema* relacionando variáveis lógicas, podemos imediatamente escrever um outro teorema trocando os sinais + e · e trocando os 0s e 1s. Os dois teoremas assim relacionados são chamados *teoremas duais* e as expressões derivadas através das duas trocas são chamadas duais uma da outra.

Um lembrete de cautela. Vamos supor que exista uma função f de variáveis lógicas e uma segunda função g das mesmas variáveis e que, embora f e g pareçam diferentes, consigamos estabelecer que $f = g$ para todos os valores possíveis das variáveis. Teremos, assim, estabelecido o

teorema $f = g$. De acordo com o princípio da dualidade, se formarmos as funções duais f_D e g_D, teremos um teorema igualmente válido $f_D = g_D$. O princípio da dualidade *não* diz, todavia, que $f_D = f$ ou que $g_D = g$.

Voltando agora à listagem dos teoremas, temos inicialmente um teorema muito importante, embora evidente por si só. O fato de uma variável só poder assumir um de dois valores nos conduz ao complemento do complemento de uma variável A ser a própria variável A, isto é:

$$\bar{\bar{A}} = A \tag{1.15-1}$$

A Eq. 1.15-1 nos leva às equações duais $\bar{0} = 1$ e $\bar{1} = 0$. Continuando, temos:

$$A + 0 = A \qquad (1.15\text{-}2a) \qquad\qquad A \cdot 1 = A \qquad (1.15\text{-}2b)$$

$$A + 1 = 1 \qquad (1.15\text{-}3a) \qquad\qquad A \cdot 0 = 0 \qquad (1.15\text{-}3b)$$

$$A + A = A \qquad (1.15\text{-}4a) \qquad\qquad A \cdot A = A \qquad (1.15\text{-}4b)$$

$$A + \bar{A} = 1 \qquad (1.15\text{-}5a) \qquad\qquad A \cdot \bar{A} = 0 \qquad (1.15\text{-}5b)$$

Estes oito teoremas envolvem uma unica variável e foram listados na forma de pares duais; aplicando-se o princípio da dualidade à Eq. (1.15-2a), produz-se a Eq. (1.15-2b) etc. A prova da validade de qualquer um dos teoremas é facilmente obtida considerando o teorema para todos os valores possíveis da variável e mostrando que ele é verdadeiro em cada caso. Como a variável só tem dois valores possíveis, este método é factível. Consideremos, por exemplo, a Eq. (1.15-5a): se $A = 0$, a equação diz que $0 + \bar{0} = 0 + 1 = 1$, o que é correto. Se $A = 1$, a equação diz que $1 + \bar{1} = 1 + 0 = 1$, que também é correto.

Listaremos agora alguns teoremas envolvendo duas e três variáveis, novamente sob a forma de pares duais.

$$A + AB = A \tag{1.15-6a}$$

$$A(A + B) = A \tag{1.15-6b}$$

$$AB + A\bar{B} = A \tag{1.15-7a}$$

$$(A + B)(A + \bar{B}) = A \tag{1.15-7b}$$

$$A + \bar{A}B = A + B \tag{1.15-8a}$$

$$A(\bar{A} + B) = AB \tag{1.15-8b}$$

$$A + BC = (A + B)(A + C) \tag{1.15-9a}$$

$$A(B + C) = AB + AC \tag{1.15-9b}$$

$$AB + \bar{A}C = (A + C)(\bar{A} + B) \tag{1.15-10a}$$

$$(A + B)(\bar{A} + C) = AC + \bar{A}B \tag{1.15-10b}$$

$$AB + \bar{A}C + BC = AB + \bar{A}C \tag{1.15-11a}$$

$$(A + B)(\bar{A} + C)(B + C) = (A + B)(\bar{A} + C) \tag{1.15-11b}$$

A Eq. (1.15-9b) é de interesse especial, pois indica que a álgebra de variáveis lógicas é *distributiva*, isto é, dada uma expressão envolvendo parênteses, como $A(B + C)$, podemos remover

os parênteses através da "multiplicação" como se se tratasse de álgebra comum e obter $A(B + C) = AB + AC$. Por outro lado, o teorema indica que um "fator" comum pode ser "fatorado". Assim, em $AB + AC$ o A é um fator comum aos dois termos e pode ser fatorado para que se obtenha $A(B + C)$. Este teorema é aceito intuitivamente graças à sua correspondência com a álgebra comum; seu dual, Eq. (1.15-9a), por outro lado, parece um pouco estranho e é necessário acostumar-se a ele.

Qualquer um dos teoremas listados anteriormente pode ser verificado substituindo todos os valores possíveis das variáveis na equação. Quando houver duas variáveis, teremos quatro combinações; quando três variáveis estiverem envolvidas, teremos oito combinações.

Exemplo 1.15-1 Provar que a Eq. (1.15-8a) é correta.

SOLUÇÃO: A verificação da Eq. (1.15-8a) pode ser feita através da comparação das tabelas verdade de $A + \bar{A}B$ e de $A + B$:

A	B	$Z_1 = A + \bar{A}B$	$Z_2 = A + B$
0	0	$0 + 1 \cdot 0 = 0$	$0 + 0 = 0$
0	1	$0 + 1 \cdot 1 = 1$	$0 + 1 = 1$
1	0	$1 + 0 \cdot 0 = 1$	$1 + 0 = 1$
1	1	$1 + 0 \cdot 1 = 1$	$1 + 1 = 1$

Os valores na coluna de Z_1 são, em todos os casos, idênticos aos valores na coluna de Z_2; portanto, $Z_1 = Z_2$ e, conseqüentemente, a Eq. (1.15-8a) fica comprovada.

Exemplo 1.15-2 Provar que a Eq. (1.15-11b) é correta.

SOLUÇÃO: Para verificar a Eq. (1.15-11b), comparamos as tabelas verdade de $(A + B)(\bar{A} + C)(B + C)$ e de $(A + B)(\bar{A} + C)$. São necessárias tabelas verdade de três variáveis, que são construídas usando o processo descrito na Seção 1.14.

A	B	C	$Z_1 = (A + B)(\bar{A} + C)(B + C)$	$Z_2 = (A + B)(\bar{A} + C)$
0	0	0	$(0 + 0)(1 + 0)(0 + 0) = 0$	$(0 + 0)(1 + 0) = 0$
0	0	1	$(0 + 0)(1 + 1)(0 + 1) = 0$	$(0 + 0)(1 + 1) = 0$
0	1	0	$(0 + 1)(1 + 0)(1 + 0) = 1$	$(0 + 1)(1 + 0) = 1$
0	1	1	$(0 + 1)(1 + 1)(1 + 1) = 1$	$(0 + 1)(1 + 1) = 1$
1	0	0	$(1 + 0)(0 + 0)(0 + 0) = 0$	$(1 + 0)(0 + 0) = 0$
1	0	1	$(1 + 0)(0 + 1)(0 + 1) = 1$	$(1 + 0)(0 + 1) = 1$
1	1	0	$(1 + 1)(0 + 0)(1 + 0) = 0$	$(1 + 1)(0 + 0) = 0$
1	1	1	$(1 + 1)(0 + 1)(1 + 1) = 1$	$(1 + 1)(0 + 1) = 1$

Como $Z_1 = Z_2$ para todas as combinações possíveis de A, B e C, o teorema fica verificado.

Se escrevermos expressões da álgebra booleana em termos de operações AND, OR e NOT, os teoremas básicos são aqueles que envolvem uma variável, Eqs. (1.15-2) a (1.15-5), e os dois teoremas sobre distributividade, Eqs. (1.15-9a e b). Todos os outros teoremas, inclusive os

mostrados nesta seção, podem ser provados usando os teoremas básicos e manipulações algébricas. Por exemplo, os teoremas das Eqs. (1.15-6*a* e *b*) são provados assim:

Eq. (1.15-6*a*)	Eq. (1.15-6*b*)
$A + AB = A \cdot 1 + AB$	$A(A + B) = (A + 0)(A + B)$
$= A(1 + B)$	$= A + 0 \cdot B$
$= A \cdot 1$	$= A + 0$
$= A$	$= A$

Observe-se que, quando usamos um teorema básico para provar a Eq. (1.15-6*a*), usamos seu dual para provar a Eq. (1.15-6*b*). Assim, quando de um lado usamos $A = A \cdot 1$, do outro lado usamos $A = A + 0$; quando de um lado usamos a lei distributiva na forma da Eq. (1.15-9*a*), do outro usamos sua dual da Eq. (1.15-9*b*). Assim, generalizando, estabelecemos o princípio da dualidade, pois, se podemos provar determinado teorema seguindo uma seqüência de passos, então podemos provar seu dual simplesmente substituindo cada passo por seu dual.

1.16 O TEOREMA DE DE MORGAN

Um teorema tem suficiente importância para ser examinado à parte. Conhecido como *teorema de De Morgan*, aplica-se a um número arbitrário de variáveis e, em suas formas duais, é dado por

$$\overline{A \cdot B \cdot C \cdots} = \bar{A} + \bar{B} + \bar{C} + \cdots \tag{1.16-1}$$

$$\overline{A + B + C + \cdots} = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdots \tag{1.16-2}$$

Estes teoremas querem dizer que (1) o complemento de um produto de variáveis é igual à soma dos complementos de cada variável e (2) que o complemento de uma soma de variáveis é igual ao produto dos complementos de cada uma das variáveis.

O teorema é facilmente comprovado para duas variáveis construindo uma tabela verdade para testar as quatro possíveis combinações. Após provar que $\overline{AB} = \bar{A} + \bar{B}$, podemos examinar o caso de três variáveis $\overline{ABC}$. Fazemos $Z = AB$ e, portanto, $\overline{ABC} = \overline{ZC}$. Este é um caso de duas variáveis e podemos escrever que $\overline{ZC} = \bar{Z} + \bar{C}$. Aplicando, agora, o teorema de duas variáveis a $\bar{Z} = \overline{AB}$, temos $\overline{ABC} = \overline{ZC} = \bar{Z} + \bar{C} = \bar{A} + \bar{B} + \bar{C}$. Tendo provado o teorema para $2 + 1 = 3$ variáveis, podemos seguir o mesmo processo e prová-lo para $3 + 1 = 4$ variáveis e assim por diante para um número arbitrário de variáveis. O teorema dual $\overline{A + B + C \ldots} = \bar{A} \cdot \bar{B} \cdot \bar{C} \ldots$ é provado de modo similar.

1.17 DIAGRAMAS DE VENN

Existe uma formação geométrica interessante que pode ser colocada em correspondência biunívoca com os conceitos de variáveis lógicas. Estas formas, conhecidas como *diagramas de Venn*, são úteis na visualização geométrica dos teoremas da álgebra booleana.

Na Fig. 1.17-1 o espaço dentro do retângulo é o nosso universo de interesse. Dividimos este universo em duas regiões mutuamente exclusivas: a região dentro da circunferência e a região fora dela. Como existem apenas duas regiões e elas são mutuamente exclusivas, podemos associá-las

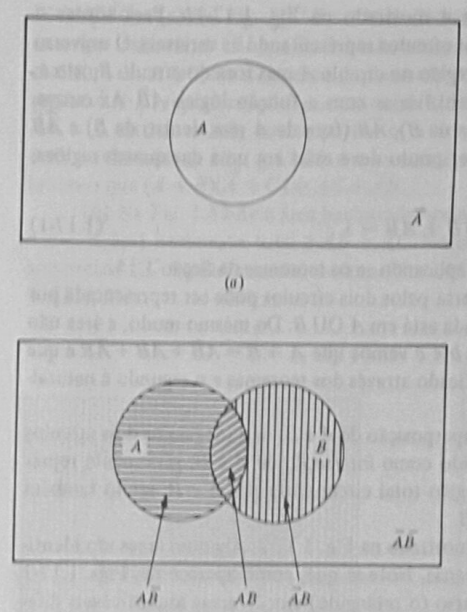


Fig. 1.17-1 (*a*) Diagrama de Venn para uma variável. (*b*), (*c*) Diagramas de duas variáveis. Algumas áreas são identificadas em função de suas representações booleanas.

com a variável lógica A. Associamos a região dentro da circunferência com a variável lógica A e a região fora da circunferência com a variável lógica $\bar{A}$. O teorema booleano $A + \bar{A} = 1$ pode receber a interpretação geométrica que é sempre verdade (1 lógico) que um ponto está dentro de uma circunferência *ou* fora dela. Logicamente, o 1 em $A + \bar{A} = 1$ representa a verdade inevitável que um ponto deve estar dentro ou fora da circunferência. Geometricamente, o 1 representa o universo de interesse, isto é, a totalidade dos pontos do retângulo.

Um diagrama de Venn para duas variáveis é mostrado na Fig. 1.17-1*b*. Para tornar o diagrama mais geral, superpusemos parcialmente os círculos representando as variáveis. O universo agora está dividido em quatro regiões. Existe uma região no círculo A mas fora do círculo B, isto é, no círculo A e *não* no círculo B. Esta região identifica-se com a função lógica $A\bar{B}$. As outras regiões correspondem, então, a AB (dentro de A e de B), $\bar{A}B$ (fora de A mas dentro de B) e $\bar{A}\bar{B}$ (fora de A e, também, fora de B). Como qualquer ponto deve estar em uma das quatro regiões, obtemos

$$A\bar{B} + AB + \bar{A}B + \bar{A}\bar{B} = 1 \qquad (1.17\text{-}1)$$

A validade da Eq. (1.17-1) é facilmente verificada aplicando-se os teoremas da Seção 1.15.

Como vemos na Fig. 1.17-1*c*, a região coberta pelos dois círculos pode ser representada por $A + B$, isto é, qualquer ponto na região sombreada está em A OU B. Do mesmo modo, a área não sombreada é $\overline{A + B}$. Comparando as Figs. 1.17-1*b* e *c* vemos que $A + B = A\bar{B} + AB + \bar{A}B$ e que $\bar{A}\bar{B} = \overline{A + B}$. O primeiro resultado pode ser verificado através dos teoremas e o segundo é naturalmente conhecido como o teorema de De Morgan.

Na Fig. 1.17-1*b* a região AB é a região de superposição de A e B; nesta região os dois círculos se intersecionam. Por isso AB também é conhecido como *interseção* de A e B, geralmente representada por $A \cap B$. Similarmente, $A + B$ é a região total circunscrita por A e B, sendo também chamada *união* de A e B e representada por $A \cup B$.

Um diagrama de Venn de três variáveis é mostrado na Fig. 1.17-2. Algumas áreas são identificadas em termos de suas representações booleanas. Note-se que, como aparece nas Figs. 1.17-1 e 1.17-2, os diagramas de Venn dividem o universo (o retângulo) em 2^n áreas identificáveis diferentes, onde n é o número de variáveis. O número de áreas é, pois, igual ao número de linhas na tabela verdade de mesmo número de variáveis.

Os diagramas de Venn são úteis quando se deseja a visualização geométrica de funções booleanas, podendo também ser usados para estabelecer ou verificar teoremas algébricos booleanos, como no exemplo seguinte.

Exemplo 1.17-1 (*a*) Usar um diagrama de Venn para verificar o teorema da Eq. (1.15-10*b*), isto é, $(A + B)(\bar{A} + C) = AC + \bar{A}B$. (*b*) Usar um diagrama de Venn para verificar o teorema da Eq. (1.15-11*a*), ou seja, $AB + \bar{A}C + BC = AB + \bar{A}C$.

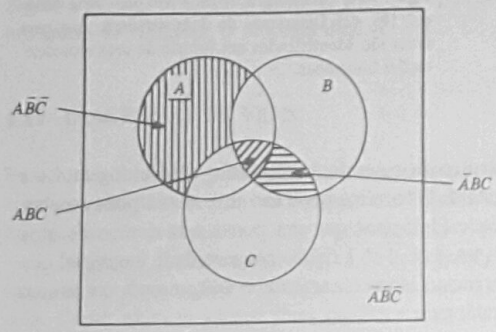


Fig. 1.17-2 Um diagrama de Venn de três variáveis. Algumas áreas são identificadas em função de suas representações booleanas.

SOLUÇÃO: (*a*) Na Fig. 1.17-3*a* o hachuramento horizontal cobre a região $A + B$, isto é, toda a área que está em A OU em B. O hachuramento vertical cobre a região $\bar{A} + C$, isto é, a área que está fora de A OU em C. A área duplamente sombreada está tanto na região $A + B$ E também na região $\bar{A} + C$, representando, portanto, $(A + B)(\bar{A} + C)$. Na Fig. 1.17-3*b* o hachurado horizontal é AC, e o vertical é $\bar{A}B$, sendo, portanto, a área sombreada total $AC + \bar{A}B$. Finalmente, como a área sombreada na Fig. 1.17-3*b* é a mesma área duplamente sombreada na Fig. 1.17-3*a*, estabelecemos que $(A + B)(\bar{A} + C) = AC + \bar{A}B$.

(*b*) Na Fig. 1.17-3*c* a área hachurada horizontalmente é AB e a área hachurada verticalmente é $\bar{A}C$. A área hachurada total é $AB + \bar{A}C$. Notamos que esta área inclui BC embora não tenhamos sombreado BC explicitamente. Assim, $AB + \bar{A}C$ automaticamente inclui BC, e $AB + \bar{A}C = AB + \bar{A}C + BC$.

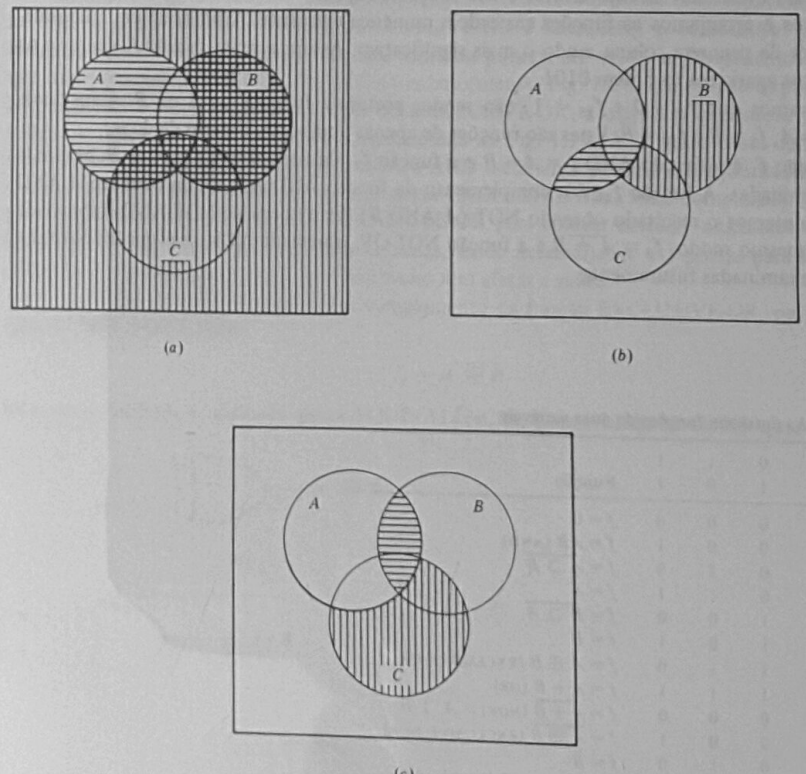


Fig. 1.17-3 (*a*) Hachuramento horizontal é $A + B$. Hachuramento vertical é $\bar{A} + C$. Hachuramento duplo é $(A + B)(\bar{A} + C)$. (*b*) Hachuramento horizontal é AC. Hachuramento vertical é $\bar{A}B$. A área hachurada total é $AC + \bar{A}B$. (*c*) Hachuramento horizontal é AB, hachuramento vertical é $\bar{A}C$. Área hachurada total $AB + \bar{A}C$ inclui BC.

Diagramas de Venn para mais de três variáveis não são muito usados por serem difíceis de desenhar e de aparência confusa (ver Prob. 1.17-1).

1.18 AS FUNÇÕES DE DUAS VARIÁVEIS

Na Seção 1.15 mostramos que, em princípio, existem dezesseis funções de duas variáveis e examinamos as funções AND e OR. Nesta seção vamos examinar as funções restantes.

Uma tabela verdade de duas variáveis tem quatro linhas e nesta tabela uma função é definida especificando-se as quatro entradas V ou F (ou 0 ou 1) na coluna de $Z = f(A, B)$. Como cada entrada permite dois valores e existem quatro entradas, o número total de funções é $2 \times 2 \times 2 \times 2 = 16$. As dezesseis funções são mostradas na Tab. 1.18-1, onde usamos a notação 0, 1. Por conveniência de desenho, o que normalmente aparece como linhas aqui aparece como colunas e vice-versa. Para evitar um arranjo aleatório das funções, supusemos que os valores lógicos 0 e 1 eram números e arranjamos as funções em ordem numérica crescente. Arbitrariamente, consideramos o valor da primeira coluna sendo o mais significativo. Assim, a sexta função f_5 é aquela na qual os dígitos aparecem na ordem 0101.

Observamos que $f_0 = 0$ e $f_{15} = 1$, não sendo, portanto, funções de A ou B. Além disso, $f_3 = A$, $f_{12} = A$, $f_5 = B$ e $f_{10} = \bar{B}$. Estas são funções de apenas uma variável e não de duas.

A função f_1 é a função AND $f = A \cdot B$ e a função f_7 é a função OR $f = A + B$, ambas já por nós examinadas. A função f_{14} é o complemento da função AND $\overline{AB}$; tendo formado A AND B, complementamos o resultado obtendo NOT (A AND B). Esta operação NOT-AND é abreviada NAND. Do mesmo modo, $f_8 = \overline{A + B}$ é a função NOT-OR, abreviada NOR. As funções NAND e NOR serão examinadas futuramente.

Tab. 1.18-1 As dezesseis funções de duas variáveis

A	0	0	1	1	
B	0	1	0	1	Função
f_0	0	0	0	0	$f = 0$
f_1	0	0	0	1	$f = AB$ (AND)
f_2	0	0	1	0	$f = \overline{A \supset B}$
f_3	0	0	1	1	$f = A$
f_4	0	1	0	0	$f = \overline{B \supset A}$
f_5	0	1	0	1	$f = B$
f_6	0	1	1	0	$f = A \oplus B$ (EXCLUSIVE-OR)
f_7	0	1	1	1	$f = A + B$ (OR)
f_8	1	0	0	0	$f = \overline{A + B}$ (NOR) $= A \downarrow B$
f_9	1	0	0	1	$f = \overline{A \oplus B}$ (EXCLUSIVE-NOR)
f_{10}	1	0	1	0	$f = \bar{B}$
f_{11}	1	0	1	1	$f = B \supset A$ (B implica A)
f_{12}	1	1	0	0	$f = \bar{A}$
f_{13}	1	1	0	1	$f = A \supset B$ (A implica B)
f_{14}	1	1	1	0	$f = \overline{AB}$ (NAND) $= A \uparrow B$
f_{15}	1	1	1	1	$f = 1$

1.19 A FUNÇÃO EXCLUSIVE-OR (OU-EXCLUSIVO)

A função f_6 é chamada função EXCLUSIVE-OR e é representada pelo símbolo $\oplus$:

$$f_6 = f(A, B) = A \oplus B \tag{1.19-1}$$

O nome desta função origina-se do fato que $Z = A \oplus B = 1$ quando uma das variáveis A ou B, na *exclusão* da outra, tiver valor lógico 1. Assim, $Z = 1$ se A = 1 ou se B = 1, mas não se ambos A e B, forem 1. Analogamente, a função OR, já estudada (f_7 da Tab. 1.18-1), deveria ser chamada função INCLUSIVE-OR (OU-INCLUSIVO). Geralmente, porém, usa-se apenas OR quando se deseja o significado de INCLUSIVE-OR.

A operação EXCLUSIVE-OR é comutativa ($A \oplus B = B \oplus A$) e associativa $[(A \oplus B) \oplus C = A \oplus (B \oplus C)]$. Estas propriedades podem facilmente ser verificadas (Prob. 1.19-1) usando a tabela verdade.

O símbolo da porta que implementa a função EXCLUSIVE-OR de duas variáveis é mostrado na Fig. 1.19-1*a*. Como a operação EXCLUSIVE-OR é associativa e comutativa, poderíamos imaginar que o mesmo símbolo com três entradas poderia ser usado se desejássemos uma porta que implementasse $A \oplus B \oplus C$. Isto foi feito anteriormente (Fig. 1.8-1) em relação às portas AND e OR, mas não é feito em relação às portas EXCLUSIVE-OR; a estrutura necessária para implementar a função de três variáveis é a representada na Fig. 1.19-1*b*. A razão desta complicação é que a implementação física das portas OR e AND facilmente permite a ligação de entradas adicionais, ao passo que a estrutura de uma porta EXCLUSIVE-OR é mais complicada. Não existe, pois, maneira de alterar uma porta EXCLUSIVE-OR para colocar entradas adicionais, e as portas EXCLUSIVE-OR disponíveis têm duas entradas, sendo necessárias $n - 1$ portas para acomodar n entradas. As entradas podem ser intercambiadas sem afetar a saída.

A função f_9 da Tab. 1.18-1 é o complemento da função EXCLUSIVE-OR, ou seja, f_9 é a função EXCLUSIVE-NOR:

$$f_9 = \overline{A \oplus B} \tag{1.19-2}$$

Esta porta também é chamada porta EQUIVALÊNCIA, pois a saída só será 1 quando $A = B$.

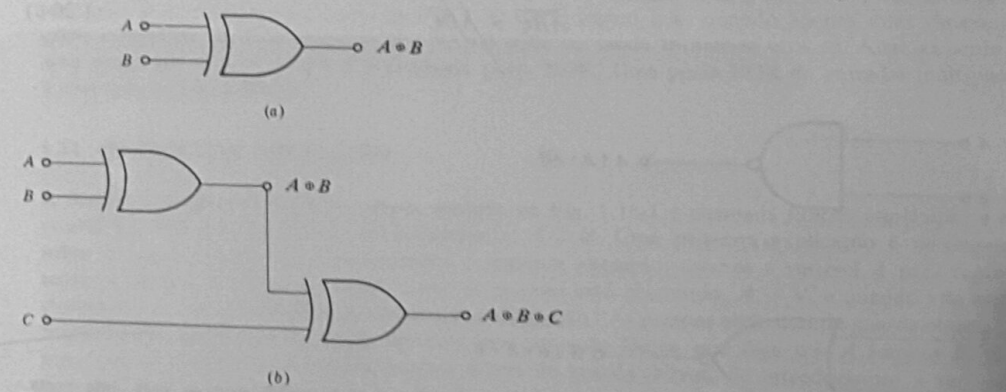


Fig. 1.19-1 (*a*) O símbolo da porta EXCLUSIVE-OR. (*b*) Duas portas usadas para obter $A \oplus B \oplus C$.

1.20 AS FUNÇÕES NAND E NOR

A função f_{14}, representada por seu próprio símbolo, escreve-se

$$f_{14} = A \uparrow B \tag{1.20-1}$$

Como pode ser verificado, comparando f_{14} com f_1 na Tab. 1.18-1

$$f_{14} = A \uparrow B = \overline{AB} \tag{1.20-2}$$

Isto é, f_{14} = NOT (A AND B), que é abreviada por A NAND B. Da mesma forma, podemos representar uma porta NAND por uma porta AND com um círculo de inversão ligado à saída, como na Fig. 1.20-1*a*.

Analogamente, notamos que a função f_8 assume valores 0 e 1 que, em cada caso, são o complemento dos valores da função $f_7 = A$ OR B. Representada por seu símbolo próprio, f_8 é escrita

$$f_8 = A \downarrow B \tag{1.20-3}$$

e

$$f_8 = A \downarrow B = \overline{A + B} \tag{1.20-4}$$

Como f_8 = NOT (A OR B), a função pode ser abreviada por A NOR B. O símbolo para uma porta NOR é mostrado na Fig. 1.20-1*b*.

Tanto a operação NAND como a operação NOR são comutativas; como se pode observar da tabela verdade

$$A \uparrow B = \overline{AB} = B \uparrow A = \overline{BA} \tag{1.20-5}$$

e

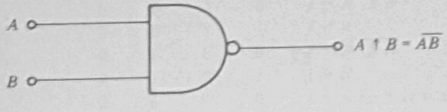
$$A \downarrow B = \overline{A} + \overline{B} = B \downarrow A = \overline{B + A} \tag{1.20-6}$$

Por outro lado, nem a operação NAND nem a operação NOR são associativas (Prob. 1.20-1), ou seja:

$$(A \uparrow B) \uparrow C \neq A \uparrow (B \uparrow C) \tag{1.20-7}$$

ou, na representação alternativa:

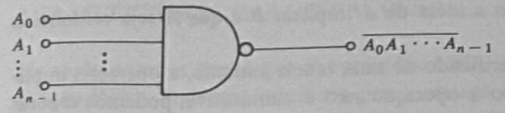
$$\overline{\overline{AB}C} \neq A\overline{\overline{BC}} \tag{1.20-8}$$



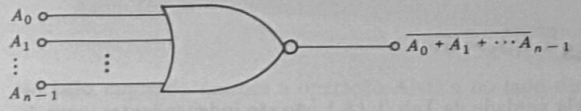
(*a*)

(*b*)

Fig. 1.20-1 (*a*) O símbolo para uma porta NAND. (*b*) O símbolo para uma porta NOR.



(*a*)



(*b*)

Fig. 1.20-2 (*a*) Porta NAND de entradas múltiplas. (*b*) Porta NOR de entradas múltiplas.

Do mesmo modo

$$(A \downarrow B) \downarrow C \neq A \downarrow (B \downarrow C) \tag{1.20-9}$$

ou

$$\overline{\overline{A + B} + C} \neq A\overline{\overline{B + C}} \tag{1.20-10}$$

A operação NAND, como todas as outras operações, envolve apenas duas variáveis. Se desejarmos gerar uma função envolvendo somente a operação NAND e mais de duas variáveis, é necessário especificar a ordem em que as variáveis são operadas por NAND, pois $(A \uparrow B) \uparrow C$ não é o mesmo que $A \uparrow (B \uparrow C)$ que, por sua vez, é diferente de $B \uparrow (A \uparrow C)$ etc. Existe, porém, outra maneira de usar a operação NAND em relação a mais de duas variáveis. Neste caso o termo NAND aplicado a três variáveis significa simplesmente ABC: as variáveis são inicialmente combinadas através da operação AND e, a seguir, a "variável" ABC é complementada. Como a operação AND é comutativa, NAND empregada neste sentido também é comutativa, e não é necessário especificar a ordem das variáveis. Com NAND empregada desta maneira, uma porta NAND de entradas múltiplas seria representada como na Fig. 1.20-2*a*. É lógico que, quando apenas duas variáveis estão envolvidas, as duas maneiras que NAND pode ser usada tornam-se uma só. Analogamente, uma porta que gera $A + \in + C$ é chamada porta NOR. Uma porta NOR de entradas múltiplas é mostrada na Fig. 1.20-2*b*.

1.21 AS FUNÇÕES IMPLICAÇÃO

A função definida como f_{13} na tabela verdade da Fig. 1.18-1 é chamada *função implicação* e é escrita $f_{13} = A$ implica B ou, simbolicamente, $A \supset B$. Uma pequena explicação é necessária sobre o termo *implicação*. Consideremos o seguinte exemplo: usemos a variável A para representar a afirmação "Está chovendo"; isto é, quando está chovendo, A = V, e quando não está chovendo, A = F; usemos B para representar a afirmação "As pessoas estão usando guarda-chuvas". Suponhamos agora que se deseje escrever uma equação lógica que diga que A implica B, ou seja, que a ocorrência de chuva implica o uso de guarda-chuvas. Se introduzirmos a variável $Z = A \supset B$, Z será relacionada a A e B pela maneira descrita por f_{13} na Tab. 1.18-1. Se a situação com respeito a A e B for consistente com a idéia de que A implica B, então Z será verdadeira. A

única possibilidade que não é consistente com a idéia de A implicar B é que A seja verdadeiro e B seja falso. Neste caso Z será falsa.

Como é óbvio, ou pode ser facilmente verificado de uma tabela verdade, a operação implicação não é associativa nem comutativa. Como a operação não é comutativa, podemos esperar encontrar a operação comutada, que é $f_{11} = B \supset A$. Finalmente, as duas operações restantes na Tab. 1.18-1 são $f_2 = A \overline{\supset} B$ e $f_4 = B \overline{\supset} A$.

1.22 RELAÇÃO ENTRE OPERAÇÕES

As operações lógicas definidas pela tabela verdade da Tab. 1.18-1 não são independentes umas das outras. Para mostrar que isto é verdade, apresentamos todas as funções então definidas expressas usando apenas as funções AND e OR e a operação complementação. Temos

$$
\begin{aligned}
&f_0 = A\bar{A} = B\bar{B} = 0 && f_1 = AB && f_2 = A\bar{B} \\
&f_3 = A && f_4 = \bar{A}B && f_5 = B \\
&f_6 = A\bar{B} + \bar{A}B && f_7 = A + B && f_8 = \overline{A + B} = \bar{A}\bar{B} \\
&f_9 = \bar{A}\bar{B} + AB && f_{10} = \bar{B} && f_{11} = A + \bar{B} \\
&f_{12} = \bar{A} && f_{13} = \bar{A} + B && f_{14} = \overline{AB} = \bar{A} + \bar{B} \\
&f_{15} = A + \bar{A} = B + \bar{B} = 1
\end{aligned}
\qquad (1.22\text{-}1)
$$

A correção de cada uma das expressões da Eq. (1.22-1) pode ser facilmente verificada usando uma tabela verdade. Por exemplo, consideremos $f_6 = \bar{A}B + A\bar{B}$, que afirmamos ser a mesma função que $A \oplus B$. Temos

A	B	$\bar{A}$	$\bar{B}$	$A\bar{B}$	$\bar{A}B$	$A\bar{B} + \bar{A}B$
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	0

Comparando a coluna de $\bar{A}B + A\bar{B}$ com a coluna de $f_6 = A \oplus B$ na Tab. 1.18-1, verificamos que realmente $\bar{A}B + A\bar{B} = A \oplus B$. Do mesmo modo, as demais equivalências indicadas na Eq. (1.22-1) podem ser verificadas. Na tabulação das equivalências usamos o teorema de De Morgan para escrever f_8 e f_{14} de duas maneiras diferentes.

Note-se que qualquer função lógica de variáveis pode ser expressa em termos das operações OR e AND (que ligam duas variáveis) e da operação NOT (que é aplicada a uma variável). Esta situação ocorre quer haja duas variáveis ou mais. No caso de diversas variáveis, as funções são geradas através da aplicação repetida de operações que ligam duas variáveis. Notamos, ainda, a facilidade de uso das operações OR e AND, uma vez que ambas são comutativas e associativas. Assim, não é surpreendente que, em geral, as funções lógicas sejam expressas em termos das operações AND, OR e NOT.

1.23 SUFICIÊNCIA DAS OPERAÇÕES

Na seção anterior vimos que as operações AND, OR e NOT são suficientes para expressar qualquer função lógica. Podemos seguir um passo adiante e observar que a operação NOT, juntamente com a operação AND *ou* juntamente com a operação OR, é suficiente. Verifica-se facilmente que a operação AND pode ser expressa em termos de OR e NOT ou, alternativamente, que OR pode ser expressa em termos de AND e NOT.

Assim temos

$$AB = \overline{\bar{A} + \bar{B}} \qquad (1.23\text{-}2)$$

No lado esquerdo usamos a operação AND e no lado direito as operações OR e NOT. De modo similar, temos:

$$A + B = \overline{\bar{A}\bar{B}} \qquad (1.23\text{-}3)$$

onde substituímos uma operação OR por uma operação AND e uma NOT. As Eqs. (1.23-2) e (1.23-3) podem ser reconhecidas como simples aplicações do teorema de De Morgan.

Apesar de ser possível dispensar o uso da operação OR ou da operação AND, geralmente isto não é feito.

1.24 SUFICIÊNCIA DE NAND; SUFICIÊNCIA DE NOR

Tendo observado a suficiência da combinação de AND e NOT e da combinação de OR e NOT, verificaremos agora que a operação NAND é por si só suficiente, do mesmo modo que a operação NOR é por si só suficiente. Consideremos, por exemplo, a operação NAND. Temos

$$A \uparrow A = A\bar{A} = \bar{A} \qquad (1.24\text{-}1)$$

e

$$(A \uparrow A) \uparrow (B \uparrow B) = \overline{\overline{A\bar{A}}\overline{B\bar{B}}} = \bar{A}\bar{B} = A + B \qquad (1.24\text{-}2)$$

observamos, pois, da Eq. (1.24-1), que, se aplicarmos a variável A a ambas as entradas de uma porta NAND de duas entradas, a saída será $\bar{A}$; a operação NAND pode, pois, executar a operação NOT sobre uma variável A. Da Eq. (1.24-2) observamos que a aplicação repetida da operação NAND nos permite obter a função $A + B$. [A igualdade à direita da Eq. (1.24-2) é uma aplicação do teorema de De Morgan.]

Finalizando, comprovamos que a operação NAND pode substituir tanto a operação OR como a operação NOT. Como foi visto anteriormente, a combinação de OR e NOT é suficiente para gerar qualquer função, e segue-se que a operação NAND é, *por si só*, suficiente. De modo similar podemos estabelecer a suficiência de NOR.

Freqüentemente é conveniente projetar um circuito lógico usando somente um tipo de porta lógica. As portas NAND e NOR são apropriadas para este propósito, pois as operações NAND e NOR são suficientes e, conseqüentemente, qualquer operação lógica pode ser efetuada por um arranjo de portas idênticas. A implementação lógica com portas NAND ou NOR é, todavia, inconveniente por não serem as operações associativas. O procedimento usual é, então, projetar uma estrutura usando portas AND, OR e NOT e, posteriormente, transformá-la em uma estrutura equivalente que use apenas portas NAND ou NOR conforme seja desejado. Este processo é muito simples e é apresentado na Seção 2.7.

1.25 EXEMPLOS DE APLICAÇÃO DOS TEOREMAS DA ÁLGEBRA DE BOOLE

Introduzimos o conceito de variável lógica e o conceito de função de uma variável e de duas variáveis. Uma operação lógica que une duas variáveis, como a função AND etc., é muitas vezes denominada de *conetivo*. Funções de mais de duas variáveis são formadas pela aplicação repetida destes conetivos de duas variáveis.

Com freqüência encontraremos expressões algébricas booleanas suscetíveis de simplificação pela aplicação dos teoremas booleanos [Eqs. (1.15-2 a 1.15-11)] em virtude de, conforme descrito na Seção 1.22, poderem as operações lógicas ser expressas em função umas das outras, além de algumas delas serem comutativas, associativas ou ambas.

Nesta seção apresentaremos um conjunto de exemplos destas simplificações. O exemplo será resolvido no lado esquerdo, enquanto o princípio, teorema ou outra justificação para o processo será mencionado à direita. Devemos evitar confusão entre os símbolos usados no exemplo e os símbolos usados para expressar um teorema. Por esta razão usaremos letras maiúsculas do princípio do alfabeto para expressar os teoremas (A, B, C etc.) e letras minúsculas do final do alfabeto para os exemplos ($\ldots, x, y, z$).

Exemplo 1.25-1 Simplificar $w = xy + \bar{y}\bar{x}z$.

$w = xy + \bar{x}\bar{y}z$ — $AB = BA$ (comutatividade de AND)

Se $v = xy$; — A função de variáveis lógicas é, por si só, uma variável

então $v = xy$

e $w = v + \bar{u}z$

$= v + z = xy + z$ — $A + \bar{A}B = A + B$ [Eq. (1.15-8a)]

Observe-se que $\overline{xy}$ é o complemento de xy, ao passo que $\bar{x}\bar{y}$ *não* é o complemento de xy.

Exemplo 1.25-2 Simplificar $w = x(\bar{x} + y)$.

$w = x(\bar{x} + y) = xy$ — $A(\bar{A} + B) = AB$ [Eq. (1.15-8b)]

ou, alternativamente:

$w = x(\bar{x} + y) = x\bar{x} + xy$ — $A(B + C) = AB + AC$ [Eq. (1.15-9b)]

$= 0 + xy$ — $A\bar{A} = 0$

$= xy$ — $A + 0 = A$

Exemplo 1.25-3 Simplificar $w = \bar{x}(x + y) + \bar{z} + zy$.

$\bar{x}(x + y) + \bar{z} + zy = \bar{x}(x + y) + \bar{z} + y$ — $A + \bar{A}B = A + B$ [Eq. (1.15-8a)]; A é identificado com $\bar{z}$, de modo que $\bar{A} = z$

$= \bar{x}x + \bar{x}y + \bar{z} + y = \bar{x}y + \bar{z} + y$ — $A(B + C) = AB + AC$ (1.15-9b) e $\bar{A}A = A\bar{A} = 0$

$= y + \bar{x}y + \bar{z} = y + y\bar{x} + z$ — $A + B = B + A$; $AB = BA$ (comutatividade de OR e AND)

$= y + \bar{z}$ — $A + AB = A$ (1.15-6a)

Os exemplos seguintes ilustram o uso do teorema de De Morgan.

Exemplo 1.25-4 Simplificar $v = \overline{w + w\bar{x} + yz}$.

$\overline{w + w\bar{x} + yz} = \bar{w} \cdot \overline{w\bar{x}} \cdot \overline{yz}$ — Teorema de De Morgan [Eq. (1.16-2)]

$= \bar{w}(\bar{w} + \bar{\bar{x}})(\bar{y} + \bar{z})$ — Teorema de De Morgan novamente

$= \bar{w}(\bar{w} + x)(\bar{y} + \bar{z})$ — $\bar{\bar{A}} = A$ [Eq. (1.15-1)]

$= \bar{w}(\bar{y} + \bar{z})$ — $A(A + B) = A$ [Eq. (1.15-6b); $A = \bar{w}$, $\bar{A} = \bar{w}$]

Alternativamente, poderíamos simplificar antes de aplicar o teorema de De Morgan. Neste caso teríamos

$\overline{w + w\bar{x} + yz} = \overline{w + yz}$ — $A + AB = A$ [Eq. (1.15-6b)]

$\overline{w + yz} = \overline{w} \cdot \overline{yz}$ — Teorema de De Morgan [Eq. (1.16-2)]

$= \bar{w}(\bar{y} + \bar{z})$ — Teorema de De Morgan novamente

Exemplo 1.25-5 Simplificar $v = \overline{w[(x + y(z + \bar{w})]}$.

$\overline{w[(x + y(z + \bar{w})]} = \bar{w} + \overline{x + y(z + \bar{w})}$ Teorema de De Morgan [Eq. (1.16-1)]

$= \bar{w} + \bar{x} \cdot \overline{y(z + \bar{w})}$ — Teorema de De Morgan novamente [Eq. (1.16-2)]

$= \bar{w} + \bar{x}(\bar{y} + \overline{z + \bar{w}})$ — Novamente

$= \bar{w} + \bar{x}(\bar{y} + \bar{z}w)$ — Novamente e $\bar{\bar{A}} = A$

$= \bar{w} + \bar{x}\bar{y} + \bar{x}\bar{z}w$ — $A(B + C) = AB + AC$ [Eq. (1.15-9b)]

$= \bar{w} + \bar{x}\bar{y} + \bar{x}\bar{z}$ — $A + \bar{A}B = A + B$ [Eq. (1.15-8a)]; $A = \bar{w}$, $\bar{A} = w$

Podemos fatorar $\bar{x}$ dos dois últimos termos levando a uma expressão alternativa e talvez mais simples

$\bar{w} + \bar{x}\bar{y} + \bar{x}\bar{z} = \bar{w} + \bar{x}(\bar{y} + \bar{z})$ — $AB + AC = A(B + C)$ [Eq. (1.15-9b)]

O exemplo seguinte ilustra o uso do teorema da Eq. (1.15-11). Note-se na Eq. (1.15-11a) que os termos nos dois lados da equação são idênticos, exceto que o termo BC (que é o produto dos coeficientes de A e $\bar{A}$) pode ser incluído ou excluído à vontade. O comentário correspondente aplica-se ao fator $B + C$ na Eq. (1.15-11b).

Exemplo 1.25-6 Simplificar $v = wx + x\bar{y} + yz + x\bar{z}$.

$v = wx + x\bar{y} + yz + x\bar{z} + xy$	Da Eq. (1.15-11*a*); como temos os termos zy e $x\bar{z}$, podemos adicionar o termo xy
$= wx + x(\bar{y} + y) + yz + x\bar{z}$	Fatorando x do segundo e quinto termos conforme permitido por $AB + AC = A(B + C)$ [Eq. (1.15-9*b*)]
$= wx + x + yz + x\bar{z}$	$\bar{A} + A = 1$ [Eq. (1.15-5*a*)] e $A \cdot 1 = A$ [Eq. (1.15-2*b*)]
$= x + yz + x\bar{z}$	$A + AB = A$ [Eq. (1.15-6*a*)]
$= x + yz$	$A + AB = A$

Exemplo 1.25-7 Simplificar $v = (w + x + y)(w + \bar{x} + y)(\bar{y} + z)(w + z)$.

$v = (w + y)(\bar{y} + z)(w + z)$	Da Eq. (1.15-7*b*) aplicada aos primeiros dois fatores de v; $A = w + y$, $B = x$
$= (w + y)(\bar{y} + z)$	Da Eq. (1.15-11*b*)

1.26 . EXEMPLOS ADICIONAIS

Esta seção traz alguns exemplos adicionais que mostram como afirmações e condições relativamente complexas podem ser expressas sob a forma de expressões algébricas booleanas. A partir daí usamos os teoremas da álgebra booleana para encontrar expressões mais simples e equivalentes, ilustrando, assim, como o conceito de variável lógica e sua álgebra associada é usado para executar um processo de raciocínio lógico.

Exemplo 1.26-1 Uma estudante consulta o catálogo da universidade e fica sabendo que pode matricular-se em determinada disciplina de eletrônica somente se ela satisfizer uma das seguintes condições:

1. Já completou sessenta créditos e é uma estudante de engenharia regularmente matriculada.
2. Ou completou sessenta créditos e é uma estudante de engenharia e tem o consentimento do departamento.
3. Ou completou menos de sessenta créditos e é uma estudante de engenharia com matrícula especial.
4. Ou é uma estudante regularmente matriculada e tem o consentimento do departamento.
5. Ou é uma estudante de engenharia e não tem o consentimento do departamento.

Encontre uma expressão mais simples que indique a possibilidade de a estudante matricular-se no curso.

SOLUÇÃO: Introduzimos as variáveis w, x, y, z e v para representar as seguintes situações:

w = a estudante completou sessenta créditos
x = a estudante é aluna de engenharia
y = a estudante tem matrícula regular
z = a estudante tem consentimento do departamento
v = a estudante pode matricular-se na disciplina

Assim, y = V (verdadeiro) ou $y = 1$ representa a proposição que é verdade que a estudante tem matrícula regular. De maneira correspondente, $\bar{y} = 1$ ou $y = 0$ significa que a estudante tem matrícula especial. Quando as variáveis w, x, y e z assumem valores tais que v = verdadeiro, a estudante pode matricular-se na disciplina. A especificação de quando isto ocorre pode ser feita pela equação algébrica lógica

$$v = wxy + wxz + \bar{w}x\bar{y} + yz + x\bar{z} \tag{1.26-1}$$

Se a condição 1 for satisfeita, isto é, $w = 1$, $x = 1$ e $y = 1$ simultaneamente, então o termo wxy na Eq. (1.26-1) é $wxy = 1$, e conseqüentemente $v = 1$, independentemente dos valores lógicos dos demais termos da Eq. (1.26-1). Os diversos termos da Eq. (1.26-1) são ligados pelo conetivo OR correspondente à palavra "OU" na especificação original das condições.

A simplificação pode ser efetuada da seguinte maneira:

$v = wxy + \bar{w}x\bar{y} + yz + x(\bar{z} + zw)$	Fatorar x do segundo e quinto termos
$= wxy + \bar{w}x\bar{y} + yz + x(\bar{z} + w)$	Da Eq. (1.15-8*a*)
$= wxy + \bar{w}x\bar{y} + yz + x\bar{z} + wx$	Da Eq. (1.15-6*b*)
$= wx(y + 1) + \bar{w}x\bar{y} + yz + x\bar{z}$	Fatorando wx do primeiro e último termos
$= wx + \bar{w}x\bar{y} + yz + x\bar{z}$	$A + 1 = 1$; $A \cdot 1 = A$
$= x(w + \bar{w}\bar{y}) + yz + x\bar{z}$	Fatorando x do primeiro e segundo termos
$= x(w + \bar{y}) + yz + x\bar{z}$	Da Eq. (1.15-8*a*)
$= xw + x\bar{y} + yz + x\bar{z}$	Da Eq. (1.15-6*b*)

A expressão resultante é idêntica à expressão do exemplo 1.25-6, portanto:

$$v = x + yz \tag{1.26-2}$$

Assim, a estudante poderá matricular-se na disciplina ($v = 1$) se for uma aluna de engenharia ($x = 1$) ou se simultaneamente tiver matrícula regular e consentimento do departamento ($y = z = 1$).

Exemplo 1.26-2 Há cinco livros em uma prateleira v, w, x, y e z. Você deve selecionar alguns entre eles de modo a satisfazer todas as condições a seguir:

1. Selecionar v ou w ou ambos.
2. Selecionar x ou z mas não ambos.
3. Selecionar v e z juntos ou nenhum dos dois.

4. Se selecionar y também deve selecionar z.

5. Se selecionar w também deve selecionar v e y.

Façamos com que as variáveis v, w, x, y e z (que usamos para representar os livros) também representem as proposições "o livro v foi selecionado", "o livro w foi selecionado" etc. Façamos u representar a proposição que a seleção feita obedece a todos os requisitos. Neste caso a relação lógica entre as variáveis é

$$u = (v + w)(x \oplus y)\overline{(v \oplus z)}(y \supset z)(w \supset vy) \qquad (1.26\text{-}3)$$

Cada uma das expressões entre parênteses se refere a uma das condições acima. Como o problema exige que todas as condições sejam satisfeitas, elas são ligadas pela função AND. O primeiro parênteses envolve a função INCLUSIVE-OR, conforme a especificação da condição 1, enquanto o segundo parênteses envolve a função EXCLUSIVE-OR, conforme a condição 2. O conetivo EXCLUSIVE-OR requer a seleção de uma ou outra das variáveis, mas não ambas nem nenhuma. De maneira correspondente, o complemento de EXCLUSIVE-OR requer a seleção de ambas as variáveis ou nenhuma delas, exatamente como especificado na condição 3 para as variáveis v e z. Assim, o terceiro parênteses da Eq. (1.26-3) é $\overline{v \oplus z}$. A condição 4 especifica que a seleção de y *implica* a seleção de z, isto é, $y \supset z$, e, finalmente, a condição 5 requer que w implique ambos, v e y, ou seja, $w \supset vy$.

Para permitir a manipulação da Eq. (1.26-3) reescrevemos a equação usando somente as operações AND, OR e NOT. Da Tab. 1.18-1 e Eq. (1.22-1) obtemos:

$$u = (v + w)(x\bar{z} + \bar{x}z)(vz + \bar{v}\,\bar{z})(\bar{y} + z)(\bar{w} + vy) \qquad (1.26\text{-}4)$$

Temos, então:

$u = (vz + \bar{v}w\bar{z})(x\bar{z} + \bar{x}z)(\bar{y} + z)(\bar{w} + vy)$ — Multiplicando o primeiro e o terceiro parênteses e usando $AA = A$, $A\bar{A} = 0$ e $A + AB = A$

$= (v\bar{w}z + vyz)(x\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{x}z)$ — Multiplicando o primeiro e o quarto parênteses e também o segundo e terceiro parênteses e usando $AA = A$ e $A\bar{A} = 0$

Multiplicando os parênteses e usando $AA = A$ e $A\bar{A} = 0$

$= v\bar{w}\bar{x}\bar{y}z + v\bar{w}\bar{x}z + v\bar{x}yz$ — Fatorando o fator comum conforme $AB + AC = A(B + C)$ [Eq. (1.15-9*b*)]

$= v\bar{x}z(\bar{w}\bar{y} + \bar{w} + y)$ — $A + AB = A$ [Eq. (1.15-6*a*)]

$= v\bar{x}z(\bar{w} + y)$

O resultado seria $u = v\bar{x}z(\bar{w} + y)$, cuja interpretação é: devemos selecionar v e z e rejeitar x e, ao mesmo tempo, se selecionarmos $\bar{w}$, devemos selecionar y (se w for rejeitado, não importa se selecionamos ou rejeitamos y). Alternativamente, o resultado poderia ser escrito $u = v\bar{x}z\bar{w} + v\bar{x}zy$, ou seja, podemos selecionar v e z rejeitando x e w (y opcional), ou ainda podemos selecionar v, z e y rejeitando x (w opcional).

1.27 DIAGRAMAS LÓGICOS

Existe uma correspondência entre uma expressão lógica e um diagrama de portas lógicas. Por exemplo, consideremos a expressão para u dada pela Eq. (1.26-4). Uma estrutura de portas lógicas que gera a função lógica u a partir de v, w, x, y e z é mostrada na Fig. 1.27-1 usando

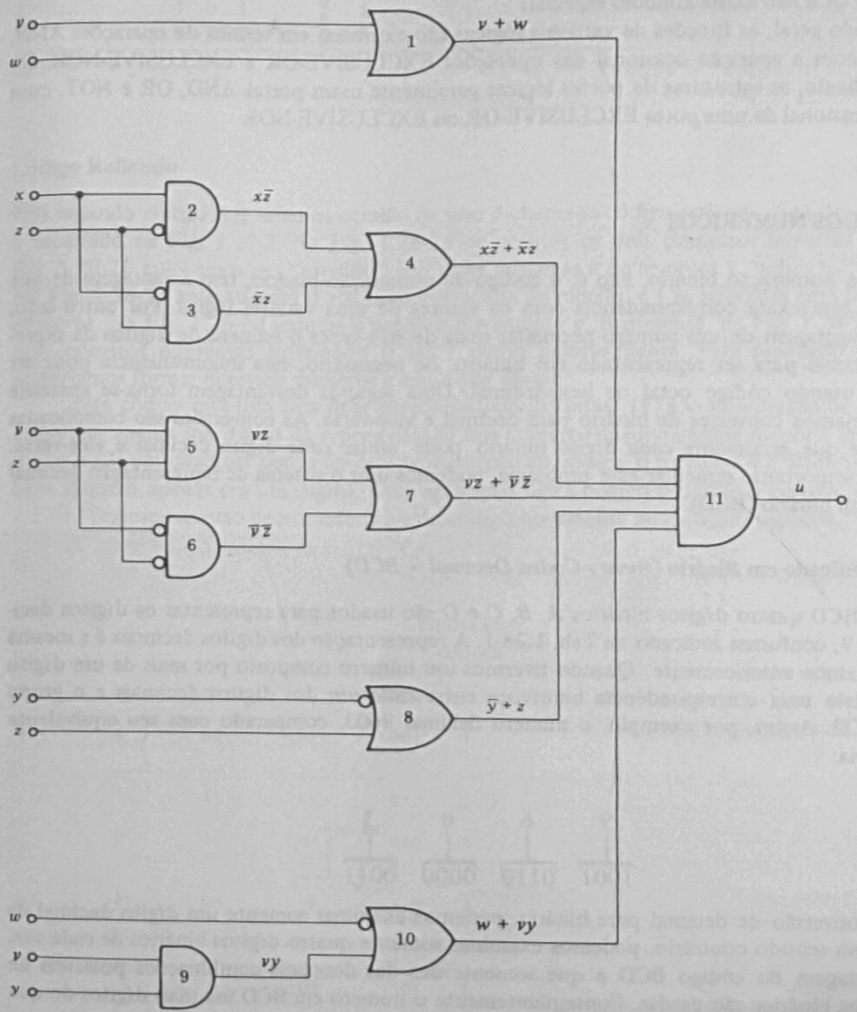


Fig. 1.27-1 O diagrama lógico de uma estrutura de portas que gera a função $u = (v + w)(x\bar{z} + \bar{x}z)(vz + \bar{v}\bar{z})(\bar{y} + z)(\bar{w} + vy)$.

portas AND e OR com a inversão lógica representada por círculos afixados às portas. Em cada caso são especificadas as variáveis ou funções que servem como entradas às portas e que são obtidas como saídas. As três portas marcadas 2, 3 e 4 geram $x \oplus z\ (= x\bar{z} + \bar{x}z)$. Estas três portas poderiam ser substituídas por uma porta EXCLUSIVE-OR, cujo símbolo é dado na Fig. 1.19-1. A saída $vz + \bar{v}\bar{z}$ da porta 7 é, conforme vimos na seção anterior, $vz + \bar{v}\bar{z} = \overline{v \oplus z}$. Assim, as portas 5, 6 e 7 poderiam ser substituídas por uma porta EXCLUSIVE-NOR, cujo símbolo é uma porta EXCLUSIVE-OR com um círculo de inversão na saída. As portas 8 e 10 geram a função implicação, para a qual não existe símbolo especial.

De modo geral, as funções de variáveis lógicas são expressas em termos de operações AND, OR e NOT com a aparição ocasional das operações EXCLUSIVE-OR e EXCLUSIVE-NOR. De modo semelhante, as estruturas de portas lógicas geralmente usam portas AND, OR e NOT, com a aparição ocasional de uma porta EXCLUSIVE-OR ou EXCLUSIVE-NOR.

1.28 CÓDIGOS NUMÉRICOS

O sistema de numeração binário, isto é, o código de numeração binário, tem a vantagem de que seus dígitos têm exata correspondência com os valores de uma variável lógica. Por outro lado, existe a desvantagem de um número necessitar mais de três vezes o número de dígitos da representação decimal para ser representado em binário. Se necessário, esta inconveniência pode ser contornada usando código octal ou hexadecimal. Uma segunda desvantagem torna-se aparente quando desejamos converter de binário para decimal e vice-versa. As conversões são complicadas pelo fato de que geralmente *cada* dígito binário pode afetar *cada* dígito decimal e vice-versa. Quando for importante remediar este problema, podemos usar o sistema de representação decimal codificado em binário (BCD).

Decimal Codificado em Binário (*Binary-Coded Decimal – BCD*)

No sistema BCD quatro dígitos binários *A, B, C* e *D* são usados para representar os dígitos decimais de 0 a 9, conforme indicado na Tab. 1.28-1. A representação dos dígitos decimais é a mesma que encontramos anteriormente. Quando tivermos um número composto por mais de um dígito decimal, existe uma correspondência biunívoca entre *cada um* dos dígitos decimais e o grupo binário *ABCD*. Assim, por exemplo, o número decimal 9603, comparado com seu equivalente em BCD, seria:

9	6	0	3
1001	0110	0000	0011

Assim, na conversão de decimal para binário, podemos examinar somente um dígito decimal de cada vez e, no sentido contrário, podemos examinar somente quatro dígitos binários de cada vez. Uma desvantagem do código BCD é que somente dez das dezesseis combinações possíveis de quatro dígitos binários são usadas. Conseqüentemente o número em BCD usa mais dígitos do que se fosse representado em binário. Uma segunda desvantagem aparece quando tentamos executar operações aritméticas com números BCD (ver Seção 5.14).

Tabela 1.28-1 Representação BCD dos dígitos decin s

Dígito decimal	BCD *A*	*B*	*C*	*D*	Dígito decimal	BCD *A*	*B*	*C*	*D*
0	0	0	0	0	5	0	1	0	1
1	0	0	0	1	6	0	1	1	0
2	0	0	1	0	7	0	1	1	1
3	0	0	1	1	8	1	0	0	0
4	0	1	0	0	9	1	0	0	1

Código Refletido

Um segundo código que teremos ocasião de usar é chamado *código refletido*. Seu desenvolvimento é mostrado na Fig. 1.28-2. Na Fig. 1.28-2*a* colocamos os dois primeiros números binários. Na Fig. 1.28-2*b* colocamos um "espelho" sob esses números e escrevemos a "reflexão" dos mesmos. Na Fig. 1.28-2*c* antepusemos 0 aos números acima do espelho e 1 às reflexões para formar quatro números. Na Fig. 1.28-2*d* usamos o espelho novamente e, finalmente, na Fig. 1.28-2*e* formamos oito números diferentes antepondo 0 ou 1 como anteriormente. Pela repetição do processo, poderíamos formar dezesseis números e assim por diante.

Este código refletido tem duas características distintas: (1) a representação de cada número difere em apenas *um* dígito da representação dos números imediatamente menor e imediatamente maior e (2) em qualquer ponto do desenvolvimento do código, o primeiro e o último número também diferem apenas em um dígito. Uma aplicação deste código refletido é considerada no Prob. 1.27-1. Teremos ocasião de nos referir a este código novamente no capítulo seguinte.

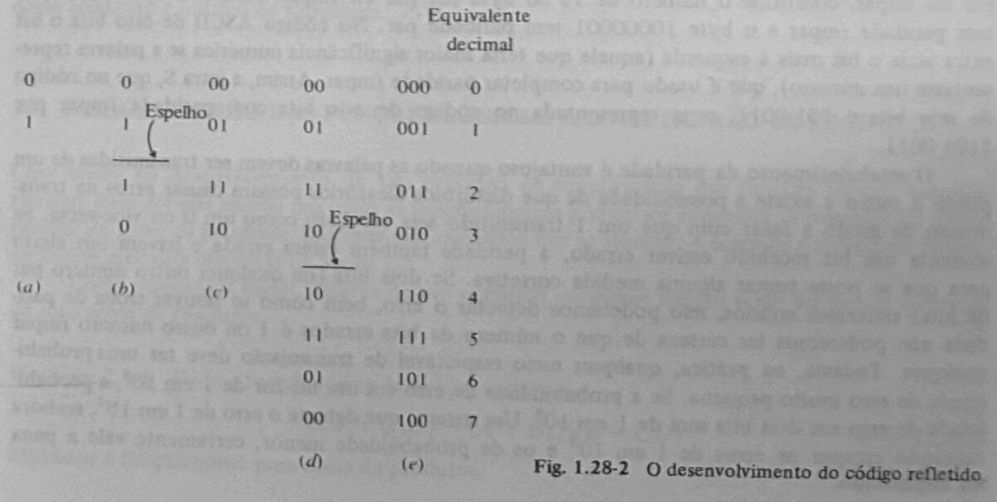


Fig. 1.28-2 O desenvolvimento do código refletido.

1.29 NOMENCLATURA

Os valores possíveis de uma variável lógica que podem ou não representar os dígitos numéricos do sistema binário são chamados dígitos binários (*binary digits*) e abreviados como *bits*. Um arranjo ou grupo de bits que carrega consigo um item de informação é chamado *palavra* (*word*). Por razões que ficarão abundantemente claras, é conveniente fazer com que as palavras constem de um número de bits que seja potência de 2; esta é a prática usualmente adotada em sistemas digitais.

Um grupo de quatro bits é chamado *nybble* e um grupo de oito bits é chamado *byte*. Se, por acaso, um nybble ou um byte contém um item completo de informação, então o nybble ou byte também é uma palavra. Uma palavra de 32 bits ou de dezesseis bits pode ser também chamada de palavra de quatro bytes ou de dois bytes, respectivamente.

1.30 CÓDIGOS DE DADOS

Conjuntos de bits, isto é, palavras, podem ser usados para representar dados e informações em geral, além de representar números. Basta que haja um entendimento, ou seja, um código conhecido pelo transmissor e pelo receptor de dados. Um código largamente usado para a transmissão de dígitos decimais, letras do alfabeto, sinais de pontuação e instruções para teletipo é o código ASCII (American Standard Code for Information Interchange). O código usa sete bits, de modo que $2^7 = 128$ itens diferentes de informação podem ser transmitidos. O código é dado no Apêndice A.

Bit de Paridade

Freqüentemente, um oitavo bit é adicionado ao código ASCII de sete bits, completando a palavra do código para uma de um byte. Em muitos casos o espaço destinado a este bit seria perdido se ele não fosse adicionado, pois muitos sistemas manejam os dados em unidades de bytes inteiros. Esse bit extra pode ser efetivamente usado para dar ao byte *paridade*. Um byte tem paridade par ou ímpar, conforme o número de 1s no byte for par ou ímpar. Assim, o byte 10000011 tem paridade ímpar e o byte 10000001 tem paridade par. No código ASCII de oito bits o bit extra seria o bit mais à esquerda (aquele que teria maior significância numérica se a palavra representasse um número), que é usado para completar paridade ímpar. Assim, a letra S, que no código de sete bits é 101 0011, seria representada no código de oito bits com paridade ímpar por 1101 0011.

O estabelecimento da paridade é vantajoso quando as palavras devem ser transmitidas de um ponto a outro e existe a possibilidade de que distúrbios aleatórios possam causar erros na transmissão de modo a fazer com que um 1 transmitido seja recebido como um 0 ou vice-versa. Se somente um bit recebido estiver errado, a paridade também estará errada e haverá um alerta para que se possa tomar alguma medida corretiva. Se dois bits (ou qualquer outro número par de bits) estiverem errados, não poderemos detectar o erro, bem como se houver troca de paridade não poderemos ter certeza de que o número de bits errados é 1 ou outro número ímpar qualquer. Todavia, na prática, qualquer meio respeitável de transmissão deve ter uma probabilidade de erro muito pequena. Se a probabilidade de erro em um bit for de 1 em 10^4, a probabilidade de erro em dois bits será de 1 em 10^8. Um sistema que detecte o erro de 1 em 10^4, embora deixando escapar os erros de 1 em 10^8 e os de probabilidade menor, certamente vale a pena ser implantado.

CAPÍTULO 2
FUNÇÕES LÓGICAS

No Cap. 1 introduzimos os conceitos de variável lógica e de funções de uma variável lógica. Desenvolvemos, também, alguns teoremas da álgebra booleana que pudemos usar para simplificar expressões algébricas booleanas. O processo pelo qual tais simplicações foram alcançadas não era totalmente satisfatório, pois dependia de nossa habilidade em reconhecer quais teoremas poderiam ser usados com eficiência e como estes teoremas poderiam ser usados. O assunto deste capítulo são os processos mais gerais e sistemáticos que levam a simplificações eficazes. De especial utilidade neste contexto é o mapa de Karnaugh.

2.1 FORMAS PADRÃO DAS FUNÇÕES LÓGICAS: A SOMA PADRÃO DE PRODUTOS

Com vistas ao desenvolvimento de um processo de simplificação das funções lógicas, introduziremos duas *formas padrão* nas quais as funções lógicas podem ser expressas. A primeira é a *forma padrão de soma de produtos*, ilustrada nos exemplos que se seguem. A segunda é o *produto padrão de somas*, que será discutida na Seção 2.2.

Como veremos a seguir, através de um exemplo, qualquer função pode ser escrita na forma de uma *soma de produtos*.

Exemplo 2.1-1 Dada a função lógica de quatro variáveis

$$f(A,B,C,D) = (\bar{A} + BC)(B + \bar{C}D) \tag{2.1-1}$$

expressar a função como uma soma de produtos.

SOLUÇÃO: Usaremos a lei distributiva [Eq. (1.15-9*b*)], que nos permite remover parênteses através da multiplicação, como na álgebra comum, obtendo

$$f(A,B,C,D) = \bar{A}(B + \bar{C}D) + BC(B + \bar{C}D) \tag{2.1-2}$$

$$= \bar{A}B + \bar{A}\bar{C}D + BBC + BC\bar{C}D \tag{2.1-3}$$

$$= \bar{A}B + \bar{A}\bar{C}D + BC \tag{2.1-4}$$

Como, conforme o teorema da Eq. (1.15-4*b*), $BB = B$, e como $C\bar{C} = 0$, conforme o teorema da Eq. (1.15-5*b*), notamos na Eq. (2.1-4) que f é expressa como uma soma de termos e que cada um dos termos é o produto das variáveis lógicas individuais que aparecem complementadas em alguns casos e não-complementadas em outros.

Exemplo 2.1-2 Dada a função lógica de cinco variáveis

$$f(A,B,C,D,E) = (A + \overline{BC})(\overline{D + BE}) \tag{2.1-5}$$

expressar a função como uma soma de produtos.

SOLUÇÃO: Usando o teorema de De Morgan [Eqs. (1.16-1) e (1.16-2)] e a lei distributiva obtemos

$$f(A,B,C,D,E) = (A + \overline{BC})(\overline{D + BE}) \tag{2.1-6}$$

$$= (A + \bar{B} + \bar{C})[\bar{D}(\overline{BE})] \tag{2.1-7}$$

$$= (A + \bar{B} + \bar{C})[\bar{D}(\bar{B} + \bar{E})] \tag{2.1-8}$$

$$= (A + \bar{B} + \bar{C})(\bar{B}\bar{D} + \bar{D}\bar{E}) \tag{2.1-9}$$

$$= A\bar{B}\bar{D} + A\bar{D}\bar{E} + \bar{B}\bar{D} + \bar{B}\bar{D}\bar{E} + \bar{B}\bar{C}\bar{D} + \bar{C}\bar{D}\bar{E} \tag{2.1-10}$$

Novamente, como no exemplo precedente, a função aparece como uma soma de produtos.

Estes exemplos indicam como uma expressão lógica arbitrária pode ser escrita como uma soma-de-produtos. Se somente variáveis individuais aparecem complementadas, como no primeiro exemplo, necessitamos apenas usar a lei distributiva. Se um sinal de complemento aparecer sobre uma combinação de variáveis, como no segundo exemplo, usaremos o teorema de De Morgan tantas vezes quantas necessárias até que o sinal de complemento apareça apenas sobre variáveis individuais. Em qualquer caso, é sempre possível escrever uma expressão lógica como uma simples soma de termos em que cada termo é um produto de alguma combinação de variáveis, algumas complementadas, outras não. A mesma variável nunca precisa aparecer duas vezes em um produto, pois, se tal repetição de uma variável ou de uma variável complementada ocorresse, poderia ser eliminada através do uso dos teoremas $AA = A$ ou $\bar{A}\bar{A} = \bar{A}$. Se, por outro lado, encontrarmos um termo $A\bar{A}$ em um produto, o termo pode ser descartado pois $A\bar{A} = 0$.

Notamos que nas expressões soma de produtos das Eqs. (2.1-4) e (2.1-10) os termos individuais não envolvem o mesmo número de variáveis e, nestes exemplos em particular, nenhum termo envolve todas as variáveis. Uma padronização maior pode ser efetuada, o que nos levaria a uma expressão na qual *todos* os termos envolvem *todas* as variáveis (complementadas ou não-complementadas). Esta padronização é mostrada no exemplo seguinte.

Exemplo 2.1-3 Consideremos a função lógica de três variáveis

$$f(A,B,C) = A + BC \tag{2.1-11}$$

Esta função já está na forma de uma soma-de-produtos, embora o primeiro termo produto A seja um termo de somente uma variável. Reescrever a Eq. (2.1-11) de modo que se torne uma expressão em que todas as três variáveis apareçam em cada termo produto.

SOLUÇÃO: No primeiro termo da Eq. (2.1-11) não aparece a variável B nem a variável C. Multiplicaremos este termo por $(B + \bar{B})(C + \bar{C})$. Esta operação não altera o valor lógico da função, pois $B + \bar{B} = C + \bar{C} = 1$, conforme a Eq. (1.15-5*a*). De modo similar, o segundo termo não inclui a variável A, portanto multiplicamos este termo por $A + \bar{A}$. Obtemos

$$f(A,B,C) = A(B + \bar{B})(C + \bar{C}) + (A + \bar{A})BC \tag{2.1-12}$$

Multiplicando os termos fatorados, ou seja, usando a lei distributiva, temos

$$f(A,B,C) = ABC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C} + ABC + \bar{A}BC \tag{2.1-13}$$

Notamos que o termo ABC aparece duas vezes na Eq. (2.1-13). Como $ABC + ABC = ABC$, podemos desprezar um destes termos e obter

$$f(A,B,C) = ABC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C} + \bar{A}BC \tag{2.1-14}$$

A função $f(A, B, C)$ agora está na forma padrão de soma-de-produtos, onde "padrão" significa que *cada* uma das variáveis A, B e C aparece (às vezes complementada, às vezes não) em cada um dos termos produto. Cada um destes produtos é chamado *mintermo*. Certamente, a expressão da Eq. (2.1-14) parece bem mais complicada que a expressão original da Eq. (2.1-11), mas, como veremos, o desenvolvimento que estamos seguindo nos levará a um processo muito útil de simplificação e minimização de funções lógicas.

2.2 O PRODUTO PADRÃO DE SOMAS

Pelo princípio da dualidade podemos imaginar que uma expressão lógica também pode ser escrita na forma de um *produto padrão de somas*. Veremos a seguir que isto é verdade. Vamos considerar novamente as funções dos exemplos anteriores e verificar como podemos escrevê-las na forma de produto de somas.

Exemplo 2.2-1 Dada a função lógica de quatro variáveis

$$f(A,B,C,D) = (\bar{A} + BC)(B + \bar{C}D) \tag{2.2-1}$$

expressar a função como um produto-de-somas.

SOLUÇÃO: Anteriormente, para gerar a soma-de-produtos, usamos a lei distributiva na forma da Eq. (1.15-9*b*). Ela será usada, agora, na forma dual dada pela Eq. (1.15-9*a*), obtendo-se

$$f(A,B,C,D) = (\bar{A} + B)(\bar{A} + C)(B + \bar{C})(B + D) \tag{2.2-2}$$

Notamos que, na Eq. (2.2-2), f é expressa na forma de um produto de termos, cada termo sendo a *soma* das variáveis lógicas individuais, podendo uma variável aparecer na forma complementada ou não-complementada.

Exemplo 2.2-2 Dada a função lógica de cinco variáveis

$$f(A,B,C,D,E) = (A + \overline{BC})(\overline{D + BE}) \tag{2.2-3}$$

expressar a função como um *produto-de-somas*.

SOLUÇÃO: Como no Exemplo 2.1-2, devemos primeiramente usar o teorema de De Morgan para que o sinal de complementação apareça somente sobre variáveis individuais. Esta aplicação do teorema de De Morgan já foi feita no exemplo anterior. Podemos, portanto, começar com a Eq. (2.1-8):

$$f(A,B,C,D,E) = (A + \bar{B} + \bar{C})[\bar{D}(\bar{B} + \bar{E})] \tag{2.2-4}$$

Desejamos conseguir um resultado em que f apareça como um produto-de-somas de termos em parênteses. Usaremos a lei distributiva na forma da Eq. (1.15-9*b*) para obter

$$f(A,B,C,D,E) = (A + \bar{B} + \bar{C})(\bar{B}\bar{D} + \bar{D}\bar{E}) \tag{2.2-5}$$

Pela aplicação repetida da lei distributiva, na forma da Eq. (1.15-9*a*), obtemos

$$f(A,B,C,D,E) = (A + \bar{B} + \bar{C})(\bar{B}\bar{D} + \bar{D})(\bar{B}\bar{D} + \bar{E}) \tag{2.2-6}$$

$$= (A + \bar{B} + \bar{C})(\bar{B} + \bar{D})(\bar{D} + \bar{D})(\bar{B} + \bar{E})(\bar{D} + \bar{E}) \tag{2.2-7}$$

$$= (A + \bar{B} + \bar{C})(\bar{B} + \bar{D})(\bar{D})(\bar{B} + \bar{E})(\bar{D} + \bar{E}) \tag{2.2-8}$$

Poderíamos ter simplificado a Eq. (2.2-6), pois $\bar{B}\bar{D} + \bar{D} = \bar{D}$, mas não o fizemos porque nosso interesse presente é mostrar como se gera uma forma particular de uma expressão lógica, e não obter sua forma mais simples. Observamos que na Eq. (2.2-8) chegamos a uma forma onde f é expressa como um produto de termos, cada termo sendo uma soma de variáveis, algumas vezes complementadas, outras não. Em um caso, neste exemplo, esta "soma" constou em apenas uma variável.

Em cada soma, uma variável aparece apenas uma vez, complementada ou não-complementada. Se houvesse repetição de uma variável, esta repetição poderia ser eliminada, pois $A + A = A$ etc. Se encontrássemos dentro de parênteses uma variável e seu complemento, como, por exemplo, $A + \bar{A}$, usaríamos a propriedade $A + \bar{A} = 1$, e, como 1 somado a qualquer outra quantidade resulta em 1, os parênteses seriam substituídos por 1 e não precisariam ser escritos explicitamente.

Como no caso da soma padrão de produtos, e também no caso presente, a forma *padrão* para o produto de somas é aquela em que cada parênteses contém todas as variáveis (complementadas ou não-complementadas). O processo de geração desta forma padronizada é ilustrado pelo exemplo seguinte.

Exemplo 2.2-3 Considere a função lógica de três variáveis

$$f(A,B,C) = A(\bar{B} + C) \tag{2.2-9}$$

Esta função já se encontra na forma de um produto de somas. Reescreva a Eq. (2.2-9) de modo a tornar-se uma expressão na qual as três variáveis apareçam em cada um dos termos produto.

SOLUÇÃO: No termo soma $(\bar{B} + C)$ não aparece a variável A (ou $\bar{A}$), enquanto no termo "soma" A não aparecem as variáveis B ou C. Deste modo devemos adicionar $B\bar{B} + C\bar{C}$ à A e devemos adicionar $A\bar{A}$ à soma $(\bar{B} + C)$. Isto pode ser feito pois $A\bar{A} = B\bar{B} = C\bar{C} = 0$. Assim obtemos

$$f(A,B,C) = (A + B\bar{B} + C\bar{C})(A\bar{A} + \bar{B} + C) \tag{2.2-10}$$

Aplicando repetidamente a lei distributiva na forma da Eq. (1.15-9*a*), obtemos

$$f = (A + B\bar{B} + C)(A + B\bar{B} + \bar{C})(A + \bar{B} + C)(\bar{A} + \bar{B} + C) \tag{2.2-11}$$

$$= (A + B + C)(A + \bar{B} + C)(A + B + \bar{C})(A + \bar{B} + \bar{C})(A + \bar{B} + C)(\bar{A} + \bar{B} + C) \tag{2.2-12}$$

Observamos que o termo $(A + \bar{B} + C)$ é duplicado. Eliminando esta duplicação, o que é possível pois $XX = X$, temos

$$f = (A + B + C)(A + \bar{B} + C)(A + B + \bar{C})(A + \bar{B} + \bar{C})(\bar{A} + \bar{B} + C) \tag{2.2-13}$$

A Eq. (2.2-13) apresenta a função na forma *padrão* de produto-de-somas em que cada variável (complementada ou não) aparece em cada parênteses. Cada um desses termos soma completos é denominado um *maxtermo*.

2.3 NUMERAÇÃO DOS MINTERMOS E MAXTERMOS

Tendo introduzido organização e sistematização na expressão de funções lógicas, torna-se útil criar um método que permita a atribuição de números aos maxtermos (somas completas) e mintermos (produtos completos).

Consideremos, primeiramente, a numeração dos mintermos. Para sermos específicos, imaginemos uma função lógica envolvendo três variáveis A, B e C. Um mintermo incluirá cada variável apenas uma vez (complementada ou não). Atribui-se o *número* binário 0 a cada variável complementada e o *número* binário 1 a cada variável não-complementada. Suponhamos que o mintermo envolva A, B e $\bar{C}$; os números binários atribuídos seriam 1, 1 e 0, respectivamente. A seguir compõe-se um número binário de três dígitos a partir dos dígitos individuais. Este número é o *número do mintermo*. O valor numérico vai, obviamente, depender de qual dígito for colocado na posição numérica mais significativa e de qual dígito for colocado na posição menos significativa etc. Se decidirmos que o dígito atribuído à variável A ocupará a posição mais significativa, seguido pelo dígito atribuído a B, com o dígito atribuído a C ocupando a posição menos significativa, o número seria $110 = 6$. O mintermo seria então chamado de mintermo 6 e seria representado por m_6. Se houvéssemos decidido inverter a ordem de significância numérica, o número seria $011 = 3$ e o mintermo seria m_3.

Observamos que é necessário adotar um método consistente (embora arbitrário) para atribuir os valores numéricos. O método geralmente aceito é o de fixar inicialmente a ordem em que aparecem as variáveis e atribuir os valores numéricos na mesma ordem. Assim, se o mintermo envolvendo A, B e $\bar{C}$ for escrito $AB\bar{C}$ (e nunca $A\bar{C}B$ ou $BA\bar{C}$ etc.), então o mintermo será 110 e $AB\bar{C} = m_6$. Por outro lado, se decidíssemos escrevê-lo $\bar{C}BA$, então $\bar{C}BA = 011 = m_3$. Outras ordenações produziriam numerações diferentes para o mesmo mintermo.

Quando tratamos com maxtermos, a regra para atribuir 1 ou 0 é *invertida*: uma variável complementada recebe o dígito 1 e uma variável não-complementada recebe o dígito 0. Assim, o maxtermo $\bar{A} + B + C$ recebe o número $100 = 4$ e é representado por M_4. Os comentários feitos acima em relação à ordenação das variáveis dos mintermos aplicam-se integralmente aos maxtermos.

2.4 ESPECIFICAÇÃO DE FUNÇÕES EM TERMOS DE MINTERMOS E MAXTERMOS

Uma função lógica pode ser especificada de maneira muito conveniente usando a convenção adotada para a numeração dos mintermos e maxtermos. Consideremos, por exemplo, a função $f(A, B, C)$ da Eq. (2.1-14). Esta função estava expressa na forma padrão de soma-de-produtos, isto é, soma de mintermos. Reescrevendo a Eq. (2.1-14) e colocando os mintermos em ordem de numeração crescente, obtemos

$$f(A,B,C) = \bar{A}BC + A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C} + ABC \qquad (2.4\text{-}1)$$

011 100 101 110 111

3 4 5 6 7

Sob cada variável colocamos 0s e 1s conforme requerido, gerando a representação binária do número de cada mintermo; na terceira linha substituímos os números binários por seus equivalentes decimais. Podemos, agora, escrever

$$f(A,B,C) = m_3 + m_4 + m_5 + m_6 + m_7 \qquad (2.4\text{-}2)$$

ou, em forma mais compacta

$$f(A,B,C) = \Sigma m(3, 4, 5, 6, 7) \qquad (2.4\text{-}3)$$

ou ainda

$$f(A,B,C) = \Sigma(3, 4, 5, 6, 7) \qquad (2.4\text{-}4)$$

Consideremos a função da Eq. (2.2-11), que representa uma função lógica na forma padrão de soma-de-produtos, ou seja, de soma de maxtermos. Temos

$$f(A, B, C) = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)(A + \bar{B} + \bar{C})(\bar{A} + \bar{B} + C) \qquad (2.4\text{-}5)$$

000 001 010 011 110

0 1 2 3 6

de modo que

$$f(A,B,C) = M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_6 \qquad (2.4\text{-}6)$$

ou

$$f(A,B,C) = \Pi M(0, 1, 2, 3, 6) \qquad (2.4\text{-}7)$$

ou, finalmente

$$f(A,B,C) = \Pi(0, 1, 2, 3, 6) \qquad (2.4\text{-}8)$$

2.5 RELAÇÃO ENTRE MINTERMOS, MAXTERMOS E A TABELA VERDADE

Uma função lógica pode ser expressa em uma tabela verdade como uma soma de mintermos ou como um produto de maxtermos. As relações entre estes modos de representação são mostradas

Linha nº	A	B	C	$f(A, B, C)$
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

$f = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C} + ABC$
$= \Sigma(0, 2, 3, 6, 7)$
$= (A + B + \bar{C})(\bar{A} + B + \bar{C})$
$= \Pi(1, 4, 5)$

Fig. 2.5-1 Uma tabela verdade de três variáveis. A cada linha foi atribuído o número que resultaria se os valores lógicos fossem lidos como dígitos binários na ordem A, B, C. A função definida pela tabela é expressa como uma soma de mintermos e como um produto de maxtermos.

na Fig. 2.5-1 para uma função selecionada arbitrariamente. Cada linha da tabela verdade recebeu o número que resultaria se os valores nas colunas A, B e C fossem lidos como dígitos binários.

A primeira linha da tabela verdade especifica que $f = 1$ quando $A = 0$, $B = 0$ e $C = 0$. Se pretendemos escrever a função como uma soma de mintermos, torna-se aparente que podemos garantir que $f = 1$ se $A = 0$, $B = 0$, $C = 0$ incluindo o mintermo $\bar{A}\bar{B}\bar{C}$; ao incluir este mintermo, estamos seguros de que $f = 1$ se $A = B = C = 0$, independentemente de que outros termos possam ser adicionados posteriormente, pois 1 + qualquer termo = 1. Esta primeira linha recebe o mesmo número do mintermo que devemos incluir para satisfazer a especificação da linha. Do mesmo modo, para satisfazer a especificação contida na segunda linha da tabela verdade para que $f = 1$, devemos incluir m_2 na soma de produtos. Generalizando, uma função expressa na forma de soma de mintermos deve incluir precisamente os mintermos cujos números correspondem às linhas na tabela verdade onde $f = 1$.

Do mesmo modo, podemos estabelecer que, quando uma função é expressa na forma de produto de maxtermos, o número dos maxtermos que devem ser incluídos correspondem aos números das linhas da tabela verdade onde $f = 0$. Consideremos, por exemplo, a linha 1 da tabela verdade da Fig. 2.5-1. Para assegurar, como se deseja, que $f = 0$ quando $A = 0$, $B = 0$ e $C = 1$, devemos incluir o maxtermo $(A + B + \bar{C})$. Tendo incluído este maxtermo teremos assegurado que $f = 0$ para $A = 0$, $B = 0$, $C = 1$, independentemente de que outros maxtermos possam ser adicionados posteriormente (pois 0 multiplicado por qualquer termo = 0). Note-se que como nas colunas correspondentes a A e B na tabela verdade ocorrem 0s, é necessário usar A e B *não-complementadas* no maxtermo. A variável C aparece complementada no maxtermo porque existe um 1 na coluna correspondente à variável C.

Em uma tabela verdade, linhas que não contêm $f = 1$, contêm $f = 0$ e vice-versa. Cada $f = 1$ gera um mintermo e cada $f = 0$ gera um maxtermo. Vamos supor que exista uma função expressa por uma soma de mintermos e que desejamos seja expressa como um produto de maxtermos. Para tal basta incluirmos na lista de maxtermos precisamente aqueles maxtermos cujos números *não* aparecem na lista de mintermos. Suponhamos, por exemplo, que tenhamos uma função de três variáveis, com números de 0 a 7, onde $f = \Sigma(0, 3, 6, 7)$. A *mesma* função pode ser escrita como $f = \Pi(1, 2, 4, 5)$. Do mesmo modo, $g = \Pi(0, 2, 6)$ pode ser escrita $g = \Sigma(1, 3, 4, 5, 7)$.

Por outro lado, se desejamos escrever o *complemento* de uma função, devemos trocar cada $f = 0$ por $f = 1$ na tabela verdade. Do mesmo modo, devemos trocar cada mintermo pelo maxtermo de mesmo número e vice-versa. Assim, dada $f = \Sigma(0, 3, 6, 7)$, teremos $\bar{f} = \Pi(0, 3, 6, 7)$.

Resumindo, uma função pode ser expressa como uma soma de mintermos cujos números correspondem aos números das linhas da tabela verdade onde a função tem valor 1. Pode, também, ser expressa por um produto de maxtermos cujos números correspondem às linhas da tabela verdade onde a função tem valor 0.

2.6 ESTRUTURAS DE DOIS NÍVEIS COM PORTAS

Consideremos uma função lógica expressa na forma de soma de produtos ou de produto de somas. Como a forma não precisa ser padrão, os termos individuais não precisam ser maxtermos ou mintermos. Quando estas formas lógicas são fisicamente implementadas com portas, a estrutura resultante é uma estrutura de portas de dois níveis. Por exemplo, consideremos a função

$$f(A,B,C) = A\bar{B}C + \bar{A}\bar{C} + \bar{A}B \tag{2.6-1}$$

que é implementada através da estrutura de portas da Fig. 2.6-1. Cada produto é gerado por uma porta AND e as saídas das portas AND são somadas logicamente por uma porta OR. Generalizando, se uma função aparece como a soma de n termos produto, a estrutura física de portas teria n portas AND e uma única porta OR.

Do mesmo modo, uma função expressa como um produto de somas seria implementada como um arranjo de portas OR seguidas por uma única porta AND. Se uma função $g(A, B, C)$ fosse expressa por

$$g(A,B,C) = (\bar{A} + B + \bar{C})(\bar{A} + \bar{B})(\bar{B} + C) \tag{2.6-2}$$

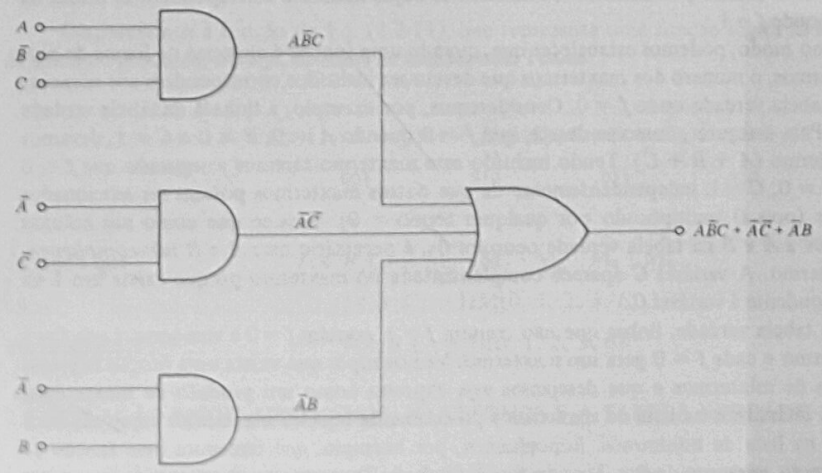


Fig. 2.6-1 Uma estrutura de portas que gera a função $A\bar{B}C + \bar{A}\bar{C} + \bar{A}B$.

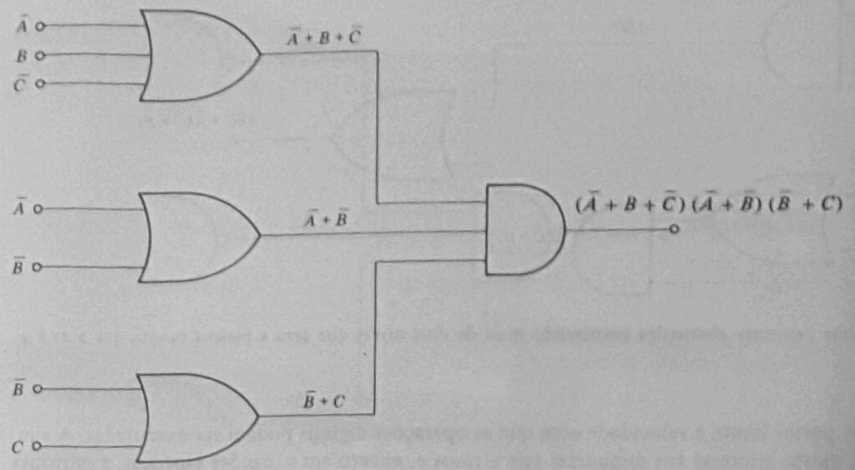


Fig. 2.6-2 Uma estrutura de portas que implementa a função $(\bar{A} + B + \bar{C})(\bar{A} + \bar{B})(\bar{B} + C)$.

a estrutura de portas correspondente seria a mostrada na Fig. 2.6-2. Se a função tiver n termos soma a estrutura terá n portas OR seguidas por uma única porta AND.

Estruturas de portas como as das Figs. 2.6-1 e 2.6-2 são chamadas estruturas de *dois níveis*, pois uma variável lógica deve passar através de duas portas antes que seu efeito seja sentido na saída. Como foi mencionado anteriormente, os valores verdade das variáveis lógicas são geralmente representados por tensões elétricas. Uma troca no valor de uma variável é representada por uma variação de tensão. Quando esta variação de tensão tem lugar na entrada de uma porta, seu efeito não se faz sentir na saída imediatamente. Existe um atraso, chamado *atraso de propagação da porta*, entre a alteração na entrada e a correspondente alteração na saída, se houver. (Nem toda a alteração na entrada causa uma alteração na saída. Se uma entrada de uma porta AND for mantida em 0, a saída não responderá a alterações no valor lógico da outra entrada.) Na montagem de portas das Figs. 2.6-1 e 2.6-2 o atraso entre a alteração das entradas e a correspondente alteração na saída corresponde ao atraso de *duas portas*, de onde decorre a caracterização da estrutura como sendo de dois níveis.

Qualquer função pode ser escrita como uma soma de produtos ou como um produto de somas e pode, conseqüentemente, ser implementada através de uma estrutura de dois níveis AND-OR (Fig. 2.6-1) ou OR-NOR (Fig. 2.6-2). Se assim o desejarmos, podemos implementar funções através de estruturas que usam mais do que dois níveis. Se reescrevermos a Eq. (2.6-1), teremos

$$f(A,B,C) = A\bar{B}C + \bar{A}\bar{C} + \bar{A}B = A\bar{B}C + \bar{A}(\bar{C} + B) \tag{2.6-3}$$

que leva diretamente à implementação da Fig. 2.6-3. Aqui temos uma estrutura de portas de três níveis, pois algumas das variáveis lógicas terão que se propagar através de três portas. Dada uma função lógica complexa, podemos, se assim o desejarmos, implementá-la de forma a envolver muitos níveis de portas. Todavia, como veremos, os atrasos de propagação encontrados em uma

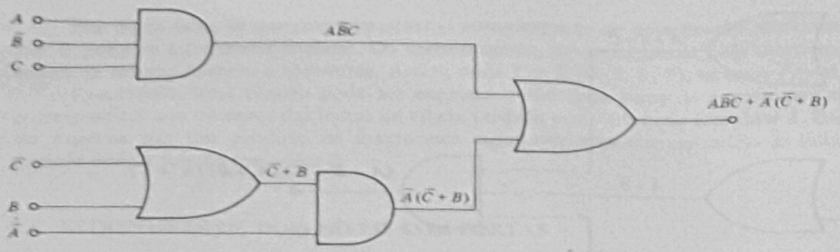


Fig. 2.6-3 Uma estrutura alternativa envolvendo mais de dois níveis que gera a mesma função que a da Fig. 2.6-1.

estrutura de portas limita a velocidade com que as operações digitais podem ser executadas. Assim, geralmente, existe interesse em minimizar tais atrasos e, exceto em situações especiais, a estrutura de portas de dois níveis é a preferida.

As estruturas de dois níveis das Figs. 2.6-1 e 2.6-2 podem gerar qualquer função arbitrária das variáveis de entrada, mas, em geral, requerem que certas variáveis estejam disponíveis tanto na forma complementada como não-complementada. Freqüentemente acontece que, quando uma variável está disponível, também o esteja seu complemento. Caso contrário é necessário usar inversores, aumentando, conseqüentemente, o atraso de propagação.

2.7 ESTRUTURAS USANDO UM TIPO DE PORTA

Como veremos no capítulo seguinte, devido à disponibilidade comercial de portas lógicas, muitas vezes é conveniente gerar uma função lógica qualquer usando somente um tipo de porta. As estruturas das Figs. 2.6-1 e 2.6-2 não satisfazem este requisito, pois elas requerem três tipos diferentes de portas: AND, OR e NOT (inversor). Existe um processo simples para converter as estruturas das Figs. 2.6-1 e 2.6-2 em estruturas que envolvam somente portas NAND ou somente portas NOR.

Consideremos a soma de produtos expressa pela Eq. (2.6-1). Vamos complementar a função duas vezes, deixando-a, portanto, inalterada. Temos

$$f(A,B,C) = \overline{\overline{A\bar{B}C + \bar{A}\bar{C} + \bar{A}B}} \tag{2.7-1}$$

Usando o teorema de De Morgan, encontramos

$$f(A,B,C) = \overline{(\overline{A\bar{B}C})(\overline{\bar{A}\bar{C}})(\overline{\bar{A}B})} \tag{2.7-2}$$

A função $\overline{A\bar{B}C}$ pode ser obtida aplicando A, $\bar{B}$ e C às entradas de uma porta NAND (o mesmo se aplicando aos termos $\overline{\bar{A}\bar{C}}$ e $\overline{\bar{A}B}$. Finalmente, a função $f(A, B, C)$ pode ser gerada aplicando os termos $\overline{A\bar{B}C}$, $\overline{\bar{A}\bar{C}}$ e $\overline{\bar{A}B}$ à outra porta NAND. A função completa pode ser gerada pela estrutura de portas NAND de dois níveis da Fig. 2.7-1. Observe-se que, de maneira formal, a conversão da estrutura AND-OR de dois níveis para NAND-NAND somente requer a substituição de *todas* as portas por portas NAND.

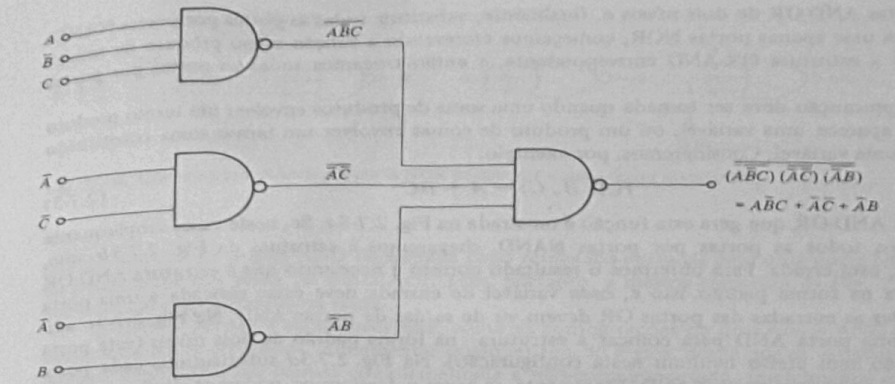


Fig. 2.7-1 A estrutura de portas NAND-NAND que gera a mesma função que a estrutura AND-OR da Fig. 2.6-1.

De modo similar, podemos estabelecer que uma estrutura OR-AND de dois níveis pode ser convertida em uma estrutura NOR-NOR de dois níveis simplesmente substituindo todas as portas por portas NOR. Como exemplo, considere-se a estrutura OR-NOR da Fig. 2.6-2. Esta estrutura é mostrada após a conversão para uma estrutura NOR-NOR na Fig. 2.7-2.

Se desejarmos gerar uma função usando somente portas NAND, devemos inicialmente exprimir a função como soma de produtos. O próximo passo é desenhar a correspondente estru-

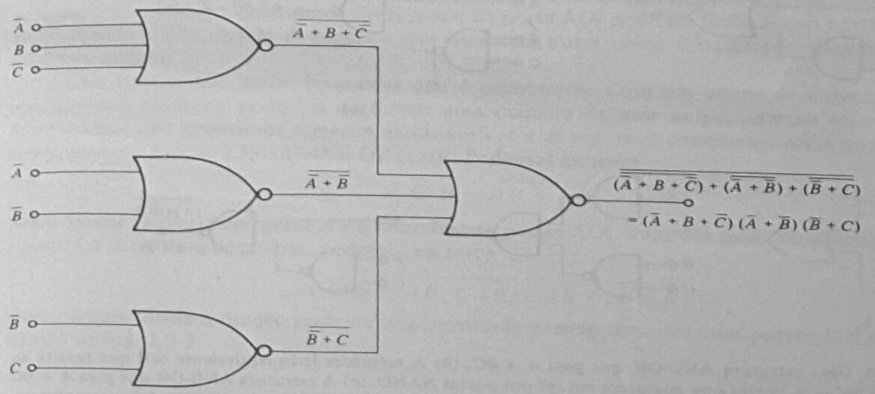


Fig. 2.7-2 A estrutura OR-NOR da Fig. 2.6-2 convertida em uma estrutura que só usa portas NOR.

tura de portas AND-OR de dois níveis e, finalmente, substituir todas as portas por portas NAND. Se queremos usar apenas portas NOR, começamos escrevendo a função como produto de somas, desenhando a estrutura OR-AND correspondente, e então trocamos todas as portas por portas NOR.

Uma precaução deve ser tomada quando uma soma de produtos envolver um termo produto em que só aparece uma variável, ou um produto de somas envolver um termo soma constituído por apenas uma variável. Consideremos, por exemplo:

$$f(A, B, C) = A + BC \tag{2.7-3}$$

A estrutura AND-OR que gera esta função é mostrada na Fig. 2.7-3*a*. Se, neste caso, simplesmente substituirmos todos as portas por portas NAND, chegaremos à estrutura da Fig. 2.7-3*b*, que, obviamente, está errada. Para obtermos o resultado correto é necessário que a estrutura AND-OR inicial esteja na forma padrão, isto é, *cada* variável de entrada deve estar aplicada a uma porta AND e *todas* as entradas das portas OR devem vir de saídas de portas AND. Na Fig. 2.7-3*c* adicionamos uma porta AND para colocar a estrutura na forma padrão de dois níveis (esta porta adicional não tem efeito nenhum nesta configuração). Na Fig. 2.7-3*d* substituímos cada porta da Fig. 2.7-3*c* por uma porta NAND. Finalmente, se assim o desejarmos, podemos eliminar a porta NAND de uma entrada e compensarmos esta remoção substituindo A por $\bar{A}$, como na Fig. 2.7-3*e*.

Se for necessário o complemento de uma variável de entrada, podemos usar uma porta NAND ou NOR para esta finalidade. Pode-se verificar facilmente que uma porta NAND ou NOR usada da maneira indicada na Fig. 2.7-4 efetua uma inversão lógica.

Portas AND-OR-INVERT (AOI)

Outro tipo de porta (na realidade uma estrutura de portas) disponível comercialmente, que pode ser usada sozinha para gerar qualquer função lógica, é a porta AND-OR-INVERT (AOI). Consiste

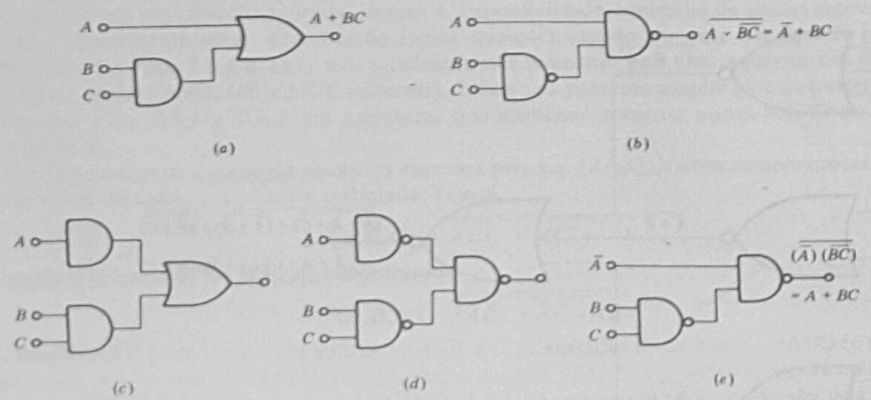


Fig. 2.7-3 (*a*) Uma estrutura AND-OR que gera $A + BC$. (*b*) A estrutura [não equivalente (*a*)] que resulta ao substituirmos todas as portas que aparecem em (*a*) por portas NAND. (*c*) A estrutura AND-OR que gera $A + BC$ com dois níveis para todas as variáveis de entrada. (*d*) A estrutura NAND-NAND que substitui (*a*) corretamente. (*e*) A porta NAND de uma entrada removida com a variável A substituída por $\bar{A}$.

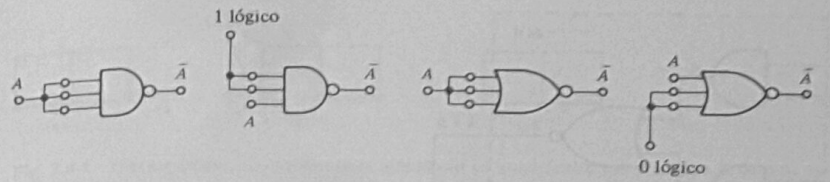


Fig. 2.7-4 Como a porta NAND e a porta NOR podem ser usadas como inversores.

em um número de portas AND de entrada cujas saídas são entradas para uma porta NOR. A saída da porta NOR é a saída da porta AOI. Tem, portanto, a mesma estrutura da Fig. 2.6-1, exceto por uma inversão adicional na saída.

Podemos primeiro estabelecer que, como as configurações de dois níveis AND-OR ou OR-AND, uma porta AOI pode implementar qualquer função lógica, desde que as variáveis de entrada estejam disponíveis nas formas complementada e não-complementada. Inicialmente escreveremos a função desejada como um produto de somas. Assim, vamos supor que a função seja

$$f = (A + \bar{B} + C)(D + \bar{E})(\bar{F} + G) \tag{2.7-4}$$

Usando o teorema de De Morgan, podemos escrever

$$f = \bar{\bar{f}} = \overline{\overline{(A + \bar{B} + C)(D + \bar{E})(\bar{F} + G)}} \tag{2.7-5}$$

$$= \overline{\overline{(A + B + \bar{C})} + \overline{(\bar{D} + E)} + \overline{(F + \bar{G})}} \tag{2.7-6}$$

$$= \overline{\bar{A}B\bar{C} + \bar{D}E + F\bar{G}} \tag{2.7-7}$$

que está exatamente na forma AND-OR-INVERT. Notamos ainda que devido à inversão adicional existente na porta AOI, esta pode ser usada para obter os complementos das variáveis de entrada se assim for desejado. Verificamos ainda que uma porta AOI pode ser usada como porta NAND ou NOR (Prob. 2.7-3), donde deduzimos que nenhuma outra porta é necessária para implementar qualquer função.

Usar portas AOI como inversores não é econômico. Com um pouco de engenhosidade na manipulação algébrica podemos encontrar uma maneira de usar as portas mais eficientemente. Suponhamos que possuímos somente as variáveis A e B sem seus complementos e que desejamos implementar a função EXCLUSIVE-OR $A \oplus B$. Podemos escrever

$$A \oplus B = \overline{AB + \bar{A}\bar{B}} \tag{2.7-8}$$

Nesta forma poderíamos gerar $A \oplus B$ usando três portas AOI, duas das quais simplesmente produziriam $\bar{A}$ e $\bar{B}$. Alternativamente, podemos escrever

$$A \oplus B = \overline{AB + \bar{A}\bar{B}} = \overline{AB + \overline{A + B}} \tag{2.7-9}$$

Nesta última forma a função pode ser implementada usando somente duas portas AOI, como mostramos na Fig. 2.7-5.

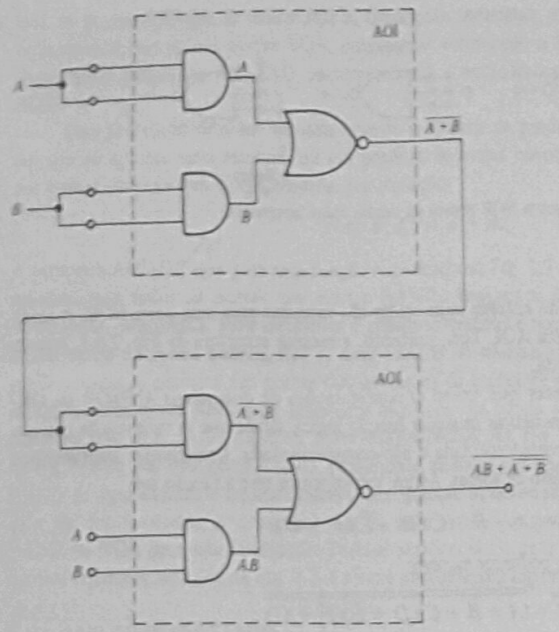


Fig. 2.7-5 Duas portas AND-OR-INVERT usadas para implementar a função EXCLUSIVE-OR.

2.8 MAPAS DE KARNAUGH

O mapa de Karnaugh é um dispositivo extremamente útil na simplificação e minimização das funções algébricas booleanas. Nesta seção e nas seguintes discutiremos os mapas K e resolveremos exemplos com estes mapas para encorajar o desenvolvimento da familiaridade do leitor com seu uso.

Um mapa K é uma figura geométrica que contém uma região (quadrículo) para cada linha de uma tabela verdade. Já observamos que existe uma correspondência biunívoca entre as linhas da tabela verdade e os maxtermos e mintermos em potencial. Observamos, adicionalmente, que existe uma correspondência entre os quadrículos do mapa K e os mintermos, bem como entre os quadrículos e os maxtermos. Inicialmente, consideraremos a construção dos mapas e a identificação dos quadrículos com as linhas da tabela verdade, com os maxtermos e com os mintermos, deixando para mais tarde a explicação sobre o uso dos mapas.

O mapa K para uma variável, no caso a variável A, é mostrado na Fig. 2.8-1. Consiste simplesmente em dois quadrículos adjacentes, correspondendo ao fato de que uma tabela verdade de uma variável contém apenas duas linhas. As Figs. 2.8-1*a*, *b* e *c* mostram três maneiras como os quadrículos podem ser identificados com as linhas da tabela verdade. Na Fig. 2.8-1*a* os quadrículos são numerados no canto superior esquerdo. O quadrículo esquerdo corresponde à linha 0 e o quadrículo direito corresponde à linha 1. Na Fig. 2.8-1*b* a mesma informação é mostrada pela correspondência do quadrículo esquerdo com a linha da tabela verdade em que $A = 0$ e do quadrículo direito com a linha em que $A = 1$. Na Fig. 2.8-1*c* o entendimento é que o quadrículo sob a chave denominada A é o que corresponde à linha da tabela verdade onde $A = 1$. Na Fig. 2.8-2 desenhamos o mapa de uma variável ao lado da tabela verdade para mostrar a correspondência. Na tabela verdade a função $f(A)$ foi deixada em branco para chamar atenção para o fato de que a forma do mapa K depende somente do número de variáveis incluídas e não da expressão booleana para a qual o mapa será usado.

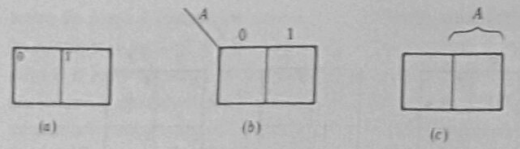
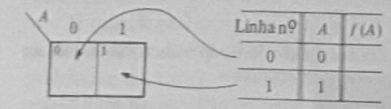


Fig. 2.8-1 Três esquemas alternativos para identificar os quadrículos em um mapa K de uma variável.

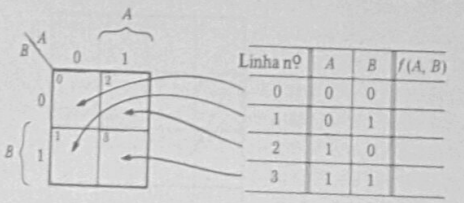


Linha nº	A	$f(A)$
0	0	
1	1	

Fig. 2.8-2 A correspondência entre uma tabela verdade de uma variável e um mapa K de uma variável.

Uma tabela verdade de duas variáveis e um mapa K para duas variáveis são mostrados na Fig. 2.8-3. Observe-se novamente os três métodos alternativos indicados para identificar os quadrículos. O quadrículo 2 não só é numerado como 2 mas também é colocado na interseção de $A = 1$ e $B = 0$, que são os valores da linha 2 nas colunas correspondentes a A e B, respectivamente. O quadrículo 2 também se encontra na região da chave A e fora da região da chave B. Um mapa K alternativo para duas variáveis bem como mapas K para um número maior de variáveis serão considerados mais tarde. No momento usaremos o mapa K da Fig. 2.8-3 para substituir uma tabela verdade.

Uma função determinada é definida pela tabela verdade da Fig. 2.8-4*a*. Esta mesma função é representada no mapa K da Fig. 2.8-4*b*. Simplesmente preenchemos os quadrículos com os 1s e 0s, conforme o caso, das linhas correspondentes da tabela verdade. Como sabemos que, se um valor não for 1, será 0, e vice-versa, podemos simplesmente registrar os 1s como na Fig. 2.8-4*c*



Linha nº	A	B	$f(A, B)$
0	0	0	
1	0	1	
2	1	0	
3	1	1	

Fig. 2.8-3 Uma tabela verdade de duas variáveis e seu mapa K.

Linha nº	A	B	$f(A, B)$
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	1

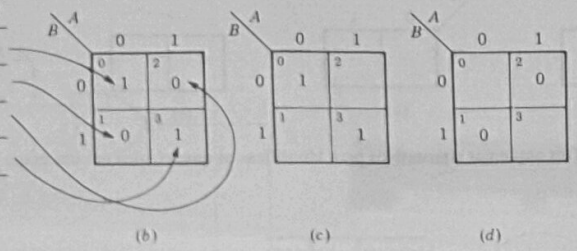


Fig. 2.8-4 (*a*) Uma tabela verdade definindo uma função. (*b*) A definição da tabela verdade representada em um mapa K. (*c*) Um mapa com os 1s registrados. (*d*) Um mapa com os 0s registrados.

ou registrar os 0s como na Fig. 2.8-4*d*. (De modo geral, em uma tabela verdade registramos *tanto* os 1s como os 0s e em um mapa K registramos *ou* os 1s *ou* os 0s. Como veremos a seguir, na leitura do mapa K concentrar-nos-emos nos 1s ou nos 0s.

Podemos verificar facilmente que a função definida na Fig. 2.8-4 é

$$f(A,B) = A\bar{B} + AB = m_0 + m_3 \tag{2.8-1}$$

$$= (A + \bar{B})(\bar{A} + B) = M_1 \cdot M_2 \tag{2.8-2}$$

Fica assim confirmado que uma função expressa por uma soma dos mintermos m_0 e m_3 é representada por 1s nos quadrículos 0 e 3 do mapa K, isto é, o quadrículo 0 é associado com m_0 e o quadrículo 3 é associado com m_3. Como a função pode ser expressa pelo produto dos maxtermos 1 e 2, encontramos 0s nos quadrículos 1 e 2. Assim, mesmo que não tivéssemos a tabela verdade explicitamente, poderíamos representar a função no mapa K através dos maxtermos ou mintermos.

Um ponto deve ser especialmente notado em relação ao registro de maxtermos e mintermos no mapa K. O quadrículo em que algum mintermo, por exemplo, m_i, deve ser registrado é o *mesmo* em que se registra o maxtermo M_i. Assim, se uma função f tem um maxtermo $f = \bar{A} + B = M_2$, então um 0 é registrado no quadrículo 2. Se uma (outra) função g possui um mintermo $A\bar{B} = m_2$, então um 1 deve ser registrado no *mesmo* quadrículo.

Um mapa K alternativo para duas variáveis é mostrado na Fig. 2.8-5. Observe-se especialmente a ordenação dos quadrículos: os números são ordenados 0, 1, 3, 2 em vez da ordem natural 0, 1, 2, 3. Como veremos, este tipo de ordenação aparece também em mapas para um número maior de variáveis. O propósito desta ordenação será discutido e explicado a seguir. A numeração

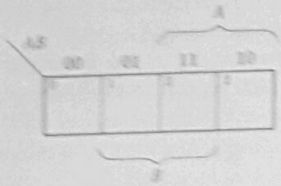


Fig. 2.8-5 Um mapa alternativo para duas variáveis.

acima do mapa é consistente com a numeração dos quadrículos. Em cada caso o dígito esquerdo se relaciona à variável A e o dígito direito se relaciona à variável B. Observe-se que, ao percorrermos o topo do mapa de um quadrículo para o seguinte, os números de dois dígitos exibem mudança em apenas um dígito de cada vez, nunca nos dois ao mesmo tempo. Veremos que esta é uma característica essencial da ordenação proposta. Finalmente, observe-se que no terceiro método de identificação dos quadrículos a chave A inclui os quadrículos correspondentes a $A = 1$ e a chave B inclui os quadrículos correspondentes a $B = 1$.

O mapa K para três variáveis é mostrado na Fig. 2.8-6. Novamente seguimos o esquema de numeração das colunas adotado no mapa alternativo para duas variáveis da Fig. 2.8-5.

O mapa K para quatro variáveis é mostrado na Fig. 2.8-7. Seguimos o esquema de numeração usado no mapa da Fig. 2.8-5 tanto para as linhas como para as colunas. Horizontalmente, os quadrículos da terceira coluna são numerados por último, ao passo que, verticalmente, os da terceira linha são os últimos a serem numerados.

É possível desenhar mapas de Karnaugh para números maiores de variáveis; um mapa para cinco variáveis teria $2^5 = 32$ quadrículos, enquanto um mapa para seis variáveis teria $2^6 = 64$ quadrículos. Por enquanto, nos concentraremos no uso dos mapas na simplificação de funções lógicas, deixando os mapas de cinco e seis variáveis para a Seção 2.12.

Note-se que existe certa medida de arbitrariedade na atribuição das variáveis às linhas e colunas de um mapa e também na atribuição de valores numéricos às variáveis. Na Fig. 2.8-7 o dígito atribuído à variável A foi tomado como sendo o mais significativo numericamente, o atribuído à variável B como o segundo mais significativo, e assim por diante. Assim, ao mintermo $A\bar{B}C\bar{D}$ foi atribuído o número 1000 = 8, e o mintermo passou a ser m_8. Esta atribuição poderia ser diferente. Se, por exemplo, invertêssemos-a ordem de significância numérica, o mapa K apare-

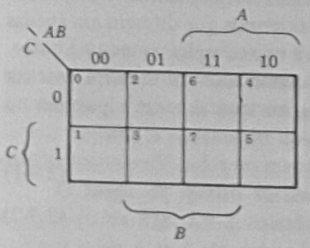


Fig. 2.8-6 O mapa K para três variáveis.

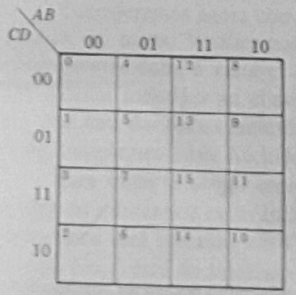


Fig. 2.8-7 O mapa K para quatro variáveis.

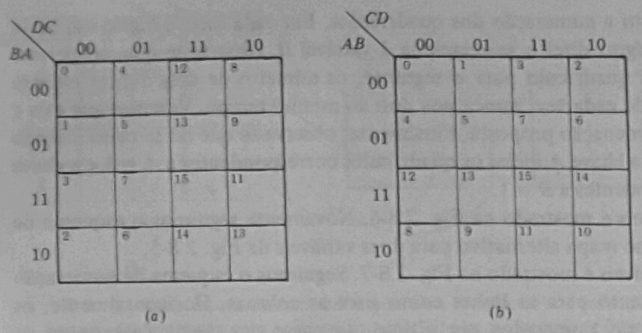


Fig. 2.8-8 Mostrando que atribuições alternativos de significância numérica e de associação de variáveis com linhas e colunas são permitidas.

ceria como na Fig. 2.8-8*a*. Poderíamos, ainda, manter as significâncias numéricas da Fig. 2.8-7 mas associando as variáveis A e B às linhas e as variáveis C e D às colunas. Neste caso o mapa K apareceria como na Fig. 2.8-8*b*. Podemos fazer qualquer atribuição de significância numérica e qualquer associação de variáveis com linhas e colunas, desde que, feitas a atribuição e a associação, elas sejam observadas consistentemente.

2.9 SIMPLIFICAÇÃO DE FUNÇÕES LÓGICAS COM MAPAS DE KARNAUGH

A característica essencial dos mapas K é que quadrículos adjacentes horizontalmente e verticalmente (mas não diagonalmente) correspondem a mintermos ou maxtermos que diferem em apenas uma variável. Esta variável aparece complementada em um termo e não-complementada no outro. É exatamente por isso que os quadrículos foram ordenados e numerados da maneira descrita previamente. Consideremos, por exemplo, os mintermos m_8 e m_{12}, horizontalmente adjacentes no mapa K da Fig. 2.9-1. Temos

$$m_8\,(8 = 1000) = A\bar{B}\bar{C}\bar{D} \tag{2.9-1}$$

$$m_{12}\,(12 = 1100) = AB\bar{C}\bar{D} \tag{2.9-2}$$

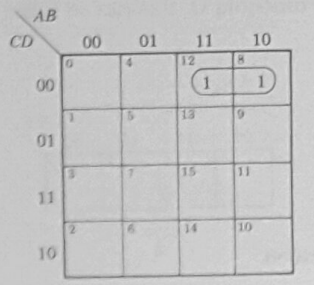


Fig. 2.9-1 Mintermos em quadrículos adjacentes podem ser combinados.

Estes dois mintermos diferem apenas pela variável B aparecer complementada em um e não-complementada no outro. Esses mintermos podem ser combinados para obter

$$A\bar{B}\bar{C}\bar{D} + AB\bar{C}\bar{D} = A\bar{C}\bar{D}(\bar{B} + B) = A\bar{C}\bar{D} \tag{2.9-3}$$

Assim, dois termos, cada um envolvendo quatro variáveis, podem ser substituídos por um termo envolvendo três variáveis. A variável que aparecia complementada em um termo e não-complementada no outro foi eliminada. Se os termos da Eq. (2.9-3) aparecessem junto com outros termos em uma função lógica, poderíamos, através de comparações sucessivas, determinar a possibilidade de sua combinação. Se a presença destes dois mintermos na função fosse anotada através da colocação de 1s nos quadrículos apropriados de um mapa K, como é mostrado na Fig. 2.9-1, a possibilidade de combinação seria determinada imediatamente, pois os mintermos correspondem a *quadrículos adjacentes*. O mérito do mapa K é possibilitar a determinação das combinações de termos através da visualização geométrica. No mapa K da Fig. 2.9-1 a combinação dos mintermos m_8 e m_{12} é indicada pela linha que enlaça os 1s colocados nos quadrículos adjacentes.

Um princípio geral que se aplica ao mapa K é que *qualquer par de mintermos adjacentes pode ser combinado em um único termo que inclui uma variável a menos que as incluídas pelos mintermos*. O termo combinado é determinado através dos dois mintermos da eliminação da variável que aparece complementada em um mintermo e não-complementada no outro. Apliquemos esta regra aos mintermos m_8 e m_{12} que aparecem na Fig. 2.9-1 Notamos que em ambos os mintermos as variáveis A, C e D são associadas (através da numeração existente acima e ao lado do mapa) com os *mesmos* dígitos (A associada com 1 e C e D com 0). Por outro lado, a variável B é associada com 1 no mintermo 12 e com 0 no mintermo 8, sendo, conseqüentemente eliminada. Os dois mintermos são combinados em um único termo em que a variável A aparece não-complementada (pois em *ambos* os mintermos ela está associada com 1) e as variáveis C e D aparecem complementadas (pois em *ambos* os mintermos estão associadas com 0). Assim

$$m_8 + m_{12} = A\bar{C}\bar{D} \tag{2.9-4}$$

Observe-se que a leitura do mapa K da Fig. 2.9-1 foi feita usando a numeração acima e ao lado do mapa; não usamos explicitamente a numeração individual dos mintermos existente nos quadrículos. Quando uma função lógica é representada através de seus mintermos ou de sua tabela verdade, a numeração dos mintermos nos quadrículos é útil, pois facilita a localização dos mintermos especificados no mapa.

O mapa de quatro variáveis com os mintermos m_8 e m_{12} indicados como na Fig. 2.9-1 é repetido na Fig. 2.9-2, usando o método alternativo de identificação dos quadrículos. As duas colunas onde $A = 1$ são enlaçadas por uma chave e marcadas A. Analogamente, colocamos chaves e identificamos as colunas em que $B = 1$ e os pares de linhas que correspondem a $C = 1$ e $D = 1$.

Consideremos agora como a representação da Fig. 2.9-2 seria usada para identificar m_8 que aparece no mapa. Verificamos que m_8 se encontra em *uma* das colunas incluídas na chave A; conseqüentemente, a variável A aparece *não-complementada*. Notamos, ainda, que m_8 *está fora* das colunas incluídas na chave B; a variável B aparecerá, pois, *complementada*. Analogamente, m_8 *está fora* das linhas incluídas pelas chaves C e D; conseqüentemente, as variáveis C e D aparecerão complementadas. Ao todo, $m_8 = A\bar{B}\bar{C}\bar{D}$. De modo similar obtemos $m_{12} = AB\bar{C}\bar{D}$.

Para obter o termo correspondente ao *par* m_8 e m_{12}, procederíamos da seguinte maneira: *ambos* os mintermos estão incluídos em A, portanto A aparece *não-complementada*. *Nenhum* dos mintermos está incluído em C ou D, portanto C e D aparecem *complementadas*. Um dos mintermos (m_{12}) está *no* intervalo correspondente a B, ao passo que o outro está *fora* deste intervalo, eliminando-se, portanto, a variável B. Assim, $m_8 + m_{12} = A\bar{C}\bar{D}$.

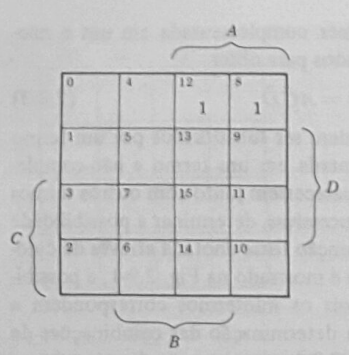


Fig. 2.9-2 Um método alternativo de identificação dos quadrículos em um mapa K.

2.10 ADJACÊNCIAS LÓGICAS ADICIONAIS

Observamos que mintermos que são *geometricamente* adjacentes no mapa K são também *logicamente* adjacentes, isto é, diferem apenas em uma variável. Há casos em que os quadrículos não são geometricamente adjacentes mas os mintermos são. Pode-se facilmente verificar que os quadrículos na coluna mais à esquerda são logicamente adjacentes aos quadrículos situados na mesma linha e na coluna mais à direita. Nas Figs. 2.9-1 e 2.9-2 m_0 é adjacente a m_8, m_1 é adjacente a m_9 etc. Similarmente, os quadrículos na linha superior são adjacentes aos situados na linha inferior e na mesma coluna, de modo que m_0 é adjacente a m_2, m_4 é adjacente a m_6 etc. Podemos

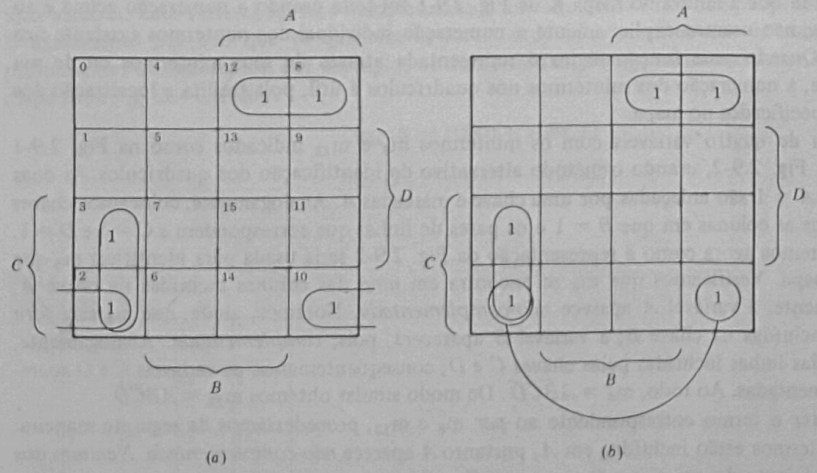


Fig. 2.10-1 Simbolismos alternativos que indicam quadrículos do mapa que não são geometricamente adjacentes.

visualizar uma adjacência geométrica entre as colunas da esquerda e da direita imaginando o mapa enrolado sobre um cilindro vertical. Do mesmo modo, podemos visualizar uma adjacência geométrica entre as linhas superior e inferior imaginando o mapa enrolado sobre um cilindro horizontal. Todas estas adjacências podem ser visualizadas ao mesmo tempo se imaginarmos o mapa enrolado sobre um toróide.

Consideremos um mapa K preenchido como o da Fig. 2.10-1. Conforme indicado pelos círculos, podemos combinar os pares de quadrículos geometricamente adjacentes com o resultado

$$m_8 + m_{12} = A\bar{C}\bar{D} \tag{2.10-1}$$

e

$$m_2 + m_3 = \bar{A}\bar{B}C \tag{2.10-2}$$

A seguir podemos combinar m_{10} com m_8 ou com m_2. Dois simbolismos alternativos para indicar a combinação de m_{10} com m_2 são mostrados nas Figs. 2.10-1*a* e *b*. Usando a combinação de m_{10} com m_2, obtemos

$$m_2 + m_{10} = \bar{B}C\bar{D} \tag{2.10-3}$$

Neste caso a função lógica definida pelo mapa K, de acordo com as Eqs. (2.10-1) a (2.10-3), é

$$f(A,B,C,D) = \Sigma m(2,3,8,10,12) = A\bar{C}D + \bar{A}\bar{B}C + \bar{B}C\bar{D} \tag{2.10-4}$$

como mostrada na Fig. 2.10-2. Se assim o desejarmos podemos combinar o mintermo m_{10} com m_8, em vez de com m_2, obtendo, então,

$$m_8 + m_{10} = A\bar{B}\bar{D} \tag{2.10-5}$$

e, das Eqs. (2.10-1), (2.10-2) e (2.10-5), teremos

$$f(A,B,C,D) = \Sigma m(2,3,8,10,12) = A\bar{C}D + \bar{A}\bar{B}C + A\bar{B}\bar{D} \tag{2.10-6}$$

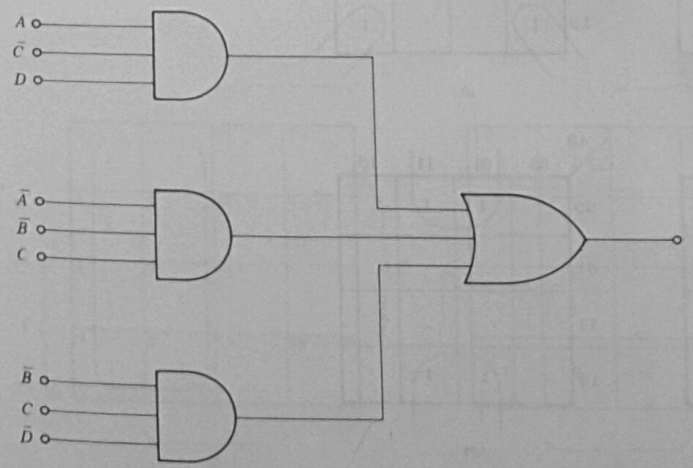


Fig. 2.10-2 Implementação da função da Eq. (2.10-4).

Embora a Eq. (2.10-6) pareça ser diferente da Eq. (2.10-4), elas são idênticas. Se derivarmos tabelas verdade para $f(A, B, C, D)$ usando primeiro uma equação e depois a outra, as tabelas serão idênticas. Do ponto de vista de economia de portas lógicas, qualquer um dos resultados é igualmente aceitável. Nos dois casos são usadas uma porta OR com três entradas e três portas AND com três entradas cada uma.

Para chegar às Eqs. (2.10-4) e (2.10-6), usamos um mesmo mintermo duas vezes. No primeiro caso combinamos m_2 com m_3 e novamente m_2 com m_{10}. No segundo caso combinamos m_8 duas vezes (com m_{10} e com m_{12}). Esse uso repetitivo de um mesmo mintermo é possível, pois, ao usarmos m_2 duas vezes, simplesmente tiramos partido do teorema da Eq. (1.15-5*a*), que, neste caso, significa

$$m_2 = \bar{A}\bar{B}C\bar{D} = \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}C\bar{D} + \cdots \tag{2.10-7}$$

2.11 AGRUPAMENTOS MAIORES EM UM MAPA K

Vimos que dois quadrículos adjacentes em um mapa K podem ser combinados, resultando um termo do qual uma variável é eliminada. De modo similar, quando 2^n quadrículos são adjacentes,

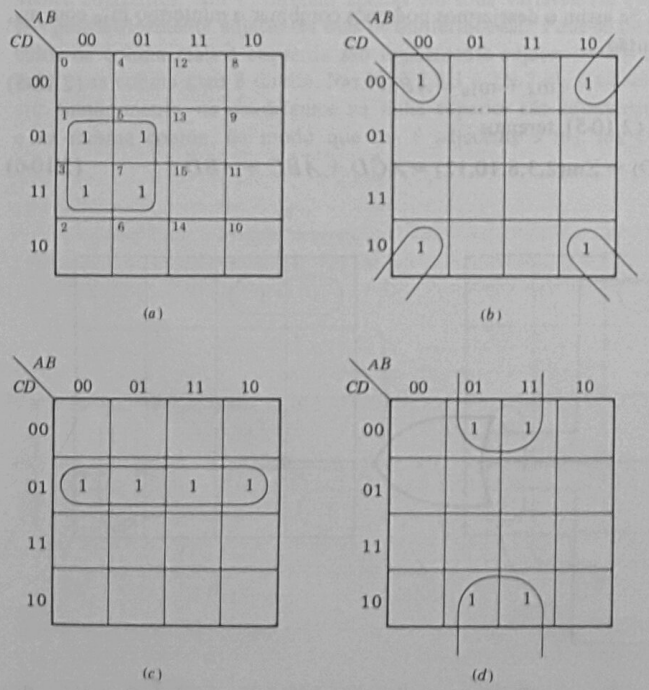


Fig. 2.11-1 Adjacências representativas de quatro quadrículos.

eles podem ser combinados resultando um termo do qual n variáveis são eliminadas. Grupos típicos de quatro quadrículos são indicados na Fig. 2.11-1. Na Fig. 2.11-1*a* as combinações $m_1 + m_5$ e $m_3 + m_7$ produzem

$$m_1 + m_5 = \bar{A}\bar{C}D \tag{2.11-1}$$

$$m_3 + m_7 = \bar{A}CD \tag{2.11-2}$$

de modo que

$$(m_1 + m_5) + (m_3 + m_7) = \bar{A}\bar{C}D + \bar{A}CD = \bar{A}D(\bar{C} + C) = \bar{A}D \tag{2.11-3}$$

O resultado seria o mesmo se o agrupamento tivesse sido feito na ordem $(m_1 + m_3) + (m_5 + m_7)$.

A determinação do grupo $m_1 + m_3 + m_5 + m_7$ poderia ser feita diretamente da seguinte maneira: os quatro mintermos estão em quadrículos pertencentes a colunas onde $A = 0$; portanto, esta variável deve permanecer na forma *complementada*. Em uma coluna, $B = 1$, e na outra, $B = 0$; portanto, a variável B deve ser eliminada. De modo similar, determinamos que a variável C deve ser eliminada e que a variável D deve permanecer na forma não-complementada, pois em ambas as linhas $D = 1$.

Lendo os demais mapas da Fig. 2.11-1, obtemos da Fig. 2.11-1*b*

$$f(A,B,C,D) = \Sigma m(1,5,9,13) = \bar{C}D \tag{2.11-4}$$

pois os quatro 1s estão em uma linha que corresponde a $C = 0$ e $D = 1$, mas estão em colunas que correspondem a $A = 0$ e $A = 1$, bem como a $B = 0$ e $B = 1$. Da Fig. 2.11-1*c*, como os quatro cantos são adjacentes, obtemos

$$f(A,B,C,D) = \Sigma m(0,2,8,10) = \bar{B}\bar{D} \tag{2.11-5}$$

Finalmente, da Fig. 2.11-1*d*, obtemos

$$f(A,B,C,D) = \Sigma m(4,6,12,14) = B\bar{D} \tag{2.11-6}$$

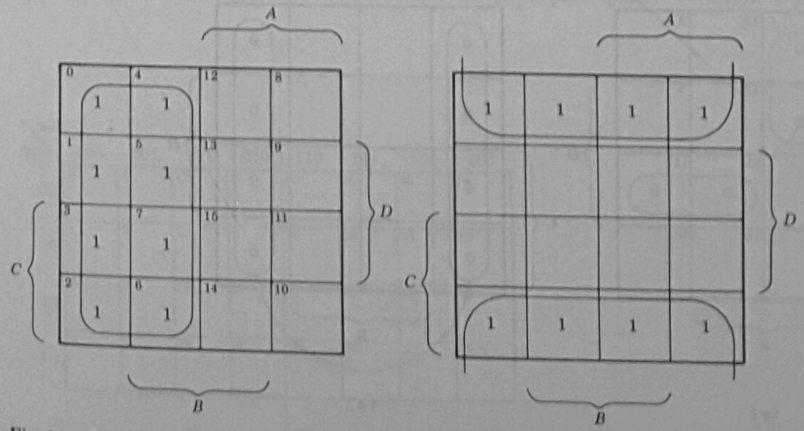


Fig. 2.11-2 Adjacências representativas de oito quadrículos.

Grupos típicos de oito quadrículos são mostrados na Fig. 2.11-2. Da Fig. 2.11-2*a* obtemos $f = \bar{A}$, pois os oito 1s estão fora do intervalo da variável A mas estão dentro e fora dos intervalos das demais variáveis. Da Fig. 2.11-2*b* obtemos $f = \bar{D}$. No caso de quatro variáveis, dezesseis 1s no mapa K indicariam que $f = 1$, independentemente de qualquer variável. Tais agrupamentos de dezesseis 1s são significativos no caso de mapas K de cinco variáveis (Seção 2.12) que possuem 32 quadrículos etc.

Consideremos agora alguns casos em que os valores no mapa K são 0s, representando maxtermos, em vez de 1s, representando mintermos. A regra para agrupar 0s é a mesma usada para agrupar 1s. A regra que determina se uma determinada variável é eliminada ou não também permanece a mesma, mas quando lemos um grupo de 0s o resultado é uma *soma* e não um *produto*; além disso, a regra que determina se uma variável aparece complementada ou não é invertida. Na Fig. 2.11-3*a* o grupo de dois 0s leva a

$$M_{11} \cdot M_{15} = \bar{A} + \bar{C} + \bar{D} \tag{2.11-7}$$

e o grupo de quatro 0s leva a

$$M_0 \cdot M_1 \cdot M_4 \cdot M_5 = A + C \tag{2.11-8}$$

Na Fig. 2.11-3*b* o grupo de oito 0s é lido como

$$M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_8 \cdot M_9 \cdot M_{10} \cdot M_{11} = B \tag{2.11-9}$$

Finalmente, os seguintes pontos devem ser observados:

1. O número de quadrículos que deve ser considerado como grupo deve ser uma potência de 2. Isto quer dizer que podemos agrupar $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$ etc., mas não podemos agrupar três quadrículos, por exemplo, mesmo que sejam todos adjacentes.

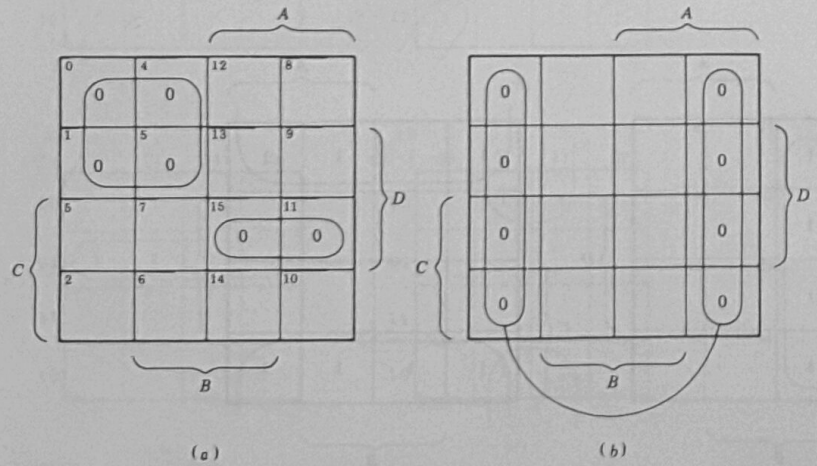


Fig. 2.11-3 Adjacências representativas usando 0s.

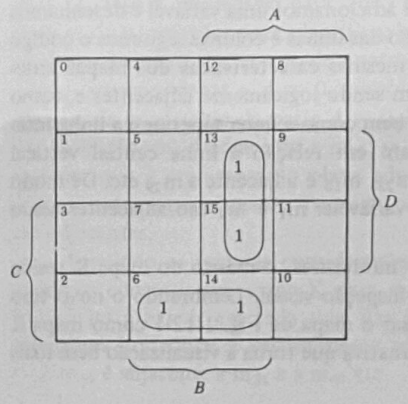


Fig. 2.11-4 Pares de mintermos que são fisicamente adjacentes mas não podem ser combinados.

2. Suponhamos a situação da Fig. 2.11-4, onde combinamos $m_6 + m_{14}$ e $m_{13} + m_{15}$. Poderíamos combinar estes dois grupos de dois em um grupo de quatro? A resposta é não, embora os dois grupos pareçam ser adjacentes; um grupo foi formado por uma combinação horizontal, ao passo que o outro foi formado por uma combinação vertical. Assim, a variável (A) eliminada de $m_6 + m_{14}$ ($= BCD$) é diferente da variável (C) eliminada de $m_{13} + m_{15}$ ($= ABD$) e não é possível efetuar combinações adicionais.

2.12 MAPAS DE KARNAUGH PARA CINCO E SEIS VARIÁVEIS

Suponhamos que para desenhar um mapa K para cinco variáveis seguíssemos o mesmo processo que nos levou do mapa para uma variável ao mapa para quatro variáveis. Obteríamos, então,

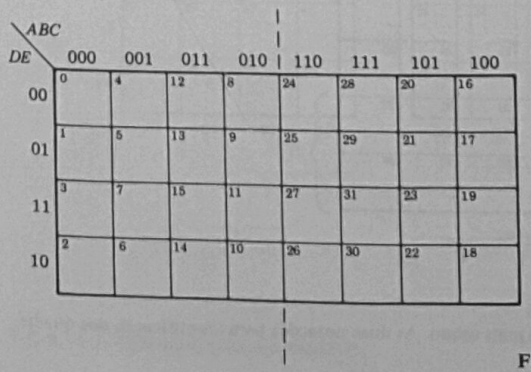


Fig. 2.12-1 Um possível mapa K para cinco variáveis.

um mapa para cinco variáveis como o da Fig. 2.12-1, onde adicionamos uma variável e desenhamos dois mapas para quatro variáveis lado a lado. Na numeração das linhas e colunas seguimos o código binário refletido (Seção 1.28). Este mapa mantém as mesmas características dos mapas anteriores: quadrículos geometricamente adjacentes continuam sendo logicamente adjacentes e, como antes, as colunas da direita e da esquerda são adjacentes, bem como a linha superior e a linha inferior. Agora, porém, quadrículos situados simetricamente em relação à linha central vertical também são adjacentes. Por exemplo, m_7 é adjacente a m_{23}, m_{13} é adjacente a m_{29} etc. De modo similar, persistem as adjacências do mapa K de quatro variáveis: m_1 e m_9 são adjacentes assim como m_2 e m_{10} etc.

Como necessitamos tirar partido da adjacência de mintermos, o mérito do mapa K reside em tornar evidentes tais adjacências através de simples inspeção visual. Lembrando o novo tipo de adjacência introduzida, seria perfeitamente possível usar o mapa da Fig. 1.12-1 como mapa K para cinco variáveis. Existe, todavia, uma distribuição alternativa que torna a visualização bem mais

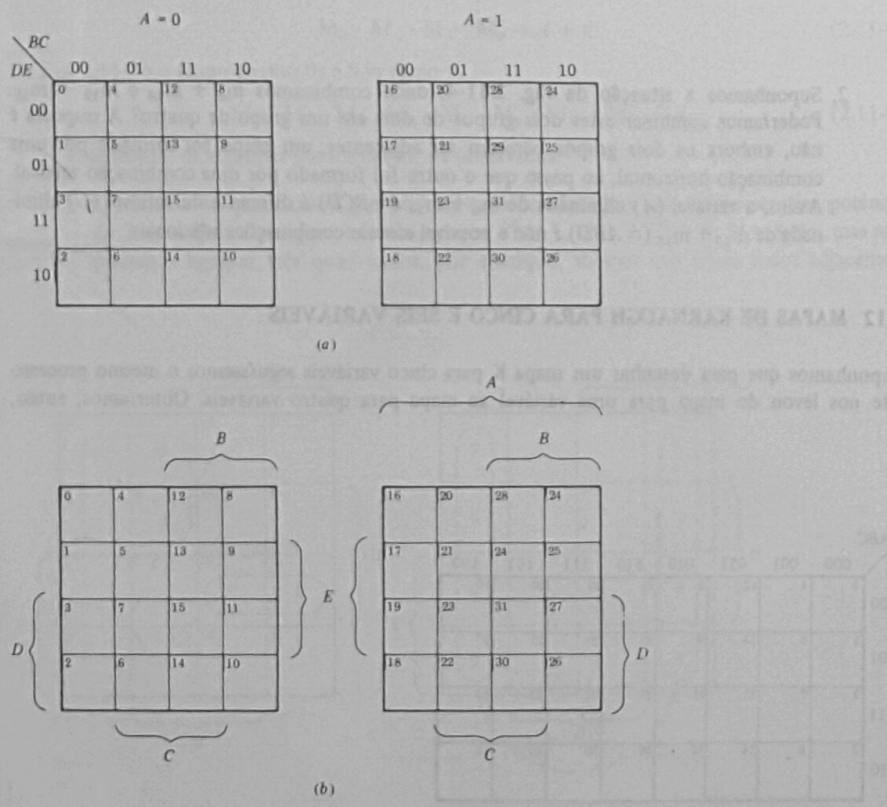


Fig. 2.12-2 O mapa K de cinco variáveis geralmente mais usado. As duas notações para identificação dos quadrículos são mostrados em (*a*) e (*b*).

fácil. O mapa K alternativo para cinco variáveis que geralmente é usado é mostrado na Fig. 2.12-2 com as duas notações para identificação dos quadrículos. Aqui todas as adjacências previamente estabelecidas para o mapa K de quatro variáveis continuam a existir tanto na parte esquerda do mapa, correspondente a $A = 0$, como na parte direita do mapa, correspondente a $A = 1$. Além disso, cada quadrículo da parte $A = 0$ é adjacente ao quadrículo correspondente da parte $A = 1$. Por exemplo, m_5 é adjacente a m_{21}, m_{15} é adjacente a m_{31} e assim por diante. Estas adjacências sugerem que a representação plana adotada corresponde a colocarmos uma das partes do mapa sobre a outra especialmente. Torna-se, então, fácil lembrar que dois quadrículos na mesma vertical são adjacentes.

Seguindo as mesmas considerações que nos levaram a construir o mapa de cinco variáveis, podemos construir um mapa K para seis variáveis, como o da Fig. 2.12-3. As adjacências usuais continuam a existir em cada uma das quatro partes do mapa. Adicionalmente, existem termos adjacentes horizontal e verticalmente entre as diversas partes do mapa; assim, m_5 é adjacente a m_{21}, m_{63} é adjacente a m_{31} e a m_{47} etc.

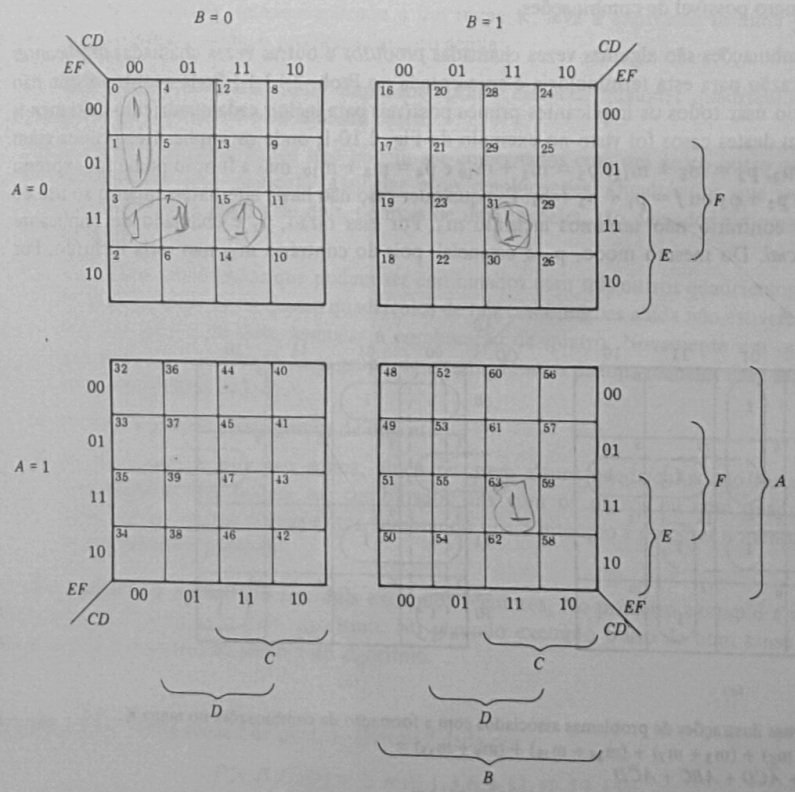


Fig. 2.12-3 O mapa K para seis variáveis.

Como já foi dito, o mérito do mapa de Karnaugh é permitir a visualização de termos adjacentes. Quando o número de variáveis cresce (sete ou mais, por exemplo), o mapa torna-se tão grande, que seu valor no reconhecimento de termos adjacentes torna-se questionável. Assim, para o caso de muitas variáveis, os métodos tabulares são preferidos.

2.13 USO DOS MAPAS DE KARNAUGH

Quando uma função lógica for expressa na forma padrão em termos de mintermos, o mapa K pode ser usado para simplificar a função pela aplicação dos seguintes princípios:

1. A combinação de quadrículos (mintermos) que for selecionada deve incluir todos os quadrículos pelo menos uma vez. Como já foi visto, um quadrículo qualquer pode aparecer em mais de uma combinação.
2. As combinações devem ser selecionadas com a finalidade de incluir o maior número possível de quadrículos de tal modo que todos os quadrículos sejam incluídos pelo menor número possível de combinações.

As combinações são algumas vezes chamadas *produtos* e outras vezes chamadas *implicantes primos*. (A razão para esta terminologia é apresentada no Prob. 2.13-1.) Pode acontecer que não seja necessário usar todos os implicantes primos possíveis para incluir cada quadrículo pelo menos uma vez. Um destes casos foi visto no exemplo da Fig. 2.10-1, onde os implicantes primos eram $p_1 = m_2 + m_3$, $p_2 = m_8 + m_{12}$, $p_3 = m_2 + m_{10}$ e $p_4 = m_8 + m_{10}$, mas a função podia ser expressa por $f = p_1 + p_2 + p_3$ ou $f = p_1 + p_2 + p_4$. Em qualquer caso não havia alternativa quanto ao uso de p_1, pois do contrário não teríamos incluído m_3. Por essa razão, p_1 é chamado de implicante primo *essencial*. Do mesmo modo, p_2 é essencial, pois do contrário m_{12} não seria incluído. Por

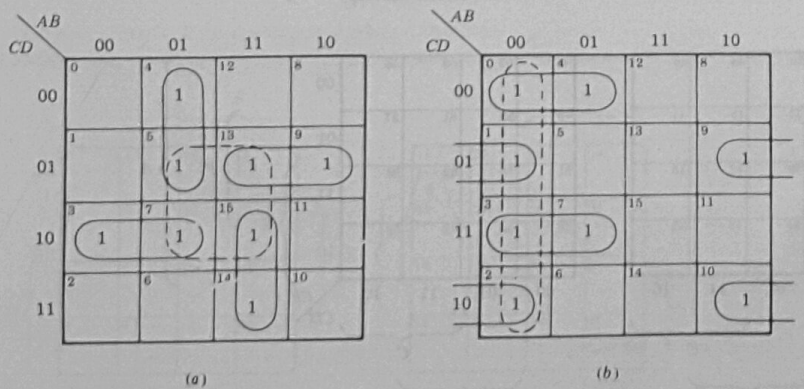


Fig. 2.13-1 Duas ilustrações de problemas associados com a formação de combinações no mapa K:
(*a*) $f = (m_4 + m_5) + (m_3 + m_7) + (m_{14} + m_{15}) + (m_9 + m_{13}) =$
$= \bar{A}B\bar{C} + \bar{A}CD + ABC + A\bar{C}D$
(*b*) $f = (m_0 + m_4) + (m_1 + m_9) + (m_3 + m_7) + (m_2 + m_{10}) =$
$= \bar{A}\bar{C}\bar{D} + \bar{B}\bar{C}D + \bar{A}CD + \bar{B}C\bar{D}$

outro lado, como podemos ou não selecionar p_3, este implicante primo não é essencial. A mesma observação aplica-se a p_4.

Uma vez expressa uma função sob a forma de soma de implicantes primos, torna-se necessário usar uma porta AND para cada implicante. Além disso, o número de entradas dessa porta diminui à medida que aumenta o número de quadrículos incluídos no implicante primo. A economia de uma estrutura de portas é julgada primeiro pelo número de portas usadas. Para diferentes estruturas com o mesmo número de portas a economia é julgada pelo número total de entradas.

A preocupação de encontrar implicantes primos num mapa K que incluam o maior número possível de quadrículos pode causar um problema, como ilustrado na Fig. 2.13-1*a*, onde poderíamos tentar combinar $m_5 + m_7 + m_{13} + m_{15}$ conforme indicado pelo círculo tracejado. Isto feito, ainda seria necessário adicionar quatro outros implicantes primos para incluir os quatro 1s restantes. Uma vez incluídos estes quatro implicantes primos, notamos que os quatro quadrículos que constituíam a combinação original são automaticamente incluídos, o que torna a combinação original supérflua. A função está escrita sob o mapa K. Um exemplo similar é mostrado na Fig. 2.13-1*b*.

O algoritmo abaixo, quando aplicado a um mapa K, leva à expressão mínima para uma função lógica, evitando, ainda, o problema descrito acima:

1. Assinalar e considerar como implicante primo essencial qualquer quadrículo que não possa ser combinado com nenhum outro.
2. Identificar os quadrículos que podem ser combinados com um único outro quadrículo somente de uma maneira. Assinalar estas combinações. Quadrículos que podem ser combinados em grupos de dois, de mais de uma maneira, são deixados temporariamente de lado.
3. Identificar quadrículos que podem ser combinados com três outros quadrículos somente de uma maneira. Se os quatro quadrículos de tais combinações ainda não estiverem incluídos em grupos de dois, assinalar a combinação de quatro. Novamente um quadrículo que pode ser combinado num grupo de quatro de mais de uma maneira deve ser deixado temporariamente de lado.
4. Repetir o processo para grupos de oito etc.
5. Se, encerrado o processo acima, ainda restarem alguns quadrículos não incluídos em grupamentos, eles podem ser combinados uns com os outros ou com quadrículos já incluídos em outros grupamentos, lembrando que a intenção é de obter o menor número de grupamentos possível.

Este algoritmo é ilustrado nos dois exemplos seguintes. No primeiro exemplo a solução é completamente determinada pelo algoritmo. No segundo exemplo o uso de bom senso nos permite satisfazer o requisito do passo 5 do algoritmo.

Exemplo 2.13-1 Uma função de quatro variáveis é dada por

$$f(A,B,C,D) = \Sigma\, m(0,1,3,6,9,11,12,13,15) \qquad (2.13\text{-}1)$$

Use um mapa K para minimizar a função.

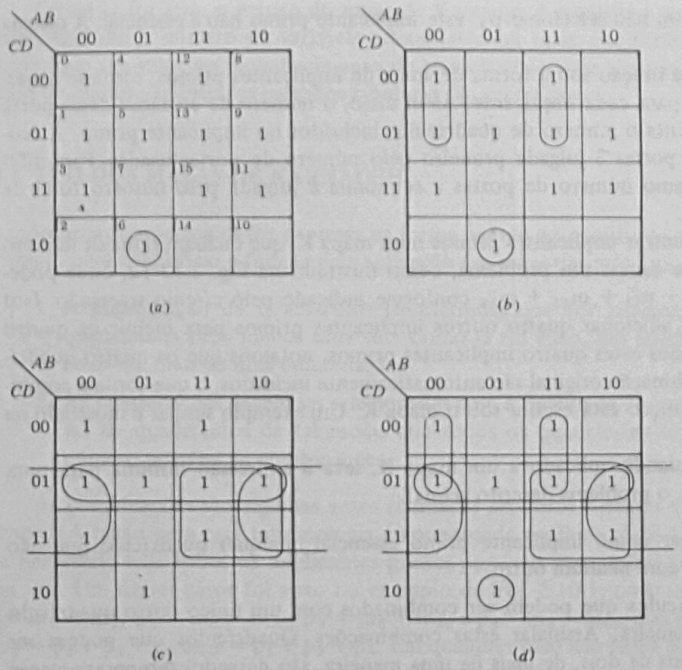


Fig. 2.13-2 Mapa K do Exemplo 2.13-1.

SOLUÇÃO: O mapa K para a função da Eq. (2.13-1) é mostrado na Fig. 2.13-2*a*. Notamos que m_6 não pode ser combinado com nenhum outro quadrículo. Conseqüentemente, nós o assinalamos e o aceitamos como implicante primo essencial. A seguir notamos que m_0 e m_{12} só podem ser combinados em grupos de dois de uma única maneira; conseqüentemente, assinalamos cada um destes grupos de dois conforme mostrado na Fig. 2.13-2*b*. Quadrículos que podem ser combinados em grupos de dois de mais de uma maneira são deixados de lado. Observamos que m_3, m_5 e m_{15} podem ser incorporados em grupos de quatro somente de uma maneira e notamos que os grupos assim formados incluem quadrículos ainda não incorporados a grupos de dois. Assim, assinalamos estes três grupos de quatro conforme mostra a Fig. 2.13-2*c*. Finalmente, na Fig. 2.13-2*d* agrupamos todas as combinações e observamos que todos os quadrículos foram incluídos. Deste mapa, obtemos

$$f(A,B,C,D) = \bar{A}BC\bar{D} + \bar{A}\bar{B}\bar{C} + AB\bar{C} + \bar{C}D + \bar{B}D + AD \tag{2.13-2}$$

Exemplo 2.13-2 Uma função de quatro variáveis é dada por

$$f(A,B,C,D) = \Sigma\, m(0,2,3,4,5,7,8,9,13,15) \tag{2.13-3}$$

Use um mapa K para minimizar a função.

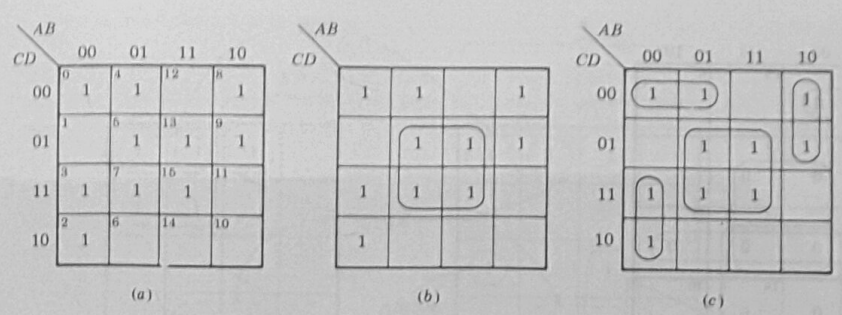


Fig. 2.13-3 Mapa K do Exemplo 2.12-2.

SOLUÇÃO: O mapa K para a função da Eq. (2.13-3) é mostrado na Fig. 2.13-3*a*. Aplicando os passos 1 e 2, não se seleciona nenhum implicante primo. Os quadrículos m_5, m_7, m_{13} e m_{15} satisfazem as condições do passo 3. A aplicação do processo do passo 3 nos leva ao grupamento da Fig. 2.13-3*b*. O passo 4 não se aplica no caso presente. Verificamos que um número de quadrículos ainda não foi agrupado; conforme indica o passo 5, podemos combiná-los arbitrariamente. É óbvio que as combinações indicadas na Fig. 2.13-3*c* nos conduzem a um número mínimo de implicantes primos adicionais. A solução obtida da Fig. 2.13-3*c* é, portanto:

$$f(A,B,C,D) = \bar{A}\bar{C}\bar{D} + \bar{A}\bar{B}C + A\bar{B}\bar{C} + BD \tag{2.13-4}$$

Para variar, consideremos agora um exemplo em que a função é especificada em termos de 0s em vez de 1s, isto é, em termos de maxtermos ao invés de mintermos. Neste caso 0s e não 1s serão colocados no mapa K e a solução será na forma de produto de somas e não de soma de produtos. O algoritmo para combinação de mintermos aplica-se igualmente a maxtermos, mas com uma alteração de terminologia. Correspondendo ao termo implicante primo, definido como um termo produto em uma soma representando a função, temos o termo *implicado primo*, definido como um termo soma em um produto de somas.

Exemplo 2.13-3 Uma função de quatro variáveis é dada por

$$f(A,B,C,D) = \Pi M(0,3,4,5,6,7,11,13,14,15) \tag{2.13-5}$$

Use um mapa K para minimizar a função.

SOLUÇÃO: O mapa K é mostrado na Fig. 2.13-4. O algoritmo leva, de maneira única, aos grupamentos indicados. Diretamente do mapa obtemos

$$f(A,B,C,D) = (A + C + D)(\bar{C} + \bar{D})(\bar{B} + \bar{D})(\bar{B} + \bar{C}) \tag{2.13-6}$$

A implementação da Eq. (2.13-6) usando portas AND e OR é mostrada na Fig. 2.13-5. Observe-se que cada termo da Eq. (2.13-6) requer uma porta OR e que as saídas de todas as portas OR são levadas à entrada de uma porta AND. A Fig. 2.13-5 deve ser comparada com a Fig. 2.10-2, onde uma função na forma de soma de produtos foi implementada com a ordem das portas OR e AND invertida.

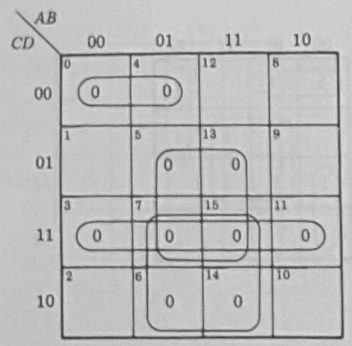


Fig. 2.13-4 Mapa K para o Exemplo 2.13-3.

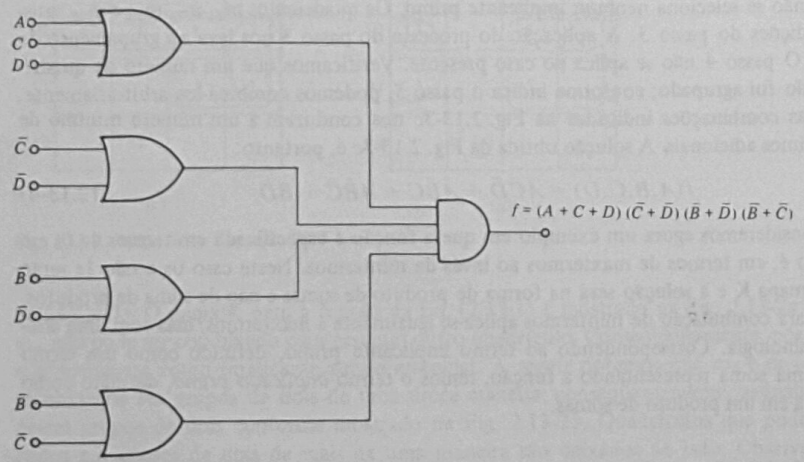


Fig. 2.13-5 Implementação de portas da Eq. (2.13-6).

Exemplo 2.13-4 Os mintermos de uma função de cinco variáveis foram indicados no mapa K da Fig. 2.13-6. Interpretar o mapa.

SOLUÇÃO: Seguindo o algoritmo visto anteriormente, os quadrículos com 1s foram combinados conforme indicado. A leitura das combinações foi indicada no mapa, resultando em

$$f(A,B,C,D,E) = A\bar{B}\bar{C}DE + ABCD + \bar{A}\bar{B}\bar{D} + B\bar{D}\bar{E} + C\bar{E} \qquad (2.13\text{-}7)$$

Exemplo 2.13-5 Consideremos um mapa K como o da Fig. 2.13-6, que foi interpretado no Exemplo 2.13-4, mas contendo 0s em vez de 1s, de modo a que o mapa represente uma função

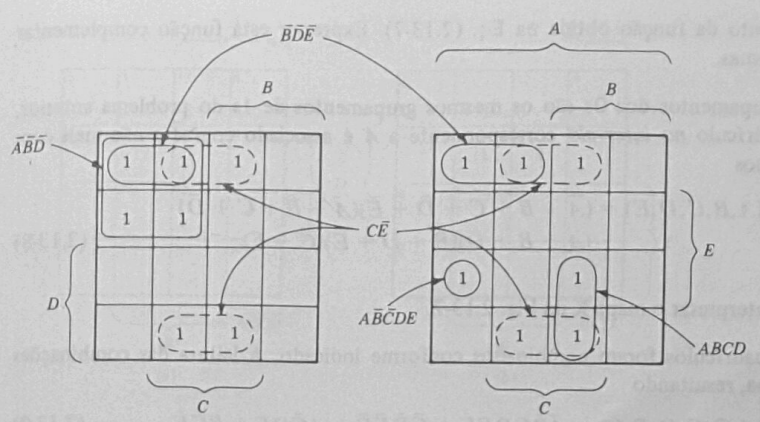


Fig. 2.13-6 Exemplo de um mapa K de cinco variáveis.

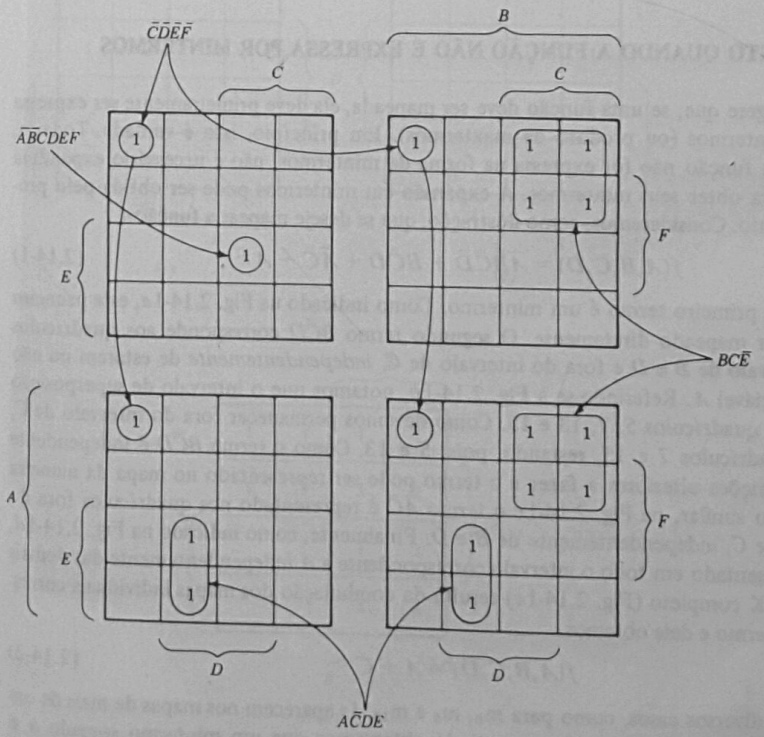


Fig. 2.13-7 Exemplo de um mapa K de seis variáveis.

que é o complemento da função obtida na Eq. (2.13-7). Expressar esta função complementar como produto de somas.

SOLUÇÃO: Os grupamentos dos 0s são os mesmos grupamentos de 1s do problema anterior, mas agora um quadrículo *no intervalo* correspondente a A é associado com $\bar{A}$ e não mais com A etc. Assim, obtemos

$$f(A,B,C,D,E) = (\bar{A} + B + C + \bar{D} + \bar{E})(\bar{A} + \bar{B} + \bar{C} + \bar{D})$$
$$(A + B + D)(\bar{B} + D + E)(\bar{C} + E) \qquad (2.13\text{-}8)$$

Exemplo 2.13-6 Interpretar o mapa K da Fig. 2.13-7.

SOLUÇÃO: Os quadrículos foram combinados conforme indicado. A leitura das combinações foi indicada no mapa, resultando

$$f(A,B,C,D,E,F) = \bar{A}\bar{B}CDEF + \bar{C}\bar{D}\bar{E}\bar{F} + A\bar{C}DE + BC\bar{E} \qquad (2.13\text{-}9)$$

2.14 MAPEAMENTO QUANDO A FUNÇÃO NÃO É EXPRESSA POR MINTERMOS

Nossa discussão sugere que, se uma função deve ser mapeada, ela deve primeiramente ser expressa como soma de mintermos (ou produto de maxtermos). Em princípio, isto é verdade. Todavia, na prática, se uma função não for expressa na forma de mintermos, não é necessário expandi-la algebricamente para obter seus mintermos. A expansão em mintermos pode ser obtida pelo processo de mapeamento. Consideremos, como ilustração, que se deseje mapear a função

$$f(A,B,C,D) = \bar{A}\bar{B}\bar{C}\bar{D} + B\bar{C}D + A\bar{C} + A \qquad (2.14\text{-}1)$$

na qual somente o primeiro termo é um mintermo. Como indicado na Fig. 2.14-1*a*, este primeiro mintermo pode ser mapeado diretamente. O segundo termo $B\bar{C}D$ corresponde aos quadrículos que estão no intervalo de B e D e fora do intervalo de C, *independentemente* de estarem ou não no intervalo da variável A. Referindo-se à Fig. 2.14-1*b*, notamos que o intervalo de superposição de B e D inclui os quadrículos 5, 7, 13 e 15. Como devemos permanecer fora do intervalo de C, eliminamos os quadrículos 7 e 15, restando, pois, 5 e 13. Como o termo $B\bar{C}D$ é independente de A, não há restrições ulteriores a fazer e o termo pode ser representado no mapa da maneira indicada. De modo similar, na Fig. 2.14-1*c* o termo $\bar{A}\bar{C}$ é representado nos quadrículos fora do intervalo de A e de C, independentemente de B e D. Finalmente, como indicado na Fig. 2.14-1*d*, o termo A é representado em todo o intervalo correspondente a A independentemente das demais variáveis. O mapa K completo (Fig. 2.14-1*e*) resulta da combinação dos mapas individuais correspondentes a cada termo e dele obtemos

$$f(A,B,C,D) = A + \bar{C} \qquad (2.14\text{-}2)$$

Notamos que, em diversos casos, como para m_0, m_5 e m_{13}, 1s aparecem nos mapas de mais de um termo. Esta situação não causa problemas, pois já observamos que um mintermo somado a si mesmo resulta em apenas um mintermo.

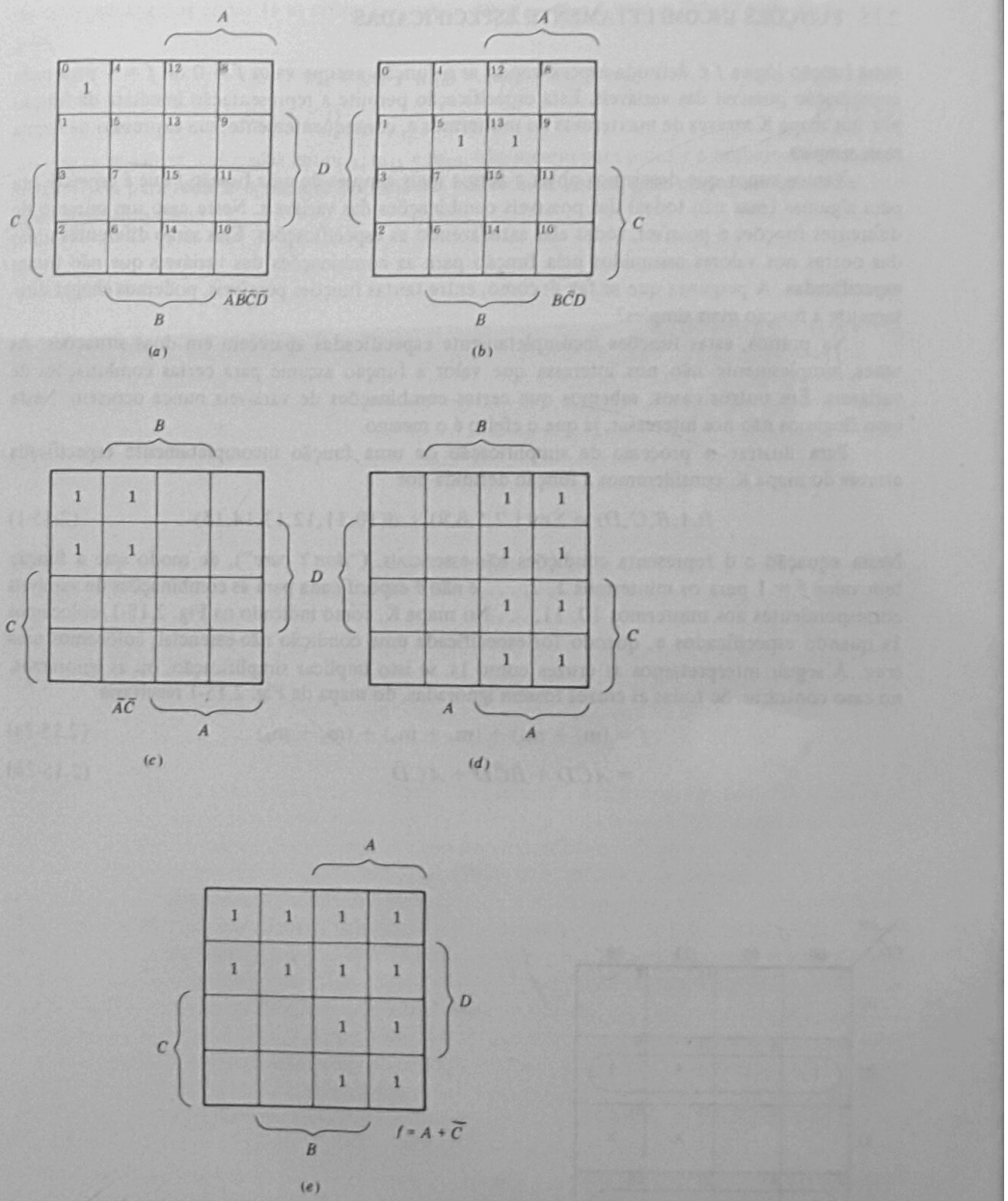


Fig. 2.14-1 Uma função lógica pode ser expandida em seus mintermos diretamente no mapa K.

2.15 FUNÇÕES INCOMPLETAMENTE ESPECIFICADAS

Uma função lógica f é definida especificando se a função assume valor $f = 0$ ou $f = 1$ para cada combinação possível das variáveis. Esta especificação permite a representação imediata da função por um mapa K através de maxtermos ou mintermos e, conseqüentemente, sua expressão na forma mais simples.

Vamos supor que desejemos obter a forma mais simples de uma função f que é especificada para algumas (mas não todas) das possíveis combinações das variáveis. Neste caso um número de diferentes funções é possível, todas elas satisfazendo as especificações. Elas serão diferentes umas das outras nos valores assumidos pela função para as combinações das variáveis que não foram especificadas. A pergunta que se faz é: como, entre tantas funções possíveis, podemos chegar diretamente à função mais simples?

Na prática, estas funções incompletamente especificadas aparecem em duas situações. Às vezes, simplesmente não nos interessa que valor a função assume para certas combinações de variáveis. Em outros casos, sabemos que certas combinações de variáveis nunca ocorrem. Neste caso fingimos não nos interessar, já que o efeito é o mesmo.

Para ilustrar o processo de simplificação de uma função incompletamente especificada através do mapa K, consideremos a função definida por

$$f(A,B,C,D) = \Sigma m(1,2,5,6,9) + d(10,11,12,13,14,15) \qquad (2.15\text{-}1)$$

Nesta equação o d representa condições não-essenciais, ("*don't care*"), de modo que a função tem valor $f = 1$ para os mintermos 1, 2, . . . e não é especificada para as combinações de variáveis correspondentes aos mintermos 10, 11, . . . No mapa K, como indicado na Fig. 2.15-1, colocamos 1s quando especificados e, quando for especificada uma condição não-essencial, colocamos uma cruz. A seguir interpretamos as cruzes como 1s, se isto implicar simplificação, ou as ignoramos, no caso contrário. Se todas as cruzes fossem ignoradas, do mapa da Fig. 2.15-1 resultaria

$$f = (m_1 + m_5) + (m_1 + m_9) + (m_2 + m_6) \qquad (2.15\text{-}2a)$$

$$= \bar{A}\bar{C}D + \bar{B}\bar{C}D + \bar{A}C\bar{D} \qquad (2.15\text{-}2b)$$

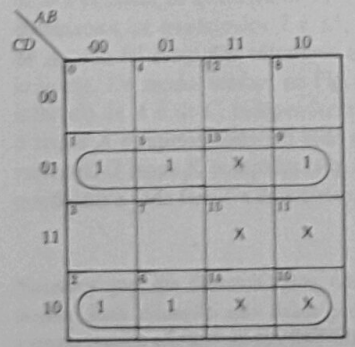


Fig. 2.15-1 Uma função incompletamente especificada.

Se interpretássemos como 1s as cruzes colocadas em m_{10}, m_{13} e m_{14}, a função seria simplificada para

$$f = (m_1 + m_5 + m_9 + m_{13}) + (m_2 + m_6 + m_{10} + m_{14}) \qquad (2.15\text{-}3a)$$

$$= \bar{C}D + C\bar{D} \qquad (2.15\text{-}3b)$$

As cruzes restantes, colocadas em m_{11}, m_{12} e m_{15}, não servem para reduzir o número de termos da função nem para reduzir o número de variáveis nos termos. Assim, estas cruzes são ignoradas, isto é, interpretadas como 0s.

CAPÍTULO 3

CIRCUITOS COMBINACIONAIS BÁSICOS

3.1 INTRODUÇÃO

Sistemas digitais são construídos a partir de portas. O principal fator determinante da velocidade com que um sistema digital pode executar sua função é a velocidade com que operam as portas. O fator mais importante relacionado com a velocidade de uma porta é o atraso de propagação t_{ap} (*propagation-delay time*). É o tempo requerido pela saída de uma porta para responder a uma mudança no nível lógico da entrada da porta. Portas que devem operar rapidamente, com t_{ap} no intervalo de microssegundos a nanossegundos (10^{-9}s) empregam dispositivos eletrônicos.

Num sistema digital eletrônico a tensão em qualquer ponto do circuito tem um valor ou outro representando um ou outro dos dois níveis lógicos. Correspondentemente os dispositivos eletrônicos do sistema devem conduzir uma quantidade substancial de corrente ou conduzir muito pouca ou nenhuma corrente. Enquanto o sistema puder distinguir entre as duas situações, isto é, correntes grandes ou pouca ou nenhuma corrente, o valor real da corrente não é importante. Esta situação é bem diferente da que ocorre quando usamos dispositivos eletrônicos em sistemas analógicos, onde, pelo menos em princípio, qualquer mudança na corrente, não importa quão pequena, interessa. Resumindo, em um sistema digital um componente eletrônico sozinho faz *muito pouco*. Conseqüentemente, para construir um sistema digital de certo significado são necessários muitos componentes eletrônicos – milhares, centenas de milhares e até milhões.

Até 1955, os componentes eletrônicos disponíveis para construir sistemas digitais eram os díodos semicondutores e as válvulas a vácuo. Os díodos são relativamente pequenos, com dimensões da ordem de milímetros, e consomem relativamente pouca potência. As válvulas, por outro lado, são grandes, tendo dimensões da ordem de vários centímetros e consomem quantidades relativamente grandes de potência, tipicamente da ordem de alguns watts. Embora em sua maioria as portas pudessem ser construídas com díodos e resistores, também era necessário usar válvulas em grandes quantidades. Como resultado, qualquer sistema digital era grande, caro e usava muita potência. A situação melhorou consideravelmente com a invenção do transistor nos anos 50. Um transistor, normalmente substituindo uma válvula, consome muito menos potência (da ordem de dezenas de miliwatts) e, como o díodo semicondutor, quando encapsulado individualmente, tem dimensões da ordem de centímetros.

Até 1965, somente eram disponíveis semicondutores encapsulados individualmente e os engenheiros montavam portas e sistemas digitais a partir destes componentes individualmente encapsulados e de resistores. Dispositivos semicondutores são fabricados através da aplicação repetida de certos processos físicos e químicos à superfície de uma pastilha de silício extremamente puro. Os detalhes da fabricação de semicondutores não nos interessam exceto no que tange às dimensões do dispositivo na superfície do silício, que são da ordem de mícrons. A maior parte da dimensão de um dispositivo encapsulado individualmente envolve não o dispositivo mas o próprio encapsulamento e o suporte mecânico necessário para as conexões elétricas. Assim iniciou-se uma série de progressos tecnológicos (que prosseguem sem que se possa vislumbrar seu final) que levaram à fabricação de um dispositivo semicondutor chamado *circuito integrado* (CI). Em um circuito integrado muitos transistores e díodos são fabricados, isto é, integrados, sobre a mesma pastilha de silício; na mesma estrutura são também integrados os resistores e até mesmo as interligações necessárias para fabricar uma porta completa, muitas portas e, até mesmo, um sistema digital completo.

Circuitos integrados comercialmente disponíveis são classificados como com integração em pequena escala (SSI), integração em média escala (MSI), integração em larga escala (LSI) e integração em muito larga escala (VLSI). Uma convenção é que um circuito com até doze portas é considerado SSI, de treze a 99 portas é considerado MSI, de cem a mil portas é considerado LSI e acima de mil portas, VLSI.

A origem dos conceitos envolvidos em sistemas digitais perde-se na Antiguidade. Sem dúvida, os povos primitivos podiam contar, e o ábaco, que compete favoravelmente com uma calculadora mecânica, tem pelo menos 2500 anos. Uma calculadora mecânica digital simples que podia somar e subtrair foi construída no século XVII. Um dispositivo mecânico incorporando alguns dos conceitos envolvidos nos computadores modernos foi construído no princípio do século XIX. Calculadoras de mesa mecânicas produzidas em massa e, conseqüentemente, baratas tornaram-se disponíveis logo após 1930, com capacidade de somar, subtrair, multiplicar e dividir. Nos anos 40, circuitos eletrônicos usando válvulas deram aos sistemas digitais velocidades que não podiam ser atingidas pelos sistemas mecânicos, mas estes sistemas eram grandes e consumiam grandes potências. Os transistores apareceram nos anos 50 e os circuitos integrados começaram a evoluir nos anos 60. No início da década de 80, o desenvolvimento dos circuitos integrados atingiu o ponto de se obter um circuito que cabe na palma da mão, dissipa em torno de 1 W e supera sistemas existentes há 30 anos que envolviam uma sala cheia de equipamento e consumiam milhares de watts.

Neste capítulo descreveremos alguns circuitos integrados com integração em pequena e média escala (SSI e MSI) e que contêm os circuitos básicos a partir dos quais se constroem os sistemas digitais. Serão considerados somente os circuitos *combinacionais*, isto é, circuitos cujas saídas dependem somente dos níveis das entradas atuais e não têm relação com a história passada do circuito.

3.2 FAMÍLIAS DE CIRCUITOS LÓGICOS

Existe um número de *famílias* de circuitos integrados lógicos que se distinguem umas das outras pelo tipo de dispositivo semicondutor que incorporam e pela maneira como os dispositivos semicondutores (e resistores, quando usados) são interligados para formar portas. As características dos tipos de transistor e as características elétricas das diversas famílias são descritas em detalhe em outros livros.* Comentaremos o assunto brevemente com a finalidade de explicar por que, neste capítulo onde nosso interesse fica confinado à integração em pequena e média escala, limitaremos nossa discussão a uma só família.

Existem dois tipos de transistores. Um é o transistor de *semicondutor-metal-óxido* (MOS). Este transistor pode ser fabricado de modo que a corrente que por ele circula seja conduzida por cargas elétricas negativas (n), sendo, conseqüentemente, chamado MOS *canal-n* (tipo n) ou por cargas elétricas positivas (p) sendo, então, chamado MOS *canal-p* (tipo p). A grande vantagem desta família é que não é necessário usar resistores. Como os transistores ocupam espaço muito pequeno na pastilha de silício, são muito apropriados para integração em larga e muito larga escala. Uma família MOS usa transistores tipo p ou transistores tipo n. Esta família possui características elétricas que se tornam inconvenientes para integração em pequena e média escala, não sendo, portanto, usada nestas aplicações e encontrando maior uso na fabricação de circuitos com integração em larga e muito larga escala. Uma segunda família usa transistores do tipo p e do tipo n no mesmo circuito, sendo chamada família MOS de *simetria complementar* (CMOS) A família CMOS é usada em integração em larga escala e, por não possuir algumas deficiências da família MOS, também em integração em pequena e média escala.

O segundo tipo de transistor é o *transistor bipolar*, atualmente fabricado em três famílias. Uma delas é chamada *lógica de injeção integrada* (IIL). Como MOS e CMOS, não requer o uso de resistores no circuito sendo, portanto, apropriada para integração em larga escala.

Uma segunda família usando transistores bipolares é chamada de *lógica acoplada pelo emissor* (ECL). Usa muitos transistores por porta; mesmo uma porta de duas entradas requer cinco transistores. É a família lógica de maior velocidade, pois usa transistores bipolares rápidos em vez de transistores MOS mais lentos e os usa de modo eficiente para fazê-los responder com grande velocidade. A própria velocidade da família ECL faz com que se torne difícil de usar. Devido à rapidez das transições de um nível lógico para outro, faz com que qualquer comprimento de fio que seja usado para interligar os circuitos deva ser tratado como se fosse uma linha de transmissão. A família ECL existe em integração em pequena e média escala, mas só é usada quando sua maior velocidade se faz realmente necessária.

Finalmente, chegamos à família lógica mais usada para circuitos com integração em pequena e média escala. É a *lógica transistor-transistor* (TTL), desenvolvida principalmente pela Texas Instrument Company mas também produzidos por outros fabricantes. Para a família TTL, a Texas Instrument usa a designação genérica SN, significando rede semicondutora ('*semiconductor network*"). Outros fabricantes usam outras designações, como DM (digital monolítico). Existem duas séries, uma com o prefixo identificador 54 e a outra com o prefixo 74. A série 54 é usada para aplicações militares, onde as exigências são maiores, e pode operar no intervalo de temperatura de -55 a $+125°C$. A série 74 é uma versão industrial de custo menor que pode operar de 0 a 70°C.

* Por exemplo, H. Taub e D. L. Schilling: *Eletrônica Digital*, McGraw-Hill, São Paulo, 1981.

3.3 A SÉRIE TTL

Em circuitos eletrônicos semicondutores geralmente é possível melhorar a velocidade de operação (isto é, reduzir o atraso de propagação e reduzir o tempo necessário para a transição de um nível lógico para outro) sacrificando a potência. Como a maior potência envolve maiores correntes, as capacitâncias "parasitas" podem ser carregadas e descarregadas mais rapidamente. Estes capacitores parasitas não são introduzidos deliberadamente no circuito, mas são o resultado inevitável das dimensões e geometria do circuito. A disponibilidade de correntes maiores torna possível ligar e desligar os transistores mais rapidamente. Quando usamos mais potência com a finalidade de obter maior velocidade, gostaríamos de saber se o aumento de velocidade compensa o sacrifício de potência. Uma figura de mérito útil neste contexto é o *produto velocidade-potência*, que é o produto do atraso de propagação pela dissipação de potência de uma porta.

Quando transistores bipolares comuns funcionam em circuitos digitais e são ligados de modo a conduzir corrente, a operação geralmente se dá na região conhecida como *saturação*. Em virtude da saturação o transistor leva um tempo relativamente longo para ser desligado. Conseqüentemente, os circuitos digitais padrão usando transistores comuns sofrem uma desvantagem em relação à velocidade. Com uma despesa adicional pode-se, todavia, fabricar um tipo especial de transistor denominado *Schottky*, que não satura, podendo, conseqüentemente, operar em velocidades mais altas.

Devido ao balanço possível entre velocidade e potência e devido à possibilidade de fabricar transistores comuns ou tipo Schottky, a família TTL existe em cinco séries distintas, que são listadas, com suas características, na Tab. 3.3-1. A razão da popularidade da série LS torna-se aparente, embora outras séries possam ser escolhidas caso haja restrições quanto à velocidade, à dissipação possível ou ao custo.

Capacidade de Saída (Fan-Out)

Uma fonte de um sinal digital aplicado à entrada de uma porta deve ser capaz de estabelecer naquela entrada uma ou outra tensão correspondente a um ou outro nível lógico. Em qualquer um dos níveis a fonte deve satisfazer os requisitos de corrente da porta acionada. Como a saída de uma porta freqüentemente é usada como fonte para a entrada de outra porta, é necessário conhecer a capacidade de acionamento de uma porta, isto é, precisamos saber quantas entradas de portas a serem acionadas podemos ligar à saída de uma porta acionadora. Os fabricantes geralmente fornecem esta informação ao especificar a capacidade de saída (*fan-out*). No caso de TTL,

Tabela 3.3-1 Características típicas da família 54/74 SSI

Séries	Tipo de transistor e potência	Atraso de propagação, ns	Dissipação de potência, mW	Produto velocidade-potência, pJ
54LS/74LS	Schottky, baixa potência	9,5	2	19
54L/74L	Comum, baixa potência	33	1	33
54S/74S	Schottky, potência normal	3	19	57
54/74	Comum, potência normal	10	10	100
54H/74H	Comum, alta potência	6	22	132

desde que cada porta acione portas da mesma série, a capacidade de saída é de 10 para portas das séries padrão e de alta potência e de 20 para as séries de baixa potência. Quando uma porta aciona portas de outras séries, é necessário referir-se à literatura do fabricante para determinar a necessidade de corrente de entrada e a disponibilidade de corrente de saída e assegurar-se de que não há carga excessiva para a saída de uma porta.

Margem de Ruído

Como a família TTL opera com uma tensão de alimentação de 5 V, todas as tensões em um sistema TTL estão no intervalo de 0 a 5 V. Quando uma porta acionadora não estiver carregada pela ligação a entradas de outras portas, sua tensão de saída baixa, correspondente ao nível lógico 0, pode ser 0,1 V ou até menor para a série 54/74. A tensão alta, correspondente ao nível lógico 1, fica em torno de 3,4 V. Quando a saída for baixa, a porta acionadora deve permitir o fluxo de corrente da porta acionada para si própria. A porta acionadora é descrita como *drenando* corrente da carga. Quando a saída estiver no nível alto, a porta acionadora serve como fonte de corrente para a carga e é descrita como *suprindo* corrente. No nível de saída baixo a corrente drenada eleva a tensão de saída, e no nível de saída alto a corrente suprida diminui a tensão de saída. Para a série 54/74, o fabricante garante que, mesmo que uma porta esteja carregada até sua capacidade máxima de saída especificada acima, a tensão de saída baixa não sobe acima de 0,4 V e a tensão de saída alta não desce abaixo de 2,4 V. O fabricante também especifica que uma tensão igual ou menor que 0,8 V sempre será interpretada por uma porta acionada como correspondendo à tensão *baixa* (0 lógico) e que uma tensão de entrada maior que 2 V sempre será interpretada como tensão *alta* (1 lógico). As duas tensões de saída e as duas tensões de entrada são representadas pelos símbolos V_{OH}, V_{OL}, V_{IH} e V_{IL} e têm as seguintes definições:

V_{OH}: A tensão de saída *mínima* que uma porta fornece quando sua saída estiver no nível *alto*.

