C

A P Ê N D I C E

Fundamentos do Projeto Lógico

Eu sempre gostei dessa palavra: Booleana.

Claude Shannon
IEEE Spectrum, abril *de 1992*
(A tese do professor Shannon mostrou que a álgebra inventada por George Boole no século XIX poderia representar o funcionamento das chaves elétricas.)

C.1 Introdução

Este apêndice oferece uma rápida discussão sobre os fundamentos do projeto lógico. Ele não substitui um curso sobre projeto lógico nem permitirá projetar sistemas lógicos funcionais significativos. Contudo, se você tiver pouca ou nenhuma experiência com projeto lógico, este apêndice oferecerá uma base suficiente para entender todo o material deste livro. Além disso, se você quiser entender parte da motivação por trás da forma como os computadores são implementados, este material servirá como uma introdução útil. Se a sua curiosidade for aumentada, mas não saciada com este apêndice, as referências ao final oferecem fontes de informação adicionais.

A Seção C.2 introduz os blocos de montagem básicos da lógica, a saber, *portas lógicas*. A Seção C.3 utiliza esses blocos de montagem para construir sistemas lógicos *combinacionais* simples, que não contêm memória. Se você já teve alguma experiência com sistemas lógicos ou digitais, provavelmente estará acostumado com o material dessas duas primeiras seções. A Seção C.5 mostra como usar os conceitos das Seções C.2 e C.3 para projetar uma ALU para o processador MIPS. A Seção C.6 mostra como criar um somador rápido e pode ser pulada sem problemas se você não estiver interessado nesse assunto. A Seção C.7 é uma introdução rápida ao assunto de clocking, necessário para discutirmos como funcionam os elementos de memória. A Seção C.8 introduz os elementos de memória, e a Seção C.9 a estende para focalizar memórias de acesso aleatório; ele descreve as características importantes de entender e como elas são usadas nos Capítulos 5 e 6, e a base que motiva muitos dos aspectos do projeto de hierarquia de memória no Capítulo 7. A Seção C.10 descreve o projeto e o uso das máquinas de estados finitos, que são blocos lógicos sequenciais. Se você pretende ler o Apêndice D, deverá entender totalmente o material das Seções de C.2 a C.10. No entanto, se você pretende ler apenas o material sobre controle, no Capítulo 4, poderá passar superficialmente pelos apêndices, mas deverá ter alguma familiaridade com todo o material, exceto a Seção C.11. A Seção C.11 serve para aqueles que desejam ter um conhecimento mais profundo das metodologias de clocking e temporização. Ele explica os fundamentos de como funciona o clock acionado por transição, introduz outro esquema de clocking e descreve rapidamente o problema de sincronizar entradas assíncronas.

Ao longo do apêndice, onde for apropriado, também incluímos segmentos em Verilog para demonstrar como a lógica pode ser representada em Verilog, que apresentaremos na Seção C.4. Um tutorial mais extenso e completo sobre Verilog pode ser encontrado em outra parte do CD.

C.2 Portas, tabelas verdade e equações lógicas

A eletrônica dentro de um computador moderno é *digital* e opera apenas com dois níveis de voltagem: alta e baixa. Todos os outros valores de voltagem são temporários e ocorrem na transição entre os valores. (Conforme discutimos mais adiante nesta seção, uma armadilha possível no projeto digital é a amostragem de um sinal quando ele não é nitidamente alto ou baixo.) O fato de que os computadores são digitais também é um motivo fundamental para eles usarem números binários, pois um sistema binário corresponde à abstração básica inerente à eletrônica. Em diversas famílias lógicas, os valores e os relacionamentos entre os dois valores de voltagem diferem. Assim, em vez de se referir aos níveis de voltagem, falamos sobre sinais que são (logicamente) verdadeiros, ou são 1, ou são **ativos**; ou sinais que são (logicamente) falsos, ou 0, ou **inativos**. Os valores 0 e 1 são chamados *complementos* ou *inversos* um do outro.

sinal ativo Um sinal que é (logicamente) verdadeiro, ou 1.

sinal inativo Um sinal que é (logicamente) falso, ou 0.

