

# C

## **Mapeando o Controle no Hardware**

*Um formato personalizado como este é escravo da arquitetura do hardware e do conjunto de instruções a que ele serve. O formato precisa atingir uma conciliação apropriada entre tamanho da ROM, decodificação da saída da ROM, tamanho dos circuitos e tempo de execução da máquina.*

**Jim McKeiv** *et al.* – Relatório de projeto do 8086, 1977

<b>C.1</b>	<b>Introdução</b>	<b>C-2</b>
<b>C.2</b>	<b>Implementando unidades de controle combinacionais</b>	<b>C-3</b>
<b>C.3</b>	<b>Implementando controle com máquinas de estados finitos</b>	<b>C-6</b>
<b>C.4</b>	<b>Implementando a função de próximo estado com um sequenciador</b>	<b>C-16</b>
<b>C.5</b>	<b>Traduzindo um microprograma para hardware</b>	<b>C-20</b>
<b>C.6</b>	<b>Comentários finais</b>	<b>C-24</b>
<b>C.7</b>	<b>Exercícios</b>	<b>C-24</b>

---

## C.1

## Introdução

O controle normalmente possui duas partes: uma parte combinacional sem estados e uma unidade de controle seqüencial que manipula seqüenciação e o controle principal em um projeto multiciclo. As unidades de controle combinacionais normalmente são usadas para tratar parte do processo de decodificação e controle. O controle da ALU no Capítulo 5 é um bom exemplo. Uma implementação de ciclo único como a do Capítulo 5 também pode usar um controlador combinacional, já que ele não exige múltiplos estados. A Seção C.2 examina a implementação dessas duas unidades combinacionais por meio das tabelas verdade do Capítulo 5.

Como as unidades de controle seqüencial são maiores e em geral mais complexas, há uma variedade maior de técnicas para implementar uma unidade de controle seqüencial. A utilidade dessas técnicas depende da complexidade do controle, das características como o número médio de próximos estados para um determinado estado e da tecnologia de implementação.

A maneira mais simples de implementar uma função de controle seqüencial é com um bloco de lógica que toma como entradas o estado atual e o campo opcode do registrador Instruction e produz como saída os sinais de controle do caminho de dados e o valor do próximo estado. A representação seqüencial pode ser um diagrama de estados finitos ou um microprograma. No último caso, cada microinstrução representa um estado. Em uma implementação usando um controle de estados finitos, a função de próximo estado será computada com lógica. A Seção C.3 constrói essa implementação para uma ROM e uma PLA.

Um método alternativo de implementação calcula a função de próximo estado usando um contador que incrementa o estado atual para determinar o próximo estado. Quando o próximo estado não segue seqüencialmente, outra lógica é usada para determinar o estado. A Seção C.4 explora esse tipo de implementação e mostra como ele pode ser usado para o controle de estados finitos criado no Capítulo 5.

Na Seção C.5, mostramos como uma representação de microprograma do controle seqüencial é traduzida para lógica de controle.

## C.2

## Implementando unidades de controle combinacionais

Nesta seção, mostraremos como a unidade de controle da ALU e a unidade de controle principal para o projeto de clock único são mapeadas para o nível de portas lógicas. Com sistemas de CAD modernos, esse processo é completamente mecânico. Os exemplos incluem como um sistema de CAD tira vantagem da estrutura da função de controle, incluindo a presença de “don’t cares”.

### Mapeando a função de controle da ALU para portas lógicas

A Figura C.2.1 mostra a tabela verdade para a função de controle da ALU desenvolvida na Seção 5.3. Um bloco lógico que implementa essa função de controle da ALU terá três saídas distintas (chamadas Operation2, Operation1 e Operation0), cada uma correspondendo a um dos três bits do controle da ALU na última coluna da Figura C.2.1. A função de lógica para cada saída é construída combinando todas as entradas da tabela verdade que definem essa saída em especial. Por exemplo, o bit menos significativo do controle da ALU (Operation0) é definido pelas duas últimas entradas da tabela verdade na Figura C.2.1. Portanto, a tabela verdade para Operation0 terá essas duas entradas.

A Figura C.2.2 mostra a tabela verdade para cada um dos três bits de controle da ALU. Tiramos proveito da estrutura comum em cada tabela verdade para incorporar “don’t cares” adicionais. Por exemplo, as cinco linhas na tabela verdade da Figura C.2.1 que definem Operation1 são reduzidas a apenas duas entradas na Figura C.2.2. Um programa de minimização lógica usará “don’t cares” para reduzir o número de portas lógicas e o número de entradas para cada porta lógica em uma concepção em portas lógicas dessas tabelas verdade.

Da tabela verdade simplificada na Figura C.2.2, podemos gerar a lógica mostrada na Figura C.2.3, que chamamos o *bloco de controle da ALU*. Esse processo é simples e pode ser feito com um programa de CAD (Computer-Aided Design – projeto auxiliado por computador). Um exemplo de como as portas lógicas podem ser derivadas das tabelas verdade é apresentado na legenda da Figura C.2.3.

Essa lógica de controle da ALU é simples porque há somente três entradas e apenas algumas combinações de entrada possíveis precisam ser reconhecidas. Se um grande número de códigos de função da ALU possíveis precisasse ser transformado em sinais de controle da ALU, esse método simples não seria eficiente. Em vez disso, você poderia usar um decodificador, uma memória ou um array estruturado de portas lógicas. Essas técnicas são descritas no Apêndice B e veremos exemplos quando examinarmos a implementação do controlador multiciclo na Seção C.3.

OpALU		Campo Funct						Operation
OpALU1	OpALU2	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

**FIGURA C.2.1** A tabela verdade para os bits de controle da ALU (chamados Operation) como uma função de OpALU e do campo de código Function. Essa tabela é igual à mostrada na Figura 5.13.

OpALU		Campos de código Function					
OpALU1	OpALU0	F5	F4	F3	F2	F1	F0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

a. A tabela verdade para Operation2 = 1 (essa tabela corresponde ao bit mais à esquerda do campo Operation na Figura C.2.1)

OpALU		Campos de código Function					
OpALU1	OpALU0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

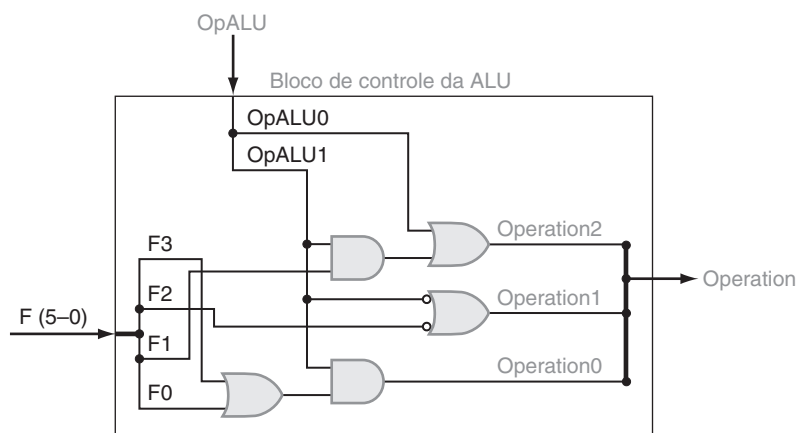
b. A tabela verdade para Operation1 = 1

OpALU		Campos de código Function					
OpALU1	OpALU0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X

c. A tabela verdade para Operation0 = 1

**FIGURA C.2.2 As tabelas verdade para as três linhas de controle da ALU.** Apenas as entradas para as quais a saída é 1 são mostradas. Os bits em cada campo são numerados da direita para a esquerda começando com 0; assim, F5 é o bit mais significativo do campo Function, e F0 é o bit menos significativo. Da mesma forma, os nomes dos sinais correspondentes ao código de operação de 3 bits fornecidos para a ALU são Operation2, Operation1 e Operation0 (com o último sendo o bit menos significativo). Portanto, a tabela verdade anterior mostra as combinações de entrada para as quais o controle da ALU deve ser 010, 001, 110 ou 111 (as combinações 011, 100 e 101 não são usadas). Os bits de OpALU são denominados OpALU1 e OpALU0. Os três valores de saída dependem do campo OpALU de 2 bits e, quando esse campo é igual a 10, do código de função de 6 bits na instrução. De igual modo, quando o campo OpALU não é igual a 10, não nos importamos com o valor do código de função (ele é representado por um X). Veja o Apêndice B para saber mais sobre “don’t cares”.

**Detalhamento:** em geral, uma equação lógica e uma representação de tabela verdade de uma função lógica são equivalentes. (Discutimos isso em mais detalhes no Apêndice B.) Entretanto, quando uma tabela verdade apenas especifica as entradas que resultam em saídas não zero, ela pode não descrever completamente a função lógica. Uma tabela verdade completa indica todas as entradas “don’t care”. Por exemplo, a codificação 11 para OpALU sempre gera um “don’t care” na saída. Portanto, uma tabela verdade completa teria XXX na parte da saída para todas as entradas com 11 no campo OpALU. Esses “don’t cares” nos permitem substituir o campo OpALU 10 e 01 por 1X e X1, respectivamente. Incorporar os “don’t cares” e minimizar a lógica é algo complexo e propenso ao erro, e, portanto, mais apropriado para um programa.



**FIGURA C.2.3 O bloco de controle da ALU gera os três bits de controle da ALU, com base nos bits do código Function e de OpALU.** Essa lógica é gerada diretamente da tabela verdade da Figura C.2.2. Considere a saída Operation2, gerada por duas linhas na tabela verdade para Operation2. A segunda linha é o AND dos dois termos (F1 = 1 e OpALU1 = 1); a porta AND superior de duas entradas corresponde a esse termo. O outro termo que faz com que Operation2 seja ativado é OpALU0. Esses dois termos são combinados com uma porta OR cuja saída é Operation2. As saídas Operation0 e Operation1, do mesmo modo, são derivadas da tabela verdade.