V_{OL}: A tensão de saída *máxima* que uma porta fornece quando sua saída estiver no nível *baixo*.

V_{IH}: A tensão *mínima* que pode ser aplicada à entrada de uma porta e reconhecida como nível *alto*.

V_{IL}: A tensão *máxima* que pode ser aplicada à entrada de uma porta e reconhecida como nível *baixo*.

Para as séries 54/74, estas tensões são as especificadas na Fig. 3.3-1 e levam à característica de tensão entrada-saída (idealizada) mostrada na Fig. 3.3-2. (A porta a que se refere a figura é uma porta NAND ou NOR com todas as entradas ligadas de modo a que a porta se comporte como um inversor.) Quando a tensão de entrada V_I estiver no intervalo de 0 a 0,8 V ou no intervalo acima de 2,0 V, a saída V_O é constante e vale 2,4 V ou 0,4 V, respectivamente. Para V_I no intervalo de 0,8 a 2,0 V, a saída varia de seu nível alto de 2,4 V até seu nível baixo de 0,4 V.

A importância destas tensões pode ser vista nas seguintes considerações. Suponhamos que uma porta acionadora está em seu nível baixo de 0,4 V. A porta acionada reconhece que sua entrada é baixa porque a tensão é menor que 0,8 V mas, como em qualquer outra implementação física de uma configuração de portas, tensões espúrias e aleatórias (ruído) podem ser superpostas através das conexões que ligam uma parte da configuração a outra. A margem para erro neste nível baixo é a diferença $V_{IL} - V_{OL}$, representada pelo símbolo $\Delta 0$, conforme indicado na Fig. 3.3-1, e chamada *margem de ruído*[*] do nível baixo que tem no caso presente o valor $\Delta 0 = 0{,}4$ V. A margem de ruído do nível alto $\Delta 1 = V_{OH} - V_{IH}$ também tem o valor 0,4 V. Assim, se uma

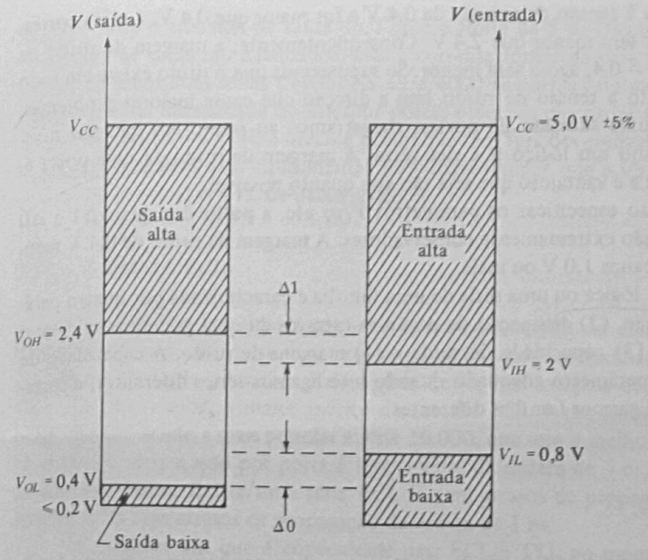


Fig. 3.3-1 Níveis de tensão na lógica TTL.

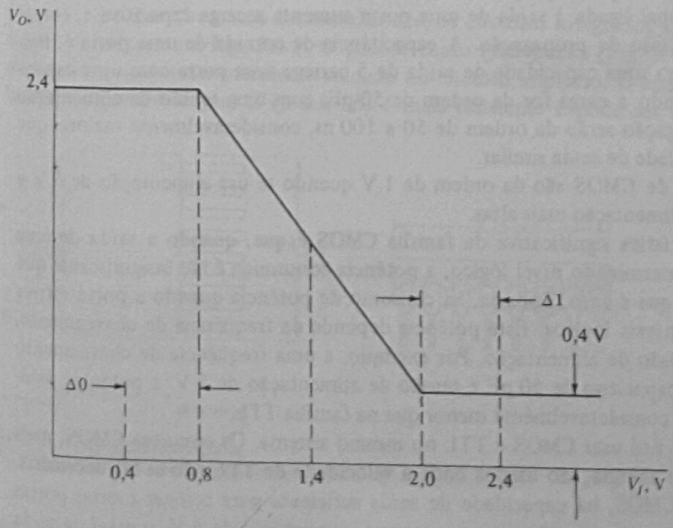


Fig. 3.3-2 Gráfico idealizado do pior caso de entrada-saída para uma porta TTL.

tensão de ruído for adicionada à tensão de entrada de 0,4 V e for maior que 0,4 V, a saída correspondente da porta adicionada será menor que 2,4 V. Conseqüentemente, a margem de ruído na saída, originalmente $2{,}4 - 2{,}0 = 0{,}4$, agora será menor. Se supusermos que o ruído existe em todo o sistema e que em cada ponto a tensão de ruído tem a direção que causa maiores problemas, podemos imaginar que, após uma sucessão de portas, chegaríamos ao ponto em que um nível lógico 1 seria interpretado como um lógico 0 e vice-versa. A margem de ruído de uma porta é, pois, um parâmetro importante e é vantajoso que seja tão alta quanto possível.

Devemos observar que, ao especificar os parâmetros V_{OH} etc. a partir dos quais $\Delta 1$ e $\Delta 0$ são calculados, os fabricantes são extremamente conservadores. A margem de ruído de 0,4 V mencionada acima normalmente alcança 1,0 V ou mais.

Resumindo, uma família lógica ou uma série de uma família é caracterizada por quatro parâmetros: (1) atraso de propagação, (2) dissipação de potência (através da qual podemos calcular o produto velocidade-potência), (3) capacidade de saída e (4) margens de ruído. A capacidade de saída muitas vezes não é um parâmetro adequado quando interligamos séries diferentes, e certamente não o é no caso de interligarmos famílias diferentes.

3.4 A FAMÍLIA CMOS

A família CMOS (semicondutor de metal-óxido de simetria complementar) usa uma tensão de alimentação no intervalo de 5 a 15 V. Uma característica desta família é que a corrente de entrada requerida por uma porta é insignificante, sendo da ordem de 1 pA ou menos. Como a corrente de saída disponível é da ordem de 1 mA, a capacidade de saída é astronômica, já que depende exclusivamente da corrente de saída disponível e da corrente de entrada requerida. Conseqüentemente, a capacidade de saída que os projetistas se permitem usar está relacionada com a velocidade. Cada entrada adicional ligada à saída de uma porta aumenta a carga capacitiva e, conseqüentemente, aumenta o atraso de propagação. A capacitância de entrada de uma porta é, tipicamente, de 5 a 10 pF, logo uma capacidade de saída de 5 carrega uma porta com uma capacitância de 25 a 50 pF. Quando a carga for da ordem de 50 pF, com uma tensão de alimentação de 5 V, os atrasos de propagação serão da ordem de 50 a 100 ns, consideravelmente maiores que na família TTL, com capacidade de saída similar.

As margens de ruído de CMOS são da ordem de 1 V quando se usa alimentação de 5 V e aumentam com tensões de alimentação mais altas.

Uma segunda característica significativa da família CMOS é que, quando a saída de uma porta se mantém fixa em determinado nível lógico, a potência consumida é tão insignificante que se pode dizer sem exagero que é zero. Todavia, há consumo de potência quando a porta estiver sendo chaveada entre dois níveis lógicos. Esta potência depende da freqüência de chaveamento, da carga capacitiva e da tensão de alimentação. Por exemplo, a uma freqüência de chaveamento de 10^5 Hz com uma carga capacitiva de 50 pF e tensão de alimentação de 5 V, a potência dissipada é da ordem de 0,2 mW, consideravelmente menor que na família TTL.

Há ocasiões em que é útil usar CMOS e TTL no mesmo sistema. Os circuitos CMOS, mais baratos e economizadores de energia, são usados onde a velocidade de TTL não se faz necessária. Quando TTL deve acionar CMOS, há capacidade de saída suficiente para acionar muitas portas CMOS, mas, mesmo quando as duas famílias usam a mesma alimentação de 5 V, o nível de saída V_{OH} (TTL) não é suficientemente alto para CMOS. O problema pode ser resolvido ligando um resistor entre o terminal de saída do TTL e a fonte de 5 V. Quando CMOS aciona TTL, os níveis de tensão de saída são adequados, mas a corrente fornecida pela porta CMOS pode não ser suficiente. Existem duas séries CMOS. Na série 4000 as portas geralmente não podem suprir corrente suficiente e há necessidade de interpor portas especiais chamadas *buffers*. A segunda série CMOS tem a designação 54C/74C e oferece portas que têm funções equivalentes às da família TTL 54/74. Estas portas CMOS são, tipicamente, 50% mais rápidas que a série 4000 e têm capacidade para acionar duas entradas TTL de baixa potência.

3.5 A FAMÍLIA ECL

A família ECL de circuitos integrados opera com tensão de alimentação negativa de $-5{,}2$ V. Os níveis lógicos são, portanto, negativos, alcançando aproximadamente $-0{,}75$ e $-1{,}6$ V. A capacidade de saída permitida é da ordem de 25, e as margens de ruído são da ordem de 0,3 V. Como já observamos, o maior mérito da família ECL é sua velocidade. Existem diversas séries nesta família, sendo a mais popular a série 10.000, que tem o melhor produto velocidade-potência. O atraso de propagação por porta é tipicamente da ordem de 2 ns, e a dissipação é da ordem de 25 mW por porta. Existe uma série ECL II com atrasos de propagação da ordem de 4 ns e uma série ECL III com atrasos de propagação da ordem de 1 ns.

Há ocasiões em que é conveniente usar ECL e TTL no mesmo sistema. Para acomodar tal possibilidade, os fabricantes dispõem de *deslocadores de nível lógico* que aceitam os níveis lógicos positivos de TTL e os transformam nos níveis negativos de ECL e vice-versa.

3.6 ENCAPSULAMENTO

De modo geral, embora não invariavelmente, os circuitos integrados são encapsulados de um dos modos mostrados na Fig. 3.6-1. O pacote chato (*flat pack*) (Fig. 3.6-1*a*) é mais compacto e é destinado a ser permanentemente soldado no circuito impresso. O encapsulamento DIP (encapsulamento em linha dupla) (Fig. 3.6-1*b*) é mais resistente e pode ser encaixado em um soquete.

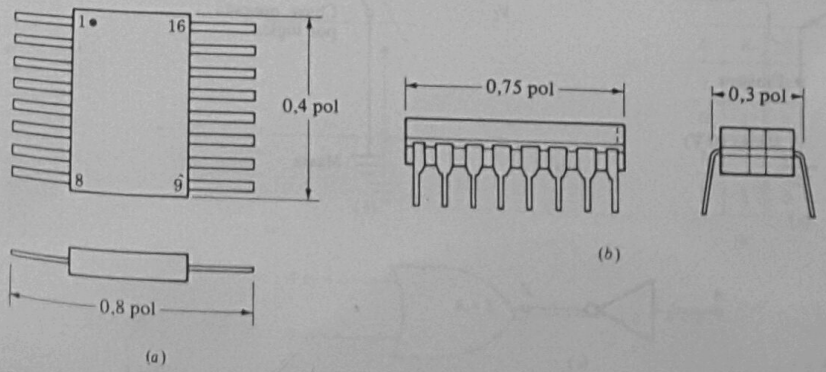


Fig. 3.6-1 Aparência e dimensões aproximadas dos encapsulamentos para CI de dezesseis pinos: (*a*) pacote chato e (*b*) encapsulamento em linha dupla (DIP).

3.7 A CHAVE OPERADA POR LÓGICA

Deliberadamente, evitamos qualquer discussão dos circuitos eletrônicos usados para implementar fisicamente os circuitos digitais, já que o objetivo deste livro é o estudo de sistemas digitais e não de eletrônica. Todavia, o conhecimento de um circuito simples e elementar que usa um transistor apenas é muito útil. Embora superficial, o estudo deste circuito evita a necessidade de explicações complicadas.

O transistor é um dispositivo semicondutor com três terminais: o *emissor*, a *base* e o *coletor*. Consideremos o transistor usado no circuito da Fig. 3.7-1*a*. Uma tensão denominada V_{CC} é aplicada aos terminais do coletor e do emissor do transistor através de um resistor R. (O índice de V_{CC} indica que o terminal mais próximo da tensão da fonte é o do coletor.) O circuito recebe energia elétrica de uma fonte de alimentação que pode ser uma bateria cujo terminal positivo é ligado ao ponto denominado V_{CC} e cujo terminal negativo é chamado *massa* (terra). A massa é o ponto do circuito com referência ao qual todas as tensões são medidas. Para maior simplicidade, a bateria não é mostrada na figura. Esperamos aplicar à *base* do transistor, isto é, entre a base e a massa, uma de duas tensões V_H ou V_L. V_H é uma tensão de *entrada* correspondente ao nível *alto* de tensão, isto é, nível lógico 1 na convenção de lógica positiva, correspondendo V_L à tensão baixa (0 lógico).

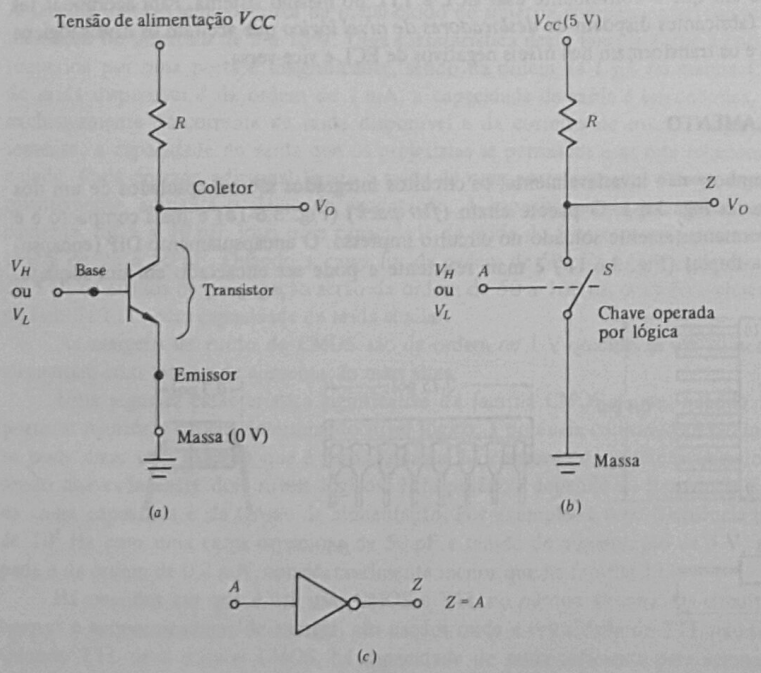
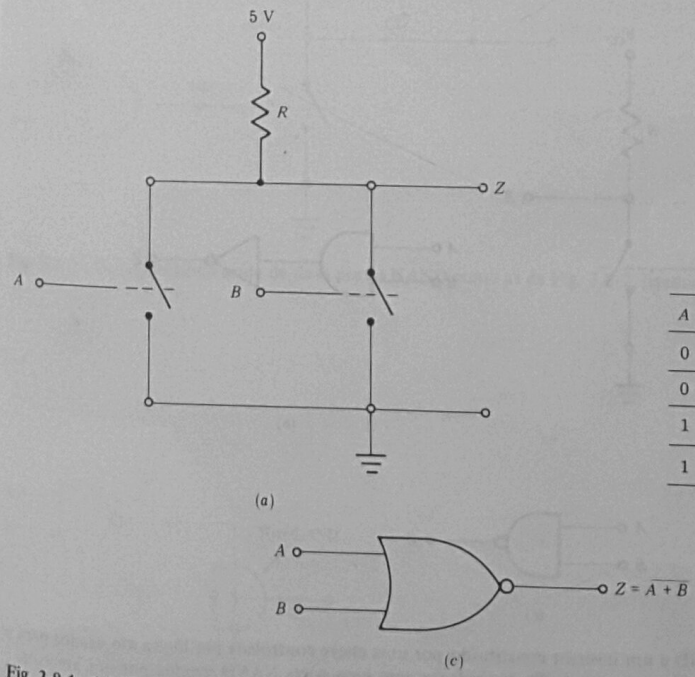


Fig. 3.7-1 (*a*) O transistor conduz ou não conduz conforme a entrada for V_H ou V_L. (*b*) Um circuito equivalente usando uma chave operada por lógica. (*c*) O circuito de (*b*) executa uma inversão lógica.

O que necessitamos saber sobre o circuito é que, quando a tensão aplicada à base for V_L, o transistor não conduz corrente, não havendo, portanto, corrente em R e a tensão de saída é $V_O = V_{CC}$. Quando a entrada for V_H, o transistor conduz corrente e a tensão V_O é praticamente igual à tensão da massa (0 V). Com boa aproximação, o circuito a transistor da Fig. 3.7-1*a* pode ser substituído pelo circuito da Fig. 3.7-1*b*. Aqui temos uma *chave operada por lógica*. Quando a tensão V_H é aplicada ao terminal de controle, a chave fecha; e quando a tensão do terminal de controle é V_L, a chave abre. Observa-se facilmente que o circuito pode ser usado como um *inversor*. Se pretendermos usar o circuito com níveis de tensão correspondentes a TTL, faremos $V_{CC} = 5$ V. Se a entrada estiver no nível 0, lógico S estará aberta e $V_O = 5$ V, correspondente ao 1 lógico, pois qualquer tensão acima de 2,4 V corresponde a 1 lógico. Se a entrada estiver no nível correspondente ao 1 lógico ou maior, S estará fechada e $V_O = 0$ V, correspondente ao 0 lógico, pois qualquer tensão menor que 0,4 V corresponde a 0 lógico. Assim, o nível lógico na saída será sempre o complemento do nível na entrada e o circuito pode ser representado como na Fig. 3.7-1*c*.

3.8 A CHAVE OPERADA POR LÓGICA: PORTA OR; A CONEXÃO WIRED-AND

Como pode ser facilmente verificado, o circuito de duas chaves da Fig. 3.8-1*a* é uma porta NOR com a tabela verdade como a da Fig. 3.8-1*b* e o símbolo lógico como o da Fig. 3.8-1*c*. A saída Z



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

(b)

Fig. 3.8-1 (*a*) Chaves controladas por lógica usadas para obter uma porta NOR; (*b*) tabela verdade; (*c*) símbolo.

só estará no nível lógico 1 quando ambas as entradas A e B estiverem no nível lógico 0 e, conseqüentemente, as duas chaves estiverem abertas.

Consideremos a seguir a configuração mostrada na Fig. 3.8-2*a*. Ela consiste em uma porta AND seguida por um inversor, como na Fig. 3.8-2*b*, constituindo-se, portanto, em uma porta NAND, conforme mostrado na Fig. 3.8-2*c*. A relevância desta configuração para nossa discussão é que muitas portas TTL usam exatamente esta disposição.

Finalmente, considere-se o circuito da Fig. 3.8-3, onde examinamos a conexão das saídas de duas portas NAND como as da Fig. 3.8-2 (conexão tracejada). Esta conexão tracejada combina as duas chaves operadas por lógica para formar uma porta NOR, como a da Fig. 3.8-1. Após esta conexão, a saída será, de acordo com o teorema de De Morgan:

$$Z = \overline{AB + CD} = (\overline{AB}) \cdot (\overline{CD}) \tag{3.8-1}$$

Na ausência da conexão, as duas saídas individuais seriam $\overline{AB}$ e $\overline{CD}$. A simples conexão das saídas por um fio efetivamente adicionou uma porta AND. Esta conexão (nem sempre possível, mas que funciona com as portas do tipo mostrado na Fig. 3.8-2) é chamada de conexão *wired*-AND e é representada pelo simbolismo da Fig. 3.8-4.

A Fig. 3.8-3 mostra a conexão wired-AND aplicada a portas NAND. Verifica-se facilmente que a conexão produz um efeito de wired-AND independente da lógica executada pelas portas individuais antes da ligação. A conexão pode, pois, ser usada com portas AND, OR, NOR etc.

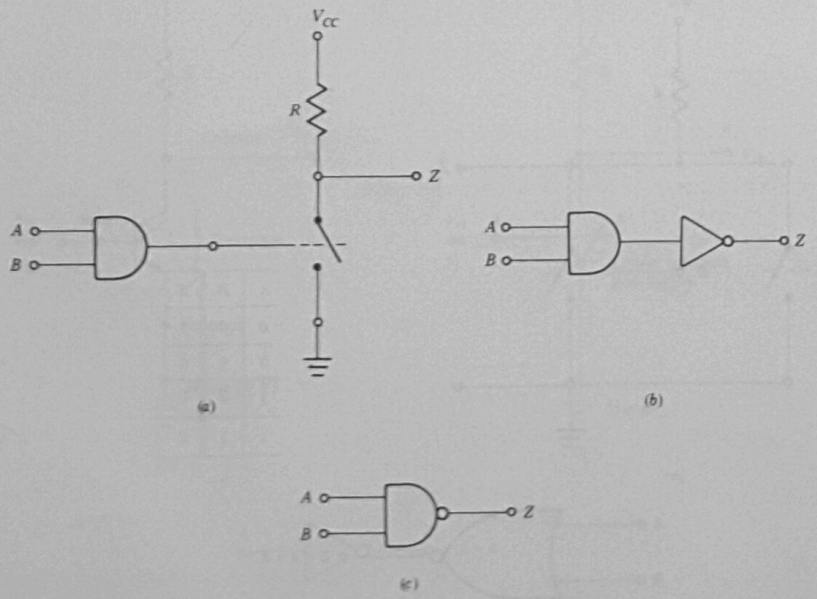


Fig. 3.8-2 (*a*) Uma porta AND e um inversor constituído por uma chave controlada por lógica são usados para obter uma porta NAND. (*b*) Uma representação equivalente com uma porta NAND seguida por um inversor. (*c*) A representação como uma porta NAND.

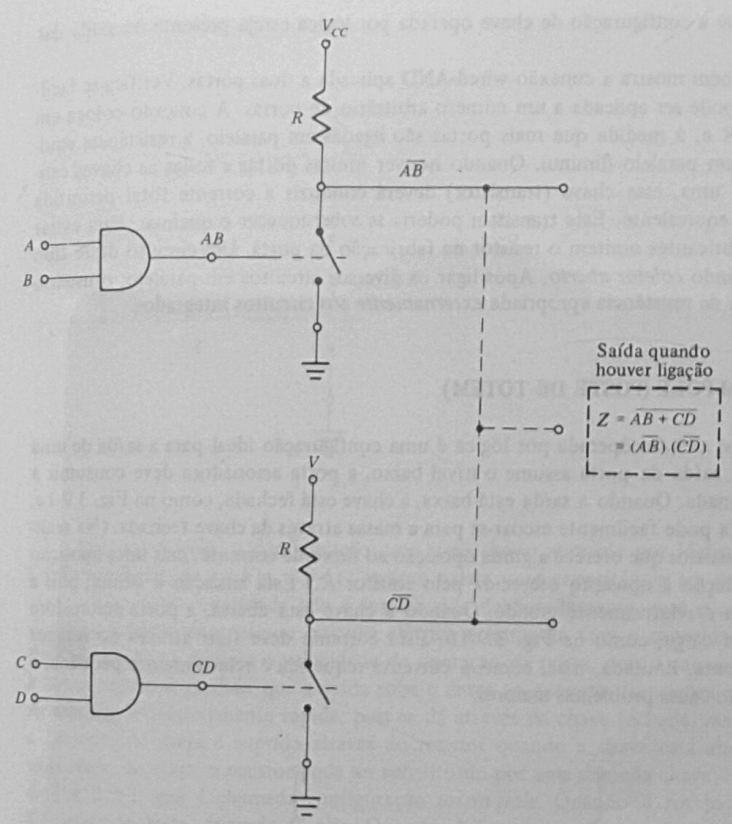


Fig. 3.8-3 Os terminais de saída de duas portas NAND como as da Fig. 3.8-2, ligados juntos.

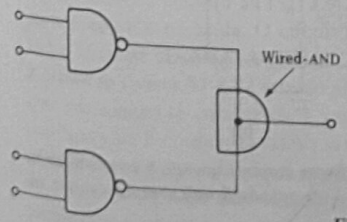


Fig. 3.8-4 O símbolo usado para representar uma conexão wired-AND.

O que é essencial é que a configuração de chave operada por lógica esteja presente na saída das portas usadas.

A Fig. 3.8-3 também mostra a conexão wired-AND aplicada a duas portas. Verifica-se facilmente que a conexão pode ser aplicada a um número arbitrário de portas. A conexão coloca em paralelo os resistores R e, à medida que mais portas são ligadas em paralelo, a resistência equivalente aos resistores em paralelo diminui. Quando houver muitas portas e todas as chaves estiverem abertas, exceto uma, essa chave (transistor) deverá conduzir a corrente total permitida pelo pequeno resistor equivalente. Este transistor poderia se sobreaquecer e queimar. Para evitar esta dificuldade, os fabricantes omitem o resistor na fabricação da porta. Um circuito deste tipo é descrito como possuindo *coletor aberto*. Após ligar os diversos circuitos em paralelo, o usuário ligará um único resistor de resistência apropriada *externamente* aos circuitos integrados.

3.9 A SAÍDA TOTEM-POLE (POSTE DE TOTEM)

Sob um ponto de vista, a chave operada por lógica é uma configuração ideal para a saída de uma porta TTL. Quando a saída da porta assume o nível baixo, a porta acionadora deve consumir a corrente da porta acionada. Quando a saída está baixa, a chave está fechada, como na Fig. 3.9-1*a*, e a corrente consumida pode facilmente escoar-se para a massa através da chave fechada. (Na realidade, a chave é um transistor que oferece alguma oposição ao fluxo de corrente, mas uma oposição muito pequena em relação à oposição oferecida pelo resistor R.) Esta situação é ótima, pois a corrente a ser escoada é relativamente grande. Quando a chave está aberta, a porta acionadora fornece corrente a sua carga, como na Fig. 3.9-1*b*. Esta corrente deve fluir através do resistor R e é, conseqüentemente, limitada, mas, como a corrente requerida é relativamente pequena, a presença do resistor não causa problemas maiores.

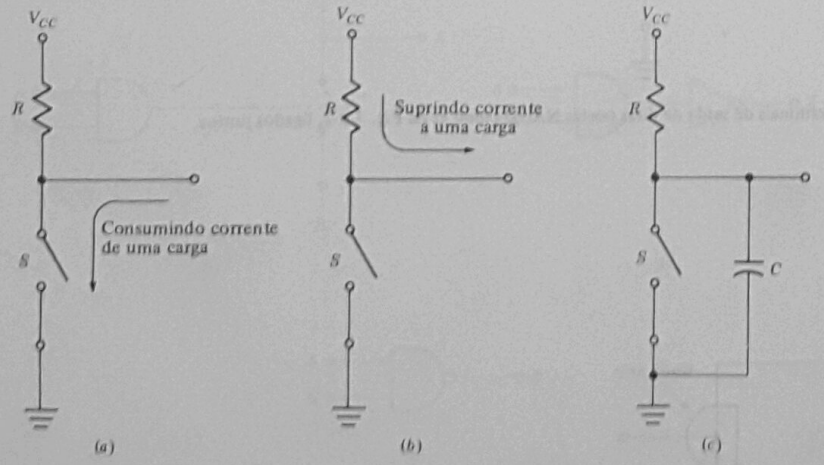


Fig. 3.9-1 Usada na saída de uma porta, a chave operada por lógica consome corrente quando S estiver fechada como em (*a*) e fornece corrente quando S estiver aberta como em (*b*); e (*c*) quando S está aberta, corrente de carga para o capacitor C é suprida através do resistor R.

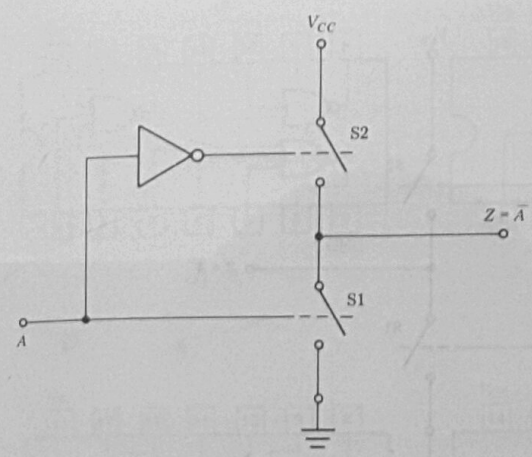


Fig. 3.9-2 A configuração totem-pole.

Por outro lado, a configuração com uma chave acionada por lógica na saída não é ideal. Conforme indicado na Fig. 3.9-1*c*, sempre existe uma capacitância em paralelo com a saída de uma porta. Esta capacidade não é introduzida deliberadamente, mas é conseqüência da disposição geométrica de qualquer circuito elétrico e aumenta com a capacidade de saída (*fan-out*). Esta capacitância diminui a velocidade de operação da porta, pois o capacitor deve ser carregado e descarregado à medida que a saída sobe e desce ao executar transições entre os níveis lógicos. A descarga é relativamente rápida, pois se dá através da chave fechada, mas a carga é lenta, pois a corrente de carga é suprida através do resistor quando a chave está aberta. Para aumentar a velocidade de carga, o resistor pode ser substituído por uma segunda chave, levando à configuração da Fig. 3.9-2, que é chamada configuração *totem-pole*. Quando A for baixa, S1 estará aberta e S2 estará fechada, fazendo Z alta. Quando A for alta, as chaves invertem seu funcionamento e Z será baixa.

3.10 A SAÍDA DE TRÊS-ESTADOS

Uma modificação útil da saída totem-pole é mostrada na Fig. 3.10-1. Enquanto o terminal de controle, denominado HABILITA (*ENABLE*), estiver no nível lógico 1, o circuito funciona exatamente como o circuito da Fig. 3.9-2, e a variável lógica de entrada A é complementada e o complemento aparece na saída. O circuito está, pois, *habilitado* a executar a função para a qual foi projetado. Quando HABILITA (*ENABLE*) for 0 lógico, as saídas de ambas as portas AND terão nível 0 lógico e as chaves S1 e S2 estarão abertas e o circuito estará *desabilitado*. Nestas condições, a saída, que normalmente conduziria à variável lógica $Z = \bar{A}$, fica totalmente *desligada* do circuito. Esta configuração é conhecida como saída de *três-estados*, dois dos quais se referem aos níveis lógicos e o terceiro correspondendo à condição desligada. Veremos posteriormente as vantagens da saída de três-estados, cujo simbolismo em uma porta é mostrado na Fig. 3.10-2*a*. As Figs. 3.10-2*b* e *c* mostram o simbolismo quando aplicado a um inversor e a um buffer.

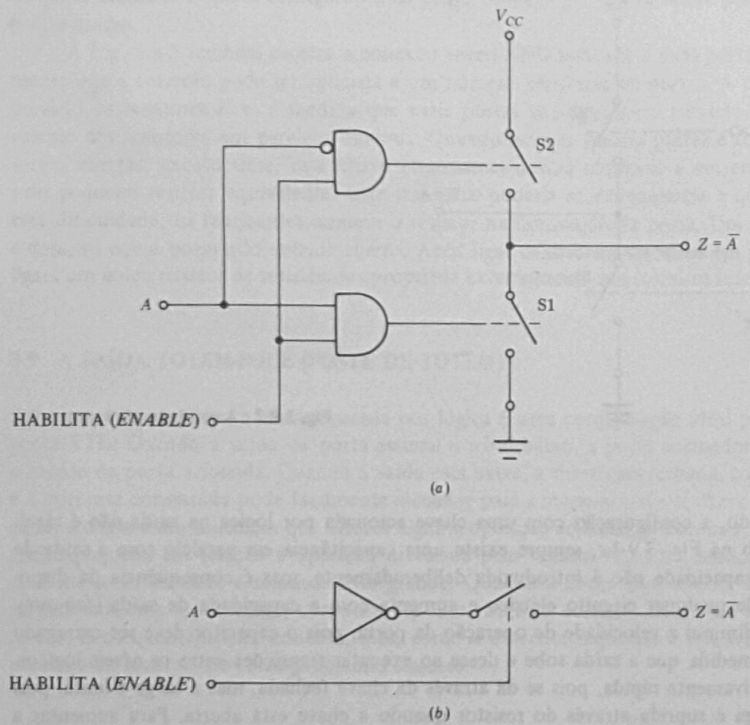


Fig. 3.10-1 (*a*) Saída totem-pole com característica de três-estados. (*b*) Representação equivalente de uma saída de três-estados.

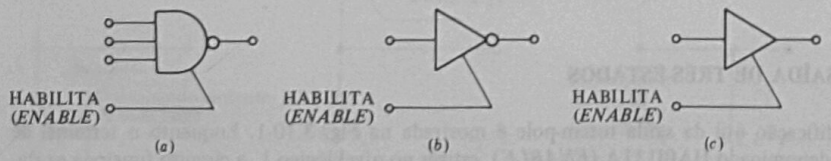


Fig. 3.10-2 Simbolismo para representar uma saída de três-estados em (*a*) uma porta, (*b*) um inversor e (*c*) um buffer.

3.11 EXEMPLOS DE PORTAS LÓGICAS CI

Como exemplos de circuitos integrados contendo portas disponíveis comercialmente, listamos alguns circuitos TTL representativos na Tab. 3.11-1. O apóstrofo antes do número do dispositivo representa a família e a série. Por exemplo, a porta da Texas Instruments SN54LS/74LS02 é

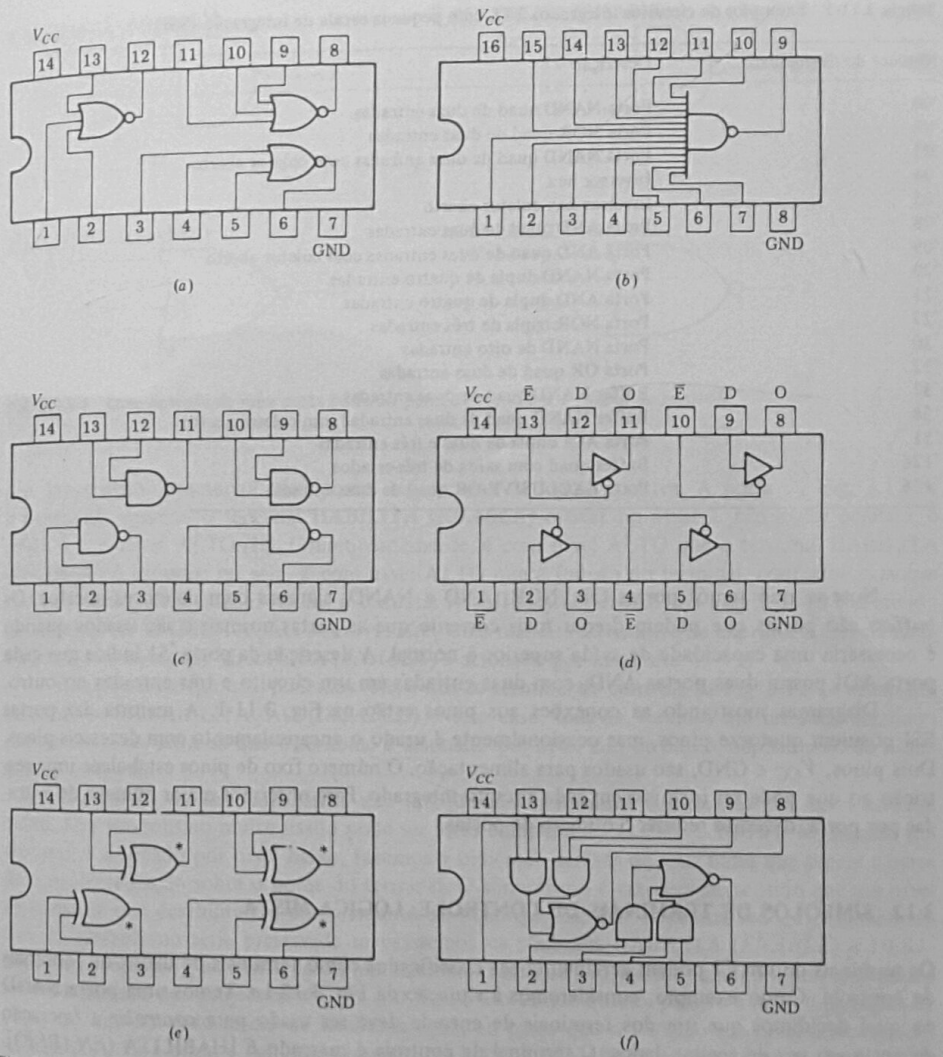


Fig. 3.11-1 Diagramas lógicos de alguns circuitos integrados TTL: (*a*) Porta NOR tripla de três entradas ('27); (*b*) porta NAND de treze entradas ('133); (*c*) porta NAND quad de duas entradas ('00); (*d*) buffer inversor quad com saída de três-estados e HABILITA (*ENABLE*) por nível baixo ('125); (*e*) porta EXCLUSIVE-OR quad ('136) (as estrelas indicam saídas coletor aberto) e (*f*) porta AOI dupla de três e duas entradas.

representada por '02 etc. Os termos "duplo", "triplo", "quad" e "hex" indicam que o circuito integrado contém, respectivamente, duas, três, quatro ou seis unidades idênticas e independentes internamente.

Tabela 3.11-1 Exemplos de circuitos integrados TTL com pequena escala de integração (SSI)

Número do dispositivo	Descrição
'00	Porta NAND quad de duas entradas
'02	Porta NOR quad de duas entradas
'03	Porta NAND quad de duas entradas com coletor aberto
'04	Inversor hex
'05	Inversor hex coletor aberto
'08	Porta AND quad de duas entradas
'09	Porta AND quad de duas entradas com coletor aberto
'20	Porta NAND dupla de quatro entradas
'21	Porta AND dupla de quatro entradas
'27	Porta NOR tripla de três entradas
'30	Porta NAND de oito entradas
'32	Porta OR quad de duas entradas
'37	Buffer NAND quad de duas entradas
'38	Buffer NAND quad de duas entradas com coletor aberto
'51	Porta AOI dupla de duas e três entradas
'126	Buffer quad com saída de três-estados
'136	Porta EXCLUSIVE-OR quad de duas entradas

Note-se que temos portas OR, NOR, AND e NAND, algumas com coletores abertos. Os *buffers* são portas que podem drenar mais corrente que as portas normais e são usados quando é necessária uma capacidade de saída superior à normal. A descrição da porta '51 indica que cada porta AOI possui duas portas AND, com duas entradas em um circuito e três entradas no outro.

Diagramas mostrando as conexões aos pinos estão na Fig. 3.11-1. A maioria das portas SSI possuem quatorze pinos, mas ocasionalmente é usado o encapsulamento com dezesseis pinos. Dois pinos, V_{CC} e GND, são usados para alimentação. O número fixo de pinos estabelece uma restrição ao que pode ser incluído em cada circuito integrado. Para obtermos maior número de entradas por porta, devemos reduzir o número de portas.

3.12 SÍMBOLOS DE TERMINAIS DE CONTROLE: LÓGICA MISTA

Os terminais de um CI podem geralmente ser classificados como terminais de dados ou terminais de controle. Como exemplo, consideremos a situação da Fig. 3.12-1*a*. Temos uma porta NAND na qual decidimos que um dos terminais de entrada deve ser usado para *controlar* a operação da porta em vez de aceitar dados. O terminal de controle é marcado E [HABILITA (*ENABLE*)]. Quando o terminal E estiver no lógico 1, o dispositivo funciona como uma porta NAND de três entradas, isto é, está *habilitado* para executar a função para a qual foi projetado. Quando $E = 0$, todavia, a saída da porta estará no nível lógico 1 (o que é de pouco interesse) e, mais importante, a saída da porta é independente das entradas A, B e C e não responde a suas variações. A porta foi *desabilitada* ou *inibida*. Já vimos um exemplo semelhante de terminal de controle no terminal HABILITA (*ENABLE*) associado à saída de três-estados.

Já observamos que uma mesma configuração física de componentes eletrônicos executa operações lógicas diferentes dependendo da convenção lógica adotada. Presentemente, a maioria

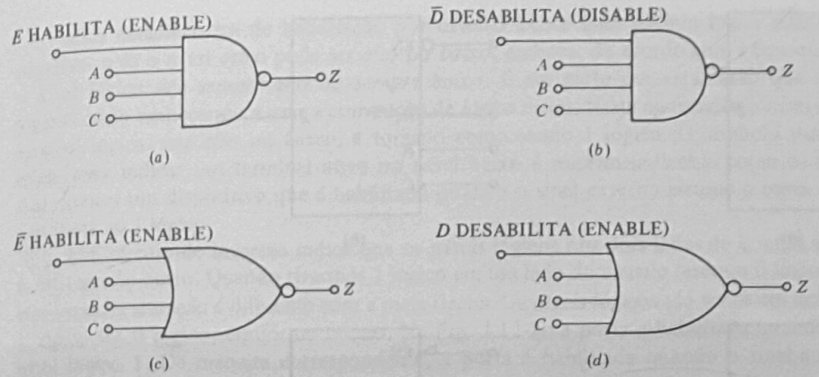


Fig. 3.12-1 Uma entrada de uma porta pode ser pensada como um terminal de *controle*.

dos fabricantes caracteriza suas portas em termos de lógica positiva. A porta da Fig. 3.12-1*a* é habilitada quando o terminal HABILITA (*ENABLE*) estiver no nível 1. Em lógica positiva, o nível 1 é o nível ALTO (H). Conseqüentemente, é com nível ALTO que o terminal HABILITA (*ENABLE*) é *ativado*, ou seja, é com nível ALTO que a função do terminal, *conforme expressa por seu nome*, é executada. Assim, o terminal HABILITA (*ENABLE*) na Fig. 3.12-1*a* é chamado de terminal de HABILITAÇÃO por *nível alto*. Uma maneira alternativa de exprimir a mesma idéia é dizer que o terminal "HABILITA (*ENABLE*) é acionado por nível alto".

Se assim o desejarmos, podemos referir-nos ao terminal de controle da Fig. 3.12-1*a* como um terminal de DESABILITAÇÃO (*DISABLE*). Nesse caso, todavia, teremos um terminal acionado por *nível baixo*. Note-se que o terminal é acionado por nível alto ou baixo, dependendo do nome que dermos a ele.

Precisamos de um simbolismo que indique se um terminal é acionado por nível alto ou baixo. Um simbolismo muito usado pode ser visto comparando as Figs. 3.12-1*a* e *b*. Quando um terminal é acionado por nível baixo, fazemos a indicação através de uma barra que sugere a barra de complementação sobre o nome do terminal. O simbolismo é razoável no sentido que um nível de controle que desabilita é o complemento do nível que habilita. Por outro lado, esta característica do simbolismo seria preservada se usássemos os símbolos HABILITA ($\overline{ENABLE}$) e DESABILITA (*DISABLE*), ao contrário do que fizemos. O fato de não usarmos o simbolismo alternativo reflete a atitude dos projetistas de que o 0 lógico é o estado quieto, passivo e inativo da lógica de controle, não havendo início de nenhuma ação, ao passo que o 1 lógico é o nível ativo. Assim, na Fig. 3.12-1*a* a porta é habilitada quando HABILITA (*ENABLE*) vai para o nível lógico 1. Na Fig. 3.12-1*b* o objetivo do terminal de controle é desabilitar em vez de habilitar. Supomos, então, que existe um nível lógico virtual DESABILITA (*DISABLE*) que assume o valor lógico 1 quando a porta deve ser desabilitada e que o complemento deste nível lógico, isto é, DESABILITA ($\overline{DISABLE}$), é o sinal que realmente é aplicado ao terminal de controle.

A nomenclatura do terminal de controle de uma porta NOR é mostrada nas Figs. 3.12-1*c* e *d*. Se denominado DESABILITA (*DISABLE*), o terminal de controle é ativo no nível alto corresponde ao nível lógico 1 no caso de lógica positiva. Se denominado HABILITA (*ENABLE*), é ativo no nível baixo e recebe o nome de HABILITA ($\overline{ENABLE}$).

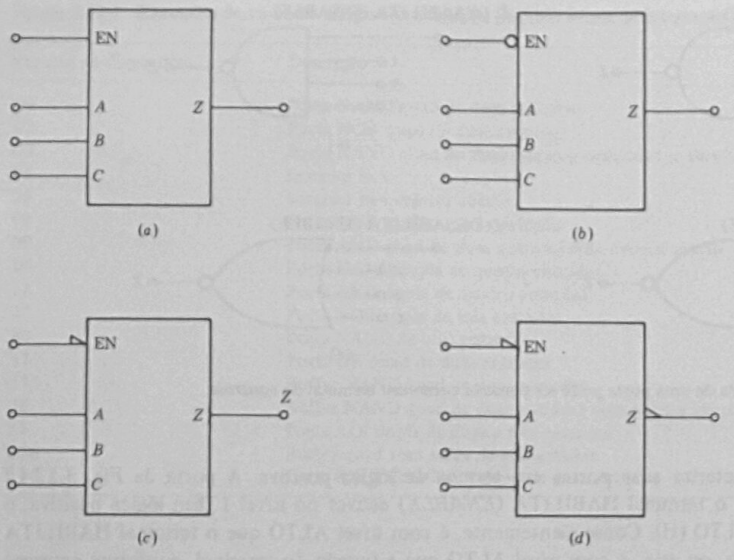


Fig. 3.12-2 Símbolos retangulares para estruturas lógicas: (*a*) entrada habilitadora ativa no nível alto; (*b*) entrada habilitadora ativa no nível baixo; (*c*) entrada habilitadora ativa no nível baixo usando o simbolismo novo e (*d*) habilitação e saída ativas no nível baixo.

Na maioria dos casos, especialmente quando se trata de circuitos MSI e LSI, não é possível nem útil mostrar todas as portas incluídas no circuito, que é, então, representado por um retângulo. Os símbolos retangulares para as portas das Figs. 3.12-1*a* e *c* são mostrados nas Figs. 3.12-2*a* e *b*. Aqui usamos o mnemônico EN, que é universalmente representativo de HABILITA (*ENABLE*). Note-se que todos os nomes são colocados dentro do retângulo. Para indicar que um terminal de controle é ativo no nível baixo, coloca-se um círculo de inversão no ponto em que a linha de controle entra no retângulo. Se tivéssemos chamado o terminal de controle de DIS [DESABILITA (*DISABLE*)] o círculo de inversão apareceria na Fig. 3.12-2*a* e não na Fig. 3.12-2*b*. De qualquer modo, tanto na Fig. 3.12-2*a* como na Fig. 3.12-2*b* o símbolo EN aparece não-complementado. Podemos, pois, supor que em algum lugar no retângulo existe um nível lógico que vai para 1 quando o dispositivo for habilitado. Em geral, é irrelevante e não se pode determinar se isto realmente acontece. A razão do uso do simbolismo é permitir a descrição do funcionamento do dispositivo sem conhecer sua estrutura interna.

Embora o simbolismo descrito seja amplamente usado na literatura e nos catálogos dos fabricantes, ele tem a feliz característica de sugerir que existe uma inversão lógica no caso de um terminal ativo no nível baixo. Como se pode ver na Fig. 3.12-1*c*, esta inversão não ocorre. Também, no caso de um terminal ativo no nível baixo, embora seja possível supor que em algum lugar existe uma variável que assume nível lógico 1 para efetuar a habilitação, na realidade *o sinal aplicado ao terminal externo assume nível 0*.

Esta característica de habilitação por nível 0 existe quer usemos lógica positiva ou lógica negativa, pois o nível *ativo* pode ser *alto* ou *baixo*, embora, de acordo com a convenção adotada, o nível *lógico* seja *sempre alto* ou *sempre baixo*. É em parte por esta razão que os projetistas lógicos estão começando a usar a convenção de *lógica mista*. Nesta convenção o nível *ativo* de uma variável lógica, seja *alto* ou *baixo*, é tomado como sendo 1 lógico. O símbolo usado na lógica mista para indicar um terminal ativo no nível baixo é uma meia-flecha, como na Fig. 3.12-2*c*. Aqui temos um dispositivo que é habilitado quando o sinal externo assume o nível *baixo* correspondente ao 1 lógico.

Um círculo de inversão indica que os níveis lógicos nos dois lados do círculo são o complemento um do outro. Quando tivermos 1 lógico em um lado do círculo teremos 0 lógico no outro, e vice-versa. A situação é diferente com a meia-flecha. Os níveis lógicos são *iguais* em ambos os lados, 1 lógico ou 0 lógico, conforme o caso. Na Fig. 3.12-2*c* a porta é habilitada quando EN assume nível lógico 1. De maneira correspondente, a porta é habilitada quando o sinal aplicado externamente assumir o nível lógico 1. Agora existe concordância entre os níveis lógicos internos e externos nos quais o circuito é habilitado ou desabilitado. Assim podemos preservar nosso preconceito de que a habilitação (ou qualquer outra função que um terminal de controle deva executar) é executada no nível lógico 1. Da Fig. 3.12-2*c* não podemos determinar o que acontece dentro do circuito. Não sabemos que sinal assume que nível de tensão para habilitar o circuito, mas o simbolismo nos diz que, para habilitar o circuito, é necessário que o sinal aplicado externamente assuma o nível de tensão *baixo*, isto é, HABILITA (*ENABLE*) é *acionado baixo*, sendo que este nível de tensão baixo equivale ao nível lógico 1.

A convenção de lógica mista não oferece vantagens quando nosso interesse é nos aspectos teóricos do projeto de circuitos lógicos, mas, como veremos, é muito útil quando tratamos com sistemas e dispositivos reais caracterizados em termos de atividade de terminais que respondem a tensões reais altas ou baixas.

Consideremos, finalmente, a possibilidade de a saída Z, digamos, de uma porta NAND como a da Fig. 3.12-1*a*, ser aplicada ao terminal de entrada de controle de outro circuito. Especificamente, vamos supor que o propósito da porta NAND seja prover um sinal de controle que acione o outro circuito quando todas as três entradas A, B e C forem altas. Então a saída Z é um sinal de controle que estará no nível que permite acionamento quando for baixa. Neste caso o símbolo retangular apropriado é o da Fig. 3.12-1*d*. Novamente a meia-flecha na saída Z indica que o sinal é *ativo quando baixo*. Verificamos, ainda, uma segunda vantagem da flecha, pois sua direção indica se o terminal é de entrada ou de saída. Generalizando, a qualquer terminal onde é necessário um sinal de controle de entrada ativo quando baixo ou onde tal sinal se origina adicionamos o símbolo da meia-flecha, cuja direção distingue terminais de entrada de terminais de saída.

No início de nossa discussão distinguimos entre terminais de controle e outros terminais que denominamos de terminais de dados. Na realidade, esta distinção é artificial e sempre podemos pensar em um terminal de entrada ou de saída de uma porta ou de um circuito integrado como tendo uma função determinada que pode ser exercida individualmente ou em conjunto com outros terminais. Assim, os comentários que fizemos a respeito dos terminais de controle aplicam-se igualmente a todos os terminais de uma porta ou de um circuito integrado. Geralmente, é útil e até necessário dar a cada terminal um nome funcionalmente descritivo e indicar se o terminal é ativo quando baixo ou quando alto. Na maioria dos casos uma descrição funcional implicaria o uso de muitas palavras, sendo costume abreviar tal descrição através do uso de mnemônicos curtos. O símbolo mais moderno e mais aceito para indicar que um terminal é ativo quando baixo é a meia-flecha, embora muitos autores continuem usando o círculo de inversão.

3.13 SÍMBOLOS DE SINAIS DE CONTROLE

Quando temos um CI com um ou mais terminais de controle, devemos gerar sinais de controle apropriados para aplicar a estes terminais de controle. Como estes sinais de controle executam transições de um nível para outro ao longo do tempo, eles são funções do tempo. Em ocasiões anteriores usamos "função" para indicar que o valor lógico de uma variável era função do valor lógico de outras variáveis. Aqui usamos "função" para indicar a dependência temporal de uma variável lógica.

Uma função de controle típica aparece na Fig. 3.13-1. Alguns sinais de controle têm uma dependência do tempo que é perfeitamente previsível, como, por exemplo, os sinais de sincronismo que descreveremos no Cap. 4. Outros sinais não têm uma dependência previsível, como, por exemplo, os sinais de controle gerados em uma calculadora, pois dependem de quais teclas o usuário aciona e quando elas são acionadas. Em todo o caso vamos supor que um circuito deva ser habilitado em certos momentos e sob certas condições. Vamos supor, ainda, que seja gerado um sinal de controle que atinja um determinado nível lógico (ainda não especificado como alto ou baixo) quando o circuito deva ser habilitado, permanecendo no outro nível durante o resto do tempo. Seria, então, razoável darmos a este sinal de controle o nome HABILITA (*ENABLE*). Mais especificamente, se o sinal for ALTO quando o circuito necessitar habilitação, a convenção é que o sinal seja chamado HABILITA (*ENABLE*) [se for BAIXO, o nome será $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$)]. Se o *terminal* de habilitação for ativo quando alto, o sinal HABILITA (*ENABLE*) pode ser ligado a ele diretamente. Se o terminal for ativo quando baixo, um sinal $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$) pode ser ligado a ele diretamente. Caso contrário devemos interpor um inversor entre a fonte de sinal e o terminal.

As terminologias HABILITA (*ENABLE*) e $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$) têm uma limitação inconveniente. Suponhamos que no diagrama do circuito lógico de um sistema digital haja uma linha contendo um sinal denominado $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$) e um terminal de controle ativo quando baixo. Podemos presumir que o sinal é apropriado para aplicação ao terminal. Se o terminal não for evidente, todavia, não saberemos se $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$) é um sinal habilitador ativo quando baixo ou o complemento de um sinal habilitador ativo quando alto. Assim, se o sinal da Fig. 3.13-1 fosse chamado $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$), a não ser que conhecêssemos a denominação do

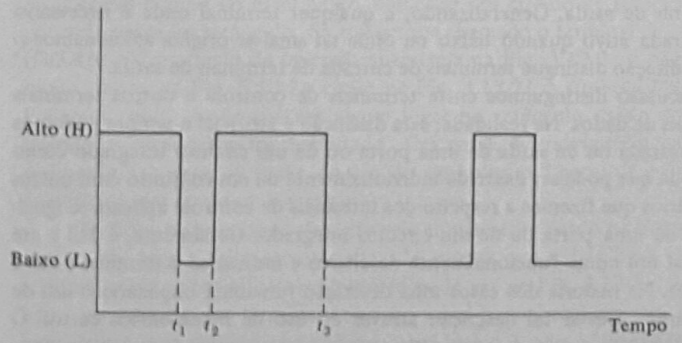


Fig. 3.13-1 Um sinal de controle arbitrário.

terminal no circuito controlado, não saberíamos se o circuito deve ser habilitado durante o intervalo t_1 a t_2 ou durante o intervalo t_2 a t_3.

Para contornar esta desvantagem, a nova terminologia substitui HABILITA (*ENABLE*)-H e HABILITA (*ENABLE*)-L por HABILITA (*ENABLE*) e $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$). HABILITA (*ENABLE*)-H é um sinal ativo quando alto (*high*) e HABILITA (*ENABLE*)-L é um sinal ativo quando baixo (*low*). Se o sinal da Fig. 3.13-1 for denominado, por exemplo, HABILITA (*ENABLE*)-L, saberemos que ele habilita durante o intervalo t_1 a t_2. É claro que, se o circuito a ser habilitado possuir um terminal de controle ativo quando alto, será necessário inverter o sinal HABILITA (*ENABLE*)-L antes de ligá-lo ao circuito.

Um sinal HABILITA (*ENABLE*) seria apropriado para ligação a um dispositivo como o da Fig. 3.12-2*a*, e $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$) seria apropriado para o dispositivo representado em *b*. Aqui o simbolismo sugere que $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$), cujo nível ativo é baixo, isto é, 0 lógico, é aplicado a um inversor e aparece dentro do dispositivo como HABILITA (*ENABLE*) que habilita o circuito ao passar para o nível 1. Quando a convenção de lógica mista é usada, a terminologia para caracterizar os sinais é HABILITA (*ENABLE*)-H e HABILITA (*ENABLE*)-L. O sinal HABILITA (*ENABLE*)-H é ativo e tem nível 1 quando for alto, e o sinal HABILITA (*ENABLE*)-L é ativo e tem nível 1 quando for baixo. Se tivermos um sinal HABILITA (*ENABLE*)-H e o terminal de controle que requer o sinal habilitador for marcado com uma meia-flecha, será necessário interpor um inversor para inverter os níveis de tensão mas não para complementar os níveis lógicos.

Um sinal HABILITA (*ENABLE*)-H é um sinal que desabilita quando for baixo, ou seja, é DESABILITA (*DISABLE*)-L. Como DESABILITA (*DISABLE*) = $\overline{\text{HABILITA}}$ ($\overline{\textit{ENABLE}}$), temos

$$\text{HABILITA } (\textit{ENABLE})\text{-H} = \overline{\text{HABILITA}}\ (\overline{\textit{ENABLE}})\text{-L} \tag{3.13-1}$$

e

$$\overline{\text{HABILITA}}\ (\overline{\textit{ENABLE}})\text{-H} = \text{HABILITA } (\textit{ENABLE})\text{-L} \tag{3.13-2}$$

Por outro lado, um sinal que está em determinado nível sempre que a habilitação é necessária, e no outro nível em todas as outras ocasiões, é um sinal habilitador mesmo que seu nível seja sempre o *nível errado*. Isto porque este sinal faz uma *distinção correta entre os tempos* em que a habilitação é necessária e aqueles em que não é necessária. Mesmo se invertermos um sinal HABILITA (*ENABLE*)-H, ainda teremos um sinal HABILITA (*ENABLE*); como o sinal está no nível baixo quando deve ser ativo, temos

$$\overline{\text{HABILITA } (\textit{ENABLE})\text{-H}} = \text{HABILITA } (\textit{ENABLE})\text{-L} \tag{3.13-3}$$

e

$$\overline{\text{HABILITA } (\textit{ENABLE})\text{-L}} = \text{HABILITA } (\textit{ENABLE})\text{-H} \tag{3.13-4}$$

Resumindo, temos

$$\text{HABILITA } (\textit{ENABLE})\text{-H} = \overline{\text{HABILITA}}\ (\overline{\textit{ENABLE}})\text{-L} = \overline{\text{HABILITA}}\ (\overline{\textit{ENABLE}})\text{-L} \tag{3.13-5}$$

e

$$\text{HABILITA } (\textit{ENABLE})\text{-L} = \overline{\text{HABILITA}}\ (\overline{\textit{ENABLE}})\text{-H} = \overline{\text{HABILITA}}\ (\overline{\textit{ENABLE}})\text{-H} \tag{3.13-6}$$

3.14 LÓGICA MISTA APLICADA A ESTRUTURAS COM PORTAS

Tanto em lógica positiva como em lógica negativa, uma porta AND gera uma saída com nível lógico 1 quando todas as entradas estiverem no nível lógico 1, e uma porta OR gera esta mesma

A	B	Z
Inativo	Inativo	Inativo
Inativo	Ativo	Inativo
Ativo	Inativo	Inativo
Ativo	Ativo	Ativo

(*a*)

A	B	Z
Inativo	Inativo	Inativo
Inativo	Ativo	Ativo
Ativo	Inativo	Ativo
Ativo	Ativo	Ativo

(*b*)

Fig. 3.14-1 Tabelas verdade com lógica mista para (*a*) função AND, $Z = AB$ e (*b*) função OR, $Z = A + B$.

saída quando uma ou mais entradas estiverem em 1. Em lógica mista, o 1 lógico não é associado aos níveis ALTO ou BAIXO de tensão, mas à *atividade*. Assim, em lógica mista as funções AND e OR podem ser definidas pelas tabelas verdade das Figs. 3.14-1*a* e *b*. Uma porta AND gera um nível ativo somente quando ambas as entradas estiverem em níveis ativos. É claro que, independentemente da convenção lógica usada, todas as estruturas de portas que tenham um mesmo propósito específico devem ser equivalentes. Por outro lado, como os exemplos seguintes mostram, os processos usados para chegar-se à estrutura de portas desejada são diferentes. Vamos supor que estão disponíveis dois sinais de controle, ambos ativos quando baixos, ALFA-L e BETA-L, e que necessitamos de um sinal ativo quando alto, GAMA-H, que seja ativo somente quando ALFA-L *e* BETA-L forem ativos. Seguindo a convenção normal de lógica positiva, podemos preparar tabelas verdade como as da Fig. 3.14-2. Na Fig. 3.14-2*a* a tabela é construída em termos de atividade e especifica que GAMA é ativo somente quando ALFA e BETA são ativos. Na Fig. 3.14-2*b* inserimos 1s e 0s usando a convenção de lógica positiva. Como ALFA-L é ativo quando baixo, o estado inativo alto é 1 lógico etc. Da Fig. 3.14-2*b* deduzimos que a lógica requerida é a lógica fornecida por uma porta NOR. Como usamos lógica positiva para preencher a Fig. 3.14-2*b*, necessitamos de uma porta NOR de lógica positiva como as que existem em um CI '02.

Alternativamente, com lógica mista procedemos da seguinte maneira. Deseja-se uma lógica que gere uma saída ativa, nível lógico 1, somente quando as duas entradas forem ativas, isto é, tiverem nível lógico 1. Esta especificação requer uma porta AND, como a da Fig. 3.14-3*a*. A seguir notamos que as entradas são ativas quando baixas. Um sinal ativo quando baixo só é efetivo quando aplicado a um terminal ativo quando baixo. Assim, como na Fig. 3.14-3*b*, colocamos as meias-flechas nas entradas. Uma meia-flecha em uma porta de um sistema de lógica mista

ALFA-L	BETA-L	GAMA-H
Inativo	Inativo	Inativo
Inativo	Ativo	Inativo
Ativo	Inativo	Inativo
Ativo	Ativo	Ativo

(*a*)

ALFA-L	BETA-L	GAMA-H
1	1	0
1	0	0
0	1	0
0	0	1

(*b*)

Fig. 3.14-2 (*a*) Tabela verdade construída em termos dos níveis de atividade requeridos das entradas e saída. (*b*) A tabela com 1s e 0s substituindo os níveis de atividade e supondo a convenção de lógica positiva.

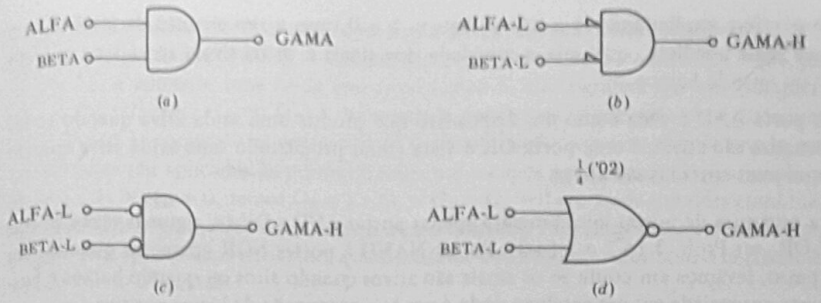


Fig. 3.14-3 Um exemplo de desenvolvimento de uma estrutura com lógica mista.

equivale a um círculo de inversão em um sistema de lógica positiva. Somos, portanto, levados à estrutura de lógica positiva da Fig. 3.14-3*c*. Devemos encontrar entre os CI disponíveis comercialmente um CI cujas portas sejam equivalentes à porta da Fig. 3.14-3*c*. Usando o teorema de De Morgan, chegamos novamente à porta NOR, como na Fig. 3.14-3*d*. A porta requerida é uma das quatro existentes no CI tipo '02. É claro que nem todos esses passos são realmente necessários. Se a discussão não servisse como exemplo ilustrativo, poderíamos ter começado com a Fig. 3.14-3*b* e chegado diretamente à Fig. 3.14-3*d*.

Como segundo exemplo, vamos supor que temos os sinais ALFA-L, BETA-L e GAMA-H e que precisamos gerar um sinal DELTA-L. É necessário que DELTA seja ativo quando GAMA for ativo *e* ALFA *ou* BETA for ativo. Inicialmente ignoramos as especificações relativas a ser um sinal ativo quando alto ou quando baixo. As palavras em itálico *e* e *ou* na frase que especifica a lógica requerem uma porta AND e uma porta OR, levando diretamente à configuração da Fig. 3.14-4*a*. Levando em conta os níveis de atividade específicos, chegamos ao circuito da Fig. 3.14-4*b*. Aplicando o teorema de De Morgan, notamos que ambas as portas são portas NAND de lógica positiva e que (ver Fig. 3.11-1*c*) a estrutura usa duas das portas de um CI tipo '00.

O processo para lógica mista usado nesses dois exemplos para deduzir a estrutura de portas necessária tem as seguintes características:

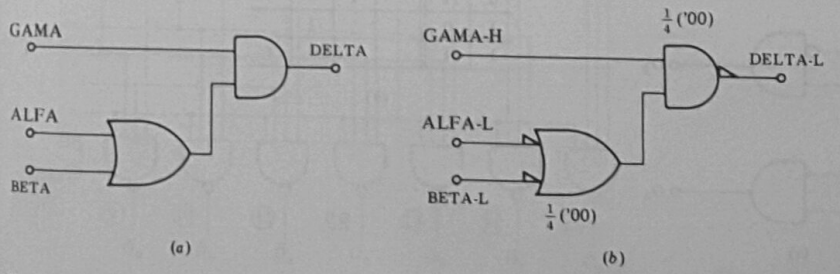


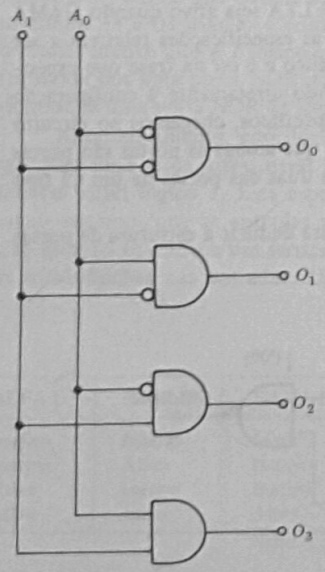
Fig. 3.14-4 Um exemplo.

1. Não se refere explicitamente a níveis lógicos 1 e 0, mas a um assunto de importância prática mais imediata, qual seja, a atividade dos sinais e se os sinais são ativos quando altos ou quando baixos.
2. Uma porta AND é vista como um dispositivo que produz uma saída ativa quando todas as entradas são ativas, e uma porta OR é vista como produzindo uma saída ativa quando uma ou mais entradas são ativas.

Geralmente, a estrutura de portas inicial mostra apenas portas AND e OR (e, algumas vezes, portas EXCLUSIVE-OR; ver Prob. 3.14-2 e 3.14-3). Portas NAND e portas NOR aparecem quando, em um segundo passo, levamos em conta se os sinais são ativos quando altos ou quando baixos e buscamos uma porta apropriada em um catálogo onde é usada a convenção de lógica positiva.

3.15 DECODIFICADORES

Um número de estruturas com portas envolvendo portas múltiplas encontra aplicação tão vasta que existe sob a forma de circuitos integrados. O número de portas incluídas nestes circuitos faz com que sejam classificados como circuitos com integração em média escala (MSI). Nesta seção discutiremos uma destas estruturas, chamada *decodificador*; nas seções seguintes discutiremos outras estruturas muito usadas.



(*a*)

A_1	A_0	O_0	O_1	O_2	O_3
0	0	1			
0	1		1		
1	0			1	
1	1				1

(*b*)

Fig. 3.15-1 Um decodificador: (*a*) estrutura de portas e (*b*) tabela verdade.

Um decodificador é mostrado na Fig. 3.15-1*a*; em sua tabela verdade (Fig. 3.15-1*b*) todos os valores não preenchidos são 0. A característica importante do decodificador é que, para cada entrada A_1A_0, e somente uma saída tem nível lógico 1. Isto significa que em qualquer instante uma das saídas (O_0, O_1, O_2 ou O_3) é diferenciada das outras. Neste caso ela é diferenciada por ser de nível lógico 1, enquanto as outras são 0. O decodificador consiste simplesmente em portas AND. As entradas são aplicadas às portas diretamente ou após uma inversão lógica, conforme necessário. Quando $A_1 = A_0 = 0$, temos $O_0 = 1$ e $O_1 = O_2 = O_3 = 0$ etc. Cada uma das combinações de entradas possíveis para A_0 e A_1 individualiza uma das saídas. Para simplificar o desenho, o nosso decodificador tem apenas duas entradas e quatro saídas. Um decodificador com n entradas necessitaria 2^n portas e teria 2^n saídas.

Para entender por que o arranjo da Fig. 3.15-1 é chamado de decodificador, consideremos o seguinte. Vamos supor que a saída de cada uma das portas AND fosse ligada a um circuito elétrico que acendesse uma lâmpada quando a saída da porta fosse de nível lógico 1. Imaginemos, ainda, que cada lâmpada iluminasse um cartão com os números de 0 a 3 que não são visíveis quando a lâmpada correspondente está apagada. O efeito deste conjunto seria tornar *explícito* o que só foi apresentado implicitamente. Dados os níveis lógicos das entradas, podemos determinar qual o número representado, mas, se olharmos para a saída do sistema, o número representado é explicitamente mostrado, isto é, a informação de entrada, recebida como uma palavra de código, é decodificada.

Se for apropriado, não é necessário apresentar o conjunto completo de 2^n saídas. Se, por exemplo, as entradas forem quatro linhas usadas para representar os dígitos decimais de 0 a 9 na forma decimal codificado em binário (BCD), então somente dez lâmpadas serão necessárias e o

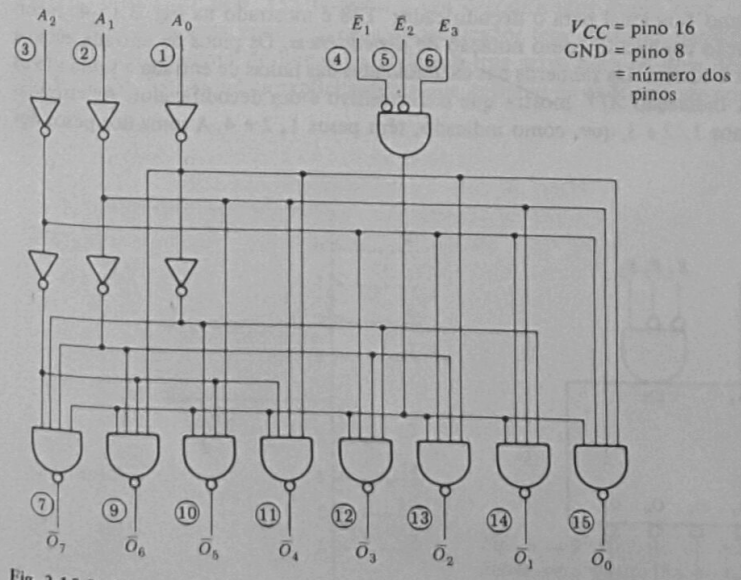


Fig. 3.15-2 Diagrama lógico do decodificador tipo '138.

decodificador deve apresentar somente dez linhas de saída, em vez do número total possível de $2^4 = 16$. Neste caso só serão necessárias dez portas AND e nem todas as portas precisam ter quatro entradas. Isto ocorre porque há condições não-essenciais que correspondem aos números de 10 a 15, que nunca são apresentados à entrada. Por outro lado, um decodificador que deve fornecer m saídas individualmente selecionáveis deve ter $2^n \geq m$.

O símbolo A para as entradas (A_1, A_0) do decodificador foi escolhido para sugerir que estes bits de entrada coletivamente constituem um *endereço*, isto é, assim como um endereço seleciona uma única casa dentre as muitas existentes em uma cidade, o *endereço* de entrada A_1A_0 seleciona uma linha de saída dentre muitas. Freqüentemente, teremos ocasião para referirmo-nos a este conceito de endereçamento.

O CI decodificador MSI tipo '138 mostrado na Fig. 3.15-2 é fabricado em um invólucro de dezesseis pinos e os números dentro dos círculos representam o número dos pinos. Este circuito é uma unidade de três entradas e oito saídas. Como usa portas NAND em vez de portas AND, as saídas são ativas quando baixas ao invés de ativas quando altas, como na Fig. 3.15-1. Entradas que não requerem inversão lógica passam através de dois inversores. Estes inversores não são necessários para fins lógicos, mas são usados para promover isolação que reduz a carga imposta pela unidade à fonte dos bits de endereçamento de entrada. Cada porta tem quatro entradas em vez das três necessárias para acomodar os bits de endereçamento. As entradas extras são ligadas juntas para proporcionar uma entrada de controle HABILITA (*ENABLE*). Há três entradas HABILITA (*ENABLE*), $\overline{E}_1$, $\overline{E}_2$ e E_3. Podemos usar estas entradas para obter um controle ativo quando alto ou ativo quando baixo ou, ainda, de forma mais genérica, para permitir a habilitação somente quando os três sinais tiverem os níveis lógicos apropriados simultaneamente. O símbolo lógico do dispositivo é mostrado na Fig. 3.15-3. Outros decodificadores, tal como o decodificador de 1-de-4 duplo '155, são também acessíveis.

O novo simbolismo funcional para o decodificador '138 é mostrado na Fig. 3.15-4. Incorpora um tipo de notação conhecido como notação de *dependência*. Os pinos de entrada estão à esquerda e os de saída à direita. Os números nas extremidades das linhas de entrada e saída são os números dos pinos. A indicação X/Y mostra que o dispositivo é um decodificador. As entradas de endereço são os pinos 1, 2 e 3, que, como indicado, têm pesos 1, 2 e 4. A soma dos pesos dos

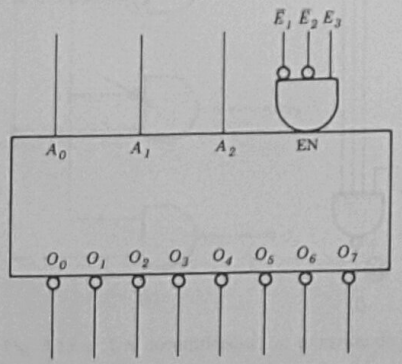


Fig. 3.15-3 Símbolo lógico do decodificador tipo '138.

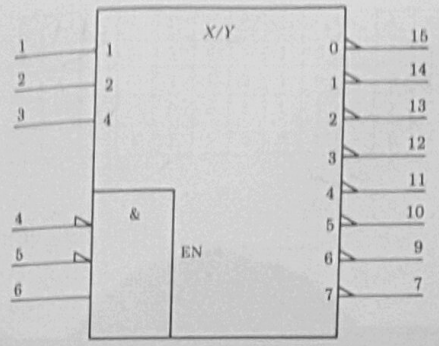


Fig. 3.15-4 O novo simbolismo funcional para o decodificador tipo '138.

pinos de entrada ativos determina o número da linha de saída que é ativada. Assim, se os pinos 1 e 3, que têm pesos 1 e 4, respectivamente, forem ativados, a linha de saída 5 (1 + 4), identificada como pino 10, será ativada. As saídas, como indicado, são ativas quando baixas. O pequeno retângulo marcado & e suas entradas associadas e a denominação EN indicam que as linhas de entrada 4, 5 e 6 são combinadas em uma porta AND para gerar um sinal de controle habilitador interno, que as linhas 4 e 5 são ativas quando baixas e que a linha 6 é ativa quando alta.

O tipo '139 é um decodificador duplo com duas linhas de entrada e quatro linhas de saída no qual os dois decodificadores são independentes. Sua nova representação simbólica é mostrada na Fig. 3.15-5. O tipo '155 também é um decodificador duplo com duas linhas de entrada e quatro de saída, mas, enquanto cada decodificador tem controles habilitadores separados, existe somente um conjunto de linhas de endereço que serve para os dois decodificadores. O simbolismo da Fig. 3.15-6 indica explicitamente que as linhas de endereço são compartilhadas.

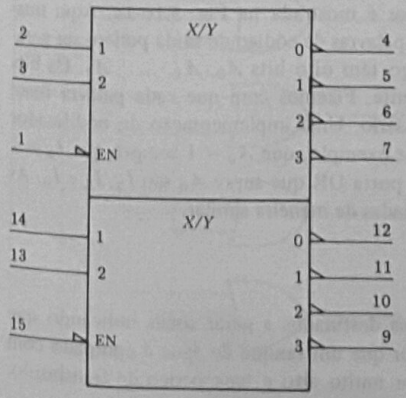


Fig. 3.15-5 Símbolo do decodificador duplo de duas linhas para quatro linhas tipo '139. As unidades são independentes.

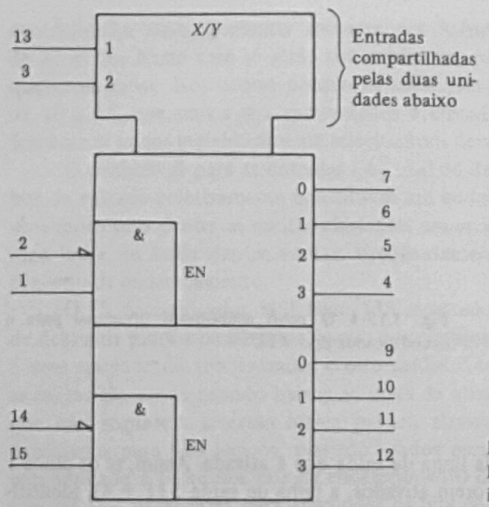


Fig. 3.15-6 Símbolo do decodificador duplo de duas linhas para quatro linhas tipo 155. O simbolismo indica que as entradas são compartilhadas.

3.16 CODIFICADORES

Um *codificador* executa uma função que é inversa à do decodificador. Um codificador opera com entradas que têm a característica de, em qualquer momento, uma ser escolhida para ter um nível lógico diferente de todas as outras. (As entradas do codificador muitas vezes são as saídas de um decodificador.) Para cada linha de entrada escolhida, uma *palavra de código* correspondente, com bits $A_0, A_1, \ldots,$ aparece nas linhas de saída. Em geral, não é necessário que exista uma relação especial entre o número de linhas de entrada e o número de linhas de saída.

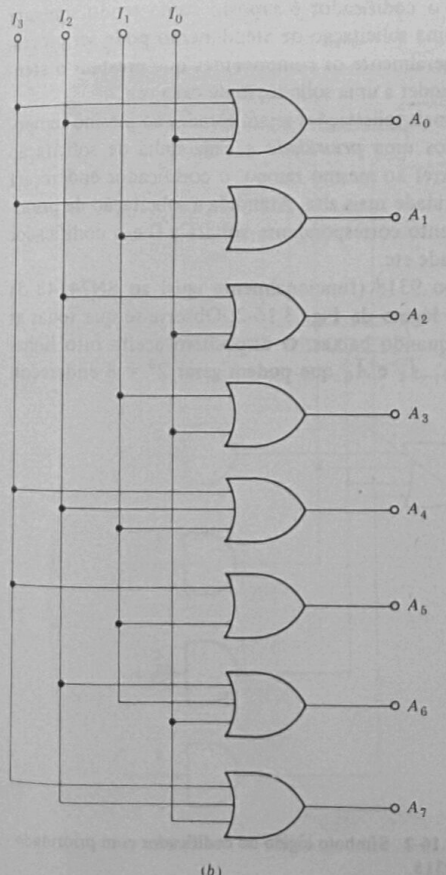
Uma possível tabela verdade para um codificador é mostrada na Fig. 3.16-1*a*. Aqui imaginamos quatro linhas de entrada, de modo que quatro palavras de código de saída podem ser geradas. Fizemos a suposição de que as palavras de código têm oito bits $A_0, A_1, \ldots, A_7$. Os bits das palavras de código foram escolhidos arbitrariamente. Fizemos com que cada palavra fosse diferente, embora, em princípio, isto não fosse necessário. Uma implementação do codificador com portas é mostrada na Fig. 3.16-1*b*. Notamos, por exemplo, que $A_0 = 1$ sempre que $I_3 = 1$ ou que $I_2 = 1$ ou que $I_0 = 1$. Assim, as entradas da porta OR que supre A_0 são I_3, I_2 e I_0. As entradas apropriadas para as demais portas são determinadas de maneira similar.

Codificador com Prioridade

Sistemas digitais freqüentemente incluem componentes destinados a gerar sinais indicando que alguma ação é necessária. Por exemplo, podemos supor que um tanque de água é equipado com uma chave que é operada quando o nível da água for muito alto e haja perigo de transbordo.

I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1				1	0	1	1	0	0	1	1
	1			1	1	0	1	0	1	0	1
		1		0	1	1	1	1	0	1	0
			1	1	1	0	0	1	1	0	1

(*a*)



(*b*)

Fig. 3.16-1 (*a*) Uma possível tabela verdade para um codificador (0s devem ser imaginados nos quadrículos vazios). (*b*) Uma implementação com portas da tabela verdade de (*a*).

Podemos facilmente criar uma linha que normalmente esteja em nível lógico 0 e que passe para 1 quando a chave for atuada. A mudança de nível na linha de nível lógico 0 para 1 é uma indicação de que algo deve ser feito, ou seja, é uma *solicitação de atendimento*. Este mesmo sistema pode ter componentes que geram a resposta desejada, ou seja, *atendem a solicitação*. O atendimento no caso do tanque de água pode consistir no fechamento da válvula de entrada, abertura da válvula de saída etc. Freqüentemente, em sistemas digitais há um número de linhas que indicam solicitações de atendimento e, correspondentemente, um número de componentes que provêem o atendimento. O componente encarregado do atendimento que é ativado depende naturalmente da linha que apresenta a solicitação de atendimento. Em geral, cada componente encarregado do atendimento é diferenciado dos outros componentes do mesmo tipo por um endereço, ou seja, um conjunto distinto de bits (um código de endereço) pelo qual o componente é acessado através de um decodificador.

Um codificador normalmente seria usado para aceitar como entradas as linhas de solicitação de atendimento e apresentar como saídas o código de bits que constitui o endereço do componente que atenderia a solicitação. Como o codificador é suposto como tendo somente uma linha identificada a cada instante, somente uma solicitação de atendimento pode ser processada de cada vez. Esta situação é aceitável, pois geralmente os componentes que prestam o atendimento têm elementos em comum e só podem atender a uma solicitação de cada vez.

Suponhamos, por outro lado, que duas ou mais solicitações sejam geradas ao mesmo tempo. Esta situação pode ser contornada se atribuirmos uma *prioridade* a cada linha de solicitação de atendimento. Se mais de uma solicitação ocorrer ao mesmo tempo, o codificador endereçará o componente que atende a solicitação com prioridade mais alta. Atendida a solicitação de prioridade mais alta, a linha de solicitação de atendimento correspondente voltará a 0 e o codificador atenderá a solicitação seguinte de mais alta prioridade etc.

O codificador com prioridade Fairchild tipo 9318 (funcionalmente igual ao SN74148 da Texas Instruments) é representado pelo símbolo lógico da Fig. 3.16-2. Observe-se que todas as entradas, saídas e linhas de controle são ativas quando baixas. O dispositivo aceita oito linhas de entrada $\overline{I}_7, \overline{I}_6, \ldots, \overline{I}_0$ e provê três saídas $\overline{A}_2$, $\overline{A}_1$ e $\overline{A}_0$ que podem gerar $2^3 = 8$ endereços.

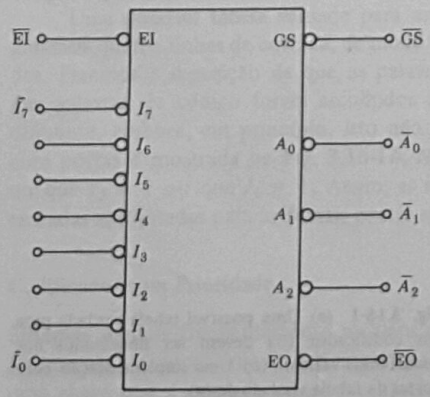
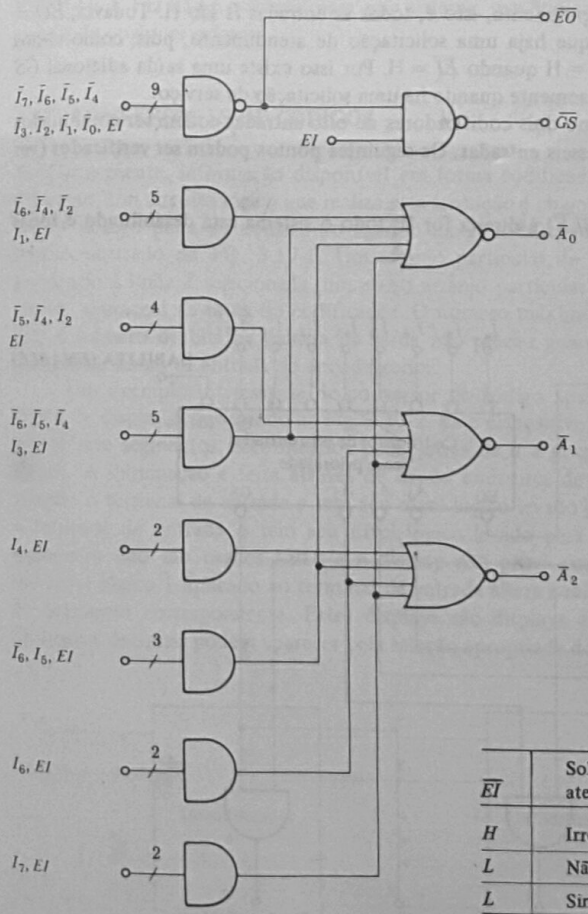


Fig. 3.16-2 Símbolo lógico do codificador com prioridade tipo 9318.

$\overline{EI}$ é uma entrada habilitadora (*ENABLE*) e o propósito das saídas EO e GS será considerado em seguida.

A lógica do codificador é mostrada na Fig. 3.16-3. As portas AND requerem entradas múltiplas. Para evitar a complicação do desenho, representamos as entradas múltiplas por uma única linha cruzada por uma barra inclinada com o número de entradas escrito ao lado. Como todas as entradas são ativas quando baixas, os *I*s de entrada são complementados $\overline{I}_0, \overline{I}_1$ etc. Onde aparece um *I* não-complementado ou um *EI* não-complementado, foi usado um inversor, que não é mostrado para simplificar o desenho. As entradas apresentadas a cada porta AND, quer venham direta-



(a)

$\overline{EI}$	Solicitação de atendimento?	$\overline{EO}$	$\overline{GS}$
H	Irrelevante	*H*	*H*
L	Não	*L*	*H*
L	Sim	*H*	*L*

(b)

Fig. 3.16-3 (*a*) Diagrama lógico do codificador com prioridade tipo 9318. (*b*) Tabela verdade para EO e GS.

mente do mundo exterior, quer sejam geradas após inversão no próprio circuito, são listadas ao lado de cada porta.

O codificador é habilitado quando a habilitação de entrada $\overline{EI}$ = L (baixo) e fica desabilitado quando $\overline{EI}$ = H (alto). Quando desabilitado, o codificador apresenta todas as saídas com nível alto independentemente das entradas. Como pode ser verificado (Prob. 3.16-2), a prioridade mais alta é atribuída a I_7 e a mais baixa a I_0. Quando o codificador está habilitado e $\overline{I}_7 =$ = L, a saída é $\overline{A}_2\overline{A}_1\overline{A}_0$ = LLL = 7, independentemente dos valores dos outros $\overline{I}$s. Quando $\overline{I}_6 =$ = L, $\overline{A}_2\overline{A}_1\overline{A}_0$ = LLH = 6 desde que I_7 *não* seja L e independentemente dos outros $\overline{I}$, e assim por diante. Quando duas ou mais entradas $\overline{I}$s forem baixas simultaneamente, a entrada com prioridade mais alta é codificada e as demais são ignoradas. Quando a saída $\overline{EO}$ for $\overline{EO}$ = L, o codificador indica que *não* há solicitação de atendimento, isto é, todas as entradas $\overline{I}$s são H. Todavia, $\overline{EO}$ = = H não significa necessariamente que haja uma solicitação de atendimento, pois, como vimos na Fig. 3.16-3*b*, sempre temos $\overline{EO}$ = H quando $\overline{EI}$ = H. Por isso existe uma saída adicional $\overline{GS}$ (sinal de grupo) que tem nível baixo somente quando há uma solicitação de serviço.

Na Fig. 3.16-4 mostramos como dois codificadores de oito entradas podem ser combinados para formar um codificador de dezesseis entradas. Os seguintes pontos podem ser verificados (ver Prob. 3.16-3).

1. Quando $\overline{\text{HABILITA}}$ (*ENABLE*) à direita for H, todo o sistema está desabilitado e todas as saídas são H.

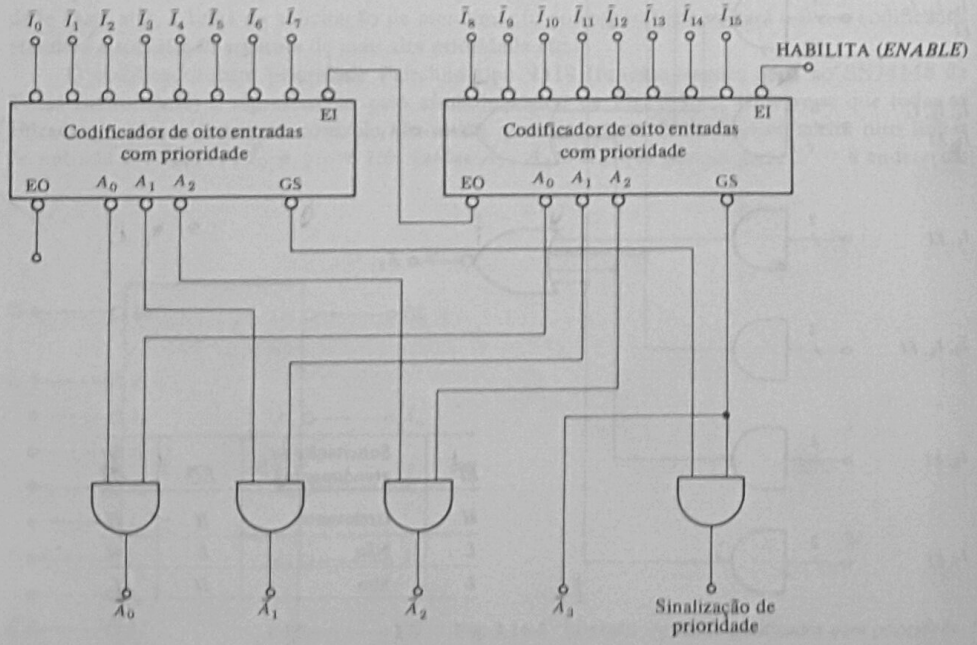


Fig. 3.16-4 Dois codificadores de oito entradas usados para formar um codificador de dezesseis entradas.

2. Quando o sistema estiver habilitado, o endereço de saída $\overline{A}_3\overline{A}_2\overline{A}_1\overline{A}_0$ indica a linha de mais alta prioridade que estiver baixa. $\overline{I}_{15}$ tem a prioridade mais alta e, se $\overline{I}_{15}$ = L, então $\overline{A}_3\overline{A}_2\overline{A}_1\overline{A}_0$ = LLLL = 15, independentemente de que outras entradas possam ser baixas.
3. O esquema pode ser expandido. Três codificadores de oito entradas podem ser combinados em um codificador de 24 entradas etc.

Em sistemas digitais, os projetistas muitas vezes provêem uma linha cujo nível lógico é usado para alertar o sistema ou o usuário do sistema de alguma situação de especial interesse ou importância. Esta indicação de um único bit é chamada *sinalização* (*flag*), análoga à sinalização feita com um semáforo. Esta sinalização, a *sinalização com prioridade*, foi incorporada ao sistema da Fig. 3.16-4. Ela não é essencial para este sistema, mas indica se existe uma solicitação de atendimento de qualquer origem.

3.17 CONVERSORES DE CÓDIGOS

Freqüentemente, informação disponível em forma codificada deve ser traduzida para um código diferente. Um circuito lógico que realiza esta tradução é chamado *conversor de código*. Um conversor de código pode ser construído ligando um decodificador e um codificador em cascata, conforme mostrado na Fig. 3.17-1. Um arranjo particular de bits identifica uma linha Z. Correspondendo à linha Z selecionada, um outro arranjo particular de bits, ou seja, uma *palavra* característica, aparecerá na saída do codificador. O número máximo de palavras de saída possível é 2^m, mas o número de bits na palavra de saída não precisa guardar nenhuma relação especial com o número de linhas de entrada do decodificador.

Um exemplo interessante de conversor de código aparece em conexão com o dispositivo digital de display representado na Fig. 3.17-2. Este dispositivo, chamado *display de sete segmentos*, possui sete segmentos, identificados pelas letras de *a* a *g*, que podem ser iluminados individualmente. A iluminação é feita através de díodo emissores de luz (LEDs). O segmento *a* é ligado quando o terminal de entrada *a* tem seu nível lógico levado para 1; o segmento *b* é ligado quando o terminal de entrada *b* tem seu nível lógico levado para 1 etc. (Em outros displays de sete segmentos não são usados LEDs e o display não provê sua própria iluminação. Em vez disso, um nível lógico 1 aplicado ao terminal de entrada altera a reflexão ou a transmissão de luz através do segmento correspondente. Estes displays são displays de cristal líquido, abreviados LCDs.) Os dígitos decimais podem aparecer pela seleção apropriada dos segmentos iluminados.

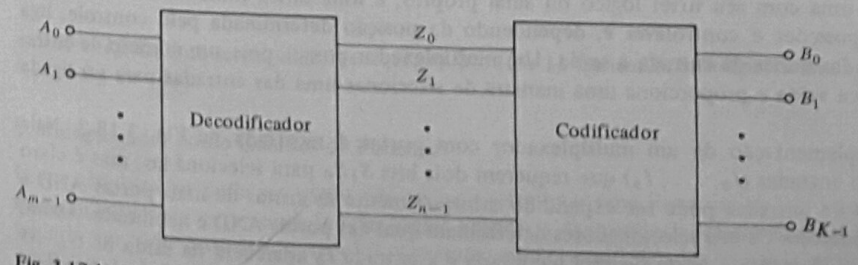


Fig. 3.17-1 Um conversor de código construído ligando um decodificador e um codificador em cascata.

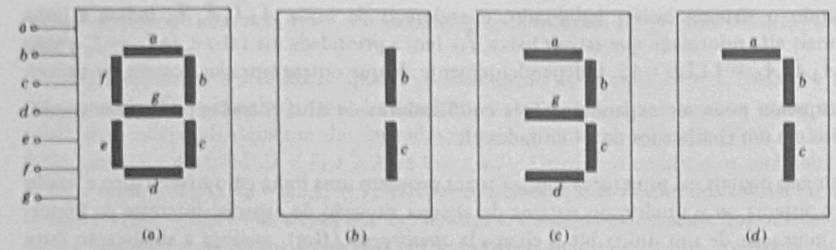


Fig. 3.17-2 (*a*) Um display de sete segmentos. Em (*b*), (*c*) e (*d*) são mostrados os números 1, 3 e 7.

As Figs. 3.17-2*b* a *d* mostram a seleção de segmentos que gera os números 1, 3 e 7, respectivamente.

Se os dígitos decimais forem apresentados em formato BCD e desejarmos mostrá-los em um display de sete segmentos, necessitaremos de um *conversor de código* que traduza o código BCD de quatro bits de entrada para uma palavra de código de saída que ilumine os segmentos apropriados. Por exemplo, se a entrada BCD for $ABCD = 0011$, correspondente ao decimal 3, é necessário que as saídas do conversor de código (identificadas de modo a corresponder às entradas de sete segmentos como na Fig. 2.18-2*a*) sejam $a = b = c = d = g = 1$ e $e = f = 0$. Os detalhes do projeto lógico do conversor de código BCD para sete segmentos são propostos como problema (Prob. 3.17-1).

Infelizmente, não há concordância completa sobre a terminologia a ser aplicada a conversores de código. Por exemplo, o conversor de código BCD para sete segmentos descrito acima, que é um componente comercial disponível, é chamado, por alguns fabricantes, decodificador, porque o conversor de código acoplado ao display recebe a entrada em BCD e mostra a mesma *explicitamente*. Vista desta maneira, a combinação de conversor e display constitui-se em um *decodificador* e, por extensão, o termo é aplicado ao conversor de código também.

3.18 MULTIPLEXADORES

Um multiplexador executa a função indicada na Fig. 3.18-1. Aqui temos um número de linhas de entrada, cada uma com seu nível lógico ou sinal próprio, e uma única linha de saída. A chave de múltiplas posições é controlável e, dependendo da posição determinada pelo controle, liga uma ou outra das linhas de entrada à saída. Um multiplexador possui, pois, um número de entradas e uma única saída e proporciona uma maneira de selecionar uma das entradas para ser ligada à saída.

Uma implementação de um multiplexador com portas é mostrada na Fig. 3.18-2. Nela existem quatro entradas ($I_0, \ldots, I_3$) que requerem dois bits S_1S_0 para selecioná-las, mas é claro que o número de entradas pode ser expandido arbitrariamente às custas de mais portas AND e mais bits de endereço. Os bits selecionadores determinam qual das portas AND é habilitada. Assim, se $A_1 = A_0 = 0$, somente a porta G_0 será habilitada e a entrada I_0 aparecerá na saída de G_0. As saídas das demais portas AND estarão no nível lógico 0, portanto a saída Z da porta OR também

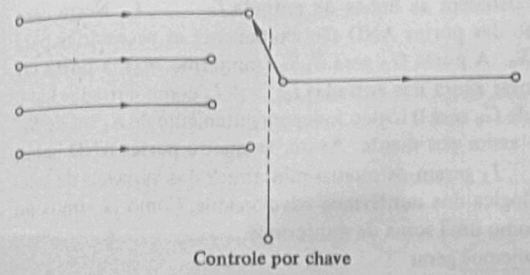


Fig. 3.18-1 A função de um multiplexador.

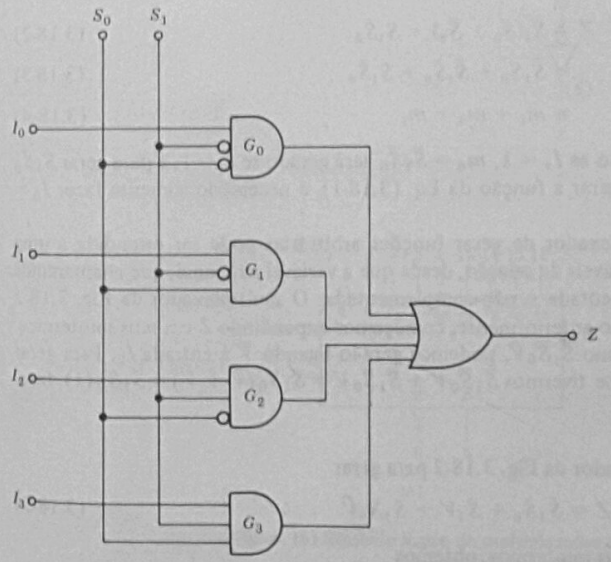


Fig. 3.18-2 Estrutura lógica de um multiplexador de quatro linhas para uma linha.

será $Z = I_0$. Os bits selecionadores $A_1A_0 = 01$, 10 e 11 transmitirão, respectivamente, as entradas I_1, I_2 e I_3.

O Multiplexador como Gerador de Funções

Um multiplexador pode ser usado não só para selecionar uma linha dentre um número de linhas mas também para gerar uma função lógica arbitrária das *variáveis selecionadoras*. Este resultado poderia ter sido antecipado se observássemos que o multiplicador consiste em uma estrutura AND-OR de dois níveis e que existem inversores nas linhas de seleção.

Vamos supor inicialmente que não existissem as linhas de entrada $I_0, \ldots, I_3$. Neste caso (Fig. 3.18-2) as linhas de entrada de seleção das portas AND são exatamente as necessárias para gerar todos os mintermos das variáveis S_1S_0. A porta G_0 gera $\bar{S}_1\bar{S}_0$ (mintermo m_0), a porta G_1 gera $\bar{S}_1S_0$ (mintermo m_1) etc. Podemos pensar agora nas entradas $I_0, \ldots, I_3$ como entradas habilitadoras. Se, por exemplo, $I_0 = 0$, a saída de G_0 será 0 lógico independentemente de S_1 ou de S_0. Mas, se $I_0 = 1$, a saída de G_0 será $\bar{S}_1\bar{S}_0$, e assim por diante. Assim, as quatro portas AND habilitadas pelas entradas correspondentes $I_0, \ldots, I_3$ geram os quatro mintermos das variáveis de seleção e a saída do multiplexador é a soma lógica dos mintermos selecionados. Como já vimos no Cap. 2, qualquer função pode ser expressa como uma soma de mintermos.

Vamos supor, como exemplo, que desejemos gerar

$$Z = \bar{S}_1 + S_1\bar{S}_0 \tag{3.18-1}$$

Expandimos Z em mintermos com o resultado

$$Z = \bar{S}_1(S_0 + \bar{S}_0) + S_1\bar{S}_0 \tag{3.18-2}$$

$$= \bar{S}_1S_0 + \bar{S}_1\bar{S}_0 + S_1\bar{S}_0 \tag{3.18-3}$$

$$= m_1 + m_0 + m_2 \tag{3.18-4}$$

O mintermo $m_1 = \bar{S}_1S_0$ será gerado se $I_1 = 1$, $m_0 = \bar{S}_1\bar{S}_0$ será gerado se $I_0 = 1$, e para gerar $S_1\bar{S}_0$ necessitamos $I_2 = 1$. Assim, para gerar a função da Eq. (3.18-1), é necessário somente fazer $I_0 = I_1 = I_2 = 1$ e $I_3 = 0$.

A habilidade de um multiplexador de gerar funções arbitrárias pode ser estendida a uma variável mais que o número de variáveis de seleção, desde que a variável adicional, que chamaremos de V, exista nas formas complementada e não-complementada. O multiplexador da Fig. 3.18-2 pode gerar $Z = f(S_1, S_0, V)$. Como anteriormente, começamos expandindo Z em seus mintermos. Se, por exemplo, temos o mintermo $\bar{S}_1\bar{S}_0\bar{V}$, podemos gerá-lo ligando $\bar{V}$ à entrada I_0. Para gerar $\bar{S}_1\bar{S}_0V$ aplicamos V à entrada I_0. Se tivermos $\bar{S}_1\bar{S}_0\bar{V} + \bar{S}_1\bar{S}_0V = \bar{S}_1\bar{S}_0(\bar{V} + V) = \bar{S}_1\bar{S}_0(1)$, fazemos $I_0 = 1$ etc.

Exemplo 3.18-1 Usar o multiplexador da Fig. 3.18-2 para gerar

$$Z = \bar{S}_1\bar{S}_0 + \bar{S}_1V + S_1S_0\bar{V} \tag{3.18-5}$$

SOLUÇÃO: Expandindo Z em seus mintermos, obtemos

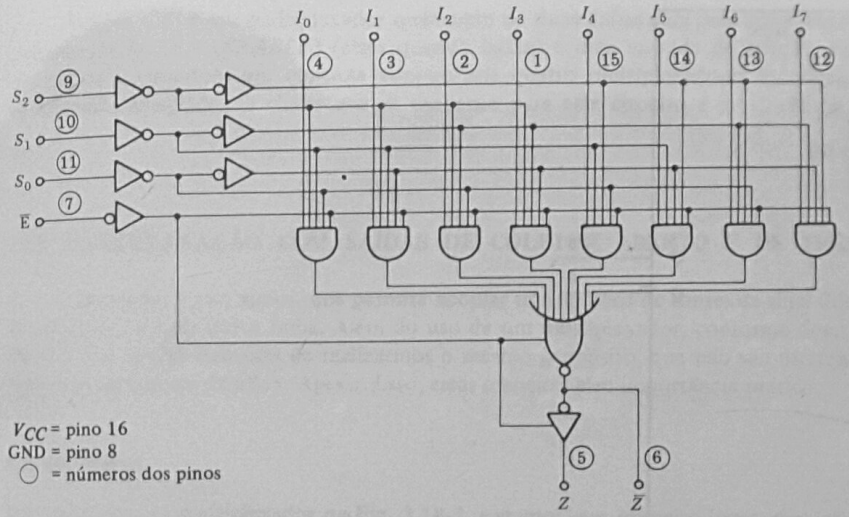
$$Z = \bar{S}_1\bar{S}_0V + \bar{S}_1\bar{S}_0\bar{V} + \bar{S}_1S_0V + S_1S_0\bar{V} \tag{3.18-6}$$

$$= \bar{S}_1\bar{S}_0(V + \bar{V}) + \bar{S}_1S_0V + S_1S_0\bar{V} \tag{3.18-7}$$

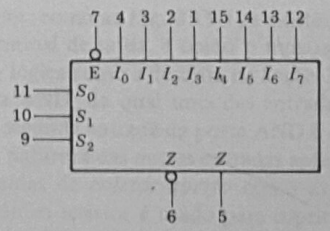
$$= \bar{S}_1\bar{S}_0(1) + \bar{S}_1S_0V + S_1S_0\bar{V} \tag{3.18-8}$$

Conseqüentemente, é necessário que $I_0 = 1, I_1 = V, I_3 = \bar{V}$, e como não existe termo envolvendo $S_1\bar{S}_0$, fazemos $I_2 = 0$.

O CI tipo '251 é um multiplexador de oito entradas. Seu diagrama lógico é mostrado na Fig. 3.18-3*a* e sua representação mais antiga (mas ainda mais comum) é mostrada na Fig. 3.18-3*b*. Note-se que possui HABILITA (*ENABLE*) ativa quando baixa e saída tanto ativa quando alta como ativa quando baixa, além de as saídas serem do tipo três-estados. O novo símbolo é mostrado na Fig. 3.18-4. Os números do lado de fora do retângulo são os números dos pinos. A palavra MUX identifica um multiplexador. O pino 7 é uma entrada HABILITA (*ENABLE*) ativa quando baixa. Os triângulos nas saídas (pinos 5 e 6) indicam que as mesmas são do tipo três-estados. Os pinos 9, 10 e 11 são as entradas de seleção, possuindo, respectivamente, pesos $2^0 = 1$, $2^1 = 2$ e $2^2 = 4$. O símbolo $\frac{0}{7}$ indica que estas entradas de seleção, dependendo de quais são ativas, isto é, altas, podem selecionar uma ou outra das entradas de sinal que são numeradas de 0 a 7. O símbolo G significa que entre as entradas de sinal e as entradas de seleção existe uma *dependência* do tipo AND. Isto significa, por exemplo, que internamente a entrada 6 (pino 13) é aplicada a uma porta AND que é habilitada quando os pinos 9 e 10 forem altos (ativos) e o pino 11 baixo (inativo), pois neste caso o peso das entradas de seleção é $2^2 + 2^1 = 6$.



(*a*)



(*b*)

Fig. 3.18-3 (*a*) Diagrama lógico. (*b*) Símbolo lógico do multiplexador de oito entradas tipo '151.

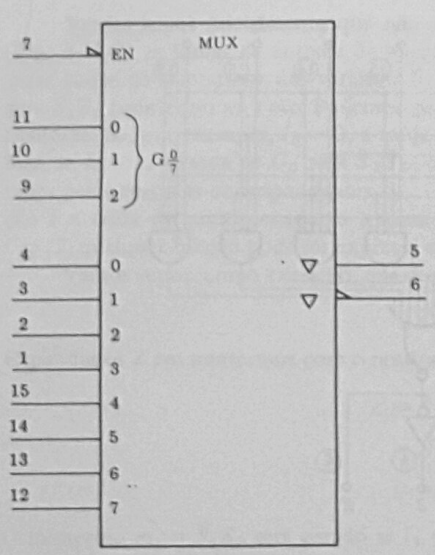


Fig. 3.18-4 O novo símbolo para o multiplexador tipo '151.

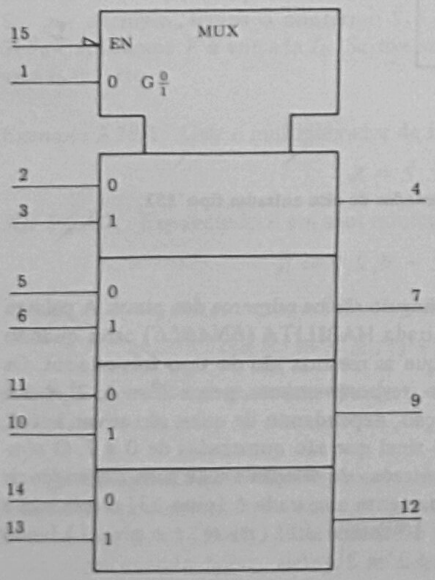


Fig. 3.18-5 Símbolo para o multiplexador quádruplo de duas linhas para uma linha tipo '157.

O tipo '157 é um multiplexador quádruplo de duas linhas para uma linha. Possui uma única entrada HABILITA (*ENABLE*) (ativa quando baixa) e uma entrada de seleção. Estas entradas de controle executam um controle comum aos quatro multiplexadores individuais existentes no circuito integrado. O símbolo mais moderno para este circuito é mostrado na Fig. 3.18-5.

3.19 MULTIPLEXAÇÃO COM SAÍDAS DE COLETOR ABERTO E DE TRÊS-ESTADOS

A multiplexação, como vimos, nos permite acoplar um número de fontes de sinal diferentes, uma de cada vez, a uma única linha. Além do uso de um multiplexador, conforme descrito na seção anterior, há outras maneiras de realizarmos o mesmo propósito, que não são diferentes em princípio mas apenas em detalhes. Apesar disso, estas maneiras têm importância prática.

Coletor Aberto

Em referência ao multiplexador da Fig. 3.18-2, notamos, em primeiro lugar, que, se a porta OR fosse substituída por uma porta NOR, o circuito continuaria a funcionar como um multiplexador. Poderíamos, facilmente, anular a inversão provocada pela porta NOR adicionando um inversor à linha *Z*.

Consideremos, a seguir, como na Fig. 3.19-1, a existência de um número de CI idênticos cujo circuito, levando a um terminal de saída, é como o mostrado. Esta saída é obtida de um terminal de uma chave operada por lógica como a descrita na Seção 3.7 (ver Fig. 3.7-1). Além disso, a chave é acionada por uma porta AND, da qual uma das entradas é acionada por um sinal HABILITA (*ENABLE*) externo. Uma segunda entrada da porta AND é um sinal qualquer gerado dentro do CI. A natureza deste sinal e a natureza das outras entradas aplicadas externamente são irrelevantes. Os CIs são equipados com saídas de *coletor aberto* como as descritas na Seção 3.8. As saídas são ligadas em paralelo e um único resistor é usado para suprir a tensão de alimentação, resistor este externo ao circuito.

Lembrando que o arranjo em paralelo das chaves operadas por lógica se constitui em uma porta NOR (ver Fig. 3.8-1), a configuração da Fig. 3.19-1 constitui-se em um conjunto de portas AND seguidas pela porta NOR que forma um multiplexador através do qual podemos transmitir o sinal de um ou outro CI (*chip*) até a saída. Logicamente, devemos prover um decodificador para assegurarmo-nos de que somente um CI (*chip*) seja habilitado de cada vez. É possível que um circuito decodificador extra não seja necessário, pois, como vimos, muitos CIs (*chips*) possuem entradas HABILITA (*ENABLE*) (EN) múltiplas, conforme mostrado nas Figs. 3.15-2 e 3.15-3. Neste caso basta suprir as entradas de seleção, pois a decodificação é executada no próprio CI (*chip*).

Resumindo, se tivermos diversos CIs (*chips*) com saídas de coletor aberto e necessitarmos multiplexar estas saídas, não é necessário usar um CI (*chip*) multiplexador adicional para este propósito; basta ligar todas as saídas a uma linha comum e fazer a ligação à fonte de alimentação através de um resistor.

No simbolismo antigo não há um símbolo para representar um coletor aberto e só é possível descobrir a existência desta característica lendo o manual do fabricante. A representação de um coletor aberto no novo simbolismo é mostrada na Fig. 3.19-1*b*.

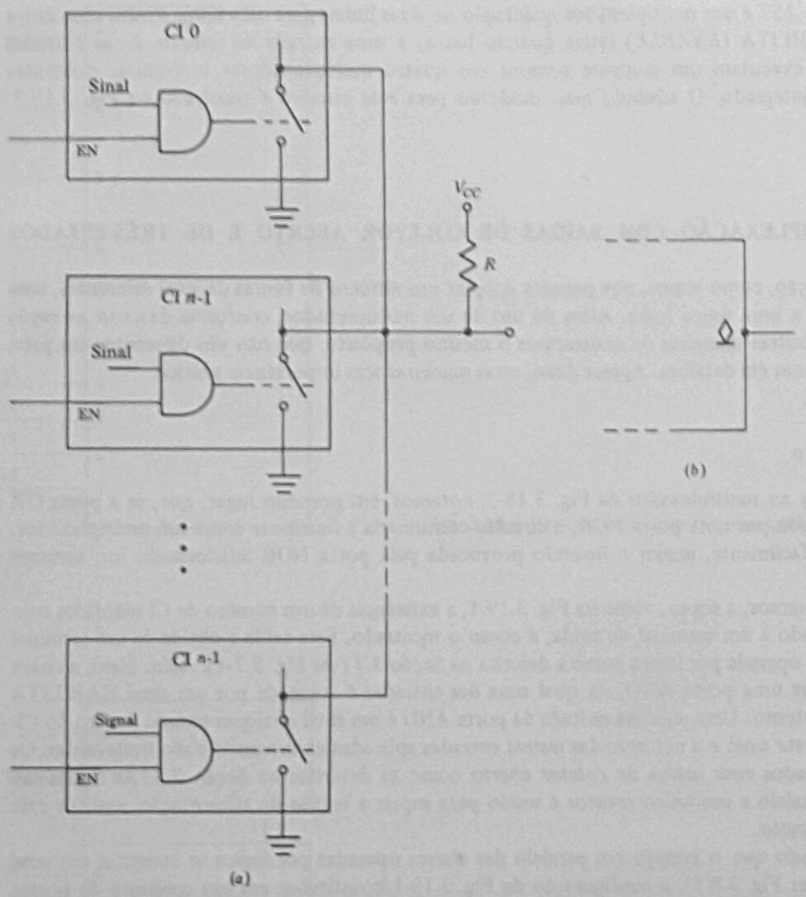


Fig. 3.19-1 (*a*) Saídas de coletor aberto podem ser multiplexadas em uma linha comum sem a adição explícita de um multiplexador. (*b*) Símbolo para uma saída de coletor aberto.

Saída de Três-Estados

Circuitos integrados com saídas de três-estados também podem ser multiplexados sem o uso explícito de um multiplexador, pois, com uma saída de três-estados, um CI, cuja saída está desabilitada está efetivamente desligado da linha de saída. Assim, para obter multiplexação, basta ligar todas as saídas a uma única linha e fazer com que somente um circuito seja habilitado de cada vez, seja através do uso de um decodificador externo ou da decodificação existente no próprio circuito integrado.

A razão principal para a incorporação da característica de três-estados em um CI é exatamente a possibilidade de obter multiplexação sem o uso explícito de um multiplexador. O mesmo comentário aplica-se à característica coletor aberto. Consideremos a situação mostrada na Fig 3.19-2, onde temos dois CIs que suprem não uma saída singela, mas uma saída que consiste em uma

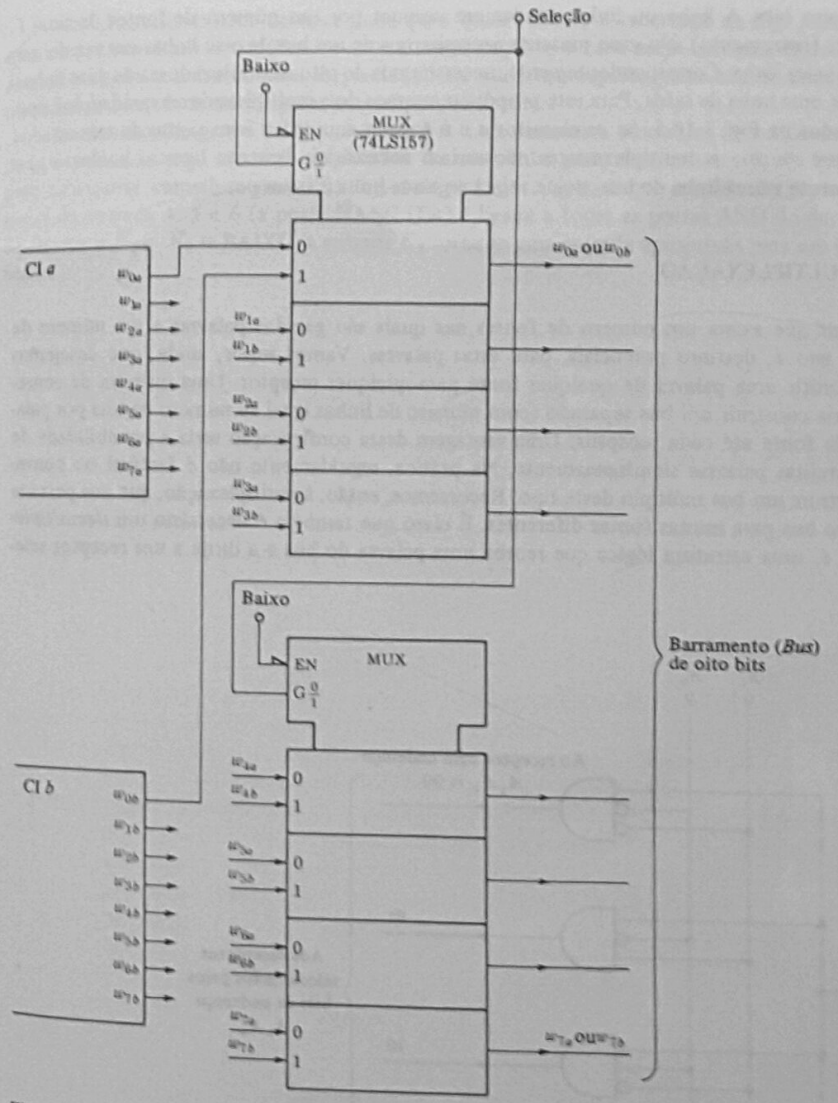


Fig. 3.19-2 Dois CI gerando palavras de saída de oito bits que são multiplexadas em um barramento de oito bits.

palavra de oito bits. A linha ou linhas usadas em comum por um número de fontes de sinal é chamada *bus* (*barramento*). No caso presente necessitamos de um bus de oito linhas em vez de um bus de uma única linha. Correspondentemente, necessitamos de oito multiplexadores de duas linhas de entrada e uma linha de saída. Para este propósito, usamos dois multiplexadores quádruplos tipo '157 mostrados na Fig. 3.18-5. Se os circuitos *a* e *b* fossem equipados com saídas de três-estados ou de coletor aberto, os multiplexadores não seriam necessários. Bastaria ligar as saídas w_{0a} e w_{0b} diretamente a uma linha do bus, w_{1a} e w_{1b} à segunda linha e assim por diante.

3.20 DEMULTIPLEXAÇÃO

Vamos supor que exista um número de fontes nas quais são geradas palavras e um número de receptores, isto é, destinos potenciais, para estas palavras. Vamos supor, ainda, que desejemos poder transmitir uma palavra de qualquer fonte para qualquer receptor. Uma maneira de conseguir isto seria construir um bus separado (com número de linhas igual ao número de bits por palavra) de cada fonte até cada receptor. Uma vantagem desta configuração seria a possibilidade de transmitir muitas palavras simultaneamente. Na prática, seguidamente não é factível ou conveniente construir um bus múltiplo deste tipo. Recorremos, então, à multiplexação, que nos permite o uso de um bus para muitas fontes diferentes. É claro que também é necessário um *demultiplexador*, isto é, uma estrutura lógica que receba uma palavra do bus e a dirija a um receptor selecionado.

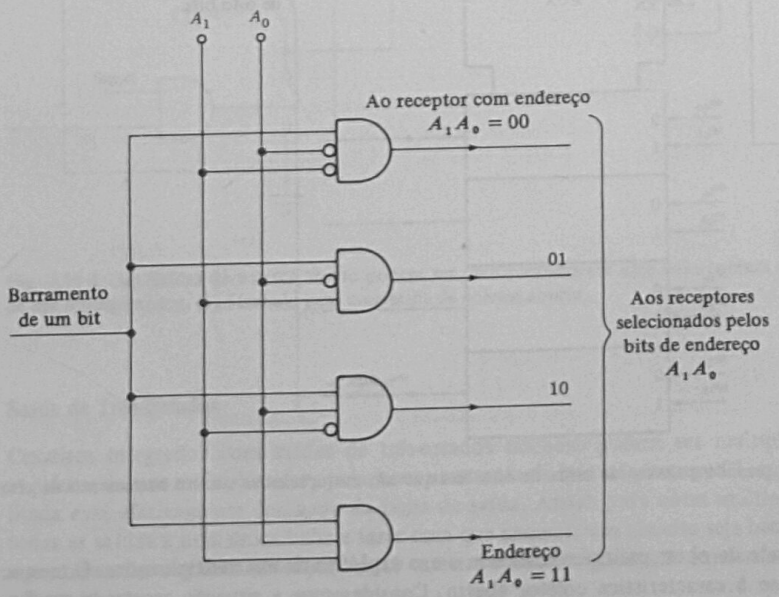


Fig. 3.20-1 A estrutura lógica de um demultiplexador.

Um demultiplexador de uma linha para quatro linhas é mostrado na Fig. 3.20-1. Note-se que a estrutura é um *decodificador* com a modificação que cada porta tem uma entrada adicional à qual é ligada a linha do bus. Dependendo do endereço suprido pelos bits de endereço A_1A_0, os dados de entrada serão dirigidos a um ou outro destino.

O circuito integrado da Fig. 3.15-2, que caracterizamos como um decodificador, também pode ser usado como um demultiplicador de uma linha para oito linhas e é, de fato, descrito pelo fabricante como um decodificador-multiplexador. Note-se que a saída da porta ligada aos pinos de entrada 4, 5 e 6 (a porta HABILITA) é ligada a todas as portas AND do decodificador. Se fizermos $\overline{E}_1 = \overline{E}_2 =$ BAIXO, a entrada E_3 torna-se uma entrada apropriada para um bus de uma linha.

CAPÍTULO 4

FLIP-FLOPS, REGISTRADORES E CONTADORES

4.1 INTRODUÇÃO

O circuito da Fig. 4.1-1, que consiste em um par de inversores acoplados, é a estrutura básica de um circuito lógico básico e muito importante chamado *latch estático*. Na Fig. 4.1-1*a* a estrutura é desenhada de modo a mostrar explicitamente que a saída do primeiro inversor é ligada à entrada do segundo e que a saída do segundo é ligada à entrada do primeiro. Na Fig. 4.1-1*b* o circuito é desenhado de uma maneira mais conveniente. Denominamos os dois terminais acessíveis de Q e $\overline{Q}$, antecipando o fato, que verificaremos em seguida, de serem os níveis lógicos desses terminais complementares. Na discussão seguinte usaremos a convenção de lógica positiva em que H (o nível de tensão mais alto) representa o nível lógico 1 e L (o nível de tensão mais baixo) representa o lógico 0.

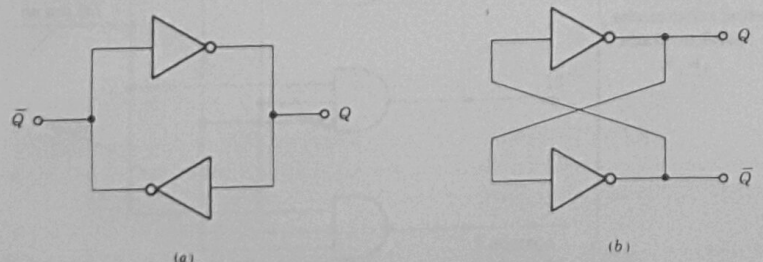


Fig. 4.1-1 (*a*) Inversores acoplados constituindo a estrutura básica de um latch. (*b*) O circuito redesenhado de modo mais conveniente.

Sem intervenção externa, o latch permanece indefinidamente em uma de duas situações possíveis chamadas *estados*. Vamos supor que Q = L; neste caso a saída do inversor inferior seria, de fato, $\overline{Q}$ = H. O postulado Q = L é autoconsistente e, se o latch se encontrar inicialmente no estado Q = L e $\overline{Q}$ = H, permanecerá indefinidamente neste estado. O segundo estado é aquele onde Q = H e $\overline{Q}$ = L, como pode ser facilmente verificado.

O elemento *novo* e essencial que o latch introduz às estruturas lógicas até agora consideradas é que o latch pode ser usado para estabelecer e manter um nível lógico sem qualquer intervenção externa. Em todas as estruturas lógicas que examinamos anteriormente, a saída de uma porta dependia das entradas da porta que eram estabelecidas por uma fonte externa. Na ausência de entradas não é possível conceber saídas de portas sem incorrer em ambigüidades.

Devido a esta independência do latch das entradas externas, ele pode ser usado para *armazenar*, isto é, *registrar* ou *lembrar* um bit lógico. Um conjunto de k latches pode ser usado para *registrar* uma palavra de k bits. Vamos supor, por exemplo, que desejemos guardar a palavra de oito bits 11001010 para uso futuro. Construiríamos um conjunto de oito latches e decidiríamos que terminais de cada latch seriam Q e que terminais seriam $\overline{Q}$ e os identificaríamos. A seguir estabeleceríamos os estados dos latches, conforme mostrado na Fig. 4.1-2, de modo a haver uma correspondência biunívoca entre os estados dos latches e os bits da palavra a ser lembrada. A palavra foi *escrita* no *registrador* e lá permanecerá enquanto desejarmos. Um *registrador* é, pois, o "papel" no qual podemos escrever qualquer palavra que desejemos lembrar. Note-se, também, na Fig. 4.1-2, que nos terminais $\overline{Q}$ do registrador existe uma segunda palavra na qual cada bit é o complemento do bit da palavra armazenada. A disponibilidade desta segunda palavra é, muitas vezes, conveniente.

Os dois estados de um latch são chamados estado *set* e estado *reset*. O estado set é aquele em que o terminal que decidimos chamar Q é Q = H ($\overline{Q}$ = L). O estado reset é aquele em que Q = = L ($\overline{Q}$ = H). O estado reset é freqüentemente chamado estado limpo (*clear*). É muito fácil colocar um latch em um estado ou no outro, conforme desejarmos. Vamos supor que queremos colocar o latch no estado set. Para isto, basta ligar temporariamente o terminal Q a um ponto externo que esteja no nível de tensão alto. Esta conexão temporária leva Q ao estado alto e $\overline{Q}$ ao estado baixo e, quando a conexão for removida, o latch *permanecerá* no estado Q = H e $\overline{Q}$ = L. De modo correspondente, uma conexão *temporária* do terminal Q a um ponto externo com nível de tensão

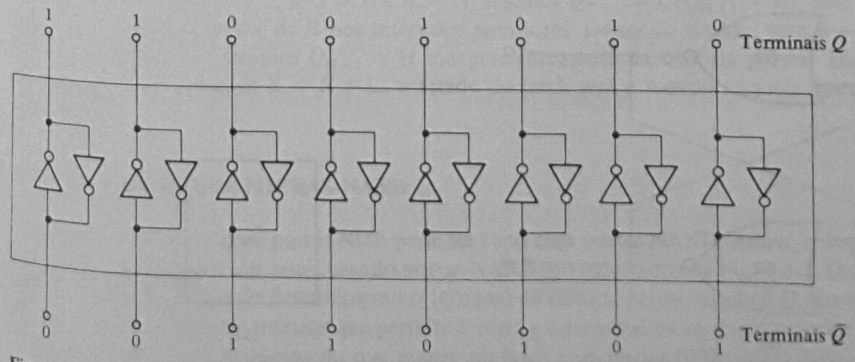


Fig. 4.1-2 Um conjunto de latches servindo como registrador para armazenar uma palavra.

baixo tornará reset (limpará) o latch. O latch permanecerá no estado de reset *indefinidamente*, desde que não seja perturbado.

O latch estático é um dos vários circuitos lógicos que têm capacidade de armazenar um bit. Outros circuitos relacionados que estudaremos incluem o latch dinâmico e o flip-flop.

4.2 UM LATCH COM PORTAS NOR

Para maior conveniência na manipulação do latch, é vantajoso substituir os inversores por portas NOR ou NAND. Os terminais adicionais de entrada servem como terminais de *controle* e permitem maneiras adicionais de acesso ao latch.

O diagrama lógico e o símbolo lógico de um latch usando portas NOR são mostrados na Fig. 4.2-1. Os terminais de controle das portas são chamados R e S por razões que aparecerão a seguir. Se $R = S =$ L, as portas NOR estão habilitadas e, em relação à outra entrada de cada porta, as portas comportam-se como simples inversores, como requerido para o funcionamento do latch. Assim, para $R = S =$ L, o latch pode estar em qualquer um de seus dois estados possíveis. O estado no qual ele se encontra vai depender da história passada. Quando $R = S =$ L, o latch não é influenciado pelos terminais de entrada R e S.

Vamos supor que R passe a $R =$ H; a porta G1 é desabilitada e Q passa a $Q =$ L, enquanto $\bar{Q}$ passa a $\bar{Q} =$ H. Assim, o fato de R passar a $R =$ H reajustou (reset) o latch. Por esta razão, o terminal R foi chamado de R significando *reset*. Se o latch estivesse originalmente no estado reset, fazendo $R =$ H, não produziria alteração no estado. Se o latch estivesse originalmente no estado set, fazendo $R =$ H, causaria uma troca do estado set para o estado reset. Mais importante, no estado reset $\bar{Q}$, que vale $\bar{Q} =$ H e é uma das entradas da porta G1, desabilitou G1, que não pode responder ao nível lógico da entrada R. Conseqüentemente, se retornarmos R ao nível $R =$ L, o latch *permanecerá* no estado reset.

Ao todo temos o seguinte resultado: com $R = S =$ L, qualquer dos dois estados é possível. Se R passa a $R =$ H temporária ou permanentemente, o latch será reajustado (reset). De modo similar, começando com $R = S =$ L, se S (significando *set*) passar para $S =$ H temporária ou permanentemente, o latch irá para o estado set ou permanecerá neste estado se já estiver nele.

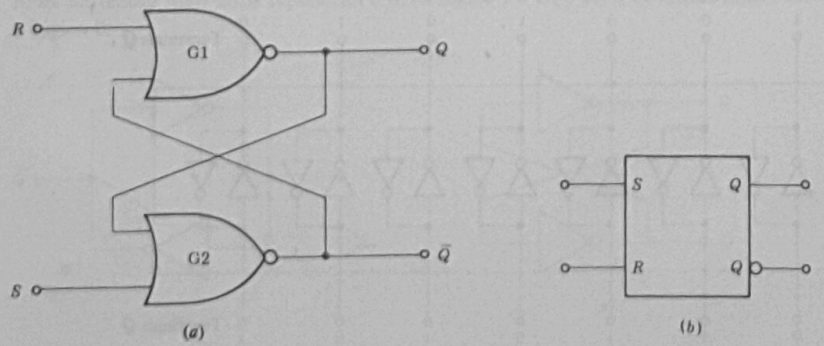


Fig. 4.2-1 Um latch usando portas NOR: (*a*) diagrama lógico e (*b*) símbolo lógico.

S	R	Q	$\bar{Q}$
L	L	L ou H	H ou L
L	H	L	H
H	L	H	L
H	H	Não usada	

(*a*)

S	R	Q_{n+1}	$\bar{Q}_{n+1}$
L	L	Q_n	$\bar{Q}_n$
L	H	L	H
H	L	H	L
H	H	Não usada	

(*b*)

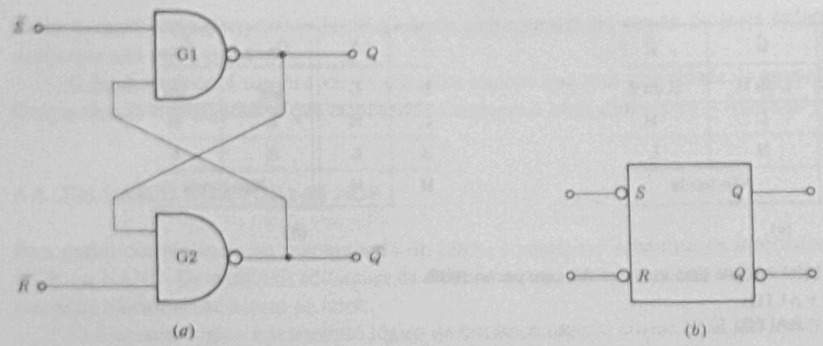
Fig. 4.2-2 Tabelas verdade para um latch SR com portas NOR.
(H = ALTO)
(L = BAIXO)

Quando desejarmos que o latch permaneça inalterado para lembrar um bit, faremos $S = R =$ L. Quando desejarmos armazenar um 0 lógico no latch, faremos com que $S =$ L enquanto $R =$ H, pelo menos temporariamente. Para armazenar um 1 lógico, faremos com que $S =$ H enquanto $R =$ L. Não há utilidade nenhuma em fazermos com que S e R sejam $S = R =$ H simultaneamente. Na verdade há uma desvantagem; se, a partir de $S = R =$ H, fizermos com que S e R passem a $S = R =$ L simultaneamente, o estado resultante do latch não pode ser previsto. Como a simultaneidade absoluta não é possível, se R mudar de nível primeiro, passaremos pela situação intermediária $S =$ H e $R =$ L. Como resultado, o latch passaria para o estado set e permaneceria neste estado quando S e R passassem para $S = R =$ L. De modo similar, se S mudasse primeiro, o latch iria para o estado reset. Incidentalmente, notamos que com $S = R =$ H teríamos $Q = \bar{Q} =$ L e a implicação da Fig. 4.2-1 de que as saídas são complementares seria incorreta.

O comportamento do latch é convenientemente resumido por qualquer uma das tabelas verdade da Fig. 4.2-2. A tabela da Fig. 4.2-2*a* não requer explicação adicional, exceto para notar que pelas razões apresentadas acima o caso $S = R =$ H não é usado. Em relação à tabela da Fig. 4.2-2*b*, contemplemos o caso em que as entradas S e R estivessem mudando de forma arbitrária. Haveria intervalos em que $S = R =$ L, intervalos em que $S =$ L e $R =$ H e intervalos em que $S =$ H e $R =$ L. Novamente, conforme notado anteriormente, excluímos o caso em que $S = R =$ H. Vamos numerar os intervalos como 1, 2, . . . , n, $n + 1$, . . . na ordem em que eles ocorrem no tempo. No intervalo $n + 1$, se $S =$ L e $R =$ H, teremos $Q_{n+1} =$ L ($\bar{Q}_{n+1} =$ H), não importando qual o nível lógico de S e de R nos intervalos anteriores. De modo similar, com $S =$ H e $R =$ L no intervalo $n + 1$, teremos $Q_{n+1} =$ H independentemente da história prévia. Todavia, se no intervalo $n + 1$ tivermos $S = R =$ L, o estado do latch será o mesmo naquele intervalo que no intervalo n.

4.3 UM LATCH COM PORTAS NAND

O que pode ser feito com portas NOR pode ser feito com portas NAND. Assim, o diagrama lógico e o símbolo lógico de um latch usando portas NAND são mostrados na Fig. 4.3-1. Como anteriormente, de modo arbitrário denominamos o terminal de saída superior terminal Q. No caso presente é o terminal de controle inferior que permite o reset e o terminal de controle superior que permite o set do latch. Isto é o oposto do que ocorre no latch com portas NOR. Além disso, os terminais de controle são *ativos quando baixos*, sendo, portanto, denominados $\bar{S}$ e $\bar{R}$ ao invés de S e R.



(*a*) (*b*)

Fig. 4.3-1 Um latch usando portas NAND: (*a*) diagrama lógico e (*b*) símbolo lógico.

$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$
L	L	Não usada	
L	H	H	L
H	L	L	H
H	H	L ou H	H ou L

(*a*)

$\bar{S}$	$\bar{R}$	Q_{n+1}	$\bar{Q}_{n+1}$
L	L	Não usada	
L	H	H	L
H	L	L	H
H	H	Q_n	$\bar{Q}_{n+1}$

(*b*)

Fig. 4.3-2 Tabelas verdade para um latch *SR* com portas NAND.
(H = ALTO)
(L = BAIXO)

As tabelas verdade do latch com portas NAND são mostradas na Fig. 4.3-2 e podem ser facilmente verificadas. Quando $\bar{S} = \bar{R} =$ H, ambas as portas estão habilitadas e qualquer um dos estados é possível, sendo o estado determinado pelos níveis anteriores de $\bar{S}$ e $\bar{R}$. Começando com $\bar{S} = \bar{R} =$ H, se passarmos $\bar{R}$ de H para L, a porta G2 será desabilitada e $\bar{Q}$ passará (ou permanecerá) a $\bar{Q} =$ H. Ambas as entradas de G1 são altas e Q passa (ou permanece) a $Q =$ L. A saída $Q =$ L de G1 é uma entrada de G2 que é desabilitada, podendo, conseqüentemente, $\bar{R}$ passar a $\bar{R} =$ H sem que haja alteração de estado. Em resumo, uma troca de $\bar{R}$ de nível alto para nível baixo, mesmo que temporária, serve para reajustar (reset) o latch. Do mesmo modo, podemos estabelecer que uma troca de $\bar{S}$ de nível alto para nível baixo, mesmo que temporária, serve para ajustar (set) o latch.

O diagrama lógico (usando lógica positiva) de um latch set-reset quádruplo tipo '279 é mostrado na Fig. 4.3-3. Note-se que, dispondo de dezesseis pinos no encapsulamento, o fabricante adicionou dois controles $\bar{S}$ adicionais a dois dos latches. Assim, sem o uso de circuitos adicionais, estes dois latches podem ser ajustados (set) por um ou outro de dois sinais de controle (ou, é claro, por ambos). Os terminais de controle são ativos quando baixos e poderíamos desejar mostrar esta característica através do simbolismo da lógica mista, em vez de usar as barras sobre S e R. Neste caso um latch seria representado como na Fig. 4.3-4.

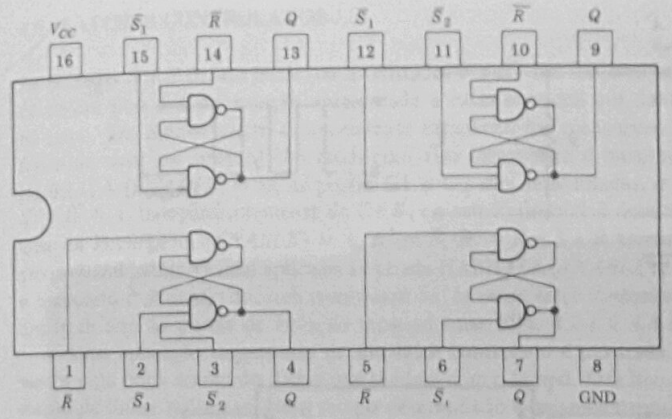
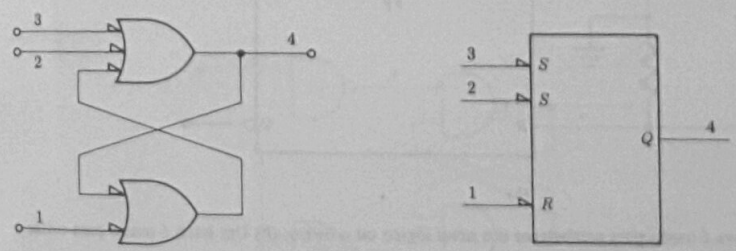


Fig. 4.3-3 Diagrama lógico de um latch *SR* quádruplo tipo '279. A lógica usada é positiva.



(*a*) (*b*)

Fig. 4.3-4 (*a*) Diagrama lógico. (*b*) Símbolo lógico de um dos latches do tipo '279 usando lógica mista.

4.4 A CHAVE SEM TREPIDAÇÃO

Consideremos a situação representada na Fig. 4.4-1 como uma aplicação útil, embora simples, de um latch. Na Fig. 4.4-1*a* uma chave é usada para tornar disponível uma tensão $V_0 = 0$ ou $V_0 = V$. Estas tensões podem ser as tensões correspondentes aos níveis lógicos de um sistema digital. Se for usada lógica positiva, 0 V, representada na Fig. 4.4-1 como massa (*ground*), representaria o nível lógico 0, ao passo que V volts representaria o nível 1. O arranjo permite que um ou outro nível lógico seja manualmente apresentado à entrada de um sistema digital. O nível lógico 0 aparece em V_0 quando a chave está na posição B e o nível 1 aparece quando a chave está na posição A. Se o resistor R não estivesse presente, o nível lógico de V_0 seria ambíguo quando o braço móvel da chave estivesse entre as posições de contato. Ligamos R à massa (0 V representando nível 0) de modo que V_0 é sempre 0, exceto quando a chave faz contato em A. Poderíamos igualmente ter ligado o resistor entre o braço da chave e V.

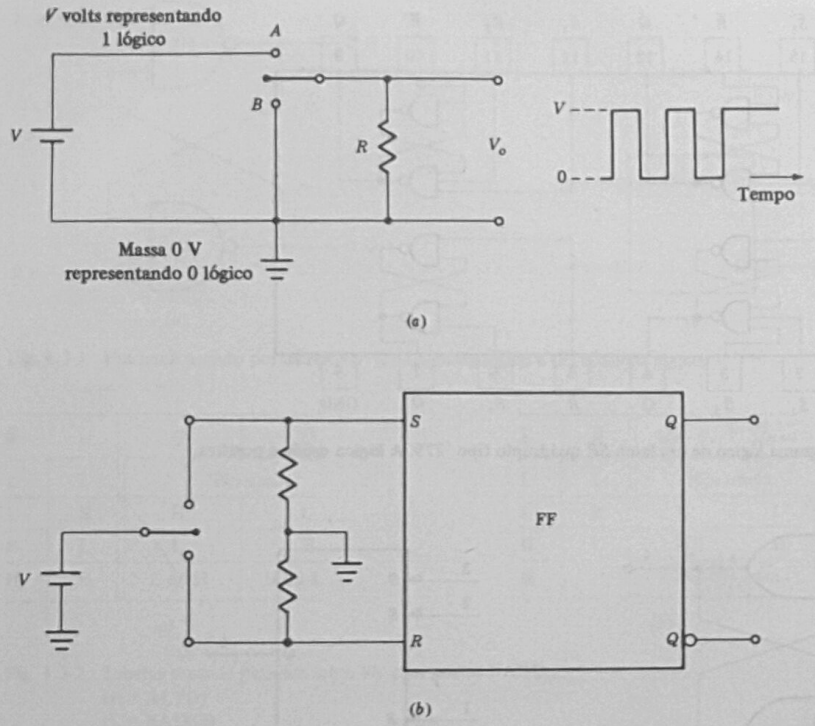


Fig. 4.4-1 (*a*) Uma chave é usada para estabelecer um nível lógico ou o outro. (*b*) Um latch é usado para evitar o efeito da trepidação da chave mecânica.

Uma característica comum às chaves mecânicas é que quando o braço é movido de uma posição para a outra ele *trepida* (*bounces*) diversas vezes antes de repousar na nova posição de contato. Se movermos o braço da chave da posição B para a posição A com a intenção de mudar V_0 de 0 lógico para 1 lógico, o braço da chave entrará em contato com A e trepidará várias vezes. (Normalmente, a trepidação não é suficiente para que o braço volte a fazer contato com B.) Conforme mostrado, a tensão V_0 faria diversas transições, ao invés de uma só, entre o nível lógico 0 e o nível lógico 1.

Uma chave *sem trepidação* é mostrada na Fig. 4.4-1*b* usando um latch set-reset. Os terminais S e R são ligados à massa (nível lógico 0) através de resistores, de modo a permanecer no nível lógico 0 exceto quando estiverem ligados a V (nível lógico 1) pela chave. A chave pode ser acionada em qualquer direção para estabelecer nível 1 em S ou em R. Se a chave estiver em R, então $Q = 0$; se estiver em S, então $Q = 1$. Vamos supor, como exemplo, que a chave esteja em R, de modo a obter $Q = 0$. Se movermos a chave para S, quando houver contato com S, teremos $Q = 1$. Se a chave trepidar (mas não o suficiente para voltar a R), teremos $S = R = 0$. Mas com estas entradas qualquer um dos estados do latch é permitido e ele permanecerá em $Q = 1$, eliminando o efeito da trepidação.

4.5 LATCHES CONTROLADOS

As entradas R e S de um latch (ou as entradas $\bar{S}$ e $\bar{R}$) são freqüentemente chamadas de entradas de dados, pois é a informação apresentada a estas entradas que determina o que é armazenado no latch. Em muitos casos, é conveniente introduzir um mecanismo que permita ligar o latch à fonte de sinal ou isolá-lo. Um latch com esse mecanismo é mostrado na Fig. 4.5-1*a*. Quando HABILITA (*ENABLE*) = 0, as portas G1 e G2 são desabilitadas, o latch é isolado dos dados, $S' = R' = 1$, independentemente de S e R, e o estado do latch é determinado pela história passada. Quando HABILITA (*ENABLE*) = 1, $S' = \bar{S}$, $R' = \bar{R}$ e S e R tornam-se entradas de set e reset ativas quando altas. O sinal aplicado à entrada HABILITA (*ENABLE*) é o sinal de *controle* (*strobe*) e o circuito é chamado de latch *controlado* ou, às vezes, latch *dinâmico*. De modo correspondente, um latch sem as portas de isolação mostrado nas Figs. 4.2-1 e 4.3-1 é chamado latch estático.

Uma aplicação importante de um latch controlado é mostrada na Fig. 4.5-1*b*. Nela observamos uma linha contendo dados que mudam com o tempo. Esta linha pode ser uma de um barramento de linhas múltiplas. Num tempo determinado desejamos capturar e manter o dado D (= 0 ou 1) da linha. Para conseguir isto, habilitamos o latch. Como $S = D$ e $R = \bar{D}$, sempre teremos $Q = D$, como pode ser facilmente verificado da tabela verdade do latch. Assim, enquanto HABILITA (*ENABLE*) for ativa, a saída Q *seguirá a entrada de dados* D. A característica desta ope-

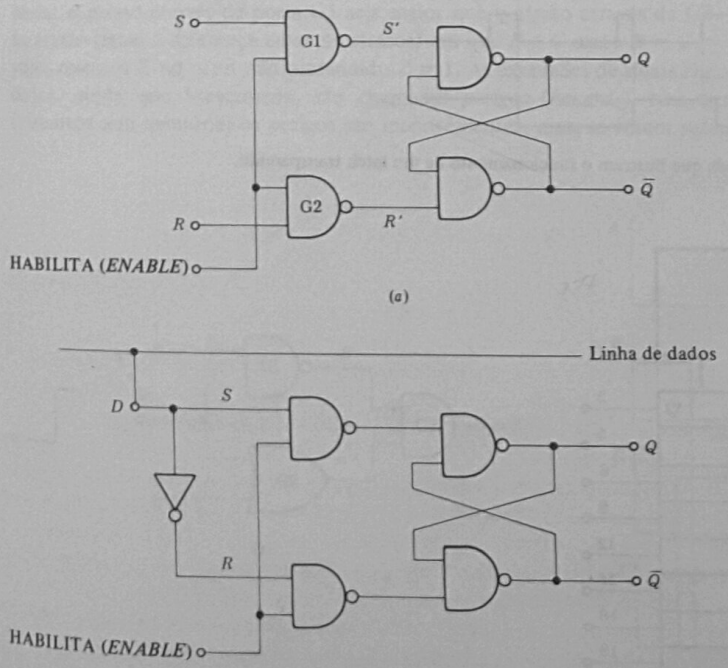


Fig. 4.5-1 (*a*) Um latch controlado. (*b*) Um latch controlado tipo D.

ração é apresentada ao descrever o latch como *transparente*. Quando desejarmos capturar e manter o dado, desabilitamos o latch. A parte estática do latch ficará na situação em que qualquer dos estados é permitido, e o estado armazenado será determinado pelo valor de D imediatamente antes de desabilitarmos o latch dinâmico. Se, na realidade, tivermos um barramento de linhas múltiplas conduzindo uma palavra, a captura da palavra inteira requer tantos latches controlados quantos forem os bits da palavra e todos estes latches seriam acionados pelo mesmo sinal HABILITA (*ENABLE*). Um latch com uma única entrada D, como na Fig. 4.5-1*b*, é chamado um latch tipo D.

Formas-de-onda que ilustram o funcionamento de um latch transparente são mostradas na Fig. 4.5-2. No instante t_0 o latch está desabilitado, pois C (a entrada habilitadora das portas de isolação) está no nível baixo. Supomos, arbitrariamente, que Q = L. De t_1 a t_2, o latch está habilitado e Q segue a entrada de dados D. Em t_2, o latch é desabilitado e o dado D = H é armazenado. Outras ocasiões de armazenamento são mostradas em t_3 e t_4.

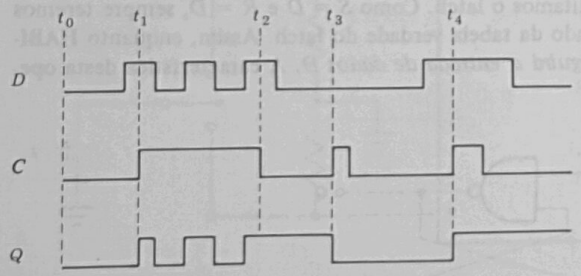


Fig. 4.5-2 Formas de onda que ilustram o funcionamento de um latch transparente.

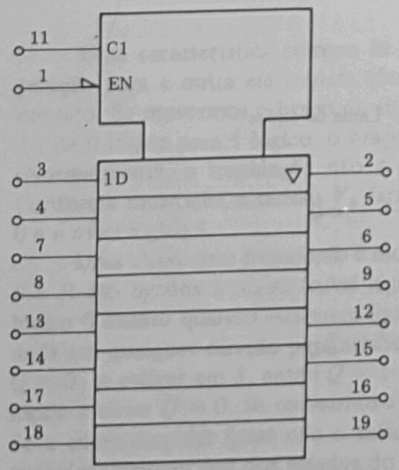


Fig. 4.5-3 Símbolo do latch óctuplo transparente tipo D '373.

O circuito integrado tipo 373 é um latch transparente tipo D óctuplo. Seu símbolo lógico é mostrado na Fig. 4.5-3. Cada um dos oito latches tem uma saída de três-estados, todas elas controladas por uma entrada HABILITA (*ENABLE*) ativa quando baixa. O sinal de entrada HABILITA (*ENABLE*) comum aqui é chamado C e o 1 que aparece no símbolo C1 indica que esta entrada *controla* as entradas que têm o mesmo número 1. Assim, o 1 no nome 1D indica que a entrada D é *controlada* por C1 (entrada de *clock* ou relógico).

4.6 SINCRONISMO (CLOCKING)

Uma característica universal das portas lógicas é que elas introduzem atrasos de propagação nos sinais transmitidos através delas. Estes atrasos podem variar de uma para outra amostra de um mesmo tipo de porta de um mesmo fabricante e podem, ainda, variar em uma mesma amostra devido ao envelhecimento, temperatura ou outras alterações ambientais. Sinais lógicos que passam através de muitas portas sofrem atrasos de propagação totais que dificilmente podem ser preditos e que estão sujeitos a consideráveis variações. Atrasos que não sejam apropriadamente levados em conta podem gerar sinais que vão para níveis lógicos diferentes dos pretendidos. Consideremos, por exemplo, a situação mostrada na Fig. 4.6-1. A entrada A inicialmente é $A = 0$ e vai para $A = 1$ em $t = t_0$. Quando $A = 0$, $B = 1$ e $C = 0$, de modo que $Z = 0$. Quando $A = 1$, teremos $B = 0$ e $C = 1$, de modo que, novamente, $Z = 0$. Mas suponhamos que, conforme indicado pela forma-de-onda, o atraso através da porta G1 seja maior que o atraso através da G2. Neste caso haverá um intervalo (igual à diferença entre os atrasos) em que B e C serão $B = C = 1$ e, durante este intervalo, teremos Z no nível não pretendido $Z = 1$. As excursões de sinais lógicos a níveis não pretendidos, ainda que brevemente, são chamadas *perigos* (*hazards*). Nos circuitos combinacionais (circuitos sem memória) os perigos são inconvenientes, mas, se forem suficientemente breves e os

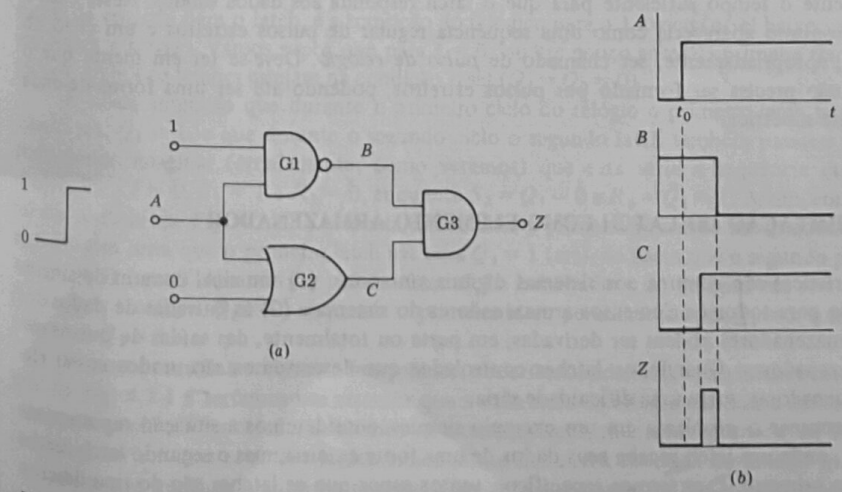


Fig. 4.6-1 (*a*) Um circuito lógico que gera um perigo. (*b*) Formas de onda.

circuitos que poderiam responder a eles forem suficientemente lentos, eles podem não causar preocupações sérias. Quando causam problemas, às vezes é possível usar processos relativamente simples para eliminá-los (veja Prob. 4.6-1). Vamos supor que tenhamos um sistema com memória, ou seja, com latches. Como já vimos, mesmo um breve nível incorreto nas entradas de set ou reset afeta o estado do latch e o estado incorreto correspondente pode persistir indefinidamente.

É, portanto, de grande utilidade construir um latch no qual o terminal de dados determina o estado eventual do latch, mas que possua a característica de que o exato momento de resposta aos dados seja determinado por um sinal auxiliar. Neste caso seria possível restringir a resposta do latch até o momento em que todos os níveis lógicos que pudessem influenciá-lo tivessem atingido seus níveis definitivos.

O latch controlado da Fig. 4.5-1 inclui (em princípio, pelo menos) o mecanismo restritivo requerido e serve como base para a operação *síncrona* de circuitos digitais. Em um sistema síncrono aplicamos à entrada HABILITA (*ENABLE*) um sinal que executa transições entre os níveis alto e baixo, habilitando e desabilitando as portas de isolação G1 e G2. Estas portas são mantidas desabilitadas até que os dados de entrada (S e R ou D) tenham atingido seu valor final apropriado, isto é, as portas são habilitadas quando os dados forem *válidos*.

Em um sistema digital que usa este registro retardado de dados em seus dispositivos de armazenagem, cada ciclo do sinal aplicado à HABILITA (*ENABLE*) avança o processamento digital um passo. A velocidade com que se dá o processamento é determinada pela velocidade com que ocorrem esses ciclos e o sinal HABILITA (*ENABLE*) é, apropriadamente, chamado de sinal de *relógio* ou de *sincronismo*. Na maioria dos sistemas os ciclos de sincronismo ocorrem regularmente como o tique-taque de um relógio e o termo é bastante apropriado. Como todo o sistema deve operar em sincronismo com o relógio, um sistema que o possui é chamado um *sistema síncrono*.

O tempo destinado a um ciclo do relógio depende da velocidade (especialmente dos atrasos de propagação) dos dispositivos físicos empregados. Dispositivos com atrasos de propagação mais longos requerem um ciclo de relógio correspondentemente mais longo. Pode ser conveniente manter o latch isolado tanto quanto possível em cada ciclo do relógio, habilitando as portas de isolação somente o tempo suficiente para que o latch responda aos dados válidos. Neste caso o sinal de sincronismo apareceria como uma seqüência regular de pulsos estreitos e um ciclo do sinal poderia, apropriadamente, ser chamado de *pulso de relógio*. Deve-se ter em mente que o sincronismo não precisa ser formado por pulsos estreitos, podendo até ser uma forma-de-onda completamente simétrica.

4.7 UMA LIMITAÇÃO DO LATCH COMO ELEMENTO ARMAZENADOR

Duas características são comuns aos sistemas digitais síncronos: (1) um sinal comum de sincronismo é usado para todos os elementos armazenadores do sistema e (2) as entradas de dados dos elementos armazenadores podem ser derivadas, em parte ou totalmente, das saídas de outros elementos armazenadores. Quando os latches controlados que descrevemos são usados como elementos armazenadores, surge uma dificuldade séria.

Para examinar o problema em um exemplo simples, consideremos a situação representada na Fig. 4.7-1, onde um latch recebe seus dados de uma fonte externa, mas o segundo latch recebe seus dados do primeiro. Para sermos específicos, vamos supor que os latches são do tipo descrito na Fig. 4.5-1*a*, onde a transição do sincronismo, isto é, a transição que permite a transferência dos

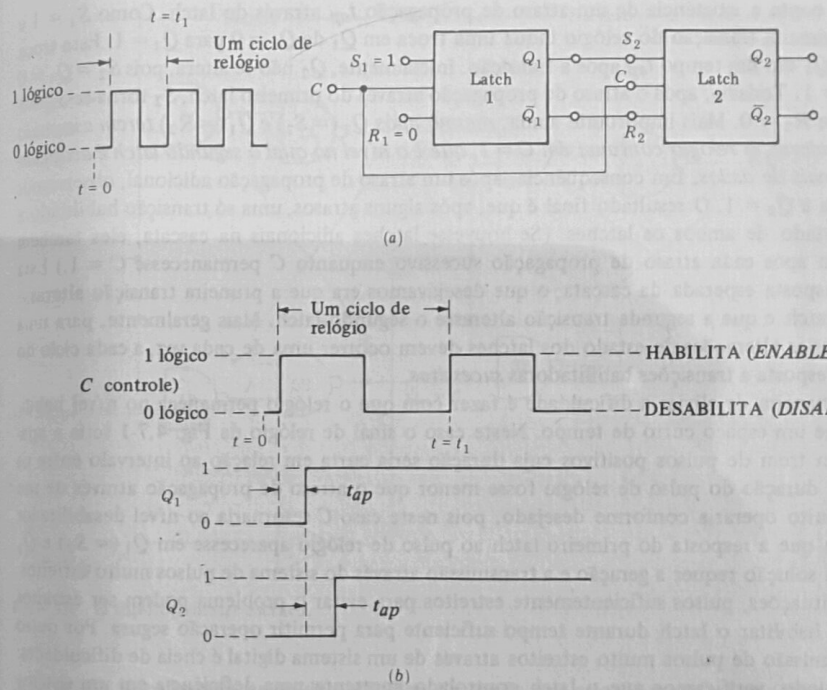


Fig. 4.7-1 (*a*) Uma cascata de dois latches. (*b*) Resposta da cascata ao sinal de relógio quando $S_1 = 1$ e $R_1 = 0$.

dados de entrada para o latch, é a transição do 0 lógico para o 1 lógico (nível baixo para nível alto em lógica positiva). Vamos supor que para $t = 0$, ou um pouco antes da primeira transição habilitadora, ambos os latches estejam na condição reset ($Q_1 = Q_2 = 0$).

É nossa intenção que durante o primeiro ciclo do relógio o primeiro latch passasse para o estado set, $Q_1 = 1$, e que durante o segundo ciclo o segundo latch também passasse para $Q_2 = 1$. Poderíamos imaginar (erradamente, como veremos) que esta seria a seqüência que realmente ocorre. Para $t = 0$, $S_1 = 1$ e $R_1 = 0$, enquanto $S_2 = Q_1 = 0$ e $R_2 = \bar{Q}_1 = 1$. Assim, como vemos da tabela verdade da Fig. 4.2-2, a resposta dos latches à primeira transição habilitadora do sinal de sincronismo seria que o primeiro latch iria para $Q_1 = 1$ (set), ao passo que o segundo permaneceria em $Q_2 = 0$ (reset). Quando da próxima transição habilitadora do relógio, em $t = t_1$, teríamos $S_2 = Q_1 = 1$ e $R_2 = \bar{Q}_1 = 0$; enquanto o primeiro latch permaneceria em $Q_1 = 1$, o segundo latch passaria a $Q_2 = 1$.

Se nosso esquema realmente funcionasse, poderíamos adicionar mais latches controlados aos dois da Fig. 4.7-1 e teríamos um circuito que a cada ciclo do relógio deslocaria dados ao longo da cascata de latches. Esta cascata de elementos armazenadores, chamada *registrador de deslocamento* (*shift register*), é uma estrutura lógica básica muito útil e será discutida novamente na Seção 4.17.

Podemos verificar, conforme mostrado nas formas-de-onda da Fig. 4.7-1*b*, que a seqüência antecipada não ocorre. Aqui desenhamos os sinais do relógio e dos níveis lógicos de Q_1 e Q_2,

levando em conta a existência de um atraso de propagação t_{ap} através do latch. Como $S_1 = 1$ e $R_1 = 0$, a primeira transição do relógio induz uma troca em Q_1 de $Q_1 = 0$ para $Q_1 = 1$. Esta troca aparece em Q_1 em um tempo t_{ap} após a transição. Inicialmente, Q_2 não se altera, pois $S_2 = Q_1 = 0$ e $R_2 = \bar{Q}_1 = 1$. Todavia, após o atraso de propagação através do primeiro latch, S_2 torna-se $S_2 = 1$ e R_2 torna-se $R_2 = 0$. Mais importante ainda, *mesmo após* $Q_1 (= S_2)$ *e* $\bar{Q}_1 (= R_2)$ *terem assumido seus novos valores, o relógio continua em* $C = 1$, *que é o nível no qual o segundo latch é acoplado a seus terminais de dados*. Em conseqüência, após um atraso de propagação adicional, observamos que Q_2 passa a $Q_2 = 1$. O resultado final é que, após alguns atrasos, uma só transição habilitadora alterou o estado de ambos os latches. (Se houvesse latches adicionais na cascata, eles também responderiam após cada atraso de propagação sucessivo enquanto C permanecesse $C = 1$.) Esta não era a resposta esperada da cascata; o que desejávamos era que a primeira transição alterasse o primeiro latch e que a segunda transição alterasse o segundo latch. Mais geralmente, para uma cascata longa, as alterações de estado dos latches devem ocorrer uma de cada vez, a cada ciclo do relógio, em resposta a transições habilitadoras *sucessivas*.

Uma maneira de aliviar a dificuldade é fazer com que o relógio permaneça no nível habilitador durante um espaço curto de tempo. Neste caso o sinal de relógio da Fig. 4.7-1 teria a aparência de um trem de pulsos positivos cuja duração seria curta em relação ao intervalo entre os pulsos. Se a duração do pulso de relógio fosse menor que o atraso de propagação através de um latch, o circuito operaria conforme desejado, pois neste caso C retornaria ao nível desabilitador $C = 0$ antes que a resposta do primeiro latch ao pulso de relógio aparecesse em $Q_1 (= S_2)$ e $\bar{Q}_1 (= R_2)$. Esta solução requer a geração e a transmissão através do sistema de pulsos muito estreitos. Em muitas situações, pulsos suficientemente estreitos para evitar o problema podem ser estreitos demais para habilitar o latch durante tempo suficiente para permitir operação segura. Por outro lado, a transmissão de pulsos muito estreitos através de um sistema digital é cheia de dificuldades.

Resumindo, verificamos que o latch controlado apresenta uma deficiência em um sistema síncrono, pois o latch tem sua entrada de dados derivada inteiramente, ou em parte, da saída de outros latches. Esta deficiência está obviamente relacionada com o fato de o latch controlado ser transparente. Temos, pois, problemas quando tentamos, em um mesmo período do relógio, *ler* o dado previamente armazenado no latch e *escrever* um novo dado no latch. Por outro lado, se desejamos um dispositivo de armazenamento no qual desejamos escrever e o qual desejamos ler, mas não ambos no mesmo ciclo de relógio, o latch é perfeitamente satisfatório. Nas próximas seções consideraremos circuitos que têm a característica que falta ao latch. Estes circuitos são chamados *flip-flops*. (multivibrador biestável)

4.8 O FLIP-FLOP MESTRE-ESCRAVO

Um tipo de dispositivo armazenador síncrono que não é transparente é o flip-flop mestre-escravo da Fig. 4.8.1. Consiste em dois latches individuais do tipo mostrado na Fig. 4.5-1. Um latch é chamado *mestre* (*master*) e o outro *escravo* (*slave*). Há, adicionalmente, portas de *entrada* de dados, 1A e 1B, e um segundo conjunto de portas de *acoplamento*, 3A e 3B, através das quais, quando habilitadas, os dados do mestre podem ser transferidos ao escravo. O sinal de sincronismo aplicado às portas de entrada é complementar ao aplicado às portas de acoplamento. Quando o sinal de sincronismo está no nível que habilita as portas de entrada, os dados de entrada nos terminais S e R são registrados no latch mestre. Todavia, quando as portas de entrada estão habilitadas, as de transferência estão desabilitadas, de modo que não há transferência do mestre para o escravo. Quando as portas de acoplamento são habilitadas, os dados são transferidos do mestre para o escravo, mas, como neste momento as portas de entrada estão desabilitadas, novos dados não são registrados no mestre.

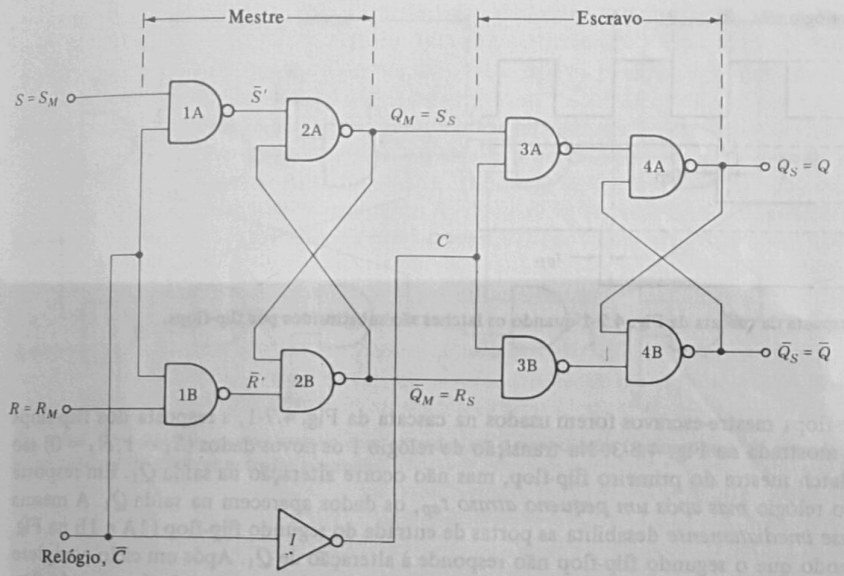


Fig. 4.8-1 O flip-flop mestre escravo.

Enquanto o relógio sobe do nível baixo para o alto, para habilitar as portas de entrada, as portas de acoplamento devem ser desabilitadas *antes* que as portas de entrada permitam a transferência para o mestre, para evitar a transferência de dados para a saída. Além disso, quando o relógio desce do nível alto para o baixo, as portas de entrada devem ser desabilitadas *antes* que as portas de acoplamento sejam habilitadas. Assim, os níveis de tensão em que as portas de entrada e de acoplamento são habilitadas e desabilitadas são ajustados de maneira que a seqüência de eventos seja a representada na Fig. 4.8-2. Um ciclo de relógio é mostrado com tempos de subida e descida finitos para que níveis de tensão diferentes sejam atingidos em tempos diferentes.

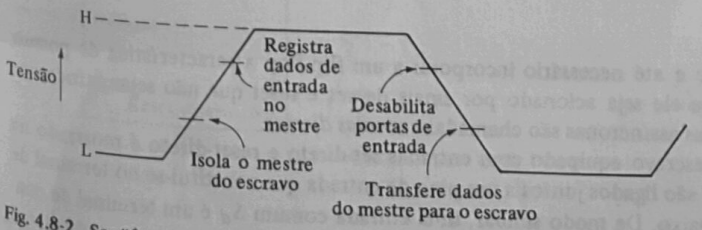


Fig. 4.8-2 Seqüência de eventos em um flip-flop mestre-escravo durante um ciclo de relógio.

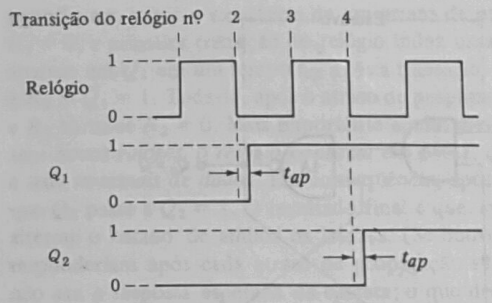


Fig. 4.8-3 Resposta da cascata da Fig. 4.7-1 quando os latches são substituídos por flip-flops.

Se flip-flops mestre-escravos forem usados na cascata da Fig. 4.7-1, a resposta dos flip-flops seria aquela mostrada na Fig. 4.8-3. Na transição de relógio 1 os novos dados ($S_1 = 1, R_1 = 0$) são escritos no latch mestre do primeiro flip-flop, mas não ocorre alteração na saída Q_1. Em resposta à borda 2 do relógio *mas após um pequeno atraso* t_{ap}, os dados aparecem na saída Q_1. A mesma borda 2 *quase imediatamente* desabilita as portas de entrada do segundo flip-flop (1A e 1B na Fig. 4.8-1), de modo que o segundo flip-flop não responde à alteração de Q_1. Após um ciclo completo do relógio, os novos dados terão sido escritos no primeiro flip-flop, mas o estado do segundo flip-flop corresponde aos dados ($Q_1 = S_2 = 0, \overline{Q}_1 = R_2 = 1$) mantidos no primeiro flip-flop antes do início do ciclo do relógio que contém as bordas 1 e 2. Do mesmo modo, o próximo ciclo com as bordas 3 e 4 transferirá os dados de entrada ($S_1 = 1, R_1 = 0$) para o segundo flip-flop.

Em outras palavras, o flip-flop mestre-escravo tem performance aceitável porque a alteração na saída, se existir, ocorre em resposta à transição do relógio que *desabilita as portas de entrada*. Assim, a borda do relógio que atualiza os terminais de saída do flip-flop em relação aos dados armazenados no mesmo torna o flip-flop insensível aos novos dados presentes à sua entrada.

Note-se que a resposta da saída do flip-flop, se houver, ocorre nominalmente na transição negativa (do nível alto para o baixo) do sinal de relógio. A convenção, neste caso, é denominar o sinal de relógio de $\overline{\text{relógio}}$, como na Fig. 4.8-1, reservando a designação relógio (sem barra) para o caso em que a resposta é à transição positiva.

A tabela verdade para um flip-flop mestre-escravo *SR* é idêntica à tabela verdade do latch *SR* da Fig. 4.2-2. No caso em questão, os símbolos Q_n e Q_{n+1} referem-se, respectivamente, aos valores de Q nos intervalos n e $(n + 1)$ do relógio.

Entradas Diretas

Muitas vezes é conveniente e até necessário incorporar a um flip-flop a característica de possuir entradas que permitam que ele seja acionado por sinais de set e reset que não sejam síncronos com o relógio. Estas entradas assíncronas são chamadas *entradas diretas*.

Um flip-flop mestre-escravo equipado com entradas set-direto e reset-direto é mostrado na Fig. 4.8-4. Os terminais $\overline{R}_d$ são ligados juntos a um pino de entrada que constitui-se no terminal de reset-direto ativo quando baixo. De modo similar, uma entrada comum $\overline{S}_d$ é um terminal de controle ativo quando baixo. Quando $\overline{R}_d = 0$ e $\overline{S}_d = 1$, a saída Q do flip-flop vai para o estado $Q = 0$

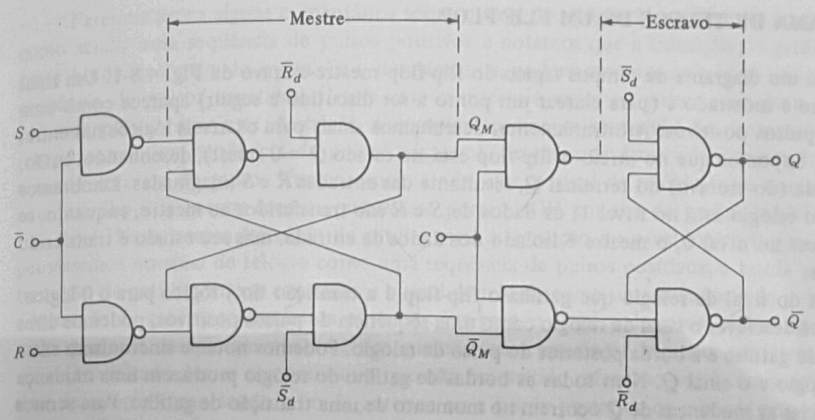


Fig. 4.8-4 Um flip-flop mestre escravo com entradas set e reset diretas (assíncronas). As entradas diretas prevalecem totalmente sobre as entradas síncronas.

e as entradas de dados S e R e a presença do relógio não têm efeito sobre a saída. Correspondentemente, $\overline{R}_d = 1$ e $\overline{S}_d = 0$ levam o flip-flop ao estado set. A condição $\overline{S}_d = \overline{R}_d = 0$ não deve ser usada e, quando $\overline{S}_d = \overline{R}_d = 1$, o flip-flop pode responder às entradas de dados S e R sincronamente com o relógio. As portas AND extras que não são encontradas na Fig. 4.8-1 servem para evitar uma resposta, ainda que transitória, à entrada de relógio; isto significa que os controles diretos *prevalecem* totalmente sobre as entradas síncronas (ver Prob. 4.8-1). O símbolo do flip-flop é mostrado na Fig. 4.8-5.

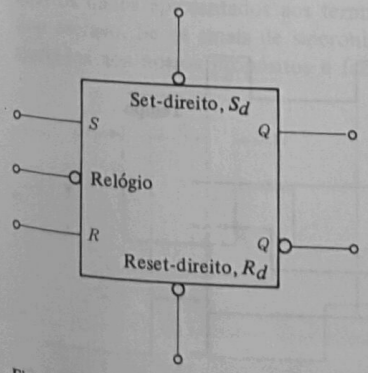


Fig. 4.8-5 Símbolo lógico para o flip-flop da Fig. 4.8-4.

4.9 DIAGRAMA DE TEMPOS DE UM FLIP-FLOP

A Fig. 4.9-1 é um diagrama de tempos típico do flip-flop mestre-escravo da Fig. 4.8-1. Um sinal de sincronismo é mostrado e (para clarear um ponto a ser discutido a seguir) aparece como uma seqüência de pulsos positivos. Arbitrariamente, desenhamos sinais para os níveis lógicos nas entradas de dados. Supomos que no início o flip-flop está no estado $Q = 0$ (reset), desenhando, então, o sinal de saída (do escravo) no terminal Q, resultante das entradas R e S imaginadas. Lembramos que, quando o relógio está no nível 1, os dados de S e R são transferidos ao mestre, enquanto, se o relógio estiver no nível 0, o mestre é isolado dos dados de entrada, mas seu estado é transferido para o escravo.

A borda do sinal de relógio que gatilha o flip-flop é a transição do 1 lógico para o 0 lógico. Se resolvermos descrever o sinal de relógio como uma seqüência de pulsos positivos, podemos dizer que a borda de gatilho é a borda posterior do pulso de relógio. Podemos notar o sincronismo entre o sinal de relógio e o sinal Q. Nem todas as bordas de gatilho do relógio produzem uma mudança em Q, mas todas as mudanças de Q ocorrem no momento de uma transição de gatilho. Para sermos mais exatos, o diagrama de tempos deveria mostrar um pequeno atraso de propagação entre as alterações no sinal Q e a transição de gatilho, mas, para simplificar, ignoramos este atraso na figura.

Examinando o diagrama de tempos, observamos o seguinte: quando ocorre a transição de gatilho ao final do pulso 1 temos $S = 1$ e $R = 0$, portanto o flip-flop irá para o estado set $Q = 1$. Entre a borda posterior do pulso 1 e a borda anterior do pulso 2 há um intervalo em que tanto S como R têm nível lógico 1. Normalmente isto não é permitido, mas, como ocorre em um período em que as entradas do flip-flop estão desabilitadas, não nos causa nenhum problema.

Quando do início do pulso 2 e no intervalo que segue, nós obtemos $S = 0$ e $R = 1$. Como resultado, o flip-flop *mestre* será reajustado (reset) pela borda *anterior* do pulso 2. Mais tarde, durante este pulso, R retorna a $R = 0$. Todavia, esta alteração de R não afetará o estado do flip-flop mestre, que continuará reajustado (reset). Quando da borda de gatilho do pulso 2, o estado reset do mestre será transferido para o escravo, conforme indicado na figura para Q.

Durante o pulso 3 temos $S = 1$ e $R = 0$. Assim, quando da transição de gatilho do pulso 3, notamos que Q passa para $Q = 1$. Durante o pulso 4, e quando de sua transição de gatilho, temos $S = R = 0$. A transição de gatilho não produz, portanto, alteração e Q permanece em $Q = 1$. Finalmente, durante a transição de gatilho do pulso 5, Q passa a $Q = 0$, pois $S = 0$ e $R = 1$.

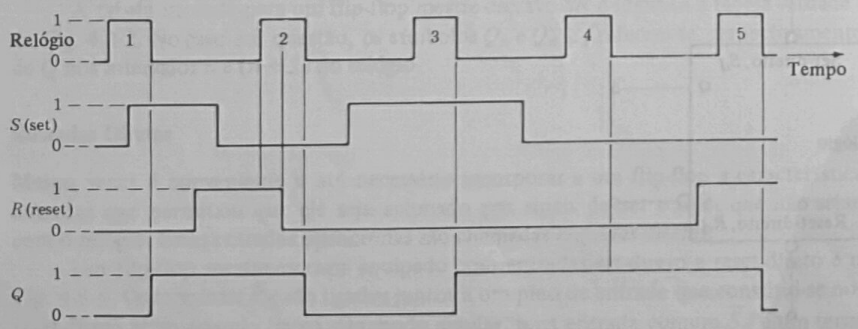


Fig. 4.9-1 Diagrama de tempos para o flip flop mestre escravo da Fig. 4.8-1.

Faremos agora alguns comentários sobre o sinal de relógio. Nos referimos ao sinal de relógio como sendo uma seqüência de pulsos positivos e notamos que a transição de gatilho do relógio era a borda posterior do pulso. Se o desejássemos, poderíamos nos referir ao relógio como uma seqüência de pulsos negativos. (Neste caso o intervalo entre os pulsos seria mais curto que os próprios pulsos, mas esta descrição é complementamente razoável.) Se pensarmos no sinal de relógio como uma seqüência de pulsos negativos, deveríamos dizer que a transição de gatilho é a borda *anterior* do pulso. Vamos supor, ainda, que o sinal de relógio fosse usado em conexão com um flip-flop mestre-escravo que usasse portas NOR em vez de portas NAND, como o da Fig. 4.8-1. Neste caso a transição de gatilho seria a transição do nível lógico 0 para o nível 1. Se pensássemos no sinal de relógio como uma seqüência de pulsos positivos, a borda anterior seria a transição de gatilho, mas se olhássemos o sinal de relógio como uma seqüência de pulsos negativos, a transição de gatilho seria a borda posterior. Em grande número de casos, o sinal de relógio é uma onda quadrada completamente simétrica que não tem a aparência de pulsos negativos ou positivos.

O resultado destes comentários é que não faz sentido referirmo-nos à transição de gatilho do sinal de relógio como sendo a borda anterior ou a borda posterior ou a borda positiva ou a borda negativa. O essencial é que a resposta do flip-flop é síncrona com a transição do relógio que leva o flip-flop da condição em que está acoplado aos terminais de entrada de dados à condição em que está desacoplado. Não é necessário, todavia, que sejamos rigorosos sobre isto; desde que se mantenha claro o que é essencial, pode ser útil, conveniente e sucinto falar sobre o pulso de relógio e suas bordas anteriores e posteriores.

4.10 SINCRONISMO DE DUAS FASES

Cada flip-flop mestre-escravo, como o da Fig. 4.8-1, requer um inversor para que os sinais de relógio aplicados ao mestre e ao escravo sejam complementares. Num sistema digital grande, onde são usados muitos flip-flops e, conseqüentemente, muitos inversores, pode ser vantajoso dispensar o uso de inversores individuais. Isto pode ser conseguido fazendo a inversão uma vez e levando os dois sinais complementares de relógio a cada flip-flop, um para o mestre e o outro para o escravo.

Examinando novamente a Fig. 4.8-1, verificamos que não deve haver um mecanismo pelo qual os dados apresentados aos terminais de dados possam ser transferidos diretamente ao flip-flop escravo. Se os sinais de sincronismo forem gerados externamente aos flip-flops, poderemos ajustá-los aos nossos propósitos e fazer com que não haja possibilidade de tal comunicação. Os

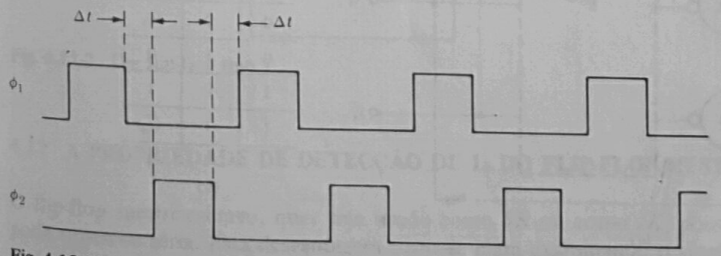


Fig. 4.10-1 O sinal de um relógio de duas fases.

dois sinais de relógio gerados com esta consideração em vista geralmente têm a forma indicada na Fig. 4.10-1 e são denominados sinal de relógio de fase 1 (ϕ_1) e de fase 2 (ϕ_2). Observe-se que com estes sinais as portas 1A e 2A da Fig. 4.8-1 e as portas 3A e 3B nunca serão habilitadas ao mesmo tempo. Observe-se ainda, na Fig. 4.10-1, que existe uma margem de segurança Δt entre o momento em que um conjunto de portas é desabilitado e o momento em que o outro conjunto é habilitado.

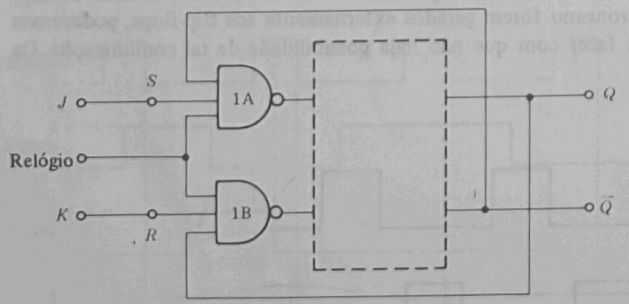
4.11 O FLIP-FLOP *JK*

Até agora temos evitado fazer S e R tal que $S = R = 1$, pois tal procedimento tentaria ajustar (set) e reajustar (reset) o flip-flop ao mesmo tempo, e o resultado seria ambíguo. Vamos modificar o flip-flop para permitir $S = R = 1$ e observaremos que o flip-flop modificado possui a propriedade que, quando $S = R = 1$, ele chaveia, isto é, muda de estado a cada transição de gatilho do relógio.

O flip-flop modificado é mostrado na Fig. 4.11-1. O retângulo tracejado representa o flip-flop completo, exceto pelas portas de entrada que são mostradas explicitamente. A modificação consiste em prover terminais adicionais nas portas de entrada e fazer ligações entre as saídas e as entradas conforme mostrado. O terminal de dados anteriormente chamado S é, agora, chamado J, e o terminal de dados R é chamado K. Na ausência desta modificação, os níveis lógicos em S e R "dirigiam" o sinal de relógio, isto é, dependendo de S e R, uma ou outra das portas de entrada 1A e 1B era habilitada e o relógio ajustava ou reajustava (set ou reset) o flip-flop. A razão da modificação é fazer com que a "direção" do sinal de relógio seja determinada não só por S e R, mas também pelo estado do flip-flop.

A operação do flip-flop *JK* pode ser determinada da tabela verdade da Fig. 4.11-1*b*. Sejam $J = K = 0$. Então ambas as portas de entrada estarão desabilitadas e o relógio não altera o estado do flip-flop; conseqüentemente, $Q_{n+1} = Q_n$.

A seguir, vamos supor que $J = 0$ e $K = 1$ e que o flip-flop esteja no estado $Q = 0$. Como $J = 0$, a porta 1A fica desabilitada do mesmo modo que 1B fica desabilitada porque $Q = 0$. Assim, o relógio não altera o estado do flip-flop. Por outro lado, se $J = 0$ e $K = 1$, mas o flip-flop estiver



(*a*)

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\bar{Q}_n$

(*b*)

Fig. 4.11-1 (*a*) O flip-flop *JK*. (*b*) A tabela verdade.

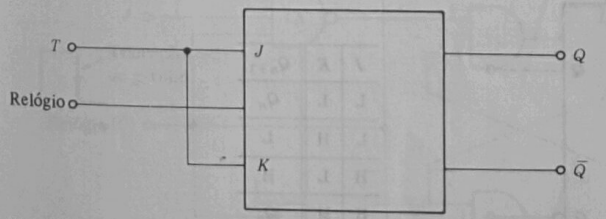
no estado $Q = 1$, a porta 1B será habilitada e o relógio causará a transferência do flip-flop para o estado reset $Q = 0$. De modo similar, com $J = 1$ e $K = 0$, o relógio transferirá o flip-flop para o estado $Q = 1$ se ele já não se encontrar neste estado.

Finalmente, façamos $J = K = 1$. Neste caso, qual das portas 1A e 1B será habilitada dependerá inteiramente de Q_n e de $\bar{Q}_n$, isto é, do estado do flip-flop. Se $Q_n = 0$, a porta 1A será habilitada e o relógio levará o flip-flop ao estado $Q_{n+1} = 1$. Se $Q_n = 1$, o relógio levará o flip-flop ao estado $Q_{n+1} = 0$. Assim, a cada ciclo do sinal de relógio, o estado do flip-flop se alterará. A palavra *chave* (*toggle*) é usada para descrever esta mudança de estado causada por cada ciclo do relógio. Na tabela verdade isso é especificado por $Q_{n+1} = \bar{Q}_n$ para $J = K = 1$. Note-se que, ao contrário dos tipos anteriores *SR*, não existe na tabela verdade nenhuma combinação de entradas marcada "não usada".

O flip-flop *JK* é uma configuação em que a saída de um flip-flop é ligada à entrada de um flip-flop. Neste caso acontece que a saída e a entrada pertencem ao mesmo flip-flop. Mesmo assim já observamos que quando tal conexão é feita não devemos usar latches transparentes. Seria de interesse pesquisarmos o que ocorreria se fizéssemos ligações da saída para a entrada como as da Fig. 4.11-1 usando o latch da Fig. 4.5-1. Se o latch estivesse no estado $Q = 0$, $\bar{Q} = 1$ com HABILITA (*ENABLE*) alta, a porta G1 seria habilitada e G2 desabilitada; o latch iria para o estado $Q = 1$, $\bar{Q} = 0$. Neste estado G1 seria desabilitada e G2 habilitada, e o latch retornaria ao estado $Q = 0$, $\bar{Q} = 1$ e recomeçaria o ciclo. Como resultado, o latch oscilaria entre os dois estados. Afora a desvantagem associada a esta oscilação, o estado ao qual o latch retornaria quando HABILITA (*ENABLE*) baixasse seria imprevisível.

Flip-Flop Tipo *T*

Um flip-flop com uma única entrada de controle que chaveia quando o controle for ativo e não responde quando o controle for inativo é chamado flip-flop tipo T (*toggle*). Conforme mostrado na Fig. 4.11-2, um flip-flop tipo T pode ser construído ligando juntos os terminais J e K de um flip-flop *JK*. O terminal comum T é o terminal de controle. O flip-flop chaveia quando $T = 1$.



(multivibrador monoestável)

Fig. 4.11-2 Um flip-flop tipo *T*.

4.12 A PROPRIEDADE DE DETECÇÃO DE 1s DO FLIP-FLOP MESTRE-ESCRAVO

O flip-flop mestre-escravo, quer seja usado como *SR* ou como *JK*, possui uma desvantagem que pode tornar-se séria. Esta desvantagem torna-se clara examinando o sinal da Fig. 4.12-1. Já vimos que o latch mestre é ligado aos terminais de dados quando o relógio é alto e que a resposta de

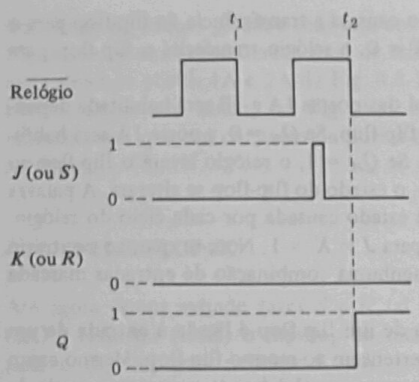
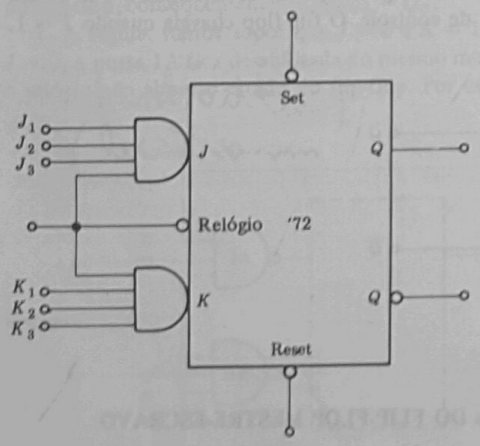


Fig. 4.12-1 Sinais ilustrando a propriedade de detecção de 1s do flip-flop mestre-escravo.

saída, se houver, ocorre na transição negativa do relógio nos tempos t_1 e t_2. Durante o primeiro ciclo do relógio, tudo funciona a contento. Temos $Q = 0$ e, como $J = K = 0$, não deve haver alteração em Q em t_1, o que em realidade acontece. Todavia, durante o segundo ciclo do relógio, em t_2, temos, novamente, $J = K = 0$, mas Q passa para $Q = 1$. A razão deste comportamento é que, enquanto o relógio estava alto, J permaneceu brevemente em $J = 1$. Como resultado, o latch mestre foi para o estado set. Com o retorno de J para $J = 0$, o latch mestre *permanece no estado set*. Em t_2 o estado do mestre é transferido para o escravo e Q passa para $Q = 1$. Se K houvesse excursionado brevemente ao nível 1, em t_2 a saída teria respondido passando para $Q = 0$, caso já



J	K	Q_{n+1}
L	L	Q_n
L	H	L
H	L	H
H	H	$\overline{Q}_n$

$J = J_1 \cdot J_2 \cdot J_3$
$K = K_1 \cdot K_2 \cdot K_3$

(b)

Fig. 4.12-2 (*a*) Símbolo lógico do flip-flop mestre-escravo tipo '72. (*b*) Tabela verdade em termos de níveis de tensão H (alto) e L (baixo).

não se encontrasse naquele nível. Se ambos, J e K, houvessem excursionado ao nível 1, o flip-flop chavearia mesmo que ao tempo da resposta, t_2, tivéssemos $J = K = 0$.