Os blocos lógicos são categorizados como um dentre dois tipos, dependendo se contêm memória ou não. Os blocos sem memória são chamados *combinacionais*; a saída de um bloco combinacional depende apenas da entrada atual. Nos blocos com memória, as saídas podem depender das entradas e do valor armazenado na memória, chamado *estado* do bloco lógico. Nesta seção e na seguinte, focalizaremos apenas a **lógica combinacional**. Depois de introduzir diferentes elementos de memória na Seção C.8, descreveremos como é projetada a **lógica sequencial**, que é a lógica que inclui estado.

lógica combinacional Um sistema lógico cujos blocos não contêm memória e, portanto, calculam a mesma saída dada à mesma entrada.

lógica sequencial Um grupo de elementos lógicos que contêm memória e, portanto, cujo valor depende das entradas e também do conteúdo atual da memória.

Tabelas verdade

Como um bloco lógico combinacional não contém memória, ele pode ser especificado completamente definindo os valores das saídas para cada conjunto de valores de entrada possíveis. Essa descrição normalmente é dada como uma *tabela verdade*. Para um bloco lógico com n entradas, existem 2^n entradas na tabela verdade, pois existem todas essas combinações possíveis de valores de entrada. Cada entrada especifica o valor de todas as saídas para essa combinação de entrada em particular.

EXEMPLO

Tabelas verdade

Considere uma função lógica com três entradas, *A*, *B* e *C*, e três saídas, *D*, *E* e *F*. A função é definida da seguinte maneira: *D* é verdadeiro se pelo menos uma entrada for verdadeira, *E* é verdadeiro se exatamente duas entradas forem verdadeiras, e *F* é verdadeiro somente se todas as três entradas forem verdadeiras. Mostre a tabela verdade para essa função.

RESPOSTA

A tabela verdade terá $2^n = 8$ entradas. Aqui está ela:

Entradas			Saídas		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

As tabelas verdade podem descrever qualquer função lógica combinacional; porém, elas aumentam de tamanho rapidamente e podem não ser fáceis de entender. Às vezes, queremos construir uma função lógica que será 0 para muitas combinações de entrada e usamos um atalho para especificar apenas as entradas da tabela verdade para as saídas diferentes de zero. Essa técnica é usada no Capítulo 4 e no Apêndice D.

Álgebra Booleana

Outra técnica é expressar a função lógica com equações lógicas. Isso é feito com o uso da *álgebra Booleana* (que tem o nome de Boole, um matemático do século XIX). Em álgebra Booleana, todas as variáveis possuem os valores 0 ou 1 e, nas formulações típicas, existem três operadores:

- O operador OR é escrito como +, como em $A + C$. O resultado de um operador OR é 1 se uma das variáveis for 1. A operação OR também é chamada *soma lógica*, pois seu resultado é 1 se um dos operandos for 1.
- O operador AND é escrito como ., como em $A \,.\, C$. O resultado de um operador AND é 1 somente se as duas entradas forem 1. A operação AND também é chamada de *produto lógico*, pois seu resultado é 1 apenas se os dois operandos forem 1.
- O operador unário NOT é escrito como $\overline{A}$. O resultado de um operador NOT é 1 somente se a entrada for 0. A aplicação do operador NOT a um valor lógico resulta em uma inversão ou negação do valor (ou seja, se a entrada for 0, a saída será 1, e vice-versa).

Existem sete leis da álgebra Booleana úteis na manipulação de equações lógicas:

- Lei da identidade: $A + 0 = A$ e $A \cdot 1 = A$.
- Leis de zero e um: $A + 1 = 1$ e $A \,.\, 0 = 0$.
- Leis inversas: $A + \overline{A} = 1$ e $A \cdot \overline{A} = 0$.
- Leis comutativas: $A + B = B + A$ e $A \cdot B = B \cdot A$.
- Leis associativas: $A + (B + C) = (A + B) + C$ e $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.
- Leis distributivas: $A \,.\, (B + C) = (A \cdot B) + (A \cdot C)$ e $A + (B \cdot C) = (A + B) \cdot (A + C)$.