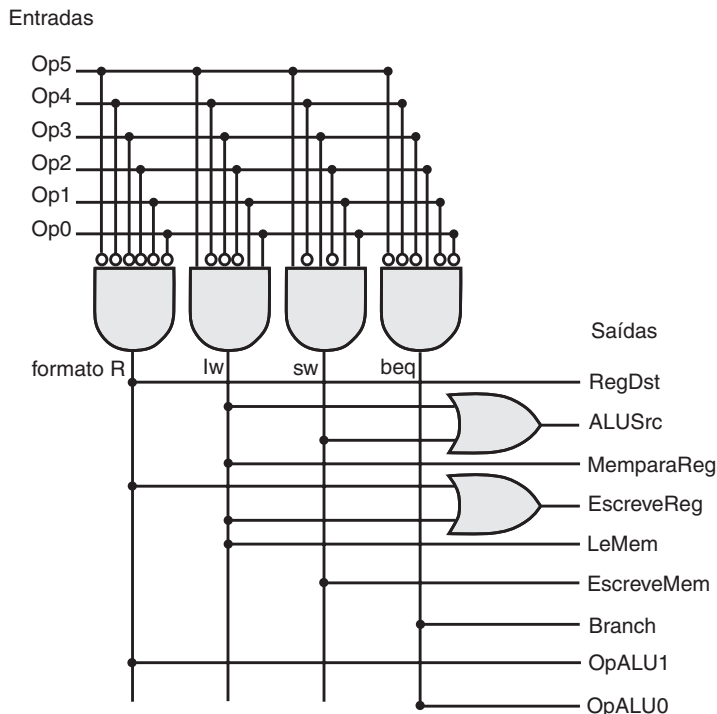
Controle	Nome do sinal	formato R	lw	sw	beq
Entradas	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Saídas	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemparaReg	0	1	X	X
	EscreveReg	1	1	0	0
	LeMem	0	1	0	0
	EscreveMem	0	0	1	0
	Branch	0	0	0	1
	OpALU1	1	0	0	0
	OpALU0	0	0	0	1

**FIGURA C.2.4** A função de controle para a implementação simples de clock único é completamente satisfeita por essa tabela verdade. Essa tabela é igual à mostrada na Figura 5.22.

### Mapeando a função de controle principal em portas lógicas

Implementar a função de controle principal com uma coleção desestruturada de portas lógicas, como fizemos para o controle da ALU, é uma solução razoável porque a função de controle nem é complexa nem grande, como podemos ver na tabela verdade mostrada na Figura C.2.4. Entretanto, se a maioria dos 64 opcodes possíveis fosse usada e houvesse muitas outras linhas de controle, o número de portas lógicas seria muito maior e cada porta lógica teria muito mais entradas.

Como qualquer função pode ser calculada em dois níveis de lógica, outra maneira de implementar uma função lógica é com um array lógico estruturado de dois níveis. A Figura C.2.5 mostra essa implementação. Ela usa um array de portas AND seguido de um array de portas OR. Essa estrutura é chamada de *array lógico programável* (PLA). Uma PLA é um dos meios mais comuns de implementar uma função de controle. Retornaremos ao tema de usar elementos lógicos estruturados para implementar controle quando implementarmos um controlador de estados finitos na próxima seção.



**FIGURA C.2.5 A implementação estruturada da função de controle como descrita pela tabela verdade na Figura C.2.4.** A estrutura, chamada de array lógico programável (PLA), usa um array de portas AND seguido de um array de portas OR. As entradas para as portas AND são as entradas de função e seus inversos (as bolhas indicam a inversão de um sinal). As entradas para as portas OR são as saídas das portas AND (ou, como um caso degenerado, as entradas da função e seus inversos). As saídas das portas OR são as saídas da função.

## C.3

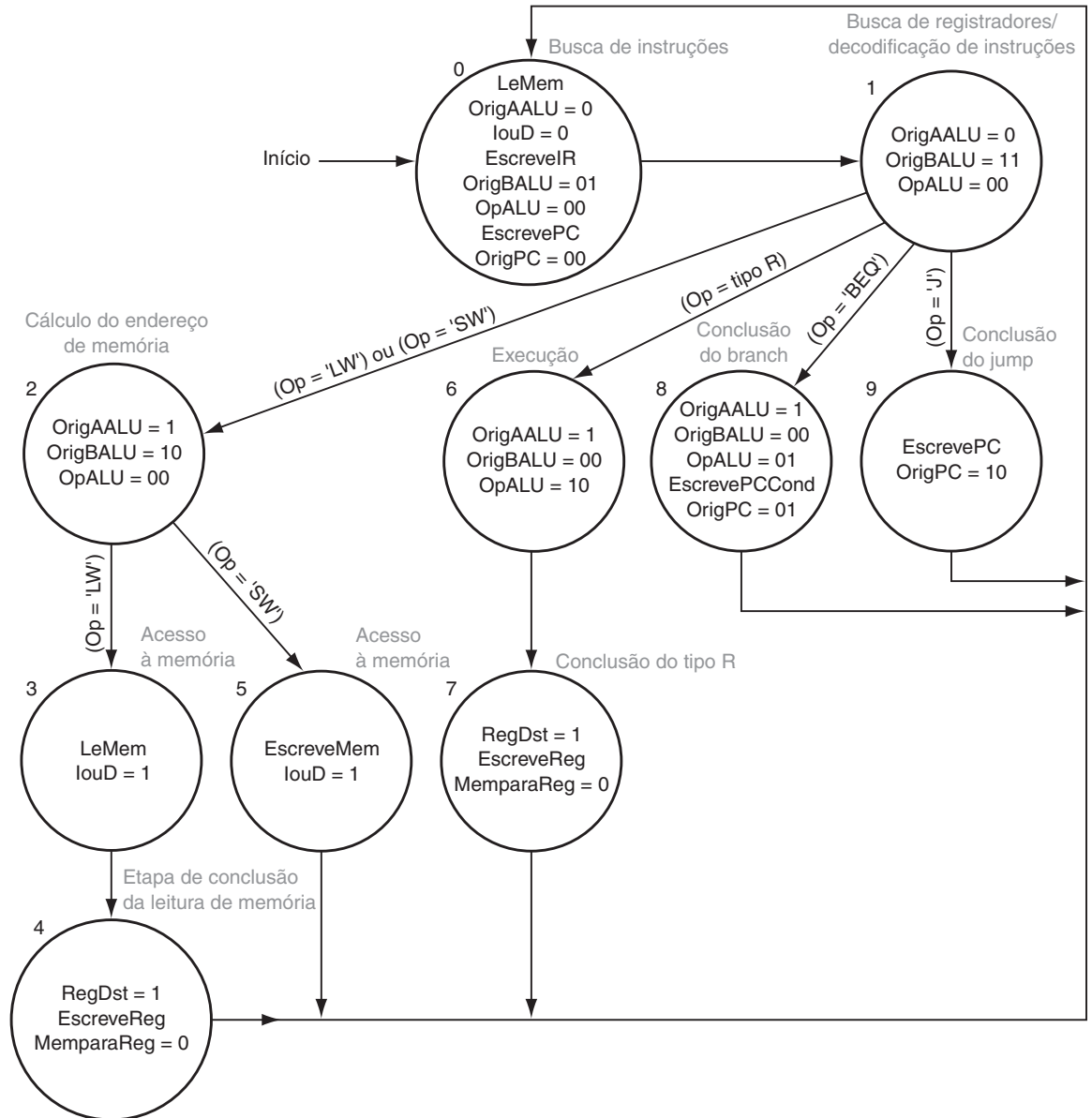
### Implementando controle com máquinas de estados finitos

Para implementar o controle como uma máquina de estados finitos, precisamos primeiro atribuir um número a cada um dos 10 estados; qualquer estado poderia usar qualquer número, mas, para simplificar, usaremos a numeração sequencial, como no Capítulo 5. (A Figura C.3.1 é uma cópia do diagrama de estados finitos da Figura 5.38, reproduzida para facilidade de acesso.) Com 10 estados, precisaremos de 4 bits para codificar o número do estado e chamamos esses bits de estado de S3, S2, S1 e S0. O número do estado atual será armazenado em um registrador de estado, como mostra a Figura C.3.2. Se os estados foram atribuídos sequencialmente, o estado  $i$  será codificado usando os bits de estado como o número binário  $i$ . Por exemplo, o estado 6 é codificado como 0110<sub>bin</sub> ou S3 = 0, S2 = 1, S1 = 1, S0 = 0, que também pode ser escrito como

$$\overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0}$$

A unidade de controle possui saídas que especificam o próximo estado. Essas são escritas no registrador de estado na transição do clock e se tornam o novo estado no início do próximo ciclo de clock seguinte à transição ativa do clock. Essas saídas são denominadas NS3, NS2, NS1, NS0. Uma vez determinados os números das entradas, os estados e as saídas, sabemos como se parecerá a estrutura básica da unidade de controle, como vemos na Figura C.3.2.

O bloco rotulado como “lógica de controle” na Figura C.3.2 é lógica combinacional. Podemos pensar nele como uma grande tabela dando o valor das saídas em função das entradas. A lógica nesse bloco implementa as duas partes diferentes da máquina de estados finitos. Uma parte é a lógica que

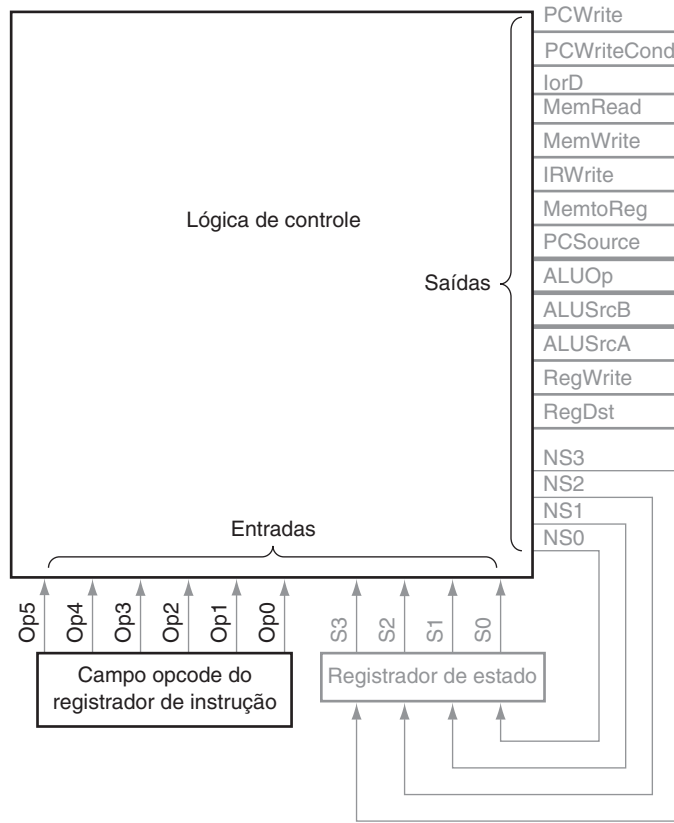


**FIGURA C.3.1** O diagrama de estados finitos desenvolvido no Capítulo 5; ele é idêntico à Figura 5.38.

determina a definição das saídas de controle do caminho de dados, que dependem apenas dos bits de estado. A outra parte da lógica de controle implementa a função de próximo estado; essas equações determinam os valores dos bits de próximo estado e as outras entradas (o opcode de 6 bits).