Verifica-se, portanto, que a resposta do flip-flop é determinada por qual entrada esteve por último no nível lógico 1, enquanto as portas de entrada estavam habilitadas. Esta característica é *propriedade de detecção de 1s* do flip-flop. (Se o flip-flop tivesse sido construído com portas NOR seria a detecção de 0s.) Esta característica pode, muitas vezes, ser inaceitável, pois torna o flip-flop suscetível a perigos e a distúrbios espúrios (ruído).

Um flip-flop *JK* mestre-escravo típico dos disponíveis comercialmente em integração de pequena escala é o tipo '72. Seu símbolo lógico e tabela verdade são mostrados na Fig. 4.12-2. Há três terminais de dados J e três K e o dispositivo responde a $J = J_1 \cdot J_2 \cdot J_3$ e a $K = K_1 \cdot K_2 \cdot K_3$. A saída responde à borda negativa do pulso de relógio. Entradas de controle assíncronas (diretas) de set e reset estão disponíveis. O tipo '73 é uma unidade dupla com apenas uma entrada J e uma K e somente uma entrada de set-direto para cada flip-flop.

4.13 UM FLIP-FLOP JK GATILHADO PELA BORDA

A dificuldade do uso de um latch em sistemas síncronos é seguidamente chamada o *problema da corrida* pela seguinte razão. Consideremos o caso em que os dados de entrada de um latch são a saída de um outro latch L_1, como na Fig. 4.7-1. Quando o relógio habilita as portas de entrada de ambos os latches, L_2 responde a seus dados de entrada, e o mesmo ocorre com L_1. Assim, logo após a habilitação das portas de entrada, os dados apresentados a L_2 mudam. Já vimos que uma solução para o problema é habilitar os latches com um pulso de relógio de curta

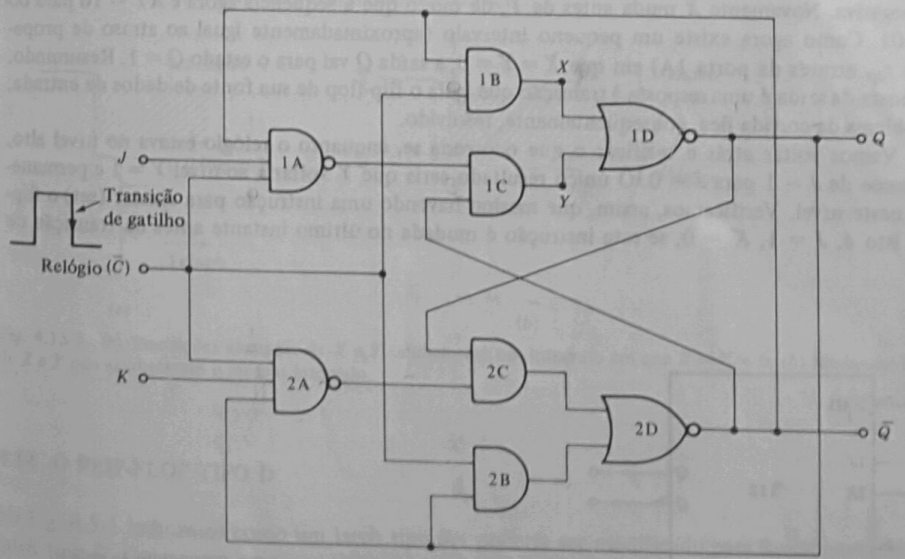


Fig. 4.13-1 Um flip-flop *JK* gatilhado pela borda.

duração, esperando desabilitar as portas antes que os dados de entrada possam mudar. Neste caso temos uma *corrida* entre a velocidade com que podemos desabilitar as portas e a velocidade com que os dados de entrada podem mudar. Já vimos como o flip-flop mestre-escravo resolve o problema da corrida. Ele o faz forçando a *mudança de saída do flip-flop, se houver, a ser a resposta à borda do pulso de relógio que desabilita o flip-flop a qualquer resposta aos dados de entrada*. O flip-flop mestre-escravo resolve o problema da corrida mas nos deixa com o problema da detecção de 1s. O problema da detecção de 1s, por sua vez, é resolvido projetando um flip-flop *gatilhado pela borda*. Neste tipo de flip-flop a resposta de saída é a resposta à entrada, mas somente aos dados presentes *imediatamente antes* da transição de gatilho do sinal de relógio.

Como verificaremos a seguir, o flip-flop da Fig. 4.13-1 resolve o problema da corrida e, também, o da detecção de 1s, sendo, pois, um flip-flop gatilhado pela borda tipo *JK*. A transição de gatilho, conforme indicado, é a borda negativa do relógio. A operação do flip-flop usa, com vantagem, o fato de que existe um atraso de propagação através de uma porta.

Consideremos que o flip-flop está no estado $Q = 0, \overline{Q} = 1$ e que $J = 1$ e $K = 0$ e vamos seguir a operação durante um ciclo completo do relógio do nível baixo até o alto e de volta ao nível baixo. Antecipamos, obviamente, que, em resposta à transição de gatilho, o flip-flop irá para o estado $Q = 1$.

Como pode facilmente ser verificado, inicialmente o terminal X está em 0 e o terminal Y está em 1. Quando o sinal de relógio subir, X irá para 1 e Y irá para 0. Para efetuar estas alterações, o sinal de relógio tem que se propagar através de uma porta (1B) para chegar a X, mas através de duas portas (1A e 1C) para chegar a Y. Conseqüentemente, a troca de estado em X ocorre *antes* da troca em Y. A seqüência de níveis na entrada da porta 1D é $XY = 01$ para 11 para 10. Como X ou Y é sempre 1, o efeito líquido é que a saída da porta 1D permanece em $Q = 0$ e não há alteração no estado do flip-flop. Observemos o que ocorre quando o relógio executa sua transição negativa. Novamente X muda antes de Y, de modo que a seqüência agora é $XY = 10$ para 00 para 01. Como agora existe um pequeno intervalo (aproximadamente igual ao atraso de propagação t_{ap} através da porta 1A) em que $X = Y = 0$, a saída Q vai para o estado $Q = 1$. Resumindo, a resposta da saída é uma resposta à transição que isola o flip-flop de sua fonte de dados de entrada; o problema da corrida fica, conseqüentemente, resolvido.

Vamos voltar atrás e verificar o que ocorreria se, enquanto o relógio estava no nível alto, J passasse de $J = 1$ para $J = 0$. O único resultado seria que Y voltaria ao nível $Y = 1$ e permaneceria neste nível. Verificamos, assim, que mesmo havendo uma instrução para ajustar (set) o flip-flop, isto é, $J = 1$, $K = 0$, se esta instrução é mudada no último instante antes da transição de

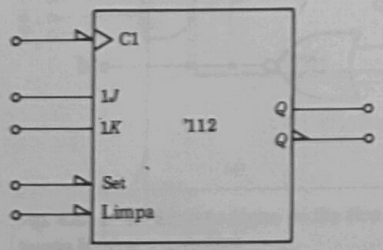


Fig. 4.13-2 Símbolo lógico para o flip-flop gatilhado pela borda tipo '112.

gatilho, o flip-flop responde à instrução atualizada. O flip-flop é, pois, um dispositivo gatilhado pela borda e a dificuldade da detecção de 1s é contornada.

O tipo '112 é um exemplo típico de CI flip-flop gatilhado pela borda que usa o circuito da Fig. 4.13-1. O CI é um flip-flop duplo e o novo símbolo lógico para qualquer uma das duas unidades é mostrado na Fig. 4.13-2. A inicial C representa o relógio (*clock*). A meia-flecha na linha de entrada indica que a transição de gatilho é a borda negativa do sinal de relógio. A marca triangular na entrada de relógio indica que o flip-flop é *gatilhado pela borda*. Conforme indicado, o flip-flop tem entradas de controle set-direto e reset-direto ativas quando baixas. O 1 que aparece em C1 e em $1J$ e $1K$ indica que o relógio controla as entradas J e K. Em geral, um número que aparece no *final* de um identificador indica que o identificador se refere a um sinal *controlador*, enquanto um número que aparece como *primeiro* símbolo em um identificador identifica um sinal que é *controlado*. O simbolismo da Fig. 4.13-2 indica, pois, que o momento exato em que as entradas J e K são efetivas é controlado pelo relógio.

Discutiremos agora uma dificuldade que pode ocorrer com um flip-flop gatilhado pela borda como o da Fig. 4.13-1. Como já foi visto, para que a operação seja apropriada, é necessário que exista um instante em que $X = Y = 0$ para que Q possa passar a $Q = 1$. Se as transições do sinal de relógio entre os níveis lógicos forem abruptas, a situação é como a representada na Fig. 4.13-3*a*. Em determinado momento, X cai subitamente de 1 para 0, passando rapidamente por uma região de ambigüidade (sombreada) em que não assume valor lógico 1 ou 0. Após um intervalo t_{ap}, Y sobe de maneira igualmente abrupta e não há ambigüidade sobre o fato de que existe um intervalo em que $X = Y = 0$, de modo que Q pode passar a $Q = 1$. Na Fig. 4.13-3*b* desenhamos X e Y superpostas levando em conta os tempos de subida e descida finitos destes sinais. Observamos que não há um intervalo onde $X = Y = 0$ sem ambigüidade, o que conduz ao resultado que se acionarmos o flip-flop com um sinal de relógio que executa transições lentas entre os níveis lógicos é possível que o flip-flop não opere de modo apropriado.

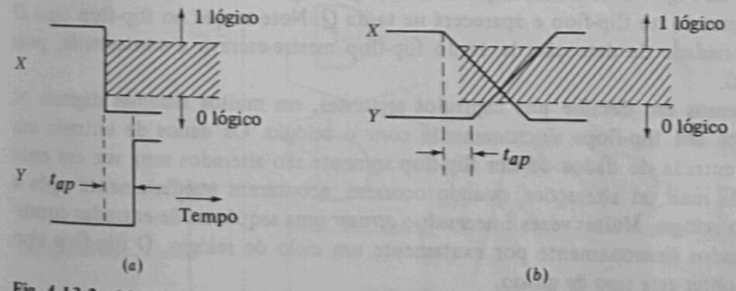


Fig. 4.13-3 (*a*) Transições abruptas de X e Y estabelecem um intervalo em que $X = Y = 0$. (*b*) Mudanças lentas de X e Y não estabelecem o mesmo intervalo.

4.14 O FLIP-FLOP TIPO D

Na Fig. 4.5-1 indicamos como um latch tipo *SR* poderia ser modificado para funcionar como um latch tipo *D*. Do mesmo modo, um flip-flop tipo *SR* pode ser modificado para tornar-se um flip-flop tipo *D*. Se tomarmos o flip-flop mestre-escravo da Fig. 4.8-1 e aplicarmos o sinal da linha de

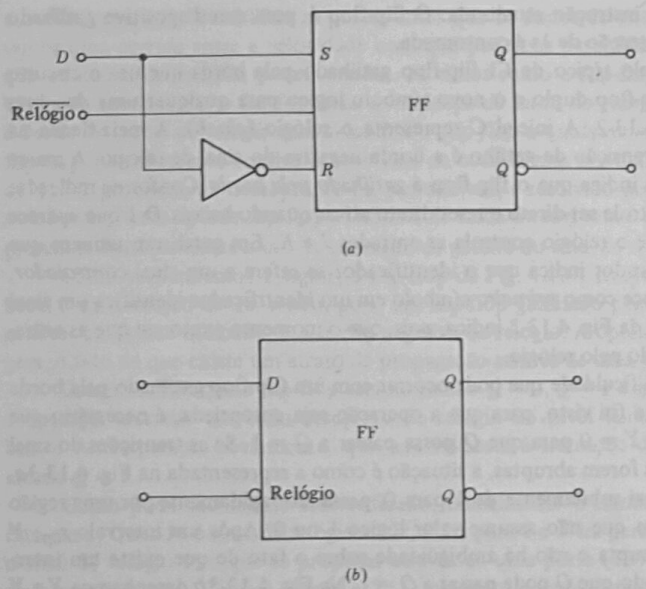


Fig. 4.14-1 (*a*) Um flip-flop *SR* alterado para funcionar como um flip-flop tipo *D*. (*b*) Seu símbolo lógico.

dados a S e seu complemento a R, como indicado na Fig. 4.14-1*a*, teremos um flip-flop tipo D com o símbolo lógico da Fig. 4.14-1*b*. Em resposta à transição negativa do relógio, o nível lógico da entrada D será registrado no flip-flop e aparecerá na saída Q. Note-se que no flip-flop tipo D da Fig. 4.14-1 a propriedade de detecção de 1s do flip-flop mestre-escravo é *contornada*, pois nunca temos $S = R = 0$.

Como examinaremos em detalhe nos capítulos seguintes, em muitos sistemas digitais os dados são apresentados aos flip-flops sincronamente com o relógio. Os dados de entrada em qualquer terminal de entrada de dados de um flip-flop somente são alterados uma vez em cada ciclo do relógio. Ainda mais, as alterações, quando ocorrem, acontecem imediatamente após a transição de gatilho do relógio. Muitas vezes é necessário *atrasar* uma seqüência de entradas constituída de bits apresentados sincronamente por exatamente um ciclo de relógio. O flip-flop tipo D pode ser usado para obter este tipo de atraso.

Os sinais que ilustram esta operação de atraso são mostrados na Fig. 4.14-2. Fizemos parecer que a transição de gatilho do relógio ocorre exatamente quando os dados estão mudando. Se isso realmente acontecesse, a resposta do flip-flop seria ambígua. Na prática, as alterações nos dados ocorrem imediatamente *após* a transição de gatilho. O atraso entre a transição de gatilho e a mudança nos dados seria da ordem de um atraso de propagação de uma porta ou de algum outro flip-flop. Este seria o atraso encontrado se os dados fossem derivados da saída de outro flip-flop acionado pelo mesmo sinal de relógio. Existe também um atraso de propagação (não mostrado na figura) entre a saída Q e a borda de gatilho do sinal de relógio.

O flip-flop tipo D que desenvolvemos é eminentemente apropriado para uso em sistemas

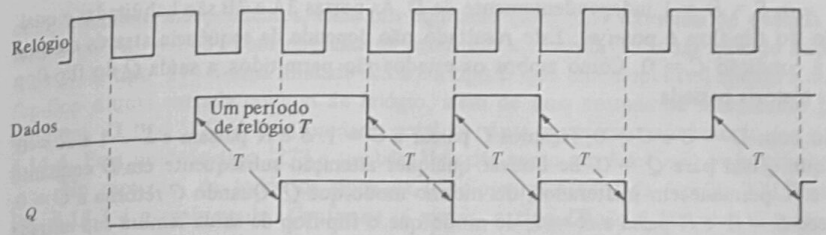


Fig. 4.14-2 Um flip-flop tipo D usado para atrasar uma entrada síncrona de dados durante um período de relógio.

síncronos. Ele resolve o problema da corrida, pois incorpora um arranjo mestre-escravo; resolve, também, o problema de detecção de 1s, pois não permite que $S = R = 0$. Não é afetado pelos tempos de subida ou descida do sinal de relógio, pois sua operação não depende de atrasos de propagação encontrados pelo sinal de relógio. Ao criar o flip-flop tipo D, perdemos, por outro lado, um pouco da versatilidade do flip-flop *SR* mestre-escravo com seus dois terminais de dados e com a possibilidade de ser facilmente transformado no flip-flop *JK*, ainda mais versátil. Se estivermos satisfeitos com o flip-flop tipo D, existe um circuito alternativo disponível mais econômico que o flip-flop mestre-escravo modificado.

Este tipo alternativo de flip-flop tipo D é mostrado na Fig. 4.14-3; consideremos sua operação. O flip-flop de saída que produz as saídas Q e $\overline{Q}$ consiste nas portas NAND 3A e 3B; as portas 1A, 1B, 2A e 2B constituem dois flip-flops direcionadores interligados. Pode ser verificado (Prob. 4.14-1) que as seguintes afirmações são válidas.

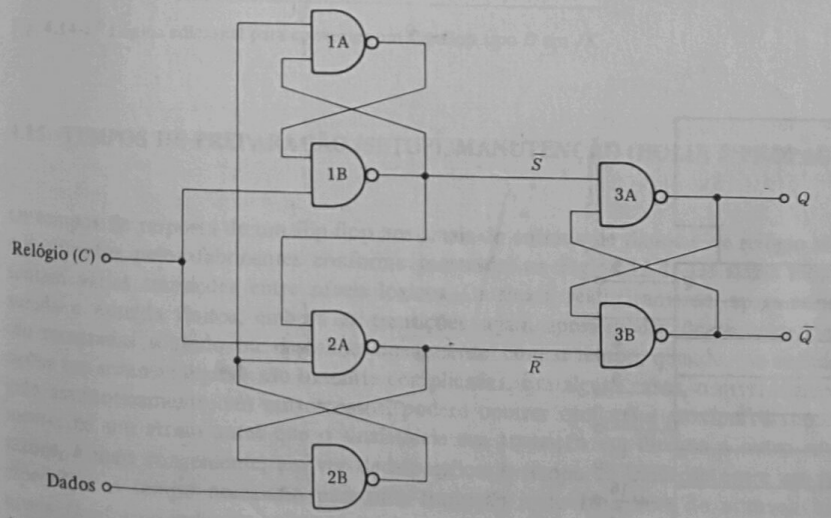


Fig. 4.14-3 O flip-flop tipo D.

1. Quando $C = 0$, $\overline{S} = \overline{R} = 1$ independentemente de D. As portas 3A e 3B são habilitadas e qualquer estado do flip-flop é possível. Este resultado não depende da seqüência através da qual chegamos à condição $C = 0$. Como ambos os estados são permitidos, a saída Q do flip-flop depende da história passada.
2. Começando com $D = 0$ e $C = 0$, façamos C passar a $C = 1$. $\overline{S}$ e $\overline{R}$ passam a $\overline{S} = 1$ e $\overline{R} = 0$, de modo que Q vai para $Q = 0$. Se houver qualquer alteração subseqüente em D enquanto $C = 1$, $\overline{S}$ e $\overline{R}$ permanecem inalterados, do mesmo modo que Q. Quando C retorna a $C = 0$, $\overline{S}$ permanece $S = 1$ e R passa a $\overline{R} = 1$, de modo que o flip-flop de saída lembra sua entrada de dados anterior e Q permanece em $Q = 0$.
3. Começando com $D = 1$ e $C = 0$, fazemos C passar a $C = 1$. $\overline{S}$ e $\overline{R}$ passam a $\overline{S} = 0$ e $\overline{R} = 1$, de modo que Q vai para $Q = 1$. Se houver qualquer alteração subseqüente em D enquanto $C = 1$, $\overline{S}$ e $\overline{R}$ permanecem inalterados, do mesmo modo que Q. Quando C retorna a $C = 0$, $\overline{R}$ permanece $\overline{R} = 1$ e S passa a $\overline{S} = 1$, de modo que o flip-flop de saída lembra sua entrada de dados anterior e Q permanece em $Q = 1$.

Resumindo, quando C executa uma transição positiva de $C = 0$ para $C = 1$, Q torna-se $Q = 0$ se D for $D = 0$ e Q torna-se $Q = 1$ se D for $D = 1$. Não existe outra maneira de alterar Q. Alternações de D quando $C = 1$ não afetam Q. Assim, exceto pelo fato de que no caso presente o gatilhamento tem lugar na borda positiva da transição do relógio, o comportamento é exatamente o mesmo do flip-flop tipo D anterior. O flip-flop da Fig. 4.14-3 é gatilhado pela borda, não faz demandas em relação à velocidade de transição do relógio e a saída responde a uma transição do relógio que também desabilita o flip-flop a responder a uma alteração dos dados de entrada.

Um flip-flop tipo D integrado típico que usa o circuito da Fig. 4.14-3 é o flip-flop duplo tipo D gatilhado pela borda tipo '74. Este dispositivo de quatorze pinos contém dois flip-flops

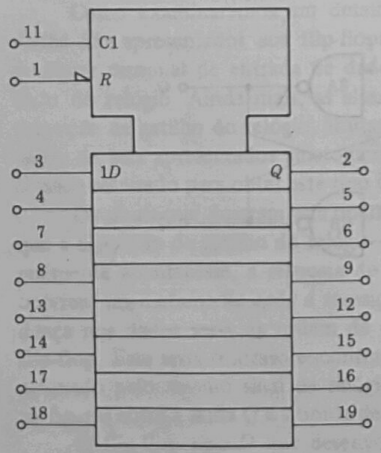


Fig. 4.14-4 Símbolo lógico do flip-flop octal tipo D tipo '273.

completamente independentes, cada um equipado (além dos terminais de entrada e saída mostrados na Fig. 4.14-3) com entradas de set-direto e reset-direto ativas quando baixas (ver Prob. 4.14-2). O tipo '174 é uma unidade sêxtupla tipo D que tem disponível apenas a saída Q de cada flip-flop e uma entrada comum de relógio, além de uma entrada de reset-direto para todos os flip-flops. O tipo '273 é uma unidade octal de vinte pinos cujo símbolo lógico é dado na Fig. 4.14-4. Esta unidade tipo '273 é também descrita como um *latch não-transparente* de oito bits.

No presente, de todos os tipos de flip-flops integrados da família TTL existentes, o da Fig. 4.14-3 é o mais usado. Conforme já vimos, o tipo D não é tão versátil quanto o tipo JK, pois não responde imediatamente às situações $J = K = 0$ e $J = K = 1$, ou seja, permanecer no mesmo estado ou chavear. Esta situação é facilmente remediável pelo uso de lógica adicional. Para fazer com que o flip-flop permaneça no estado atual, basta ligar a entrada D à saída Q, e para fazê-lo chavear basta ligar D à $\overline{Q}$. Para converter um flip-flop tipo D em JK, é necessário adicionar portas que gerem $D = J\overline{Q} + \overline{K}Q$ (Prob. 4.14-3), conforme mostrado na Fig. 4.14-5.

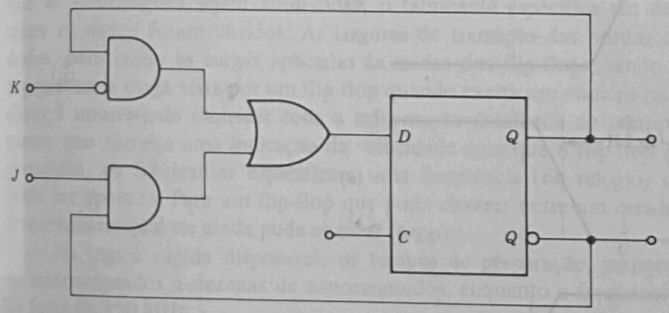


Fig. 4.14-5 Lógica adicional para converter um flip-flop tipo D em JK.

4.15 TEMPOS DE PREPARAÇÃO (SETUP), MANUTENÇÃO (HOLD) E PROPAGAÇÃO

Os tempos de resposta de um flip-flop aos sinais de entrada de dados e de relógio são geralmente especificados pelos fabricantes conforme mostrado na Fig. 4.15-1. Os sinais mostrados representam várias transições entre níveis lógicos. Os sinais, realisticamente, apresentam tempos de subida e descida finitos, embora as transições sejam apresentadas de maneira idealizada. Elas são mostradas subindo ou descendo linearmente com o tempo, quando, na realidade, as transições em sistemas digitais são bastante complicadas. Em alguns casos, o nível lógico final é atingido assintoticamente; em outros casos, podem ocorrer oscilações próximas à transição e, geralmente, há um atraso antes que o sinal inicie sua transição em direção a outro nível. Por estas razões, é mais conveniente, em vez de especificar o tempo de transição entre um nível e outro, especificar o tempo necessário para uma transição ir de 10 a 90% do intervalo total entre os níveis. Conforme indicado na transição do sinal de relógio da Fig. 4.15-1, este tempo é chamado largura de transição t_w.

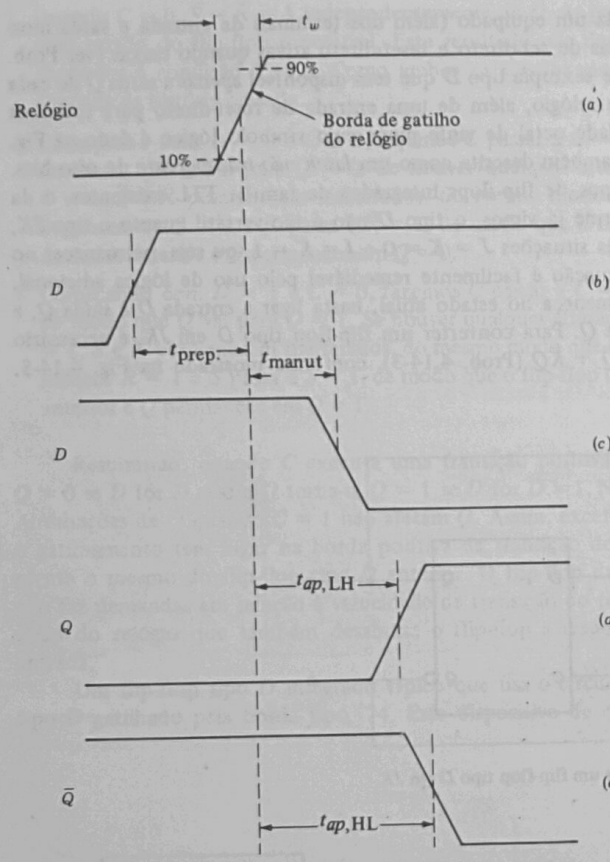


Fig. 4.15-1 Diagramas de tempo usados para definir os tempos de propagação, manutenção e o atraso de propagação de um flip-flop.

Os sinais mostrados aplicam-se a um flip-flop tipo D em que a transição de gatilho do relógio é suposta como sendo a transição positiva do nível lógico 0 para o nível 1. Esta transição de gatilho é mostrada na Fig. 4.15-1*a* e a transição da entrada de dados D do nível lógico 0 para o nível 1 é mostrada na Fig. 4.15-1*b*. A relação entre os sinais das Figs. 4.15-1*a* e *b* pretende mostrar a necessidade de a transição de D preceder a transição de gatilho do relógio por um tempo chamado *tempo de preparação* t_{prep}. Se o flip-flop deve reconhecer e responder apropriadamente à nova entrada de dados D, com certeza a mudança em D deve preceder a borda do relógio por um intervalo que não pode ser menos que t_{prep}. O sinal da Fig. 4.15-1*c* representa a transição de D de 1 para 0. A relação entre os sinais das Figs. 4.15-1*a* e *c* pretende mostrar que, se a alteração em D deve ser reconhecida sem falta, o nível $D = 1$ deve ser mantido pelo menos por um tempo t_{manut} após a borda do relógio. Ao todo, então, se o flip-flop deve responder a uma alteração em D, ela deve ocorrer pelo menos t_{prep} antes da borda do relógio e deve persistir após a borda do relógio pelo menos durante t_{manut}. Em alguns casos é permissível que D volte ao nível original *um pouco antes* da ocorrência da borda do relógio. O intervalo entre a liberação de D e a borda do relógio é, às vezes, especificado como tempo de *liberação* t_{lib} e outras vezes como tempo de manutenção *negativo*. Finalmente, notamos que os tempos de ocorrência das diversas transições são normalmente tomados como sendo o tempo em que as transições se encontram a meio caminho entre os dois níveis lógicos.

As relações entre a borda do relógio e o sinal das Figs. 4.15-1*d* e *e* indicam que, se os tempos de preparação e manutenção forem atendidos, haverá um *atraso de propagação* t_{ap} entre a borda do relógio e a resposta de Q e $\bar{Q}$ na saída do flip-flop. O atraso na saída que vai do nível *baixo* para o nível *alto* é $t_{ap,\,LH}$ e o outro atraso é $t_{ap,\,HL}$. Note-se que $t_{ap,\,LH}$ e $t_{ap,\,HL}$ podem ser diferentes. Na verdade, geralmente se constata que os tempos de preparação e manutenção são ligeiramente diferentes se a transição for positiva ou negativa.

Os fabricantes especificam informações de tempos não só em relação às entradas de dados mas também relativas às entradas de set e reset-direto, se existirem. De qualquer modo, para que as informações sejam entendidas, o fabricante especifica em detalhe as circunstâncias nas quais os dados foram obtidos. As larguras de transição das bordas do relógio e dos dados são dadas, bem como as cargas aplicadas às saídas dos flip-flops, sendo estas cargas ajustadas para representar a carga vista por um flip-flop quando excita um número razoável de portas. Em muitos casos é interessante esquecer toda a informação detalhada de tempos e ter disponível um parâmetro que forneça uma indicação da velocidade com que o flip-flop pode ser operado. Para este propósito, os fabricantes especificam uma freqüência (de relógio) máxima na qual o flip-flop pode ser operado. Para um flip-flop que pode chavear entre um estado e outro, $f_{máx}$ é a máxima freqüência na qual ele ainda pode chavear (*toggle*).

Na lógica rápida disponível, os tempos de preparação, manutenção e propagação variam de nanossegundos a dezenas de nanossegundos, enquanto a freqüência máxima de operação está na faixa de 150 MHz.

4.16 TRANSFERÊNCIAS DE REGISTRADOR PARA REGISTRADOR

Já observamos que um latch ou um flip-flop pode *armazenar* ou *lembrar*, isto é, *registrar* um bit. Um conjunto de tais latches ou flip-flops é, pois, um *registrador* no qual podemos *escrever* uma palavra e dos quais podemos *ler* a palavra armazenada quando desejarmos. Os registradores são, pois, o "papel de escrita" onde podemos escrever palavras para referência futura.

Uma grande parte das operações de um computador ou de outro processador digital consiste em nada mais que a transferência de palavras de um registrador para outro. Que isso aconteça não é surpreendente se notarmos as manipulações efetuadas quando efetuamos um cálculo ou uma operação de manejo de dados (escrituração) usando papel e lápis. Por exemplo, para efetuar um cálculo aritmético, é necessário, em primeiro lugar, *transferir* para o papel no qual vamos efetuar o cálculo os números que estão *escritos* em outro lugar, como, por exemplo, uma tabela. Pode, então, ser necessário *transferir* os números para uma coluna para que possamos somá-los e talvez seja necessário *transferir* a soma para outro lugar. Não é, pois, exagero dizer que as manipulações executadas em processos digitais consistem em um grande número de transferências de registrador para registrador misturadas com um pouco de operação lógica ou aritmética.

A Fig. 4.16-1 mostra dois registradores, o registrador A (RA) e o registrador B (RB) com a possibilidade de limpar RB e de transferir o conteúdo de RA para RB. Os registradores consistem em elementos armazenadores de quatro bits cada um, que podem ser latches controlados ou flip-flops, conforme for apropriado. Se desejarmos ler uma palavra de um registrador e transferir uma palavra para o mesmo registrador durante o mesmo ciclo de relógio, necessitaremos flip-flops. Caso contrário, batariam simples latches. Aqui usamos registradores de quatro bits, mas os registradores podem ser estendidos para qualquer número de bits apropriado. Quando a linha de *limpeza* (*clear*) for acionada, todos os latches ou flip-flops são zerados. Esta operação é descrita por

$$0 \rightarrow \text{RB}$$

Quando a linha de *transferência* estiver ativada, na primeira ocasião em que o relógio passar para o nível ativo a palavra que estiver em RA será transferida para RB. Esta operação é descrita por

$$\text{RA} \rightarrow \text{RB}$$

[Quando nos referimos a um registrador e há possibilidade de dúvida se estamos falando sobre o registrador ou sobre seu conteúdo, usamos parênteses, isto é, (RA), para indicar o conteúdo. Com esta notação escreveríamos (RA) → (RB) para indicar que o conteúdo de RA passou a ser, também, o conteúdo de RB.] A operação de transferência é manifestamente síncrona, pois depende do relógio. Geralmente, o sinal da linha de *limpeza* também é gerado por um circuito acionado pelo relógio, de modo que a operação de limpeza também é síncrona. No circuito da Fig. 4.16-1 não é necessário limpar RB antes de transferir a palavra vinda de RA. Quando uma palavra é transferida, o conteúdo de RA permanece inalterado.

A Fig. 4.16-2 mostra o uso de um *bus* (barramento) com tantas linhas quantos forem os bits dos registradores para permitir transferências de qualquer um de três registradores para qualquer

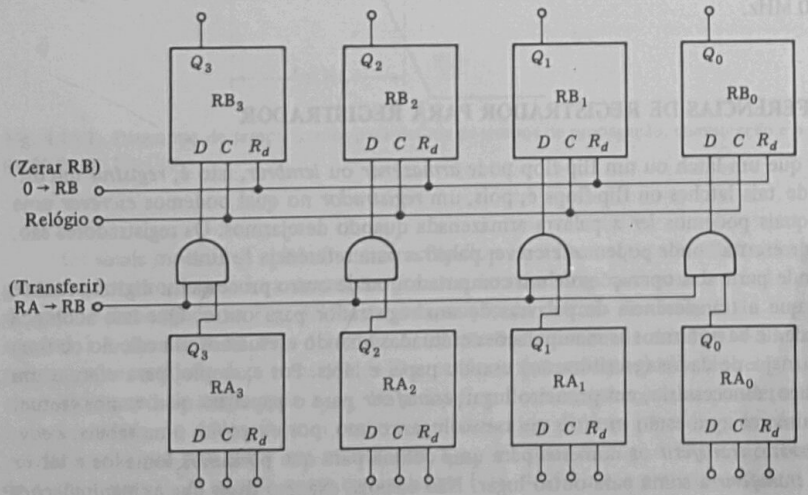


Fig. 4.16-1 Registradores ligados para permitir a transferência de conteúdos do registrador A para o registrador B.

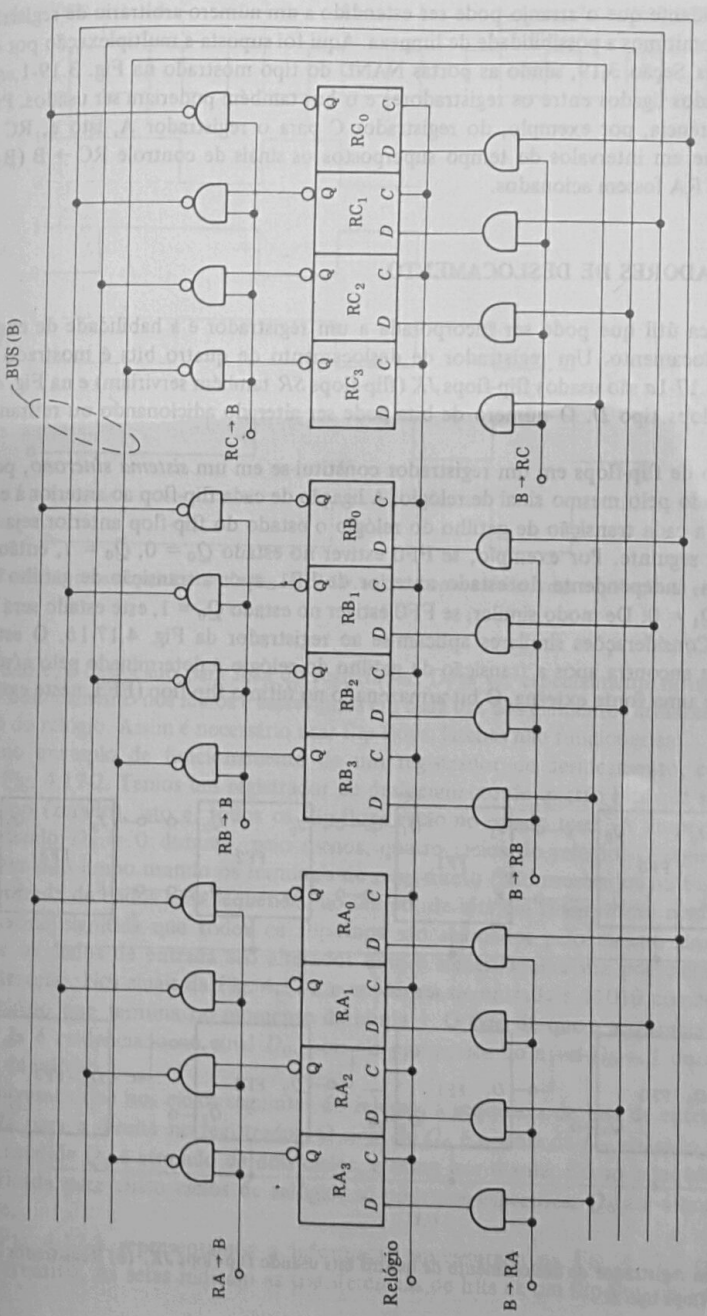


Fig. 4.16-2 Um bus comum usado para transferir o conteúdo de um para outro registrador entre muitos.

outro deles. É evidente que o arranjo pode ser estendido a um número arbitrário de registradores. Para simplificar, omitimos a possibilidade de limpeza. Aqui foi suposta a multiplexação por coletor aberto descrita na Seção 3.19, sendo as portas NAND do tipo mostrado na Fig. 3.19-1. Acionadores de três-estados ligados entre os registradores e o bus também poderiam ser usados. Para efetuar uma transferência, por exemplo, do registrador C para o registrador A, isto é, RC → RA, faríamos com que em intervalos de tempo superpostos os sinais de controle RC → B (B significando bus) e B → RA fossem acionados.

4.17 REGISTRADORES DE DESLOCAMENTO

Uma característica útil que pode ser incorporada a um registrador é a habilidade de executar a operação de deslocamento. Um registrador de deslocamento de quatro bits é mostrado na Fig. 4.17-1. Na Fig. 4.17-1*a* são usados flip-flops *JK* (flip-flops *SR* também serviriam) e na Fig. 4.17-1*b* são usados flip-flops tipo *D*. O número de bits pode ser alterado adicionando ou retirando flip-flops.

O conjunto de flip-flops em um registrador constitui-se em um *sistema síncrono*, pois cada flip-flop é acionado pelo mesmo sinal de relógio. A ligação de cada flip-flop ao anterior à esquerda é feita para que a cada transição de gatilho do relógio o estado do flip-flop anterior seja transferido ao flip-flop seguinte. Por exemplo, se FF0 estiver no estado $Q_0 = 0$, $\overline{Q}_0 = 1$, então $J_1 = 0$ e $K_1 = 1$. Assim, independente do estado anterior de FF1, após a transição de gatilho teremos FF1 no estado $Q_1 = 0$. De modo similar, se FF0 estiver no estado $Q_0 = 1$, este estado será transferido para FF1. Considerações similares aplicam-se ao registrador da Fig. 4.17-1*b*. O estado em que o flip-flop se encontra após a transição de gatilho do relógio é determinado pelo nível lógico aplicado a *D* por uma fonte externa. O bit armazenado no último flip-flop (FF3, neste exemplo) é

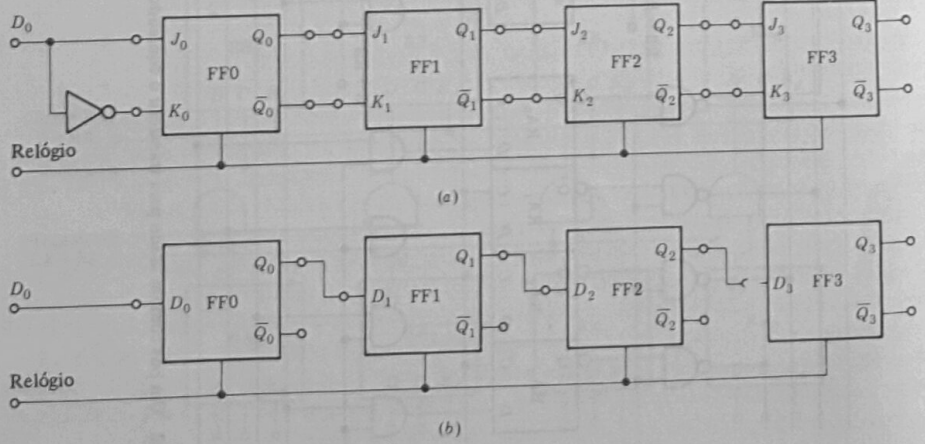


Fig. 4.17-1 (*a*) Um registrador de deslocamento de quatro bits usando flip-flops *JK*. (*b*) Registrador de deslocamento usando flip-flops tipo *D*.

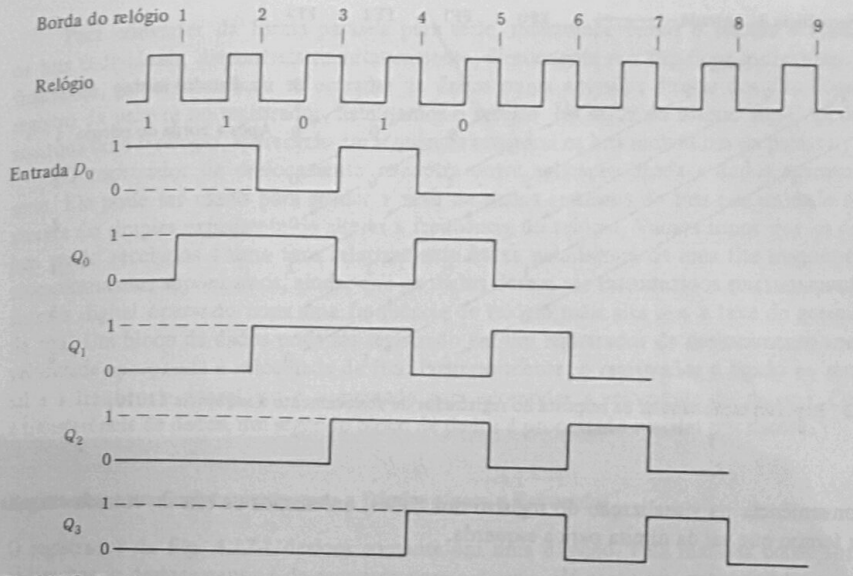


Fig. 4.17-2 Sinais para o registrador da Fig. 4.17-1*b*, inicialmente zerado, com seqüência de entrada 11010.

perdido, isto é, é deslocado para fora do registrador. Deve ser especialmente notado que no registrador de deslocamento nós lemos e escrevemos em cada um dos elementos armazenadores durante cada ciclo do relógio. Assim é necessário usar flip-flops; latches não funcionariam.

Como exemplo de funcionamento de um registrador de deslocamento, consideremos os sinais da Fig. 4.17-2. Temos um registrador de deslocamento de quatro bits que supomos inicialmente limpo (zerado), isto é, todos os flip-flops estão no estado reset. A limpeza pode ter sido feita mantendo $D_0 = 0$ durante, pelo menos, quatro ciclos do relógio. O registrador poderia, também, ter sido limpo usando os terminais de reset-direto (não mostrados na Fig. 4.17-1). Aplicamos à entrada de dados uma seqüência, no tempo, de bits em *sincronismo* com o relógio. Operação *síncrona* significa que todos os flip-flops são acionados pelo mesmo sinal de relógio e, ainda, que os dados de entrada são alterados uma e somente uma vez por ciclo de relógio, se houver alteração. Nos sinais da Fig. 4.17-2 a seqüência de entrada é 11010 começando no intervalo de relógio que termina no momento da borda 1. O fato de que a seqüência 11010 começa com dois 1s é evidenciado no sinal D_0, pois ele permanece no nível $D_0 = 1$ durante *dois* ciclos sucessivos de relógio.

Observamos que nos ciclos seguintes do relógio, a seqüência de bits de entrada é registrada e deslocada para a direita no registrador. O sinal de Q_0 é o sinal de D_0 atrasado de um ciclo de relógio, o sinal de Q_1 é atrasado de dois ciclos, e assim por diante. Como a seqüência de entrada foi especificada para cinco ciclos de relógio, só podemos especificar Q_0 até a borda 6, Q_1 até a borda 7 etc.

Na Fig. 4.17-3 representamos a informação apresentada na Fig. 4.17-2 em um formato tabular alternativo. As setas indicam as transferências de bits de um flip-flop para o seguinte. Para

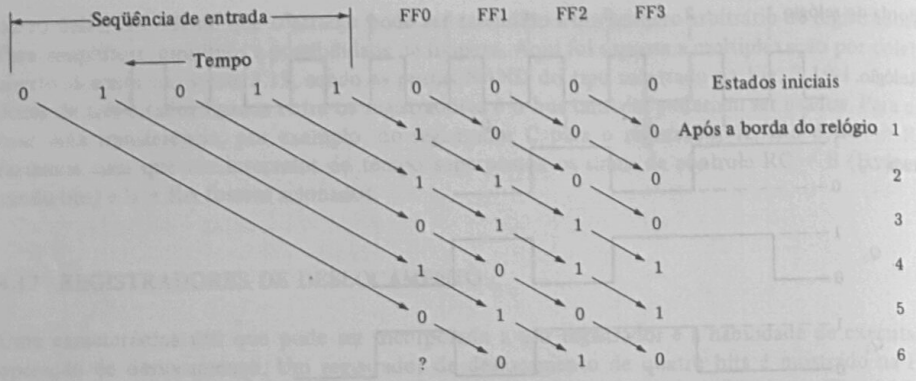


Fig. 4.17-3 Representação tabular da resposta do registrador de deslocamento à seqüência 11010.

maior conveniência na visualização do registro dos dados, ordenamos os bits de entrada em uma escala de tempo que vai da direita para a esquerda.

4.18 CARACTERÍSTICAS ADICIONAIS E USOS DOS REGISTRADORES DE DESLOCAMENTO

Nesta seção examinaremos alguns usos interessantes dos registradores de deslocamento geralmente envolvendo alguma modificação do circuito básico da Fig. 4.17-1.

Formatos de Dados Série e Paralelo

Dados digitais podem ser apresentados na forma série ou paralela. Na apresentação série, os bits individuais de uma palavra são conduzidos por um único fio (além do fio de retorno) e são apresentados em seqüência temporal, um bit após o outro. Na apresentação paralela, todos os bits da palavra são apresentados simultaneamente usando tantos fios quantos forem os bits (mais um fio de retorno). O formato série economiza circuitos, mas o formato paralelo economiza tempo. Consideremos, por exemplo, que determinado nível lógico leve 1 μs para ser reconhecido. O formato série usaria 8 μs para transmitir uma palavra de oito bits, ao passo que o formato paralelo levaria apenas 1 μs.

Seguidamente, é necessário passar de uma forma de apresentação para a outra, ou seja, de série para paralela e vice-versa. Um registrador de deslocamento pode ser usado para este propósito. Vamos supor que uma palavra de oito bits deva passar da forma série para a forma paralela. Podemos usar um registrador de deslocamento de oito bits e apresentar a palavra série de oito bits à entrada de dados do registrador com os bits individuais apresentados em sincronismo com o relógio. Após oito ciclos de relógio, desligando-se o relógio, a palavra estará registrada no registrador e os oito bits estarão simultaneamente disponíveis, isto é, em paralelo, nas oito saídas do flip-flop.

Para converter da forma paralela para série, interromperíamos o relógio e registraríamos os bits individuais, disponíveis simultaneamente, diretamente nos flip-flops individuais. Para esta finalidade, poderíamos usar as entradas de dados ou as entradas diretas dos flip-flops. Após o registro da palavra no registrador, habilitamos o relógio. Na saída do último flip-flop, de maneira síncrona com o relógio, aparecerão em seqüência temporal os bits individuais da palavra.

O registrador de deslocamento encontra outra aplicação ligada a dados apresentados em série. Ele pode ser usado para mudar a taxa de dados (número de bits por unidade de tempo) através do simples expediente de alterar a freqüência do relógio. Vamos supor que os dados estejam sendo recebidos a uma taxa relativamente baixa pela leitura de uma fita magnética movida mecanicamente; suponhamos, ainda, que os dados devem ser introduzidos sincronamente em um sistema digital operando com uma freqüência de relógio mais alta que a taxa de geração de bits da fita. Um bloco de dados pode ser registrado em um registrador de deslocamento usando uma velocidade apropriada à velocidade da fita. Posteriormente, o registrador é ligado ao sistema digital e a freqüência do relógio é aumentada para acomodar a velocidade do sistema. Completada a transferência de dados, um segundo bloco de dados é processado e assim por diante.

Registrador com Deslocamento para a Direita e para a Esquerda

O registrador da Fig. 4.17-1 desloca somente em uma direção. Pela maneira como foram feitas as ligações, o deslocamento é da esquerda para a direita. Há casos em que é útil termos um registrador que desloca em ambas as direções. Para inverter a direção de deslocamento, basta inverter a ordem das ligações entre os flip-flops. Para deslocar da direita para a esquerda, é necessário ligar a entrada do FF2 à saída do FF3 e assim por diante. Um registrador que incorpora portas que invertem as conexões em resposta a um controle de direção é chamado registrador de deslocamento para a direita e para a esquerda.

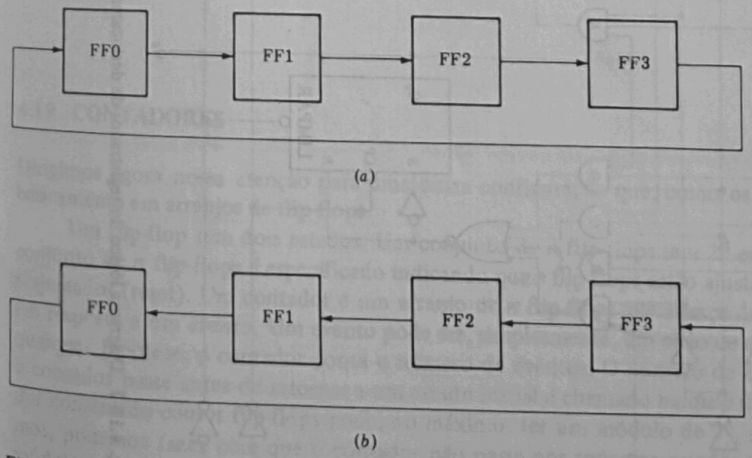


Fig. 4.18-1 Rotação em um registrador de deslocamento: (*a*) rotação para a direita e (*b*) rotação para a esquerda.

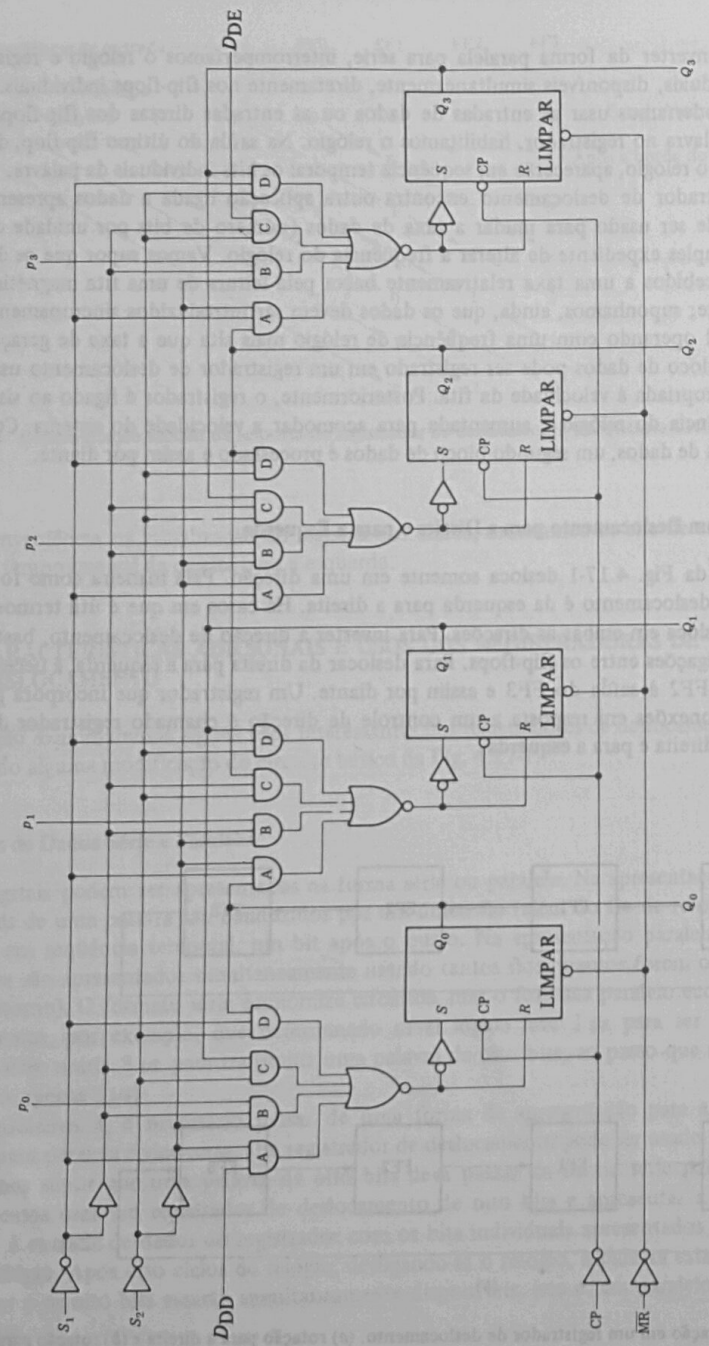


Fig. 4.18-2 Diagrama lógico do registrador de deslocamento universal bidirecional de quatro bits tipo '194.

Registrador com Rotação

Já observamos que o bit deslocado do último registrador é perdido e que é necessário suprir um bit de entrada ao primeiro flip-flop através de uma fonte externa. Podemos, todavia, ligar a saída do último flip-flop à entrada do primeiro. Neste caso não perdemos um bit e não necessitamos de uma entrada externa. Este modo de operação pode ser útil e a operação resultante, chamada *rotação*, é representada na Fig. 4.18-1. As setas indicam a direção do deslocamento dos dados. Na Fig. 4.18-1*a* temos uma rotação para a direita e na Fig. 4.18-1*b*, uma rotação para a esquerda.

Um exemplo de um registrador de deslocamento integrado é o CI tipo '194, cujo diagrama lógico é mostrado na Fig. 4.18-2. É descrito pelo fabricante como um registrador de deslocamento universal bidirecional de quatro bits. Possui uma entrada de zeramento mestre ($\overline{MR}$) ativa quando baixa que se sobrepõe a todas as outras entradas e limpa todos os flip-flops. Os flip-flops são do tipo *SR*, mas estão ligados de modo a operar como flip-flops tipo *D*. Duas entradas de seleção S_1 e S_0 determinam a operação do dispositivo. Os inversores na linha de seleção juntamente com cada grupo de quatro portas AND A, B, C e D constituem um multiplexador de quatro linhas para uma linha que determina a fonte das entradas de dados de cada flip-flop. Quando a porta A está habilitada, cada entrada de dados é a saída do flip-flop à esquerda ou a entrada D_{DD} (dados a serem deslocados para a direita) e o deslocamento é para a direita. Quando a porta B está habilitada, os dados de entrada são os níveis lógicos presentes nas linhas de *entrada paralelas* p_0, p_1, p_2 e p_3. Quando a porta C está habilitada, cada entrada de dados provém da direita ou da entrada D_{DE} (dados a serem deslocados para a esquerda). Finalmente, quando a porta *D* está habilitada, a entrada de dados de cada flip-flop é ligada à *sua própria* saída de dados, de modo que cada flip-flop registra seu próprio estado. Nestas circunstâncias, nada se altera e o registrador é descrito como estando na situação de *manutenção* (*hold*).

Existem outros tipos de registradores de deslocamento integrados, alguns de oito bits, outros de dezesseis bits. Na Fig. 4.18-3*a* mostramos o diagrama lógico do registrador de deslocamento de entrada série e saída paralela tipo '164. Seu símbolo lógico é mostrado na Fig. 4.18-3*b*. A notação SRG8 indica um registrador de deslocamento de oito bits. A seta apontando para a direita indica que a direção de deslocamento é para baixo na Fig. 4.18-3*b* (e para a direita na Fig. 4.18-3*a*).

4.19 CONTADORES

Dirigimos agora nossa atenção para uma outra configuração que, como os registradores, consiste basicamente em arranjos de flip-flops.

Um flip-flop tem dois estados. Um conjunto de n flip-flops tem 2^n estados. O estado de um conjunto de n flip-flops é especificado indicando quais flip-flops estão ajustados (set) e quais estão reajustados (reset). Um contador é um arranjo de n flip-flops que avança de um estado para outro em resposta a um *evento*. Um evento pode ser, simplesmente, um ciclo de um sinal de relógio. Em qualquer hipótese, o contador conta o número de eventos. O número de estados através do qual o contador passa antes de retornar a um estado inicial é chamado *módulo* do contador. Um contador construído com n flip-flops pode, no máximo, ter um módulo de 2^n, mas, se assim o desejarmos, podemos fazer com que o contador não passe por todos os estados possíveis. Neste caso o módulo seria menor que 2^n.

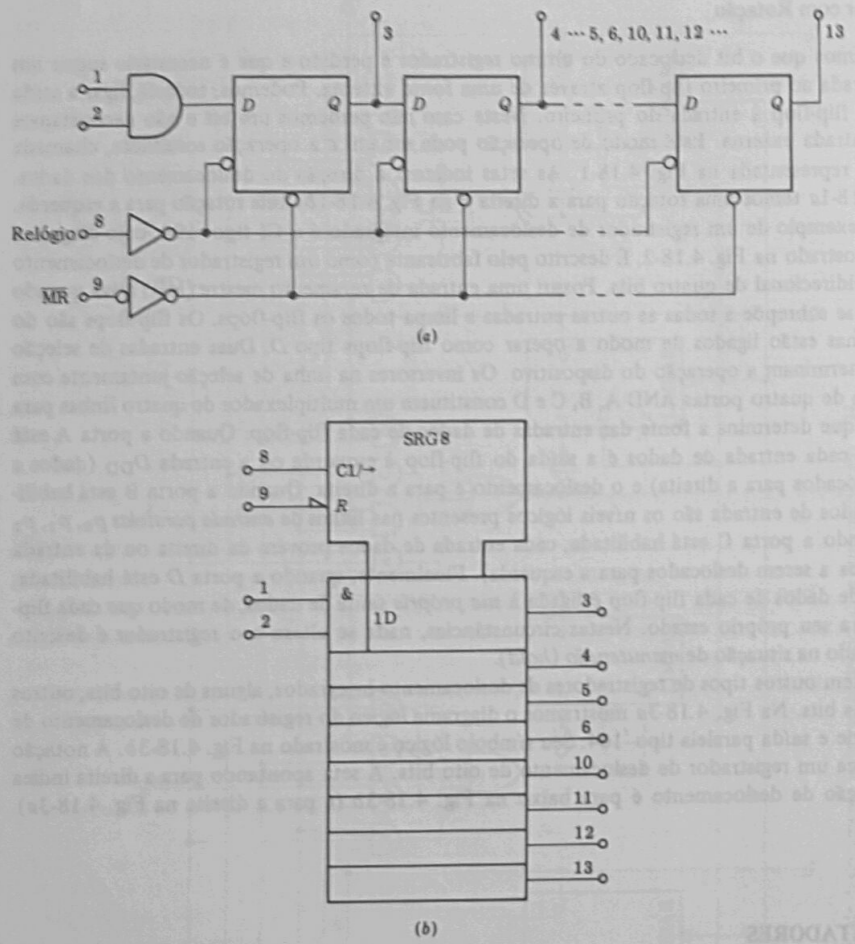


Fig. 4.18-3 (*a*) Diagrama lógico do registrador de deslocamento de entradas série e saída paralela tipo '164. (*b*) Símbolo lógico.

4.20 O CONTADOR EM ANEL

O circuito da Fig. 4.20-1 é um *contador em anel* com módulo 4. Nele reconhecemos um registrador de deslocamento (usando flip-flop tipo D, embora outros tipos também servissem) ligado de modo a permitir rotação para a direita. Primeiro, o terminal *inicia* é levado brevemente ao nível 1, fazendo $Q = 1$ no FF0 e zerando os demais flip-flops. A seguir, o sinal de relógio é aplicado e, como resultado, o contador passa a contar os ciclos do relógio com módulo 4. Cada borda de

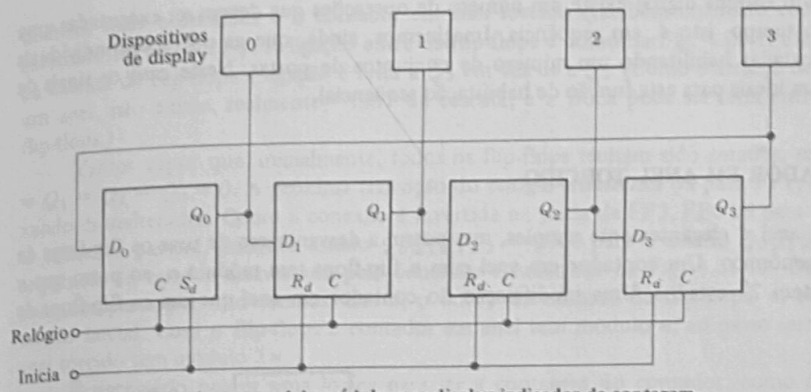


Fig. 4.20-1 Um contador em anel de módulo com display indicador de contagem.

gatilho do relógio avança o 1 para o flip-flop seguinte e, após quatro ciclos sucessivos do relógio, ele retorna ao estado inicial.

A leitura da contagem registrada no contador em anel é extremamente simples. Para determinar a contagem, basta saber qual o flip-flop que se encontra no estado $Q = 1$. Para isto foram incluídos quatro dispositivos de display. Cada um dos dispositivos é ligado à saída Q de um flip-flop. Imaginemos que os dispositivos sejam lâmpadas elétricas que se iluminam quando uma tensão correspondente ao nível lógico 1 for aplicada a elas. No nível lógico 0 elas estarão apagadas. Imaginemos, ainda, que em frente às lâmpadas coloquemos placas transparentes com os números 0, 1, 2 e 3, como indicado na figura. Durante a contagem veremos os números iluminados na ordem . . . 0, 1, 2, 3, 0, . . . e teremos uma indicação direta da contagem registrada.

Os sinais do contador em anel são mostrados na Fig. 4.20-2. Foram desenhados os sinais do relógio e das saídas Q dos quatro flip-flops. Começamos, arbitrariamente, no instante em que $Q_0 = 1$ e $Q_1 = Q_2 = Q_3 = 0$. Estes sinais são muito úteis para o propósito de *seqüenciar*. Imagi-

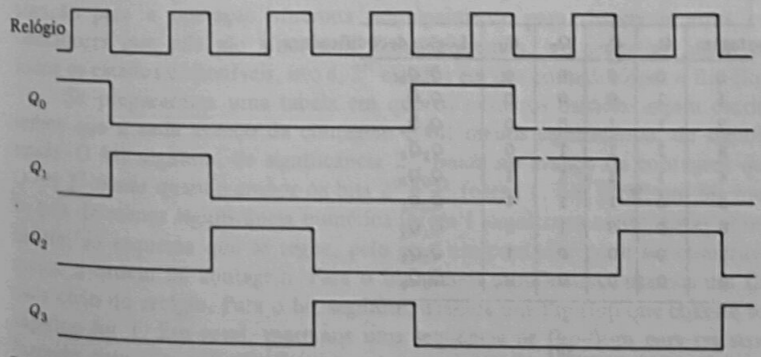
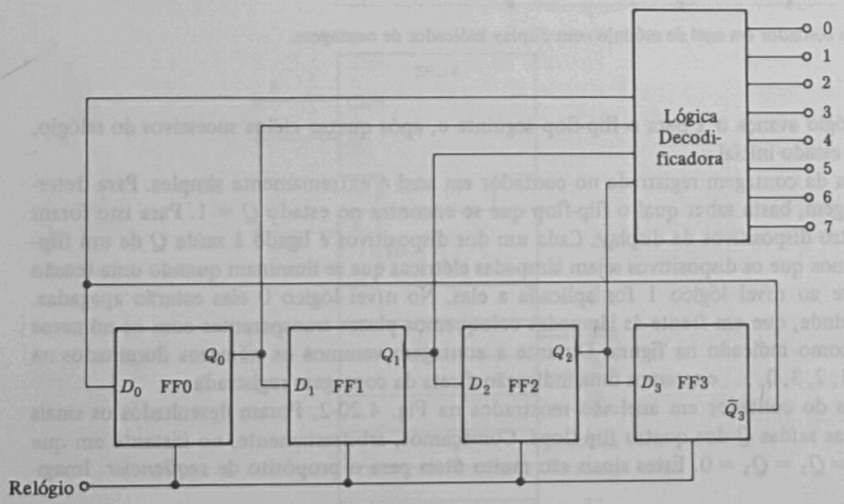


Fig. 4.20-2 Sinais do contador em anel de módulo 4 da Fig. 4.20-1.

nemos que em um sistema digital existe um número de operações que devem ser executadas uma após a outra no tempo, isto é, em seqüência. Imaginemos, ainda, que as operações individuais possam ser executadas habilitando um número de conjuntos de portas. Neste caso os sinais da Fig. 4.20-2 seriam ideais para esta função de habilitação seqüencial.

4.21 O CONTADOR EM ANEL TORCIDO

O contador em anel é elegantemente simples, mas possui a desvantagem de usar os flip-flops de modo pouco econômico. Um contador em anel com n flip-flops tem módulo n, ao passo que n flip-flops permitem 2^n estados. Uma modificação do contador em anel que usa os flip-flops de



(*a*)

Contagem	Q_0	Q_1	Q_2	Q_3	Lógica decodificadora
0	0	0	0	0	$\bar{Q}_0\bar{Q}_3$
1	1	0	0	0	$Q_0\bar{Q}_1$
2	1	1	0	0	$Q_1\bar{Q}_2$
3	1	1	1	0	$Q_2\bar{Q}_3$
4	1	1	1	1	Q_0Q_3
5	0	1	1	1	$\bar{Q}_0Q_1$
6	0	0	1	1	$\bar{Q}_1Q_2$
7	0	0	0	1	$\bar{Q}_2Q_3$
0	0	0	0	0	$\bar{Q}_0\bar{Q}_3$

(*b*)

Fig. 4.21-1 (*a*) Um contador em anel torcido. (*b*) Seqüência de contagem e lógica de decodificação.

maneira mais econômica é o contador em *anel torcido* (também chamado contador Johnson), mostrado na Fig. 4.21-1. A ligação entre os flip-flops é como na Fig. 4.20-1, exceto que no final da cascata de flip-flops a ligação é feita a $\bar{Q}_3$ em vez de a Q_3. (Como o arranjo de flip-flops forma um anel, não existe, realmente, "fim" da cascata, e a troca pode ser feita entre quaisquer dois flip-flops.)

Vamos supor que, inicialmente, todos os flip-flops tenham sido zerados, sendo, pois, $Q_0 = Q_1 = Q_2 = Q_3 = 0$. A próxima transição do relógio transferirá 0s para FF1, FF2 e FF3, deixando-os inalterados. Como a conexão é invertida na saída de FF3, FF0 irá para o estado $Q_0 = 1$. O contador passará, pois, do estado $Q_0Q_1Q_2Q_3 = 0000$ para o estado $Q_0Q_1Q_2Q_3 = 1000$. A seqüência de estados atravessada pelo contador é mostrada na Fig. 4.21-1*b*. Observamos que o contador passa por oito estados, isto é, o dobro do número de flip-flops e, então, retorna ao estado inicial. Com n flip-flops o contador em anel tem módulo n, ao passo que o contador em anel torcido tem módulo $2n$.

É necessário prover uma lógica para ler a contagem do contador; como no contador em anel, precisamos tantas saídas individuais quantos forem os estados do contador. Para cada estado, uma saída deve ser individualizada, assumindo o estado 1 enquanto todas as demais saídas permanecem em 0 (ou o contrário). Em outras palavras, é necessário construir um *decodificador* para o contador (ver Seção 3.15). Na Fig. 4.21-1*a* incluímos um bloco onde pode ser colocado o arranjo necessário de circuitos combinacionais (portas, mas não flip-flops) para a decodificação. As equações lógicas que relacionam as saídas com as entradas do bloco de lógica de decodificação são dadas na tabela da Fig. 4.21-1*b* (Prob. 4.21-1). Outras características dos contadores em anel e em anel torcido, como a autopartida, são apresentadas no Prob. 4.21-2.

4.22 OUTROS CONTADORES SÍNCRONOS

O contador em anel e o contador em anel torcido são contadores síncronos, pois o sinal de relógio (que também fornece os eventos a serem contados) é aplicado simultaneamente a todos os flip-flops. Estes contadores sofrem da limitação de não usarem os flip-flops economicamente. Neste aspecto o contador em anel torcido é melhor que o contador em anel, mas, em compensação, requer decodificação, enquanto o contador em anel não requer. Consideremos, agora, contadores síncronos que usam todos (se o desejarmos) ou quase todos os estados disponíveis. Chamamos atenção para a operação síncrona principalmente para distinguir estes contadores de outros contadores que não são síncronos. Consideraremos, em primeiro lugar, contadores que usam todos os estados disponíveis, isto é, 2^n estados em um contador com n flip-flops.

Se prepararmos uma tabela em que os números binários sejam escritos em ordem, notaremos que a cada avanço da contagem o bit menos significativo, de significância numérica 2^0, *muda*. O bit seguinte, de significância 2^1, *muda* no avanço da contagem quando o bit 2^0 for 1. O bit 2^2 *muda* quando *ambos* os bits 2^0 e 2^1 forem 1. Em geral, um bit muda sempre que todos os bits de menor significância numérica forem 1 simultaneamente. Estas observações levam, obviamente, ao esquema que se segue, pelo qual um contador pode ser construído para seguir exatamente a ordem de contagem. Para o bit menos significativo, usamos um flip-flop que *chaveie* a cada ciclo do relógio. Para o bit seguinte, usamos um flip-flop que chaveie sempre que o primeiro flip-flop for 1. Em geral, usaremos uma seqüência de flip-flops para registrar os bits individuais, fazendo com que cada flip-flop chaveie quando *todos* os flip-flops correspondentes a bits menos significativos tiverem, simultaneamente, $Q = 1$.

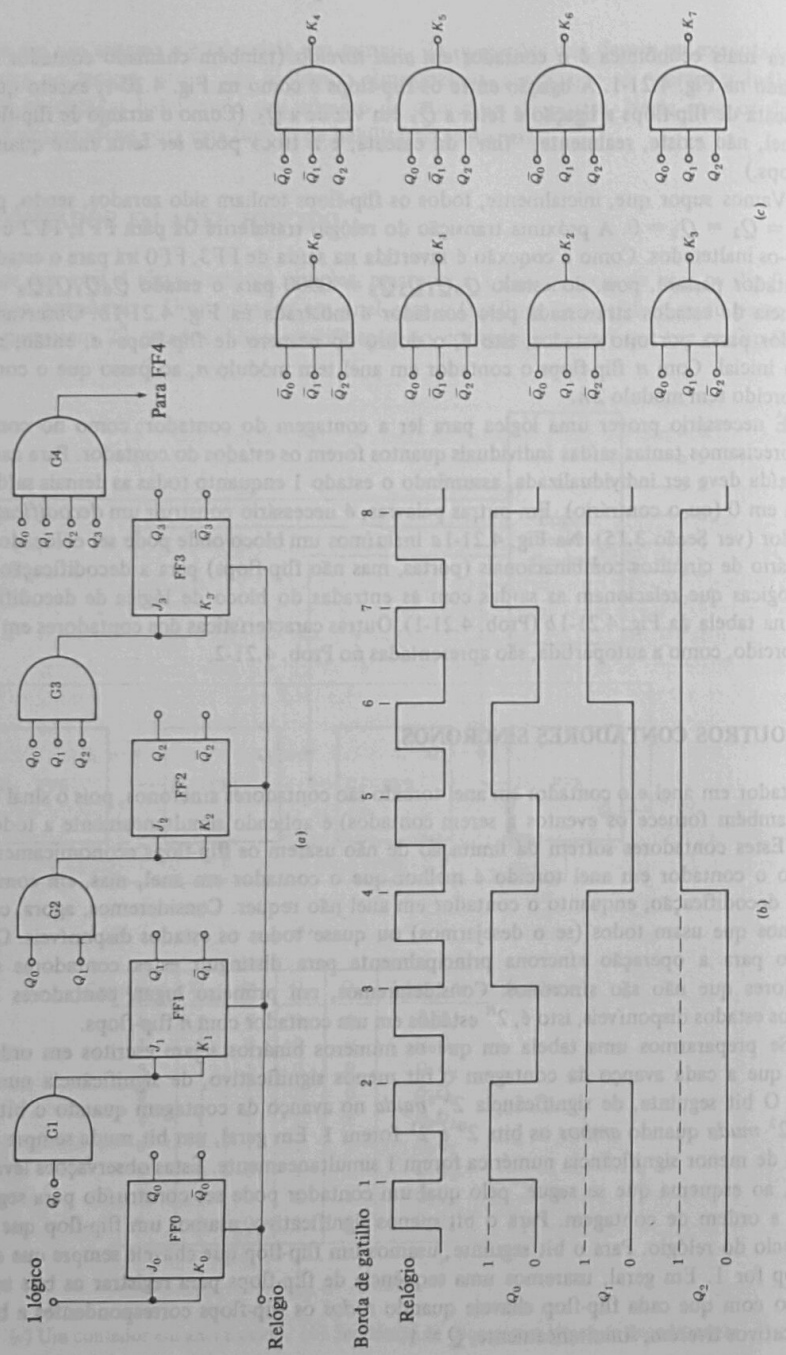


Fig. 4.22-1 Um contador síncrono com módulo 2^n que usa todos os estados disponíveis: *(a)* os diagramas lógicos, *(b)* os sinais e *(c)* a lógica de decodificação.

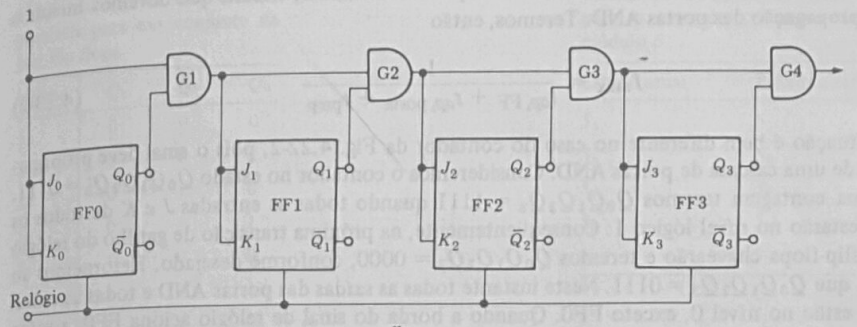


Fig. 4.22-2 Um contador síncrono módulo 2^n em que a capacidade de entrada das portas não aumenta com n (número de flip-flops).

O contador é mostrado na Fig. 4.22-1. Se n flip-flops forem usados no contador, ele terá módulo 2^n, ou seja, módulo igual ao número de estados. Os flip-flops usados são do tipo JK e chaveiam quando $J = K = 1$. Por exemplo, FF3 chaveia quando $J_3 = K_3 = 1$ e isto ocorre quando $Q_0 = Q_1 = Q_2 = 1$, conforme desejado. Obviamente, a porta G1 não serve a propósito algum e foi incluída somente para fazer a figura mais ordenada.

O contador da Fig. 4.22-1 apresenta a desvantagem da capacidade de entrada das portas AND aumentar com n e poder se tornar inconvenientemente grande. Uma possibilidade alternativa que elimina esta característica é mostrada na Fig. 4.22-2. (A porta 1 não é necessária.) Novamente observa-se que os terminais J e K de um flip-flop qualquer estão no nível 0, não permitindo o chaveamento do flip-flop, exceto quando *todos* os flip-flops precedentes estiverem na condição $Q_0 = Q_1 = Q_2 \ldots = 1$.

4.23 COMPARAÇÃO DE VELOCIDADE DE CONTADORES SÍNCRONOS

É de interesse compararmos as velocidades, isto é, as freqüências de relógio em que podem operar os contadores síncronos que discutimos até agora. Os contadores mais rápidos são os contadores em anel (simples ou torcido). Consideremos que a transição de gatilho do relógio para esse contador tenha ocorrido em $t = 0$, causando, como resposta, o avanço na contagem. Quanto tempo devemos esperar antes que a próxima transição possa ocorrer se queremos ter certeza de que a transição seguinte terá efeito sobre o contador? Nos contadores em anel somente um flip-flop, digamos o flip-flop FF_k, muda de estado a cada transição de gatilho, sendo esta mudança em resposta aos dados de entrada. Esta mudança deve ser transmitida ao próximo flip-flop, FF_{k+1}. Devemos, pois, esperar um tempo t_{ap} igual ao atraso de propagação do flip-flop FF_k. A seguir devemos permitir que o novo dado para o flip-flop FF_{k+1}, isto é, a saída de FF_k, permaneça constante durante um tempo igual ao tempo de preparação t_{prep}, conforme discutido na Seção 4.15. O tempo de manutenção do flip-flop não importa, pois o dado de entrada de um flip-flop só se altera a cada n ciclos do relógio. Ao todo, a freqüência máxima de relógio permitida é

$$f_{\text{máx}} = \frac{1}{t_{ap,\,\text{FF}} + t_{\text{prep}}} \qquad (4.23\text{-}1)$$

No contador módulo 2^n, da Fig. 4.22-1, a situação é semelhante, exceto que devemos incluir o atraso de propagação das portas AND. Teremos, então

$$f_{máx} = \frac{1}{t_{ap,\,FF} + t_{ap,\,porta} + t_{prep}} \qquad (4.23\text{-}2)$$

A situação é bem diferente no caso do contador da Fig. 4.22-2, pois o sinal deve propagar-se através de uma cascata de portas AND. Consideremos o contador no estado $Q_0Q_1Q_2Q_3 = 0111$. Na próxima contagem teremos $Q_0Q_1Q_2Q_3 = 1111$ quando todas as entradas J e K de todos os flip-flops estarão no nível lógico 1. Conseqüentemente, na próxima transição de gatilho do relógio todos os flip-flops chavearão e teremos $Q_0Q_1Q_2Q_3 = 0000$, conforme desejado. Retornemos ao ponto em que $Q_0Q_1Q_2Q_3 = 0111$. Neste instante todas as saídas das portas AND e todas as entradas J e K estão no nível 0, exceto FF0. Quando a borda do sinal de relógio aciona FF0, a saída de G1 só passa a 1 após um atraso de propagação. Haverá um segundo atraso de propagação antes que a saída de G2 passe a 1 e um terceiro antes que a saída de G3 passe a 1, e assim por diante. A troca de nível em Q_0 de $Q_0 = 0$ para $Q_0 = 1$ é propagada, como em dominós, através das portas AND. Esta transmissão seqüencial através de uma estrutura lógica após a outra, com cada transmissão acompanhada por um atraso, é chamada de pulsação (*ripple*), isto é, a alteração de Q_0 pulsa através das portas AND. Se a próxima borda do relógio ocorrer antes que a pulsação esteja completa, nem todas as entradas J e K estarão no nível 1. Alguns flip-flops não chavearão e a contagem estará errada. Um contador com n flip-flop necessita $n - 2$ portas AND; teríamos, pois, no presente caso:

$$f_{máx} = \frac{1}{t_{ap,\,FF} + (n-2)\,t_{ap,\,porta} + t_{prep}} \qquad (4.23\text{-}3)$$

O fator $(n - 2)$ aparece em vez de $(n - 1)$ pois, como já foi dito, a porta G1 na Fig. 4.22-2 não é realmente necessária.

4.24 CONTADORES SÍNCRONOS COM MÓDULO ARBITRÁRIO

Os contadores das Figs. 4.22-1 e 4.22-2 usam flip-flops economicamente, mas têm a desvantagem de que seu módulo é necessariamente 2^n, sendo n o número de flip-flops. Muitas vezes é necessário que um contador tenha módulo arbitrário. Para contar no sistema decimal, por exemplo, com o qual nos sentimos mais confortáveis, é necessário um contador com módulo 10.

Para construir um contador com módulo arbitrário, começamos com um número de flip-flops suficiente para prover um número de estados igual ou maior que o módulo. Se desejássemos contadores com módulo 5, 6 ou 7, dos flip-flops não bastariam, pois podem prover um máximo de quatro estados. Três flip-flops, todavia, seriam suficientes, pois três flip-flops provêem $2^3 = 8$ estados. Devemos, a seguir, decidir quais estados dentre os possíveis devem ser eliminados (esta decisão é, muitas vezes, totalmente arbitrária) e, finalmente, devemos decidir (talvez arbitrariamente) sobre a ordem em que o contador deve passar pelos estados não eliminados. Finalmente, usamos lógica combinacional para fazer com que o contador proceda de estado a estado conforme desejamos. Para ilustrar o processo, faremos o projeto de um contador com módulo 5.

O contador com módulo 5 requer três flip-flops. Os estados possíveis estão listados na Tab. 4.24-1. O estado S_0 é o estado em que os flip-flops estão zerados com $Q_2 = Q_1 = Q_0 = 0$ etc.

Tabela 4.24-1 Os oito estados possíveis para um conjunto de três flip-flops

Estado	Q_2	Q_1	Q_0
S_0	0	0	0
S_1	0	0	1
S_2	0	1	0
S_3	0	1	1
S_4	1	0	0
S_5	1	0	1
S_6	1	1	0
S_7	1	1	1

Tabela 4.24-2 Tabela de estados para o contador síncrono de módulo 5

Estado atual	Próximo estado
S_3	S_4
S_4	S_5
S_5	S_6
S_6	S_7
S_7	S_3

Arbitrariamente, decidimos usar os cinco últimos estados da Tab. 4.24-1 e que o contador deve passar pelos estados na ordem em que eles se encontram na tabela. A seqüência será, pois, $S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_3$ etc. Finalmente, decidimos usar flip-flops tipo JK (SR ou D também serviriam).

O que desejamos de nosso contador é mostrado pelas Tabs. 4.24-2 e 4.24-3. Na Tab. 4.24-2 indicamos que se o contador estiver no estado S_3, isto é, em um "estado atual" S_3, na próxima borda de gatilho do relógio o contador deve passar para um "próximo estado", que é S_4, e assim por diante, conforme a tabela. Uma tabela como a 4.24-2 é chamada *tabela de estados*. Na Tab. 4.24-3 repetimos a tabela de estados, mas também especificamos os estados individuais dos flip-flops que correspondem a cada estado do contador. Os símbolos Q_2^n etc. representam os estados dos flip-flops antes, e Q_2^{n+1} etc. representam os estados dos flip-flops depois da *enésima* borda do relógio. A Tab. 4.24-3 é chamada de *tabela de transição*.

Com o contador em qualquer dos estados usados é necessário dar a cada flip-flop instruções sobre que estado ele deve assumir quando da próxima transição de gatilho do relógio. Sabemos quais são estas instruções. As regras são dadas pela tabela verdade da Fig. 4.11-1*b*. Para nossos propósitos, é mais conveniente ter a informação da tabela verdade em uma forma alternativa.

Tabela 4.24-3 Tabela de transição para o contador síncrono com módulo 5

Estado atual			Próximo estado		
Q_2^n	Q_1^n	Q_0^n	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	1	1

Tabela 4.24-4 Uma forma alternativa para a tabela verdade de um flip-flop *JK*

$Q_n \rightarrow Q_{n+1}$		J	K
0	0	0	×
0	1	1	×
1	0	×	1
1	1	×	0

A tabela verdade da Fig. 4.11-1*b* nos diz diretamente o que cada flip-flop fará, dados os níveis lógicos de J e K. Seria preferível que a informação disponível na tabela estivesse sob uma forma que nos dissesse diretamente que níveis lógicos J e K deveriam assumir para que o flip-flop executasse determinada transição. Tal tabela verdade alternativa é mostrada na Tab. 4.24-4. Ela indica que, se um flip-flop se encontra inicialmente na condição $Q = 0$ e se desejarmos que ele se mantenha nesta condição após a transição de gatilho do relógio, então devemos ter $J = 0$ e que o nível lógico de K não interessa. Da tabela verdade original da Fig. 4.11-1*b* podemos verificar que a Tab. 4.24-4 está correta.

As Tabs. 4.24-2 e 4.24-3 mostram a seqüência de estados pelos quais o contador deve passar, e a Tab. 4.24-4 nos indica como instruir os flip-flops. Destas três tabelas podemos deduzir qual o nível lógico que devemos ter em cada entrada J e K a cada estado do contador. A forma mais conveniente de registrar os resultados deduzidos para cada entrada é incluí-los diretamente em mapas K. Há três flip-flops e dois terminais de entrada para cada flip-flop, J e K. Existem, pois, seis mapas K, conforme indicado na Fig. 4.24-1.

Os valores nos mapas K são determinados da seguinte maneira. Vamos supor que o contador esteja no estado S_3 com $Q_2Q_1Q_0 = 011$. Das Tabs. 4.24-2 e 4.24-3, o próximo estado deve ser S_4 com $Q_2Q_1Q_0 = 100$. Q_2 deve ir de $Q_2 = 0$ para $Q_2 = 1$. Da Tab. 4.24-4, temos que, para o flip-flop ir de $Q_2 = 0$ para $Q_2 = 1$, é necessário que $J_2 = 1$, enquanto K_2 pode ser $K_2 = \times$ (condição irrelevante). Nos mapas K de J_2 e K_2 e nos quadrículos correspondentes a $Q_2Q_1Q_0 = 011$,

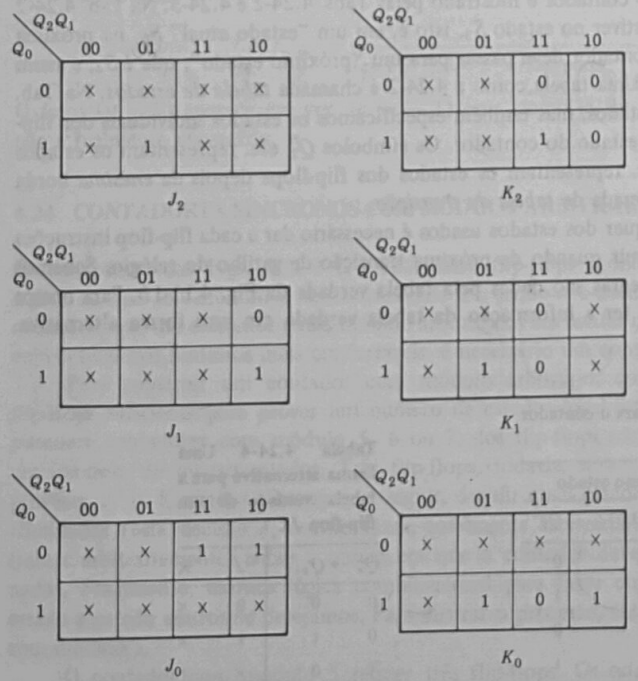


Fig. 4.24-1 Mapas de Karnaugh para o contador com módulo 5.

colocamos $J_2 = 1$ e $K_2 = \times$. Os valores dos mapas dos demais flip-flops são obtidos da mesma maneira.

Este processo detalhado assegura que os valores registrados nos mapas fazem o contador ir do estado S_3 para o estado S_4. A seguir, repetimos o processo para o estado S_4, e assim por diante, lembrando que após o estado S_7 o contador passa ao estado S_3. Como os estados S_0, S_1 e S_2 não ocorrem, colocamos condições irrelevantes em todos os mapas nos quadrículos correspondentes a estes estados.

Dos mapas K obtemos

$$\begin{aligned} J_2 &= 1 & K_2 &= Q_1Q_0 \\ J_1 &= Q_0 & K_1 &= \bar{Q}_2 \\ J_0 &= 1 & K_0 &= \bar{Q}_2 + \bar{Q}_1 \end{aligned} \qquad (4.24\text{-}1)$$

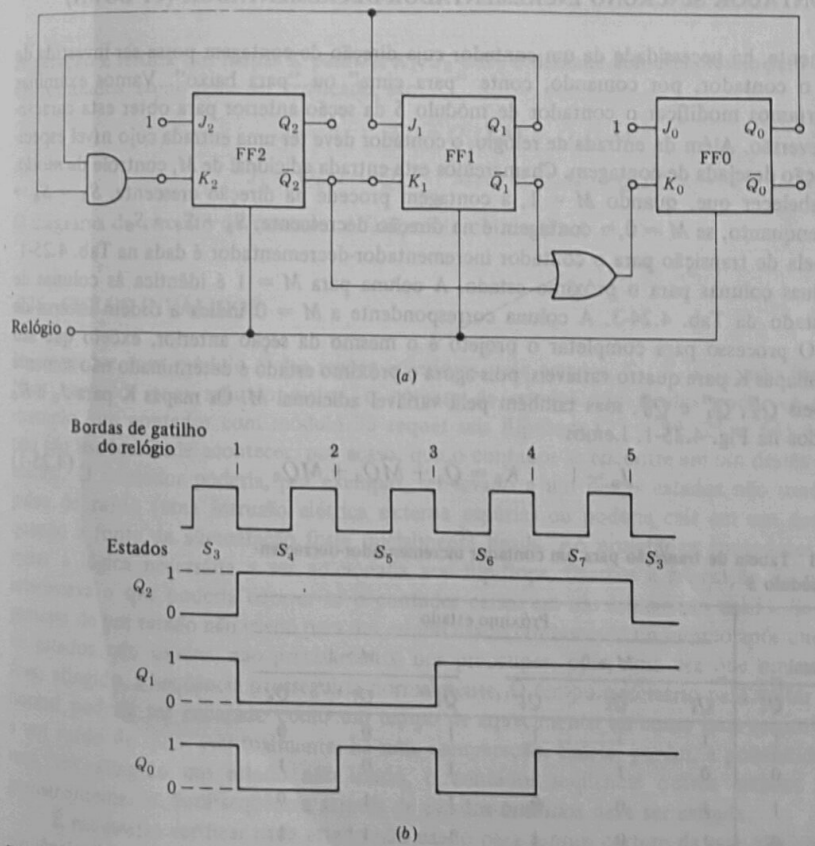


Fig. 4.24-2 (*a*) Diagrama lógico do contador com módulo 5. (*b*) Sinais.

Na Fig. 4.24-2*a* desenhamos o circuito do contador começando com os três flip-flops e fazendo as interligações conforme especificado pela lógica da Eq. (4.24-1). Adicionamos, ainda, um sinal de relógio comum a todos os flip-flops. Os sinais do contador são mostrados na Fig. 4.24-2*b*, começando, arbitrariamente, com o estado S_3. Note-se que o estado S_7 é seguido pelo estado S_3 e que os estados S_0, S_1 e S_2 nunca são usados.

Os estados do contador e a ordem em que o contador passa pelos estados selecionados foram escolhidos arbitrariamente. Naturalmente, nos perguntamos se o contador resultante não seria mais simples se houvéssemos escolhido outros estados ou usado uma seqüência diferente ou ambos. Enquanto o número de flip-flops for fixo, não seria possível, talvez, usar menos lógica, isto é, menos portas externas aos flip-flops? A resposta é sim. A seleção dos estados e seu ordenamento fazem diferença, mas o assunto não é de importância para nós no momento e será considerado brevemente nos problemas do Cap. 7.

4.25 O CONTADOR SÍNCRONO INCREMENTADOR-DECREMENTADOR (UP-DOWN)

Freqüentemente, há necessidade de um contador cuja direção de contagem possa ser invertida de modo que o contador, por comando, conte "para cima" ou "para baixo". Vamos examinar como poderíamos modificar o contador de módulo 5 da seção anterior para obter esta característica de reversão. Além da entrada de relógio, o contador deve ter uma entrada cujo nível especifique a direção desejada de contagem. Chamaremos esta entrada adicional de M, controle de *modo*. Vamos estabelecer que, quando $M = 1$, a contagem procede na direção crescente, $S_3 \rightarrow S_4 \rightarrow \rightarrow S_5 \ldots$, enquanto, se $M = 0$, a contagem é na direção decrescente, $S_3 \rightarrow S_7 \rightarrow S_6 \ldots$

A tabela de transição para o contador incrementador-decrementador é dada na Tab. 4.25-1. Agora há duas colunas para o próximo estado. A coluna para $M = 1$ é idêntica às colunas de próximo estado da Tab. 4.24-3. A coluna correspondente a $M = 0$ indica a ordem inversa de contagem. O processo para completar o projeto é o mesmo da seção anterior, exceto que são necessários mapas K para quatro variáveis, pois agora o próximo estado é determinado não somente pelas variáveis Q_2^n, Q_1^n e Q_0^n, mas também pela variável adicional M. Os mapas K para J_0 e K_0 são mostrados na Fig. 4.25-1. Lemos

$$J_0 = 1 \qquad K_0 = \bar{Q}_1 + \bar{M}Q_2 + M\bar{Q}_2 \tag{4.25-1}$$

Tabela 4.25-1 Tabela de transição para um contador incrementador-decrementador com módulo 5

Estado atual			Próximo estado					
			$M = 0$			$M = 1$		
Q_2^n	Q_1^n	Q_0^n	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}
0	1	1	1	1	1	1	0	0
1	0	0	0	1	1	1	0	1
1	0	1	1	0	0	1	1	0
1	1	0	1	0	1	1	1	1
1	1	1	1	1	0	0	1	1

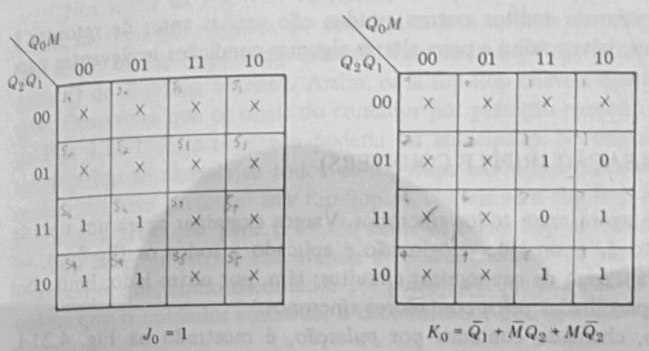


Fig. 4.25-1 Os mapas para J_0 e K_0 para o contador incrementador-decrementador especificado na tabela de transição da Tab. 4.25-1.

A geração e leitura dos mapas K para J_1, K_1, J_2 e K_2 é deixada como exercício para o estudante. Os resultados, como pode ser verificado, são

$$J_2 = 1 \qquad K_2 = Q_1Q_0M + \bar{Q}_1\bar{Q}_0\bar{M} \tag{4.25-2}$$

$$J_1 = \bar{Q}_0M + Q_0 \qquad MK_1 = \bar{Q}_2M + Q_0\bar{M} \tag{4.25-3}$$

O diagrama do circuito do contador é facilmente obtido (Prob. 4.25-1).

4.26 CICLOS INVÁLIDOS

No contador com módulo 5 das seções anteriores, três estados, S_0, S_1 e S_2, não foram usados. Num contador com módulo maior, o número de estados não usados pode ser grande. Por exemplo, um contador com módulo 33 requer seis flip-flops ($2^5 = 32$, $2^6 = 64$) e 31 estados não são usados. Pode acontecer, por acaso, que o contador se encontre em um destes estados não usados. O contador poderia, por exemplo, ser levado a um destes estados não usados por um pulso de ruído (uma intrusão elétrica externa espúria) ou poderia cair em um destes estados quando a fonte de alimentação fosse inicialmente ligada. Ao projetar os contadores, para minimizar a lógica necessária a ser adicionada aos flip-flops, fizemos a suposição de que não nos interessava o que poderia ocorrer se o contador caísse em um estado não usado. Se o contador passasse de um estado não usado para um estado usado diretamente ou mesmo após uma seqüência de estados não usados, não precisaríamos nos preocupar, pois, uma vez que um estado usado fosse atingido, a seqüência prosseguiria normalmente. O tempo necessário para voltar à seqüência normal poderia ser encarado como um tempo de aquecimento ou como uma resposta inevitável a um ruído do qual, eventualmente, há uma recuperação. Existe, porém, a possibilidade de que, uma vez atingido um estado não usado, o contador seqüencie outros estados não usados permanentemente. Esta seqüência através de estados *inválidos* deve ser evitada.

É necessário verificar cada estado não usado para termos certeza de que não conduz a uma seqüência de estados inválidos. Quando isto ocorre, a situação pode ser remediada alterando as condições irrelevantes para condições relevantes. Mesmo que não ocorra uma seqüência de estados

inválidos, um contador pode percorrer muitos outros estados não usados antes de retornar à seqüência normal. Aqui, também, talvez valha a pena alterar algumas condições irrelevantes para relevantes.

4.27 CONTADORES POR PULSAÇÃO (RIPPLE COUNTERS)

Os contadores considerados até agora eram todos síncronos. Vamos examinar agora um tipo de contador que é assíncrono, isto é, o sinal de relógio não é aplicado a todos os flip-flops. Os contadores assíncronos têm a vantagem de economizar circuitos; têm, por outro lado, limitações de velocidade que não são compartilhadas pelos contadores síncronos.

Um contador assíncrono, chamado contador por *pulsação*, é mostrado na Fig. 4.27-1. Os flip-flops devem ser operados no *modo de chaveamento*. Podem, pois, ser flip-flops tipo JK em que as entradas J e K são fixadas no nível lógico 1. Alternativamente, podem ser flip-flops tipo D com a saída $\overline{Q}$ ligada à entrada D.

Note-se, na Fig. 4.27-1, que o sinal externo de relógio, isto é, o sinal de relógio cujos ciclos serão contados, só é aplicado a C_0, isto é, à entrada de relógio do primeiro flip-flop. Todos os outros flip-flops derivam o sinal aplicado às entradas de relógio C_1, C_2, C_3 etc. dos terminais de saída Q_0, Q_1, Q_2 etc. do flip-flop anterior. Para que o efeito do relógio externo se faça sentir num flip-flop determinado, a influência do relógio deve propagar-se, isto é, pulsar, através de todos os flip-flops anteriores.

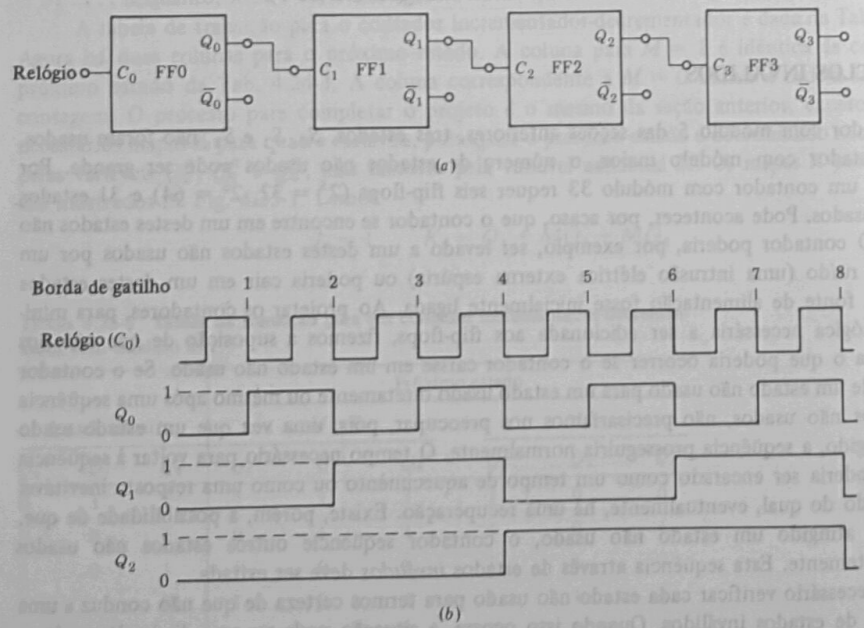


Fig. 4.27-1 Um contador por pulsação: (*a*) diagrama lógico e (*b*) sinais.

Nos sinais da Fig. 4.27-1*b* fizemos a suposição de que os flip-flops respondem, isto é, chaveiam, na transição negativa do sinal aplicado à entrada de relógio. Iniciamos num momento em que todos os flip-flops estão zerados. Para cada flip-flop, exceto FF0, a entrada de relógio é a saída Q do flip-flop anterior. Assim, cada flip-flop chaveia quando o flip-flop anterior retorna a zero. Observe-se que os sinais do contador por pulsação parecem iguais aos do contador síncrono da Fig. 4.22-1. Este resultado poderia ser antecipado. No contador síncrono estabelecemos que um flip-flop só chaveia se todos os flip-flops anteriores estão no estado $Q = 1$. A situação é a mesma no caso presente: um flip-flop só chaveia se o flip-flop anterior estiver no estado $Q = 1$ e faz uma transição para o estado zero. Este flip-flop anterior só faz sua transição quando o flip-flop anterior estiver executando a mesma transição (1 para 0) e assim por diante. Notamos, finalmente, que, como o contador por pulsação seqüencia através dos mesmos estados e na mesma ordem que o contador síncrono, o decodificador da Fig. 4.22-1*c* também serve para o contador assíncrono.

Na verdade, existe uma pequena diferença entre os sinais do contador síncrono e do assíncrono. Esta diferença resulta dos atrasos de propagação dos flip-flops, que, para simplificar, não levamos em conta nas Figs. 4.22-1 e 4.27-1. A situação é aclarada na Fig. 4.27-2, onde redesenhamos, na Fig. 4.27-2*a*, o sinal do contador síncrono da Fig. 4.22-1 na vizinhança da borda de gatilho do relógio 4 e, na Fig. 4.27-2*b*, a porção correspondente dos sinais do contador por pulsação da Fig. 4.27-1. Observe-se que na Fig. 4.27-2*a* existe um atraso fixo, o atraso de propagação de um flip-flop t_{ap}, entre a borda do relógio e a transição de nível de qualquer dos sinais. Na Fig. 4.27-2*b*, todavia, o atraso é progressivamente maior, aumentando de t_{ap} cada vez que passamos de um flip-flop para o seguinte.

Esta acumulação de atrasos de propagação causa dificuldades na leitura da contagem, como veremos a seguir. Consideremos as indicações de saída K_0 etc. do decodificador da Fig.

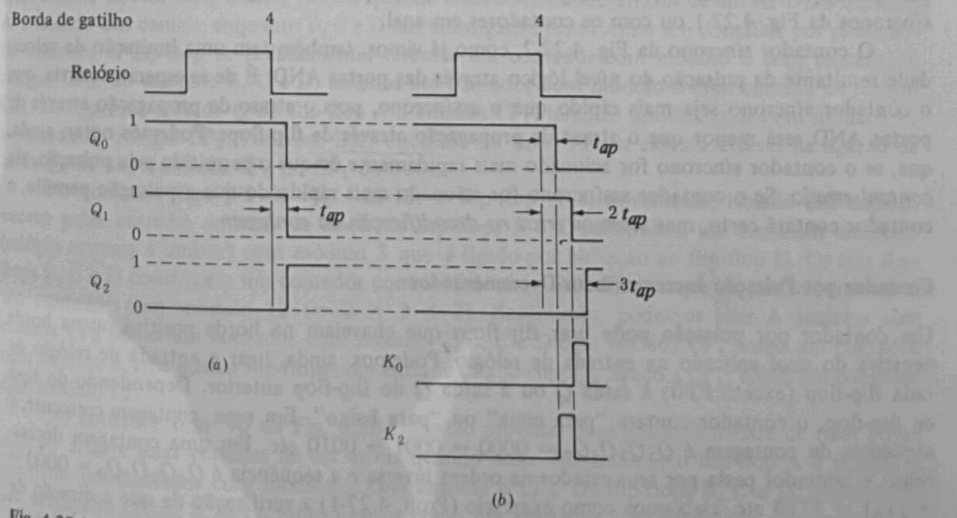


Fig. 4.27-2 Sinais na vizinhança da borda de gatilho do relógio 4: (*a*) o contador síncrono da Fig. 4.22-1 e (*b*) o contador por pulsação da Fig. 4.27-1.

4.22-1*c*. Antes da borda 4, temos $Q_2Q_1Q_0 = 011$ e $K_3 = 1$ enquanto todos os demais K são 0, conforme desejado. Em um tempo t_{ap} após a borda 4, teremos $Q_2Q_1Q_0 = 100$ e K_4 passará a $K_4 = 1$ enquanto os demais K serão zero. Existe, obviamente, um atraso t_{ap} antes que o decodificador registre a nova contagem corretamente, mas este atraso, em geral, não causa preocupações. Por outro lado, consideremos a situação da Fig. 4.27-2*b*. Novamente, antes da borda 4, somente $K_3 = 1$. Note-se que, agora, há intervalos em que $K_0 = 1$ e $K_2 = 1$. O decodificador fornece informação errada, pois, no intervalo descrito: todos os K, exceto K_3 e K_4, deveriam permanecer em 0.

Como vimos, os contadores por pulsação geram *erros de decodificação* que persistem durante parte do intervalo de um ciclo de relógio. Para mascarar estes erros comumente se adiciona uma entrada extra a cada porta do decodificador da Fig. 4.22-1. Estas entradas adicionais são ligadas a um terminal comum de entrada e um *pulso de controle* é aplicado a este terminal. Este pulso só permanece no nível 1 durante aquela parte do ciclo de relógio em que não ocorrem erros de decodificação. Enquanto os erros ocorrem, o controle permanece em 0, todas as portas de decodificação ficam desabilitadas e todos os K permanecem no nível 0.

O intervalo durante o qual ocorrem erros de decodificação da contagem aumenta com o número de flip-flops na cascata. O pior caso ocorre quando todos os flip-flops estão no estado $Q = 1$ e o contador mostra 11 . . . 11. Na próxima borda do relógio, a contagem deve passar para 00 . . . 00. Somente o primeiro flip-flop, todavia, responde diretamente ao relógio e a transição de 1 para 0 pulsa através de toda a cascata de flip-flops. Se existirem n flip-flops, a pulsação não se completa antes de um tempo nt_{ap}. Suponhamos, agora, que o período de relógio seja menos que nt_{ap}. Então, antes que a contagem 00 . . . 00 tenha sido registrada no contador, o primeiro flip-flop, no mínimo, já terá sido chaveado para o estado 1. Nunca ocorrerá, portanto, um momento em que o contador leia 00 . . . 00, e será impossível decodificar o contador corretamente. Resumindo, quando for necessário decodificar a contagem corretamente, o contador por pulsação tem uma séria limitação de velocidade quando comparado com os contadores síncronos da Fig. 4.22-1 ou com os contadores em anel.

O contador síncrono da Fig. 4.22-2, como já vimos, também tem uma limitação de velocidade resultante da pulsação do nível lógico através das portas AND. É de se esperar, todavia, que o contador síncrono seja mais rápido que o assíncrono, pois o atraso de propagação através de portas AND será menor que o atraso de propagação através de flip-flops. Podemos notar, ainda, que, se o contador síncrono for acionado mais rapidamente do que o permitido pela pulsação, ele *contará errado*. Se o contador assíncrono for acionado mais rápido do que a pulsação permite, o contador contará certo, mas teremos *erros na decodificação da contagem*.

Contador por Pulsação Incrementador-Decrementador

Um contador por pulsação pode usar flip-flops que chaveiam na borda positiva ou na borda negativa do sinal aplicado na entrada de relógio. Podemos, ainda, ligar a entrada de relógio de cada flip-flop (exceto FF0) à saída Q ou à saída $\overline{Q}$ do flip-flop anterior. Dependendo do tipo de flip-flop, o contador contará "para cima" ou "para baixo". Em uma contagem crescente, a seqüência de contagem é $Q_3Q_2Q_1Q_0 = 0000 \rightarrow 0001 \rightarrow 0010$ etc. Em uma contagem decrescente, o contador passa por seus estados na ordem inversa e a seqüência é $Q_3Q_2Q_1Q_0 = 0000 \rightarrow$ $\rightarrow 1111 \rightarrow 1110$ etc. Deixamos como exercício (Prob. 4.27-1) a verificação de que a direção de contagem é a da Tab. 4.27-1. Também deixamos como exercício (Prob. 4.27-2) a modificação do circuito da Fig. 4.27-1 para transformá-lo em um contador incrementador-decrementador.

Tabela 4.27-1 Direção de contagem em um contador por pulsação A entrada de relógio de cada flip-flop é a mostrada, exceto para o primeiro flip-flop, que é acionado por um sinal externo

Ligação da entrada de relógio	Direção da borda do relógio para chaveamento	Direção de contagem
Q	Negativa	Crescente
Q	Positiva	Decrescente
$\overline{Q}$	Negativa	Crescente
$\overline{Q}$	Positiva	Decrescente

Do mesmo modo que os contadores síncronos, os contadores por pulsação podem ser construídos com módulo arbitrário. Alguns dos métodos usados são apresentados nos Probs. 4.27-3 e 4.27-4.

4.28 CIRCUITOS INTEGRADOS CONTADORES

Existe um grande número de circuitos integrados contadores (em torno de vinte) que podem ser encontrados em um catálogo de TTL. Todos envolvem quatro flip-flops e alguns são bastante elaborados. Os diagramas lógicos de três dos mais simples são mostrados na Fig. 4.28-1. O tipo '93, mostrado na Fig. 4.28-1*a*, é um contador por pulsação. Os terminais J e K não indicados são, na realidade, ligados ao nível lógico 1 através de conexões internas. Todos os flip-flops têm $J = K = 1$ e estão no modo de chaveamento. Observamos que, exceto através dos controles de reset-mestre diretos MR_1 e MR_2 (ativos quando altos e acionando através de um AND), o flip-flop A é isolado dos demais, enquanto B, C e D são interligados para formar um contador por pulsação. Se usarmos o flip-flop A isoladamente, teremos um contador com módulo 2 com entrada de relógio $\overline{CP}_0$. Se usarmos B, C e D, teremos um contador com módulo 8 com entrada de relógio $\overline{CP}_1$. Se ligarmos a saída Q do flip-flop A à entrada $\overline{CP}_1$, teremos um contador com módulo 16 e entrada de relógio $\overline{CP}_0$. Podemos ligar circuitos do tipo '93 em cascata através da ligação da saída Q_3 de um à entrada $\overline{CP}_0$ ou $\overline{CP}_1$ do outro.

No contador tipo '92 da Fig. 4.28-1*b*, o flip-flop A é, novamente, isolado dos demais, exceto pelas entradas comuns de reset. Os flip-flops B e C constituem um contador síncrono (relógio comum a ambos) com módulo 3 que é ligado por pulsação ao flip-flop D. Os três flip-flops B, C e D constituem um contador com módulo 6 (= 3 × 2) e os quatro juntos constituem um contador com módulo 12 (= 2 × 3 × 2). Resumindo, podemos usar A sozinho com entrada $\overline{CP}_0$ como um contador com módulo 2; B, C e D com entrada $\overline{CP}_1$ como um contador com módulo 6, e os quatro flip-flops com entrada $\overline{CP}_0$, e com Q_0 ligado a $\overline{CP}_1$, como um contador com módulo 12.

No contador tipo '90 da Fig. 4.28-1*c*, estão previstas duas entradas mestre de reset ativas quando altas e duas entradas de set ativas quando altas. Novamente, o flip-flop A é isolado e constitui um contador com módulo 2 cuja entrada é $\overline{CP}_0$. Os flip-flops B, C e D, como pode ser verificado (Prob. 4.28-1), estão ligados como um contador com módulo 5 cuja entrada de relógio é $\overline{CP}_1$. Se ligarmos Q_0 a $\overline{CP}_1$, teremos um contador com módulo 10 (= 2 × 5), isto é, um contador de década. Com esta ligação, o sinal de relógio seria aplicado a $\overline{CP}_0$. Se desejarmos,

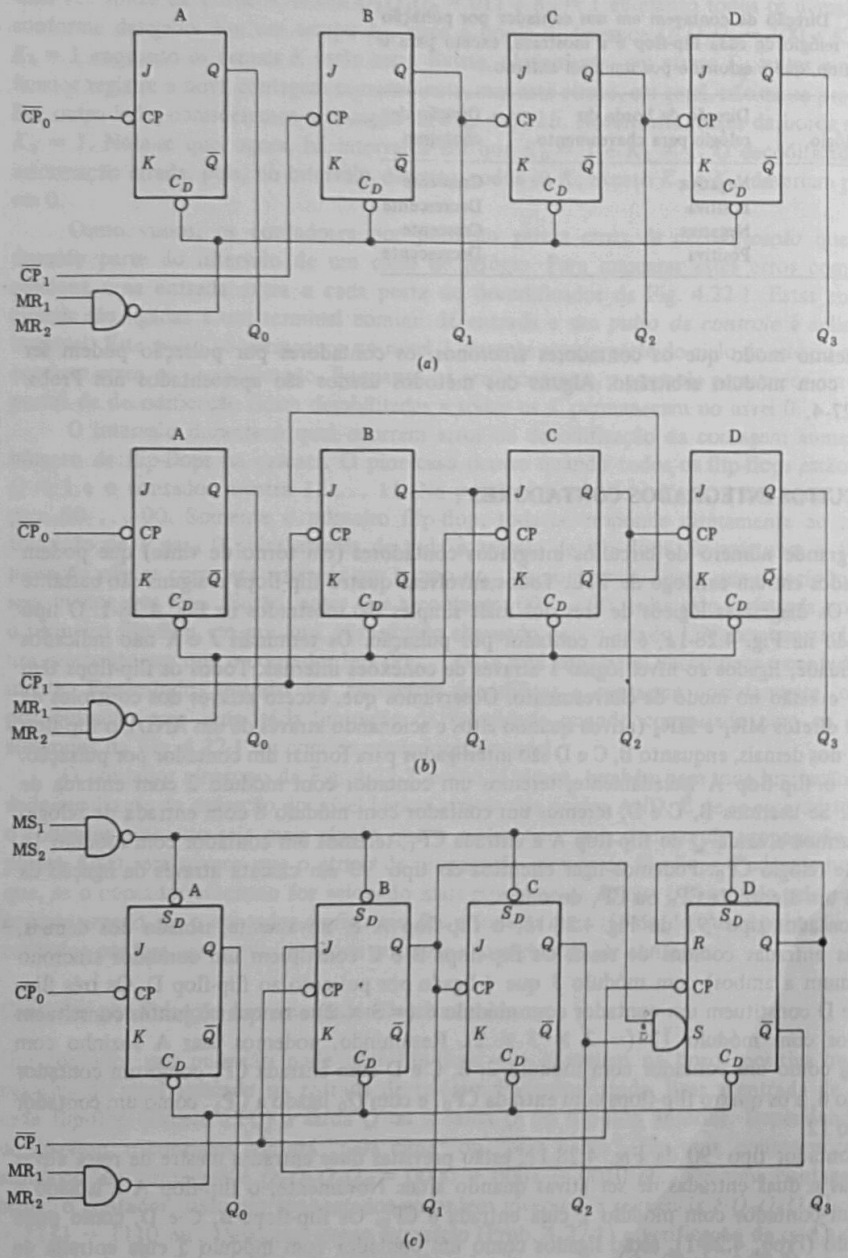


Fig. 4.28-1 Contadores integrados: (*a*) contador por pulsação tipo '93; (*b*) contador com módulo 12 tipo '92 e (*c*) contador de década tipo '90.

podemos ligar Q_3 a $\overline{CP}_0$ e usar $\overline{CP}_1$ como entrada de relógio. Em qualquer dos casos, o módulo do contador seria o mesmo, mas os sinais gerados seriam diferentes (Prob. 4.28-2).

Típico dos contadores mais elaborados disponíveis é o tipo '190. Este dispositivo é um contador incrementador-decrementador ("sobe-desce") com previsão para carga assíncrona. Tem disponíveis sinais de saída que facilitam a ligação em cascata com outros contadores iguais de modo síncrono ou por pulsação.

CAPÍTULO 5

ARITMÉTICA

Neste capítulo descreveremos como se representam números, tanto positivos como negativos, e como eles são manipulados nos sistemas digitais. Consideraremos, a seguir, a execução de operações aritméticas. Enfatizaremos a operação de soma, uma vez que todas as demais operações aritméticas podem ser executadas através da soma.

5.1 REPRESENTAÇÃO DE NÚMEROS COM SINAL

Números, como palavras digitais, são armazenados, isto é, escritos em registradores. Quando os estados set e reset dos flip-flops de um registrador são usados para representar os *numerais* 0 e 1 é conveniente e razoável (embora não seja absolutamente necessário) que o estado set (saída Q do flip-flop com nível 1) represente o numeral 1 e que o estado reset represente o numeral 0. Assim, se o estado de um registrador de quatro bits for $A_3A_2A_1A_0 = 1101$ e o registrador estiver armazenando um número, o número será 1101 (base 2) = 13 (base 10). A representação de um registrador de quatro bits é mostrada na Fig. 5.1-1*a*. O registrador tem dezesseis posições e pode ser usado para registrar um número de 0 a 15.

O registrador da Fig. 5.1-1*a* é desenhado em forma circular para lembrar que um registrador tem um módulo. Isto significa que possui um número finito de estados e que, se percorresse todos os seus estados, um a um, eventualmente retornaria ao estado inicial e recomeçaria o ciclo. O registrador da Fig. 5.1-1 é descrito como um registrador com módulo 16 ou um registrador com um módulo de quatro casas. O ponteiro indica que o registrador pode ser usado para guardar uma contagem ou mesmo para somar e subtrair. Podemos contar avançando o ponteiro uma posição para indicar uma contagem adicional. Para somar 2 e 3, primeiro limparíamos o registrador fazendo-o igual a 0000. A seguir, avançaríamos o ponteiro duas posições e depois três

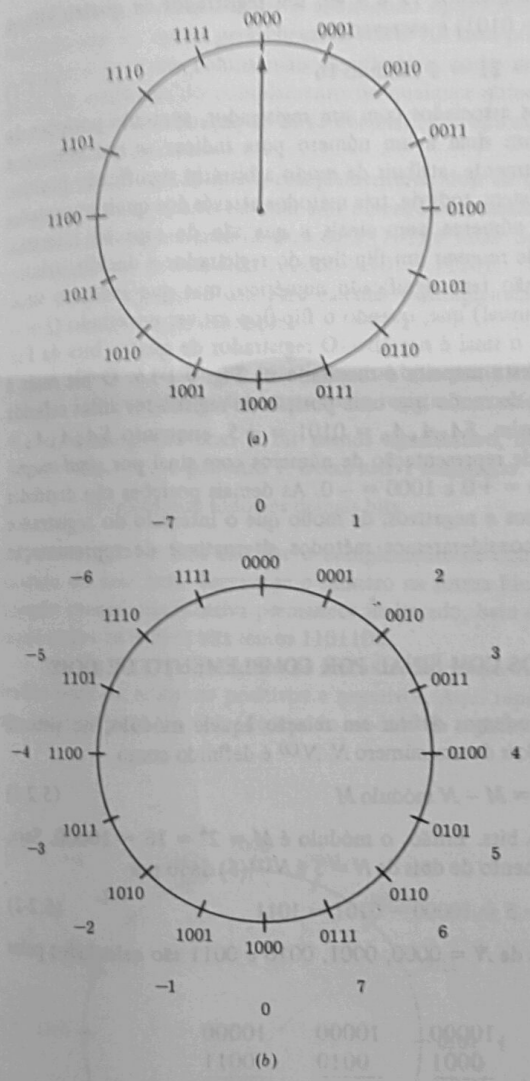


Fig. 5.1-1 (*a*) Um registrador de quatro bits com módulo 16. (*b*) O método sinal-magnitude de atribuir posições do registrador a números positivos e negativos.

posições. O avanço total seria de cinco posições e o ponteiro indicaria, corretamente, 0101 (= 5). Para subtrair 2 de 3, limparíamos o registrador, avançaríamos o ponteiro três posições e a seguir o retardaríamos duas posições. A posição final indicaria, corretamente, 0001 (= 1). O resultado final de uma adição ou subtração seria correto desde que o resultado fosse um número positivo no intervalo de 0 a 15. Se o resultado estiver fora deste intervalo, estará correto somente dentro do

módulo do registrador. Por exemplo, se somarmos 12 e 9 em um registrador de quatro bits, o registrador indicará $(12 + 9) - 16 = 5$ (= 0101) e escrevemos

$$12 + 9 = 21 = 5 \text{ módulo } 16 \tag{5.1-1}$$

Não há sinais de mais ou de menos associados com um registrador, somente posições do registrador. Como poderíamos associar um sinal a um número para indicar se um número é positivo ou negativo? Poderíamos, simplesmente, atribuir de modo arbitrário significado positivo ou negativo a posições do registrador. Existem, todavia, três métodos através dos quais as posições do registrador podem ser associadas a números com sinais e que são de especial interesse.

Um método que se apresenta é o de reservar um flip-flop do registrador e decidir, arbitrariamente, que o estado deste flip-flop não tem significado numérico, mas que indica o sinal. Podemos arbitrar (e este é o significado usual) que, quando o flip-flop estiver no estado $Q = 0$, o sinal é positivo, e que, quando $Q = 1$, o sinal é negativo. O registrador de quatro bits da Fig. 5.1-1*a* com as posições redistribuídas desta maneira é mostrado na Fig. 5.1-1*b*. O bit mais à esquerda dos quatro será o bit de sinal S, de modo que uma posição do registrador antes referida como $A_3A_2A_1A_0$ torna-se $SA_2A_1A_0$. Assim, $SA_2A_1A_0 = 0101 = +5$, enquanto $SA_2A_1A_0 = 1101 = -5$. Este método é chamado de representação de números com sinal por *sinal-magnitude*. Observe-se que há dois zeros, $0000 = +0$ e $1000 = -0$. As demais posições são divididas eqüitativamente entre os números positivos e negativos, de modo que o intervalo do registrador é de -7 a $+7$. Nas próximas seções consideraremos métodos alternativos de representação que são geralmente mais úteis.

5.2 A REPRESENTAÇÃO DE NÚMEROS COM SINAL POR COMPLEMENTO DE DOIS

Tendo sido especificado o módulo M, podemos definir em relação àquele módulo, no sistema binário de aritmética, o *complemento de dois* de um número N. $N^{(2)}$ é definido como

$$N^{(2)} = M - N \text{ módulo } M \tag{5.2-1}$$

Por exemplo, seja um módulo de quatro bits. Então, o módulo é $M = 2^4 = 16 = 10000$. Seja, ainda, o número $N = 5$ (0101). O complemento de dois de $N = 5$ é $N^{(2)}(5)$ dado por

$$N^{(2)}(5) = 2^4 - 5 = 10000 - 0101 = 1011 \tag{5.2-2}$$

Como exemplo, os complementos de dois de $N = 0000$, 0001, 0010 e 0011 são calculados pelas subtrações

	10000	10000	10000	10000
$N =$	0000	0001	0010	0011
$N^{(2)} =$	10000	1111	1110	1101

No primeiro caso, a subtração formal de 0 a 2^4 resulta em 10000, mas, como notamos da Eq. (5.1-1), o resultado deve ser escrito módulo M e 10000 módulo 16 = 0000; o resultado é, pois, 0000 e não 10000.

A avaliação do complemento de dois de um número envolve uma sucessão de subtrações, bit a bit, do número binário 10 (= 2 decimal). Por exemplo, na subtração de 0001 começamos subtraindo 1 de 10. Estamos, pois, procurando um número que adicionado a 1 resulta em 10, isto é,

um número que *complemente* o subtraendo significando que o subtraendo e a diferença complementam um ao outro para chegar à soma 10. Esta primeira subtração gera um empréstimo que é levado à próxima coluna mais significativa, onde, novamente, subtraímos 1 de 10 (= 2). Em geral, na avaliação do complemento de qualquer número após o primeiro 1 do subtraendo cada subtração é uma subtração de 10 (2 decimal). Por esta razão, a quantidade $N^{(2)}$ é chamada *complemento de dois* de N.

Deve ser notado que o complemento de dois de um número *não pode ser especificado com certeza a não ser que o módulo seja especificado*. Assim, quando o módulo for 16 (quatro casas binárias), o complemento de dois de 5 (101) é 1101. Se o módulo fosse 32 (cinco casas binárias), o complemento de dois seria $100000 - 101 = 11101$.

Há um algoritmo útil para calcular o complemento de dois de um número sem o procedimento formal de uma subtração:

1. Escreva o número usando tantos bits quantos forem apropriados para o módulo, isto é, tantos bits quantos forem os do registrador onde será armazenado o número.
2. Começando com o bit menos significativo, deixe inalterados os bits que forem 0 até encontrar o primeiro 1. Deixe este 1 inalterado.
3. Complemente todos os demais bits.

Como exemplo, vamos calcular o complemento de dois do número +20 (decimal) supondo um módulo de sete bits. Escreve-se o número na forma binária com sete bits com 0010100. O 1 na posição menos significativa permanece inalterado, bem como os zeros a sua direita. Complementando todos os outros bits temos 1101100.

Na Fig. 5.2-1, mostramos os métodos de *complemento de dois* de atribuir posições no registrador para os números positivos e negativos. Aqui representamos um número negativo $-N$ por seu complemento de dois (dentro do módulo do registrador) de N; $-N$ é representado por $M - N$.

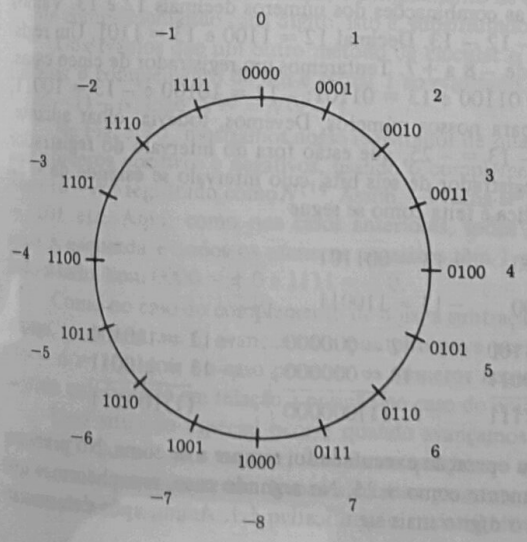


Fig. 5.2-1 A representação por complemento de dois.

No caso presente de um módulo de quatro bits, a representação de -1 é $2^4 - 1 = 10000 - 0001 = 1111$ etc. Observe-se que neste caso há um único zero. Note-se, ainda, que, como no caso da representação por sinal-magnitude, o bit mais à esquerda serve para indicar o sinal. Se este bit for 0, o número será positivo; se o bit for 1, o número será negativo.

O método de atribuição de posições no registrador por complemento de dois tem o mérito de permitir, desde que o resultado esteja dentro do intervalo do registrador, soma pelo avanço do ponteiro e subtração pelo seu retardo. Vamos supor que desejamos encontrar $4 - 6$. Começando em 0, avançamos o ponteiro quatro posições e depois o retardamos seis posições. Se quisermos, podemos efetuar as operações na ordem inversa (atrasar primeiro e depois avançar). Em qualquer caso chegaríamos a 1110, que é a representação correta em complemento de dois de -2 $(= 4 - 6)$. Observe-se que, se a mesma operação houvesse sido tentada com o método de sinal-magnitude, não teríamos chegado a um resultado correto.

A seguir, devemos observar que o registrador de complemento de dois nos permite efetuar a operação de subtração como uma operação de *adição* de um número negativo. Assim, vamos supor que desejamos $N_1 - N_2$. Podemos escrever $N_1 - N_2 = N_1 + (-N_2)$. Como $-N_2$ é representado por $M - N_2$ temos

$$N_1 - N_2 = N_1 + (-N_2) = N_1 + (M - N_2) \tag{5.2-3}$$

Isto significa que, para *subtrair* N_2 de N_1, ajustamos o ponteiro para N_1 e *avançamos* (como se estivéssemos *somando*) o ponteiro um número de casas igual ao complemento de dois de N_2. Um exemplo tornará claro o funcionamento do processo. Consideremos novamente a subtração de $4 - 6$. Após ajustar o ponteiro em 4, devemos atrasá-lo seis posições. Existem dezesseis posições no registrador de quatro bits. Assim, em vez de *atrasar* o ponteiro seis posições, podemos chegar ao mesmo ponto adiantando-o dez posições, pois $10 + 6 = 16$ ou, em outras palavras, atrasar o ponteiro seis posições equivale a adiantá-lo um número de posições equivalente ao complemento de dois de 6.

Um exemplo tornará claro como podemos combinar números quando o método do complemento de dois for seguido. Consideremos as combinações dos números decimais 12 e 13. Vamos calcular $12 + 13$, $12 - 13$, $-12 + 13$ e $-12 - 13$. Decimal $12 = 1100$ e $13 = 1101$. Um registrador de quatro casas abarca o intervalo de -8 a $+7$. Tentaremos um registrador de cinco casas. Em um registrador de cinco casas, $12 = 01100$ e $13 = 01101$; $-12 = 10100$ e $-13 = 10011$, sendo, portanto, o intervalo adequado para nossos números. Devemos, todavia, olhar adiante. Sabemos que $12 + 13 = 25$ e que $-12 - 13 = -25$, que estão fora do intervalo do registrador de cinco bits. Devemos, pois, usar um registrador de seis bits, cujo intervalo se estende de -32 a $+31$, como pode ser verificado. A arimética é feita como se segue

$+12 = 001100$ $\quad +13 = 001101$

$-12 = 110100$ $\quad -13 = 110011$

$+12 = 001100$	$+12 = 001100$	$-12 = 000000$	$-12 = 110100$
$+13 = 001101$	$-13 = 110011$	$+13 = 000000$	$-13 = 110011$
011001	111111	$(1)000000$	$(1)100111$

Note-se que em cada um dos quatro casos a operação executada foi sempre a de soma. No primeiro caso, obtemos 011001, que é lido diretamente como $+25$. No segundo caso, reconhecemos que o resultado é um número negativo, pois o dígito mais significativo é 1. Assim, após determinarmos a magnitude do resultado, devemos acrescentar o sinal de menos. A magnitude é calculada encontrando o complemento de dois de 111111, que é 000001, ou 1. O resultado é, pois, -1. Observe-se que, enquanto o número negativo for mantido dentro do processador digital, ele permanecerá na forma de complemento de dois. Assim, -1 será registrado como 111111. As operações de leitura do sinal e determinação da magnitude geralmente só são necessárias quando se deseja que o resultado seja lido por um operador humano.

No terceiro e no quarto casos, aparecem vai-uns para a sétima casa. Como estes vai-uns que marcamos com parênteses estão fora do intervalo do registrador de seis bits, eles não serão registrados e os números lidos do registrador serão, respectivamente, 000001 e 100111.

5.3 A REPRESENTAÇÃO DE NÚMEROS COM SINAL POR COMPLEMENTO DE UM

Uma vez definido o módulo $M = 2^n$, o complemento de um $N^{(1)}$ de um número N é definido por

$$N^{(1)} = (M - 1) - N \tag{5.3-1}$$

Assim, com um módulo de quatro casas $M - 1 = 2^n - 1 = 1111$ e os complementos de um de $N = 0000$, 0001, 0010 e 0011 são calculados através das subtrações

	1111	1111	1111	1111
$N =$	0000	0001	0010	0011
$N^{(1)} =$	1111	1110	1101	1100

Observe-se que, em cada caso, a determinação do complemento de um é feita através de uma seqüência de subtrações, dígito a dígito, de 1, donde a denominação de *complemento de um*. O cálculo do complemento de um é mais simples que o cálculo do complemento de dois, pois no caso atual não há empréstimo da casa seguinte. Na verdade, não necessitamos efetuar subtrações, pois, como é facilmente observado, o complemento de um de um número pode ser gerado simplesmente complementando cada dígito, isto é, substituindo 0s por 1s e 1s por 0s.

Observamos que um outro método de calcular o complemento de dois de um número seria calcular o complemento de um e somar 1 ao resultado. Isto ocorre porque $N^{(2)} = M - N$ e $N^{(1)} = (M - 1) - N$, logo, $N^{(2)} = N^{(1)} + 1$.

Na Fig. 5.3-1, mostramos nosso registrador de quatro bits onde o método de atribuir posições aos números positivos e negativos, usando o complemento de um, é usado. Neste registrador um número $-N$ é registrado como $N^{(1)}$. Assim, -1 torna-se $1111 - 0001 = 1110$, $-2 = 1111 - 0010 = 1101$ etc. Aqui, como nos casos anteriores, todos os números positivos têm 0 como dígito mais à esquerda e todos os números negativos têm 1 naquela posição. Existem dois zeros, agora lado a lado, com $0000 = +0$ e $1111 = -0$.

Como no caso do complemento de dois, a subtração pode ser efetuada avançando o ponteiro. Assim, para subtrair 6, avançamos o ponteiro nove posições $(15 - 6 = 9)$ em vez de dez posições $(16 - 6 = 10)$, pois no caso presente os números negativos foram deslocados de uma posição no sentido anti-horário em relação à posição no caso do complemento de dois.

Uma situação especial ocorre quando avançamos tanto o registrador que o ponteiro passa por 0000. Quando isto ocorrer, saberemos que houve um erro de uma contagem, pois o ponteiro terá passado por dois zeros. Somos alertados para este fato pela geração de um vai-um na posição uma casa mais significativa que a posição mais signitiva do registrador. Se detectarmos este vai-

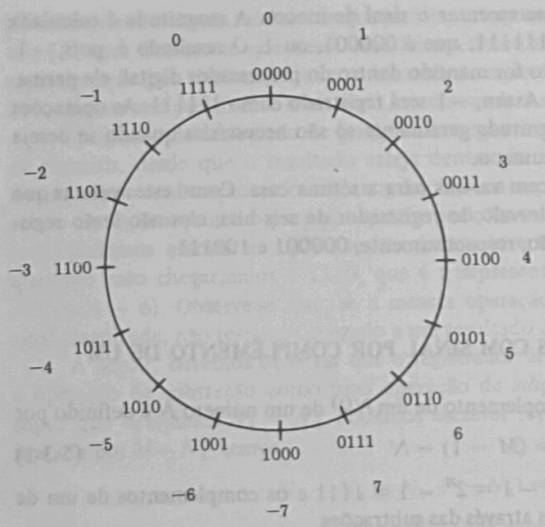


Fig. 5.3-1 A representação por complemento de um.

um, poderemos corrigir o resultado, quando o vai-um assim indicar, somando 1 ao resultado obtido.

Como exemplo do funcionamento do registrador por complemento de um, combinaremos, novamente, os números 12 e 13. Temos

$$+12 = 001100 \qquad +13 = 001101$$
$$-12 = 110011 \qquad -13 = 110010$$

$$\begin{array}{llll} +12 = 001100 & +12 = 001100 & -12 = 110011 & -12 = 110011 \\ +13 = \underline{001101} & -13 = \underline{110010} & +13 = \underline{001101} & -13 = \underline{110010} \\ 011001 & 111110 & 1000000 & 1100101 \\ & & \longrightarrow 1 & \longrightarrow 1 \\ & & 000001 & 100110 \end{array}$$

No primeiro caso, encontramos o resultado 011001 = +25. No segundo caso, reconhecemos que o resultado é negativo. Para encontrar a magnitude, calculamos o complemento de um, que é 000001 = 1, logo, o resultado é −1. No terceiro e no quarto caso, encontramos um vai-um na sétima casa. Devemos, pois somar 1 ao resultado e, isto feito, ignorarmos o vai-um.

5.4 SOMA DE DOIS NÚMEROS BINÁRIOS

Vamos supor que tenhamos dois números binários de n dígitos $A = A_{n-1}A_{n-2} \ldots A_0$ e $B = B_{n-1}B_{n-2} \ldots B_0$ e que desejemos encontrar sua soma. A soma S possivelmente terá um dígito a mais que os dois números, de modo que $S = S_nS_{n-1}S_{n-2} \ldots S_0$. Cada dígito S_i da soma é

unicamente determinado pelos dígitos A_i e B_i dos números que estão sendo somados. Podemos, pois, preparar uma tabela verdade mostrando S_i para todas as combinações possíveis de A_i e B_i. Esta tabela verdade poderia ser implementada por uma estrutura de portas que não precisaria envolver mais de dois níveis de portas. Obviamente, seriam necessárias tantas estruturas quantos fossem os dígitos da soma. Se existisse tal conjunto de estruturas e no tempo $t = 0$ os níveis lógicos que representam os numeros A e B fossem aplicados às entradas das portas, a soma estaria disponível na saída das portas após um tempo $t = 2t_{ap}$, onde t_{ap} seria o tempo de atraso de propagação de uma porta.

Embora esse método de soma numérica seja válido em princípio, geralmente ele não é factível. Não raro os números A e B têm 32 dígitos cada um. Neste caso o número de colunas no lado

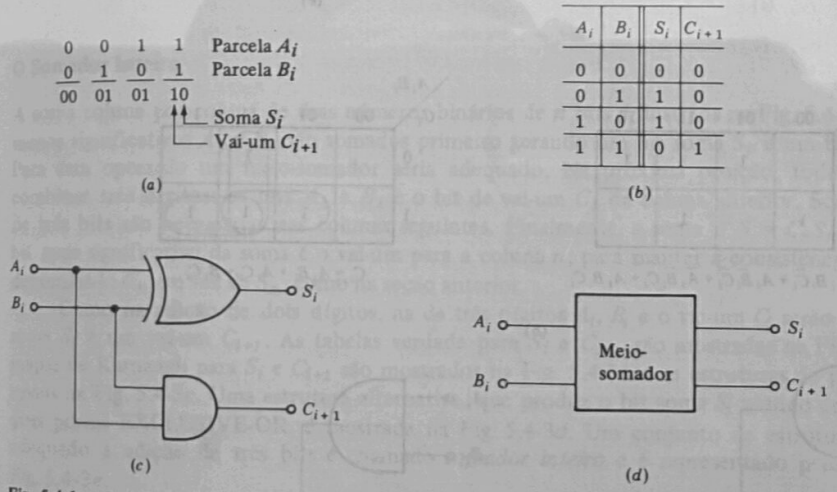


Fig. 5.4-1 (*a*) Adição de dois dígitos binários. (*b*) Tabela verdade para os bits soma e vai-um. (*c*) Realização da tabela verdade usando portas. (*d*) Símbolo do meio-somador.

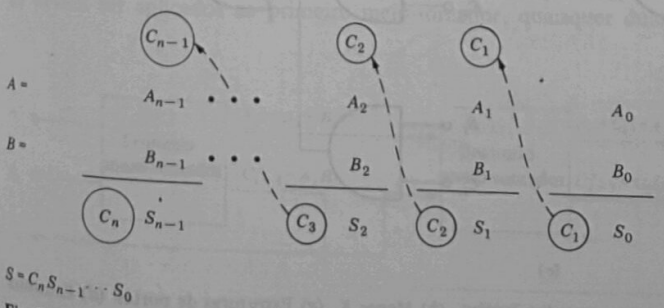
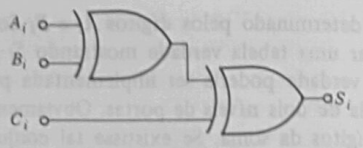


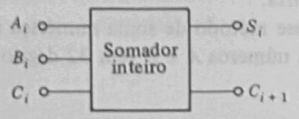
Fig. 5.4-2 A soma de dois números de n dígitos.

Vai-um de entrada C_i	Parcela A_i	Parcela B_i	Soma S_i	Vai-um de saída C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

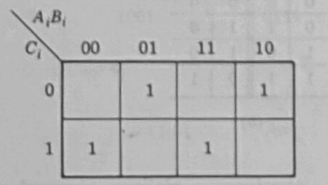
(a)



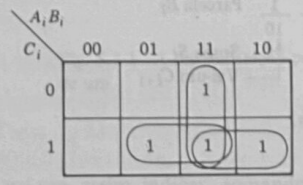
(d)



(e)

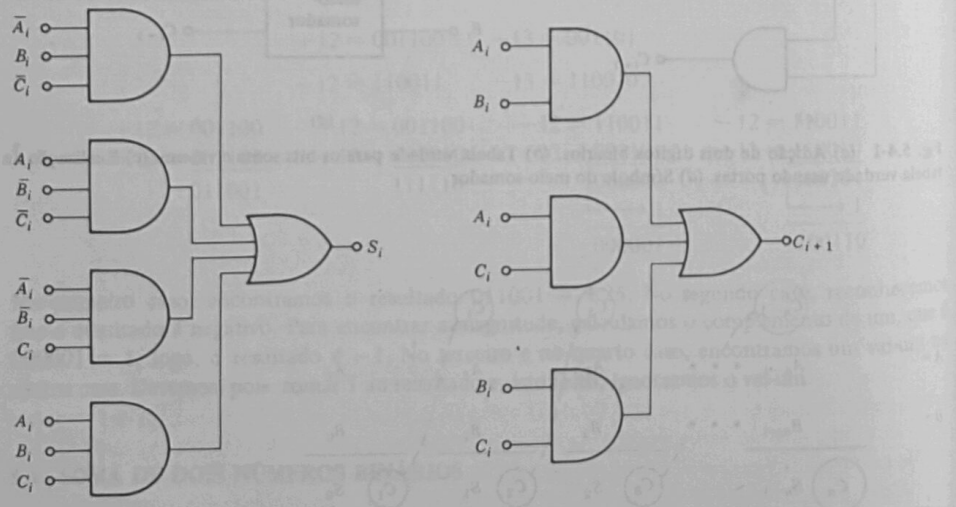


$S_i = \bar{A}_i B_i \bar{C}_i + A_i \bar{B}_i \bar{C}_i + \bar{A}_i \bar{B}_i C_i + A_i B_i C_i$



$C_{i+1} = A_i B_i + A_i C_i + B_i C_i$

(b)



(c)

Fig. 5.4-3 (*a*) Tabela verdade para um somador inteiro. (*b*) Mapas K. (*c*) Estruturas de portas. (*d*) Estrutura alternativa para a soma S_i. (*e*) Símbolo do somador inteiro.

da entrada da tabela verdade é 64, o número de linhas na tabela verdade é $2^{64} \simeq 10^{19}$ e o número de portas envolvidas é similarmente astronômico. Como resultado, os métodos factíveis de soma geralmente envolvem a soma de dígitos coluna a coluna, como no caso de um calculador humano.

Na Fig. 5.4-1*a* somamos uma parcela de um dígito A_i a uma parcela de um dígito B_i. Esta soma gera um dígito soma S_i na mesma coluna e, conseqüentemente, de mesma significância numérica que A_i e B_i. A soma pode também gerar um dígito de vai-um C_{i+1} que possui significância numérica maior que os dígitos que estão sendo somados. A tabela verdade para a geração dos bits de soma e de vai-um é mostrada na Fig. 5.4-1*b*. O bit soma é gerado por uma operação EXCLUSIVE-OR, $S_i = A_i \oplus B_i$ e o vai-um é gerado por uma operação AND, $C_{i+1} = A_i B_i$. As portas são mostradas na Fig. 5.4-1*c*. Esta estrutura de portas é chamada *meio-somador* e representada pelo símbolo da Fig. 5.4-1*d*.

O Somador Inteiro

A soma coluna por coluna de dois números binários de n bits é ilustrada na Fig. 5.4-2. Os dígitos menos significativos A_0 e B_0 são somados primeiro gerando um bit soma S_0 e um bit vai-um C_1. Para esta operação um meio-somador seria adequado. Na próxima posição, todavia, devemos combinar três dígitos: os bits A_1 e B_1 e o bit de vai-um C_1 da coluna anterior. Somas similares de três bits são necessárias nas colunas seguintes. Finalmente, a soma é $S = C_n S_{n-1} \ldots S_0$. O bit mais significativo da soma é o vai-um para a coluna n; para manter a consistência resolvemos denominá-lo C_n em vez de S_n, como na seção anterior.

Como na adição de dois dígitos, na de três dígitos A_i, B_i e o vai-um C_i serão gerados uma soma S_i e um vai-um C_{i+1}. As tabelas verdade para S_i e C_{i+1} são mostradas na Fig. 5.4-3*a*. Os mapas de Karnaugh para S_i e C_{i+1} são mostrados na Fig. 5.4-3*b* e as estruturas de portas de dois níveis na Fig. 5.4-3*c*. Uma estrutura alternativa, que produz o bit soma S_i usando uma cascata de duas portas EXCLUSIVE-OR, é mostrada na Fig. 5.4-3*d*. Um conjunto de estruturas de portas adequado à adição de três bits é chamado *somador inteiro* e é representado pelo símbolo da Fig. 5.4-3*e*.

Um somador inteiro pode ser montado usando dois meio-somadores e uma porta OR, conforme mostrado na Fig. 5.4-4. O primeiro meio-somador gera os bits soma e vai-um S_i' e C_{i+1}' para A_i e B_i, e o segundo meio-somador combina a entrada de vai-um C_i com S_i' para gerar a saída soma S_i e um vai-um C_{i+1}''. O vai-um final é $C_{i+1} = C_{i+1}' + C_{i+1}''$. Embora a figura sugira que A_i e B_i devam ser aplicados ao primeiro meio-somador, quaisquer duas das três entradas C_i, A_i e B_i

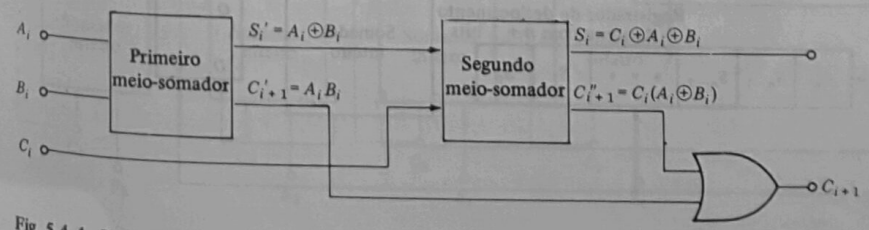


Fig. 5.4-4 Um somador inteiro constituído por dois meio-somadores e uma porta OR.

podem ser entradas para o primeiro meio-somador. A terceira entrada será, então, introduzida no segundo meio-somador.

5.5 UM SOMADOR SÉRIE

Na Fig. 5.5-1 mostramos como um único somador inteiro pode ser usado para somar dois números de n bits $A = A_{n-1}A_{n-2} \ldots A_0$ e $B = B_{n-1}B_{n-2} \ldots B_0$. Aqui a soma tem lugar *em série*, isto é, uma coluna de cada vez, em sincronismo com os ciclos sucessivos do sinal de relógio.

Inicialmente, as parcelas são carregadas em um registrador de deslocamento de n bits. Um registrador de deslocamento de $(n + 1)$ bits está disponível para receber a soma. Os registradores deslocam para a direita e o carregamento dos registradores é feito de tal maneira que os bits menos significativos A_0 e B_0 são colocados nas posições mais à direita dos registradores. Inicialmente, também, supomos que os flip-flops tipo D estejam no estado reset, de modo que $Q = C_i = 0$. Os bits soma serão deslocados para o registrador da esquerda, um bit de cada vez. Durante a entrada da soma, os bits que porventura estiverem no registrador da soma serão deslocados para fora e perdidos. Não há, pois, necessidade de limpar o registrador da soma no início da operação. A soma $S = C_n S_{n-1} S_{n-2} \ldots S_0$, indicada como sendo registrada no registrador de soma, só estará registrada ao final da soma e não no princípio.

Inicialmente, então, antes da primeira borda de gatilho do sinal de relógio, os bits A_0 e B_0 estarão presentes às entradas do somador inteiro enquanto a entrada C_i será $C_i = 0$. No mesmo instante, a soma S_0 e o vai-um C_0 gerados por estes bits estarão presentes às saídas do somador S_i e C_{i+1} (supomos que o atraso de propagação através do somador inteiro, que é um circuito combinacional, é pequeno em relação ao período do relógio). Na primeira borda de gatilho do relógio ocorrerão diversos eventos. O bit soma S_0 será registrado no flip-flop mais à esquerda

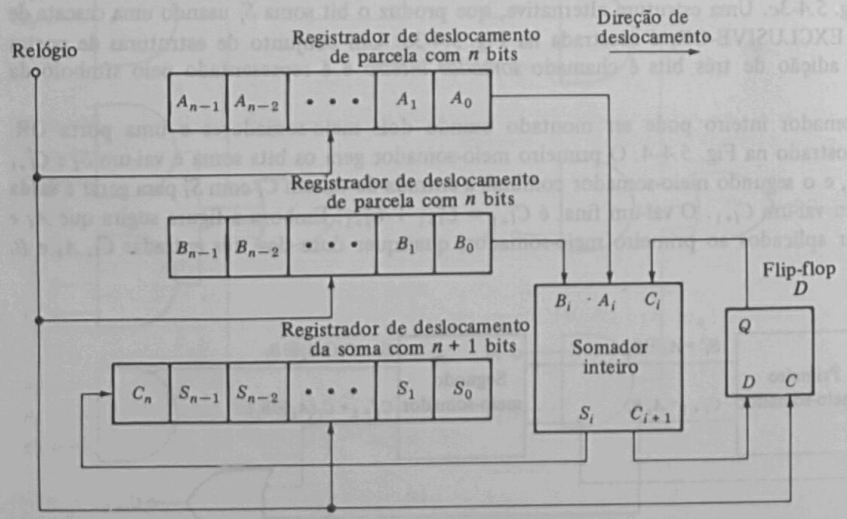


Fig. 5.5-1 Um somador série.

do registrador de soma. Os registradores das parcelas deslocarão uma posição, de modo que os bits seguintes, A_1 e B_1, aparecerão nas entradas do somador inteiro. A entrada C_1 do flip-flop tipo D será transferida para a saída, de modo que C_1 aparecerá na entrada C_i do somador. O propósito do flip-flop é prover um atraso de um ciclo de relógio, de modo que o vai-um gerado na soma de uma coluna apareça somente quando deva ser somado aos bits da coluna seguinte.

Após esta primeira borda do relógio, o primeiro bit da soma terá sido registrado, as entradas do somador serão A_1, B_1 e C_1 e as saídas do somador serão S_1 e C_2. A segunda borda move S_0 uma casa para a direita no registrador soma e registra o novo bit soma calculado S_1 no flip-flop mais à esquerda do registrador de soma. Esta mesma borda avança A_2 e B_2 para o somador inteiro e transfere C_2 para a entrada C_i. Assim, de ciclo em ciclo do relógio, os bits soma são calculados, entram no registrador de soma pela esquerda e são deslocados para a direita para fazer lugar para os bits calculados sucessivamente. Após n ciclos de relógio, os registradores A e B terão sido esgotados; teremos $A_i = B_i = 0$ e C_i será o vai-um C_n gerado pela soma $A_{n-1} + B_{n-1} + C_{n-1}$. A $(n + 1)$ésima borda do relógio transfere este último bit para o registrador de soma e a operação se completa. Neste instante o relógio deve ser desligado.

5.6 SOMA EM PARALELO

O método de soma série da seção anterior é econômico do ponto de vista de hardware, mas é relativamente lento. É necessário um sistema síncrono e uma soma de n bits requer $(n + 1)$ ciclos de relógio. Um método de soma mais rápido em que todos os bits das parcelas são somados *em paralelo*, isto é, simultaneamente, é mostrado na Fig. 5.6-1. O método usa mais hardware, pois requer tantos somadores completos quantos forem os bits dos números a serem somados. Em princípio, o primeiro somador usado para combinar A_0 e B_0 poderia ser um meio-somador. Se um somador inteiro for usado, a entrada de vai-um do somador deve ser $C_{in} = C_0 = 0$. O primeiro somador combina, então, A_0 e B_0 para obter um bit de soma S_0 e um bit de vai-um C_1. Este bit de vai-um é combinado no somador seguinte com os bits A_1 e B_1 para gerar S_1 e C_2, e assim por diante.

É necessário que todos os bits A_i e B_i estejam disponíveis ao mesmo tempo para serem aplicados às entradas dos somadores. Assim, deve ser entendido na Fig. 5.6-1 que os números

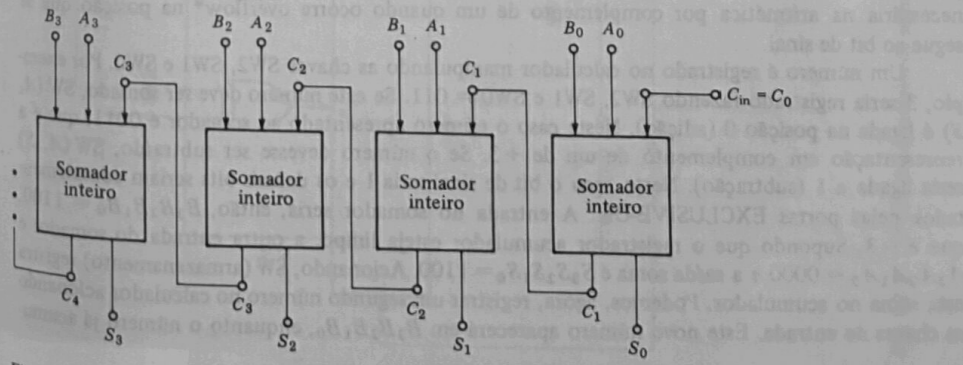


Fig. 5.6-1 Um somador paralelo usando um conjunto de somadores inteiros.

A e B estão disponíveis em registradores. Estes registradores, que não aparecem na figura, não precisam ser registradores de deslocamento, mas, simplesmente, *registradores de armazenamento*.

5.7 UM CALCULADOR SIMPLES DE SOMA E SUBTRAÇÃO

O diagrama da Fig. 5.7-1 mostra um calculador simples onde combinamos, por soma e subtração, uma seqüência de números binários. Os números são registrados de uma maneira conveniente para o operador humano. Como normalmente ocorre com calculadores, o operador registrará a magnitude do número e a indicação da operação a ser executada, adição ou subtração. A indicação de saída, para conveniência do operador, será na forma sinal-magnitude. Dentro do calculador os números aparecerão na representação de complemento de um e as manipulações numéricas serão feitas de forma adequada a esta representação. O resultado acumulado da combinação das entradas é guardado em um registrador de armazenamento denominado, apropriadamente, *acumulador*. Supomos que existe uma maneira de limpar o registrador através, por exemplo, do acionamento das entradas de reset-direto dos flip-flops do registrador. Para simplificar, esta característica não é mostrada no desenho. O registrador tem uma linha de relógio ligada a todas as entradas de relógio dos flip-flops. Supomos que os flip-flops tipo D respondem à transição do relógio de 1 para 0. Supomos, ainda, que, quando a chave SW (armazenamento) estiver aberta, as entradas de relógio são mantidas no nível 0 por um circuito que não é mostrado. O fechamento momentâneo da chave SW (armazenamento) ligará a linha de relógio ao nível lógico 1, fazendo com que a linha de relógio passe de 0 para 1 e depois retorne a 0. Assim, o rápido acionamento de SW (armazenamento) aplica um ciclo de relógio à linha de relógio. Embora o calculador possa ser expandido para um número arbitrário de bits, aqui o restringimos a quatro bits para simplificar o desenho. O intervalo da soma acumulada deve ser, pois, de $+7$ a -7. O bit na posição mais significativa é o bit de sinal.

No somador de quatro bits há quatro somadores inteiros individuais. Os terminais de entrada $A_3A_2A_1A_0$ e $B_3B_2B_1B_0$ são mostrados, bem como os terminais de saída $S_3S_2S_1S_0$. Também aparecem a entrada de vai-um C_{in} do primeiro (menor ordem) somador inteiro e a saída de vai-um C_4 do último somador inteiro. As demais ligações de vai-um e vem-um (C_1 a C_2, C_2 a C_3 etc.) não são mostradas explicitamente, pois não precisamos ter acesso a estes terminais. Note-se, em particular, a ligação do vai-um de saída C_4 à entrada de vai-um C_{in}. Esta ligação provê a correção necessária na aritmética por complemento de um quando ocorre overflow* na posição que se segue ao bit de sinal.

Um número é registrado no calculador manipulando as chaves SW2, SW1 e SW0. Por exemplo, 3 seria registrado fazendo SW2, SW1 e SW0 = 011. Se este número deve ser somado, SW (A, S) é ligada na posição 0 (adição). Neste caso o número apresentado ao somador é 0011, que é a representação em complemento de um de $+3$. Se o número devesse ser subtraído, SW (A, S) seria ligada a 1 (subtração). Neste caso o bit de sinal seria 1 e os demais bits seriam complementados pelas portas EXCLUSIVE-OR. A entrada do somador seria, então, $B_3B_2B_1B_0 = 1100$, que é -3. Supondo que o registrador acumulador esteja limpo, a outra entrada do somador é $A_3A_2A_1A_0 = 0000$ e a saída soma é $S_3S_2S_1S_0 = 1100$. Acionando, SW (armazenamento) registra esta soma no acumulador. Podemos, agora, registrar um segundo número no calculador acionando as chaves de entrada. Este novo número aparecerá em $B_3B_2B_1B_0$, enquanto o número já acumu-

* Transbordamento. Ocorre, no caso, se o número da soma cair fora do intervalo $+7$ a -7.

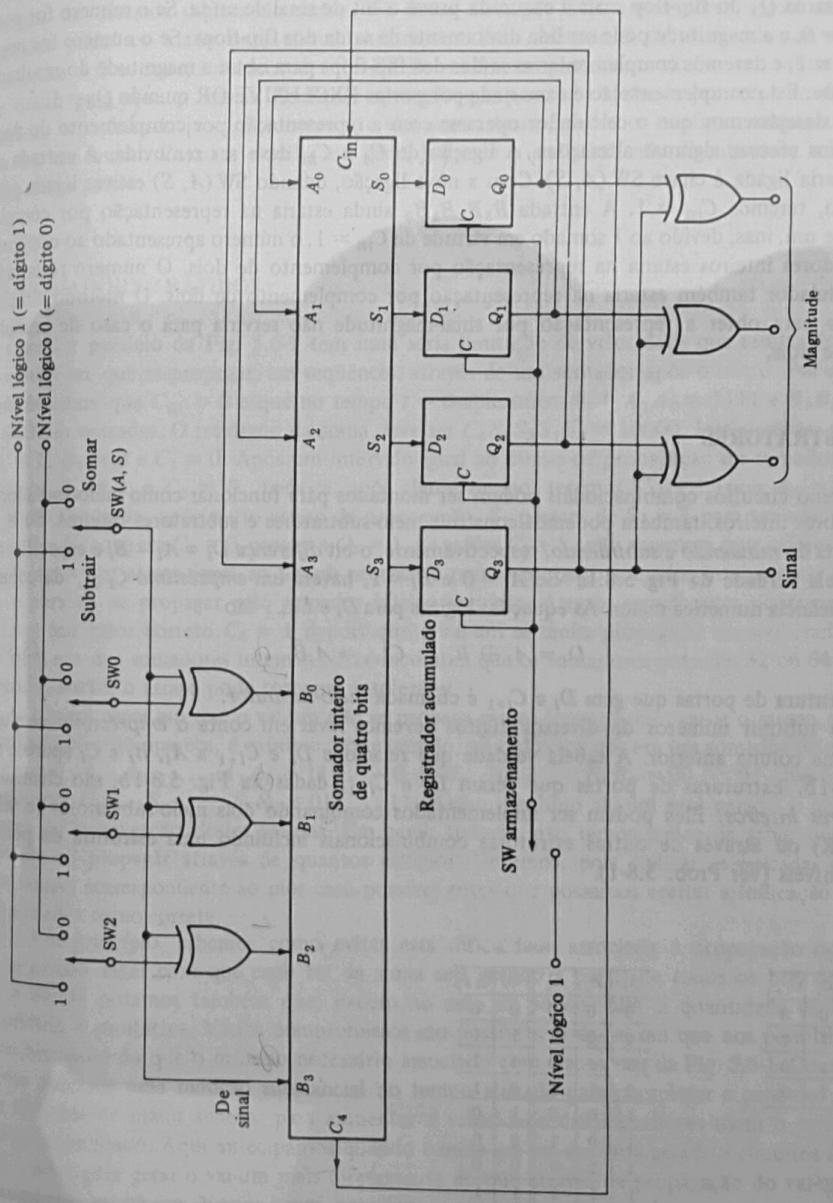


Fig. 5.7-1 Um calculador simples de soma e subtração.

lado aparecerá em $A_3A_2A_1A_0$. Acionando, SW (armazenamento) adicionará a nova entrada à soma já existente no acumulador e registrará a nova soma no acumulador.

A saída Q_3 do flip-flop mais à esquerda provê o bit de sinal de saída. Se o número for positivo, $Q_3 = 0$, e a magnitude pode ser lida diretamente da saída dos flip-flops. Se o número for negativo, $Q_3 = 1$, e devemos complementar as saídas dos flip-flops para obter a magnitude do resultado acumulado. Esta complementação é executada por portas EXCLUSIVE-OR quando $Q_3 = 1$.

Se desejássemos que o calculador operasse com a representação por complemento de dois, deveríamos efetuar algumas alterações. A ligação de C_4 a C_{in} deve ser removida. A entrada de vai-um seria ligada à chave SW (A, S). Com a nova ligação, quando SW (A, S) estiver ligada para subtração, teremos $C_{\text{in}} = 1$. A entrada $B_3B_2B_1B_0$ ainda estaria na representação por complemento de um, mas, devido ao 1 somado em virtude de $C_{\text{in}} = 1$, o número apresentado ao conjunto de somadores inteiros estaria na representação por complemento de dois. O número registrado no acumulador também estaria na representação por complemento de dois. O método simples mostrado para obter a representação por sinal-magnitude não serviria para o caso de complemento de dois.

5.8 SUBTRATORES

Assim como circuitos combinacionais podem ser montados para funcionar como meio-somadores e somadores inteiros, também podemos construir meio-subtratores e subtratores inteiros. Se A_i e B_i são bits do *minuendo* e *subtraendo*, respectivamente, o bit *diferença* $D_i = A_i - B_i$ é especificado pela tabela verdade da Fig. 5.8.1*a* Se $A_i = 0$ e $B_i = 1$, haverá um *empréstimo* C_{i+1} da coluna de significância numérica maior. As equações lógicas para D_i e C_{i+1} são

$$D_i = A_i \oplus B_i \qquad C_{i+1} = \bar{A}_i B_i \tag{5.8-1}$$

Uma estrutura de portas que gera D_i e C_{i+1} é chamada *meio-subtrator*.

Ao subtrair números de diversos dígitos devemos levar em conta o *empréstimo* vem-um C_i de uma coluna anterior. A tabela verdade que relaciona D_i e C_{i+1} a A_i, B_i e C_i aparece na Fig. 5.8-1*b*. Estruturas de portas que geram D_i e C_{i+1}, dadas na Fig. 5.8-1*b*, são chamadas *subtratores inteiros*. Eles podem ser implementados combinando dois meio-subtratores (e uma porta OR) ou através de outras estruturas combinacionais incluindo uma estrutura de portas de dois níveis (ver Prob. 5.8-1).

A_i	B_i	D_i	C_{i+1}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	1	0

(*a*)

A_i	B_i	C_i	D_i	C_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

(*b*)

Fig. 5.8-1 Tabela verdade para (*a*) um meio-subtrator e (*b*) um subtrator inteiro.

Um conjunto de subtratores inteiros em cascata inteiramente análogo ao conjunto de somadores inteiros da Fig. 5.6-1 pode ser usado para efetuar a operação de subtração de um número de diversos dígitos de outro. Na nossa discussão sobre somadores, vimos como um somador pode ser usado para efetuar subtrações. Para subtrair B de A usando um somador, formamos $-B$ e, após, somamos $A + (-B)$. De modo completamente similar, um subtrator pode ser usado para efetuar somas. Qualquer operação que possa ser efetuada por um somador, pode também ser efetuada por um subtrator, e vice-versa. Assim, podemos confinar nossa atenção em um ou em outro. Como a escolha em geral favorece o somador, não discutiremos subtratores além deste ponto.

5.9 SOMADORES RÁPIDOS

O somador paralelo da Fig. 5.6-1 tem uma séria limitação de velocidade que resulta do fato de o vai-um ter que se propagar, em seqüência, através de um somador após o outro. Por exemplo, consideremos que $C_{\text{in}} = 0$ e que no tempo $t = 0$ aplicamos $A_3A_2A_1A_0 = 1111$ e $B_3B_2B_1B_0 = 0001$ às entradas. O resultado da soma deve ser $C_4S_3S_2S_1S_0 = 10000$. Inicialmente, em $t = 0$ $A_1 = 1$, $B_1 = 0$ e $C_1 = 0$. Após um intervalo igual ao atraso de propagação do somador inteiro, teremos $S_1 = 1$ e $C_2 = 0$. Todavia, após algum tempo, teremos $C_1 = 1$ (pois $A_0 = B_0 = 1$) e, como resultado, após outro atraso de propagação, S_1 passará de $S_1 = 1$ para seu valor correto $S_1 = 0$ e C_2, que era $C_2 = 0$, passará a $C_2 = 1$. As saídas C_2 e S_1 não assumem seus valores corretos senão após um atraso igual ao tempo necessário para o primeiro somador inteiro gerar o vai-um C_1 e para C_1 se propagar pelo somador inteiro seguinte. Atrasos semelhantes ocorrem e C_4 só assume seu valor correto $C_4 = 1$ depois que o vai-um se tenha propagado em seqüência através de cada um dos somadores inteiros. Não é incomum que os somadores possuam 32 ou 64 estágios, quando, então, o atraso pode tornar-se intolerável.

Haverá casos em que o vai-um não se propaga muito longe; nestes casos o atraso, até que o resultado correto apareça, é mínimo. Por exemplo, suponhamos que em um somador de n estágios tenhamos $A_{n-1}A_{n-2} \ldots A_1A_0 = 11 \ldots 10$ e $B_{n-1}B_{n-2} \ldots B_1B_0 = 00 \ldots 01$. Neste caso a soma será $C_nS_{n-1}S_{n-2} \ldots S_1S_0 = 011 \ldots 11$. Assim, nenhum vai-um será gerado e não haverá atraso de propagação destes vai-uns. Em geral, todavia, não temos meios de saber que vai-uns deverão se propagar através de quantos estágios. Devemos, pois, aplicar as entradas e esperar um tempo correspondente ao pior caso possível antes que possamos aceitar a indicação de saída do somador como correta.

Em princípio, sabemos como evitar esta dificuldade associada à propagação do vai-um; é necessário fazer com que *cada* bit da soma seja gerado a partir de *todos* os bits de entrada A_i e B_i. Já notamos também que, exceto no caso de poucos bits, a quantidade de hardware necessária é proibitiva. Vários compromissos são possíveis, todavia, em que nos permitimos usar mais hardware do que o mínimo necessário associado com o somador da Fig. 5.6-1. Como recompensa obtemos uma redução substancial no tempo alocado para completar o processo da soma. Os métodos de maior sucesso para aumentar a velocidade dos somadores usam o princípio do *vai-um antecipado*. Aqui antecipamos quando e onde um vai-um seria gerado e circuitos adicionais são usados para gerar o vai-um mais diretamente do que através da propagação do vai-um através de estágios anteriores. Vamos agora considerar o esquema mais popular para se atingir esta finalidade.

5.10 O SOMADOR COM VAI-UM ANTECIPADO

Considerando novamente que A_i e B_i sejam os bits de entrada na iésima coluna da soma de um número com diversos dígitos, definimos as quantidades G_i e P_i como

$$G_i = A_i B_i \tag{5.10-1}$$

$$P_i = A_i \oplus B_i \tag{5.10-2}$$

onde G_i é chamada *variável geração* e P_i *variável propagação*. Os termos são apropriados pelas seguintes razões: se $G_i = 1$ é porque A_i e B_i são $A_i = B_i = 1$. Neste caso há um vai-um de saída C_{i+1}, independente da existência de um vai-um de entrada C_i. O vai-um é *gerado* pelos valores dos bits na coluna i. De modo similar, se $P_i = 1$ é porque $A_i = 1$ ou $B_i = 1$ (mas *não ambos*) e só haverá um vai-um de saída C_{i+1} se existir um vai-um de entrada C_i. Isto significa que o vai-um de saída não é gerado na coluna, mas, se existir um vai-um de entrada, será *propagado* através da coluna.

Agora é possível expressar todos os bits de soma e de vai-um do somador da Fig. 5.6-1 explicitamente em função do vai-um de entrada C_i e das variáveis G_i e P_i. Inicialmente temos

$$S_i = A_i \oplus B_i \oplus C_i \tag{5.10-3a}$$

$$= P_i \oplus C_i \tag{5.10-3b}$$

Também temos

$$C_{i+1} = A_i B_i + A_i C_i + B_i C_i \tag{5.10-4}$$

Isto é, $C_{i+1} = 1$ sempre que duas ou mais das entradas A_i, B_i e C_i forem 1. Uma forma alternativa de escrever C_{i+1} é

$$C_{i+1} = A_i B_i + C_i(A_i \oplus B_i) \tag{5.10-5}$$

Esta expressão para C_{i+1} não é tão simples quanto a da Eq. (5.10-4), mas serve a nossos propósitos, pois agora podemos reescrever a Eq. (5.10-4) na forma

$$C_{i+1} = G_i + P_i C_i \tag{5.10-6}$$

Aplicando a Eq. (5.10-6) a cada estágio, encontramos

$$C_1 = G_0 + P_0 C_0 \tag{5.10-7a}$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 \tag{5.10-7b}$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \tag{5.10-7c}$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \tag{5.10-7d}$$

e assim por diante. Estas equações, quando postas em palavras, são razoáveis. Considere a equação de C_4. Diz que o vai-um aparece na saída do estágio 3 se for gerado naquele estágio OU se for gerado no estágio 2 E propagado através do estágio 3 OU etc.

Um somador de diversos bits é construído da seguinte maneira. Primeiro substituímos o somador inteiro da Fig. 5.6-1 por unidades como a da Fig. 5.10-1. Estas unidades são chamadas

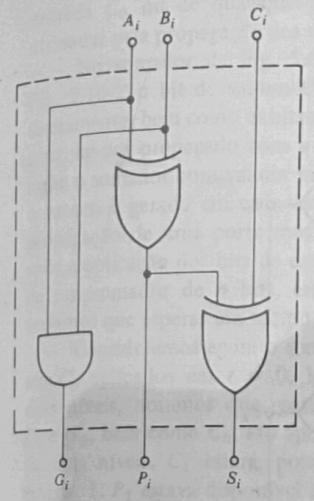


Fig. 5.10-1 Uma unidade somadora que recebe A_i, B_i e C_i e gera S_i, G_i e P_i.

unidades somadoras para distingui-las do somador inteiro da Fig. 5.6-1. Estas unidades têm algumas características em comum com o somador inteiro. Ambos aceitam três entradas, dois bits de entrada e a entrada de vai-um da coluna anterior. Ambos apresentam a saída de soma. Enquanto o somador inteiro fornece o vai-um para o estágio seguinte, a unidade somadora fornece uma saída de propagação P_i e uma de geração G_i. O vai-um para o estágio seguinte é fornecido por uma estrutura auxiliar. A conjugação de unidades somadoras e estruturas auxiliares constitui-se em um sistema completo de soma. Um sistema deste tipo, que acomoda uma entrada de vai-um C_0 e dois números de quatro bits, é mostrado na Fig. 5.10-2. Como pode ser antecipado da Eq. (5.10-7) e pode ser visto na Fig. 5.10-2, os circuitos, para gerarem o bit de vai-um em cada estágio, tornam-se progressivamente mais complexos à medida que avançamos de estágio para estágio.

Na Fig. 5.10-2 adicionamos as portas G1 e G2, que não seriam necessárias se, continuando o modo pelo qual os vai-uns são gerados, simplesmente desejássemos produzir o vai-um C_4. Neste caso, G1 seriam dispensáveis e incluiríamos as ligações tracejadas. A modificação foi feita para obter dois sinais lógicos adicionais, *propagação de grupo* $P(0-3)$ e *geração de grupo* $G(0-3)$, cujos propósitos serão considerados na seção seguinte. No momento registramos, apenas, que

$$P(3{-}0) = P_3 P_2 P_1 P_0 \tag{5.10-8}$$

enquanto

$$G(3{-}0) = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 \tag{5.10-9}$$

Se $P(3-0) = 1$, um vai-um de entrada C_0 será propagado através de todo o grupo, enquanto, se $G(3-0) = 1$, haverá um vai-um de saída porque um vai-um é gerado em um dos quatro estágios e propagado até a saída. Note-se que $P(3-0)$ e $G(3-0)$ não dependem do vai-um de

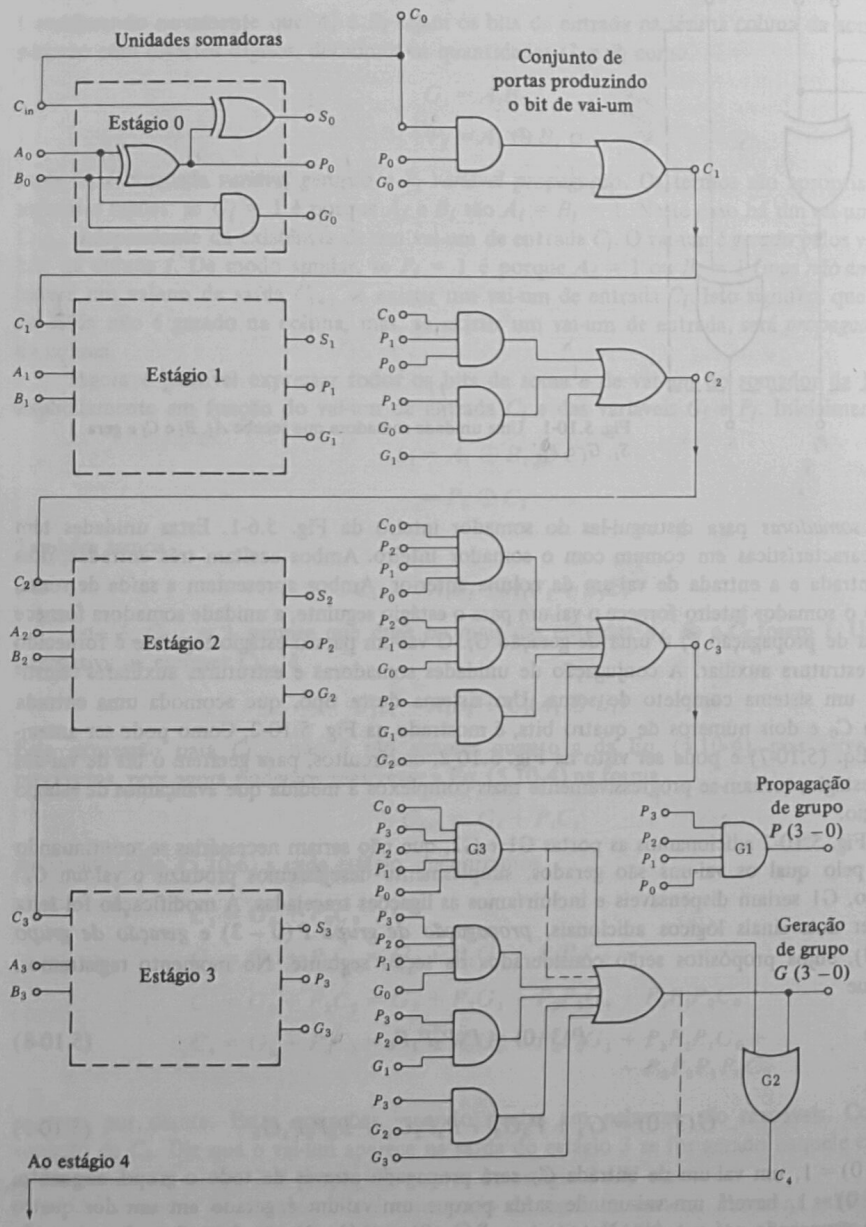


Fig. 5.10-2 Um somador para diversos bits com vai-um antecipado.

entrada C_0 ou de quaisquer vai-uns gerados dentro da unidade. Assim, estas funções não são atrasadas pela propagação dos vai-uns.

No somador da Fig. 5.6-1 o vai-um se propaga de estágio para estágio. No arranjo da Fig. 5.10-2 o bit de vai-um C_{i+1} é gerado em cada estágio usando os bits de entrada A_i e B_i diretamente, bem como os bits de entrada de ordem inferior. Comparemos a velocidade do somador com vai-um propagado com a do somador com vai-um antecipado. Consideremos em primeiro lugar o somador com vai-um propagado. Notamos, da Fig. 5.4-2, que (como geralmente é o caso) o vai-um é gerado em cada estágio em uma estrutura de portas de dois níveis. Se o atraso de propagação de uma porta média for t_{ap}, em cada estágio o vai-um aparece em um tempo $2t_{ap}$ após a aplicação dos bits de entrada. Se o vai-um tem que se propagar através de todos os estágios de um somador de n bits, este é o pior caso possível; após a aplicação dos bits de entrada, teríamos que esperar um tempo $2n\ (t_{ap})$ antes de podermos ler a saída do somador com confiança.

Consideremos agora o somador com vai-um antecipado. Sejam os bits de entrada e o vai-um em C_0 aplicados em $t = 0$. Lembrando que uma porta EXCLUSIVE-OR é uma estrutura de dois níveis, notamos que, para $t = 2t_{ap}$, teremos P_0 e G_0 e que para $t = 4t_{ap}$ teremos S_0. P_0 e G_0, bem como C_0, são aplicados ao primeiro gerador de vai-um, que também é uma estrutura de dois níveis. C_1 estará, pois, disponível em $t = 4t_{ap}$. Então C_1 e P_0 serão combinados no estágio 1. P_1 estava disponível em $t = 2t_{ap}$. Para obter o bit soma S_1, teremos que esperar mais $2t_{ap}$ para a propagação através da segunda porta EXCLUSIVE-OR. S_1 estará, pois, disponível em $t = 6t_{ap}$. Assim como C_1 estava disponível em $t = 4t_{ap}$, também C_2, C_3 etc. estarão. Conseqüentemente, os bits soma S_2, S_3 etc. também estarão disponíveis em $t = 6t_{ap}$. Logo, após ultrapassado o primeiro estágio, não há mais atraso na geração dos bits soma. Em um somador de poucos estágios, a diferença entre a velocidade dos dois tipos de somador é pequena. Para $n = 4$ o atraso é de $8t_{ap}$ para o somador por propagação e $6t_{ap}$ para o somador antecipado. Por outro lado, quando o número de estágios for grande, a diferença é substancial.

Apresentamos P_i como sendo dado por $P_i = A_i \oplus B_i$. A atribuição desta função lógica a P_i é intuitivamente agradável e a unidade somadora da Fig. 5.10-1 tem, em função disto, a característica simplificadora que P_i é gerado pela mesma estrutura usada para gerar S_i. Na realidade, P_i pode ser, também, dado por $P_i = A_i + B_i$ e, como a porta OR tem um atraso de propagação menor que a porta EXCLUSIVE-OR, o uso da operação OR pode aumentar a velocidade de funcionamento. Podemos facilmente verificar que P_i pode ser escrito como $P_i = A_i + B_i$ preparando um mapa K para uma função das três variáveis A_i, B_i e C_i e registrando no mapa a função C_{i+1} da Eq. (5.10-6), que é

$$C_{i+1} = G_i + P_iC_i = A_iB_i + (A_i \oplus B_i)C_i$$
$$= A_iB_i + A_i\bar{B}_iC_i + \bar{A}_iB_iC_i \qquad (5.10\text{-}10)$$

Lendo o mapa de modo a obter o resultado mais simples, encontramos

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i = A_iB_i + (A_i + B_i)C_i \qquad (5.10\text{-}11)$$

A razão para este resultado é, claramente, que, se $A_i = B_i = 1$, um vai-um será gerado e transferido para o estágio seguinte de qualquer maneira. É *irrelevante*, pois, se o mecanismo de propagação também gerar um vai-um neste caso. Isto significa que podemos adicionar à operação EXCLUSIVE-OR $A_i\bar{B}_i + \bar{A}_iB_i$ um termo irrelevante A_iB_i para obter $A_i\bar{B}_i + \bar{A}_iB_i + A_iB_i = A_i + B_i$.

5.11 VAI-UM ANTECIPADO APLICADO A GRUPOS

É aparente que se o método da Fig. 5.10-2 fosse aplicado à soma de números com muitos bits, os circuitos envolvidos tornar-se-iam extremamente complicados. Computadores grandes freqüentemente usam números de 64 bits. Neste caso a estrutura necessária para gerar somente o último vai-um usaria 65 portas com até 65 entradas. Imaginamos, então, se não seria melhor usarmos o princípio do vai-um antecipado em grupos menores de bits. Poderíamos dividir os bits de entrada em grupos de 4, usar o método de vai-um antecipado dentro de cada grupo e deixar que o vai-um se propagasse de grupo para grupo. (A partição em grupos de 4 é sugerida pelo fato de os circuitos integrados comerciais disponíveis serem, geralmente, projetados para acomodar quatro bits.)

Com estas considerações em mente, observemos o arranjo da Fig. 5.11-1, que é um somador projetado para acomodar dezesseis bits. Cada bloco retangular representa a montagem completa da Fig. 5.10-2. Observe-se a convenção pela qual pudemos reduzir o número de linhas que precisam ser desenhadas para representar as entradas e saídas do bloco. Por exemplo, as quatro linhas de entrada A_3, A_2, A_1 e A_0 são indicadas por A_3-A_0. Uma única seta de linha de entrada é, então, desenhada. Todavia, o traço desenhado cortando a linha e o número 4 tornam claro que a linha representa quatro linhas de entrada. Dentro de cada bloco o princípio do vai-um antecipado é usado, mas de um bloco para o outro temos a propagação simples do vai-um. As saídas $P(3-0)$ e $G(3-0)$ da primeira unidade e as saídas correspondentes das demais unidades não são usadas.

Vamos supor, novamente, que as entradas A_i e B_i, bem como o vai-um de entrada C_0, são aplicados em $t = 0$. Quanto tempo teremos que esperar para ter certeza de que a indicação de saída do somador $C_{16}S_{15}S_{14} \ldots S_1S_0$ está correta? Como não estamos usando as saídas de propagação e geração de grupo P e G, vamos supor, na Fig. 5.10-2, que a ligação através da qual o último vai-um C_4 é produzido é aquela indicada pela linha tracejada. Neste caso o atraso de propagação através da porta G2 é eliminado.

Como já vimos, o vai-um C_4 da primeira unidade de quatro bits estará disponível em $t = 4\,t_{ap}$, ao passo que os bits soma S_3-S_0 estarão disponíveis para $t = 6\,t_{ap}$. Na segunda unidade de quatro bits, o vai-um de entrada C_4 é aplicado em $t = 4\,t_{ap}$, logo, seus vai-uns C_5, C_6, C_7 e C_8 estarão disponíveis em $t = 6\,t_{ap}$ e os bits soma S_7-S_4 estarão disponíveis em $t = 8\,t_{ap}$. Isto ocorre porque as entradas A_7-A_4 e B_7-B_4 foram aplicadas $4\,t_{ap}$ antes da aplicação de C_4. Durante este tempo a propagação necessária através das unidades somadoras do segundo grupo já terá ocorrido. Enquanto o primeiro somador de quatro bits requer um tempo de espera de $6\,t_{ap}$, o segundo somador de quatro bits requer um tempo adicional $4\,t_{ap}$ ao tempo $4\,t_{ap}$ necessário para gerar C_4, num total de $8\,t_{ap}$. De modo correspondente, doze bits estarão disponíveis após $4\,t_{ap} + 4\,t_{ap} + 4\,t_{ap} = 12\,t_{ap}$ e dezesseis bits e o último vai-um C_{16} estarão disponíveis após $16\,t_{ap}$. Um somador de dezesseis bits sem vai-um antecipado necessitaria $32\,t_{ap}$. Em geral, se usamos um único somador de quatro bits, o tempo é $6\,t_{ap}$ contra $8\,t_{ap}$. Todavia, se usamos uma cascata de mais de um somador de quatro bits, o somador com vai-um antecipado apresenta a saída correta na metade do tempo necessário para o somador sem vai-um antecipado.

O princípio do vai-um antecipado pode ser incorporado a um somador de maneira mais interessa com maior vantagem em relação à velocidade, como veremos na seção seguinte.

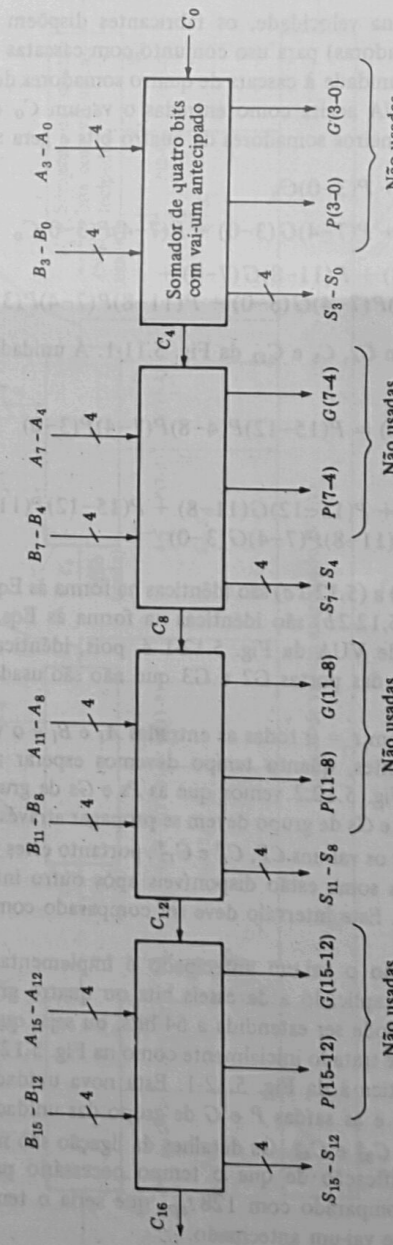


Fig. 5.11-1 Uma cascata de quatro somadores de quatro bits com vai-um antecipado. Cada somador de quatro bits com vai-um antecipado é como mostrado na Fig. 5.10-2. A cascata acomoda dezesseis bits.

5.12 USO DE VAI-UM ANTECIPADO ADICIONAL

Para efetuar melhoria adicional na velocidade, os fabricantes dispõem de unidades de vai-um antecipado (sem as unidades somadoras) para uso conjunto com cascatas de somadores de quatro bits. O modo de conexão de tal unidade à cascata de quatro somadores de quatro bits é mostrado na Fig. 5.12-1. A unidade de VUA aceita como entradas o vai-um C_0 e as saídas de geração e propagação de grupo dos três primeiros somadores de quatro bits e gera saídas

$$C_4^* = G(3-0) + P(3-0)C_0 \tag{5.12-1a}$$

$$C_8^* = G(7-4) + P(7-4)G(3-0) + P(7-4)P(3-0)C_0 \tag{5.12-1b}$$

$$C_{12}^* = G(11-8) + P(11-8)G(7-4) + P(11-8)P(7-4)G(3-0) + P(11-8)P(7-4)P(3-0)C_0 \tag{5.12-1c}$$

Estas saídas de vai-um substituem C_4, C_8 e C_{12} da Fig. 5.11-1. A unidade VUA gera saídas (não usadas aqui) dadas por

$$P(15-0) = P(15-12)P(4-8)P(7-4)P(3-0) \tag{5.12-2a}$$

e

$$G(15-0) = G(15-12) + P(15-12)G(11-8) + P(15-12)P(11-8)G(7-4) + P(15-12)P(11-8)P(7-4)G(3-0) \tag{5.12-2b}$$

Observamos que as Eqs. (5.12-1*a*) a (5.12-1*c*) são idênticas na forma às Eqs. (5.10-7*a*) a (5.10-7*c*), enquanto as Eqs. (5.12-2*a*) e (5.12-2*b*) são idênticas na forma às Eqs. (5.10-8) e (5.10-9). A estrutura de portas da unidade de VUA da Fig. 5.12-1 é, pois, idêntica à parte da Fig. 5.10-2 que gera o vai-um, com exceção das portas G2 e G3 que não são usadas e das conexões tracejadas que não são feitas.

Supondo, novamente, que em $t = 0$ todas as entradas A_i e B_i e o vai-um de entrada C_0 são aplicados, perguntamos, como antes, quanto tempo devemos esperar até que a soma correta esteja disponível na saída? Da Fig. 5.10.2 vemos que as Ps e Gs de grupo estão disponíveis em $t = 4\,t_{ap}$. Na Fig. 5.12-1 estas Ps e Gs de grupo devem se propagar através de dois níveis adicionais na unidade de VUA para produzir os vai-uns C_4^*, C_8^* e C_{12}^*; portanto estes vai-uns estão disponíveis em $t = 6\,t_{ap}$. Finalmente, os bits soma estão disponíveis após outro intervalo de $2t_{ap}$; logo, o tempo de espera total é de $8\,t_{ap}$. Este intervalo deve ser comparado com $32\,t_{ap}$ de um somador que não usa vai-um antecipado.

Vimos, primeiramente, como o vai-um antecipado é implementado com quatro bits. A seguir vimos como o princípio é aplicado a dezesseis bits ou quatro grupos de quatro bits. De modo similar, a implementação pode ser estendida a 64 bits, ou seja, quatro grupos de dezesseis bits. Cada grupo de dezesseis bits é tratado inicialmente como na Fig. 5.12-1. A seguir adicionamos mais uma unidade de VUA idêntica à da Fig. 5.12-1. Esta nova unidade de VUA aceita como entradas o vai-um de entrada C_0 e as saídas P e G de grupo das unidades de VUA de primeiro nível, fornecendo os vai-uns C_{16}^*, C_{32}^* e C_{48}^*. Os detalhes da ligação são mostrados na Fig. 5.12-2. Deixamos como exercício a verificação de que o tempo necessário para completar a soma é $14\,t_{ap}$. Este intervalo deve ser comparado com $128\,t_{ap}$, que seria o tempo necessário para um somador de 64 bits que não usasse vai-um antecipado.

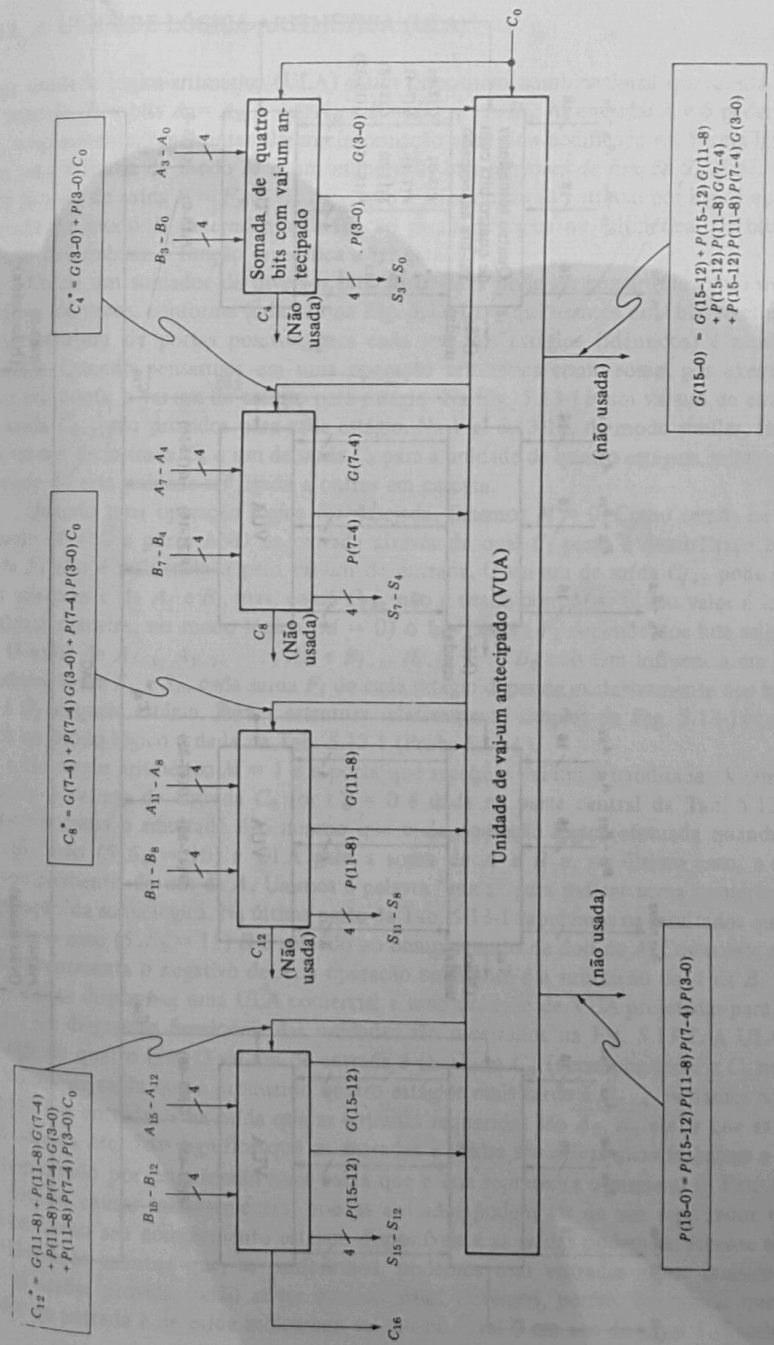


Fig. 5.12-1 Uma unidade de vai-um antecipado (VUA) adicionada à cascata da Fig. 5.11-1.

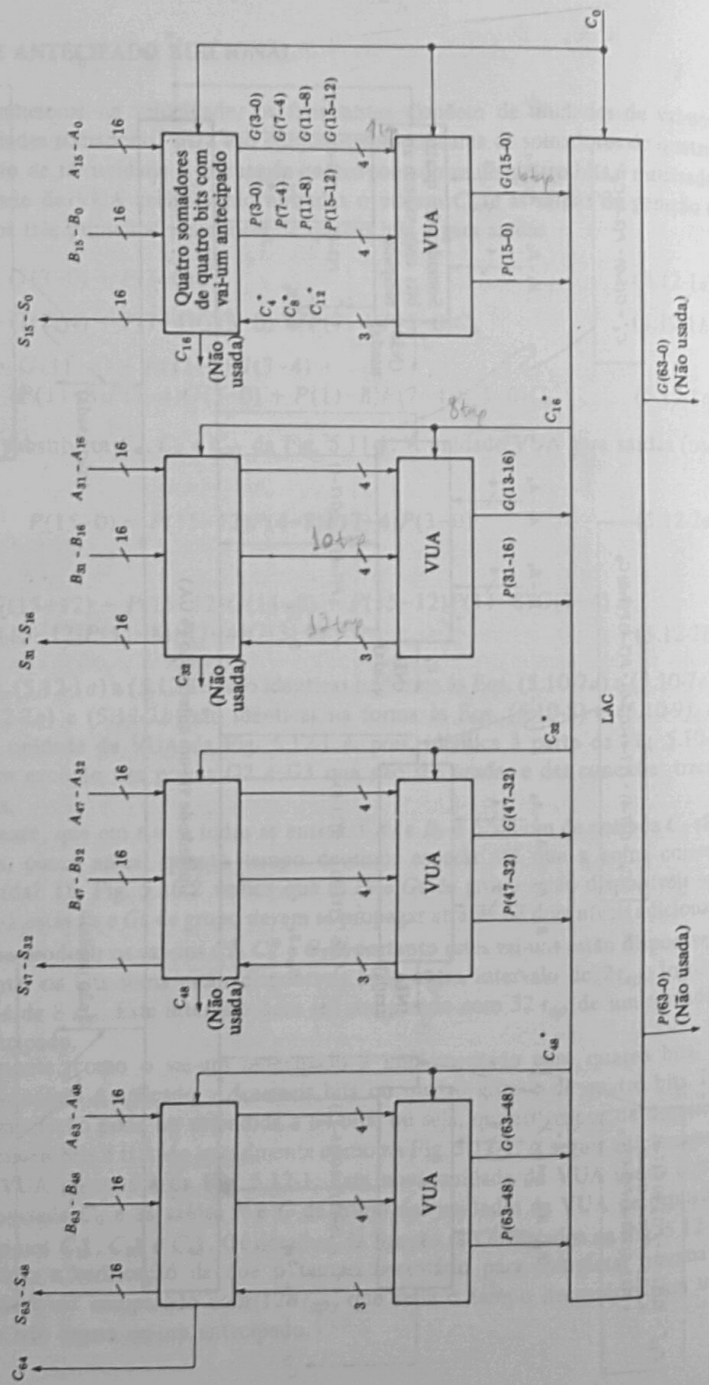


Fig. 5.12-2 Um total de cinco unidades de VUA são usadas em um somador de 64 bits.

5.13 A UNIDADE LÓGICA-ARITMÉTICA (ULA)

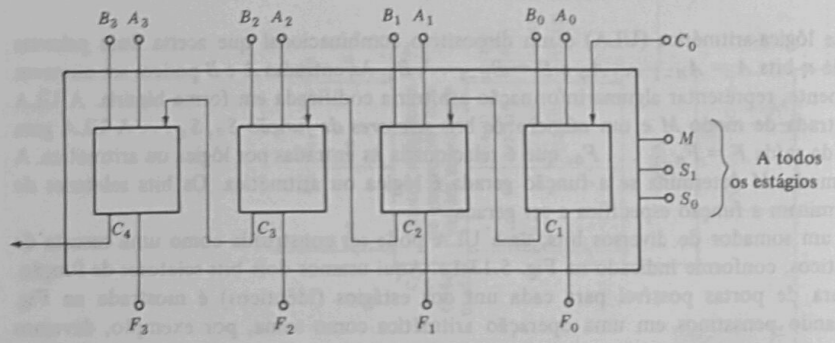
Uma unidade lógica-aritmética (ULA) é um dispositivo combinacional que aceita duas palavras de entrada de n bits $A = A_{n-1} \ldots A_0$ e $B = B_{n-1} \ldots B_0$. As entradas A e B podem ser números ou, simplesmente, representar alguma informação arbitrária codificada em forma binária. A ULA tem uma entrada de modo M e um número de bits *seletores de função* $S_0, S_1 \ldots$ A ULA gera uma *função* de saída $F = F_{n-1} \ldots F_0$, que é relacionada às entradas por lógica ou aritmética. A entrada de modo M determina se a função gerada é lógica ou aritmética. Os bits seletores de função determinam a função específica a ser gerada.

Como um somador de diversos bits, uma ULA pode ser construída como uma cascata de estágios idênticos, conforme indicado na Fig. 5.13-1*a*. Aqui usamos dois bits seletores de função. Uma estrutura de portas possível para cada um dos estágios (idênticos) é mostrada na Fig. 5.13-1*b*. Quando pensarmos em uma operação aritmética como soma, por exemplo, devemos levar em conta o vai-um de estágio para estágio. Na Fig. 5.13-1*b* um vai-um de entrada C_i e um de saída C_{i+1} são providos para cada estágio. Na Fig. 5.13-1*a*, de modo similar, foram providos um vai-um de entrada C_0 e um de saída C_4 para a unidade de quatro estágios, antecipando a possibilidade de esta unidade ser ligada a outras em cascata.

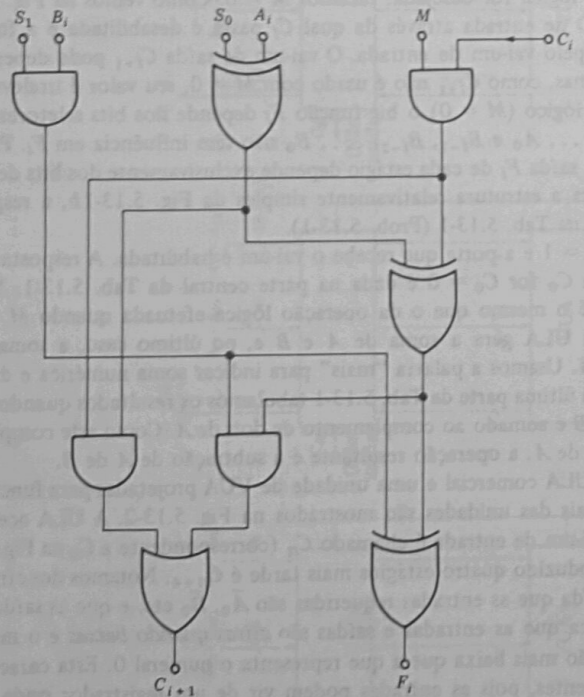
Quando uma operação lógica for desejada, fazemos $M = 0$. Como vemos na Fig. 5.13-1*b*, quando $M = 0$ a porta AND de entrada através da qual C_i passa é desabilitada e a função de saída F_i não é influenciada pelo vai-um de entrada. O vai-um de saída C_{i+1} pode depender dos bits seletores e de A_i e B_i, mas, como C_{i+1} não é usado com $M = 0$, seu valor é irrelevante. De qualquer maneira, no modo lógico ($M = 0$) o bit função F_i depende dos bits seletores de A_i e B_i. O valor de $A_{i-1}, A_{i-2}, \ldots, A_0$ e $B_{i-1}, B_{i-2}, \ldots, B_0$ não têm influência em F_i. Para cada combinação de S_1 e S_0, cada saída F_i de cada estágio depende exclusivamente dos bits de entrada A_i e B_i daquele estágio. Para a estrutura relativamente simples da Fig. 5.13-1*b*, a resposta da ULA no modo lógico é dada na Tab. 5.13-1 (Prob. 5.13-1).

No modo aritmético $M = 1$ e a porta que recebe o vai-um é habilitada. A resposta da ULA quando o vai-um de entrada C_0 for $C_0 = 0$ é dada na parte central da Tab. 5.13-1. Nos dois primeiros casos o resultado é o mesmo que o da operação lógica efetuada quando $M = 0$. No terceiro caso ($S_1S_0 = 10$) a ULA gera a soma de A e B e, no último caso, a soma de B e do complemento de um de A. Usamos a palavra "mais" para indicar soma numérica e distinguir a operação da soma lógica. Na última parte da Tab. 5.13-1 tabulamos os resultados quando $C_0 = 1$. No último caso ($S_1S_0 = 11$) B é somado ao complemento de dois de A. Como este complemento de dois representa o negativo de A, a operação resultante é a subtração de A de B.

Existe disponível uma ULA comercial e uma unidade de VUA projetadas para funcionarem juntas. Os diagramas funcionais das unidades são mostrados na Fig. 5.13-2. A ULA aceita duas entradas de quatro bits. O vai-um de entrada é chamado C_n (correspondente a C_0 na Fig. 5.13-1) e o vai-um de saída que é produzido quatro estágios mais tarde é C_{n+4}. Notamos dos círculos de inversão nas entradas e na saída que as entradas requeridas são $\overline{A}_0$, $\overline{B}_0$ etc. e que as saídas fornecidas são $\overline{F}_0$ etc. Isto significa que as entradas e saídas são *ativas quando baixas* e o numeral 1 é representado por uma tensão mais baixa que a que representa o numeral 0. Esta característica não precisa causar inconvenientes, pois as entradas podem vir de um registrador onde tanto a variável como seu complemento estejam disponíveis e as saídas podem facilmente ser invertidas. (O fabricante acentua que, se preferirmos, podemos usar entradas *ativas quando altas*. Neste caso as saídas providas serão ativas quando altas. Devemos, porém, considerar que as linhas de vai-um de entrada e de saída indicariam vai-um no nível 0 em vez do nível 1. Também a função



(*a*)



(*b*)

Fig. 5.13-1 (*a*) Uma ULA construída como uma cascata de estágios idênticos. (*b*) Uma estrutura de portas possível para um estágio.

Tabela 5.13-1 As funções geradas pela estrutura lógica da Fig. 5.13-1

Função lógica com $M = 0$			
S_1	S_0	Função lógica F_i	Comentário
0	0	$F_i = A_i$	Entrada A transferida para a saída
0	1	$F_i = \overline{A_i}$	Entrada A complementada e transferida
1	0	$F_i = A_i \oplus B_i$	EXCLUSIVE-OR
1	1	$F_i = \overline{A_i \oplus B_i}$	EXCLUSIVE-NOR

Funções aritméticas com $M = 1$ e $C_0 = 0$			
S_1	S_0	Função aritmética F	Comentário
0	0	$F = A$	Entrada A transferida para a saída
0	1	$F = \overline{A}$	Complemento de um de A
1	0	$F = A$ mais B	Soma de A e B
1	1	$F = \overline{A}$ mais B	Soma de B e complemento de um de A

Funções aritméticas com $M = 1$ e $C_0 = 1$			
0	0	$F = A$ mais 1	Incrementa A
0	1	$F = \overline{A}$ mais 1	Complemento de dois de A
1	0	$F = A$ mais B mais 1	Incrementa soma de A e B
1	1	$F = \overline{A}$ mais B mais 1	B menos A

lógica ou aritmética executada para um conjunto fixo de bits seletores seria diferente no caso de entradas ativas quando altas e quando baixas.)

A função gerada pela ULA depende do controle de modo M e dos bits seletores S_3, S_2, S_1 e S_0. Estas funções estão listadas na Tab. 5.13-2. As funções lógicas são geradas quando $M = 1$ e, como há quatro bits seletores, existem dezesseis funções disponíveis. Devemos lembrar que (ver Seção 1.5) existem precisamente dezesseis funções de duas variáveis lógicas e, como podemos verificar da tabulação, cada uma delas aparece aqui.

As funções aritméticas são geradas quando o vai-um da ULA é habilitado fazendo $M = 0$. Somos levados a interpretar a saída como um número quando uma operação aritmética é executada, mas é de interesse notar que alguns dos resultados das operações aritméticas são precisamente os mesmos das operações lógicas. Todavia, quando os resultados são interpretados como números, a presença do vai-um de entrada ($C_n = 1$) aumenta a magnitude do número de uma unidade.

Uma palavra sobre a operação aritmética quando $S_3S_2S_1S_0 = 0110$ com $C_n = 1$. Notamos que $F = A$ menos B. Esta subtração é obtida somando A ao complemento de um de B (o

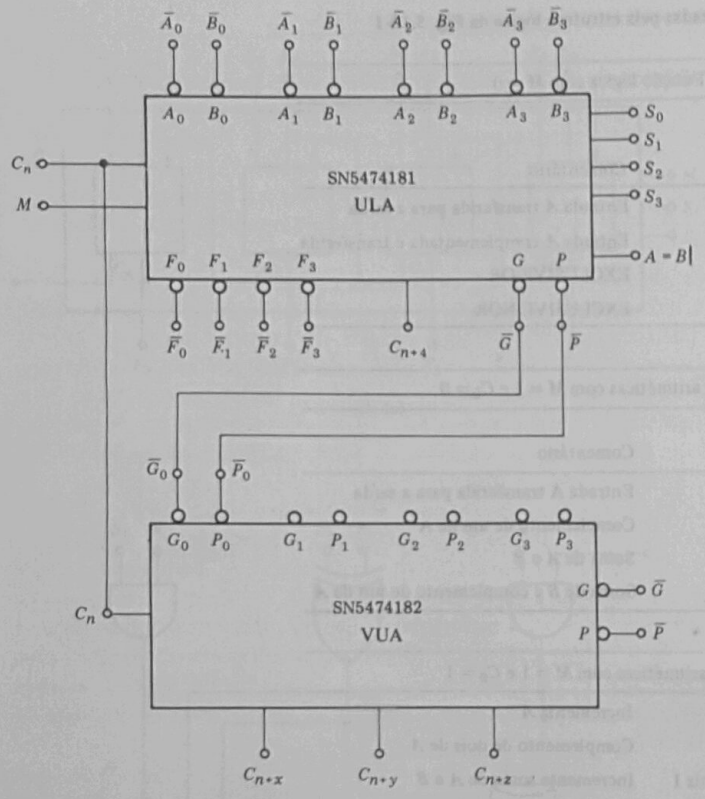


Fig. 5.13-2 A configuração de terminais da ULA e da unidade de VUA integradas disponíveis comercialmente. As interligações estão indicadas.

número B com todos seus bits complementados). Se desejarmos que o resultado seja válido na representação por complemento de um, devemos usar o vai-um deslocado do final como na Fig. 5.7-1 ou, se desejarmos a representação por complemento de dois, devemos fazer $C_n\ (= C_0) = = 1$, conforme especificado na tabulação.

Quando preparado desta maneira para subtração com $S_3S_2S_1S_0 = 0110$ e $C_n = 1$, se $A = B$, o resultado é $F_3F_2F_1F_0 = 0000$ e a saída da ULA é $\bar{F}_3\bar{F}_2\bar{F}_1\bar{F}_0 = 1111$. Na ULA, uma porta AND existe cujas entradas são $\bar{F}_3$, $\bar{F}_2$, $\bar{F}_1$ e $\bar{F}_0$ e cuja saída é o terminal externo marcado "$A = B$". A ULA pode, pois, ser usada como comparador digital que registra a equivalência $A = B$ fazendo o terminal "$A = B$" igual a 1. Neste mesmo modo subtrativo de operação, a ULA pode ser usada para indicar as magnitudes relativas dos números A e B. Para as entradas e saídas ativas quando baixas especificadas na Fig. 5.13-2 (como pode ser verificado, Prob. 5.13-2), as regras para determinação da magnitude relativa são as seguintes:

Tabela 5.13-2 As funções lógicas e aritméticas geradas pela ULA SN5474181

Entrada de seleção				$M = 1$	$M = 0$ Operações aritméticas	
S_3	S_2	S_1	S_0	Funções lógicas	$C_n = 0$	$C_n = 1$
0	0	0	0	$F_i = \bar{A}_i$	$F = A$ menos 1	$F = A$
0	0	0	1	$F_i = \overline{A_iB_i}$	$F = AB$ menos 1	$F = AB$
0	0	1	0	$F_i = \bar{A}_i + B_i$	$F = A\bar{B}$ menos 1	$F = A\bar{B}$
0	0	1	1	$F_i = 1$	$F =$ menos 1*	$F =$ zero
0	1	0	0	$F_i = \overline{A_i + B_i}$	$F = A$ mais $(A + \bar{B})$	$F = A$ mais $(A + \bar{B})$ mais 1
0	1	0	1	$F_i = \bar{B}_i$	$F = AB$ mais $(A + \bar{B})$	$F = AB$ mais $(A + \bar{B})$ mais 1
0	1	1	0	$F_i = \overline{A_i \oplus B_i}$	$F = A$ menos B menos 1	$F = A$ menos B
0	1	1	1	$F_i = A_i + \bar{B}_i$	$F = A + \bar{B}$	$F = (A + \bar{B})$ mais 1
1	0	0	0	$F_i = \bar{A}_iB_i$	$F = A$ mais $(A + B)$	$F = A$ mais $(A + B)$ mais 1
1	0	0	1	$F_i = A_i \oplus B_i$	$F = A$ mais B	$F = A$ mais B mais 1
1	0	1	0	$F_i = B_i$	$F = A\bar{B}$ mais $(A + B)$	$F = A\bar{B}$ mais $(A + B)$ mais 1
1	0	1	1	$F_i = A_i + B_i$	$F = A + B$	$F = (A + B)$ mais 1
1	1	0	0	$F_i = 0$	$F = A + A$**	$F = A$ mais A mais 1
1	1	0	1	$F_i = A_i\bar{B}_i$	$F = AB$ mais A	$F = AB$ mais A mais 1
1	1	1	0	$F_i = A_i\bar{B}_i$	$F = A\bar{B}$ mais A	$F = A\bar{B}$ mais A mais 1
1	1	1	1	$F_i = A_i$	$F = A$	$F = A$ mais 1

* Na representação por complemento de dois.

** Cada bit é deslocado para a posição mais significativa seguinte.

C_n	C_{n+4}	Indicação
0	0	$A < B$
0	1	$A > B$
1	0	$A < B$
1	1	$A > B$

O circuito integrado da ULA tem incorporada a característica de VUA já descrita. A estrutura lógica da ULA é descrita em detalhe na Fig. B-1 do Apêndice B. Lá observamos que, como na Fig. 5.13-2, a unidade fornece saídas de vai-um de grupo $\bar{G}$ e propagação de grupo $\bar{P}$, de maneira que quatro ULAs podem usar um gerador de VUA para estender a característica de VUA para dezesseis bits. A estrutura lógica da unidade de VUA é mostrada na Fig. B-2 do Apêndice B. Os detalhes das ligações necessárias para esta finalidade são mostrados na Fig. 5.13-3. Aqui, para simplificar, incluímos apenas os terminais necessários para mostrar as interligações. Finalmente notamos que a unidade de VUA possui terminais de saída de propagação e geração de grupo (não usados aqui) que permitem a extensão da unidade além de dezesseis bits.

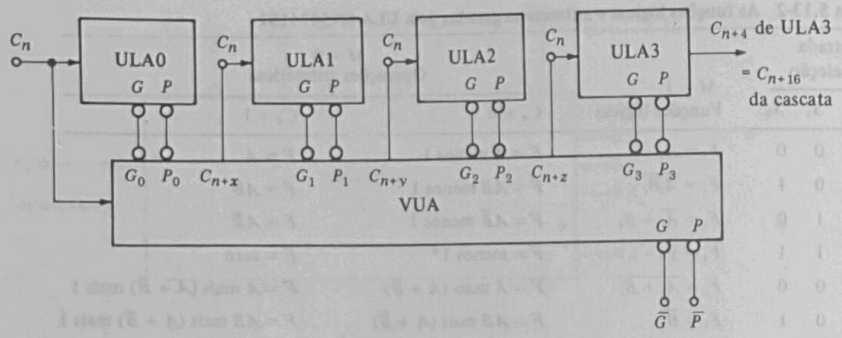


Fig. 5.13-3 As interligações entre a ULA e a unidade de VUA necessárias para acomodar dezesseis bits.

5.14 SOMA BCD

No sistema BCD (decimal codificado em binário) de representação de números, os dígitos binários são agrupados em nibbles de quatro bits, cada nibble representando um dígito decimal. O código BCD tem a desvantagem de usar somente dez das dezesseis combinações de quatro bits. A representação binária dos números decimais de 10 a 15 é excluída. Por outro lado, a representação BCD tem uma vantagem que, às vezes, é muito importante. Consideremos, por exemplo, o problema de converter um número binário normal em seu equivalente decimal e vice-versa. Para converter de binário para decimal, devemos executar uma computação que na determinação de *cada* dígito decimal deve levar em conta *cada* dígito binário. Na conversão inversa, *cada* dígito binário geralmente depende de *cada* dígito decimal. Como resultado, os circuitos necessários para efetuar a conversão crescem exponencialmente, com o número de dígitos. No caso de BCD, cada dígito decimal é associado com apenas quatro bits. O circuito para conversão de $2N$ dígitos é somente duas vezes maior que para conversão de N dígitos.

Suponhamos agora que exista um processador digital de algum tipo ao qual apresentamos dados numéricos em forma decimal e que após processamento (algum tipo de computação, por exemplo) os resultados devem ser fornecidos em forma decimal. Se o processamento fosse extenso e muito mais complicado que as conversões decimal-binário e binário-decimal, seria vantajoso executar as conversões, mas, se o processamento fosse simples, talvez fosse mais conveniente evitar as conversões e executar o processamento com os dados na forma BCD. Estas considerações nos levam ao assunto da soma de números em código BCD.

```
3 = 0 0 1 1          7 = 0 1 1 1          9 =   1 0 0 1
4 = 0 1 0 0          5 = 0 1 0 1          8 =   1 0 0 0
7 = 0 1 1 1         12 = 1 1 0 0 (não BCD) 17 = (1) 0 0 0 1 (não BCD)
                     6 = 0 1 1 0          6 =   0 1 1 0
                   (1) 0 0 1 0 (=12 BCD)    (1) 0 1 1 1 (=17 BCD)
```

(a) (b) (c)

Fig. 5.14-1 Um processo para somar números BCD.

Na Fig. 5.14-1*a* somamos os números 3 + 4 = 7 e a soma correspondente dos equivalentes BCD é direta. Na Fig. 5.14-1*b* somamos 7 + 5 (selecionando para obter uma soma maior que 9). O resultado 1100 é correto como representação binária, mas inaceitável como uma representação BCD. Em BCD o resultado deve ser 12 = 00010010. A dificuldade surge, obviamente, porque os seis números de 10 a 15 são excluídos do BCD. Como veremos, a situação é facilmente corrigida adicionando 6 = 0110. O resultado muda para a representação do decimal 2 e, conforme desejado, o vai-um é gerado (indicado pelo 1 dentro do círculo). Na Fig. 5.14-1*c* somamos 9 + 8 e encontramos o resultado 0001 = 1. Mas a geração do vai-um assinalado nos mostra que algo está errado. Novamente o problema é corrigido pela soma de 0110 = 6. Resumindo, o processo de soma de números codificados em BCD principia com a soma como se as parcelas fossem números binários diretos. Se o resultado não for um número BCD, isto é, não for de 0 a 9, ou um vai-um for gerado, então 0110 = 6 é somado ao número.

O circuito para efetuar a soma de um dígito decimal em código BCD, incluindo previsão para um vai-um da posição imediatamente menos significativa, é mostrado na Fig. 5.14-2. A linha superior de somadores inteiros soma os números $A_1B_1C_1D_1$ e $A_2B_2C_2D_2$. Uma ou outra das portas AND terá saída 1 quando a soma gerada estiver no intervalo de 10 a 15. A saída

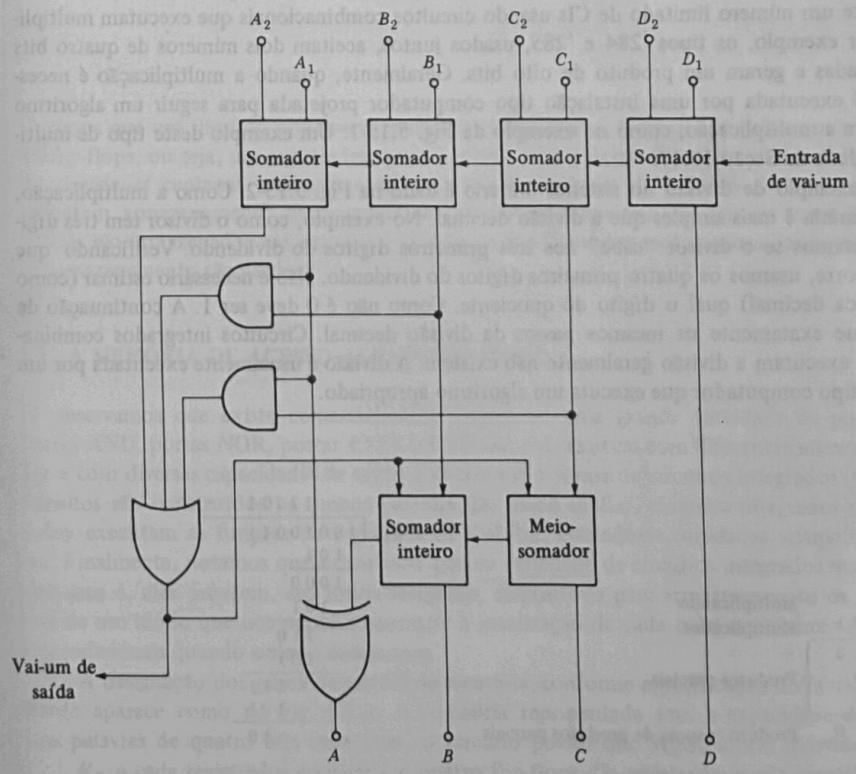


Fig. 5.14-2 Um estágio de lógica para somar números BCD.

da porta OR será 1 quando a soma estiver no intervalo de 10 a 15 ou quando um vai-um for gerado. Sempre que a saída da porta OR for 1, o número 6 = 0110 será somado, como necessário (ver Prob. 5.14-1).

5.15 MULTIPLICAÇÃO E DIVISÃO

Multiplicação e divisão no sistema binário são efetuadas da mesma maneira que no sistema decimal. Um exemplo de multiplicação binária é mostrado na Fig. 5.15-1. O *multiplicando* é multiplicado por cada um dos dígitos individuais do *multiplicador*, cada uma destas multiplicações formando um produto *parcial*. Se o bit do multiplicador for 0, o produto parcial será 0. Se o bit do multiplicador for 1, o produto parcial é o próprio multiplicando. A "tabuada de multiplicação" que deve ser decorada é, pois, bastante simples, não havendo geração de vai-um na multiplicação por um bit do multiplicador. Começamos com o bit menos significativo e, como mostrado na Fig. 5.15-1, os produtos parciais sucessivos são deslocados uma posição para a esquerda. O produto é a soma dos produtos parciais. No caso geral, o produto pode ter um número de bits que é um a mais que a soma do número de bits do multiplicando e do multiplicador.

Existe um número limitado de CIs usando circuitos combinacionais que executam multiplicações. Por exemplo, os tipos '284 e '285, usados juntos, aceitam dois números de quatro bits como entradas e geram um produto de oito bits. Geralmente, quando a multiplicação é necessária, ela é executada por uma instalação tipo computador projetada para seguir um algoritmo que executa a multiplicação, como no exemplo da Fig. 5.15-1. Um exemplo deste tipo de multiplicação é dado na Seção 10.17.

Um exemplo de divisão no sistema binário é dado na Fig. 5.15-2. Como a multiplicação, a divisão binária é mais simples que a divisão decimal. No exemplo, como o divisor tem três dígitos, perguntamos se o divisor "cabe" nos três primeiros dígitos do dividendo. Verificando que isto não ocorre, usamos os quatro primeiros dígitos do dividendo. Não é necessário estimar (como na aritmética decimal) qual o dígito do quociente. Como não é 0 deve ser 1. A continuação da divisão segue exatamente os mesmos passos da divisão decimal. Circuitos integrados combinacionais que executam a divisão geralmente não existem. A divisão é usualmente executada por um dispositivo tipo computador que executa um algoritmo apropriado.

```
      1 1 1 0   Multiplicando
        1 0 1   Multiplicador
      -------
      1 1 1 0 }
    0 0 0 0   } Produtos parciais
  1 1 1 0     }
  -----------
  1 0 0 0 1 1 0   Produto = soma de produtos parciais
```

Fig. 5.15-1 Um exemplo de multiplicação em aritmética binária.

```
          11101 ···
101 | 10010011
      101
      ----
      1000
       101
      ----
        110
        101
        ---
         111
         101
         ---
          10
```

Fig. 5.15-2 Um exemplo de divisão em aritmética binária.

CAPÍTULO 6
MEMÓRIA

Já vimos que um flip-flop pode servir para armazenar, isto é, lembrar, um bit e que um conjunto de flip-flops, ou seja, um registrador, pode lembrar uma palavra. Freqüentemente, há necessidade de armazenar centenas, milhares e até dezenas de milhares de palavras. Os componentes que executam armazenamento nesta escala são chamados *memórias*. Em alguns casos os bits individuais são armazenados nas memórias através de flip-flops. Em outros casos, outros mecanismos de armazenamento são usados.

6.1 A MEMÓRIA DE ACESSO ALEATÓRIO (RAM)

Já observamos que existe comercialmente disponível uma grande variedade de portas lógicas. Portas AND, portas NOR, portas EXCLUSIVE-OR etc. existem com diferentes números de entradas e com diversas capacidades de saída. Existem sob a forma de circuitos integrados (CI) e muitos circuitos são construídos na mesma pastilha. De modo similar, circuitos integrados mais complicados executam as funções de registradores multibit, contadores, unidades aritméticas multibit etc. Finalmente, notamos que existe uma grande variedade de circuitos integrados que são *memórias*, isto é, eles provêem, em forma integrada, dispositivos para armazenamento de muitas palavras de um modo que nos permite conhecer a localização de cada palavra e chamar de volta palavras individuais quando assim o desejarmos.

A designação dos pinos de um CI de memória, conforme especificação do fabricante, tipicamente aparece como na Fig. 6.1-1. A memória representada tem a capacidade de armazenar oito palavras de quatro bits cada uma. O circuito possui oito registradores individuais R_0, R_1, ..., R_7 e cada registrador consiste em quatro flip-flops. Os registradores são identificados pelos níveis lógicos nos terminais A_2, A_1 e A_0. Estes bits A_2, A_1 e A_0 são chamados bits de *endereço* e o conjunto de bits de endereço é conhecido como o *endereço*. Quando $A_2A_1A_0 = 000$, pode-

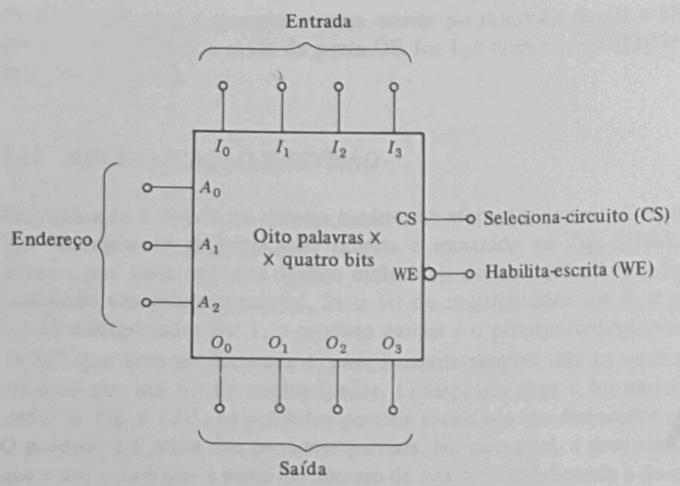


Fig. 6.1-1 A designação dos pinos de um CI de memória com oito palavras de quatro bits.

mos acessar o registrador R_0; quando $A_2A_1A_0 = 001$, podemos acessar o registrador R_1, e assim por diante. Como existem três bits de endereço, podemos endereçar $2^3 = 8$ registradores. Cada registrador tem, pois, um *endereço* e um registrador qualquer é acessado ao colocarmos os bits de endereço daquele registrador nos terminais de endereço.

Os quatro bits que devem ser armazenados, isto é, *escritos*, no registrador endereçado são apresentados ao circuito de memória através dos terminais de entrada I_3, I_2, I_1 e I_0. Para escrever uma palavra em um registrador, o registrador deve ser endereçado, a palavra deve ser apresentada à entrada e *ambas* as entradas *habilita-escrita* (*write-enable*) ($\overline{\text{WE}}$) e *seleciona-circuito* (CS) devem ser habilitadas, isto é, tornadas ativas. Com base na convenção geralmente aceita (já discutida no Cap. 3), o diagrama da Fig. 6.1-1 indica que o terminal externo $\overline{\text{WE}}$ é ativo quando baixo (ativo quando $\overline{\text{WE}} = 0$ em um sistema de lógica positiva) e CS é ativo quando alto.

O processo de *leitura* da memória traz aos terminais de saída O_3, O_2, O_1 e O_0 a palavra armazenada no registrador endereçado. Para ler a memória, apresentamos o endereço, habilitamos a entrada CS e colocamos a entrada $\overline{\text{WE}}$ em $\overline{\text{WE}} = 1$. Assim $\overline{\text{WE}} = 0$ *escreve* na memória enquanto $\overline{\text{WE}} = 1$ *lê* a memória. Por esta razão alguns fabricantes denominam os terminais de habilita-escrita de leitura/escrita ($R/\overline{W}$).

Tanto para leitura como para escrita, a entrada seleciona-circuito (CS) deve estar habilitada, isto é, ativa. Se CS não estiver ativa, o circuito estará isolado do mundo externo. Não será possível escrever na memória ou lê-la.

O número total de bits na memória da Fig. 6.1-1 é de $8 \times 4 = 32$ bits. O modo como os bits são arranjados em palavras é descrito pela *organização* da memória. A memória da Fig. 6.1-1 é descrita como sendo uma memória de oito palavras com quatro bits/palavra.

Como podemos *ler* palavras da memória e podemos *escrever* palavras nela, a memória descrita é, com razão, chamada memória de *leitura-escrita*. Além disso, podemos acessar qualquer registrador desejado aleatoriamente para ler ou escrever uma palavra. Por esta razão a memória

também é conhecida como uma *memória de acesso aleatório* (MAA) ou RAM (*random-access memory*).

6.2 ESTRUTURA DE UMA RAM SEMICONDUTORA

Em princípio, a estrutura lógica de uma memória semicondutora do tipo que discutimos é mostrada na Fig. 6.2-1. Aqui mostramos quatro palavras de dois bits cada uma, isto é, uma organização de 4×2. Nosso interesse está na estrutura lógica da memória e não nos detalhes de sua eletrônica. (A eletrônica é tratada em outros textos. Ver nota ao pé da página 76.) Assim, tomamos a liberdade de incorporar algumas simplificações em comparação com uma memória real. O arranjo usa a chave controlada por lógica introduzida na Seção 3.7. Quando o nível lógico da linha de controle (linha tracejada passando através da chave) for 1, a chave estará fechada e a conexão será feita; quando o nível lógico da linha de controle for 0, a chave estará aberta.

Os bits são armazenados nos flip-flops mais elementares, constituídos por inversores com acoplamento cruzado (ver Seção 4.1). Os bits de endereço A_1 e A_0 são aplicados a um decodificador. Quando, por exemplo, tivermos $A_1 = 1$ e $A_0 = 0$, a saída da porta G_2 estará no nível 1, enquanto as saídas das demais portas AND do decodificador estarão no nível 0. Assim, somente os flip-flops do registrador para a palavra 2 serão acessados, uma vez que somente as chaves destes flip-flops estarão fechadas para fazer contato com as linhas de bits. Assim, o endereço $A_1A_0 = 10$ endereça a palavra 2 e somente a palavra 2.

Se *seleciona-circuito* (CS) estiver no nível 0, as saídas de ambas as portas G_0 e G_1 estarão no nível lógico 0 e as linhas de bits não estarão ligadas nem à entrada de dados nem aos terminais de saída. Se CS = 1 e WE = 1, os flip-flops estarão ligados aos terminais de saída de modo a permitir a leitura da palavra armazenada.

Circuitos integrados de memória comerciais disponíveis no presente incorporam até 65.536 (2^{16}) bits. É devido à habilidade dos projetistas e fabricantes que tais circuitos estão disponíveis a preços que permitem seu uso até por amadores. Devido ao grande número de componentes ativos que devem ser incorporados a circuitos de memória tão grandes, poderíamos imaginar se não seria vantajoso remover a parte do decodificador, tornando-o disponível como um circuito separado. Existe uma razão prática para que isto não ocorra. Consideremos, como exemplo, uma memória com $2^8 = 256$ palavras. Se o decodificador estiver incluído no CI, somente oito pinos de conexão são necessários na periferia do circuito como entradas de endereço. Se o decodificador fosse externo, seriam necessários 256 pinos de conexão. Uma das limitações envolvidas na fabricação é o espaço disponível na periferia da pastilha para ligação dos pinos.

6.3 LIGAÇÃO DE CIRCUITOS DE MEMÓRIA EM PARALELO

Freqüentemente, ocorre que o número de palavras disponível em um CI não é adequado ou o número de bits por palavra não é adequado ou ambos. A solução é ligar memórias em paralelo.

A ligação de CIs de memória em paralelo para aumentar o número de bits por palavra (mas não para aumentar o número de palavras) é ilustrada na Fig. 6.3-1. Aqui ligamos em paralelo dois CIs de oito palavras e quatro bits/palavra para construir uma memória em que o número de palavras permanece em oito mas o número de bits aumenta de quatro para oito. O endereço de entrada de três bits é aplicado aos pinos de endereçamento de ambas as memórias. Os terminais

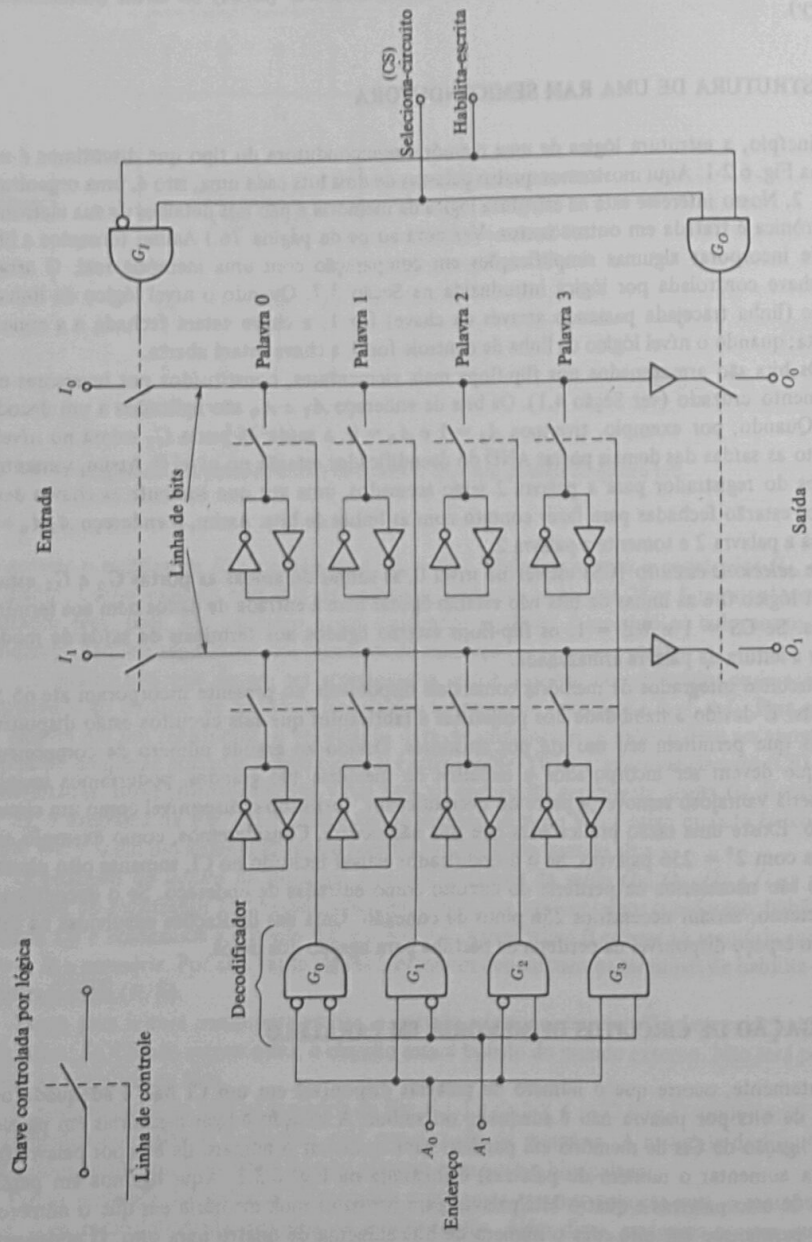


Fig. 6.2-1 Uma estrutura para uma RAM de escrita-leitura com quatro palavras e dois bits/palavra.

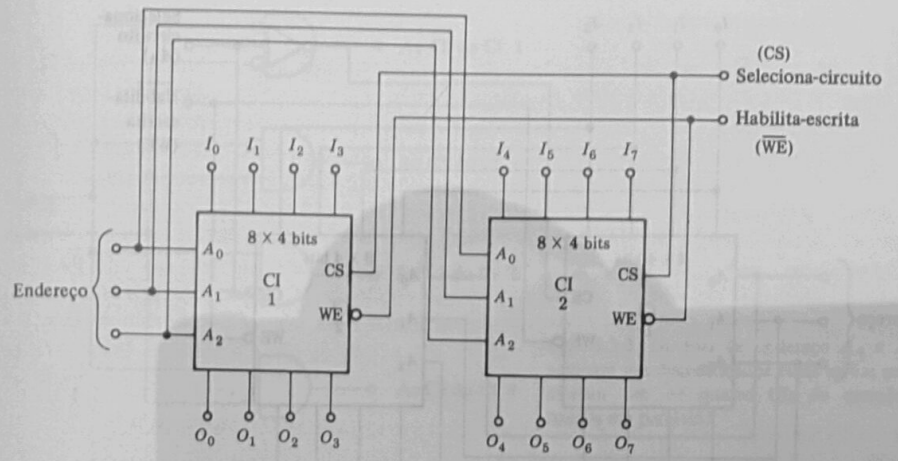


Fig. 6.3-1 Dois CIs de oito palavras e quatro bits/palavra ligados em paralelo para formar uma memória de oito palavras e oito bits/palavra.

CS dos dois CIs são interligados, bem como os terminais $\overline{WE}$. *Seleciona-circuito* e *habilita-escrita* selecionam e habilitam os CIs simultaneamente. O CI1 aceita e armazena quatro bits (0, 1, 2 e 3) e o CI2 aceita e armazena mais quatro bits (4, 5, 6 e 7). O paralelismo pode ser estendido a circuitos adicionais. Três CIs de oito palavras e quatro bits/palavra produzem uma memória de oito palavras e doze bits/palavra, e assim por diante. Conforme os CIs são adicionados, os endereços são aplicados simultaneamente aos terminais de entrada de endereço de todos os CIs. De modo similar, todos os pinos CS são ligados juntos para formar uma entrada CS única; as entradas $\overline{WE}$ são tratadas da mesma maneira.

Para possibilitar máxima flexibilidade em matéria de bits por palavra, os fabricantes oferecem circuitos de memória com um só bit por palavra. Assim, encontramos nos catálogos dos fabricantes memórias com organizações de 256 × 1, 1024 × 1, 4096 × 1 etc. Após selecionar um circuito com o número adequado de palavras, construímos uma memória com n bits/palavra simplesmente através da ligação de n circuitos em paralelo.

Na Fig. 6.3-2 mostramos como dois CIs de oito palavras e quatro bits/palavra podem ser ligados em paralelo para obter uma memória de dezesseis palavras e quatro bits/palavra. Como antes, os três bits de endereço são aplicados a ambos os circuitos, mas, em vez de termos um bit CS comum aplicado a ambos os circuitos, quando a entrada CS de um circuito é ativada a entrada CS do outro circuito é desativada. A entrada CS do sistema de memória (aquela na entrada do inversor) é um *bit adicional de endereço*, que chamamos A_4. Quando $A_4 = 1$, o CI endereçado é o CI2, e, quando $A_4 = 0$, o CI endereçado é o CI1. O bit de endereço de seleção de circuito A_4 seleciona um ou outro dos CIs; os bits de endereço A_0, A_1 e A_2 selecionam uma localização particular no circuito selecionado. Os bits de dados de entrada são aplicados em comum aos dois circuitos, bem como à entrada $\overline{WE}$.

Uma ocorrência especial existe aqui que não aparece quando os circuitos são ligados em paralelo para aumentar somente o número de bits por palavra, como na Fig. 6.2-1. Aqui, a palavra de

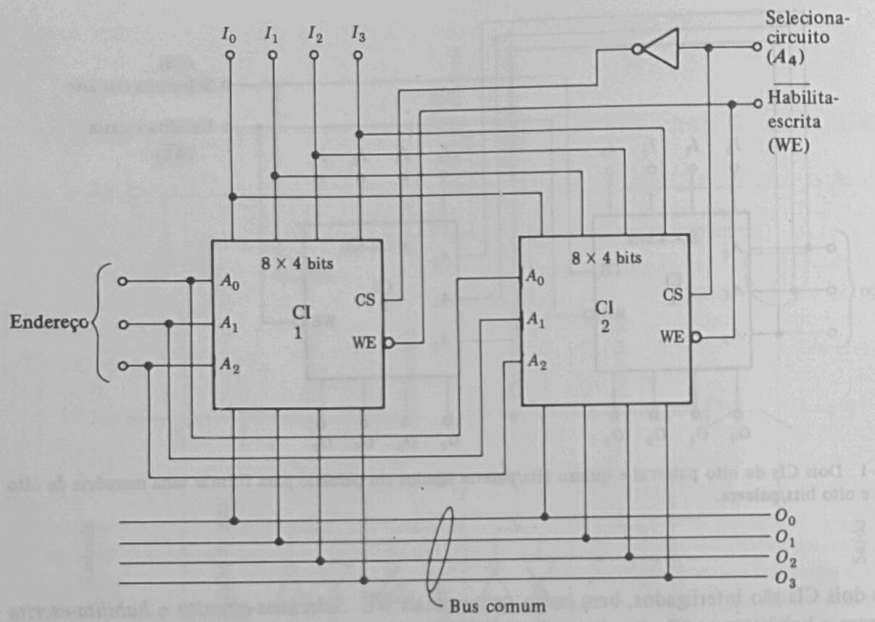


Fig. 6.3-2 Ligação de circuitos de memória em paralelo para aumentar o número de palavras mantendo fixo o número de bits por palavra.

saída é lida algumas vezes de um circuito e outras vezes do outro circuito. Presumivelmente, a palavra deve ser transmitida ao mesmo destino, independentemente de qual circuito se originou. As saídas dos circuitos devem, pois, ser ligadas a um bus comum e por esta razão tal bus é mostrado na Fig. 6.3-2. O uso de um bus comum para um número de fontes de entrada foi discutido na Seção 3.18 e seguintes sob o tópico geral de *multiplexação*. Lá notamos que uma maneira que permitia o uso comum de um bus por diversas fontes era equipar as fontes de sinal com saídas de três-estados. Assim, na Fig. 6.2-1 equipamos nossos circuitos de memória com este tipo de saída. Um circuito que não for selecionado tem suas linhas de bits desligadas de seus pinos de saída. O bus comum pode, pois, conduzir os níveis lógicos de saída dos circuitos selecionados sem interferência dos circuitos não selecionados. Nem todas as memórias comerciais são equipadas com saídas de três-estados. Algumas possuem saídas que podem ser ligadas em paralelo através do uso da ligação coletor-aberto descrita na Seção 3.19.

O método de ligação em paralelo da Fig. 6.3-2 pode ser estendido a mais circuitos. Suponha que desejássemos usar aquele método para ligar quatro circuitos em paralelo formando uma memória de 32 palavras e quatro bits/palavra. Introduziríamos um quinto bit de endereço A_5. Os três bits de endereço $A_0A_1A_2$ seriam, como anteriormente, aplicados em comum aos pinos de entrada de endereço dos CIs. Os bits de endereço A_4 e A_5 seriam aplicados a um decodificador, como mostrado na Fig. 6.3-3. As saídas do decodificador seriam usadas para ativar as entradas de seleção de circuito dos quatro CIs. Em muitos casos os fabricantes incluem na memória uma porta AND de entradas múltiplas e fazem com que a saída desta porta seja o nível lógico de

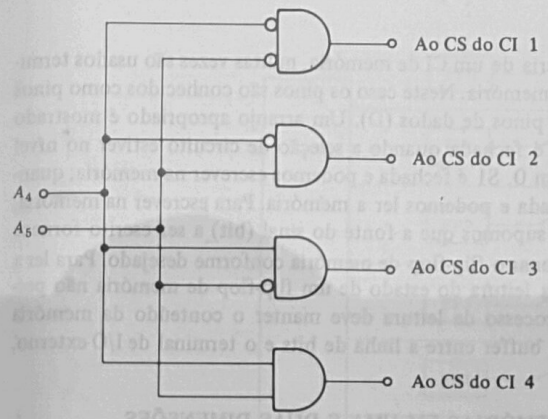


Fig. 6.3-3 Os bits de endereço A_4 e A_5 acionam um decodificador cujas saídas selecionam um de quatro CIs de memória ligados em paralelo.

entrada CS para o CI. Quando tais portas são incluídas, o decodificador externo de CS não é necessário e a decodificação de seleção de circuito pode ser feita no próprio CI. Em um caso típico, um CI de memória pode ter três entradas de seleção de circuito CS1, CS2 e CS3 e o CI só será selecionado quando CS1 = CS2 = CS3 = 1. Neste caso, até 8 (= 2^3) CIs podem ser ligados em paralelo sem decodificação externa.

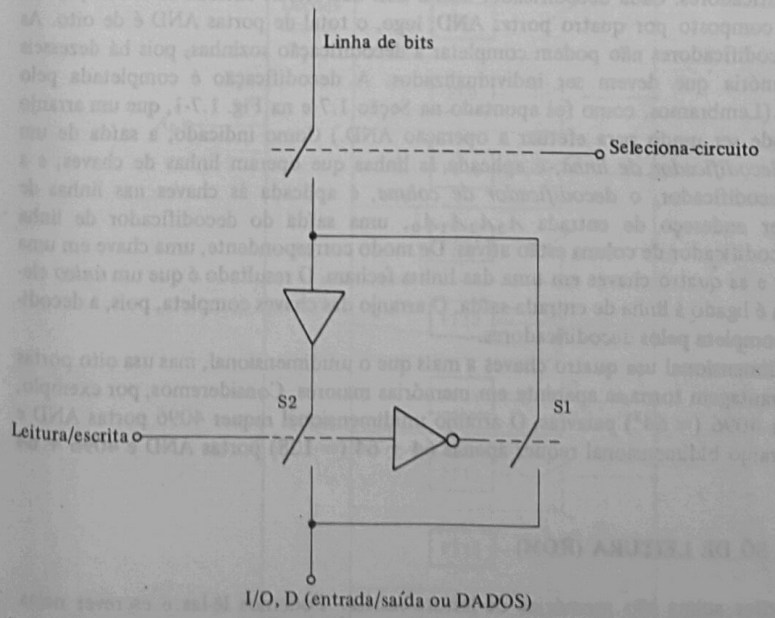


Fig. 6.3-4 Um arranjo que permite o uso do mesmo terminal para entrada e para saída.

Terminais Comuns de Entrada-Saída

Para reduzir o número de pinos na periferia de um CI de memória, muitas vezes são usados terminais comuns para a entrada e a saída da memória. Neste caso os pinos são conhecidos como pinos de entrada-saída (I/O) ou, simplesmente, pinos de dados (D). Um arranjo apropriado é mostrado na Fig. 6.3-4. A chave na linha de bits é fechada quando a seleção de circuito estiver no nível lógico 1. Quando leitura/$\overline{\text{escrita}}$ estiver em 0, S1 é fechada e podemos escrever na memória; quando leitura/$\overline{\text{escrita}}$ estiver em 1, S2 é fechada e podemos ler a memória. Para escrever na memória, provemos acesso direto à linha de bits e supomos que a fonte do sinal (bit) a ser escrito fornece a tensão e a corrente necessárias para acionar o flip-flop de memória conforme desejado. Para ler a memória, queremos ter certeza de que a leitura do estado de um flip-flop de memória não perturbe o estado do flip-flop, isto é, o processo de leitura deve manter o conteúdo da memória inalterado. Por esta razão colocamos um buffer entre a linha de bits e o terminal de I/O externo.

6.4 ORGANIZAÇÃO INTERNA DE MEMÓRIAS EM UMA E DUAS DIMENSÕES

A organização externa do decodificador e das chaves na memória da Fig. 6.2-1 é caracterizada como sendo unidimensional ou linear. Em geral, é vantajosa uma alternativa bidimensional. A distinção entre as duas organizações é mostrada nas Figs. 6.4-1 e 6.4-2. Em ambos os casos supomos uma memória de dezesseis palavras e um bit/palavra. No primeiro caso, há dezesseis elementos de memória (flip-flops), dezesseis chaves e um decodificadr que possui quatro entradas de bits de endereço. O decodificador possui, pois, dezesseis portas AND. No segundo caso, na Fig. 6.4-2, usamos dois decodificadores. Cada decodificador aceita dois dos bits de entrada de endereço, tem quatro saídas e é composto por quatro portas AND; logo, o total de portas AND é de oito. As oito saídas dos decodificadores não podem completar a decodificação sozinhas, pois há dezesseis elementos de memória que devem ser individualizados. A decodificação é completada pelo arranjo das chaves.(Lembramos, como foi apontado na Seção 1.7 e na Fig. 1.7-1, que um arranjo série de chaves pode ser usado para efetuar a operação AND.) Como indicado, a saída de um decodificador, o *decodificador de linha*, é aplicada às linhas que operam linhas de chaves; e a saída do outro decodificador, o *decodificador de coluna*, é aplicada às chaves nas linhas de bits. Para qualquer endereço de entrada $A_3A_2A_1A_0$, uma saída do decodificador de linha e uma saída do decodificador de coluna estão ativas. De modo correspondente, uma chave em uma linha de bits fecha e as quatro chaves em uma das linhas fecham. O resultado é que um *único* elemento de memória é ligado à linha de entrada-saída. O arranjo das chaves completa, pois, a decodificação deixada incompleta pelos decodificadores.

O arranjo bidimensional usa quatro chaves a mais que o unidimensional, mas usa oito portas AND a menos. A vantagem torna-se aparente em memórias maiores. Consideremos, por exemplo, uma memória com 4096 ($= 64^2$) palavras. O arranjo unidimensional requer 4096 portas AND e 4096 chaves. O arranjo bidimensional requer apenas 64 + 64 (= 128) portas AND e 4096 + 64 (= 4160) chaves.

6.5 A MEMÓRIA SÓ DE LEITURA (ROM)

As memórias descritas acima são memórias de leitura-escrita. Podemos lê-las e escrever nelas. Uma memória somente de leitura (*read-only memory*) (ROM) é uma memória que podemos ler

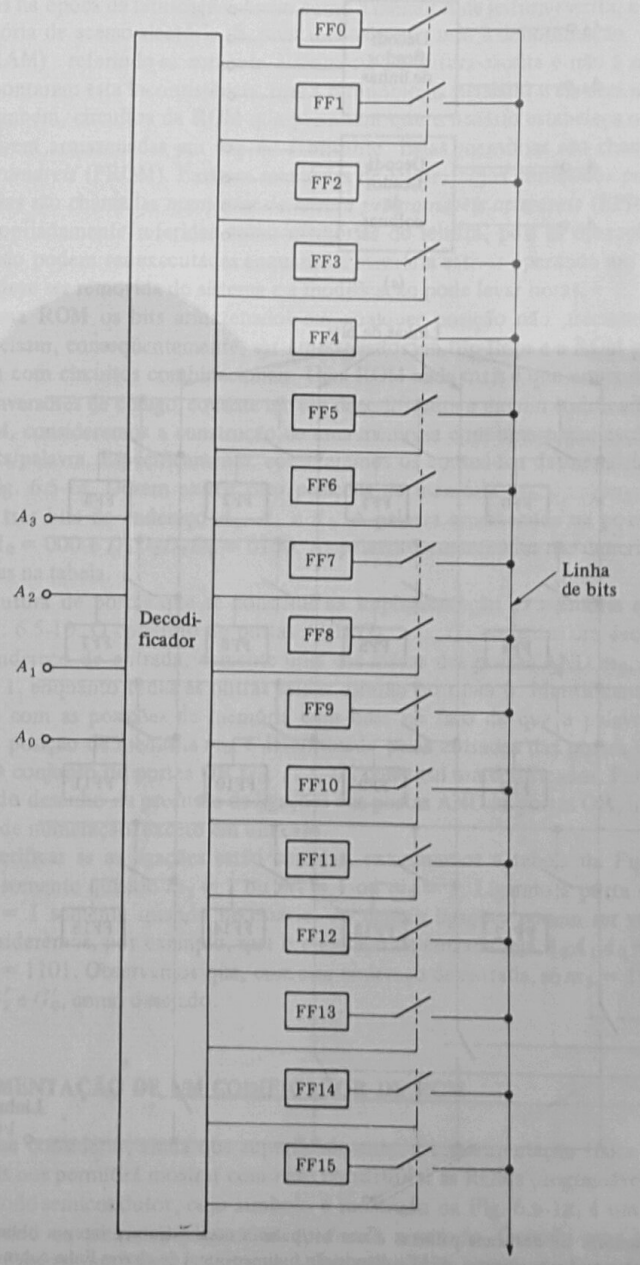


Fig. 6.4-1 Uma memória de dezesseis palavras e um bit/palavra com estrutura interna unidimensional.

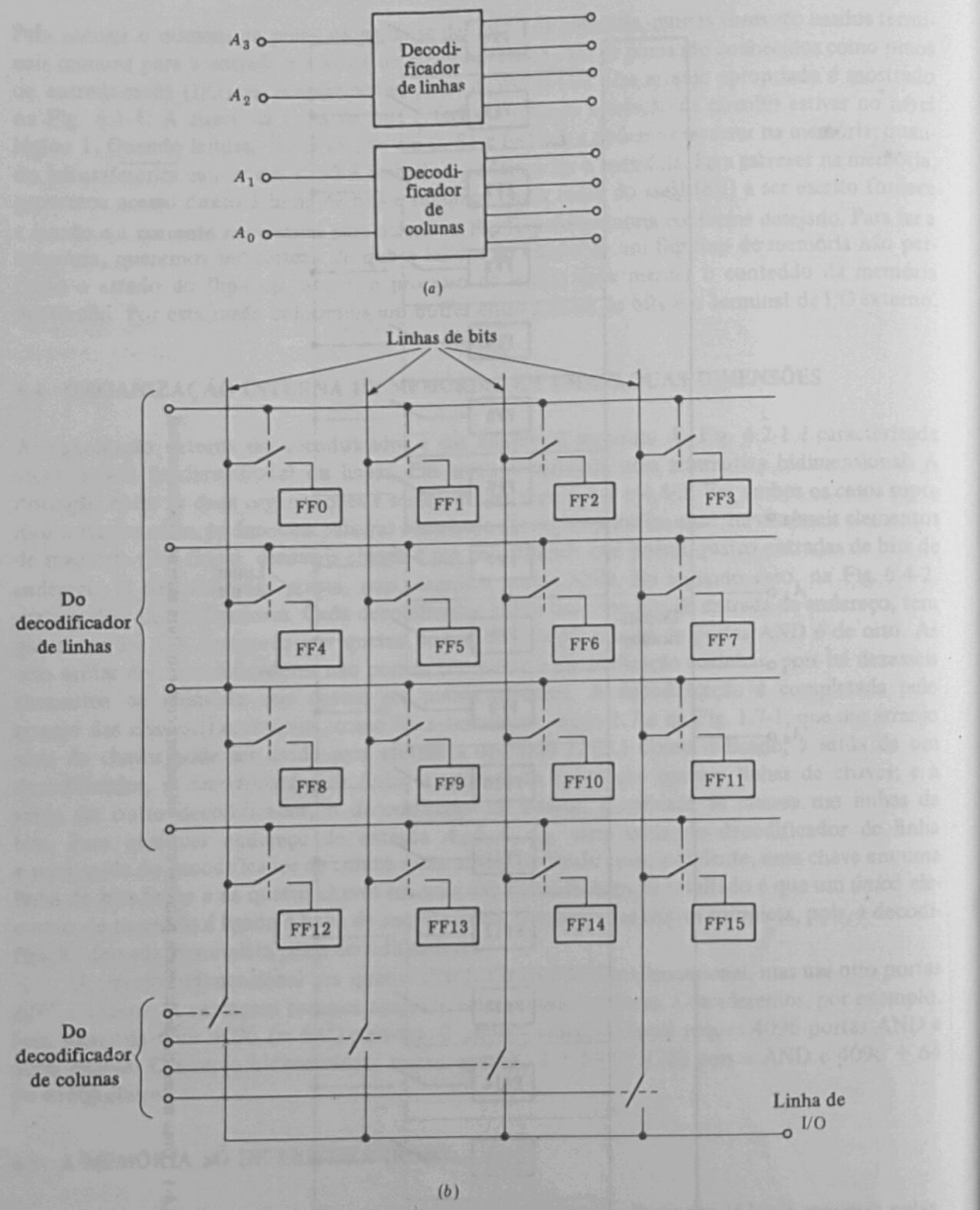


Fig. 6.4-2 Uma memória de dezesseis palavras e um bit/palavra com estrutura interna bidimensional: (*a*) o decodificador de quatro saídas necessário e (*b*) a disposição bidimensional de chaves linha-coluna.

mas na qual não podemos escrever. Os conteúdos da memória são fixos e inalterados, tendo sido estabelecidos na época da fabricação. Assim como a memória de leitura-escrita, a memória de leitura é uma memória de acesso aleatório. É, pois, inconsistente usar a denominação memória de acesso aleatório (RAM) referindo-se somente à memória de leitura-escrita e não à memória de escrita. Muitos já apontaram esta inconsistência, mas a terminologia persistiu e ela será usada aqui.

Há, também, circuitos de ROM que permitem que o usuário estabeleça o conjunto de informações a serem armazenadas em vez do fabricante. Estas memórias são chamadas *memórias de leitura programáveis* (PROM). Existem memórias de leitura cujos conteúdos podem ser alterados. Tais memórias são chamadas *memórias de leitura programáveis apagáveis* (EPROM). Estas memórias são apropriadamente referidas como memórias de leitura, pois as operações de apagamento e reescrita não podem ser executadas enquanto a memória estiver operando em um sistema digital. A memória deve ser removida do sistema e a modificação pode levar horas.

Em uma ROM os bits armazenados em qualquer posição não precisam ser alterados. Os bits não precisam, conseqüentemente, ser armazenados em flip-flops e a ROM pode ser construída inteiramente com circuitos combinacionais. Uma ROM nada mais é que um conversor de código e, como os conversores de código, consiste em um decodificador e em um codificador. Como exemplo de uma ROM, consideremos a construção de uma memória com uma organização de oito palavras e quatro bits/palavra. Especificamente, consideremos os conteúdos da memória especificada pela tabela da Fig. 6.5-1*a*. Devem existir oito posições de memória $m_0, \ldots, m_7$, que serão endereçadas pelos três bits de endereço A_2, A_1 e A_0. A palavra armazenada na posição m_0 com endereço $A_2A_1A_0 = 000$ é $D_3D_2D_1D_0 = 0100$. As palavras armazenadas nas demais posições também são mostradas na tabela.

A estrutura de portas que se constitui na implementação da memória especificada é mostrada na Fig. 6.5-1*b*. O conjunto de portas AND $G_0, \ldots, G_7$ constitui um decodificador. Dependendo do endereço de entrada, somente uma das saídas das portas AND $m_0, \ldots, m_7$ estará no nível lógico 1, enquanto todas as outras saídas estarão no nível 0. Identificamos estas saídas das portas AND com as posições de memória com base no fato de que a palavra armazenada, por exemplo, na posição de memória m_0, é determinada pelas entradas das portas OR às quais m_0 for ligada etc. O conjunto de portas OR $G_0', \ldots, G_3'$ constitui um codificador. Para evitar a confusão que adviria do desenho da profusão de ligações das portas AND às portas OR, indicamos estas ligações através de numeração, exceto em um caso.

Para verificar se as ligações estão corretas, examinamos a tabela da Fig. 6.5-1*a*. Notamos que $D_3 = 1$ somente quando $m_2 = 1$ ou $m_3 = 1$ ou $m_5 = 1$. Ligando a porta G_3' a m_2, m_3 e m_5, teremos $D_3 = 1$ somente quando necessário. As demais ligações podem ser verificadas de modo similar. Consideremos, por exemplo, que o endereço de entrada seja $A_2A_1A_0 = 011$. Devemos ter $D_3D_2D_1D_0 = 1101$. Observamos que, com esse endereço de entrada, só $m_3 = 1$ e m_3 são ligados às portas G_3', G_2' e G_0', como desejado.

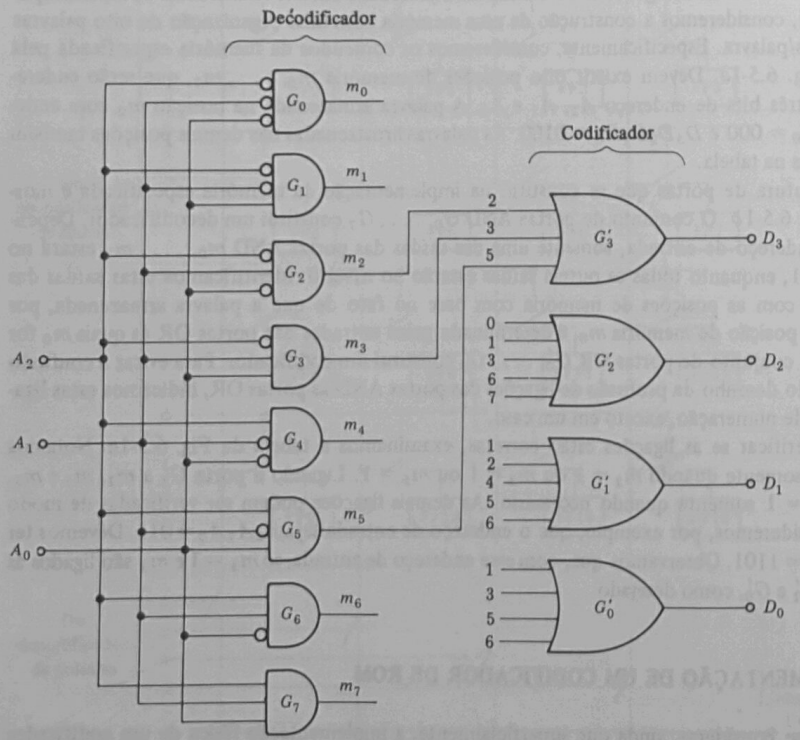
6.6 IMPLEMENTAÇÃO DE UM CODIFICADOR DE ROM

É de interesse considerar, ainda que superficialmente, a implementação física de um codificador de ROM, pois nos permitirá mostrar como são construídas as ROMs programáveis.

Um díodo semicondutor, cujo símbolo é mostrado na Fig. 6.6-1*a*, é um dispositivo de dois terminais, sendo os terminais chamados de *ânodo* e *cátodo*. Quando uma tensão for aplicada ao díodo na direção que torna o ânodo positivo em relação ao cátodo (isto é, V na Fig. 6.6-1*a* é

Posição de memória	Endereço			Palavra de dados			
	A_2	A_1	A_0	D_3	D_2	D_1	D_0
m_0	0	0	0	0	1	0	0
m_1	0	0	1	0	1	1	1
m_2	0	1	0	1	0	1	0
m_3	0	1	1	1	1	0	1
m_4	1	0	0	0	0	1	0
m_5	1	0	1	1	0	1	1
m_6	1	1	0	0	1	1	1
m_7	1	1	1	0	1	0	0

(*a*)



(*b*)

Fig. 6.5-1 (*a*) Especificação de uma ROM de oito palavras e quatro bits/palavras. (*b*) O decodificador-codificador (conversor de código) que é uma implementação da memória em (*a*).

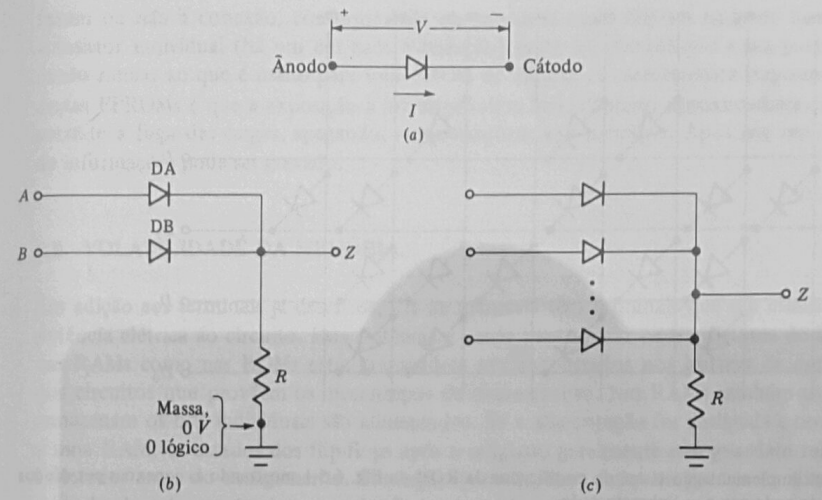


Fig. 6.6-1 (*a*) Símbolo de um diodo. (*b*) Uma porta OR. (*c*) Uma porta OR com múltiplas entradas.

um número positivo), uma corrente I circulará através do diodo na direção indicada pela seta. Quando a tensão aplicada for invertida, não circulará corrente. O diodo é um dispositivo unilateral e só permite circulação de corrente em uma direção.

O circuito da Fig. 6.6-1*b*, que envolve dois diodos e um resistor, é uma porta OR de duas entradas. Consideramos que "massa", isto é, 0 V é a tensão $V(0)$ corresponde ao nível lógico 0, ao passo que uma tensão positiva $V(1)$ corresponde ao nível 1. Se as tensões em A e B forem ambas $V(0)$, ou seja, correspondentes ao nível 0, então nenhum dos diodos conduzirá e a saída Z estará também no nível 0. Se A estiver em $V(1)$ (nível lógico 1) e B estiver em $V(0)$ (nível 0), então o diodo DA conduzirá, o diodo DB terá sobre si uma tensão na direção inversa, de modo que não conduzirá, e a saída Z estará no nível lógico 1. De modo similar, se A estiver em $V(0)$ e B em $V(1)$, DB conduzirá e Z estará no nível 1. Se A e B estiverem no nível 1, Z estará também no nível 1. Conforme é requerido de uma porta OR, a saída estará no nível 1 se uma ou outra ou ambas as entradas estiverem no nível 1. Para acomodar um número maior de entradas à porta OR, basta adicionar diodos da maneira indicada na Fig. 6.6-1*c*. Deve ser notado que o essencial nesta aplicação de um diodo é sua propriedade unilateral. Outros dispositivos eletrônicos, como transistores, bipolares ou MOS, que também possuem propriedades unilaterais, podem substituir os diodos.

O codificador da ROM da Fig. 6.5-1 consiste em um conjunto de portas OR e, como acabamos de verificar, uma porta OR pode ser construída com diodos e um resistor (Seção 1.8). A implementação do codificador da ROM da Fig. 6.5-1 é mostrada na Fig. 6.6-2. Nesta figura vemos uma matriz de linhas de endereço e de linhas de saída de dados cruzadas, geradas pelos decodificadores. Os três diodos ligados à linha de dados D_3, juntamente com o resistor ligado àquela linha, constituem a porta OR de três diodos mostrada na Fig. 6.5-1 como G'_3. Assim, como desejado, D_3 estará no nível 1 se m_2 ou m_3 ou m_5 estiver em 1. De modo similar, podemos verificar que os

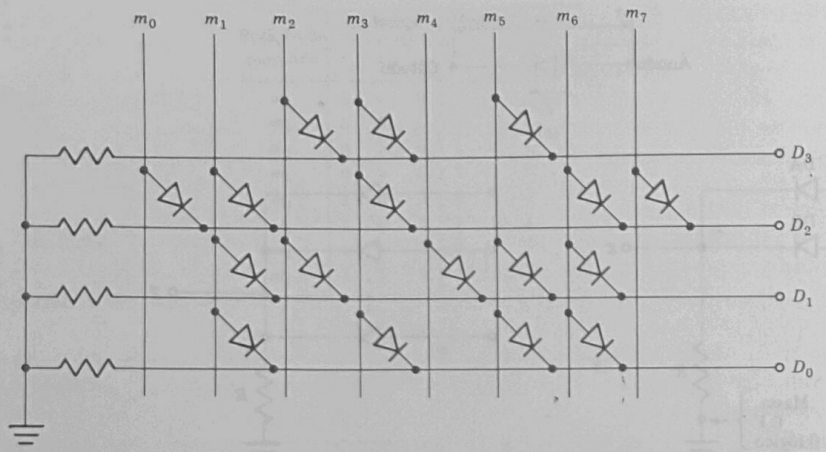


Fig. 6.6-2 Uma implementação física do codificador da ROM da Fig. 6.5-1 mostrando as conexões por díodos entre as linhas de endereço e as linhas de dados.

demais díodos estão apropriadamente localizados. Há tantos díodos no codificador quantos 1s nas colunas da tabela da Fig. 6.5-1*a*. No circuito da Fig. 6.6-2 sempre que um 1 aparecer na interseção da linha m_j com a coluna D_k da tabela, localizamos um díodo ligando a linha de endereço m_j à linha de dados D_k. Em uma RAM necessitamos um flip-flop para cada bit. Em uma ROM um díodo ou sua ausência estabelece um bit permanentemente.

6.7 ROMs PROGRAMÁVEIS E APAGÁVEIS

Uma ROM *programável* é aquela em que o fabricante inclui uma ligação em *cada* interseção da grade de linhas de endereço e de dados. (A conexão consiste em um díodo, como na Fig. 6.6-2, ou em um transistor de algum tipo.) Em série com cada conexão, o fabricante inclui um fusível que pode ser derretido e, conseqüentemente, aberto através da passagem de uma corrente suficientemente grande por ele. Podemos individualmente selecionar um díodo pela seleção de uma linha de dados e do endereço de entrada apropriado. Por exemplo, suponhamos que na ROM das Figs. 6.5-1 e 6.6-2 aplicássemos o endereço $A_2A_1A_0 = 010$. A linha de saída do decodificador m_2 estaria em 1. Vamos supor que a linha de dados D_3 fosse ligada a uma fonte de tensão cuja tensão fosse menor que a da linha m_2. Neste caso somente um díodo conduziria corrente (aquele ligado entre m_2 e D_3). Se diminuirmos a tensão a que está ligada D_3, alcançaremos um ponto em que o fusível em série com o díodo abrirá e a conexão entre m_2 e D_3 deixará de existir. Deste modo podemos percorrer sistematicamente as interseções de uma PROM fazendo desconexões conforme for de nosso desejo. Uma PROM pode ter milhares de conexões, mas existe equipamento que permite a programação de maneira conveniente em um tempo razoável.

Nas ROMS *programáveis* e *apagáveis*, o elemento usado para fazer as conexões entre as linhas do decodificador e as linhas de dados não é um díodo, mas um transistor MOS. Estes dispositivos

fazem ou não a conexão, conforme haja ou não uma carga elétrica na *porta* do transistor. Um transistor individual (há um em cada interseção) pode ser identificado e sua porta carregada de modo similar ao que é usado para uma PROM de díodos. A característica importante e distintiva destas EPROMs é que a exposição à luz ultravioleta forte (durante aproximadamente 30 minutos) permite a fuga das cargas, apagando, em conseqüência, a memória. Após isto um novo conjunto de informações pode ser gravado.

6.8 VOLATILIDADE DA MEMÓRIA

Em adição aos terminais já descritos, CIs de memória têm terminais que são usados para entregar potência elétrica ao circuito. Esta potência é usada para operar os transistores do circuito. Tanto nas RAMs como nas ROMs estes transistores são encontrados nos buffers de entrada e saída e nos circuitos que provêem os mecanismos de chaveamento. Nas RAMs também os flip-flops que armazenam os bits individuais são alimentados. Se a alimentação for desligada e novamente ligada a uma RAM, os estados dos flip-flops após a religação geralmente não guardam relação útil com os estados antes do desligamento. Conseqüentemente, uma falta temporária de alimentação causa a perda da informação da memória. Por esta razão uma RAM semicondutora é descrita como *memória volátil*.

Vimos que em uma ROM a informação armazenada é determinada pela localização dos díodos na matriz de linhas de saída e do decodificador. Em alguns casos as conexões por díodos são substituídas por uma conexão por transistor que deve ser alimentada como qualquer outro transistor. O ponto importante é que os bits armazenados são determinados somente pela localização da conexão por diodo ou por transistor. Assim, na ROM o conjunto de informações é permanente e não-volátil e a memória retorna inalterada após a religação da alimentação.

Quando a perda de memória em uma RAM devida à falta de alimentação deve ser evitada, é necessário estabelecer um sistema de suporte, geralmente um sistema à bateria. Existem sistemas e componentes comerciais que são projetados para sentir a queda na tensão de alimentação e transferir a RAM para a fonte de suporte rapidamente de modo a evitar uma perda de memória.

6.9 TEMPOS DE CHAVEAMENTO DAS MEMÓRIAS

Já observamos que as portas lógicas requerem certo tempo para responder a alterações na entrada. As limitações de velocidade das portas são, geralmente, expressas sob a forma de *atrasos de propagação* entre as alterações na entrada e as respostas na saída. Ocasionalmente, para portas muito rápidas, os fabricantes fornecem informação sobre os *tempos de subida* e *descida* das formas-de-onda na saída. Para dispositivos mais complicados, a informação fornecida é mais extensa. Para flip-flops (como vimos na Seção 4.15), os fabricantes fornecem não só tempos de propagação mas também tempos de *preparação* e *manutenção* e *máxima freqüência de chaveamento*. Memórias são ainda mais complicadas. Os CIs de memória são geralmente considerados circuitos com integração em larga escala em comparação com circuitos com integração em média escala, como contadores e registradores de deslocamento, e em pequena escala, como portas lógicas simples.

A complicação na especificação dos tempos de chaveamento das memórias ocorre principalmente devido ao grande número de tipos de terminais: terminais de endereço, terminais de

entrada, terminais de saída (estes dois últimos às vezes combinados como terminais de dados), terminais de habilitação de circuito e terminais de leitura-escrita. Alguns fabricantes incorporam características especiais que requerem ainda maior número de terminais. Um dicionário típico de definições de símbolos de chaveamento, encontrado em um catálogo de componentes de memória, pode conter mais de vinte símbolos. A situação é piorada pelo fato de que fabricantes diferentes usam símbolos completamente diferentes. A seguir consideraremos as especificações de chaveamento mais importantes e mais comumente usadas.

Um conjunto típico de sinais de chaveamento para uma ROM é mostrado na Fig. 6.9-1. Os endereços de entrada são aplicados em $t = t_0$. Há um número de terminais de endereço e, quando um endereço é aplicado, algumas entradas irão de 0 para 1 e outras de 1 para 0. Assim, a entrada de endereços é representada não por um sinal que executa uma transição de 0 para 1 ou 1 para 0, mas por dois sinais executando transições opostas e se cruzando para $t = t_0$. A saída de dados geralmente aparece em um número de terminais (exceto em memórias com palavras de um bit); portanto, o sinal de saída de dados também é representado por dois sinais cruzados. Observamos que, se o endereço for aplicado em $t = t_0$, a palavra endereçada aparece nos terminais de saída da memória com todos os bits corretos, isto é, os dados de saída tornam-se *válidos*, somente no instante t_2 após um tempo de *acesso após o endereço* t_{AA}. Nenhum dado aparecerá na saída até que a entrada de seleção de circuito (ou, mais geralmente, as múltiplas entradas) tenha sido ativada. O tempo de resposta da memória às entradas de seleção do circuito é o tempo de *habilitação de circuito* $t_{CE} = t_2 - t_1$. Os sinais são desenhados de forma a reconhecer que $t_{CE} < t_{AA}$. Os sinais indicam que, se as seleções de circuito são acionadas um tempo t_{CE} antes de t_2, o atraso da ativação $(t_1 - t_0)$ não causa atraso adicional no aparecimento da palavra de saída válida. Alguns fabricantes não especificam t_{CE}, mas sim $t_1 - t_0$ ou o tempo que a ativação da seleção de circuito pode ser atrasada em relação à aplicação do endereço sem causar atraso adicional (além de t_{AA}) no aparecimento dos dados de saída válidos.

O diagrama da Fig. 6.9-1 indica que o endereço é alterado em t_3. Todavia, os dados de saída correspondentes ao endereço aplicado em t_0 permanecem válidos até $t = t_4$. O intervalo em que os dados permanecem válidos após a mudança de endereço é, geralmente, chamado t_{OH}

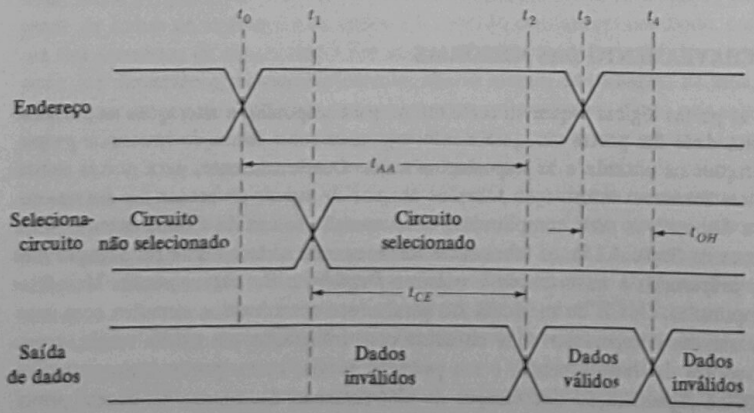


Fig. 6.9-1 Sinais de tempos de memória ilustrando os efeitos de uma mudança de endereço.

(manutenção da saída). Após $t = t_4$ os dados de saída começam a mudar em resposta ao novo endereço. Em um intervalo t_{AA} após t_3 os dados de saída serão válidos novamente e corresponderão ao endereço aplicado em $t = t_4$.

Nos sinais da Fig. 6.9-1 medimos os tempos a partir do momento em que os níveis lógicos se encontram no meio da transição. Alguns fabricantes preferem levar em conta os tempos de subida e descida dos sinais. Os tempos, então, são medidos a partir do momento em que um sinal crescente atinge 90% do nível lógico 1 ou que um sinal decrescente chegou a 10% do nível 0. (Fazemos a aproximação razoável que os pontos de 10 e 90% são atingidos simultaneamente.) Julgamos, então, que um novo endereço ou que novos dados não são válidos até que tenham completado 90% de qualquer alteração de nível requerida. Se os dados estiverem mudando, os dados antigos deixam de ser válidos se uma mudança de 10% tiver lugar. Com este ponto de vista, desenhamos na Fig. 6.9-2 um diagrama de tempos mostrando a mesma informação que a Fig. 6.9-1. Na Fig. 6.9-2, todavia, consideramos que em $t = t_3$, e após dados válidos aparecerem, o circuito foi desselecionado em vez de se alterar o endereço. O atraso na resposta do circuito à desseleção é indicado por t_{CD} (desseleção de circuito). Quando mais que um circuito for acoplado a um *bus* (barramento) comum, é importante que, ao transferir o barramento de um circuito para outro, o primeiro circuito seja desselecionado antes que o segundo seja selecionado. Para facilitar esta ocorrência, os fabricantes fazem com que t_{CD} seja substancialmente mais curto que t_{OH}.

A discussão precedente relativa a tempos envolveu apenas ROMs. Consideremos, a seguir, RAMs. Quando uma RAM é lida, as considerações são quase idênticas às feitas para as ROMs. O terminal de *habilita-escrita* pode ser pensado como um terminal de *seleção de circuito* adicional. Com este entendimento, os sinais das Figs. 6.9-1 e 6.9-2 aplicam-se tanto às ROMs como às RAMs, quando estas forem lidas.

Quando desejarmos escrever em uma RAM, o terminal *habilita-escrita* deve ser forçado ao nível correspondente à escrita. Para que isto seja efetivo, algumas considerações especiais devem ser feitas, conforme indicado na Fig. 6.9-3. Um pulso de entrada de escrita deve ser aplicado e supomos, pois geralmente este é o caso, que a entrada *habilita-escrita* é ativa quando baixa. Para que

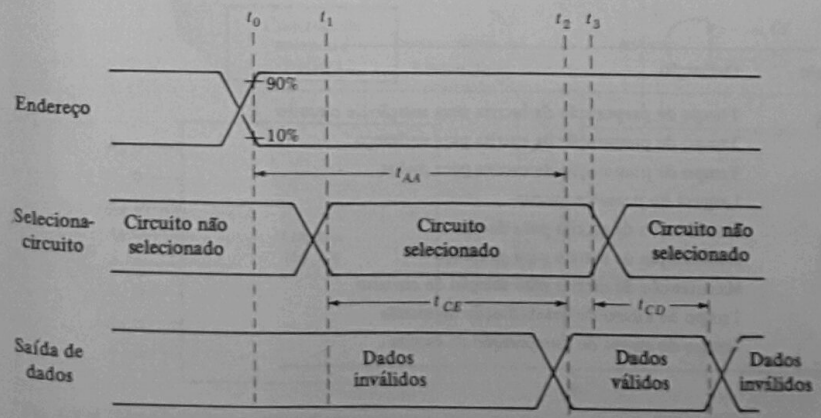
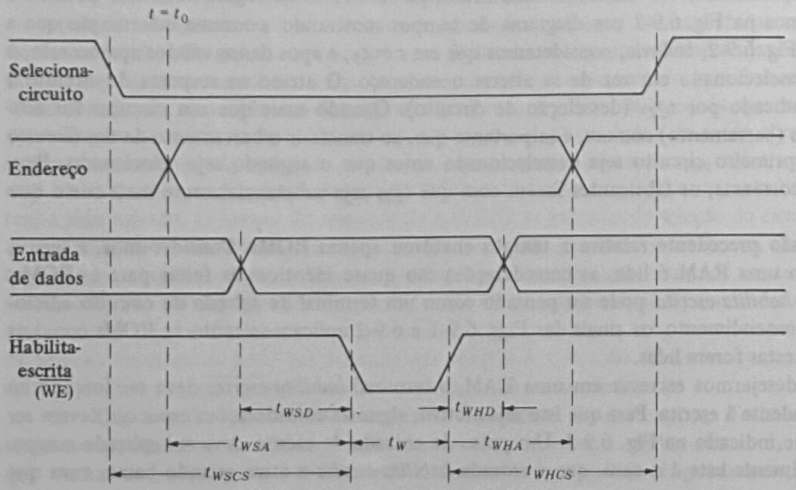


Fig. 6.9-2 Sinais de tempos de memória ilustrando o efeito da desseleção do circuito.

este pulso seja efetivo, é necessário que antes da ocorrência de sua borda dianteira a seleção de circuito seja ativada e que tanto o endereço como os dados a serem escritos estejam presentes às entradas da memória. Isto quer dizer que a seleção de circuito, os dados e o endereço devem ser *preparados* antes do pulso de escrita. Os tempos de *preparação* são indicados no diagrama da Fig. 6.9-3*a*, e as definições dos símbolos são dadas na Fig. 6.9-3*b*. Há uma limitação mínima de largura t_W do pulso de escrita. Finalmente, após o fim do pulso de escrita, as entradas devem ser mantidas por algum tempo para que os dados de entrada possam ser confiavelmente estabelecidos em seus endereços. Há, pois, tempos de *manutenção* para seleção de circuito, para endereço e para dados. Os sinais da Fig. 6.9-3 foram desenhados para permitir a fácil indicação dos diversos tempos.



(*a*)

Símbolo	Definição
t_{WSCS}	Tempo de preparação de escrita para seleção de circuito
t_{WSA}	Tempo de preparação de escrita para endereço
t_{WSD}	Tempo de preparação de escrita para dados
t_W	Largura do pulso de escrita
t_{WHD}	Manutenção de escrita para dados
t_{WHA}	Manutenção de escrita para endereço
t_{WD}	Manutenção de escrita para seleção de circuito
t_{WR}	Tempo de atraso de desabilitação de escrita
	Tempo de atraso de recuperação de escrita

(*b*)

Fig. 6.9-3 (*a*) Sinais associados com escrita na memória. (*b*) Definições dos símbolos.

A figura faz parecer que os tempos de preparação e manutenção são maiores para seleção de circuito etc. Isto não ocorre necessariamente. Os tempos de preparação são freqüentemente iguais, como o são os de manutenção, e ambos são menores que a largura mínima do pulso de escrita.

Outros tempos são de interesse em relação à operação de escrita. Quando $\overline{WE}$ for 1, a memória está no modo de leitura e os buffers de saída estão habilitados, enquanto os de entrada estão desabilitados. Quando $\overline{WE}$ passa a 0, a situação se inverte. O tempo necessário para passar do modo de leitura para o de escrita é freqüentemente chamado *tempo de atraso de escrita*, simbolizado por t_{WD} ou t_{WS} (preparação de escrita). Se uma memória tem saída de três-estados, a passagem para o modo de escrita coloca as saídas no estado de alta impedância (Z alta) e o tempo t_{WD} pode ser representado por t_{ZWS}. O tempo necessário para voltar ao modo de leitura é o *tempo de recuperação de escrita* t_{WR}.

A entrada WE habilita a entrada *ou* a saída da memória. A entrada de seleção de circuito serve para desabilitar a entrada e a saída ao *mesmo tempo*. Muitas vezes é conveniente ter uma entrada que possa habilitar ou desabilitar só as ligações de saída. Isto sempre será necessário quando o bus de dados a que está ligada a memória for necessário para outro propósito. Muitas memórias oferecem esta entrada de controle chamado OD (desabilita saída) ou EO (habilita saída). Este terminal adicional causa a introdução de novos parâmetros, os tempos de habilitação e desabilitação para os buffers de saída, para os quais não há simbolismo geralmente aceito.

Na Fig. 6.9-4 representamos a lógica de controle de uma memória típica. Há uma entrada $\overline{WE}$, duas entradas de seleção de circuito CS e $\overline{CS}$ e uma entrada de desabilitação da saída $\overline{OD}$. Supomos, como é usual, que as chaves são fechadas pelas linhas de controle em nível 1 e abertas em nível 0. Quando, por exemplo, $\overline{OD} = 0$, as chaves no circuito de saída estão abertas e a saída é desabilitada, como se deseja.

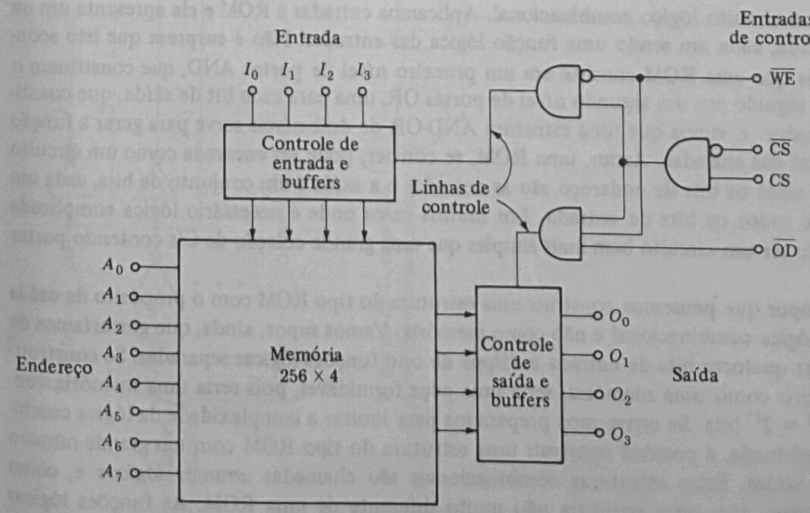


Fig. 6.9-4 Uma memória comercial típica mostrando entradas de controle ($\overline{WE}$, CS, $\overline{CS}$, $\overline{OD}$), entradas, saídas e endereços.

Nem todos os fabricantes especificam as características das memórias da mesma maneira. A notação varia de fabricante para fabricante. A maioria dos fabricantes apresenta um glossário completo dos símbolos usados.

Há possibilidade de confusão na maneira que os fabricantes usam para especificar valores para certas características. Por exemplo, o CI 93411 da Fairchild tem um tempo de acesso de endereço *típico* de $t_{AA} = 40$ ns, enquanto o máximo valor deste parâmetro é 45 ns. Não há problema nesta especificação. Adiante, porém, encontramos largura *típica* do pulso de escrita $t_W = 25$ ns, enquanto o *valor mínimo é especificado* $t_W = 40$ ns. A interpretação desta aparente inconsistência é que a unidade tipicamente operará propriamente com $t_W = 25$ ns, mas algumas unidades requerem t_W de até 40 ns. A mínima largura que o usuário deve usar é, pois, 40 ns. De modo geral, quando encontramos uma especificação mínima maior que a típica, esta especificação deve ser interpretada como uma condição imposta ao usuário.

A velocidade com que uma memória pode operar é facilmente estimada em função da informação descrita. O tempo mínimo para leitura é, aproximadamente, o tempo de acesso t_{AA}. O tempo mínimo para uma operação de escrita é aproximadamente a soma do tempo de preparação (o mais longo) com a largura do pulso de escrita e o tempo de manutenção. Além das especificações usuais, alguns fabricantes fornecem os tempos mínimos recomendados para que se execute uma leitura ou escrita com confiabilidade. Estes são os tempos de ciclo de leitura t_{RC} e de ciclo de escrita t_{WC}. Estes tempos geralmente são comparáveis e, quando há diferença, o ciclo de leitura é quase sempre o mais longo. O tempo t_{RC} para as unidades comercialmente disponíveis varia de 50 a 1500 ns.

6.10 O ARRANJO LÓGICO PROGRAMÁVEL

Uma ROM é um circuito lógico combinacional. Aplicamos entradas à ROM e ela apresenta um ou mais bits de saída, cada um sendo uma função lógica das entradas. Não é surpresa que isto aconteça, pois vimos que uma ROM consiste em um primeiro nível de portas AND, que constituem o decodificador, seguido por um segundo nível de portas OR, uma para cada bit de saída, que constituem o codificador. E vimos que uma estrutura AND-OR de dois níveis serve para gerar a função lógica mais geral das entradas. Assim, uma ROM, se convier, pode ser encarada como um circuito combinacional onde os bits de endereço são as entradas e a saída é um conjunto de bits, cada um uma função de todos os bits de entrada. Em muitos casos onde é necessário lógica complicada uma ROM pode ser um circuito bem mais simples que uma grande coleção de CIs contendo portas AND e OR.

Vamos supor que pensemos construir uma estrutura do tipo ROM com o propósito de usá-la somente para lógica combinacional e não como memória. Vamos supor, ainda, que gostaríamos de poder acomodar quatorze bits de entrada e dispor de oito funções lógicas separadas. Se construirmos o dispositivo como uma memória, seria uma peça formidável, pois seria uma memória contendo $2^{14} \times 2^3 = 2^{17}$ bits. Se estivermos preparados para limitar a complexidade da lógica combinacional a ser efetuada, é possível construir uma estrutura do tipo ROM com um grande número de entradas e saídas. Estas estruturas combinacionais são chamadas *arranjos lógicos* e, como podemos antecipar, têm uma estrutura não muito diferente de uma ROM. As funções lógicas geradas por arranjos lógicos podem ser *programadas* durante a fabricação pela seleção dos lugares em que são feitas ligações em um arranjo cruzado de linhas de modo similar ao usado para pro-

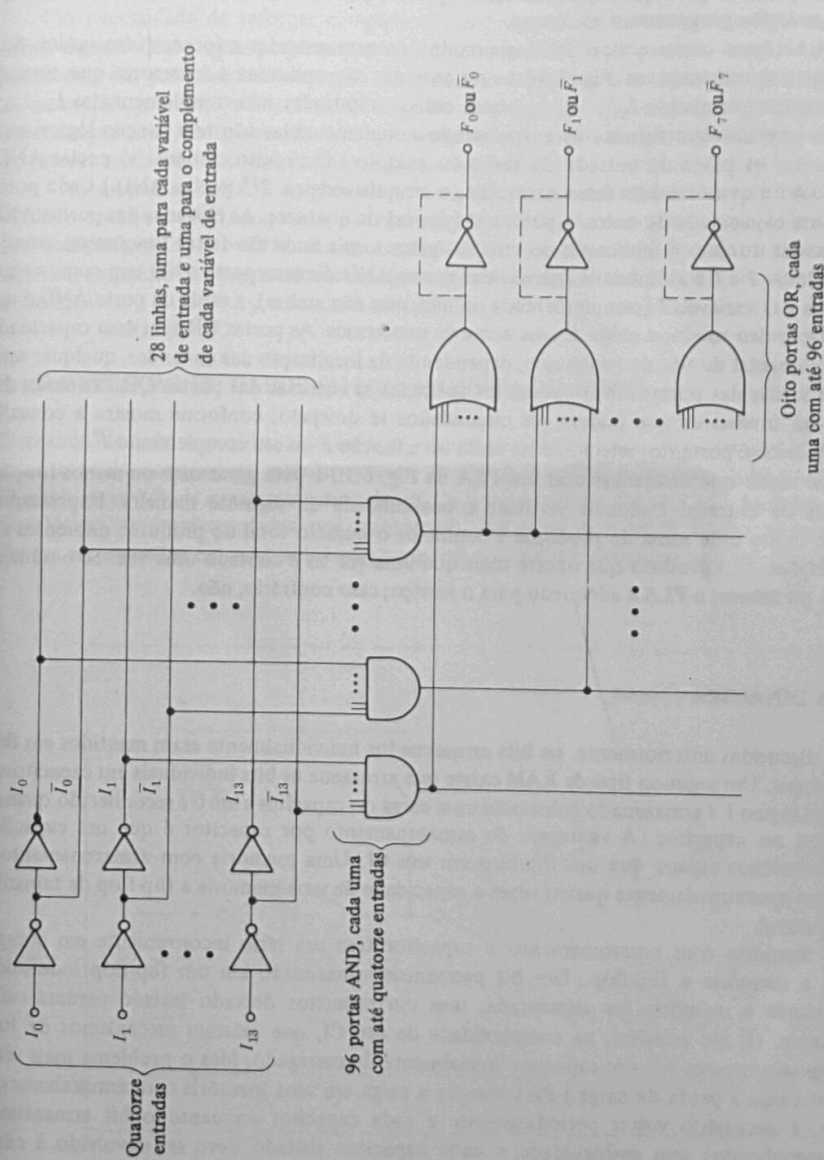


Fig. 6.10-1 A estrutura de portas lógicas de um arranjo lógico programável comercial (tipo 7575).

gramar uma ROM. Por esta razão o arranjo é chamado *arranjo lógico programável* (PLA). Se o fabricante faz provisão para que a programação seja feita pelo usuário, o PLA é conhecido como FPLA, *arranjo lógico programável no campo*.

Um PLA típico como o tipo 7575 acomoda quatorze entradas e fornece oito saídas. Sua estrutura lógica é mostrada na Fig. 6.10-1. As entradas são aplicadas a inversores que tornam disponíveis os complementos $\overline{I}_0, \ldots, \overline{I}_{13}$, bem como as entradas não-complementadas $I_0, \ldots, I_{13}$. (A dupla inversão que fornece as entradas não-complementadas não tem função lógica, mas serve para isolar os pinos de entrada do resto do circuito.) O circuito contém 96 portas AND. (Se uma ROM de quatorze bits fosse necessária, o circuito exigiria 2^{14} portas AND.) Cada porta AND tem uma capacidade de entrada potencial (*fan-in*) de quatorze. As entradas das portas AND são estabelecidas durante a fabricação do circuito pelos locais onde são feitas ligações no arranjo cruzado de linhas I e $\overline{I}$ e as linhas de entrada das portas AND. Se uma porta AND tem como entradas cada uma das variáveis I (complementada ou não, mas não ambas), a saída da porta AND é um mintermo. De outro modo, a saída é uma soma de mintermos. As portas OR têm uma capacidade de entrada potencial de 96, de modo que, dependendo da localização das conexões, qualquer uma ou todas as saídas das portas AND podem ser aplicadas às entradas das portas OR. Na saída das portas OR há inversores que podem ser circundados se desejado, conforme mostra a conexão tracejada. Podemos, portanto, selecionar na saída ou a função F ou seu complemento $\overline{F}$.

Vamos supor que desejemos usar um PLA da Fig. 6.10-1 para gerar oito ou menos funções das variáveis de entrada. Podemos verificar a possibilidade da seguinte maneira. Expressamos cada função como uma soma de produtos e contamos o número total de produtos diferentes em todas as funções. Um produto que ocorre mais que uma vez só é contado uma vez. Se o número total for 96 ou menor, o PLA é adequado para o serviço; caso contrário, não.

6.11 RAM DINÂMICA

Nas RAMs discutidas anteriormente, os bits armazenados individualmente eram mantidos em flip-flops individuais. Um segundo tipo de RAM existe que armazena os bits individuais em capacitores, isto é, um bit lógico 1 é armazenado colocando uma carga no capacitor e um 0 é reconhecido quando não há carga no capacitor. A vantagem do armazenamento por capacitor é que um capacitor ocupa muito menos espaço que um flip-flop em um CI. Uma memória com armazenamento a capacitor tem aproximadamente quatro vezes a capacidade de uma memória a flip-flop de tamanho físico comparável.

Uma memória com armazenamento a capacitor tem um sério inconveniente em comparação com a memória a flip-flop. Um bit permanece armazenado em um flip-flop indefinidamente enquanto a memória for alimentada, mas um capacitor deixado isolado perderá carga inevitavelmente. (É até possível, na complexidade de um CI, que existam mecanismos de fuga tais que depositem carga em um capacitor inicialmente descarregado. Mas o problema mais sério é a fuga que causa a perda de carga.) Para manter a carga em uma memória com armazenamento a capacitor, é necessário voltar periodicamente a cada capacitor enquanto o bit armazenado ainda for reconhecível sem ambigüidade; e cada capacitor visitado deve ser devolvido à carga completa correspondente ao bit armazenado. Esta operação de restaurar a carga inicial é descrita como operação de *refrescamento*. Tipicamente, em memórias comerciais cada bit tem que ser reforçado a intervalos que não excedam 2 ms. Por causa da constante atividade de reforço requerida, memórias com armazenamento a capacitor são chamadas memórias *dinâmicas*, em contraposição às memórias com flip-flop, que são chamadas memórias *estáticas*.

A necessidade de reforçar complica a incorporação de uma memória dinâmica em um sistema digital. Onde uma pequena capacidade de memória é requerida, cerca de 64×10^3 bits ou menos, em geral, não há vantagem em usar memórias dinâmicas; quando aparece a necessidade de memórias grandes, em torno de 128×10^3 ou maiores, é que surge uma vantagem de custo ao usar as memórias dinâmicas. Entre estes intervalos a escolha é determinada por uma variedade de fatores relacionados com o sistema digital completo.

No momento a RAM estática disponível com maior capacidade tem dezesseis kilobits (na verdade $16.384 = 2^{14}$), enquanto a maior memória dinâmica tem uma capacidade de 64 kilobits (na realidade $65.536 = 2^{16}$). RAMs estáticas têm comprimentos de palavra de oito, quatro e um bit, enquanto as RAMs dinâmicas quase invariavelmente têm palavras de comprimento igual a um bit.

6.12 DETECÇÃO DE DADOS EM UMA MEMÓRIA DINÂMICA

Deixando de lado o problema do reforço, vamos considerar como um capacitor armazenador de bit é tornado acessível ao mundo externo ao CI. A interface entre uma linha de dados externa D e um *capacitor de armazenamento de bit* C_B é mostrada na Fig. 6.12-1. As chaves controladas por lógica são implementadas por transistores, em geral tipo MOS. Como veremos, as chaves devem ser operadas em uma seqüência temporal especial. Os geradores que provêem níveis lógicos

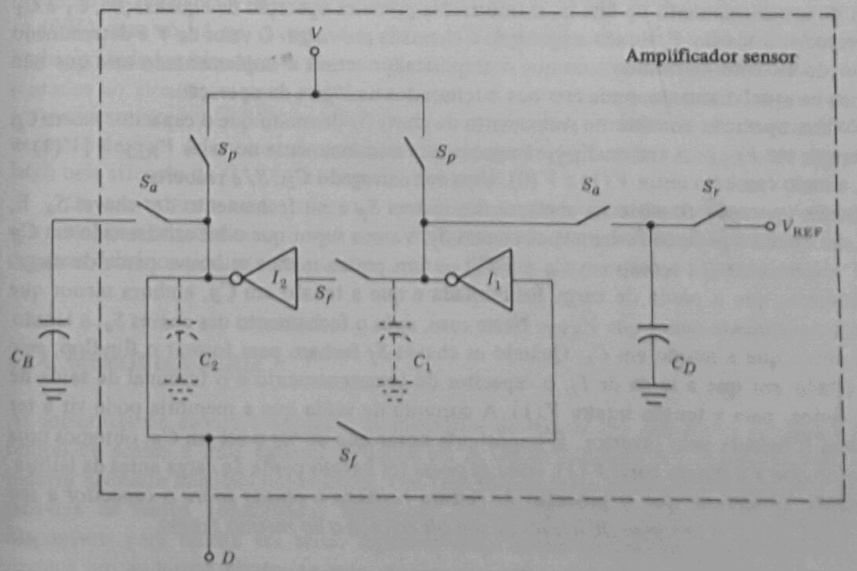


Fig. 6.12-1 Um amplificador sensor usado para ler e reforçar o bit armazenado em C_B.

para controlar estas chaves estão incorporados ao circuito. Como nosso interesse é na lógica da memória e não na eletrônica, tomamos algumas liberdades com o intuito de simplificar sem distorcer o princípio. O circuito entre C_B e D é chamado *amplificador sensor*.

Quando as duas chaves S_f estiverem fechadas (elas abrem e fecham juntas), os dois inversores estão ligados de modo a constituir um flip-flop. A saída de um inversor é $V(1)$, correspondente ao nível 1, e a do outro é $V(0)$, correspondente ao nível 0. Estas são as mesmas tensões que idealmente esperamos encontrar em C_B antes que haja qualquer degradação devido à fuga. Os capacitores C_1 e C_2 não são componentes deliberadamente introduzidos no circuito, mas sim capacitâncias parasitas inevitáveis, presentes devido ao comprimento do condutor ligado às saídas dos inversores.

Consideremos, agora, como o amplificador sensor é operado para sentir o nível lógico de C_B e para trazer o bit de informação para o terminal de dados. Inicialmente, todas as chaves estão abertas. A primeira operação a ter lugar é o fechamento das duas chaves S_p, trazendo C_1 e C_2 à mesma tensão. Esta equalização de tensões é feita para que o flip-flop (constituído quando as chaves S_f fecharem) não tenha lembrança de sua história passada, isto é, não tenha lembrança de seu estado anterior. Vamos supor que, com S_f fechada, o flip-flop tenha permanecido em um estado com a saída do inversor I_1 em $V(1)$ e com a saída de I_2 em $V(0)$. Quando S_f abrisse o circuito, as tensões de saída seriam preservadas por algum tempo em C_1 e C_2. Se fechássemos S_f novamente logo após sua abertura e sem equalização, o flip-flop reverteria a seu estado original. Mesmo supondo que com S_f aberta procurássemos fazer com que o flip-flop se encontrasse no estado alternado quando do fechamento de S_f (através do manejo das demais chaves), a tensão preservada em C_1 e C_2 tenderia a fazer o flip-flop retornar ao estado original, dificultando sua passagem para o outro estado. Desejamos, pois, assegurar falta de polarização no flip-flop porque o amplificador sensor é usado para sentir, em rápida sucessão, os bits armazenados em um grande número de posições de armazenamento de bits na memória. Na primeira operação de equalização, C_1 e C_2 são *pré-carregados* à tensão V, donde a operação é chamada *pré-carga*. O valor de V é determinado por detalhes do circuito eletrônico com que o amplificador sensor é implementado mas que não são relevantes na atual discussão, onde estamos interessados na lógica da operação.

A próxima operação consiste no fechamento da chave S_r de modo que o capacitor *morto* C_D possa se carregar até V_{REF}. A tensão V_{REF} é selecionada nominalmente no valor $V_{REF} = [V(1) + V(0)]/2$, a meio caminho entre $V(1)$ e $V(0)$. Uma vez carregado C_D, S_r é reaberta.

A terceira operação consiste na abertura das chaves S_p e no fechamento das chaves S_a. E, finalmente, na quarta operação fechamos as chaves S_f. Vamos supor que o bit armazenado em C_B fosse um 1 lógico; então, a tensão em C_B é $V(1)$ ou um pouco menos se houve perda de carga. Supomos, todavia, que a perda de carga foi limitada e que a tensão em C_B, embora menor que $V(1)$, é confortavelmente maior que V_{REF}. Neste caso, após o fechamento das chaves S_a, a tensão em C_2 será maior que a tensão em C_1. Quando as chaves S_f fecham para formar o flip-flop, este irá para o estado em que a saída de I_2, o capacitor de armazenamento e o terminal de saída de dados irão, todos, para a tensão inteira $V(1)$. A corrente de saída que a memória pode vir a ter que suprir será fornecida pelo inversor. É importante notar que ao ler o bit em C_B, obtemos uma tensão de saída que é a tensão *total* $V(1)$, embora possa ter havido perda de carga antes da leitura. Mais importante, observe-se que o processo de leitura *restaura a tensão sobre o capacitor a seu valor pleno* $V(1)$. *Isto é, o processo de leitura de um bit reforça-o ao mesmo tempo.*

6.13 CARACTERÍSTICAS DE UMA MEMÓRIA DINÂMICA

A organização de uma memória dinâmica de dezesseis palavras e um bit/palavra é mostrada na Fig. 6.13-1. Note-se que o arranjo interno da memória é bidimensional. Uma linha ativa do decodificador de colunas fecha uma chave em uma linha de bit e ativa o amplificador sensor ligado àquela linha de bit. Os decodificadores de linha e coluna, juntos, individualizam um capacitor e provêem acesso àquela posição de armazenamento de um bit, tanto para leitura como para escrita.

Ao reforçar, não é necessário reforçar um capacitor de cada vez; todos os capacitores de uma linha podem ser reforçados ao mesmo tempo. Há uma provisão no circuito para ativar todos os amplificadores sensores e fechar simultaneamente todas as chaves apropriadas nas linhas de bits. Esta ativação simultânea reforça todos os capacitores na linha selecionada pelo decodificador de linha. Consideremos uma memória de quatro kilobits e um bit/palavra. Esta memória tem 64 linhas e 64 colunas de capacitores. Como cada linha deve ser reforçada a cada 2 ms, o intervalo permissível entre o reforço de uma linha e o reforço de outra é $2000/64 = 31\ \mu s$. Mesmo que concedamos $1\ \mu s$ para a operação de reforço (uma estimativa conservativa em um caso típico), temos o resultado que o reforço ocupa apenas 3% do tempo total de operação. De qualquer modo, quando se dá o reforço, a memória não fica disponível para outros propósitos e o sistema ao qual a memória é incorporada deve ser alertado sobre a não-disponibilidade.

Uma memória dinâmica de um kilobit, como a do tipo 1103, é fabricada em um invólucro de dezoito pinos. Há dez pinos de endereço ($2^{10} = 1024$), um pino de habilitação de circuito, um pino de leitura-escrita, pinos de entrada e saída, um pino de pré-carga e pinos para alimentação. Uma memória de quatro kilobits, como a do tipo 2104, é fabricada em um invólucro de dezesseis pinos. Como a de 1103, tem pinos para alimentação, um pino de leitura-escrita, um pino de habilitação de circuito e pinos de entrada e saída, não tem pino de pré-carga, pois aparentemente a operação de pré-carga é executada automaticamente. Doze bits de endereço são necessários ($2^{12} = 4096$), mas só há seis pinos de endereço. Os doze bits necessários são colocados na memória multiplexando os bits através dos pinos de endereço. Seis bits com endereço de linha são apresentados ao circuito e, com a ativação de um pino chamado *sincronismo do endereço de linha* (*row-address strobe*) (RAS), estes bits são transferidos para um latch de seis bits no CI. Em seguida os seis endereços de coluna são apresentados aos pinos de entrada e transferidos para um segundo latch pela ativação do CAS (*column-address strobe*).

Os comentários sobre reforço e multiplexação de endereços tiveram a intenção de indicar que uma memória dinâmica é algo mais difícil de usar que uma memória estática. Os fabricantes dispõem de vários circuitos de interface para auxiliar no uso das memórias dinâmicas.

6.14 MEMÓRIAS SÉRIE

As memórias de escrita e leitura e as memórias de leitura são memórias de acesso aleatório. O tempo de acesso a uma palavra de dados em qualquer endereço é independente do endereço da palavra acessada imediatamente antes. Um tipo diferente de memória é a *memória série*. Aqui as palavras de dados armazenadas tornam-se disponíveis para leitura, ou localizações tornam-se disponíveis para escrita em série, seguindo uma ordem seqüencial predeterminada. Se selecionarmos um endereço aleatoriamente, notaremos que o tempo de acesso depende do número de endereços interpostos entre a localização acessada no presente e a localização à qual se deseja

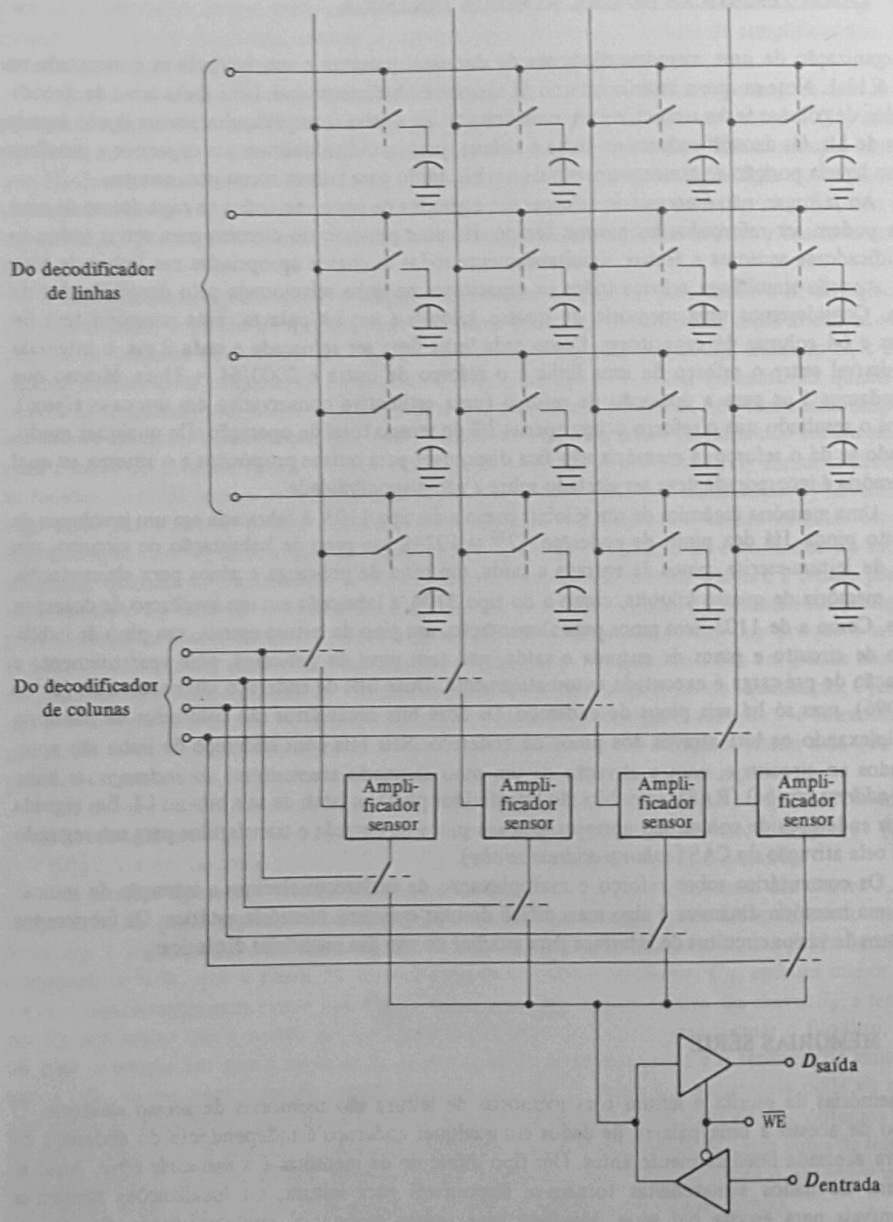


Fig. 6.13-1 A organização interna de uma memória dinâmica de dezesseis palavras e um bit/palavra.

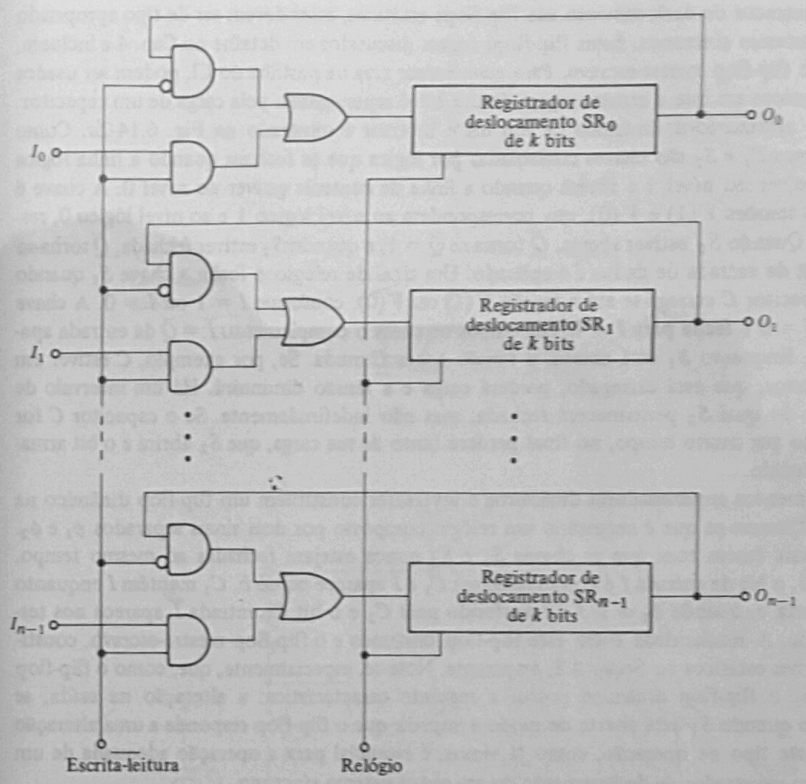


Fig. 6.14-1 Registradores de deslocamento usados para construir uma memoria série com k palavras e n bits/palavra.

acesso. Em aplicações em que as memórias série são aceitáveis, elas apresentam a vantagem de serem mais econômicas que as memórias de escrita e leitura de acesso aleatório.

Uma memória série é mostrada na Fig. 6.14-1. Existem n registradores de deslocamento e cada registrador de deslocamento é de k bits. A memória tem uma organização $k \times n$, ou seja, k palavras de n bits. Para carregar a memória, palavras de entrada são aplicadas às linhas de entrada de bits $I_0, I_1, \ldots, I_{n-1}$ em sincronismo com o relógio. Para permitir a escrita na memória, a entrada *escrita-*$\overline{\textit{leitura}}$ deve estar no nível lógico 1. Uma palavra de entrada aparecerá nos pinos de saída $O_0, O_1, \ldots, O_{n-1}$ após k ciclos de relógio. Em geral, exceto pelo atraso que ocorre enquanto a palavra é deslocada através dos registradores, as palavras de entrada aparecem na saída na mesma seqüência em que foram lidas. Uma memória deste tipo é chamada memória *first in-first out* (FIFO). Quando a memória tiver sido preenchida, os dados armazenados podem ser preservados fazendo a entrada *escrita-*$\overline{\textit{leitura}}$ igual a 0. Os bits de saída serão reintroduzidos no registrador através das entradas e as palavras *recirculação*. As palavras armazenadas estarão disponíveis uma após a outra, em seqüência, sincronamente com o relógio.

Se o registrador de deslocamento usa flip-flops estáticos, estes devem ser de tipo apropriado para uso em sistemas síncronos. Estes flip-flops foram discutidos em detalhe no Cap. 4 e incluem, entre outros, o flip-flop mestre-escravo. Para economizar área na pastilha do CI, podem ser usados flip-flops dinâmicos em que a armazenagem de um bit é representada pela carga de um capacitor. Um elemento armazenador dinâmico de um bit e inversor é mostrado na Fig. 6.14-2*a*. Como sempre as chaves S_1 e S_2 são chaves controladas por lógica que se fecham quando a linha lógica de controle estiver no nível 1 e abrem quando a linha de controle estiver no nível 0. A chave é ligada entre as tensões $V(1)$ e $V(0)$, que correspondem ao nível lógico 1 e ao nível lógico 0, respectivamente. Quando S_2 estiver aberta, $\overline{Q}$ torna-se $\overline{Q} = 1$, e quando S_2 estiver fechada, $\overline{Q}$ torna-se $\overline{Q} = 0$. Um bit de entrada de dados I é aplicado. Um sinal de relógio ϕ fecha a chave S_1 quando $\phi = 1$ e o capacitor C carrega-se até a tensão $V(1)$ ou $V(0)$, conforme $I = 1$ ou $I = 0$. A chave S_2 abre para $I = 0$ e fecha para $I = 1$. Em ambos os casos o complemento $\overline{I} = \overline{Q}$ da entrada aparece na saída. Enquanto S_1 está aberta, a tensão sobre C muda. Se, por exemplo, C estiver em $V(1)$, o capacitor, que está carregado, perderá carga e a tensão diminuirá. Há um intervalo de tensão dentro do qual S_2 permanecerá fechada, mas não indefinidamente. Se o capacitor C for deixado isolado por muito tempo, no final perderá tanto de sua carga, que S_2 abrirá e o bit armazenado será perdido.

Dois elementos armazenadores dinâmicos e inversores constituem um flip-flop dinâmico na Fig. 6.14-2*b*. Observe-se que é necessário um relógio composto por dois sinais separados ϕ_1 e ϕ_2. Estes dois sinais fazem com que as chaves S_1 e S_3 nunca estejam fechadas ao mesmo tempo. Quando $\phi_1 = 1$, o bit de entrada I é armazenado em C_1 e $\overline{I}$ aparece no nó b. C_1 mantém I enquanto S_1 estiver aberta e, quando $\phi_2 = 1$, $\overline{I}$ é transferido para C_2 e o bit de entrada $\overline{I}$ aparece nos terminais de saída. A similaridade entre este flip-flop dinâmico e o flip-flop mestre-escravo, constituído por latches estáticos na Seção 4.8, é aparente. Note-se, especialmente, que, como o flip-flop mestre-escravo, o flip-flop dinâmico possui a seguinte característica: a alteração na saída, se houver, ocorre quando S_1 está aberta de modo a impedir que o flip-flop responda a uma alteração na entrada. Este tipo de operação, como já vimos, é essencial para a operação adequada de um flip-flop em um registrador de deslocamento ou em outro sistema síncrono.

O registrador de deslocamento recirculante dinâmico de 1024 bits tipo 2405 da Intel é típico dos registradores de deslocamento dinâmicos existentes no mercado e que podem ser usados em uma memória série. A lógica incorporada ao circuito, além do registrador próprio, é mostrada na Fig. 6.14-3. Como as memórias de acesso aleatório, este circuito integrado possui, além dos terminais de entrada e saída de dados, dois terminais de seleção de circuito $\overline{CS}_x$ e $\overline{CS}_y$. Possui, ainda, um terminal $W/\overline{R}$ que aqui significa *escrita-recirculação* e corresponde a *escrita-leitura* em uma memória de acesso aleatório. O sinal de relógio de entrada é usado para acionar um gerador de relógio no próprio circuito que produz o relógio de duas fases requerido pelos flip-flops dinâmicos. Na saída existe previsão para a ligação do terminal marcado "saída" ao terminal R_L e conseqüente uso do resistor e da chave controlada por lógica como um inversor e isolador (*buffer*) de saída. A ligação pode não ser completada, possuindo, então, o circuito uma *saída com coletor aberto* que permite o uso da ligação WIRED-AND (ver Seção 3.8).

O tipo 2045 da Intel opera a uma freqüência máxima de relógio de 1 MHz. Devido ao caráter dinâmico da memória, há uma freqüência mínima permitida de aproximadamente 1 kHz. Esta velocidade mínima de relógio é necessária para permitir uma operação de deslocamento e, conseqüentemente, uma operação de reforço antes que um capacitor perca tanta carga que o nível lógico armazenado se torne ambíguo.

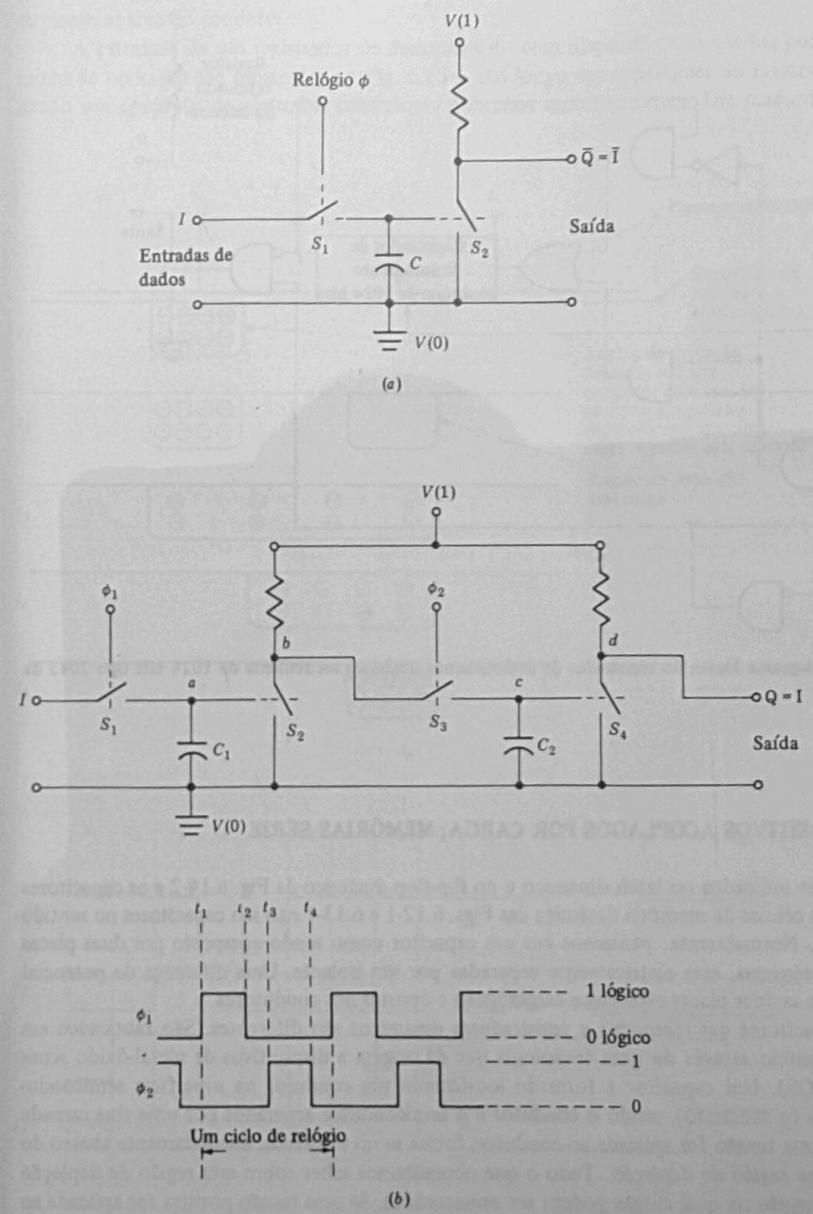


Fig. 6.14-2 (*a*) Um elemento armazenador dinâmico de um bit e inversor. (*b*) Duas unidades armazenadoras-inversoras constituem um flip-flop dinâmico.

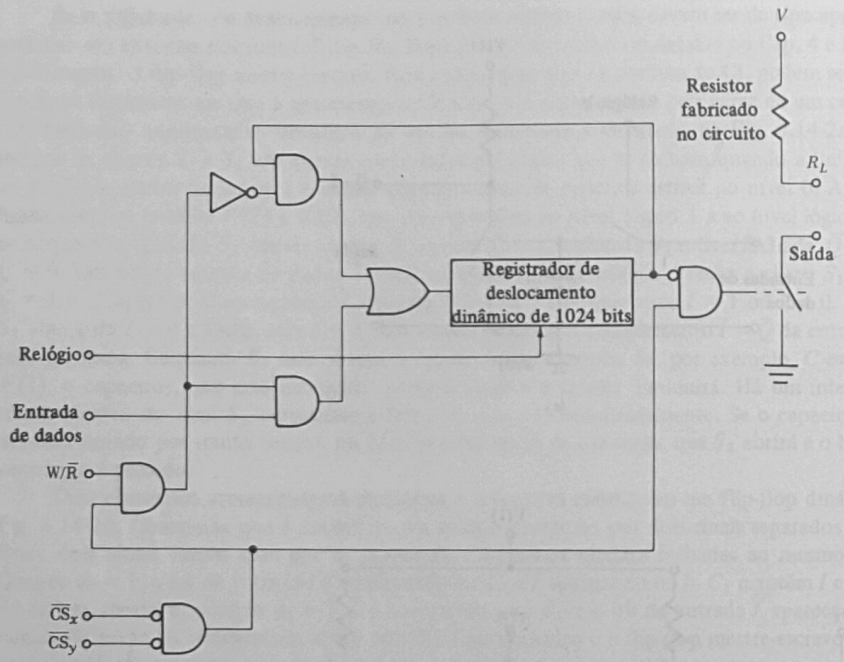


Fig. 6.14-3 Diagrama lógico do registrador de deslocamento dinâmico recirculante de 1024 bits tipo 2045 da Intel.

6.15 DISPOSITIVOS ACOPLADOS POR CARGA; MEMÓRIAS SÉRIE

Os capacitores indicados no latch dinâmico e no flip-flop dinâmico da Fig. 6.14-2 e os capacitores indicados nas células da memória dinâmica das Figs. 6.12-1 e 6.13-1 não são capacitores no sentido convencional. Normalmente, pensamos em um capacitor como sendo composto por duas placas condutoras próximas, mas eletricamente separadas por um isolante. Uma diferença de potencial aplicada entre as duas placas estabelece cargas iguais e opostas nos condutores.

Os capacitores das memórias e registradores dinâmicos são diferentes. São fabricados em pastilhas de silício através de uma tecnologia que dá origem a dispositivos de metal-óxido semicondutor (MOS). Um capacitor é formado localizando um condutor na superfície semicondutora de silício (o substrato), sendo o condutor e o semicondutor separados por uma fina camada isolante. Se uma tensão for aplicada ao condutor, forma-se no substrato, imediatamente abaixo do condutor, uma *região de depleção*. Tudo o que necessitamos saber sobre esta região de depleção é que é uma região na qual cargas podem ser armazenadas. Se uma tensão positiva for aplicada ao condutor, forma-se a região de depleção e, se injetarmos uma carga negativa na região de depleção, a carga será armazenada. Assim, as cargas que em um capacitor convencional são armazenadas

nas placas condutoras, em um dispositivo MOS são armazenadas no condutor e na região de depleção abaixo do condutor.

A estrutura de um registrador de deslocamento com dispositivos acoplados por carga e seu modo de operação são mostrados na Fig. 6.15-1. Ao longo das superfícies do substrato está localizado um conjunto de eletrodos condutores próximos uns dos outros. Um conjunto de quatro

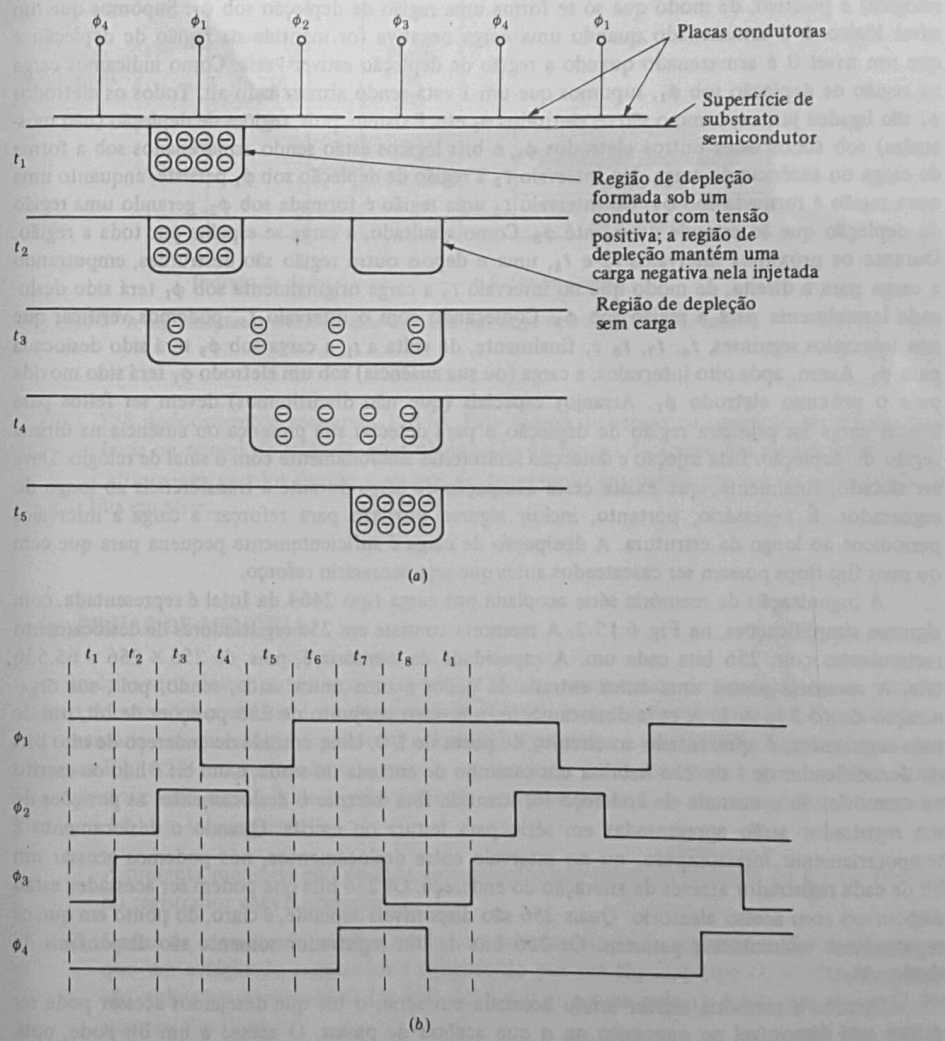


Fig. 6.15-1 (*a*) A estrutura de um registrador de deslocamento acoplado por carga mostrando a distribuição de carga nos vários instantes identificados em (*b*). (*b*) Os sinais de tensão aplicados aos eletrodos de metal.

eletrodos adjacentes acionado por quatro sinais de relógio ϕ_1, ϕ_2, ϕ_3 e ϕ_4, mostrados na Fig. 6.15-1*b*, constitui-se no mecanismo de um flip-flop dinâmico. Observe-se que os sinais ϕ_1 e ϕ_3 são idênticos em sua forma, mas deslocados no tempo um em relação ao outro. O mesmo ocorre com os sinais ϕ_2 e ϕ_4. O arranjo elaborado de quatro fases é necessário para dar ao flip-flop as características de operação de um flip-flop mestre-escravo, permitindo, assim, seu uso em um registrador de deslocamento e assegurando que os dados se movam em uma única direção.

Como pode ser visto nas Figs. 6.15-1*a* e *b*, durante o intervalo t_1 somente ϕ_1 (dos quatro relógios) é positivo, de modo que só se forma uma região de depleção sob ϕ_1. Supomos que um nível lógico 1 é armazenado quando uma carga negativa for mantida na região de depleção e que um nível 0 é armazenado quando a região de depleção estiver vazia. Como indicamos carga na região de depleção sob ϕ_1, supomos que um 1 está sendo armazenado ali. Todos os eletrodos ϕ_1 são ligados juntos, como o são os eletrodos ϕ_2 etc. Existem, pois, regiões de depleção (não mostradas) sob todos estes outros eletrodos ϕ_1, e bits lógicos estão sendo armazenados sob a forma de carga ou ausência de carga. No intervalo t_2 a região de depleção sob ϕ_1 persiste, enquanto uma nova região é formada sob ϕ_3. No intervalo t_3 uma região é formada sob ϕ_2, gerando uma região de depleção que se estende de ϕ_1 até ϕ_3. Como resultado, a carga se espalha por toda a região. Durante os próximos intervalos t_4 e t_5, uma e depois outra região são destruídas, empurrando a carga para a direita, de modo que no intervalo t_5 a carga originalmente sob ϕ_1 terá sido deslocada lateralmente para a região sob ϕ_3. Começando com o intervalo t_5, podemos verificar que nos intervalos seguintes, t_6, t_7, t_8 e, finalmente, de volta a t_1, a carga sob ϕ_3 terá sido deslocada para ϕ_1. Assim, após oito intervalos, a carga (ou sua ausência) sob um eletrodo ϕ_1 terá sido movida para o próximo eletrodo ϕ_1. Arranjos especiais (que não discutiremos) devem ser feitos para injetar carga na primeira região de depleção e para detectar sua presença ou ausência na última região de depleção. Esta injeção e detecção serão feitas sincronamente com o sinal de relógio. Deve ser notado, finalmente, que existe certa dissipação de carga durante a transferência ao longo do registrador. É necessário, portanto, incluir alguma provisão para reforçar a carga a intervalos periódicos ao longo da estrutura. A dissipação de carga é suficientemente pequena para que cem ou mais flip-flops possam ser cascateados antes que seja necessário reforço.

A organização da memória série acoplada por carga tipo 2464 da Intel é representada, com algumas simplificações, na Fig. 6.15-2. A memória consiste em 256 registradores de deslocamento recirculantes com 256 bits cada um. A capacidade da memória é, pois, de $256 \times 256 = 65.536$ bits. A memória possui uma única entrada de dados e uma única saída, sendo, pois, sua organização de 65.536×1. A cada deslocamento, um novo conjunto de 256 posições de bit, um de cada registrador, é apresentado ao circuito de porta de I/O. Uma entrada de endereço de oito bits do decodificador de 1 de 256 habilita um caminho de entrada ou saída, e um bit é lido ou escrito na memória. Se a entrada de endereço for mantida fixa durante o deslocamento, as posições de um registrador serão apresentadas em série para leitura ou escrita. Quando o deslocamento é temporariamente interrompido, ou no intervalo entre deslocamentos, nós podemos acessar um bit de cada registrador através da alteração do endereço. Os 256 bits que podem ser acessados estão disponíveis com acesso aleatório. Quais 256 são disponíveis depende, é claro, do ponto em que os registradores recirculantes pararam. Os 256 bits de um registrador somente são disponíveis de modo série.

Quando a memória estiver sendo acessada em série, o bit que desejamos acessar pode ser o que está disponível no momento ou o que acabou de passar. O acesso a um bit pode, pois, requerer nenhum deslocamento ou 256 deslocamentos. Na média, o número de deslocamentos necessários será de $256/2 = 128$ deslocamentos. Como o 2464 pode ser deslocado à razão de

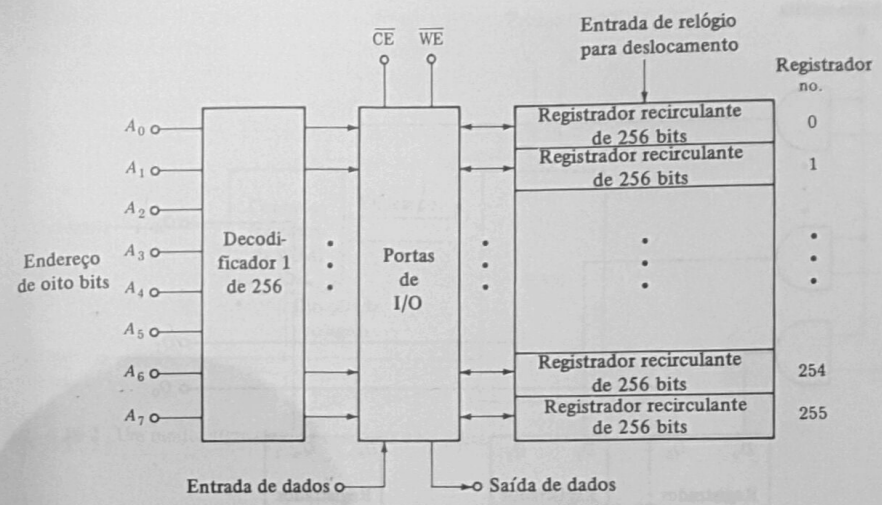


Fig. 6.15-2 A organização da memória série acoplada por carga tipo 2464 CCD da Intel.

1 MHz, 128 deslocamentos requerem 128 μs. Este tempo de acesso na operação série é denominado *latência* ou *tempo de latência*, para distingui-lo do tempo de acesso das memórias não-série, onde os atrasos são gerados devido a atrasos de propagação em portas em vez da necessidade de acessar bits em série.

6.16 PILHAS DE MEMÓRIA

As memórias série descritas nas seções anteriores têm a característica de operação que a primeira palavra a entrar na memória é a primeira a sair (FIFO); mas, freqüentemente, há necessidade de uma memória em que a última palavra a entrar seja a primeira a sair (LIFO). Tal memória é mostrada na Fig. 6.16-1. Aqui também usamos n registradores de deslocamento, cada um com k bits para construir uma memória de k palavras, cada uma com n bits. Esta memória difere da memória da Fig. 6.14-1 em três aspectos:

1. A presente memória não permite recirculação, não havendo conexão do último flip-flop do registrador para o primeiro.
2. Dados de entrada são escritos e lidos no mesmo estágio do registrador. Se considerarmos que um estágio do registrador é constituído por um flip-flop tipo D, os bits de entrada são aplicados à entrada D (dados) do flip-flop e uma palavra é lida das saídas Q dos mesmos flip-flops.
3. Os registradores de deslocamento podem ser deslocados em uma ou outra direção, dependendo do nível lógico da entrada de direção de deslocamento.

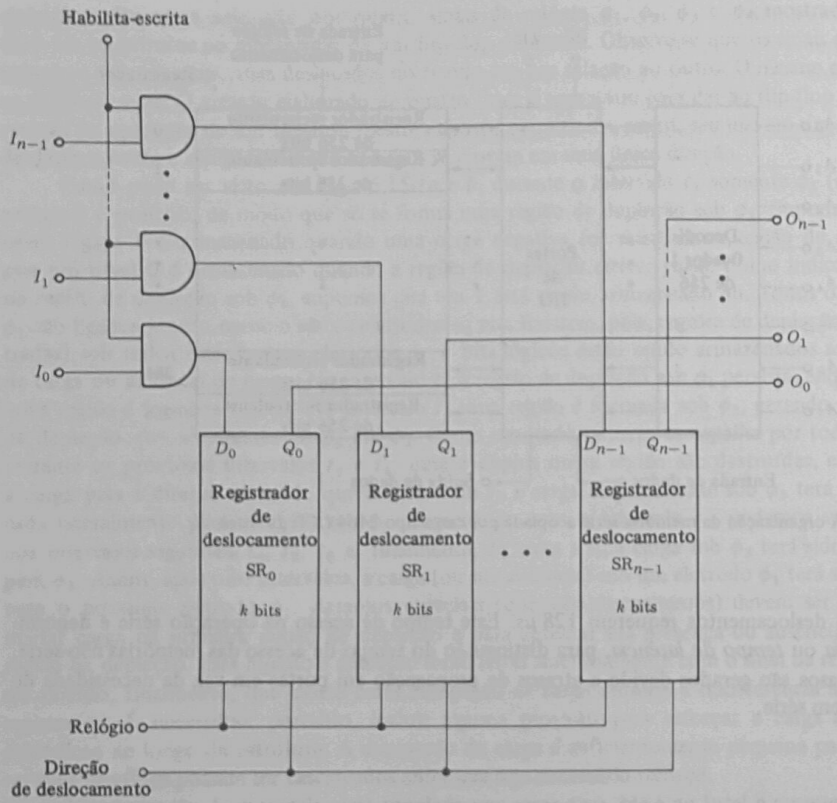


Fig. 6.16-1 Uma memória série do tipo último a entrar-primeiro a sair (LIFO), freqüentemente chamada pilha.

Vamos supor que a entrada de habilitação de escrita esteja no nível 1 e que o controle de direção de deslocamento esteja ajustado para deslocar em direção ao fim do registrador. Vamos supor, ainda, que uma seqüência de palavras seja apresentada aos terminais de entrada I_0, I_1, . . . , I_{n-1} em sincronismo com o relógio. Estas palavras serão armazenadas em localizações sucessivas na memória. Se, por outro lado, o controle de direção de deslocamento for invertido, as palavras armazenadas aparecerão nos terminais de saída O_0, O_1, . . . , O_{n-1} novamente em sincronismo com o relógio. As palavras serão lidas na ordem inversa à que foram escritas, isto é, última a ser escrita será a primeira a ser lida.

A operação da memória lembra o método usado para empilhar pratos em um restaurante, onde os pratos são empilhados um sobre o outro em uma plataforma suportada por uma mola. Quando não há pratos na plataforma, ela se nivela com o balcão. Quando um prato é colocado na plataforma, o peso faz com que ela afunde. À medida que mais pratos são empilhados, a plataforma afunda mais, deixando somente o último prato acessível. A adição de mais um prato *empurra* (push) a pilha para baixo, ao passo que a retirada de um prato permite que a pilha salte

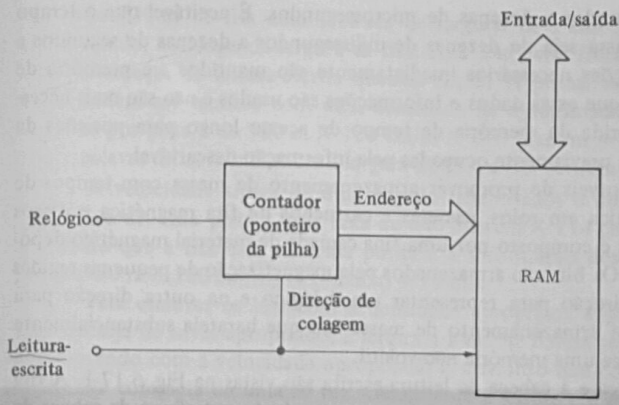


Fig. 6.16-2 Um modo alternativo de construir uma pilha.

(pop). Estas considerações explicam por que a memória série da Fig. 6.16-1 é chamada *pilha*, o processo de escrever uma palavra é chamado de *empurrar a pilha* (*push the stack*) e a leitura de uma palavra é um *salto* (*pop*). A terminologia é apropriada. Quando uma palavra é escrita na memória, todas as palavras na memória são *empurradas* um estágio para dar lugar para a nova palavra. Quando uma palavra é lida da memória, todas as palavras remanescentes saltam para cima.

Uma maneira alternativa de construir uma pilha é mostrada na Fig. 6.16-2. Aqui uma memória de acesso aleatório é acoplada a um contador que fornece o endereço da localização que é acessado. Quando o controle de leitura-escrita estiver no nível de escrita, o contador é ajustado para contar, por exemplo, no sentido crescente. A cada ciclo de relógio, uma nova posição de memória é acessada. Se palavras forem apresentadas à entrada da memória em sincronismo com o relógio, estas palavras serão escritas em posições seqüenciais na memória. Quando o controle de leitura-escrita for para leitura, a direção do contador é invertida e as palavras são lidas da memória na ordem inversa à em que foram escritas. Na memória da Fig. 6.16-2 empurrar e saltar ainda são usados para indicar uma operação de escrita e de leitura, respectivamente, embora a terminologia não seja realmente descritiva neste caso. Em uma memória LIFO, como a da Fig. 6.16-2, o contador ou outro registrador que guarda o endereço da posição de memória presentemente acessada é chamado *apontador de pilha*.

6.17 MEMÓRIA DE MASSA

Memórias de acesso aleatório semicondutoras permitem armazenamento com capacidades de até 64 kilobits. A capacidade pode ser aumentada ligando diversos CIs em paralelo. Em qualquer sistema digital com alguma sofisticação, capacidades de armazenamento da ordem de milhões de bits são muitas vezes necessárias. Obter e usar eficientemente capacidades desta ordem com memórias semicondutoras, embora não seja impossível, em geral é inconveniente.

Geralmente, porém, estes sistemas digitais de grande porte não requerem acesso rápido a toda informação armazenada na memória. O termo "acesso rápido" é usado para indicar tempos

de acesso de dezenas de nanossegundos a dezenas de microssegundos. É aceitável que o tempo de acesso de uma memória de massa seja de dezenas de milissegundos a dezenas de segundos e até de minutos. Dados e informações necessários imediatamente são mantidos em memória de tempo de acesso curto. À medida que estes dados e informações são usados e não são mais necessários, nova informação é transferida da memória de tempo de acesso longo para posições da memória de tempo de acesso curto, previamente ocupadas pela informação descartável.

Meios comercialmente disponíveis de promover armazenamento de massa com tempos de acesso longos incluem fita magnética em rolos, cassetes e cartuchos de fita magnética e discos magnéticos. Cada um destes meios é composto por uma fina camada de material magnético depositado sobre um meio de suporte. Os bits são armazenados pela magnetização de pequenas regiões no meio magnetizável em uma direção para representar um 0 lógico e na outra direção para 1 lógico. Os grandes méritos deste armazenamento de massa são que barateia substancialmente o custo por bit de memória e oferece uma memória não-volátil.

Uma porção de fita magnética e a cabeça de leitura-escrita são vistas na Fig. 6.17-1. A fita relativamente estreita move-se na direção de seu comprimento através do entreferro da cabeça de leitura-escrita. Para escrever na fita, usamos uma corrente que percorre a bobina em uma direção ou em outra, magnetizando o recobrimento magnético da fita em uma ou outra direção. Com o movimento da fita em frente à cabeça, os bits são armazenados na fita adjacentes uns aos outros ao longo do comprimento da fita. Para ler a fita fazemos com que ela passe em frente à cabeça. A cada mudança na magnetização da fita, uma tensão é induzida na bobina da cabeça, cuja polaridade depende da direção da mudança de magnetização. O mecanismo de armazenamento em disco magnético é o mesmo do armazenamento em fita. A diferença é que na fita a camada de material magnético é suportada por uma fita plástica longa e muito flexível, ao passo que no disco o meio de suporte é um disco rígido ou pouco flexível (o chamado *disco floppy*), cuja geometria é semelhante à de um disco fonográfico.

Fita Magnética

Em um caso extremo, amadores da computação doméstica usam para armazenagem em fita uma fita cassete para áudio de boa qualidade. No outro extremo, estão as fitas usadas em grandes instalações de computadores, fornecidas em grandes rolos. Tipicamente, um rolo de 26 cm desta fita

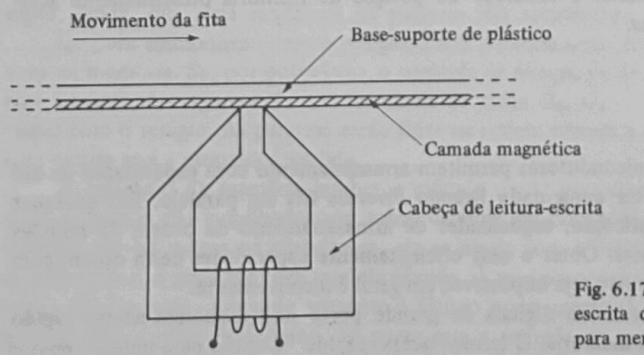


Fig. 6.17-1 A fita e a cabeça de leitura-escrita de uma unidade de fita magnética para memória de massa.

contém 730 m de fita com 1,25 cm de largura. Esta fita permite que nove bits sejam escritos lado a lado, ao longo da largura da fita. Para usar estas fitas de pistas múltiplas, é necessário usar tantas cabeças de leitura-escrita quantas forem as pistas. Ao longo do comprimento da fita, os bits podem ser escritos com uma densidade de aproximadamente 640 bits/cm. O número total de bits por pista é de 730 m × 100 cm/m × 640 bits/cm = 46 milhões. Como podemos escrever uma palavra de nove bits sobre a largura da fita, a fita contém 9 × 46 = 414 milhões de bits.

A velocidade típica de uma fita de computador é de 114 cm/s. Nesta velocidade os bits são lidos de cada pista a uma taxa de 640 bits/cm × 114 cm/s = 72 × 10^3 bits/s. No pior caso, supondo que a fita está em uma ponta e a informação desejada na outra, o tempo de latência seria (730 m × 100 cm/m)/(114 cm/s) ≈ 10 min.

Para escrever os bits com a densidade apropriada e para assegurar que a saída durante a leitura seja de nível apropriado, a leitura e a escrita devem ser feitas somente quando a fita estiver se movendo com a velocidade apropriada. É duvidoso que desejemos ou possamos ler ou escrever uma fita completa de uma só vez. Geralmente a leitura e a escrita são feitas sobre pequenas porções da fita de tempos em tempos e, quando isto ocorre, é necessário acelerar e desacelerar a fita. Devemos evitar a leitura e a escrita durante estes intervalos em que a fita não está se movendo com a velocidade apropriada. Assim, a informação é armazenada na fita em blocos de tamanho conveniente separados por espaços usados para aceleração e desaceleração.

Uma unidade de fita de um computador de grande porte, incluindo os mecanismos de acionamento da fita e de controle e todo o equipamento necessário para o interfaceamento com o computador, é uma máquina eletromecânica formidável e grande. Entre este extremo e a simples fita cassete de áudio, os fabricantes oferecem um grande número de unidades de fita intermediárias, incluindo cassetes e cartuchos digitais especiais.

Discos Magnéticos

Em memórias de discos magnéticos a informação é armazenada em trilhas concêntricas em um disco semelhante ao fonográfico. Enquanto no disco fonográfico a trilha é em espiral de fora para dentro, havendo, na realidade, somente uma trilha longa, no disco magnético cada trilha é um círculo fechado separado dos outros. Como na memória de fita, a informação é lida e escrita por uma cabeça de leitura-escrita. Por causa da geometria circular do disco, ele pode ser mantido girando constantemente mesmo quando não se processa leitura ou escrita. A perda de tempo e espaço associada com a aceleração e desaceleração é evitada. A densidade de bits ao longo da trilha varia de 1250 a 17.500 bits/cm. Uma velocidade típica de rotação é 3600 rpm.

Existe, comercialmente, uma grande variedade de memórias de disco. Há unidades com diferentes números de discos e, enquanto algumas escrevem e lêem em apenas uma das superfícies, outras usam os dois lados do disco. O número total de superfícies de escrita pode chegar a 24. Quando uma só cabeça de leitura-escrita é usada, a unidade deve conter um mecanismo para mover a cabeça de uma trilha para a outra. As trilhas estão separadas, tipicamente, por 0,25 mm e sua largura vai de 0,09 a 0,12 mm. A rotação do disco causa a presença de uma fina camada de ar que gira junto com o disco devido ao atrito viscoso entre o disco e o ar. O formato da cabeça é tal que esta camada de ar mantém a cabeça separada do disco alguns décimos de mícrons; esta separação evita o desgaste do disco e da cabeça. Outras unidades possuem diversas cabeças de leitura-escrita para diminuir o tempo perdido no movimento da cabeça.

Para acessar uma determinada parte da superfície do disco é necessário, em primeiro lugar, mover a cabeça para a trilha apropriada. O tempo requerido para esta operação é chamado tempo de *busca*. Após isto há um tempo de *latência* até que a parte apropriada da trilha chegue à cabeça. O tempo de acesso é a soma do tempo de busca e do tempo de latência e é tipicamente de 30 ms. A densidade das trilhas no disco é da ordem de oitenta trilhas/cm e a densidade de bits por trilha é de cerca de 1600 bits/cm. A capacidade de armazenamento de uma memória de disco que possua vinte superfícies é da ordem de centenas de milhões de bits.

O Disco Flexível

A memória de massa com disco magnético possui a característica de, até certo ponto, ser uma memória de acesso aleatório. Em uma memória de fita, se estivermos em uma extremidade da fita e desejarmos acesso à outra extremidade, devemos reenrolar a fita. Em uma memória de disco, se estivermos na trilha mais externa, podemos mover a cabeça diretamente para a trilha mais interna, sem parar nas trilhas intermediárias, que podem ser centenas. Assim, os tempos de acesso do disco são medidos em dezenas de milissegundos, ao passo que na fita se estendem a segundos. As unidades que descrevemos, todavia, são dispositivos mecânicos sofisticados que, além de caros, são construídos com tolerâncias extremamente pequenas. Os discos giram a alta velocidade e devem ser muito rígidos para manter o pequeno intervalo de ar entre o disco e a cabeça.

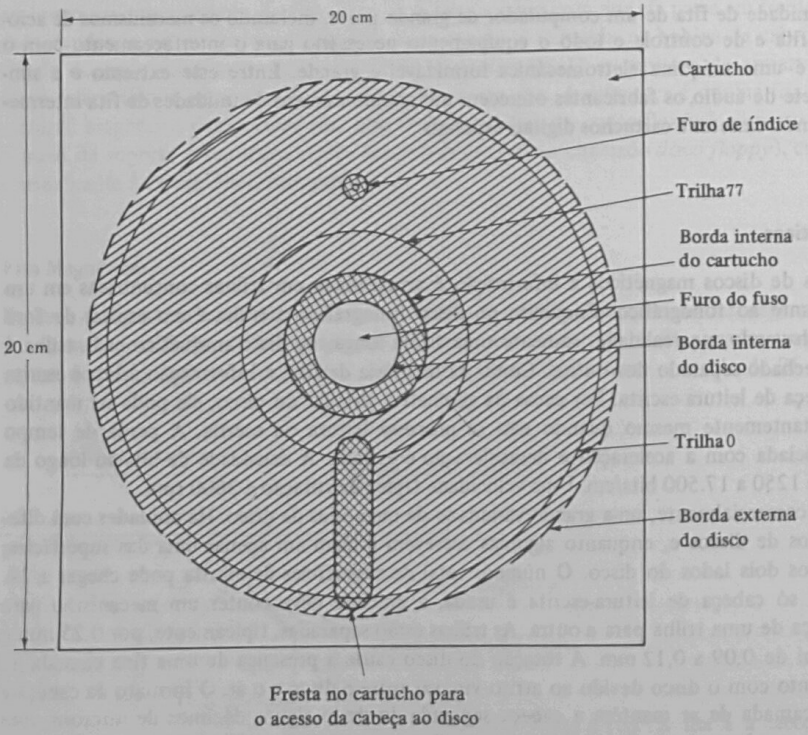


Fig. 6.17-2 Geometria de um disco flexível em seu cartucho. O disco é sombreado. Sombreado simples mostra o disco escondido pelo cartucho. Sombreado duplo mostra as partes do disco expostas pelas aberturas do cartucho.

Para satisfazer a necessidade de uma memória de disco mais barata, muitos fabricantes produzem um tipo de memória de disco conhecido como sistema de armazenamento de *disco flexível*. No sistema de disco flexível, o disco e a cabeça fazem contato como a fita e a cabeça em um sistema de fita. Conseqüentemente, o disco não precisa ser rígido e pode, inclusive, ser suficientemente fino para tornar-se ligeiramente flexível. A velocidade do disco flexível é de 300 rpm, comparada com 3600 rpm para o disco rígido. O número de trilhas é de 77, comparado com quinhentas ou mais. A densidade das trilhas é de 19/cm, comparada com 80/cm. A capacidade total de armazenamento de um disco flexível é de trinta milhões de bits por superfície, em comparação com dez vezes mais para um disco rígido. O tempo de acesso de um disco flexível é da ordem de centenas de milissegundos, comparado com dezenas de milissegundos para o sistema de disco rígido.

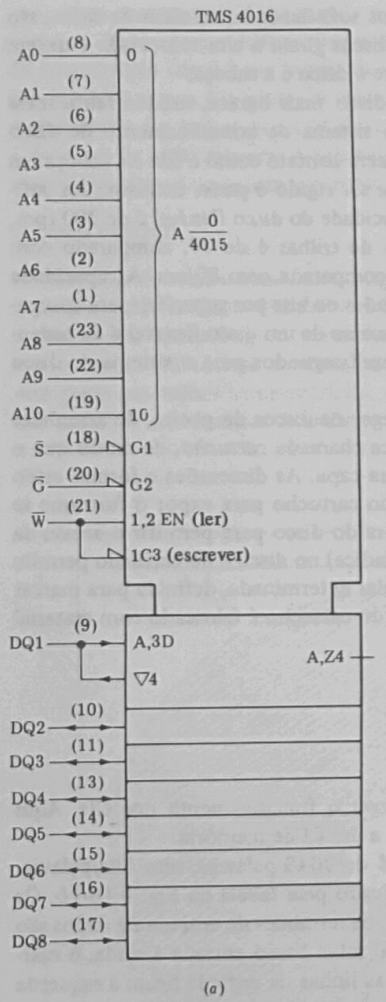
Para permitir um manuseio conveniente e para proteger os discos da poeira, de arranhões etc., eles são permanentemente selados em uma capa plástica chamada *cartucho*, de modo que o conjunto se parece com um disco fonográfico selado em sua capa. As dimensões e formas estão mostradas na Fig. 6.17-2. Existe uma abertura no centro do cartucho para expor o furo que se ajusta ao fuso. Uma fresta alongada estende-se sobre a largura do disco para permitir o acesso da cabeça única usada com discos flexíveis. Um pequeno furo (índice) no disco e no cartucho permite a passagem da luz quando o disco está em uma posição angular determinada, definida para marcar o início da trilha. O disco gira dentro do cartucho. O interior do cartucho é fabricado com material que minimiza o atrito com o disco e o limpa.

6.18 SIMBOLISMO

Como já notamos, existe um novo simbolismo para descrever o funcionamento dos CIs. Aqui consideramos um exemplo de como este simbolismo se aplica a um CI de memória.

O tipo TMS 4016 da Texas Instruments é uma RAM de 2048 palavras, oito bits/palavra. Seu símbolo funcional é mostrado na Fig. 6.18-1*a* e clarificado pela tabela na Fig. 6.18-1*b*. Os números entre parênteses são os números dos pinos. Em geral, os terminais de entrada de dados são marcados D e os terminais de saída Q. Quando um terminal serve como entrada e saída, é marcado DQ. Se os terminais de entrada e saída são separados, as linhas de entrada ficam à esquerda e as de saída à direita. Quando, como neste caso, entrada e saída devem ser combinadas, setas são usadas para indicar o fluxo de dados em ambas as direções. A seção de controle fica no topo e, abaixo, estão os oito retângulos representando as oito linhas de entrada-saída controladas. Como controle idêntico é exercido sobre as oito linhas, somente um retângulo de linha de entrada-saída é marcado com indicações explanatórias.

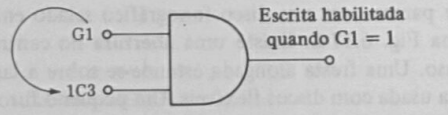
A chave com seu símbolo indica que os onze pinos incluídos por ela estabelecem um endereço e que os números das posições de endereço vão de 0 a 4015. $\overline{S}$ é um terminal de *seleção de circuito* ativo quando baixo; $\overline{G}$ é um terminal de habilitação da saída ativo quando baixo; e $\overline{W}$ é a entrada *leitura-escrita*. $\overline{W}$ é um controle de escrita ativo quando baixo e serve também como controle de leitura ativo quando alto. A operação do dispositivo é mostrada na tabela. Quando $\overline{W} = \overline{S} = L$, escrevemos os dados de entrada na memória. O nível de $\overline{G}$ não interessa, pois só



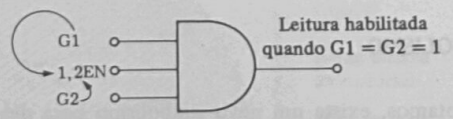
(a)

W̄	S̄	Ḡ	DQ1 – DQ8	Modo
L	L	X	Dados de entrada	Escrita
H	L	L	Dados de saída	Leitura
X	H	X	Z-alta	Dispositivo desabilitado
H	L	H	Z-alta	Dispositivo desabilitado

(b)



(c)



(d)

Fig. 6.18-1 (*a*) Símbolo da RAM TMS 4016. (*b*) Tabela verdade. (*c*) e (*d*). Explicação do simbolismo AND-dependência.

afeta a saída de dados. Quando $\overline{W}$ = H, $\overline{S}$ = L e $\overline{G}$ = L, os dados aparecerão na saída, pois $\overline{W}$ está no nível de leitura, o circuito foi selecionado e a saída está habilitada. O CI tem saída de três-estados e, quando não for selecionado ($\overline{S}$ = H), a saída irá para o estado de alta impedância (*Z*-alta) e será desligada das linhas de entrada-saída. Finalmente, quando a saída for desabilitada ($\overline{G}$ = H), mesmo que o CI seja selecionado ($\overline{S}$ = L), o dispositivo será desabilitado desde que

$\overline{W}$ esteja no nível de leitura ($\overline{W}$ = H). Vejamos como esta mesma informação é fornecida pelo simbolismo da Fig. 6.18-1*a*.

Devemos lembrar que a lógica mista é usada. Assim, quando $\overline{S}$ = L, o sinal no pino 18 é ativo e, portanto, tem nível lógico 1. De modo correspondente, o sinal interno G1 também está no nível 1. Se este sinal interno é alto ou baixo não nos interessa (felizmente, pois não conhecemos a estrutura interna do circuito). Os símbolos internos G (que não são relacionados com o símbolo externo G no pino 20) indicam uma *dependência* de AND. Efetivamente, o sinal interno G é combinado com outros sinais internos em uma porta AND. O número após o G indica qual sinal externo é controlado pelo sinal G. O número que precede a identificação de um sinal interno indica qual sinal interno exerce controle. Assim, o sinal G1 exerce controle AND sobre o sinal 1C3 do modo indicado na Fig. 6.18-1*c*. Se G1 = 0, a porta AND é desabilitada e o sinal 1C3 não é efetivo. Como indicado na Fig. 6.18-1*d*, o terminal de controle 1,2EN não pode ser efetivo a não ser que G1 e G2 estejam no nível lógico 1.

O C na identificação 1C3 significa *controle*. Como é seguido pelo número 3, o sinal 1C3, que é controlado por G1, por sua vez controla o efeito no circuito do que ocorre no terminal A,3D. Este é o terminal que (quando 1C3 = 1) aceita dados (D) e permite seu armazenamento na posição especificada pelo endereço (A).

Quando o sinal interno LEITURA estiver ativo, o dispositivo apresenta dados cuja fonte é especificada pelo endereço (A). Como já dissemos, uma linha de saída normalmente apareceria à direita. Neste caso, como há uma linha comum de entrada-saída, a linha de saída deve ser transferida. O símbolo Z representa esta transferência. Assim, A,Z4 significa que tomamos o dado que está no endereço A e o transferimos para o terminal marcado 4. O triângulo neste terminal indica que a saída é de três-estados.

Notamos a diferença no tratamento das identificações de entrada e de saída independente da necessidade de transferir a saída da direita para a esquerda. A linha interna de entrada chamada A,3D identifica sua fonte de controle. A linha de saída não identifica. A razão para esta distinção é que os dados escritos no dispositivo alteram o conteúdo da memória, enquanto os dados lidos do dispositivo não alteram o conteúdo.

CAPÍTULO 7

CIRCUITOS SEQÜENCIAIS

Os circuitos lógicos em que todas as entradas de portas são supridas por fontes externas, não havendo realimentação, isto é, nenhuma ligação de retorno das saídas das portas para as entradas das portas, constituem os *circuitos combinacionais*. Em tais circuitos, as saídas das portas são determinadas pelas entradas das portas. Se, no instante $t = t_0$, algumas das entradas das portas devem variar, então, em conseqüência, algumas das saídas das portas podem variar. Mas as novas saídas das portas dependem apenas das entradas das portas depois de $t = t_0$ e não dependem absolutamente do que as entradas das portas eram antes do instante $t = t_0$. Os circuitos combinacionais não têm memória. As *saídas atuais* dependem somente do que as *entradas atuais* são, e as saídas não dependem dos primitivos valores de entrada. Naturalmente, quando as entradas variam, há um intervalo muito breve em que as saídas não refletem ainda os valores de entrada antes da variação. Mas este breve intervalo é conseqüência simplesmente do retardo de propagação finito através das portas. Em princípio, pelo menos, os retardos de propagação das portas podem ser desprezíveis.

Outros circuitos que envolvem realimentação exibem a característica de que as saídas não dependem apenas das entradas presentes, mas, também, até certo ponto, da história das entradas no passado. Assim, as saídas presentes dependem da *seqüência* de valores lógicos na entrada que conduzem até o presente e não somente dos valores de entrada presentes. Tais circuitos lógicos são chamados circuitos lógicos *seqüenciais*. O simples latch estático constitui um exemplo de um circuito seqüencial. Portanto, eles também são contadores.

7.1 ESTADOS

Cada estágio através do qual um circuito seqüencial avança é chamado um *estado*. Em cada estado, o circuito armazena uma recordação de sua história passada de modo que ele possa saber o que fazer a seguir. Um estado é distinguido de um outro por sua recordação armazenada. Parece que, quando o tempo progride, novos itens necessitam ser acrescentados à memória e que,

conseqüentemente, uma seqüência ilimitada de novos e diferentes estados necessita se desenvolver. Revela-se geralmente, no entanto, que nem toda a história passada é relevante, que nem todos os estados através dos quais o sistema progride são diferentes uns dos outros e que o número total de estados diferentes é bastante limitado. Um exemplo servirá para ilustrar o assunto.

Suponhamos que nos foi dada a tarefa de observar uma fileira de cinco lâmpadas numeradas de 1 a 5. Estas lâmpadas estão normalmente desligadas. Uma ou outra, mas apenas uma de cada vez, lampeja brevemente de vez em quando. Estamos instruídos de que, se a lâmpada 1 cintilar, depois a lâmpada 2, e assim por diante na ordem de numeração até a lâmpada 5, devemos esperar brevemente e então soar um alarme. Estamos instruídos, ainda, de que, se a ordem de cintilação não acompanhar a ordem de numeração, devemos ignorar a cintilação até que a luz 1 cintile. Assim, na seqüência 1, 2, 3, 4, 5, fazemos soar o alarme. Se virmos a seqüência 1, 2, 4, . . . , à observação da cintilação da lâmpada 4 podemos ignorar tudo o que acontecer a seguir, porque a ordem requerida não foi ainda seguida. Podemos continuar a ignorar a cintilação até que a lâmpada 1 cintile novamente, uma vez que esta cintilação da lâmpada 1 pode ser o começo das cintilações das lâmpadas na ordem requerida.

É possível julgar facilmente quantos estados diferentes ou distintos este sistema possui imaginando que no fim de nosso turno de serviço chegue nosso substituto e precisemos dizer a ele o que é importante que ele saiba. Uma mensagem que podemos transmitir a ele é que nenhuma das lâmpadas cintilou, uma segunda mensagem possível é que a lâmpada 1 cintilou, uma terceira mensagem possível é que vimos a seqüência 1 e depois 2, e assim por diante, até a possível mensagem de que vimos a seqüência 1, 2, 3, 4, 5 e estamos prestes a fazer soar o alarme. Há um total de seis diferentes mensagens que podemos necessitar transportar. Daí, há seis lembranças e seis diferentes *estados*.

O número de diferentes seqüências de cintilações ou lampejos que devemos observar é realmente ilimitado, mas nem todas as diferentes seqüências são diferentes de maneira relevante. Suponhamos, por exemplo, que observamos a seqüência 3, 2, 4, 5, 1, 2, 5, 3, 1, 4, 2, 1, 2. Tudo o que necessitamos lembrar a respeito desta seqüência é que as duas últimas cintilações foram 1 e 2, o que é tudo o que teríamos que relatar a nosso substituto. Assim, esta seqüência coloca o sistema no mesmo estado que a seqüência simples 1, 2 colocaria. Suponhamos, novamente, que vimos uma seqüência 1, 2, 3, 4, 2, 5, 3, 2, 1, 3. Então o sistema seria deixado no mesmo estado, como se nenhuma das lâmpadas tivesse cintilado.

Podemos escolher implementar um sistema seqüencial com seres humanos, como no exemplo acima, em cujo caso dependemos da memória humana; mas, se escolhermos implementar um sistema com componentes lógicos, usaremos flip-flops como memória. Se o sistema tiver, por exemplo, seis estados de lembrança, como em nosso exemplo, necessitaremos, correspondentemente, ter seis estados de flip-flop. Para ter seis estados de flip-flop, necessitaríamos de três flip-flops. Os flip-flops forneceriam, então, oito estados, que se estendem de $Q_2Q_1Q_0 = 000$ a $Q_2Q_1Q_0 = 111$. Uma vez que temos oito estados de flip-flop e somente seis estados de lembrança, dois dos estados do flip-flop permanecerão sem utilidade.

7.2 CONTADORES COMO SISTEMAS SEQÜENCIAIS

Admitiremos, inicialmente, que nossos sistemas seqüenciais são síncronos; assim, todos os flip-flops são comandados ou excitados pela mesma forma-de-onda de sincronização. Naturalmente,

deve ser entendido que, como em todos os sistemas síncronos, os flip-flops usados devem ser do tipo que envolve o problema de disparo (ou corrida) descrito no Cap. 4. Assim, por exemplo, os flip-flops mestre-escravo devem ser adequados. Começando na Seção 7.11, consideraremos os sistemas seqüenciais que não são síncronos.

Retornemos agora ao contador síncrono já discutido no Cap. 4. Tais contadores são nitidamente sistemas seqüenciais. De fato, eles são um tanto especiais no sentido de que não há nenhuma entrada lógica apresentada ao contador e a única forma-de-onda é o relógio, que serve apenas para proporcionar sincronismo à operação dos flip-flops.

Consideremos, como um exemplo, um contador síncrono de módulo 4. Um tal contador tem quatro itens a lembrar. Se quiséssemos ler a contagem convenientemente, poderíamos arranjar o circuito contador de forma a fornecer duas saídas Z_1 e Z_0, que leriam $Z_1Z_0 = 00, 01, 10, 00$ etc., pois o contador realiza um ciclo de contagem a contagem. Sem qualquer referência a qualquer detalhe do conjunto de circuito do contador, descrevemos a operação do contador no diagrama da Fig. 7.2-1. Aqui indicamos quatro caixas que representam os quatro estados de lembrança. Dentro

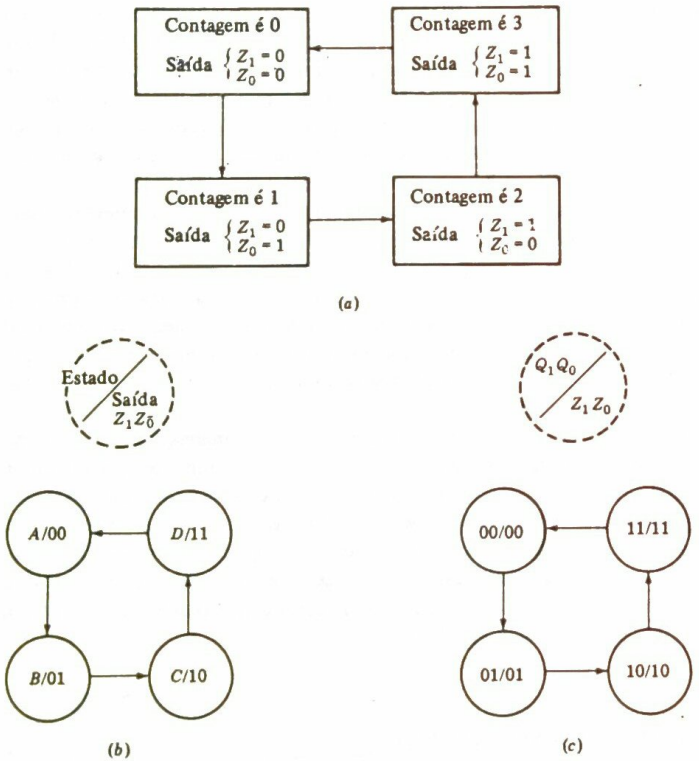


Fig. 7.2-1 O diagrama de estados de um contador de módulo 4: (*a*) estado descrito em palavras num diagrama de fluxo; (*b*) estado representado por um símbolo num diagrama de estados; (*c*) símbolo de estado substituído por uma atribuição de estado.

de cada caixa copiamos o que deve ser lembrado e também escrevemos a informação concernente a que saídas devem ser fornecidas em cada estado. As setas indicam a seqüência que o sistema deve seguir. Ele deve ir desde o estado com lembrança de que a contagem é 0 até o estado com lembrança de que a contagem é 1, e assim por diante. O diagrama da Fig. 7.2-1 dificilmente parece contribuir com alguma coisa para a discussão. Apesar disto, conforme veremos particularmente em sistemas de maior complexidade, o projeto muitas vezes é iniciado em maior eficiência pela construção de um tal diagrama de estado. Quando, como na Fig. 7.2-1*a*, as caixas que representam os estados dão a descrição detalhada do estado, o diagrama é geralmente denominado *diagrama de fluxo* (*fluxograma*).

Uma vez que compreendemos muito bem o desempenho requerido de um sistema seqüencial, podemos achar mais conveniente representar a operação do sistema como na Fig. 7.2-1*b*. Aqui os círculos representam os estados, e arbitrariamente designamos os estados por *A*, *B*, *C* e *D*. (Não há nenhuma significação particular no fato de que fomos de retângulos a círculos para representar os estados. Trata-se, simplesmente, de prática padronizada resultante, sem dúvida, do fato de que, se houver uma grande quantidade para escrever, os retângulos são mais convenientes.) A representação da Fig. 7.2-1*b* é chamada *diagrama de estado*.

Uma vez que o contador tem quatro estados, necessitaremos de dois flip-flops, FF1 e FF0. As saídas dos flip-flops são Q_1 e Q_0, e os quatro estados disponíveis dos flip-flops são $Q_1Q_0 = 00$, 0,1 10 e 11. Temos agora que fazer uma associação entre os estados de lembrança, *A*, *B*, *C* e *D*, e os estados dos flip-flops, 00, 01, 10 e 11. Assim, temos que fazer o que é denominado *atribuição de estado*. Por exemplo, estamos inicialmente em liberdade para decidir arbitrariamente que o estado *A* de lembrança deve ser representado pelo estado $Q_1Q_0 = 00$ do flip-flop, ou pelo estado $Q_1Q_0 = 01$ do flip-flop etc. Havendo selecionado um estado do flip-flop como o estado *A* de lembrança, fazemos depois uma *atribuição de estado* para o estado *B* dentre os três estados remanescentes do flip-flop, e assim por diante para os estados *C* e *D*.

Um tanto genericamente, diferentes atribuições de estado num sistema seqüencial conduzem a diferentes circuitos lógicos, alguns dos quais são melhores (mais simples ou mais econômicos em componentes). No entanto, adiaremos a discussão da seleção de atribuição de estado. No caso simples presente do contador síncrono de módulo 4, é evidente que uma atribuição de estado eminentemente razoável é

Estado de lembrança	Estado do flip-flop
A	$Q_1Q_0 = 00$
B	01
C	10
D	11

A vantagem desta atribuição é que as saídas Z_1 e Z_0 do sistema podem então ser tomadas diretamente dos terminais Q_1 e Q_0 do flip-flop. O diagrama de estado do flip-flop que usa esta atribuição de estado é dado na Fig. 7.2-1*c*, onde se vê que $Q_1 = Z_1$ e $Q_0 = Z_0$. Por conseguinte, nenhum decodificador será requerido para gerar os níveis lógicos Z_1 e Z_0 das saídas. Em cada borda de gatilho do sinal de relógio, o sistema seqüencial avançará de estado a estado na direção indicada pelas setas.

A informação nos diagramas de estado das Figs. 7.2-1*b* e *c* pode ser colocada em forma tabular, conforme mostrado nas Figs. 7.2-2*a* e *b*. Nestas tabelas, como usual, o estado presente é o

Estado presente	Saída presente Z_1Z_0	Estado seguinte
A	00	B
B	01	C
C	10	D
D	11	A

(*a*)

Estado presente $Q_1^nQ_0^n$	Saída presente Z_1Z_0	Estado seguinte $Q_1^{n+1}Q_0^{n+1}$
00	00	01
01	01	10
10	10	11
11	11	00

(*b*)

Fig. 7.2-2 Tabelas de estado dando em forma tabular a mesma informação que a fornecida pelo diagrama da Fig. 7.2-1: (*a*) tabela de estados com representação simbólica de estados; (*b*) tabela de transição, isto é, uma tabela de estados em que foi feita uma atribuição de estado.

estado no n-ésimo intervalo de relógio; após a próxima borda de gatilho do relógio, no $(n+1)$-ésimo intervalo de relógio, o sistema está em seu estado seguinte. Uma tabela de estados como na Fig. 7.2-2*b*, em que foi feita uma atribuição de estado, é chamada *tabela de transição*.

Prosseguindo ainda neste projeto do contador, devemos fazer uma seleção, novamente um tanto arbitrária, do tipo de flip-flop que nos propomos a usar – tipo *D*, *RS*, *JK*, chaveador (*T*) etc. Especificando, admitamos que usaremos flip-flops tipo *D*. Então achamos da Fig. 7.2-2*b* que, quando $Q_1Q_0 = 00$, o próximo estado é $Q_1Q_0 = 01$. Assim, Q_1 deve permanecer em $Q_1 = 0$, enquanto Q_0 deve ir para $Q_0 = 1$. Podemos certificar-nos de que tal será realmente o caso se, no estado presente, arranjarmos para que a entrada D em Q_1 seja $D_1 = 0$, enquanto $D_0 = 1$. De maneira idêntica, podemos determinar, da Fig. 7.2-2*b*, os valores de D no estado presente requeridos em todos os estados presentes. Esta informação foi introduzida nos mapas K da Fig. 7.2-3, dos quais achamos que

$$D_1 = Q_1\bar{Q}_0 + \bar{Q}_1Q_0 \qquad (7.2\text{-}1a)$$

$$D_0 = \bar{Q}_0 \qquad (7.2\text{-}1b)$$

Estas equações são chamadas *equações de excitação* dos flip-flops. Elas dizem o que as entradas, isto é, as excitações, dos flip-flops devem ser para garantir que os flip-flops *respondam* conforme requerido na borda de gatilho do relógio.

Podemos agora usar as equações de excitação para desenhar o circuito completo do contador de módulo 4. O circuito é dado na Fig. 7.2-4. Redesenhamos o circuito na Fig. 7.2-5, onde dividimos o circuito em duas partes. Uma parte consiste nos flip-flops por si próprios e é chamada

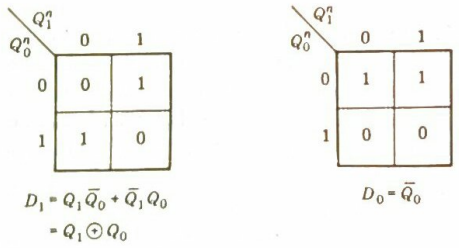


Fig. 7.2-3 Mapas K usados para determinar a equação de excitação do contador de módulo 4 usando flip-flops tipo *D*.

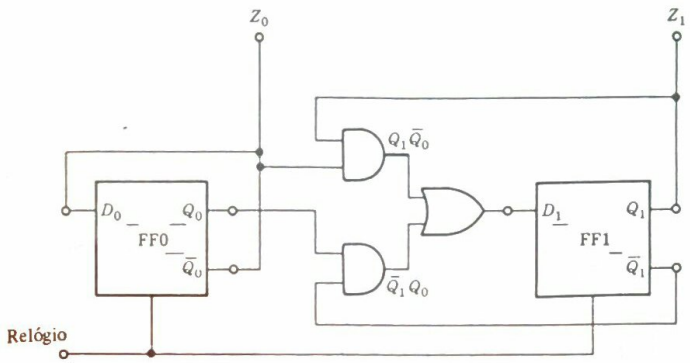


Fig. 7.2-4 O diagrama lógico do contador de módulo 4.

memória, enquanto a segunda parte consiste em portas e é chamada *lógica*. O inversor na caixa lógica e os dois círculos de inversão nas entradas das portas AND não são essenciais. Poderíamos ter evitado tanto o inversor como os círculos de inversão se tivéssemos escolhido usar a saída $\bar{Q}_1$ de FF1, conforme fizemos na Fig. 7.2-4. Preferimos não fazer esta escolha, a fim de reduzir o número de linhas associadas à memória ao mínimo essencial.

Em termos da arquitetura da Fig. 7.2-5, podemos descrever a operação seqüencial do sistema nos seguintes termos. A história passada relevante do sistema e o repositório da informação que dirige a próxima etapa do sistema são a memória, e a informação na memória é feita disponível nas saídas Q_1 e Q_0 do flip-flop. Estes níveis lógicos em Q_1 e Q_0 constituem o *estado presente* do sistema. Este estado presente é apresentado à *lógica*, que gera, por outro lado, a saída Z_1 e Z_0 apropriada a este estado. (É útil manter em mente que todo o propósito do sistema é precisamente gerar esta saída.) Por outro lado, a lógica também gera níveis lógicos que constituem o *estado seguinte* do sistema. Assim, na base do registro presentemente relevante da história passada, isto é, o estado presente, a lógica gera e apresenta, às entradas D_1 e D_0 do flip-flop, níveis lógicos que atualizam a memória. Estes níveis lógicos de entrada constituem o *estado seguinte*. Na borda de gatilho do relógio, as entradas do estado seguinte tornam-se o novo estado presente, e a lógica então também é capaz de gerar um novo estado seguinte atualizado.

Será apreciado que nossa presente discussão do contador síncrono de módulo 4 dificilmente se destina apenas à finalidade de chegarmos a um projeto de um tal contador. Já consideramos o projeto de tais contadores e mesmo de contadores mais complicados no Cap. 4. Em vez disto, nosso propósito é estabelecer uma sistematização do projeto de circuitos seqüenciais, e nosso contador serviu como um exemplo fácil que nos permitiu especificar e evitar generalizações vagas. Em qualquer caso, na base de nossa discussão, neste ponto podemos instituir um procedimento de projeto.