A Figura C.3.3 mostra as equações lógicas: a parte superior mostrando as saídas e a parte inferior mostrando a função de próximo estado. Os valores nessa tabela foram determinados a partir do diagrama de estados na Figura C.3.1. Sempre que uma linha de controle está ativa em um estado, esse estado é inserido na segunda coluna da tabela. Da mesma forma, as entradas de próximo estado são feitas sempre que um estado é um sucessor para outro.

Na Figura C.3.3, usamos a abreviatura *stateN* para indicar o estado atual *N*. Portanto, *stateN* é substituído pelo termo que codifica o número de estado *N*. Usamos *NextStateN* para representar a definição das saídas de próximo estado para *N*. Essa saída é implementada usando as saídas de próximo estado (NS). Quando *NextStateN* está ativo, os bits NS[3-0] são definidos de acordo com a versão binária do valor *N*. É claro que, como um determinado bit de próximo estado é ativado em múltiplos



**FIGURA C.3.2 A unidade de controle para MIPS consistirá em alguma lógica de controle e em um registrador para conter o estado.** O registrador de estado é escrito na transição ativa do clock e é estável durante o ciclo de clock.

próximos estados, a equação para cada estado será o OR dos termos que ativam esse sinal. Da mesma forma, quando usamos um termo como ( $Op = 'lw'$ ), isso corresponde a um AND das entradas de opcode que especifica a codificação do opcode  $lw$  em 6 bits, exatamente como fizemos para a unidade de controle simples na seção anterior deste capítulo. É simples traduzir as entradas na Figura C.3.3 em equações lógicas para as saídas.

### EQUAÇÕES LÓGICAS PARA AS SAÍDAS DE PRÓXIMO ESTADO

Forneça a equação lógica para o bit de próximo estado menos significativo, NS0.

**EXEMPLO**

O bit de próximo estado NS0 deve estar ativo sempre que o próximo estado tiver  $NS0 = 1$  na codificação de estado. Isso é verdadeiro para NextState1, NextState3, NextState5, NextState7 e NextState9. As entradas para esses estados na Figura C.3.3 fornecem as condições quando esses valores de próximo estado devem estar ativos. A equação para cada um desses próximos estados é fornecida a seguir. A primeira equação diz que o próximo estado é 1 se o estado atual é 0; o estado atual é 0 se cada um dos bits de entrada de estado for 0, que é o que indica o termo de produto mais à direita.

**RESPOSTA**

$$NextState1 = State0 = \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot \overline{S0}$$

$$NextState3 = State2 \cdot (Op[5-0] = lw)$$

$$= \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot Op5 \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot Op1 \cdot Op0$$



Saída	Estados atuais	Op
EscrevePC	state0 + state9	
EscrevePCCond	state8	
louD	state3 + state5	
LeMem	state0 + state3	
EscreveMem	state5	
EscreveIR	state0	
MemparaReg	state4	
OrigPC1	state9	
OrigPC0	state8	
OpALU1	state6	
OpALU0	state8	
OrigBALU1	state1 + state2	
OrigBALU0	state0 + state1	
OrigAALU	state2 + state6 + state8	
EscreveReg	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

**FIGURA C.3.3 As equações lógicas para a unidade de controle mostradas de uma forma resumida.** Lembre-se de que “+” significa OR em equações lógicas. As entradas de estado e as saídas das entradas NextState precisam ser expandidas usando a codificação de estado. Qualquer entrada em branco é um “don’t care”.

$$\begin{aligned}
 \text{NextState5} &= \text{State 2} \cdot (\text{Op}[5-0] = \text{sw}) \\
 &= \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot \text{Op5} \cdot \overline{\text{Op4}} \cdot \text{Op3} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0} \\
 \text{NextState7} &= \text{State6} = \overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0} \\
 \text{NextState9} &= \text{State1} \cdot (\text{Op}[5-0] = \text{jmp}) \\
 &= \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot S0 \cdot \overline{\text{Op5}} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \overline{\text{Op0}}
 \end{aligned}$$

NS0 é a soma lógica de todos esses termos.

Como já vimos, a função de controle pode ser expressa como uma equação lógica para cada saída. Esse conjunto de equações lógicas pode ser implementado de duas maneiras: correspondente a uma tabela verdade completa ou correspondente a uma estrutura lógica de dois níveis que permite uma codificação esparsa da tabela verdade. Antes de olharmos essas implementações, vejamos a tabela verdade para a função de controle completa.

É mais simples se dividirmos a função de controle definida na Figura C.3.3 em duas partes: as saídas de próximo estado, que podem depender de todas as entradas, e as saídas de sinal de controle, que

dependem apenas dos bits do estado atual. A Figura C.3.4 mostra as tabelas verdade para todos os sinais de controle do caminho de dados. Como esses sinais realmente dependem apenas dos bits de estado (e não do opcode), cada uma das entradas em uma tabela na Figura C.3.4, na verdade, representa  $64 (= 2^6)$  entradas, com os 6 bits denominados Op tendo todos os valores possíveis; isto é, os bits Op são bits “don’t care” para determinar as saídas de controle do caminho de dados. A Figura C.3.5 mostra a tabela verdade para os bits de próximo estado NS[3-0], que dependem dos bits de entrada de estado e dos bits de instrução, que formam o opcode.

**Detalhamento:** existem muitas oportunidades de simplificar a função de controle observando semelhanças entre dois ou mais sinais e usando a semântica da implementação. Por exemplo, os sinais EscrevePCCond, OrigPC0 e OpALU0 são todos ativados exatamente em um estado, o estado 8. Esses três sinais de controle podem ser substituídos por um único sinal.

s3	s2	s1	s0
0	0	0	0
1	0	0	1

a. Tabela verdade para EscrevePC

s3	s2	s1	s0
1	0	0	0

b. Tabela verdade para EscrevePCCond

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c. Tabela verdade para louD

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d. Tabela verdade para LeMem

s3	s2	s1	s0
0	1	0	1

e. Tabela verdade para EscreveMem

s3	s2	s1	s0
0	0	0	0

f. Tabela verdade para EscreveIR

s3	s2	s1	s0
0	1	0	0

g. Tabela verdade para MemparaReg

s3	s2	s1	s0
1	0	0	1

h. Tabela verdade para OrigPC1

s3	s2	s1	s0
1	0	0	0

i. Tabela verdade para OrigPC0

s3	s2	s1	s0
0	1	1	0

j. Tabela verdade para OpALU1

s3	s2	s1	s0
1	0	0	0

k. Tabela verdade para OpALU0

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l. Tabela verdade para OrigBALU1

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m. Tabela verdade para OrigBALU0

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

n. Tabela verdade para OrigAALU

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Tabela verdade para EscreveReg

s3	s2	s1	s0
0	1	1	1

p. Tabela verdade para RegDst

**FIGURA C.3.4 As tabelas verdade são mostradas para os 16 sinais de controle do caminho de dados que dependem apenas dos bits da entrada de estado atual, que são mostrados para cada tabela.** Cada tabela verdade corresponde a 64 entradas: uma para cada valor possível dos 6 bits Op. Observe que algumas das saídas estão ativas sob quase as mesmas circunstâncias. Por exemplo, no caso de EscrevePCCond, OrigPC0 e OpALU0, esses sinais estão ativos apenas no estado 8 (veja b, i e k). Esses três sinais poderiam ser substituídos por um sinal. Há outras oportunidades de reduzir a lógica necessária para implementar a função de controle tirando proveito de outras semelhanças nas tabelas verdade.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1

a. A tabela verdade para a saída NS3, que está ativa quando o próximo estado é 8 ou 9. Esse sinal é ativado quando o estado atual é 1.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	0	1	1
X	X	X	X	X	X	0	1	1	0

b. A tabela verdade para a saída NS2, que está ativa quando o próximo estado é 4, 5, 6 ou 7. Essa situação ocorre quando o estado atual é 1, 2, 3 ou 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1
1	0	0	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0

c. A tabela verdade para a saída NS1, que está ativa quando o próximo estado é 2, 3, 6 ou 7. O próximo estado é 2, 3, 6 ou 7 apenas se o estado atual é 1, 2 ou 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
X	X	X	X	X	X	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0
0	0	0	0	1	0	0	0	0	1

d. A tabela verdade para a saída NS0, que está ativa quando o próximo estado é 1, 3, 5, 7 ou 9. Isso ocorre apenas se o estado atual é 0, 1, 2 ou 6.

**FIGURA C.3.5 As quatro tabelas verdade para os quatro bits de saída de próximo estado (NS[3-0]).** As saídas de próximo estado dependem do valor de Op[5-0], que é o campo opcode, e do estado atual, fornecido por S[3-0]. As entradas com X são “don’t cares”. Cada entrada com um “don’t care” corresponde a duas entradas, uma com essa entrada em 0 e uma com essa entrada em 1. Portanto, uma entrada com  $n$  “don’t cares”, na verdade, corresponde a  $2^n$  entradas da tabela verdade.

### Uma implementação usando ROM

Provavelmente, a maneira mais simples de implementar a função de controle é codificar as tabelas verdade em uma ROM. O número de entradas na memória para as tabelas verdade das Figuras C.3.4 e C.3.5 é igual a todos os valores possíveis das entradas (os 6 bits de opcode mais os 4 bits de estado), que é  $2^{\text{entradas}} = 2^{10} = 1024$ . As entradas para a unidade de controle se tornam as linhas de endereço para a ROM, que implementa o bloco de lógica de controle mostrado na Figura C.3.2. A largura de cada entrada (ou word na memória) é de 20 bits, já que há 16 saídas de controle do caminho de dados e 4 bits de próximo estado. Isso significa que o tamanho total da ROM é  $2^{10} \times 20 = 20$  Kbits.

A definição dos bits em uma word na ROM depende de quais saídas estão ativas nessa word. Antes de vermos as words de controle, precisamos ordenar os bits dentro das words de entrada (o endereço) e de saída (o conteúdo) do controle, respectivamente. Iremos numerar os bits usando a ordem da Figura C.3.2, com os bits de próximo estado sendo os bits menos significativos da *word* de controle e os bits de entrada de estado atual sendo os bits menos significativos do *endereço*. Isso significa que a saída de EscrevePC será o bit mais significativo (o bit 19) de cada word

de memória e o NS0 será o bit menos significativo. O bit de endereço mais significativo será fornecido por Op5, que é o bit mais significativos da instrução, e o bit de endereço menos significativo será fornecido por S0.

Podemos construir o controle da ROM montando a tabela verdade inteira de uma forma em que cada linha corresponda a uma das  $2^n$  combinações de entrada únicas, e um conjunto de colunas indique que saídas estão ativas para essa combinação de entrada. Não temos espaço aqui para mostrar todas as 1.024 entradas na tabela verdade. Entretanto, podemos fazê-lo separando o controle do caminho de dados e as saídas de próximo estado, já que as saídas de controle do caminho de dados dependem apenas do estado atual. A tabela verdade para as saídas de controle do caminho de dados é mostrada na Figura C.3.6. Incluímos somente as codificações das entradas de estado que estão em uso (ou seja, os valores de 0 a 9 correspondentes aos 10 estados da máquina de estados).

Saídas	Valores de entrada (S[3-0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
EscrevePC	1	0	0	0	0	0	0	0	0	1
EscrevePCCond	0	0	0	0	0	0	0	0	1	0
IouD	0	0	0	1	0	1	0	0	0	0
LeMem	1	0	0	1	0	0	0	0	0	0
EscreveMem	0	0	0	0	0	1	0	0	0	0
EscreveIR	1	0	0	0	0	0	0	0	0	0
MemparaReg	0	0	0	0	1	0	0	0	0	0
OrigPC1	0	0	0	0	0	0	0	0	0	1
OrigPC0	0	0	0	0	0	0	0	0	1	0
OpALU1	0	0	0	0	0	0	1	0	0	0
OpALU0	0	0	0	0	0	0	0	0	1	0
OrigBALU1	0	1	1	0	0	0	0	0	0	0
OrigBALU0	1	1	0	0	0	0	0	0	0	0
OrigAALU	0	0	1	0	0	0	1	0	1	0
EscreveReg	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

**FIGURA C.3.6 A tabela verdade para as 16 saídas de controle do caminho de dados, que dependem apenas das entradas de estado.** Os valores são determinados pela Figura C.3.4. Embora haja 16 valores possíveis para o campo de estado de 4 bits, apenas 10 são usados e mostrados aqui. Os 10 valores possíveis são mostrados no alto; cada coluna mostra o valor das saídas de controle do caminho de dados para o valor de entrada de estado que aparece no alto da coluna. Por exemplo, quando as entradas de estado são 0011 (estado 3), as saídas de controle do caminho de dados são IouD e LeMem.

A tabela verdade na Figura C.3.6 fornece diretamente o conteúdo dos 16 bits mais significativos de cada word na ROM. O campo de entrada de 4 bits fornece os quatro bits menos significativos do endereço de cada word, e a coluna fornece o conteúdo da word nesse endereço.

Se mostrássemos uma tabela verdade inteira para os bits de controle do caminho de dados com o número de estado e os bits de opcode como entradas, as entradas de opcode seriam todas indiferentes. Quando construímos a ROM, não podemos ter quaisquer “don’t cares”, pois os endereços na ROM precisam estar completos. Assim, as mesmas saídas de controle do caminho de dados ocorrerão muitas vezes na ROM, já que essa parte da ROM é a mesma sempre que os bits de estado são idênticos, independente do valor das entradas de opcode.

## ENTRADAS DO CONTROLE NA ROM

### EXEMPLO

Para quais endereços de ROM o bit correspondente a *EscrivePC*, o bit mais significativo da word de controle, será 1?

### RESPOSTA

*EscrivePC* está ativo nos estados 0 e 9; isso corresponde aos endereços com os 4 bits menos significativos sendo 0000 ou 1001. O bit estará ativo na word de memória independente das entradas *Op[5-0]*, de modo que os endereços com o bit ativo são 000000000, 0000001001, 0000010000, 0000011001, ..., 1111110000, 1111111001. A forma geral disso é XXXXXX0000 ou XXXXXX1001, onde XXXXXX é qualquer combinação de bits e corresponde ao opcode de 6 bits do qual essa saída não depende.

Mostraremos o conteúdo inteiro da ROM em duas partes para facilitar a compreensão. A Figura C.3.7 mostra os 16 bits mais significativos da word de controle; isso vem diretamente da Figura C.3.6. Essas saídas de controle do caminho de dados dependem apenas das entradas de estado, e esse conjunto de words seria duplicado 64 vezes na ROM inteira, como discutimos anteriormente. Como as entradas correspondentes aos valores de entrada 1010 a 1111 não são usadas, não nos importamos com o que elas contêm.

A Figura C.3.8 mostra os 4 bits menos significativos da word de controle correspondentes às saídas de próximo estado. A última coluna da tabela na Figura C.3.8 corresponde a todos os valores possíveis do opcode que não coincidem com os opcodes especificados. No estado 0, o próximo estado é sempre o estado 1, já que a instrução ainda está sendo buscada. Após o estado 1, o campo opcode precisa ser válido. As tabelas indicam isso pelas entradas marcadas como ilegais; veremos como lidar com esses opcodes ilegais na Seção 5.6.

Essa representação como duas tabelas separadas não só é uma maneira mais compacta de mostrar o conteúdo da ROM, mas também é um meio mais eficiente de implementar a ROM. A maioria das saídas (16 de 20 bits) depende apenas de 4 das 10 entradas. O número de bits no total quando o controle é implementado como duas ROMs separadas é  $2^4 \times 16 + 2^{10} \times 4 = 256 + 4096 = 4,3\text{Kbits}$ , que é aproximadamente um quinto do tamanho de uma única ROM, que requer  $2^{10} \times 20 = 20\text{Kbits}$ . Há algum overhead associado com qualquer bloco de lógica estruturada, mas, nesse caso, o overhead adicional de uma ROM extra seria muito menor do que as economias advindas da divisão da ROM única.

4 bits menos significativos do endereço	Bits 19-4 da word
0000	1001010000001000
0001	0000000000011000
0010	0000000000010100
0011	0011000000000000
0100	0000001000000010
0101	0010100000000000
0110	0000000001000100
0111	0000000000000011
1000	0100000010100100
1001	1000000100000000

**FIGURA C.3.7 O conteúdo dos 16 bits mais significativos da ROM dependem apenas das entradas de estado.** Esses valores são iguais aos da Figura C.3.6, simplesmente girados em 90°. Esse conjunto de words de controle seria duplicado 64 vezes para todo valor possível dos 6 bits mais significativos do endereço.

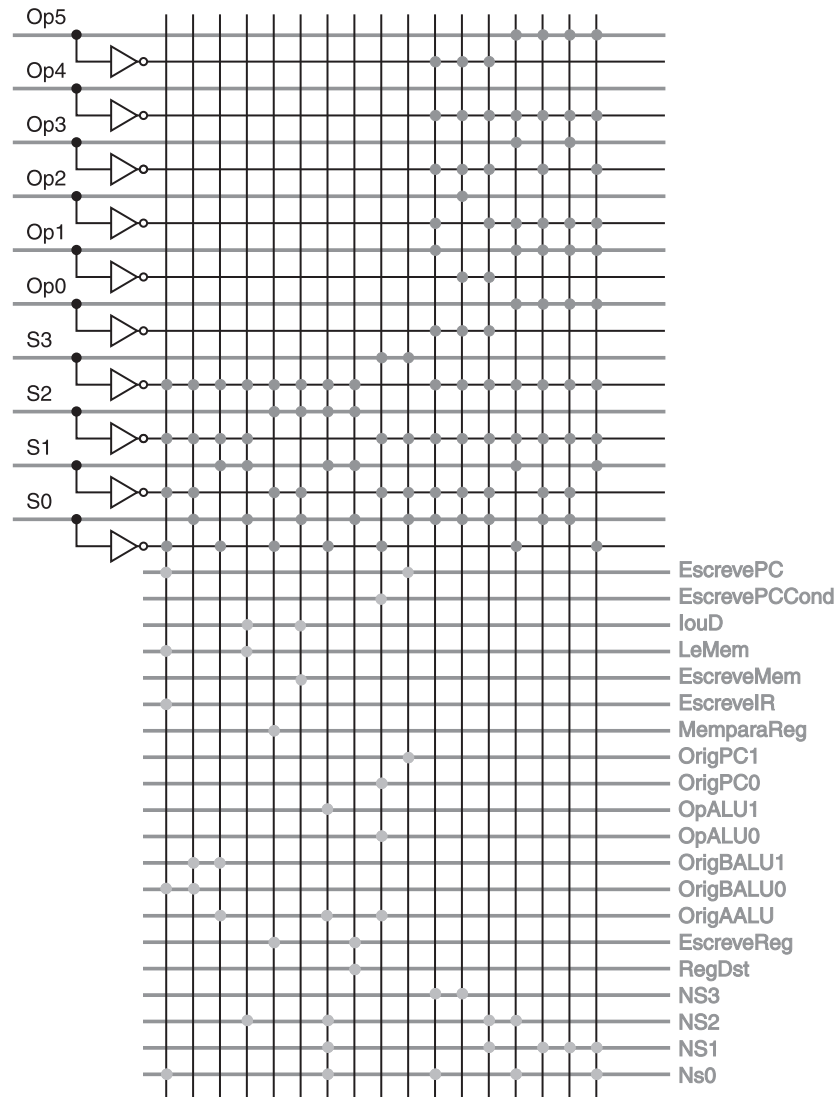
Estado atual S[3-0]	Op[5-0]					
	000000 (formato R)	000010 (jmp)	000100 (beq)	100011 (lw)	101011 (sw)	Qualquer outro valor
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	ilegal
0010	XXXX	XXXX	XXXX	0011	0101	ilegal
0011	0100	0100	0100	0100	0100	ilegal
0100	0000	0000	0000	0000	0000	ilegal
0101	0000	0000	0000	0000	0000	ilegal
0110	0111	0111	0111	0111	0111	ilegal
0111	0000	0000	0000	0000	0000	ilegal
1000	0000	0000	0000	0000	0000	ilegal
1001	0000	0000	0000	0000	0000	ilegal