Etapa 1. Devemos pensar até chegar à conclusão sobre a seqüência de operações através da qual o sistema deve progredir. Neste processo de reflexão, devemos ter em mente quais saídas necessitamos que o sistema gere a cada etapa na seqüência. Devemos também ser bastante inteligentes e capazes de distinguir entre a história passada que é importante e deve ser lembrada e a história

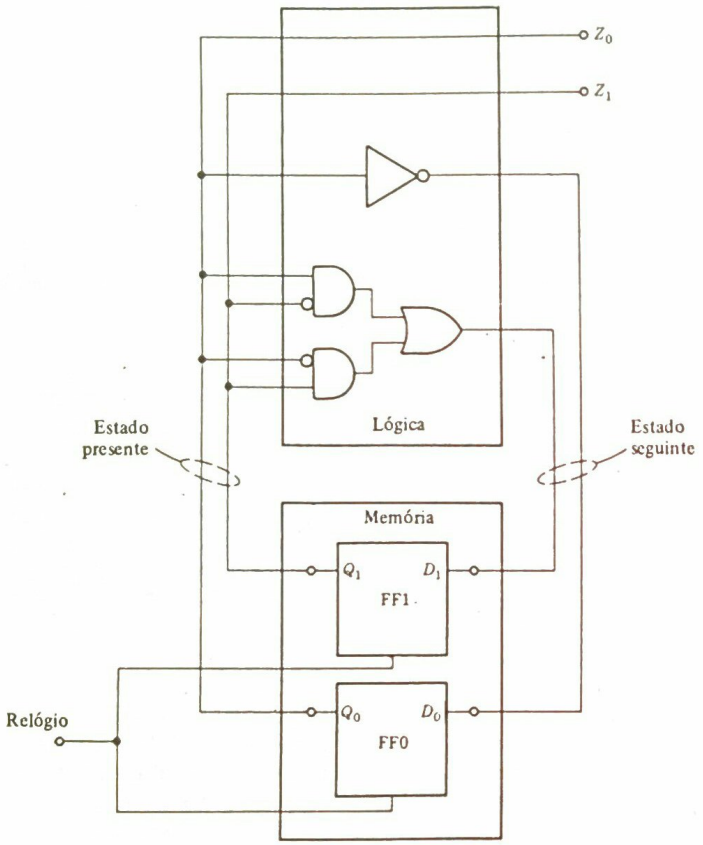


Fig. 7.2-5 O diagrama lógico da Fig. 7.2-4 redesenhado para dividi-lo numa parte lógica e numa memória.

passada que é irrelevante. Ao levar a cabo este processo, um diagrama de fluxo como aquele na Fig. 7.2-1 constitui um inestimável e muitas vezes imprescindível auxílio. No contador de módulo 4, este processo de reflexão foi um tanto trivial, mas em sistemas de alguma complexidade o processo pode ser o mais difícil. Havendo decidido quais e quantos estados são requeridos, podemos depois substituir um diagrama de fluxo, como na Fig. 7.2-1, por um *diagrama de estado de memória*, como na Fig. 7.2-1*b*.

Etapa 2. A partir do número de portas podemos determinar o número requerido de flip-flops. Um número n de flip-flops será requerido se o número de estados estiver na faixa de $2^{n-1} + 1$ a 2^n. Uma *atribuição de estado* deve ser feita associando cada estado do diagrama de estados a um estado do flip-flop. Por enquanto, admitiremos que qualquer atribuição de estado é tão boa quanto qualquer outra. Se houver mais estados de flip-flop do que estados de memória, simplesmente não usaremos alguns dos estados do flip-flop.

Etapa 3. Tendo feito uma atribuição de estado, podemos construir uma tabela de transição, como na Fig. 7.2-2*b*. A seguir devemos decidir que tipo de flip-flop nos propomos a usar (tipo *D*, *JK* etc.). Novamente admitimos aqui que a escolha é um tanto arbitrária e pode mesmo ser determinada pela disponibilidade. Havendo selecionado um tipo de flip-flop, podemos agora desenvolver equações lógicas de excitação de terminais de entrada de dados dos flip-flops a partir da tabela de estados. Os flip-flops, com dois terminais de dados, como os flip-flops *JK* ou *RS*, terão duas vezes mais equações de excitação do que os flip-flops de um simples terminal de dados como o do tipo *D* e os flip-flops chaveadores. Contudo, um tanto genericamente, onde as equações forem mais numerosas, elas serão também mais simples. Em qualquer caso, quando as equações de excitação dos flip-flops e as equações lógicas de saída tiverem sido escritas, o circuito pode ser desenhado.

7.3 UM CONTADOR DE MÓDULO 4 INCREMENTADOR-DECREMENTADOR

Como um exemplo de um circuito seqüencial ligeiramente mais complicado, consideremos um contador síncrono de módulo 4 incrementador-decrementador. Como o contador incrementador-decrementador descrito na Seção 4.25, este contador tem um terminal M de controle do modo de entrada. Quando o nível lógico aplicado a M for $M = 1$, o contador conta na direção direta 0, 1, 2, 3, 1, 2, . . . Quando $M = 0$, o contador conta na direção inversa, 0, 3, 2, 1, 0, 3, . . .

O fluxograma do contador incrementador-decrementador é mostrado na Fig. 7.3-1. Conforme anteriormente, há quatro caixas retangulares que descrevem os estados de lembrança e as correspondentes saídas requeridas. Há também os *losangos de decisão*, que descrevem como o sistema responde ao nível lógico do *controle M de entrada*. Por exemplo, se o sistema estiver no estado em que a contagem for 1 e se $M = 1$, então na próxima borda de gatilho de relógio o sistema vai para a contagem 2. Se, no entanto, $M = 0$, então, conforme indicado, o sistema vai para o estado em que a contagem é zero. Deve ser tido em mente que o sistema gastará o tempo de um ciclo de relógio em cada estado e daí em cada caixa retangular. O sistema não gasta nenhum tempo nos losangos. Os losangos servem apenas para especificar a direção que o sistema segue.

O diagrama de estados correspondente ao fluxograma da Fig. 7.3-1 é mostrado na Fig. 7.3-2*a*. Novamente, demos de modo arbitrário aos estados os nomes *A*, *B*, *C* e *D*. Num diagrama de estados não usamos losangos de decisão, e um expediente mais simples esclarece como a seqüência de estado a estado é afetada pela entrada M de controle. Por exemplo, as setas e a designação das setas esclarecem que, quando o sistema estiver no estado *A* e $M = 1$, então na próxima borda de gatilho de relógio o sistema irá à frente para o estado *B*. Por outro lado, se $M = 0$, o sistema retorna ao estado *D*. A mesmíssima informação que aparece no diagrama de estados na Fig. 7.3-2*a* está repetida em forma tabular na Fig. 7.3-2*b*. Note que agora há duas colunas de estado seguinte, uma para $M = 0$ e uma para $M = 1$. Novamente, usamos a atribuição de estado que usávamos no caso do contador unidirecional. Esta atribuição de estado conduz ao diagrama de estados na Fig. 7.3-2*c* e à tabela de estados na Fig. 7.3-2*d*. Esta tabela final de estados na Fig. 7.3-2*d*, que inclui a atribuição de estados, nos permite deduzir as equações de excitação dos flip-flops (novamente em número de dois), pelo menos tão logo que decidamos qual tipo de flip-flop nos propomos a usar. Novamente, concordemos em usar o flip-flop tipo *D*.

No caso presente, o estado seguinte do sistema depende não somente do estado presente, que está especificado pelas duas variáveis Q_1 e Q_0, mas também da entrada M de controle. Por

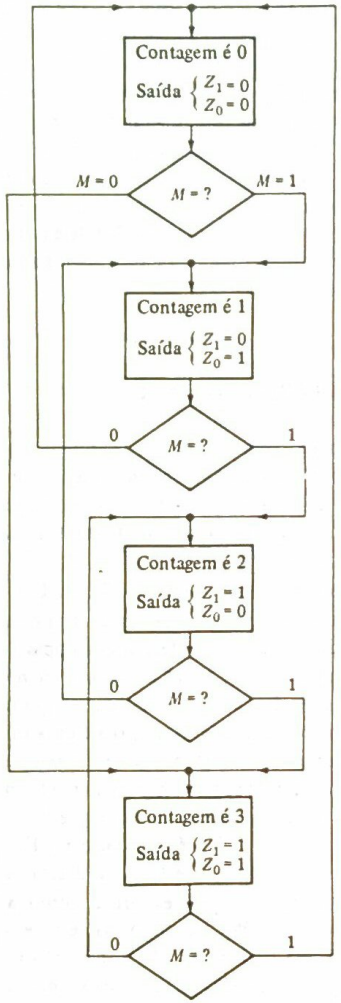
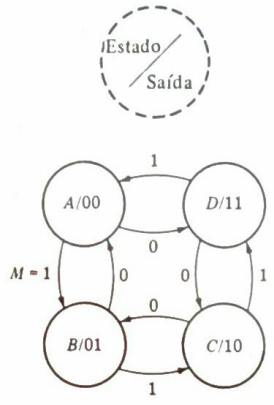
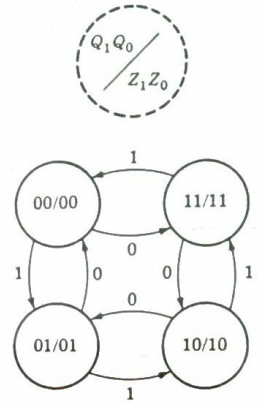


Fig. 7.3-1 Fluxograma de um contador incrementador-decrementador de módulo 4.

conseguinte, o estado seguinte é determinado pelas três variáveis Q_1, Q_0 e M. Correspondentemente, os mapas K de duas variáveis da Fig. 7.2-3 necessitam ser substituídos pelos mapas de três variáveis da Fig. 7.3-3. Vejamos como se dá uma entrada típica. Observemos da tabela da Fig. 7.3-2*d* que quando o estado presente é $Q_1^n Q_0^n = 00$ e $M = 0$, o estado seguinte requerido é Q_1^{n+1} ... $Q_0^{n+1} = 11$. Portanto, requeremos que $D_1 = 1$ e $D_0 = 1$, e daí, no quadrículo do mapa K $MQ_1Q_0 = 000$, para D_1 introduzimos 1 e também introduzimos 1 no quadrículo correspondente a D_0. As outras entradas são feitas em conseqüência de considerações idênticas. (Na tabela usamos



(*a*)



(*c*)

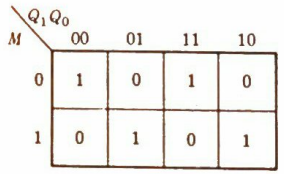
Estado presente	Saída presente Z_1Z_0	Estado seguinte	
		$M = 0$	$M = 1$
A	00	*D*	*B*
B	01	*A*	*C*
C	10	*B*	*D*
D	11	*C*	*A*

(*b*)

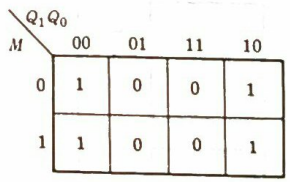
Estado presente $Q_1^n Q_0^n$	Saída presente Z_1Z_0	Estado seguinte $Q_1^{n+1} Q_0^{n+1}$	
		$M = 0$	$M = 1$
00	00	11	01
01	01	00	10
10	10	01	11
11	11	10	00

(*d*)

Fig. 7.3-2 Diagramas de estado e tabelas correspondentes ao fluxograma da Fig. 7.3-1: (*a*), (*b*) estados representados pelos símbolos; (*c*), (*d*) símbolos dos estados substituídos pelas atribuições de estado.



$D_1 = \bar{M}\bar{Q}_1\bar{Q}_0 + \bar{M}Q_1Q_0 + M\bar{Q}_1Q_0 + MQ_1\bar{Q}_0$



$D_0 = \bar{Q}_0$

Fig. 7.3-3 Mapas K usados para determinar as equações de excitação do contador incrementador-decrementador de módulo 4.

a notação Q^n e Q^{n+1} desde que explicitamente queremos distinguir estado presente de estado seguinte. Nos mapas K, os Qs representam os níveis lógicos, a qualquer tempo, nas saídas dos flip-flops. Por conseguinte, nos mapas K usamos os símbolos Q_1 e Q_0.)

Lemos dos mapas K que

$$D_0 = \bar{Q}_0 \tag{7.3-1a}$$

$$D_1 = \bar{M}\,\bar{Q}_1\bar{Q}_0 + \bar{M}\,Q_1Q_0 + M\bar{Q}_1Q_0 + MQ_1\bar{Q}_0 \tag{7.3-1b}$$

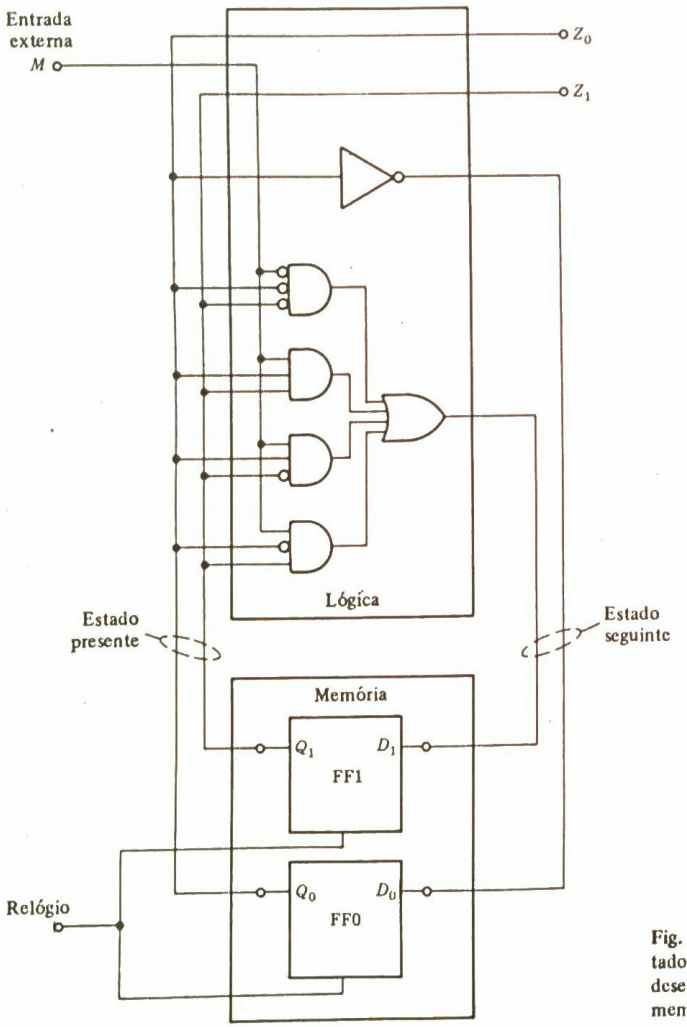


Fig. 7.3-4 O diagrama lógico do contador incrementador-decrementador desenhado para exibir uma parte da memória e uma parte da lógica.

e, naturalmente, já que rearranjamos assim:

$$Z_0 = Q_0 \tag{7.3-2a}$$

$$Z_1 = Q_1 \tag{7.3-2b}$$

O diagrama de circuito está desenhado na Fig. 7.3-4. Aqui novamente arranjamos a arquitetura num bloco lógico e num bloco de memórias. Comparando a Fig. 7.3-4 com a Fig. 7.2-4, notamos que o circuito presente é um bit mais complicado do que o circuito primitivo, como seria antecipado. O ponto essencial a notar, no entanto, é que o circuito presente tem uma entrada M externa que penetra na lógica do sistema, um aspecto que não existe no caso primitivo.

7.4 UM DETECTOR DE SEQÜÊNCIAS

Para aprofundar ainda mais a nossa análise de sistemas seqüenciais síncronos, consideremos nesta seção o projeto de um detector de seqüências. A arquitetura de nosso sistema deve ser conforme indicado na Fig. 7.4-1. Como anteriormente, o sistema consiste em um bloco lógico e em um bloco de memórias. Não sabemos de início quantos flip-flops serão necessários, mas, qualquer que seja o número deles, estes flip-flops devem ser comandados sincronamente, isto é, todos pela mesma forma-de-onda de relógio. Se os flip-flops forem do tipo D, o número de linhas n que transportam os próximos níveis lógicos de estado será igual ao número de flip-flops. Se forem usados flip-flops JK, as entradas lógicas na memória terão que ser fornecidas para a entrada J e para a entrada K de cada flip-flop. Neste caso o número n será o dobro do número de flip-flops. Se as saídas do estado presente da memória forem tomadas somente das saídas Q, o número m de tais linhas será igual ao número de flip-flops. Se forem usadas tanto as saídas Q quanto as saídas $\bar{Q}$, o número de linhas pode ser da ordem do dobro do número de flip-flops.

O sistema que nos propomos a projetar tem uma única entrada X e uma única saída Z. Como a variável M de entrada na seção precedente, a entrada X deve fazer parte da lógica que determina o estado seguinte. Postulamos de início que X deve ser uma entrada *síncrona*; assim, variações em

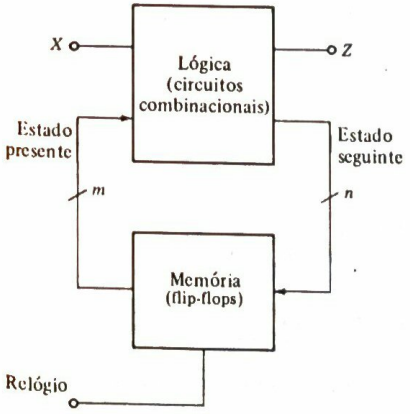


Fig. 7.4-1 A arquitetura de um sistema seqüencial síncrono.

X devem ocorrer sincronamente com a forma-de-onda de relógio. A noção de uma entrada síncrona será mais clara após um exame da situação representada na Fig. 7.4-2. Aqui desenhamos uma forma-de-onda de relógio e, admitindo que os flip-flops respondam à borda (ou transição) crescente negativa do relógio, numeramos as bordas de gatilho. Uma forma-de-onda X de entrada representativa é também mostrada. No intervalo de relógio que termina na borda 1 de gatilho temos $X = 0$. Por conseguinte, retornando ao sistema da Fig. 7.4-1, se se revela que o sistema necessita lembrar do valor X neste intervalo, ele se lembrará de que $X = 0$ e este bit de informação será armazenado na memória, isto é, provocará alguma variação no estado do flip-flop, no instante da borda 1 de gatilho do relógio. Uma vez que, conforme vimos, os flip-flops têm tempos de *manutenção* finitos (ver Seção 4.15), será necessário que o valor $X = 0$ persista, como indicado, ligeiramente além da borda de gatilho do relógio.

No ciclo de relógio que termina na borda 2 temos $X = 1$ e, no ciclo que termina na borda 3, temos $X = 0$. Nos próximos *dois* ciclos de relógio, X persiste em $X = 1$; daí a entrada ser lida não como $X = 1$ mas como $X = 11$. Depois disto a entrada X vai para $X = 0$ durante três ciclos e é lida $X = 000$. Totalmente, no tempo mostrado, a entrada X é lida $X = 010110001$. Esta leitura resulta porque a entrada está sendo sincronizada contra o relógio, isto é, está sendo lida e interpretada numa base síncrona. Se não houvesse nenhum relógio contra o qual X devesse estar sincronizada, leríamos simplesmente X como $X = 010101$, não levando em conta a duração do tempo dos níveis lógicos individuais.

Um tanto genericamente, quando entradas síncronas forem desenhadas sobre a mesma escala de tempos, como a forma-de-onda de relógio, o tempo de manutenção é ignorado e a forma-de-onda de entrada está desenhada como na Fig. 7.4-2*c*. Neste caso, a transição de relógio e a transição X de entrada (se houver) parecem ocorrer precisamente ao mesmo tempo, e pode parecer que há uma certa ambigüidade em torno do valor de X na borda de gatilho de relógio. Porém, conforme temos observado, o que se pretende é que o valor de X a ser lido seja o valor de X imediatamente *antes* da variação aparentemente coincidente.

Finalmente, admitamos, por simplificação, que os flip-flops sejam gatilhados pela borda (ver Seção 4.13) e consideremos agora que a entrada X apareça como na Fig. 7.4-2*d*. Tenhamos

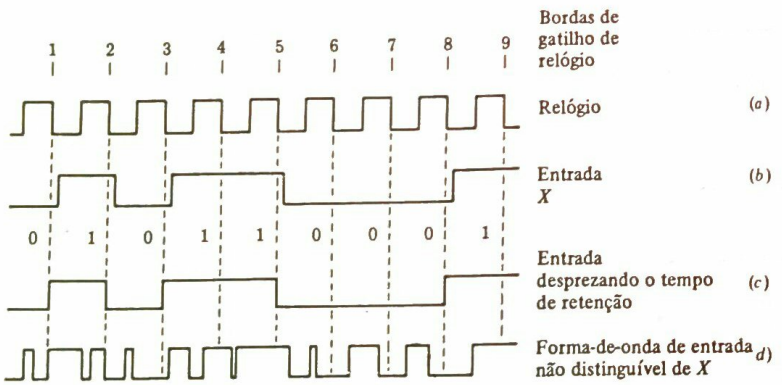


Fig. 7.4-2 Formas-de-onda para ilustrar a idéia de uma entrada síncrona num sistema seqüencial sincronizado (clocado).

em mente que os flip-flops gatilhados pela borda respondem somente à entrada de dados presente imediatamente precedente à borda de gatilho de relógio. Neste caso, como pode ser verificado, o sistema da Fig. 7.4-1 não será capaz de distinguir a forma-de-onda de entrada na Fig. 7.4-1*d* da forma-de-onda de entrada nas Figs. 7.4-1*c* ou *b*. Este aspecto é novamente o resultado do fato de que o sistema lê a entrada numa base síncrona contra o padrão do relógio.

Havendo empregado estes preliminares concernentes ao significado de uma entrada síncrona, formulemos um problema simples no projeto de um sistema seqüencial que envolva uma entrada e uma saída, como na Fig. 7.4-1. O problema consiste em projetar o sistema de modo que Z será $Z = 1$ quando e somente quando X for $X = 1$ durante *três ou mais* intervalos de relógio consecutivos. Para que nos certifiquemos de que não há ambigüidade na especificação do problema, indicamos na Fig. 7.4-3 uma possível seqüência de bits de entrada e a saída resultante. Admitiremos, conforme temos estado supondo todo o tempo, que o nível Z lógico de saída depende apenas do estado da máquina; assim, a saída está conectada (possivelmente através de portas) somente às saídas dos flip-flops. Não há nenhuma ligação direta entre X e Z. Daí, quando o sistema estiver num estado fixo (como está durante um ciclo inteiro de relógio), as variações em X não afetarão a saída Z de nenhuma maneira. Naturalmente, o valor de X num ciclo de relógio pode afetar o estado no ciclo seguinte, e depois Z pode ser afetado pela mudança de estado.

Com referência agora à Fig. 7.4-3, consideremos que o ciclo de relógio que designamos por 1 é o primeiro ciclo disponível para nosso exame. (A máquina pode ter estado operando durante algum tempo antes que conseguíssemos observá-la.) Neste primeiro intervalo não somos capazes de indicar o que é Z, porque o valor de Z depende dos valores de X nos três intervalos precedentes. No ciclo 2, no entanto, podemos dizer que $Z = 0$ porque vemos que não é possível que X tenha sido $X = 1$ nos três ciclos que precederam o ciclo 2. Se X tivesse sido $X = 1$ no ciclo 1, não teríamos sido capazes de determinar Z no ciclo 2. Prosseguindo ao longo da seqüência, achamos que $Z = 1$ no ciclo 8 porque $X = 1$ nos ciclos 5, 6 e 7. Similarmente, $Z = 1$ no ciclo 12 porque $X = 1$ nos ciclos 9, 10 e 11. Além disto, $X = 1$ nos ciclos 13 e 14 porque em ambos os casos X tinha sido 1 durante mais do que três ciclos precedentes. No ciclo 14, $Z = 1$, independentemente do valor de X neste ciclo.

Voltemos agora ao projeto do sistema. Notamos no começo que no caso geral, especificamente onde o projeto é de alguma complexidade, o ponto de partida é um fluxograma. Nos casos mais simples, como no caso presente, podemos começar pelo diagrama de estados.

Ao começar um diagrama de estados, é necessário partir de um estado, denominá-lo estado A, definido de tal maneira que, conforme o desenvolvimento do diagrama de estados se desdobre, nunca teremos que procurar informação adicional a respeito da seqüência que leva até o estado A. No caso presente, portanto, definimos o estado A como sendo o estado em que o sistema

Ciclo de relógio	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$X =$	0	1	1	0	1	1	1	0	1	1	1	1	1	
$Z =$	?	0	0	0	0	0	0	1	0	0	0	1	1	1

Fig. 7.4-3 Uma seqüência de entrada típica de valor de X e a correspondente saída Z requerida ao sistema seqüencial sob projeto.

achar-se-á num ciclo de relógio se no ciclo que o precede imediatamente X foi $X = 0$. Neste caso, também, no estado A a saída é $Z = 0$, conforme indicado na Fig. 7.4-4. Suponhamos, por outro lado, que tivéssemos definido A como um estado alcançado, por exemplo, no ciclo k de relógio, depois que X foi $X = 1$ no ciclo $k-1$ de relógio. Então não conheceríamos Z no ciclo k, nem seríamos capazes de continuar a desenvolver o diagrama de estados. Necessitaríamos conhecer o valor de X nos ciclos $k-2$ e $k-3$.

O estado A que selecionamos como um ponto de partida é adequado, mas ele não é o único ponto de partida adequado. Seria aceitável definir o estado A como um estado atingido após três ou mais ciclos sucessivos em que $X = 1$.

No ciclo em que o sistema está no estado A, teremos $X = 0$ ou $X = 1$. Se $X = 0$, o sistema necessita ir para um estado diferente? A resposta é não. No estado A o sistema deve simplesmente armazenar a informação que não ocorreu com $X = 1$ em nenhum dos ciclos imediatamente precedentes. Se então X deve ser $X = 0$, novamente a informação que necessita ser armazenada não se modificou. Por conseguinte, se o sistema estiver no estado A, um $X = 0$ deixa o sistema no estado A.

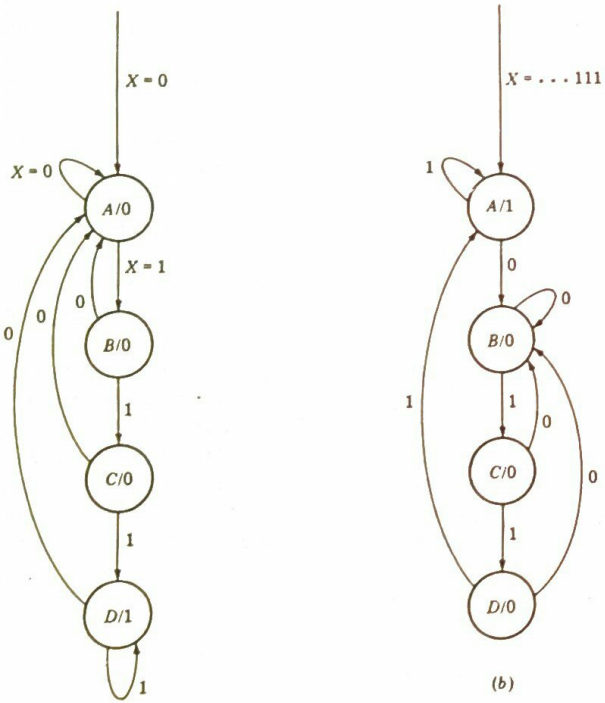


Fig. 7.4-4 Diagrama de estado do detector de seqüências. (*a*) $X = 0$ indica que A está no estado em que o sistema achar-se-á num ciclo de relógio se X for $X = 0$ no ciclo de relógio imediatamente precedente. (*b*) $X = \ldots 111$ indica que A está no estado em que o sistema achar-se-á se X for $X = 1$ nos três ou mais ciclos de relógio imediatamente precedentes.

Estado presente	Saída presente	Estado seguinte	
	Z	$X = 0$	$X = 1$
A	0	A	B
B	0	A	C
C	0	A	D
D	1	A	D

(*a*)

Estado	Q_1Q_0
A	00
B	01
C	10
D	11

(*b*)

Estado presente $Q_1^nQ_0^n$	Saída presente Z	Estado seguinte $Q_1^{n+1}Q_0^{n+1}$ $X = 0$	$X = 1$
00	0	00	01
01	0	00	10
10	0	00	11
11	1	00	11

(*c*)

$Q^n \to Q^{n+1}$	J	K
0 → 0	0	×
0 → 1	1	×
1 → 0	×	1
1 → 1	×	0

(*d*)

X \ Q_1Q_0	00	01	11	10
0	0	0	×	×
1	0	1	×	×

$J_1 = XQ_0$

X \ Q_1Q_0	00	01	11	10
0	×	×	1	1
1	×	×	0	0

$K_1 = \overline{X}$

(*e*)

X \ Q_1Q_0	00	01	11	10
0	0	×	×	0
1	1	×	×	1

$J_0 = X$

X \ Q_1Q_0	00	01	11	10
0	×	1	1	×
1	×	1	0	×

$K_0 = \overline{X} + \overline{Q}_1 = \overline{XQ}_1$

Q_1 \ Q_0	0	1
0	0	0
1	0	1

$Z = Q_1Q_0$

(*f*)

Fig. 7.4-5 Conclusão do projeto do detector de seqüências: (*a*) tabela de estados; (*b*) atribuição de estado arbitrária; (*c*) tabela de estados usando atribuição de estado; (*d*) tabelas verdade dos flip-flops JK; (*e*) mapas K dos Js e Ks dos flip-flops; e (*f*) mapa K da saída.

Este resultado está representado na Fig. 7.4-4 pela seta marcada 0, que se origina e também termina no estado A. Se, quando o sistema estiver em A, um $X = 1$ deve ocorrer, então, naturalmente, ele necessita ser lembrado (memorizado) e, portanto, como indicado, em $X = 1$, o sistema vai do estado A para o estado B. No estado B a saída é novamente $Z = 0$ porque somente um $X = 1$ ocorreu. Continuando, um segundo $X = 1$ leva o sistema de B para C, e um terceiro $X = 1$ leva o sistema para o estado D, onde $Z = 1$. Uma vez que a especificação do problema não faz nenhuma distinção entre três 1s sucessivos e mais do que três 1s sucessivos, o quarto e outros 1s sucessivos simplesmente deixam o sistema no estado D. Se no estado B, C ou D um $X = 0$ deve ocorrer, o sistema necessita apenas lembrar que a seqüência de 1s consecutivos foi interrompida, e esta é precisamente a memória armazenada no estado A.

Sempre que, como no presente, houver uma única variável X de entrada, cada estado no diagrama de estados deve ter duas setas que saem dele, uma vez que uma seta corresponde a $X = 0$ e a outra a $X = 1$. Uma seta conta como saindo mesmo se ela retornar ao mesmo estado. Se houver duas variáveis de entrada, X_1 e X_2, temos quatro possibilidades $X_1X_2 = 00, 01, 10, 11$, e um diagrama de estados não seria completo até que tivéssemos indicado quatro setas saindo de cada estado.

Na Fig. 7.4-4*b* desenhamos um diagrama de estado partindo com um estado alcançado após X ter sido $X = 1$ durante três ou mais ciclos sucessivos. É deixado como exercício ao estudante verificar se o diagrama é válido. Há outros pontos de partida igualmente válidos (ver Prob. 7.4-1).

O restante do projeto é executado na Fig. 7.4-5. Na Fig. 7.4-5*a* transformamos o diagrama de estados da Fig. 7.4-4*a* numa tabela de estados. Na Fig. 7.4-5*b* fizemos uma atribuição de estados arbitrária. Na Fig. 7.4-5*c* o diagrama de estados foi repetido usando a atribuição de estados da Fig. 7.4-5*b*. Neste ponto decidimos, por causa da variedade, usar flip-flops *JK*. Na Fig. 7.4-5*d* consideramos a tabela (apresentada anteriormente; ver Tab. 4.24-4) que especifica quais níveis lógicos são requeridos para J e K em qualquer transição do flip-flop. A seguir, na Fig. 7.4-5*e* usamos mapas K para deduzir as equações de excitação de J_1, K_1, J_0 e K_0, as quatro entradas de dados nos dois flip-flops. Uma vez que o estado seguinte depende do estado presente (Q_1 e Q_0) e também da entrada X, um mapa de três variáveis é exigido. Como um exemplo de como são feitas as entradas do mapa K, consideremos dizer que $Q_1^nQ_0^n = 00$ enquanto $X = 1$; depois, no estado seguinte, devemos ter $Q_1^{n+1}Q_0^{n+1} = 01$. Portanto, em Q_1 necessitamos de uma *transição*

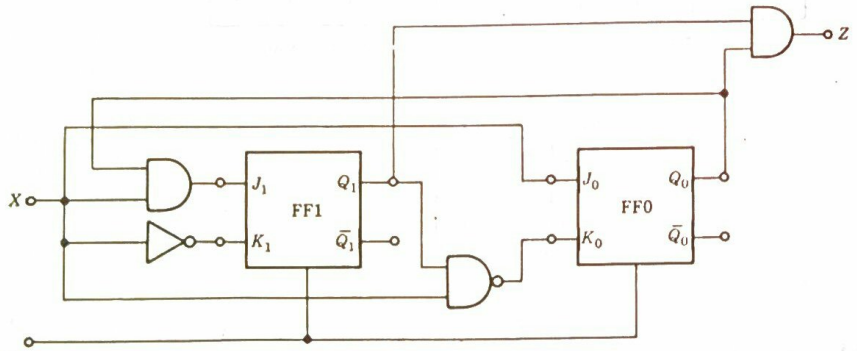


Fig. 7.4-6 O circuito lógico do detector de seqüências.

$0 \rightarrow 0$, que requer que $J_1 = 0$ e $K_1 = X$, enquanto em Q_0 necessitamos da transição $0 \rightarrow 1$, que requer $J_0 = 1$ e $K_0 = 0$. A saída Z depende apenas do estado Q_1Q_0, e, por conseguinte, Z é lido de um mapa K de duas variáveis. Achamos em conjunto que

$$J_1 = XQ_0 \qquad K_1 = \bar{X} \tag{7.4-1a}$$

$$J_0 = X \qquad K_0 = \bar{X} + \bar{Q}_1 = \overline{XQ_1} \tag{7.4-1b}$$

$$Z = Q_1Q_0 \tag{7.4-1c}$$

O circuito está desenhado na Fig. 7.4-6, que não faz nenhuma tentativa para indicar uma separação em memória e lógica.

7.5 CIRCUITOS MOORE E MEALY

Nos exemplos das seções anteriores, providenciamos para que as saídas fossem funções apenas do estado do sistema. Com um tal arranjo, o valor de uma entrada no ciclo k de relógio não pode afetar a saída até, no mais próximo, ciclo $k + 1$. As mudanças na entrada devem primeiro ter uma influência no estado do sistema antes que possam afetar a saída. Pareceria então que algumas economias poderiam ser efetuadas se arranjássemos para que as saídas fossem funções diretas não somente do estado mas também das entradas. Um circuito em que as saídas são funções apenas do estado é chamado *circuito Moore*. Um circuito em que as saídas são funções tanto do estado quanto das entradas é chamado *circuito Mealy*. Na Seção 7.4 projetamos o detector de seqüências como um circuito Mealy. O problema pode agora ser exposto sob outra forma da seguinte maneira: projetar um circuito que no intervalo k de relógio forneça uma saída $Z = 1$ sempre que, levando em conta o valor de X no intervalo k, houver uma seqüência de três ou mais valores sucessivos $X = 1$.

O diagrama de estados do circuito Mealy é dado na Fig. 7.5-1, e deve ser comparado ao diagrama da Fig. 7.4-4. Em ambos os casos, o estado A é um estado atingido após X ter sido $X = 0$, de modo que qualquer seqüência de 1s de entrada tenha sido interrompida e Z não possa

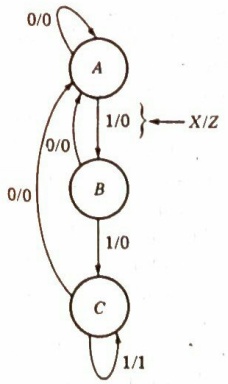


Fig. 7.5-1 O diagrama de estados do detector de seqüências projetado como um circuito Mealy.

ser $Z = 1$ novamente até que uma passagem de pelo menos três 1s sucessivos ocorra. Notar que no sistema Moore a saída Z é introduzida no círculo que representa o estado. Uma tal entrada pode ser feita porque Z depende apenas do estado. O simbolismo no sistema Mealy é diferente. Aqui, partindo do estado A, quando $X = 1$, vamos (como no caso Moore) a um novo estado B, uma vez que, como compreendemos, o sistema deve lembrar que ocorreu uma primeira etapa em direção a uma seqüência bem-sucedida. A seta que vai do estado A ao estado B tem associada a ela a notação 1/0. A entrada à esquerda da linha inclinada significa $X = 1$, e a entrada à direita da linha inclinada significa $Z = 0$. O simbolismo global deve ser lido como se segue: "Suponhamos que, no intervalo k de relógio, o sistema esteja no estado A, e suponhamos que, neste mesmo intervalo k, X tenha o valor $X = 1$. Depois, novamente, neste mesmo intervalo k, teremos $Z = 0$ e, após a próxima borda de gatilho de relógio, quando estivermos no intervalo $k + 1$, o sistema estará no estado B". Se indagarmos agora a respeito do valor de Z quando, no intervalo $k + 1$, o sistema estiver no estado B, podemos não ter nenhuma resposta até que especifiquemos o valor de X no intervalo $k + 1$. Podemos, se quisermos, achar a informação voltando a examinar o diagrama, onde observamos que no estado B tanto $X = 0$ quanto $X = 1$ produzem $Z = 0$.

O estado C na Fig. 7.5-1 é um estado em que é lembrado que houve dois 1s sucessivos. Por conseguinte, no intervalo daquele mesmo estado, um $X = 1$ será reconhecido como um terceiro 1, e a saída será $Z = 1$. Observemos, agora, que o sistema Mealy tem apenas três estados, enquanto o sistema Moore tem quatro estados. Naturalmente, em ambos os casos requereremos dois flip-flops, mas no caso mais geral está claro que uma redução no número de estados pode reduzir o número requerido de flip-flops e a complexidade geral do circuito.

O projeto do detector de seqüência Mealy é executado na Fig. 7.5-2. Na Fig. 7.5-2*a* a informação no diagrama de estados da Fig. 7.5-1 foi transferida para uma tabela de estados. Notar a diferença entre esta tabela e a tabela Moore da Fig. 7.4-4*a*. Na Fig. 7.5-2*b* foi feita uma atribuição de estados arbitrária. Somente três dos quatro estados do flip-flop foram usados, e não usamos o estado $Q_1Q_0 = 11$ do flip-flop. Na Fig. 7.5-2*c* introduzimos a atribuição de estado da Fig. 7.5-2*b* na tabela de estados da Fig. 7.5-2*a*. Tendo decidido usar novamente os flip-flops JK, consideramos outra vez na Fig. 7.5-2*d* os requisitos gerais de excitação dos flip-flops JK. Os mapas K de J_1, K_1, J_0 e K_0 são dados na Fig. 7.5-2*e*. Um mapa K de Z aparece na Fig. 7.5-2*f*, onde um mapa de três variáveis é requerido porque Z depende de X, bem como do estado Q_1Q_0. O diagrama de circuito está desenhado na Fig. 7.5-3; desta vez ele é uma forma de memória lógica. Notar especialmente que Z é a saída de uma porta da qual uma das entradas é X.

Mais exatamente antecipamos que uma versão Mealy de um circuito seqüencial será mais econômica de componentes físicos (*hardware*) do que uma versão Moore. Tal é geralmente (embora nem sempre) o caso. Em algumas situações, como em nosso detector de seqüências, a diferença dificilmente parece importante. Ainda podemos duvidar se há ou não algum detalhe no projeto de um sistema como um circuito Moore. A opinião geral é sim. Uma razão é que os estados extras de um sistema Moore e a independência das saídas desde a entrada tornam mais fácil acompanhar a operação de um sistema quando ele caminha através de seus estados e, por conseguinte, torna mais fácil a depuração (eliminação de falhas). Uma segunda razão é que num sistema Moore a entrada necessita ser lida apenas num instante imediatamente antes da ocorrência de uma borda de gatilho de relógio. Valores incorretos de entrada em outros instantes não terão um efeito sobre a saída. Estes valores de entrada incorretos podem resultar quer de alguma limitação da velocidade da entrada, quer de alguma perturbação (ruído) espúria.

Nas Figs. 7.5-4 e 7.5-5, em proveito da integridade e para introduzir algum simbolismo adicional, desenhamos fluxogramas para os detectores de seqüência Moore e Mealy. No diagrama

Estado presente	Estado seguinte/Saída presente para Entrada presente	
	$X = 0$	$X = 1$
A	$A/0$	$B/0$
B	$A/0$	$C/0$
C	$A/0$	$C/1$

(*a*)

Estado	Q_1Q_0
A	00
B	01
C	10

(*b*)

PS $Q_1^nQ_0^n$	NS/Z	
	$X = 0$	$X = 1$
00	00/0	01/0
01	00/0	10/0
10	00/0	10/1

(*c*)

$Q^n \to Q^{n+1}$	J	K
$0 \to 0$	0	×
$0 \to 1$	1	×
$1 \to 0$	×	1
$1 \to 1$	×	0

(*d*)

X \ Q_1Q_0	00	01	11	10
0	0	0	×	×
1	0	1	×	×

$J_1 = XQ_0$

X \ Q_1Q_0	00	01	11	10
0	×	×	×	1
1	×	×	×	0

$K_1 = \bar{X}$

(*e*)

X \ Q_1Q_0	00	01	11	10
0	0	×	×	0
1	1	×	×	0

$J_0 = X\bar{Q}_1$

X \ Q_1Q_0	00	01	11	10
0	×	1	×	×
1	×	1	×	×

$K_0 = 1$

X \ Q_1Q_0	00	01	11	10
0	0	0	×	0
1	0	0	×	1

$Z = XQ_1$

(*f*)

Fig. 7.5-2 Conclusão do projeto de um detector de seqüências de circuito Mealy: (*a*) tabela de estados; (*b*) atribuição de estados arbitrária; (*c*) tabela de estados usando atribuição de estados; (*d*) tabela verdade para os flip-flops JK; (*e*) mapas K para os Js e Ks dos flip-flops; e (*f*) mapa K para a saída.

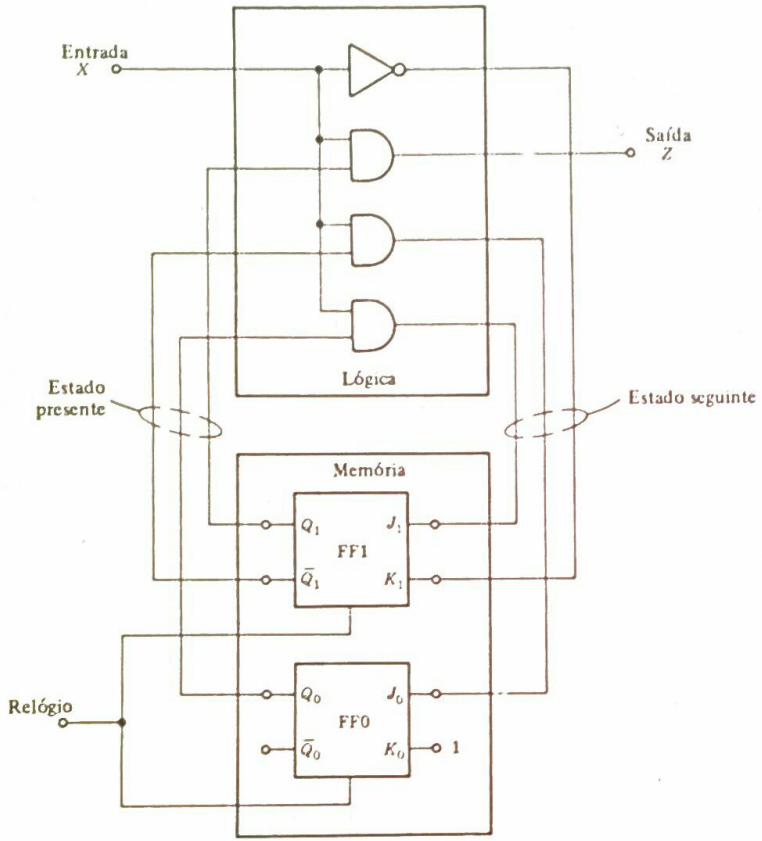


Fig. 7.5-3 Projeto concluído do detector de seqüências de circuito Mealy.

da Fig. 7.5-4 os únicos símbolos novos são os círculos exatamente acima do estado *A* e do estado *D*. Eles não têm nenhuma significação especial e são usados por questão de conveniência quando inúmeras setas provenientes de inúmeras fontes necessitam ser orientadas para uma única terminação. Na Fig. 7.5-5 as novas caixas ovais são usadas para especificar saídas num sistema Mealy. Assim, por exemplo, lemos que, se o sistema estiver no estado *C* e $X = 1$, então $Z = 1$ e o próximo estado estável é novamente *C*. Se $X = 0$, então $Z = 0$ e o próximo estado é *A*.

Finalmente, notamos novamente que a arquitetura de um sistema seqüencial é conforme mostrado na Fig. 7.5-6. O número de entradas pode ser zero (um simples contador) ou n, as entradas sendo $X_0, X_1, \ldots, X_{n-1}$. As saídas são em número de m, $Z_0, Z_1, \ldots, Z_{m-1}$. Deve haver pelo menos uma saída, pois, se não houver saídas, o sistema não faz nada. A caixa lógica é um circuito combinacional, podendo ser montada a partir de portas individuais, ou, caso se revele ser

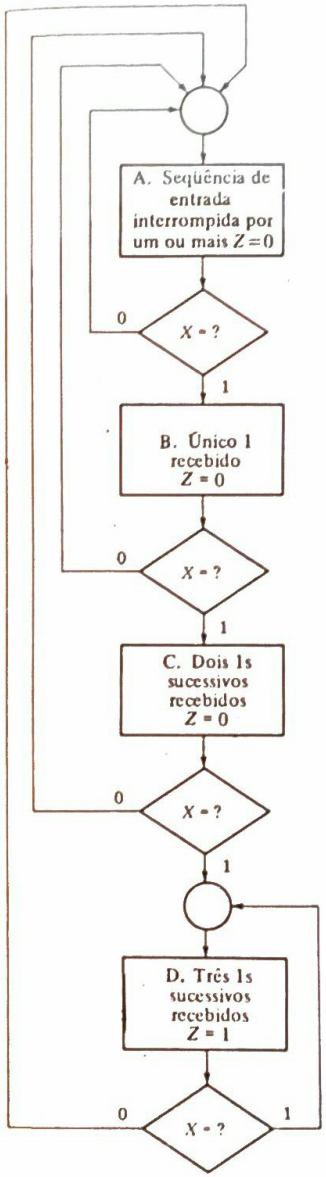


Fig. 7.5-4 Fluxograma do detector de seqüências de circuito Moore.

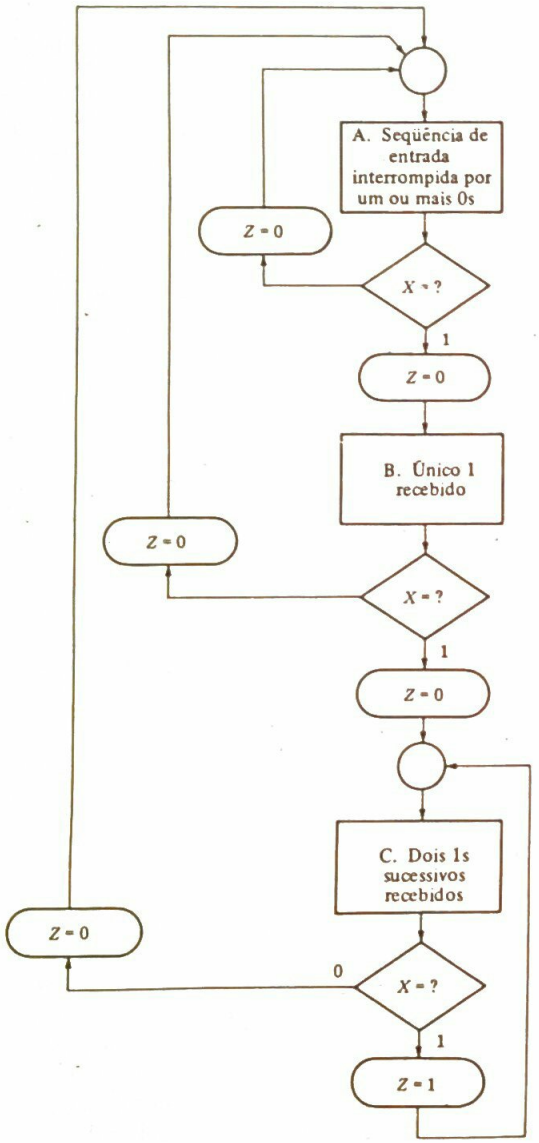


Fig. 7.5-5 Fluxograma do detector de seqüências de circuito Mealy.

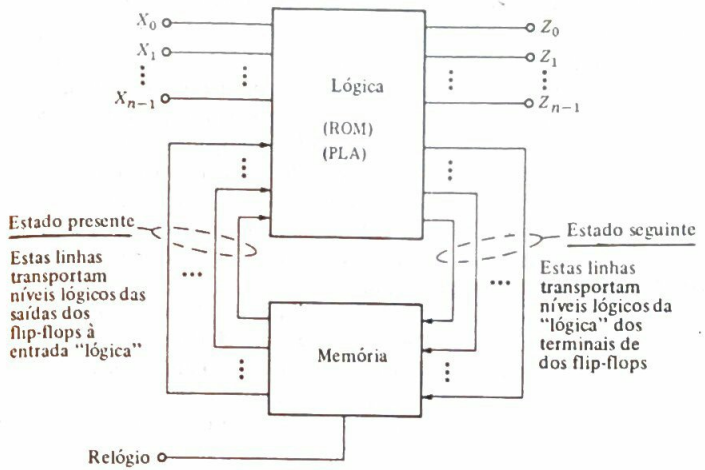


Fig. 7.5-6 A arquitetura de um circuito seqüencial.

economicamente viável, o bloco lógico pode ser uma ROM ou um PLA. O número de flip-flops deve ser adequado para proporcionar todos os estados requeridos.

Se não houver nenhuma linha de entrada, a tabela de estados para o sistema terá apenas uma única coluna no estado seguinte. Uma entrada irá gerar duas colunas no estado seguinte, e n entradas irão gerar 2^n colunas no estado seguinte. Quando n for grande, uma tabela de estados completamente detalhada pode ser uma visão assustadora; seis entradas irão gerar 64 colunas no estado seguinte.

7.6 ELIMINAÇÃO DE ESTADOS REDUNDANTES

Conforme observamos, a primeira etapa no projeto de um sistema seqüencial consiste em considerar cuidadosamente os estados de lembrança requeridos. Este processo conduz a um fluxograma; num caso mais simples, podemos desviar o fluxograma e ir diretamente a um diagrama de estados. Suponhamos que, ao considerarmos cuidadosamente o assunto, falhemos em um ou mais pontos em reconhecer que é requerido um novo estado. Num tal caso, o sistema simplesmente não funcionará conforme requerido. Se acharmos que o sistema não funciona adequadamente, não há nenhuma alternativa para começar tudo de novo, refletindo sobre o assunto mais cuidadosamente. Por outro lado, suponhamos que em vários pontos no desenvolvimento de um fluxograma ou tabela de estados imaginemos por engano que alguma coisa nova para lembrar tenha aparecido e correspondentemente acrescente novos estados. No último caso não houve prejuízo, pois, conforme veremos, existem procedimentos simples para eliminar estes estados supérfluos.

O importante a lembrar no trato com estados redundantes é que um sistema seqüencial é construído com o único propósito de fornecer uma determinada seqüência de níveis lógicos em sua saída ou saídas em resposta à seqüência ou às seqüências aplicadas em suas entradas. Portanto, suponhamos que temos dois sistemas seqüenciais. Suponhamos que um tem quatro flip-flops que

são usados para gerar dezesseis estados e suponhamos que, como ele mostra suas habilidades, ele realmente ocupa, em certa época ou em outra, cada um de seus dezesseis estados. Suponhamos que o outro sistema tenha apenas dois flip-flops e somente quatro estados. Suponhamos ainda, finalmente, que ambos os sistemas forneçam as mesmas seqüências de saída para as mesmas seqüências de entrada arbitrárias. Então, no que nos diz respeito, os dois sistemas são idênticos e indistintos. Sem dúvida, o sistema de dezesseis estados pode ter a *potencialidade* de proporcionar saídas que o sistema de quatro estados não pode proporcionar. Ainda, novamente, até um tal instante em que nos propomos a tirar vantagem desta potencialidade, os sistemas são indistintos. Por conseguinte, sob tais circunstâncias, julgaríamos doze dos dezesseis estados do sistema de dezesseis estados como sendo redundantes. Com estas considerações em mente, vejamos agora como os estados redundantes podem ser reconhecidos numa tabela de estados.

Consideremos a situação representada na tabela de estados da Fig. 7.6-1. Aqui contemplamos um sistema Mealy com uma única entrada X e uma única saída Z. (Nossos comentários também se aplicarão a um número arbitrário de entradas e saídas, bem como a um sistema Moore.) A tabela de estados inteira não foi especificada; em vez disto, chamamos a atenção para o fato de que achamos na tabela dois estados especiais p e q. Estes estados têm a característica especial de que, se o estado presente for p ou se ele for q, então, não importa qual o valor de X, a saída é a mesma e os estados seguintes também são os mesmos. Assim, achamos dois estados p e q na coluna do estado presente cuja saída Z presente e os estados seguintes são idênticos. Então está intuitivamente claro que estes estados são indistintos e um ou outro pode ser dispensado. Portanto, se o estado presente for p ou se ele for q, a saída é a mesma. Também se, por exemplo, $X = 0$, então novamente em ambos os casos o estado seguinte é r e o futuro do sistema depois disto depende do estado r e não de se p ou q precederam r.

Um exemplo de uma tabela de estados, da qual podem ser eliminados os estados redundantes, está indicado na Fig. 7.6-2*a*. Examinando sistematicamente as filas desta tabela, achamos que os estados B e D têm os itens de dados de saída e do estado seguinte idênticos. Portanto, julgamos que o estado D é desnecessário porque ele é o mesmo que o estado B, e eliminamos o estado D e sua fila. (Naturalmente, poderíamos ter eliminado o estado B, se o escolhêssemos.) Na tabela de estados reduzida na Fig. 7.6-2*b* o estado D foi eliminado sempre que aparecia e substituído por B. Em conseqüência do fato de modificarmos todos os Ds para Bs, notamos agora na Fig. 7.6-2*b* que um novo conjunto de estados A e E exibem sua equivalência. Eliminamos o estado E, e a nova tabela de estados reduzida aparece na Fig. 7.6-2*c*. Não são possíveis outras reduções.

	NS/Z	
PS	$X = 0$	$X = 1$
⋮	⋮	⋮
p	$r/0$	$s/1$
⋮	⋮	⋮
q	$r/0$	$s/1$
⋮	⋮	⋮

Fig. 7.6-1 Uma tabela de estados em que dois estados, p e q, têm os mesmos estados seguintes e saídas para todos os valores de X.

	NS/Z	
PS	$X = 0$	$X = 1$
A	$B/0$	$C/1$
B	$C/0$	$A/1$
C	$D/1$	$B/0$
~~D~~	~~$C/0$~~	~~$A/1$~~
E	$D/0$	$C/1$

(*a*)

	NS/Z	
PS	$X = 0$	$X = 1$
A	$B/0$	$C/1$
B	$C/0$	$A/1$
C	$B/1$	$B/0$
E	$B/0$	$C/1$

(*b*)

	NS/Z	
PS	$X = 0$	$X = 1$
A	$B/0$	$C/1$
B	$C/0$	$A/1$
C	$B/1$	$B/0$

(*c*)

Fig. 7.6-2 (*a*) Uma tabela de estados em que os estados B e D são os mesmos. (*b*) A tabela de estados reduzida após o estado D ter sido eliminado. (*c*) A tabela de estados reduzida ainda mais pela eliminação do estado E.

7.7 ELIMINAÇÃO DE ESTADOS REDUNDANTES POR PARTIÇÃO

Na seção anterior vimos que dois estados numa tabela de estados que tem estados seguintes e saídas idênticos são realmente estados idênticos, de modo que um estado pode ser eliminado. Veremos agora que uma tabela de estados pode ter estados redundantes mesmo quando a tabela de estados não exibe linhas com estados seguintes e saídas idênticos.

Para ilustrar o detalhe e verificar como os estados redundantes são discernidos numa tabela de estados do último tipo, consideremos a tabela de estados na Fig. 7.7-1*a*. [Ignoremos, por enquanto, os índices nos itens de dados (entradas) do estado presente.] Um exame da tabela de estados não revela linhas com itens de dados de saída e do estado seguinte idênticas. Lembrando que o pormenor do sistema seqüencial consiste em fornecer *saídas* conforme requerido, examinemos os *itens de dados de saída* na tabela. Notemos que os itens de dados de saída são os mesmos em todos os estados, exceto F. Portanto, está claro que, independente de qualquer outro estado que possamos determinar, o estado F deve ser diferente de todos os outros estados. Todos os outros estados produzem uma saída $Z = 0$ quando $X = 0$, enquanto o estado F produz $Z = 1$ quando $X = 0$. Para perseguir esta ênfase em outras saídas, tomemos, por tentativa, a atitude de que, uma vez que todos os estados, exceto F, têm as mesmas saídas, estes estados são, na realidade, todos o mesmo estado. Nesta base, dividimos os estados em duas classes, ou partições, conforme indicado pelos subscritos, ou índices. Colocamos todos os estados na posição 1, enquanto o estado F colocamos na partição 2. Esta partição em classes é indicada pelos subscritos nos estados na coluna (PS) de estado presente. Até que achemos a hipótese insustentável ou injustificável, admitiremos que todos os estados na mesma partição realmente são o mesmo estado.

Na Fig. 7.7-1*b* copiamos a tabela da Fig. 7.7-1*a*, exceto que indicamos a classe de estados não somente na coluna de estados presentes mas também nas colunas (NS) de estados seguintes. Não copiamos as saídas porque não mais necessitamos delas. As saídas conforme dadas na Fig. 7.7-1*a* serviram para indicar uma partição inicial em duas classes e, para nosso propósito imediato, não podem fornecer nenhuma outra informação. Agora, examinando a tabela na Fig. 7.7-1*b*, achamos que o estado D, inicialmente colocado na partição 1, realmente não pode ser o mesmo estado que os outros estados na partição 1. Isto pode ser visto do fato de que, quando $X = 1$, todos os outros estados na partição 1 vão para os estados na mesma partição (partição 1) enquanto

PS	NS/Z	
	$X=0$	$X=1$
A_1	$B/0$	$C/0$
B_1	$D/0$	$E/0$
C_1	$G/0$	$E/0$
D_1	$H/0$	$F/0$
E_1	$G/0$	$A/0$
F_1	$G/1$	$A/0$
G_1	$D/0$	$C/0$
H_1	$H/0$	$A/0$

(a)

PS	NS	
	$X=0$	$X=1$
A_1	B_1	C_1
B_1	D_1	E_1
C_1	G_1	E_1
D_1	H_1	F_2
E_1	G_1	A_1
F_2	G_1	A_1
G_1	D_1	C_1
H_1	H_1	A_1

(b)

PS	NS	
	$X=0$	$X=1$
A_1	B_1	C_1
B_1	D_3	E_1
C_1	G_1	E_1
D_3	H_1	F_2
E_1	G_1	A_1
F_2	G_1	A_1
G_1	D_3	C_1
H_1	H_1	A_1

(c)

PS	NS	
	$X=0$	$X=1$
A_1	B_4	C_1
B_4	D_3	E_1
C_1	G_4	E_1
D_3	H_1	F_2
E_1	G_4	A_1
F_2	G_4	A_1
G_4	D_3	C_1
H_1	H_1	A_1

(d)

PS	NS	
	$X=0$	$X=1$
A_5	B_4	C_5
B_4	D_3	E_5
C_5	G_4	E_5
D_3	H_1	F_2
E_5	G_4	A_5
F_2	G_4	A_5
G_4	D_3	C_5
H_1	H_1	A_5

(e)

PS	NS/Z	
	$X=0$	$X=1$
a	$b/0$	$a/0$
b	$c/0$	$a/0$
c	$e/0$	$d/0$
d	$b/1$	$a/0$
e	$e/0$	$a/0$

(f)

$a = (A, C, E)$
$b = (B, G)$
$c = D$
$d = F$
$e = H$

(g)

Fig. 7.7-1 (*a*) Uma tabela de estados. Na coluna de estados presentes os estados são separados de acordo com a saída. (*b*) a (*f*) Estágios na redução das tabelas de estado. (*g*) Os estados restantes e sua relação com os estados iniciais.

D vai para um estado (F) numa partição diferente (partição 2). Suponhamos que começamos com o sistema, de um lado, por exemplo, no estado A e, por outro lado, no estado D. Então, como pode ser verificado da tabela na Fig. 7.7-1*a*, se $X = 0$, em ambos os casos teríamos $Z = 0$. Se, no entanto, o valor seguinte de X fosse novamente $X = 0$, então num caso teríamos $Z = 0$ e no outro caso $Z = 1$. Através de uma fácil generalização desta observação, chegamos ao seguinte princípio: *Dois estados cujos estados seguintes em cada coluna de estados seguintes não estão nas mesmas partições devem ser estados diferentes.* Aplicando este princípio à tabela na Fig. 7.7-1*b*, achamos que podemos persistir ainda na hipótese de que os estados A, B, C, E, G e H são o mesmo estado e daí deixarmos todos na partição 1. No entanto, o estado D deve ser removido desta

partição. Portanto, colocamos o estado D na partição 3. Copiamos novamente a tabela de estados na Fig. 7.7-1*c* e indicamos em toda a tabela que o estado D está na partição 3.

Usando o princípio estabelecido acima; achamos agora da tabela na Fig. 7.7-1*c* que nem o estado B nem o estado G podem permanecer na partição 1. Ambos estes estados vão para os estados em cada coluna de estados presentes que estão na mesma partição. Daí, enquanto devemos removê-los da partição 1, podemos colocá-los ambos na mesma partição, como indicado na tabela da Fig. 7.7-1*d*. Da Fig. 7.7-1*d*, alternativamente, achamos que os estados C e E devem ser removidos da partição 1, mas podem permanecer na mesma partição; isto é, partição 5. Finalmente, na Fig. 7.7-1*e* notamos que nenhuma outra proliferação de partições é requerida. Todos os estados que estão na mesma partição (A, C e E na partição 1 e B e G na partição 4) têm itens de dados (entradas) de estado seguinte em cada coluna que estão na mesma partição.

Quando a partição estiver completa, todos os estados ainda na mesma partição constituem o mesmo estado. Por exemplo, se começarmos pelo estado A ou C ou E, e se supusermos uma seqüência inteiramente arbitrária de entrada Xs, então em cada caso a seqüência de saídas será a mesma. Daí, conforme indicado na figura, uma vez que temos cinco partições, há somente cinco estados. Usando letras minúsculas para designar estes estados, temos $a = (A, C, E)$, $b = (B, G)$, $c = D$, $d = F$, e $e = H$.

A tabela final de cinco estados reduzidos na Fig. 7.7-1*f* é alcançada da seguinte maneira. Introduzimos o estado a na coluna de estados presentes. A seguir retornamos à tabela de estados original na Fig. 7.7-1*a*. Não importa se, ao olharmos a linha de estados A, C ou E, achamos $Z = 0$ para $X = 0$ e $X = 1$; daí fazermos entradas correspondentes na tabela reduzida. Se olharmos novamente a linha de A, achamos que para $X = 0$ o estado seguinte é B, que é agora incluído no estado b. Se olharmos a linha de C ou E, achamos que para $X = 0$ o próximo estado é G, que é também incluído no estado b. Por conseguinte, na tabela de estados reduzida, o próximo estado para o estado a e para $X = 0$ é o estado b. O restante da tabela reduzida é completado de maneira idêntica. Nesta discussão, conseguimos mostrar em detalhes que, se o estado a for visto como A ou como C ou como E, o resultado na tabela reduzida é o mesmo. Realmente, não é necessário mostrar que tal é o caso. Portanto, se A, C e E forem de fato o mesmo estado, o resultado é, naturalmente, inevitável.

7.8 UM EXEMPLO

Nesta seção projetaremos um outro detector de seqüências para proporcionar compreensão adicional na geração de um diagrama de estados a partir de uma especificação do problema enunciado em palavras. Além disto, o exemplo nos dará uma oportunidade de aplicar os processos disponíveis para eliminação de estados redundantes.

Nosso detector de seqüências deve ser um sistema Mealy e deve ter uma única entrada X e uma única saída Z. Em qualquer intervalo de relógio em que $X = 0$, a saída deve ser $Z = 1$, contanto que nos quatro intervalos precedentes a entrada seja $X = 1001$. Assim, uma seqüência bem-sucedida que leva a $Z = 1$ é $X = 10010$. Para evitar incerteza em torno do que se pretende, uma possível seqüência de entrada e a correspondente saída foram especificadas na Fig. 7.8-1. Os valores de X nos intervalos de relógio 2, 3, 4, 5 e 6 constituem uma seqüência bem-sucedida e, conseqüentemente, no intervalo 6 temos $Z = 1$. Observemos que é permitida a superposição de seqüências bem-sucedidas. Assim, os últimos dois bits da primeira seqüência bem-sucedida consti-

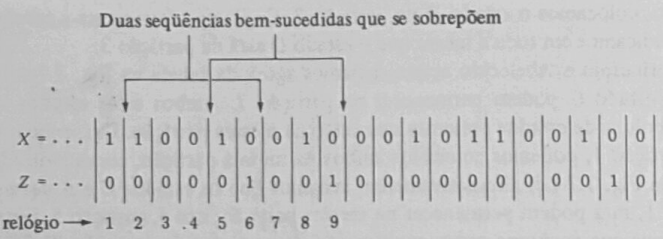


Fig. 7.8-1 Uma seqüência de entrada e a correspondente seqüência de saída de um detector que procura a seqüência $X = 10010$ (sobreposição permitida).

tuem os primeiros dois bits de uma segunda seqüência bem-sucedida que se segue imediatamente à primeira. Notar também que, se não tivéssemos especificado X no intervalo 1 de relógio, não teríamos sabido se o $X = 0$ no intervalo 3 constituía o fim de uma seqüência bem-sucedida e, também, não teríamos sabido o valor de Z no intervalo 3.

Desviaremos o fluxograma e começaremos pelo diagramas de estados. Conforme notamos de início, devemos definir um primeiro estado especificando bastante da história que conduz até este estado para evitar ter que retornar além deste primeiro estado. Há muitas maneiras pelas quais o primeiro estado pode ser definido no caso presente. Consideraremos o primeiro estado como sendo o estado atingido após haver tido uma sucessão de dois (ou mais) 1s sucessivos. (Outros possíveis primeiros estados são deixados para consideração do estudante; ver Prob. 7.8-1.) Este primeiro estado é o estado em que o sistema acha-se no intervalo 3 de relógio na Fig. 7.8-1. (Lembremo-nos de que o estado em qualquer intervalo de relógio não é influenciado pela entrada neste intervalo, mas somente pelas entradas nos intervalos anteriores.) Em qualquer evento, este primeiro estado, estado A na Fig. 7.8-2, é um estado em que o primeiro bit de uma seqüência potencialmente bem-sucedida foi recebido e qualquer coisa que aconteceu de início (antes dos dois 1s sucessivos) é irrelevante.

Este primeiro estado A (atingido depois de dois intervalos sucessivos em que $X = 1$ e em cujos intervalos Z será $Z = 0$) está indicado na Fig. 7.8-2*a*. Se agora houver um terceiro $X = 1$, este terceiro 1 substituirá o segundo 1 como o primeiro bit de uma seqüência potencialmente bem-sucedida. Em qualquer caso, nenhum progresso foi feito com vistas a uma seqüência bem-sucedida. O que o sistema deve lembrar é o mesmo que antes. Daí, como indicado pela seta que começa e termina no estado A, quando $X = 1$, então $Z = 0$ e o estado "seguinte" é o próprio estado A. Agora admitamos que os próximos quatro bits de entrada sejam $X = 0010$; assim, aparece uma seqüência bem-sucedida. Então, a ocorrência de cada nova etapa deve ser lembrada. Em conseqüência, cada nova entrada gera um novo estado, como indicado na Fig. 7.8-2*a*. O último $X = 0$ gera uma saída $Z = 1$, e na próxima borda de gatilho de relógio o sistema vai para o estado E.

O diagrama de estados está completo na Fig. 7.8-2*b*. (Se desenvolvemos o diagrama de estados correta ou incorretamente, o diagrama não está completo até que duas setas saiam de cada estado.) Se, quando o sistema está no estado B, tivermos $X = 1$, a seqüência torna-se não-válida e não podemos ir de B para C; assim, introduzimos um novo estado F. Para futura referência, notamos aqui que F é um estado alcançado depois da seqüência 1101 e, por conseguinte, é um estado que lembra que o primeiro bit, isto é, o último 1, de uma seqüência potencialmente

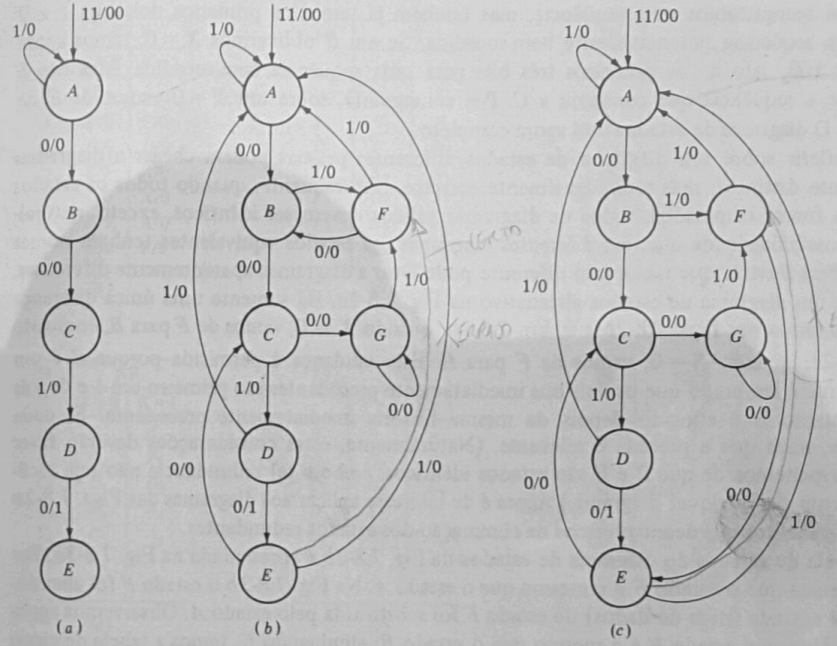


Fig. 7-8-2 (*a*) Uma tabela de estados incompleta mostrando o desenvolvimento de uma seqüência bem-sucedida. (*b*) Um diagrama de estados completo. (*c*) Um diagrama de estados alternativo.

bem-sucedida foi recebido. Em F, suponhamos que a entrada seguinte é $X = 0$. Então, a seqüência até agora é 11010. Tudo o que necessitamos agora para lembrar a respeito da seqüência 11010 é que os dois últimos de seus bits são potencialmente os dois primeiros bits de uma seqüência bem-sucedida. Mas esta lembrança é precisamente a lembrança armazenada em B. Por conseguinte, conforme indicado, em F, um $X = 0$ produz $Z = 0$, sendo o estágio seguinte novamente o estado B. Se, em F, o bit seguinte for $X = 1$, então os dois últimos bits são 1s sucessivos e o sistema vai para o estado A, pois o estado A foi precisamente definido como o estado do sistema após dois 1s sucessivos.

Em C, se a entrada seguinte for $X = 0$, temos três 0s sucessivos, a seqüência é não-válida e não podemos ir de C a D. Portanto, introduzimos o novo estado G. O estado G é um estado que lembra que a seqüência foi tornada não-válida porque houve uma seqüência de mais do que dois 0s e que nenhum progresso com vistas a uma seqüência bem-sucedida será feito até que apareça um 1. Em G, mais 0s não modificam o que tem que ser lembrado, de modo que $X = 0$ deixa o sistema no estado G. Por outro lado, em G, se $X = 1$, novamente temos o primeiro bit de uma seqüência potencialmente bem-sucedida. O estado F é um estado que lembra precisamente que este primeiro bit foi recebido. Daí um $X = 1$ produz $Z = 0$ e leva o sistema de G de volta a F.

Em D, um $X = 1$ significa que dois 1s sucessivos foram recebidos, e isto é a lembrança armazenada em A. Por conseguinte, quando $X = 1$, vamos de D a A enquanto $Z = 0$. Finalmente, lembremo-nos de que a superposição de seqüências é permitida. Daí, quando alcançamos o estado E,

não somente completamos uma seqüência, mas também já temos os primeiros dois bits 1 e 0 de uma nova seqüência potencialmente bem-sucedida. Se em E obtivermos $X = 0$, temos agora a seqüência 100, isto é, os primeiros três bits para uma seqüência bem-sucedida. Mas isto é precisamente a seqüência que conduziu a C. Por conseguinte, sobre um $X = 0$ vamos de E de retorno a C. O diagrama de estados está agora completo.

Ao refletir sobre um diagrama de estados, diferentes pessoas podem chegar a diagramas aparentemente distintos, mas todos igualmente corretos. Naturalmente, quando todos os estados redundantes forem eliminados, todos os diagramas válidos devem ser idênticos, exceto, naturalmente, a possibilidade de que, em diferentes diagramas, os estados equivalentes tenham nomes diferentes. Para ilustrar que raciocínio diferente pode levar a diagramas aparentemente diferentes, desenhamos um diagrama de estados alternativo na Fig. 7.8-2*c*. Há somente uma única diferença entre os diagramas nas Figs. 7.8-2*b* e *c*. Em 7.8-2*b*, quando $X = 0$, vamos de F para B, enquanto na Fig. 7.8-2*c*, quando $X = 0$, vamos de F para E. Esta mudança é permitida porque B é um estado em que é lembrado que os dois bits imediatamente precedentes são primeiro um 1 e depois um 0, enquanto E é atingido depois da mesma história imediatamente precedente. E, dada esta história, nada que a preceda é relevante. (Naturalmente, estas considerações deverão fazer com que suspeitemos de que B e E são estados idênticos, embora tal redundância não seja facilmente evidente em qualquer diagrama.) Agora é de interesse aplicar aos diagramas das Figs. 7.8-2*b* e *c* os procedimentos que desenvolvemos na eliminação dos estados redundantes.

A tabela de estados do diagrama de estados da Fig. 7.8-2*b* é apresentada na Fig. 7.8-3*a*. Por inspeção, vemos que o estado F é o mesmo que o estado A. Na Fig. 7.8-3*b* o estado F foi abandonado e cada entrada (item de dados) do estado F foi substituída pelo estado A. Observemos agora na Fig. 7.8-3*b* que o estado E é o mesmo que o estado B; eliminando E, temos a tabela de cinco estados na Fig. 7.8-3*c*. Não são possíveis outras reduções de estado por inspeção. Podemos agora tentar a partição para verificar se é possível uma outra redução. Verifica-se imediatamente que não é possível.

Na Fig. 7.8-4*a* preparamos a tabela de estados correspondente ao diagrama de estados da Fig. 7.8-2*c*. Aqui parece que não é possível nenhuma redução de estado por inspeção. Portanto tentemos o método de partição. O processo é executado etapa por etapa nas tabelas da Fig. 7.8-4. Na Fig. 7.8-4*e* achamos finalmente que nenhuma outra partição é solicitada. Aqui também

PS	NS/Z	
	$X=0$	$X=1$
A	$B/0$	$A/0$
B	$C/0$	$F/0$
C	$G/0$	$D/0$
D	$E/1$	$A/0$
E	$C/0$	$A/0$
~~F~~	~~$B/0$~~	~~$A/0$~~
G	$G/0$	$F/0$

(*a*)

PS	NS/Z	
	$X=0$	$X=1$
A	$B/0$	$A/0$
B	$C/0$	$A/0$
C	$G/0$	$D/0$
D	$E/1$	$A/0$
~~E~~	~~$C/0$~~	~~$A/0$~~
G	$G/0$	$A/0$

(*b*)

PS	NS/Z	
	$X=0$	$X=1$
A	$B/0$	$A/0$
B	$C/0$	$A/0$
C	$G/0$	$D/0$
D	$B/1$	$A/0$
G	$G/0$	$A/0$

(*c*)

Fig. 7.8-3 (*a*) A tabela de estados correspondente ao diagrama de estados na Fig. 7.8-2*b*. (*b*) Uma tabela de estados parcialmente reduzida. (*c*) A tabela de estados completamente reduzida.

PS	NS/Z	
	$X=0$	$X=1$
A	$B/0$	$A/0$
B	$C/0$	$F/0$
C	$G/0$	$D/0$
D	$E/1$	$A/0$
E	$C/0$	$A/0$
F	$E/0$	$A/0$
G	$G/0$	$F/0$

(*a*)

PS	NS	
	$X=0$	$X=1$
A_1	B_1	A_1
B_1	C_1	F_1
C_1	G_1	D_2
D_2	E_1	A_1
E_1	C_1	A_1
F_1	E_1	A_1
G_1	G_1	F_1

(*b*)

PS	NS	
	$X=0$	$X=1$
A_1	B_1	A_1
B_1	C_3	F_1
C_3	G_1	D_2
D_2	E_1	A_1
E_1	C_3	A_1
F_1	E_1	A_1
G_1	G_1	E_1

(*c*)

PS	NS	
	$X=0$	$X=1$
A_1	B_4	A_1
B_4	C_3	F_1
C_3	G_1	D_2
D_2	E_4	A_1
E_4	C_3	A_1
F_1	E_4	A_1
G_1	G_1	F_1

(*d*)

PS	NS	
	$X=0$	$X=1$
A_1	B_4	A_1
B_4	C_3	F_1
C_3	G_5	D_2
D_2	E_4	A_1
E_4	C_3	A_1
F_1	E_4	A_1
G_5	G_5	F_1

(*e*)

PS	NS/Z	
	$X=0$	$X=1$
A	$B/0$	$A/0$
B	$C/0$	$A/0$
C	$G/0$	$D/0$
D	$B/1$	$A/0$
G	$G/0$	$A/0$

(*f*)

Fig. 7.8-4 (*a*) A tabela de estados correspondente ao diagrama de estados da Fig. 7.8-2*c*. (*b*) a (*f*) Estágios na redução da tabela de estados por partição.

achamos que A e F estão ainda na mesma partição que B e E. Abandonando E e F, chegamos, finalmente, à tabela na Fig. 7.8-4*f*, que é a mesma tabela da Fig. 7.8-3*c*.

Notamos assim que, às vezes, um diagrama ou tabela de estados fornecerá redução por inspeção e, às vezes, será necessária a partição. Num caso um tanto geral, uma tabela pode fornecer certa redução por inspeção e pode fornecer, ainda, redução por partição. Uma vez que a inspeção é mais simples, sempre é sensato tentar primeiro a inspeção e depois aplicar partição.

Quando uma tabela de estado sem estados redundantes tiver sido gerada, o restante do projeto acompanha o procedimento especificado de início (ver Fig. 7.5-2). No caso presente, uma vez que há cinco estados, necessitaremos de três flip-flops. Uma vez que o próximo estado depende do estado presente (determinado pelos três flip-flops) e depende igualmente da entrada X, as entradas de excitação dos flip-flops dependerão de quatro variáveis. Se, como na Fig. 7.5-2, escolhermos usar flip-flops JK, teremos que gerar seis mapas K para as excitações dos flip-flops, isto é, um mapa para cada J e um mapa para cada K de cada um dos três flip-flops. Teremos também um mapa K para a saída Z. Cada um destes mapas K envolverá quatro variáveis. Quando os mapas tiverem sido preenchidos, lemos as equações lógicas de todos os Js e Ks e de Z. Depois disto o diagrama de circuito pode ser desenhado diretamente. A conclusão destes procedimentos de projeto é deixada como um problema (Prob. 7.8-2).

7.9 ATRIBUIÇÕES DE ESTADOS

Até o presente ponto temos sido inteiramente arbitrários em nossa seleção de atribuições de estados. Surge naturalmente a pergunta se algumas atribuições de estado são melhores do que outras. A resposta é sim. Se observarmos novamente as Figs. 7.4-4 e 7.5-2, é evidente que a atribuição de estado que escolhemos terá um efeito marcante sobre onde os 1s, 0s e Xs aparecem nos mapas K para as excitações de flip-flops e mesmo no mapa de Z. Naturalmente, gostaríamos que os 1s, 0s e Xs caíssem num padrão que produzisse as expressões lógicas mais simples possíveis para as excitações de entrada dos flip-flops. Revela-se que não há regras facilmente aplicáveis que fornecerão uma seleção da melhor atribuição de estado. Há, no entanto, inúmeras regras cuja aplicação conduzirá a uma boa atribuição de estado – não necessariamente a melhor, porém geralmente melhor do que as atribuições selecionadas aleatoriamente. Algumas destas regras, bem como a verificação de que elas são eficientes, são consideradas nos Probs. 7.9-3 e 7.9-4.

Uma palavra a respeito do número de possíveis atribuições diferentes. Elas são em número menor do que possamos antecipar, mas também bem maior do que possamos argumentar. Por exemplo, consideremos que temos quatro estados A, B, C e D e, conseqüentemente, usamos dois flip-flops com saídas Q_1 e Q_0. Uma possível atribuição de estados seleciona $A = Q_1Q_0 = 00$ etc., como na Fig. 7.2-2. Mas a atribuição de A pode ser selecionada fora de quatro possibilidades, deixando três possibilidades para B, duas para C e, finalmente, uma para D. Podemos então esperar que o número total de atribuições seja $4 \cdot 3 \cdot 2 \cdot 1 = 24$. No entanto, trocando as colunas de 1s e de 0s sob Q_1 e Q_0, simplesmente troca-se os nomes dos flip-flops, de modo que Q_1 torna-se Q_0 e vice-versa. Também podemos trocar 1s e 0s em qualquer ou em ambas as colunas, uma vez que uma tal troca simplesmente altera a designação das saídas de um flip-flop, o terminal Q sendo agora chamado $\bar{Q}$ e vice-versa. Está intuitivamente claro que uma mudança de nome ou mudança de designação pode não ter nenhum efeito sobre a complexidade de um circuito físico. Revela-se, então, que no caso de quatro estados há somente três diferentes atribuições de estado possíveis, e neste caso é viável experimentar todas elas; no entanto, com cinco, seis, sete, oito e nove estados etc., revela-se que o número de atribuições diferentes é de, respectivamente, 140, 420, 840, 10.810.800 etc.

7.10 PROJETOS ALTERNATIVOS

Na discussão precedente, nosso interesse foi na economia de um projeto de sistema seqüencial. Procuramos usar o número mínimo de flip-flops e o número mínimo possível de portas lógicas adicionais. Atualmente, quando o hardware é relativamente tão pouco dispendioso, esta filosofia de projeto não é necessariamente a melhor, e pode mesmo ser vantajoso montar um sistema na base do que parece intuitivamente natural e razoável. Um tal procedimento de projeto heurístico pode ser especialmente eficiente se nos permitir usar componentes comerciais facilmente encontrados e se gera um sistema em que a função de cada parte componente se distingue nitidamente.

Como um exemplo de um tal projeto intuitivo, baseado em nossa experiência e sofisticação com sistemas digitais, retornemos ao problema do detector de seqüências introduzido na Seção 7.8. O detector devia responder à seqüência $X = 10010$. Ocorre naturalmente para nós que o problema aqui origina-se essencialmente do fato de que os bits de entrada nos são apresentados um de cada vez, em vez de todos imediatamente. No entanto, estando cientes dos progressos dos

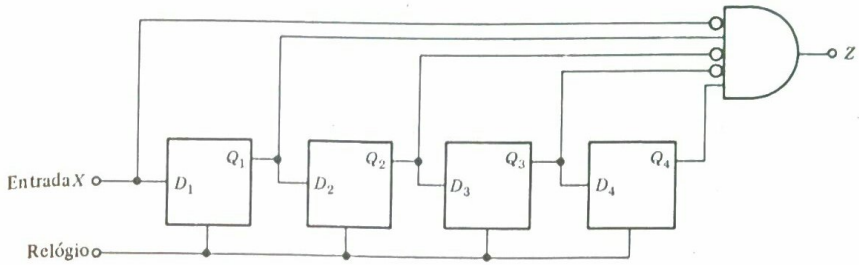
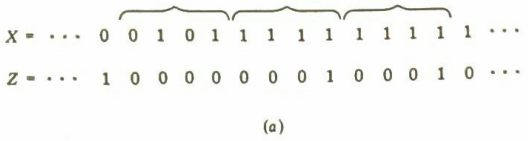


Fig. 7.10-1 Um projeto heurístico do detector de seqüências que responde à seqüência $X = 10010$.

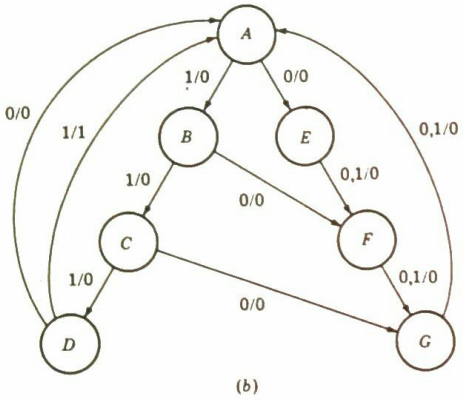
registradores de deslocamento, também compreendemos que um registrador de deslocamento pode ser usado para capturar uma seqüência de tempo de bits de modo que todos os bits tornem-se disponíveis num certo instante. Nesta base somos naturalmente levados a "projetar" nosso detector de seqüências da maneira indicada na Fig. 7.10-1. Por enquanto, suponhamos que num intervalo de relógio específico tenhamos $X = 0$ e que nos quatro intervalos de relógio precedentes X tenha sido 1001. Então, no intervalo de relógio específico, temos $X = 0$, $Q_1 = 1$, $Q_2 = Q_3 = 0$ e $Q_4 = 1$. Então a saída Z da porta AND será $Z = 1$, conforme requerido. Para nos certificarmos, o sistema da Fig. 7.10-1 usa quatro flip-flops que possibilitam dezesseis estados, enquanto, como observado de início, o sistema mais econômico de estados requer apenas cinco estados. Ainda quando consideramos que as frações de pastilha de circuitos integrados são pouco dispendiosas e proporcionam registradores de deslocamento de quatro (e mais) estágios, o sistema da Fig. 7.10-1 pode muito bem ser o projeto de escolha.

Como um segundo exemplo de projetos alternativos, um baseado em nosso procedimento formalizado e um baseado na intuição e experiência, consideremos o seguinte problema de detector de seqüência. Queremos um detector que proporcione uma saída $Z = 1$ sempre que tiver havido uma seqüência de quatro 1s sucessivos. Proibimos a sobreposição e requeremos que cada grupo de quatro intervalos de relógio seja examinado separadamente. Uma possível seqüência X e sua correspondente saída Z são mostradas na Fig. 7.10-2*a*. O diagrama de estados do circuito seqüencial é dado na Fig. 7.10-2*b*. Deixa-se como um exercício para o estudante verificar que o diagrama de estados é válido e que ele não tem estados redundantes. O estado A é o estado em que o sistema se acha depois de ter acabado de completar o exame de um grupo de quatro bits e é o estado em que ele está preparado para receber o primeiro bit de um novo grupo de quatro bits. Partindo de A, uma seqüência de quatro 1s leva o sistema para B, C e D, e de volta para A. A saída é $Z = 1$ quando $X = 1$ e o sistema está no estado D. Se, partindo de A, a seqüência recebida dos quatro bits seguintes for alguma coisa diferente de 1111, os estados E, F e G juntos com A simplesmente ignoram a contagem dos quatro intervalos, e levam o sistema de volta a A no fim, com o nível de Z mantendo-se em $Z = 0$ em toda parte. Uma vez que o diagrama de estados exibe sete estados, são requeridos três flip-flops. Para completar o projeto, será necessário fazer uma atribuição de estados, selecionar um tipo de flip-flop, deduzir a lógica de excitação do flip-flop e desenhar o circuito. Estes assuntos são deixados como exercícios para o estudante.

A seguir, consideremos como podemos formar o sistema na base do que chega naturalmente. Primeiro, usaremos, é claro, um registrador de deslocamentos para tornar os quatro bits na seqüência disponíveis todos ao mesmo tempo, tal como fizemos no primeiro exemplo desta seção.



(*a*)



(*b*)

Fig. 7.10-2 Um detector de seqüências que responde a uma seqüência de quatro 1s sucessivos com sobreposição proibida: (*a*) possível seqüência de entrada e a saída correspondente; e (*b*) diagrama de estados do detector.

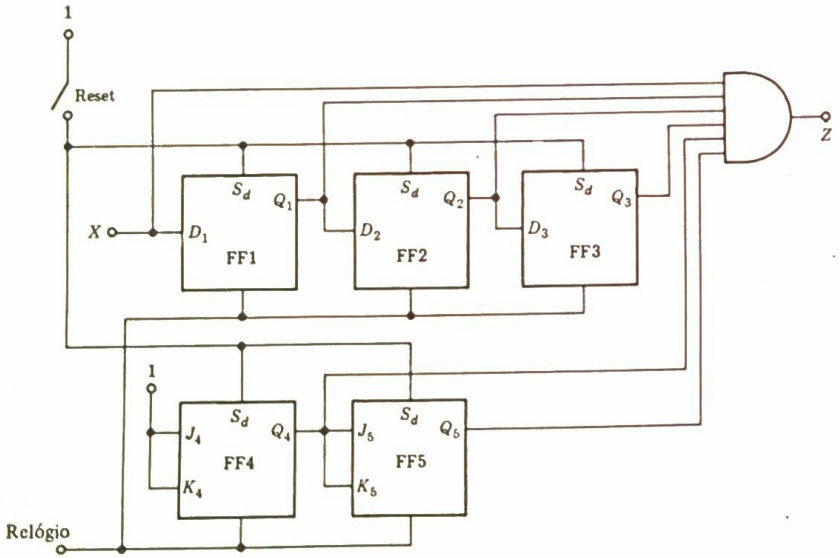
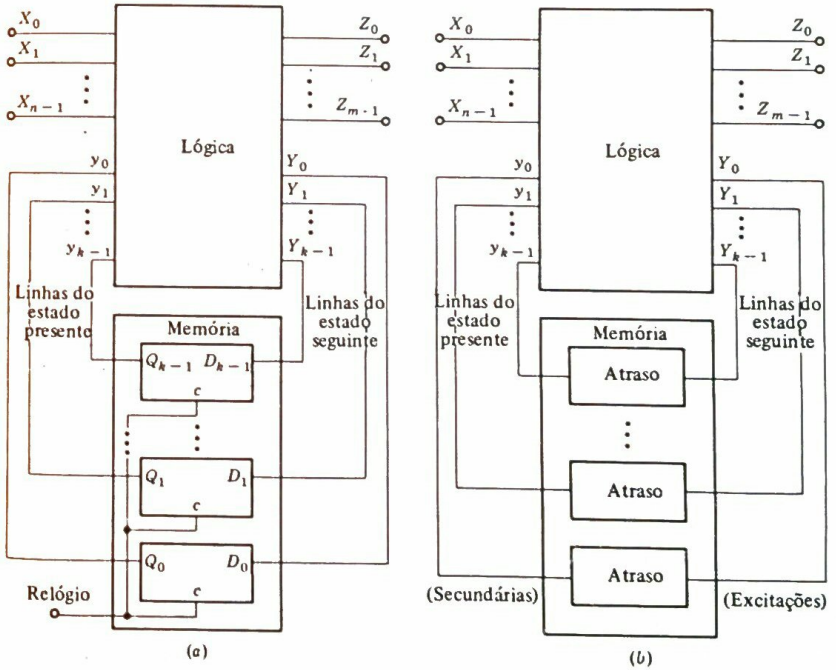


Fig. 7.10-3 Um detector de seqüências de quatro 1s sucessivos projetado numa base heurística.

A seguir, necessitamos de um meio de separação dos bits em grupos independentes de 4. Para este propósito, será eficiente um contador de módulo 4, que é usado para contar ciclos de relógio. O circuito é mostrado na Fig. 7.10-3. Os flip-flops FF1, FF2 e FF3 constituem o registrador de deslocamento de três bits, enquanto os flip-flops FF4 e FF5 constituem um contador síncrono de módulo 4. Fizemos também preparativos para restabelecer (*reset*) o sistema pelo uso dos terminais S_d de reset-direto dos flip-flops. Após a chave de reset ter sido fechada por um tempo curto e depois aberta, o sistema está pronto para receber um primeiro bit, que será aceito como o primeiro de um grupo de 4. Por conseguinte, na abertura da chave de reset, o relógio pode ser ligado e a entrada X aplicada sincronamente. Notar que a porta AND não somente observa a coincidência de quatro 1s em X, Q_1, Q_2 e Q_3, mas também serve como o decodificador para o contador. Este novo projeto usa cinco flip-flops, que permitem 32 estados, e deve ser comparado ao primeiro projeto, que necessitava apenas de três flip-flops.

7.11 CIRCUITOS SEQÜENCIAIS NO MODO FUNDAMENTAL

Na Fig. 7.11-1*a* apresentamos novamente a estrutura lógica de um circuito seqüencial. Se quisermos, podemos razoavelmente tomar a atitude de que o propósito do flip-flop é proporcionar



(*a*) (*b*)

Fig. 7.11-1 (*a*) A estrutura de um circuito seqüencial síncrono. (*b*) A estrutura de um circuito seqüencial (assíncrono) no modo fundamental.

um *atraso* entre as variações nos níveis lógicos na linha dos estados seguintes e as variações correspondentes subseqüentes nas linhas do estado presente. As variações nas linhas de entrada $X_0, X_1, \ldots, X_{n-1}$, se houver, ocorrem nominalmente no instante de uma transição de gatilho de relógio, e as linhas $Y_p, Y_1, \ldots, Y_{k-1}$ do estado presente respondem imediatamente. No entanto, as variações correspondentes nas linhas $y_0, y_1, \ldots, y_{k-1}$ são *atrasadas* até a ocorrência da próxima transição de gatilho do relógio. Estas considerações sugerem que a estrutura da Fig. 7.11-1*b* pode igualmente bem constituir um circuito seqüencial. Aqui os flip-flops foram substituídos por elementos que introduzem um atraso. O tipo de atraso contemplado é o atraso que resulta quando um sinal elétrico é transmitido ao longo de um comprimento de fio ou através de uma porta lógica ou de uma cascata de portas. A diferença essencial entre o atraso fornecido nas Figs. 7.11-1*a* e *b* é que no último caso o atraso é inteiramente determinado pelo próprio elemento de atraso e não por um agente externo (a forma-de-onda de relógio). Na prática, revela-se geralmente que os elementos de atraso no conjunto de circuitos da Fig. 7.11-1*b* não são elementos deliberadamente introduzidos, mas são os atrasos de propagação das portas na parte lógica do sistema.

Quando o sistema na Fig. 7.11-1*a* estiver repousando quiescentemente num de seus estados, as entradas $y_0, y_1, \ldots, y_{k-1}$ podem ser diferentes da saída lógica $Y_0, Y_1, \ldots, Y_{k-1}$. Na Fig. 7.11-1*b*, numa tal condição quiescente, as entradas lógicas devem ser as mesmas que as saídas lógicas. Ainda em resposta a uma variação numa linha de entrada $X_0, X_1, \ldots, X_{n-1}$, as linhas $Y_0, Y_1, \ldots, Y_{k-1}$ variarão e, pelo menos durante o tempo de atraso, os Ys podem ser diferentes dos ys. Uma vez que os Ys e os ys diferem uns dos outros apenas transitoriamente e para um intervalo que não é controlável externamente, há uma relutância por parte de alguns em chamá-las variáveis de *estado seguinte* e variáveis de *estado presente*. Em vez disto, os Ys muitas vezes são chamados variáveis de *excitação*, e os ys são chamados variáveis *secundárias*.

O circuito seqüencial na Fig. 7.11-1*a* que usa k flip-flops e tem k linhas de estado pode ter um número de 2^k estados distintos. Correspondentemente, esperamos que o circuito na Fig. 7.11-1*b*, que usa elementos de atraso, terá também 2^k estados.

Conforme temos observado, os circuitos seqüenciais que usam flip-flops sincronizados (clocados) são chamados *sistemas síncronos*. Ao contrário, os circuitos que usam elementos de atraso são chamados *sistemas de modo fundamental*. Historicamente, os sistemas do modo fundamental apareceram antes dos sistemas síncronos, mas os sistemas síncronos possuem muitas vantagens: eles são mais fáceis de projetar e oferecem liberdade de interesse quanto aos atrasos de propagação variáveis e imprevisíveis. Os sistemas do modo fundamental, portanto, têm aplicação um tanto limitada atualmente.

7.12 UM EXEMPLO DE ANÁLISE

Um exemplo simples de um circuito de modo fundamental é mostrado na Fig. 7.12-1. Nós o desenhamos no padrão da Fig. 7.11-1*b*. Ele é facilmente reconhecido como um latch estático, e designamos apropriadamente os terminais. As entradas X_0 e X_1 são as entradas de *reset* e *set*, e a única saída Z é o terminal Q do flip-flop. Neste presente exemplo há uma única linha Y de estado conectada a y, e acontece que ela é a mesma que a linha Z. O atraso na linha que fornece *realimentação* da saída lógica para a entrada lógica não está explicitamente mostrado. Conforme temos observado, este atraso é realmente fornecido pelos atrasos de propagação das portas. Uma vez que o

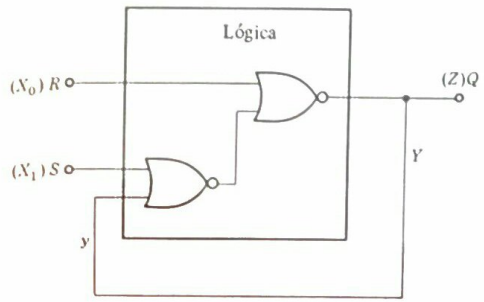


Fig. 7.12-1 Um exemplo de um circuito do modo fundamental.

estado está sendo determinado pelo nível lógico numa única linha, devemos esperar que o sistema tenha dois estados.

O desempenho do circuito da Fig. 7.12-1 é descrito pela tabela da Fig. 7.12-2. Aqui fornecemos um quadrículo correspondente a cada uma das oito possíveis combinações das três variáveis lógicas S, R e y de entrada. Em cada quadrículo introduzimos o valor correspondente da saída $Y\,(=Z)$. *Quando R e S forem invariáveis e o sistema estiver repousando quiescentemente num de seus estados estáveis, evidentemente devemos ter $Y = y$.* Para indicar tais estados estáveis, a entrada Y correspondente na tabela foi envolvida por um círculo. As entradas que não estão envolvidas por um círculo correspondem a situações que podem persistir apenas transitoriamente e são denominadas estados instáveis.

Observemos, de modo coerente com nossa expectativa, que, quando $S = R = 0$, há dois estados estáveis, um em que a *variável de estado* $y = 0$ e um em que $y = 1$. Suponhamos que agora mudamos para $S = 0$, $R = 1$. Então, conforme indicado pela seta em linha cheia, o sistema permanece no mesmo estado estável com $y = 0$. Uma outra mudança para $S = 1$, $R = 1$ ainda deixa o sistema no estado $y = 0$. A seguir, mudemos as entradas de $S = R = 1$ para $S = 1, R = 0$. A despeito da maneira pela qual o diagrama da Fig. 7.12-1 está desenhado, consideremos que não há atraso de propagação através das portas e que em vez disto há um elemento de atraso entre Y e y. Então, conforme indicado nesta tabela, com $S = 1$, $R = 0$ e $y = 0$, teremos $Y = 1$. Esta situação não persistirá. Depois de um atraso, o valor $Y = 1$ aparecerá em y, e teremos $y = 1$. Achamos, da tabela, que $S = 1$, $R = 0$ e $y = Y = 1$ é um estado estável. Assim, sem exceção, quando estivermos no estado $y = 0$ com $SR = 11$, uma mudança para $SR = 10$ leva o sistema transitoriamente através de um estado instável, onde $y = 0$ e $Y = 1$, e dali para um novo estado estável em que $y = 1$. Esta transição do estado estável inicial através de um estado instável transitório para um novo estado estável é indicada pela seta tracejada na Fig. 7.12-2. De maneira idêntica, partindo novamente

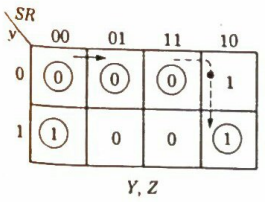


Fig. 7.12-2 A tabela de transição para o circuito da Fig. 7.12-1.

PS \ SR	00	01	11	10
a	(a)	(a)	(a)	b
b	(b)	a	a	(b)

NS

Fig. 7.12-3 A tabela (fluxo) de estados para o circuito da Fig. 7.12-1.

de $SRy = 000$, se mudássemos S para $S = 1$, novamente passaríamos pelo estado instável em $SRy = 100$ e terminaríamos no estado estável em $SRy = 101$.

Partindo do estado estável em $SRy = 101$, uma mudança para $SR = 00$ levaria ao estado $y = 1$, e uma mudança para $SR = 11$ ou $SR = 01$ levaria de volta ao estado $y = 0$.

A tabela da Fig. 7.12-2 é chamada *tabela de transição*; nela os estados foram identificados pelos níveis lógicos sobre as linhas de estado. Um tanto mais genericamente, se quisermos simplesmente denominar os estados sem fazer referência específica aos níveis lógicos, a tabela aparecerá como na Fig. 7.12-3. Os estados são aqui simplesmente identificados como estados a e b, e esta tabela pode razoavelmente ser chamada *tabela de estados*. Vemos, naturalmente, uma correspondência entre a transição presente e as tabelas de estado e entre a transição e as tabelas de estado de sistemas síncronos. Há uma restrição que se aplica no caso assíncrono (modo fundamental) e que não ocorre no caso síncrono. No caso assíncrono, em cada linha deve haver pelo menos uma entrada NS (*next state* = estado seguinte) que é a mesma que a PS (*present state* = estado presente). Do contrário, o estado contemplado não seria um estado estável. Além disto, no caso síncrono, PS e NS referem-se aos estados em um período de relógio e no período sucessivo. No caso assíncrono, PS e NS referem-se aos níveis lógicos nas linhas y e Y. No caso assíncrono, a tabela da Fig. 7.12-3 é muitas vezes chamada *tabela de fluxo*.

7.13 UM EXEMPLO DE PROJETO

Quando analisamos um circuito no caso síncrono, fomos do circuito para uma tabela de transição, para uma tabela de estados e, depois, para um diagrama de estados. No processo de síntese, invertemos a ordem. Partindo de uma descrição verbal dos requisitos, deduzimos logicamente uma tabela de estados e voltamos a trabalhar com o circuito. Um procedimento semelhante é apropriado no presente caso assíncrono, exceto que geralmente podemos desviar-nos do diagrama de estados e partir da tabela de estados.

Como um exemplo simples de projeto, consideremos um circuito que deve ter uma única entrada X e duas saídas Z_1 e Z_0. O circuito deve contar, em módulo 4, o número de inversões de nível lógico sobre a entrada X. Assim, partindo de $X = 0$ e $Z_1Z_0 = 00$, requeremos que, quando X passe pela seqüência $X = 0 \to 1 \to 0 \to 1 \to 0$ etc., a saída passe pela seqüência $Z_1Z_0 = 00 \to$ $\to 01 \to 10 \to 11 \to 00$ etc.

É evidente que necessitamos de quatro estados estáveis e somos levados inevitavelmente ao diagrama de estados da Fig. 7.13-1*a*. Quando $X = 0$, estamos num estado estável a e a saída é $Z_1Z_0 = 00$. Quando X varia de 0 a 1, queremos ir para o estado b com $Z_1Z_0 = 01$. De b queremos ir para c, e dali para d, e finalmente de volta para a. Na Fig. 7.13-1*b* fizemos uma atribuição de

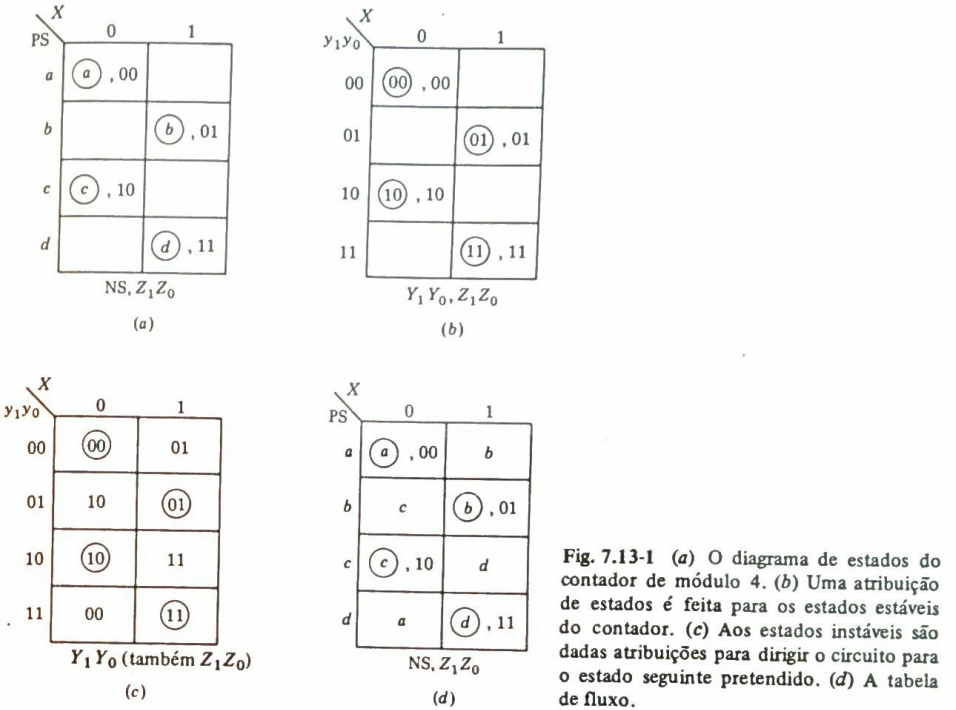


Fig. 7.13-1 (*a*) O diagrama de estados do contador de módulo 4. (*b*) Uma atribuição de estados é feita para os estados estáveis do contador. (*c*) Aos estados instáveis são dadas atribuições para dirigir o circuito para o estado seguinte pretendido. (*d*) A tabela de fluxo.

estado para que o estado estável a tenha a atribuição $Y_1Y_0 = y_1y_0 = 00$ etc. A atribuição foi selecionada para fazer $Y_1Y_0 = Z_1Z_0$ em cada caso, de modo que o conjunto de circuitos lógicos possa ser simplificado.

Retornando agora ao estado a, notamos que, quando mudamos X de 0 para 1, vamos para uma coluna em que há *dois* estados estáveis. Como asseguraremos que terminamos no estado correto, isto é, b e não d? Um esquema que imediatamente se recomenda está indicado na Fig. 7.13-1*c*. Aqui, no local da tabela para $y_1y_0 = 00$ e $x = 1$, fizemos a entrada $Y_1Y_0 = 01$. Por conseguinte, quando o sistema estiver no estado $y_1y_0 = 00$ (estado a), uma modificação em X de 0 para 1 causa uma modificação imediata em Y_1 e Y_0 para $Y_1Y_0 = 01$. (Lembrar que admitimos que o atraso está nas conexões de realimentação e não nas portas.) Em um curto espaço de tempo mais tarde (o tempo de atraso), y_1y_0 assume os mesmos valores que Y_1Y_0, e temos $Y_1Y_0 =$ $= y_1y_0 = 01$, que é precisamente como requerido para estabelecer o circuito no estado b. Se a entrada no local $y_1y_0X = 001$ tivesse sido $Y_1Y_0 = 11$, então correspondentemente teríamos ido do estado a para o estado d. Em geral, quando nos movimentamos para uma coluna em que mais do que um estado estável está disponível, fazemos entradas apropriadas nos locais correspondentes a estados instáveis, para dirigir o sistema para o estado seguinte estável adequado. Estas direções, que não estão incluídas na tabela estável da Fig. 7.13-1*a*, estão incluídas na tabela da Fig. 7.13-1*d*.

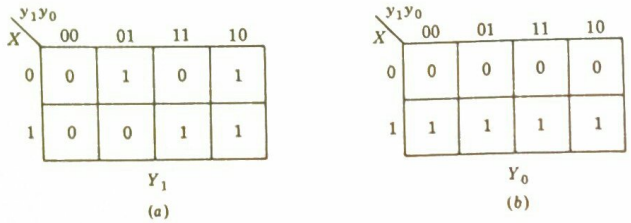


Fig. 7.13-2 Os mapas K para Y_0 e Y_1 do contador.

Na Fig. 7.13-1*d* não especificamos Z_1Z_0 na tabela em locais com estados instáveis. Nós (por tentativa) tomamos a atitude de que, desde que estes estados são de vida muito curta, os valores de Z_1Z_0 são irrelevantes (ou não-especificados).

Podemos construir agora o diagrama lógico a partir da informação na tabela da Fig. 7.13-1*c*. A partir desta tabela, podemos fazer entradas nos mapas de Karnaugh para Y_1 e Y_0 como nas Figs. 7.13-2*a* e *b*. Lemos

$$Y_0 = X \tag{7.13-1a}$$

$$Y_1 = \bar{X}\bar{y}_1y_0 + Xy_1 + y_1\bar{y}_0 \tag{7.13-1b}$$

e o circuito é desenhado na Fig. 7.13-3.

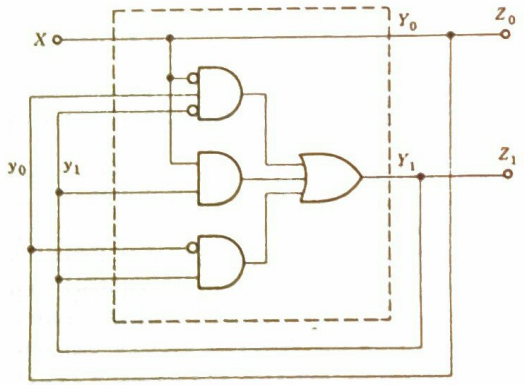


Fig. 7.13-3 O diagrama de circuito do contador de módulo 4.

7.14 CORRIDAS

Ao anteciparmos pela tabela da Fig. 7.11-1*c* que nosso circuito assíncrono realmente irá operar conforme requerido, admitimos tacitamente que o atraso Δ_0 entre y_0 e Y_0 e o atraso Δ_1 entre y_1 e Y_1 são precisamente os mesmos. Por questão de prática, não é possível fazer os dois atrasos

exatamente iguais e manter esta igualdade indefinidamente. Veremos agora que a desigualdade de atraso fará com que o circuito opere de maneira diferente da pretendida.

Suponhamos que nosso sistema esteja no estado *b* estável ($Y_1Y_0 = y_1y_0 = 01$) e que modifiquemos X de 1 para 0, sendo nossa intenção que o sistema deva ir agora para o estado *c* ($Y_1Y_0 = y_1y_0 = 10$). A mudança em X fará com que Y_1Y_0 mude *imediatamente* para $Y_1Y_0 = 10$. Se os atrasos Δ_0 e Δ_1 forem os mesmos, então, após um atraso $\Delta_0 = \Delta_1$, y_1y_0 se tornará $y_1y_0 = 10$; mas esta última mudança em y_1y_0 não causará nenhuma outra mudança em Y_1Y_0, e o sistema realmente foi estabelecido no estado *c*.

Consideremos a seguir a mesma transição admitindo que $\Delta_1 > \Delta_0$. As formas-de-onda que descrevem a seqüência de eventos são mostradas na Fig. 7.14-1. A modificação em X de $X = 1$ para $X = 0$ ocorre em $t = t_0$. Antes de $t = t_0$, o sistema está no estado *b* com $Y_1Y_0 = y_1y_0 = 01$. Em $t = t_0$, quando X muda para $X = 0$, conforme requerido pela tabela da Fig. 7.13-1*c*, há uma mudança imediata em Y_1Y_0 para $Y_1Y_0 = 10$. Depois de um tempo Δ_0, y_0 responde a Y_0 e muda para $y_0 = 0$. Neste instante ($t_1 = t_0 + \Delta_0$) temos $y_1y_0 = 00$. Da tabela achamos que, com $X = 0$ e $y_1y_0 = 00$, devemos ter $Y_1Y_0 = 00$. Por conseguinte, começando em $t = t_1$, temos $Y_1Y_0 = y_1y_0 = 00$; isto é, estamos no estado *a* e não no estado *c*.

Finalmente, após um tempo Δ_1, as mudanças em Y_1 aparecerão em y_1. Conforme indicado pelas setas curvadas, estas mudanças em y_1 induzirão outras mudanças em Y_1, que após um segundo atraso Δ_1 novamente aparecerão em y_1 etc. Portanto, haverá um intervalo de duração Δ_0 quando realmente teremos $Y_1Y_0 = y_1y_0 = 10$; assim, o sistema estará no estado *c*. Em conjunto, então, parece que o sistema oscilará entre os estados *a* e *c*. Examinando o circuito da Fig. 7.13-3, observamos que o atraso Δ_0 é nominalmente zero. Quando X se modifica, y_0 pode responder imediatamente, enquanto a modificação em y_1, se houver, requer que um sinal se propague através de dois níveis de portas. Portanto, parece que as formas-de-onda da Fig. 7.14-1, após $t = t_0 + \Delta_0$, correspondem a uma situação em que o sistema está no estado *a* ($Y_1Y_0 = y_1y_0 = 00$), mas perturbado a intervalos Δ_1 por um pulso muito estreito nas formas-de-onda de Y_1 e y_1. Este pulso de duração Δ_0 aparece em Y_1, completa a lógica e o atraso e reaparece em y_1, perseguindo a si próprio ao redor da malha ou laço de realimentação.

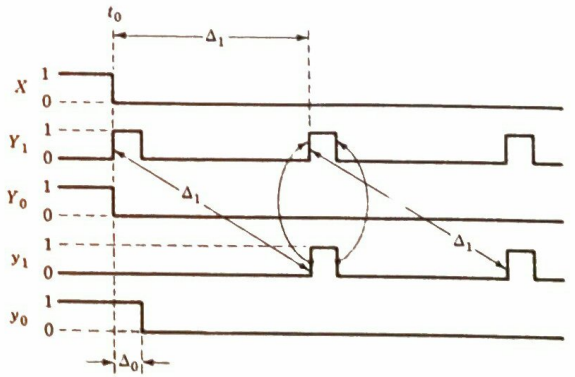


Fig. 7.14-1 Formas-de-onda do circuito da Fig. 7.13-3 para a mudança pretendida do estado *b* para *c*, supondo $\Delta_1 > \Delta_0$.

Realmente as formas-de-onda da Fig. 7.14-1 e a descrição acima de um pulso que corre ao redor da malha ou laço são irrealísticas. Num sistema real, como uma transição é propagada através das portas, ela sofre não somente atrasos de propagação, mas também um prolongamento de seu tempo de transição; o tempo de transição torna-se progressivamente mais longo a cada propagação através de uma porta. Por conseguinte, como nosso pulso estreito corre através da malha de realimentação e seu tempo de crescimento e de queda tornam-se cada vez maiores em comparação com a duração do pulso, o pulso tornar-se-á debilitado ou esgotado e desaparecerá. Em conjunto, então, quando fazemos nossa mudança em X de $X = 1$ para $X = 0$, esperando que o sistema vá de b para c, pode haver alguma breve perturbação de atividade, mas o resultado líquido é que o sistema irá em vez disso para a.

A dificuldade que acabamos de descrever resulta do fato de que a transição que procurávamos induzir requer mudanças nas duas variáveis de estado seguinte Y_1 e Y_0 e correspondentemente nas duas variáveis de estado presente. Sempre que uma variação é requerida em mais do que uma de tal variável, a situação que resulta é denominada *corrida* (*race*). As variações nos Ys correm entre si através dos elementos de atraso e inevitavelmente chegam aos ys em tempos diferentes. As corridas são chamadas críticas ou não-críticas, dependendo de elas afetarem ou não adversamente a operação do sistema. A corrida que examinamos acima era crítica. Deixa-se como exercício para o estudante verificar que, se tivéssemos suposto $\Delta_0 > \Delta_1$, a corrida resultante seria ainda crítica. Às vezes as corridas não são críticas. Por exemplo, suponhamos que nos comprometemos a fazer uma transição para um estado estável que é o único estado estável numa determinada coluna da tabela de estados. Se nos movermos para esta coluna, devemos, no final, terminar neste único estado estável. Às vezes, as corridas são críticas ou não-críticas, dependendo dos atrasos relativos, e as corridas críticas podem ser eliminadas pela judiciosa adaptação dos atrasos.

7.15 ELIMINAÇÃO DE CORRIDAS CRÍTICAS

No projeto de um circuito assíncrono, quando se desenvolvem as corridas críticas, elas devem ser eliminadas. Às vezes, esta correção pode ser feita simplesmente selecionando-se uma nova atribuição de estado. Acontece ser este o caso com nosso contador de módulo 4 da Seção 7.13. Uma nova atribuição de estado, substituindo a atribuição dada na Fig. 7.13-1, é dada na Fig. 7.15-1. Aqui conseguimos que somente uma *única* variável de estado se modifique em cada transição solicitada. Em conseqüência, não podem se desenvolver corridas. Se continuarmos a usar

y_1y_0 \ X	0	1
$a = 00$	(00)	01
$b = 01$	11	(01)
$c = 11$	(11)	10
$d = 10$	00	(10)

Y_1Y_0

Fig. 7.15-1 Uma tabela de transição para o contador de módulo 4 em que a atribuição de estado foi selecionada para evitar corridas.

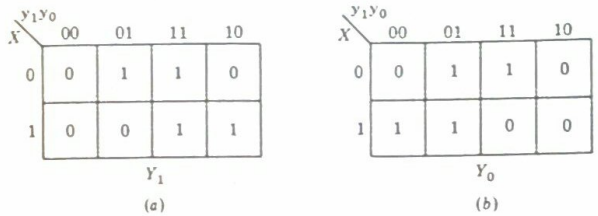


Fig. 7.15-2 Mapas K para Y_1 e Y_0 para a atribuição de estado isenta de corridas.

Y_1 e Y_0 como terminais de saída Z_1 e Z_0, o contador contará em código Gray. Os mapas de Karnaugh de Y_1 e Y_0 são dados na Fig. 7.15-2, e lemos

$$Y_1 = \bar{X}y_0 + Xy_1 \qquad (7.15\text{-}1a)$$

$$Y_0 = \bar{X}y_0 + X\bar{y}_1 \qquad (7.15\text{-}1b)$$

Mais genericamente, nenhuma atribuição de estado é possível com a qual cada transição de estado solicita apenas uma mudança numa única variável. Em tais casos há inúmeros recursos disponíveis, todos dependendo de usar mais estados do que são realmente solicitados. Ilustramos um método como exemplo.

Na Fig. 7.15-3 mostramos uma tabela de estados com quatro estados. Ordinariamente, usaríamos duas variáveis de estado. No entanto, é facilmente verificado que, neste caso, nenhuma atribuição de estado eliminará todas as corridas. Em conseqüência, usaremos três variáveis de estado y_2, y_1 e y_0, acrescentando deste modo quatro estados redundantes. Agora temos oito estados e com três variáveis de estado podemos fazer uma atribuição de estado a cada estado. O estado original a e seu estado redundante equivalente serão chamados a_1 e a_2. Similarmente, temos b_1 e b_2, c_1 e c_2, e d_1 e d_2. Naturalmente, as saídas do sistema, não especificadas na Fig. 7.15-3, são as mesmas nos estados a_1 e a_2 etc., e o mundo externo nunca saberá se o sistema está no estado a_1 ou no estado a_2 etc.

A atribuição de estado que usaremos agora em nossos oito estados é dada na Fig. 7.15-4. Se estivermos no estado a_1, podemos ir para b_1 ou c_1 sem gerar uma corrida, isto é, sem modificar mais do que uma variável de estado, mas não podemos ir para d_1. Se então estamos no estado a_1 e necessitamos ir para o estado d, podemos ir para o estado d_2, que externamente não é dis-

PS \ X_1X_0	00	01	11	10
a	(a)	(a)	c	d
b	a	c	(b)	(b)
c	d	(c)	(c)	b
d	(d)	c	b	(d)

Fig. 7.15-3 Uma tabela de estados para a qual nenhuma atribuição de quatro estados produzirá uma atribuição isenta de corridas.

y_0 \ y_2y_1	00	01	11	10
0	a_1	b_1	c_2	d_2
1	c_1	d_1	a_2	b_2

Fig. 7.15-4 A atribuição de estado para os oito estados usados para evitar corridas.

tinto de d_1. Se de d_2 necessitamos ir para um estado b, não podemos ir para b_1, mas podemos ir para b_2. Em conjunto, como é facilmente verificado, podemos fazer qualquer transição sem gerar uma corrida. A tabela de oito estados expandida é mostrada na Fig. 7.15-5. Como um exemplo de como chegarmos às entradas, consideremos a primeira linha na Fig. 7.15-3. Aqui achamos que, se estivermos no estado a e X_1X_0 tornar-se $X_1X_0 = 11$, iremos para o estado c. Na tabela da Fig. 7.15-4 achamos que podemos ir para c_1, mas não para c_2. Por conseguinte, a entrada c_1 na coluna $X_1X_0 = 11$. No entanto, novamente da Fig. 7.15-4, achamos que não podemos ir de a_1 para d_1. Por conseguinte, vamos de a_1 para d_2 e, correspondentemente, a entrada sob $X_1X_0 = 10$ na tabela da Fig. 7.15-5 é d_2.

	$y_2y_1y_0$	X_1X_0 00	01	11	10
a_1	0 0 0	(a_1)	(a_1)	c_1	d_2
b_1	0 1 0	a_1	c_2	(b_1)	(b_1)
c_1	0 0 1	d_1	(c_1)	(c_1)	b_2
d_1	0 1 1	(d_1)	c_1	b_1	(d_1)
a_2	1 1 1	(a_2)	(a_2)	c_2	d_1
b_2	1 0 1	a_2	c_1	(b_2)	(b_2)
c_2	1 1 0	d_2	(c_2)	(c_2)	b_1
d_2	1 0 1	(d_2)	c_2	b_2	(d_2)

Fig. 7.15-5 A tabela de quatro estados da Fig. 7.15-3 expandida numa tabela de oito estados isenta de corridas.

7.16 UM EXEMPLO

No exemplo do contador de módulo 4 discutido acima foi óbvio, naturalmente, que quatro estados estáveis eram requeridos e a geração da tabela de estados da Fig. 7.13-1 foi igualmente óbvia. Nesta seção consideraremos um exemplo menos simples.

O problema é enunciado como se segue. Requeremos um circuito assíncrono com duas entradas X_1 e X_0 e uma única saída Z. Enquanto $X_0 = 0$, a saída Z deve ser $Z = 0$ e não deve responder a variações em X_1. A primeira variação em X_1 que ocorre enquanto $X_0 = 1$ deve fazer com que Z se torne $Z = 1$. Uma vez que Z foi para $Z = 1$, Z deve novamente ser insensível a X_1 e Z deve retornar a $Z = 0$ somente quando X_0 retornar a $X_0 = 0$. Admitiremos que X_1 e X_0 não variem ao mesmo tempo e que as variações em X_1 e X_0 são separadas por um intervalo adequado para assegurar que o sistema seja estável antes de cada variação em qualquer variável de entrada. As formas-de-onda de X_1, X_0 e Z num caso típico são mostradas na Fig. 7.16-1.

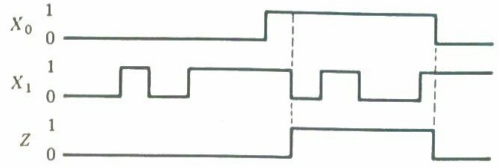


Fig. 7.16-1 Formas-de-onda que ilustram o comportamento específico de um sistema no modo fundamental.

Ao refletirmos sobre o projeto de um circuito síncrono, achamos útil começar por um diagrama de estados que foi depois convertido numa tabela de estados. Em circuitos assíncronos geralmente revela-se mais conveniente pular o diagrama de estados e, em vez disto, começar por uma tabela de estados. Num diagrama de estados, seja num caso síncrono ou num caso assíncrono, há uma linha separada para cada estado. Daí, ao raciocinarmos sobre a operação do circuito, cada vez que suspeitamos de que geramos um novo estado, isto é, imaginamos ter gerado algo novo a ser lembrado, devemos fornecer uma nova linha. Um modo de ajudar a assegurar que não foram omitidos estados essenciais consiste em fornecer um novo estado e uma nova linha correspondente cada vez que mudarmos uma variável de entrada, exceto quando estivermos confiantes de que uma mudança nos faz voltar a um estado já estabelecido. Um tal procedimento leva a uma tabela de transição com apenas um estado estável por linha, chamada tabela de transição *primitiva*.

Uma tabela primitiva para nosso problema é dada na Fig. 7.16-2. Conforme o costume, um estado estável é envolvido por um círculo. Em cada caso um estado estável envolvido introduzido na tabela é o mesmo que o estado presente listado ao longo do lado esquerdo da tabela. Também introduzimos a saída Z para cada estado estável. Partindo de um estado a com $X_1X_0 = 00$, uma

PS \ X_1X_0	00	01	11	10
a	(a) /0	b/–	–/–	c/–
b	a/–	(b) /0	d/–	–/–
c	a/–	–/–	e/–	(c) /0
d	–/–	f/–	(d) /1	c/–
e	–/–	f/–	(e) /0	c/–
f	a/–	(f) /1	d/–	–/–

NS/saída, Z

Fig. 7.16-2 Uma tabela de fluxos primitiva para um sistema especificado.

mudança em X_0 leva-nos a um estado b, a saída Z permanecendo em $Z = 0$. No quadrículo correspondente a PS a e $X_1X_0 = 01$, introduzimos o estado instável b para assegurar que, quando vamos de a na coluna $X_1X_0 = 00$ para a coluna $X_1X_0 = 01$, realmente terminamos em b em vez de em algum outro estado estável que possa aparecer na coluna $X_1X_0 = 01$. Conforme fizemos num caso anterior, tentamos deixar como irrelevante o item de dados de entrada extraído no estado instável. Novamente, de a uma mudança para $X_1X_0 = 10$ leva-nos a um estado estável c. Admitimos que não mudaremos X_1 e X_0 simultaneamente. Por conseguinte, na linha a e na coluna $X_1X_0 = 1$ deixamos tanto o estado quanto a saída como irrelevantes. De b uma outra mudança para $X_1X_0 = 11$ faz a saída ir para $Z = 1$, e assim fornecemos um novo estado d obviamente requerido com saída $Z = 1$. Por outro lado, partindo de c, uma mudança para $X_1X_0 = 1$ não faz com que $Z = 1$, porque a ordem de mudança de X_1 e X_0 não é conforme a requerida. Daí, uma tal mudança para $X_1X_0 = 1$ deve levar-nos a um outro estado diferente de d. Este novo estado é e, e a saída correspondente é $Z = 0$. Se de e mudarmos agora X_1 de volta a $X_1 = 0$, temos uma mudança em X_1 quando $X_0 = 1$ e devemos ir para um estado em que novamente $Z = 1$. Este estado é f. As entradas restantes na tabela são verificáveis sem dificuldade.

7.17 ELIMINAÇÃO DE ESTADOS REDUNDANTES

Podemos antecipar que haverá estados redundantes na tabela da Fig. 7.16-2. Infelizmente, os métodos dados inicialmente para eliminação de tais estados não são adequados ao presente caso por causa das entradas irrelevantes (ou não-especificadas). Num mapa de Karnaugh, em que uma condição irrelevante é decomposta ou em um 0 ou em um 1, podemos facilmente decidir como usar as condições irrelevantes (*don't-cares*) para possibilitar a simplificação ótima. Numa tabela em que uma condição irrelevante pode ser decomposta de tantas maneiras quantos são os estados, a questão é mais complicada. Apresentamos agora um novo processo de redução que é eficiente mesmo quando as condições irrelevantes estiverem presentes. Desenvolveremos o procedimento usando a tabela da Fig. 7.16-2 como uma ilustração.

Na Fig. 7.17-1*a* preparamos uma carta que nos permite comparar cada estado com qualquer *outro* estado. Na tabela de estados, quando comparamos a linha a com a linha b, achamos que estamos inteiramente em liberdade para ajustar as condições irrelevantes de modo que os estados a e b possam tornar-se o mesmo único estado. Em cada coluna, ou as entradas são as mesmas, ou, se houver uma entrada numa coluna de uma linha, a mesma coluna da outra linha tem uma condição irrelevante. Na carta indicamos esta *compatibilidade* introduzindo uma marca de prova. As outras marcas de prova na carta resultam da mesma consideração aplicada às linhas apropriadas.

A seguir, consideremos as linhas b e c. Observemos da coluna $X_1X_0 = 11$ que, se supusermos que os estados b e c devem ser tornados equivalentes, exigiríamos que o estado d fosse equivalente ao estado e. Assim, uma hipótese de que $b = c$ *implica* $d = e$. Do mesmo modo, todos os outros pares de linhas são examinados para verificar quais equivalentes estão implícitos pelas equivalências supostas, e os resultados são introduzidos na *carta de implicações* da Fig. 7.17-1. Notar que a equivalência suposta $b = e$ implica tanto $b = f$ como $d = e$. Finalmente, achamos que $b = f$ *não* é possível pela razão de que, na coluna $X_1X_0 = 01$, achamos $Z = 0$ num caso e $Z = 1$ no outro; analogamente, devido à diferença na saída $d \neq e$. Estas duas desigualdades ($b \neq f$ e $d \neq e$) são introduzidas como cruzes na carta de implicação da Fig. 7.17-1*a*.

A seguir, redesenhamos a carta de implicação na Fig. 7.17-1*b*. Uma vez que temos o resultado da Fig. 7.17-1*a* de que $d \neq e$ e $b \neq f$, completamos a carta na Fig. 7.17-1*b* colocando cruzes nos quadrículos que têm entradas d, e ou b, f, ou ambas. No caso presente achamos que todos os quadrículos estão com cruzes, exceto aqueles com verificações. (Num caso mais geral, esta segunda passagem geraria novas desigualdades e passagens adicionais seriam requeridas. O processo continua até que não sejam geradas novas desigualdades.)

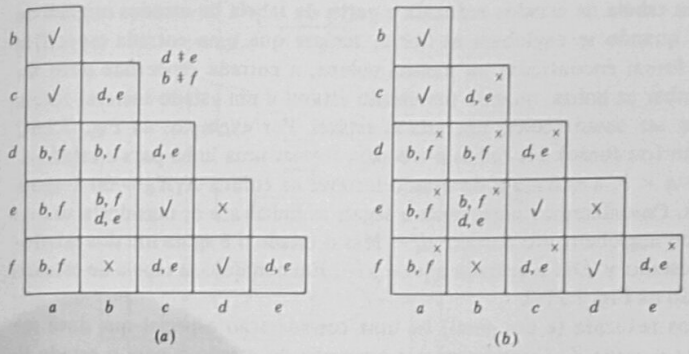


Fig. 7.17-1 Cartas de implicação usadas num procedimento para eliminar estados redundantes da tabela da Fig. 7.16-2. (*a*) Informação introduzida diretamente da tabela de fluxos primitiva. (*b*) As desigualdades em *a* levaram a outras desigualdades.

Achamos agora dos quadrículos sem cruzes na Fig. 7.17-1*b* que os seguintes pares de estados são compatíveis uns com os outros:

$$(a, b), (a, c), (c, e) \quad \text{e} \quad (d, f)$$

Note que o que temos tabelado aqui é a *compatibilidade* e não a equivalência. Se a fosse equivalente a b, $a = b$, e se $a = c$, teríamos também $b = c$, que, conforme aparece na Fig. 7.15-1*b*, não é o caso. Na base destas compatibilidades, exploramos agora o que aconteceria se decompusésssemos as condições irrelevantes (ou não-especificadas) de tal maneira que $a = b$, $c = e$ e $d = f$; assim, englobamos os estados compatíveis (a, b) num estado α, os estados compatíveis (c, e) num estado β e os estados compatíveis (d, f) no estado γ. Estes três estados α, β e γ incluem todos os estados originais. Ignoramos o par compatível (a, c) primeiro porque a hipótese $a = c$ é incoerente com a hipótese $a = b$. Além disto, se tivéssemos usado (a, c), (c, e) e (d, f), o estado b não seria levado em conta.

PS \ X_1X_0	00	01	11	10
α	(α) /0	(α) /0	γ/–	β/–
β	α/–	γ/0	(β) /0	(β) /0
γ	α/–	(γ) /1	(γ) /1	β/–

NS/saída, Z

Fig. 7.17-2 A tabela de estados reduzida.

Construamos agora uma tabela de estados reduzida a partir da tabela de estados original da Fig. 7.16-2. Evidentemente, quando se englobam as linhas, sempre que uma entrada específica e uma condição irrelevante forem encontradas na mesma coluna, a entrada específica deve ser listada. Além disto, ao englobar as linhas, quando um estado estável e um estado instável forem encontrados, a entrada deve ser listada como um estado estável. Por exemplo, na Fig. 7.16-2 consideremos o englobamento (ou fusão) das linhas a e b para formar uma linha para o estado α. Uma vez que decidimos que $a = b$, a entrada a do estado instável na coluna $X_1X_0 = 00$ é agora realmente o estado a estável. Consideremos novamente a seguir as linhas a e b, mas desta vez na coluna $X_1X_0 = 1$. A fusão ou englobamento fornece $d/-$. Mas o estado d é agora um dos estados englobados ou fundidos no estado γ. Daí a entrada aqui é $\gamma/-$. Em conjunto, a tabela de estados reduzida é conforme mostrado na Fig. 7.17-2.

Nesta tabela de estados reduzida (e em geral) há uma consideração especial que deve ser levada em conta e que afeta a saída Z. Consideremos a transição do estado α para o estado β. Em ambos os estados, inicial e final, $Z = 0$. No entanto, na transição o sistema passa pelo estado β instável, onde a saída é deixada como uma condição irrelevante. Portanto, é possível que, na transição, Z possa ir de $Z = 0$, passe transitoriamente por $Z = 1$ e se estabeleça em $Z = 0$. Geralmente, preferiríamos evitar a breve e desnecessária transição, e podemos fazer isto substituindo a condição irrelevante pelo nível lógico Z nos estados inicial e final. Por conseguinte, é apropriado na Fig. 7.17-2 substituir Z = condição irrelevante por $Z = 0$ na linha α da coluna $X_1X_0 =$ $= 10$. Quando, por outro lado, deve haver uma variação em Z, podemos igualmente deixar $Z =$ $=$ condição irrelevante no estado intermediário. Se esta condição irrelevante se revelar ser um 1 ou um zero, ela simplesmente determina se a variação em Z ocorre um bit mais cedo ou um bit mais tarde.

Na Fig. 7.17-3 selecionamos uma atribuição de estado e construímos uma tabela de transição correspondente.

y_1y_0 \ X_1X_0	00	01	11	10
$\alpha = 00$	(00) /0	(00) /0	01/–	11/0
$\beta = 11$	00/–	01/0	(11) /0	(11) /0
$\gamma = 01$	00/–	(01) /1	(01) /1	11/–

Y_1Y_0/Z

Fig. 7.17-3 A tabela de transição correspondente à atribuição de estado $\alpha = 00, \beta = 11$ e $\gamma = 01$.

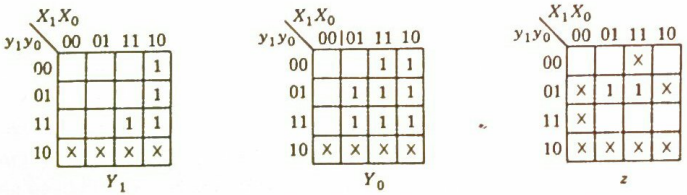


Fig. 7.17-4 Mapas K para as variáveis Y_1 e Y_0 de estado e para a saída Z.

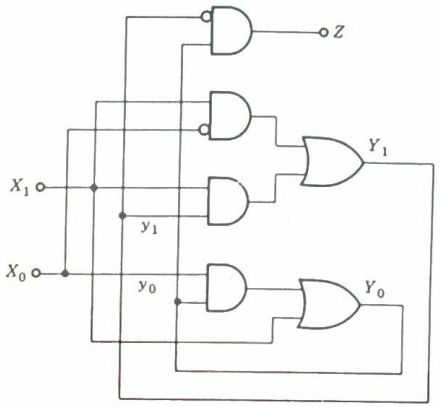


Fig. 7.17-5 O diagrama de circuito.

Modificamos Z na casa um, conforme notado acima, e evitamos corridas arranjando para que somente uma única variável seja envolvida em cada mudança de estado. Os mapas de Karnaugh de Y_1, Y_0 e Z são dados na Fig. 7.17-4. Lemos

$$Y_1 = X_1\bar{X}_0 + X_1y_1 \qquad (7.17\text{-}1a)$$

$$Y_0 = X_1 + X_0y_0 \qquad (7.17\text{-}1b)$$

$$Z = \bar{y}_1y_0 \qquad (7.17\text{-}1c)$$

Este circuito lógico é dado na Fig. 7.17-5.

7.18 OUTRO EXEMPLO DE ELIMINAÇÃO DE ESTADOS REDUNDANTES

Na seção anterior descrevemos um procedimento pelo qual os estados redundantes na tabela da Fig. 7.16-2 podem ser eliminados. Aquele exemplo não proporcionou nenhuma oportunidade de fazer todos os comentários indispensáveis à explicação do procedimento em estados redundantes quando estiverem envolvidas condições irrelevantes. O assunto é de interesse porque condições irrelevantes também podem ocorrer em tabelas de estado em sistemas síncronos. Em conseqüência, consideremos um segundo exemplo.

Consideremos, então, a tabela de fluxos primitiva da Fig. 7.18-1*a*. A carta de implicações é dada na Fig. 7.18-1*b*. Onde houver apenas implicações tautológicas, não fazemos nenhuma entrada. (Por exemplo, achamos que a hipótese de que os estados 2 e 3 são os mesmos, o que implica que apenas o estado 2 é o mesmo que 2 e o estado 3 é o mesmo que 3; daí, nenhuma entrada é feita. Similarmente, achamos que a hipótese de que os estados 8 e 9 são os mesmos, o que implica que 8 e 9 são os mesmos.) Os quadrículos sem entradas são marcados por verificações. Quando achamos estados com saídas diferentes na mesma coluna, no correspondente quadrículo da carta de implicação, isto é, o quadrículo na intersecção da linha e coluna correspondentes, está feita uma cruz. Temos a informação conforme observado na Fig. 7.18-1*b* que $1 \neq 10$, $2 \neq 4$, $2 \neq 6$, $3 \neq 8$ e $3 \neq 9$. Em conseqüência completamos a carta fazendo cruzes em todos os quadrículos

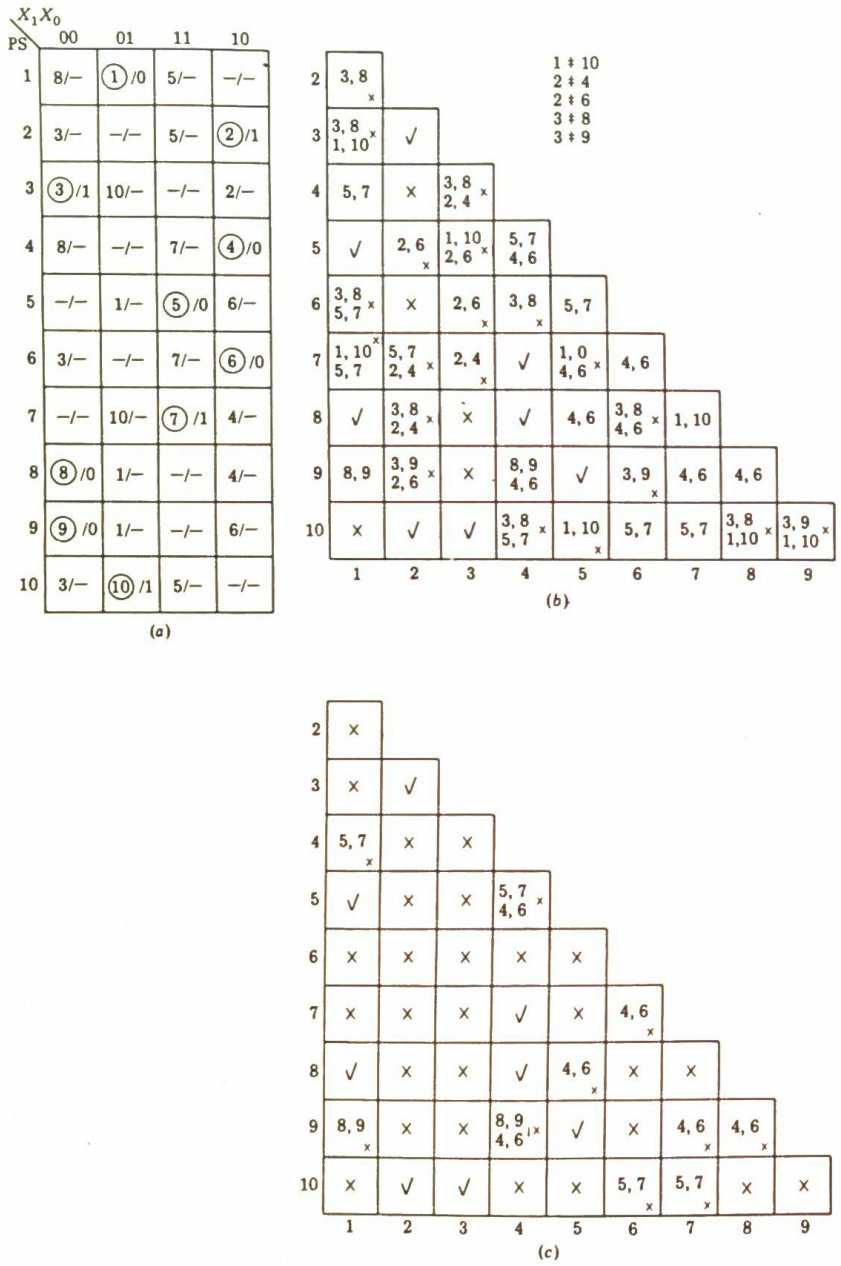


Fig. 7.18-1 (*a*) Um exemplo de uma tabela de fluxos primitiva. (*b*) Desenvolvimento parcial da carta de implicação. (*c*) Conclusão do desenvolvimento da carta de implicação.

em que ocorrem tais estados aos pares. Este processo de fazer cruzes gera novas desigualdades. Necessitamos, portanto, completar a carta novamente para fazer cruzes, nesta segunda passagem, nos quadrículos cujas entradas incluem estas novas desigualdades. Por conveniência, antes de fazer esta segunda passagem, reproduzimos a carta na Fig. 7.18-1*c* eliminando todas as entradas nos quadrículos assinalados com cruzes.

Agora observemos sistematicamente a carta da Fig. 7.18-1*c*, partindo, por exemplo, do canto inferior direito. O primeiro quadrículo, que notamos que ainda tem uma entrada, é o quadrículo na intersecção dos estados 8 e 9, e a entrada é 4, 6. Olhamos agora para o quadrículo correspondente aos estados 4 e 6 e achamos que ele está assinalado com cruz. Por conseguinte, o estado 8 não é equivalente ao estado 9 e as novas cruzes são então geradas para o quadrículo na intersecção dos estados 8 e 9 e para todos os outros quadrículos com entradas 4 e 6. Similarmente, trouxemos da Fig. 7.16-1*b* a informação de que $5 \neq 7$; assim, acrescentamos cruzes igualmente nos quadrículos com a entrada 5, 7.

Se tivéssemos construído uma terceira carta, acharíamos que somente os quadrículos com entradas 8, 9 não foram assinalados com cruzes. Mas achamos que o quadrículo na intersecção dos estados 8 e 9 foram assinalados com cruzes. Por conseguinte, marcamos com cruzes também os quadrículos com entradas 8, 9. (Não temos desenhada uma terceira carta, mas marcamos com cruzes estes quadrículos na Fig. 7.16-1*c*.) Portanto, por meio de exemplo, tendo achado inicialmente que $3 \neq 8$, porque estes estados têm saídas diferentes para $X_1X_0 = 00$, determinamos que $4 \neq 6$; e porque $4 \neq 6$, achamos ainda que $8 \neq 9$.

Podemos agora ler da Fig. 7.16-1*c* quais dos estados *não* são incompatíveis com outros. (Revela-se aqui que os únicos quadrículos da carta de implicação que não estão marcados com cruz são aqueles que foram verificados originalmente. Num caso mais extenso, de maneira geral, tal não ocorrerá, e alguns dos quadrículos com entradas também permanecerão sem cruzes.) Achamos que temos os seguintes pares de estados compatíveis, incluindo o estado 6, que é compatível apenas consigo próprio:

(5, 9) (2, 10) (1, 8)
(4, 8) (2, 3) (1, 5)
(4, 7) (3, 10) (6)

Com vista a reduzir o número de estados a um mínimo, tentamos agora formar uma *classe compatível máxima*. Uma classe compatível é um grupo de estados, *cada* um dos quais sendo compatível com *cada* um dos outros no grupo. (Lembremo-nos de que, se *a* for compatível com *b* e *b* com *c*, desta maneira não está implícito que *a* seja compatível com *c*. Daí a formação de uma classe compatível requerer que cada membro prospectivo da classe seja verificado quanto à compatibilidade contra cada outro membro da classe.) Uma classe compatível é *máxima* quando não é possível achar quaisquer membros adicionais da classe. Achamos no presente caso que a única classe máxima que podemos formar com mais que dois membros é a classe (2, 3, 10); de modo que as classes máximas são

(2, 3, 10)
(5, 9) (1, 8)
(4, 8) (1, 5)
(4, 7) (6)

Propomos agora usar algumas destas classes compatíveis como estados de uma tabela de estados reduzida. Devemos, naturalmente, selecionar classes suficientes para assegurar que cada estado esteja incluído na tabela de estados. Imediatamente achamos que a seleção mínima que podemos fazer que satisfaça a este requisito é a seleção de cinco estados que designamos como se segue:

$$\begin{aligned} a &= (2, 3, 10) \\ b &= (5, 9) \\ c &= (4, 7) \\ d &= (1, 8) \\ e &= (6) \end{aligned} \qquad (7.18\text{-}1)$$

Observamos que, se não tivéssemos combinado as classes (2, 10), (2, 3) e (3, 10) na única máxima compatível, teríamos requerido seis estados.

Encerramento

A maneira pela qual selecionamos os estados de uma tabela proposta de estados reduzidos ainda não garante que a seleção é aceitável. Este é mais um requisito que deve ser satisfeito, chamado condição de *encerramento*.

Suponhamos, por exemplo, que o estado presente de nosso sistema é $a = (2, 3, 10)$. Quando quisermos saber o que é o estado seguinte em qualquer coluna, olhamos para a tabela de estados original. Olhamos para ver o estado seguinte ao estado 2 ou ao estado 3 ou ao estado 10. A condição de encerramento impõe o requisito óbvio, da coerência, que em cada caso o estado seguinte na tabela original deve estar no mesmo estado na tabela reduzida. Assim, suponhamos, com referência aos nomes de estados na Eq. (7.18-1), que achamos que os estados presentes 2, 3 e 10 vão para os estados seguintes que são ou 5 ou 9. Então o estado presente a vai para o estado seguinte b. Mas suponhamos que os estados 2 e 3 vão para 5 e 9, respectivamente, mas o estado 10 vai para o estado 4. Então estaremos numa perda para uma entrada coerente no estado seguinte na tabela reduzida. A condição de encerramento terá sido violada, e nossa seleção de estados reduzidos é inaceitável.

Nossa intenção em formar classes máximas de compatíveis foi a de ser capaz de incluir todos os estados originais num número mínimo de estados reduzidos. Por causa do requisito de encerramento, esta preocupação com a maximização das classes às vezes invalida seu propósito. Pode-se ressaltar que há uma vantagem no uso de classes compatíveis que não sejam máximas. Um exemplo de um tal caso é dado no Prob. 7.18-1.

No caso presente, podemos facilmente verificar que nossa seleção de estados para a tabela reduzida satisfaz a condição de encerramento, e a tabela reduzida é dada na Fig. 7.18-2.

Em resumo, o procedimento para eliminação de estados redundantes de uma tabela de estados que tem entradas irrelevantes é como se segue:

1. Preparamos uma carta de implicações que faz a provisão de cada par dos estados listados no diagrama de estados. Nesta carta introduzimos as implicações de admitir que cada membro de um par é equivalente ao outro membro. Quando os estados de um par tive-

PS \ X_1X_0	00	01	11	10
a	(a) /1	(a) /1	b/–	(a) /1
b	(b) /0	d/–	(b) /0	e/–
c	d/–	a/–	(c) /1	(c) /0
d	(d) /0	(d) /0	b/–	c/–
e	a/–	–/–	c/–	(e) /0

Fig. 7.18-2 A tabela de estados reduzida.

rem saídas diferentes, estes estados não podem ser equivalentes e introduzimos uma cruz no quadrículo correspondente da carta.

2. A seguir atravessamos a carta, quadrículo por quadrículo, fazendo cruzes nos quadrículos com entradas de pares de estados já estabelecidas como não-equivalentes; isto é, removemos as incoerências. Este processo é continuado, passagem após passagem pela carta, sendo feito até que não restem mais outras incoerências.
3. Listamos todos os estados compatíveis restantes e os agrupamos em classes compatíveis máximas.
4. Por tentativa, adotamos, como candidatos à tabela de estados reduzida, o menor número possível destas classes compatíveis máximas coerentes com a condição de que cada estado original seja incluído pelo menos uma vez e que a restrição de encerramento seja satisfeita.
5. Pelo método das tentativas, verificamos se o número de estados reduzidos pode ser reduzido usando-se menos do classes máximas para possibilitar mais fácil satisfação da restrição de encerramento.

7.19 PERIGO E CIRCUITOS ASSÍNCRONOS

Já observamos na Seção 4.6 que atrasos de propagação desiguais nas portas podem dar origem a perigos. Um perigo é uma breve excursão a um nível lógico não pretendido. Consideremos, por exemplo, a função lógica definida pelo mapa K da Fig. 7.19-1*a*. Combinamos os termos mínimos (mintermos) m_2 e m_6 em $B\overline{C}$ e os termos mínimos (mintermos) m_5 e m_7 em AC. Na Fig. 7.19-1*b* a porta p gera AC e a porta q gera $B\overline{C}$. Suponhamos agora que as entradas sejam $ABC = 111$, correspondendo a m_7, e que mudamos C para $C = 0$ de modo que ABC se torne $ABC = 110$. Esta transição está indicada no mapa K por uma seta. Inicialmente, teremos p_0, a saída da porta p, como $p_0 = 1$, e a saída de q será $q_0 = 0$, de modo que $Z = 1$. Depois da mudança em C, teremos $p_0 = 0$ e $q_0 = 1$, de modo que Z permanecerá $Z = 1$. No entanto, se p_0 mudar para 0 antes que q_0 mude para 1, durante um intervalo teremos $Z = 0$. Uma tal situação, em que a saída deve

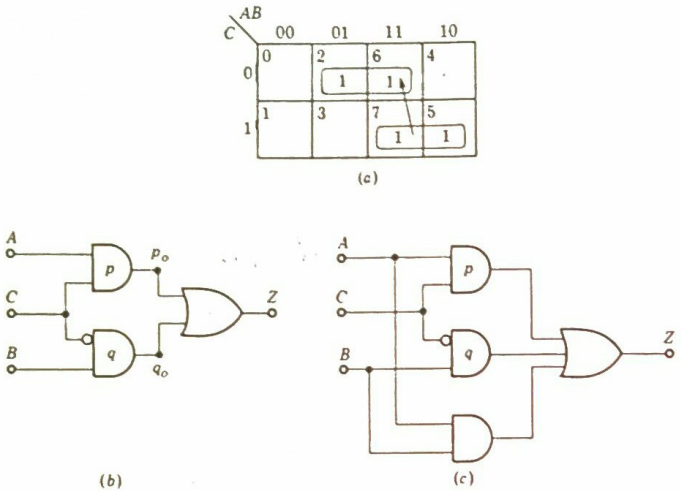


Fig. 7.19-1 (*a*) Um mapa K cuja realização física requer duas portas AND. (*b*) A realização física. (*c*) Uma terceira porta é acrescentada para remover um perigo ou risco.

permanecer em 1 mas se torna 0 transitoriamente, é chamada *perigo estático* nos 1s. Circuitos com perigos estáticos nos 0s são, naturalmente, igualmente possíveis. Nos circuitos síncronos, conforme temos visto, os perigos não necessitam causar nenhum problema, mas nos circuitos do modo fundamental os perigos podem causar transições não desejadas de estado a estado. Em tais circuitos é, portanto, necessário verificar se há perigos (quer pela análise, quer pela experiência) para determinar se eles causam operação inadequada e, se assim for, tomar providências para suprimi-los.

Como pode ser visto nas Figs. 7.19-1*a* e *b*, o perigo é gerado porque a mudança em C representa uma transferência do par de mintermos $m_5 + m_7$ para o par $m_2 + m_6$. Por correspondência, a saída $Z = 1$ depende primeiro da saída p_0, sendo $p_0 = 1$, e depois da saída q_0 , sendo $q_0 = 1$. O recurso consiste em acrescentar um par de mintermos *redundantes*, $m_6 + m_7 = AB$, e correspondentemente uma porta redundante que gere um 1 lógico independentemente do valor de C. Uma tal porta acrescentada é mostrada na Fig. 7.19-1*c*, que está agora livre do perigo. Este esquema de acrescentar uma porta redundante pode ser usado bem genericamente para retirar os perigos estáticos.

Num segundo tipo de um perigo, denominado *perigo dinâmico*, uma mudança numa saída é realmente pretendida, ou seja, a mudança $0 \rightarrow 1$. No entanto, a resposta do circuito é $0 \rightarrow 1 \rightarrow 0 \rightarrow \rightarrow 1$. Assim, uma vez que a mudança tenha ocorrido, há um breve retorno ao nível inicial antes de uma fixação no novo nível. Como os perigos (ou riscos) estáticos, os perigos (ou riscos) dinâmicos podem causar operação inadequada nos circuitos assíncronos. Revela-se, felizmente, que, se a estrutura lógica de um circuito do modo fundamental é uma estrutura de porta de dois níveis que gera somas de produtos e se conseguirmos eliminar os perigos ou riscos estáticos nos 1s, a estrutura também estará livre de perigos ou riscos dinâmicos.

Um tipo adicional de perigo que pode ser encontrado num circuito assíncrono é chamado

perigo essencial. Consideremos um circuito com pelo menos dois caminhos de realimentação com variáveis de estado y_1 e y_0. Pode acontecer que a geração e a realimentação para fornecer y_1 sejam relativamente tão rápidas que y_1 mude antes que o conjunto de circuitos que gera y_0 tenha sido capaz de responder completamente à mudança na entrada. Uma tal circunstância pode novamente causar operação defeituosa. Ela não é corrigível pelo acréscimo de portas redundantes e requer, em vez disto, ajustes dos atrasos.

CAPÍTULO 8

CONTROLADORES

No Cap. 7 introduzimos algumas idéias úteis e importantes concernentes a sistemas lógicos seqüenciais, isto é, sistemas com memória. Para ilustrar o projeto de sistemas seqüenciais, usamos o exemplo do detector de seqüências. Um tipo de sistema seqüencial que tem uma faixa de aplicabilidade muito mais ampla é o *controlador*. Os controladores são sistemas seqüenciais que fornecem níveis lógicos apropriados em tempos apropriados para controlar uma seqüência de operações lógicas simples que, juntas, executam uma operação complicada. Consideremos primeiro como estas operações lógicas simples podem ser efetuadas.

8.1 TRANSFERÊNCIAS DE REGISTRADORES

As operações elementares, lógicas ou aritméticas, que podem ser efetuadas sobre palavras lógicas são muito simples. Uma operação básica típica e a mais importante consiste simplesmente na transferência do conteúdo de um registrador para um segundo registrador. Um mecanismo de uma tal transferência é mostrado na Fig. 8.1-1, onde temos dois registradores, o registrador A e o registrador B. Os registradores são considerados aqui como consistindo de um arranjo de inúmeros latches estáticos (RS) de set-reset, como na Fig. 4.2-1. Somente um latch do registrador A, o latch A_i, e um do registrador B, o latch B_i, estão mostrados. Os outros latches estão representados pelos pontos que aparecem em ambos os lados dos latches. Duas portas AND estão interpostas entre os latches. Estas portas AND podem ser habilitadas elevando-se o *terminal de controle* assinalado "Move A para B" ao nível lógico 1. Em tal habilitação, S de B_i assumirá o valor Q de A_i, e R assumirá o valor $\bar{Q}$ de A_i. O latch B_i assumirá, portanto, o estado de A_i. Quando o terminal de controle "Move A para B" retorna ao nível lógico 0, tanto R quanto S tornar-se-ão 0 e o estado de A_i será retido em B_i. Em resumo, a colocação do terminal de controle em seu nível ativo gera um

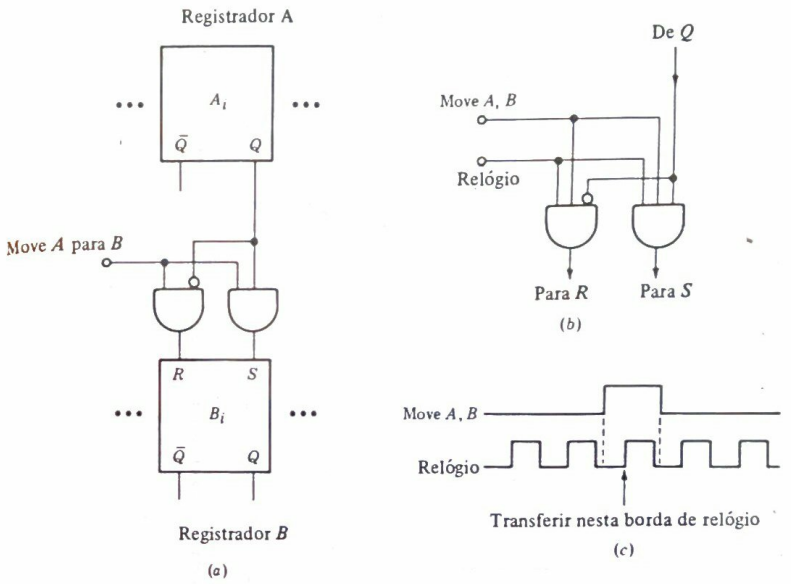


Fig. 8.1-1 (*a*) Elevação do terminal de controle "Move A para B" para lógica 1 brevemente movimenta o conteúdo do registrador A para o registrador B. (*b*) Um relógio é acrescentado para operação síncrona. (*c*) Temporização da operação de movimentar.

comando a que o circuito responde. O terminal de controle é em conseqüência muitas vezes denominado *terminal de comando*. Notar que neste processo a palavra armazenada no registrador A permanecerá inalterada, mas, naturalmente, qualquer palavra armazenada no registrador B antes da transferência será perdida.

Se, por exemplo, os registradores A e B forem registradores de dezesseis bits, então são requeridas dezesseis linhas de conexão, bem como dezesseis pares de portas AND. A inversão exigida na porta AND que precede a entrada R seria desnecessária se tivéssemos escolhido acrescentar uma segunda linha de conexão da saída $\bar{Q}$ de A_i. Escolhemos não fazer isto com base em que fisicamente a implementação de uma inversão muitas vezes é mais simples do que o acréscimo de uma conexão.

Mais importante, notamos que transferir o conteúdo do registrador A para o registrador B requer simplesmente que em curto tempo habilitemos um arranjo de portas mudando o nível lógico de um terminal de controle. Se estivermos tratando com um sistema síncrono (o caso usual), podemos igualmente querer que a transferência ocorra em sincronismo com o relógio. Tal transferência síncrona pode ser efetuada acrescentando-se uma entrada de relógio às portas AND, conforme mostrado na Fig. 8.1-1*b*. A Fig. 8.1-1*c* mostra uma forma-de-onda de relógio e uma mudança do nível de habilitação em "Move A para B" que seleciona um ciclo de relógio particular durante o qual ocorre a transferência.

Suponhamos que temos muitos registradores e requeremos a facilidade de efetuar uma transferência de qualquer registrador para qualquer outro registrador. Se fornecermos conexões

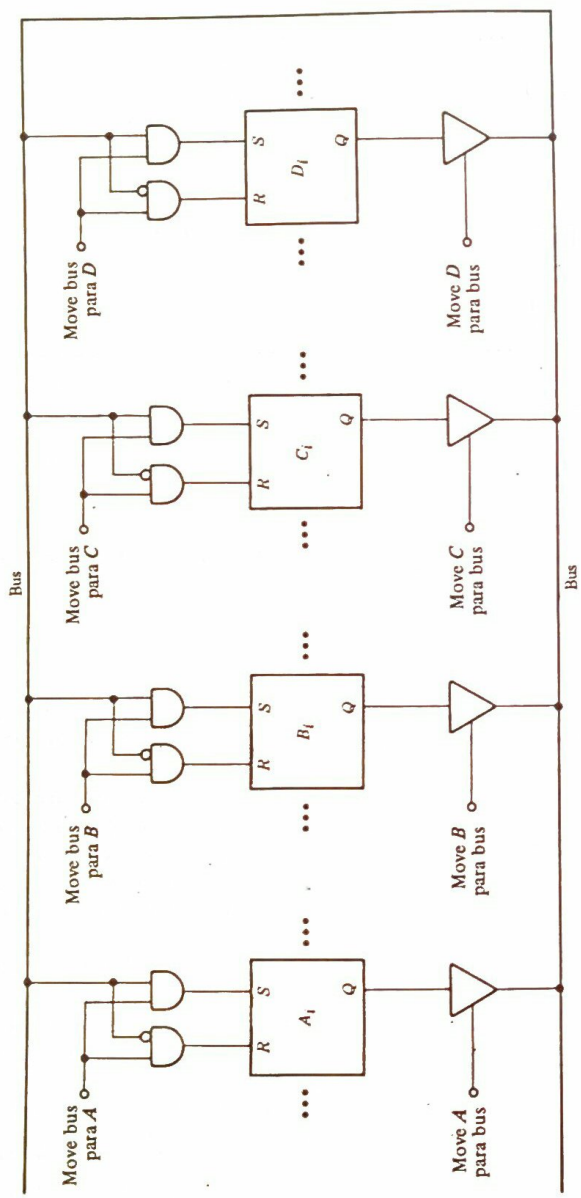


Fig. 8.1-2 Inúmeros registradores compartilham de um bus comum. Elevando-se para 1 lógico um controle de entrada e um controle de saída de cada vez, as transferências entre registradores são realizadas.

individuais de cada registrador para todos os outros, o número de tais conexões pode igualmente descontrolar-se. Num tal caso podemos também querer usar um esquema de multiplexação (ver Seção 3.19) em que um bus (*linha de transmissão de dados*) comum é empregado. O bus é mostrado na Fig. 8.1-2 e serve como uma interconexão entre muitos registradores. Somente quatro registradores são indicados, e para cada registrador somente um latch é mostrado. Correspondentemente, apenas um fio do bus é mostrado. Se os registradores fossem, por exemplo, registradores de dezesseis bits, haveria dezesseis fios separados no bus. Uma vez que toda informação deve ser transferida sobre um único bus, é possível fazer somente uma transferência de cada vez. Se, por exemplo, quisermos fazer uma transferência de A para C, elevaríamos para 1 lógico o nível no terminal de controle "Move A para o bus" e o estado do registrador A seria colocado no bus. (Na Fig. 8.1-2 usamos a conexão de três-estados para acoplar as saídas de muitos registradores a um único bus. Alternativamente, a conexão de *coletor aberto* da Seção 3.19 poderia ser usada.) Se agora também colocarmos em nível lógico 1 o terminal marcado "Move bus para C"; a transferência será realizada. Mas, novamente, o ponto importante a observar é que a operação de transferência é obtida pelas portas de habilitação, neste caso dois conjuntos de portas, fazendo linhas disponíveis sobre as quais o nível lógico 1 é mudado para o nível de habilitação.

8.2 OUTRAS OPERAÇÕES

Complementação

Uma segunda operação lógica simples comumente requerida consiste na complementação de uma palavra. Aqui temos em mente a substituição de uma palavra num registrador por uma nova palavra, sendo cada bit da nova palavra o complemento do bit correspondente da palavra original. A Fig. 8.2-1 mostra a palavra originalmente no registrador A; o complemento deve aparecer no registrador B. Esta transferência e complementação é realizada simplesmente elevando-se para 1 lógico os terminais "Complemento" e "Move A para B". Se for empregado um relógio, como indicado, "Complementa" e "Move A para B" terão que estar em 1 lógico quando o relógio também se eleva para 1 lógico, e a transferência ocorrerá sincronamente com a borda de relógio. Na Fig. 8.2-1*a* o registrador pode consistir simplesmente em um arranjo ou conjunto de latches.

Suponhamos, por outro lado, que não queremos usar um segundo registrador em que o complemento deve aparecer. Em vez disto, queremos usar apenas um registrador e queremos que a palavra original seja substituída por seu complemento. Depois propomo-nos a ler do registrador (para ver qual bit está em cada flip-flop) e, no mesmo ciclo de relógio, a escrever no flip-flop, isto é, escrever no flip-flop o complemento do bit que acabamos de ler. Para este propósito requeremos que os flip-flops no registrador sejam do tipo especial (mestre-escravo etc., conforme discutido no Cap. 4), o que permite leitura e escrita simultâneas. Na Fig. 8.2-1*b* usamos os flip-flops *JK* no registrador. Quando a linha "Complementa" estiver em 1, o flip-flop chaveará na transição de gatilho de relógio, substituindo desta maneira cada bit por seu complemento.

Deslocamento

A Fig. 8.2-2 mostra um arranjo que possibilita uma transferência do registrador A para o registrador B. Dependendo de se S_O, S_L ou S_R está em lógica 1, o deslocamento será direto ou será acompanhado por um deslocamento numa direção ou em outra. São mostrados três flip-flops adja-

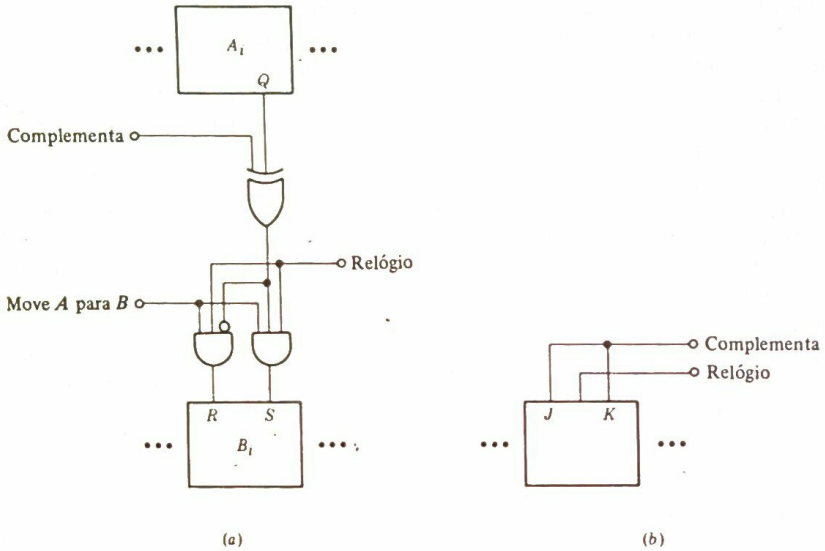


Fig. 8.2-1 (*a*) Um arranjo que permite uma transferência do registrador A para o registrador B do conteúdo de A (Complemento = 0) ou o complemento de A bit por bit (Complemento = 1). (*b*) Um arranjo que possibilita a complementação sem transferência.

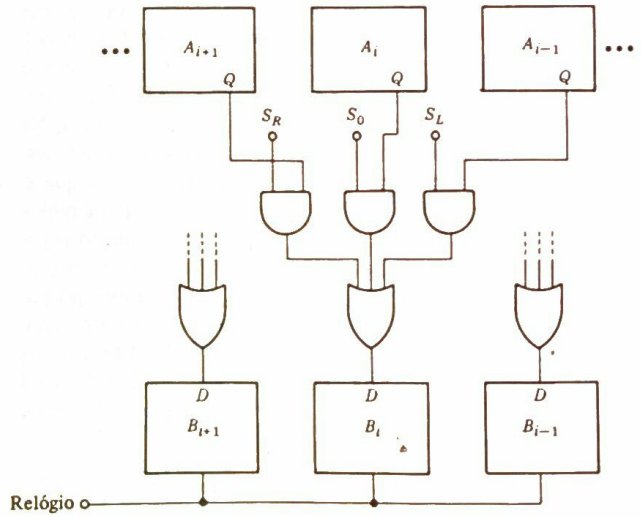


Fig. 8.2-2 Um arranjo que possibilita a transferência do registrador A para o registrador B sem deslocamento ($S_RS_OS_L = 10$), com deslocamento à esquerda ($S_RS_OS_L = 001$) ou com deslocamento à direita ($S_RS_OS_L = 100$).

centes do registrador A e três do registrador B. O registrador B é aqui suposto compreender os flip-flops do tipo D. Há uma estrutura de quatro portas (três portas AND e uma porta OR) associada à entrada D de cada flip-flop do registrador B, mas somente uma estrutura é mostrada completamente. Se somente S_O for $S_O = 1$, o conteúdo de A_i será transferido para B_i na ocorrência da borda de gatilho de relógio. Se somente S_L for $S_L = 1$, haverá um deslocamento à esquerda e transferência. $S_R = 1$ alcançará um deslocamento à direita e transferência. Naturalmente, num deslocamento à esquerda teremos que fazer provisão especial para o bit que deve ser deslocado para o flip-flop mais à direita, uma vez que este bit deve ser fornecido por uma fonte externa. Um comentário semelhante se aplica ao bit a ser deslocado para o flip-flop mais à esquerda num deslocamento à direita. Em qualquer caso, e mais importante, notamos novamente que a operação a ser realizada é conseguida mantendo em lógica 1, durante um ciclo de relógio, um ou outro dos terminais de controle S_O, S_L ou S_R.

Suponhamos que não necessitamos da facilidade de transferir e deslocar uma palavra, mas somente de deslocá-la, mantendo-a no mesmo registrador. Depois devemos requerer nada mais do que o registrador de deslocamento à direita-à esquerda já descrito na Seção 4.18.

Incrementação e Decrementação

Muitas vezes devemos armazenar um número num registrador e incorporar no registrador a facilidade de alterar o número armazenado por +1 ou por -1. Estas operações são chamadas *incrementação* e *decrementação*. Um registrador que responderá ao comando de *altera contagem*, mostrado na Fig. 8.2-3, consiste em um contador incrementador-decrementador (de qualquer tipo, assíncrono ou síncrono) com a modificação de que o relógio não é aplicado diretamente à entrada de relógio do contador. Em vez disto, uma porta AND é acrescentada e o relógio é passado através dessa porta acrescentada. A outra entrada da porta AND é uma entrada de habilitação chamada *incremento*. Se o terminal de altera contagem for mantido em lógica 1 na direção de um ciclo de relógio, o contador incrementará ou decrementará seu registro de 1, dependendo do nível lógico do controle de modos.

Reajuste e Ajuste (Resetting and Setting)

Suponhamos que necessitamos da facilidade de limpar um registrador (colocar Q de cada flip-flop em $Q = 0$) ou ajustar (set) um registrador (cada $Q = 1$). Um flip-flop individual de um tal registrador é mostrado na Fig. 8.2-4. Um flip-flop *JK* está sendo empregado aqui. Se a entrada

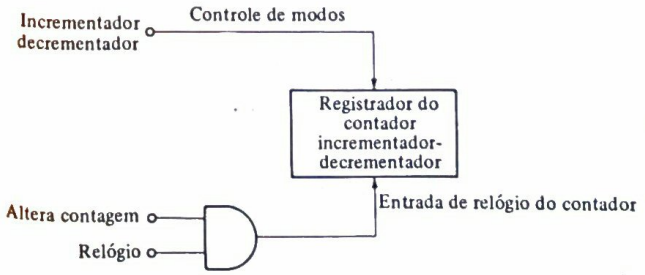


Fig. 8.2-3 O registrador do contador incrementará ou decrementará sua contagem, dependendo do nível lógico do controle de modo, se o terminal "Altera contagem" for mantido em lógica 1 na duração de um ciclo de relógio.

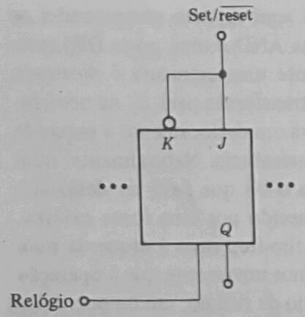


Fig. 8.2-4 Um flip-flop *JK* usado como parte de um registrador e dotado da facilidade de ser limpo ou ajustado (set).

de set/reset estiver em lógica 1, então $J = 1$ e $K = 0$, de modo que em uma borda de gatilho de relógio o flip-flop se ajustará. Se a entrada de set/reset estiver em lógica 0, teremos $J = 0$ e $K = 1$, de modo que o flip-flop se reajustará (reset).

8.3 REGISTRADOR SENSÍVEL A COMANDOS MÚLTIPLOS

Vimos que podemos construir um registrador que responderá a um *comando* ou a outro. O comando é transportado colocando-se o nível lógico de um certo *terminal de controle* no nível que habilita uma porta ou um conjunto de portas. Num sistema síncrono que usa um relógio, o comando geralmente será executado na borda de gatilho de uma forma-de-onda de relógio. Suponhamos que num certo sistema digital requeremos a facilidade de comandar inúmeras operações. Podemos então escolher efetuar operações individuais em registradores separados ou construir um registrador que seja capaz de responder a inúmeros comandos diferentes. A primeira alternativa oferece a vantagem da flexibilidade; a segunda alternativa pode-nos possibilitar a economia de algum hardware.

Como exemplo de um registrador que pode responder a inúmeros comandos, projetemos um registrador que é capaz de responder a cinco comandos. Estes cinco comandos e o símbolo que deve ser associado a cada comando estão listados na Tab. 8.3-1. Portanto, nosso registrador deve ter cinco terminais de controle *W*, *R*, *I*, *C* e *Z*. A qualquer instante apenas um destes terminais deve ser mantido em lógica 1, enquanto todos ou outros devem estar em lógica 0. Se, por exemplo,

Tabela 8.3-1 Comandos aos quais responde o registrador

Comando	Símbolo
1. Escrever a palavra do bus no registrador (*write*)	*W*
2. Ler a palavra do registrador no bus (*read*)	*R*
3. Incrementar o registrador	*I*
4. Complementar o registrador	*C*
5. Limpar o registrador de modo que todos os *Q*s sejam zero (reset)	*Z*

tivermos $W = 1$, então na borda de gatilho da forma-de-onda de relógio a palavra sobre o bus (barramento) deve ser introduzida no registrador. Os n flip-flops no registrador são FF_0, FF_1, . . . , FF_i, . . . , FF_{n-1}. Uma linha B_i do bus de n linhas é mostrada, e a lógica associada exatamente a um flip-flop FF_i é mostrada na Fig. 8.3-1. Decidimos aqui arbitrariamente que os flip-flops *JK* devem ser usados. A fim de introduzir a lógica na linha B_i do cabo no FF_i em uma transição de gatilho de relógio e, quando $W = 1$, requeremos que

$$J_i = B_i W \qquad \text{e} \qquad K_i = \bar{B}_i W \tag{8.3-1}$$

(Usamos o símbolo B_i tanto para representar a *i*-ésima linha do bus quanto o nível lógico na linha.) Para transferir o bit em FF_i para o bus, isto é, para ler o flip-flop sobre o bus quando $R = 1$, requeremos que

$$B_i = Q_i R \tag{8.3-2}$$

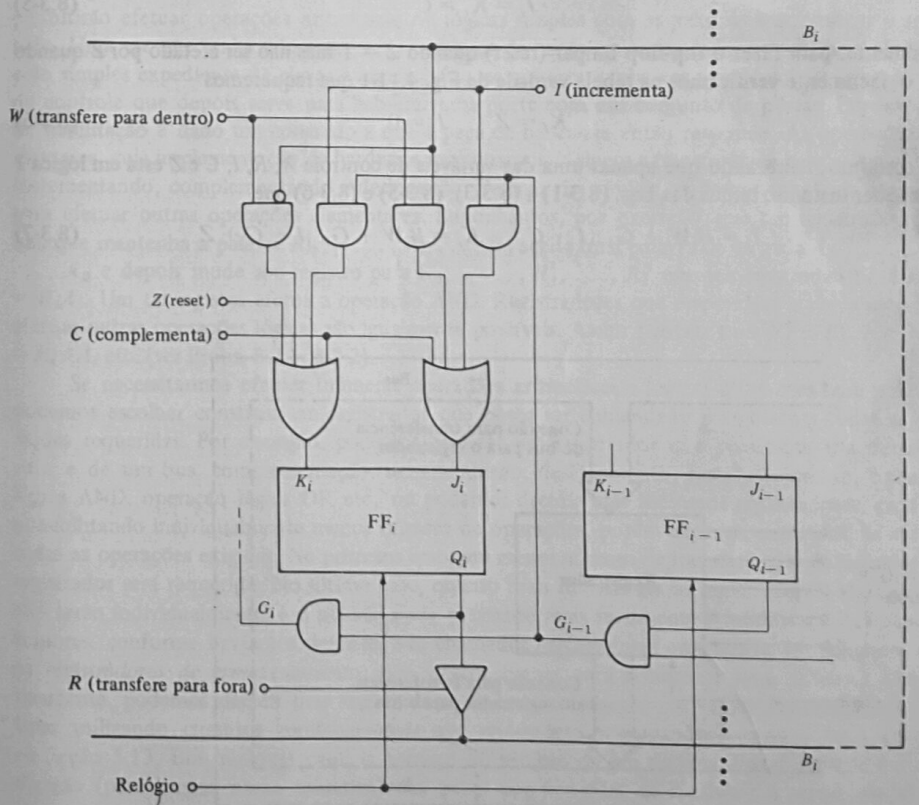


Fig. 8.3-1 Um estágio de um registrador que responderá a cinco comandos.

Notar que esta transferência ocorrerá logo que $R = 1$ e a sincronização (temporização) da transferência não depende do relógio. Naturalmente, se a palavra assim colocada no bus deve ser colocada noutro registro, esta colocação ocorrerá em uma transição de gatilho de relógio.

Para incrementar o registrador em $I = 1$, interconectamos estágios de flip-flop da maneira como fizemos em um contador. Escolhemos arbitrariamente usar um contador assíncrono, de ondulação (contador por pulsação). O contador de ondulação requer que as entradas J e K de cada flip-flop sejam ligadas juntas e depois conectadas à saída Q do flip-flop precedente (ver Fig. 4.27-1). Portanto, requeremos que no i-ésimo flip-flop

$$J_i = K_i = G_{i-1}I \tag{8.3-3}$$

A Eq. (8.3-3) aplica-se a todos os flip-flops, exceto ao primeiro flip-flop FF_0, que não tem estágio precedente. Neste caso especial, requeremos

$$J_0 = K_0 = 1 \cdot I \tag{8.3-4}$$

Para fazer com que um flip-flop complemente, isto é, chaveie, quando $C = 1$, estabelecemos

$$J_i = K_i = C \tag{8.3-5}$$

Finalmente, para fazer o flip-flop limpar (reset) quando $Z = 1$ mas não ser afetado por Z quando $Z = 0$, facilmente verificamos na tabela verdade da Fig. 4.11-1 que requeremos

$$K_i = Z \qquad J_i = 0 \tag{8.3-6}$$

Em conjunto, lembrando que apenas uma das variáveis de controle W, R, I, C e Z está em lógica 1 a qualquer instante, temos das Eqs. (8.3-1) a (8.3-3), (8.3-5) e (8.3-6) que

$$J_i = B_iW + G_{i-1}I \cdot C \qquad K_i = \bar{B}_iW + G_{i-1}I + C + Z \tag{8.3-7}$$

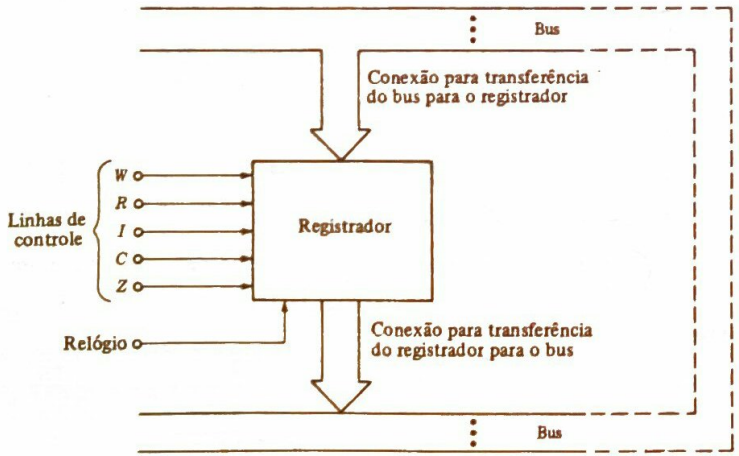


Fig. 8.3-2 Uma representação funcional do registrador cujo único estágio é dado na Fig. 8.3-1.

e

$$B_i = Q_iR \tag{8.3-8}$$

O primeiro flip-flop FF_0 é especial quanto a que os termos $G_{I-1}I$ na Eq. (8.3-7) são substituídos simplesmente pelos termos $1 \cdot I = I$, como indicado na Eq. (8.3-4).

Na Fig. 8.3-1 são mostradas as portas associadas a FF_i requeridas pelas Eqs. (8.3-7) e (8.3-8). Naturalmente, uma estrutura idêntica de portas (não mostrada) é requerida para cada flip-flop no registrador.

Um diagrama em blocos do registrador mostrando suas conexões ao bus é mostrado na Fig. 8.3-2. As cinco linhas de comando de controle estão indicadas, como também a entrada de relógio.

8.4 UM CONTROLADOR SIMPLES

Vimos que é viável construir de maneira direta circuitos combinacionais e registradores que nos permitirão efetuar operações aritméticas ou lógicas simples com as palavras e armazenar o resultado. Uma vez que tenha sido construída uma peça de hardware apropriada, a operação é efetuada pelo simples expediente de elevar à lógica 1 (ou, se quisermos, abaixar para lógica 0) alguma linha de controle que depois serve para habilitar uma porta com um conjunto de portas. Por este ato de habilitação é dado um *comando* a que a peça de hardware então responde. As operações elementares, cuja implementação de hardware examinamos, incluem a transferência *para* e *de* um bus, incrementando, complementando e deslocando. De maneira similar, podemos construir hardware para efetuar outras operações elementares. Suponhamos, por exemplo, que um registrador de n bits que mantenha a palavra $R_{n-1}, \ldots, R_i, \ldots, R_0$ aceite uma palavra de entrada $A_{n-1}, \ldots, A_i, \ldots, A_0$ e depois mude seu registro para $R'_{n-1}, \ldots, R'_1, \ldots, R'_i$, em que cada novo bit é $R'_i = R_iA_i$. Um tal registro efetua a operação AND. Registradores que respondem a comandos para efetuar outras operações lógicas são igualmente possíveis. Assim podemos ter $R'_i = R_i + A$, $R'_i = R_i \oplus A_i$ etc. (ver Probs. 8.2-1, 8.2-2).

Se necessitarmos efetuar inúmeras operações aritméticas e lógicas diferentes com palavras, podemos escolher construir um registrador que possa ser comandado para efetuar todas as operações requeridas. Por exemplo, podemos projetar um registrador que possibilite transferências para e de um bus, complementação, incrementação, deslocamento, restabelecimento, operação lógica AND, operação lógica OR etc., ou podemos decidir usar inúmeros registradores, cada um possibilitando individualmente menor número de operações, porém capazes, entre eles, de efetuar todas as operações exigidas. No primeiro caso, um menor número de transferências de registrador a registrador será requerido. No último caso, quanto mais numerosos os registradores, mais simples eles serão individualmente, e o sistema pode prestar-se mais facilmente à modificação. (Tais registradores, conforme havíamos descrito, são chamados *registradores operativos*, em contraste com os *registradores de armazenamento*, que servem apenas para armazenar uma palavra.) Alternativamente, podemos decidir usar registradores de armazenamento em vez de registradores operativos utilizando circuitos combinacionais que respondem a comandos, como a ULA descrita na Seção 5.13. Em qualquer caso, o começo do projeto de um sistema digital consiste em uma decisão (pelo menos numa tentativa) das peças componentes de hardware a serem usadas, de que comandos os componentes devem ser capazes de responder e de como os componentes devem ser interligados. Estes aspectos característicos do sistema são denominados *arquitetura do*

sistema. Uma vez que a arquitetura tenha sido estabelecida, podemos construir um *controlador*, que fornecerá comandos na seqüência correta às linhas de controle dos componentes, conforme requerido, para fazer com que o sistema execute sua função.

Exceto nos casos muito mais simples, não há procedimento de projeto que conduza à melhor arquitetura. As arquiteturas são selecionadas pelos projetistas na base da experiência e no simples bom senso. Dois projetistas podem desenvolver dois sistemas individuais de arquiteturas diferentes que desempenham a mesma função. Pode ser impossível julgar um projeto superior ao outro sem que haja ambigüidade. Um projeto pode ter méritos em certas direções; o outro pode ter vantagens em outras áreas. A situação é um tanto análoga à que prevalece nos negócios (ou em outras instituições humanas). Consideremos, por exemplo, inúmeras companhias industriais que fabricam bugigangas, todas tendo quase o mesmo rendimento bruto anual. Todas elas necessitam de sistemas de escrituração contábil. O resultado final em cada caso deve ser o mesmo. Elas devem todas guardar os registros e fazer cálculos de modo que possam arrecadar suas contas a receber, saldar suas contas a pagar, pagar seus empregados por hora de trabalho executado etc. Mas o grupamento de funções em departamentos pode bem diferir de companhia para companhia, e dentro de um departamento a divisão de trabalho entre os empregados também pode ser diferente. Ao comparar duas companhias, é muito improvável que haja uma correspondência precisa um a um entre departamentos ou entre os serviços de empregados individuais.

Voltemos agora ao sistema simples cuja arquitetura é dada na Fig. 8.4-1. Aqui temos em

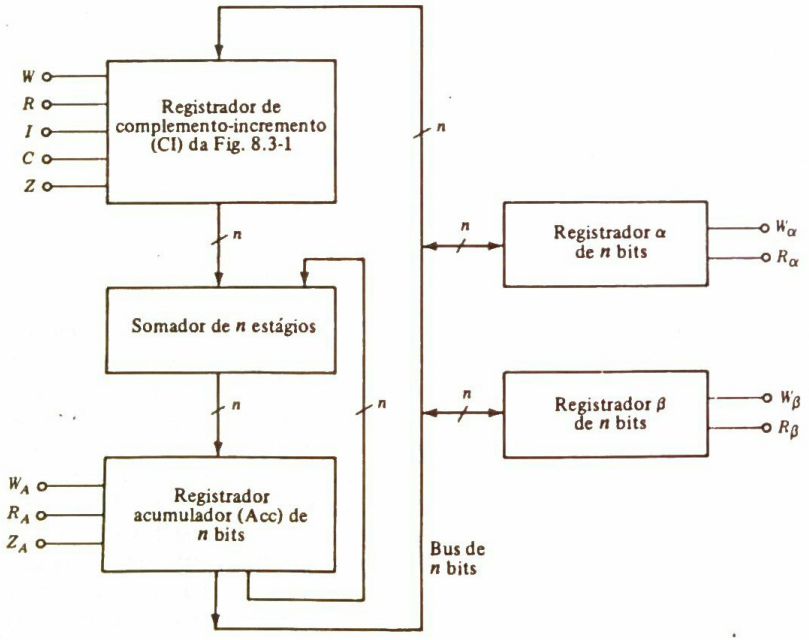


Fig. 8.4-1 Uma arquitetura para combinar aritmeticamente o conteúdo dos registradores α e β.

mente um sistema para calcular o valor da soma ou diferença aritméticas de dois números binários de n bits. Especificamente, queremos calcular as somas e diferenças $\alpha + \beta$, $\alpha - \beta$, $-\alpha + \beta$ e $-\alpha - \beta$ dos números nos registradores α e β. O mecanismo pelo qual estes números são introduzidos nos registradores α e β não está indicado na figura. Admitamos que estes números sejam introduzidos assincronamente através dos terminais de *set-direto* e *reset-direto* dos flip-flops individuais que constituem o registrador. Se um flip-flop tiver que ter um 0 introduzido nele, seu terminal de *reset-direto* é trazido brevemente para lógica 1. Se um 1 deve ser introduzido, o terminal de *set-direto* é trazido brevemente para lógica 1. Quando um número tiver sido introduzido desta maneira, todos os terminais *diretos* são permitidos retornar à lógica 0, de modo que o flip-flop possa agora responder à transição de gatilho de relógio conforme solicitado pelos terminais de controle. A quantidade $\pm\alpha \pm \beta$ é finalmente preparada no registrador acumulador e também no registrador α ou no β.

Todos os registradores e o somador acomodam n bits. O registrador (CI) de complementa-incrementa está conectado ao somador, o somador está conectado ao acumulador e o acumulador está conectado de volta ao somador, todos por conexões de n linhas. Uma vez que estas conexões de n bits são *dedicadas*, isto é, cada uma serve a uma única função de transferência, elas não são buses. Há também um bus de n bits para e a partir do qual podemos transferir os conteúdos do registrador α e do registrador β. Esta viabilidade de transferência de duas vias é indicada pela seta bidirecional. Uma forma-de-onda de sincronização comum, não mostrada, é aplicada a todos os registradores. Quando $W_\alpha = 1$, na borda de gatilho de relógio uma palavra é transferida do bus para o registrador α; assim, uma palavra é "escrita" no registrador. Quando $R_\alpha = 1$, uma palavra é "lida" do registrador no bus. Comentários similares se aplicam à resposta do registrador β a W_β e R_β. Quando $W_A = 1$, a saída do somador será registrada no acumulador, e quando $R_A = 1$, o conteúdo do acumulador será colocado no bus. O conteúdo do acumulador está permanentemente conectado à conexão de n bits que retorna ao somador. Não há nenhum controle sobre esta conexão. O acumulador é limpo (levado a zero) na transição de gatilho de relógio se $Z_A = 1$.

Consideremos como o sistema da Fig. 8.4-1 pode ser orientado para achar $\alpha + \beta$ e para armazenar o resultado no registrador α. Inúmeras operações elementares, chamadas *microoperações*, são necessárias, cada uma requerendo um ciclo de relógio. A seqüência está mostrada na Tab. 8.4-1.

Notar que nesta seqüência de operações elementares tanto α quanto β foram passados através de CI. Naturalmente, não há necessidade de se fazer assim, e o tempo de dois ciclos de relógio foi desperdiçado desta maneira. Ainda, por causa da arquitetura de nosso sistema, não tivemos nenhuma escolha. Se tivéssemos partido de uma arquitetura mais elaborada, que fornecesse acesso direto ao somador, teríamos sido capazes de desviar o registrador CI. Notar que não há necessidade de limpar o registrador CI.