**FIGURA C.3.8** Esta tabela contém os 4 bits menos significativos da word de controle (as saídas de NS), que dependem das entradas de estado, S[3-0], e do opcode, Op[5-0], que correspondem ao opcode da instrução. Esses valores podem ser determinados a partir da Figura C.3.5. O nome de opcode é mostrado sob a codificação no cabeçalho. Os 4 bits da word de controle cujo endereço é dado pelos bits de estado atual e os bits Op são mostrados em cada entrada. Por exemplo, quando os bits de entrada de estado são 0000, a saída é sempre 0001, independente das outras entradas; quando o estado é 2, o próximo estado é don't care para três das entradas, é 3 para lw e é 5 para sw. Juntamente com as entradas na Figura C.3.7, essa tabela especifica o conteúdo da ROM da unidade de controle. Por exemplo, a word no endereço 1000110001 é obtida encontrando-se os 16 bits mais significativos na tabela da Figura C.3.7 usando apenas os bits de entrada de estado (0001) e concatenando os 4 bits menos significativos encontrados usando o endereço inteiro (0001 para localizar a linha e 100011 para localizar a coluna). A entrada da Figura C.3.7 produz 0000000000011000, enquanto a entrada apropriada na tabela imediatamente acima é 0010. Portanto, a word de controle no endereço 1000110001 é 00000000000110000010. A coluna "Qualquer outro valor" se aplica apenas quando os bits de Op não correspondem a um dos opcodes especificados.

Embora essa codificação de ROM da função de controle seja simples, ela é extravagante, mesmo quando dividida em duas partes. Por exemplo, os valores das entradas do registrador Instruction normalmente não são necessários para determinar o próximo estado. Portanto, a ROM de próximo estado possui muitas entradas duplicadas ou "don't cares". Considere o caso quando a máquina está no estado 0: existem  $2^6$  entradas na ROM (já que o campo opcode pode ter qualquer valor), e essas entradas terão todas o mesmo conteúdo (a saber, a word de controle 0001). A razão por que tanta ROM é desperdiçada é que a ROM implementa a tabela verdade completa, fornecendo a oportunidade de ter uma saída diferente para cada combinação de entradas. Entretanto, a maioria das combinações das entradas nunca ocorre ou é redundante!

## Uma implementação usando PLA

Podemos reduzir a quantidade de armazenamento de controle necessária com o custo de usar decodificação de endereço mais complexa para as entradas de controle, o que codificará apenas as combinações de entrada necessárias. A estrutura lógica mais usada para isso é um array lógico programável (PLA), já mencionado e ilustrado na Figura C.2.5. Em uma PLA, cada saída é o OR lógico de um ou mais mintermos. Um *mintermo*, também chamado de *termo de produto*, é simplesmente um AND lógico de uma ou mais entradas. As entradas podem ser imaginadas como o endereço para a indexação da PLA, enquanto os mintermos selecionam quais de todas as combinações de endereço possíveis são interessantes. Um mintermo corresponde a uma única entrada em uma tabela verdade, como aquelas na Figura C.3.4, incluindo possíveis "don't cares". Cada saída consiste em um OR desses mintermos, que corresponde exatamente a uma tabela verdade completa. Contudo, diferente de uma ROM, apenas as entradas da tabela verdade que produzem uma saída ativa são necessárias, e apenas uma cópia de cada mintermo é necessária, mesmo se o mintermo contiver "don't cares". A Figura C.3.9 mostra a PLA que implementa essa função de controle.



**Figura C.3.9** Essa PLA implementa a lógica da função de controle para a implementação multiciclo.

As entradas para o controle aparecem à esquerda e as saídas, à direita. A metade superior da figura é o plano AND que calcula todos os mintermos. Os mintermos são transportados para o plano OR nas linhas verticais. Cada ponto em cinza corresponde a um sinal que compõe o mintermo transportado nessa linha. Os termos de soma são calculados desses mintermos, com cada ponto cinza representando a presença do mintermo que intercepta esse termo de soma. Cada saída consiste em um único termo de soma.

Como podemos ver pela PLA na Figura C.3.9, existem 17 mintermos únicos – 10 que dependem apenas do estado atual e 7 outros que dependem de uma combinação do campo Op e os bits de estado atual. O tamanho total da PLA é proporcional a  $(\#entradas \times \#termos \text{ de produto}) + (\#saídas \times \#termos \text{ de produto})$ , como podemos ver simbolicamente na figura. Isso significa que o tamanho total da PLA na Figura C.3.9 é proporcional a  $(10 \times 17) + (20 \times 17) = 510$ . Por comparação, o tamanho de uma única ROM é proporcional a 20Kbits e mesmo a ROM de duas partes possui um total de 4,3Kbits. Como o tamanho de uma célula de PLA será apenas ligeiramente maior do que o tamanho de um bit em uma ROM, uma PLA será uma implementação muito mais eficiente para essa unidade de controle.

É claro que, assim como dividimos a ROM em duas, podemos dividir a PLA em duas PLAs: uma com 4 entradas e 10 mintermos que gera as 16 saídas de controle, e uma com 10 entradas e 7 mintermos que gera as 4 saídas de próximo estado. A primeira PLA teria um tamanho proporcional a  $(4 \times 10) + (10 \times 16) = 200$ , e a segunda PLA teria um tamanho proporcional a  $(10 \times 7) + (4 \times 7) = 98$ . Isso



produziria um tamanho total proporcional a 298 células de PLA, cerca de 55% do tamanho de uma PLA única. Essas duas PLAs serão consideravelmente menores do que uma implementação usando duas ROMs. Para obter mais detalhes sobre PLAs e sua implementação, bem como as referências para livros sobre projeto de lógica, veja o ■ Apêndice B.

## **C.4 Implementando a função de próximo estado com um seqüenciador**

Vamos olhar cuidadosamente a unidade de controle construída na última seção. Se você examinar as ROMs que implementam o controle nas Figuras C.3.7 e C.3.8, poderá ver que muito da lógica é usada para especificar a função de próximo estado. Na verdade, para a implementação usando duas ROMs separadas, 4.096 dos 4.368bits (94%) correspondem à função de próximo estado! Além disso, imagine como seria a lógica de controle se o conjunto de instruções tivesse muito mais tipos de instrução diferentes, alguns dos quais exigindo muitos clocks para serem implementados. Haveria muito mais estados na máquina de estados finitos. Em alguns estados, poderíamos estar desviando para um número maior de estados diferentes dependendo do tipo de instrução (como fizemos no estado 1 da máquina de estados finitos na Figura C.3.1). Entretanto, muitos dos estados continuariam de uma maneira seqüencial, exatamente como os estados 3 e 4 na Figura C.3.1.

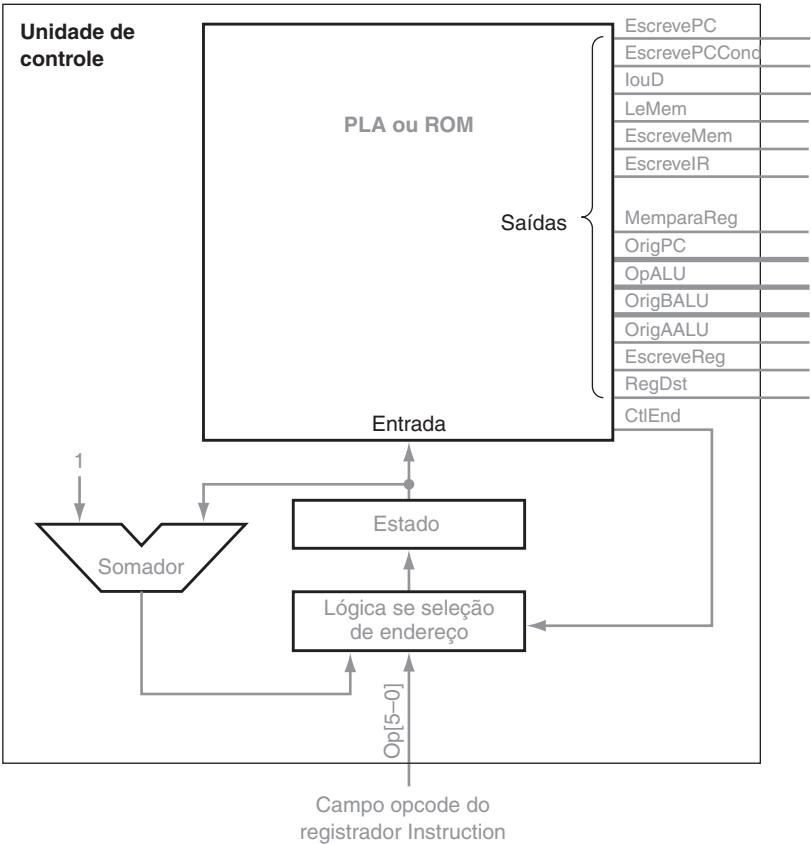
Por exemplo, se incluíssemos ponto flutuante, veríamos uma seqüência sucessiva de muitos estados que implementam uma instrução de ponto flutuante multiciclo. Como alternativa, considere como o controle pode se parecer para uma máquina que pode ter múltiplos operandos de memória por instrução. Ela exigiria que muito mais estados buscassem múltiplos operandos de memória. O resultado disso seria que a lógica de controle será dominada pela codificação da função de próximo estado. Além disso, muito da lógica será dedicada a seqüências de estados com apenas um caminho entre eles que se parecem com os estados 2 a 4 na Figura C.3.1. Com mais instruções, essas seqüências consistirão em estados numerados muito mais seqüencialmente do que para nosso subconjunto simples.