Projetaremos agora um controlador que tornará disponíveis os níveis lógicos requeridos para pôr em seqüência nossa máquina através de suas etapas. É evidente que o controlador deve ser uma máquina seqüencial com seis estados, uma vez que têm que ser efetuadas seis operações separadas. No entanto, no caso presente encontramos uma situação que não surgiu no Cap. 7, quando projetamos detectores de seqüências. Num detector de seqüências estávamos satisfeitos em permitir que a máquina funcionasse continuamente desde que considerássemos que a seqüência de entrada também era contínua. No caso presente queremos ser capazes de interromper ou parar a máquina após a última etapa na seqüência. Do contrário, a máquina continuará de um lado para outro de sua seqüência de operações. Após uma volta, o novo registro no registrador α será $\alpha' = \alpha + \beta$. Após a volta seguinte, teremos $\alpha'' = \alpha' + \beta$ etc. E, se usarmos um relógio com

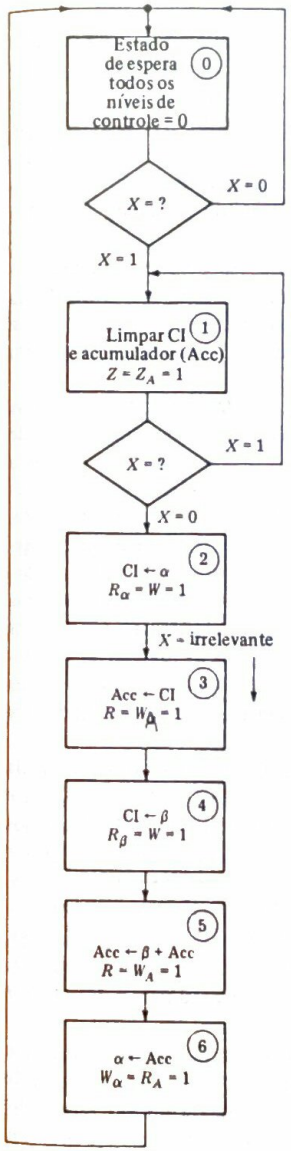
Tabela 8.4-1

Ciclo de relógio	Linhas de controle a serem postas em lógica 1	Comentário
1	Z_A	Registrador do acumulador limpo de qualquer número que possa ter restado de uma operação anterior
2	R_α, W	Ler α no bus e escrever palavra no bus dentro do registrador CI
3	R, W_A	Conteúdo de CI passado pelo somador (outra entrada do somador é zero) e registrado no acumulador
4	R_β, W	Conteúdo de β transferido para CI
5	R, W_A	β acrescentado ao acumulador
6	R_A, W_α	Conteúdo do acumulador transferido para o registrador α

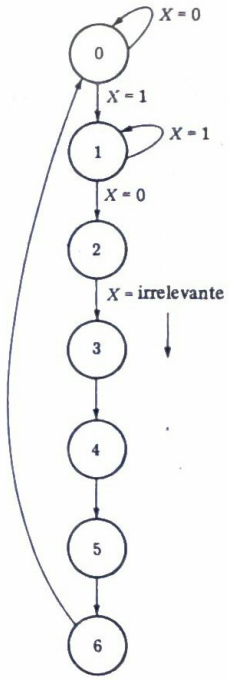
uma freqüência comumente usada em sistemas eletrônicos digitais (100 kHz ou mais alta), não teremos mesmo tempo de ler o registrador α antes que ele mude.

Portanto, aos seis estados correspondentes às seis micro operações acrescentemos um sétimo, em que o controlador irá parar e *esperar* quando a seqüência tiver sido completada. Este estado extra nos dará tempo para ler o resultado de nossa computação e para colocar novos números nos registradores α e β. Depois consideramos que, para tirar o controlador fora do estado de espera, há uma entrada X no controlador proveniente de alguma fonte externa. Esta entrada pode ser fornecida por uma chave de botão de calcar. Quando o botão for calcado, $X = 1$; do contrário, $X = 0$. Quando $X = 0$, o controlador, tendo iniciado sua seqüência, continuará até que ele alcance o estado de *espera*. A seqüência seguinte não começará até que o botão tenha sido calcado de modo que X novamente se torne $X = 1$. Conforme vemos agora, no entanto, não resolvemos nosso problema completamente. Suponhamos que, tendo calcado o botão, deixamos de liberá-lo antes que a seqüência tenha sido completada. Então o controlador pode atravessar diversos ciclos antes que finalmente chegue a se restabelecer no estado de *espera* (a uma taxa de relógio de 100 kHz, a seqüência de seis etapas é concluída em apenas 60 μs). Em conjunto, somos levados a um controlador descrito pelo fluxograma da Fig. 8.4-2*a*.

Há sete estados que numeramos de 0 a 6. O número do estado é dado no círculo no canto superior direito dos quadrículos de estado na Fig. 8.4-2*a*. No estado 0, o estado de espera, todas as entradas de controle na unidade aritmética da Fig. 8.4-1 estão em lógica 0. Enquanto $X = 0$, o controlador permanece no estado 0. Quando o botão for calcado e $X = 1$, o controlador vai para o estado 1, enquanto $Z_A = 1$, de modo que o registrador do acumulador está limpo. O controlador não sai do estado 1 até que o botão tenha sido liberado. Na liberação do botão o controlador vai para o estado 2, onde $R_\alpha = W = 1$, de modo que o conteúdo do registrador α se deslocará para o registrador CI. Depois do estado 2, o progresso para o estado 3 e o restante da seqüência prossegue independentemente de se $X = 0$ ou $X = 1$. O controlador finalmente termina no estado 0 e permanece ali até que o botão seja calcado novamente. O diagrama de estados é dado na Fig. 8.4-2*b*.



(*a*)



(*b*)

Fig. 8.4-2 (*a*) O fluxograma de um controlador que usa o sistema da Fig. 8.4-1 através das microoperações requeridas para somar os conteúdos de R_α e R_β e armazenar a soma em R_α. (*b*) O diagrama de estados.

8.5 IMPLEMENTAÇÃO DO CONTROLADOR

Uma vez que o controlador da seção anterior tem sete estados, necessitamos de um circuito seqüencial de três flip-flops. Decidimos arbitrariamente usar flip-flops do tipo D. Uma tabela de transição é dada na Fig. 8.5-1*a*. Na primeira coluna, para identificação de estados, listamos os estados por sua numeração na Fig. 8.4-2*a*. Arbitrariamente fizemos a atribuição de estado que está indicada na segunda coluna. Fizemos simplesmente, para cada estado, uma atribuição $Q_2Q_1Q_0$ que, quando lida como um número binário, é igual ao número de identificação decimal. Os estados seguintes, para $X = 0$ e $X = 1$, são tomados diretamente do fluxograma ou diagrama de estados da Fig. 8.4-2.

O controlador é uma máquina Moore. As saídas são totalmente dependentes do estado do controlador, isto é, dos níveis lógicos em Q_2, Q_1 e Q_0. A única entrada X não tem influência direta sobre as saídas, uma vez que X serve apenas para determinar se o controlador está em seqüência através de seus estados ou se é mantido em repouso. As saídas listadas nas colunas restantes podem também ser lidas diretamente do fluxograma.

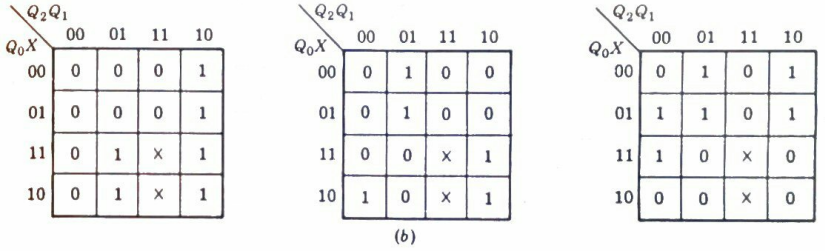
Na Fig. 8.5-1*b* construímos mapas K para as excitações D_2, D_1, D_0 dos três flip-flops. A leitura de cada mapa é dada diretamente sob o mapa. Na Fig. 8.5-1*c* desenhamos o diagrama de circuito daquela parte do controlador que determina a ação de pôr em seqüência de estado a estado. (A parte do controlador que gera as muitas saídas requeridas não é mostrada.) As três estruturas de porta que geram as três excitações D_2, D_1 e D_0 não são mostradas explicitamente, mas estão indicadas pelas caixas (quadrículos) designadas por "lógica". As entradas nas caixas lógicas são Q_2, Q_1 e Q_0 e a variável, X. As saídas das caixas lógicas são expressas na figura como funções booleanas. Naturalmente, se escolhermos, podemos substituir as três caixas lógicas usando portas individuais com uma ROM. As entradas na ROM, isto é, o endereço, seriam então $Q_2Q_1Q_0$ e X e as saídas, isto é, a palavra lida da ROM, seriam D_2, D_1 e D_0. Finalmente, a parte de geração de seqüências do controlador (Fig. 8.5-1*c*) é representada como um bloco na Fig. 8.5-1*d*. Os únicos terminais explicitamente em evidência são o relógio, a entrada X e os terminais dos flip-flops requeridos para implementar o decodificador que irá gerar as saídas.

O decodificador de geração das saídas é dado na Fig. 8.5-2. Como exemplo, notamos da tabela da Fig. 8.5-1*a* que Z e Z_A estão em lógica 1 quando e somente quando $Q_2 = 0$, $Q_1 = 0$ e $Q_1 = 1$. Por conseguinte, um único AND com entradas conforme indicado gera Z e Z_A. Notamos que R e W_A estão em lógica 1 quando o estado for $Q_2Q_1Q_0 = 011$ e também quando $Q_2Q_1Q_0 = 101$. Por conseguinte, aqui duas portas AND e uma porta OR são requeridas conforme mostrado. O restante do decodificador é lido de maneira idêntica à da Fig. 8.5-1*a*, que, no que concerne às saídas, é uma tabela verdade que mostra a relação das saídas com Q_2, Q_1 e Q_0. As Figs. 8.5-1*c* e 8.5-2, juntas, constituem o controlador completo. Quando o controlador é usado em conjunto com a arquitetura mostrada na Fig. 8.4-1, temos uma máquina que somará dois números binários.

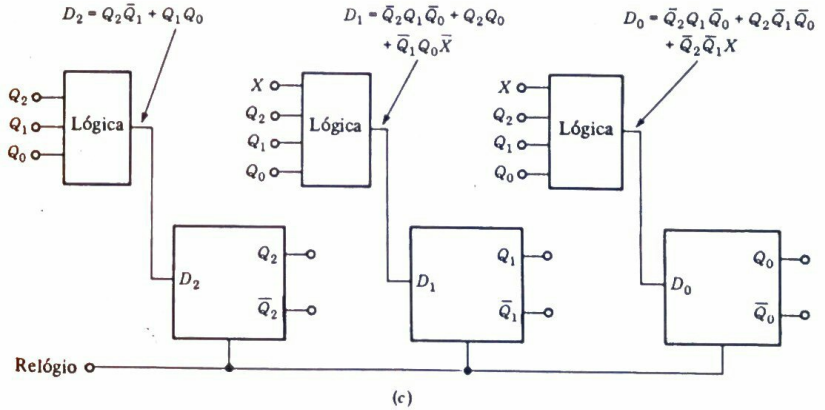
As formas-de-onda da máquina de adição estão mostradas na Fig. 8.5-3. Admitimos aqui que os flip-flops das seqüências respondem à transição (borda) crescente negativa da forma-de-onda de relógio. A forma-de-onda de X representa a operação da chave. Em um ponto arbitrário, num ciclo de relógio, a chave está fechada e X vai para lógica 1. A chave permanece fechada durante um número indeterminado de ciclos de relógio e depois abre, novamente num ponto arbitrário no ciclo de relógio. As formas-de-onda mostradas são aquelas na saída do decodificador e também a "forma-de-onda" de $\bar{Q}_2\bar{Q}_1\bar{Q}_0$. Esta última não é realmente gerada em qualquer lugar no sistema e foi incluída de modo que será evidente quando a seqüência estiver no estado de espera.

Estado presente	Estado presente	Estado seguinte		Saídas								
	$Q_2Q_1Q_0$	$X=0$	$X=1$	Z	Z_A	R_α	W	R	W_A	R_β	W_α	R_A
0	000	000	001	0	0	0	0	0	0	0	0	0
1	001	010	001	1	1	0	0	0	0	0	0	0
2	010	011	011	0	0	1	1	0	0	0	0	0
3	011	100	100	0	0	0	0	1	1	0	0	0
4	100	101	101	0	0	0	1	0	0	1	0	0
5	101	110	110	0	0	0	0	1	1	0	0	0
6	110	000	000	0	0	0	0	0	0	0	1	1

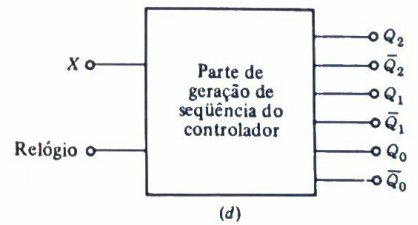
(*a*)



(*b*)



(*c*)



(*d*)

Fig. 8.5-1 O projeto do controlador cujos fluxograma e tabela de estado são dados na Fig. 8.4-2: (*a*) tabela de transição; (*b*) mapas K para as excitações de entrada dos flip-flops; (*c*) diagrama lógico da parte de geração de seqüência do controlador; e (*d*) símbolo do sistema lógico em (*c*).

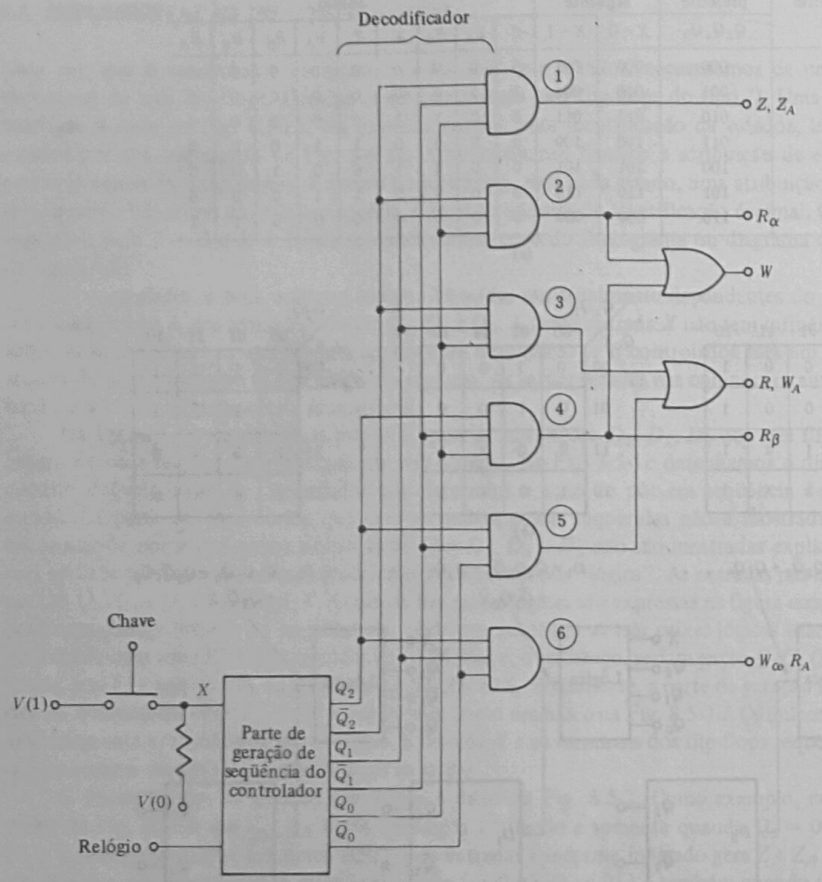


Fig. 8.5-2 O decodificador a ser usado com o gerador de seqüências da Fig. 8.5-1.

As formas-de-onda começam num instante em que $X = 0$ (chave ainda não calcada) e o sistema está no estado de espera. No instante da primeira transição de relógio crescente negativa após a chave ter sido calcada, o sistema sai do estado de espera (estado 0) e vai para o estado seguinte (estado 1), em que os estados seguintes Z e Z_A são 1. Este estado 1 persiste até que o tempo da próxima transição de relógio negativa depois de X volte a $X = 0$. Além das formas-de-onda de habilitação Z e Z_A, todas as outras formas-de-onda crescem até o nível de habilitação de um único ciclo de relógio em um instante. Algumas formas-de-onda fazem esta excursão para a lógica 1 apenas uma vez quando as seqüências completam um ciclo através de seus estados; outras fazem-na duas vezes.

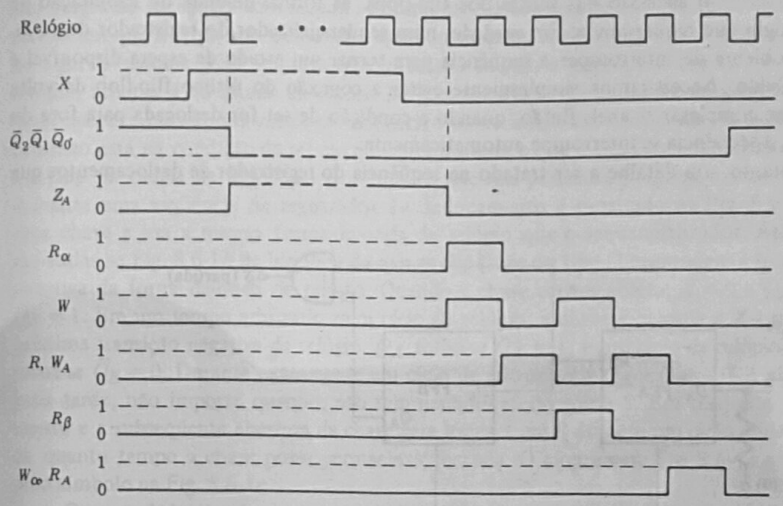


Fig. 8.5-3 Formas-de-onda de relógio e formas-de-onda geradas nas saídas do decodificador da Fig. 8.5-2.

8.6 O CONTROLADOR DO REGISTRADOR DE DESLOCAMENTOS

O controlador das seções anteriores é projetado para requerer um número mínimo de flip-flops. Este critério de projeto de minimização do número de estados, de modo que o número de flip-flops seja mínimo, também foi observado no Cap. 7, onde projetamos detectores de seqüências. No Cap. 7 notamos, no entanto, que havia um método alternativo de projeto usando registradores de deslocamento. O projeto do registrador de deslocamentos usa mais flip-flops, mas geralmente menos lógica, isto é, menor número de portas para gerar funções lógicas. E enquanto, no final, um projeto de registrador de deslocamentos possa não ser tão econômico em hardware quanto um projeto de mínimo de flip-flops, o projeto do registrador de deslocamentos tem pelo menos um grande mérito, que é o de ser mais ordenado e sistematizado no sentido de que podemos determinar fácil e precisamente o que faz cada flip-flop. Tal não é geralmente o caso num projeto com o mínimo de número de estados.

Estas considerações levam-nos a examinar um projeto de registrador de deslocamentos no caso presente. Observemos primeiro que, enquanto o controlador projetado nas seções precedentes usa apenas três flip-flops, ele usa muita lógica nas caixas lógicas da Fig. 8.5-1 e no decodificador da Fig. 8.5-2. Segundo, notemos que as formas-de-onda da Fig. 8.5-3 são rememorativas das formas-de-onda encontradas geralmente nos circuitos registradores de deslocamentos. É, em princípio pelo menos, muito óbvio como usar um registrador de deslocamentos para construir uma seqüência para nosso controlador de máquina de somar. Temos apenas que construir um contador em anel do registrador de deslocamento de seis estados e providenciar para que, no início, o primeiro flip-flop esteja ajustado (set) e todos os outros reajustados (reset). Depois, com cada ciclo de relógio, a condição de set ($Q = 1$) se moverá ao longo do registrador e, ciclo a ciclo,

teremos disponíveis, em sucessão nas saídas dos flip-flops, as formas-de-onda de habilitação de um ciclo de relógio que requeríamos. Em verdade, num seqüencializador do registrador de deslocamentos, o problema de interromper a seqüência para tornar um estado de espera disponível é facilmente resolvido. Necessitamos simplesmente evitar a conexão do último flip-flop de volta ao primeiro para completar o anel. Então, quando a condição de set for deslocada para fora do último flip-flop, a seqüência se interrompe automaticamente.

Há, no entanto, um detalhe a ser tratado na seqüência do registrador de deslocamentos que

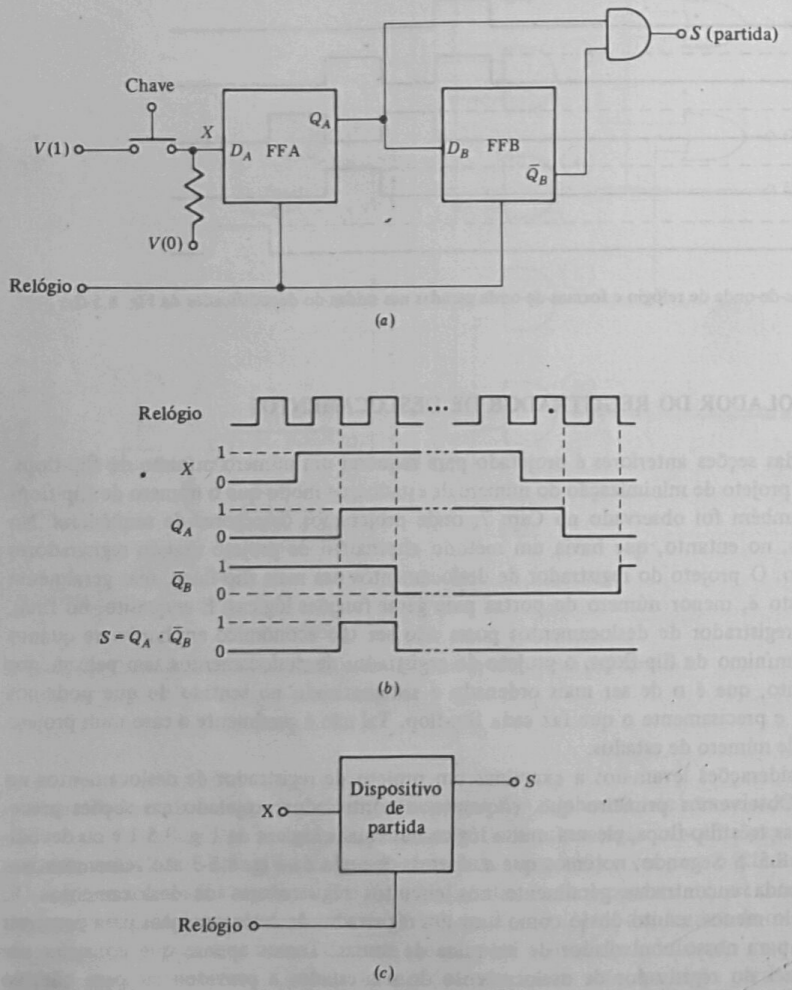


Fig. 8.6-1 (*a*) Uma estrutura lógica que, no fechamento da chave, gera uma transição de S para o nível lógico 1 que persiste durante um ciclo de relógio. (*b*) Formas-de-onda. (*c*) Símbolo do dispositivo de partida.

não surgiu na seqüência com mínimo de estados. Ali não tínhamos nenhum interesse a respeito do estado em que a seqüência podia estar quando fosse aplicada energia de alimentação para acionar ou ligar o sistema. Não importa qual o estado inicial, a seqüência, depois de alguns ciclos de relógio, achar-se-á no estado de espera. Na seqüência do registrador de deslocamentos, por outro lado, necessitamos de um modo de assegurar que no início um e somente um registrador de deslocamento está na condição de set em qualquer instante e que o flip-flop assim ajustado é o primeiro flip-flop na cadeia de flip-flops. Um circuito que nos permitirá preparar e iniciar (*start*) adequadamente uma seqüência de registrador de deslocamento é mostrado na Fig. 8.6-1*a*. Ele envolve uma chave e usa a mesma forma-de-onda de relógio que o seqüencializador. As formas-de-onda são dadas na Fig. 8.6-1*b* na hipótese de que os flip-flops do tipo D respondem à transição crescente negativa da forma-de-onda de relógio. Quando a chave estiver aberta, $X = 0$ e $Q_A = 0$ enquanto $\bar{Q}_B = 1$. Em um tempo arbitrário num ciclo de relógio, a chave é calcada e X torna-se $X = 1$. Na próxima transição negativa de relógio, Q_A torna-se $Q_A = 1$, e um ciclo de relógio, mais tarde, $\bar{Q}_B$ torna-se $\bar{Q}_B = 0$. Durante exatamente um ciclo de relógio, $S = Q_A \cdot \bar{Q}_B = 1$. A abertura da chave mais tarde, não importa quando, não tem nenhum efeito sobre S. Em conjunto, então, o fechamento e a subseqüente abertura da chave gera lógica 1 em S durante um ciclo, independentemente de quanto tempo a chave possa permanecer fechada. O circuito na Fig. 8.6-1*a* será representado pelo símbolo na Fig. 8.6-1*c*.

O controlador inteiro com esta unidade de dispositivo de partida é mostrado na Fig. 8.6-2. Quando a chave está aberta, $X = 0$ e $S = 0$. Depois de alguns ciclos de relógio o registrador inteiro de deslocamentos terá voltado a zero (estará limpo) e todas as saídas estarão em lógica 0. O fechamento da chave ajustará $S = 1$ durante um ciclo de relógio, e esta condição de set propagar-se-á para baixo no registrador de deslocamentos. É bem verdadeiro que usamos aqui muito mais flip-flops que no controlador de mínimo de estados, e em conseqüência há muitos estados não utilizados; mas observemos que muito menos lógica é usada no presente caso. Observemos também que aqui há uma correspondência de um para um entre o flip-flop e os estados através dos quais o controlador completa o ciclo. E finalmente observemos quanto mais simples seria reparar defeitos no presente sistema.

8.7 RESPOSTA CONDICIONAL DE CONTROLADORES

O controlador das seções precedentes seguia uma seqüência fixa de produção de ciclos através dos estados, fazendo disponível uma seqüência fixa de níveis lógicos de habilitação para efetuar uma seqüência fixa de microoperação. No caso mais geral queremos um controlador para acompanhar diferentes seqüências sob diferentes circunstâncias. Para orientar o controlador, haverá disponível para ele inúmeras entradas X_0, X_1 etc. Em alguns casos, uma entrada, por exemplo X_0, virá de uma fonte externa tanto ao controlador, quanto ao processador que está sendo controlado, e o nível lógico de X_0 será determinado pela intervenção humana. Por exemplo, em nosso sistema acima, o resultado final da adição é, como uma última etapa, transferido de volta ao registrador R_α, mas podemos querer ser capazes de escolher se o resultado será transferido para R_α ou para R_β. E podemos depois arranjar, por exemplo, para que, quando pusermos $X_0 = 0$, a transferência seja para R_α, e quando pusermos $X_0 = 1$, a transferência seja para R_β. Em outros casos, uma entrada, por exemplo X_1, pode ser determinada por uma conseqüência ou resultado do processamento. Por exemplo, suponhamos que queremos o resultado final transferido para R_α, se o resultado revelar-se um número positivo, e transferido para R_β se revelar um número negativo. Supo-

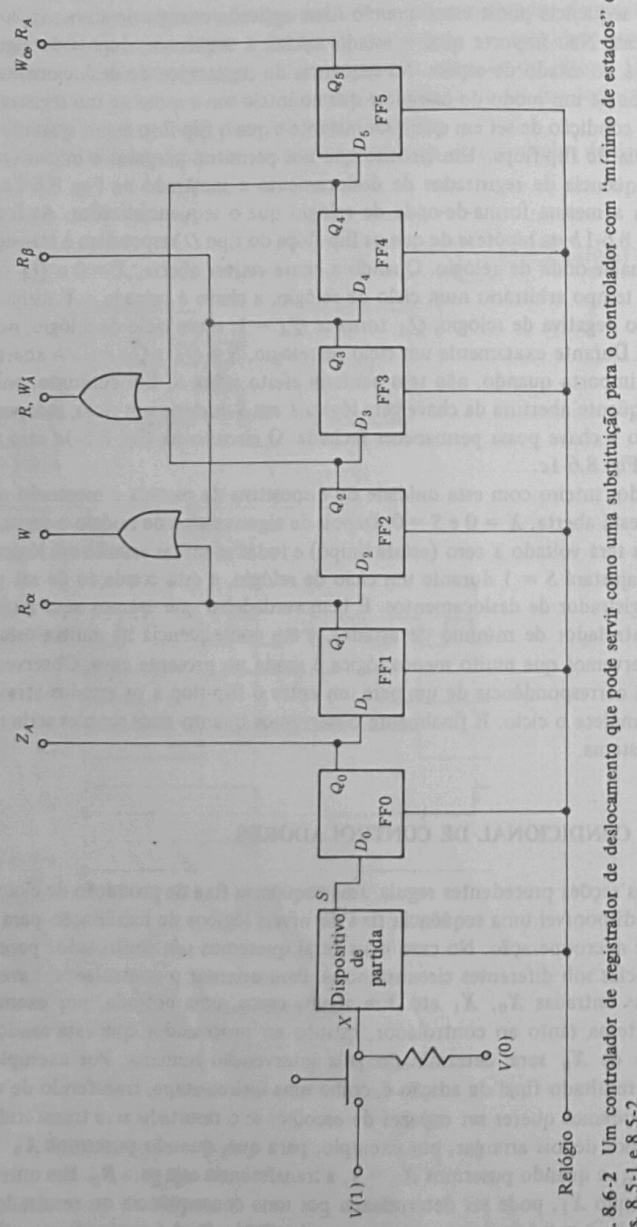


Fig. 8.6-2 Um controlador de registrador de deslocamento que pode servir como uma substituição para o controlador com "mínimo de estados" das Figs. 8.5-1 e 8.5-2.

nhamos, ainda, que números negativos sejam representados em forma de complemento de dois de modo que o bit de sinal seja 0 ou 1, conforme o número for positivo ou negativo. Então a entrada X_1 do controlador seria o bit de sinal no registrador do acumulador, e quereríamos arranjar uma transferência para R_α ou R_β dependendo de X_1 ser 0 ou 1. Estas entradas, X_0, X_1 etc., sejam de uma fonte inteiramente externa, sejam de um resultado do processamento, são precisamente as entradas X_0, X_1 etc. na Fig. 7.5-6, que representava um circuito seqüencial generalizado. As entradas geradas são chamadas *entradas de realimentação.*

Num projeto de controlador de mínimo de estados, sempre preparamos uma tabela de estados. Numa tal tabela de estados há uma coluna de estados seguintes correspondente a cada combinação possível de entradas. Se houver uma entrada, há duas de tais colunas, uma para $X = 0$ e outra para $X = 1$. Se houver duas entradas, há quatro colunas; três entradas, oito colunas, e assim por diante. Depois de ter sido deduzida a tabela de estados, e após havermos decidido a respeito do tipo de flip-flop a ser usado e termos feito uma atribuição de estado, o restante do projeto que leva ao controlador é bastante automático. Por outro lado, são requeridos alguns comentários para esclarecer como as entradas X_0, X_1 etc., muitas vezes chamadas *entradas condicionais* ou *modificadores*, são manipuladas nos controladores de registradores de deslocamento.

Suponhamos, por exemplo, que queremos ser realmente capazes de selecionar se R_α ou R_β deve ser o repositório final de nosso resultado e que providenciamos para este propósito um *seletor de registrador final* (FRS) de entrada de controle. Então podemos ver na Fig. 8.7-1 como esta entrada de controle é feita eficiente. Todo o controlador da Fig. 8.6-5 permanece inalterado, exceto na saída do último flip-flop na cadeia, uma vez que somente a última microoperação é afetada. A última saída é modificada conforme mostrado pelo acréscimo de duas portas AND. Se FRS = 1, W_α tornar-se-á $W_\alpha = 1$ durante o último ciclo da seqüência e o resultado será escrito em R_α. Se RFS = 0, R_β será escolhido.

Suponhamos, por outro lado, que queremos selecionar o registrador R_α ou R_β não na base da entrada externa, mas na base do sinal do resultado. Então na Fig. 8.7-1, substituiríamos RFS pela conexão ao flip-flop do acumulador que guarda ou retém o bit de sinal. É viável fazer assim porque, durante o intervalo em que deve ser feita a transferência final para R_α ou R_β, o acumulador ainda não está limpo e o bit de sinal está disponível.

Consideremos a seguir um caso em que há uma entrada f de realimentação no controlador, e suponhamos que é exigido que o valor lógico de f durante o k-ésimo ciclo de relógio tem que selecionar a microoperação a ser executada durante o $(k + 1)$-ésimo ciclo de relógio. Um modo de se conseguir esta finalidade consiste em acrescentar um flip-flop (e não parte da cadeia do registrador de deslocamentos), no qual armazenamos o valor de f de modo que ele será encontrado quando necessário. Então, como na Fig. 8.7-1, executaríamos uma ou outra microoperação

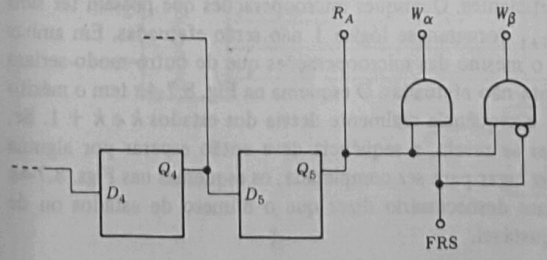


Fig. 8.7-1 Uma modificação do controlador da Fig. 8.6-2 que permite uma resposta do controlador determinada pelo nível lógico de um FRS (seletor do registrador final) de entrada.

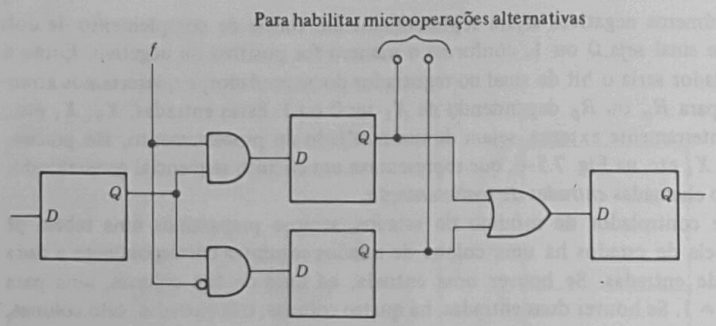


Fig. 8.7-2 Um controlador do registrador de deslocamentos que possibilita o nível lógico de uma entrada f de realimentação estabelecida durante um intervalo de relógio controlar a microoperação realizada durante o próximo intervalo.

no $(k + 1)$-ésimo intervalo de relógio. Uma segunda possibilidade é permitir que o controlador siga um de dois caminhos alternativos após o intervalo k, dependendo do valor de f. Um tal arranjo que fornece caminhos alternativos é mostrado na Fig. 8.7-2. Aqui, se $f = 1$, o flip-flop superior é incluído na cadeia de registradores de deslocamento, enquanto o flip-flop inferior é excluído. Se $f = 0$, o flip-flop inferior é usado e o flip-flop superior, excluído. Daí, dependendo de qual flip-flop seja usado, uma ou outra de duas microoperações alternativas será efetuada.

Na Fig. 8.7-2 os caminhos são alternativos durante o tempo de um ciclo de relógio. Após este período de relógio, a seqüência retorna a um único caminho comum. Num caso mais geral, pode haver mais do que dois caminhos e os caminhos alternativos podem ter diferentes comprimentos. Uma tal situação está representada na Fig. 8.7-3, onde estão indicados quatro caminhos alternativos, todos de diferentes comprimentos. O caminho é selecionado pelas entradas f_1 e f_2 de controle de seqüência. Dependendo dos valores lógicos de f_1 e f_2, uma e somente uma das linhas C_0, C_1, C_2 ou C_3 estará em lógica 1, e um caminho de seqüência correspondente será selecionado.

Outras manipulações que são facilmente incorporadas numa seqüência de registrador de deslocamento estão indicadas na Fig. 8.7-4. Na Fig. 8.7-4*a*, se $f = 1$, a seqüência progride do estado $k - 1$ até k até $k + 1$ até $k + 2$. (Definimos aqui o estado k como sendo o estado em que o flip-flop k está no estado set.) Quando, no entanto, $f = 0$, os estados k e $k + 1$ são desviados e avançamos do estado $k - 1$ ao estado $k + 2$. Na Fig. 8.7-4*b*, quando $f = 1$, os estados k e $k + 1$ não são desviados, mas são tornados ineficientes. Quaisquer microoperações que possam ter sido efetuadas quando Q_k e, mais tarde, Q_{k+1} tornaram-se lógica 1 não serão efetuadas. Em ambas as Figs. 8.7-4*a* e *b* o resultado final é o mesmo das microoperações que de outro modo seriam efetuadas nos estados k e $k + 1$ realmente não efetuadas. O esquema na Fig. 8.7-4*a* tem o mérito potencial de economizar tempo porque a seqüência realmente desvia dos estados k e $k + 1$. Se, no entanto, conforme não poucas vezes se revela, a seqüência deve então esperar por alguma operação sendo efetuada em algum outro lugar para ser completada; os esquemas nas Figs. 8.7-4*a* e *b* são comparáveis. Finalmente, é quase desnecessário dizer que o número de estados ou de microoperações saltados é inteiramente ajustável.

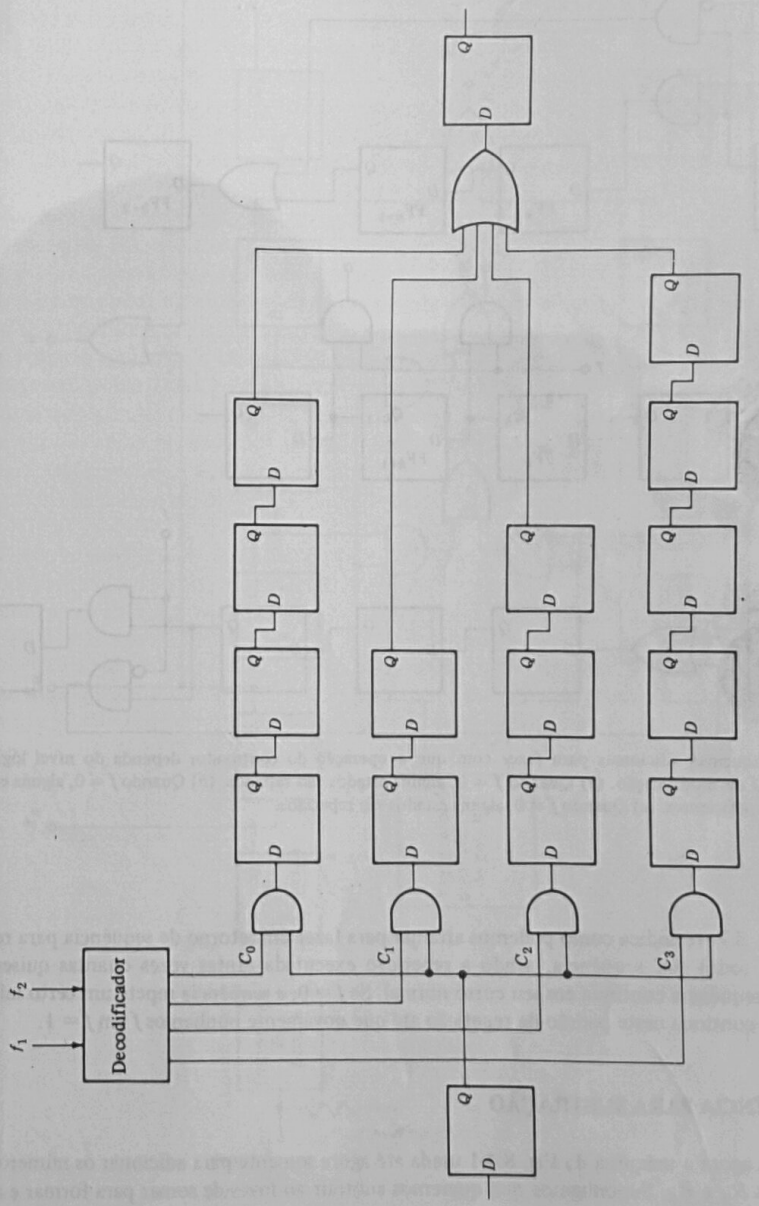


Fig. 8.7-3 Um controlador que permite quatro diferentes seqüências de comprimento variáveis, dependendo das entradas f_1 e f_2.

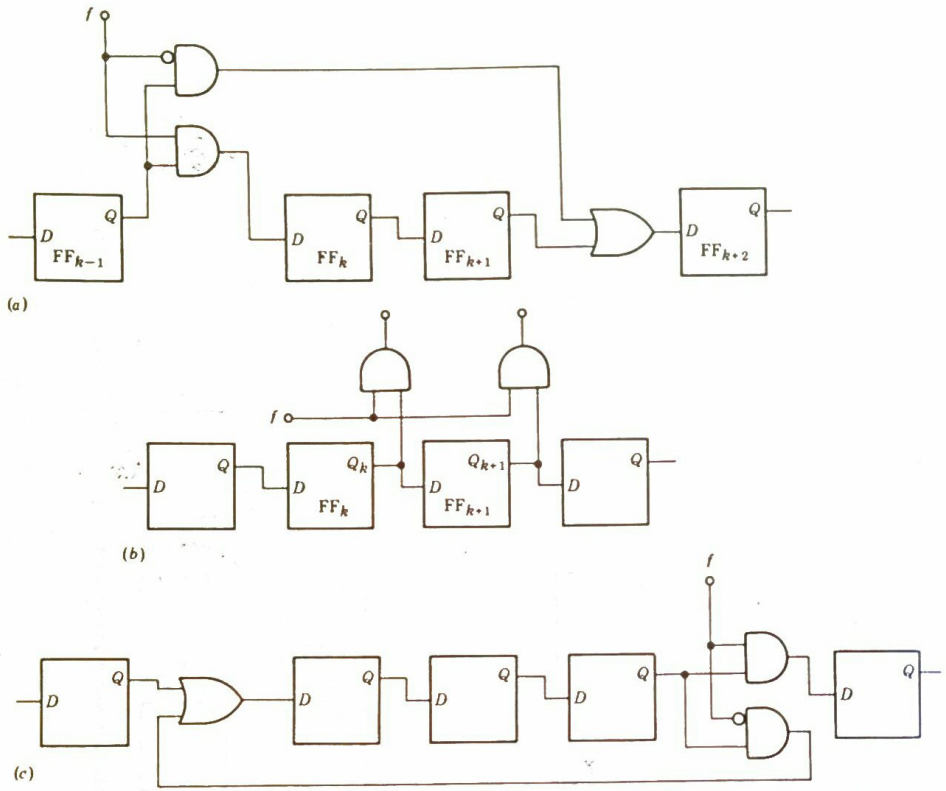


Fig. 8.7-4 Esquemas adicionais para fazer com que a operação do controador dependa do nível lógico de uma entrada f de modificação. (*a*) Quando $f = 0$, alguns estados são saltados. (*b*) Quando $f = 0$, alguns estados são tornados ineficientes. (*c*) Quando $f = 0$, alguns estados são repetidos.

A Fig. 8.7-4*c* indica como podemos arranjar para fazer um retorno de seqüência para repetir alguma (ou toda) sua seqüência, sendo a repetição executada tantas vezes quantas quisermos. Se $f = 1$, a seqüência continua em seu curso normal. Se $f = 0$, a seqüência repete um certo número de estados e continua neste padrão de repetição até que novamente ponhamos f em $f = 1$.

8.8 SEQÜÊNCIA PARA SUBTRAÇÃO

Retornemos agora à máquina da Fig. 8.4-1 usada até agora somente para adicionar os números nos registradores R_α e R_β. Suponhamos que queremos subtrair ao invés de somar para formar e armazenar $R_\alpha - R_\beta$. É, na verdade, com este detalhe na mente que incorporamos à máquina o regis-

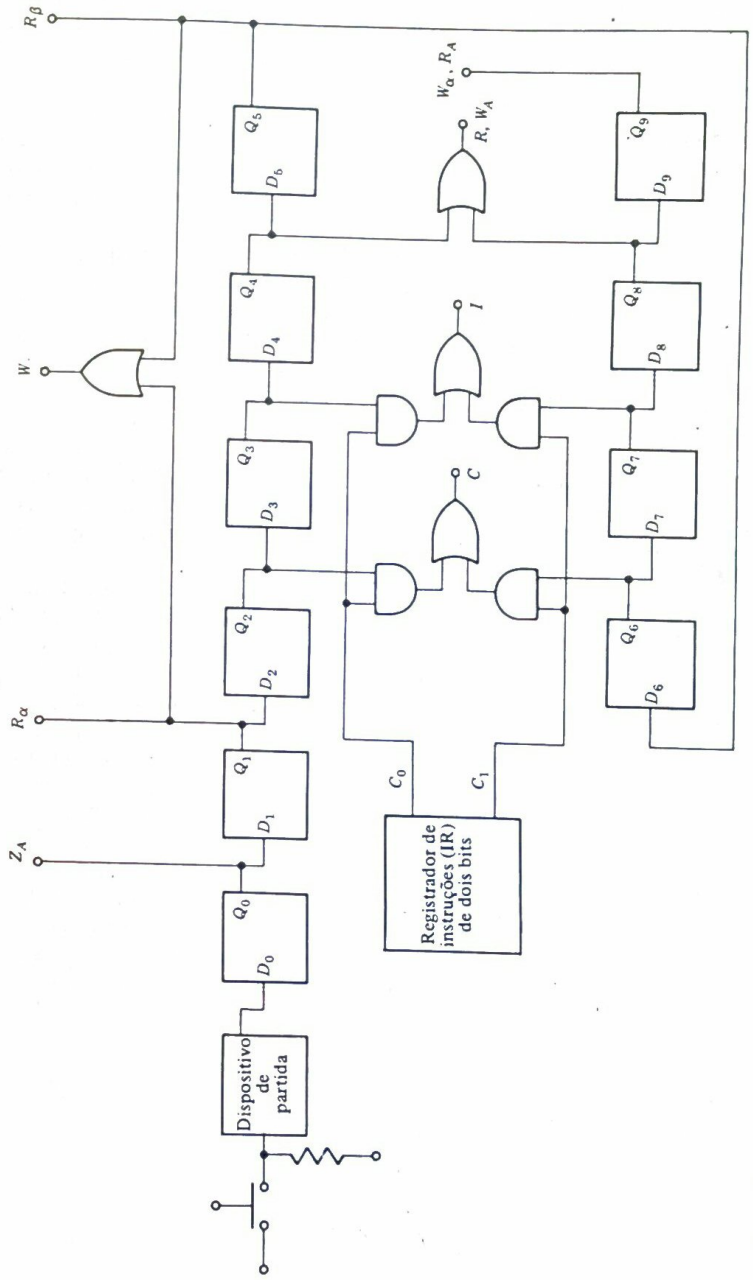


Fig. 8.8-1 Um controlador de registrador de deslocamentos a ser usado com a arquitetura da Fig. 8.4-1. O controlador colocará no registrador R_α uma das quatro quantidades $\pm R_\alpha$ ou $\pm R_\beta$ dependendo da instrução no registrador de instruções.

trador de complementar-incrementar (CI), que realmente não serve a nenhum propósito na formação da soma. Se quisermos formar $R_\alpha - R_\beta$, temos apenas que inverter o sinal do conteúdo de R_β, formando, desta maneira, $-R_\beta$, e depois adicionar $R_\alpha + (-R_\beta)$. Admitimos, como antes, que números negativos devem ser representados em forma de complemento de dois. O complemento de dois é gerado complementando-se cada bit individual do número e depois incrementando-se o número de 1. Daí, em conjunto, para formar $R_\alpha - R_\beta$, primeiro transferimos R_α através de CI para o registrador do acumulador como antes; depois transferimos R_β para CI, onde, então, primeiro *complementamos* e depois *incrementamos* antes da transferência de CI para o acumulador.

Para realizar a subtração, necessitamos modificar o controlador para proporcionar duas microoperações adicionais, que por sua vez exigem dois estados adicionais. Se projetarmos um controlador com mínimo de estados, a tabela de estados terá dois estados adicionais. Uma vez que nosso controlador original tinha sete estados, o novo controlador terá nove estados e quatro flip-flops serão necessários. Além disto, teremos que modificar a lógica que usa a seqüência através de seus passos, e teremos que acrescentar lógica ao decodificador. Se projetarmos um controlador do registrador de deslocamentos, teremos que acrescentar dois flip-flops à cadeia do registrador de deslocamentos. Os detalhes de cada um destes projetos é deixado como exercício ao estudante.

A seguir, suponhamos que queremos formar $-R_\alpha + R_\beta$. Partindo então novamente do controlador original da Fig. 8.5-2 ou Fig. 8.6-2, acrescentaríamos duas microoperações, desta vez para formar o negativo do número em R_α. Se quiséssemos $-R_\alpha - R_\beta$, teríamos que acrescentar quatro microoperações.

Poderíamos também decidir que quatro diferentes controladores para realizar as quatro operações $R_\alpha + R_\beta$, $R_\alpha - R_\beta$, $-R_\alpha + R_\beta$ e $-R_\alpha - R_\beta$ não são realmente exigidos. Em vez disto poderíamos projetar um simples controlador que pudesse efetuar qualquer uma das quatro operações dependendo da *instrução* que déssemos a ele. Um tal controlador é dado na Fig. 8.8-1. Com o propósito de guardar a *instrução*, acrescentamos um *registrador de instrução de dois bits*. Consideremos aqui que a instrução está colocada no registrador de instruções pela manipulação manual usando chaves, exatamente como registramos os números a serem combinados, isto é, os operandos, nos registradores R_α e R_β. Ao controlador original da Fig. 8.6-2 também acrescentamos quatro estados adicionais. Nestes estados os números no registrador CI serão complementados e incrementados, conforme requerido, para trocar o sinal dos operandos. A complementação no registrador CI é efetuada quando o terminal C na Fig. 8.8-1 (conectado a C na Fig. 8.4-1) vai para lógica 1. A incrementação é efetuada quando o terminal I vai para lógica 1. Quando as linhas de instruções de bit são $C_0 = C_1 = 0$, todas as portas AND na Fig. 8.8-1 estão desativadas, C e I estão sempre em lógica 0, e o resultado líquido é que a operação total efetuada é para gerar a soma $R_\alpha + R_\beta$. A operação total para todas as quatro instruções possíveis é dada na Fig. 8.8-2.

Instrução		Operação total
C_1	C_0	
0	0	$R_\alpha + R_\beta$
0	1	$-R_\alpha + R_\beta$
1	0	$R_\alpha - R_\beta$
1	1	$-R_\alpha - R_\beta$

Fig. 8.8-2 O código C_1C_0 de instruções para as quatro instruções do controlador da Fig. 8.8-1.

8.9 UM COMPUTADOR SIMPLES

O *controlador* da Fig. 8.8-1, operando em conjunto com *registradores de armazenamento* e *registradores controláveis* organizados na arquitetura da Fig. 8.4-1, nos permitirá combinar dois números aritmeticamente por adição e subtração. Para usar a máquina, colocaríamos números em R_α e R_β, presumivelmente de forma manual usando chaves, uma vez que não tomamos outras providências. Depois, manualmente colocaríamos uma instrução no registrador de instruções (IR). Depois disto, tudo prossegue automaticamente. Necessitamos somente calcar o botão de partida e, depois de certo tempo, acharemos nosso resultado em R_α.

Suponhamos, agora, que queremos tornar nossa máquina mais elaborada. Primeiro, queremos ser capazes de lidar com mais do que dois números, ou operandos. Assim, podemos efetuar uma adição de apenas dois operandos, mas queremos ser capazes de selecionar a partir de um grande número deles, ou seja, dispondo de um grande número de operandos, combinar muitos deles. Uma modificação óbvia que é depois exigida consiste em substituir os registradores R_α e R_β por um grande conjunto de registradores. Este grande conjunto de registradores é, naturalmente, uma memória, como descrita no Cap. 6. Se quisermos ser capazes de mudar operandos facilmente, é requerida uma RAM'

No controlador da Fig. 8.8-1 inúmeras etapas da seqüência tratam do operando armazenado em R_α, e inúmeras etapas tratam do operando em R_β. É evidente que a obediência a este padrão levaria a uma seqüência muito longa se muitos operandos estivessem envolvidos. Cada novo operando acrescenta etapas à seqüência. Reconhecemos, no entanto, que cada operando é realmente submetido às mesmas microoperações. O operando é transferido de um registrador de armazenamento (ou melhor, de um local na memória) para o registrador CI e daí através do somador para o acumulador. Se necessitarmos mudar o sinal do operando, complementamos e incrementamos no registrador CI. Do contrário, o registrador CI não faz nada. Esta seqüência de microoperações é repetida sem cessar para cada operando. Daí parece que realmente podemos dispor de um controlador que tem uma seqüência curta, uma seqüência que processará apenas um operando. No entanto, com este controlador de seqüências, necessitaríamos de algum mecanismo para ajustar a instrução ao controlador, isto é, mudar o sinal do operando ou não mudar o sinal, quando cada novo operando for processado. Um modo de ajustar a instrução é simplesmente interromper a seqüência de controle depois que ele processa cada operando e em seguida mudar manualmente a instrução antes de permitir que a seqüência processe o operando seguinte. Mas, já que temos uma memória, podemos armazenar nela a informação concernente a como cada operando deve ser processado e a operação inteira pode ser feita de forma automática. Com estas considerações em mente, voltemos agora à Fig. 8.9-1; ela expõe um sistema que nos permitirá (com alguma intervenção manual) combinar aritmeticamente um grande número de operandos automaticamente.

O sistema tem uma memória RAM. Para sermos explícitos, admitamos uma memória com 64 palavras, cada palavra tendo oito bits. Um local da memória é endereçado por um endereço de seis bits ($2^6 = 64$). A memória tem uma entrada de HABILITA (*ENABLE*) e uma entrada de *leitura/escrita* (*read/*$\overline{write}$). Quando *HABILITA* (*ENABLE*) = 1, a memória está conectada ao bus (oito bits), e quando *HABILITA* (*ENABLE*) = 0, o bus está isolado da memória. Quando *HABILITA* (*ENABLE*) = 1, a memória lerá uma palavra sobre o bus ou escreverá uma palavra na memória, dependendo de *leitura/*$\overline{escrita}$ ser 1 ou 0. O local da memória a partir do qual uma palavra é lida ou no qual uma palavra é escrita é determinado pelos seis bits de endereço. A seta de ponta dupla que conecta a memória ao bus indica que a transferência de informações entre o bus e a memória é bidirecional.

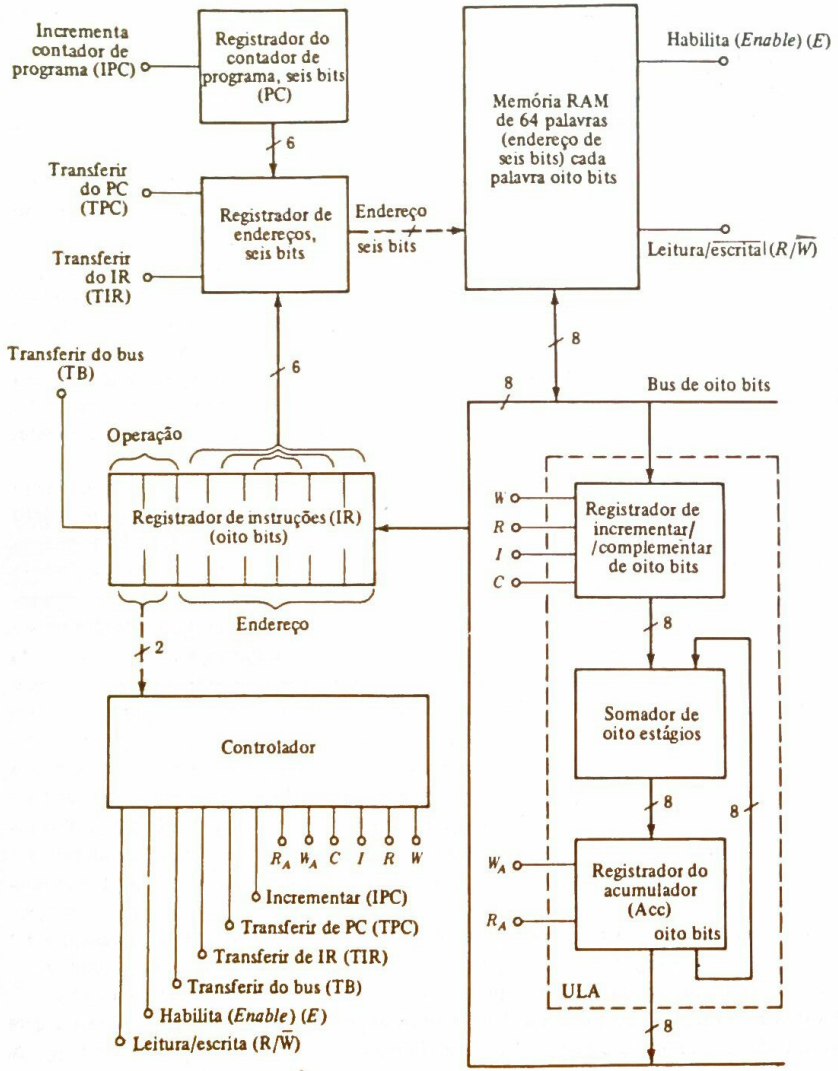


Fig. 8.9-1 A arquitetura de um "computador" que permitirá a combinação aritmética de um grande número de operandos.

Na Fig. 8.9-2 ilustramos o que tipicamente pode estar contido na memória que serve à nossa máquina um tanto simples. Na Fig. 8.9-2*a* escrevemos por extenso em palavras e números decimais o conteúdo de alguns dos locais na memória. Nos locais 0 a 6 escrevemos um *programa* de instruções.

Local da memória	(a)	Local da memória	(c)
			Subtrair de Acc — Local 59 da memória
0	Subtrair de Acc conteúdo de local 59 da memória	0 0 0 0 0 0	1 0 1 1 1 0 1 1
1	Somar ao Acc conteúdo de local 60 da memória	0 0 0 0 0 1	0 1 1 1 1 1 0 0
2	Subtrair de Acc conteúdo de local 61 da memória	0 0 0 0 1 0	1 0 1 1 1 1 0 1
3	Somar ao Acc conteúdo de local 62 da memória	0 0 0 0 1 1	0 1 1 1 1 1 1 0
4	Somar ao Acc conteúdo de local 63 da memória	0 0 0 1 0 0	0 1 1 1 1 1 1 1
5	Transferir conteúdo de Acc para local 39 da memória	0 0 0 1 0 1	1 1 1 0 0 1 1 1
6	Parar (*stop*)	0 0 0 1 1 0	0 0 x x x x x x
59	49	1 1 1 0 1 1	0 0 1 1 0 0 0 1
60	−79	1 1 1 1 0 0	1 0 1 1 0 0 1 1
61	−52	1 1 1 1 0 1	1 1 0 0 1 1 0 0
62	121	1 1 1 1 1 0	0 1 1 1 1 0 0 1
63	82	1 1 1 1 1 1	0 1 0 1 0 0 1 0

Código	Instrução
00	Parar (*stop*)
01	Somar a Acc
10	Subtrair de Acc
11	Transferir conteúdo de Acc para

(b)

Fig. 8.9-2 (*a*) Um possível conteúdo da memória na Fig. 8.9-1 escrito para transportar o intento do conteúdo da memória. (*b*) Um código de instruções. (*c*) O conteúdo real de bits binários dos locais da memória.

No outro extremo da memória armazenamos alguns operandos. Se pudermos arranjar para que a máquina execute as instruções dadas na ordem listada, a máquina computará a quantidade $-(49) + (-79) - (-52) + (121) + (82) = +127$, como pode ser visto observando os conteúdos dos locais 59 a 63 da memória. A máquina então transferirá esta soma acumulada do registrador do acumulador para o local 39 da memória e depois interromperá e esperará pela intervenção humana. Conforme veremos, há um propósito na colocação das instruções em locais sucessivos na memória. Por outro lado, não é importante que os operandos (49, −79 etc.) estejam localizados em locais sucessivos. Nem é importante que tenhamos colocado as instruções nas extremidades opostas da memória. Tudo o que é requerido é que as instruções sejam colocadas (em locais sucessivos) numa parte da memória e os operandos sejam colocados em outro lugar. A ordem indicada na Fig. 8.9-2 é sugerida principalmente por um respeito característico à escrituração contábil bem organizada. Os locais da memória não indicados na figura têm naturalmente algum conteúdo. (Um local limpo da memória com um conteúdo 00 . . . 00 tem um conteúdo apesar disto.) Mas o conteúdo destes locais da memória é irrelevante para nossa presente discussão. Propomos armazenar nosso resultado no local 39 (não mostrado). O conteúdo do local 39 é desconhecido e irrelevante. Quando for executada a instrução que grava o resultado no local 39, o conteúdo anterior deste local será perdido.

Notemos que usamos quatro instruções: somar, subtrair, transferir e parar. Arbitrariamente, usamos o código de dois bits da Fig. 8.9-2*b* para representar estas instruções. Na Fig. 8.9-2*c* reescrevemos a memória em forma binária. Nos locais da memória que retêm operandos, simplesmente substituímos números decimais por números binários. Em locais da memória que retêm instruções, arbitrariamente arranjamos para que os dois bits mais à esquerda representassem a instrução e os seis bits restantes especificassem o local que retém o operando. Esta atribuição de significância de bit é expressamente indicada para o primeiro local da memória. Os locais 0 a 5 da memória retêm, todos, instruções que exigem uma operação e se referem a um local de memória específico quando o operando é armazenado. O local 6 da memória requer uma operação, mas, uma vez que nenhum operando é envolvido, não nos interessamos pelos seis bits de endereço.

Observemos que as palavras da memória têm oito bits de comprimento. Se propusermos representar números negativos em forma de complemento de dois, a faixa de números que podemos acomodar estende-se de +127 a −128. Devemos, então, ser cuidadosos para assegurar que quando nossa máquina acumula os números que estamos combinando, nunca exijamos que a soma acumulada se estenda para fora da faixa permitida. Os números introduzidos nos locais 59 a 63 da memória foram selecionados arbitrariamente, exceto que observamos esta restrição a respeito da faixa.

Retornando à Fig. 8.9-1, notamos que a seção na caixa (quadrículo) desenhada é o sistema da Fig. 8.4-1 sem os registradores R_α e R_β. (Estes registradores foram substituídos pela parte da memória que armazena os operandos.) Este pequeno sistema efetua aritmética (somando e incrementando) e lógica (complementando) e é com razão denominado Unidade Lógica e Aritmética (ULA), embora difira em muitos aspectos da ULA da Fig. 5.13-1. Por simplificação, deixamos de fora o terminal Z_A de *limpar*, uma vez que o acumulador pode ser limpo usando as instruções da Fig. 8.9-2*b* (Prob. 8.9-1).

Continuando nosso exame da Fig. 8.9-1, notamos a presença de um registrador contador de programa (PC), de um registrador de instruções (IR), de um registrador de endereços da memória (MAR) e, finalmente, do controlador que é para colocar nosso pequeno computador à prova e que temos ainda que projetar. O propósito detalhado de cada registrador será discutido em breve. No momento, observemos simplesmente os caminhos disponíveis para transferências de palavras

(ou de partes de palavras) e as facilidades que foram incorporadas a cada registrador. Notemos aqui que há um bus de oito bits a que a memória tem uma conexão bidirecional. A ULA pode aceitar uma palavra do bus sobre seu lado de entrada e fornecer uma palavra ao bus de seu lado de saída. O registrador de instruções pode aceitar uma palavra do bus. O registrador de endereços da memória tem duas entradas de controle que podem ser usadas para transferir para o MAR a palavra de seis bits no contador de programa ou os seis bits mais à direita do registrador de instruções. Os dois bits mais à esquerda do registrador de instruções são tornados disponíveis (não transferidos) para o controlador. Por conseguinte, esta conexão de dois bits é indicada por uma linha tracejada ao invés de linha cheia. A única operação que o contador de programas pode executar é a operação de incrementar. Finalmente, o controlador tem uma linha de saída de controle correspondente a cada linha de entrada de controle de cada registrador e da memória. Quaisquer microoperações que sejam efetuadas serão executadas sempre que a correspondente linha de controle for para o nível de habilitação. Com exceção da memória, todos os registradores e o controlador são sincronizados (clocados), e o momento real em que um registrador aceita uma transferência e incrementa é o instante de tempo da transição de gatilho do relógio. A linha da forma-de-onda de relógio, que é distribuída em todo o sistema da Fig. 8.9-1, não está indicada no desenho.

8.10 OPERAÇÃO DO COMPUTADOR

Para ver como opera nosso computador, consideremos que nossa memória é carregada como na Fig. 8.9-2*c*. Podemos imaginar que para efetuar este carregamento a memória foi temporariamente desconectada do sistema e que as entradas (endereços, dados, *habilita* (*enable*), leitura/escrita) foram aplicadas manualmente. Admitamos também que, no início, o contador do programa (PC) e o registrador do acumulador estão limpos. (Não importa se os outros registradores estão limpos inicialmente.) Listaremos agora, nas Tabs. 8.10-1 e 8.10-2, ciclo de relógio por ciclo de relógio, a seqüência de microoperações através das quais o controlador deve regular o andamento do sistema para que este tome nota da primeira instrução, execute seu intento e prepare para ler a próxima instrução. Quando for viável efetuar mais do que uma microoperação durante o curso de um ciclo de relógio, tiraremos vantagem deste aspecto característico a fim de economizar tempo. Deve ser notado especialmente que as únicas operações especificadas na tabulação são aquelas que serão afetadas pelo fato de que uma ou mais saídas de controle do controlador vão para o nível de habilitação.

Nas operações listadas na Tab. 8.10-1 trouxemos as primeiras instruções da memória para dentro do registrador de instruções. Esta parte do ciclo de operações da máquina é chamada *ciclo de busca* (*fetch cycle*). Agora que a primeira instrução está disponível, a máquina prosseguirá para responder às instruções. O ciclo das operações pelas quais esta resposta é executada é chamado *ciclo de executar*. As operações do ciclo de busca são, naturalmente, executadas sob o controle do controlador, mas a operação do controlador durante o ciclo de busca é independente dos dois bits mais à esquerda no registrador de instruções, que estão disponíveis para o controlador. Em verdade, já que não limpamos o registrador de instruções, nem mesmo soubemos quais eram estes bits. No entanto, agora que buscamos esta primeira instrução e a registramos no registrador de instruções, o processamento a partir deste ponto seguirá um curso que depende da

Tabela 8.10-1 Ciclo de busca (fetch cycle)

Ciclo de relógio	Descrição simbólica da operação	Linha de controle a ser habilitada
1. Transferir conteúdo do contador de programa para o registrador de endereços da memória	PC → MAR	TPC
2. Transferir instrução endereçada (no local 000000) para o registrador de instruções por (1) habilitação da memória para conectar a memória ao bus, (2) colocação de $R/\overline{W}$ em 1 para ler memória, e (3) transferência da palavra no bus para o registrador de instruções; incrementar contador de programa para preparar para fazer aparecer a próxima instrução quando a primeira instrução tiver sido concluída	M → IR ("M" representa palavra de memória endereçada) PC + 1 → PC	E, $R/\overline{W}$, TB IPC

instrução transportada pelos dois *bits de operação* da instrução. Uma vez que a primeira instrução requer *subtração*, a *execução* da instrução prossegue conforme mostrado na Tab. 8.10-2.

A máquina agora executou as primeiras instruções. Providenciaremos para que, correspondentemente, o controlador que projetamos tenha completado toda sua seqüência e retornado a seu início. Por conseguinte, a próxima operação a ser executada é novamente transferir o contador do programa para o registrador de endereços da memória. Mas lembremos que incrementamos o contador de programas. Em conseqüência, a instrução buscada será a segunda instrução (no local 00001). A segunda instrução será executada como a primeira, exceto que, uma vez que a adição é requerida ao invés da subtração, as operações de complementar e incrementar serão des-

Tabela 8.10-2 Ciclo de executar

Ciclo de relógio	Descrição simbólica da operação	Linha de controle a ser habilitada
3. Transferir parte do endereço do registrador (seis bits à direita) para o registrador de endereços da memória (endereço é 59)	IR (ADD) → MAR	TIR
4. Transferir palavra endereçada da memória para o bus e do bus para o registrador CI	M → BUS BUS → CI	E, $R/\overline{W}$, W
5. Complementar CI	$\overline{CI}$ → CI	C
6. Incrementar CI	CI + 1 → CI	I
7. Registrar saída do somador no registrador do acumulador	Adder → Acc	W_A

viadas. Assim vemos que, quando o controlador se põe em seqüência, ele busca e depois executa, busca depois executa etc. A operação de busca é sempre a mesma; as operações durante o executar dependem, naturalmente, da instrução.

Ao construir o sistema simples da Fig. 8.9-1 tomamos inúmeras decisões (algumas das quais arbitrárias e algumas baseadas na experiência). Estas decisões referem-se ao número e à função dos registradores, aos tipos de interligações entre os registradores e entre os registradores e a memória, ao número de palavras na memória, ao número de bits por palavra, ao número e tipo de operações que a ULA pode efetuar etc. Estes aspectos constituem a arquitetura e a organização do computador. Uma vez que tenham sido estabelecidas uma arquitetura e uma organização, permanece a tarefa do projeto do controlador. (O controlador de nossa máquina é projetado na próxima seção.)

Há, naturalmente, muitas arquiteturas e organizações que podem ser admitidas por uma peça da maquinaria de computação. Depois de muitos anos de experimentação com uma ampla faixa de possibilidades, a maquinaria de computação da época atual geralmente incorpora alguns aspectos comuns, alguns dos quais devem ser vistos em nosso computador simples e que destacamos agora.

Notemos primeiro que a máquina tem uma memória, em que *armazenamos* de início todas as instruções requeridas para executar uma computação. Por esta razão, a máquina é denominada *computador de programa armazenado*. Uma vez que a máquina está plenamente instruída, não teríamos que interrompê-la para proporcionar outra direção para sua computação. A memória armazena não somente as instruções mas também os operandos e os resultados da computação. Daí, teremos que fazer freqüentes referências à memória para ler dela e escrever nela. O local endereçado é achado no *registrador de endereços da memória* (MAR) e não é surpreendente que haja acesso ao MAR de inúmeras direções. Em nosso caso, podemos chegar ao MAR a partir do contador de programas e do registrador de instruções. (Em computadores mais sofisticados, meios de acesso adicionais diretos e indiretos ao MAR são fornecidos.)

Observemos a seguir que nossa máquina tem uma unidade lógica e aritmética em que todas as operações aritméticas e lógicas são efetuadas. O resto da máquina (além do controlador) consiste em nada mais que um conjunto de registradores. Mesmo a memória nada mais é que uma coleção de registradores de armazenamento. E os registradores de armazenamento não fazem nada. Eles constituem apenas o "papel de escrita" digital em que escrevemos as palavras digitais que necessitamos guardar para futura referência. Nossa ULA também é um tanto simples. Ela complementa, incrementa e adiciona. ULAS mais elaboradas proporcionam estas funções e também efetuam operações lógicas (AND, OR etc.), deslocam à esquerda ou à direita etc.

Observemos que na manipulação da máquina providencia-se para que inúmeros assuntos sejam "entendidos". Quando uma instrução é executada, é "entendido" *que a próxima instrução está no próximo local da memória. Daí, o local da próxima instrução não necessita ser especificado*. Novamente, quando deve ser efetuada uma adição, a instrução especifica somente um dos operandos envolvidos na adição. Está "entendido" que o outro operando está no registrador do acumulador. Finalmente, a instrução não dá nenhuma indicação de onde o resultado da adição deve ser armazenado. Está "entendido" que o resultado deve ser deixado no acumulador. Todos estes entendimentos efetuam uma considerável economia no comprimento das palavras. Graças a estes entendimentos que foram incorporados à máquina, uma instrução necessita apenas especificar uma operação e um endereço de um operando. Sem estes entendimentos uma instrução teria que especificar a operação, a fonte do primeiro operando, a fonte do segundo operando, a casa onde o resultado deve ser armazenado e a fonte da próxima instrução. Naturalmente,

uma tal instrução elaborada exigiria muito mais bits do que as instruções mais simples possíveis, devido aos entendimentos.

8.11 PROJETO DO CONTROLADOR DO COMPUTADOR

Um projeto do controlador requerido na Fig. 8.9-1 é mostrado na Fig. 8.11-1. Neste projeto, a fim de obter simplificação, apresentamos um deliberado e extremo descaso pela economia de hardware. Como sempre, o *elemento de partida* e todos os flip-flops são comandados por uma forma-de-onda de sincronização comum (não mostrada explicitamente). O fechamento da chave dá partida à seqüência do registrador de deslocamentos. Quando a saída de cada flip-flop sucessivo cresce alternadamente para lógica 1, é executada uma microoperação. Estabelecemos em palavras quais microoperações são executadas em cada etapa e indicamos entre parênteses quais terminais de controle devem ser tornados ativos para que as microoperações sejam executadas. As duas primeiras etapas na seqüência *buscam* (*fetch*) a instrução e carregam-na dentro do registrador de instruções. Esta parte de busca da operação do controlador é a mesma, independentemente da instrução buscada. Depois de ter sido concluída a parte de busca da seqüência, começa a parte de *executar*. A operação a ser executada é transmitida ao controlador como um código de operação de dois bits. No controlador estes bits são aplicados a um decodificador com quatro linhas de saída, Quando o código de operação for 10, é pretendida a subtração. A linha de saída do decodificador 10 sozinha está em lógica 1 e apenas a porta AND superior está habilitada. A seqüência do registrador de deslocamentos continua através da fila superior de flip-flops, arranjando desta maneira para executar a seqüência apropriada à subtração. Um código 01 de operação habilita a segunda porta AND da parte superior. A seqüência resultante é a mesma da subtração, exceto que são descartadas as microoperações de *complementar* e *incrementar*. Um código 11 de operação exige uma transferência do conteúdo do registrador do acumulador de volta para a memória. Aqui são requeridas duas microoperações. A primeira, como nas operações de subtração e adição, transfere o endereço do registrador de instruções para o registrador de endereços da memória. A segunda *escreve* na memória em vez de ler da memória, daí a inversão de nível lógico aplicada ao controle $R/\bar{W}$ da memória.

Se a instrução for *subtrair, somar* ou *transferir*, a seqüência de executar, quando concluída, leva imediatamente de volta através da porta OR ao começo do ciclo de busca. A próxima instrução é buscada, e começa uma nova execução etc. Se, por outro lado, a instrução for 00, o que significa *parar* (*stop*), não há retorno ao ciclo de busca. A operação pára depois da saída da porta AND mais inferior ter estado em lógica 1 durante um ciclo de relógio. Esta saída "acabada" pode ser usada, se quisermos, para indicar que o computador concluiu seu trabalho e agora requer outra intervenção manual.

A porção de executar do controlador da Fig. 8.11-1 usa flip-flops um tanto mais livremente do que o necessário. Uma porção alternativa de executar mostrada na Fig. 8.11-2 é mais econômica em flip-flops, embora às expensas de exigir mais portas. São usados cinco flip-flops, permitindo uma seqüência de executar com outras tantas cinco microoperações. Em cada caso, exceto quando a instrução for parar (*halt*), a primeira etapa da seqüência exige que o terminal TIR seja elevado ao nível ativo. Na segunda etapa, as instruções *somar* e *subtrair* exigem uma microoperação comum, enquanto a instrução *armazenar* requer uma microoperação alternativa. A menos que a instrução seja subtrair, as terceira e quarta etapas da seqüência são descartadas. Se for requerida a subtração, as terceira e quarta microoperações são *complementar*

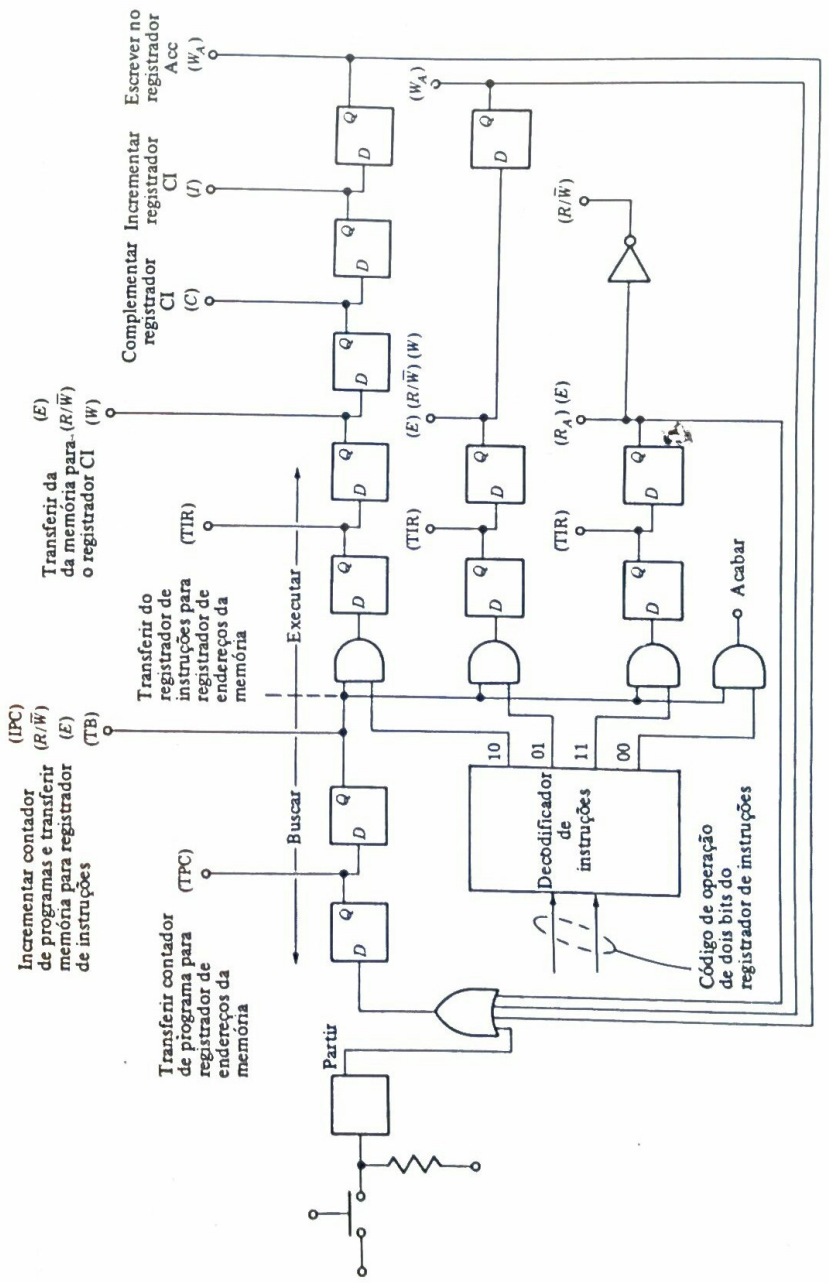


Fig. 8.11-1 O controlador do "computador" da Fig. 8.9-1.

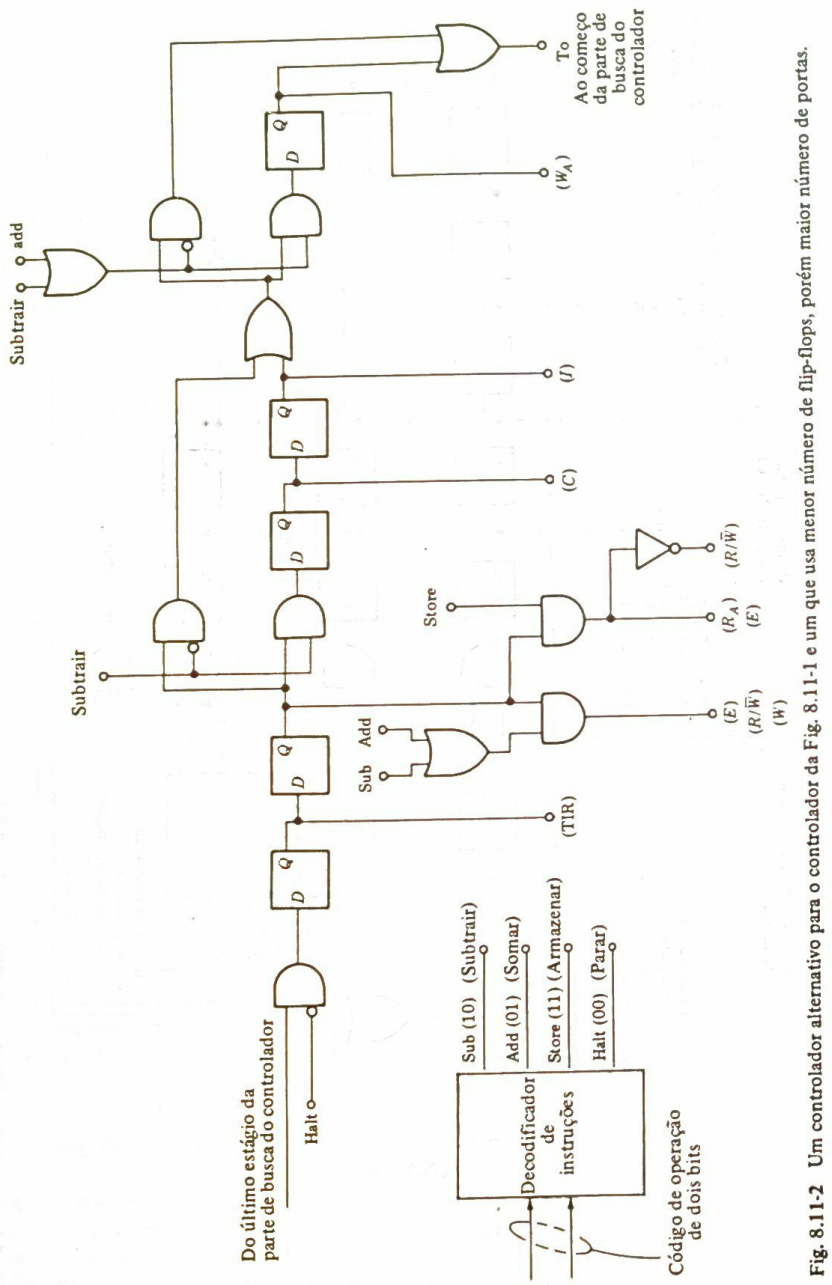


Fig. 8.11-2 Um controlador alternativo para o controlador da Fig. 8.11-1 e um que usa menor número de flip-flops, porém maior número de portas.

e *incrementar*. Finalmente, outra vez, se for requerida *somar* ou *subtrair*, há uma ativação comum da entrada W_A, conforme proporcionado pela quinta etapa na seqüência. Do contrário, esta quinta etapa é descartada.

8.12 INTERRUPÇÃO

Uma vez posto a funcionar, o controlador da Fig. 8.11-1 corre em todas as direções. Ele executa uma seqüência de busca e depois uma seqüência de execução, volta para uma seqüência de busca etc. repetidas vezes até que, finalmente, ele encontra uma instrução de parar (*halt*). O controlador opera o processador digital de idealização um tanto simples da Fig. 8.9-1, que é capaz de fazer não muito mais do que somar e subtrair uma coluna de números. No entanto, o modo de operação deste controlador (buscar, executar, buscar etc.) é de importância fundamental porque os controladores na maioria dos sofisticados computadores digitais operam neste mesmo modo.

Freqüentemente, acontece que é necessário *interromper* esta operação cíclica ou repetitiva de buscar-executar e convocar o controlador para gerar uma seqüência de sinais de comando de habilitação para executar alguma operação especial. A fonte que requer a operação especial é descrita como *requerendo uma interrupção*, e a resposta correspondente do controlador é descrita como *respondendo a um pedido de serviço*.

A Fig. 8.12-1 mostra como o controlador pode ser aumentado para incorporar a capacidade de responder a um tal pedido de serviço. Se o latch estático estiver no estado reset com $\bar{Q} = 1$, a porta AND designada G_N em operação normal é habilitada, e a porta de uma operação de interrupção G_I é desabilitada. Depois do botão de partida ter sido calcado, começará uma seqüência normal, primeiro pela porção de busca do controlador, depois pela porção de executar do controlador, depois de volta à busca, e assim por diante. Suponhamos que num certo instante um sinal seja recebido indicando que a seqüência de serviço é requerida. O sinal que requer a interrupção necessita apenas ser uma breve excursão para lógica 1 aplicada à entrada set do latch. Quando o latch se ajusta ou se estabelece ($Q = 1, \bar{Q} = 0$), será habilitada G_I em vez de G_N. Suponhamos que a primeiríssima etapa da seqüência de busca já tenha começado antes do sinal de interrupção ser recebido. Então a seqüência de busca continuará e será seguida pela seqüência de executar. Assim, a instrução que está no processo de ser buscada e executada não será interrompida, mas quando a execução tiver sido concluída, desde que G_I esteja habilitada e G_N não, a colocação em seqüência prosseguirá através do gerador de seqüências auxiliar para fazer o serviço da interrupção. Quando a seqüência prossegue assim, serão gerados sinais de comando para ativar quaisquer microoperações que sejam requeridas. Providenciamos para que o último sinal de comando, além de qualquer microoperação que ele efetue, também venha a reajustar (reset) o latch. Com o latch reajustado, o controlador voltará para operação normal.

O latch na Fig. 8.12-1 é requerido para armazenar a informação de que alguma fonte fez um pedido de uma interrupção porque ela requer prestação de serviço. Tal armazenamento é requerido porque o sinal de entrada de interrupção pode ser de curta duração e nenhuma resposta pode ser feita pelo controlador até que ele tenha completado o tratamento ou procedimento com a instrução no processo a ser executado. "Registradores" de um bit, como o latch, que não registram uma palavra mas servem para armazenar uma certa parte de informação de um bit para alertar o controlador em alguma situação, são chamados *flags* (bandeiras). Assim, a chamada de uma interrupção *eleva o flag de interrupção* e, quando a interrupção for atendida, o flag será abaixado.

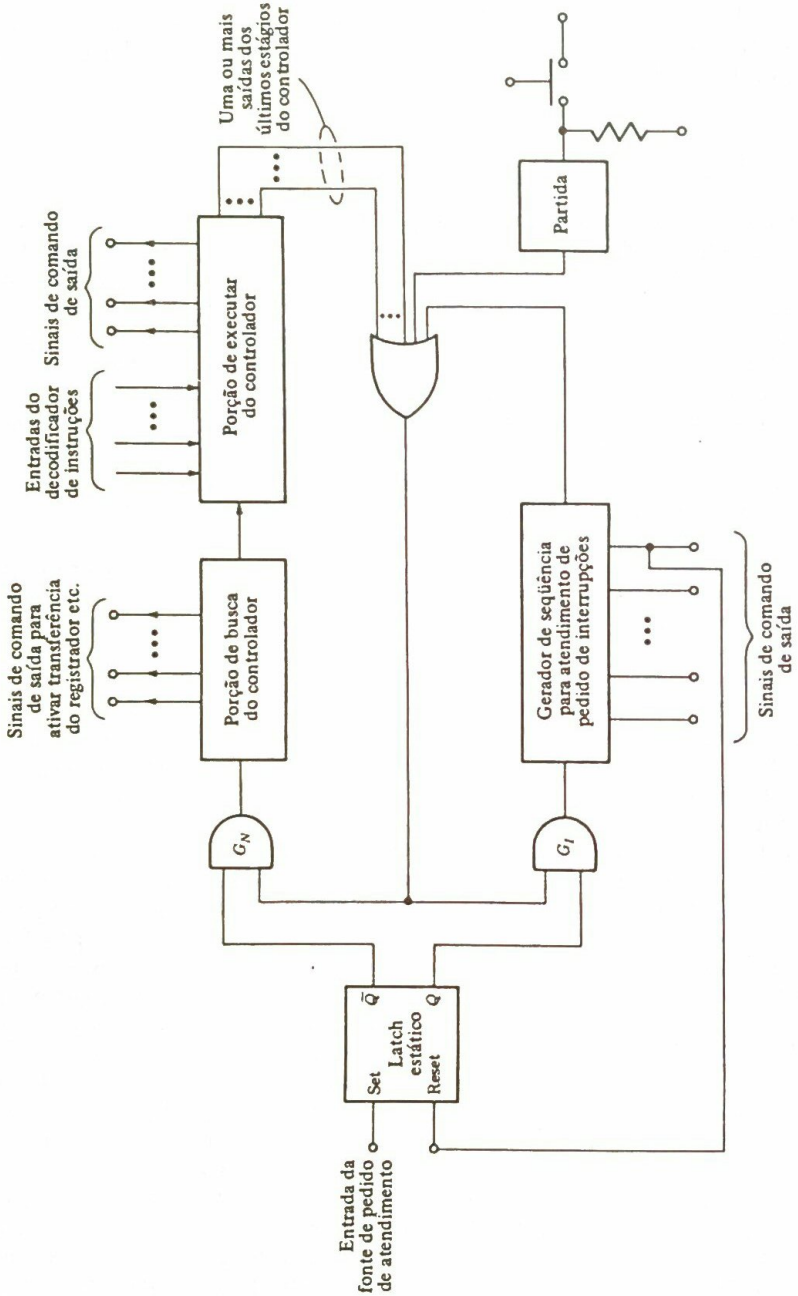


Fig. 8.12-1 Uma modificação do controlador que permite resposta a um pedido de *interrupção*.

Geralmente, um processador digital, seja ele um computador digital ou mesmo algum processador de simplicidade comparável à nossa máquina da Fig. 8.9-1, não está sozinho. Inevitavelmente, ele irá operar em conjunto com algum equipamento que é exterior ao próprio processador. Em muitos casos, este equipamento exterior serve como o recurso pelo qual o processador pode comunicar-se com os seres humanos. Este equipamento pode consistir, por exemplo, em uma máquina de escrever eletromecânica através da qual as pessoas podem *introduzir* dados e instruções no processador e através da qual o processador pode *lançar* seus resultados numa forma que é convenientemente entendida pelas pessoas. Este hardware externo é então denominado equipamento de entrada-saída (I/O); quase todo processador necessita incorporar uma capacidade de controlar este hardware de entrada-saída em seu controlador. Freqüentemente, a facilidade de interrupção de um controlador é requerida precisamente para auxiliar na manipulação do hardware de I/O.

Um simples exemplo de como uma chamada de interrupção pode ser atendida está indicado na Fig. 8.12-2. Aqui consideramos que nosso processador de somar-subtrair da Fig. 8.9-1 está combinando uma longa lista de números, e a taxa de processamento (determinada pela taxa de relógio) é bastante lenta para nós para que razoavelmente tenhamos um interesse, de vez em quando, em testar para ver qual é a acumulação total presente no acumulador. Além disto, a nosso pedido, queremos arranjar para que o registro no acumulador seja datilografado numa máquina de escrever. Admitimos a disponibilidade de uma máquina de escrever eletromecânica que responderá, conforme exigimos, à ativação lógica de linhas de entrada.

Na Fig. 8.12-2 reproduzimos, da Fig. 8.9-1, somente o bus e a ULA e, da ULA, somente o acumulador. O restante do processador não é relevante para nossos propósitos atuais. Para conseguir nosso propósito, calcamos um botão de partida, que fornecemos para indicar ao processador que queremos que ele interrompa sua operação normal e datilografe o último registro do acumulador. O acumulador deve transferir para o registrador externo. Aqui os quatro bits menos significativos e os quatro bits mais significativos são separadamente decodificados por dois decodificadores hexadecimais. Cada decodificador proporciona dezesseis linhas de saída, somente uma de cada dezesseis estando em lógica 1, dependendo das significações numéricas dos códigos hexadecimais de entrada. Suponhamos uma máquina de escrever com os dezesseis caracteres hexadecimais e admitamos igualmente que, quando o terminal "Datilografar MSB" ("Datilografar Bit Mais Significativo") for elevado à lógica 1, a máquina de escrever ativará a tecla correspondente à linha ativa na saída do decodificador do MSB. Uma entrada correspondente à entrada da máquina de escrever do LSB é admitida. E, finalmente, admitimos uma entrada que desloca o carro de um espaço de modo que cada número hexadecimal de dois dígitos pode ser separado dos outros. Conforme também é mostrado na figura, para realizar o que é requerido, o gerador de seqüências para atendimento do pedido de interrupção deve ter quatro sinais de comando. Cada vez, então, que calcamos o botão de interrupção, o processador completará o tratamento com sua instrução presente e irá para a seqüência de atendimento para responder nosso pedido.

Uma dificuldade é imediatamente evidente no esquema da Fig. 8.11-2. Se construirmos nosso gerador de seqüências como os exemplos apresentados de início, teremos permitido apenas um intervalo de relógio para cada operação (transferência, datilografia e deslocamento). A transferência do acumulador para o registrador externo é facilmente conseguida num intervalo de relógio, mas, a menos que a taxa de relógio seja muito lenta, um intervalo de relógio não será adequado para as respostas mecânicas requeridas da máquina de escrever. Um modo de resolver dificuldades devidas à resposta lenta consiste em aumentar de um para muitos o número de

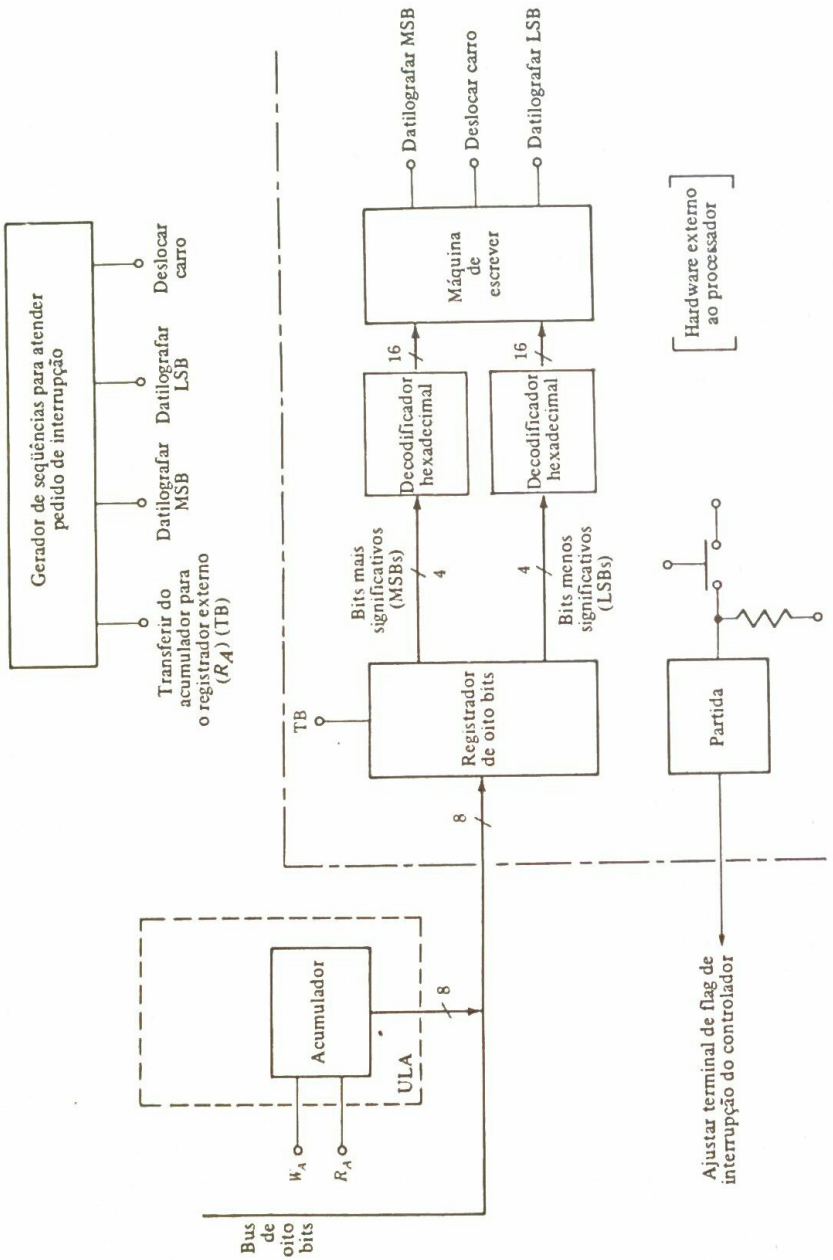


Fig. 8.12-2 Um exemplo. O atendimento de uma interrupção que solicitou uma leitura de saída do conteúdo do acumulador.

flip-flops num gerador de seqüências entre dois pontos a partir dos quais os sinais de comando são atendidos. Este método, naturalmente, é desperdiçador de hardware. Em qualquer caso, este problema de fazer a interface entre sistemas que não operam com a mesma velocidade é um problema que reaparece regularmente. Na seção seguinte nós trataremos deste assunto.

8.13 CONFIRMAÇÃO DE CONEXÃO (HANDSHAKING)

Um método pelo qual podemos arranjar comunicação entre dois sistemas que não operam sincronicamente e muitas vezes com velocidades não relacionadas e amplamente diferentes é chamado *confirmação de conexão* (*handshaking*). Consideremos o assunto especificamente em relação à transmissão de dados de um sistema de transmissão para um sistema de recepção.

No handshaking, o transmissor torna disponível para o receptor, além dos dados a serem transmitidos, uma linha que transporta um sinal lógico chamado "*dados válidos*" (DAV). O receptor torna disponível para o transmissor uma linha que transporta um sinal lógico chamado "*dados aceitos*" (DAC). O arranjo é ilustrado na Fig. 8.13-1. Presumivelmente, o transmissor transmitirá palavra após palavra ao receptor sobre um bus de dados de n bits. Quando DAV = 1, significa que quaisquer processamento e operações que necessitam ser efetuados para colocar uma nova palavra no bus de dados foram concluídos e a palavra no bus está estabelecida, estável e pronta para ser lida pelo receptor; assim, a palavra é *válida*. Quando DAV = 0, a palavra não é válida. Uma seqüência de níveis lógicos no DAV desde 1 até 0 e de volta a 1 acompanha cada mudança de palavra.

Quando DAC = 0, significa que o receptor está completamente pronto para receber a palavra de dados sobre o bus, mas o fará somente se achar que DAV = 1. Se DAC = 1, o receptor não aceitará os dados; assim, o receptor está, efetivamente, desconectado do bus. Transmissor e receptor comunicam-se sobre estas linhas DAV e DAC. O receptor diz ao transmissor quando a palavra de dados foi aceita, de modo que o transmissor possa colocar a próxima palavra na linha. O transmissor diz ao receptor quando uma nova palavra foi de maneira estável estabelecida no bus de modo que ela possa ser lida.

Suponhamos que partimos durante um intervalo em que DAV = 1 e DAC = 0, isto é, uma palavra está sendo transmitida e no processo de ser aceita. Então devemos admitir que DAV não se torna DAV = 0 até que tenhamos DAC = 1. Além disto, quando, com DAV = 1 e DAC = 0, uma palavra é aceita, esta palavra não deve ser aceita novamente até que DAV mude para DAV = 0 e depois novamente de volta a DAV = 1. De outro modo, o receptor aceitará a mesma

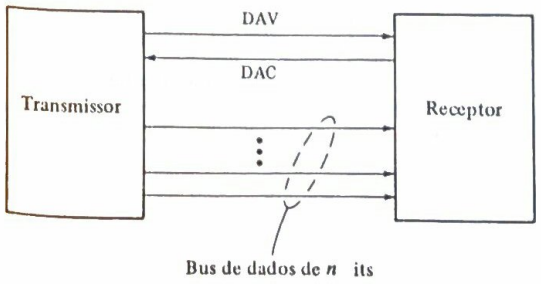


Fig. 8.13-1 A operação de um transmissor e receptor de dados é coordenada pelo uso de linhas DAV e DAC de confirmação de conexão.

Tabela 8.13-1 Seqüência de níveis lógicos DAV e DAC quando uma palavra é transmitida e feita a preparação para a palavra seguinte

Intervalo de tempo	Transferir DAV	Receber DAC	Comentário
1	1 válido	0 aceita	Palavra transferida
2	1 válido	1 não aceita	Certificar-se de que os dados permanecem válidos até depois que o receptor se desconecta
3	0 não-válido	1 não aceita	Palavra sendo mudada
4	0 não-válido	0 aceita	Nenhuma transferência porque DAV = 0
5	1	0	Nova palavra transferida

palavra duas vezes. Com um tal sistema haverá seguramente intervalos em que o transmissor deve *esperar* para que o receptor complete algum processamento ou em que o receptor deve *esperar* pelo transmissor. Na verdade, o ponto preciso de todo o arranjo consiste em assegurar que o transmissor ou receptor *esperará* quando requerido. A seqüência que as linhas DAV e DAC atravessam quando uma palavra é transferida e são feitas preparações para transferência da palavra seguinte é mostrada na Tab. 8.13-1.

Como ilustração do uso das linhas DAV e DAC, consideremos a situação na Fig. 8.13-2, onde temos duas memórias de acesso aleatório de organização idêntica e é nosso propósito transferir o conteúdo da memória A (M_A) para a memória B (M_B). Cada palavra deve ocupar o mesmo endereço em B conforme ocupou quando em A. Permanentemente, ajustamos a entrada E_A de habilitar e a $(R/\bar{W})_A$ em $E_A = (R/\bar{W})_A = 1$ de modo que haja sempre uma palavra extraída de M_A, sendo esta palavra determinada por um contador cuja contagem constitui o endereço da palavra selecionada. Em M_B ajustamos $(R/\bar{W})_B = 0$ de modo que M_B esteja permanentemente no modo de escrita, e a operação de escrita ocorrerá quando o terminal de habilitação for para $E_B = 1$.

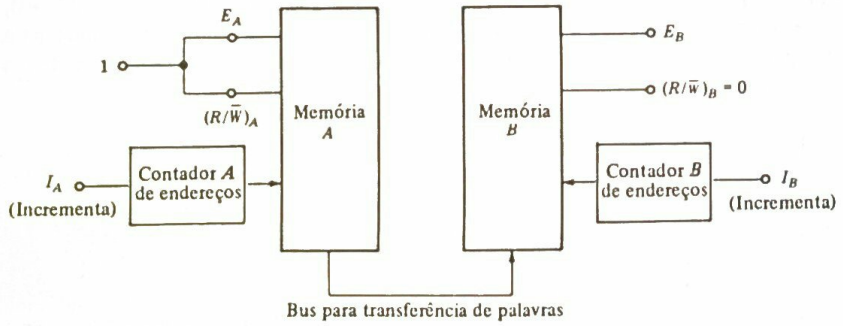


Fig. 8.13-2 Duas RAMs idênticas estão conectadas para efetuar a transferência do conteúdo da memória A para a memória B.

Se o sistema inteiro fosse operar como um simples sistema síncrono, isto é, comandado por uma única forma-de-onda comum de sincronização, o projeto de um controlador para efetuar a transferência de M_A para M_B seria bem elementar. Na verdade, requereríamos apenas um único contador de endereços que forneceria um endereço comum tanto para M_A quanto para M_B. Um controlador de dois estados faria o serviço. Num estado o controlador ajustaria $E_B = 1$ para ler a palavra em M_B. No segundo estado o contador de endereços seria incrementado. O controlador realizaria a seqüência através dos estados escrever, incrementar, escrever, incrementar etc. Se o número de estados do contador for o mesmo que o número de palavras nas memórias, não importará qual seja o endereço inicial no contador. Além disto, se o controlador continuar a funcionar

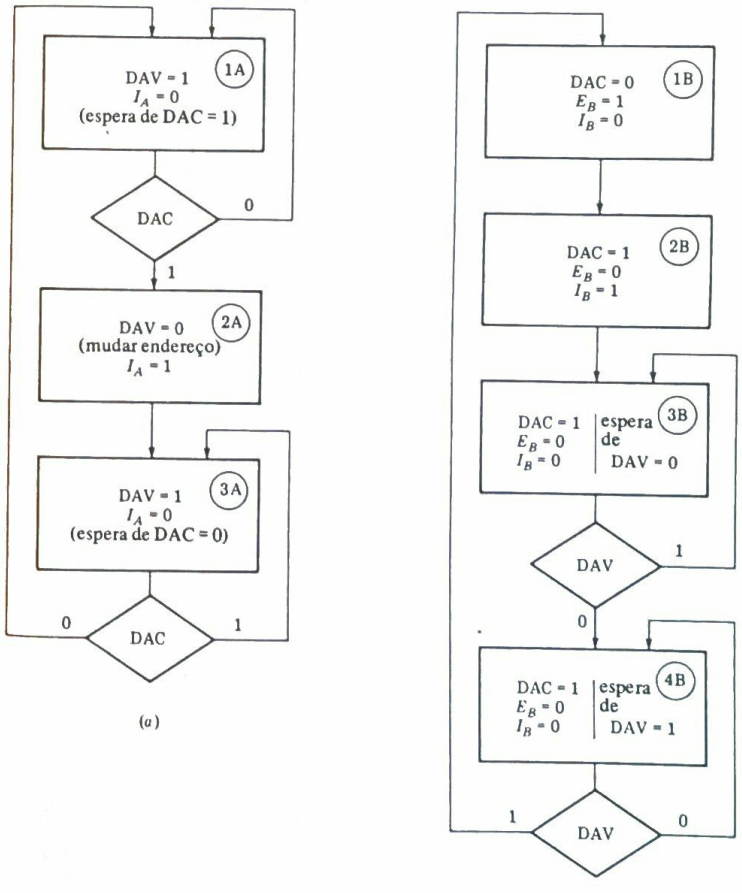


Fig. 8.13-3 O fluxograma de um controlador para operar o sistema da Fig. 8.13-2 e para tornar disponíveis os sinais DAV e DAC.

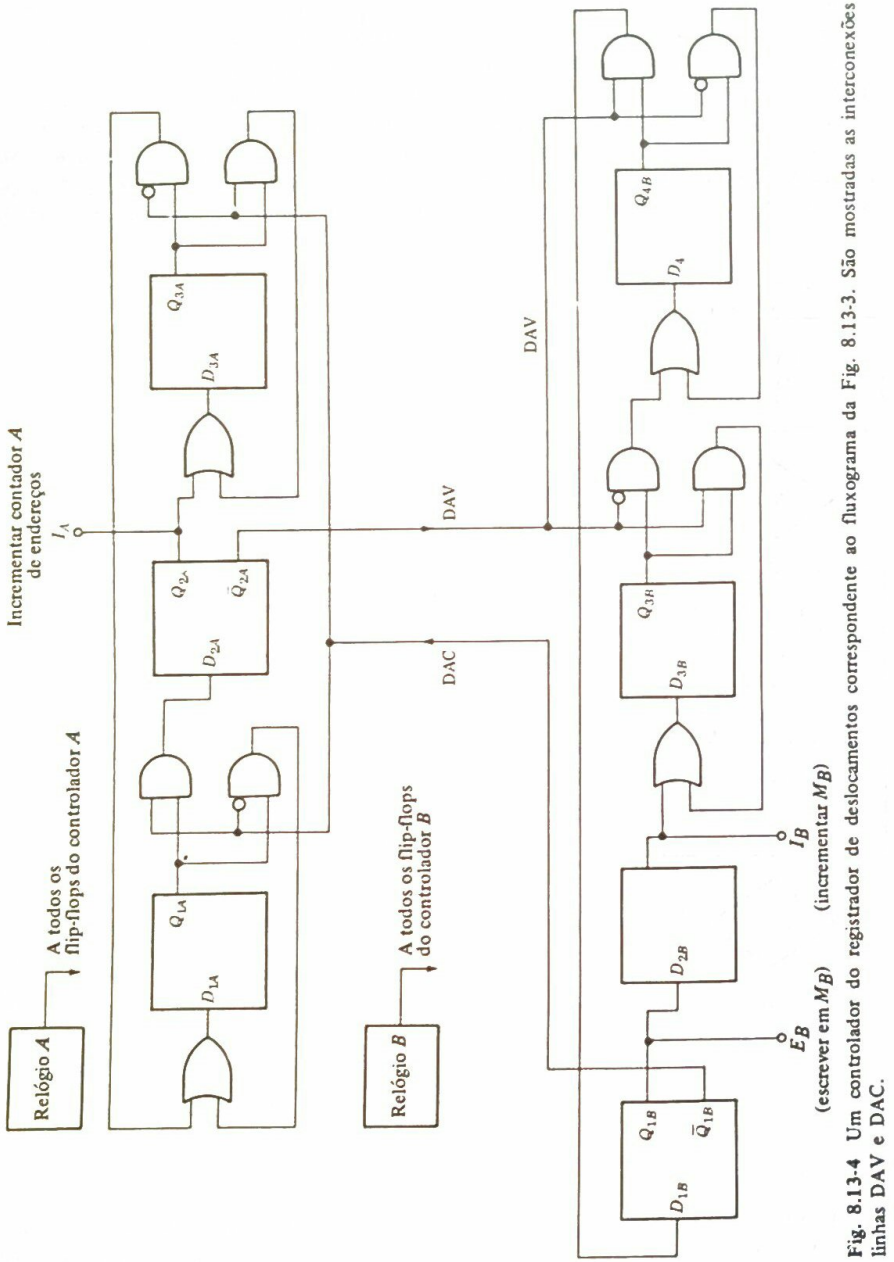


Fig. 8.13-4 Um controlador do registrador de deslocamentos correspondente ao fluxograma da Fig. 8.13-3. São mostradas as interconexões das linhas DAV e DAC.

depois do serviço ser concluído, não haverá prejuízo, uma vez que simplesmente estaremos escrevendo as mesmas palavras nos mesmos lugares.

A seguir suponhamos que as duas porções do sistema A e B devam ser comandadas pelas duas formas-de-onda $C1_A$ e $C1_B$ de sincronização não-relacionadas, separadas e independentes com freqüências f_A e f_B. Pode ser que $f_A > f_B$ ou vice-versa. Pode ser mesmo que f_A e f_B variem com o tempo de maneira não relacionada uma à outra. Os dois sistemas são conectados *somente* pelo bus sobre o qual a palavra é transferida, e interligaremos os dois controladores separados agora requeridos pelas linhas DAC e DAV. Consideremos agora o projeto dos dois controladores.

O fluxograma do controlador A está mostrado na Fig. 8.13-3*a*. Definimos o primeiro estado, $1A$, como sendo um estado em que o contador A de endereços está quiescente em um endereço fixo e a palavra endereçada está sobre o bus. A palavra sobre o bus é a palavra que queremos ler; nenhum de seus bits está no processo de modificação, de modo que a palavra é válida e temos DAV = 1. Este primeiro estado também definimos como sendo um estado que é atingido em conseqüência do fato de que $C1_A$ num instante da transição de gatilho de C1 a linha DAC do controlador B para o controlador A estava DAC = 0. Este estado então é o estado durante o qual uma palavra válida é aplicada a um receptor de aceitação de modo que a palavra será escrita em M_B. A escrita real requer apenas que o controlador coloque E_B em lógica 1 durante este estado.

Desde que DAC permaneça DAC = 0, o transmissor deve guardar a mesma palavra no bus e, correspondentemente, manter DAV = 1. Quando DAC tornar-se DAC = 1, isto é, quando o receptor tenha se desconectado do bus, o controlador do transmissor pode ir para o estado seguinte, $2A$, onde DAV será DAV = 0 e o contador A de endereços estará avançado. Depois do endereço ter sido modificado, o transmissor novamente tem uma palavra válida sobre o bus e, portanto, o controlador vai para um estado $3A$, em que novamente DAV = 1. Neste estado o controlador espera até que DAC novamente se torne DAC = 0, em cujo tempo o controlador retorna a seu estado inicial. Note particularmente que, uma vez que um incremento de endereço tenha sido feito, a próxima mudança de endereço não ocorre até que DAC primeiro se torne DAC = 0 e, depois, DAC = 1. Desta maneira o controlador se certifica de que o receptor aceitou a palavra transmitida antes que uma nova palavra seja colocada no bus.

O fluxograma do controlador B é mostrado na Fig. 8.13-3*b*. O estado $1B$ corresponde ao estado $1A$. Quando os controladores A e B estiverem nestes estados, os contadores de endereço não estão sendo avançados, os dados são válidos, E_B está em $E_B = 1$ e uma transferência de palavra está em processo. Note que o controlador A não pode sair do estado $1A$ até que o controlador B tenha saído do estado $1B$. Note, ainda, que o controlador não retornará ao estado $1B$ até que primeiro DAV tenha ido para DAV = 0 e depois retornado a DAV = 1. Desta maneira está assegurado que a próxima palavra transferida será uma palavra nova.

Um controlador correspondente ao fluxograma da Fig. 8.13-3 é mostrado na Fig. 8.13-4.

CAPÍTULO 9

COMPUTADORES

O sistema da Fig. 8.8-1 pode, com certa benevolência, ser chamado computador. Seus méritos essenciais são sua simplicidade e, além disto, ele apresenta alguns aspectos característicos importantes que são achados em computadores mais sofisticados. Por outro lado, nosso "computador" tem a séria deficiência de não ser capaz de fazer muito. Neste capítulo consideraremos algumas elaborações de projetos de computador.

9.1 UMA ARQUITETURA APRIMORADA

Conforme temos observado, o projeto de um computador ou de outro processador digital geralmente começa pela consideração da arquitetura. Decisões têm que ser tomadas a respeito do tipo de blocos componentes a serem usados, isto é, números de registradores, tipos de registradores, operações que os registradores e as ULAs devem ser capazes de executar e, mais importante, que registradores e componentes devem ser interconectados de modo que os dados possam ser transferidos. Não há nenhum procedimento de projeto pelo qual uma "melhor" arquitetura possa ser selecionada. Estruturas arquitetônicas são geralmente atingidas na base da experiência do projetista e muitas vezes simplesmente "aprimorando" um projeto existente. Um aprimoramento de sistema da Fig. 8.8-1 é mostrado na Fig. 9.1-1. Em geral, como também podemos esperar, aprimoramentos que permitam que um sistema faça mais conduzem a maior complexidade.

Em nossa nova máquina, a memória, que é naturalmente essencial num computador de programa armazenado, é especificada como uma memória que armazenará palavras de doze bits. Realmente, não faremos nenhum uso preciso do comprimento da palavra, e o comprimento é especificado simplesmente de modo que possamos ser mais específicos na discussão que se segue.

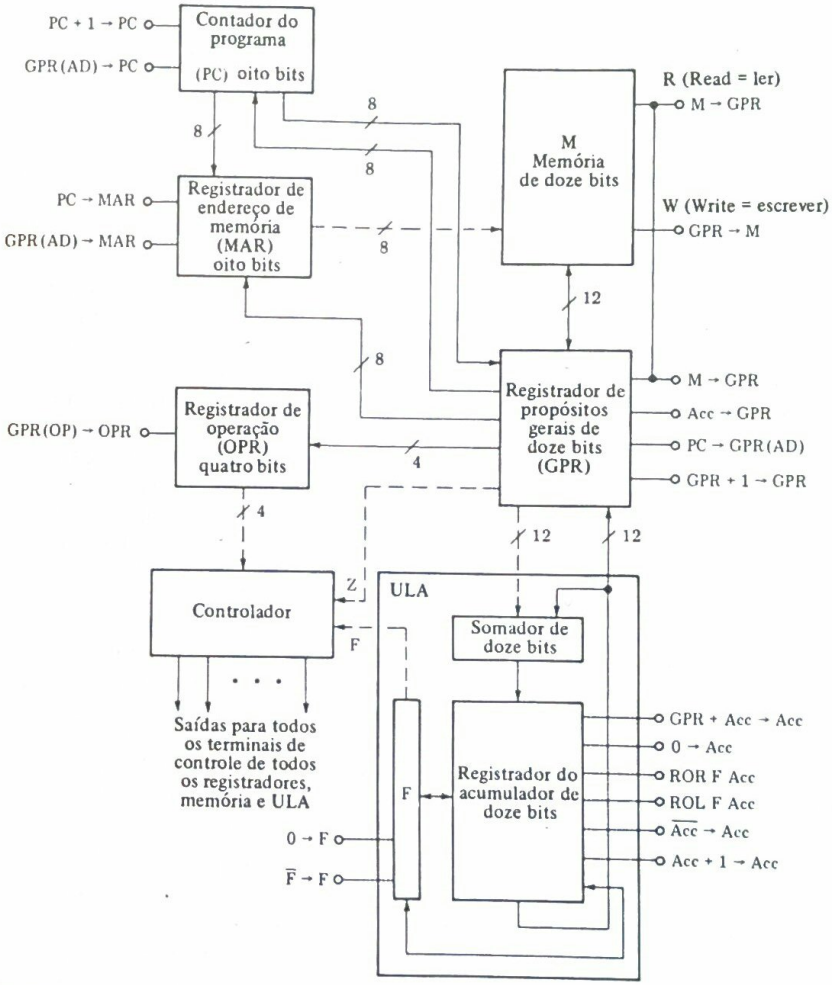


Fig. 9.1-1 Arquitetura de um computador.

Nossa máquina terá um repertório de dezesseis instruções, para as quais necessitaremos de quatro bits de instrução. Se tivéssemos permanecido com palavras de oito bits, teríamos sido reduzidos a um endereço de quatro bits, permitindo apenas uma memória de dezesseis palavras.

Em ocasiões anteriores indicamos que as memórias de acesso aleatório são geralmente equipadas com dois terminais de controle e, possivelmente, com outros. Um destes é um terminal (E) de HABILITA (*ENABLE*), e o outro é um terminal de leitura/$\overline{\text{escrita}}$ (R/$\overline{\text{W}}$). A memória permitirá que uma palavra seja escrita (gravada) nela ou lida dela somente se E = 1. Para escrever, colocamos

$R/\overline{W} = 0$ e, para ler, colocamos $R/\overline{W} = 1$. Em nossa presente aplicação será mais conveniente ter a memória equipada com dois terminais de controle marcados simplesmente escrever (gravar) (W) e ler (R). Quando W = 1, a palavra presente nos terminais de dados da memória é escrita na memória, e quando R = 1, a palavra na memória é lida até o fim nos terminais de dados. Quando W = R = 0, não ocorrerão nem leitura nem escrita, e W = R = 1 não é permitida. Podemos prontamente acrescentar os circuitos-porta entre os terminais W e R e os terminais E e $R/\overline{W}$ (Prob. 9.1-1). As portas são supostas estarem incorporadas à memória da Fig. 9.1-1 e não estão mostradas explicitamente.

Conforme anteriormente, temos um contador (PC) de programa e um registrador (MAR) de endereços da memória. Não temos um registrador de instruções, mas agora temos um registrador de propósitos gerais de doze bits (GPR) e um registrador (OPR) de operações que tem quatro bits de comprimento e reterá a parte de operação da instrução. Quando uma instrução (que consiste em uma parte de operação e em uma parte de endereço do operando) for lida da memória, a instrução será primeiro lida no GPR. A parte de operação da instrução será depois transferida para o OPR. A parte de endereços do operando da instrução será transferida para o registrador (MAR) de endereços da memória. Quando o GPR está apenas retendo o endereço de oito bits, quatro bits da capacidade do GPR não estarão em uso. Por outro lado, a capacidade plena do GPR será usada quando o GPR estiver armazenando um operando em vez do endereço de um operando. O presente arranjo oferece-nos a conveniência de sermos capazes de separar a parte de operação de uma instrução da parte de endereços do operando. Então seremos capazes de efetuar algumas manipulações úteis da parte de endereço sem envolver a parte de operação.

Como antes, incorporamos ao contador do programa a capacidade de ser incrementado. Mas agora providenciamos ligação direta entre o contador do programa e o registrador de propósitos gerais, de modo que o conteúdo de cada registrador possa ser transferido para o outro. Mais ainda, fornecemos realmente dois caminhos separados, um de PC para GPR e um de GPR para PC, de modo que o conteúdo dos registradores possa ser trocado durante um único ciclo de relógio. Naturalmente, para que ocorra esta troca, os flip-flops destes dois registradores devem ser do tipo a partir do qual podemos ler e no qual podemos escrever no mesmo ciclo de relógio. Conforme assinalamos, flip-flops mestre-escravos bem como de inúmeros outros tipos são adequados.

O registrador de propósitos gerais é capaz de quatro microoperações:

1. Pode transferir para si próprio a palavra de memória endereçada. Para efetuar esta transferência, elevaremos à lógica 1 o terminal de controle GPR marcado M → GPR e, simultaneamente, colocaremos em lógica 1 o terminal READ da memória. O terminal de controle READ da memória é usado sem nenhum outro propósito senão o de permitir que uma palavra da memória seja lida no GPR; daí, conforme indicado, estamos livres para conectar o terminal READ ao terminal GPR marcado M → GPR.
2. Pode transferir para si próprio o conteúdo do acumulador.
3. Pode transferir para si próprio os oito bits no contador do programa, colocando os bits, naturalmente, nas oito posições do registrador reservadas para o endereço.
4. Pode ser incrementado.

Nossa nova unidade lógica e aritmética é projetada diferentemente da ULA da Fig. 8.9-1 e, comparada com aquele projeto inicial, tem um repertório expandido. A nova ULA tem apenas um registrador de doze bits mais um registrador F de um bit extra, isto é, um flip-flop. A

ULA, naturalmente, tem uma quantidade substancial de lógica, isto é, conjunto de circuitos lógicos combinatórios, que não é mostrado. A ULA pode efetuar oito microoperações:

1. Pode adicionar a seu registro presente o número de doze bits sobre a linha de doze bits que vem do registrador da memória intermediária.
2. Pode limpar o acumulador.
3. Pode limpar F.
4. Pode complementar F.
5. Pode complementar o acumulador.
6. Pode incrementar o acumulador.
7. Pode girar para a direita através de F (ROR F, Acc).
8. Pode girar para a esquerda através de F (ROL F, Acc).

Na operação ROR F, Acc todos os bits no acumulador são deslocados de uma posição para a direita. O bit no flip-flop F é deslocado para a posição do acumulador mais à esquerda, e o bit originalmente na posição do acumulador mais à direita é deslocado para o flip-flop F. Daí, se visualizarmos o acumulador e o flip-flop F como um contador em anel, a operação ROR F, Acc gira os bits de uma posição para a direita. Correspondentemente, ROL F, Acc gira os bits de uma posição para a esquerda. As duas operações estão representadas em diagrama na Fig. 9.1-2.

Para simplificar, admitimos aqui que cada trajeto de transferência é reservado para uma única função de transferência; assim, nenhum dos buses (barramentos) tem que ser compartilhado. Então, os doze bits do registrador de propósitos gerais estão sempre disponíveis para a ULA aceitar ou não, como quiser; e os doze bits do registrador do acumulador estão sempre disponíveis para o registrador de propósitos gerais. Todos os trajetos sobre os quais a transferência de palavra ocorrerá somente em resposta à ativação de alguma entrada de controle

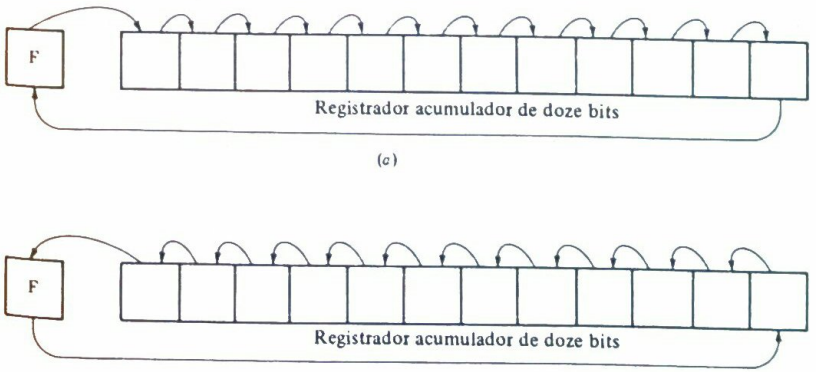


Fig. 9.1-2 (*a*) Operação ROR F, Acc. (*b*) Operação ROL F, Acc.

estão desenhados como linhas cheias. As linhas de trajeto tracejadas indicam trajetos de transferência que não requerem nenhuma destas ativações.

O controlador fornecerá sinais de comando a todos os terminais de controle dos diversos registradores e para a memória; assim, em ciclos de relógio apropriados, como denominados pela operação no registrador de operação, um ou mais terminais de controle serão elevados ao nível lógico 1. Há ocasiões, no entanto, em que se exige que os comandos emitidos pelo controlador dependam mais da parte da operação da instrução. Em vez disto, no curso da colocação em seqüência através das microoperações requeridas na execução de uma única instrução, a próxima microoperação na seqüência pode depender dos resultados produzidos pelas microoperações precedentes. Com isto em mente, fizemos provisão de duas entradas adicionais para o controlador. Uma delas vem do GPR e é designada Z. Esta linha marcada Z estará em lógica 1 quando (e

Tabela 9.1-1 Componentes e Operações de Controle do Sistema

Componente	Simbolismo de controle	Explicação
Memória	1. GPR → M	Escreve conteúdo de GPR na localização da memória endereçada
Contador do programa (PC)	2. PC + 1 → PC	Incrementa PC
	3. GPR(AD) → PC	Transfere bits de endereço do registrador de propósitos gerais para PC
Registrador de endereço da memória (MAR)	4. PC → MAR	Transfere do PC para MAR
	5. GPR(AD) → MAR	Transfere bits de endereço do registrador de propósitos gerais para MAR
Registrador de operação (OPR)	6. GPR(OP) → OPR	Transfere bits de operação do registrador de propósitos gerais para OPR
Registrador de propósitos gerais (PR)	7. M → GPR	Transfere palavra endereçada para GPR
	8. Acc → GPR	Transfere conteudo do Acc para GPR
	9. PC → GPR(AD)	Transfere conteúdo do contador de programa para a parte de endereços do MAR
	10. GPR + 1 → GPR	Incrementa GPR
Unidade lógica e aritmética (ULA)	11. GPR + Acc → Acc	Adiciona número no GPR para número no Acc e deixa soma no Acc
	12. 0 → Acc	Limpa Acc
	13. ROR F, Acc	Gira Acc para a direita através de F
	14. ROL F, Acc	Gira Acc para a esquerda através de F
	15. 0 → F	Reset flip-flop F
	16. $\overline{F} \rightarrow F$	Complementa flip-flop F
	17. $\widehat{Acc} \rightarrow Acc$	Complementa Acc
	18. Acc + 1 → Acc	Incrementa Acc

somente quando) todos os doze bits no GPR estiverem em zero. A outra entrada F vem do flip-flop flag (bandeira) associado ao registrador do acumulador. Indicamos estas ligações para servir de argumento quando dissermos mais tarde que o controlador segue uma seqüência ou outra, dependendo do nível lógico Z ou do fato de F estar ajustado (set) ou reajustado (reset).

A Tab. 9.1-1 lista todos os componentes de nosso sistema, exceto o controlador, e todos os símbolos usados para designar os terminais de controle; por conveniência, ela estabelece novamente, para cada terminal de controle, a microoperação que é executada quando o terminal for ativado.

9.2 INSTRUÇÕES

Como antes, quando for dada partida ao controlador, ele funcionará em todas as direções, buscando uma instrução, executando-a, buscando uma outra instrução etc. Suponhamos que ajustamos manualmente o contador de programa (PC) no endereço da primeira instrução. A seqüência de microoperações seguida no presente caso para *buscar* (*fetch*) uma instrução é como se segue:

Ciclo de relógio	Microoperação	Explicação
1	PC → MAR	Transfere local da instrução do contador de programa para o registrador de endereço da memória
2	M → GPR PC + 1 → PC	Transfere palavra endereçada para o registrador de propósitos gerais; incrementa contador de programa
3	GPR(OP) → OPR	Transfere parte da operação de instrução para o registrador de operações

Quando a seqüência das microoperações de busca estiver completa, a parte da operação da instrução foi transferida para o registrador de operação. Se houver um operando envolvido na instrução, o endereço deste operando é deixado na parte de endereço do registrador de propósitos gerais GPR (AD). Naturalmente, nem todas as instruções envolvem um operando; em tais casos, o conteúdo do GPR (AD) no fim da seqüência de busca não é de importância particular.

Definiremos agora inúmeras instruções para nosso computador. Naturalmente, uma instrução é viável somente se puder ser efetuada elevando-se ao nível ativo, em alguma seqüência, os terminais de comando expostos na Fig. 9.1-1. Se mantivermos nossas instruções simples e poucas, manteremos o controlador simples. Em tais casos, programas de instrução escritos para obter um certo propósito computacional útil necessariamente tornar-se-ão mais longos. Se introduzirmos instruções complicadas e numerosas, a escrita de programas torna-se mais simples, mas o controlador torna-se correspondentemente mais elaborado. Favoreceremos a alternativa de instruções poucas e simples, sendo as instruções, apesar disto, representativas das instruções usadas em componentes comerciais mais sofisticados.

Introduzimos primeiro inúmeras instruções que são bastante elementares no sentido de que cada uma requer um único terminal de comando para ser elevada ao nível ativo. Daí, cada uma

pode ser executada num único ciclo de relógio e consiste em uma única microoperação. Nenhuma destas envolve uma referência à memória, quer de leitura, quer de escrita e, por conseguinte, nenhuma envolve um endereço. Temos:

Instrução (microoperação)	Explicação	Mnemônico
$0 \rightarrow Acc$	Limpar acumulador	CRA
$\overline{Acc} \rightarrow Acc$	Complementar acumulador	CTA
$Acc + 1 \rightarrow Acc$	Incrementar acumulador	ITA
$0 \rightarrow F$	Limpar flip-flop F	CRF
$\overline{F} \rightarrow F$	Complementar flip-flop F	CTF
$PC + 1 \rightarrow PC$	Saltar (skip) para a instrução seguinte se F for zero	SFZ
Girar à direita	Girar à direita através de F e Acc	ROR
Girar à esquerda	Girar à esquerda através de F e Acc	ROL

A instrução $PC + 1 \rightarrow PC$ merece algum comentário. Suponhamos que esta instrução esteja localizada no endereço K da memória. Durante a seqüência de *busca* de microoperações que lê esta instrução fora da memória, o contador PC do programa é incrementado de modo que PC esteja em $K + 1$. No entanto, a execução desta instrução, como a de qualquer instrução, causa um incremento adicional, de modo que depois da execução temos $PC = K + 2$ e a instrução no endereço $K + 1$ da memória é saltada. Notamos ainda que este salto (skip) é condicional no valor de F. Se $F = 0$, a microoperação de salto (skip) $PC + 1 \rightarrow PC$ é realmente executada. Por outro lado, se no instante em que esta operação de salto (skip) for provocada, revela-se que $F = 1$, então a instrução não executa nenhuma operação. Na preparação da arquitetura da Fig. 9.1-1 pretendíamos que esta operação fosse condicional em F. É por esta razão que F é uma das entradas para o controlador. Veremos como esta instrução de salto (skip) em cooperação com uma instrução JUMP (apresentada abaixo) possibilitará que nossa máquina selecione um curso de ação ou outro, dependendo do valor de F.

Consideremos agora instruções adicionais que requerem uma *seqüência* de microoperações para sua execução.

ADD, address (ADICIONAR, endereço)

Adicionar ao presente conteúdo do acumulador o número (operando) localizado no endereço de memória que está especificado como parte da instrução.

A parte de busca do ciclo transferiu a parte de operação desta instrução para o registrador de operação (OPR), e a parte de endereço do operando da instrução é deixada na parte de endereço do registrador de propósitos gerais GPR (AD). A instrução é agora executada pela seguinte seqüência de microoperações:

Ciclo de relógio	Microoperação	Explicação
1	$GPR(AD) \rightarrow MAR$	Transfere endereço do operando de GPR (AD) para MAR
2	$M \rightarrow GPR$	Lê da memória a palavra no local cujo endereço está em MAR
3	$GPR + Acc \rightarrow Acc$	Adiciona conteúdo de GPR ao conteúdo de Acc, deixando a soma em Acc

Freqüentemente, em vez de adicionar ao acumulador um operando da memória, simplesmente queremos transferir um tal operando para o acumulador. Uma instrução desta espécie seria descrita como "carregar o acumulador" e pode igualmente ter o mnemônico LDA. No entanto, em benefício da economia, não introduziremos uma tal instrução. Em vez disto, quando necessitarmos de uma tal operação de "carregar", efetuá-la-emos usando, em seqüência, as duas instruções CRA (*clear the accumulator* = limpar o acumulador) e ADD (*add to accumulator* = adicionar ao acumulador).

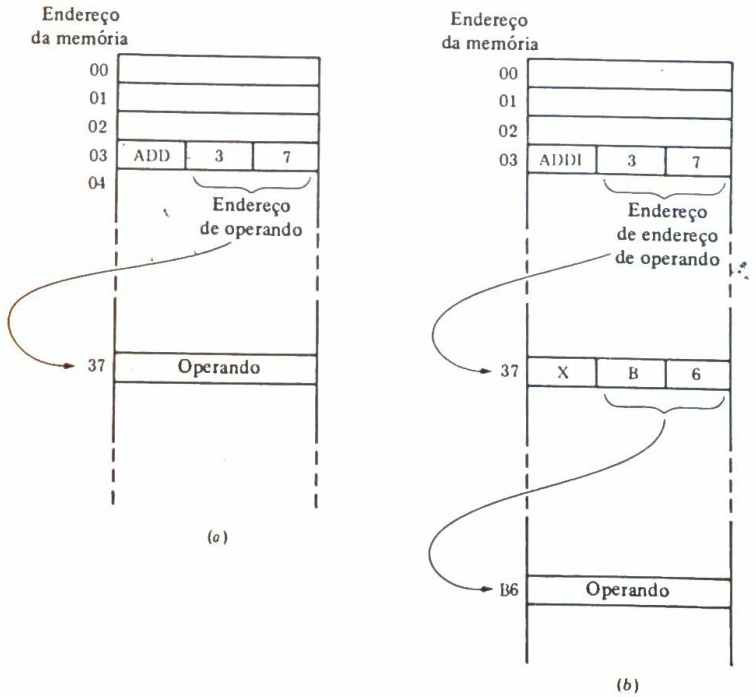


Fig. 9.2-1 A diferença entre (*a*) uma instrução de endereço direto e (*b*) uma instrução indireta.

A instrução ADD fornece o endereço do operando que deve ser adicionado ao acumulador. Conforme veremos, é muitas vezes de grande conveniência ter uma instrução que, em vez de fornecer o endereço do operando, retém o endereço de um local da memória que alternativamente retém o endereço do operando. Quando um endereço de operando é fornecido desta maneira circundante ou indireta, dizemos que o endereço é *indireto*. Para indicar que uma instrução envolve um endereço indireto, acrescentamos o símbolo I ao mnemônico da instrução. A diferença entre ADD (adicionar ao acumulador) e ADDI (adicionar ao acumulador, indireto) é tornada clara na Fig. 9.2-1. Aqui, para simplificar, escrevemos as palavras binárias em notação hexadecimal (ver Seção 1.13). Na Fig. 9.2-1*a* consideramos que o local 03 da memória contém a instrução ADD 37, que significa que o acumulador deve ter adicionado a ele o operando localizado no local 37 da memória. (A parte da operação da instrução usa quatro bits, como faz cada um dos números 3 = 0011 e 7 = 0111.) O operando é agora achado diretamente no local 37 da memória. Na Fig. 9.2-1*b*, onde se pretende o endereçamento indireto, achamos no local 37 da memória não o operando, mas o endereço do operando. (Uma vez que o local 37 contém apenas um endereço, somente oito dos doze bits são relevantes e os primeiros quatro bits são redundantes ou não-especificados.) No local 37 achamos que o operando está no local B6 da memória, e este é o operando que é adicionado ao acumulador. A seqüência de microoperações solicitada pela instrução ADDI é:

Ciclo de relógio	Microoperação	Explicação
1	GPR(AD) → MAR	Transfere endereço de GPR para MAR
2	M → GPR	Transfere conteúdo da memória no local endereçado para GPR (GPR terá então endereço do operando)
3	GPR(AD) → MAR	Transfere endereço do operando para MAR
4	M → GPR	Transfere operando endereçado para GPR
5	GPR + Acc → Acc	Adiciona conteúdo de GPR ao Acc

Se necessitarmos efetuar um carregamento indireto do acumulador, podemos fazê-lo pela seqüência de instruções CRA (limpar acumulador) seguida pela ADDI.

A seqüência de instruções CRA e ADD provoca um operando da memória: isto é, elas *lêem* a memória e transferem a palavra assim lida para dentro do acumulador. A instrução inversa *escreve* na memória o conteúdo do acumulador. O mnemônico desta instrução é STA (*store the accumulator* = armazenar o acumulador). Para esta instrução STA temos, então, a seguinte definição:

STA, address (STA, endereço)

Armazenar o conteúdo do acumulador em um endereço de memória especificado.

A seqüência de microoperações é:

Ciclo de relógio	Microoperação	Explicação
1	GPR(AD) → MAR	Transferir endereço de GPR para MAR
2	Acc → GPR	Transferir conteúdo de Acc para GPR
3	GPR → M	Escrever conteúdo de GPR na memória no endereço retido em MAR

Observamos que normalmente as instruções são listadas na memória na ordem em que devem ser executadas, e temos acesso a uma instrução após outra incrementando o contador do programa. Contudo, conforme veremos, às vezes é útil não seguir um tal procedimento, mas saltar (jump), para uma instrução que não esteja em ordem seqüencial. As diversas instruções seguintes proporcionam exatamente uma tal aptidão.

JMP, address

Saltar (jump) para a instrução localizada no local da memória especificado na instrução.

Para esclarecer a intenção de sua instrução, suponhamos que, quando o controlador busca e executa instrução após instrução, ele busca a instrução JMP 29. Então, a única execução que o controlador efetua consiste em transferir o endereço, isto é, 29, não para o registrador de endereços da memória (MAR), mas para o contador do programa (PC). A única microoperação solicitada é:

Ciclo de relógio	Microoperação	Explicação
1	GPR(AD) → PC	Transfere endereço da próxima instrução do GPR para PC

Daí, a *próxima* instrução que será buscada é a instrução no local 29 da memória; e, desde que não sejam encontradas outras instruções de salto (jump) a máquina prosseguirá depois em ordem para a instrução no local 30, 31 etc. da memória.

JMPI, address [salto (jump), endereço]

Saltar (jump) para a instrução cujo endereço está localizado no endereço de memória especificado na instrução.

Esta instrução é a forma *indireta* da instrução JMP dada acima. A parte de endereço da instrução JMPI é o endereço do local da memória que retém ou guarda o *endereço da próxima instrução*. A seqüência da microoperação é:

Ciclo de relógio	Microoperação	Explicação
1	GPR(AD) → MAR	Transfere endereço para MAR
2	M → GPR(AD)	Endereço lido da proxima instrução da memória
3	GPR(AD) → PC	Transfere endereço da próxima instrução do GPR para PC

Antes da apresentação da próxima instrução convém darmos alguma explicação. Suponhamos que escrevemos um programa (uma lista de instruções para o computador executar a fim de conseguir algum propósito) para nosso computador. Suponhamos, ainda, que achamos que muitas vezes durante a execução do programa um conjunto particular de instruções reapareça regularmente. Por exemplo, o programa pode muitas vezes requerer a multiplicação de dois números. Conforme veremos, nossa máquina não será capaz de executar uma tal instrução de multiplicação diretamente. Em vez disto, para efetuar uma multiplicação, teremos que providenciar para que o computador complete uma lista de instruções mais simples, isto é, uma *rotina* de multiplicação, a que ele possa responder. Se quisermos, podemos simplesmente incluir esta rotina em nosso programa cada vez que for requerida uma multiplicação. Há, no entanto, um procedimento alternativo que tem méritos óbvios. Em algum lugar na memória, podemos armazenar esta rotina apenas uma vez. Depois, sempre que a multiplicação for solicitada, podemos saltar (jump) para a rotina armazenada, e, quando a rotina da multiplicação tiver sido concluída, podemos retornar ao local em nosso programa a partir do qual saltamos (jump).

Este processo de abandonar a seqüência ordenada de instruções para ir para uma rotina especial é denominado *chamada de uma sub-rotina* ou *salto (jump) para uma sub-rotina*. O fim de uma chamada de sub-rotina deve sempre ser um retorno ao local no programa a partir do qual o salto (jump) foi feito.

Antes de saltar (jump) para uma sub-rotina, devemos *selecionar um local na memória* onde nos propomos a armazenar o endereço a ser rearmazenado para o contador de programa quando a sub-rotina tiver sido comcluída. O endereço deste *local selecionado* deve ser incluído na instrução que chama a sub-rotina. Devemos também incluir na instrução o endereço na memória onde começa a sub-rotina. Por conseguinte, esta instrução CSR (*call subroutine* = chamar sub-rotina) realmente envolve dois endereços. No entanto, desde que nossa instrução CSR, como todas as outras instruções, pode acomodar somente um endereço, deve haver algum arranjo ou entendimento anterior.

O meio pelo qual nos propomos a chamar uma sub-rotina e depois retornar ao programa principal interrompido é mostrado na Fig. 9.2-2. A instrução CSR está na memória no endereço CALL. A sub-rotina para a qual estamos para saltar (jump) é armazenada em r locais sucessivos $K + 1, K + 2, \ldots, K + r$. Como com o programa principal, admitimos que as instruções na sub-rotina estejam armazenadas em sucessivos locais da memória na ordem em que elas devem ser executadas. Tomamos a deliberação de arranjar para que os locais K e $K + r + 1$ da memória estejam disponíveis para nossos propósitos. Agora, para chamar a sub-rotina e depois retornar dela, escrevemos em nosso programa, em locais apropriados, duas instruções. A primeira instrução é a instrução CSR presentemente sob discussão e tendo a definição:

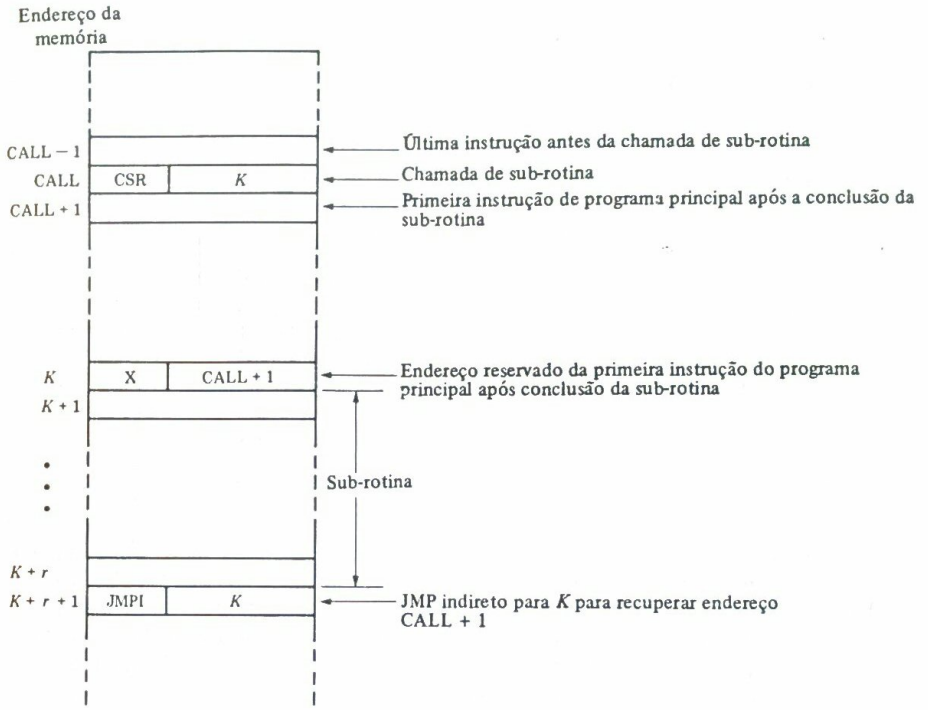


Fig. 9.2-2 As manipulações envolvidas numa chamada de sub-rotina.

CSR, address (CSR, endereço)

Chamar a sub-rotina. Armazenar o endereço de retorno ao programa principal (isto é, o endereço CALL + 1) no local da memória especificado nesta instrução (isto é, endereço = = K) e buscar a próxima a partir do local $K + 1$ da memória.

A segunda instrução é escrita no local $K + r + 1$ da memória e é a instrução JMPI K de salto (jump) indireto que retorna o programa para a instrução em CALL + 1. A seqüência de microoperações requerida para executar a instrução CSR está detalhada na página seguinte.

Uma explicação é novamente requerida antes da descrição da próxima instrução. Há ocasiões em que uma seqüência de instruções deve ser seguida repetidamente um grande número de vezes. Uma tal seqüência repetitiva é efetuada pela execução de um *laço* (loop) *de programa*. A idéia é que, uma vez que a seqüência tenha sido concluída, a próxima instrução encontrada é uma instrução de salto (jump) que faz o programa retornar ao começo da seqüência. A instrução ISZ é usada para terminar este processo de reversão quando ele tiver sido repetido no número

Ciclo de relógio	Microoperação	Explicação
1	GPR(AD) → MAR	Transferir para MAR o endereço onde o endereço de retorno deve ser armazenado; este endereço de armazenamento é *K*
2	GPR(AD) → PC PC → GPR(AD)	Trocar os conteúdos de PC e GPR(AD) [após a troca, PC retém o endereço *K* e GPR(AD) retém CALL + 1, uma vez que PC foi incrementado de CALL para CALL + 1 durante o ciclo de busca]
3	GPR(AD) → M	Transferir GPR(AD) para a memória [o resultado é que endereço CALL + 1 será escrito no local *K* da memória]
4	PC + 1 → PC	Incrementar PC [PC guardará então endereço *K* + 1; próxima instrução será então tirada do local *K* + 1, que é a primeira instrução de sub-rotina

requerido de vezes. Suponhamos que a reversão (ou laço) necessite ser feita *n* vezes. Então o número negativo $-n$ é armazenado em algum local apropriado na memória. A cada laço (ou reversão) o número $-n$ é lido da memória, incrementado e retornado à memória. Quando a reversão tiver sido efetuada *n* vezes, o número terá sido incrementado a zero. Neste ponto a "instrução seguinte", que é a instrução para saltar (jump) de volta ao começo do laço, será saltada (skip) e o programa avançará para fora do laço. A definição precisa é:

ISZ, address (ISZ, endereço)

Incrementar e saltar se zero. Ler o número no local da memória especificado, incrementá-lo e retorná-lo a seu local original. Se, depois da incrementação, o número for zero, saltar a próxima instrução.

As microoperações requeridas para esta instrução ISZ são:

Ciclo de relógio	Microoperações	Explicação
1	GPR(AD) → MAR	Transferir para MAR o local do número a ser incrementado
2	M → GPR	Ler número da memória
3	GPR + 1 → GPR	Incrementar número
4	GPR → M	Retornar número à memória
5	PC + 1 → PC (se GPR = 0)	Saltar próxima instrução se GPR = 0

Note que, para executar esta instrução, o controlador necessita saber quando o registro em GPR é zero. Mas, conforme será lembrado, e como está indicado na Fig. 9.1-1, tal informação é feita disponível ao longo da linha marcada Z (zero). Note ainda que o contador do programa está incrementado durante o ciclo de busca. Daí, se ele for incrementado novamente, como no ciclo 5 de relógio, PC terá sido incrementado duas vezes e a instrução de salto (jump) que retorna o programa ao começo do laço será saltada (skip).

9.3 RESUMO DAS INSTRUÇÕES

Para permitir uma visão global do computador que postulamos, resumimos aqui todo o repertório de instruções:

1. CRA limpar acumulador
2. CTA complementar acumulador
3. ITA incrementar acumulador
4. CRF limpar flip-flop F
5. CTF complementar flip-flop F
6. SFZ saltar (skip) próxima instrução se F = 0
7. ROR girar à direita
8. ROL girar à esquerda
9. ADD adicionar ao acumulador
10. ADDI adicionar ao acumulador, indireto
11. STA armazenar na memória do acumulador
12. JMP saltar (jump)
13. JMPI saltar (jump) indireto
14. CSR chamar sub-rotina
15. ISZ incrementar e saltar (skip) se Z = 0
16. HLT Parar

Notar que, além das instruções que descrevemos, acrescentamos a instrução essencial HALT (PARAR).

As seções seguintes darão alguns exemplos expondo como nosso computador pode ser programado para executar funções úteis.

9.4 ADIÇÃO E SUBTRAÇÃO

Na Fig. 9.4-1 fizemos entradas na memória conforme requerido para preparar nosso computador para calcular a soma de três números e depois armazenar a soma na memória. (Todos os números são expressos em código hexadecimal.) Os primeiros seis locais, 00 a 05 da memória, guardam

Local da memória				Explicação
00	CRA	x	x	Limpar Acc
01	ADD	0	6	Adicionar conteúdo de 06 no Acc
02	ADD	0	7	Adicionar conteúdo de 07 no Acc
03	ADD	0	8	Adicionar conteúdo de 08 no Acc
04	STA	0	9	Armazenar conteúdo de Acc no local 09
05	HLT	x	x	Parar
06	0	1	7	Operando em 06
07	0	0	B	Operando em 07
08	0	1	C	Operando em 08
09				03B será armazenado aqui no fim do programa

Fig. 9.4-1 O conteúdo da memória para um programa achar a soma de três números.

instruções. Os quatro bits mais à esquerda de uma instrução definem a operação. Uma vez que não instituímos um código para as operações, deixamos as operações escritas em forma mnemônica. As quatro primeiras instruções têm endereços. A instrução HLT não tem endereço, de modo que a parte do endereço desta instrução tem entradas irrelevantes. Os operandos que devem ser adicionados (017 + 00B + 01C = 03B) são armazenados em locais 06, 07 e 08. A soma deve ser armazenada no local 09. A primeira instrução limpa e carrega o acumulador com o conteúdo do local 06. A segunda adiciona o conteúdo do local 07 etc. Armazenamos o resultado no local 09. Suponhamos, no entanto, que, uma vez que a computação tenha sido concluída, não mais necessitamos do programa nem dos operandos. Neste caso, poderíamos ter igualmente armazenado o resultado em qualquer um dos locais 00 a 08.

No programa simples da Fig. 9.4-1 fomos capazes de ver no início que necessitaríamos de cinco locais, 00 a 05, para escrever a seqüência de instruções. Por conseguinte, fomos capazes imediatamente de localizar o primeiro operando no local 06 e escrever deste modo as duas primeiras instruções como CRA e ADD 06. No caso mais geral será muito mais conveniente não ter que especificar um local da memória numérico preciso para cada operando até depois do programa ter sido escrito completamente. É mais conveniente especificar endereços de operandos em forma simbólica, sendo os símbolos no final substituídos por números quando o programa estiver completo. Um tal procedimento também evitará ter-se que mudar muitos números sempre que uma etapa de programa ou operando for acrescentada ou eliminada.

O programa da Fig. 9.4-1 está duplicado na Tab. 9.4-1 com os endereços em forma simbólica. Uma vez que não há necessidade de fazer referência no programa aos locais das primeiras cinco entradas no programa, não foram atribuídos endereços simbólicos a eles. Quando o programa for apresentado na forma da Tab. 9.4-1, finalmente alguém (ou alguma coisa) terá que tomar inúmeras decisões. Portanto, em que local da memória deve começar o programa? Na Fig. 9.4-1 arbitrariamente iniciamos o programa no local 00, mas é claro que pode ser de nossa conveniência começar em outro lugar. Uma vez que o local inicial do programa tenha sido decidido, todas as instruções subseqüentes devem seguir-se em ordem em locais sucessivos. No entanto, não há necessidade de que os operandos sigam imediatamente após as instruções, nem é necessário que os operandos sejam colocados em locais sucessivos.

Tabela 9.4-1 O programa da Fig. 9.4-1 escrito atribuindo endereços simbólicos aos locais da memória a que devem ser feitas referências

Local simbólico	Conteúdo do local	Comentário
	CRA	Limpar Acc
	ADD W	Adicionar conteúdo de W a Acc
	ADD X	Adicionar conteúdo de X a Acc
	ADD Y	Adicionar conteúdo de Y a Acc
	STA Z	Armazenar conteúdo de Acc no local Z
	HLT	Parar
W	017	Operando em W
X	00B	Operando em X
Y	01C	Operando em Y
Z	XXX	Local de armazenamento do resultado

A seguir usamos nosso computador para subtrair um número de outro, por exemplo 0B7 – 09C (hex). Nossa máquina mais simples da Fig. 8.9-1 tinha uma instrução de subtração. Uma vez que nossa máquina presente tem uma organização e arquitetura que não incluem uma tal instrução, teremos que escrever um programa. Para efetuar uma subtração, devemos formar o negativo do subtraendo. Esta mudança de sinal pode ser executada no acumulador exercendo-se a capacidade do acumulador de complementar e incrementar. Devemos inicialmente carregar o subtraendo no acumulador; pois, se o minuendo fosse carregado antes, o acumulador não mais estaria disponível para mudar o sinal. O programa da subtração é dado na Tab. 9.4-2. Aqui a terminologia foi modificada de "endereço simbólico" para o termo mais usual *rótulo* (*label*).

Tabela 9.4-2 Programa da subtração

Rótulo (Label)	Conteúdo	Comentário
	CRA	Limpar Acc
	ADD SUB	Adicionar subtraendo a Acc
	CTA	Complementar Acc
	ITA	Incrementar Acc
	ADD MIN	Adicionar minuendo a Acc
	STA DIF	Armazenar em local rotulado "DIF"
	HLT	Parar
SUB	09C	Subtraendo em local rotulado "SUB"
MIN	0B7	Minuendo em local rotulado "MIN"
DIF	XXX	Local onde a diferença será armazenada (a diferença será 0B7 – 09C = 01B)

Tabela 9.4-3 Programa da subtração que envolve dois subtraendos

Rótulo	Conteúdo	Comentário
	CRA	Limpar Acc
	ADD SUB (1)	Adicionar ao Acc conteúdo do local "SUB (1)" (subtraendo 1)
	CTA	Complementar Acc
	ITA	Incrementar Acc
	STA NSUB (1)	Armazenar no local "NSUB (1)" (negativo de subtraendo 1)
	CRA	Limpar Acc
	ADD SUB (2)	Adicionar subtraendo (2) ao Acc
	CTA	Complementar Acc
	ITC	Incrementar Acc
	ADD MIN	Adicionar ao Acc conteúdo do local "MIN" (minuendo)
	ADD NSUB (1)	Adicionar conteúdo do local "NSUB (1)"
	STA RES	Armazenar no local "RES" (resultado)
	HLT	Parar
MIN	0B7	Minuendo no local "MIN"
SUB (1)	09C	Subtraendo 1 no local "SUB (1)"
SUB (2)	005	Subtraendo 2 no local "SUB (2)"
NSUB (1)	XXX	Local de armazenamento de negativo de SUB (1)
RES	XXX	Local de armazenamento do resultado

A Tab. 9.4-3 mostra o programa que calcula 0B7 − 09C − 005. Aqui, uma vez que há dois subtraendos, o registrador do acumulador deve ser usado duas vezes para trocar sinais. Quando uma mudança de sinal for efetuada, o resultado deve ser armazenado na memória para deixar o acumulador disponível para trocar o sinal do segundo subtraendo.

9.5 USO DE JMP E ISZ

Como um exemplo da utilidade das instruções JMP e ISZ e da característica indireta, vejamos como nossa máquina pode ser usada para formar a soma de um grande número (seja 100) de adendos. Em qualquer caso, teremos que colocar nossos cem adendos na memória. Depois disto podemos escrever um programa em que simplesmente incluímos cem instruções ADD sucessivas, que diferem uma da outra somente no endereço do operando. Um programa menor e mais elegante está listado na Tab. 9.5-1.

O programa provê um local (ANA) da memória que retém o endereço do próximo adendo a ser somado. Após cada adição, este endereço é incrementado. Um segundo local de contador

Tabela 9.5-1 Um programa para somar um grande número de adendos

Rótulo (Label)	Conteúdo	Comentário
	CRA	Limpar Acc
LOOP	ADDI ANA	Adicionar ao Acc o número cujo endereço está no local "ANA" (endereço do próximo adendo)
	ISZ ANA	Incrementar endereço de adendo
	ISZ CTR	Incrementar número em local "CTR" e saltar próxima instrução se incremento faz o conteúdo de CTR igual a 0
	JMP LOOP	JMP de volta à instrução rotulada "LOOP"
	STA RES	Armazenar resultado em local "RES"
	HLT	Parar
RES	XXX	Resultado a ser armazenado aqui
ANA	FAD	Este local guarda endereço do próximo adendo
CTR	F9C	Este local guarda contagem de número de adições a serem feitas (representação de − 100 em complemento de dois)
FAD	ADD (1)	100 adendos a serem somados
	ADD (2)	
	⋮	
	ADD (100)	

de memória especial (CTR) é usado para reter a contagem do número de adendos somados. As adições repetitivas são programadas fornecendo um laço repetidas vezes até que o serviço seja feito. O local do contador retém o número −100 (decimal) = F9C (hex) no início. Em cada adição o contador é incrementado até que finalmente atinja zero.

Com referência agora ao programa na Tab. 9.5-1, achamos que, depois do acumulador ter sido limpo, a próxima instrução fornece uma adição *indireta*. O local ANA da memória guarda não o adendo, mas o endereço do adendo. No início, o endereço em ANA é o primeiro endereço (FAD) do adendo e notamos que no endereço FAD temos o primeiro adendo. Tendo colocado ADD(1) no acumulador, agora incrementamos o endereço mantido em ANA. Para esta operação de incrementação usamos uma instrução ISZ. Embora não antecipemos que a entrada em ANA tornar-se-á sempre zero, achamos conveniente usar esta instrução para incrementar, mesmo que não tiremos vantagem da operação inteira que a instrução pode executar. A seguir incrementamos o número no local CTR. Desde que o local CTR não tenha ainda alcançado zero, a próxima instrução não é saltada (skip). Esta próxima instrução enlaça o programa de volta à instrução ADDI, em conseqüência de que o próximo adendo é adicionado ao acumulador. Notar que, desde que necessário, em um outro ponto, no programa, para recorrer novamente à instrução ADDI, é necessário especificamente atribuir um rótulo a seu endereço. O programa reverterá cem vezes até que finalmente o conteúdo do local do contador seja zero. Neste zero, a instrução JMP será saltada (skip), o resultado será armazenado e o computador irá parar.

9.6 PROGRAMA DA MULTIPLICAÇÃO

Um modo de programar multiplicação consiste simplesmente em usar uma sucessão de adições. Assim, para multiplicar 18 por 16, adicionamos 18 a si próprio dezesseis vezes ou 16 a si próprio dezoito vezes. Este esquema um tanto deselegante não deve ser desaprovado quando usado com computadores muito pequenos e lentos. Mesmo tão bem concebido, um minicomputador como o pdp-8 DEC não proporciona nenhum outro método em seu esquema básico de instruções. Enquanto este expediente também estiver disponível para nós, consideraremos igualmente o método de multiplicação que usa deslocamento e adição, a maneira pela qual a multiplicação é ordinariamente feita com lápis e papel (Fig. 5.15-1). Somos capazes de usar este método porque incorporamos em nossa máquina provisão para deslocamento do conteúdo do acumulador e porque providenciamos uma entrada para o controlador a partir do flip-flop F, que é uma extensão do acumulador.

O esquema é como se segue. Na memória providenciamos locais para armazenamento do multiplicando, do multiplicador e da soma dos produtos parciais (multiplicando = MD, multiplicador = MR, soma dos produtos parciais = SP). Transferimos o multiplicador para o registrador do acumulador e circulamos à direita, deslocando deste modo o bit menos significativo do multiplicador para dentro do flip-flop F flag. O multiplicador deslocado é então retornado a seu local MR, de modo que possamos usar o acumulador para tratar do multiplicando. O multiplicando é então transferido para o acumulador. Se F = 1, o conteúdo do local SP (SP inicialmente é um local de memória limpo) é adicionado ao multiplicando no registrador do acumulador e o conteúdo do acumulador é depois retornado ao local SP. Depois retornamos o multiplicando ao acumulador, giramos à esquerda e retornamos o multiplicando deslocado a seu local MD de armazenamento. Se F = 0, a adição do multiplicando ao conteúdo de SP não é feita, mas o deslocamento do multiplicando é ainda efetuado. A rotação à direita do multiplicador é feita de modo que possamos verificar cada bit do multiplicador e tomar um curso ou outro, dependendo do bit ser 1 ou 0. Esta verificação e a conseqüente variação no curso de ação podem ser feitas somente pelo deslocamento do bit no flip-flop F, uma vez que somente este flip-flop na ULA tem uma ligação com o controlador. O deslocamento à esquerda do multiplicando é feito para multiplicá-lo pelos fatores sucessivos de 2.

Em nossa máquina, uma vez que os dados numéricos têm doze bits, este processo de adicionar produtos parciais (sempre que F = 1) e de deslocamento deve ser feito doze vezes. Devemos, portanto, incorporar um *laço* ao nosso programa e ter um local de *contador* na memória, que serve para contar os números de vezes que o laço é completado.

O programa da multiplicação é dado na Tab. 9.6-1. Admitimos aqui que o multiplicando e o multiplicador são tais que o produto possa estar contido em doze bits. (Num máximo o produto de dois números de doze bits se estenderá a 24 bits.) As instruções nos locais 1 e 2 da memória limpam o local SP, onde a soma de produtos parciais deve ser armazenada. Não temos nenhum meio direto de limpar um local da memória. Por conseguinte, para limpar SP, limpamos, em vez disto, o registrador do acumulador e depois deslocamos para SP o conteúdo do acumulador. Os locais 3, 4 e 5 colocam o bit mais à direita do multiplicador no flip-flop F flag, deslocam o multiplicador e armazenam-no de volta na memória. Os locais 6, 7 e 8, juntos, determinam o curso resultante do programa, dependendo de ser F = 1 ou F = 0. Os locais 9, 10, 11 e 12, que são parte do programa se F = 1, adicionam o multiplicando ao conteúdo do local SP. Os locais 14, 15, 16 e 17 deslocam o multiplicando, retornando-o a seu local de armazenamento. A operação

Tabela 9.6-1 Programa da multiplicação

Local da memória	Rótulo (*Label*)	Conteúdos	Comentário
1		CRA	Limpar local onde a soma dos produtos SP, isto é, resultado da multiplicação, será armazenada
2		STA SP	
3	LOOP	ADD MR	Carregar o conteúdo do local MR em Acc
4		ROR	Girar à direita para deslocar bit mais à direita do multiplicador dentro de F
5		STA MR	Colocar multiplicador deslocado de volta ao local MR
6		SFZ	Saltar (skip) próxima instrução se F = 0
7		JMP I	Saltar (jump) para instrução em local rotulado "1" (desde que F ≠ 0)
8		JMP 0	Saltar (jump) para instrução em local rotulado "0" (desde que F = 0)
9	1	CRA	Carregar conteúdo de local MD em Acc
10		ADD MD	
11		ADD SP	Adicionar conteúdo do local SP a Acc
12		STA SP	Armazenar conteúdo de Acc no local SP
13		CRF	Limpar flip-flop F
14	0	CRA	Carregar conteúdo do local MD em Acc
15		ADD MD	
16		ROL	Girar à esquerda
17		STA MD	Colocar multiplicando deslocado de volta ao local MD
18		ISZ CTR	Incrementar contador
19		JMP LOOP	Contador não zero, saltar (jump) de volta para instrução em "LOOP"
20		HLT	Parar
21	CTR,	FF4	Representação hexadecimal de −12
22	MD	Multiplicando	Local para guardar multiplicando
23	MR	Multiplicador	Local para reter multiplicador
24	SP	XXX	Local para reter resultado (limpo pelas duas primeiras instruções)

de limpeza na linha 13 é necessária para evitar deslocamento de um 1 para dentro do local mais à direita do acumulador quando o multiplicando for deslocado para multiplicar por 2. O local 18 incrementa a entrada no local (CTR) do contador e o local 19 transporta o programa de volta para LOOP. A entrada em CTR é FF4, que é a representação de complemento de dois hexadecimais de −12 (decimal).

9.7 PROGRAMA QUE ILUSTRA A CHAMADA DE SUB-ROTINA

Para ilustrar uma chamada de sub-rotina, consideremos um programa para computar a quantidade $N_1N_2 + N_3N_4 + N_5N_6 + \ldots$, sendo os Ns números positivos sem sinais. Aqui há multiplicações e adições repetidas. O programa da adição é tão simples, que podemos escrever um programa de adição repetidas vezes e tão freqüentemente quanto for solicitado. O programa da multiplicação, por outro lado, é bastante longo para tornar conveniente instalar a rotina da multiplicação numa sub-rotina de modo que possamos chamá-lo repetidamente. O programa na Tab. 9.7-1 calcula a quantidade $N_1N_2 + N_3N_4$. O programa é escrito de modo que, se quisermos acrescentar termos adicionais (N_5N_6 etc.) temos apenas que introduzir os números adicionais e proporcionar etapas adicionais de programa para fazer a adição.

Os locais 1 a 4 da memória retêm os dados. Uma vez que deveremos ter acesso a estes números, seus locais foram identificados por rótulos. O local 5, rotulado novamente para acesso, simplesmente retém o número -12. Será lembrado que, na rotina da multiplicação, partimos do número -12 armazenado num local da memória rotulado "CTR". Quando a multiplicação estiver concluída, este CTR estará limpo. Daí necessitarmos ter disponível o número -12 para que possamos carregá-lo em CTR no começo de cada nova chamada na sub-rotina da multiplicação. As instruções nos locais 6 a 11 carregam os números N_1 e N_2 naqueles locais da memória associados à sub-rotina da multiplicação que estão reservados para o multiplicando e para o multiplicador.

O local 12 chama a sub-rotina da multiplicação. A parte desta sub-rotina entre os locais 32 e a linha 50 é idêntica ao programa da Tab. 9.6-1. O local 28 guarda o endereço do local da memória onde a próxima instrução deve ser achada depois da sub-rotina ter sido completada. Não há nenhuma instrução neste local da memória. Esta linha é rotulada ("MULT") uma vez que teremos que ler este endereço para fazer voltar ao programa principal. A primeira vez que a rotina da multiplicação é chamada, o endereço colocado neste local (28) será o endereço 13. Para antecipar, podemos notar que há uma segunda chamada de sub-rotina no local 23. Quando a instrução da linha 23 tiver sido executada, ela fará com que o endereço 24 de retorno (que é o novo valor do endereço de retorno) seja armazenado na linha 28. A instrução no local 51 é um salto (jump) de volta ao local "MULT" para apanhar o endereço de retorno, que nos leva de volta ao programa principal. No programa da Tab. 9.6-1, onde apenas uma simples multiplicação foi contemplada, a instrução correspondente (linha 18 da Tab. 9.6-1) foi HLT.

Uma vez que retornamos da sub-rotina, as instruções nos locais 13 a 16 limpam um local da memória rotulado "PR" (resultados parciais), onde o produto N_1N_2 é então armazenado. Os locais 17 a 22 carregam N_3 e N_4 na rotina da multiplicação, e o local 22 chama novamente a sub-rotina. Os locais 24 e 26 adicionam N_1N_2 a N_2N_3 e colocam a soma de volta ao local PR. Os termos adicionais N_5N_6 etc. podem ser formados e adicionados do mesmo modo e depois, finalmente, uma instrução HLT terá que ser anexada.

9.8 MICROPROGRAMAÇÃO

No Cap. 8 consideramos inúmeras maneiras pelas quais um controlador pode ser projetado. Consideremos agora uma técnica adicional de realizar controle chamada microprogramação. Como veremos, um computador controlado por microprograma é, de certo modo, um computador dentro

Tabela 9.7-1 Programa em que o programa da multiplicação aparece como uma sub-rotina

Local da memória	Rótulo (*Label*)	Conteúdo	Comentário
1	A	N_1	Locais A, B, C e D da memória retêm números N_1, N_2, N_3 e N_4 a serem combinados em $N_1N_2 + N_3N_4$
2	B	N_2	
3	C	N_3	
4	D	N_4	
5	T	FF4	Representação hexadecimal de -12
6		CRA	Coloca N_1 no local "MD", reservado ao multiplicando na sub-rotina da multiplicação
7		ADD A	
8		STA MD	
9		CRA	Coloca N_2 no local "MR", reservado ao multiplicador na sub-rotina da multiplicação
10		ADD B	
11		STA MR	
12		CSR MULT	Chama sub-rotina no endereço "MULT"
13		CRA	Limpa local "PR" a ser usado para armazenar o resultado parcial N_1N_2
14		STA PR	
15		ADD SP	Armazena resultado parcial N_1N_2 no local "PR"
16		STA PR	
17		CRA	Coloca N_3 e N_4 em "MD" e "MR", onde eles são acessíveis para sub-rotina da multiplicação
18		ADD C	
19		STA MD	
20		CRA	
21		ADD D	
22		STA MR	
23		CSR MULT	Chama novamente sub-rotina da multiplicação
24		CRA	Carrega Acc com N_3N_4 do local "SP"
25		ADD SP	
26		ADD PR	Adiciona N_1N_2 do local "PR" para formar $N_1N_2 + N_3N_4$
27		HLT	Pára, deixando $N_1N_2 + N_3N_4$ em Acc
28	MULT		Retém endereço de retorno (mutável)
29		CRA	Coloca -12 no local da memória rotulado "CTR"
30		ADD T	
31		STA CTR	
32		CRA	Sub-rotina da multiplicação
33		STA SP	
34	LOOP	ADD MR	
35		ROR	
36		STA MR	
37		SFZ	
38		JMP 1	
39		JMP 0	
40	1	CRA	
41		ADD MD	
42		ADD SP	
43		STA SP	

Tabela 9.7-1 (continuação)

Local da memória	Rótulo (*Label*)	Conteúdo	Comentário
			Continuação da sub-rotina da multiplicação
44		CRF	
45	0	CRA	
46		ADD MD	
47		ROL	
48		STA MD	
49		ISZ CTR	
50		JMP LOOP	
51		JMPI MULT	Retorna ao programa principal
52	CTR		Locais da memória rotulados reservados ao contador, multiplicando, multiplicador, soma de produtos parciais e aos resultados parciais
53	MD		
54	MR		
55	SP		
56	PR		

de um computador. Por esta razão retardamos a descrição do método até que tivéssemos alguma experiência elementar com uma estrutura do tipo de computador.

Vimos que as microoperações efetuadas por nosso computador são executadas em resposta a sinais aplicados aos terminais de controle dos registradores, à ULA e à memória. Estes sinais assim aplicados são chamados *sinais de comando*. Em qualquer intervalo de relógio uma ou mais entradas de controle devem ser elevadas ao nível lógico 1 pelos sinais de comando aplicados.

Agora, com referência à nossa máquina na Fig. 9.1-1, podemos verificar que há dezoito terminais de controle de entrada que em um instante de tempo ou noutro irão requerer um comando para se tornarem ativos. Imaginemos então uma memória de leitura exclusiva, como na Fig. 9.8-1, em que as palavras têm dezoito bits de comprimento, de modo que podemos, então, fazer uma associação um a um entre os bits da palavra e os terminais de entrada de controle. Na Fig. 9.8-1, numa base inteiramente arbitrária, atribuímos o bit 3 da palavra da memória ao terminal de controle que, quando ativado, realiza a microoperação GPR(AD) → MAR. Similarmente, atribuímos o bit 7 a M → MBR e bit 11 a GPR + Acc → Acc. A saída de três bits da ROM está conectada ao terminal de controle na Fig. 9.8-1 assinalado GPR(AD) → MAR etc. Vejamos agora, como exemplo, como podemos arranjar para usar a ROM para executar em seqüência adequada as microoperações necessárias à execução de uma instrução.

Consideremos, especificamente, a instrução ADDI, cujas microoperações, em número de cinco, estão detalhadas na página 358. Providenciemos, então, para que a ROM tenha escrito nela, em cinco locais sucessivos de palavra, a seqüência de palavras listada na Fig. 9.8-2. E, finalmente, admitamos que o registrador de endereços da Fig. 9.8-1 tenha partido no endereço da primeira palavra e depois incrementado através dos endereços remanescentes em sincronismo com o relógio que comanda o computador da Fig. 9.1-1. Durante o primeiro ciclo de relógio, somente o bit 3 está em lógica 1, de modo que apenas o terminal de controle GPR(AD) → MAR é tornado ativo conforme solicitado. No segundo intervalo de relógio, somente o bit 7 está em lógica 1, de modo que a microoperação M → GPR é realizada. No terceiro intervalo de relógio, temos novamente

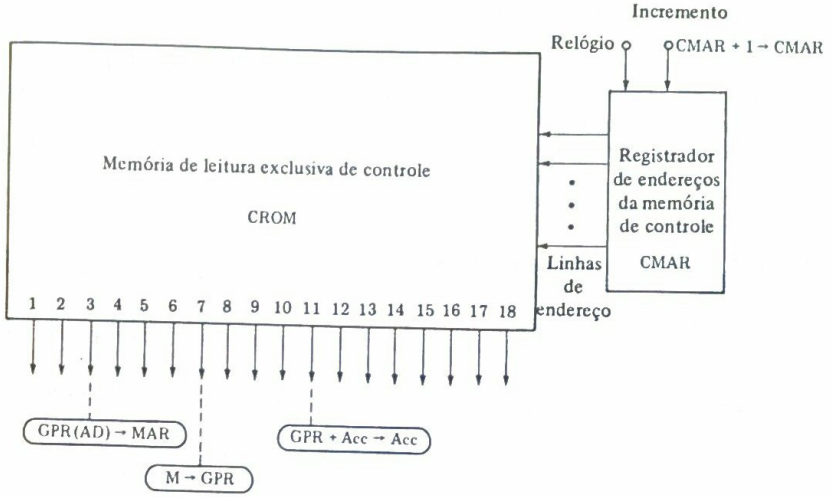


Fig. 9.8-1 Palavras lidas em sucessão a partir de uma ROM de controle proporcionam os sinais de comando seqüenciais requeridos para controlar um processador.

GPR (AD) → MAR. No quarto intervalo o bit 7 está novamente em lógica 1 e, portanto, novamente a microoperação M → GPR será executada. Finalmente, no quinto intervalo de relógio teremos GPR + Acc → Acc. Neste exemplo, revela-se que em cada ciclo de relógio somente uma operação necessita ser executada. Daí, em cada palavra CROM somente um bit está em lógica 1. Quando várias microoperações necessitam ser executadas no mesmo ciclo de relógio, um número correspondente de bits estará em lógica 1.

Assim, em conjunto, pela leitura de uma sucessão de palavras, chamadas *palavras de controle* ou *micropalavras*, executamos uma seqüência de microoperações armazenadas na memória de leitura exclusiva e o resultado final é que executamos uma operação conforme requerido. As palavras na ROM que identificam as microoperações são também chamadas *microinstruções* (em contraste com as *instruções*, que são armazenadas na RAM de leitura-escrita do computador). Para evitar ambigüidade, as instruções de uma máquina podem ser chamadas *macroinstruções* para

	Número do bit																	
Palavra	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Fig. 9.8-2 As cinco micropalavras sucessivas na ROM da Fig. 9.8-1 que fornecem as microinstruções para causar a execução das microoperações requeridas pela instrução ADDI.

distingui-las das microinstruções armazenadas na ROM de controle. Um computador cujo controlador opera conforme descrito aqui é chamado computador *microprogramado*. Notar que um tal computador tem duas memórias: (1) como todos os computadores, ele tem uma RAM, que contém as instruções e dados, e (2) em seu controlador ele tem uma ROM, que contém a microinstrução usada para executar as instruções. Esta segunda memória é geralmente denominada *memória de leitura exclusiva de controle* (CROM) e seu registrador de endereços é, correspondentemente, chamado *registrador de endereços da memória de controle* (CMAR).

9.9 DESVIO (BRANCH) DO MICROPROGRAMA

O esquema simples de microprogramação da seção precedente descreve o princípio da microprogramação, mas não é versátil o bastante para ser de muito uso numa situação real. Nesta e nas seções seguintes consideraremos algumas sofisticações.

No arranjo da Fig. 9.8-1 somos capazes de ler microinstruções somente na ordem em que elas estão escritas na memória, mas gostaríamos de ser capazes de desviar ou de saltar (jump) para um local arbitrário na memória. Um arranjo que permite um tal desvio é mostrado na Fig. 9.9-1. Aqui admitimos que há N bits de comando para fornecer N entradas para os terminais de controle dos registradores do computador e que o número de microinstruções a serem armazenadas é tal que é requerido um endereço da memória de controle de M bits. Providenciamos então para que o número de bits na palavra da ROM seja $N + M + 1$. Os N bits são bits de comando, e os M bits constituem o endereço em que um salto (jump) deve ocorrer, se realmente um salto (jump) deve ocorrer. O bit extra é usado para determinar se um salto (jump) é solicitado ou não. Este bit extra é o *bit de controle de carga*. Se este bit estiver em lógica 0, a entrada de incremento está em lógica 1 e o registrador de endereços será incrementado em cada intervalo de relógio. Se, no entanto, o bit de controle de carga estiver em lógica 1, não ocorrerá a incrementação. Em vez disto, teremos lógica 1 no terminal de entrada de endereço de desvio de carga do registrador de endereços e na borda de gatilho do relógio o endereço de M bits a que queremos

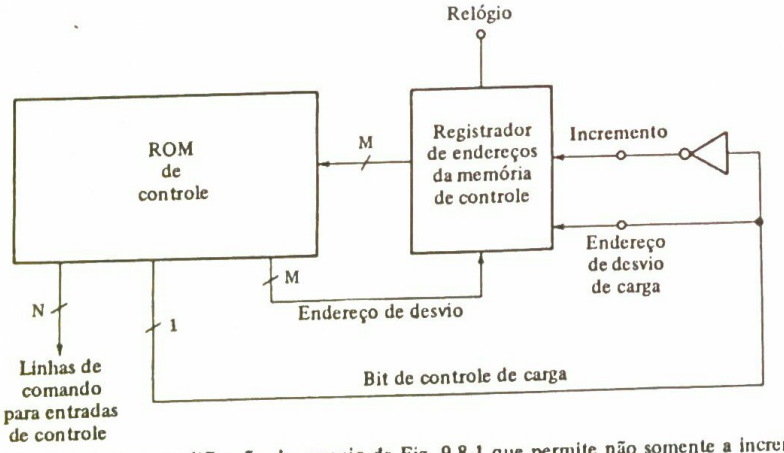


Fig. 9.9-1 Uma modificação do arranjo da Fig. 9.8-1 que permite não somente a incrementação do endereço da CROM, mas também o desvio para um endereço especificado pela micropalavra precedente.

saltar (jump) será carregado no registrador de endereços. Naturalmente, o registrador de endereços da Fig. 9.9-1 é, no assunto de implementação de hardware, mais complicado do que o registrador da Fig. 9.8-1. O primeiro registrador deve ser capaz somente de incrementar, enquanto o segundo registrador deve ser capaz de incrementar ou de aceitar uma carga paralela.

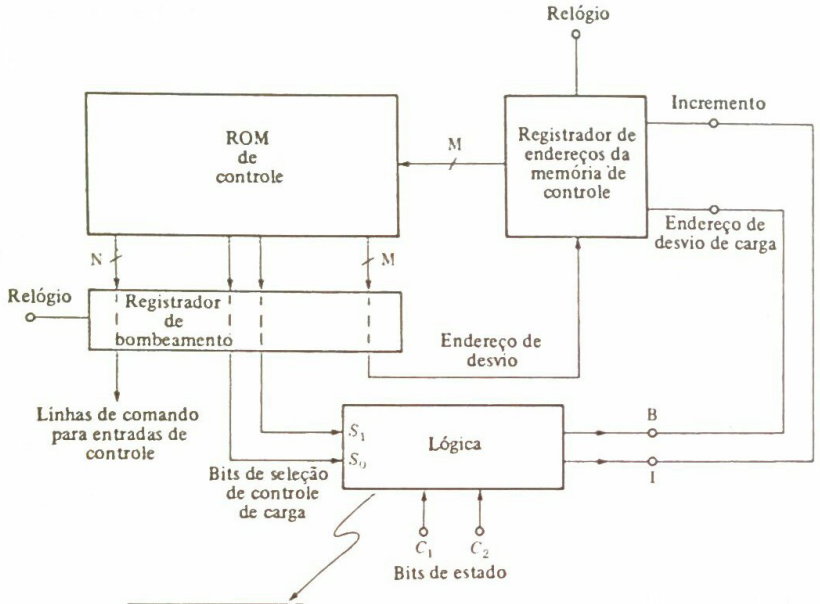
Na ROM de controle do controlador microprogramado da Fig. 9.9-1, providenciamos suficientes bits por palavra para que uma palavra contenha tanto um endereço como uma microinstrução. Esta proliferação de bits é algo que geralmente é evitado no estabelecimento do comprimento da palavra da memória RAM. Há duas boas razões para esta diferença na inclinação para estender o comprimento da palavra: (1) RAMs são mais dispendiosas que ROMs de igual tamanho e (2) maior comprimento de palavra na RAM, que retém instruções (em vez de microinstruções) e dados, exigiria, correspondentemente, comprimento de palavra mais longo em quase todos os registradores do computador. Tal não é o caso quando o comprimento de palavra da ROM é aumentado. Verificamos, portanto, que, geralmente, enquanto os projetistas acrescentam comprimento ao tamanho da palavra de instruções e de dados, somente após uma decisão deliberada de construir um computador mais dispendioso, bits são acrescentados livremente à memória de controle. Atualmente, o controle microprogramado está chegando a ser olhado mais favoravelmente. Ele constitui a maneira mais sistematizada e ordenada de projetar um controlador e as ROMs requeridas em sua implementação de hardware são agora pouco dispendiosas e prontamente encontradas.

9.10 DESVIO CONDICIONAL

Consideremos a seguir o arranjo da Fig. 9.10-1. O *registrador de bombeamento* (*pipeline*) mostrado não é importante em nossa presente discussão e foi incluído na figura de modo que possamos usar esta figura novamente na discussão da seção seguinte. Por enquanto, então, ignoramos o registrador de bombeamento e consideramos as linhas tracejadas através deste registrador simplesmente como sendo continuações das linhas de saída da ROM de controle.

O arranjo da Fig. 9.10-1 introduz flexibilidade adicional no controlador. Aqui providenciamos dois bits S_1 e S_0 de seleção de controle de carga. Os bits C_1 e C_2 são *bits de estado*, que são derivados de algum local no computador e destinados a indicar se alguma condição é satisfeita ou não. Por exemplo, o bit C_1 pode ser a entrada Z do controlador na Fig. 9.1-1. Se C_1 (= Z) for $C_1 = 1$, o registrador de propósitos gerais está limpo. Se $C = 0$, este registrador não está limpo. Assim, o bit C_1 fornece informação a respeito do estado do GPR. Similarmente, C_2 pode fornecer informação de estado a respeito de algum outro componente do computador. Dessa forma, citando novamente a Fig. 9.1-1, C_2 pode ser F. Estes bits de estado, quando tornados disponíveis para o controlador, permitem que o controlador siga uma seqüência ou outra dependendo do nível lógico do bit de estado.

O bloco *lógico* na Fig. 9.10-1 é uma estrutura combinacional cuja tabela verdade é conforme dada (ver Prob. 9.10-1); quando $S_1S_0 = 00$, a saída do bloco lógico I = 1 e a saída B = 0. Admitamos que os terminais de controle do registrador de endereços da memória de controle sejam ativos em lógica 1. Por conseguinte, quando $S_1S_0 = 00$, o registrador incrementará. Similarmente, quando $S_1S_0 = 01$, o registrador desviará para o endereço especificado pelo campo de endereço da micropalavra cuja instrução está sendo implementada. Em ambos estes casos os bits de estado são ignorados. Quando $S_1S_0 = 10$, o bit C_1 de estado determina se ocorre a incrementação ou o desvio, enquanto quando $S_1S_0 = 11$, o *bit de estado* C_2 determina



S_1	S_0	C_1	C_2	B	I	
0	0	×	×	0	1	Incremento: ignora bits de estado
0	1	×	×	1	0	Desvio: ignora bits de estado
1	0	0	×	0	1	Responde somente a C_1: incrementa se
1	0	1	×	1	0	$C_1 = 0$; desvia se $C_1 = 1$
1	1	×	0	0	1	Responde somente a C_2: incrementa se
1	1	×	1	1	0	$C_2 = 0$, desvia se $C_2 = 1$

Fig. 9.10-1 Uma outra modificação de um controlador microprogramado que permite que a operação do controlador seja influenciada pelos bits de estado.

se ocorre a incrementação ou o desvio. (Uma vez que I e B são sempre complementares entre si, podíamos ter permitido apenas uma única linha de saída do bloco lógico e arranjado a complementaridade como na Fig. 9.9-1. Não fizemos assim porque num ponto subseqüente faremos algumas modificações que permitem I e B estarem simultaneamente em lógica 0.)

9.11 BOMBEAMENTO (PIPELINING)

Veremos nesta seção como o acréscimo do *registrador de bombeamento* serve para aumentar a velocidade com que o controlador microprogramado pode operar. Consideremos primeiro, na *ausência* do registrador de bombeamento, os atrasos de tempo associados ao controlador e ao sistema que ele controla. Suponhamos, na ausência do registrador de bombeamento, que ocorre

uma transição de gatilho de relógio no instante de tempo $t = t_0$. O tempo t_0 é o tempo de início para a operação de estabelecimento de um novo endereço no registrador de endereços da memória de controle quer por incrementação, quer por carregamento. Quanto tempo devemos esperar agora antes que possamos permitir a próxima transição de gatilho? Primeiro devemos esperar um tempo t_A para permitir que um endereço válido seja estabelecido na saída do registrador de endereços. (O endereço é válido depois de todas as M linhas de saída de registrador de endereços apresentarem o novo bit adequado à ROM e não devem ocorrer outras modificações.) A seguir devemos esperar o tempo adicional t_M do atraso de propagação através da memória de controle. Depois de um tempo $t_A + t_M$, a microinstrução endereçada será validamente estabelecida na saída da memória de controle. Devemos agora esperar, ainda, que o sistema controlado responda às microinstruções, complete sua resposta e esteja pronto para a próxima instrução. Se chamarmos de t_R o tempo de resposta do sistema, o período mínimo de relógio permissível é $t_A + t_M + t_R$. O mais longo destes componentes de tempo é t_R, que deve ser bastante longo para acomodar a microoperação mais lenta, geralmente a operação de leitura de ou escrita em uma RAM.

O processo de estabelecer uma nova micropalavra de saída da ROM *começa* no momento da ocorrência da transição de gatilho de relógio quando o registrador de endereços começa a incrementar ou a carregar um endereço de desvio. Agora nos ocorre que podemos economizar tempo *começando* a estabelecer uma nova micropalavra antes do sistema controlado ter concluído sua resposta à micropalavra anterior.

Seja t_X o tempo de atraso da propagação da caixa lógica na Fig. 9.10-1 e seja o período de relógio $t_C = t_A + t_M + t_X$. Em, por exemplo, $t = t_0$, uma borda de relógio dá partida ao registrador de endereços no processo de incrementação e carregamento de um endereço de desvio. Em $t = t_0 + t_A + t_M$ uma nova micropalavra é validamente estabelecida na saída da ROM, e em $t = t_0 + t_A + t_M + t_X = t_0 + t_C$ um bit de controle de carga válido foi fornecido ao registrador de endereços de modo que possa ocorrer a próxima borda de gatilho de relógio. Com o período de relógio estabelecido em $t_C = t_A + t_M + t_X$, novas micropalavras válidas aparecerão na saída da ROM em sucessão na taxa de relógio. Suponhamos, agora, que o tempo t_R de resposta do sistema seja menor do que o período do relógio. Então o sistema controlado terá tempo adequado para responder a cada microinstrução. Por exemplo, suponhamos que o período do relógio seja $t_C = t_A + t_M + t_X = 2 + 3 + 1 = 6\ \mu s$ e que o tempo de resposta do sistema seja $t_R = 4\ \mu s$. Novas microinstruções serão apresentadas ao sistema controlado a cada 6 μs, e o sistema concluirá sua resposta em 4 μs. Bordas de relógio em $t = 0, 6, 12\ \mu s$ irão gerar microinstruções válidas em $t = 6, 12, 18\ \mu s$. O sistema responderá à microinstrução que for apresentada em $t = 6$ durante o intervalo de $t = 6$ até $t = 10$. O sistema depois espera para receber sua próxima microinstrução em $t = 12$ e faz sua resposta correspondente de $t = 12$ a $t = 16$ etc. O ponto importante é que o tempo é economizado porque o sistema faz sua resposta a uma microoperação *no mesmo tempo* que a próxima microoperação está fazendo seu caminho através dos dutos de bombeamento, isto é, através da caixa lógica combinacional do registrador de endereços e da ROM. O período de relógio é 6 μs em vez de 10 μs, como seria se esperássemos que a resposta do sistema estivesse concluída antes que começássemos a gerar uma nova microoperação.

O argumento do parágrafo anterior, que indica como a taxa de relógio pode ser aumentada, realmente *não é válido*. Admite-se ali que, se em $t = t_0$ ocorre uma transição de gatilho, uma nova micropalavra de saída da ROM aparecerá em $t_0 + t_A + t_M$, que no intervalo interveniente de t_0 a $t_0 + t_A + t_M$ não ocorrerá *nenhuma variação* na saída da ROM e que durante o intervalo

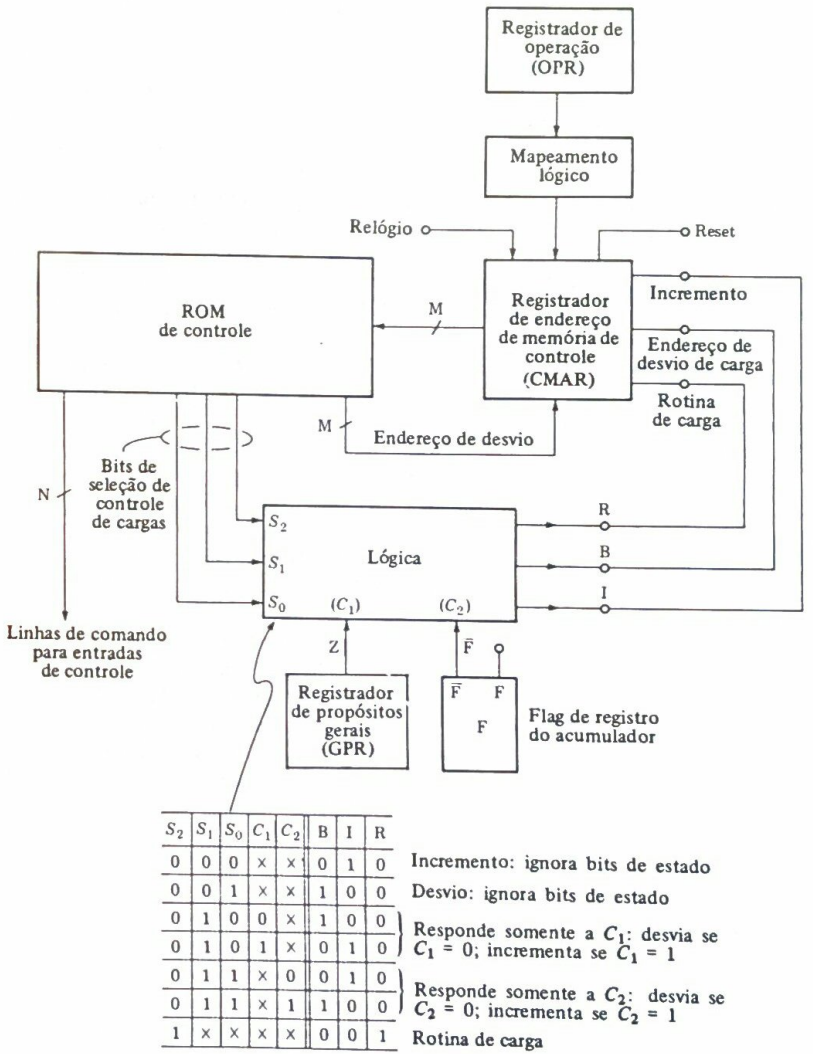
a saída da ROM reterá a micropalavra anterior. Tal não é o caso. É verdade que necessitamos esperar um tempo $t_A + t_M$ para nos certificarmos de que *todas* as variações na saída da ROM que vão ocorrer realmente ocorreram. No entanto, algumas variações podem ocorrer após um intervalo muito menor. Assim, enquanto podem ser necessários 5 μs para que seja estabelecida uma nova saída da ROM completamente válida, algumas saídas da ROM podem modificar-se em aproximadamente 0,1 μs. Uma vez que *não* devemos permitir variações na saída da ROM até que o sistema tenha completado sua resposta, devemos esperar por uma conclusão da resposta do sistema antes que nós mesmos *comecemos* o processo de mudança de endereço.

Esta dificuldade é aliviada acrescentando-se o *registrador de bombeamento* indicado na Fig. 9.10-1, conforme veremos agora. Este registrador não constitui realmente o duto do bombeamento, uma vez que, na realidade, são os atrasos de propagação através do bloco lógico, do registrador de endereços e da ROM que são análogos a um duto de bombeamento. O registrador de bombeamento serve como uma "válvula" no final do duto de bombeamento, permitindo que o duto de bombeamento opere eficientemente. O registrador de bombeamento é acionado pela mesma forma-de-onda de relógio que aciona o registrador de endereços. A transição de relógio que incrementa ou carrega o registrador de endereços é a mesma transição que carrega o registrador de bombeamento com a micropalavra de saída da ROM. A resposta do sistema controlado pode agora começar no instante em que o registrador de bombeamento for carregado. *Neste exato momento* podemos começar a mudar o endereço a fim de fazer aparecer a próxima microinstrução. Enquanto o sistema controlado estiver completando sua resposta, os bits da palavra de saída da ROM podem realmente estar mudando, mas estas mudanças não causam nenhum problema porque o sistema está separado da microinstrução variável por causa do isolamento proporcionado pelo registrador de bombeamento.

9.12 UM CONTROLADOR MICROPROGRAMADO

No computador da Fig. 9.1-1 incluímos um controlador para fornecer sinais de habilitação de comando a todos os registradores componentes, de modo que cada registrador executará uma das microoperações de seu repertório quando for necessário. Para ilustrar o princípio da microprogramação, consideraremos agora como o controlador pode ser realizado como um controlador microprogramado. O controlador microprogramado é mostrado na Fig. 9.12-1 e é padronizado após o controlador da Fig. 9.10-1. Para simplificar, omitimos o aspecto de bombeamento, uma vez que o bombeamento não afeta o princípio de operação, apenas sua velocidade.

No diagrama do sistema de computador completo (Fig. 9.1-1) notamos que o controlador tem entradas do registrador de operação (OPR), do registrador de propósitos gerais (GPR) e do indicador do acumulador F. Estes dois registradores e o registrador indicador estão incorporados na Fig. 9.12-1. O bit C_1 de desvio de condição na Fig. 9.10-1 torna-se o bit Z fornecido pelo GPR. Este bit fornece informação concernente ao *estado* do GPR. Quando o registro no GPR for zero, Z = 1; do contrário, Z = 0. Similarmente, a linha lógica F, substituindo C_2, fornece um bit de *estado* ao controlador informando ao controlador se o flip-flop flag F está ajustado (set) ou reajustado (reset). No controlador da Fig. 9.10-1 fizemos provisão quer para incrementar o registrador de endereços da memória de controle (CMAR), quer para carregar nele o endereço de desvio. No caso presente, fizemos provisão, igualmente, para permitir o carregamento no CMAR de um endereço determinado pelo código de operação registrado no registrador de operações.



S_2	S_1	S_0	C_1	C_2	B	I	R	
0	0	0	×	×	0	1	0	Incremento: ignora bits de estado
0	0	1	×	×	1	0	0	Desvio: ignora bits de estado
0	1	0	0	×	1	0	0	Responde somente a C_1: desvia se $C_1 = 0$; incrementa se $C_1 = 1$
0	1	0	1	×	0	1	0	
0	1	1	×	0	0	1	0	Responde somente a C_2: desvia se $C_2 = 0$; incrementa se $C_2 = 1$
0	1	1	×	1	1	0	0	
1	×	×	×	×	0	0	1	Rotina de carga

Fig. 9.12-1 Uma outra elaboração do controlador microprogramado que faz provisão de carregamento no CMAR do endereço da CROM de partida de uma rotina.

(O endereço real é determinado pelo OPR e pela lógica de mapeamento combinacional, como veremos.) O sinal de comando para carregamento deste endereço determinado pelo registrador de operações é assinalado "rotina de carga". Notar que o campo de seleção de controle de carga con-

siste agora em três bits, em vez de dois bits como na Fig. 9.10-1. Descreveremos agora como opera o controlador microprogramado.

A seqüência de microoperações (escrita na ROM de controle como uma seqüência de micropalavras) que executa as etapas necessárias para buscar uma instrução é chamada *rotina de busca* (*fetch*). Similarmente, uma seqüência que causa a execução de uma instrução é chamada *rotina de executar*. Num certo local na ROM de controle, seja começando num local que chamaremos ADDR(FETCH), escreveremos a rotina de busca. O controlador entrará em seqüência através da rotina de busca incrementando o CMAR após cada microoperação. No fim da rotina de busca, a operação solicitada pela instrução buscada terá sido carregada no registrador de operações. Suponhamos, por exemplo, que a instrução que foi buscada é a instrução ADDI. A rotina de execução da instrução ADDI será localizada na ROM de controle começando num local ADDR(ADDI). É, portanto, necessário, no final da rotina de busca, que haja um *desvio* para a rotina ADDI que comece em ADDR(ADDI). Este endereço ADDR(ADDI) é, naturalmente, determinado pelo *código de operação* no registrador de operações. Seria muito conveniente se fôssemos capazes de arranjar para que o código OP da ADDI fosse o mesmo que o endereço ADDR(ADDI). Se tal fosse o caso, uma transferência para a rotina ADDI poderia ser realizada pelo carregamento do código OP no CMAR. Um tanto genericamente, contudo, não é viável arranjar esta característica conveniente e, portanto, é necessário interpor entre o registrador de operações e o CMAR um bloco combinacional apropriado que aceita o código OP como entrada e fornece como saída o endereço que dá partida à rotina da instrução. Esta tradução do código OP no endereço de partida de uma rotina é chamada *mapeamento*, e na Fig. 9.12-1 é realizada pelo bloco combinacional da lógica de mapeamento (que possivelmente será uma ROM). Em qualquer caso, agora é necessário proporcionar uma facilidade não somente para incrementação de CMAR e carregamento dele com o endereço de *desvio*, mas também para carregamento dele com a rotina de execução, cujo código OP está no OPR. Já incorporamos exatamente uma tal facilidade no controlador da Fig. 9.12-1.

A tabela verdade do bloco lógico na Fig. 9.12-1 é uma ligeira elaboração da tabela verdade na Fig. 9.10-1; isto é permitido porque providenciamos um bit de seleção de controle de carga adicional, S_2. Desde que $S_2 = 0$, R = 0 e o restante da tabela verdade na Fig. 9.12-1 é o mesmo que o dado na Fig. 9.10-1. Quando $S_2 = 1$, R = 1 e o endereço da rotina de execução da instrução especificado pelo registrador de operações será carregado no CMAR.

9.13 CONTEÚDO DA ROM DE CONTROLE

Consideremos agora o que podem ser conteúdos adequados da ROM de controle de um controlador microprogramado para nosso simples computador da Fig. 9.1-1. Estes conteúdos são mostrados na Fig. 9.13-1. Os bits das micropalavras recaem em três classes, ou campos. O *campo de operação* especifica a microoperação que deve ser efetuada. O *campo de seleção de endereços* especifica como o controlador deve selecionar o endereço na ROM de controle onde a próxima microoperação deve ser achada. Quando o endereço da próxima microoperação constitui uma parte da micropalavra, este próximo endereço será achado no *próximo campo de endereço*. Quando uma próxima microinstrução for selecionada pela incrementação do CMAR ou pelo carregamento neste registrador de um endereço determinado pelo registrador de operações, o conteúdo do próximo campo de endereço será irrelevante.

Nos locais ADDR(FETCH), ADDR(FETCH) + 1 e ADDR(FETCH) + 2 da ROM de controle, achamos as microoperações que executam a rotina de busca (ver página 355). Após a primeira microoperação e também após a segunda, o registrador de endereços deve ser incrementado; daí achamos nestes primeiros dois casos que o campo do endereço lê $S_2S_1S_0 = 000$. Após a terceira microoperação no local ADDR(FETCH) + 2 ter sido executada, o registrador de operações (cujo registro é decodificado pela lógica de mapeamento) retém o endereço CROM do começo da rotina de instrução que deve ser executada. Por conseguinte, neste caso a entrada da rotina de carga do registrador de endereços deve ser ativada e, correspondentemente, achamos que $S_2S_1S_0 = 1XX$.

Por exemplo, se a instrução assim carregada no registrador de endereços fosse a instrução ADDI, a seguir completaríamos a seqüência em ADDR(ADDI), ADDR(ADD) + 1 etc., que constitui a rotina de execução da instrução ADDI. Quando a instrução tiver sido executada, devemos retornar para a rotina de busca. Então a última micropalavra da rotina ADDI tem o código $S_2S_1S_0 = 001$ de seleção de endereços (desviar para o endereço dado no campo de endereços) e o campo de endereços retém o endereço ADDR(FETCH). Bastante genericamente, desta mesma maneira a última micropalavra de toda rotina de execução de instrução causará um desvio de volta a ADDR(FETCH).

A seguir, consideremos a rotina de ISZ [incrementar palavra da memória e saltar (skip) a próxima instrução se o número incrementado tiver o valor zero]. Esta instrução é especial porque envolve o bit de estado Z (também chamado C_1). A rotina desta instrução está escrita na Fig.

Localização de memória	Campo de operação	Campo de seleção de endereço			Campo seguinte de endereço
		S_2	S_1	S_0	
ADDR(FETCH)	PC → MAR	0	0	0	
ADDR(FETCH) + 1	M → GPR; PC + 1 → PC	0	0	0	
ADDR(FETCH) + 2	GPR(OP) → OPR	1	×	×	
⋮					
ADDR(ADDI)	GPR(AD) → MAR	0	0	0	
ADDR(ADDI) + 1	M → GPR	0	0	0	
ADDR(ADDI) + 2	GPR(AD) → MAR	0	0	0	
ADDR(ADDI) + 3	M → GPR	0	0	0	
ADDR(ADDI) + 4	GPR + Acc → Acc	0	0	1	ADDR (FETCH)
⋮					
ADDR(ISZ)	GPR (AD) → MAR	0	0	0	
ADDR(ISZ) + 1	M → GPR	0	0	0	
ADDR(ISZ) + 2	GPR + 1 → GPR	0	0	0	
ADDR(ISZ) + 3	GPR → M	0	1	0	ADDR(FETCH)
ADDR(ISZ) + 4	PC + 1 → PC	0	0	1	ADDR(FETCH)
⋮					
ADDR(SFZ)	NOP	0	1	1	ADDR(FETCH)
ADDR(SFZ) + 1	PC + 1 → PC	0	0	1	ADDR(FETCH)

Fig. 9.13-1 Conteúdos parciais da CROM da Fig. 9.12-1 se o controlador deve ser usado com o computador da Fig. 9.1-1.

9.13-1, começando no local ADDR(ISZ) da ROM (ver também página 362). As três primeiras micropalavras têm o campo de seleção de endereços $S_2S_1S_0 = 000$ para solicitar incrementação do CMAR. O GPR é incrementado durante a terceira microoperação. Depois desta operação de incrementação ter sido completada, saberemos se o GPR está limpo, em cujo caso Z = 1, ou não, em cujo caso Z = 0. A quarta microoperação tem o campo 010 de seleção de endereços, que indica que o próximo endereço da microoperação deve ser determinado por Z (C_1). Se Z = 0, haverá um desvio de volta para ADDR(FETCH), como indicado pela tabela verdade da Fig. 9.12-1. Se Z = 1, o CMAR não será carregado com o endereço de desvio, mas sim incrementado. Se tal incrementação ocorrer, a quinta microoperação PC + 1 → PC será executada e, depois, haverá um desvio para ADDR(FETCH).

Consideremos a seguir a instrução SFZ [saltar (skip) próxima macroinstrução se F = 0]. Como pode ser visto na Fig. 9.12-1, temos usado F como o bit C_2 de estado, de modo que $C_2 = \overline{F}$ quando F = 0. As microoperações requeridas para executar SFZ estão listadas na Fig. 9.13-1, começando em ADDR(SFZ). A primeira "microoperação" não constitui realmente nenhuma operação (NOP). Ela é conseguida arranjando-se para que todos os bits de comando da ROM de controle sejam 0s. Durante o intervalo da NOP, o controlador terá tempo para determinar se $C_2 (= \overline{F})$ é 0 ou 1. O campo de seleção de endereços nesta micropalavra é $S_2S_1S_0 = 011$, que, como mostra a tabela verdade da Fig. 9.12-1, indica que o controlador deve responder a C_2. Se $C_2 = 0$ (F = 1), haverá um desvio para o endereço no campo de endereços. O resultado será que realmente nenhuma operação terá sido efetuada e o controlador voltará para a rotina de busca. Se, no entanto, $C_2 = 1$, o CMAR será incrementado para ADDR(SFZ) + 1. Neste local achamos a microinstrução PC + 1 → PC, de modo que o contador de programa será incrementado e depois a rotina de busca começará. O resultado final é que uma macroinstrução terá sido saltada (skip).

Outras instruções podem ser escritas na memória de controle seguindo o mesmo padrão conforme aplicado às instruções na Fig. 9.13-1. Os detalhes são deixados como exercícios para os estudantes.

9.14 MÉTODOS DE ENDEREÇAMENTO

Vimos que um tanto genericamente (embora não necessariamente) uma instrução consiste em uma parte de operação e em uma parte do endereço. A parte de endereço pode guardar o endereço de um operando a ser usado na execução da instrução. Em outras ocasiões, a parte de endereço da instrução pode guardar não o endereço do operando, mas o endereço em que o endereço do operando deve ser achado. No primeiro caso, o endereço é descrito como um *endereço direto*; no segundo caso, ele é um *endereço indireto*. Em computadores, minicomputadores e microcomputadores, uma ampla faixa de modos de endereçamento é empregada, dos quais consideramos vários nesta seção.

Direto. No *endereçamento direto*, como já observado, a instrução contém o endereço do local na memória onde o operando deve ser achado.

Indireto. No *endereçamento indireto*, novamente como observado, a instrução contém não o local da memória onde o operando deve ser achado, mas o endereço do local da memória onde o endereço do operando deve ser achado.

Relativo. No *endereçamento relativo* a parte de endereço da instrução contém um número N.

O endereço, na memória, do operando é achado pela adição do número N ao número no contador do programa. Por exemplo, se o contador do programa (PC) guardar o número 17, e se $N = 14$, o endereço do operando é PC + N = 17 + 14 = 31. O número N pode ser positivo, como em nosso exemplo, ou negativo, de modo que o endereço efetivo PC + N possa ser mais alto ou mais baixo do que o endereço no contador do programa.

Indexado. No *endereçamento indexado*, como no endereçamento relativo, a parte do endereço da instrução contém um número N, que pode ser positivo ou negativo. Contudo, a fim de usar endereçamento indexado, o computador deve ser equipado com um registrador especial (diferente do contador de programa) usado para permitir endereçamento indexado e chamado *registrador de índices* (I). O endereço do local da memória onde o operando está localizado é achado pela adição I + N. (No computador simples da Fig. 9.1-1 não temos nenhum registrador de índices, e não fizemos nenhuma provisão para endereçamento indexado.)

Indireto de registrador. Um computador que incorpora a facilidade de *endereçamento indireto de registrador* tem um registrador especial, muitas vezes chamado *registrador indicador* ou *apontador* (P). Este registrador P guarda o endereço do operando; isto é, ele aponta para o local da memória do operando. Uma instrução que invoca endereçamento indireto do registrador realmente não tem bits significativos em sua parte de endereço. Em vez disto, a instrução inteira é incluída nos bits atribuídos à parte de operação da instrução. Uma instrução típica que usa endereçamento indireto do registrador pode especificar "carregar o acumulador com o operando que está localizado na memória no endereço dado no registrador P".

Outros esquemas comuns para localização de fontes de operandos ou destinos de operandos, descritos como *modos de endereçamento*, incluem:

Imediato. No *endereçamento imediato* a parte de endereço da instrução não contém o endereço de um operando mas o próprio operando. Assim, a instrução "carregar acumulador *direto* com 37" significa carregar o acumulador com o conteúdo do local 37 da memória. Por outro lado, "carregar acumulador *imediato* com 37" significa carregar o acumulador com o número 37.

Inerente. Ordinariamente, um endereço que é parte de uma instrução refere-se a um local na memória. Quando uma instrução indica uma fonte ou um destino de alguma amostra de dados e nenhum endereço está especificamente em evidência porque nenhuma referência é feita a um local da memória, a instrução é descrita como tendo um *endereço inerente*. Por exemplo, na instrução "limpar o acumulador", os "dados" sendo movimentados constituem uma palavra em que todos os bits são 0s e o "endereço" do destino destes dados é o registrador do acumulador. Novamente, na instrução "mover o conteúdo do registrador R1 para o registrador R2", R1 e R2 são os "endereços" dos quais uma palavra é lida e nos quais a palavra é escrita.

9.15 PILHAS (STACKS) DE MEMÓRIA

Ao ter acesso à informação armazenada na memória já notamos que, quando possível, é útil ter um entendimento a respeito do endereço da próxima palavra a ser escrita ou lida. Assim, achamos muito útil armazenar instruções sucessivas nos sucessivos locais da memória, o que significava que tínhamos que tratar do endereço da próxima instrução apenas infreqüentemente. Exceto

quando forem encontradas instruções de salto (jump), a provisão do endereço da próxima instrução requer somente que um contador, o contador de programa, seja incrementado.

De maneira idêntica, é muito útil (quando for viável fazê-lo) armazenar dados numa facilidade da memória de tal maneira que novos endereços possam ser estabelecidos simplesmente incrementando (ou decrementando) um contador. Tal facilidade da memória é chamada pilha. Já consideramos as pilhas na Seção 6.16, onde notamos a razão do uso do nome pilha e o motivo pelo qual a escrita de uma palavra numa pilha é chamada colocação (*pushing*) e a leitura de uma palavra é chamada extração ou retirada (*popping*).

Qualquer conjunto de registradores pode ser usado como uma pilha. Usualmente, no entanto, a facilidade de armazenamento usada para implementar uma pilha é uma seção reservada de uma RAM. Tal seção de uma RAM, junto com o contador, está indicada na Fig. 9.15-1. Na programação, queremos ser capazes de lembrar qual seção da RAM está sendo reservada para a pilha de memória. Por conseguinte, como na Fig. 9.15-1, admitimos uma memória de capacidade de N palavras com endereços dos locais estendendo-se de 0 a $N-1$, e reservamos para a pilha o final da memória com endereços $N-1, N-2$ etc. O registrador do contador que é incrementado ou decrementado para ir de um local de pilha para o seguinte é chamado *apontador* ou *indicador de pilha* (SP). A terminologia é apropriada, já que o apontador de pilha guarda o endereço de, isto é, aponta para o local da pilha a que se tem acesso para leitura ou escrita.

A ordem em que os dados são colocados na pilha é, primeiro, dados A no local $N-1$, depois dados B no local $N-2$ etc. A pilha é mostrada na Fig. 9.15-1, depois de quatro palavras de dados A, B, C e D terem sido colocadas nela. Na leitura, isto é, extração ou retirada, primeiro extraímos D, depois C etc. Um modo pelo qual podemos implementar o processo de colocação de uma palavra na pilha é o seguinte:

1. Escrever a palavra no local da memória especificado pelo indicador de pilha.
2. Depois decrementar o indicador de pilha.

Tendo colocado uma palavra na pilha, suponhamos que queremos extrair esta mesma palavra. O procedimento é como se segue:

1. Incrementar o apontador de pilha.
2. Em seguida, ler a palavra do local da memória especificado pelo apontador de pilha.

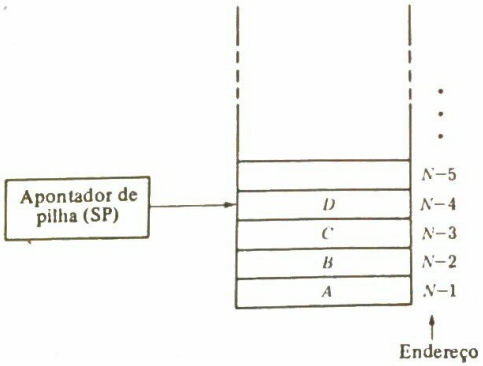


Fig. 9.15-1 Uma seção de uma RAM é reservada para uma pilha. Os endereços da pilha são fornecidos por um registrador (SP) apontador de pilhas.

No procedimento acima, a colocação foi feita primeiro escrevendo e depois decrementando. Não há razão pela qual não possamos modificar o procedimento de modo que a colocação seja efetuada primeiro decrementando e depois escrevendo. Se este procedimento inverso for empregado, naturalmente o procedimento da extração também deve ser invertido; isto é, devemos primeiro ler e depois incrementar. Suponhamos que usando este procedimento invertido queremos colocar no local $N-1$ da memória. Depois devemos primeiro ajustar o indicador de pilha em N de modo que a operação de decremento coloque o indicador de pilha, isto é, o contador, em $N-1$. Se o contador do indicador de pilha tiver um módulo N, então a colocação do contador em N realmente ajusta o contador em zero; isto é, 00 ... 00, decrementado de 1, torna-se 11 ... 11.

A RAM da Fig. 9.15-1 já é parte de nosso computador na Fig. 9.1-1. Acrescentamos agora um apontador de pilha, mas dificilmente necessitamos nos alongar sobre o ponto de que este apontador de pilha acrescentado não faz por si próprio uma pilha disponível para um computador. Para acrescentar tal facilidade de pilha, é necessário acrescentar ao repertório de instruções o que o controlador do sistema é capaz de executar. Deve haver uma instrução que carregará no apontador de pilha o endereço em que nos propomos a iniciar a pilha. Deve haver uma instrução que carregue o endereço do apontador de pilha no registrador de endereços da memória. Deve haver uma instrução que colocará, isto é, incrementará SP e escreverá na memória, e também uma instrução que retirará, isto é, decrementará SP e lerá da memória. Alguns computadores têm instruções que efetuarão inúmeras operações de colocar ou retirar, em seqüência, todas em resposta a esta única instrução.

Não é com freqüência que acontece de um programa chamar sub-rotinas que estejam encaixadas umas nas outras; isto é, uma sub-rotina, seja a sub-rotina SRA, é solicitada. Dentro da sub-rotina SRA torna-se necessário chamar a sub-rotina SRB. Antes da sub-rotina SRB ter sido completada, a sub-rotina SRC deve ser chamada, e assim por diante. Lembremos (Seção 9.2) que, sempre que uma sub-rotina for chamada, devemos guardar o endereço da instrução a que devemos retornar depois da sub-rotina ter sido completada. Graças à sua característica de espera (*last in-first out* = = LIFO), uma pilha torna um arranjo ideal para armazenamento de endereço de retorno quando as sub-rotinas estiverem encaixadas. Como uma última operação antes de deixar o programa principal para sub-rotina SRA, colocamos na pilha o endereço de retorno ADDR(MAIN) do programa principal. Ao deixar a sub-rotina SRA para a sub-rotina SRB, colocamos na pilha o endereço de retorno ADDR(SRA) da sub-rotina SRA, e assim por diante. Daí, os endereços de retorno são colocados na pilha na ordem ADDR(MAIN), ADDR(SRA), ADDR(SRB) etc. O último endereço colocado na pilha é o primeiro endereço requerido quando a última sub-rotina estiver concluída, e assim por diante. Portanto, o processo de retornar finalmente ao programa principal é uma questão de retirada da pilha para tornar disponíveis os endereços precisamente na ordem oposta àquela em que eles foram colocados na pilha.

Podemos igualmente achar que, quando uma sub-rotina é chamada, necessitamos poupar não somente o endereço de retorno para o programa principal, mas também os dados no acumulador, o registrador de estados e outros registradores de modo que o programa principal possa continuar depois da sub-rotina ter sido completada. A pilha também é usada eficientemente neste armazenamento temporário de dados.

CAPÍTULO 10

MICROPROCESSADORES

10.1 INTRODUÇÃO

Usamos o termo *computador de grande porte* (*large computer*) para fazer referência aos computadores cujas dimensões físicas se estendem de alguns quadros de relés de equipamento até muitas salas cheias de equipamentos e cujos custos variam de dezenas de milhares a centenas de milhares de dólares. Os minicomputadores são menores e menos dispendiosos, custando de milhares a dezenas de milhares de dólares e tendo dimensões físicas que variam do tamanho de uma cesta de pães até o tamanho de um quadro de relés. Os computadores de grande porte são usados para resolver uma ampla faixa de complexos problemas científicos ou de engenharia ou para manipular e processar grandes quantidades de dados. Os minicomputadores muitas vezes são usados em conexão com algum propósito simples onde o computador de grande porte não é nem necessário nem justificável.

Os computadores de grande porte são fabricados por um número relativamente pequeno de empresas, tal como a IBM. A maioria dos engenheiros acha que seu único encontro com estas grandes máquinas consiste em ter, de vez em quando, que escrever um programa para o computador em alguma linguagem de programação conveniente, tal como Fortran. Por conseguinte, dada a importância do assunto, o projeto de computadores de grande porte é algo a respeito de que nem todo engenheiro necessita estar amplamente informado. Uma situação semelhante prevalece em conexão com os minicomputadores. Os minicomputadores geralmente são comprados como unidades completas dos fabricantes e a tarefa do usuário consiste simplesmente em aprender como programá-los e usá-los.

Por outro lado, uma percentagem muito grande de engenheiros achará inevitavelmente necessário envolver-se no projeto de pequenos sistemas digitais que incorporam alguma modesta facilidade computacional. Tais pequenos computadores digitais, isto é, os *microcomputadores*,

acham uma gama de aplicação muito ampla. Numa loja de venda a varejo, um vendedor coloca um item numa balança e introduz o preço por libra num pequeno teclado. Um microcomputador, incorporado à balança, calcula o produto do peso pelo preço por libra e apresenta uma indicação visual digital do preço total. No equipamento eletrônico de teste, tal como osciloscópios, voltímetros digitais, sintetizadores de freqüência etc., microcomputadores incorporados aos instrumentos proporcionam muitas funções úteis, por exemplo, selecionando limites de variação e de amplitudes, controlando exposições e recalibrando. Os microcomputadores são usados em itens do consumidor como sofisticadas câmaras e jogos de TV. A indústria de automóveis está usando microcomputadores para controlar o fornecimento de combustível e a regulação do centelhamento para assegurar melhor rendimento da gasolina por percurso quilométrico. Os grandes veículos, por exemplo, os aviões e os caminhões com reboque, são equipados com freios antiderrapantes operados por computador. Nos aeroportos e em outros terminais de transporte, os microcomputadores são usados na coleta de preço de passagens, fazendo troco, emitindo passagens e confirmando reservas. A perspectiva futura da faixa de aplicação dos microcomputadores é quase sem limites.

Conforme vimos, um computador rudimentar consiste nos seguintes itens: (1) uma memória, (2) uma unidade de controle, (3) uma unidade lógica e aritmética (ULA), e (4) inúmeros registradores. Já observamos no Cap. 6 que pastilhas de memória LSI há muito tempo são encontradas. Atualmente pastilhas de memória de escrita com 64 kilobits estão começando a aparecer. Desde alguns anos também têm sido encontradas pastilhas de LSI em grande quantidade, cada uma contendo os componentes restantes do computador, isto é, a unidade de controle, a ULA e inúmeros registradores. Uma tal pastilha é denominada *unidade de microprocessador* (MPU) ou *microprocessador*. Assim, uma pastilha de microprocessador e uma pastilha de memória constituem juntas um computador rudimentar.

A disponibilidade da pastilha de memória e do microprocessador torna agora viável projetar sistemas digitais um tanto sofisticados com hardware pouco dispendioso que, além disto, ocupa um pequeno volume. Por esta razão, é provável que qualquer pessoa que esteja envolvida com o projeto de sistemas digitais necessitará saber como efetuar projetos que incorporam microprocessadores. A situação é diferente da que prevalece com os computadores de grande e de médio portes, onde a maioria das pessoas necessita saber pouco mais do que como eles são programados.

Os microprocessadores encontram aplicação não somente onde for requerida alguma facilidade computacional modesta, mas também na realização de sistemas digitais de modo geral. Suponhamos, por exemplo, que nos propomos a construir algum sistema digital seqüencial cuja implementação requer de nós que incorporemos facilidade de contagem, de deslocamento de palavras, de incrementação, de combinação de palavras e números numérica e logicamente etc. Podemos, então, construir um sistema em que fornecemos: um contador físico para contar, registradores de deslocamento físicos para deslocar, somadores para somar e incrementar e estruturas de porta para efetuar a lógica requerida. Um tal sistema seria descrito como sendo implementado em *hardware*, o termo hardware referindo-se aos dispositivos físicos reais. Alternativamente, podemos tirar vantagem do fato de que uma pastilha de microprocessador (em conjunção com uma ou mais pastilhas, tal como uma memória) constitui um processador digital que pode ser programado para efetuar todas aquelas mesmas operações de contagem, deslocamento, incrementação, combinação de palavras lógica e numericamente etc. Assim, podemos estar inclinados a substituir o extenso hardware do primeiro método de implementação pelo esquema alternativo de usar muito menos hardware num sistema de microprocessadores. No entanto, o sistema de microprocessadores terá que ser usado em conjunto com um programa especialmente escrito que dirigirá o micropro-

cessador para executar qualquer algoritmo que seja requerido para efetuar o processamento digital. Um programa muitas vezes é denominado *software*. Daí, uma tal substituição de componentes físicos por um programa é descrita como *substituição de hardware por software*.

Neste capítulo e no seguinte estudaremos o microprocessador 8080 da Intel como representativo dos microprocessadores de modo geral. Comentaremos sucintamente, também, o microprocessador 6800 da Motorola. Atualmente, estes constituem as pastilhas de microprocessador mais amplamente conhecidas e usadas dentre outras.

10.2 ARQUITETURA DO PROGRAMADOR DE UM MICROPROCESSADOR

Já vimos como projetar as partes componentes de um microprocessador. Vimos como projetar registradores de armazenamento para armazenar palavras e números. Examinamos como o conteúdo armazenado de tais registradores pode ser transferido de registrador a registrador usando, se necessário, um único bus para todas as transferências. Consideramos como projetar registradores de processamento, isto é, registradores que não somente armazenam dados mas que podem também realizar operações lógicas e numéricas com seus conteúdos, deslocamento, incrementação, realização de uma operação AND ou de uma operação OR com os conteúdos de outros registradores etc. Vimos como projetar uma unidade lógica e aritmética (ULA) para realizar funções lógicas e aritméticas, se quisermos assim, num circuito combinacional em vez de num registrador de processamento. Finalmente, estudamos como projetar a unidade de controle, o sistema seqüencial que avança um processador digital através de uma série especificada de etapas. Todos estes componentes, junto com a compreensão que adquirimos quando os estudamos, estão envolvidos no microcomputador simples, isto é, microprocessador mais memória, apresentado na Fig. 9.1-1.

Deve ser observado que na Fig. 9.1-1 há um mínimo de detalhes. Os registradores são simplesmente representados por retângulos. Onde um registrador de processamento estiver envolvido, não há detalhes que indiquem como as portas são incorporadas ao registrador para permitir o processamento solicitado. Por exemplo, o contador do programa tem que ser capaz de aceitar uma entrada do bus e tem que ser capaz de incrementar. Dispomos do assunto desenhando um retângulo, rotulando-o "contador do programa" e fornecendo dois terminais de entrada. Um terminal é assinalado "PC + 1 → PC" (isto é, incrementar) e o segundo é assinalado "GPR(AD) → → PC" (isto é, carregar bits de endereço do registrador de propósitos gerais). Nossa atitude é que, no ponto onde o assunto foi introduzido em nossa discussão, foi satisfatório dizer que "todos sabem" como construir um tal registrador que responderá quando solicitado a uma ativação de um terminal de entrada ou outro. Uma situação idêntica aplica-se ao controlador. Aqui novamente temos simplesmente indicado um retângulo com linhas de comando de saída e uma conexão aos registradores de instrução. Admite-se que, uma vez que tenhamos mais ou menos arbitrariamente decidido quanto à palavra de código binário que deve ser usada para representar cada instrução, "todos sabem" como projetar um controlador para fornecer a seqüência correta de comandos em resposta a cada código de instrução de entrada. Afinal, projetamos e apresentamos na Fig. 8.11-1 um controlador que foi capaz de responder a quatro instruções. O controlador na Fig. 9.1-1 não é diferente em qualquer aspecto que envolve um princípio. Simplesmente é que o último controlador deve responder a mais instruções e fornecer mais saídas de comando. Na Seção 9.12, começamos de fato um projeto do controlador maior, mas ele foi empreendido para ilustrar o princípio da microprogramação e não porque tivéssemos qualquer interesse especial no

projeto do controlador. Para ilustrar ainda nosso ponto, lembremos que no Cap. 4 consideramos como projetar contadores com módulos diferentes de 2^n. Ilustramos o procedimento projetando um contador de módulo 3, um contador de módulo 5, e outros contadores de módulos pequenos. Poderíamos, naturalmente, ter usado como exemplo um contador de módulo 137, mas, evidentemente, ele não teria servido para nenhum propósito útil. Nossa explicação (ou desculpa) por não fornecer projetos detalhados dos componentes na Fig. 9.1-1 é que os princípios de projeto são conhecidos e que, por causa da relativa complexidade, isto é, mais ou menos a mesma já encontrada em sistemas menores, tal projeto detalhado seria de redação muito longa, tedioso e improdutivo.

Apesar de todas nossas justificativas, poderíamos, entretanto, ter devotado mais atenção ao projeto de registradores de processamento e de controladores de grande porte se os engenheiros fossem freqüentemente chamados a projetar tais componentes. No decorrer de um outro tipo de análise poderíamos ter chegado através de atalhos e procedimentos alternativos com vantagens nas diferentes situações e, também, ter desenvolvido discernimento intuitivo do tipo que não vem sempre com a familiaridade e experiência. Mas o fato é que a disponibilidade de pastilhas de microprocessador (e de outras pastilhas de suporte) geralmente torna óbvia a necessidade de projetar qualquer controlador elaborado ou de projetar processador. Nos dias de hoje, quando alguém tiver que montar um sistema digital moderadamente sofisticado, geralmente o método mais frutífero consiste em evitar projetar o requisito hardware e implementar o sistema com um microprocessador e software.

Voltando agora à questão das pastilhas de microprocessador, notamos que elas são dispositivos um tanto elaborados nos quais foram integrados cerca de 5 000 componentes eletrônicos organizados em centenas ou mesmo milhares de portas. Estas portas são depois organizadas em registradores, controladores etc. Por exemplo, não podemos deixar de nos impressionar com fato de que o 6800 incorporou a ele a capacidade de executar 72 instruções distintas. A maioria destas instruções pode ser associada a mais do que um modo de endereçamento. Revela-se que então o número total de instruções é 197. O 8080 tem um comparável repertório de instruções. Podemos bem imaginar que o projeto destes e de microprocessadores similares ocupou a atenção de muitas pessoas durante um longo tempo.

Propomo-nos agora a começar um exame da arquitetura e do uso de nossos microprocessadores representativos. Podemos novamente estar inclinados a investigar os detalhes do conjunto de circuitos do microprocessador através das portas e flip-flops individuais. Mesmo se tais detalhes fossem fornecidos pelo fabricante (geralmente não é o caso), tal curso não seria produtivo. Poderia ser essencial a um pequeno e especializado grupo de engenheiros que profissionalmente projeta microprocessadores, mas, para engenheiros de modo geral, por mais interessante que um estudo detalhado possa ser, as recompensas não seriam proporcionais ao esforço – mesmo porque, como acontece, a falta de um estudo detalhado não impede necessariamente o usuário de fazer uso total e efetivo do dispositivo. No que se segue dirigir-nos-emos aos assuntos que os engenheiros geralmente necessitam saber, e não aos assuntos que se enquadram na competência do especialista.

Na Fig. 10.2-1 desenhamos um diagrama do "programador" do microprocessador 8080; isto é, indicamos apenas aqueles componentes da unidade que serão consultados explicitamente na escrita de um programa. Naturalmente, o 8080 tem uma ULA e uma unidade de controle, mas nós não o indicamos. O microprocessador também possui registradores adicionais àqueles indicados, mas estes registradores adicionais são "transparentes" para o programador. Não há instruções que façam referência explícita a eles e, portanto, preferimos não os mostrar. Também mostrada na figura está uma RAM externa, que reterá os dados e as instruções.

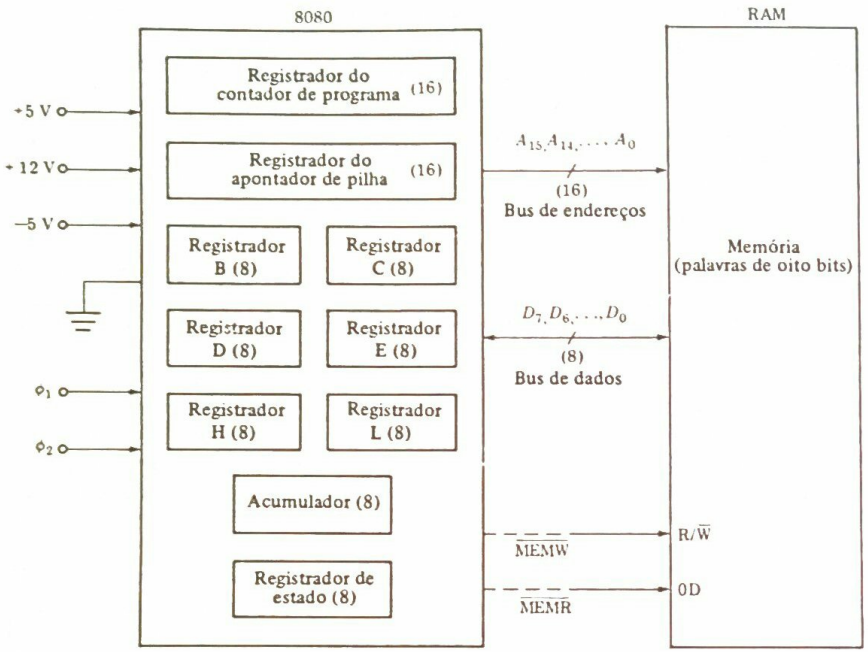


Fig. 10.2-1 Uma vista do programador do microprocessador 8080.

Voltando agora aos registradores que são mostrados, achamos que há um registrador do contador de programa com dezesseis bits de comprimento, conforme indicado pelo número entre parênteses. Há um registrador apontador de pilha, que também tem dezesseis bits de comprimento. Uma vez que o contador do programa e o apontador de pilha se destinam a guardar endereços da memória, achamos, muito coerentemente, que o 8080 tem um bus de endereços de dezesseis bits com linhas rotuladas $A_{15}, A_{14}, \ldots, A_0$ que devem ser conectadas aos pinos de entrada de endereço da pastilha de memória. As dezesseis linhas do bus de endereços possibilitam uma memória da ordem de $2^{16} = 65\,536$ palavras, mas também pode ser usada uma memória menos extensa. Uma vez que o 8080 se destina a ser usado com uma memória em que as palavras individuais têm oito bits de comprimento, achamos que os outros registradores são correspondentemente do mesmo comprimento de oito bits. Há um bus bidirecional de oito bits com linhas rotuladas $D_7, D_6, \ldots, D_0$ que transportam dados da pastilha do microprocessador para a memória externa e da memória para a pastilha. Os registradores B e C, D e E, e H e L são registradores de propósitos gerais; eles podem ser usados individualmente ou, conforme veremos, aos pares, de modo que B e C, D e E ou H e L possam ser usados como um único registrador de dezesseis bits. O registrador do acumulador, como o acumulador de nossos primitivos processadores mais simples, é especial quanto a que ele pode ser usado para proporcionar uma certa manipulação dos dados que ele retém. Assim, o acumulador possibilita deslocamento, incrementação etc. Ele também geralmente retém um dos

dois operandos envolvidos numa operação aritmética ou lógica e, depois, retém o resultado da operação.

O registrador de estado é diferente dos outros registradores (ele é chamado registrador por tolerância, afinal, necessitamos de um nome para ele). Ele consiste em oito flip-flops, três dos quais estão permanentemente no estado set e, por conseguinte, em princípio, não servem para nenhum propósito. Os outros cinco biestáveis são flip-flops flag, cujos estados são observados numa base individual e muitas vezes não sustentam nenhuma ligação uns com os outros.

A linha $\overline{\text{MEMW}}$ ($\overline{\text{MEMORY WRITE}}$) e as linhas $\overline{\text{MEMR}}$ ($\overline{\text{MEMORY READ}}$) da MPU para a memória determinam se deve haver uma transferência de informação entre MPU e a memória e se a memória deve ser lida ou escrita. Quando não deve haver nenhuma leitura nem uma escrita, tanto $\overline{\text{MEMW}}$ quanto $\overline{\text{MEMR}}$ estão em lógica 1. Nestas condições, a entrada de memória R/$\overline{\text{W}}$ estando em lógica 1, a memória é ativada para ser lida, mas a entrada OD (impossibilitar saída) está em lógica 1, as conexões de saída de três-estados da memória para o cabo de dados estão no estado de alta impedância e a memória está, desta maneira, realmente desconectada do cabo de dados. Se deve ocorrer uma operação de escrita, a $\overline{\text{MEMW}}$ é trazida para lógica 0. O bus de dados é sempre conectado às conexões de entrada da memória, e, mesmo com $\overline{\text{MEMR}}$ em lógica 1, uma operação de escrita ocorrerá quando MEMW for para lógica 0. Naturalmente, antes que possa ser permitida uma operação de leitura ou escrita, a MPU deve estabelecer uma memória válida no bus de endereços.

Na Fig. 10.2-1 convencionamos a utilização de linhas tracejadas nas partes das linhas $\overline{\text{MEMW}}$ e $\overline{\text{MEMR}}$ que se conectam à MPU. Pretendemos deste modo indicar que estas linhas não se originam na realidade na MPU. Como veremos mais tarde, elas se originam numa unidade auxiliar que, por sua vez, é controlada pelos sinais da MPU. No entanto, a maneira pela qual procedemos aqui nos permite simplificar as interligações sem violar qualquer princípio. Colocaremos a questão corretamente mais tarde.

Indicamos também na Fig. 10.2-1 que o 8080 requer três tensões de alimentação, aplicadas, como indicado, entre os terminais de entrada de voltagem e a terra. A unidade também requer um sinal de entrada de relógio. Realmente o fabricante especifica que a entrada de relógio consiste em duas partes síncronas, ϕ_1 e ϕ_2. Alguns detalhes concernentes aos sinais de relógio necessários são dados no próximo capítulo. A unidade 8080A, um tanto mais rápida do que a 8080, aceitará uma freqüência de relógio até 2 MHz. Uma vez que os elementos de armazenamento na unidade são do tipo dinâmico, que requerem reabastecimento periódico, há também uma freqüência de relógio mínima de cerca de 0,5 MHz.

Notamos ainda na Fig. 10.2-1 que mostramos dezesseis linhas de endereço, oito linhas de dados, quatro linhas de fontes de alimentação (incluindo terra) e duas linhas de relógio, num total de trinta. Uma vez que a pastilha do 8080 tem quarenta pinos, dez pinos não são atualmente considerados. Prosseguiremos neste assunto no próximo capítulo.

10.3 INSTRUÇÕES DE UM, DOIS E DE TRÊS BYTES NO 8080

Em nosso computador simples da Fig. 9.1-1, providenciamos para que cada instrução armazenada na memória pudesse ser armazenada nos doze bits de um único local da memória. No 8080, que é muito mais sofisticado, há somente oito bits num local da memória. Achamos, portanto, não surpreendentemente, que muitas instruções do 8080 necessitam de dois locais da memória ou

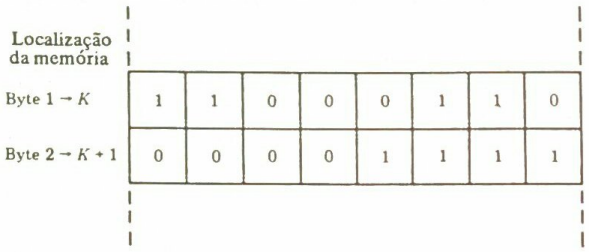


Fig. 10.3-1 A instrução de dois bytes "adicionar 15 (decimal) ao conteúdo do acumulador".

mesmo de três locais para armazenar uma instrução. Uma vez que oito bits constituem um byte, achamos, por conseguinte, que o 8080 tem instruções de um, dois e três bytes.

Um exemplo de uma instrução de um byte é a instrução para INCREMENTAR O ACUMULADOR. O mnemônico desta instrução é INR A, e o código binário para ele ocupa um local da memória. Um exemplo de uma instrução de dois bytes é a instrução ADD (ao acumulador) IMMEDIATE, ou ADI data. A parte de operação da instrução ocupa um byte, e os dados ocupam o byte imediatamente posterior. Para especificarmos, notamos que o fabricante nos diz que o código da instrução de ADI é 11000110; suponhamos que o número a ser adicionado é o decimal 15 = = 00001111. Então a instrução inteira apareceria em dois locais sucessivos da memória, conforme indicado na Fig. 10.3-1. Finalmente, como exemplo de uma instrução de três bytes, temos a instrução LOAD THE ACCUMULATOR (CARREGAR O ACUMULADOR) (com um operando da memória). Esta instrução deve ter incluído nela o endereço de dezesseis bits do operando. O mnemônico aqui é LDA. A parte operacional da instrução requer um byte. O endereço requer dezesseis bits e, por conseguinte, usa dois bytes. O fabricante nos diz que o código desta instrução é 00111010. O endereço tem oito bits $A_{15}, A_{14}, \ldots, A_8$ de ordem *mais alta* e oito bits $A_7, A_6, \ldots, A_0$ de ordem *mais baixa*. Suponhamos então que o endereço seria

$$A_{15}, A_{14}, \ldots, A_8; A_7 A_6, \ldots, A_0 = 11110000\ 10101010$$

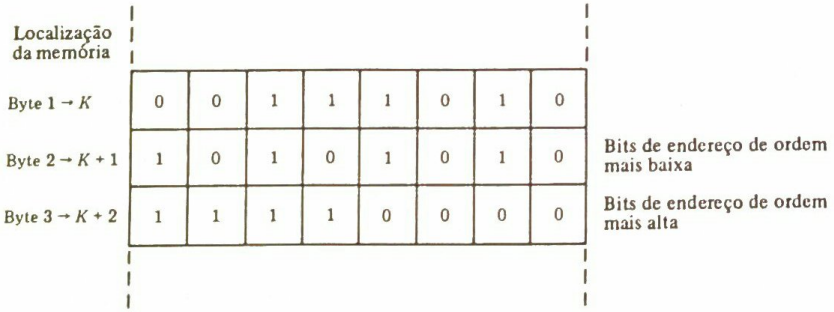


Fig. 10.3-2 A instrução de três bytes "carregar acumulador com conteúdo de local de memória $A_{15}, \ldots, A_0 = 1111\ 0000\ 1010\ 1010$".

Então a instrução completa apareceria em três locais sucessivos da memória, conforme mostrado na Fig. 10.3-2. Notar que no 8080 os bits de endereço de ordem mais baixa aparecem no byte 2 seguidos pelos bits de ordem mais alta no byte 3. (Em algumas MPUs a ordem é invertida, os bits de ordem mais alta antes dos de ordem mais baixa.)

Em nosso computador mais simples e primitivo, cada instrução ocupava não mais do que um local da memória. Neste caso, o controlador buscaria uma instrução, colocaria a parte operacional no registrador de instruções e o endereço (se houvesse) num registrador de endereços, e começaria a executar a instrução imediatamente. No presente caso de múltiplos bytes, o primeiro byte dirá ao controlador se a instrução tem mais do que um byte, e, se for assim, a execução da instrução será adiada até que todos os bytes tenham sido buscados.

10.4 INSTRUÇÕES PARA MOVIMENTO DE DADOS

Nesta e nas seções que se sucedem descreveremos as instruções que constituem o repertório do 8080. Aqui consideramos as instruções que transferem o conteúdo de um registrador para outros registradores ou que efetuam uma transferência de dados entre registradores e memória.

O mnemônico da linguagem do montador para o movimento de dados é MOV. A instrução que transfere o conteúdo de um registrador r2 para um registrador r1 é escrita

MOV r1, r2

É importante ter em mente que esta instrução MOV simplesmente copia em r1 o conteúdo de r2 e que o conteúdo de r2 *permanece inalterado*. Assim, após MOV r1, r2, r1 e r2 têm os mesmos conteúdos, isto é, aqueles presentes em r2 antes da execução da instrução. Naturalmente, o conteúdo anterior de r1 está perdido. Os registradores de fonte podem ser quaisquer dos registradores A, B, C, D, E, H ou L, e os registradores de destino também podem ser quaisquer destes mesmos registradores. Por exemplo, MOV L, B copiará o conteúdo do registrador B no registrador L. O 8080 aceitará como válida até mesmo uma instrução MOV na qual fonte e destino são os mesmos, tal como MOV C, C. Esta instrução, naturalmente, não fará nada.

A instrução MOV também pode ser usada para mover dados para ou da memória; no entanto, já que a memória não é um simples registrador mas pode ser da ordem de 2^{16} registradores, primeiro é necessário estabelecer um endereço na memória a partir do qual os dados devem ser lidos ou no qual os dados devem ser escritos. Este endereço é fornecido pelos conteúdos dos registradores H e L. O registrador H retém os bits altos de endereço, isto é, os bits mais significativos $A_{15}, A_{14}, \ldots, A_8$, e o registrador L retém os baixos, isto é, os bits menos significativos. Assim, a instrução

MOV A, M; MOV M, A

move para o registrador A o conteúdo de um local da memória ou para a memória o conteúdo de A, sendo o local selecionado da memória estabelecido pelos dezesseis bits nos registradores H e L. As instruções MOV H, M: MOV L, M; MOV M, H; MOV M, L são todas permitidas. A instrução MOV M, M revela-se não ser inútil, mas faz com que o microprocessador pare.

Um segundo tipo de instrução MOVE é a MOVE IMMEDIATE, instrução de dois bytes com o mnemônico MVI. A palavra "immediate" significa que a instrução fornece os próprios dados em vez de fornecer o endereço de um local da memória onde os dados devem ser achados. Tipi-

camente, se quisermos colocar na memória uma instrução que moverá para dentro do registrador C os dados 11001100, devemos escrever na memória, em dois locais sucessivos, as entradas

MVI C
11001100

Mais genericamente, escrevemos a instrução

MVI r
$\langle B_2 \rangle$

em que r é o registrador de *destino* dos dados e $\langle B_2 \rangle$ significa o segundo byte da instrução de dois bytes. O registrador de destino pode ser qualquer um dos sete registradores de propósitos gerais A, B, C, D, H ou L. A instrução

MVI M
$\langle B_2 \rangle$

moverá o byte B_2 para dentro da memória, sendo o local na memória dado pelo endereço retido no par de registradores H e L.

10.5 INSTRUÇÕES QUE ENVOLVEM DIRETAMENTE O ACUMULADOR

De todos os registradores de propósitos gerais, o acumulador é o mais versátil. Não é, portanto, surpreendente que haja inúmeras instruções concernentes ao movimento de dados para dentro do acumulador ou do acumulador para algum outro repositório. Assim, temos a instrução LOAD ACCUMULATOR DIRECT. Esta é uma instrução de três bytes e é escrita

LDA
$\langle B_2 \rangle$
$\langle B_3 \rangle$

em três locais sucessivos da memória. LDA é a parte operacional da instrução e B_2 e B_3 são, respectivamente, os bytes de endereço da memória de baixa e de alta ordem. A instrução significa que deve ser carregado no acumulador o conteúdo do local B_3B_2 da memória.

O inverso da instrução de carregar acumulador é a instrução STORE ACCUMULATOR DIRECT, escrita

STA
$\langle B_2 \rangle$
$\langle B_3 \rangle$

Esta instrução transfere para o local B_3B_2 da memória o conteúdo do acumulador. Como usual, o conteúdo do acumulador permanece inalterado.

As próximas duas instruções são similares às instruções LDA e STA, respectivamente, e envolvem a noção de um par de registradores. De tais pares, há três possíveis: o par B e C, o par D e E e o par H e L. Estes pares são invocados quando necessitarmos, efetivamente, de um registrador de dezesseis bits para reter um endereço da memória. Já vimos como H e L foram usados como par. No par B e C, B sempre retém os bits de endereço de ordem alta e C retém os bits de ordem baixa. Similarmente, no par D e E, D retém os bits de ordem alta e E os bits de ordem baixa. Quando for pretendido o par B e C, nós citaremos o par simplesmente como registrador B; quando for pretendido o par D e E, citaremos registrador D; e quando o par H e L for pretendido, citaremos registrador H. Estará claro do contexto se um único registrador ou um par de registradores é pretendido. Se não estiver claro, citaremos especificamente *par de registradores* B etc.

A instrução LOAD ACCUMULATOR INDIRECT

LDAX rp

carrega o acumulador com o conteúdo do local da memória cujo endereço é dado pelo par rp de registradores, mas nesta instrução rp deve ser ou rp = B ou rp = D. O par rp = H não é permitido. A instrução inversa de LDAX rp é a instrução STORE ACCUMULATOR DIRECT

STAX rp

que armazena os dados do acumulador nos locais da memória especificados no par rp de registradores. Novamente rp = H não é permitido.

Podemos notar que nas instruções LDAX rp e STAX rp não há necessidade de rp = H; portanto, se quisermos carregar ou armazenar o acumulador e quisermos que o endereço da memória seja o par H e L, podemos usar as instrucões MOV A, M e MOV M, A.

As instruções adicionais de movimento de dados no repertório do 8080 são tabeladas na Tab. 10.5-1, junto com aquelas já discutidas. A primeira coluna dá o nome descritivo da instrução. A segunda coluna dá o mnemônico da linguagem do montador e indica se a instrução é uma instrução de um, dois ou três bytes. Quando uma instrução requer um segundo byte $\langle B_2 \rangle$ ou mesmo um terceiro byte $\langle B_3 \rangle$, estes bytes são escritos na mesma linha que a parte operacional da instrução. Está entendido, naturalmente, que três bytes ocupam três locais sucessivos da memória e devem mais adequadamente aparecer um sob o outro. A microoperação ou microoperações reais efetuadas pela instrução estão especificadas na terceira coluna, onde o símbolo M significa a memória, M [H, L] representa o local da memória dado pelo conteúdo do par H de registradores e M [$\langle B_3 \rangle$ $\langle B_2 \rangle$] é um local da memória especificado pelo endereço B_3B_2. Também escrevemos com palavras precisamente o que faz cada instrução.

Não deve ser recomendado que o leitor procure memorizar agora esta lista de instruções. Os leitores que eventualmente encontrarem-se fazendo uso extensivo do 8080 inevitavelmente tornar-se-ão cada vez mais familiarizados com estas e as outras instruções do 8080, ainda a serem consideradas. Neste caso, os usuários eventualmente achar-se-ão bem à vontade com as instruções, tanto quanto nós nos achamos à vontade com qualquer nova linguagem ou procedimento complicado a que nos aplicamos diligentemente. Mas mesmo para o usuário casual sugerimos que vale o esforço de desenvolver uma certa familiaridade com estas e outras instruções com base em que elas são bastante características das instruções a serem encontradas genericamente em outros microprocessadores. O tempo dedicado à familiarização com um microprocessador é muito útil, não importa que microprocessador seja finalmente escolhido.

Tabela 10.5-1 As instruções do 8080 que efetuam transferências de palavras de registrador a registrador e entre registradores e locais da memória

MOVE REGISTER	MOV r1, r2	r2 → r1
Mover conteúdo do registrador r2 para o registrador r1, deixando conteúdo do registrador r2 inalterado		
MOVE FROM MEMORY	MOV r, M	M[H, L] → r
Mover para registrador r conteúdo do local da memória cujo endereço é mantido no par H de registradores		
MOVE TO MEMORY	MOV M, r	r → M[H, L]
Mover conteúdo do registrador r para local da memória cujo endereço é mantido no par H de registradores		
MOVE IMMEDIATE	MVI r, ⟨B_2⟩	⟨B_2⟩ → r
Mover byte ⟨B_2⟩ para dentro do registrador r de destino		
MOVE TO MEMORY IMMEDIATE	MVI M, ⟨B_2⟩	⟨B_2⟩ → M[H, L]
Mover byte ⟨B_2⟩ para o local da memória cujo endereço é mantido no par H de registradores		
LOAD REGISTER PAIR IMMEDIATE	LXI rp, ⟨B_2⟩⟨B_3⟩	⟨B_2⟩ → rp[low] ⟨B_3⟩ → rp[high]
Mover byte ⟨B_2⟩ para a parte de ordem baixa do par rp de registradores e mover byte ⟨B_3⟩ para a parte de ordem alta do par rp de registradores		
LOAD ACCUMULATOR DIRECT	LDA ⟨B_2⟩⟨B_3⟩	M[⟨B_3⟩⟨B_2⟩] → A
Mover para Acc o conteúdo do local da memória cujo endereço é dado por ⟨B_3⟩ (bits de endereço de ordem alta) e ⟨B_2⟩ (bits de endereço de ordem baixa)		
STORE ACCUMULATOR DIRECT	STA ⟨B_2⟩⟨B_3⟩	A → M[⟨B_3⟩⟨B_2⟩]
Mover conteúdo de Acc para o local da memória cujo endereço é dado por ⟨B_3⟩ (bits de endereço de ordem alta) e ⟨B_2⟩ (bits de endereço de ordem baixa)		
LOAD H AND L DIRECT	LHLD ⟨B_2⟩⟨B_3⟩	M[⟨B_3⟩⟨B_2⟩] → L M[⟨B_3⟩⟨B_2⟩ + 1] → H
Mover para registrador L o conteúdo do local da memória cujo endereço é ⟨B_3⟩⟨B_2⟩. Mover para registrador H conteúdo do local da memória seguinte		
STORE H AND L DIRECT	SHLD ⟨B_2⟩⟨B_3⟩	L → M[⟨B_3⟩⟨B_2⟩] H → M[⟨B_3⟩⟨B_2⟩ + 1]
Mover conteúdo do registrador L para local da memória cujo endereço é ⟨B_3⟩⟨B_2⟩. Mover conteúdo do registrador H para o local da memória seguinte		
LOAD ACCUMULATOR INDIRECT	LDAX rp	M[rp] → A (rp ≠ H)
Mover para Acc o conteúdo da memoria no endereço mantido no par rp de registradores; o par rp não deve ser o par H		
STORE ACCUMULATOR INDIRECT	STAX rp	A → M[rp] (rp ≠ H)
Mover conteúdo de Acc para local da memória cujo endereço é dado pelo par rp de registradores; o par rp não deve ser o par H		
EXCHANGE H AND L WITH D AND E	XCHG	H ⟷ D L ⟷ E
Trocar conteúdos dos registradores H e D e também conteúdos dos registradores L e E		

10.6 INSTRUÇÕES ARITMÉTICAS

Consideremos agora um segundo tipo de instrução do 8080, denominado *instruções aritméticas*. Um propósito destas instruções é somar e subtrair os conteúdos dos registradores ou somar e subtrair os conteúdos dos registradores e locais da memória. Quando uma adição ou subtração estiver envolvida, devemos especificar a fonte de cada um dos operandos e o lugar onde o resultado deve ser armazenado. Daí, uma tal operação, envolvendo três endereços da memória ou identificações do registrador, pode requerer um grande número de bytes, o que não é econômico. Para contornar esta dificuldade, constitui prática geral em microprocessadores (e em outras pequenas instalações de computação) ter entendido que *um dos operandos deve ser achado no acumulador e que o resultado deve ser colocado no acumulador*. Já encontramos esta idéia no Cap. 9, e achamos, igualmente, que este arranjo é quase sempre empregado no 8080. Um segundo propósito das instruções aritméticas é incrementar ou decrementar registradores ou o conteúdo dos locais da memória.

Quando a UCP responde às instruções que solicitam uma operação aritmética (adição, subtração, incremento e decremento), a UCP, naturalmente, usa sua unidade lógica e aritmética (ULA). A ULA é também usada nas operações lógicas. Quando a ULA é envolvida, freqüentemente acontece que, após a operação da ULA, a seqüência de instruções a ser seguida depende do resultado na saída da ULA. Por esta razão, o 8080 fornece inúmeros flags (bandeiras), cinco ao todo, que juntos constituem o registrador de estado da Fig. 10.2-1. Estes flags monitoram a saída da ULA e cada um serve para armazenar um bit de saída da ULA que pode ser necessário em outro desenvolvimento do programa. Quatro destes flags estão listados na Tab. 10.6-1 junto com seus significados. O flag Z armazena um registro de se o resultado é zero ou não. O flag S registra se o bit mais significativo é 0 ou 1. Se a saída da ULA for um número, e se estivermos usando a representação em complemento de dois dos números negativos, o flag S (sinal) registra se o número é positivo ou negativo. O flag CY registra se há um transporte vai-um ou um empréstimo do bit de ordem mais alta, como pode ocorrer numa operação de adição ou de subtração. O flag P de paridade registra se a paridade do resultado é ímpar ou par. Este flag não é, geralmente, tão útil quanto os flags Z, S e CY, mas tem sua função em certas aplicações quando o microprocessador for usado como um componente num sistema de transmissão de dados. A função do quinto flag será descrita mais tarde.

Tabela 10.6-1 Quatro dos flags do 8080

Nome do flag	Símbolo	Significado do registro
Zero	Z	= 0 resultado não é zero = 1 resultado é zero
Sinal	S	= 0 resultado é positivo = 1 resultado é negativo
Transporte (*Carry*)	CY	= 0 não ocorreu transporte nem empréstimo = 1 ocorreu um transporte ou empréstimo
Paridade	P	= 0 resultado tem paridade par = 1 resultado tem paridade ímpar

Em qualquer caso, quando a execução de uma operação aritmética tiver sido concluída, o registrador de estado imediatamente refletirá o estado do resultado na *saída da ULA*. Assim, por exemplo, se um número for somado ao conteúdo do acumulador, então, no instante em que o resultado for estabelecido no acumulador, o registrador de estado já saberá se houve um transporte, se o resultado é positivo, negativo ou zero e qual a paridade do resultado. É importante notar, contudo, que nem todas as instruções que envolvem a ULA afetam todos os flags. Se um flag particular não for afetado por uma instrução, este flag é deixado no estado em que ele estava no fim da execução da instrução anterior. No exame das instruções a serem consideradas abaixo, às vezes acharemos que alguns dos flags afetados dificilmente podem servir a um propósito útil. Podemos racionalmente admitir que esta característica foi permitida simplesmente porque simplificava o projeto do controlador. Afinal, podemos sempre ignorar um flag se preferirmos fazer assim.

Se estivéssemos limitados a usar números de oito bits para acomodar registradores e palavras de memória do tamanho de um byte, não seríamos capazes de obter precisão aritmética, mas a dificuldade é fácil de contornar. Por exemplo, se quisermos trabalhar com número de dezesseis bits, podemos simplesmente dividir os dezesseis bits em dois bytes e guardar o número de dezesseis bits em dois locais da memória. Suponhamos, então, que queremos somar dois números N_1 e N_2 de dezesseis bits. N_1 tem os oito bits *mais* significativos N_1 (M) e os oito bits *menos* significativos N_1 (L). Similarmente, N_2 é composto das partes N_2 (M) e N_2 (L). Estes números podem agora ser somados como se segue. Primeiro, usamos uma instrução que soma N_1 (L) + N_2 (L). Providenciamos para que esta adição deixe no flip-flop flag de transporte um registro de se, nesta adição, há um transporte do bit mais significativo. A seguir, usamos uma instrução que forma a soma N_1 (M) + N_2 (M) + *conteúdo do flag de transporte*. Em conseqüência destas duas somas, realmente temos formada a soma $N_1 + N_2$. Suponhamos, ainda, que a instrução que executa a instrução "somar dois números de oito bits e somar a esta soma o conteúdo do flag de transporte" deixa no flag de transporte um registro de se esta adição, por sua vez, gera um transporte da posição do bit mais significativo. Então seremos capazes de estender a precisão de nossa aritmética ilimitadamente. Portanto, seremos capazes de somar números de três bytes (24 bits), números de quatro bytes (32 bits) etc.

Considerações idênticas aplicam-se ao processo da subtração. Necessitaremos de uma instrução que subtraia, isto é, forme $N_1 - N_2$, e de uma instrução que forme $N_1 - N_2$ – *conteúdo do flag de transporte*. Em conjunto, então, não estamos surpreendidos em achar no repertório das instruções do 8080 as instruções "somar", "somar com transporte", "subtrair", e "subtrair com empréstimo". Com estes comentários, estamos preparados para examinar algumas das instruções aritméticas.

As instruções de adição e subtração estão listadas na Tab. 10.6-2. As três primeiras fornecem adição *sem* transporte e diferem umas das outras na fonte do adendo que deve ser adicionado ao adendo no acumulador. No primeiro caso, o adendo é achado num dos registradores de propósitos gerais. No segundo caso, o operando está incluído na instrução (dois bytes). No terceiro caso, o adendo está na memória. Observemos, como no caso das instruções MOVE, que não há nenhuma instrução sozinha que nos permita especificar o endereço de um local da memória e somar o conteúdo da memória ao acumulador. Para executar esta operação devemos primeiro usar uma instrução tal como LOAD REGISTER PAIR IMMEDIATE para estabelecer o endereço da memória no par de registradores H e L. As três instruções a seguir são idênticas às três primeiras instruções, exceto que elas se somam no bit de transporte, se houver. As seis instruções seguintes são idênticas às seis primeiras, exceto que é solicitada subtração em vez de adição e um bit de

Tabela 10.6-2 Instruções Aritméticas do 8080*

ADD REGISTER	ADD r	A + r → A
Somar número no registrador r ao número em Acc e deixar soma em Acc		
ADD IMMEDIATE	ADI ⟨B_2⟩	A + ⟨B_2⟩ → A
Somar número B_2, que é o segundo byte da instrução, a Acc e deixar soma em Acc		
ADD MEMORY	ADD M	A + M[H, L] → A
Somar número na memória, no endereço dado pelos registradores H e L, ao número em Acc e deixar soma em Acc		
ADD REGISTER WITH CARRY	ADC r	A + r + CY → A
Somar a Acc o número do registrador r e somar também número no flag de transporte e deixar soma em Acc		
ADD IMMEDIATE WITH CARRY	ACI ⟨B_2⟩	A + ⟨B_2⟩ + CY → A
Somar a Acc o número B_2, que é o segundo byte da instrução, e somar também número no flag de transporte e deixar soma em Acc		
ADD MEMORY WITH CARRY	ADC M	A + M[H, L] + CY → A
Somar a Acc o número na memória cujo endereço está nos registradores H e L e somar também número no flag de transporte e deixar soma em Acc		
SUBTRACT REGISTER	SUB r	A – r → A
Subtrair número no registrador r do número em Acc e deixar diferença em Acc		
SUBTRACT IMMEDIATE	SUI ⟨B_2⟩	A – ⟨B_2⟩ → A
Subtrair número B_2, que é o segundo byte da instrução, do número em Acc e deixar diferença em Acc		
SUBTRACT MEMORY	SUB M	A – M[H, L] → A
Subtrair de Acc o número na memória cujo endereço está nos registradores H e L e deixar a diferença em Acc		
SBUTRACT REGISTER WITH BORROW	SBB r	A – r – CY → A
Subtrair de Acc o número no registrador r e subtrair também número no flag de transporte e deixar resultado em Acc		
SUBTRACT IMMEDIATE WITH BORROW	SBI ⟨B_2⟩	A – ⟨B_2⟩ – CY → A
Subtrair de Acc o número B_2, que é o segundo byte da instrução, e subtrair também número no flag de transporte e deixar resultado em Acc		
SUBTRACT MEMORY WITH BORROW	SBB M	A – M[H, L] – CY → A
Subtrair de Acc o número na memória cujo endereço está nos registradores H e L e também subtrair número no flag de transporte e deixar resultado em Acc		
DECIMAL ADJUST ACCUMULATOR	DAA	
Conteúdo de Acc é modificado como seria requerido se o conteúdo fosse resultado da adição de dois números BCD. O procedimento é: (1) Se o valor de quatro bits menos significativos em Acc for maior do que 9, ou se o flag AC for ajustado, 6 é adicionado a estes bits menos significativos. (2) Se, depois da etapa 1 estar completa, o valor dos bits mais significativos for maior do que 9, ou se o flag CY for ajustado, 6 é adicionado a estes bits mais significativos		
ADD REGISTER PAIR rp TO REGISTER PAIR H AND L	DAD rp	rp + H, L → H, L
O número de dezesseis bits no par rp de registradores é somado ao número no par H e L de registradores e a soma é deixada em H e L. O único flag afetado é CY. Se houver um transporte fora desta adição de dezesseis bits, CY é tornado set; do contrário CY é tornado reset.		

* *Todas* estas instruções aritméticas (exceto DAD rp) afetam *todos* os cinco flags.

empréstimo é subtraído em vez de um bit de transporte ser adicionado. As duas últimas instruções na Tab. 10.6-2 são incluídas porque elas também são instruções que envolvem aritmética, mas elas são um tanto especiais e são discutidas abaixo.

Consideremos agora a instrução (um tanto especial) seguinte-à-última na Tab. 10.6-2. Será lembrado que na Seção 5.14 sugerimos que às vezes poderia ser razoável fazer aritmética usando números BCD. Lá consideramos um modo pelo qual a adição de BCD podia ser efetuada e também apresentamos o hardware para efetuar a adição. O 8080 permite tal adição BCD pelo uso da instrução DECIMAL ADJUST ACCUMULATOR com mnemônico DAA. Aplicando-se o princípio de que há um mérito na substituição do software pelo hardware, o 8080 executa um pequeno programa em resposta à instrução DAA. Não especificamos as microoperações envolvidas, mas escrevemos em palavras precisamente o que a instrução faz. É necessário, contudo, acrescentar uma pequena peça de hardware na forma de um flip-flop flag adicional chamado *flag* AC *de transporte auxiliar*. Quando dois números de oito bits são somados em adição binária direta, o flag AC é ajustado (set) se houver um transporte do quarto bit, tal como o flag CY é ajustado (set) se houver um transporte do oitavo bit.

Suponhamos agora que temos um número N_1 de dois dígitos decimais (tal como podia ser representado nos oito bits binários) e um segundo número N_2 de dois dígitos decimais. Para somar estes números no 8080, primeiro somamos N_1 e N_2 como se eles fossem números binários diretos. A seguir invocamos a instrução DAA. Esta instrução examina os quatro bits menos significativos do resultado e o estado do flag AC. Se este número de quatro bits for maior do que 9, ou se AC estiver ajustado (set), 6 é somado a estes bits. A seguir, os quatro bits mais significativos são examinados. Esta segunda parte do procedimento deve esperar até depois da adição (se houver) de 6 aos bits menos significativos, uma vez que esta adição irá gerar um transporte se nenhum transporte já tiver sido gerado. Novamente, se os bits mais significativos tiverem um valor maior do que 9, ou se um transporte for gerado (o transporte aqui tornará set o flag CY), 6 é somado a estes bits. Como pode ser verificado (ver discussão na Seção 5.13), este procedimento DAA fornecerá a soma BCD corretamente. Tal como na adição binária direta, também aqui podemos usar o CY para permitir precisão aumentada estendendo o processo da adição a quatro dígitos decimais, ou a seis dígitos etc. Um programa em linguagem de montador (*assembler*) para somar N_1 e N_2, ambos números BCD, é como se segue:

Rótulo (*label*)	Conteúdo	Comentário
	SUB A	Subtrair A de A para limpar Acc
	ADI $\langle N_1 \rangle$	Somar o número N_1 a Acc
	ADI $\langle N_2 \rangle$	Somar o número N_2 a Acc
	DAA	Acc de ajuste decimal

Anteriormente, identificamos quatro flip-flops flags no registrador de estado e indicamos que havia também um quinto. O flag AC é o quinto. Em conjunto, então, os cinco flags no registrador de estado são Z (*zero*), S (*sign*), P (*parity*), CY (*carry*) e AC (*auxiliary carry*). *Todas* as instruções na Tab. 10.6-2, exceto uma, (DAD rp), afetam *todos* os cinco flags.

Finalmente, notamos a última instrução que envolve adição, que é diferente das outras pois não envolve o acumulador. Esta instrução DAD rp processa dezesseis bits (dois bytes) de uma vez e afeta *apenas* o flag CY.

10.7 ALGUNS EXEMPLOS

Consideremos agora alguns exemplos simples de programas em linguagem do montador que usam as instruções apresentadas até agora. Estas instruções servem para movimentar dados de registrador a registrador e da memória para o registrador, e vice-versa, para efetuar adição e subtração. O seguinte programa soma dois números N_1 e N_2 de um byte:

Rótulo	Conteúdo	Comentário
SUM	MVI A $\langle N_1 \rangle$	Carregar o número N_1 em Acc
	MVI B $\langle N_2 \rangle$	Carregar o número N_2 no registrador B
	ADD B	Somar conteúdo de B ao conteúdo de Acc e deixar a soma em Acc

Neste programa fizemos uma entrada SUM no campo de rótulos não porque é requerido, mas para servir como lembrança de que, geralmente, um programa tem este campo. Se a soma $N_1 + N_2$ for 11 111 111 (= octal 377 = decimal 255) ou menos, o flag CY de transporte não estará ajustado (set). Se a soma for maior do que 377, CY estará ajustado (set).

O programa a seguir soma dois números de dois bytes $N_1 + N_2$. N_1 é um número de dois bytes com byte N_1 (M) mais significativo e byte N_1 (L) menos significativo. Similarmente, $N_2 = = N_2$ (M) N_2 (L).

Rótulo	Conteúdo	Comentário
SUM	LXI B $\langle N_1(L) \rangle$ $\langle N_1(M) \rangle$	Carregar o *par* B de registradores, isto é, B e C, com número N_1(M) N_1(L) de dois bytes
	LXI D $\langle N_2(L) \rangle$ $\langle N_2(M) \rangle$	Carregar o *par* D de registradores, isto é, D e E, com o número N_2(M) N_2(L) de dois bytes
	MOVA, E	Mover para Acc o conteúdo de E, que retém byte N_2(L)
	ADD C	Somar a Acc conteúdo de C, que retém byte N_1(L)
	MOVL, A	Movimentar soma N_2(L) + N_1(L) para registrador L a fim de fazer Acc disponível para somar N_2(M) + N_2(M)

(*continua*)

(continuação)

MOV A, D	Movimentar para Acc conteúdo de D que retém byte N_2(M)
ADC B	Somar a Acc conteúdo de B, que retém byte N_1(M); somar também *bit de transporte*, se houver, gerado quando a soma N_2(L) + N_1(L) for formada
MOV H, A	Movimentar soma N_2(M) + N_1(L) + CY de Acc para registrador H, de modo que o resultado inteiro esteja agora no par H, L de registradores

10.8 INSTRUÇÕES DE INCREMENTAR-DECREMENTAR

O 8080 tem seis instruções que permitem incrementação e decrementação de registradores, pares de registradores e do conteúdo de um local da memória cujo endereço está no par H, L de registradores. Estas instruções estão listadas na Tab. 10.8-1. Conforme indicado, em quatro dos casos, os flags Z, S, P, AC (mas não CY) são afetados, e em dois dos casos nenhum dos flags é afetado. Os registradores que podem ser incrementados ou decrementados incluem o acumulador e qualquer um dos outros registradores de propósitos gerais B, C, D, E, H e L. Qualquer um dos pares de registradores B e C, ou D e E ou H e L pode ser incrementado ou decrementado.

Tabela 10.8-1 Instruções de incrementar e decrementar do 8080

INCREMENT REGISTER	INR r	r + 1 → r
Incrementar conteúdo do registrador r; todos os flags afetados, *exceto* CY		
DECREMENT REGISTER	DCR r	r − 1 → r
Decrementar conteúdo do registrador r; todos os flags afetados, *exceto* CY		
INCREMENT REGISTER PAIR	INX rp	rp + 1 → rp
Incrementar conteúdo do par rp; *nenhum* flag afetado		
DECREMENT REGISTER PAIR	DCX rp	rp − 1 → rp
Decrementar conteúdo do par rp de registradores; *nenhum* flag afetado		
INCREMENT MEMORY	INR M	M[H, L] + 1 → M[H, L]
Incrementar conteúdo do local da memória cujo endereço está no par H, L de registradores; todos os flags afetados, *exceto* CY		
DECREMENT MEMORY	DCR M	M[H, L] − 1 → M[H, L]
Decrementar conteúdo do local da memória cujo endereço está no par H, L de registradores; todos os flags afetados, *exceto* CY		

10.9 INSTRUÇÕES LÓGICAS

As instruções disponíveis no 8080 que envolvem operações lógicas são listadas na Tab. 10.9-1. As operações são em número de três e incluem AND, OR e EXCLUSIVE-OR. Para evitar confusão entre operações aritméticas e lógicas, usamos os símbolos $\wedge$ e $\vee$ para representar AND e OR. Como nas instruções de adição e subtração, um operando é sempre achado no acumulador, e o resultado é colocado no acumulador. Há três fontes permitidas para o segundo operando, um registrador de propósitos gerais, um local na memória ou a própria instrução, isto é, numa instrução imediata onde o operando é o segundo byte da instrução. Há, por conseguinte, 3 × 3 = 9 instruções lógicas.

Estas operações lógicas são efetuadas numa base de bit-por-bit. A execução de uma operação lógica entre dois bytes envolve oito operações separadas, cada uma envolvendo um bit de cada um dos dois bytes. Um exemplo é dado na Fig. 10.9-1. Aqui o conteúdo do acumulador e de um registrador r está especificado, e também está mostrado o resultado que aparecerá no acumulador depois de ser executada a instrução ANA r (registrador AND).

Todas as instruções lógicas afetam *todos* os flags. Assim, como sempre, os flags Z, S e P registrarão se a saída da ULA é zero, se o bit mais significativo é 1 ou 0 e se a paridade é ímpar ou par. Já que uma operação lógica não envolve nenhum transporte de posição de bit para posição de bit, devemos imaginar que qualquer registro escrito nos flags AC e CY seria irrelevante. Tal é realmente o caso, mas necessitamos ter em mente que a ULA incorporou nela, em cada posição de bit, lógica para gerar um transporte, como será pedido quando os bits de entrada forem interpretados como números. Quando for solicitada uma operação lógica, os transportes assim gerados não serão transferidos para a próxima posição de bit mais alta. Ainda o transporte gerado na quarta posição de bit e na oitava posição de bit são encontrados na saída da ULA e serão devidamente registrados nos flags AC e CY, a menos que alguma resposta alternativa específica seja fornecida pelo controlador. No 8080, como mostra a Tab. 10.9-1, em todos casos, exceto dois, os flags AC e CY estão limpos. Com as duas instruções ANA, somente o flag CY está limpo. Igualmente, pode ser que, quando uma operação lógica for executada, não tenhamos nenhum interesse nos flags. Ainda, em cada ponto no desenvolvimento de um programa, necessitamos estar precisa-

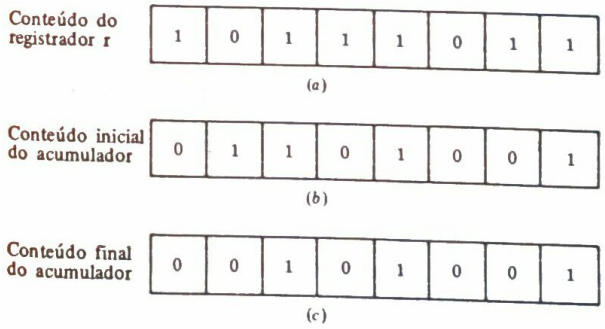


Fig. 10.9-1 Conteúdo inicial de (*a*) um registraoor de propósitos gerais e (*b*) o acumulador. (*c*) O conteúdo do acumulador após a execução da instrução ANA r.

Tabela 10.9-1 As instruções lógicas do 8080

AND REGISTER	ANA r	$A \wedge r \rightarrow A$

Conteúdo do registrador submetido a uma operação AND com conteúdo de Acc e resultado colocado em Acc; flag CY limpo

AND MEMORY	ANA M	$A \wedge M[H, L] \rightarrow A$

Conteúdo do local da memória cujo endereço está no par H, L de registradores é submetido a uma operação AND com Acc e resultado colocado em Acc; flag CY limpo

AND IMMEDIATE	ANI $\langle B_2 \rangle$	$A \wedge \langle B_2 \rangle \rightarrow A$

Segundo byte, B_2, da instrução submetido a uma operação AND com Acc e resultado colocado em Acc; flags CY e AC limpos

OR REGISTER	ORA r	$A \vee r \rightarrow A$

Conteúdo do registrador submetido a uma operação OR com conteúdo de Acc e resultado colocado em Acc; flags CY e AC limpos

OR MEMORY	ORA M	$A \vee M[H, L] \rightarrow A$

Conteúdo do local da memória cujo endereço está no par H, L é submetido a uma operação OR com conteúdo de Acc e resultado colocado em Acc; flags CY e AC limpos

OR IMMEDIATE	ORI $\langle B_2 \rangle$	$A \vee \langle B_2 \rangle \rightarrow A$

Segundo byte, B_2, da instrução submetido a uma operação OR com conteúdo de Acc e resultado colocado em Acc; flags CY e AC limpos

EXCLUSIVE-OR REGISTER	XRA r	$A \oplus r \rightarrow A$

Conteúdo do registrador r submetido a uma operação EXCLUSIVE-OR com conteúdo de Acc e resultado colocado em Acc; flags CY e AC limpos

EXCLUSIVE-OR MEMORY	XRA M	$A \oplus M[H, L] \rightarrow A$

Conteúdo do local da memória cujo endereço está no par H, L de registradores é submetido a uma operação EXCLUSIVE-OR com conteúdo de Acc e resultado colocado em Acc; flags CY e AC limpos

EXCLUSIVE-OR IMMEDIATE	XRA $\langle B_2 \rangle$	$A \oplus \langle B_2 \rangle \rightarrow A$

Segundo byte, B_2, da instrução submetido à operação EXCLUSIVE-OR com conteúdo de Acc e resultado colocado em Acc; flags CY e AC limpos

mente atentos ao estado dos flags por causa de seu efeito sobre a maneira pela qual o programa se desenrola.

10.10 AS INSTRUÇÕES DE COMPARAÇÃO

Há três instruções que nos permitem comparar os valores numéricos do número N (Acc) registrado no acumulador, com um segundo número N (r, M, I) num registrador ou na memória ou imediatamente disponível, isto é, dado como parte da instrução. Se N (Acc) = N (r, M, I), a igualdade ajustará (set) Z em Z = 1; e se N (Acc) $\neq N$ (r, M, I), então Z permanecerá ou será reajustado (reset) em Z = 0. Se N (Acc) $< N$ (r, M, I), o flag CY será ajustado (set) em CY = 1. Se N (Acc) $>$ N (r, M, I), o flag CY será deixado inalterado. Podemos, portanto, deduzir que N (Acc) $>$ N (r, M, I) se acharmos CY = 0 e se tivermos tido a previsão de limpar CY antes de usar a instrução de comparação. As instruções de comparação são descritas na Tab. 10.10-1.

Tabela 10.10-1 Instruções de comparação do 8080

COMPARE REGISTER	CMP r	A − r

Conteúdos do registrador r e de Acc aplicados à entrada da ULA. ULA instruída para efetuar subtração A − r. Resultado desta subtração *não* é registrado em Acc nem em qualquer outro registrador. Todos flags são afetados de maneira normal. Se A = r, então A − r = 0 e este resultado será registrado no flag Z, e teremos Z = 1. Se A < r, teremos Z = 0 e, ainda, a subtração A − r gerará um *empréstimo* fora da posição do bit mais significativo, e teremos CY = 1

COMPARE MEMORY	CMP M	A − M[H, L]

Operação efetuada é a mesma que em CMP r, exceto que comparação é feita não com conteúdos do registrador de propósitos gerais, mas com conteúdo do local da memória cujo endereço é mantido no par de registradores H, L

COMPARE IMMEDIATE	CPI $\langle B_2 \rangle$	A − $\langle B_2 \rangle$

Uma instrução de dois bytes; neste caso comparação é feita com $\langle B_2 \rangle$, o segundo byte da instrução

10.11 AS INSTRUÇÕES DE GIRAR (ROTATE)

Há quatro instruções que fazem com que o acumulador opere como um registrador de deslocamentos. Duas das instruções efetuam um deslocamento à direita, e duas efetuam um deslocamento à esquerda. O flag CY do registrador de estado está envolvido em todas as quatro instruções, mas nenhum dos outros flags é afetado. Os detalhes destas instruções estão escritos na Tab. 10-11-1 e na Fig. 10.11-1.

10.12 AS INSTRUÇÕES DE COMPLEMENTAR E SET

Há duas instruções de COMPLEMENTAR e uma instrução de SET, conforme listado na Tab. 10.12-1. Elas nos permitem complementar o acumulador, complementar o flag de transporte e

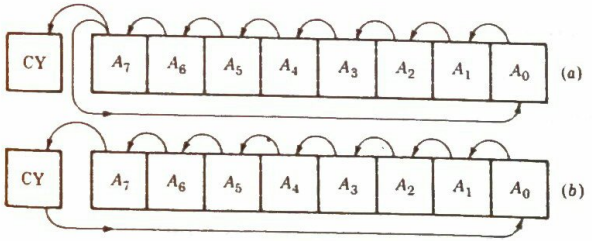


Fig. 10.11-1 As instruções (*a*) "girar à esquerda" (*rotate left*) e (*b*) "girar à esquerda por transporte" (*rotate left through carry*).

Tabela 10.11-1 As instruções de girar

ROTATE LEFT	RLC	$A_n \rightarrow A_{n+1}$ $A_7 \rightarrow A_0$; $A_7 \rightarrow$ CY	$(n \neq 7)$;

Cada bit no registrador de Acc, exceto o bit mais significativo, A_7, é deslocado uma posição de bit à esquerda. Bit A_7 é transferido para posição A_0 e também para flag CY. Nenhum dos outros flags é afetado

ROTATE RIGHT	RRC	$A_n \rightarrow A_{n-1}$ $A_0 \rightarrow A_7$; $A_0 \rightarrow$ CY	$(n \neq 0)$;

Cada bit no registrador de Acc, exceto o bit menos significativo, A_0, é deslocado uma posição de bit à direita. Bit A_0 é transferido para posição A_7 e também para flag CY. Nenhum dos outros flags é afetado

ROTATE LEFT THROUGH CARRY	RAL	$A_n \rightarrow A_{n+1}$ $A_7 \rightarrow$ CY; CY $\rightarrow A_0$	$(n \neq 7)$;

Cada bit no registrador de Acc, exceto o bit mais significativo, A_7, é deslocado uma posição de bit à esquerda. Bit A_7 é transferido para CY, e conteúdo de CY é transferido para A_0. Nenhum dos outros flags é afetado

ROTATE RIGHT THROUGH CARRY	RAR	$A_n \rightarrow A_{n-1}$ $A_0 \rightarrow$ CY; CY $\rightarrow A_7$	$(n \neq 0)$;

Cada bit no registrador de Acc, exceto o bit menos significativo, A_0, é deslocado uma posição de bit à direita. Bit A_0 é transferido para CY, e conteúdos de CY são transferidos para A_7. Nenhum dos outros flags é afetado

ajustar (set) o flag de transporte. Se quisermos reajustar (reset) o flag de transporte, podemos fazê-lo usando a instrução "registrador AND", que efetua a operação AND entre um registrador e o acumulador. Selecionamos como o registrador especificado nesta instrução o próprio acumulador. O resultado é que o acumulador é submetido a uma operação AND consigo mesmo, deixando o conteúdo do acumulador inalterado. No entanto, conforme notamos, a instrução limpa o flag CY.

10.13 UM EXEMPLO

Como um exemplo de um programa simples que usa algumas das instruções lógicas e de transferência de dados do 8080, consideremos o seguinte. Suponhamos que guardamos na memória um

Tabela 10.12-1 As instruções de complementar e set

COMPLEMENT ACCUMULATOR	CMA	$\overline{A} \rightarrow A$

Cada bit de Acc substituído por seu complemento; nenhum flag é afetado

COMPLEMENT CARRY	CMC	$\overline{CY} \rightarrow CY$

Bit no flag CY substituído por seu complemento; nenhum outro flag é afetado

SET CARRY	STC	$1 \rightarrow CY$

Flag de transporte ajustado (set) em 1; nenhum outro flag é afetado

número decimal de dois dígitos, tendo os números sido feitos disponíveis na forma de caracteres do código ASCII com paridade par. Se o número for 59, temos 5 = 00110101 e 9 = 00111001 e o número ocupa dois locais da memória M_1 e M_2. Cada local da memória tem duas partes de quatro bits M_1[H], M_1[L] e M_2[H], M_2[L]. Em cada um dos números do código ASCII, somente os quatro bits mais à direita têm significado numérico. Estes quatro bits representam o número em código BCD. Os outros quatro bits servem para estabelecer a paridade e para distinguir os números dos outros símbolos de caracteres. Suponhamos agora que queremos colocar juntos os dois códigos BCD de quatro bits num único byte. (Podemos igualmente querer efetuar uma tal consolidação para reservar um local da memória em antecipação a efetuar uma operação aritmética com o número.)

Este processo de suprimir bits que não mais servem para um propósito e de consolidar bits úteis num único byte é chamado *empacotamento* (*packing*). O programa que se segue realiza o empacotamento e retorna o byte empacotado a um local M_3 da memória.

Rótulo	Conteúdo	Comentário
	LDA $\langle M_1[L] \rangle$ $\langle M_1[H] \rangle$	Carregar Acc com conteúdo do local M_1 da memória
	ANI ⟨00001111⟩	Submeter a uma operação AND Acc com 00001111, arranjando deste modo que todos os quatro bits à esquerda se tornem 0s e todos os quatro bits à direita permaneçam inalterados (este processo de limpar locais que retenham bits irrelevantes é chamado *mascaramento*)
	RLC RLC RLC RLC	Girar quatro vezes à esquerda de modo que os bits BCD estejam nas quatro posições de bit à esquerda e as quatro posições de bit à direita sejam todas 0s
	MOV B, A	Mover conteúdo de Acc para registrador B de modo que Acc esteja disponível para processar segundo caractere
	LDA $\langle M_2[L] \rangle$ $\langle M_2[H] \rangle$	Carregar Acc com segundo caractere ASCII do local M_2
	ANI ⟨00001111⟩	Mascarar quatro posições de bit à esquerda
	ORA B	Submeter a uma operação OR o conteúdo do registrador B com Acc; os dois números BCD são agora empacotados num único byte
	STA $\langle M_3[L] \rangle$ $\langle M_3[H] \rangle$	Armazenar conteúdo de Acc no local M_3 da memória

10.14 AS INSTRUÇÕES DE JUMP (SALTO)

A despeito do grande número de instruções que já introduzimos, achamos que as computações e manipulações de dados do 8080 são ainda severamente limitadas. A versatilidade das possibilidades de programação é grandemente realçada pelas duas instruções de jump incorporadas ao repertório do 8080. Ambas são instruções de três bytes. Uma instrução de jump é incondicional, que, conforme usual, descrevemos como se segue:

JUMP	JMP $\langle B_2\rangle\langle B_3\rangle$	$\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC

Transferir para contador do programa o endereço $\langle B_3\rangle\langle B_2\rangle$ dado nos segundo e terceiro bytes de instrução.

O efeito resultante desta instrução é que, depois do JUMP ter sido executado, a próxima instrução a ser executada é a instrução no endereço $\langle B_3\rangle\langle B_2\rangle$.

O segundo jump é uma instrução de jump *condicional*, que ocorre se e somente se a condição especificada for satisfeita. As condições citadas são as condições dos quatro flags Z, S, P e CY. O flag AC não está envolvido nestas instruções de jump condicional. Há quatro flags, cada flag tem dois estados, e há uma instrução de jump condicional associada a cada estado de cada flag individual. Por conseguinte, a instrução de jump realmente expande-se nas oito instruções que estão listadas na Tab. 10.14-1.

As instruções de jump não afetam nenhum dos flags. Dependendo dos estados dos flags, um salto condicional pode ou não ocorrer, mas, em qualquer caso, depois das instruções de jump terem sido executadas, os estados dos flags são os mesmos que antes.

Tabela 10.14-1 As instruções de jump condicional

JUMP IF ZERO FLAG IS RESET (resultado não é zero)	JNZ $\langle B_2\rangle\langle B_3\rangle$	If Z = 0, $\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC
JUMP IF ZERO FLAG IS SET (resultado é zero)	JZ $\langle B_2\rangle\langle B_3\rangle$	If Z = 1, $\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC
JUMP IF SIGN FLAG IS RESET (sinal é positivo)	JP $\langle B_2\rangle\langle B_3\rangle$	If S = 0, $\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC
JUMP IF SIGN FLAG IS SET (sinal é negativo)	JM $\langle B_2\rangle\langle B_3\rangle$	If S = 1, $\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC
JUMP IF PARITY FLAG IS RESET (paridade é ímpar)	JPO $\langle B_2\rangle\langle B_3\rangle$	If P = 0, $\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC
JUMP IF PARITY FLAG IS SET (paridade é par)	JPE $\langle B_2\rangle\langle B_3\rangle$	If P = 1, $\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC
JUMP IF CARRY FLAG IS RESET (nenhum transporte ou empréstimo)	JNC $\langle B_2\rangle\langle B_3\rangle$	If CY = 0, $\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC
JUMP IF CARRY FLAG IS SET (é indicado um transporte ou empréstimo)	JC $\langle B_2\rangle\langle B_3\rangle$	If CY = 1, $\langle B_3\rangle\langle B_2\rangle \rightarrow$ PC

Como um exemplo simples que envolve uma instrução de jump condicional, consideremos o seguinte programa. Aqui supomos que temos dois números sem sinais N_1 e N_2 em dois locais sucessivos K e $K + 1$ da memória e que queremos transferir para o local $K + 2$ da memória o maior dos dois números.

Rótulo	Conteúdo	Comentário
	LXIH ⟨K[L]⟩ ⟨K[H]⟩	Carregar par H de registradores com endereço K de número N_1
	MOV A, M	Mover para Acc o conteúdo da memória cujo endereço está no par H de registradores
	INX H	Incrementar conteúdo do par H de registradores (par H de registradores agora retém endereço $K + 1$ de número N_2)
	CMP M	Subtrair de Acc o conteúdo do local da memória no par H de registradores e ajustar (set) os flags CY e Z apropriadamente. Se $N_1 = N_2$, então Z = 1, se $N_1 < N_2$, CY = 1. Conteúdo de Acc não alterado. Se CY = 0, então $N_1 > N_2$
	JNC ⟨END[L]⟩ ⟨END[H]⟩	Se CY = 0, então $N_1 \geq N_2$ e número já em Acc é número a ser armazenado em local $K + 2$ da memória. Neste caso, saltar para instrução no local da memória rotulado "END". Se CY = 1, necessitamos mover N_2 para Acc, uma transferência executada pela próxima instrução
	MOV A, M	Mover para Acc o conteúdo da memória cujo endereço está no par H de registradores
END	INX H	Incrementar conteúdo do par H de registradores (par de registradores agora retém endereço $K + 2$)
	MOV M, A	Armazenar conteúdo de Acc no local da memória cujo endereço está no par H de registradores

No programa acima, poderíamos ter feito aparecer o conteúdo da memória usando a instrução LDA. A instrução de três bytes LDA ⟨K[L]⟩ ⟨K[H]⟩ realizaria o mesmo propósito que os primeiros quatro bytes do programa conforme escrito. Por outro lado, o esquema de endereçamento indireto, onde o endereço da memória está no par H de registradores, possibilita a conveniência de ter acesso a locais sucessivos, como requerido no caso presente, simplesmente usando a instrução INX H de um byte. Notar também que, em virtude do endereço ter sido no par H de registradores, poderíamos usar a instrução CMP M, tornando possível desta maneira uma comparação com o conteúdo de um local da memória sem chamar os conteúdos num dos registradores de propósitos gerais.

10.15 AS INSTRUÇÕES DE CHAMADA E DE RETORNO

Já observamos na Seção 9.7 a vantagem de se poder chamar uma sub-rotina. Quando uma sub-rotina é chamada, é necessário reservar o endereço no programa principal a que o computador

deve retornar depois do programa da sub-rotina ter sido concluído. No esquema da Seção 9.7 reservamos um local da memória imediatamente precedendo o programa da sub-rotina para reservar este endereço de retorno. No 8080 e nos microprocessadores, geralmente um esquema alternativo e mais conveniente é usado para reservar o endereço de retorno. Uma pilha é usada com este propósito.

Como observado na Seção 6.16, uma pilha é uma memória de acesso aleatório (ou uma parte especialmente posta de lado de uma memória maior) que opera numa base "de rebaixamento" (last-in first-out = LIFO). O endereço do local da pilha é retido num contador. No 8080 (e em outros microprocessadores) este contador de endereços é chamado *apontador de pilha*, e a pilha é uma porção posta de lado da mesma memória de acesso aleatório que está proporcionando facilidade de armazenamento ao microprocessador. Esta memória, se estiver sendo usada em sua capacidade plena, tem uma capacidade de 2^{16} palavras. O endereço *superior* (topo) da memória é considerado memória com local 0000 (hex), e o endereço *inferior* da memória tem endereço FFFF (hex). Suponhamos, então, que o programa principal esteja armazenado na parte superior da memória, digamos partindo de 0000. Podemos, então, muito racionalmente, usar a parte inferior da memória para a pilha, já que necessitamos assegurar que a pilha não invada seções da memória reservadas para outros propósitos.

Na organização das operações de controle que manipulam a pilha, pode ser providenciado que, quando palavras adicionais forem armazenadas na pilha, a pilha cresça quer no sentido descendente, quer no sentido ascendente. Suponhamos que escolhemos crescimento no sentido descendente. Então devemos considerar o fato de que, depois do apontador de pilha alcançar FFFF, seu próximo registro será 0000. Assim, quando a pilha cresce, estamos preocupados com que não invada a memória principal armazenada começando no topo da memória. Esta dificuldade pode surgir mesmo se o espaço da memória entre a parte inferior do armazenamento da memória principal e a parte superior da pilha esteja disponível. Se tentarmos evitar esta dificuldade, partindo da pilha bem acima de FFFF, podemos achar memória desperdiçada na parte inferior da pilha, já que, quando começamos a escrever um programa, é muitas vezes difícil julgar quantos locais de pilha serão necessários. Por estas razões, no 8080, como em outros microprocessadores, a pilha é projetada para crescer de baixo para cima. Depois, se iniciarmos a pilha em FFFF, necessitamos apenas estar interessados em se a pilha, crescendo para cima, satisfaz os outros itens armazenados que estejam crescendo para baixo. Se ainda enfrentarmos esta dificuldade, é porque simplesmente não fornecemos capacidade de memória suficiente, e não porque falhamos em usar sabiamente a memória.

Tendo decidido que a pilha deve crescer para cima, teremos ainda uma escolha de como a pilha deve operar. Por outro lado, podemos arranjá-la de modo que, quando uma palavra nova deve ser colocada na pilha, ela seja armazenada no local endereçado pelo apontador de pilha, *depois* do que o apontador de pilha decrementa. Num tal caso, quando queremos recuperar a palavra do topo da pilha, devemos *primeiro* incrementar o apontador de pilha e depois ler. Por outro lado, podemos arranjá-la de modo que na colocação de uma nova palavra na pilha nós *primeiro* decrementemos e depois armazenemos. Neste caso, na recuperação de uma palavra nós primeiro lemos e depois incrementamos. No 8080 esta segunda alternativa é usada. Portanto, se queremos armazenar uma palavra no local FFFF, primeiro devemos colocar o apontador de pilha em 0000. Com estas observações preliminares, podemos agora considerar as instruções CALL do 8080.

A instrução CALL é uma instrução de três bytes em que o segundo e o terceiro bytes especificam o endereço do começo da sub-rotina que está sendo chamada. A execução desta instrução CALL realiza as seguintes operações: (1) armazena no topo da pilha o endereço da instrução

imediatamente seguinte à instrução CALL. Esta instrução é a instrução do programa principal, à qual o computador deve retornar depois da sub-rotina ter sido completada, e (2) transfere ao contador do programa (PC) o endereço da primeira instrução na sub-rotina que está sendo chamada. Uma vez que os endereços têm dezesseis bits de comprimento, dois locais da memória são requeridos para armazenar o endereço de retorno do programa principal. Daí, após o armazenamento de endereço estar completo, o apontador de pilha tem um registro que é 2 em vez de 1 menor do que seu registro antes da instrução CALL. Em conjunto, a instrução CALL é descrita como se segue:

CALL SUBROUTINE (Chamar sub-rotina)	CALL $\langle B_2\rangle\langle B_3\rangle$	PC [high] → SP − 1 PC [low] → SP − 2 SP − 2 → SP $\langle B_3\rangle\langle B_2\rangle$ → PC

Aqui, SP representa o registro do apontador de pilha pouco antes da instrução CALL ser executada. O símbolo PC [high] representa os bits de ordem alta do endereço da instrução imediatamente seguinte à instrução CALL. Similarmente, PC [low] são os bits de ordem baixa desta mesma instrução.

Como a instrução JUMP, a instrução CALL é disponível tanto incondicional quanto condicionalmente, sendo as condições examinadas novamente os estados de quatro dos cinco flags. Os mnemônicos destas instruções CALL condicionais são como se segue:

CNZ	Call if zero flag is reset (Z = 0, result is not zero)
CZ	Call if zero flag is set (Z = 1, result is zero)
CP	Call if sign flag is reset (S = 0, sign is positive)
CM	Call if sign flag is set (S = 1, sign is minus)
CPO	Call if parity flag is reset (P = 0, parity is odd)
CPE	Call if parity flag is set (P = 1, parity is even)
CNC	Call if carry flag is reset (CY = 0, no carry or borrow)
CC	Call if carry flag is set (CY = 1, carry or borrow indicated)

Quando tivermos chamado uma sub-rotina e a sub-rotina tiver sido completada, deve haver um retorno para o programa principal. Por esta razão, há disponível uma instrução RETURN e uma instrução RETURN condicional. A instrução RETURN põe novamente no contador do programa o endereço armazenado na pilha. A instrução RETURN é uma instrução de um byte e tem a seguinte descrição:

RETURN	RET	SP → PC [low] SP + 1 → PC [high] SP + 2 → SP

As instruções RETURN condicionais têm mnemônicos RNZ, RZ etc., como nas instruções JUMP e CALL.

Antes que possamos usar a instrução CALL, devemos carregar um endereço no apontador de pilha a fim de estabelecer o local da memória a partir do qual a pilha crescerá. Com este propósito podemos usar a instrução LXI (*"load register pair immediate"*), uma vez que o apontador de pilha

é um dos pares de registradores com que a instrução LXI pode ser usada. Alternativamente, podemos estabelecer o endereço de partida do apontador de pilha carregando o par de registradores H, L primeiro e, depois, transferindo o conteúdo deste par para o apontador de pilha. Com este propósito, há disponível a seguinte instrução:

MOVE REGISTER PAIR H TO SP	SPHL	H, L → SP

Muitas vezes acontece que, quando uma sub-rotina foi chamada, ela requer os serviços de uma segunda sub-rotina, que requer os serviços de uma terceira sub-rotina etc. Tal situação é descrita como encaixamento de sub-rotinas. Nestas circunstâncias a pilha funciona muito eficientemente. Como cada sub-rotina e sub-sub-rotina é chamada, os endereços de retorno são amontoados na pilha, e como os retornos da sub-sub-rotina e da sub-rotina devem ser efetuados, os endereços de retorno devem ser recuperados exatamente na ordem oposta àquela em que eles foram armazenados. Precisamente este aspecto de rebaixamento (LIFO) é uma característica essencial da pilha.

Como exemplo do uso das instruções CALL e RETURN, consideremos que teremos freqüentes ocasiões de converter um número na faixa de 00 (hex) a 0F (hex) (0000 a 1111 em binário) para código ASCII. Se a necessidade da conversão for bastante freqüente, gostaríamos de ser capazes de chamar uma sub-rotina de conversão, em vez de ter que reescrever o programa da conversão cada vez que for necessário. É deixado como exercício aos estudantes (Prob. 10.15-1) verificar (1) que, se o número estiver na faixa 00 a 09, a conversão é efetuada simplesmente adicionando 00110000 (30 hex) e (2) que, se o número for 10 (decimal) ou maior, a conversão é efetuada adicionando 0011 0111 = (0011 0000 + 0000 0111).

No programa que se segue fazemos aparecer o conteúdo de um local da memória *M* [H, L] e, sabendo que ele é um número na faixa de 0 a 15 (decimal), fazemos uma conversão para a representação ASCII e depois retornamos o número a seu local de memória original. Incluímos uma instrução que inicia o registro do apontador de pilha de modo que a pilha começará no local FFFF da memória.

Rótulo	Conteúdo	Comentário
	LXI SP 00 (hex) 00 (hex)	Carregar apontador de pilhas com registro 0000 (hex). Um decremento em SP iniciará pilha em FFFF
	LDA *M* [L] *M* [H]	Carregar Acc com dados no local *M* [H, L] da memória
	CALL ⟨B_2⟩ (ASC) ⟨B_3⟩	Chamar sub-rotina que começa no endereço B_3B_2 da memória (sub-rotina está no endereço rotulado ASC)
	STA *M* [L] *M* [H]	Armazenar conteúdo de Acc de volta no local *M* [H, L] da memória
	.	
	.	
	.	

(continua)

(*continuação*)

ASC	CPI 10 (decimal)	Comparar Acc com 10 (decimal). Se Acc for menor que 10, temos CY = 1. (Se CY = 0, então A ⩾ 10 e devemos somar 0011 0000 e também 0000 0111
	JC ⟨B_2⟩ (LTN) ⟨B_3⟩	Se CY = 1, saltar para instrução LTN (menor que dez)
	ADI 0000 0111	Somar imediato segundo byte (porque A ⩾ 10)
LTN	ADI 0011 0000	Somar imediato segundo byte (necessário em cada caso)
	RET	Retornar ao programa principal (STA, acima)

10.16 AS INSTRUÇÕES "PUSH AND POP"

Suponhamos que, quando chamamos uma sub-rotina, haja dados no acumulador, nos flags e nos registradores de propósitos gerais de que necessitaremos quando a sub-rotina estiver concluída. É fácil imaginar um programa de sub-rotina que pode usar os registradores de tal maneira que os dados necessitados não seriam preservados. Num tal caso, antes de chamar a sub-rotina, teríamos que armazenar os dados necessários em algum lugar do qual eles poderiam ser recuperados. Melhor ainda, poderíamos arranjar para que este processo de armazenamento fosse incorporado como parte da sub-rotina no começo da sub-rotina. Correspondentemente, as últimas instruções da sub-rotina restaurariam a cada registrador os dados presentes antes que a sub-rotina fosse chamada. Em princípio, podemos armazenar os dados em qualquer lugar na memória de acesso aleatório, mas na prática é muito mais conveniente armazenar estes dados na pilha. A porção de pilha de memória é limitada à operação de rebaixamento (*last-in, first-out operation*), mas no presente caso esta característica LIFO não é em absoluto nenhuma inconveniência.

Conforme observamos, a operação de escrita de dados no topo de uma pilha é chamada *operação "push"* (colocação de um elemento no topo de uma pilha), e o processo de leitura de dados do topo de uma pilha é chamado *operação "pop"* (retirada de um elemento do topo de uma pilha). O 8080 tem duas instruções PUSH e duas instruções POP. Os dados que aí são colocados, o são na forma de uma palavra de dezesseis bits. Assim os dados colocados constituem uma palavra de dois bytes, armazenada em dois locais da memória; daí, são requeridos dois decrementos do apontador de pilha de modo que SP possa apontar para os dois locais sucessivos da memória em que os dados devem ser armazenados. Podemos notar, entre parênteses, que a instrução CALL envolve uma operação *push* de colocação implícita e que ali, também, os dados colocados eram uma palavra de dezesseis bits, isto é, o conteúdo do contador do programa. Uma instrução PUSH é usada para colocar na pilha o conteúdo dos pares de registradores B (B e C) ou D (D e E) ou H (H e L) como se segue:

PUSH REGISTER PAIR	PUSH rp	rp [H] → SP − 1 rp [L] → SP − 2 SP − 2 → SP

Antes da execução da instrução, se o registro do apontador de pilha for SP, os bits de ordem elevada do par de registradores são colocados em SP − 1 e os bits de ordem baixa em SP − 2; quando a execução estiver concluída, o novo registro do apontador de pilha é SP − 2. O par de registradores colocado não pode ser o próprio apontador de pilha.

Correspondentemente, há uma instrução POP que é precisamente inversa desta instrução PUSH:

POP REGISTER PAIR	POP rp	SP → rp [L] SP + 1 → rp [H] SP + 2 → SP

Antes da execução da instrução, se o registro do apontador de pilha for SP, o conteúdo do local de memória SP é transferido para o registrador de ordem baixa do par rp de registradores, o conteúdo do local SP + 1 é transferido para o registrador de ordem elevada do par e, quando a execução estiver concluída, o novo registro do apontador de pilha é SP + 2. A palavra retirada não pode ser transferida para o próprio apontador de pilha.

Há uma segunda instrução PUSH. Ela coloca na pilha o conteúdo do acumulador e um segundo byte, chamado *palavra de estado do processador* (PSW). Cinco dos bits da PSW são os conteúdos dos cinco flags Z, S, P, CY e AC; os outros três são bits fantasmas e não servem a nenhum propósito, exceto suplementar os cinco bits flags de modo que uma palavra de oito bits seja formada. Correspondendo a esta instrução PUSH está uma instrução POP que realiza precisamente a operação inversa:

PUSH PROCESSOR STATUS WORD	PUSH PSW	A → SP − 1 PSW → SP − 2 SP − 2 → SP
POP PROCESSOR STATUS WORD	POP PSW	SP + 1 → A SP + 2 → PSW SP → SP + 2

Há uma instrução adicional envolvendo a pilha:

EXCHANGE STACK TOP WITH H AND L	XTHL	L ⟷ *M* [SP] H ⟷ *M* [SP + 1]

O conteúdo do registrador L é trocado pelo conteúdo do local da memória cujo endereço é especificado pelo conteúdo do registrador SP. O conteúdo do registrador H é trocado pelo conteúdo do local da memória cujo endereço é 1 mais do que o conteúdo do registrador SP. Enquanto esta instrução estiver sendo executada, o registro do apontador de pilha permanece inalterado.

Como um exemplo do uso da pilha e de algumas das instruções de pilha, consideremos agora a adição com múltipla precisão. Suponhamos que temos na memória dois números N_1 e N_2 de 64 bits e queremos somar estes números e armazenar a soma na memória. Cada um dos números terá quatro bytes e ocupará quatro locais da memória. A soma também terá que ocupar quatro locais da memória e, se houver um transporte da posição de bit 64, deixaremos este transporte no flag de transporte. A fim de evitar uma profusão de símbolos para representar locais de N_1, N_2 e da soma, usaremos números reais. Assim, usando números hexadecimais em toda parte, admitamos que N_1 seja armazenado nos locais da memória 0041, 0042, 0043 e 0044; que N_2 esteja nos locais 0051, 0052, 0053 e 0054; e que a soma deva ser armazenada nos locais 0061, 0062, 0063 e 0064. Em todos os casos supomos que os bits de ordem mais baixa estejam armazenados no local de número mais baixo etc., estando os bits de ordem mais elevada no local da memória com número mais alto.

Naturalmente, podemos fazer aparecer os bytes individuais de N_1 e de N_2 e armazenar os bytes de $N_1 + N_2$ usando as instruções de três bytes LDA $\langle B_2 \rangle \langle B_3 \rangle$ (load Acc from memory location B_3B_2) e STA $\langle B_2 \rangle \langle B_3 \rangle$ (store Acc in memory location B_3B_2). No entanto, uma vez que locais sucessivos são usados para N_1, N_2 e $N_1 + N_2$, seria vantajoso ser capaz de carregar registradores com endereços de partida e depois obter endereços sucessivos usando a instrução de incrementar um byte. Podemos usar os três pares de registradores H, D e E com este propósito; no entanto, também necessitamos de um registrador para fazer a contagem do número de bytes que adicionamos, de modo que possamos parar quando tivermos somado os quatro bytes. Uma vez que todos os pares de registradores estão consignados, e desde que o acumulador seja necessário para efetuar a adição, podemos olhar para o uso da pilha. O seguinte programa é escrito como um programa principal, que carrega a informação de partida nos registradores, e uma sub-rotina, que realmente efetua a adição com múltipla precisão. A pilha é usada de início simplesmente porque uma sub-rotina é chamada e novamente para liberar o par B de registradores. O programa também ilustra o uso das instruções XTHL para tornar disponível um par de registradores extra na pilha.

Rótulo	Conteúdo	Comentário
	LXI H 41 00	Carregar par de registradores H com o endereço de partida de N_1
	LXI D 51 00	Carregar par de registradores D com o endereço de partida de N_2
	LXI B 61 00	Carregar par de registradores B com o endereço de partida de $N_1 + N_2$
	MVI A 04	Set Acc em 4, correspondendo a quatro bytes a serem somados
	LXI SP 70 00	Iniciar SP em antecipação de uma instrução CALL (qualquer lugar é aceitável, contanto que não haja sobreposição)

(*continua*)

(continuação)

	CALL MPADD	Instrução de três bytes que chama sub-rotina de somar com precisão múltipla (MPADD)
	. . .	
MPADD	PUSH B	Empurrar conteúdo de par B de registradores para pilha para tornar registrador disponível para reter conteúdo de Acc
	MOV B, A	Mover conteúdo de Acc para registrador B
	ANA A	Submeter a uma operação AND Acc com ele mesmo a fim de limpar CY
ADD W	LDAX D	Carregar o acumulador com número no local da memória cujo endereço está no par D de registradores
	ADC M	Somar ao número no Acc número em local da memória cujo endereço está no par H, L. Somar também conteúdo de flag CY
	XTHL	Trocar conteúdo do topo da pilha por conteúdo do par H, L de registradores. Esta instrução coloca endereço de partida da soma $N_1 + N_2$ no par H, L. Este endereço, originalmente no par B de registradores, foi o último dado colocado na pilha em execução da instrução PUSH B acima
	MOV M, A	Mover conteúdo de Acc para local da memória cujo endereço está no par H, L
	INX H	Incrementar par H, L de registradores para arranjar que, quando próximo conjunto de bytes de N_1 e N_2 for acrescentado, soma seja armazenada no local da memória com próximo número mais alto
	XTHL	Trocar novamente conteúdo do topo da pilha pelo conteúdo do par de registradores H, L a fim de colocar o endereço de partida de N_1 novamente no par H, L
	INX D	Incrementar local de endereço no par D
	INX H	Incrementar local de endereço no par H
	DCR B	Decrementar registrador B

Neste ponto todos os locais de endereço de N_1, N_2 e $N_1 + N_2$ foram incrementados, e a contagem do número de bytes a serem processados foi decrementada. Agora necessitamos apenas repetir instruções das somas até que o número de bytes a ser processado tenha sido decrementado a zero. O programa continua:

	JNZ ADD W	Se o registro no registrador B não for zero, JUMP de volta para instrução cujo rótulo de endereço seja ADD W (somar palavra = *add word*); se B = 0, prosseguir para próxima instrução

Neste ponto a sub-rotina foi concluída, e gostaríamos de usar a instrução RET (RETURN) para voltar ao programa principal. O endereço em que o programa principal deve começar novamente foi armazenado na pilha pela instrução CALL. A instrução RET colocará o endereço no topo da pilha de volta ao contador do programa. Necessitamos, portanto, lembrar que depois

da instrução CALL usamos a instrução PUSH B. Devemos, por conseguinte, primeiro usar uma instrução POP B para obter o endereço de retorno ao topo da pilha. A sub-rotina é, então, concluída como se segue:

	POP B	Retirar da pilha de volta ao registrador B para obter o endereço de retorno do programa principal ao topo da pilha
	RET	Retornar ao programa principal

10.17 UM EXEMPLO: MULTIPLICAÇÃO

Nesta seção consideramos a operação aritmética da multiplicação, tanto por causa de sua importância fundamental, quanto porque ela constitui ainda um simples e interessante exemplo de programação.

A multiplicação com números binários é inteiramente direta e é efetuada exatamente como em aritmética decimal (ver Fig. 10.17-1). O multiplicando é multiplicado alternadamente por cada um dos bits do multiplicador. Assim, a multiplicação é mais fácil do que no sistema decimal, já que o resultado desta multiplicação é zero ou o próprio multiplicando, dependendo do fato do bit multiplicador ser 0 ou 1. Para levar em conta o aumento em significado numérico dos bits multiplicadores sucessivos conforme vamos para a esquerda, há um correspondente deslocamento à esquerda dos produtos parciais sucessivos. O produto é, então, formado pela adição de todos os produtos parciais. Quando a multiplicação for efetuada a lápis e papel, é realmente melhor, como na Fig. 10.17-1, formar todos os produtos parciais primeiro e depois somar; assim, desta maneira, necessitamos efetuar apenas uma operação de adição. Quando, por outro lado, a multiplicação for feita pelo microprocessador, é melhor somar cada produto parcial logo que ele esteja formado, evitando assim a necessidade de registradores para reter ou guardar os produtos parciais até que estejamos prontos para somá-los.

No procedimento da Fig. 10.17-1 partimos usando o bit multiplicador menos significativo mais à direita, depois o próximo bit mais significativo etc. Uma vez que um produto parcial tenha sido escrito, nunca necessitamos apagá-lo a fim de deslocá-lo para uma nova posição. Na escrita de nosso programa de multiplicação de microprocessador, contudo, revela-se mais conveniente dispor o procedimento de outro modo. Partimos do bit mais significativo mais à esquerda e deslocamos produtos parciais e somas de produtos parciais depois deles terem sido formados. O procedimento é mostrado etapa por etapa na Fig. 10.17-2, onde fizemos a mesma multiplicação que na Fig. 10.17-1.

```
        1 0 1 1     Multiplicando
        1 1 0 1     Multiplicador
      ---------
        1 0 1 1
      0 0 0 0       } Produtos parciais
    1 0 1 1
  1 0 1 1
---------------
1 0 0 0 1 1 1 1     Produto
```

Fig. 10.17-1 A maneira "normal" de executar uma multiplicação.

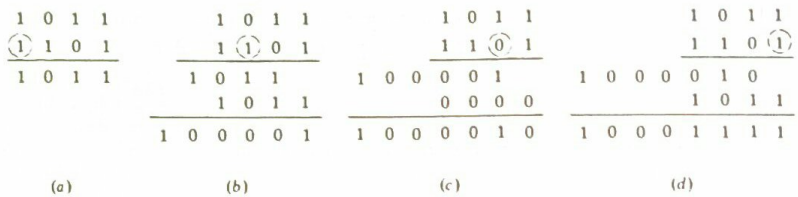


Fig. 10.17-2 Como o programa do microprocessador executa uma multiplicação.

Na Fig. 10.17-2*a* formamos o produto parcial usando o bit multiplicador mais à esquerda (envolvido em círculo). Na Fig. 10.17-2*b* copiamos o resultado da Fig. 10.17-2*a*, exceto que nós o deslocamos uma casa à esquerda. Aqui, também, formamos o próximo produto parcial e somamos. O resultado na Fig. 10.17-2*b* está copiado na Fig. 10.17-2*c*, exceto que deslocamos novamente. Finalmente, procedendo da mesma maneira, mostramos o resultado final na Fig. 10.17-2*d*.

Admitiremos agora que no local de memória 0040 (hex) temos um multiplicando de um byte e que no local 0041 da memória temos um multiplicador de um byte. O programa que se segue formará o produto e armazenará o resultado na memória. Uma vez que o produto de dois números de oito dígitos pode estender-se até dezesseis dígitos, armazenaremos o resultado nos locais 0042 e 0043 da memória.

Rótulo	Conteúdo	Comentário
	LXI H 40 00	Carregar par de registradores H, L com 0040, que é endereço do multiplicando
	MOV E, M	Mover multiplicando da memória para registrador E
	MVI D 00	Limpar registrador D, que é outra metade do par de registradores D, E (a razão aparecerá abaixo)
	INX H	Incrementar par de registradores H, L para local da memória de multiplicador
	MOV A, M	Mover multiplicando da memória para Acc
	LXI H 00 00	Limpar par de registradores H, L. Até agora H, L foi usado para guardar endereços. Agora começamos a usá-lo para guardar produtos parciais como eles são formados, para deslocar e somar
	MVI B 08	Carregar registrador B com número 8. Há oito bits multiplicadores e, decrementando o registrador B, saberemos quando usamos todos os bits e a multiplicação está completa
MULT	DAD H	Somar conteúdos do par de registradores H, L a eles próprios. Esta adição multiplica os conteúdos por um fator 2. Esta multiplicação é então equivalente a um *deslocamento à esquerda* (primeiro desempenho desta operação não tem efeito resultante, já que o par de registradores começa limpo)
	RAL	Girar Acc para a esquerda através de transporte, colocando o bit multiplicador mais significativo no flag CY

(*continua*)

(*continuação*)

	JNC CHANGE COUNT	Instrução de três bytes solicitando salto (jump) para instrução cujo rótulo de endereço é CHCNT, mas o salto é condicional e ocorrerá se não houver nenhum transporte, isto é, se CY = 0. Do contrário, próxima instrução é executada
	DAD D	Somar conteúdo do par de registradores D, E ao par de registradores H, L. Registrador E guarda multiplicando (de início limpamos registrador D)
CHCNT	DCR B	Decrementar registrador B
	JNZ MULT	Instrução de três bytes solicitando salto para instrução cujo rótulo de endereço é MULT, mas o salto é condicional e ocorrerá somente se conteúdo do registrador B não for zero. Se registro de B for zero, multiplicação está completa e o resultado pode ser armazenado conforme fornecido pela instrução seguinte
	SHLD 42 00	Armazenar conteúdo do par de registradores H, L no local 42 da memória

Um resumo das instruções do 8080 é dado no Apêndice C. Incluídas na listagem estão inúmeras instruções que serão discutidas no Cap. 11.

10.18 O MICROPROCESSADOR DO TIPO 6800

Agora é evidente que constitui um importante empreendimento tornar-se bastante familiarizado com um determinado microprocessador para usá-lo eficientemente. Neste capítulo descrevemos simplesmente a maioria das instruções do 8080 e demos alguns exemplos de escrita de programa. Muito resta a ser dito. Será lembrado que há ainda dez pinos na pastilha do 8080, cuja função ainda não descrevemos. Prosseguiremos no assunto ainda no Cap. 11. Embora não seja viável discutir aqui outras pastilhas de microprocessadores com terminalidade comparável, há um bom número de diferentes microprocessadores comercialmente disponíveis. Apesar de serem bastante semelhantes, eles têm importantes e interessantes diferenças. Para dar apenas uma idéia das similaridades e diferenças, consideremos sucintamente alguns dos aspectos característicos da pastilha de microprocessador do tipo 6800.

A Fig. 10.18-1 mostra uma vista do programador do 6800 conectado a uma RAM periférica. Como o 8080, o 6800 é um dispositivo de quarenta pinos; dezesseis dos pinos fornecem um bus de endereços, e há um bus de dados de oito bits. Há uma única linha de saída de leitura/escrita ($R/\overline{W}$). Há duas linhas de entrada que aceitam formas-de-onda de relógio ϕ_1 e ϕ_2. A unidade é energizada por uma única fonte de 5 V, que usa um pino, e há dois pinos de terra ou massa. Dos quarenta pinos, dois não são usados. Ao todo consideramos 32 pinos. Os outros oito pinos são usados tal como os dez pinos no 8080 que não consideramos ainda. Adiamos este assunto até o próximo capítulo.

No 8080 a memória periférica era controlada por duas linhas, $\overline{MEMR}$ e $\overline{MEMW}$, através das quais podíamos arranjar para ler da memória, escrever na memória, ou, quando necessário, liberar o bus de dados da memória de modo que o bus possa ser usado com algum outro propósito. No 6800 há somente uma única linha de controle $R/\overline{W}$ da memória, que determina se há uma operação

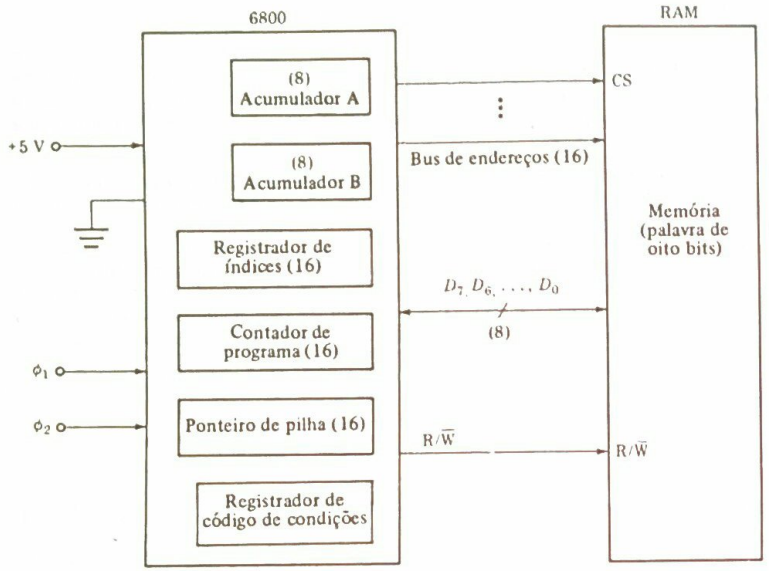


Fig. 10.18-1 Uma vista do programador do microprocessador 6800.

de leitura ou de escrita. Para liberar o bus da pastilha de memória é necessário usar um seletor de pastilha (CS) como uma linha de entrada de endereço da pastilha. Quando a linha de endereço para CS estiver em lógica 0 (ou lógica 1, quando houver uma entrada $\overline{CS}$), o bus estará livre.

O 6800 tem dois acumuladores A e B de oito bits, em vez de um acumulador como no 8080. Os acumuladores servem à mesma função genérica que no 8080. Nas operações que envolvem a ULA, sejam aritméticas ou lógicas, os acumuladores inicialmente guardam um dos operandos e, finalmente, guardam o resultado da operação. Os acumuladores do 6800 respondem às instruções de deslocar e girar similares às do 8080. O acumulador também responde a uma instrução (mnemônico ABA) que soma o conteúdo dos acumuladores, deixa a soma no acumulador A e deixa o conteúdo inicial de B inalterado. A disponibilidade dos dois registradores de acumulador um tanto versáteis constitui uma vantagem do 6800, que, considerada com outros aspectos, serve para compensar a falta dos pares B, C e D, E de registradores de propósitos gerais no 8080. Como o 8080, o 6800 tem um contador de programa de dezesseis bits e um apontador de pilha de dezesseis bits.

10.19 MODOS DE ENDEREÇAMENTO DO TIPO 6800

No 8080, quando é necessário ter acesso a um local da memória para armazenamento ou recuperação de alguns dados, geralmente há duas maneiras de incorporar o endereço do local da memória nas instruções: (1) o endereço é incluído diretamente na instrução; por exemplo, a instrução de

três bytes LDA $\langle B_2 \rangle \langle B_3 \rangle$ carrega no acumulador os dados no local da memória cujo endereço é B_3B_2, e (2) o endereço do local da memória é achado no par de registradores H, L; por exemplo, a instrução MOV A, M de um byte move para o acumulador o conteúdo do local da memória cujo endereço é achado no par de registradores H, L. No 6800 foram incorporados alguns modos adicionais de endereçamento da memória.

O primeiro modo de endereçamento disponível no 6800 é chamado *endereçamento direto*. Deste modo, o endereço da memória se constitui numa parte da instrução, mas a hipótese é feita de que o byte de ordem mais alta do endereço da memória é 00 (hex). Daí, uma vez que a instrução necessita apenas especificar o byte de endereço de ordem mais baixa, uma tal instrução com endereçamento direto é uma instrução de dois bytes em vez de três bytes. Uma limitação do modo de endereçamento direto é que somente a parte da memória de endereço 0000 (hex) a 00FF é acessível. Como o 8080, o 6800 também usa instruções em que são empregados três bytes de modo que o endereço global possa ser feito parte da instrução. No 6800 este modo de endereçamento recebe o nome *estendido* para distingui-lo do endereçamento direto.

O 6800 também incorpora uma facilidade de *endereçamento indexado*, que não é diretamente encontrado no 8080. Este modo de endereçamento usa o *registrador de índice* de dezesseis bits, que é o terceiro registrador de dezesseis bits na Fig. 10.18-1. Uma instrução que usa endereçamento indexado é uma instrução de dois bytes. O local da memória endereçado é depois calculado adicionando-se o segundo byte da instrução ao conteúdo do registrador de índices. Por exemplo, no mnemônico LDA B N, X da linguagem do montador, o X indica que é pretendido o endereçamento indexado. A instrução significa que o acumulador B deve ser carregado do local da memória calculado pela adição do número N de um byte ao conteúdo do registrador de índices. Se o conteúdo do registrador de índices fosse, por exemplo, 2615 (hex) e $N = 32$ (hex), o local da memória a que se tem acesso seria 2615 + 32 = 2647. O endereçamento indexado é útil porque ele nos possibilita estender a vantagem do endereçamento direto, isto é, instruções de dois em vez de três bytes, a qualquer porção da memória. Primeiro colocamos o registrador de índices em um endereço A_I em cuja vizinhança necessitamos ter acesso. Depois disto, uma instrução de dois bytes fornecerá acesso a todos os locais desde A_I até A_I + FF (hex).

10.20 O REGISTRADOR DE CÓDIGO DE CONDIÇÃO DO 6800

O registrador de código de condição do 6800 é comparável ao registrador de estado do 8080. Como no 8080 e como indicado na Fig. 10.20-1, o registrador de código de condição é um registrador de oito bits. Dois dos flip-flops no registrador, 6 e 7, estão permanentemente no estado set e servem apenas para preencher o registrador com oito bits. O bit 4 é um bit de máscara de interrupção que não discutiremos aqui. O bit C (transporte/empréstimo) é o mesmo que o bit CY do 8080. O bit Z (zero) é o mesmo que o bit Z do 8080. O bit N (negativo) é tornado set quando o bit b_7 de ordem mais alta na saída da ULA for $b_7 = 1$, e é o mesmo que o bit S (sinal) no 8080. O bit H (meio transporte) é o mesmo que o bit AC de transporte auxiliar no 8080 e indica um transporte de b_3 a b_4. O 6800 não tem nenhum bit de paridade, mas, no entanto, ele tem um bit V (*overflow* = estouro). Este bit de estouro – particularmente sua distinção do bit de transporte – merece comentários.

Se usarmos um registrador de um byte para guardar um número sem referência ao sinal, o número $b_7b_6, \ldots, b_0$ tem a extensão de 0 a 255 (decimal). Se somarmos os números nos dois registradores e a soma exceder 255, haverá um transporte do bit mais significativo. Este transporte

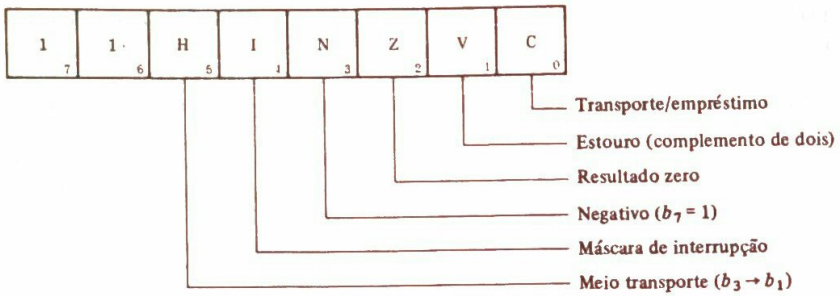


Fig. 10.20-1 A condição do registrador de código do 6800.

irá tornar set o flag CY no 8080 (ou o flag C no 6800), e, como vimos, a indicação de saída de transporte nos possibilitará fazer edição de múltiplos bytes.

Suponhamos, agora, que nos propomos a usar nosso registrador para reter ou guardar números positivos e negativos de acordo com a representação de complemento de dois. Então escrevamos o número como $sb_6b_5, \ldots, b_0$, em que o bit b_7 tornou-se o bit de sinal e b_6 é agora o bit numérico mais significativo. A extensão da representação do número agora vai de $+127$ a -128. Se somarmos os números nos dois registradores e os números somados forem de sinais diferentes, a soma estará dentro da extensão ou faixa do registrador. Se somarmos números de mesmos sinais e a soma for menos positiva do que $+127$ ou menos negativa do que -128, novamente tudo está em ordem. Mas, se somarmos números de mesmos sinais e a soma estiver fora da faixa de $+127$ a -128, o resultado apresentará erro. Por exemplo

$$\begin{array}{rl} +98 = & 01100010 \\ \underline{+54 = } & \underline{00110110} \\ +152 \neq & 10011000\ (= -104) \end{array} \qquad \begin{array}{rl} -98 = & 10011110 \\ \underline{-54 = } & \underline{11001010} \\ -152 \neq & 01101000\ (= +104) \end{array}$$

Naturalmente, estando longe de usar registradores mais longos, não há nada que possamos fazer quanto ao assunto. Mas certamente acharíamos útil sermos alertados para o fato de que ocorreu um erro. Note que um transporte não pode ser usado como uma indicação de erro. No exemplo acima, o resultado está errado em ambos os casos. Não há nenhum transporte no primeiro caso, mas há no segundo.

Agora é deixado como exercício ao estudante (Prob. 10.20-1) verificar que o seguinte esquema fornece uma indicação de que ocorreu um erro, isto é, um erro de estouro fora do alcance ou faixa do registrador. Gerar uma variável lógica C_n que é $C_n = 1$ quando houver um transporte da posição de bit (b_6) numérico de ordem mais alta e $C_n = 0$ de outro modo. Gerar uma variável lógica C_s que é $C_s = 1$ se houver um transporte da posição de bit de sinal e $C_s = 0$ de outro modo. Gerar a variável V dada por

$$V = C_n \oplus C_s \tag{10.20-1}$$

Se $V = 1$, há um erro de estouro. O flag V no registrador de código de condição do 6800 é determinado precisamente desta maneira.

CAPÍTULO 11

OPERAÇÕES DE ENTRADA-SAÍDA (I/O)

Até o presente ponto evitamos intencionalmente qualquer consideração de comunicação entre a unidade microprocessador e o mundo exterior à pastilha (com a única exceção da transferência de informação de um lado para outro entre o microprocessador e a memória). Dificilmente ainda necessitamos bater no ponto de que tal comunicação é de importância fundamental. Se a pastilha do microprocessador com sua memória deve ser usada simplesmente para efetuar uma computação ou manipulação de dados, no mínimo necessitamos de um mecanismo para introduzir dados e instruções no início e ler na saída o resultado quando a computação ou manipulação tiver sido concluída. Mais genericamente, o sistema microprocessador pode igualmente ser destinado não somente para computar e manipular dados, mas também para controlar as operações de algum processo externo. Num tal caso, o sistema microcomputador pode ter que aceitar novos dados em ocasiões imprevisíveis durante o próprio tempo em que alguma computação ou manipulação estiver em andamento, e o sistema pode também ser solicitado a fornecer sinais de controle de saída em tempos igualmente imprevisíveis. A questão é um tanto complicada pelo fato de que o hardware externo com que a comunicação é requerida (chamado *periféricos*) pode também compreender dispositivos que não sejam capazes de operar em sincronismo com a forma-de-onda de relógio do microprocessador ou em velocidades comparáveis à do microprocessador. Estes periféricos particulares podem consistir em chaves e botões de pressão mecânicos, teclas de máquinas de escrever, dispositivos de fita magnética etc. Neste capítulo discutiremos a comunicação entre microcomputador e periféricos. O assunto é geralmente chamado comunicação de *entrada-saída* (I/O = E/S).

11.1 GERAÇÃO DE SINAIS DE CONTROLE I/O NO 8080

A fim de entender como o 8080 torna disponíveis sinais de controle necessários para comunicação I/O, é necessário descrever sucintamente alguns detalhes da operação interna da pastilha do microprocessador.

Como já observamos, a temporização do 8080 é estabelecida por duas formas-de-onda de relógio ϕ_1 e ϕ_2, que devem ser fornecidas à MPU a partir de uma fonte externa. Estas formas-de-onda de relógio são mostradas nas Figs. 11.1-1*a* e *b*. O *tempo do ciclo* t_{cy} (o período) das duas formas-de-onda é o mesmo. No 8080, t_{cy} no mínimo é de quase 0,5 μs, correspondendo a uma freqüência de 2 MHz, e no máximo de quase 2 μs, correspondendo a uma freqüência de 0,5 MHz. Os ciclos de serviço das formas-de-onda, isto é, tempos relativos gastos nos níveis de tensão mais altos e mais baixos, não são estritamente especificados, exceto que, tipicamente, $t_{\phi_1} = \frac{2}{9} t_{cy}$ e $t_{\phi_2} = \frac{5}{9} t_{cy}$. Os tempos em que ϕ_1 e ϕ_2 se alongam no nível de tensão superior não se superpõem. O crescimento de ϕ_2 é retardado em cerca de 80 ns ou mais após o aumento em ϕ_1. Estas formas-de-onda de relógio comandam o controlador da MPU, e o controlador avança de estado a estado em sincronismo com estas formas-de-onda de relógio. Portanto, o tempo t_{cy} é chamado *tempo de estado*, como indicado. Pois, como observamos, um controlador e, por conseguinte, a maquinaria que ele controla avançam de um estado para o próximo a cada ciclo de relógio.

Toda a seqüência de estados através da qual o controlador passa no processamento de uma instrução é chamada *ciclo de instrução*. Incluídos nos ciclos de instrução estão os estados associados à busca da instrução e os estados associados à execução da instrução. Consideremos, por exemplo, a instrução ADD r, que soma o número no registrador r ao número no acumulador. A instrução requer uma única comunicação com um dispositivo externo à MPU. O dispositivo externo é a memória, e a comunicação tem o propósito de buscar as instruções de único byte. Revela-se que três estados são usados na extração da instrução da memória e um estado adicional

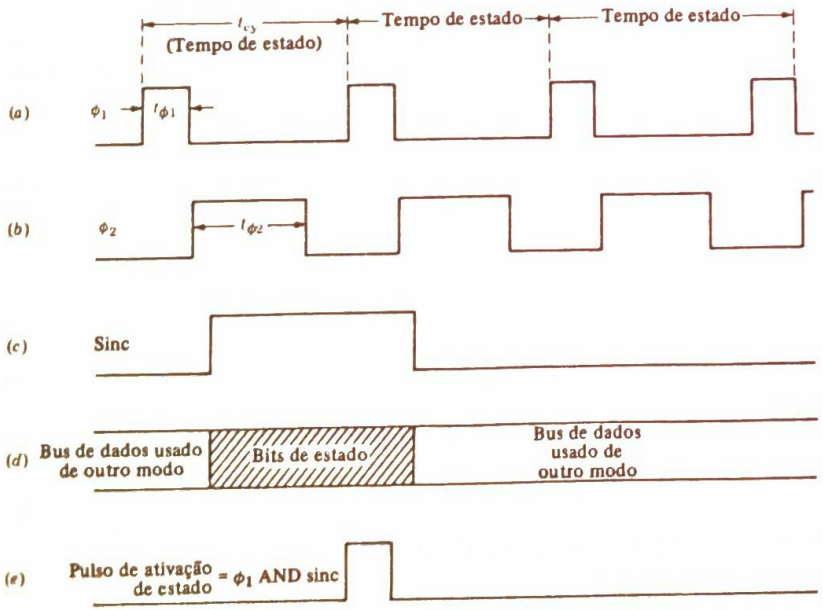


Fig. 11.1-1 (*a*) e (*b*) Formas-de-onda de relógio para a MPU 8080. (*c*) A forma-de-onda de saída de sincronismo. (*d*) Temporização de bits de estado nas linhas de dados. (*e*) Temporização do pulso de ativação de estado.

serve para efetuar a adição. Estes quatro estados, que juntos efetuam uma comunicação com o dispositivo externo (a memória) e depois processam os dados comunicados, são chamados em conjunto *ciclo de máquina*. Assim, a instrução ADD r é executada num ciclo de máquina que consiste em quatro estados. Genericamente, então, cada seqüência de estados associada com a efetivação de uma comunicação com um dispositivo externo e depois o processamento dos dados transferidos é um ciclo de máquina, e um ciclo de instrução é constituído de um ou mais ciclos de máquina. Deve haver pelo menos um ciclo de máquina, porque toda instrução requer uma operação de busca, que por sua vez requer um ciclo de máquina. No máximo, um ciclo de instrução pode consistir em cinco ciclos de máquina. Um exemplo de uma tal instrução com cinco ciclos de máquina é a instrução SHLD, que armazena o conteúdo do par de registradores H, L em dois locais sucessivos da memória. A instrução é uma instrução de três bytes; daí, a busca destes três bytes requer três ciclos de máquina; então são requeridos dois ciclos de máquina adicionais, um para armazenar o conteúdo do registrador L e outro para armazenar o conteúdo de H.

Durante o primeiro estado de cada ciclo de máquina, a pastilha 8080 torna disponível para o seu próprio mundo exterior informação concernente à operação que ela está no processo de realização. Esta informação é usada para controlar e orientar a atividade dos dispositivos periféricos, sejam eles pastilhas de memória ou outros componentes. A forma deste controle é mostrada nas Figs. 11.1-1 e 11.1-2.

Um dos terminais de saída do 8080, ao qual ainda não fizemos referência, é o terminal de saída *sinc* (sincronização). Neste terminal o 8080 gera a forma-de-onda mostrada na Fig. 11.1-1*c*. Esta forma-de-onda está normalmente em lógica zero, exceto que, começando no *primeiro* estado de cada ciclo de máquina, ela aumenta durante um tempo de estado para lógica 1. O aumento na forma-de-onda de sincronização é cronometrado para ocorrer em um curto tempo de atraso depois da borda frontal da forma-de-onda de relógio ϕ_2, e a queda em sincronismo ocorre num atraso comparável após a próxima elevação em ϕ_2. (O atraso é inferior a 120 ns.) Agora, normalmente as oito linhas de dados D_0 até D_7 na Fig. 11.1-2 são usadas para transferir dados de ou para a pastilha da UCP. No entanto, na duração do intervalo de sincronização, as linhas de dados são usadas com um propósito diferente. Durante este intervalo, a UCP coloca nestas oito linhas de dados oito bits que são chamados bits de estado, já que eles servem para dizer ao mundo exterior a respeito do estado de coisas dentro da pastilha, isto é, que operação a UCP está pretendendo efetuar.

Os bits de estado (ou alguns deles) podem também ser requeridos durante todo o ciclo de máquina. No entanto, naturalmente, as linhas de dados serão necessárias durante o mesmo ciclo de máquina para transferir dados de um lado para outro entre a UCP e a memória e entre a UCP e os periféricos. Daí necessitarmos guardar os bits de estado enquanto liberamos as linhas de dados. Como pode ser visto na Fig. 11.1-1*d*, os bits de estado são colocados nas linhas de dados somente no tempo durante o qual a *sincronização* = 1. Em todos outros instantes as linhas de dados estão disponíveis para outros propósitos. Para manter os bits de estado disponíveis, mesmo após a *sincronização* retornar à lógica 1, usamos um latch, como indicado na Fig. 11.1-2. O pulso que opera o latch, apanhando e retendo deste modo os bits de estado, é chamado *pulso de ativação de estado* (*status strobe* = STSTB); é gerado submetendo-se a uma operação AND a forma-de-onda de sincronização e a forma-de-onda ϕ_1 de relógio. Como pode ser visto na Fig. 11.1-1*e*, este pulso de ativação de estado torna-se ativo satisfatoriamente após o tempo em que os bits de estado são primeiro colocados nas linhas de dados. Por conseguinte, há bastante tempo para que os bits de estado se tornem validamente estabelecidos. E o pulso

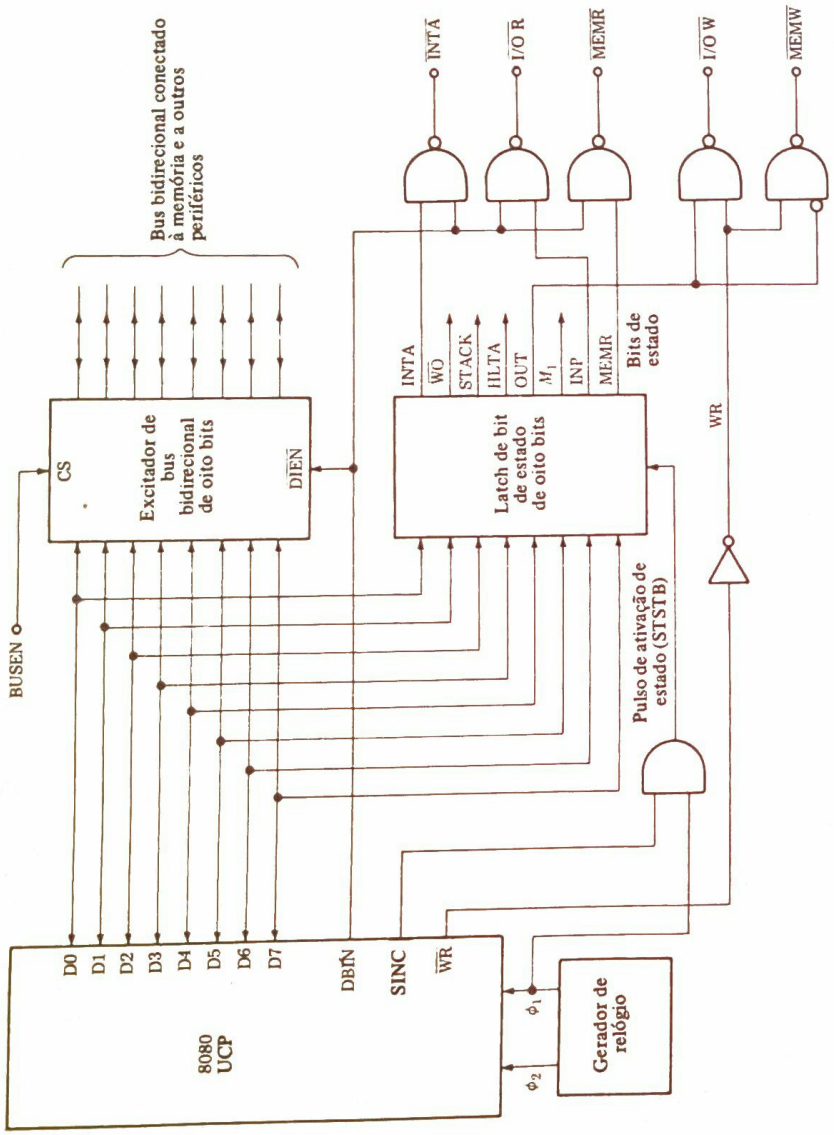


Fig. 11.1-2 As interligações entre o 8080, a memória intermediária de bus de dados bidirecional e o latch de bits de estado.

de ativação de estado torna-se inativo satisfatoriamente antes do tempo em que as linhas de dados devem ser usadas com outros propósitos.

Consideremos agora como os bits de estado disponíveis na saída do latch de estado são combinados com outros sinais do 8080 para gerar sinais a serem usados para controlar a comunicação entre o 8080 e a memória e entre o 8080 e os outros periféricos. Primeiro anotemos na Fig. 11.1-2 dois sinais de controle adicionais fornecidos pelo 8080. Um é o sinal de saída $\overline{WR}$ ($\overline{WRITE}$). Quando este sinal (baixo-ativo) estiver em lógica 0, ele indica que a UCP colocou dados nas linhas de dados e que os dados são estáveis. Daí, se tivermos uma pastilha de memória conectada às linhas de dados e colocarmos nos níveis apropriados os terminais de controle da pastilha de memória (R/$\overline{W}$ e OD, output disable), os dados serão *escritos* na memória. O segundo sinal de saída de controle é DBIN (data bus in). Quando DBIN = 1, a pastilha da UCP indica que o bus de dados está no modo de entrada; isto é, a pastilha da UCP está processando a transferência de dados (por exemplo, da memória) para dentro de um dos registradores internos à pastilha 8080. Correspondentemente, quando DBIN = 0, a transferência de dados é na outra direção, isto é, fora da UCP.

Agora consultemos novamente a Fig. 10.2-1, onde mostramos sinais de controle $\overline{MEMR}$ e $\overline{MEMW}$ conectados respectivamente nos terminais de entrada OD e R/$\overline{W}$ de uma pastilha de memória. Pode agora ser verificado (Prob. 11.1-1) que, se complementarmos DBIN para tornar disponível $\overline{DBIN}$, então $\overline{DBIN}$ pode servir como o sinal $\overline{MEMR}$ que é aplicado à entrada de memória OD e $\overline{WR}$ pode servir como o sinal $\overline{MEMW}$ aplicado à entrada de memória R/$\overline{W}$. No entanto, veremos na próxima seção que no 8080 os projetistas preferem não usar $\overline{DBIN}$ e $\overline{WR}$ diretamente para fornecer $\overline{MEMR}$ e $\overline{MEMW}$. Na próxima seção continuaremos também nossa discussão da Fig. 11.1-2.

11.2 ENTRADA-SAÍDA MAPEADA NA MEMÓRIA E ISOLADA

Conforme temos observado, os dados necessitam ser comunicados entre UCP e memória e entre UCP e outros dispositivos periféricos. Estes outros dispositivos periféricos podem ser tratados simplesmente como pastilhas de memória adicionais. Tal tratamento é indicado na Fig. 11.2-1, onde a UCP está em comunicação com a pastilha de memória e dois outros periféricos. As linhas de endereço estão conectadas às entradas de endereço da memória. Um código de endereço é reservado para distinguir cada periférico. Quando um periférico estiver endereçado, a saída de seu codificador estará em lógica 1 e o periférico estará habilitado. Depois podemos transferir dados para ou do periférico, dependendo dos níveis lógicos de $\overline{MEMR}$ e $\overline{MEMW}$. (Novamente indicamos, como visto na Fig. 11.1-2, que $\overline{MEMR}$ e $\overline{MEMW}$ não são fornecidos diretamente pela UCP.) Tendo usado dois endereços para os dois periféricos na Fig. 11.2-1, temos ainda disponíveis $2^{16} - 2$ endereços para os locais da memória. A decodificação da memória é feita direto na pastilha de memória. Muito freqüentemente, conforme notado na figura, um decodificador inteiramente separado do periférico será requerido para cada periférico. Uma vez que os decodificadores devem acomodar dezesseis bits, eles podem ser muito caros e podemos preferir evitá-los. Como um meio de eliminar os decodificadores, consideremos o seguinte arranjo. Admitamos que a pastilha de memória esteja equipada com duas entradas $\overline{CS}$ ($\overline{CHIP\ SELECT}$) de modo que, quando qualquer uma destas entradas estiver em lógica 1, a memória não é selecionada. (Estas entradas $\overline{CS}$ não são mostradas na Fig. 11.2-1.) Conectemos uma linha de endereço, seja A_{15} diretamente à entrada HABILITA (*ENABLE*) do periférico 1 e a uma entrada $\overline{CS}$ da memória e seja A_{14} diretamente à

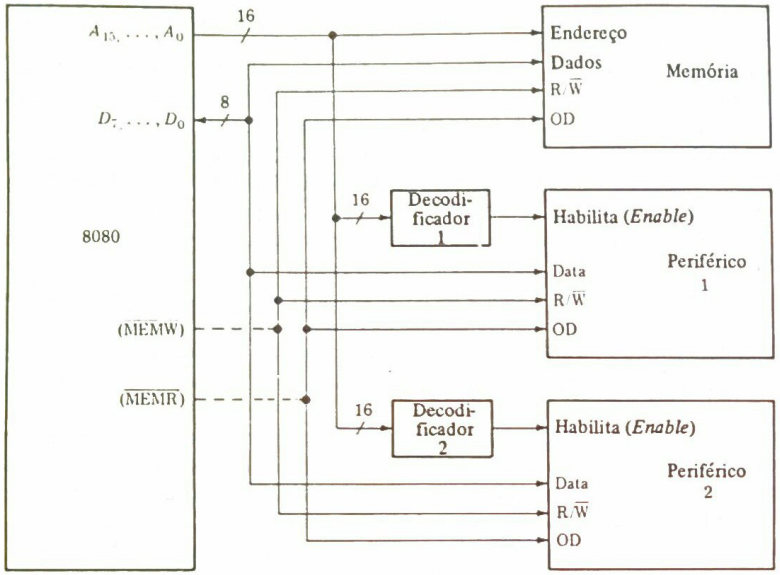


Fig. 11.2-1 Entrada-saída mapeada na memória em que periféricos são tratados como extensões da memória.

entrada HABILITA (*ENABLE*) do periférico 2 e à outra entrada $\overline{CS}$. Usemos as outras quatorze linhas de endereço $A_{13}, \ldots, A_0$ para endereçamento da memória. Então podemos dispensar os decodificadores. Quando $A_{15} = 1$, ou quando $A_{14} = 1$, endereçaremos respectivamente os periféricos 1 ou 2. Quando $A_{15} = A_{14} = 0$, o endereço refere-se a um local da memória. Naturalmente, teremos reduzido o número de locais endereçáveis de $2^{16} - 2$ para 2^{14}. Se a capacidade de endereçamento for ainda adequada, podemos achar que este esquema de eliminação de decodificadores é bastante aceitável.

Os dispositivos periféricos a que os dados são "remetidos" ou dos quais os dados são recebidos são geralmente denominados pares de terminais (*ports*). O esquema que acabamos de escrever para transferir dados entre qualquer par de terminais e a UCP, tratando qualquer par de terminais na maneira de uma memória, é chamado *entrada-saída de memória mapeada*. Trata-se de um esquema totalmente viável que pode ser usado com o 8080. Ele torna disponíveis, com o propósito de comunicação entre pares de terminais e UCP, todas as instruções que movem dados entre a memória e a UCP. I/O mapeada na memória tem a desvantagem, no entanto, de infringir um pouco sobre o número de endereços disponíveis para endereçamento da memória. Ela tem uma outra desvantagem: uma vez que um endereço da memória ou do par de terminais deve ser fornecido, um endereço de dezesseis bits deve estar incluído ou implícito na instrução. Em alguns casos, tal como LDA (load accumulator direct), o endereço está incluído na instrução de três bytes. Em outros casos, tais como MOV A, M (move to accumulator from memory), a própria instrução envolve apenas um byte, mas o endereço está no par de registradores H e em algum ponto foram requeridas etapas do programa para carregar o endereço no par de registradores H.

Não se deve imaginar que estas desvantagens são muito sérias. Inúmeros microprocessadores, por exemplo o 6800, não fornecem nenhum outro método de entrada-saída. Ainda os projetistas do 8080 decidiram que seria mais simples e mais conveniente tornar disponível um esquema de I/O adicional que isolasse o acesso à memória do acesso aos outros pares de terminais. Eles, portanto, incluíram a facilidade desta I/O *isolada* na pastilha.

Para ver como opera a I/O isolada, veja a Fig. 11.1-2. Note que um dos bits de estado retido pelo latch é denominado OUT. Seu nome completo é OUTPUT WRITE, para distingui-lo como uma operação de escrever habilitar num *par de terminais de saída* e não como uma de escrever memória, que escreveria na memória. Quando examinamos os detalhes fornecidos pelo fabricante a respeito do trabalho do 8080, achamos que OUT está em nível lógico 1 quando e somente quando a instrução a ser executada é uma operação de escrita num par de terminais diferente da memória, usando I/O isolada. Agora observamos também que, quando OUT = 1, o terminal marcado $\overline{I/O\,W}$ (input-output write) tem o valor lógico $\overline{I/O\,W} = \overline{WR}$, enquanto o terminal $\overline{MEMW}$ está mantido fixo na lógica 1. Quando OUT = 0, então $\overline{I/O\,W} = 1$ e $\overline{MEMW} = \overline{WR}$. Assim temos dois terminais separados para uso na escrita: $\overline{MEMW}$, que se torna ativo quando queremos escrever na memória, e $\overline{I/O\,W}$, que se torna ativo quando queremos escrever em algum outro periférico diferente da memória.

Todas as instruções introduzidas até agora que se referem à comunicação entre UCP e periféricos fornecem apenas comunicação com a memória. Para permitir o uso da I/O isolada para transferir dados para fora da UCP, o 8080 fornece uma instrução de dois bytes especial:

OUT
$\langle B_2 \rangle$

O segundo byte $\langle B_2 \rangle$ é o endereço do par de terminais envolvido. Este endereço de oito bits aparece nas oito linhas de endereço menos significativas $A_7, \ldots, A_0$ e é duplicado nas oito linhas $A_{15}, \ldots, A_0$ de endereço mais significativo. Uma vez que o endereço tem apenas oito bits, somente 256 pares de terminais podem ser endereçados, mas é um tanto difícil imaginar uma situação em que este número não seria mais do que adequado. Quando a instrução OUT $\langle B_2 \rangle$ for executada, os dados colocados no bus de dados são o conteúdo do acumulador. Assim, a instrução OUT $\langle B_2 \rangle$ é um tanto limitada em comparação com a instrução disponível para uso com I/O de memória mapeada. Na I/O de memória mapeada podemos envolver qualquer registrador, não apenas o acumulador.

Voltando novamente à Fig. 11.1-2, observamos que ela inclui um excitador de bus bidirecional com oito bits. O excitador (acoplador) de bus consiste em oito pares de amplificadores separados (intermediários). Dois destes pares são mostrados na Fig. 11.2-2. Quando CS estiver em CS = 0, todos os amplificadores estão inoperantes, isto é, as saídas vão para o estado de alta impedância como em saídas de três-estados, e o bus conectado à memória e aos outros pares de terminais está inteiramente isolado da UCP. Quando CS = 1, ou o amplificador superior ou o amplificador inferior de cada par torna-se operante. Quando CS = 1 e $\overline{DIEN}$ (data in to bus enable) é $\overline{DIEN} = 0$, o amplificador superior sozinho de cada par é tornado operativo para estabelecer um caminho de transmissão para os dados da UCP até o bus. Quando $\overline{DIEN} = 1$, o amplificador inferior torna-se operativo, permitindo transmissão do bus para a UCP. Os buffers não efetuam nenhuma lógica com os dados que eles transmitem. A amplificação fornecida, se houver, geralmente não é importante. Em vez disto, cada amplificador buffer opera com corrente de entrada relativamente pequena e é capaz de fornecer relativamente muito mais correntes de saída. Daí, interpondo o buffer entre o bus e a UCP, garante-se que a UCP não há de impor

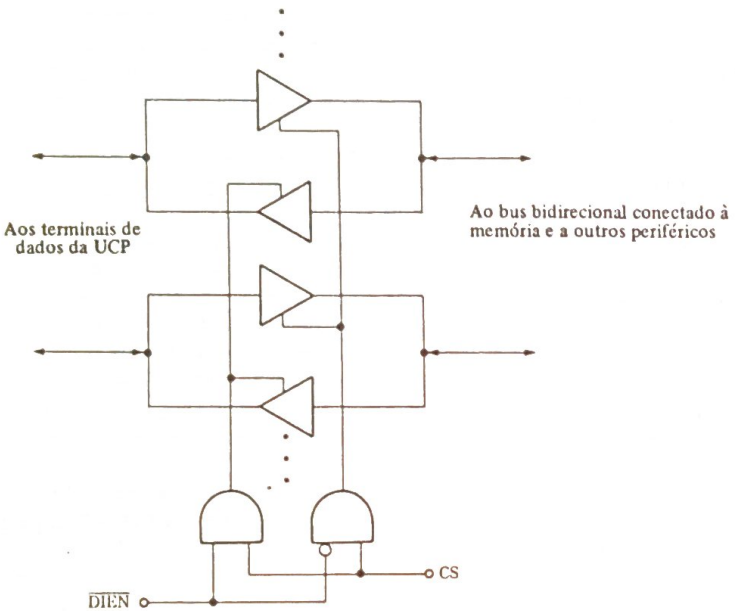


Fig. 11.2-2 Diagrama lógico de um excitador (*driver*) de bus bidirecional.

uma carga excessiva sobre um par de terminais de entrada e vice-versa. Na Fig. 11.1-2 notamos que um sinal de entrada BUSEN (bus enable) foi aplicado à entrada CS do acoplador de bus. Este sinal BUSEN não é fornecido pela UCP e, quando disponível, é normalmente fornecido de algum outro lugar do sistema microprocessador, talvez mesmo por uma chave operada manualmente. Em qualquer caso, quando BUSEN = 0, o bus está liberado da UCP e pode então ser usado na comunicação entre periféricos sem envolver a UCP. Um tal tipo de comunicação envolve a transferência direta de dados entre a memória e algum outro periférico, um tipo de transferência denominado DMA (direct memory acess).

Podemos agora resumir nossa discussão da instrução OUT $\langle B_2 \rangle$ da seguinte maneira. Durante esta execução da instrução devemos ter BUSEN = 1. Também DBIN = $\overline{\text{DIEN}}$ = 0, de modo que o excitador de bus é ajustado (set) para transferência de dados fora da UCP para o bus. Previamente alocados para a execução da instrução, os dados e o endereço dos pares de terminais são postos nas linhas de dados e de endereços e o bit de estado OUT é estabelecido em lógica 1. Depois dos bits de endereço terem se tornado estavelmente estabelecidos, isto é, válidos, a saída $\overline{\text{WR}}$ da UCP torna-se ativa ($\overline{\text{WR}}$ = 0) e a saída $\overline{\text{I/O W}}$ vai para lógica 0, escrevendo deste modo o conteúdo do acumulador no par de terminais de saída. O tempo total requerido para buscar (fetch) e executar a instrução OUT $\langle B_2 \rangle$ é o tempo de dez estados (10 X 0,5 μs = 5 μs numa taxa de relógio de 2 MHz). O sinal $\overline{\text{WR}}$ é ativo apenas durante o último estado.

Consultando novamente a Fig. 11.1-2, notamos que o latch de estado também fornece uma saída MEMR. Esta saída está em lógica 1 sempre que uma operação de leitura da memória tiver

que ser efetuada. Durante uma tal operação de leitura, DBIN também estará em lógica 1, de modo que terá $\overline{\text{MEMR}}$ = 0, fornecendo deste modo um sinal de leitura para a memória. Quando MEMR = 1, temos INP (input) em lógica 0. A UCP fornece uma instrução

IN
$\langle B_2 \rangle$

e, quando esta instrução estiver sendo executada, e somente então, a INP (entrada) da saída do latch de estado vai para lógica 1, enquanto MEMR permanece em lógica 0. Durante esta instrução, DBIN também vai para lógica 1, de modo que o sinal $\overline{\text{I/O R}}$ torna-se $\overline{\text{I/O R}}$ = 1, fornecendo assim um sinal para ler da I/O isolada.

Em resumo, então, as interligações indicadas na Fig. 11.1-2 fornecem sinais $\overline{\text{MEMR}}$ e $\overline{\text{MEMW}}$ que se tornam ativos respectivamente quando uma operação de leitura ou escrita da *memória* deve ocorrer. Tal será o caso sempre que uma operação de busca estiver sendo executada ou qualquer instrução estiver sendo executada que requeira acesso à memória. Mas quando quisermos ter acesso a qualquer par de terminais que não seja a memória, as instruções que podem ser usadas são OUT $\langle B_2 \rangle$ ou IN $\langle B_2 \rangle$. Estas duas instruções ativam as saídas $\overline{\text{I/O W}}$ e $\overline{\text{I/O R}}$, respectivamente, e permitem escrita ou leitura da I/O isolada.

Uma questão concernente à temporização das saídas $\overline{\text{I/O W}}$ e $\overline{\text{I/O R}}$ é esclarecida na Fig. 11.2-3. Estes sinais de controle de saída são ativos apenas durante o último dos dez estados sobre os quais as instruções OUT e IN se estendem. Quando a freqüência de relógio for 2 MHz, um estado dura 0,5 μs e, correspondentemente, como aparece na figura, o pulso $\overline{\text{I/O W}}$ ou $\overline{\text{I/O R}}$ usado para controlar periféricos também dura 0,5 μs. No entanto, um endereço válido é estabelecido nas linhas de endereço $A_7, A_6, \ldots, A_0$ (e duplicado em $A_{15}, A_{14}, \ldots, A_8$) cerca de 0,8 μs antes do pulso de controle tornar-se ativo. Desta maneira, permitimos tempo para que os bits de endereço se propaguem através de decodificadores onde requerido e para o periférico endereçado reconhecer adequadamente o que está sendo escolhido.

Posteriormente, teremos ocasião de considerar os sinais INTA e $\overline{\text{INTA}}$, que aparecem na Fig. 11.1-2. Isto ainda deixa quatro sinais $\overline{\text{WO}}$, STACK, HLTA e M_1. Estes sinais não são normalmente usados em simples sistemas microprocessadores, mas servem a um propósito útil no teste de um sistema para determinar se ele está funcionando adequadamente e acham aplicação quando forem empregadas memórias dinâmicas. Não os consideraremos mais, exceto para observar os seguintes detalhes: $\overline{\text{WO}}$ = 0 indica que ou uma operação de *escrever* memória ou uma operação

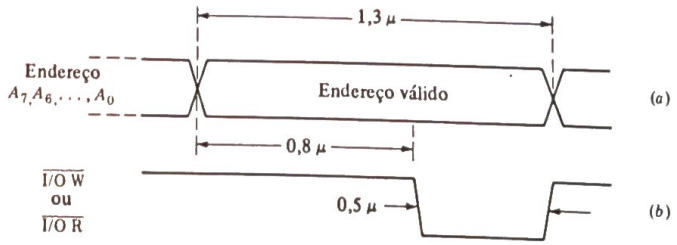


Fig. 11.2-3 A temporização do sinal I/O W ou I/O R relativa ao intervalo durante o qual um endereço válido é disponível no bus de endereços.

de *saída* de I/O isolada está em andamento. STACK = 1 indica que o bus de endereços retém o endereço da pilha como fornecido pelo apontador de pilha. HLTA = 1 quando a UCP está reconhecendo que ela parou a operação em resposta a uma instrução para fazê-lo. Finalmente, M_1 fornece um sinal para indicar que a UCP está no ciclo de busca (*fetch*) para o primeiro byte de uma instrução.

11.3 USO DE INSTRUÇÕES IN E OUT

Suponhamos que, em certo ponto de um programa, é gerado um resultado numérico que é de interesse imediato para o operador humano, que gostaria, então, de ter tal resultado disponível. Especificando, admitamos que o resultado seja um número decimal de dois dígitos representado na UCP como dois números BCD combinados em um byte. Então, usando a instrução OUT $\langle B_2 \rangle$ em conjunção com o hardware adicional mostrado na Fig. 11.3-1, o resultado pode se tornar disponível num indicador visual. Primeiro, o byte que retém o resultado seria movido para o acumulador da UCP, se ele já não estivesse localizado neste registrador. A seguir, envolveríamos a instrução OUT $\langle B_2 \rangle$. O byte $\langle B_2 \rangle$ de endereço, encontrado quer nas linhas de endereço baixo, quer nas de endereço alto, seria arranjado para ser o endereço que traria para lógica 1 a saída do decodificador de endereço. Depois do endereço e dos dados terem sido validamente estabelecidos, a instrução OUT faria com que a linha $\overline{\text{I/O W}}$ se tornasse ativa, isto é, fosse para lógica 0, e a saída da porta AND então habilitaria o latch de oito bits brevemente e depois retornaria o fatch para a condição desabilitada. Durante o breve período de habilitação, os dados de interesse seriam travados no latch e a UCP, depois disto, estaria livre para prosseguir suas outras tarefas. O observador humano estaria agora livre para ler os dados com calma nos dois indicadores visuais de sete segmentos (ver Seção 3.17). Admitamos que o hardware do indicador visual incorpore os decodificadores requeridos para converter o número BCD, cada um codificado em quatro bits binários, em tensões apropriadas para acender os segmentos adequados dos indicadores visuais. Em qualquer ponto de um programa onde queremos os dados de saída visíveis para serem atualizados, necessitamos apenas incluir a instrução OUT $\langle B_2 \rangle$.

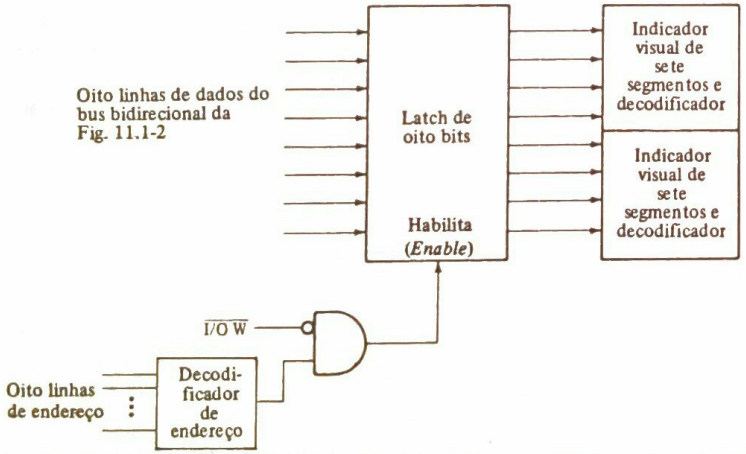


Fig. 11.3-1 Hardware requerido para permitir o uso da instrução OUT para expor o conteúdo do acumulador.

Naturalmente, na Fig. 11.3-1 um decodificador de endereços separado não é absolutamente necessário. A porta AND pode fazer a decodificação de endereços se acrescentarmos a ela entradas adicionais para acomodar tantas linhas de endereço quantas acharmos necessárias ao uso. Se, por exemplo, somente oito pares de terminais estivessem sendo usados, poderíamos providenciar para que em cada endereço do par de terminais todos os bits fossem 0, exceto um (00000001, 00000010 etc.), e uma única linha de endereço seria adequada.

Na Fig. 11.3-2 contemplamos o processo inverso da leitura de dados de algum par de terminais no acumulador da UCP. Aqui o byte de dados é estabelecido por oito chaves mecânicas. Quando uma chave está aberta, a linha de dados está em lógica 1, e quando a chave está fechada, a linha de dados está em lógica 0. No caso anterior, estávamos interessados em apanhar os dados no breve intervalo em que eles estavam disponíveis no acumulador. No caso presente, os dados estão disponíveis enquanto nós quisermos, mas não queremos usar o bus de dados mais do que o necessário. Daí usarmos uma memória intermediária unidirecional de três-estados com oito bits que é habilitada por um endereço do par de terminais e pelo sinal $\overline{\text{I/O R}}$. Quando a instrução IN $\langle B_2 \rangle$ for executada, durante o breve período (um ciclo de relógio) quando $\overline{\text{I/O R}} = 0$, os dados de entrada serão conectados ao bus, mas do contrário o bus será liberado para outros propósitos.

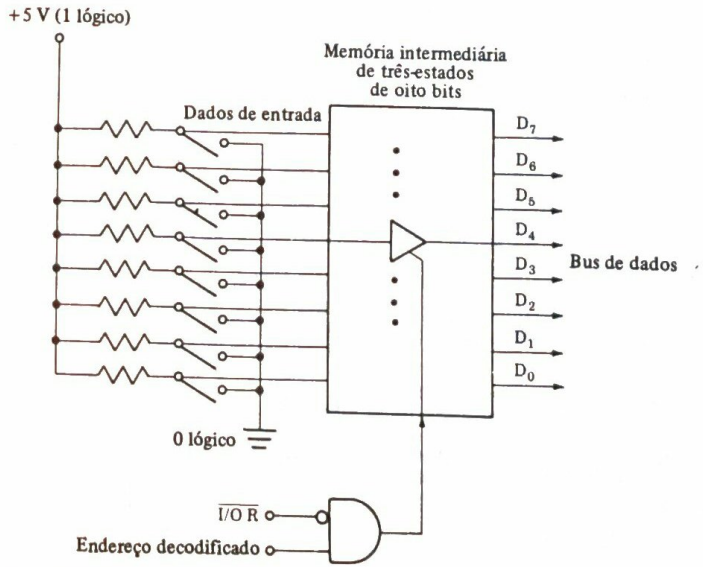


Fig. 11.3-2 Hardware requerido para permitir o uso da instrução IN para transferir dados (preparados pelas chaves) para dentro do acumulador.

11.4 UM TECLADO NÃO-CODIFICADO

Nesta seção consideramos um exemplo que é instrutivo de inúmeras maneiras. Ele ilustrará as ligações de entrada e saída, o uso das instruções IN e OUT e a substituição do software por hardware.

Contemplemos um teclado de 64 teclas que deve ser usado para introduzir dados num sistema microprocessador. Especificamente, admitamos que, em resposta à operação de uma tecla,

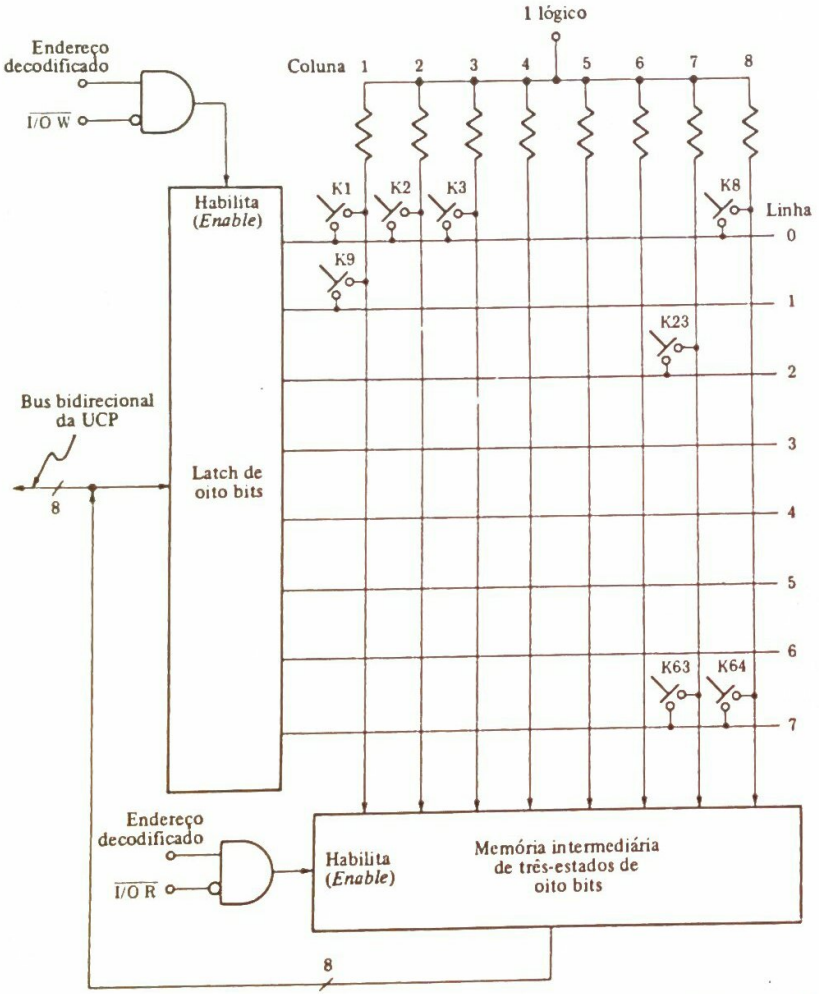


Fig. 11.4-1 Um teclado não-codificado mostrando as ligações aos buses da UCP e aos sinais de controle.

uma identificação da tecla que foi comprimida deve ser introduzida no acumulador da UCP. Depois disto, como quisermos, podemos transferir os dados para outro lugar, digamos para a memória, de modo que o acumulador estará disponível para receber mais dados. Identifiquemos as teclas pelos números de 1 a 64 e consideremos que somente uma tecla é comprimida de cada vez. Necessitamos de que, quando uma tecla for operada, o número desta tecla aparecerá no acumulador. Numeramos as teclas começando por 1 em vez de 0, de modo que 0 no acumulador possa servir como uma indicação de que nenhuma tecla foi comprimida.

Podemos providenciar para que cada tecla gere lógica 1 ou 0 numa linha, de acordo com o fato dela estar aberta ou fechada. Estas 64 linhas seriam depois aplicadas às entradas de portas OR, isto é, um codificador (ver Seção 3.16), para gerar o código binário que representa o número da tecla operada. Depois disto, necessitaríamos apenas usar a instrução IN para transferir o número da tecla codificada para dentro do acumulador. Preferimos, no entanto, dispensar o hardware de um codificador aqui e, em vez disto, gerar o número de identificação da tecla escrevendo um programa apropriado.

O arranjo que nós propomos está mostrado na Fig. 11.4-1. Temos aqui uma matriz de oito fios horizontais e oito fios verticais. Os fios não estão conectados, mas, como indicado, uma conexão entre um fio horizontal e um fio vertical pode ser feita fechando-se uma chave K1, K2 etc. Há 64 chaves K1 até K64, uma em cada intersecção (nem todas as chaves são mostradas). Os fios horizontais são comandados pelo latch de oito bits que recebe o registro do acumulador em resposta à instrução OUT $\langle B_2 \rangle$, sendo $\langle B_2 \rangle$ o endereço do latch. Os dados mantidos nos oito fios verticais serão transferidos para o acumulador em resposta à instrução IN $\langle B_2 \rangle$, onde $\langle B_2 \rangle$ é o endereço da memória intermediária de três-estados. Quando todas as chaves estiverem abertas, todas estas linhas verticais estarão em lógica 1, como deve ser notado das conexões a uma fonte de tensão de lógica 1. Suponhamos, no entanto, que uma das linhas horizontais, seja a linha 2, esteja em lógica 0, e suponhamos que a chave K23, ligando a linha 2 à coluna 7, esteja fechada. Então, por causa dos resistores na fila da coluna, a fila da coluna 7 irá para lógica 0. Em resumo, se apenas uma linha for para lógica 0 e somente uma chave estiver fechada, somente uma coluna irá para lógica 0 e estas duas filas em lógica 0 identificam a chave operada. Em verdade, a maneira pela qual as filas das colunas e das linhas foram numeradas na Fig. 11.4-1 significa que a chave fechada tem o número $N = (8 \times$ número da linha + número da coluna).

Consideremos agora um programa para conseguir nosso propósito. Primeiro, procuremos determinar se qualquer tecla foi comprimida; se não, deixemos o acumulador limpo.

Rótulo	Conteúdo	Comentário
WAIT	MVI, A 0000 0000	Mover número zero imediatamente para dentro de Acc
	OUT ⟨LATCH⟩	Saída 0000 0000 para o latch e para todas as filas das linhas
	IN ⟨BUFFER⟩	Registro de entrada de filas das colunas para Acc
	CPI 1111 1111	Comparar Acc ao padrão 1111 1111; se nenhuma tecla abaixada, temos Z = 1
	JZ ⟨WAIT⟩	Se Z = 1, nenhuma tecla abaixada, esperar, isto é, voltar para instrução rotulada "WAIT"

Se uma tecla tiver sido comprimida, teremos Z = 0, a instrução JZ não fará nada e o programa continua. O programa opera como se segue:

Rótulo	Conteúdo	Comentário
	MVI B 0000 0001	Por tentativa, supomos que tecla 1 está fechada e correspondentemente introduzimos seu número no registrador B
	MVI C 1111 1110	No registrador C estabelecemos um padrão de bits que, quando transferido para Acc e extraído para latch, trará para lógica 0 primeira linha na Fig. 11.4-1 marcada "0"
ROW	MOV A, C OUT ⟨LATCH⟩	Mover padrão de exploração para Acc e extraí-lo para o latch
	RLC	Girar à esquerda. Padrão de exploração em Acc é mudado para 1111 1101 em antecipação de que a chave ou tecla fechada pode não ser achada na linha 0. Esta operação não modifica padrão de exploração já extraído para o latch
	MOV C, A	Mover padrão de exploração modificado de volta ao registrador C porque Acc é necessário para outros propósitos
	IN ⟨BUFFER⟩	Introduzir em Acc os dados nas filas das colunas
	CPI 1111 1111	Comparar Acc ao padrão 1111 1111. Se nenhuma tecla for abaixada *na linha que está sendo testada*, temos Z = 1
	JNZ ⟨ ⟩ [COL]* ⟨ ⟩	Se Z = 0, a tecla foi abaixada. Saltar para lugar no programa rotulado "COL", que identificará coluna da tecla abaixada. Se Z = 1, nenhuma tecla foi abaixada na linha que está sendo testada. Neste caso continuar com a próxima instrução

	MOV A, B
	ADI
	⟨decimal 8⟩
	MOV B, A
		Uma vez que nenhuma tecla está abaixada na linha que acabamos de testar, somar 8 ao registrador B. Registrador B agora guarda o número da tecla na próxima linha e na coluna 1

	JMP ⟨ ⟩ [ROW] ⟨ ⟩	Saltar de volta ao lugar no programa rotulado "ROW". Próxima linha será então testada por causa da modificação do padrão de exploração já feita
COL	RAR JNC ⟨ ⟩ [END] ⟨ ⟩	Girar à direita através de transporte. Se bit movido para flag de transporte for 0, achamos tecla abaixada. Neste caso (CY = 0) acabamos
	INR B JUMP ⟨ ⟩ [COL] ⟨ ⟩	Se rotação não move um 0 para transporte, tecla na coluna testada não é abaixada. Incrementar registrador do número de teclas e voltar a testar próxima coluna
END	HLT	

* A entrada entre colchetes dá o rótulo correspondente ao endereço nos bytes não especificados acima e abaixo.

Na Fig. 11.4-1 vale a pena notar que, quando os dados são extraídos da UCP para o latch, $\overline{\text{I/O W}}$ será ativa mas $\overline{\text{I/O R}}$ será inativa. Daí, a memória intermediária de três-estados estará em seu estado de impedância elevada, e a conexão da memória intermediária ao cabo bidirecional não interferirá com a transmissão da UCP para o latch. Correspondentemente, quando os dados da memória intermediária forem introduzidos na UCP, o latch estará desabilitado.

11.5 CONTROLE DE DISPOSITIVOS PERIFÉRICOS

Vimos como os sinais $\overline{\text{I/O R}}$ e $\overline{\text{I/O W}}$ podem ser usados para ler dados dentro ou fora do acumulador. Estes sinais também podem ser usados para permitir que o microprocessador exerça uma medida de controle sobre os dispositivos periféricos mesmo quando leitura ou escrita não estiver envolvida. Assim, suponhamos que um sistema microcomputador inclua um gravador de fitas e que queremos ser capazes de incluir num programa uma instrução capaz de dar partida ou parar o gravador. O hardware mostrado na Fig. 11.5-1, junto com as instruções IN ⟨B_2⟩ e OUT ⟨B_2⟩, permitirá tal controle. Assim, suponhamos que incluímos num programa a instrução OUT ⟨B_2⟩, onde ⟨B_2⟩ é o endereço a que o decodificador na Fig. 11.5-1 está destinado a responder. Então, durante o curso da execução desta instrução, aparecerá na entrada SET do latch um pulso positivo (correspondendo ao pulso negativo na Fig. 11.2-3*b*) que fará set o latch. Correspondentemente, uma instrução IN ⟨B_2⟩ fará reset o latch. Alternativamente, o estado do latch pode ser usado para abrir ou fechar chaves eletromecânicas que podem dar partida ou parar um motor, liberar uma embreagem etc. Uma vez que, como vimos, há da ordem de 256 endereços disponíveis, podemos controlar um grande número de dispositivos para iniciar e parar movimento, para abrir e fechar válvulas, para fazer com que as teclas da máquina de escrever batam etc. Tudo o que necessitamos fazer é providenciar para que, conforme requerido, nosso programa inclua as instruções IN e OUT apropriadamente endereçadas para os latches fornecidos responderem a estas instruções.

11.6 LAÇOS DE TEMPORIZAÇÃO

Suponhamos que queremos transmitir caracteres alfanuméricos a uma máquina de escrever eletromecânica. Cada tecla da máquina de escrever pode ser equipada com o hardware da Fig. 11.5-1. Para datilografar um caractere, podemos usar uma instrução OUT ⟨B_2⟩, o byte ⟨B_2⟩ retendo o endereço da tecla a ser batida. (Se a máquina de escrever responder ao código ASCII, os bytes

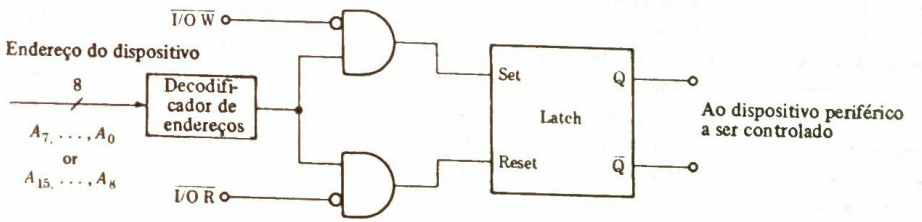


Fig. 11.5-1 Os sinais $\overline{\text{I/O W}}$ e $\overline{\text{I/O R}}$ são usados para fazer set e reset um latch endereçado. O latch, por sua vez, pode ser usado para controlar um dispositivo periférico arbitrário.

de endereço são simplesmente os códigos ASCII dos caracteres a serem impressos.) A instrução OUT $\langle B_2 \rangle$ faz set o latch, o latch energiza um eletroímã e o eletroímã puxa para baixo a tecla da máquina de escrever. Devemos agora manter o eletroímã energizado por tempo suficiente para a tecla mecânica relativamente lenta completar sua resposta. Depois disto, podemos usar uma segunda instrução IN $\langle B_2 \rangle$ para tornar reset o latch e liberar a tecla. Pode também ser necessário manter o set do latch durante muitas dezenas de milissegundos, um tempo que é muito longo comparado com a duração de um ciclo de relógio e com o tempo requerido para buscar e executar uma instrução. (O tempo do ciclo de relógio para o 8080 é na faixa de 0,5 a 2 μs, e mesmo a instrução mais complicada requer apenas dezoito ciclos de relógio.) Daí, devemos interpor um programa entre a instrução OUT e a instrução IN cujo único propósito é permitir a passagem do tempo. Tais *programas de atraso de tempo* são usados tão freqüentemente em microprocessadores com equipamento periférico associado que eles geralmente aparecem como sub-rotinas. Portanto, um programa típico que solicita a operação de uma tecla de máquina de escrever seria:

Rótulo	Conteúdo	Comentário
	OUT $\langle B_2 \rangle$	Fazer set latch para abaixar a tecla endereçada
	CALL ⟨ ⟩ [TIMING] ⟨ ⟩	Chamar incondicionalmente a sub-rotina que começa no endereço rotulado "TIMING"
RELEASE	IN $\langle B_2 \rangle$	Fazer reset latch para liberar tecla

Numa rotina de temporização partimos carregando um número num registrador. Depois decrementamos o registrador. A instrução de decrementação é seguida por uma instrução de salto condicional. Se após a decrementação o registrador não estiver em zero, nós desviamos de volta para a instrução de decrementação, isto é, revertemos de volta, mas, quando o registrador lê zero, deixamos o laço e avançamos para a próxima instrução, que é um RETURN incondicional ao programa principal, isto é, ao "RELEASE". O tempo de atraso é determinado pelo número inicialmente colocado no registrador e pela taxa de relógio. Se o tempo de atraso fornecido por um único registrador não for bastante longo, podemos usar dois registradores. O segundo registrador é decrementado por 1 cada vez que o primeiro registrador atravessar todo o ciclo de decrementação desde seu registro inicial até zero. Assim, o número total de operações de decrementação requerido para limpar ambos os registradores depende do produto (e não da soma) dos dois registros. Somente quando ambos os registradores estiverem limpos terminamos a reversão da sub-rotina. Um fluxograma para uma sub-rotina de temporização é mostrado na Fig. 11.6-1. Aqui admitimos que os registradores envolvidos são os registradores D e E. O programa da sub-rotina é como se segue:

Rótulo	Conteúdo	Comentário
TIMING	MVI E $\langle N_E \rangle$	Mover número N_E para o registrador E
LOAD	MVI D $\langle N_D \rangle$	Mover número N_D para registrador D
WAIT	DCR D	Decrementar registrador D
	JNZ ⟨ ⟩ [WAIT] ⟨ ⟩	Se registrador D não for zero, saltar de volta para decrementar instrução no endereço "WAIT". Do contrário, continuar para próxima instrução
	DCR E	Decrementar registrador E
	JNZ ⟨ ⟩ [LOAD] ⟨ ⟩	Se registrador E não for zero, saltar de volta à instrução no endereço "LOAD" para colocar N_D novamente no registrador D
	RET	Retornar incondicionalmente ao programa principal

Os fabricantes fornecem informação sobre o número de ciclos de relógio requeridos para executar uma instrução. Para o 8080 esta informação é dada no Apêndice C. O tempo de um ciclo de relógio é o inverso da freqüência de relógio. A partir do programa principal acima, que requer uma sub-rotina TIMING, a partir da própria sub-rotina e a partir dos parâmetros de temporização N_E e N_D, podemos contar o número de ciclos de relógio entre os pulsos que tornam set e reset o latch na Fig. 11.5-1. Conhecendo o tempo de um ciclo de relógio, podemos, então, determinar o intervalo total de temporização estabelecido. Os detalhes da computação são deixados como exercício aos estudantes (Prob. 11.6-1). Pode acontecer que, quando a rotina de temporização for necessária, os registradores D e E estão retendo os dados necessários. Então, no começo da sub-rotina, teremos que colocar estes conteúdos do registrador na pilha e retirá-los pouco antes do retorno.