Para codificarmos de maneira eficiente essas funções de controle mais complexas, podemos usar uma unidade de controle que tenha um contador para fornecer o próximo estado seqüencial. Esse contador elimina a necessidade de codificar a função de próximo estado explicitamente na unidade de controle. Como mostra a Figura C.4.1, um somador é usado para incrementar o estado, transformando-o essencialmente em um contador. O estado incrementado é sempre o estado que se segue na ordem numérica. Todavia, a máquina de estados finitos algumas vezes “desvia”. Por exemplo, no estado 1 da máquina de estados finitos (veja a Figura C.3.1), existem quatro próximos estados possíveis, sendo que apenas um deles é o próximo estado seqüencial. Desse modo, precisamos ser capazes de escolher entre o estado incrementado e um novo estado com base nas entradas do registrador Instruction e no estado atual. Cada word de controle incluirá linhas de controle que determinarão como o próximo estado é escolhido.

É fácil implementar a parte do sinal da saída de controle da word de controle, já que, se usarmos os mesmos números de estado, essa parte da word de controle se parecerá exatamente com o controle da ROM mostrado na Figura C.3.7. No entanto, o método para selecionar o próximo estado difere da função de próximo estado na máquina de estados finitos.

Com um contador explícito fornecendo o próximo estado seqüencial, a lógica da unidade de controle só precisa especificar como escolher o estado quando ele não é o estado seqüencialmente seguinte. Há dois métodos para fazer isso. O primeiro é um método que já vimos: A unidade de controle codifica explicitamente a função de próximo estado. A diferença é que a unidade de controle só precisa definir as linhas de próximo estado quando o próximo estado designado não é o estado que o contador indica. Se o número de estados for grande e a função de próximo estado que precisamos codificar estiver principalmente vazia, isso pode não ser uma boa escolha, já que a unidade de controle resultante terá muito espaço vazio ou redundante. Um método alternativo é usar lógica externa separada para especificar o próximo estado quando o contador não especifica o estado. Muitas unidades de controle, especialmente aquelas que implementam grandes conjuntos de instruções, usam esse método, e iremos focalizar a especificação externa do controle.





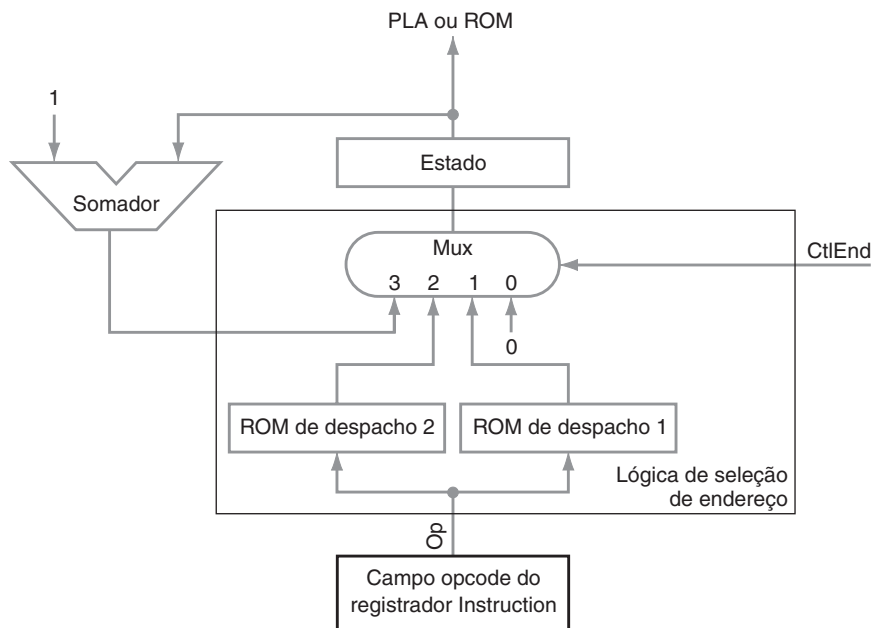
**FIGURA C.4.1 A unidade de controle usando um contador explícito para calcular o próximo estado.**

Nessa unidade de controle, o próximo estado é calculado usando um contador (pelo menos em alguns estados). Por comparação, a Figura C.3.2 codifica o próximo estado na lógica de controle para todos os estados. Nessa unidade de controle, os sinais rotulados como *CtlEnd* controlam como o próximo estado é determinado.

Embora o próximo estado não sequencial venha de uma tabela externa, a unidade de controle precisa especificar quando isso deve ocorrer e como encontrar esse próximo estado. Há dois tipos de “desvio” que precisamos implementar na lógica de seleção de endereço. Primeiro, precisamos ser capazes de saltar para um de vários estados baseado na parte do opcode do registrador Instruction. Essa operação, chamada *despacho*, normalmente é implementada usando um conjunto de ROMs ou PLAs especiais incluídas como parte da lógica de seleção de endereço. Um conjunto adicional de saídas de controle, que chamamos *CtlEnd*, indica quando um despacho deve ser feito. Olhando o diagrama de estados finitos (Figura C.3.1), vemos que existem dois estados em que fazemos um desvio baseado em uma parte do opcode. Portanto, precisaremos de duas pequenas tabelas de despacho. (Como alternativa, também poderíamos usar uma única tabela de despacho e usar os bits de controle que selecionam a tabela como bits de endereço que escolhem de que parte da tabela de despacho selecionar o endereço.)

O segundo tipo de desvio que precisamos implementar consiste em desviar de volta para o estado 0, que inicia a execução da próxima instrução MIPS. Portanto, existem quatro maneiras possíveis de escolher o próximo estado (três tipos de desvios e incrementar o número do estado atual), que podem ser codificadas em 2 bits. Vamos considerar que a codificação é a seguinte:

Valor de CtlEnd	Ação
0	Define o estado em 0
1	Despacha com ROM 1
2	Despacha com ROM 2
3	Usa o estado incrementado



**FIGURA C.4.2** Essa é a lógica de seleção de endereço para a unidade de controle da Figura C.4.1.

Se usarmos essa codificação, a lógica de seleção de endereço para essa unidade de controle pode ser implementada como mostra a Figura C.4.2.

Para completar a unidade de controle, só precisamos especificar o conteúdo das ROMs de despacho e os valores das linhas de controle de endereço para cada estado. Já especificamos a parte do controle do caminho de dados da word de controle usando o conteúdo da ROM da Figura C.3.7 (ou as partes correspondentes da PLA na Figura C.3.9). O contador de próximo estado e as ROMs de despacho substituem a parte da unidade de controle que estava calculando o próximo estado, mostrada na Figura C.3.8. Como estamos apenas implementando uma parte do conjunto de instruções, as ROMs de despacho estarão primordialmente vazias. A Figura C.4.3 mostra as entradas que precisam ser atribuídas para esse subconjunto. A Seção 5.6 do Capítulo 5 discute o que fazer com as entradas nas ROMs de despacho que não correspondem a qualquer instrução.

ROM de despacho 1		
Op	Nome do opcode	Valor
000000	formato R	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

ROM de despacho 2		
Op	Nome do opcode	Valor
100011	lw	0011
101011	sw	0101

**FIGURA C.4.3** Cada uma das ROMs de despacho possui  $2^6 = 64$  entradas com 4 bits de largura, já que esse é o número de bits na codificação de estado. Essa figura mostra as entradas na ROM de interesse para esse subconjunto. A primeira coluna de cada tabela indica o valor de Op, que é o endereço usado para acessar a ROM de despacho. A segunda coluna mostra o nome simbólico do opcode. A terceira coluna indica o valor nesse endereço na ROM.

Agora podemos determinar a definição das linhas de seleção de endereço (CtlEnd) em cada word de controle. A tabela na Figura C.4.4 mostra como o controle de endereço precisa ser definido para cada estado. Essas informações serão usadas para especificar a definição do campo CtlEnd na word de controle associada com esse estado.

Número do estado	Ação do controle de endereço	Valor de CtlEnd
0	Usa o estado incrementado	3
1	Usa a ROM de despacho 1	1
2	Usa a ROM de despacho 2	2
3	Usa o estado incrementado	3
4	Substitui o número do estado por 0	0
5	Substitui o número do estado por 0	0
6	Usa o estado incrementado	3
7	Substitui o número do estado por 0	0
8	Substitui o número do estado por 0	0
9	Substitui o número do estado por 0	0

**FIGURA C.4.4** Os valores das linhas de controle de endereço são definidos na word de controle que corresponde a cada estado.

O conteúdo da ROM de controle inteira é mostrado na Figura C.4.5. O armazenamento total necessário para o controle é muito pequeno. Existem 10 words de controle, cada uma com 18 bits de largura, para um total de 180 bits. Além disso, as duas tabelas de despacho têm 4 bits de largura e cada uma possui 64 entradas, para um total de 512 bits adicionais. Esse total de 692 bits é melhor do que a implementação que usa duas ROMs com a função de próximo estado nas ROMs (que exige 4,3Kbits).

Número do estado	Bits de word de controle 17-2	Bits de word de controle 1-0
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	000000001000100	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

**FIGURA C.4.5** O conteúdo da memória de controle para uma implementação usando um contador explícito. A primeira coluna mostra o estado, a segunda mostra os bits de controle do caminho de dados e a última coluna mostra os bits de controle de endereço em cada word de controle. Os bits 17-2 são idênticos aos da Figura C.3.7.

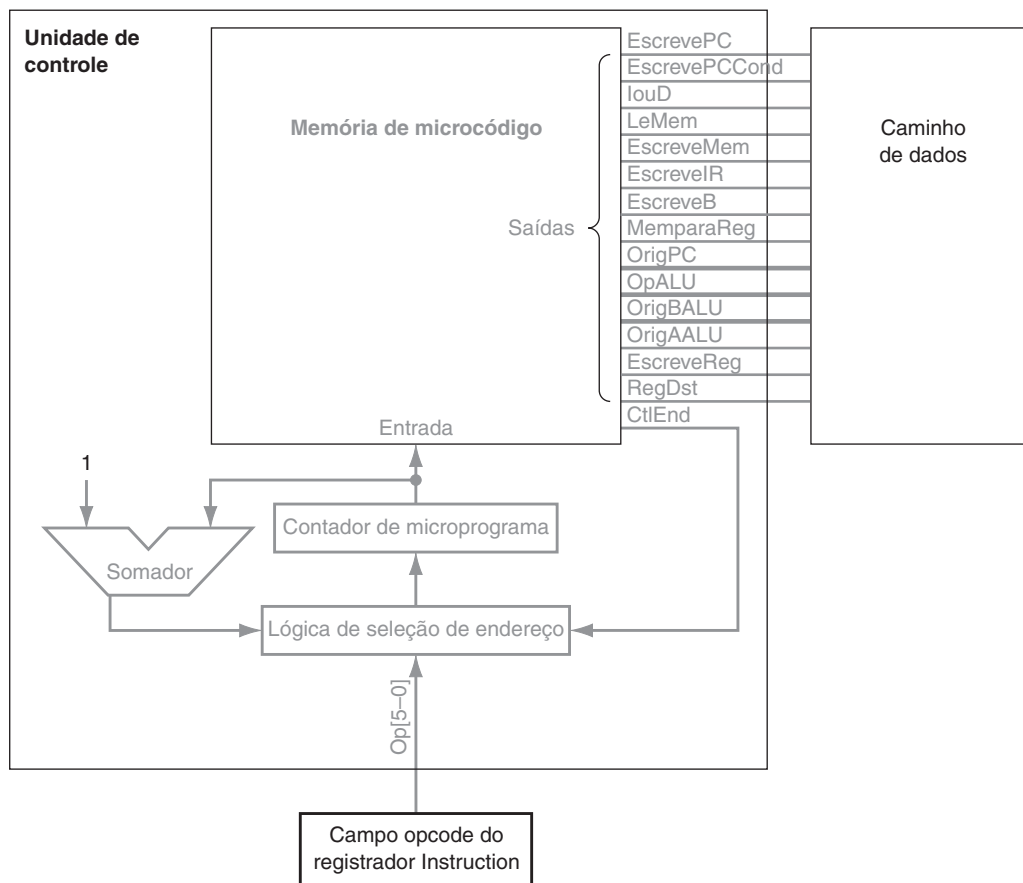
Naturalmente, as tabelas de despacho são esparsas e poderia ser implementada mais eficientemente com duas pequenas PLAs. A ROM de controle também poderia ser substituída por uma PLA.

### Otimizando a implementação do controle

Podemos reduzir mais a quantidade de lógica na unidade de controle por duas técnicas diferentes. A primeira é a *minimização de lógica*, que usa a estrutura das equações lógicas, incluindo os “don’t ca-

res”, para reduzir a quantidade de hardware necessária. O sucesso desse processo depende de quantas entradas existem na tabela verdade e de como estão relacionadas. Por exemplo, nesse subconjunto, apenas os opcodes *lw* e *sw* possuem um valor ativo para o sinal *Op5*; portanto, podemos substituir as duas entradas da tabela verdade que testam se a entrada é *lw* ou *sw* por um teste simples nesse bit; da mesma forma, podemos eliminar vários bits usados para localizar *lw* e *sw* na primeira ROM de despacho. Evidentemente, se o espaço do opcode fosse mais denso, as oportunidades para essa otimização seriam mais difíceis de localizar. Todavia, escolhendo os opcodes, o arquiteto pode fornecer oportunidades adicionais de escolher opcodes relacionados para instruções que provavelmente compartilham estados no controle.

Um tipo diferente de otimização pode ser feito atribuindo os números de estado em uma implementação de estados finitos ou microcódigo para minimizar a lógica. Essa otimização, chamada *atribuição de estados*, tenta escolher os números de estado de modo que as equações lógicas resultantes contêm mais redundância e, assim, possam ser simplificadas. Vamos considerar primeiro o caso de uma máquina de estados finitos com um controle de próximo estado codificado, já que ela permite que os estados sejam atribuídos arbitrariamente. Por exemplo, observe que na máquina de estados finitos o sinal *EscreveReg* está ativo apenas nos estados 4 e 7. Se codificássemos esses estados como 8 e 9, em vez de 4 e 7, poderíamos reescrever a equação para *EscreveReg* simplesmente como um teste no bit *S3* (que só está ligado para os estados 8 e 9). Essa renumeração permite combinar as duas entradas da tabela verdade na parte (o) da Figura C.3.4 e substituí-las por uma única entrada, eliminando um termo na unidade de controle. É claro que teríamos de renumerar os estados existentes 8 e 9, talvez como 4 e 7.



**FIGURA C.4.6 A unidade de controle como um microcódigo.** O uso do termo “micro” serve para distinguir entre o contador de programa no caminho de dados e o contador de microprograma, e entre a memória de microcódigo e a memória de instruções.

A mesma otimização pode ser aplicada em uma implementação que usa um contador de programa explícito, embora estejamos mais restritos. Como o número do próximo estado frequentemente é calculado incrementando o número do estado atual, não podemos atribuir arbitrariamente os estados. No entanto, se mantivermos os estados em que o estado incrementado é usado como o próximo estado na mesma ordem, poderemos reatribuir os estados consecutivos como um bloco. Em uma implementação com um contador de próximo estado explícito, a atribuição de estado pode permitir simplificar o conteúdo das ROMs de despacho.

Se olharmos novamente a unidade de controle na Figura C.4.1, surpreendentemente ela se parece muito com um computador. A ROM ou a PLA pode ser imaginada como instruções buscadas na memória para o caminho de dados. O estado pode ser imaginado como um endereço de instrução. Daí a origem do nome *microcódigo* ou *controle microprogramado*. As words de controle são consideradas como *microinstruções* que controlam o caminho de dados, e o registrador State é chamado de *contador de microprograma*. A Figura C.4.6 mostra uma visão da unidade de controle como um *microcódigo*. A próxima seção descreve como mapear de um microprograma para o microcódigo.

## C.5

### Traduzindo um microprograma para hardware

Para traduzir o microprograma da Seção 5.5 para hardware real, precisamos especificar como cada campo se traduz em sinais de controle. Podemos implementar o microprograma com controle de estados finitos ou uma implementação de microcódigo com um seqüenciador explícito. Se escolhermos uma máquina de estados finitos, precisaremos construir a função de próximo estado a partir do microprograma. Uma vez que essa função seja conhecida, podemos mapear um conjunto de entradas da tabela verdade para as saídas de próximo estado. Nesta seção, veremos como traduzir o microprograma considerando que o próximo estado seja especificado por um seqüenciador. Das tabelas verdade que construiremos, seria simples construir a função de próximo estado para uma máquina de estados finitos.

Considerando um seqüenciador explícito, precisamos realizar duas tarefas adicionais para traduzir o microprograma: atribuir endereços às microinstruções e preencher o conteúdo das ROMs de despacho. Esse processo é igual ao processo de traduzir um programa em assembly para instruções de máquina: Os campos em assembly ou a instrução do microprograma são traduzidos e os rótulos das instruções precisam ser convertidos em endereços.

A Figura C.5.1 mostra os diversos valores para cada campo de microinstrução que controla o caminho de dados e como esses campos são codificados como sinais de controle. Se o campo correspondente a um sinal que afeta uma unidade com estado (ou seja, Controle da Memória, Controle dos Registradores, Controle da ALU ou Controle de escrita no PC) estiver vazio, então, nenhum sinal de controle deve estar ativo. Se um campo correspondente a um sinal de controle de multiplexador ou ao controle de operação da ALU (ou seja, OpALU, SRC1 ou SRC2) estiver vazio, a saída não é utilizada, de modo que os sinais associados podem ser definidos como “don’t cares”.

O campo Seqüenciamento pode ter quatro valores: Fetch (significando ir para o estado Fetch), Dispatch 1, Dispatch 2 e Seq. Esses quatro valores são codificados para definir o controle de endereço de 2 bits exatamente como foram na Figura C.4.4: Fetch = 0, Dispatch 1 = 1, Dispatch 2 = 2, Seq = 3. Finalmente, precisamos especificar o conteúdo das tabelas de despacho para relacionarem as entradas de despacho do campo Seqüenciamento aos rótulos simbólicos no microprograma. Usamos as mesmas tabelas de despacho que usamos anteriormente na Figura C.4.3.

Um montador de microcódigo usaria a codificação do campo Seqüenciamento, o conteúdo das tabelas de despacho simbólicas na Figura C.5.2, a especificação na Figura C.5.1 e o microprograma real na Figura 5.7.3 para gerar as microinstruções.

Nome do campo	Valor	Sinais ativos	Comentário
Controle da ALU	Add	OpALU = 00	Faz com que a ALU realize uma soma.
	Subt	OpALU = 01	Faz com que a ALU realize uma subtração; isso implementa a comparação para desvios.
	Func code	OpALU = 10	Usa o código de função da instrução para determinar o controle da ALU.
SRC1	PC	OrigAALU = 0	Usa o PC como a primeira entrada da ALU.
	A	OrigAALU = 1	O registrador A é a primeira entrada da ALU.
SRC2	B	OrigBALU = 00	O registrador B é a segunda entrada da ALU.
	4	OrigBALU = 01	Usa 4 como a segunda entrada da ALU.
	Extend	OrigBALU = 10	Usa a saída da unidade de extensão de sinal como a segunda entrada da ALU.
	Extshft	OrigBALU = 11	Usa a saída da unidade de deslocamento em dois bits como a segunda entrada da ALU.
Controle dos Registradores	Read		Lê dois registradores usando os campos rs e rt do IR como os números de registrador e colocando os dados nos registradores A e B.
	Write ALU	EscreveReg, RegDst = 1, MemparaReg = 0	Escreve num registrador usando o campo rd do IR como o número de registrador e o conteúdo de SaídaALU como os dados.
	Write MDR	EscreveReg, RegDst = 0, MemparaReg = 1	Escreve num registrador usando o campo rt do IR como o número de registrador e o conteúdo de MDR como os dados.
Controle da Memória	Read PC	LeMem, louD = 0, EscreveIR	Lê a memória usando o PC como o endereço; escreve o resultado no IR (e no MDR).
	Read ALU	LeMem, louD = 1	Lê a memória usando SaídaALU como o endereço; escreve o resultado no MDR.
	Write ALU	EscreveMem, louD = 1	Escreve na memória usando SaídaALU como o endereço e o conteúdo de B como os dados.
Controle de Escrita no PC	ALU	OrigPC = 00, EscrevePC	Escreve a saída da ALU no PC.
	ALUOut-cond	OrigPC = 01, EscrevePCCond	Se a saída Zero da ALU estiver ativa, escreve o PC com o conteúdo do registrador SaídaALU.
	jump address	OrigPC = 10, EscrevePC	Escreve o PC com o endereço de jump da instrução.
Seqüenciamento	Seq	CtlEnd = 11	Escolhe a próxima microinstrução seqüencialmente.
	Fetch	CtlEnd = 00	Vai para a primeira microinstrução para iniciar uma nova instrução.
	Dispatch 1	CtlEnd = 01	Despacha usando a ROM 1.
	Dispatch 2	CtlEnd = 10	Despacha usando a ROM 2.