11.7 INTERRUPÇÕES

Conforme já temos observado, uma pastilha de microprocessador e sua memória associada operam em conjunto com equipamento periférico externo, teclados, dispositivos de indicação visual, componentes que requerem sinais de controle etc. De vez em quando um periférico ou outro requererão que o microprocessador lhe proporcione algum serviço de assistência técnica ou de reparação. Por exemplo, uma tecla será comprimida sobre um teclado, e será necessário que o microprocessador transfira a indicação da tecla para seu acumulador e depois possivelmente para algum local de armazenamento na memória. Para que o microprocessador seja capaz de assistir em serviços ao periférico, o microprocessador deve suspender qualquer outra rotina de programa que possa estar seguindo. Um modo de arranjar a questão é fazer com que o microprocessador suspenda suas operações periodicamente e determine se há um sinal do periférico para indicar que ele necessita de assis-

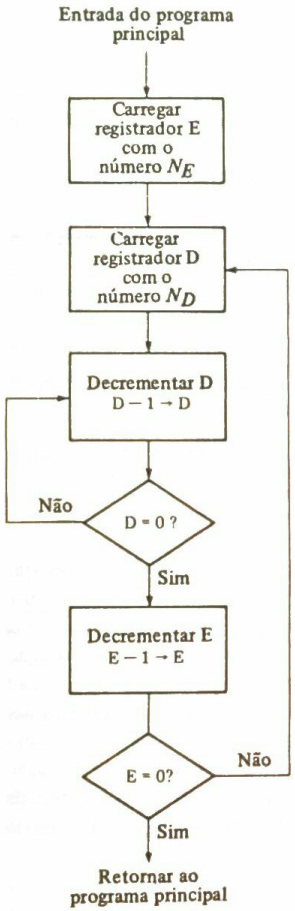


Fig. 11.6-1 Fluxograma de uma sub-rotina de temporização.

tência técnica ou de reparação. Se isto acontecer, o microprocessador desvia ou salta para uma sub-rotina de programa que proporciona a assistência técnica. Quando a assistência técnica for concluída, o microprocessador retorna ao programa interrompido. Este esquema direto de assistência técnica a um periférico é inteiramente viável, mas tem duas desvantagens: (1) o periférico que requer assistência técnica tem que esperar até que o microprocessador faça uma de suas indagações periódicas; (2) se quando as indagações forem feitas não for necessário assistência técnica, o tempo usado para fazer as indagações foi simplesmente desperdiçado.

Para evitar estas desvantagens, o 8080 (como outros microprocessadores) fornece um esquema alternativo para responder aos requisitos da assistência técnica. O 8080 tem um pino de entrada rotulado "INT" (pedido de interrupção). Quando este pino for trazido para lógica 1, mesmo brevemente (por pelo menos um tempo comparável ao período de relógio), o microprocessador completa a execução da instrução sobre a qual ele está engajado e depois começa a responder à chamada de serviço. Daí a resposta do microprocessador ser nominalmente imediata e nenhum tempo ser desperdiçado em indagações desnecessárias.

Se houvesse apenas uma única fonte potencial de um pedido de assistência técnica ou reparação, a sub-rotina de assistência técnica poderia ser iniciada em algum local único e predeterminado. O aparecimento de um sinal na entrada INT poderia então ser interpretado pelo microprocessador como uma instrução CALL para a sub-rotina neste local predeterminado. Mas já que, geralmente, haverá inúmeras fontes potenciais de pedidos de interrupção, requeremos um meio de identificar a fonte do pedido e de providenciar para que a sub-rotina chamada seja a única apropriada para a fonte. Em suma, deve haver muitos pontos de partida para as sub-rotinas de assistência técnica e um meio de obter a que seja correta.

O 8080 fornece oito endereços predeterminados em que uma sub-rotina de serviço possa iniciar. É mais conveniente especificar estes locais no sistema octal. Os bits de ordem alta destes endereços de partida são em cada caso 000. Os bits de ordem baixa são 000, 010, 020, 030, 040, 050, 060 e 070. Em notação decimal esta numeração de ordem baixa é 0, 8, 16, 24, 32, 40, 48 e 56, ou oito vezes o próximo dígito octal menos significativo. Nominalmente, então, uma sub-rotina de serviço tem disponível a faixa de endereços de ordem baixa 000 a 007, a próxima 010 a 017 etc. Se os sete locais de endereço disponíveis a uma sub-rotina não forem adequados, podemos usar uma instrução JUMP para levar o programa da sub-rotina a algum outro local disponível na memória onde a sub-rotina possa ser continuada.

Quando o 8080 recebe um pulso de interrupção, o microprocessador completa a execução da instrução que está sendo executada. Ele depois arranja (com a cooperação de hardware auxiliar, conforme veremos) para que a própria instrução seguinte carregada no registrador de instruções seja uma instrução que chame uma sub-rotina começando em um dos oito endereços de partida. Anteriormente não indicamos o código binário para instruções individuais, exceto em poucas ocasiões quando quisemos dar exemplos de instruções de um, dois e de três bytes, mas aqui é apropriado observar a forma do código binário que chama uma sub-rotina em resposta a uma interrupção. O código binário é

7	6	5	4	3	2	1	0
1	1	N_5	N_4	N_3	1	1	1

Os três dígitos binários $N_5N_4N_3$ determinam o endereço de partida da sub-rotina desde os oito tais possíveis endereços de acordo com a regra de que o endereço é oito vezes o significado numérico do número $N_5N_4N_3$. Assim, se $N_5N_4N_3 = 110\ (= 6)$, o endereço de partida é 000 060 (octal) e o endereço real é

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

Para ver precisamente como o 8080 responde a uma interrupção num caso específico, consideremos a situação representada na Fig. 11.7-1. Aqui contemplamos um teclado com as características de que, quando uma tecla for comprimida, o código da tecla é registrado em seu latch e, depois disto, um pulso de *dados válidos* é feito disponível. (Teclados deste tipo são encontrados comercialmente.) Quando uma tecla é abaixada, queremos que o microprocessador aceite os dados fornecidos pela operação da tecla e depois, presumivelmente, processe os dados de alguma maneira. Há outros periféricos além do teclado e, arbitrariamente, atribuímos ao teclado a identificação

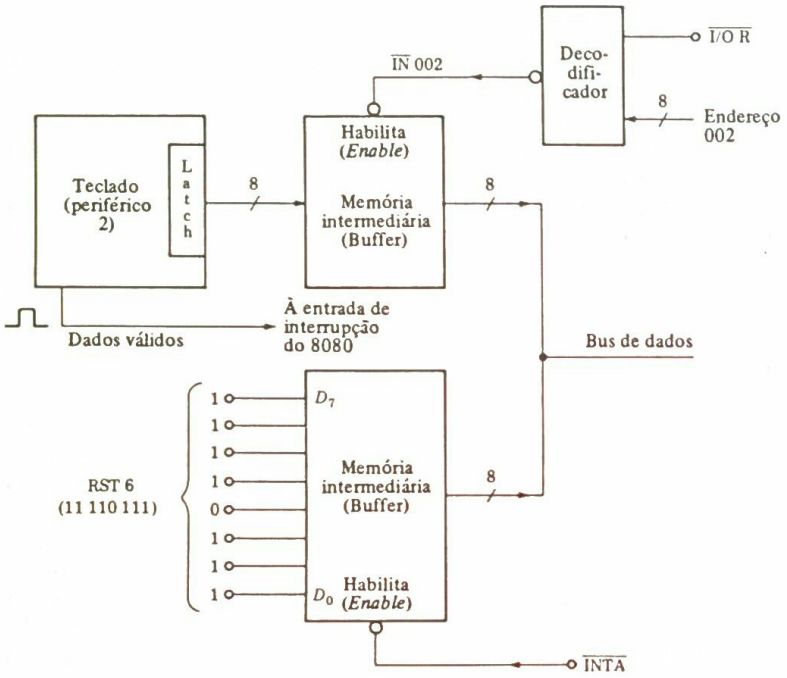


Fig. 11.7-1 O hardware associado a um sistema para permitir que a UCP responda a um pedido de interrupção.

periférico 2. A sub-rotina que presta serviço ao teclado começa no local 060 (octal) da memória. Correspondentemente, conectamos as oito entradas da memória intermediária inferior a níveis lógicos para formar o código de instrução 11 110 111 (= 367 octal), o "6" determinando que o endereço de partida total é 000 060 (octal). Agora, quando o microprocessador recebe o pulso de dados válidos em seu terminal de entrada INT, o sistema 8080 montado na Fig. 11.1-2 torna disponível um sinal de pulso negativo INTA (interrupt acknowledge). A memória intermediária é então habilitada, e o código de instrução 11 110 111 é colocado no bus de dados do microprocessador e transferido para dentro do microprocessador. No entanto, a transferência é especial quanto a que o código não é transferido para dentro do acumulador ou de outro registrador de propósitos gerais, mas para dentro do *registrador de instruções*. Daí, o resultado final é que o registrador de instruções agora retém a instrução que chama a sub-rotina requerida. Naturalmente, como sempre, quando o microprocessador desvia para uma sub-rotina, ele deve primeiro reservar na pilha o endereço de retorno ao programa principal. Esta seqüência de eventos que é iniciada por um pedido de interrupção é chamada instrução RESTART; seu mnemônico é RST *n*, onde *n* representa $N_5N_4N_3$. Mais formalmente temos:

RESTART	RST *n*	PC[high] → SP − 1 PC[low] → SP − 2 SP − 2 → SP 00000000 00$N_5N_4N_3$000 → PC

Os oito bits de ordem alta do próximo endereço da instrução são movidos para o local da memória cujo endereço é 1 menos do que o registro de SP. Os bits de ordem baixa são movidos para o local 2 menos do que o registro de SP. SP é decrementado duas vezes. A próxima instrução a ser buscada e executada é a instrução em 00000000 00$N_5N_4N_3$000.

11.8 HABILITAR E DESABILITAR INTERRUPÇÃO

Podemos facilmente imaginar situações em que, pelo menos por um tempo, não queremos que um sistema microprocessador responda a uma interrupção. Por exemplo, se o sistema estiver controlando algum processo que deva estar concluído num certo tempo fixo, pode não ser possível permitir tempo para responder aos pedidos de assistência técnica. Para acomodar este requisito, a pastilha do 8080 tem um *flip-flop de interrupção*, isto é, um *flag de interrupção*. Quando este flag estiver no estado set, o microprocessador aceitará e responderá a uma chamada de interrupção, mas quando o flag estiver no estado reset, o microprocessador não responderá à interrupção. O estado deste flag é indicado no terminal de saída do 8080 rotulado "INTE" (interrupt enable). O 8080 indica que sua facilidade de interrupção está na condição habilitada colocando INTE alta. Para proporcionar controle sobre o mecanismo de interrupção da pastilha são fornecidas duas instruções:

ENABLE INTERRUPT	EI

Habilitar o mecanismo de interrupção da pastilha do microprocessador seguindo a execução da *próxima* instrução

DISABLE INTERRUPT	DI

Desabilitar o mecanismo de interrupção da pastilha do microprocessador seguindo a execução *desta* instrução

Note que a instrução DI, como todas as outras instruções, requer uma resposta imediata, enquanto a instrução EI requer que o microprocessador execute primeiro a *próxima* instrução seguinte no programa após EI e somente depois faça o que é comandado pela instrução EI. A necessidade deste aspecto especial será explicada brevemente. Quando chegamos a um ponto num programa onde queremos rejeitar ou proibir interrupções, usamos DI. O microprocessador então deixará de responder a uma interrupção até que revoguemos a instrução usando EI.

Um microcomputador típico pode envolver mais do que apenas um só periférico. Examinaremos mais tarde o mecanismo pelo qual o 8080 identifica o periférico que requer assistência técnica ou reparação e decide quanto à ordem em que os pedidos de reparação devem ser processados. Apenas uma interrupção pode ser manipulada a qualquer tempo. No momento observemos que freqüentemente, embora nem sempre necessariamente, quando um pedido de interrupção está sendo atendido, preferimos que ele não seja interrompido alternadamente em resposta a um

segundo pedido de interrupção. O 8080 cuida deste assunto automaticamente. Imediatamente ao recebimento de um pedido de interrupção, ele reajusta o flag de interrupção. Uma vez que o flag não se ajusta automaticamente ao fim da rotina de serviço, é necessário, ao fim de toda rotina de serviço, incluir especificamente uma instrução EI. Se uma segunda interrupção for recebida enquanto o primeiro pedido estiver sendo atendido, o 8080 registrará este segundo pedido e empreenderá seu atendimento após a interrupção ter sido ajustada por EI. Somente o pedido de interrupção não atendido é lembrado.

Como em todo o programa de sub-rotina, a *última* instrução é a instrução RET (RETURN) que volta ao programa principal. A instrução EI deve então ser a instrução *próxima à última*. Uma vez que há um atraso de uma instrução na execução de EI, o mecanismo de interrupção não se torna habilitado até depois da instrução RET ter sido executada. A necessidade deste atraso pode ser vista do seguinte exemplo. Suponhamos que haja um período estendido durante o qual novos pedidos de interrupção sejam feitos antes do atendimento do pedido anterior ter sido completado. O microprocessador vai da rotina de interrupção para rotina de interrupção sem voltar sempre ao programa principal. E suponhamos que cada nova rotina de serviço comece imediatamente após a instrução EI e antes da instrução RET. Cada rotina de serviço começa colocando na pilha um endereço de retorno e possivelmente outros dados. A instrução RET retira os dados da pilha, mas, se a RET não for executada, a capacidade da memória pode igualmente ser sobrecarregada.

Um pedido de interrupção é um tipo especial de instrução de chamada. Como geralmente é o caso com instruções de chamada em que deixamos um programa principal, queremos reservar o conteúdo do acumulador, o registrador flag e os outros registradores; quando a rotina de serviço estiver completa, necessitaremos restaurar estes registradores. O padrão de tipos de instrução encontrados em conexão com uma rotina de serviço aparece, portanto, como na Fig. 11.8-1. Uma vez que o espaço da memória alocado a uma instrução RST *n* compreende apenas oito

Fig. 11.8-1 Uma sub-rotina de serviço típica.

locais de memória, pode não ser possível adaptar um programa como na Fig. 11.8-1 ao espaço disponível. Por conseguinte, um procedimento comum consiste em usar os três primeiros os locais da memória correspondentes a RST *n* para a instrução JUMP ⟨B_2⟩⟨B_3⟩, o salto (jump) levando o programa a um local da memória onde é disponível espaço adequado. Se, por outro lado, não nos propusermos a usar todas as oito instruções RST *n* disponíveis para chamar sub-rotinas de interrupção separadas, podemos permitir um programa partindo do endereço correspondente a RST *n* para avançar em RST *n* + 1 etc.

11.9 INTERRUPÇÃO COM CHAMADA SELETIVA (POLLED)

O arranjo da Fig. 11.7-1 é hardware apropriado quando houver apenas uma fonte possível de um pedido de interrupção. Consideremos agora um arranjo de hardware apropriado e seu correspondente software quando houver inúmeros periféricos a serem atendidos.

O hardware externo ao sistema 8080 da Fig. 11.1-2 é mostrado na Fig. 11.9-1. Aqui indicamos três periféricos, mas o sistema pode ser expandido para acomodar mais. Quando um periférico requer assistência técnica ou serviço, ele gera um pulso que é aplicado à entrada SET de um latch. Os latches, um para cada periférico, lembrarão agora dos pedidos e continuarão a alertar o microprocessador de que é requerido atendimento até que o microprocessador tome a ação requerida. Estes latches são depois apropriadamente denominados flags – mais especificamente flags externos, já que eles são exteriores ao microprocessador. Conforme veremos, quando um pedido de interrupção for atendido, seu flag estará abaixado; isto é, o latch estará reset.

As saídas dos flags são combinadas numa porta OR que depois fornece uma indicação para a entrada INT do 8080 de que algum (ou mais) dos periféricos requer assistência técnica. Como veremos, a partir do software que temos ainda que escrever, a resposta subseqüente do 8080 é como se segue. Primeiro ele lê em si próprio o estado de todos os flags. Depois ele observa alternadamente cada bit flag. Este processo de indagação de cada periférico alternadamente para descobrir se é requerida assistência técnica é denominado *chamada seletiva* ou interrogação sucessiva. Depois de fornecer o atendimento requerido ou de determinar que nenhuma assistência técnica é solicitada, o microprocessador move-se para o próximo periférico.

O microprocessador tem que exercer um certo controle sobre o hardware externo. Nos instantes adequados ele deve transferir para si próprio o estado dos flags e limpar alternadamente cada flag quando a chamada seletiva progredir. Com este propósito, como antes, usaremos instruções IN ⟨B_2⟩ e OUT ⟨B_2⟩, onde B_2 é o endereço do periférico a ser controlado. Como observado de início, os pulsos de controle são gerados nas linhas quando requerido, combinando-se num decodificador o endereço e as saídas $\overline{I/O\,W}$ e $\overline{I/O\,R}$ do sistema da Fig. 11.1-2. Necessitamos de três pulsos endereçados individuais para a limpeza dos flags. Eles são gerados em resposta à instrução OUT ⟨B_2⟩ e são chamados OUT ⟨1⟩, OUT ⟨2⟩ e OUT ⟨3⟩. Um único pulso de controle IN ⟨BUFFER⟩ é usado para habilitar a memória intermediária (*buffer*) e ler o estado dos flags no acumulador.

A sub-rotina de serviço total para o caso representado na Fig. 11.9-1 tem quatro partes. Uma sub-rotina toma conta da chamada seletiva e três sub-rotinas de serviço adicionais fornecem os três serviços solicitados pelas três fontes potenciais de um pedido de interrupção. A sub-rotina de chamada seletiva é como se segue:

Rótulo	Conteúdo	Comentário
POLL	IN ⟨BUFFER 2⟩	Introduzir em Acc estado dos flags habilitando BUFFER 2
	RAR	Girar Acc à direita de modo que primeiro bit D_0 aparecerá no flag de transporte, onde ele pode ser testado
	JC ⟨ ⟩ [SUBROUTINE 1] ⟨ ⟩	Se transporte = 1, saltar para sub-rotina que atende ao primeiro periférico. Endereço de sub-rotina está no segundo e no terceiro byte desta instrução. Se transporte = 0, continuar com próxima instrução
	RAR	Girar Acc novamente para testar D_1
	JC ⟨ ⟩ [SUBROUTINE 2] ⟨ ⟩	Se transporte = 1, saltar para sub-rotina que atende ao segundo periférico
	RAR	Girar Acc novamente para testar D_2
	JC ⟨ ⟩ [SUBROUTINE 3] ⟨ ⟩	Se transporte = 1, saltar para sub-rotina que atende ao terceiro periférico

Se o primeiro periférico requerer serviço, a primeira instrução JC na rotina de chamada seletiva acima causará um salto para a sub-rotina de serviço do primeiro periférico. Nesta sub-rotina de serviço, a primeira instrução deve ser $\overline{\text{OUT}}$ ⟨1⟩ para o FLAG 1. A instrução próxima à última será EI e a última será RETURN. O programa conseqüentemente voltará para o programa principal no ponto onde ele foi interrompido. Se agora o segundo ou o terceiro periférico ou ambos requererem assistência técnica ou serviço, não serão buscadas nenhumas das instruções do programa principal; em vez disto, o programa saltará de volta ao programa de chamada seletiva uma vez mais. No programa de chamada seletiva as primeiras instruções lerão novamente o estado dos flags e testarão novamente FLAG 1. Desde que FLAG 1 tenha sido limpo, no entanto, não haverá nenhum salto para a rotina de serviço do primeiro periférico e, em vez disto, a rotina de chamada seletiva se moverá para frente para testar FLAG 2. Desta maneira, todos os flags eventualmente serão testados e todos os serviços de assistência serão providenciados. Notar que, mesmo depois do serviço ter sido providenciado para o primeiro periférico, a rotina de chamada seletiva novamente testará FLAG 1 antes de FLAG 2. Quando o último programa de serviço solicitado tiver sido concluído, todos os flags terão sido limpos e o microprocessador voltará para o programa principal.

A ordem em que os flags são testados estabelece uma prioridade relativa entre os periféricos, tendo o primeiro testado a mais alta prioridade etc. Por questão de prática, a ordem de prioridade atribuída aos periféricos deve depender não somente da importância relativa, mas também do tempo consumido pela correspondente rotina de serviço. Assim, se a assistência técnica a um periférico menos importante requerer 1 ms quando um periférico mais importante requerer vários segundos, ao periférico menos importante pode bem ser atribuída uma prioridade mais alta.

No esquema de chamada seletiva descrito acima, quando um periférico de baixa prioridade iniciar um pedido de serviço (mesmo se ele for o único periférico que pede assistência ou serviço),

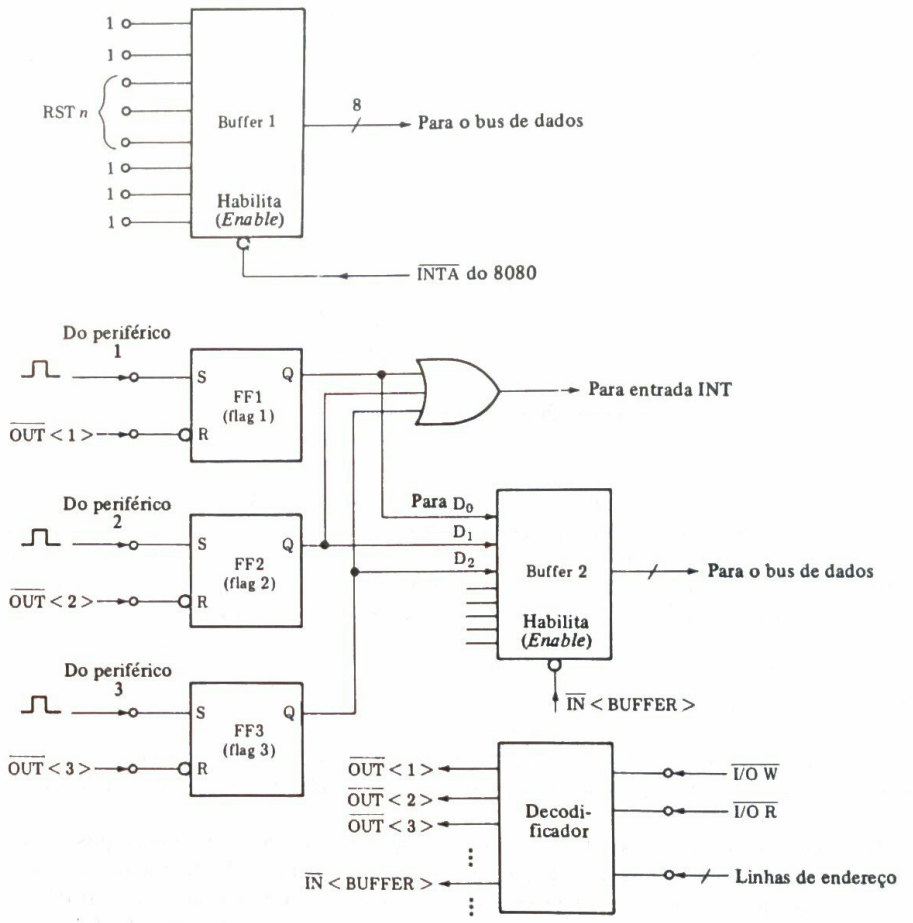


Fig. 11.9-1 Hardware associado a um sistema para permitir que a UCP responda a pedidos de assistência técnica originários de inúmeras fontes. A instrução RST n chama a sub-rotina de chamada seletiva.

ele deve esperar até que todos os outros periféricos de prioridade mais alta tenham sido seletivamente chamados. Este aspecto pode não ser sempre aceitável. Um esquema alternativo que supera a rotina de chamada seletiva está indicado na Fig. 11.9-2. Aqui um codificador de prioridade (ver Seção 3.16) é usado. Os bits flag que na Fig. 11.9-1 estão ligados numa porta OR são aqui aplicados às entradas do codificador de prioridades. A entrada 7 tem a mais alta prioridade e a entrada 0 tem a mais baixa prioridade. A saída de três bits do codificador é a representação binária do número decimal pelo qual as entradas são identificadas. Esta saída do codificador fornece os três bits da instrução RST n ($n = N_5N_4N_3$). Se qualquer periférico necessitar serviço, a saída

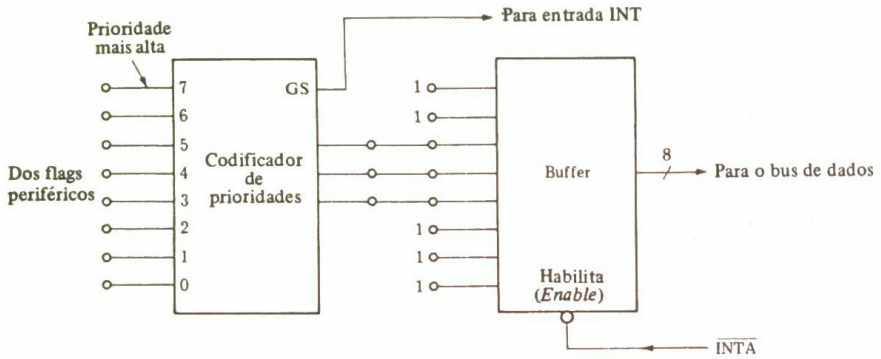


Fig. 11.9-2 Um codificador de prioridades permite que assistência técnica (serviço) seja fornecida ao periférico de prioridade mais alta requerendo serviço sem chamada seletiva.

GS do codificador fornece um pedido de interrupção ao processador. A resposta INTA do processador "força a entrada", então, no registrador de instruções de uma das instruções de reiniciar RST 7, RST 6 etc. (Na verdade, por uma razão observada na Seção 11.10, a instrução RST 0 não é geralmente usada.) Não há chamada seletiva. O processador vai diretamente para a subrotina de serviço requerida para responder imediatamente ao periférico de prioridade mais alta.

11.10 OUTRA COMUNICAÇÃO I/O

Consideremos novos pinos adicionais no 8080 através dos quais o microprocessador se comunica com o mundo exterior. Observemos primeiro o pino de entrada chamado READY e o pino de saída chamado WAIT.

Já observamos que na busca (*fetch*) e na execução de uma instrução o microprocessador atravessa inúmeros ciclos de máquina e que na transposição de cada ciclo de máquina o processador atravessa inúmeros estados. O processador vai de estado a estado junto com o relógio. Também notamos que, no primeiro estado de cada ciclo de máquina (denominado T_1), o processador coloca no bus de endereços o endereço de um local da memória ou um dispositivo I/O e faz informação de estado disponível no bus de dados. Notamos ainda que, quando o processador entra no segundo estado T_2 de um ciclo de máquina, ele realmente não avança o processamento, mas simplesmente examina a entrada READY. Se a entrada READY for alta, o processador avança para o terceiro estado T_3 e igualmente para os estados T_4 e T_5, se tais estados adicionais forem solicitados, para continuar o processamento. No estado T_2, se o processador achar que READY é baixo, no entanto, ele não vai para o estado T_3, mas para um estado WAIT T_W, onde o processador permanece inativo, não fazendo nada, esperando que a entrada READY se torne alta. O processador pode permanecer indefinidamente no estado WAIT, seu tempo em que o estado se estende sobre um número inteiro de ciclos de relógio. Quando READY torna-se alta, o processador sai do estado T_W para ir para o estado T_3 e depois continua o processamento. Quando o processador vai para o estado WAIT, ele alerta o mundo exterior para esta condição

elevando o pino WAIT de saída ao nível alto. Em resumo, a temporização é como se segue: se no começo do estado T_2 o processador achar READY baixa, ele entra no estado WAIT e no começo do estado WAIT eleva o pino WAIT para alto. Assim, há um intervalo de um ciclo de relógio entre o tempo que o processador reconhece que um WAIT é solicitado e o tempo em que ele reconhece para o mundo exterior que ele está realmente esperando.

A entrada READY permite que o processador se acomode a periféricos relativamente lentos, incluindo mesmo certos tipos de memórias que respondem muito lentamente. Tais periféricos podem ser equipados com um tipo de confirmação de conexão de saída que é aplicada à entrada READY. Esta saída do periférico será projetada para permanecer baixa até que um tempo adequado tenha decorrido para que o periférico responda adequadamente ao processador.

Como exemplo do uso dos terminais READY e WAIT do 8080, consideremos o circuito mostrado na Fig. 11.10-1. O propósito deste arranjo é possibilitar *operação de uma única etapa* do microprocessador, em que o processador permanece inativo no estado de espera até que o botão de pressão S seja abaixado. Ele então avança pelo estado restante T_3 (e possivelmente T_4 e T_5) do ciclo de máquina, retorna para atravessar T_1 e T_2 do próximo ciclo de máquina e, finalmente, chega a RESET novamente no estado WAIT T_W. A cada pressão do botão, o processador completa um ciclo de máquina, parte no ciclo seguinte e depois pára. O escalonamento simples permite-nos (com o auxílio de hardware adicional) monitorar o que está no bus de endereços e no bus de dados em cada ciclo de máquina. Este modo de operação é inestimável na depuração de um programa.

Na Fig. 11.10-1, FF2 não é, em princípio, essencial à operação, mas o 8080 tem certos requisitos de temporização tais como os tempos de preparação (*setup*) e manutenção (*hold*) a que o sinal de entrada READY deve satisfazer. O fabricante especifica que muito geralmente estas especificações podem ser encontradas pela aplicação do sinal READY não diretamente à entrada READY, mas da maneira mostrada. Aqui, uma READYIN (ready input) é aplicada a um flip-flop tipo *D* gatilhado pela borda que usa ϕ_2 como forma-de-onda de relógio. Em cada borda crescente positiva de ϕ_2, READYIN é transferida para READY, e o nível assim sincronizado (clocado) é retido pelo

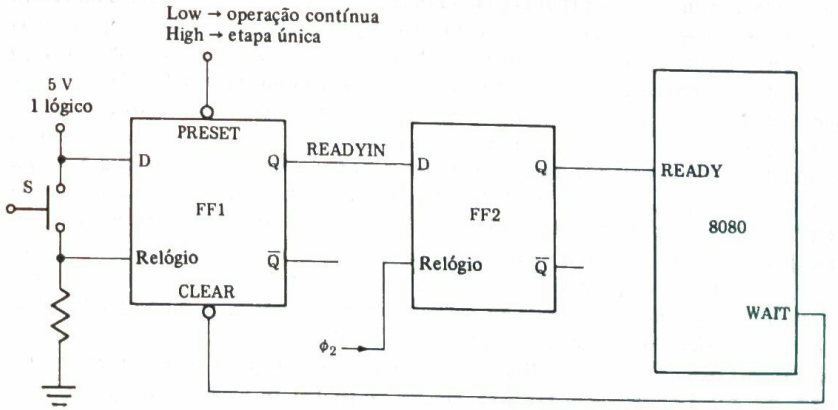


Fig. 11.10-1 Um arranjo para possibilitar operação em uma única etapa.

menos durante um ciclo de relógio. O fabricante explica por que FF2 não é incorporado diretamente na pastilha do 8080, notando que os requisitos de temporização tornam necessário que a função de FF2 seja executada por dispositivos bipolares rápidos em vez de por dispositivos MOS mais lentos fabricados na pastilha do 8080. Em qualquer caso, tendo feito estes comentários para evitar representar indevidamente a pastilha 8080, ignoremos FF2 em nossa discussão subseqüente e pretendamos que READYIN esteja conectada diretamente a READY.

O flip-flop FF1 é flip-flop do tipo *D* gatilhado pela borda crescente positiva com entradas PRESET e CLEAR ativas e baixas (tais como o tipo 7474). Quando PRESET e CLEAR forem altas, estas entradas assíncronas não têm nenhum efeito sobre o flip-flop, e a cada borda de relógio crescente positiva o nível em D é transferido para Q. As entradas assíncronas se sobrepõem à entrada sincronizada (clocada). Quando PRESET estiver alta e CLEAR estiver baixa, Q fica baixa e $\overline{Q}$ fica alta. Quando PRESET estiver baixa e CLEAR estiver alta, Q fica alta e $\overline{Q}$ fica baixa. A característica do tipo 7474 é que, quando tanto PRESET como CLEAR forem baixas, tanto Q como $\overline{Q}$ ficam altas.

Consideremos primeiro que colocamos PRESET baixa. Então READY está alta, indicando que não há necessidade de WAIT. A saída WAIT estará baixa e, portanto, CLEAR também estará. Tanto Q quanto $\overline{Q}$ estão altos. O microprocessador operará continuamente, nunca parando no estado WAIT. Agora coloquemos PRESET alta, de modo que READY fique baixa. O processador continuará em ciclos através de seus estados até que chegue ao estado T_2. Ali ele perceberá uma chamada para um WAIT e, conseqüentemente, irá para o estado T_W. No começo do estado T_W, WAIT irá alto, reconhecendo que o processador está esperando. Agora temos tanto PRESET como CLEAR altas, isto é, ambas estão no nível inativo. No entanto, uma vez que PRESET foi alta primeiro, o efeito do nível baixo em CLEAR persistirá e READY permanecerá baixa. O processador permanecerá no estado WAIT indefinidamente. Agora, abaixemos a chave S: READY torna-se alta e o processador sai do estado WAIT e entra no estado T_3. No começo do estado T_3 a saída WAIT fica baixa e READY volta para baixa. No entanto, uma vez que o processador foi liberado do estado WAIT, ele continuará através do ciclo da máquina até que volte para o estado WAIT. No estado WAIT a saída WAIT é novamente alta, de modo que o processador possa uma vez mais responder à chave S.

Consideremos a seguir os pinos HOLD e HOLDA (*hold acknowledge* = retenção conhecida). HOLD é um pino de entrada e, quando ele é posto alto, o processador suspende sua operação. O processador também libera o bus de endereços e o bus de dados. Assim, os pinos de endereço e os pinos de dados do processador são postos no estado de alta impedância (ver Seção 3.10), de modo que efetivamente o processador está desconectado dos buses de endereços e de dados. Esta desconexão permite que dispositivos externos ganhem controle dos buses. Um exemplo da utilidade de uma tal facilidade HOLD deve ser vista em conexão com o *acesso direto à memória* (DMA). Há ocasiões em que grandes quantidades de dados necessitam ser escritas na ou lidas da memória, sendo a fonte ou o destino dos dados algum dispositivo externo. Ela irá, obviamente, poupar tempo para transferir os dados diretamente entre o dispositivo e a memória, em vez de através do processador. Com este propósito, os buses devem estar disponíveis e, uma vez que o processador não pode usar os buses, o processador deve suspender sua operação. O processador reconhece que está retendo, elevando ao nível alto sua saída HOLDA.

O processador examina a entrada HOLD em cada ciclo de máquina durante o estado T_2, o mesmo estado em que a entrada READY é examinada. Se HOLD for achada alta, o processador torna set um flip-flop flag interno, mas depois vai em frente para acabar o ciclo de máquina (não necessariamente a instrução). Ele depois vai para um *modo de retenção* (não o estado WAIT),

mas o processador libera os buses no ponto do ciclo de máquina quando ele não mais os requer. Assim, nos estados T_4 e T_5 são efetuadas operações que são inteiramente interiores ao processador, e os buses são feitos disponíveis mesmo que enquanto o processador estiver atravessando T_4 e T_5 se revele que estes estados são usados no ciclo da máquina.

O pino final no 8080 é a entrada RESET. Quando a entrada RESET for elevada alta e mantida neste nível durante pelo menos três ciclos de relógio, inúmeras coisas acontecem: (1) o contador do programa (PC) será reajustado (reset) em zero; (2) a facilidade de interrupção será desativada, em reconhecimento de que a saída INTE será baixa; e (3) se o processador por acaso estiver no modo de retenção, ele sairá deste modo e, correspondentemente, a saída HOLDA ficará baixa. No entanto, os conteúdos dos flags, do acumulador, do apontador de pilha e de outros registradores não serão afetados. Quando a entrada RESET for permitida retornar baixa, o processador inicia imediatamente a operação, começando com a instrução em local zero da memória. Uma vez que o local zero da memória está envolvido na operação RESET, este local da memória não está geralmente disponível em conexão com uma sub-rotina de interrupção.

A entrada RESET deve inevitavelmente ser usada quando a energia de alimentação for aplicada ao processador. Portanto, quando a alimentação for aplicada primeiro, os registros que aparecerão nos vários registradores são inteiramente imprevisíveis, sendo determinados por fatores aleatórios. Se, usando RESET, dermos partida ao processador no local zero da memória, podemos providenciar para as primeiras instruções limparem os registradores, tornarem reset os flags e iniciarem o apontador de pilha. Deste modo o programa a ser passado na máquina não encontrará indicações de flags de estados inadequados e não pretendidos, conteúdos de registradores etc.

Se quisermos que o sistema não parta tão logo RESET retorne baixa, mas que espere por um sinal de partida explícito, podemos arranjá-lo do seguinte modo. Imediatamente após as instruções que limpam o registrador de dados aleatórios impróprios, usamos a instrução EI para habilitar a facilidade de interrupção e, depois, a instrução HLT para parar o processador. Mas, quando quisermos partir rodando o programa, introduzimos de uma fonte externa uma interrupção e forçamos (fazemos entrar) o endereço de partida do programa dentro do processador.

APÊNDICE A

CÓDIGO ASCII (hexadecimal com bit zero mais significativo)

Código ASCII	Caractere	Código ASCII	Caractere	Código ASCII	Caractere
00	NUL	14	DC4	28	(
01	SOH	15	NAK	29	)
02	STX	16	SYN	2A	*
03	ETX	17	ETB	2B	+
04	EOT	18	CAN	2C	,
05	ENQ	19	EM	2D	-
06	ACK	1A	SUB	2E	.
07	BEL	1B	ESC	2F	/
08	BS	1C	FS	30	0
09	HT	1D	GS	31	1
0A	LF	1E	RS	32	2
0B	VT	1F	US	33	3
0C	FF	20	SP	34	4
0D	CR	21	!	35	5
0E	SO	22	"	36	6
0F	SI	23	#	37	7
10	DLE	24	$	38	8
11	DC1 (X-ON)	25	%	39	9
12	DC2 (TAPE)	26	&	3A	:
13	DC3 (X-OFF)	27	'	3B	;

Código ASCII	Caractere	Código ASCII	Caractere	Código ASCII	Caractere
3C	<	53	S	6A	j
3D	=	54	T	6B	k
3E	>	55	U	6C	l
3F	?	56	V	6D	m
40	@	57	W	6E	n
41	A	58	X	6F	o
42	B	59	Y	70	p
43	C	5A	Z	71	q
44	D	5B	[	72	r
45	E	5C	\	73	s
46	F	5D	]	74	t
47	G	5E	^(↑)	75	u
48	H	5F	−(←)	76	v
49	I	60	`	77	w
4A	J	61	a	78	x
4B	K	62	b	79	y
4C	L	63	c	7A	z
4D	M	64	d	7B	{
4E	N	65	e	7C	|
4F	O	66	f	7D	} (ALT MODE)
50	P	67	g	7E	~
51	Q	68	h	7F	DEL (RUB OUT)
52	R	69	i		

APÊNDICE B

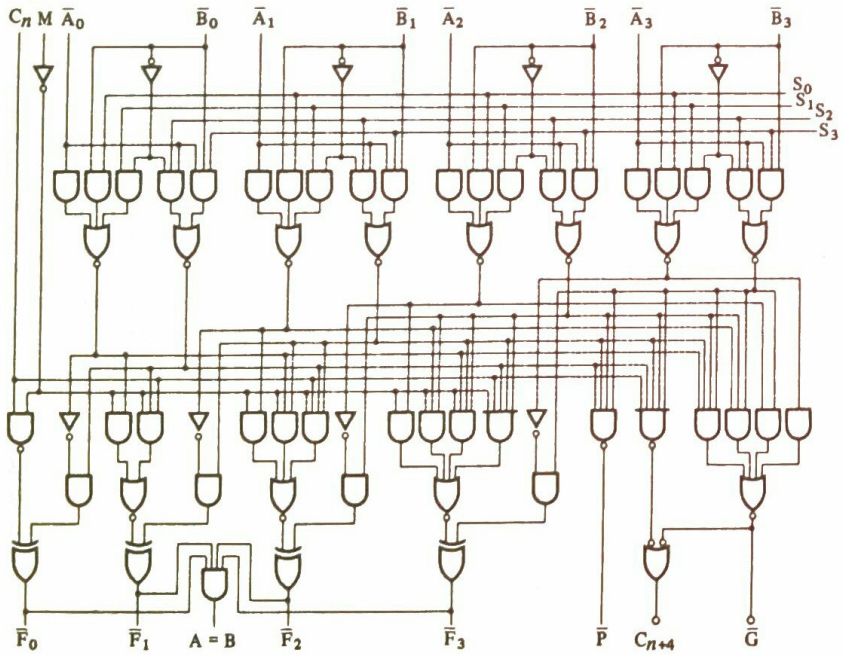


Fig. B.1 Diagrama lógico da ULA tipo 181.

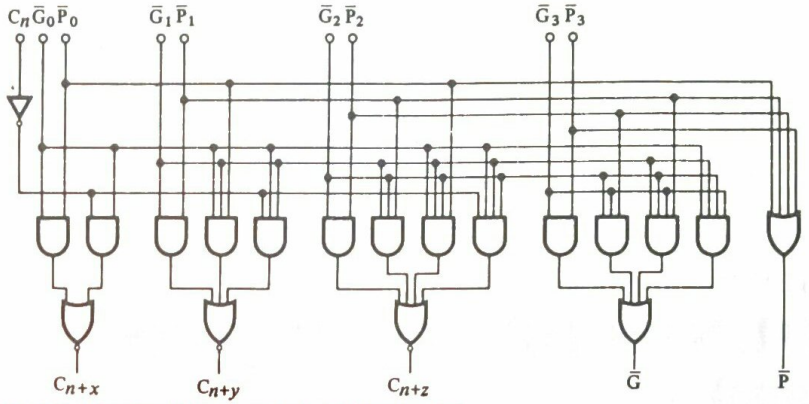


Fig. B.2 Diagrama lógico da unidade de VUA tipo 182.

APÊNDICE C
CONJUNTO DE INSTRUÇÕES DO 8080

Onde são dados dois números de ciclos de relógio, o menor número é requerido se a condição não estiver satisfeita e o maior número é requerido se a condição estiver satisfeita.

Mnemônico	Descrição	Ciclos de relógio
MOV r1, r2	Move register to register	5
MOV M, r	Move register to memory	7
MOV r,M	Move memory to register	7
HLT	Halt	7
MVI r	Move immediate register	7
MVI M	Move immediate memory	10
INR r	Increment register	5
DCR r	Decrement register	5
INR M	Increment memory	10
DCR M	Decrement memory	10
ADD r	Add register to A	4
ADC r	Add register to A with carry	4
SUB r	Subtract register from A	4
SBB r	Subtract register from A with borrow	4
ANA r	AND register with A	4
XRA r	EXCLUSIVE-OR register with A	4
ORA r	OR register with A	4
CMP r	Compare register with A	4
ADD M	Add memory to A	7
ADC M	Add memory to A with carry	7
SUB M	Subtract memory from A	7
SBB M	Subtract memory from A with borrow	7
ANA M	AND memory with A	7
XRA M	EXCLUSIVE-OR memory with A	7
ORA M	OR memory with A	7
CMP M	Compare memory with A	7

Mnemônico	Descrição	Ciclos de relógio
ADI	Add immediate to A	7
ACI	Add immediate to A with carry	7
SUI	Subtract immediate from A	7
SBI	Subtract immediate from A with borrow	7
ANI	AND immediate with A	7
XRI	EXCLUSIVE-OR immediate with A	7
ORI	OR immediate with A	7
CPI	Compare immediate with A	7
RLC	Rotate A left	4
RRC	Rotate A right	4
RAL	Rotate A left through carry	4
RAR	Rotate A right through carry	4
JMP	Jump unconditional	10
JC	Jump on carry	10
JNC	Jump on no carry	10
JZ	Jump on zero	10
JNZ	Jump on no zero	10
JP	Jump on positive	10
JM	Jump on minus	10
JPE	Jump on parity even	10
JPO	Jump on parity odd	10
CALL	Call unconditional	17
CC	Call on carry	11/17
CNC	Call on no carry	11/17
CZ	Call on zero	11/17
CNZ	Call on no zero	11/17
CP	Call on positive	11/17
CM	Call on minus	11/17
CPE	Call on parity even	11/17
CPO	Call on parity odd	11/17
RET	Return	10
RC	Return on carry	5/11
RNC	Return on no carry	5/11
RZ	Return on zero	5/11
RNZ	Return on no zero	5/11
RP	Return on positive	5/11
RM	Return on minus	5/11
RPE	Return on parity even	5/11
RPO	Return on parity odd	5/11
RST	Restart	11
IN	Input	10
OUT	Output	10
LXI B	Load immediate register pair B and C	10
LXI D	Load immediate register pair D and E	10
LXI H	Load immediate register pair H and L	10
LXI SP	Load immediate stack pointer	10
PUSH B	Push register pair B and C on stack	11
PUSH D	Push register pair D and E on stack	11
PUSH H	Push register pair H and L on stack	11
PUSH PSW	Push A and flags on stack	11
POP B	Pop register pair B and C off stack	10
POP D	Pop register pair D and E off stack	10
POP H	Pop register pair H and L off stack	10

Mnemônico	Descrição	Ciclos de relógio
POP PSW	Pop A and flags off stack	10
STA	Store A direct	13
LDA	Load A direct	13
XCHG	Exchange D and E, H and L registers	4
XTHL	Exchange top of stack, H and L	18
SPHL	H and L to stack pointer	5
PCHL	H and L to program counter	5
DAD B	Add B and C to H and L	10
DAD D	Add D and E to H and L	10
DAD H	Add H and L to H and L	10
DAD SP	Add stack pointer to H and L	10
STAX B	Store A indirect	7
STAX D	Store A indirect	7
LDAX B	Load A indirect	7
LDAX D	Load A indirect	7
INX B	Increment B and C registers	5
INX D	Increment D and E registers	5
INX H	Increment H and L registers	5
INX SP	Increment stack pointer	5
DCX B	Decrement B and C	5
DCX D	Decrement D and E	5
DCX H	Decrement H and L	5
DCX SP	Decrement stack pointer	5
CMA	Complement A	4
STC	Set carry	4
CMC	Complement carry	4
DAA	Decimal adjust A	4
SHLD	Store H and L direct	16
LHLD	Load H and L direct	16
EI	Enable interrupt	4
DI	Disable interrupt	4
NOP	No operation	4

PROBLEMAS

Capítulo 1

1.6-1 Prove que a função OR é associativa e comutativa.

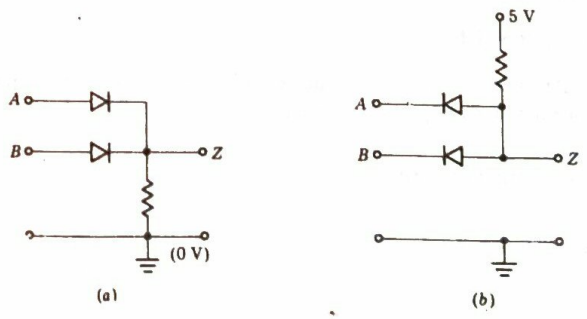
1.7-1 A tripulação de um avião consiste em dois pilotos e um engenheiro. Projete um circuito com chaves que são acionadas quando um membro da tripulação deixa sua poltrona e que gera um sinal de alarme sempre que o engenheiro deixa seu posto ou sempre que os dois pilotos deixam seus postos simultaneamente.

1.7-2 O circuito da Fig. 1.7-1 foi apresentado como um exemplo de um sistema lógico que executa a operação AND e o da Fig. 1.7-2 como exemplo de um sistema que executa a operação OR. Suponha que todas as variáveis A, B e Z tenham suas definições invertidas, isto é, A seria verdadeiro quando a chave estivesse aberta etc. Verifique que neste caso o circuito da Fig. 1.7-1 torna-se um circuito OR e o da Fig. 1.7-2 um circuito AND.

1.8-1 Suponha que o valor lógico 0 seja representado pela tensão 0 V (massa) e que o valor lógico 1 seja representado por 5 V.

(*a*) Mostre que o circuito a díodo da Fig. P1.8-1*a* é uma porta OR e que o circuito da Fig. P1.8-1*b* é uma porta AND. (Suponha que os díodos são ideais em todos os sentidos.)

(*b*) Substitua a lógica positiva de (*a*) por lógica negativa, isto é, 1 e 0 representados, respectivamente, por 0 V e 5 V. Mostre que a porta OR tornou-se AND e vice-versa.



1.11-1 Encontre os equivalentes, no sistema decimal, dos seguintes números binários: (*a*) 11001, (*b*) 10101, (*c*) 0,01101, (*d*) 10011,1011.

1.12-1 Converta para o sistema binário os seguintes números decimais: (*a*) 29, (*b*) 137, (*c*) 259, (*d*) 163.

1.12-2 Converta para o sistema binário os seguintes números decimais: (*a*) 17,74, (*b*) 27,159, (*c*) 11,111, (*d*) 9,99.

1.12-3 O processo para converter um número decimal em um número binário consiste, conforme descrito na Seção 1.12, na divisão sucessiva por 2 (isto é, pela base do sistema binário). De modo similar, a conversão de um número decimal em um número de um sistema cuja base seja B consiste de divisões sucessivas pela base B.
(*a*) Converta o número decimal 119 para o sistema base 7.
(*b*) Converta o número decimal 57 para o sistema base 3.
(*c*) Converta o número decimal 99 para o sistema base 5.

1.12-4 (*a*) Converta o número decimal 0,69 para o sistema base 3.
(*b*) Converta o número decimal 79,46 para o sistema base 4.

1.13-1 (*a*) Converta os números octais 765 e 627 para base 2.
(*b*) Converta os números hexadecimais BEEF e CAB7 para base 2.
(*c*) Converta os números decimais 987 e 526 em números hexadecimais.
(*d*) Converta o número binário 101101101,10101101 para o sistema octal e para o sistema hexadecimal.

1.14-1 Prepare tabelas verdade para as funções:
(*a*) $f(A, B, C) = A\,(B + \bar{C})(\bar{B} + C)$
(*b*) $f(A, B, C, D) = A\,[\bar{B} + \bar{C}\,(\bar{B} + D)]$

1.15-1 Verifique, examinando todos os casos possíveis, que todos os teoremas das Eqs. (1.15-2) até (1.15-5) são válidos.

1.15-2 Use uma tabela verdade para três variáveis para verificar o teorema da Eq. (1.15-11*a*).

1.15-3 Usando manipulação algébrica e os outros teoremas disponíveis, prove os teoremas das Eqs. (1.15-11*a*) e (1.15-11*b*).

1.16-1 Verifique que $\overline{A\bar{B} + \bar{A}B} = AB + \bar{A}\bar{B}$

1.16-2 Prove que, se complementarmos uma expressão lógica e substituirmos cada variável por seu complemento, formaremos o dual da expressão original.

1.16-3 Encontre o complemento das seguintes expressões:
(*a*) $(X + Y\bar{Z})(\bar{X} + \bar{W}V)$
(*b*) $A\,[B + \bar{C}\,(D + \bar{E})]$
(*c*) $A + B\,(\bar{C} + D\bar{E})$

1.17-1 Desenhe um diagrama de Venn para quatro variáveis. Comece com um diagrama para três variáveis e inclua uma quarta região fechada que divida o mapa em dezesseis regiões que tenham correspondência biunívoca com as linhas de uma tabela verdade para quatro variáveis.

1.17-2 Use um diagrama de Venn para provar que as seguintes equações são válidas:
(*a*) $\bar{A}C + \bar{A}B + \bar{B}C = A\bar{B} + B\bar{C} + \bar{A}C$
(*b*) $\bar{A} + \bar{B} = \bar{A}B + A\bar{B} + \bar{A}\bar{B} = \bar{A}\bar{B}$

1.19-1 Prove que a operação EXCLUSIVE-OR é comutativa e associativa.

1.19-2 Temos n variáveis lógicas $A_0, A_1, \ldots, A_{n-1}$ que em determinado momento podem ter valor lógico 1 ou 0. Necessitamos um circuito lógico que determine se o número de variáveis com valor lógico 1 é par ou ímpar. Explique como podemos usar portas EXCLUSIVE-OR para este fim.

1.20-1 Prove que nem a operação NAND nem a operação NOR são associativas.

1.22-1 Verifique que $f_2 = \overline{A \supset B} = A\bar{B}$ e que $f_4 = \overline{B \supset A} = \bar{A}B$.

1.25-1 Através de manipulação algébrica, usando os teoremas da álgebra booleana, verifique as seguintes equações:
(*a*) $(A + \bar{B} + AB)(A + \bar{B})\bar{A}B = 0$
(*b*) $(A + \bar{B} + A\bar{B})(AB + \bar{A}C + BC) = AB + \bar{A}\bar{B}C$
(*c*) $(AB + C + D)(C + \bar{D})(C + \bar{D} + E) = AB\bar{D} + C$
(*d*) $\bar{A}B\,(\bar{D} + D\bar{C}) + (A + D\bar{A}C)B = B$

1.25-2 Simplifique as seguintes expressões:
(*a*) $\overline{(A + B)(\bar{A} + C) + (\bar{A} + \bar{B} + \bar{A}BC)(A + \bar{A}B)(\bar{A} + \bar{B})}$
(*b*) $\overline{\bar{A}\bar{B}\bar{C} + \bar{A}D + CD + AC\bar{D} + \bar{A}B + B\bar{D} + \bar{A}\bar{D} + AB}$

1.25-3 Um circuito lógico de *maioria* fornece uma saída com nível lógico 1 quando a maioria das entradas estiver em 1. Para o caso de três entradas A, B e C escreva a expressão lógica para a variável Z que é $Z = 1$ quando a maioria das entradas for 1. Simplifique a expressão e desenhe o circuito usando portas AND e OR.

1.25-4 Simplifique as seguintes expressões:
(*a*) $A + \bar{A}B + \overline{(A + B)}C + \overline{(A + B + C)}D$
(*b*) $A\bar{B} + AC + BCD + \bar{D}$
(*c*) $\bar{A} + \bar{A}\bar{B} + BC\bar{D} + B\bar{D}$
(*d*) $A\bar{B}C + (\bar{B} + \bar{C})(\bar{B} + \bar{D}) + \overline{A + C + D}$

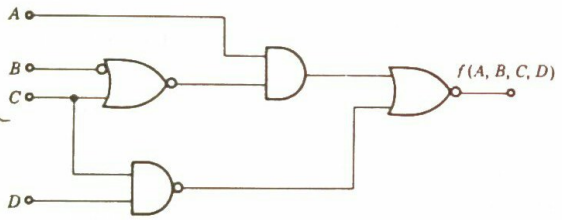
1.25-5 (*a*) Existem três chaves de parede *a*, *b* e *c*. $A = 1$ representa a condição chave *a* PARA CIMA e $A = 0$ representa a condição chave *a* PARA BAIXO. De modo similar, as variáveis B e C estão associadas às posições das chaves *b* e *c*, respectivamente. Escreva uma expressão para uma variável Z que pode ser feita $Z = 1$ ou $Z = 0$ pela operação

individual de uma chave, independentemente das posições das outras chaves. Supondo que as chaves possam ser usadas para prover níveis lógicos para as entradas das portas, desenhe uma estrutura de portas que implemente Z.

(*b*) Lembrando que chaves em série e em paralelo podem executar as operações AND e OR, conforme mostrado nas Figs. 1.7-1 e 1.7-2, desenhe o circuito de um arranjo de chaves que permite que uma lâmpada seja LIGADA ou DESLIGADA por qualquer das três chaves. (As chaves podem ser multipolares.)

1.26-1 Um técnico em um laboratório químico possui quatro produtos químicos A, B, C e D que devem ser guardados em um ou outro depósito. Por conveniência, é necessário mover um ou mais produtos de um depósito para o outro de tempos em tempos. A natureza dos produtos é tal que é perigoso guardar B e C juntos, a não ser que A esteja no mesmo depósito. Também é perigoso guardar C e D juntos se A não estiver no depósito. Escreva uma expressão para a variável lógica Z tal que $Z = 1$ sempre que exista uma combinação perigosa em qualquer dos depósitos.

1.27-1 Escreva a função $f(A, B, C, D)$ para o circuito da Fig. P1.27-1 e simplifique.



1.28-1 (*a*) Um algoritmo para a conversão do código binário para o código refletido da Seção 1.28 é o seguinte:

1. O dígito mais à esquerda do código refletido é o dígito mais à esquerda (mais significativo) do código binário.
2. O próximo dígito refletido é a soma, módulo 2, dos dois primeiros dígitos binários e assim por diante. (A soma módulo dois, também chamada soma EXCLUSIVE-OR, segue as seguintes regras: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 0.) Por exemplo,

Binário	1100	1100	1100	1100
Refletido	1	10	101	1010

Explique por que o algoritmo funciona.

(*b*) O algoritmo para a conversão inversa é mostrado no exemplo seguinte:

Refletido	1010	1010	1010	1010
Binário	1	11	110	1100

Explique por que o algoritmo funciona.

1.30-1 Em uma palavra de quatro bits, um deles é um bit de paridade usado para estabelecer paridade par. Mostre como portas EXCLUSIVE-NOR podem ser usadas para determinar se a paridade foi mantida ou não.

Capítulo 2

2.1-1 Reescreva as seguintes expressões na forma padrão de soma de produtos:

(*a*) $f(A, B, C, D) = (A\bar{B} + \bar{A}B)(C + \bar{C}D)$
(*b*) $f(X, Y, Z) = (\bar{X} + \bar{Y})(\bar{X}\bar{Y} + Z)$
(*c*) $f(A, B, C, D) = \overline{(\bar{A}\bar{B}C + D)}(B + \overline{C\bar{D}})$
(*d*) $f(A, B, C) = (\bar{A} + \bar{B})(A + \bar{A}B)(\bar{A} + \bar{B} + \bar{A}BC)$
(*e*) $f(A, B, C) = A \oplus \bar{B} \oplus C$

2.2-1 Reescreva as seguintes expressões na forma padrão de produto de somas:

(*a*) $f(A, B, C) = \overline{A\bar{B} + \bar{B}C}$
(*b*) $f(A, B, C, D) = \overline{(A + B + C)(B + C + \bar{D})}$
(*c*) $f(X, Y, Z) = (X + \bar{Y})\bar{X}Y + Z$
(*d*) $f(A, B, C) = (A \oplus B)C + \bar{A}(B \oplus C)$
(*e*) $f(A, B, C, D) = \overline{A\bar{B}} + \overline{B\bar{C}} + \overline{C\bar{D}}$

2.4-1 Expresse as seguintes funções como somas de mintermos:

(*a*) $f(A, B, C) = \overline{A + \bar{B} + C}$
(*b*) $f(A, B, C) = \overline{(A + \bar{B})(B + \bar{C})}$
(*c*) $f(A, B, C, D) = \overline{A\bar{B} + BC\bar{D}} + \bar{A}C\bar{D}$
(*d*) $f(A, B, C, D) = A\overline{(\bar{B} + C\bar{D})} + \bar{A}BC$

2.4-2 Expresse as seguintes funções como produtos de maxtermos:

(*a*) $f(A, B, C, D) = (\bar{A} + C)D + \bar{B}D$
(*b*) $f(X, Y, Z) = \overline{(\bar{X}Y + Z)}(Y + XZ)$
(*c*) $f(A, B, C, D) = \overline{A\bar{B}C\bar{D}}$
(*d*) $f(A, B, C, D) = \overline{A\bar{B}C + AB\bar{D}}$

2.5-1 Para cada uma das funções abaixo: (1) prepare uma tabela verdade, (2) expresse a função como uma soma de mintermos, (3) expresse a função como um produto de maxtermos e (4) expresse o complemento da função sob a forma de soma de mintermos e de produto de maxtermos.

(*a*) $f(A, B, C) = A\overline{(B + \bar{C})}$
(*b*) $f(A, B, C) = (\bar{A} + B)(A + B + \bar{C})(\bar{A} + C)$
(*c*) $f(A, B, C, D) = (A + \bar{B})(C + \bar{D})(\bar{A} + \bar{C})$
(*d*) $f(A, B, C, D, E) = \bar{A}E + BCD$

2.6-1 Para as expressões abaixo desenhe uma estrutura de portas que gera a função diretamente na forma em que está escrita. A seguir substitua a estrutura obtida por uma estrutura (1) AND-OR de dois níveis e (2) OR-AND de dois níveis.

(*a*) $f(A, B, C) = [\bar{A}(B + \bar{C}) + A]\bar{B} + C$
(*b*) $f(A, B, C, D) = [(A\bar{B} + C)D + \bar{A}\bar{C}]BD + A\bar{B}C\bar{D}$

2.7-1 (*a*) Desenhe uma estrutura de dois níveis, usando somente portas NAND, que gera a função $f(A, B) = A \oplus B$.
(*b*) Repita usando apenas portas NOR.

2.7-2 (*a*) Desenhe uma estrutura de portas NAND-NAND para a função $f(A, B, C, D) = A + \bar{B} + C\bar{D}$.
(*b*) Repita usando uma estrutura de portas NOR-NOR.

2.7-3 Mostre que uma estrutura AND-OR-INVERT (AOI) pode ser usada como uma porta NAND e, também, como uma porta NOR.

2.7-4 Para cada uma das funções do Prob. 2.5-1 desenhe uma estrutura de portas que implemente (*a*) usando somente portas NAND e (*b*) usando somente portas NOR.

2.7-5 Desenhe as estruturas de portas NAND-NAND e NOR-NOR para as funções:
(*a*) $f(A, B, C) = \Sigma(0, 1, 3, 7)$
(*b*) $f(A, B, C, D) = \Pi(1, 5, 6, 8, 11, 13)$

2.7-6 Implemente as funções abaixo usando uma porta AND-OR-INVERT:
(*a*) $f(X, Y, Z) = XY + \bar{X}\bar{Y}Z$
(*b*) $f(A, B, C, D) = \Sigma(2, 3, 5, 7, 9, 14)$
(*c*) $f(A, B, C, D) = \Pi(2, 3, 4, 7, 10, 11, 15)$

2.13-1 (*a*) Verifique se a função $f(A, B, C) = \Sigma(1, 3, 5)$ pode ser escrita na forma $f(A, B, C) = \bar{A}\bar{B}C + A\bar{B}C + \bar{A}C + \bar{B}C$.
(*b*) Cada um dos termos produto é chamado um *implicante* da função no sentido que, se o termo produto tem valor 1, isto implica em a função assumir valor 1. Um *implicante primo* é um implicante do qual nenhuma variável pode ser removida sem que o restante do termo perca sua propriedade de implicante. Encontre os implicantes primos e não-primos de $f(A, B, C)$.

2.13-2 Use um mapa K para encontrar as expressões mais simples das seguintes funções:
(*a*) $f(A, B, C) = \Sigma(0, 2, 3)$
(*b*) $f(A, B, C) = \Sigma(1, 2, 4, 6, 7)$
(*c*) $f(A, B, C) = \Sigma(0, 1, 2, 3)$
(*d*) $f(A, B, C) = \Sigma(0, 2, 4, 6)$
(*e*) $f(A, B, C) = \Sigma(0, 3, 5, 6)$

2.13-3 Repita o Prob. 2.13-2 fazendo com que os números que representam mintermos passem a representar maxtermos.

2.13-4 Use um mapa K para encontrar a expressão mais simples das seguintes funções:
(*a*) $f(A, B, C, D) = \Pi(0, 5, 7, 13, 14, 15)$
(*b*) $f(A, B, C, D) = \Pi(1, 4, 6, 8, 11, 13, 14)$
(*c*) $f(A, B, C, D) = \Pi(1, 2, 4, 5, 7, 8, 10, 11, 13, 14)$
(*d*) $f(A, B, C, D) = \Pi(0, 5, 7, 8, 9, 10, 11, 13)$

2.13-5 Repita o Prob. 2.13-4 fazendo com que os números que representam maxtermos passem a representar mintermos.

2.13-6 Use um mapa K para encontrar a expressão mais simples das seguintes funções:
(*a*) $f(A, B, C, D, E) = \Sigma(0, 4, 8, 12, 16, 20, 24, 28)$
(*b*) $f(A, B, C, D, E) = \Sigma(0, 2, 5, 8, 13, 15, 18, 21, 24, 29, 31)$
(*c*) $f(A, B, C, D, E) = \Sigma(3, 4, 6, 9, 11, 13, 15, 18, 25, 26, 27, 29, 31)$
(*d*) $f(A, B, C, D, E) = \Sigma(1, 5, 8, 10, 12, 13, 14, 15, 17, 21, 24, 26, 31)$

2.13-7 Repita o Prob. 2.13-6 fazendo com que os números que representam mintermos passem a representar maxtermos.

2.13-8 Use um mapa K para encontrar a expressão mais simples das seguintes funções:
(*a*) $f(A, B, C, D, E, F) = \Pi(2, 3, 6, 7, 10, 14, 18, 19, 22, 23, 27, 37, 42, 43, 45, 46)$
(*b*) $f(A, B, C, D, E, F) = \Pi(6, 9, 13, 18, 19, 25, 27, 29, 41, 45, 57, 61)$
(*c*) $f(A, B, C, D, E, F) = \Pi(4, 5, 6, 7, 8, 18, 20, 23, 25, 26, 27, 28, 37, 38, 42, 44)$

2.13-9 Repita o Prob. 2.13-8 fazendo com que os números que representam maxtermos passem a representar mintermos.

2.14-1 Use um mapa K para simplificar as seguintes funções:
(*a*) $f(A, B, C, D) = \bar{A}\bar{B}C + AD + B\bar{D} + C\bar{D} + A\bar{C} + \bar{A}\bar{B}$
(*b*) $f(A, B, C, D) = (A + B + \bar{C})(\bar{B} + \bar{D})(\bar{A} + C)(B + C)$
(*c*) $f(A, B, C, D, E) = A + BC + \bar{C}D\bar{E}$
(*d*) $f(A, B, C, D, E) = (A + B)(B + C)(C + \bar{D})(D + \bar{E})$

2.15-1 Use um mapa K para simplificar as seguintes funções:
(*a*) $f(A, B, C, D) = \Sigma m(0, 1, 4, 5, 9, 11, 14, 15) + d(10, 13)$
(*b*) $f(A, B, C, D) = \Sigma m(0, 13, 14, 15) + d(1, 2, 3, 9, 10, 11)$
(*c*) $f(A, B, C, D) = \Sigma m(0, 6, 9, 10, 13) + d(1, 3, 8)$
(*d*) $f(A, B, C, D) = \Sigma m(1, 4, 7, 10, 13) + d(5, 14, 15)$
(*e*) $f(A, B, C, D, E) = \Sigma m(1, 4, 6, 10, 20, 22, 24, 26) + d(0, 11, 16, 17)$

2.15-2 Um circuito lógico tem cinco entradas e uma saída. As quatro entradas A, B, C e D representam um dígito decimal em *BCD*. A quinta entrada é uma entrada de controle. Quando a entrada de controle estiver em 0 lógico, a saída deve ser 0 lógico se o número decimal for par e 1 lógico se o número decimal for ímpar. Quando a entrada de controle estiver em 1 lógico, a saída deve ser 0, exceto se o número for múltiplo de 3. Projete o circuito.

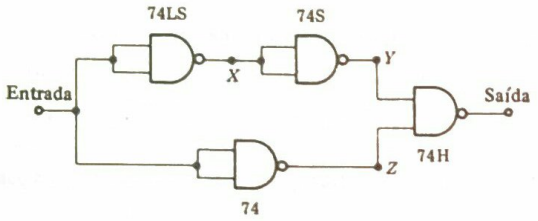
2.15-3 Um circuito lógico aceita como entradas dois números de dois bits $A = A_1A_0$ e $B = B_1B_0$ e gera, como saída, um número de quatro bits $P = P_3P_2P_1P_0$ que é o produto numérico de A e B. Projete o circuito.

Capítulo 3

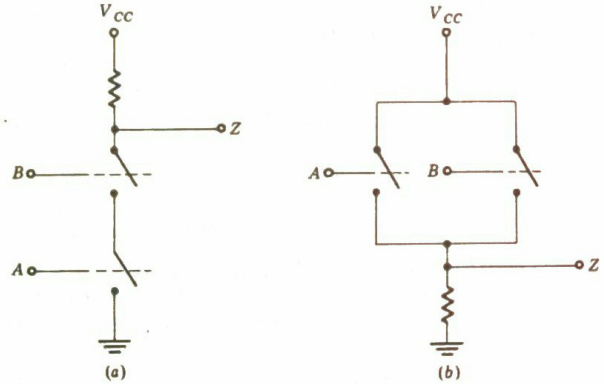
3.3-1 Portas TTL com características como as da Fig. 3.3-2 são ligadas individualmente para funcionar como inversores e então ligadas em cascata, a saída do primeiro inversor ligada à entrada do segundo, e assim por diante. A entrada do primeiro inversor é mantida a 2,4 V. Supondo que uma tensão espúria, originada de uma fonte de ruído, com uma amplitude de 0,5 V é adicionada a cada junção, sendo a polaridade do ruído sempre na direção

errada, pergunta-se: quantas portas podem existir na cascata antes que a saída de uma porta apresente uma tensão que está mais próxima do nível errado de saída que do nível certo?

3.3-2 No circuito da Fig. P3.3-2 uma transição abrupta do nível lógico 1 para o nível lógico 0 aparece na entrada para $t = 0$. Desenhe as transições dos níveis de tensão nos pontos X, Y e Z e na saída usando uma mesma escala de tempo. Ignore os tempos de subida e descida das portas.



3.8-1 Duas chaves operadas por lógica são ligadas conforme as Figs. P3.8-1*a* e *b*. Para cada caso encontre as relações lógicas entre Z e A e B.



3.11-1 Existem disponíveis as variáveis A, $\bar{A}$, B, $\bar{B}$, C e D ($\bar{C}$ e $\bar{D}$ não são disponíveis). Mostre como se pode gerar a função

$$Z = (\bar{A} + \bar{B})(\bar{C} + \bar{D})(A + \bar{C})(B + \bar{D})$$

usando apenas um dos circuitos integrados da Tab. 3.11-1.

3.11-2 Use somente um dos CIs listados na Tab. 3.11-1 para gerar simultaneamente a função $Z = (\bar{A} + B + C)(D + E)$ e a função $(A + \bar{B} + D)(C + \bar{E})$. Todas as variáveis e seus complementos são disponíveis.

3.11-3 Usando dois CIs dos listados na Tab. 3.11-1, construa um circuito que pode ser usado para determinar se um grupo de três bits tem paridade ímpar ou não.

3.14-1 Na lógica *mista* descrevemos portas como sendo portas AND ou OR (não se fazendo referência a portas NAND ou NOR) e especificamos se os terminais de entrada recebem sinais ativos quando altos ou quando baixos e o nível em que a saída gerada é ativa. Qual é a descrição com lógica *positiva* de portas que, usando lógica mista, seriam descritas como:
(*a*) Uma porta OR com entradas e saída ativas quando baixas
(*b*) Uma porta AND com entradas e saída ativas quando baixas
(*c*) Uma porta OR com entradas ativas quando altas e saída ativa quando baixa
(*d*) Uma porta AND com entradas ativas quando altas e saída ativa quando baixa
(*e*) Uma porta OR com entradas ativas quando baixas e saída ativa quando alta
(*f*) Uma porta AND com entradas ativas quando baixas e saída ativa quando alta

3.14-2 Uma porta EXCLUSIVE-OR é descrita em lógica *mista* como uma porta que aceita duas entradas, por exemplo X e Y, e fornece uma saída quando X ou Y for ativa mas não quando ambas forem ativas.
(*a*) Desenhe o símbolo para uma porta que, em lógica mista, aceite X e Y baixas e gere uma saída Z alta.
(*b*) Desenhe o símbolo para uma porta que aceite X e Y altas e gere Z baixa.
(*c*) Mostre que as portas especificadas em (*a*) e (*b*) são fisicamente como uma porta que aceita ambas as entradas ativas quando altas e gera uma saída ativa quando baixa.

3.14-3 Requer-se uma estrutura lógica com portas com quatro entradas A, B, C e D para gerar uma saída Z que é ativa quando uma ou outra das seguintes condições é verificada, mas *não* quando ambas são verificadas: (1) ambas A e B são ativas; (2) C ou D ou ambas são ativas.
(*a*) Suponha que A e B são disponíveis ativas quando altas e C e D são disponíveis ativas quando baixas e que Z deve ser ativa quando baixa.
(*b*) Suponha que todas as entradas e a saída são ativas quando altas.
Em ambos os casos mostre como as estruturas podem ser construídas usando portas encontradas comercialmente como as da Tab. 3.11-1.

3.14-4 Entradas A baixa, B alta, C baixa e D alta estão disponíveis. Desenhe as estruturas usando portas da Tab. 3.11-1 que geram saídas Z altas dadas por:
(*a*) $Z = (A + D)(\bar{B} + C)$
(*b*) $Z = A\bar{B}\bar{C} + BD$
(*c*) Repita (*a*) e (*b*) se o requisito for que Z seja ativa quando baixa.

3.14-5 Entradas A baixa, B alta, C baixa e D alta estão disponíveis. Desenhe as estruturas usando portas da Tab. 3.11-1 que geram saídas Z baixas dadas por:
(*a*) $Z = \overline{(A + B + C)}(\bar{B} + D)$
(*b*) $Z = (A + \bar{B} + C)(C \oplus D)$
(*c*) $Z = A + B + C + D$

3.14-6 Uma caixa contém uma lâmpada que é ligada em resposta à ativação de seu terminal de entrada que é ativo quando baixo. A lâmpada é controlada pelos seguintes sinais: (1) ligar a lâmpada (LL-H), (2) inibição (I-L), emergência (EMERG-L) e (4) hora errada (HE-H). A lâmpada deve acender quando houver instrução nesse sentido desde que seja hora certa e não haja inibição por um comando contrário. Por outro lado, se houver uma emergência,

a lâmpada deve acender independentemente de qualquer outra instrução. Desenhe um diagrama lógico do circuito usando lógica mista e mostre sua implementação física usando portas de lógica positiva.

3.15-1 Mostre como dois decodificadores tipo '138 (Fig. 3.15-2) podem ser combinados para construir um decodificador com quatro linhas de endereço de entrada e dezesseis linhas de saída.

3.15-2 Mostre como usar nove decodificadores tipo '138, como os da Fig. 3.15-2, (e alguns inversores) para construir um decodificador com seis linhas de endereço de entrada e 64 linhas de saída.

3.15-3 Desenhe a estrutura de portas de um decodificador que aceita a representação de dígitos decimais em BCD com quatro bits e possui dez saídas, cada uma das quais é selecionada em correspondência com um dígito decimal.

3.15-4 Na Fig. 3.15-4 quais pinos de entrada devem ser levados a que nível de tensão lógico (H ou L) para que o pino 10 tenha nível baixo?

3.16-1 Um decodificador tem oito linhas de entrada $I_0, \ldots, I_7$ e gera uma palavra de saída de três bits. A palavra de saída é a representação em código refletido do número decimal correspondente à entrada ativada. Mostre a estrutura de portas do decodificador.

3.16-2 Verifique que o codificador com prioridade da Fig. 3.16-3*a* funciona conforme descrito no texto. Verifique também os dados da tabela da Fig. 3.16-3*b*.

3.16-3 Verifique a descrição dada no texto da Seção 3.16 a respeito da operação em cascata dos codificadores com prioridade tipo 9318.

3.16-4 Desenhe o diagrama lógico de um codificador com prioridade que possua quatro entradas $I_0, \ldots, I_3$ e duas saídas A_1 e A_0. I_3 deve ter prioridade sobre I_2, e assim por diante. Inclua uma entrada de habilitação, uma saída de habilitação e uma saída com sinal de grupo. Faça com que todas as entradas e saídas sejam ativas quando altas.

3.17-1 Projete um conversor de código que aceite como entrada a representação BCD de um dígito decimal e, como saídas, linhas que acionem os segmentos apropriados de um display de sete segmentos que mostre o dígito decimal que aparece na entrada.

3.18-1 Mostre como dois multiplexadores do tipo mostrado nas Figs. 3.18-3 e 3.18-4 podem ser combinados (juntamente com algumas portas) para construir um multiplexador de dezesseis linhas para uma linha.

3.18-2 Mostre como nove multiplexadores do tipo mostrado das Figs. 3.18-3 e 3.18-4 podem ser combinados (juntamente com algumas portas) para construir um multiplexador de 64 linhas para uma linha.

3.18-3 No multiplexador descrito pelo símbolo lógico da Fig. 3.18-4 desejamos ter na saída o complemento do nível lógico do pino 14. Que entradas são necessárias e onde deve ser obtida a saída?

3.18-4 Use o multiplexador da Fig. 3.18-2 para gerar $Z = S_1 + S_0$.

3.18-5 Use o multiplexador da Fig. 3.18-2 para gerar $Z = S_1 S_0 + S_0 V + \bar{S}_1 \bar{S}_0 \bar{V}$.

3.18-6 Vimos que o multiplexador da Fig. 3.18-2 pode ser usado para gerar qualquer função de S_1 e S_0 ou qualquer função de S_1, S_0 e V desde que V esteja disponível nas formas complementada e não-complementada. Pode ser econômico em CI usar o multiplexador com mais variáveis, mesmo que seja necessário usar portas adicionais. Use o multiplexador para gerar $Z = S_1 \bar{S}_0 + S_0 W + VW + S_0 \bar{W}$.

Sugestão: Considere a possibilidade de as entradas I serem não só 1, 0, V ou $\bar{V}$ mas também alguma função lógica de V e W.

3.18-7 (*a*) Use um multiplicador de oito entradas para gerar a função $Z = \Sigma m$ (0, 3, 5, 6, 9, 10, 12, 15).
(*b*) Desenhe a estrutura que gera a mesma função usando somente portas NAND.
(*c*) Suponha que as portas NAND disponíveis são do tipo duplo de quatro entradas. Compare o número de CIs necessário para os dois tipos de implementação.

Capítulo 4

4.3-1 Considere uma configuração de portas com acoplamento cruzado como o da Fig. 4.2-1 exceto que são usadas portas OR.
(*a*) Mostre que, quando os terminais de entrada de dados habilitam ambas as portas, os dois terminais de saída podem ambos permanecer no nível lógico 0 ou no nível lógico 1 e que, portanto, o circuito constitui-se em um latch, isto é, um circuito com dois estados estáveis.
(*b*) Mostre que através da manipulação dos terminais de dados é possível induzir uma transição em uma direção, mas não na outra.
(*c*) Repita as partes (*a*) e (*b*) para portas AND.
(*d*) Se uma porta for uma porta AND e a outra uma porta OR, mostre que o circuito resultante é um latch e que é possível induzir transições em ambas as direções.

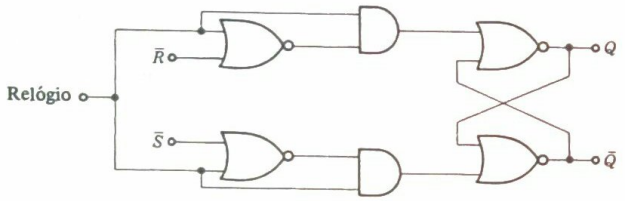
4.5-1 (*a*) No circuito de latch controlado da Fig. 4.5-1 adicione um terminal de entrada a cada uma das portas com acoplamento cruzado, isto é, as portas que estão diretamente ligadas a Q e $\bar{Q}$. Mostre que estes terminais adicionais são entradas de *set-direto* e *reset-direto* ativas quando baixas, portanto, $\bar{S}_d$ e $\bar{R}_d$ que podem ser usadas para tornar set e reset o latch antes que a entrada habilitadora torne-se ativa.
(*b*) Suponha que, na Fig. 4.5-1*b* por exemplo, $\bar{S}_d = 1$ e $\bar{R}_d = 0$, de modo que as entradas estejam ajustadas para tornar reset o latch. Suponha, ainda, que $D = 1$ e que HABILITA (*ENABLE*) seja ativada. Agora existe um conflito entre a instrução da entrada direta e a da entrada controlada. Mostre que, após HABILITA (*ENABLE*) ser desativada, o estado final do latch será determinado pelas entradas diretas, mas que, enquanto HABILITA (*ENABLE*) estiver ativada, a entrada controlada faz efeito.
(*c*) Mostre como o circuito pode ser modificado para que as entradas diretas *sempre prevaleçam* sobre a entrada controlada.

4.6-1 Na Fig. 4.6-1 considere, no caso mais geral, que o sinal ligado ao terminal de entrada superior é um sinal B e ao terminal inferior é um sinal C. Adicione mais uma porta que gere o sinal $\overline{B} + C$ e aplique este sinal à porta G_3. Mostre que neste caso a relação entre Z e A, B e C não se altera, mas que o perigo mostrado na Fig. 4.6-1*b* é eliminado. (O assunto de eliminação de perigos será considerado mais longamente na Seção 7.19.)

4.8-1 Mostre que os controles de set e reset-direto do flip-flop mestre-escravo da Fig. 4.8-4 prevalecem totalmente sobre as entradas síncronas.

4.13-1 (*a*) Mostre que o circuito da Fig. P4.13-1 é um flip-flop gatilhado pela borda cujo funcionamento depende do atraso de propagação das portas NOR de entrada.

(*b*) Mostre que o flip-flop é apropriado para uso em um sistema síncrono, pois a saída responde à transição do sincronismo que desabilita as portas NOR.

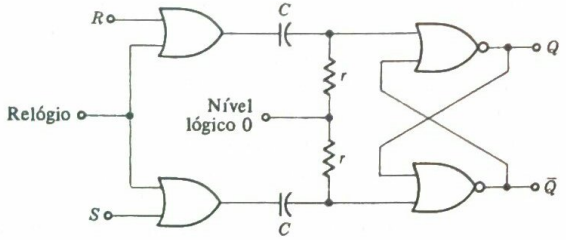


4.13-2 O circuito de flip-flop da Fig. P4.13-2 é usado em lógica acoplada pelo emissor (ECL).

(*a*) Mostre que é um flip-flop gatilhado pela borda apropriada para uso em sistemas síncronos, pois a saída responde à transição do sincronismo que desabilita as portas de entrada.

(*b*) Mostre que o circuito tem as limitações características dos circuitos que dependem do atraso de propagação, isto é, dependência da velocidade de transição do sincronismo. Especificamente, mostre que o circuito pode não operar de modo adequado se o tempo de transição da forma-de-onda de sincronismo for muito longo em comparação com a constante de tempo rC.

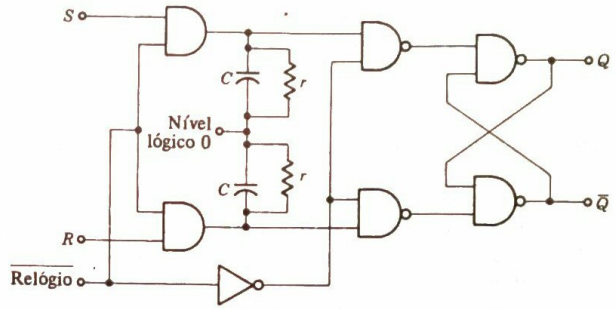
(*c*) Mostre que conexões de realimentação das saídas para as portas de entrada podem transformar o flip-flop em um flip-flop $\overline{JK}$.



4.13-3 O flip-flop da Fig. P4.13-3 é usado em lógica transistor-transistor (TTL). (Diodos de junção polarizados inversamente são usados na implementação física das combinações rC mostradas no circuito.)

(*a*) Descreva a operação do circuito e mostre que é apropriado para uso em um sistema síncrono, pois a saída responde às transições do sincronismo que desabilitam as portas de entrada.

(*b*) Leve em conta o fato de que o capacitor não pode carregar-se instantaneamente e que, uma vez carregado, o capacitor se descarrega através de r. Discuta qualitativamente as considerações de temporização correspondentes que devem ser levadas em conta relativamente às entradas de dados e à forma-de-onda de sincronismo.



4.14-1 Verifique todas as afirmações feitas na Seção 4.14 sobre o funcionamento do flip-flop tipo D da Fig. 4.14-3.

4.14-2 Suponha que todas as portas NAND da Fig. 4.14-3 fossem substituídas por portas NOR. Reescreva a descrição do funcionamento do flip-flop.

4.14-3 Sem adicionar portas, modifique o flip-flop da Fig. 4.14-3 de modo a equipá-lo com entradas set-direto e reset-direto, que sempre prevaleçam sobre a entrada síncrona.

4.14-4 Verifique que o circuito da Fig. 4.14-5 opera conforme é esperado de um flip-flop JK.

4.16-1 (*a*) Temos disponíveis dois registradores de n bits A e B onde os elementos armazenadores individuais são latches SR estáticos. Temos, ainda, um sinal habilitador de transferência RA → RB. Mostre a estrutura através da qual podemos transferir o conteúdo de RA para RB.

(*b*) Suponha, agora, que temos dois sinais: um (0 → RA) que pode ser usado para limpar RA e outro que é o sinal de transferência RA → RB. Mostre como o hardware necessário pode ser simplificado se pudermos aceitar uma operação de transferência em dois passos.

4.17-1 No registrador de deslocamento de quatro bits da Fig. 4.17-1 suponha que, inicialmente, $Q_0 = Q_1 = 0$ e $Q_2 = Q_3 = 1$. Desenhe a forma-de-onda de cada flip-flop se a seqüência de entrada 10101 for aplicada a D_0 sincronamente com o pulso de relógio.

4.18-1 Desenhe o diagrama lógico de um registrador de três-estados com deslocamento para a esquerda e deslocamento para a direita usando flip-flops tipo D. O registrador deve ter terminais D_{DD} (entrada de dados para deslocamento para a direita), D_{DE} (entrada de

dados para deslocamento para a esquerda), M (controle de modo para determinar a direção do deslocamento) e um terminal de sincronismo.

4.20-1 (*a*) Considere um contador em anel de quatro bits como o da Fig. 4.20-1 em que a seqüência de contagem é $Q_0Q_1Q_2Q_3 = 1000 \rightarrow 0100 \rightarrow 0010 \rightarrow 0001 \rightarrow 1000$ etc. Mostre que, se por um acidente devido a um ruído espúrio o contador chegar a qualquer outro estado, ele circulará pelos estados não permitidos sem retornar à seqüência desejada.

(*b*) Agora considere a modificação em que D_0 não é mais ligada a Q_3 mas a uma porta de modo a obter $D_0 = \bar{Q}_0\bar{Q}_1\bar{Q}_2$. Mostre que neste caso o contador automaticamente se corrige, sempre retornando à seqüência desejada, independentemente do estado de onde partir.

4.21-1 Projete um decodificador para um contador em anel torcido de cinco bits para obter uma indicação explícita da contagem.

4.21-2 (*a*) Considere o contador em anel torcido de quatro bits da Fig. 4.21-1 cuja seqüência de contagem é dada. Suponha que, por um acidente, quando a alimentação é aplicada ou devido a algum distúrbio, o contador se encontre em um estado fora da seqüência desejada. Mostre que o contador continuará a circular pelos estados indesejáveis sem retornar à seqüência desejada.

(*b*) Modifique o contador substituindo o FF_0 por um flip-flop JK cujas entradas serão $J_0 = \bar{Q}_2\bar{Q}_3$ e $K_0 = Q_3$. Mostre que agora o contador se corrige automaticamente, retornando sempre à seqüência desejada, independentemente do estado de onde partir.

4.21-3 O contador em anel torcido seqüencia através de um número par de estados igual a $2N$, sendo N o número de flip-flops. Considere um contador com $N = 3$ e FF_0, FF_1 e FF_2, onde FF_0 é um flip-flop JK. Construa um contador no qual Q_0 é ligada a D_1 e Q_1 a D_2 e onde $J_0 = Q_1$ e $K_0 = \bar{Q}_2$. Mostre que este contador seqüencia através de um número ímpar de estados (cinco).

4.24-1 Projete um contador síncrono de módulo 3 usando flip-flops JK. Faça com que a seqüência de estados seja $Q_1Q_0 = 00 \rightarrow 11 \rightarrow 10 \rightarrow 00$ etc.

4.24-2 Projete um contador síncrono de módulo 5 que siga a seqüência dada na Tab. 4.24-2 usando:

(*a*) flip-flops SR

(*b*) flip-flops tipo D

(*c*) flip-flops tipo T

4.24-3 Verifique os mapas K da Fig. 4.24-2 e verifique a leitura do mapa resultante na Eq. (4.24-1).

4.24-4 Projete um contador síncrono de módulo 5 que seqüencie através dos estados $S_0 \rightarrow S_2 \rightarrow S_4 \rightarrow S_6 \rightarrow S_7 \rightarrow S_0$ etc., sendo os estados definidos pela Tab. 4.24-1, usando (*a*) flip-flops JK e (*b*) flip-flops tipo D. Verifique a não-ocorrência de ciclos em estados indesejáveis.

4.24-5 Projete um contador síncrono de módulo 7 que seqüencie, em ordem, através de todos os estados da Tab. 4.24-1, exceto S_3. Faça com que o contador vá para o estado S_0 se, acidentalmente, ele se encontrar no estado S_3. Use flip-flops tipo T.

4.24-6 (*a*) Construa um contador síncrono com módulo 6 que seqüencie através dos seis primeiros estados da Tab. 4.24-1, na mesma ordem. Use flip-flops tipo D.

(*b*) Projete um contador com módulo 3 que seqüencie através dos três primeiros estados da Tab. 4.24-1. Construa um contador com módulo 6 cascateando o contador de módulo 3 com um contador com módulo 2. O contador de módulo 2 deve mudar de estado sempre que o contador de módulo 3 completar sua seqüência.

(*c*) Desenhe as formas-de-onda para ambos os projetos de contador com módulo 6.

(*d*) Para cada caso projete um decodificador de contagem.

4.24-7 Refira-se às formas-de-onda de temporização do contador síncrono de módulo 8 dadas na Fig. 4.22-1*b*. O decodificador é mostrado na Fig. 4.22-1*c*. Seja T o período do pulso de sincronismo. Suponha que o atraso de propagação de um flip-flop seja $t_{ap} = 0{,}1T$ quando o flip-flop passa do estado reset para o set e $t_{ap} = 0{,}2T$ quando passa do estado set para o reset. Redesenhe as formas-de-onda levando em conta os atrasos de propagação. Na mesma escala de tempo desenhe as formas-de-onda das saídas $K_0, \ldots, K_7$ do decodificador. Mostre que erros transitórios de decodificação ocorrem devido aos atrasos desiguais de propagação.

4.25-1 Desenhe o diagrama lógico de um contador cuja lógica é dada pelas Eqs. (4.25-1) a (4.25-3).

4.25-2 Projete um contador síncrono incrementador-decrementador com módulo 8.

4.25-3 Projete um contador síncrono incrementador-decrementador com módulo 7 que seqüencie em ordem através de todos os estados da Tab. 4.24-1, exceto o estado S_0.

4.25-4 Projete um contador síncrono incrementador-decrementador com módulo 10.

4.27-1 Verifique os registros da Tab. 4.27-1.

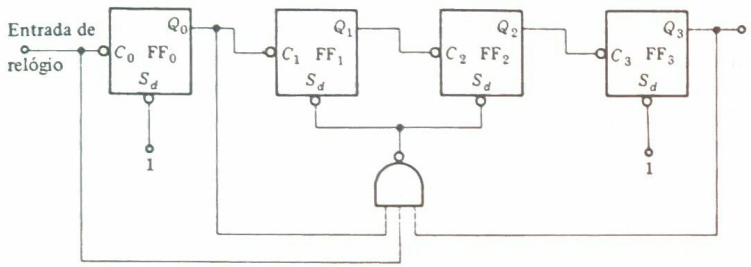
4.27-2 (*a*) Mostre que o contador assíncrono da Fig. 4.27-1 pode ser convertido em um contador incrementador-decrementador com controle de modo (direção) M substituindo a conexão direta $C_{k+1} = Q_k$ por uma estrutura com portas tal que $C_{k+1} = MQ_k + \bar{M}\bar{Q}_k$. Desenhe o circuito lógico de um contador com módulo 8.

(*b*) Mostre que o projeto de (*a*) tem a característica indesejável de mudar a contagem em algumas ocasiões em que se muda a direção de contagem. Mostre como se pode mudar o projeto para evitar esta dificuldade.

4.27-3 (*a*) Um contador assíncrono de módulo 10 (década) é mostrado na Fig. P4.27-3. Nos FF_0 e FF_3 as entradas de set-direto ativas quando baixas são mantidas permanentemente inativas pela ligação a uma tensão equivalente ao 1 lógico. Os flip-flops chaveiam na borda negativa do pulso aplicado à entrada de sincronismo. Desenhe dez ciclos de um sinal de entrada de relógio e na mesma escala temporal desenhe as formas-de-onda de Q_0, Q_1, Q_2 e Q_3 começando com o estado $Q_0Q_1Q_2Q_3 = 0000$. Mostre que após a nona transição do sinal de relógio temos $Q_0Q_1Q_2Q_3 = 1001$ (= 9). Mostre que a transição positiva do relógio imediatamente seguinte leva o contador a $Q_0Q_1Q_2Q_3 =$

= 1111 (durante este estado o decodificador de contagem deve ser desabilitado) e que a próxima borda negativa do sinal de relógio retorna o contador ao estado inicial.

(*b*) Generalize a idéia de projeto de (*a*). Desenhe os diagramas lógicos de contadores de módulo 14, 19 e 27.



4.27-4 Os flip-flops no contador assíncrono da Fig. P4.27-4 são do tipo $\overline{JK}$ com terminais de dados ativos quando baixos. Eles respondem à transição positiva do sinal aplicado à entrada de sincronismo. Possuem ainda a característica de que, quando o sincronismo for alto, uma transição positiva no terminal $\overline{J}$ torna set o flip-flop e no terminal $\overline{K}$ torna reset o mesmo. (Estes são os flip-flops apresentados no Prob. 4.13-2 modificados para $\overline{JK}$.) Na Fig. P4.27-4 todos os terminais $\overline{J}$ e $\overline{K}$ não indicados são mantidos no nível lógico 0. No último flip-flop o sincronismo é mantido no nível alto.

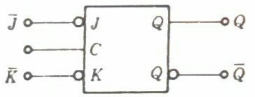
(*a*) Mostre que, se ignorarmos a conexão a $\overline{J}_0$, os N primeiros flip-flops constituem um contador assíncrono de módulo 2^N.

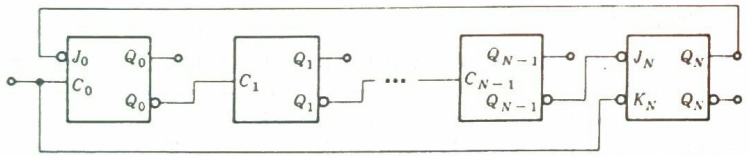
(*b*) Mostre que, em virtude da adição do último flip-flop e da realimentação $\overline{J}_0$, o contador tem módulo $2^N + 1$ (e não 2^{N+1}).

(*c*) Usando o princípio ilustrado e somente com flip-flops (sem usar portas), construa um contador com módulo 3 e um com módulo 5.

(*d*) Construa um contador com módulo 7, 11 e 25.

Sugestão: $7 = 2(2 + 1) + 1; 11 = 2(2^2 + 1) + 1; 25 = 2^3(2^1 + 1) + 1$.



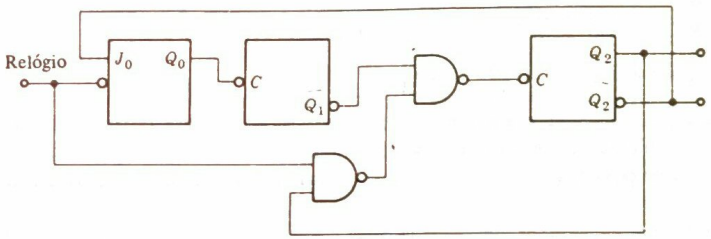


4.27-5 Contadores assíncronos de módulo arbitrário podem ser construídos adicionando portas a um arranjo contador de flip-flops de modo a fazer com que o contador salte alguns estados. Não existe método para o projeto e é necessário usar engenhosidade. Um exemplo é dado na Fig. P4.27-5. Todas as entradas J e K não mostradas explicitamente estão no nível lógico 1.

(*a*) Desenhe as formas-de-onda para o contador e verifique se possui módulo 5.

(*b*) Usando o formato sugerido pelo contador mostrado, desenhe o diagrama lógico de um contador com módulo 9.

(*c*) Tente projetar um contador com módulo 7.



4.28-1 Verifique que os flip-flops B, C e D da Fig. 4.28-1*c* formam um contador de módulo 5.

4.28-2 Desenhe as formas-de-onda para o contador da Fig. 4.28-1*c* para os seguintes casos:

(*a*) O pulso de relógio externo é aplicado a $\overline{CP_0}$.

(*b*) O pulso de relógio externo é aplicado a $\overline{CP_1}$.

Capítulo 5

5.1-1 Efetue as seguintes somas e subtrações de números no sistema binário:

(*a*) 1101 + 110
(*b*) 101011 + 101010
(*c*) 1111 + 1
(*d*) 111 + 11
(*e*) 1000 − 111
(*f*) 10100 − 1111
(*g*) 10001 − 1110
(*h*) 10101 − 1010

5.2-1 Para acomodar números cujo intervalo se estende por muitas ordens de grandeza, os computadores armazenam números binários em uma representação com ponto flutuante, sendo o número N representado por $N = F2^E$. F é uma fração cujo bit mais significativo é 1 e E é um expoente da base 2. Dois registradores são usados, um para F e outro para E. Assim $F = S_F F_{-1} F_{-2} \ldots$ e $E = S_E E_k E_{k-1} \ldots$, onde S_F e S_E são os bits de sinal. Supondo que os números negativos, sejam partes fracionais ou expoentes, são especificados na representação por complemento de dois. Escreva o conteúdo dos registradores para os seguintes números, supondo que F é um registrador de oito casas e E é um registrador de quatro casas:

(*a*) +110,101
(*b*) −110,101
(*c*) +0,00001101
(*d*) −0,00001101

5.2-2 Encontre a representação por complemento de dois no sistema binário dos seguintes números decimais:

(*a*) +17 (*c*) +32
(*b*) −17 (*d*) −32

Em cada caso use o menor registrador permissível.

5.2-3 Usando a representação por complemento de dois, efetue as seguintes somas e subtrações, expressando o resultado na forma sinal-magnitude:

(*a*) $A + B$ (*c*) $-A + B$
(*b*) $A - B$ (*d*) $-A - B$

onde a magnitude de A é 110101 e a de B é 110011.

5.2-4 No sistema decimal formamos o complemento de *dez* de um número de maneira correspondente ao complemento de *dois* no sistema binário. Assim, em um registrador decimal de duas casas com posições 00, 01, . . . , 98, 99, o complemento de dez de 37 é 100 − 37 = = 63.

(*a*) Desenhe um registrador decimal de duas casas e indique a significância numérica em sinal e magnitude das posições do registrador.
(*b*) Qual é o maior número positivo representado?
(*c*) Qual é o número negativo de maior magnitude?
(*d*) Em um registrador com um número arbitrário de posições, como podemos saber, através do dígito mais à esquerda, se a posição do registrador representa um número positivo ou negativo?
(*e*) Usando somente a operação soma, e representando números negativos na forma de complemento de dez, calcule: 46 + 39, 46 − 39, − 46 + 39 e − 46 − 39.

5.2-5 Sem usar formalmente a operação de subtração, determine o complemento de dois de:

(*a*) 110111 (*c*) 1010100
(*b*) 00001 (*d*) 1100101

5.2-6 Suponha que combinemos dois números, cada um dentro do intervalo do registrador usado na representação por complemento de dois, com o resultado caindo fora do intervalo do registrador. Demonstre que podemos detectar a ocorrência de um erro examinando se há vai-um da posição mais significativa do registrador e se há vai-um da posição imediatamente menos significativa. Mostre que, se um ou outro vai-um for gerado, mas não ambos, o intervalo do registrador foi excedido.

5.3-1 Encontre a representação por complemento de um em um registrador de oito casas dos seguintes números decimais:

(*a*) +8 (*c*) +21
(*b*) −8 (*d*) −21

5.3-2 Usando a representação por complemento de um, efetue as seguintes somas e subtrações e expresse o resultado na forma de sinal e magnitude:

(*a*) $A + B$ (*c*) $-A + B$
(*b*) $A - B$ (*d*) $-A - B$

onde a magnitude de A é 111001 e a de B é 101101.

5.3-3 No sistema decimal formamos o complemento de *nove* de um número de maneira correspondente ao complemento de *um* do sistema binário. Assim, em um registrador decimal de duas casas com posições 00, 01, . . . , 98, 99, o complemento de nove de 37 é 99 − 37 = = 62. Repita para o caso do complemento de nove todas as partes do Prob. 5.2-4, que se referia ao caso do complemento de dez.

5.4-1 O somador inteiro da Fig. 5.4-3 requer nove portas.
(*a*) Mostre que, se C_{i+1} estiver disponível, S_i pode ser gerado por

$$S_i = \bar{C}_{i+1}(A_i + B_i + C_i) + A_iB_iC_i$$

e que desta maneira o somador inteiro pode ser implementado com oito portas. A economia de uma porta é, necessariamente, uma vantagem? Compare o número de níveis de portas através dos quais um sinal deve passar.

5.4-2 (*a*) Verifique que

$$X \oplus Y = \overline{\overline{XY}\,X}\;\;\overline{\overline{XY}\,Y}$$

e que

$$XY + XZ + YZ = \overline{\overline{XY}\,\overline{(X \oplus Y)\,Z}}$$

(*b*) Use os resultados de (*a*) para mostrar que um somador inteiro pode ser implementado com nove portas NAND com duas entradas cada uma. Como se compararia o atraso de propagação deste arranjo com o da estrutura da Fig. 5.4-3?

5.4-3 Derive as equações lógicas para um somador inteiro que acomode dois números de dois bits mais um bit de vai-um de entrada. A unidade aceita entradas $A_{i+1}A_i$ e $B_{i+1}B_i$ e C_i e gera saídas $S_{i+1}S_i$ e C_{i+2}.

5.4-4 Projete um circuito lógico combinacional que aceite como entrada um número de três bits e gere uma saída que é o quadrado do número.

5.4-5 Um *comparador* é um circuito digital que aceita dois números A e B como entradas e gera três saídas. A primeira, segunda ou terceira saída será ativa se $A = B$, $A > B$ ou $A < B$. Projete um comparador combinacional que acomode dois números de três bits.

5.5-1 O somador série da Fig. 5.5-1 tem os números 01101 e 00111 armazenados nos dois registradores de parcelas de cinco bits. Desenhe gráficos de tempos para a operação do somador durante seis intervalos de relógio. Comece com o sinal de relógio e abaixo, na mesma escala de tempo, desenhe os sinais da saída do somador inteiro e da saída do flip-flop tipo D.

5.6-1 No somador paralelo da Fig. 5.6-1 suponha que os bits soma e vai-um sejam gerados pelas estruturas de portas da Fig. 5.4-3*c*. Ignore o atraso associado com a geração dos complementos das variáveis, tais como os que são necessários no circuito para gerar S_i, e suponha que o atraso é igual para todas as portas. Suponha um somador de quatro bits, $C_0 = 0$, e que imediatamente antes de $t = 0$, $A = B = 0$, de modo que os bits de soma e vai-um de saída também são 0. Em $t = 0$ aplicamos $A = 1111$ e $B = 0001$. Mostre a seqüência de trocas por que passa a saída $C_4S_3S_2S_1S_0$ iniciando em 00000 e terminando em 10000.

5.7-1 No calculador da Fig. 5.7-1 o registro no acumulador é -4. As chaves de entrada estão acionadas para subtrair 5. Quais são os valores de C_4, S_3, S_2, S_1 e S_0? Quando a chave SW (registrador) for acionada, qual será a indicação de saída subseqüente do calculador?

5.7-2 (*a*) Considere que uma seqüência de números foi somada e subtraída do acumulador da Fig. 5.7-1. Suponha que a acumulação final está dentro do intervalo do acumulador, mas que, durante a seqüência de operações, tenha havido um overflow fora do intervalo do acumulador. Mostre que, apesar de tudo, o resultado final estará correto. Mostre que este resultado ainda se aplica se for usada a representação por complemento de dois.

(*b*) Passo a passo forme a acumulação $-5-6+4$ e mostre que o resultado final é -7, embora a combinação dos dois primeiros números resulte em -11, que está fora do intervalo do acumulador.

5.8-1 (*a*) Mostre que um subtrator inteiro pode ser implementado com dois meio-subtratores e uma porta OR.

(*b*) Desenhe as estruturas de portas de dois níveis para a geração de D_i e C_{i+1} conforme a tabela verdade da Fig. 5.8-1.

5.8-2 Mostre que um somador inteiro pode ser convertido em um subtrator inteiro pelo uso de um único inversor.

5.10-1 Mostre que a Eq. (5.10-4) pode ser reescrita na forma dada na Eq. (5.10-5).

5.10-2 Verifique a expressão dada na Eq. (5.10-7) para os bits de vai-um.

5.10-3 Considere que no somador com vai-um antecipado da Fig. 5.10-2 a variável P_i é gerada por uma porta OR com atraso t_{ap} em vez de por uma porta EXCLUSIVE-OR. Encontre o atraso total do somador.

5.10-4 Em um manual de características TTL, procure o somador binário de quatro bits com vai-um rápido tipo '83A e verifique que os bits soma são gerados usando duas portas EXCLUSIVE-OR como na Fig. 5.10-2, mas que P_i é gerado como $P_i = A_i + B_i$.

5.12-1 Na Fig. 5.12-1 todos os bits A_i e B_i bem como C_0 são aplicados em $t = 0$. Quanto tempo devemos esperar para termos certeza de que a soma disponível está correta?

5.13-1 Verifique os dados da Tab. 5.13-1 para a ULA simples da Fig. 5.13-1.

5.13-2 Na Seção 5.13 há uma tabela que lista as magnitudes relativas das entradas A e B da ULA em termos da entrada de vai-um C_n aplicada e da saída de vai-um C_{n+4} observada. Verifique os valores da tabela.

5.14-1 Para o circuito da Fig. 5.14-2 mostre que, sempre que a soma gerada pela linha superior de somadores estiver no intervalo de 10 a 15, uma saída de vai-um será gerada e que $6 = 0110$ será adicionado à soma.

5.14-2 Projete um circuito combinacional que aceite um número binário de quatro bits e gere o número decimal correspondente (dois dígitos) na representação BCD.

5.14-3 Converta os números decimais 267 e 134 para BCD e execute a soma dos números.

5.15-1 Execute as seguintes multiplicações e divisões:

(*a*) 1101×101 (*c*) $1001101 \div 101$

(*b*) 1001×1111 (*d*) $110011 \div 1111$

Capítulo 6

6.1-1 Uma RAM armazena palavras de quatro bits. Tem terminais de entrada e saída separados, dois controles CS e WE e dois pinos de alimentação. A memória é encapsulada em um invólucro de 24 pinos, todos eles usados. Qual é a organização da memória?

6.3-1 Usando a memória da Fig. 6.1-1, desenhe um diagrama mostrando como ela pode constituir uma memória de oito palavras e dezesseis bits/palavra.

6.3-2 Usando a memória da Fig. 6.1-1, desenhe um diagrama mostrando como ela pode constituir uma memória de 32 palavras e quatro bits/palavra.

6.4-1 Uma RAM tem organização de memória de 16 384 palavras e um bit/palavra. Compare o número de chaves e portas necessário para acessar cada bit para organização interna uni e bidimensional.

6.5-1 O conteúdo de uma ROM de quatro palavras e oito bits/palavra é especificado na tabela. Desenhe as correspondentes estruturas do decodificador e do codificador.

Endereço		Palavra de dados							
A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	1	1	0	0	1	1	0
0	1	1	0	1	0	1	0	1	0
1	0	1	1	1	0	0	0	1	1
1	1	0	0	1	1	1	0	0	1

6.5-2 Temos uma ROM de 2048 palavras e cinco bits/palavra com onze entradas de endereço. Necessitamos uma ROM de 8 192 palavras e um bit/palavra, com treze entradas de endereço. Desenhe um diagrama mostrando como a conversão pode ser executada usando um multiplexador.

6.6-1 Desenhe a estrutura de diodos e resistores do decodificador da ROM cujo conteúdo é especificado no Prob. 6.5-1.

6.9-1 Uma ROM tem parâmetros $t_{AA} = 250$ ns, $t_{CE} = 150$ ns e $t_{OH} = 100$ ns.
(*a*) Um endereço válido é aplicado em $t = 0$, a entrada de seleção de circuito é ativada em $t = 50$ ns e um novo endereço é apresentado em $t = 300$ ns. Qual é o intervalo de tempo em que há saída válida correspondente ao primeiro endereço?
(*b*) Repita a parte (*a*) quando a seleção de circuito for ativada em $t = 200$ ns.

6.10-1 (*a*) Desejamos que o PLA da Fig. 6.10-1 seja programável pelo usuário. Cada porta AND deve ter acesso a qualquer dos Is de entrada ou seu complemento e cada porta OR deve ter acesso à saída de qualquer porta AND; podemos escolher saídas ativas quando baixas ou quando altas. Quantos fusíveis devem ser incorporados ao FPLA?
(*b*) Quantos fusíveis seriam necessários em uma PROM com o mesmo número de entradas e saídas do FPLA da parte (*a*)?

6.10-2 Uma ROM deve ser usada para gerar quatro funções F_0, F_1, F_2 e F_3 de quatro variáveis de entrada I_0, I_1, I_2 e I_3.

$$F_0 = I_0 + I_1 + I_2 + I_3 \qquad F_1 = I_0 I_1 I_2 I_3$$
$$F_2 = I_0 \oplus I_1 \oplus I_2 \oplus I_3 \qquad F_3 = (I_0 + I_1)(I_2 + I_3)$$

Projete a ROM e desenhe seu circuito lógico, isto é, a estrutura de portas AND e OR. Desenhe um segundo circuito em que a parte do codificador mostre os diodos individuais.

6.10-3 Um circuito combinacional deve ser implementado para que aceite um número de entrada de três bits e gere uma saída que seja o quadrado da entrada. Faça uma tabela verdade para o circuito. Mostre que o circuito pode ser implementado usando somente uma ROM que tenha um endereço de três bits e uma saída de quatro bits.

6.10-4 Um circuito combinacional deve ser implementado que gere o produto numérico de dois números de entrada de dois bits A_1A_0 e B_1B_0. Os bits de entrada são ativos quando altos, e as saídas devem ser ativas quando baixas. Projete o circuito seguindo o modelo de um PLA. Expresse a saída como uma soma de produtos e desenhe um diagrama da estrutura de portas necessária.

6.14-1 Reporte-se à Fig. 6.14-2*a*. Mostre que, se a posição da chave S_2 e a de seu resistor fossem trocadas, o circuito produziria uma saída I em vez de $\overline{I}$. Usando este resultado, modifique a Fig. 6.14-2*b* de modo a produzir a saída $\overline{Q}$ além da Q. Mostre que agora, se a saída for ligada à entrada, o flip-flop chaveará.

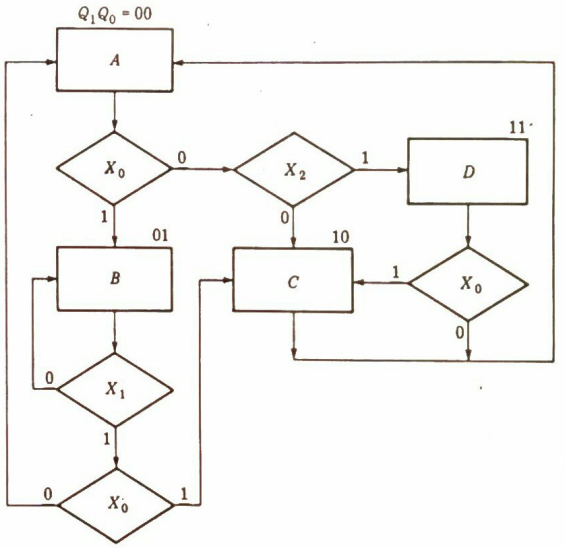
6.15-1 Suponha que, em um registrador de deslocamento acoplado por carga usado como memória série e possuindo cem estágios em cascata, a perda total de carga do início até o fim não deve exceder 5%. Qual é a eficiência de carga requerida, isto é, a fração da carga em uma posição que deve ser transferida para a posição seguinte?

Capítulo 7

7.3-1 Um sistema seqüencial síncrono deve permanecer num estado A até que uma chave mecânica seja fechada durante um tempo curto. No fechamento, o sistema deve ir para o estado

B. Em um segundo fechamento breve da chave, o sistema deve avançar para o estado C, e assim por diante. Na prática, o período de relógio será muito menor do que o intervalo em que a chave estará fechada. Desenhar um fluxograma do sistema.

7.3-2 Na carga de fluxo de quatro estados da Fig. P7.3-2 as atribuições de estado são dadas no canto superior direito acima de cada retângulo. Devem ser usados flip-flops do tipo D. Há três variáveis de entrada X_0, X_1 e X_2. Quando o sistema está no estado A e $X_0 = 1$, o estado seguinte é B, independentemente de X_1 e X_2 etc.
(*a*) Construir uma tabela de estados.
(*b*) Construir uma tabela de transições.
(*c*) Achar as equações de excitação dos flip-flops.
(*d*) Desenhar o circuito de maneira que se separe a lógica da memória.

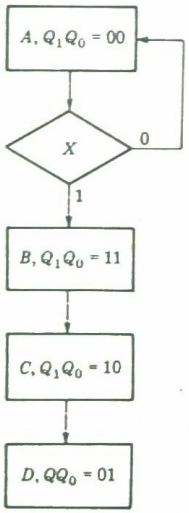


7.4-1 (*a*) Em uma interseção de tráfego de duas ruas, uma que corre para norte e sul (NS) e outra que corre para leste e oeste (EW), lâmpadas de tráfego são instaladas proporcionando indicações de vermelho, amarelo e verde. Construir um fluxograma que mostre a seqüência de estados através dos quais as lâmpadas passarão. Admitir que o sistema é comandado por um relógio cujo período é de 5 s. O tráfego em cada direção deve ser permitido durante 20 s e a transição de luz amarela deve ser permitida durante 5 s.
(*b*) Admitir que queremos tráfego NS interrompido exceto quando houver um carro na interseção querendo ir para EW. Um sensor é instalado para indicar os carros EW esperando. Modificar o fluxograma.
(*c*) Admitir que sensores separados são instalados, um para indicar um carro com destino oeste e um para indicar um carro com destino leste. Modificar o fluxograma.

(*d*) Admitir que na interseção há um botão que pode ser operado por um pedestre. Comprimindo-se o botão, faz-se com que todas as lâmpadas fiquem vermelhas durante 5 s ao fim do tempo comum alocado ao tráfego, de modo que os pedestres possam atravessar em qualquer direção. Modificar o fluxograma.

7.4-2 Uma máquina deve ser construída com a finalidade de orientar um jogador iniciante no jogo de dados. O jogo é executado com as seguintes regras. O jogador ganha se o primeiro arremesso for 7 ou 11 e o jogo está terminado. Do contrário, o jogador continua arremessando até que o número do primeiro arremesso se repita, dando um ganho, ou até que 7 ou 11 apareça, dando uma perda. A máquina opera como se segue. O jogador começa comprimindo o botão marcado NEW GAME (JOGO NOVO); em resposta a este sinal as lâmpadas acendem dizendo FIRST THROW (PRIMEIRO ARREMESSO). Tendo arremessado, o jogador ajusta as chaves para representar o número e depois comprime o botão marcado REGISTER (REGISTRADOR), transferindo deste modo o número para um registrador de armazenamento. Se o número for 7 ou 11, as lâmpadas de sinal acendem lendo YOU WIN (VOCÊ GANHA) e permanecem acesas até que NEW GAME (NOVO JOGO) seja iniciado; se o número não for 7 ou 11, as lâmpadas de sinal acendem dizendo THROW AGAIN (ARREMESSE NOVAMENTE). O jogador ajusta novamente as chaves para indicar o número e comprime de novo um REGISTER (REGISTRADOR) para fazer com que a máquina saiba que a entrada foi atualizada. O jogo continua até que um ganho ou uma perda resulte. Numa perda, uma lâmpada de sinal acende lendo YOU LOSE (VOCÊ PERDE). Desenhar um fluxograma para esta máquina de dados.

7.4-3 A Fig. P7.4-3 mostra um fluxograma incompleto de quatro estados em que as atribuições de estado são como indicados. A entrada X *é assíncrona:* $X = 0$ exceto quando uma chave mecânica é fechada por um operador humano, fazendo $X = 1$. Levar em conta o tempo de *preparação* dos flip-flops e sua variabilidade de um flip-flop para outro. Mostrar que, se



a chave estiver fechada em um tempo inadequado, o sistema pode ir de A para ou C ou D, em vez de para B. Sugerir um corretivo.

7.4-4 Num sistema síncrono admite-se que a borda ativa da forma-de-onda de relógio chegue a cada flip-flop ao mesmo tempo. Uma diferença de tempo, se ocorrer, é chamada *assimetria de relógio* (*clock skew*).

(*a*) Discutir qualitativamente as dificuldades que a assimetria de relógio (*clock skew*) pode causar.

(*b*) Considerar um contador em anel de módulo 2 com dois flip-flops FFA e FFB. Seja uma assimetria (*skew*) τ entre as formas-de-onda de relógio do flip-flop. Para cada um dos estados do contador, descrever o desempenho do circuito quando τ aumenta a partir de $\tau = 0$.

7.5-1 Um sistema síncrono de quatro estados com estados A, B, C e D tem quatro entradas X_0, X_1, X_2 e X_3 e uma saída Z. De A o sistema vai para B. De B ele vai para C se $X_0 = 0$. Enquanto em B, se $X_0 = X_1 = 1$, a saída é $Z = 1$, sendo esta a única condição para a qual uma saída é especificada. Se $X_0 = 1$ e $X_2 = 0$, o circuito vai de B para C. Se $X_0 = X_1 = 1$ e também $X_2 = 0$, a transição de estado é de B para D, mas, se $X_0 = X_1 = 1$ e $X_2 = 1$, o circuito permanece em B. De C vamos para A se $X_0X_3 = 11$, para B se $X_0X_3 = 01$ ou 10 e para D se $X_0X_3 = 00$. Desenhar o fluxograma para o sistema.

7.5-2 Desenvolver um diagrama de estados para um circuito Moore que produz uma saída $Z = 1$ durante um ciclo de relógio, contanto que tenha havido três 1s na linha X de entrada durante *exatamente* os três intervalos de relógio precedentes. Se houver 1s durante quatro ou mais ciclos, a saída é $Z = 0$.

7.5-3 Repetir o Prob. 7.5-2 exceto como um circuito Mealy, isto é, a determinação se houve ou não exatamente três 1s deve ser feita durante o tempo de ocorrência do terceiro 1.

7.5-4 Desenvolver uma tabela de estados para um circuito síncrono Mealy com uma única entrada X e uma única saída Z. O circuito faz $Z = 1$ sempre que *exatamente* um par de bits idênticos for introduzido, contanto que o par fosse precedido por exatamente um par e desde que também os bits no par sucessivo sejam diferentes. Por exemplo,

X	0	0	0	1	1	0	0	1	1	1	0	0	1	1
Z	?	?	0	0	0	0	1	0	1	0	0	0	0	1

7.5-5 (*a*) Desenvolver uma tabela de estados para um circuito síncrono Mealy com duas entradas X_0 e X_1 e uma única saída Z. Z torna-se $Z = 1$ no intervalo quando $X_0 = X_1$ contanto que as entradas tenham sido também iguais no intervalo imediatamente precedente. Uma vez que $Z = 1$, este valor é válido desde que X_0 e X_1 continuem a ser iguais.

(*b*) Repetir a parte (*a*) para um circuito Moore.

7.7-1 Eliminar os estados redundantes, se houver, das seguintes tabelas de estados. Fazer tabelas de estado reduzidas.

Estado presente	Estado seguinte/Z	
	$X=0$	$X=1$
A	$A/0$	$D/0$
B	$E/0$	$C/0$
C	$D/1$	$A/0$
D	$F/1$	$A/1$
E	$E/0$	$D/0$
F	$E/0$	$C/1$

(*a*)

Estado presente	Estado seguinte/Z	
	$X=0$	$X=1$
A	$B/0$	$C/1$
B	$A/1$	$E/0$
C	$F/1$	$C/0$
D	$D/0$	$C/1$
E	$A/1$	$B/0$
F	$B/0$	$D/1$

(*b*)

7.7-2 Eliminar os estados redundantes, se houver, das seguintes tabelas de estados. Construir tabelas de estado reduzidas.

Estado presente	Estado seguinte/Z	
	$X=0$	$X=1$
A	$E/0$	$D/0$
B	$A/1$	$F/0$
C	$C/0$	$A/1$
D	$B/0$	$A/0$
E	$D/1$	$C/0$
F	$C/0$	$D/1$
G	$H/1$	$G/1$
H	$C/1$	$B/1$

(*a*)

Estado presente	Estado seguinte/Z	
	$X=0$	$X=1$
A	$A/0$	$B/0$
B	$H/1$	$C/0$
C	$E/0$	$B/0$
D	$C/1$	$D/0$
E	$C/1$	$E/0$
F	$F/1$	$G/1$
G	$B/0$	$F/0$
H	$H/1$	$C/0$

(*b*)

7.7-3 Eliminar os estados redundantes, se houver, da seguinte tabela de estados. Construir uma tabela de estados reduzida.

Estado presente	Estado seguinte/Z $X_1X_0=$			
	00	01	10	11
A	$D/0$	$D/0$	$F/0$	$A/0$
B	$C/1$	$D/0$	$E/1$	$F/0$
C	$C/1$	$D/0$	$E/1$	$A/0$
D	$D/0$	$B/0$	$A/0$	$F/0$
E	$C/1$	$F/0$	$E/1$	$A/0$
F	$D/0$	$D/0$	$A/0$	$F/0$
G	$G/0$	$G/0$	$A/0$	$A/0$
H	$B/1$	$D/0$	$E/1$	$A/0$

7.7-4 Achar uma tabela de estados mínimos equivalente à tabela de estados dada.

Estado presente	Estado seguinte/Z	
	$X=0$	$X=1$
A	$B/0$	$C/0$
B	$D/0$	$E/0$
C	$F/0$	$G/0$
D	$H/0$	$I/0$
E	$J/0$	$K/0$
F	$D/0$	$L/0$

Estado presente	Estado seguinte/Z	
	$X=0$	$X=1$
G	$J/0$	$L/0$
H	$H/0$	$A/0$
I	$J/1$	$A/0$
J	$D/0$	$A/0$
K	$B/0$	$A/0$
L	$B/0$	$A/0$

7.8-1 Com referência ao diagrama de estados da Fig. 7.8-2, propor definições alternativas que possam ser usadas para o estado inicial A.

7.8-2 Partindo da tabela de estados reduzida da Fig. 7.8-4*f*, achar as equações de excitação dos flip-flops se flip-flops JK devem ser usados para implementar o circuito. Desenhar o diagrama lógico do detector de seqüências. Usar as atribuições de estado $Q_2Q_1Q_0 = 000$, 001, 010, 011 e 101 para os estados A, B, C, D e G, respectivamente.

7.8-3 Um circuito seqüencial de quatro estados é caracterizado pela tabela de estados dada. Usando a atribuição de estados dada, achar as equações de excitação dos flip-flops e desenhar o diagrama lógico para (*a*) os flip-flops tipo D, (*b*) os flip-flops tipo JK e (*c*) os flip-flops *toggle* (T).

Estado presente	Estado seguinte/Z	
	$X=0$	$X=1$
A	$D/1$	$C/0$
B	$A/0$	$C/0$
C	$B/0$	$C/0$
D	$D/0$	$C/0$

Estado	Q_1Q_0
A	00
B	01
C	11
D	10

7.8-4 Para a tabela de estados e a atribuição de estados dada, achar as equações de excitação dos flip-flops e desenhar o circuito lógico.

Estado presente	Estado seguinte/Z	
	$X=0$	$X=1$
A	$E/0$	$A/1$
B	$A/0$	$C/1$
C	$D/1$	$C/0$
D	$B/1$	$E/1$
E	$C/0$	$F/0$
F	$B/0$	$D/0$

Estado	Q_2	Q_1	Q_0
A	0	0	0
B	0	0	1
C	1	1	1
D	1	0	0
E	1	0	1
F	0	1	1

7.9-1 (*a*) Fazer uma atribuição de estados arbitrária para um circuito seqüencial de quatro estados envolvendo dois flip-flops com saídas Q_1Q_0. Determinar o número de maneiras em que a atribuição pode ser modificada pela troca de colunas sob Q_1 e Q_0, ou pela troca de 0s e 1s em uma ou em ambas as colunas, ou fazendo-se ambos os tipos de trocas. Mostrar que há oito de tais atribuições efetivamente equivalentes e que, por conseguinte, há somente três atribuições possíveis essencialmente diferentes. Achar estas três atribuições.

(*b*) Um sistema seqüencial tem S estados e usa F flip-flops com $2^{F-1} < S \leq 2^F$. Mostrar que o número N de atribuições de estado essencialmente diferentes é

$$N = \frac{(2^F-1)\,!}{(2^F-S)\,!\,F\,!}$$

7.9-2 São dadas as três atribuições de estado essencialmente diferentes para um sistema síncrono de quatro estados. Também é dada uma tabela de quatro estados. Para cada uma das possíveis atribuições de estado, achar os diagramas lógicos dos circuitos que implementam o diagrama de estados. Usar os flip-flops tipo D. Uma atribuição de estado fornece um circuito mais econômico do que as outras duas?

	Q_1Q_0		
Estado	1	2	3
A	00	00	00
B	01	11	10
C	11	01	01
D	10	10	11

Estado presente	Estado seguinte/Z	
	$X = 0$	$X = 1$
A	*D*/0	*B*/0
B	*C*/1	*D*/1
C	*A*/0	*B*/0
D	*A*/0	*C*/0

7.9-3 As entradas estão indicadas nas duas linhas de uma tabela de estados na Tab. P7.9-3*a*. Uma vez que há duas entradas X_1 e X_0, há quatro colunas dos estados seguintes. As linhas são especiais quanto a que, para cada estado presente, as entradas dos estados seguintes são as mesmas. As saídas (não indicadas) não são as mesmas, de modo que $A \neq B$. Admitir que haja oito estados na tabela de modo que três flip-flops são requeridos. Os flip-flops tipo D com entradas D_2, D_1 e D_0 devem ser usados.

(*a*) Usando a atribuição de estado dada na Tab. 7.9-3*b* para os quatro estados *A*, *B*, *C* e *D*, a partir da informação dada, preencher, tanto quanto possível, os mapas K dos quais possamos ler as excitações requeridas para D_2, D_1 e D_0.

(*b*) Repetir a parte (*a*) mas usar a atribuição de estados dada na Tab. P7.9-3*c*. Nesta segunda atribuição arranjamos para que as atribuições dadas ao par de estados *A* e *B* e dadas ao par de estados *C* e *D* sejam diferentes em apenas um *único* bit. Tais atribuições são denominadas *logicamente adjacentes*. Na parte (*a*), as atribuições não são logicamente adjacentes.

(*c*) Dos resultados da parte (*a*) e (*b*) formular uma regra concernente às atribuições de estado a serem feitas quando dois estados presentes tiverem as mesmas entradas de estado seguinte em cada coluna. Se algumas, mas não todas, entradas de estado seguinte forem as mesmas, a regra ainda é útil?

(*d*) Na tabela de estados do P7.9-3*a* admitir que na coluna $X_1X_0 = 01$ e $X_1X_0 = 10$ os estados *C* e *D* foram trocados. Mostrar que mesmo neste caso a atribuição do P7.9-3*c* é melhor do que a atribuição do P7.9-3*b*.

Tabela P7.9-3

Estado presente	Estado seguinte $X_1X_0 =$ 00	01	11	10
•	•	•	•	•
A	*C*	*C*	*C*	*C*
•	•	•	•	•
B	*D*	*D*	*D*	*D*
•	•	•	•	•

(*a*)

Estado	Q_2	Q_1	Q_0
A	0	0	0
B	1	0	1
C	0	1	1
D	1	0	0

(*b*)

Estado	Q_2	Q_1	Q_0
A	0	0	0
B	0	0	1
C	0	1	1
D	1	1	1

(*c*)

7.9-4 As entradas estão indicadas em duas linhas de uma tabela de estados de oito estados na Tab. P7.9-4*a*. Há duas entradas X_1 e X_0 e, portanto, há quatro colunas de estados seguintes. Devem ser usados flip-flops tipo D. São dadas duas atribuições de estado possíveis na Tab. P7.9-4*b* e *c*.

Tabela P7.9-4

Estado presente	Estado seguinte $X_1X_0 =$ 00	01	11	10
•	•	•	•	•
A	*C*	*D*	*E*	*F*
•	•	•	•	•
B	*C*	*D*	*E*	*F*
•	•	•	•	•

(*a*)

Estado	Q_2	Q_1	Q_0
A	0	0	0
B	0	0	1
C	0	1	0
D	0	1	1
E	1	1	1
F	1	1	0

(*b*)

Estado	Q_2	Q_1	Q_0
A	0	0	0
B	1	0	1
C	1	0	0
D	1	1	1
E	0	0	1
F	0	1	0

(*c*)

(*a*) Na Tab. P7.9-4*b* as atribuições são *logicamente adjacentes* (ver Prob. 7.9-3) para os pares *A* e *B*, *C* e *D*, *D* e *E*, *E* e *F* e *C* e *F*. Usando esta atribuição, a partir da informação limitada dada, preencher, tanto quanto possível, os mapas K dos quais possamos ler as excitações requeridas para as entradas dos flip-flops D_2, D_1 e D_0.

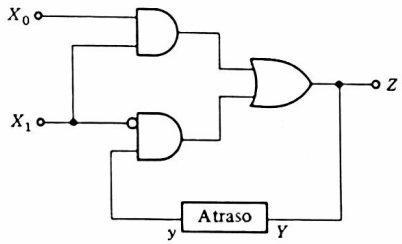
(*b*) Repetir a parte (*a*) para a atribuição da Tab. P7.9-4*c*. Neste caso os pares citados na parte (*a*) não têm atribuições adjacentes.

(*c*) Da comparação dos resultados das partes (*a*) e (*b*) formular uma regra concernente às atribuições a serem feitas aos estados seguintes numa linha de uma tabela de estados.

7.12-1 Construir uma tabela de transição para o circuito do modo fundamental da Fig. P7.12-1.

(*a*) Mostrar que o circuito tem dois estados estáveis para $X_1X_0 = 00$ e também para $X_1X_0 = 01$.

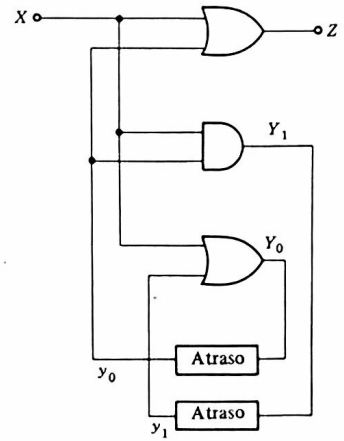
(*b*) Se o circuito estiver no estado estável $Z = 0$ com $X_1X_0 = 00$, o que deve ser feito para colocá-lo no estado $Z = 1$ com $X_1X_0 = 00$?



7.12-2 Construir uma tabela de transição para o circuito do modo fundamental da Fig. P7.12-2.

(*a*) Mostrar que o circuito tem um estado estável para $X = 0$ e um estado estável para $X = 1$.

(*b*) Admitir, conforme indicado na figura, que os atrasos são externos às portas e estão localizados nas ligações de realimentação. Os atrasos são iguais em um valor Δ. O circuito está no estado estável com $X = 0$ e, no tempo $t = 0$, X faz uma transição para $X = 1$. Desenhar as formas-de-onda de y_1, y_0, Y_1, Y_0 e de Z mostrando a seqüência de mudanças que levam o circuito para o outro estado estável.



7.13-1 Um circuito do modo fundamental tem duas entradas X_1 e X_0 e uma saída Z. Quando $X_1X_0 = 00$, o circuito tem dois estados estáveis, um com $Z = 0$ e outro com $Z = 1$. Se o circuito estiver no estado com $Z = 0$, uma mudança para $X_1X_0 = 01$ e um retorno a $X_1X_0 = 00$ causarão uma mudança de estado. Similarmente, uma mudança para $X_1X_0 = 10$ e um retorno para $X_1X_0 = 00$ causarão uma mudança inversa no estado. Na base desta descrição verbal do desempenho requerido do circuito, fazer uma tabela de transições. Fazer uma atribuição de estados arbitrária e desenhar o circuito. Verificar que o circuito é um latch de porta NOR.

7.15-1 Achar uma atribuição de estado isenta de corridas críticas para a tabela de fluxos da Fig. P7.15-1.

PS \ X_1X_0	00	01	11	10
a	*d*/0	*b*/0	*d*/1	(*a*)/1
b	*d*/1	(*b*)/1	(*b*)/0	*a*/0
c	*d*/0	(*c*)/0	*b*/0	(*c*)/0
d	(*d*)/0	*c*/0	(*d*)/1	*c*/1

NS, *Z*

7.15-2 Consultar a tabela de estados da Fig. P7.15-2. Mostrar que, se duas variáveis de estado forem usadas, não há nenhuma atribuição de estados que evite completamente as corridas críticas.

PS \ X_1X_0	00	01	11	10
a	(*a*)	*b*	*d*	(*a*)
b	*a*	(*b*)	(*b*)	*d*
c	(*c*)	(*c*)	*b*	*a*
d	*c*	*b*	(*d*)	(*d*)

NS

7.15-3 Achar uma atribuição de estados para a tabela de estados da Fig. P7.15-2 que seja inteiramente isenta ou livre de corridas críticas.

7.17-1 Na tabela de estados dada, pareceria eminentemente razoável atribuir valores às saídas irrelevantes para fazer $A = B$ e $E = F$.

Estado presente	Estado seguinte/Z	
	$X = 0$	$X = 1$
A	$B/-$	$E/0$
B	$B/1$	$E/-$
C	$F/0$	$C/0$
D	$B/1$	$A/1$
E	$C/0$	$C/-$
F	$C/-$	$C/1$

(*a*) Mostrar que uma tal atribuição conduz a uma tabela de estados reduzida com quatro estados.

(*b*) Fazer a seguir uma atribuição cujo mérito não seja óbvio; isto é, nos estados A e E substituir as saídas irrelevantes por 0 e em B e F substituir as saídas irrelevantes por 1. Mostrar que agora a tabela de estados reduzida tem apenas dois estados.

7.17-2 Eliminar os estados redundantes, se houver, e construir tabelas de estados reduzidas para as seguintes tabelas que têm entradas irrelevantes.

Estado presente	Estado seguinte/Z	
	$X = 0$	$X = 1$
A	$-/0$	$B/0$
B	$A/1$	$D/-$
C	$D/-$	$E/1$
D	$-/0$	$B/-$
E	$C/0$	$E/1$

Estado presente	Estado seguinte/Z X_1X_0			
	00	01	11	10
A	$A/0$	$E/1$	$A/0$	$C/-$
B	$B/0$	$C/0$	$C/-$	$C/-$
C	$C/0$	$B/-$	$A/1$	$C/-$
D	$D/0$	$E/-$	$E/0$	$A/-$
E	$E/0$	$B/-$	$A/-$	$D/-$

7.18-1 (*a*) Achar os máximos compatíveis para a seguinte tabela de estados.

Estado presente	Estado seguinte/Z	
	$X = 0$	$X = 1$
A	$C/0$	$D/-$
B	$C/-$	$C/-$
C	$B/0$	$D/-$
D	$A/0$	$E/-$
E	$B/-$	$E/-$

(*b*) Mostrar que, se os máximos compatíveis forem usados como estados, a tabela de estados reduzida tem três estados.

(*c*) Mostrar que usando outros que não sejam os máximos compatíveis em cada caso é possível reduzir a tabela de estados a dois estados.

7.18-2 A tabela de fluxo reduzida na Fig. P7.18-2 descreve um sistema assíncrono. Fazer uma atribuição de estado que evite as corridas críticas e desenhar o circuito lógico.

PS \ X_1X_0	00	01	11	10
a	(a)/0	$b/-$	(a)/1	$b/-$
b	$a/-$	(b)/0	$c/-$	(b)/0
c	$a/-$	(c)/1	(c)/0	$b/0$

NS, Z

7.18-3 A tabela de fluxos primitiva na Fig. P7.18-3 descreve um sistema assíncrono. Construir uma tabela de estados reduzida e desenhar o circuito lógico.

PS \ X_1X_0	00	01	11	10
a	$d/-$	$-/-$	$e/-$	(a)/1
b	$d/-$	(b)/1	$e/-$	$-/-$
c	$d/-$	$-/-$	$e/-$	(c)/0
d	(d)/0	$b/1$	$e/-$	$a/-$
e	$-/-$	$b/-$	(e)/0	$c/0$

NS, Z

7.18-4 Um circuito do modo fundamental tem entradas X_1 e X_0 e uma única saída Z. Quando $X_1X_0 = 00$, $Z = 0$. Para fazer $Z = 1$ começamos com $X_1X_0 = 0$ e primeiro mudamos X_0 para 1 e a seguir mudamos X_1 para 1. Para retornar Z para $Z = 0$, devemos retornar para $X_1X_0 = 00$, sendo a ordem de retorno sem conseqüências.

(*a*) Construir uma tabela de fluxos primitiva.

(*b*) Eliminar os estados redundantes, se houver.

(*c*) Fazer uma atribuição de estado que evite corridas críticas.

(*d*) Desenhar o circuito lógico.

Capítulo 8

8.1-1 Suponhamos que achamos aceitável realizar uma transferência de registrador a registrador em duas etapas; isto é, primeiro limpamos B e depois carregamos B com o conteúdo de A. Mostrar que neste caso podemos usar menos hardware do que na Fig. 8.1-1. Desenhar o diagrama lógico e as formas-de-onda para uma tal transferência síncrona.

8.2-1 (*a*) Um registrador com estágios individuais A_i deve ser acoplado a um bus cujas linhas sustentam os bits B_i. Os componentes do registrador A são latches estáticos *SR*. Desenhar o diagrama lógico de um circuito associado com um estágio do registrador que nos permita comandar a transferência para dentro do estágio registrador do bit que resulta quando A_i é submetido a uma operação AND com B_i; isto é, $A_iB_i \rightarrow A_i$. Repetir para (*b*) $A_i + B_i \rightarrow A_i$, (*c*) $A_i \oplus B_i \rightarrow A_i$ e (*d*) $A_i \oplus \bar{B}_i \rightarrow A_i$.

8.2-2 Repetir o Prob. 8.2-1 para flip-flops *JK* e operação síncrona.

8.2-3 Um registrador A consiste em flip-flops do tipo *D*. Para cada um dos seguintes casos, desenhar individualmente a estrutura lógica associada com um estágio A_i do registrador que nos permitirá comandar a modificação do conteúdo do registrador na maneira indicada: (*a*) $0 \rightarrow A_i$, (*b*) $1 \rightarrow A_i$, (*c*) $\bar{A}_i \rightarrow A_i$.

8.3-1 Desenhar o diagrama lógico de um registrador de três bits usando flip-flops *JK* que têm lógica associada que fornece dois terminais de controle chamados ADD e INCREMENT. Quando ativado, INCREMENT faz com que o registro seja aumentado de 1, e ADD soma o conteúdo do registrador ao número sobre um bus e deixa a soma no registrador.

8.3-2 Desenhar o diagrama lógico de um registrador de três bits que usa flip-flops tipo *D* com três terminais de controle chamados ROTATE LEFT, ROTATE RIGHT e COMPLEMENT. ROTATE RIGHT move cada bit de uma posição para a direita, o bit mais à direita transferindo para o flip-flop mais à esquerda. ROTATE LEFT causa o deslocamento inverso. COMPLEMENT substitui cada bit por seu complemento.

8.3-3 Desenhar o diagrama lógico de um registrador de três bits usando flip-flops *JK* que têm lógica associada fornecendo terminais de controle DECREMENT, COMPLEMENT, SET e RESET. DECREMENT reduz o registro de 1, COMPLEMENT substitui cada bit por seu complemento e SET e RESET, respectivamente, ajustam e reajustam cada bit.

8.3-4 Desenhar o diagrama lógico de um registrador de três bits usando flip-flops *JK* que têm lógica associada fornecendo terminais de controle ADD e SUBTRACT. Quando ativado, ADD soma o número num bus ao número no registrador. SUBTRACT subtrai o número no bus do número no registrador. Em ambos os casos o resultado é deixado no registrador.

8.4-1 Construir um fluxograma de um controlador para orientar a operação do sistema da Fig. 8.4-1. O controlador deve ter uma chave de partida representada pela variável X e deve ter uma segunda entrada Y condicional. Quando $Y = 1$, a operação a ser efetuada é o cálculo de $\alpha - \beta$, e quando $Y = 0$, ela é $\beta - \alpha$. O resultado deve ser armazenado no registrador β.

8.4-2 Necessitamos de um circuito que em resposta ao fechamento de uma chave mecânica gere um pulso positivo. O pulso é síncrono com uma chave e tem a duração de um único ciclo de relógio. A chave deve ser aberta e fechada novamente para gerar um segundo pulso.
(*a*) Construir um fluxograma de uma máquina Moore que irá gerar o pulso. Lembre-se de que a chave pode fechar em um tempo arbitrário no ciclo de relógio.
(*b*) Completar o projeto e desenhar o circuito lógico usando flip-flops tipo *D*. Tentar todas as atribuições de estado possíveis para ver se há uma melhor.
(*c*) Se o sinal da chave for aplicado a um flip-flop tipo *D*, a saída do flip-flop fornecerá uma transição síncrona, sempre atrasada, em resposta à operação da chave. Admitindo que se disponha de uma transição síncrona, construir um fluxograma de uma máquina Mealy que realizará a mesma função que o circuito Moore. Projetar e desenhar o circuito lógico.

8.4-3 Uma máquina de adicionar-subtrair deve ser construída de acordo com as seguintes especificações. Os números, em representação binária, devem ser introduzidos ajustando-se um conjunto de chaves de duas posições. O número introduzido deve ser adicionado ou subtraído do conteúdo de um registrador acumulador conforme seja comprimido um botão ADD ou um botão SUBTRACT. A representação em complemento de dois deve ser usada no registrador do acumulador.
(*a*) Estabelecer uma arquitetura para o sistema.
(*b*) Construir um fluxograma.
(*c*) Projetar o controlador do sistema usando um número mínimo de estados e desenhar o circuito lógico.

8.4-4 Um número decimal de dois dígitos é armazenado em dois registradores de quatro bits em forma BCD. O registrador M retém o dígito mais significativo e o registrador L retém o dígito menos significativo. Os números devem ser transferidos de modo que eles apareçam lado a lado num registrador R de oito bits. Disponível para efetuar a transferência está um bus de quatro bits acessível a M e a L mas somente para as quatro posições mais à direita do registrador R. A operação de transferência deve ocorrer em resposta à operação de uma chave.
(*a*) Estabelecer uma arquitetura para o sistema especificando os terminais de controle em cada registrador.
(*b*) Construir um fluxograma.
(*c*) Projetar o controlador do sistema usando um mínimo de estados e desenhar o circuito lógico.

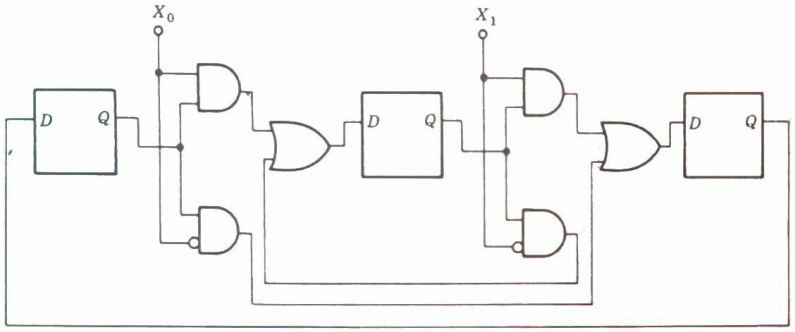
8.4-5 Uma fechadura de combinação eletrônica tem um botão X_0 de compressão para reset (reajuste) e três botões de operação, X_1, X_2 e X_3. A fechadura abrirá somente se os botões de operação forem comprimidos na ordem correta. Se a ordem estiver errada, o botão de reajuste deve ser comprimido para possibilitar uma nova tentativa. Construir um fluxograma, projetar o circuito e desenhar seu diagrama lógico.

8.5-1 Projetar um conversor de código de ROM para substituir as três caixas lógicas na Fig. 8.5-1 que geram as excitações para os flip-flops.

8.5-2 Verificar que o decodificador da Fig. 8.5-2 gera sinais de saída de controle nos instantes de tempo em que eles são necessários.

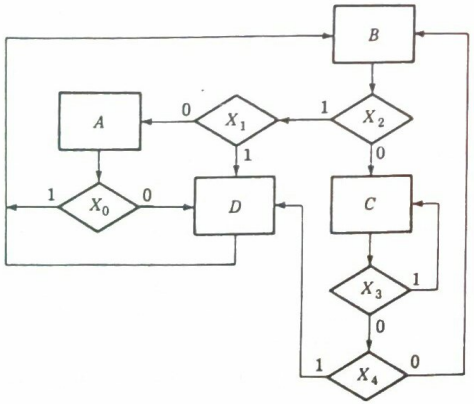
8.5-3 Deduzir as equações de excitação requeridas no projeto da Fig. 8.5-1 se os flip-flops *JK* forem substituídos pelos do tipo *D*.

8.6-1 Construir um fluxograma e uma tabela de estados para o controlador cujo diagrama lógico é dado na Fig. P8.6-1.



8.6-2 Suponhamos que um fluxograma solicite um estado de "espera" (*wait*), isto é, o controlador deve parar e não implementar nenhuma outra microoperação até que receba um sinal READY de alguma fonte. Mostrar como um tal estado de espera pode ser incorporado a um controlador registrador de deslocamentos. Mostrar a lógica para ambos os tipos de flip-flop, *JK* e *D*.

8.7-1 Um fluxograma de um sistema seqüencial é mostrado na Fig. P8.7-1. Há quatro estados *A*, *B*, *C* e *D* e cinco variáveis condicionais X_0 a X_4 que determinam a ordem de colocação em seqüência. Projetar um controlador registrador de deslocamentos para o sistema.



8.7-2 Projetar um controlador registrador de deslocamentos que incrementa a tabela de estados dada.

	NS/Z	
PS	$X = 0$	$X = 1$
A	*D*/1	*B*/0
B	*D*/0	*B*/0
C	*E*/0	*A*/0
D	*C*/0	*B*/0
E	*E*/0	*B*/0

NS = Estado seguinte
PS = Estado presente

8.8-1 (*a*) Projetar um controlador registrador de deslocamentos a ser usado com a arquitetura da Fig. 8.4-1 que forma a quantidade $R_\alpha - 2R_\beta$ e deixa o resultado no acumulador.
(*b*) Repetir para o resultado $2(R_\alpha - R_\beta)$.

8.8-2 Projetar um controlador registrador de deslocamentos a ser usado com a arquitetura da Fig. 8.4-1. O sistema deve formar a soma dos valores absolutos dos conteúdos de R_α e R_β e deve deixar o resultado no acumulador. (Note que o controlador terá que ter os bits de sinal dos números nos registradores como entradas de controle.)

8.8-3 Projetar um controlador registrador de deslocamentos a ser usado com a arquitetura da Fig. 8.4-1. Se o conteúdo de cada (ou de ambos) R_α e R_β for negativo, o controlador limpa o acumulador mas não faz nada além disto. Se ambos forem positivos, o sistema determina qual dos conteúdos é maior e deixa este número maior no acumulador. (Ver a nota no Prob. 8.8-2.)

8.9-1 (*a*) Na arquitetura da Fig. 8.9-1 não providenciamos um terminal de controle para limpar o acumulador. Mostrar que esta operação de limpeza pode ser efetuada usando apenas as instruções listadas na Fig. 8.9-2*b*.
(*b*) Normalmente não há necessidade de limpar o registrador CI. Mostrar, no entanto, que, se quisermos, podemos limpá-lo usando novamente apenas as instruções listadas.

8.9-2 Os números $N_1 = 106$, $N_2 = 16$, $N_3 = -85$, $N_4 = 45$ e $N_5 = 36$ estão armazenados nos locais da memória 50 a 54 da memória da Fig. 8.9-1. Escrever um programa na maneira da Fig. 8.9-2*c* que forma $3(N_1 - 2N_2 + N_3 + N_4 - N_5)$ e armazena o resultado no local 49 da memória.

8.11-1 Suponhamos que modificamos a estrutura da Fig. 8.9-1 como se segue: (1) reservamos três bits para instruções, de modo que temos mais do que quatro instruções, e (2) equipamos o acumulador com um terminal de saída X_0, que indica quando o registro do acumulador é zero. Esta saída do acumulador é uma das variáveis de entrada para o controlador.
(*a*) Modificar o controlador de modo que ele responderá à instrução "parar se o acumulador estiver limpo".
(*b*) Modificar o controlador de modo que ele responderá à instrução "buscar a próxima instrução não do local da memória no contador do programa mas de um local da memória cujo endereço está especificado nesta instrução".

8.13-1 (*a*) Construir tabelas de estados para os controladores cujos fluxogramas são dados na Fig. 8.13-3.
(*b*) Usando flip-flops *JK*, reprojetar os controladores de confirmação de conexão (*hand-shake*) de modo que eles usem um número mínimo de flip-flops.

Capítulo 9

9.1-1 Uma RAM tem terminais de controle *HABILITA* (*ENABLE*) (E) e *leitura-escrita* (*read-write*) (R/$\overline{W}$). Preferimos operar a memória ativando ou um terminal *leitura* (*read*) (R) ou um terminal *escrita* (*write*) (W). Projetar a lógica requerida a ser interposta entre os terminais R e W e os terminais da memória.

9.1-2 Projetar um estágio típico do registrador do acumulador da Fig. 9.1-1.

9.1-3 Projetar estágios típicos do registrador GPR e do registrador PC. Lembrar que estes registradores devem ser capazes de trocar conteúdos durante um ciclo de relógio.

9.1-4 Supor, na arquitetura da Fig. 9.1-1, que usamos uma memória que tem uma capacidade de 65 536 palavras, cada uma com 22 bits de comprimento.
(*a*) Qual é o número de instruções que podem ser incorporadas ao computador?
(*b*) Quantos bits nos registradores PC, MAR, OPR e GPR?

9.2-1 No local 17 da memória há a instrução "somar ao acumulador o operando cujo endereço deve ser achado no local 31". Este operando tem o valor -27, e o número já no acumulador é $+42$. Admitir que o código da instrução é 10. (Todos os números precedentes são decimais.) Começando no instante da conclusão da instrução anterior, especificar, tanto quanto possível, o conteúdo, em notação binária, de cada registrador na Fig. 9.1-1 após cada ciclo de relógio até que a execução da instrução esteja completa.

9.2-2 Usando a arquitetura e as microoperações da Fig. 9.1-1, escrever a seqüência de microoperações que executariam a instrução:
(*a*) que chamava para substituição o número N positivo em algum local pelo número $4N$.
(*b*) que chamava para substituição o número N por $-N$.

9.2-3 Um deslocamento *aritmético* de um conteúdo do registrador é um deslocamento à esquerda para multiplicar por 2 ou um deslocamento à direita para dividir por 2. Quando o conteúdo for um número positivo, o dígito 0 deve ser deslocado para a posição extrema vaga mais à esquerda ou mais à direita.
(*a*) Admitir que um registrador retém um número negativo em representação em complemento de um. Mostrar que, neste caso, num deslocamento aritmético, 1s devem ser deslocados para as posições extremas vagas por deslocamentos em qualquer direção.
(*b*) Admitir que um registrador retém um número negativo em representação de complemento de dois. Mostrar que, neste caso, os 0s devem ser deslocados para a posição extrema vaga mais à direita por um deslocamento à esquerda e os 1s devem ser deslocados para a posição extrema vaga mais à esquerda por um deslocamento à direita.
(*c*) Usando a arquitetura e as microinstruções da Fig. 9.1-1, escrever a seqüência de microoperações que executará a instrução solicitando a substituição do conteúdo N do acumulador pelo número $4N$. Sabe-se que N é negativo e é dado em representação de complemento de dois.
(*d*) Repetir a parte (*c*) se o conteúdo N deve ser substituído por $N/4$.

9.5-1 Escrever um programa de linguagem de montador que substituirá um número N armazenado na memória pelo número $4N$. Não se sabe se o número é positivo ou negativo. Se negativo, ele é dado em representação de complemento de dois. (Consultar o Prob. 9.2-3.)

9.5-2 Escrever um programa em linguagem de montador que deslocará o conteúdo do local 30 (hex) da memória de uma posição à esquerda. Um 1 deve ser movido para dentro da posição vaga mais à direita e o resultado deve ser colocado no local 31.

9.5-4 Escrever um programa em linguagem de montador que ajustará em 0 os quatro bits mais significativos do conteúdo do local 30 (hex) da memória e mover o resultado para o local 31.

9.5-5 Escrever um programa de linguagem de montador que divide o conteúdo do local 30 (hex) da memória em duas seções de seis bits e as armazena nos locais 31 e 32 da memória. Os seis bits mais significativos do local 30 devem ir para as posições de bit menos significativos do local 32. As posições dos bits mais significativos dos locais 31 e 32 devem estar limpas.

9.5-6 Os locais 30 e 31 (hex) da memória retêm dois números positivos. Escrever um programa de linguagem de montador que escreverá o maior dos dois números no local 32 da memória.

9.5-7 Os locais 50 a 70 (hex) da memória retêm entradas, algumas das quais são números positivos e outras números negativos. Escrever um programa em linguagem de montador que escreva no local 71 da memória o número que exprime quantas destas entradas são negativas.

9.7-1 Os locais 50 a 53 (hex) da memória retêm os números positivos N_1, N_2, N_3 e N_4. Escrever um programa em linguagem de montador que forma o produto $N_1N_2N_3N_4$ e deixa o resultado no local 54.

9.7-2 Os locais 50 a 70 (hex) da memória retêm números positivos. Usando o programa do Prob. 9.5-6, escrever um programa empregando uma sub-rotina que escreva o maior destes números no local 71 da memória.

9.8-1 Atribuir às saídas da CROM da Fig. 9.8-1 as operações especificadas pelas microoperações numeradas correspondentemente na Tab. 9.1-1. Com esta atribuição, escrever o conteúdo da CROM (como na Fig. 9.8-2) que execute as instruções (*a*) JMPI, (*b*) CSR, (*c*) ISZ.

9.8-2 A máquina da Fig. 9.1-1 não tem uma instrução de SUBTRACT. Escrever o conteúdo da CROM da Fig. 9.8-2 que forneça uma instrução que subtrai do acumulador o conteúdo de um local da memória e deixa o resultado no acumulador. Usar a atribuição indicada no Prob. 9.8-1.

9.8-3 A CROM da Fig. 9.8-1 tem dezoito saídas correspondentes às dezoito microoperações

listadas na Tab. 9.1-1. Recomendar um arranjo alternativo que permita o uso de uma CROM com menor número de bits por palavra (embora às expensas de lógica adicional). Qual é o mínimo aceitável de números de bits por palavra?

9.9-1 Projetar o registrador de endereços da memória de controle da Fig. 9.9-1 que pode ser incrementado ou carregado. Admitir que $M = 4$.

9.10-1 Desenhar a estrutura de portas da caixa lógica da Fig. 9.10-1.

9.12-1 Desenhar a estrutura de portas da caixa lógica da Fig. 9.12-1.

9.15-1 Uma seção de uma RAM deve ser usada como uma pilha. Projetar o hardware a ser associado com a RAM de modo que, quando um terminal de comando for feito ativo, um registrador apontador de pilhas será incrementado e depois uma palavra num bus será escrita na memória. Deve haver também um segundo terminal de comando que, quando ativado, lerá de volta a palavra no bus e depois decrementará o apontador de pilhas.

Capítulo 10

10.12-1 (*a*) Dentre as instruções do 8080 achar duas instruções de um byte cada uma das quais pode ser usada individualmente para limpar o acumulador.
(*b*) Achar uma instrução de dois bytes que limpará o acumulador.

10.12-2 Escrever um programa que complementa cada bit da palavra no local 0040 da memória e coloca o resultado no local 0041.

10.12-3 Escrever um programa para somar os conteúdos dos locais 0040 e 0041 da memória e colocar a soma no local 0042.

10.12-4 Escrever um programa para deslocar o conteúdo do local 0040 da memória de um bit para a esquerda, sendo o resultado retornado ao local 0041. A posição de bit mais à direita deve ser deixada limpa.

10.12-5 Escrever um programa que coloque os quatro bits menos significativos do conteúdo do local 0040 no local 0041 da memória. As quatro posições de bits mais significativos devem ser deixadas limpas.

10.12-6 Escrever um programa para limpar os locais 0040 e 0041 da memória.

10.12-7 Escrever um programa que divide o conteúdo dos locais 0040 da memória em duas porções de quatro bits e os armazena nos locais 0041 e 0042 da memória. Os quatro bits mais significativos do local 0040 devem ir para as quatro posições de bits menos significativos do local 0041. Os quatro bits menos significativos do local 0040 devem ir para as quatro posições de bits menos significativos do local 0042. As quatro posições de bits mais significativos dos locais 0041 e 0042 devem estar limpas.

10.12-8 Escrever um programa que carrega o mesmo número N nos registradores B, C, D e E. Escrevê-lo de maneira que requeira o menor número de bytes.

10.12-9 Escrever um programa que subtraia o conteúdo do registrador E do conteúdo do registrador B e reserve o resultado no registrador D.

10.12-10 Escrever um programa que subtraia o conteúdo do *par* de registrador B do conteúdo do *par* de registrador D e reserve o resultado no *par* de registrador D.

10.12-11 Escrever um programa que transfira o conteúdo dos locais 0041, 0042, 0043 e 0044 para os locais 0051, 0052, 0053 e 0054, respectivamente, e deixe os locais originais limpos.

10.12-12 O local 0040 da memória retém o número N_0, e o local 0041 retém o número N_1. Escrever um programa que transferirá para o local 0042 (*a*) o maior dos dois números e (*b*) o menor dos dois números.

10.12-13 Os locais 0060 a 0067 contêm uma tabela de quadrados dos números 0 a 7, respectivamente, isto é, o local $006x$ retém o número x^2 para x de 0 a 7. O local 0040 da memória retém um número x entre 0 e 7. Escrever um programa que usa a tabela de quadrados para colocar x^2 no local 0041 da memória.

10.12-14 Escrever um programa que substitui o número no local 0040 da memória por seu negativo. Os números negativos são expressos em representação de complemento de dois.

10.12-15 Os locais da memória 0040 e 0041 retêm um número de dezesseis bits, estando os bits mais significativos no local 0041. Escrever um programa que colocará o complemento de dois do número nos locais 0042 e 0043, sendo os bits mais significativos colocados no local 0043.

10.14-1 Há uma série de números armazenados em locais sucessivos da memória começando no local 0042. O comprimento da série é dado pelo número registrado no local 0041 da memória. Escrever um programa que somará a série de números e armazenará a soma no local 0040. Admitir que, conforme a soma é formada, não são gerados transportes e que a soma pode estar contida em oito bits.

10.14-2 Repetir o Prob. 10.14-1 admitindo desta vez que um número de dezesseis bits é requerido para reter a soma.

10.14-3 Há uma série de números armazenados em locais sucessivos da memória, começando no local 0042. O comprimento da série é dado pelo número registrado no local 0041 da memória. Escrever um programa que determine quantos dos números na série são negativos e armazene este número no local 0040 da memória.

10.14-4 Há uma série de números armazenados em sucessivos locais da memória que começam no local 0042 da memória. O comprimento da série é dado pelo número no local 0041. Escrever um programa que determine qual número da série é o maior e armazene este número no local 0040. Admitir que os números são todos números de oito bits sem sinais.

10.14-5 Escrever um programa que desloque para a esquerda o número no local 0040 da memória

tantas vezes quantas forem necessárias para mover o 1 mais significativo do número para dentro da posição de bit mais significativo. Armazenar o resultado no local 0041 da memória e armazenar no local 0042 o número de deslocamentos requeridos. Se o número no local 0040 for zero, os locais 0041 e 0042 devem estar limpos. (Se concordarmos em que a vírgula binária do número deve estar localizada depois do bit mais à esquerda, o processo aqui muda o número para notação científica, isto é, 0,0010110 torna-se $1{,}0110000 \times 2^{-3}$.)

10.14-6 Uma série de caracteres ASCII (Apêndice A) é armazenada em locais sucessivos da memória começando no local 0041. O fim da série é assinalado pelo caractere de retorno do carro CR (hex 00). Escrever um programa que determine o número de caracteres na série (excluindo o caractere CR) e escreva o número no local 0040.

10.14-7 Uma série de caracteres ASCII (Apêndice A) é armazenada em locais sucessivos da memória começando no local 0041. O comprimento da série é dado pelo número armazenado no local 0040. Os caracteres ASCII têm sete bits significativos, sendo o bit mais à esquerda, em cada caso, 0. Escrever um programa em que o bit mais à esquerda é mudado para 1 conforme requerido em cada caso para dar a *paridade par* aos caracteres. Os caracteres modificados devem permanecer em seus locais originais.

10.14-8 Uma série de caracteres ASCII (Apêndice A) é armazenada em locais sucessivos da memória começando no local 0042. O comprimento da série é dado no local 0041. Uma segunda série de comprimento idêntico começa no local 0062. Escrever um programa que limpe o local 0040 se as duas séries forem as mesmas e ajuste todas as posições de bits se as séries não se adaptarem.

10.14-9 Escrever um programa para somar dois números de múltiplos bytes. O número de bytes envolvido está no local 0030 da memória. Os próprios números começam nos locais da memória 0041 e 0061, respectivamente. Os bytes menos significativos são dados primeiro. A soma deve substituir o número que começa no local 0041.

10.14-10 Há uma lista de entradas nos locais sucessivos da memória que começam no local 0041 e que se estendem para o local $0041 + K$. O número K é armazenado no local 0040. Há também uma entrada no local 0030. Escrever um programa que verifique determinar se a entrada no local 0030 é uma das entradas na lista. Se esta entrada estiver presente, o programa não faz nada. Se a entrada não estiver presente, o programa acrescenta-a à lista no local $0040 + K + 1$ e muda o número no local 0040 para $K + 1$.

10.15-1 Um registrador de oito bits tem seus quatro bits mais significativos, todos 0s, e os bits restantes são usados para representar os dígitos hexadecimais 0 a F. Verificar, do Apêndice A, que para converter o registro em um caractere ASCII adicionamos 30 (hex) se o dígito for 9 ou menos e adicionamos 37 (hex) se o dígito for maior do que 10.

10.20-1 Verificar que um erro de estouro ocorreu se e somente se V dado pela Eq. (10.20-1) for $V = 1$.

Capítulo 11

11.1-1 Consultar as Figs. 10.2-1 e 11.1-2. Mostrar que o sinal $\overline{DBIN}$ pode servir como o sinal $\overline{MEMR}$ e que $\overline{WR}$ pode servir como $\overline{MEMW}$.

11.3-1 (*a*) Externamente ao microprocessador dispõe-se de um arranjo de chave e de memória intermediária (*buffer*) como na Fig. 11.3-2. Todas as chaves estão permanentemente abertas, exceto possivelmente uma, digamos D_3. Desejamos limpar o local 0040 (hex) da memória se esta chave estiver fechada. Se a chave estiver aberta, o local da memória deve ser ajustado em 0001 (hex). Escrever um programa apropriado.

(*b*) Considerar os casos especiais onde a chave está nas linhas D_0, D_6 ou D_7. Mostrar como as instruções RAL, RAR, ORA A ou ADD A podem ser usadas para fazer um ou outro flag fornecer a informação requerida a respeito da chave estar aberta ou fechada.

11.3-2 Externamente ao microprocessador dispõe-se de um arranjo de chave e de memória intermediária como na Fig. 11.3-2. Todas as chaves estão permanentemente abertas, exceto possivelmente uma. Escrever um programa que limpe o local 0040 (hex) da memória e depois registre neste local o número de vezes que esta chave fechou. Quando a chave for examinada e achada fechada, a condição de fechada não deve ser acrescentada à contagem, a menos que tenha havido um intervalo aberto entre os fechamentos. Admitir que a chave não tenha trepidação (*bounce*).

11.3-3 Em um ponto na execução de um programa, o microcomputador deve ser instruído para examinar a situação de uma das chaves na Fig. 11.3-2 e parar a execução de outras instruções a menos que ache a chave fechada. Escrever as etapas do programa requeridas para efetuar uma tal espera.

11.3-4 Em um ponto na execução de um programa, o microcomputador deve ser instruído para examinar uma das chaves na Fig. 11.3-2 e não prosseguir com outra execução de instruções até que ele veja a *mudança* da chave de aberta para fechada ou vice-versa. Escrever as etapas do programa requerido para efetuar tal espera.

11.3-5 No arranjo da Fig. 11.3-2, admitir que somente uma chave esteja fechada em qualquer tempo particular. Na mudança das chaves abrimos a chave inicialmente fechada antes do fechamento de uma nova chave, de modo que haja intervalos quando nenhuma chave estiver fechada. Escrever um programa que determine se uma chave está fechada e armazene o número desta chave (0 a 7) no registrador B do microprocessador.

11.3-6 As chaves da Fig. 11.3-2 são operadas como descrito no Prob. 11.3-5. Escrever um programa que tome nota de qual chave está fechada, espere até que uma nova chave esteja fechada e registre o número da nova chave no local 0040 (hex) da memória.

11.3-7 O local 0040 (hex) da memória retém um número de dois dígitos decimais em código BCD. Escrever um programa que apresente o número no indicador visual da Fig. 11.3-1 desde que os dois dígitos não sejam iguais. Se os dígitos forem iguais, o indicador visual deve ler 00.

11.3-8 Escrever um programa que determine qual das chaves está fechada na Fig. 11.3-2 e apresente o número da chave (0 a 7) num dos indicadores visuais de sete segmentos da Fig. 11.3-1.

11.3-9 Escrever um programa que monitore continuamente uma das chaves na Fig. 11.3-2 e complemente cada posição de bit no local 0040 (hex) da memória a cada novo fechamento.

11.3-10 Além da chave na Fig. 11.3-2 há uma outra chave denominada LOAD. Mostrar o hardware e escrever um programa que monitore continuamente a chave LOAD. Quando a chave LOAD for encontrada fechada, os dados de entrada na Fig. 11.3-2 são transferidos para o local 0040 (hex) da memória.

11.6-1 Escrever uma sub-rotina que gere um atraso igual a 1 ms vezes o número registrado no acumulador. Admitir que o 8080 esteja operando com um relógio de 2 MHz. Usar a informação de temporização dada no Apêndice C. Ignorar o tempo usado pelas instruções CALL e RETURN.

11.6-2 Usando o hardware da Fig. 11.5-1, escrever um programa que gere na saída Q do latch um pulso positivo cuja duração seja 10^3 ciclos de relógio.

11.6-3 Usar o hardware da Fig. 11.3-1. Escrever um programa que apresente seqüencialmente em um dos indicadores visuais de sete segmentos os números 0 a 9. Admitir que a UCP opere sobre um relógio de 2 MHz. Fornecer indicadores visuais de temporização de modo que o número exposto mude a cada segundo.

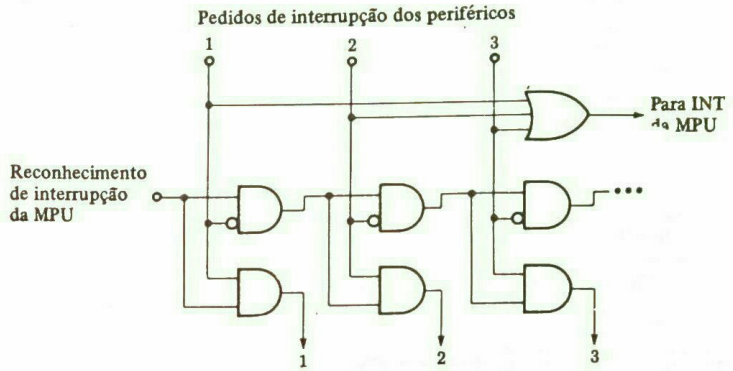
11.6-4 Escrever um programa que elimine a trepidação (*bounce*) de uma chave mecânica. Quando houver uma mudança na posição da chave, a UCP espera um tempo razoável e examina a posição da chave novamente para verificar se ela está na mesma nova posição.

11.6-5 Usando o hardware apropriado auxiliar da UCP, escrever um programa que gere uma forma-de-onda em onda quadrada. Considerar tanto uma forma-de-onda simétrica quanto uma assimétrica.

11.7-1 Consultar a Fig. 11.7-1. Escrever um programa que fará com que a UCP espere por um sinal de interrupção do teclado e depois coloque os dados do teclado num local da memória.

11.9-1 Consultar a Fig. 11.9-1. Escrever um programa que, em resposta a uma entrada INT, interrogue os periféricos e faça a manutenção deles em ordem. Localizar as sub-rotinas de serviço em locais arbitrários designados INT 1, INT 2 e INT 3. Antes de responder ao pedido de interrupção, os pares de registradores B, D e H e o registrador PSW devem ser salvos.

11.9-2 A Fig. P11.9-2 mostra um esquema "daisy chain" para atribuir prioridade a pedidos de interrupção originários dos periféricos. Explicar a operação. Que periférico tem a mais alta prioridade?



Reconhecimentos de interrupção para os periféricos

ÍNDICE ANALÍTICO

Acesso direto à memória (DMA), 450
Adição de números binários, 180-188, 204-206
  programa para, 363-366
Álgebra booleana, 7, 15-18
  aplicação de, exemplos de, 28-32
  dualidade na, 15
  princípio distributivo na, 16
  teorema de De Morgan, 18
  teoremas de, 15-18
Apontador (indicador) de pilha, 241, 384, 410
Arquitetura
  de microprocessadores
    8080, 388, 391
    6800, 419-422
  de tipo aprimorado, 350-355
  de um computador simples, 332
  de um sistema digital, 313-314
Arranjo lógico programável, 226-228
Assertiva de um sinal lógico, 90
Associatividade
  de função AND, 5
  de função EXCLUSIVE-OR, 23
  de função OR, 6
Atividade de um sinal lógico, 90
Atraso, propagação, 45, 74
  num biestável (flip-flop), 145-147

Biestáveis (ver Flip-Flops)
Bit
  controle de carga, 374
  definição de, 36
  endereço, 208
  estado no 8080, 425
  paridade, 36
Bus, 116, 307
  bidirecional, 430
Byte, 36

Chamada de sub-rotina, ilustração do programa, 370-372
Chave, operada por lógica, 82-83
  em circuito totem-pole, 86-87
  em ligação wired-AND, 85-86
  uso de: em porta OR, 83-86
Chave sem trepidação, 123-124
Ciclo de busca, 355, 431
  tabela, 336
Ciclo de execução, 336
Ciclo de máquina no 8080, 425, 450
Circuito Mealy, 263-271
Circuito Moore, 263-271
Circuitos integrados (CI), 75
  exemplos de, 88-90
Circuitos seqüenciais, 248-303
  contadores como, 249-259
  estados de, 248-249
  modo fundamental, 283-303
    exemplos de, 284-288
  síncronos, 149-283
Circuitos somadores, 180-188
  calculador, 186-188
  decimal codificado em binário (BCD), 204-206

  meio-somador, 183
  somador em paralelo, 185-186
  somador em série, 184-185
  somador inteiro (total), 183-184
  somador rápido, 189
    (ver também Somador com transporte antecipado)
Codificadores, 102-107
  prioridade, 102-107, 448
    diagrama lógico do tipo 9318, 105
Codificadores de prioridades, 102-107, 448
  diagrama lógico do tipo 9318, 105
Códigos
  ASCII, 36, 452-453
  dados, 36
  decimal codificado em binário (BCD), 34-35, 204-206
  numéricos, 34-35
  paridade, 36
  refletidos, 35
Códigos numéricos, 34-35
Complementação, 307
Componentes de um computador, 354
Computadores, 350-385
  componentes de, 354
  exemplo simples de, 331-335
  operação de, 335-338
  operações de controle de, 350-355
  programa armazenado para, 337
  projeto de controladores para, 338-341
Comutatividade
  de função AND, 5
  de função EXCLUSIVE-OR, 23
  de função OR, 6
Conexão wired-AND, 85-86
Confirmação de conexão, 345-349
  controlador de, exemplo de, 345-349
Contadores, 155-173
  anel, 156-158
  anel torcido (switchtail), 158-159
  bloqueio nos, 167-168
  circuitos integrados, 171-173
  como exemplo de circuito seqüencial, 249-259
  ondulação, pulsação (ripple), 168-171
  up-down (para cima-para baixo), 170, 309
  síncronos, 159-167
    comparações de velocidades, 161-162
    módulo arbitrário, 162-166
    para cima-para baixo, 166-167, 309
Controladores, 304-349
  implementação de, 318-321
  microprogramados, 378-382
  registradores de deslocamento, 321-323
  resposta condicional de, 323-328
  simples, 312-317
  usados com computador simples, 331-349
  usados para gerar seqüência da subtração, 328-330
Conversores de código, 107
Corridas em sistemas seqüenciais, 288-292
  eliminação de, 290-292

Decimal codificado em binário (BCD), 34-35, 204-206
Decodificadores, 98-102
  símbolos de, 100-101
  tipo '138 MSI, 100
    diagrama lógico, 99
Decrementação, 309
Demultiplexação, 116-117
Deslocamento, 307-309
Desvio do microprograma, 374-375
  condicional, 375-376
Detector de seqüências, 259-263, 264-265
  projetos alternativos de, 280-283
Diagramas de Venn, 18-22
  para duas variáveis, 19
  para três variáveis, 20
  uso de, para provar os teoremas booleanos, 20
Diagramas lógicos, 33
Dispositivo de partida, 322
Dispositivos acoplados pela carga, 236-239
Dispositivos periféricos, 423
  controle de, no 8080, 437
Divisão de números binários, 206
Dualidade, 15

Encerramento, 300-301
Endereçamento, métodos de (ver Modos de endereçamento)
Endereço
  bits de, 208
  direto, 382
  indireto, 357-358, 382
Equações de excitação, 252
Estados
  atribuição de, 249-251, 280
  condição de encerramento, 300-301
  de um latch, 119
  de um sistema seqüencial, 248-249
  diagrama de, 251
  eliminação de redundantes (ver Estados redundantes)
  redundante (ver Estados redundantes)
  representação em fluxograma, 249-255
  tabela de, 252
Estados redundantes, eliminação de
  em casos irrelevantes, 294-300
  exemplo de, 275-279
  por exame, 271-273
  por partição, 273-275

Estouro, 422
Estruturas com portas
  lógica mista aplicada às, 95-98
  utilização de dois níveis, 44-46
  utilização de um tipo de porta, 46-50

Famílias de circuitos lógicos, 76-81
  encapsulamento de, 81
  lógica acoplada pelo emissor (ECL), 76, 81
  lógica transistor-transistor (TTL), 76-81
    fan-out, 77
    margem de ruído, 78-80
    níveis de tensão de, 78-80
    séries de, 77
  semicondutor metal-óxido com simetria complementar (CMOS), 80-81
Fan-out, 77
Fechamento, (ver Encerramento)
Fetch (ver Ciclo de busca)
First-in, First-out (FIFO), 233, 239
Flags
  do microprocessador 8080, 397-401
  interrupção, 341
Flip-Flop, 130-147
  chaveamento (toggle), 137
  D, 141-145
    tipo '273, símbolo lógico de, 144
  gatilhado pela borda, 139-141
  JK, 136-137
  mestre-escravo, 130-133
    detecção de 1s nos, 137-139
    diagrama de temporização para, 134-135
    entradas assíncronas em, 132-133
    entradas diretas para, 132-133
      aspecto de prevalecimento (override), 133
    seqüência de eventos no, 139
    tipo '72, símbolo lógico para, 138
  tempo de liberação no, 147
  tempo de,
    preparação, 145-147
    manutenção, 145-147
    propagação, 145-147
Fluxogramas, 249, 253-254, 317
Formas padronizadas de funções, 37-41
  produto de somas, 39-41
  soma de produtos, 37-39
Função
  definição de, 1
  lógica (ver Função lógica)
  tabela, 2
    (ver também Função específica)
Função AND (E), 4
  exemplo de, 5
  símbolo de porta, 8
Função Equivalência, 23
Função EXCLUSIVE-NOR, 23
Função EXCLUSIVE-OR, 23
Função IMPLICAÇÃO, 25
Função lógica, 4-6, 22-27, 37-73
  de duas variáveis, 4, 22
    (ver também Mapas de Karnaugh)
  de uma única variável, 3
  formas padronizadas de, 37-41
    produto padronizado de somas, 39-41
    soma padronizada de produtos, 37-39
  função AND, 4
  função EXCLUSIVE-OR, 23
  função IMPLICAÇÃO, 25
  função NAND, 24
  função NOR, 24
  função OR, 6
Função NAND, 24-25
  suficiência de, 27
Função NOR, 24-25
  suficiência de, 27
Função OR, 6
  exemplo de, 7
  símbolo de porta, 8

Gerar e propagar transporte, 190-193

Hand Shaking (ver Confirmação de conexão)
Hardware, 387

Incrementação, 309
Indicador visual, sete segmentos, 107
Indicador (apontador) de pilhas, 241, 384, 410
Instruções
  armazenar, 358-359
  chamada de sub-rotina, 360-362
  da MPU 8080: aritmética, 397-401
    acumulador direto, 394-396
    chamar e retornar, 409-413
      tabela, 411
    complementar e set, 405-406
      tabela, 406
    complementar, tabela, 456-458
    decremento e incremento, 402
      tabela, 402
    entrada-saída (ver Instruções de entrada-saída 8080)
    envolvendo par de registradores, 394
    exemplos de uso de, 401-402, 406
    incrementar e decrementar, 402
    jump (salto), 408-409
      condicional, tabela, 408
    lógicas, 403-404
      tabela, 404
    para movimento de dados, 393-394
    push and pop, 413-417

  restart, 442-443
  rotate (de girar), tabela, 406
  tabela, 400
  tabela de comparações, 405
  transfer (de transferência), tabela, 396
  um, dois e três bytes, 391-393
  uso de, exemplos de, 401-402, 406
Instruções de entrada-saída (I/O) no 8080, 429-433
  in (IN), 428, 431
  out (OUT), 428-429
Integração em escala muito grande (VISI) (ver Circuitos integrados)
Integração em larga escala (LSI) (ver Circuitos integrados)
Integração em média escala (MSI) (ver Circuitos integrados)
Integração em pequena escala (SSI) (ver Circuitos integrados)
Interrupções, 341-345
  "daisy chain", 502
  habilitar e desabilitar (enable and disable) no 8080, 443-445
  polled, 445-448
  processamento no 8080, 439-448
Inversão, lógica, 9

Laber (ver Rótulo)
Laço (ou reversão) num programa, 361, 367-368
  temporização, 437-439
Last-in, first-out (FIFO), 239-241
Latches, 118-128
  arranjo lógico, programável, 226-228
  ciclos inválidos, 167-168
  com portas NAND, 121-123
  com portas NOR, 120-121
  com pulso de ativação, 125
  como um circuito seqüencial, 248
  controlados, 125-127
  dinâmicos, 125
  estados de set e reset dos, 119
  estáticos, 118, 125
  função lógica, 4-6, 22-27, 37-73
    de duas variáveis, 4, 22
      (ver também Mapas de Karnaugh)
    de uma única variável, 3
    formas padronizadas de, 37-41
      produto padronizado de somas, 39-41
      soma padronizada de produtos, 37-39
    função AND, 4
    função EXCLUSIVE-OR, 23
    função IMPLICAÇÃO, 25
    função NAND, 24
    função NOR, 24
    função OR, 6
  limitação de, 128
  terminais de controle de, 120
  tipo *D*, 125
  tipo '279, diagramas lógicos de, 123
  tipo '373, símbolo de, 126
  transparentes, 126
  uso em chave sem trepidação, 123-124
Lógica acoplada pelo emissor (ECL), 76, 81
Lógica de simetria complementar do semicondutor de óxido-metal (CMOS), 80-81
Lógica mista, 90-98
  aplicada às estruturas das portas, 95-98
Lógica transistor-transistor (TTL), 76-81
  fan-out, 77
  margem de ruído, 80

Macroinstrução, 373-374
Mapas K (ver Mapas de Karnaugh)
Mapas de Karnaugh, 50-73
  adjacências lógicas no, 56-61
  cinco e seis variáveis, 61-64
  definição de, 50
  duas variáveis, 51
  quatro variáveis, 53
  três variáveis, 53
  uma variável, 51
  uso de
    com funções especificadas incompletamente, 72-73
    para desenvolver função em mintermos ou em maxtermos, 70-72
    para simplificar funções, 54, 64-70
Margem de ruído, 78-80
Maxtermos, 41-44
  especificação de função com, 42
  numeração de, 41
  relação entre mintermos e tabela verdade, 42-44
Memória, 207-226, 228-247
  acesso aleatório (RAM), 207-209
  de um sistema seqüencial, 252-253
  dinâmica, 228-231
    amplificador de leitura para, 230
    características de, 231-232
    detecção (leitura) de dados em, 229
    pré-carregamento em, 230
    refrescamento de, 228
  leitura exclusiva (ROM), 214-221
    apagável (erasable), 220
    implementação de, 217-220
    programável, 220
  serial (ou em série), 231
    armazenamento de grande capacidade, 241-245
    disco flexível (floppy), 244-245
    disco magnético, 243-244
    dispositivos acoplados por carga, 236-239

first-in, first-out (FIFO), 233
fita magnética, 242-243
last-in, first-out (LIFO), 239
tempos de chaveamento de, 221-226
uso em microprogramação, 370-374, 380-382
volatilidade de, 221
Memória de acesso aleatório (RAM), 207-209
dinâmica (ver Memória, dinâmica)
Memória em série (ver Memória, serial)
Memória exclusiva de leitura (ver Memória, leitura exclusiva)
Métodos de endereçamento
da MPU 6800, 420-421
direto, 382
imediato, 383
indexado, 383
indireto, 382
indireto de registrador, 383
inerente, 383
relativo, 382-383
Microcomputadores, 386
Microinstrução, 373-374
Microoperações, 315
seqüências de, 355-363
Microprocessadores, 386-422
arquitetura de
6800, 419-422
8080, 388-391
conexões à memória
no 6800, 419-420
no 8080, 390-391
instruções de (ver Instruções, do MPU 8080)
Microprogramação, 370-382
conteúdo da ROM de controle na, 380-382
mapeamento na, 380
Mintermos, 41-44
especificação de função com, 42
numeração de, 41
relação entre maxtermos e tabela verdade, 42-44
Multiplexação, 113-116
com saídas de coletor aberto, 113-114
com saídas de três-estados, 114-116
Multiplexador, 108-113
diagrama lógico do tipo 157, 111
símbolo lógico do tipo, 151, 112
usado como um gerador de funções, 109-110
Multiplicação de números binários
programa do 8080 para, 417-419
programa para, 368-369

Notação
usando símbolos 0 e 1, 10
vírgula binária, 11-12
vírgula decimal, 11-12
Notação de dependência, 100, 247
Números com sinais, representações de, 174-180
complemento de dois, 176-179
complemento de um, 179-180
Nybble, 36

Operação de única etapa do 8080, 449
Operação síncrona, 128, 151
Operações aritméticas, 174-206
adição, 180-184, 204-206, 363-366
(ver também Circuitos somadores)
divisão, 206
multiplicação, 206, 368-369
subtração, 188-189, 363-366
Operações de controle de um computador, 350-355
Operações de entrada-saída (I/O), 423-451
interrupção (ver Interrupções)
isoladas, 427
endereçamento para, 429
memória mapeada, 428
teclado decodificado, 434-437
uso de, no 8080, 432-437
(ver também Sinais de controle de entrada-saída (I/O) no 8080)
Operações lógicas
relações entre, 26
suficiência de, 27

Palavra de estado do processador (PSW) da MPU 8080, 414
Pares de terminais no sistema digital, 428
Pastilha com coletor aberto, 86, 307
Pastilhas de memória, 207-209
colocação em paralelo de, 209-213
designações dos pinos sobre, 208
organização internadas, 214
simbolismo para, 245-247
terminais de, 208-209, 214
Pedido de serviço, 341
Pilhas de memória, 239-241, 383-385
indicador (apontador) de, 241, 384, 410
push and pop de, 240-241
Portas AND-OR-INVERTE (AOI), 48-50
estruturas de múltiplos níveis, 44-46
estruturas de um tipo, 46-50
lógica mista aplicada às, 95-98
símbolo
de AND, 8
de EXCLUSIVE-OR, 23
de NAND, 24
de NOR, 24
de OR, 8
Problema da corrida, 139
Produto velocidade-potência, 77
Programa de atraso de tempo no 8080, 438-439
Programa de instruções, 332

Raiz de um sistema de numeração, 11
Registrador de código de condição da MPU 6800, 421-422
Registrador de endereços da memória, 334, 352
Registrador de instruções, 330, 334
Registrador de operações, 352
Registrador do acumulador, 186, 314, 394-396
Registrador do contador de programas, 334, 352
Registradores
acumulador, 186, 314, 394-396
armazenamento, 186, 313
bombeamento, 376-378
definição de, 119
deslocamento (shift), 129, 151-155
deslocamento à direita, deslocamento à esquerda, 152-153
mudança de formato série-paralelo com, 152
mudança de velocidade com, 153
rotação com, 155
tipo 194, diagrama lógico de, 154
tipo 164, símbolo lógico de, 156
operação, 304
sensível a múltiplos comandos, 310-312
transferência entre, 147-150, 304-307
Representação de números com sinais por complemento de dois, 176-179
Representação de números com sinais por complemento de um, 179-180
Retentores (ver Latches)
Riscos ou perigo, 127
em circuitos assíncronos, 301-303
essenciais, 302-303
Rotação do conteúdo do registrador, 155, 353
Rótulo, 365

Saída de três-estados (tristate), 87-88
Saída totem-pole, 86-87
Saturação de transistores, 77
Shift-Register, 321-323
(ver também Registradores, deslocamento)
Sinais de controle de entrada-saída (I/O) no 8080, 423-427
bus de dados no (DBIN), 426
ciclo de busca (M_1), 431
endereço de pilha (stack), 431
escrita de entrada-saída ($\overline{I/OW}$), 426
escrita de memória ($\overline{MEMW}$), 391, 427
escrita ($\overline{WR}$), 427
interromper desabilitação (DI), 443-445
interromper habilitação (EI), 443-445
leitura de entrada-saída ($\overline{I/OR}$), 426
leitura de memória ($\overline{MEMR}$), 391, 426
pedido de interrupção (INT), 440
pulso de ativação de estado, 424
ready, 448
reconhecimento de interrupção ($\overline{INTA}$), 426
reconhecimento de parada (HLTA), 431
reconhecimento de retenção ($\overline{HOLDA}$), 450
reset, 451
reter (hold), 450
saída de escrita ($\overline{WO}$), 431
sincronização, 425
wait (espera), 448
Sinal lógico, 90
Sincronização, 127-128, 135-136
bifásica, 135-136
na MPU 8080, 425
pulsos usados para, 128
Sistema de numeração binário, 11-12
Sistema de numeração hexadecimal, 13-14
Sistema de numeração octal, 13-14
Sistemas de numeração, 11-14
base de, 11
binário, 11-12
conversão entre, 12-13
hexadecimal, 13-14
octal, 13-14
raiz de, 11
Software, 385
Somador com transporte antecipado, 190-198
aplicado a grupos, 194-198
diagrama lógico, tipo 182, 455
símbolo lógico, tipo 182, 202
uso estendido de, 196-198
Stack pointer (ver Apontador de pilha)
Sub-rotinas, encaixamento, 412
Subtração de números binários, 188-189
programa para, 363-366

Tabela de estados, 163
Tabela de transição, 159, 252
primitiva, 293
Tabela verdade, 3
relação com os mintermos e os maxtermos, 42-44
Teclado não-codificado, 434-437
Tempo de atraso de propagação, 74, 77, 147, 221
Tempo de estado no 8080, 424
Tempo de liberação em um flip-flop, 147
Tempo de preparação (set up) em um flip-flop, 146
Tempo de retenção em um flip-flop, 145-147
Tempos de chaveamento da memória, 221-226
Teorema de De Morgan, 18
Terminal de controle
exemplo de, 304, 310-312
simbolismo para, 90-95
Tipos de transistores
bipolar, 76

metal-óxido semicondutor (MOS), 76
Schottky, 77
Transparência de um latch, 126

União de variáveis lógicas, 20
Unidade do microprocessador (MPU), 387
Unidade lógica e aritmética (ULA), 199-204, 337
diagrama lógico, tipo '181, 454

Variáveis
campo de, 1
estado presente, 253
estado seguinte, 253
excitação, 284
lógicas (ver Variáveis lógicas)
secundárias, 284
Variáveis lógicas, 2-6
complemento de, 9
intersecção de, 20
número binário, relação para, 14-15
representação elétrica de, 79
união de, 20
valores de, 2
Volatilidade da memória, 221

Impressão e acabamento
(com filmes fornecidos).
EDITORA SANTUÁRIO
Fone (0125) 36-2140
APARECIDA - SP