**FIGURA C.5.1 Cada campo de microcódigo é traduzido para um conjunto dos sinais de controle a serem definidos.**

Essa tabela especifica um valor para cada um dos campos que foram especificados originalmente na Figura 5.7.3. Esses 22 valores diferentes dos campos especificam todas as combinações necessárias das 18 linhas de controle. As linhas de controle não definidas que correspondem a ações são 0 por padrão. As linhas de controle de multiplexador são definidas como 0 se a saída importa. Se uma linha de controle de multiplexador não estiver explicitamente definida, sua saída é “don’t care” e não é usada.

Como o microprograma é uma representação abstrata do controle, há uma grande flexibilidade em como o microprograma é traduzido. Por exemplo, a entrada atribuída a muitas microinstruções pode ser escolhida arbitrariamente; as únicas restrições são as impostas pelo fato de que certas microinstruções precisam ocorrer em ordem seqüencial (de modo que incrementar o registrador State gere o endereço da próxima instrução). Portanto, o montador de microcódigo pode reduzir a complexidade do controle atribuindo as microinstruções inteligentemente.

Tabela de despacho de microcódigo 1		
Campo opcode	Nome do opcode	Valor
000000	formato R	R-format1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

Tabela de despacho de microcódigo 2		
Campo opcode	Nome do opcode	Valor
100011	lw	LW2
101011	sw	SW2

**FIGURA C.5.2** As duas ROMs de despacho de microcódigo mostrando o conteúdo em forma simbólica usando as tabelas no microprograma.

### Organizando o controle para reduzir a lógica

Para uma máquina com controle complexo, pode haver uma grande quantidade de lógica na unidade de controle. A ROM ou PLA de controle podem ser bastante dispendiosas. Embora nossa implementação simples tivesse apenas uma microinstrução de 18 bits (considerando um seqüenciador explícito), existiram máquinas com microinstruções de centenas de bits de largura. Evidentemente, um projetista gostaria de reduzir o número de microinstruções e a largura.

O método ideal para reduzir o armazenamento de controle é primeiro escrever o microprograma completo em uma notação simbólica e, depois, medir como as linhas de controle estão definidas em cada microinstrução. Fazendo medições, podemos reconhecer os bits de controle que podem ser codificados em um campo menor. Por exemplo, se não mais que uma de oito linhas é definida simultaneamente na mesma microinstrução, esse subconjunto de linhas de controle pode ser codificado em um campo de 3 bits ( $\log_2 8 = 3$ ). Essa mudança economiza 5 bits em cada microinstrução e não prejudica o CPI, embora signifique o custo de hardware extra de um decodificador de 3 para 8 necessário para gerar as oito linhas de controle quando forem exigidas no caminho de dados. Isso também pode ter um pequeno impacto de ciclo de clock, já que o decodificador está no caminho do sinal. Entretanto, tirar 5 bits da largura do armazenamento de controle normalmente irá superar o custo do decodificador, e o impacto do tempo de ciclo provavelmente será pequeno ou inexistente. Por exemplo, essa técnica pode ser aplicada nos bits 13-6 das microinstruções nessa máquina, já que apenas 1 bit dos 7 bits da word de controle está sempre ativo (veja a Figura C.4.5).

Essa técnica de reduzir a largura de campo é chamada de *codificação*. Para ganhar ainda mais espaço, as linhas de controle podem ser codificadas juntas se forem definidas na mesma microinstrução apenas ocasionalmente; duas microinstruções em vez de uma são então necessárias quando as duas precisarem ser definidas. Como isso não acontece em rotinas críticas, a microinstrução mais curta pode justificar algumas words extras do armazenamento de controle.

As microinstruções podem se tornar mais curtas ainda se forem divididas em diferentes formatos e receberem um campo opcode ou *format* para distingui-las. O campo format fornece os valores padrão a todas as linhas de controle não especificadas, a fim de não mudar nada mais na máquina, e é semelhante ao opcode de uma instrução em um conjunto de instruções mais capaz. Por exemplo, para microinstruções que realizassem acessos à memória, poderíamos usar um formato diferente daqueles que realizassem operações da ALU registrador para registrador, tirando vantagem do fato de que as linhas de controle de acesso à memória não são necessárias em microinstruções controlando operações da ALU.

A redução dos custos de hardware usando campos de formato normalmente tem um custo de desempenho adicional além do necessário para mais decodificadores. Um microprograma usando um único formato de microinstrução pode especificar qualquer combinação de operações em um cami-



nho de dados e pode exigir menos ciclos de clock do que um microprograma composto de microinstruções restritas que não podem realizar qualquer combinação de operações em uma única microinstrução. Entretanto, se a capacidade total da word de microprograma mais larga não for usada intensamente, muito do armazenamento de controle será desperdiçado; além disso, a máquina pode se tornar menor e mais rápida restringindo a capacidade de microinstrução.

O método mais estreito, mas em geral mais longo, é chamado de *microcódigo vertical*, enquanto o método largo mas curto é chamado de *microcódigo horizontal*. Devemos salientar que os termos “microcódigo vertical” e “microcódigo horizontal” não possuem uma definição universal – os projetistas do 8086 consideravam sua microinstrução de 21 bits mais horizontal do que outros computadores de chip único da época. Os termos relacionados *maximamente codificado* e *minimamente codificado* provavelmente são melhores do que “vertical” e “horizontal”.

## C.6

### Comentários finais

Iniciamos este apêndice vendo como traduzir um diagrama de estados finitos para uma implementação usando uma máquina de estados finitos. Em seguida, examinamos os seqüenciadores explícitos que usam uma técnica diferente para realizar a função de próximo estado. Embora grandes microprogramas normalmente sejam destinados a implementações usando esse método de próximo estado explícito, também podemos implementar um microprograma com uma máquina de estados finitos. Como vimos, as implementações em ROM e PLA das funções lógicas são possíveis. As vantagens do próximo estado explícito *versus* codificado e da implementação em ROM *versus* PLA são resumidas abaixo.

### Colocando em perspectiva

Quer o controle seja representado como um diagrama de estados finitos ou como um microprograma, a tradução para uma implementação de controle de hardware é semelhante. Cada estado ou microinstrução ativa um conjunto de saídas de controle e especifica como escolher o próximo estado.


A função de próximo estado pode ser implementada codificando-a em uma máquina de estados finitos ou usando um seqüenciador explícito. O seqüenciador explícito é mais eficiente se o número de estados for grande e houver muitas seqüências de estados consecutivos sem desvio.

A lógica de controle pode ser implementada com ROMs ou PLAs (ou mesmo um mix). As PLAs são mais eficientes, a menos que a função de controle seja muito densa. As ROMs podem ser apropriadas se o controle for armazenado em uma memória separada, e não dentro do mesmo chip que o caminho de dados.


## C.7


### Exercícios


**C.1** [10] <§C.2> Em vez de usar 4 bits de estado para implementar a máquina de estados finitos da Figura C.3.1, use 9 bits de estado, cada um dos quais sendo um 1 apenas se a máquina de estados finitos estiver nesse estado específico (por exemplo, S1 é 1 no estado 1, S2 é 1 no estado 2 etc.). Redesenhe a PLA (Figura C.3.9).


**C.2** [5] <§C.3> Quantos termos de produto são necessários em uma PLA que implementa o caminho de dados de ciclo único para ja1, considerando as adições de controle descritas no Exercício 5.20 em  **Aprofundando o aprendizado?**



**C.3** [5] <§C.3> Quantos termos de produto são necessários em uma PLA que implemente o caminho de dados de ciclo único e controle para addi, considerando que as adições de controle que você precisasse fossem encontradas no Exercício 5.19 em  **Aprofundando o aprendizado?**

**C.4** [10] <§C.3> Determine o número de termos de produto em uma PLA que implemente a máquina de estados finitos para jal construída no Exercício 5.55 em  **Aprofundando o aprendizado?** A maneira mais fácil de fazer isso é construir as tabelas verdade para qualquer nova saída ou qualquer saída afetada pela adição.

**C.5** [10] <§C.3> Determine o número de termos de produto em uma PLA que implemente a máquina de estados finitos para addi no Exercício 5.19 em  **Aprofundando o aprendizado?** A maneira mais fácil de fazer isso é construir as adições às tabelas verdade para addi.

**C.6** [20] <§C.4> Implemente a máquina de estados finitos do Exercício 5.19 na seção  **Aprofundando o aprendizado** usando um contador explícito para determinar o próximo estado. Preencha as novas entradas para as adições à Figura C.4.5. Além disso, acrescente quaisquer entradas necessárias às ROMs de despacho da Figura C.5.2.

**C.7** [15] <§§C.3–C.6> Determine o tamanho das PLAs necessárias para implementar a máquina multiciclo do Capítulo 5 considerando que a função de próximo estado é implementada com um contador. Implemente as tabelas de despacho da Figura C.5.2 usando duas PLAs, e o conteúdo da unidade de controle principal na Figura C.4.5 usando outra PLA. Como o tamanho total dessa solução se compara com a solução de uma única PLA com o próximo estado codificado? E se as PLAs principais para os dois métodos forem divididas em duas PLAs separadas decompondo o próximo estado ou os sinais de seleção de endereço?