Além disso, existem dois outros teoremas úteis, chamados leis de DeMorgan, que são discutidos com mais profundidade nos exercícios.

Qualquer conjunto de funções lógicas pode ser escrito como uma série de equações com uma saída no lado esquerdo de cada equação e, no lado direito, uma fórmula consistindo em variáveis e os três operadores anteriores.

Equações lógicas

EXEMPLO

Mostre as equações lógicas para as funções lógicas, *D, E* e *F,* descritas no exemplo anterior.

RESPOSTA

Aqui está a equação para *D:*

$$D = A + B + C$$

F é igualmente simples:

$$F = A \cdot B \cdot C$$

E é um pouco complicada. Pense nela em duas partes: o que precisa ser verdadeiro para *E* ser verdadeiro (duas das três entradas precisam ser verdadeiras) e o que não pode ser verdadeiro (todas as três não podem ser verdadeiras). Assim, podemos escrever *E* como

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

Também podemos derivar *E* observando que *E* é verdadeiro apenas se exatamente duas das entradas forem verdadeiras. Então, podemos escrever *E* como um OR dos três termos possíveis que possuem duas entradas verdadeiras e uma entrada falsa:

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

A prova de que essas duas expressões são equivalentes é explorada nos exercícios.

Em Verilog, descrevemos a lógica combinacional sempre que possível usando a instrução assign, descrita a partir da página C-13. Podemos escrever uma definição para *E* usando o operador OR exclusivo da Verilog, como em `assign E = (A ^ B ^ C) * (A + B + C) * (A * B * C)`, que é outra maneira de descrever essa função. *D* e *F* possuem representações ainda mais simples, que são exatamente como o código C correspondente: `D = A | B | C` e `F = A & B & C`.

Portas lógicas

porta lógica Um dispositivo que implementa funções lógicas básicas, como AND ou OR.

Blocos lógicos são criados a partir de **portas lógicas** que implementam as funções lógicas básicas. Por exemplo, uma porta AND implementa a função AND, e uma porta OR implementa a função OR. Como AND e OR são comutativos e associativos, uma porta AND ou OR pode ter várias entradas, com a saída igual ao AND ou OR de todas as entradas. A função lógica NOT é implementada com um inversor que sempre possui uma única entrada. A representação padrão desses três blocos de montagem lógicos aparece na Figura C.2.1.

Em vez de desenhar inversores explicitamente, uma prática comum é acrescentar "bolhas" às entradas ou saída de uma porta lógica para fazer com que o valor lógico nessa linha de entrada ou de saída seja invertida. Por exemplo, a Figura C.2.2 mostra o diagrama lógico para a função, usando inversores explícitos à esquerda e usando as entradas e a saída em bolha à direita.

Qualquer função lógica pode ser construída usando portas AND, portas OR e inversão; vários dos exercícios dão a oportunidade de tentar implementar algumas funções lógicas comuns com portas. Na próxima seção, veremos como uma implementação de qualquer função lógica pode ser construída usando esse conhecimento.

Porta NOR Uma porta OR invertida.

Porta NAND Uma porta AND invertida.

De fato, todas as funções lógicas podem ser construídas com apenas um único tipo de porta, se essa porta for inversora. As duas portas inversoras são chamadas **NOR** e **NAND** e correspondem às portas OR e AND invertidas, respectivamente. Portas NOR e NAND são chamadas *universais*, pois qualquer função lógica pode ser construída por meio desse tipo de porta. Os exercícios exploram melhor esse conceito.

Verifique você mesmo

As duas expressões lógicas a seguir são equivalentes? Se não, encontre valores para as variáveis para mostrar que não são:

- $(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$
- $B \cdot (A \cdot \overline{C} + C \cdot \overline{A})$

FIGURE C.2.1 Padrão de desenho de uma porta AND, uma porta OR e um inversor, mostrados da esquerda pra direita. Os sinais à esquerda de cada símbolo são entradas, enquanto as saídas aparecem à direita. As portas AND e OR têm duas entradas cada. O inversor tem uma única entrada.

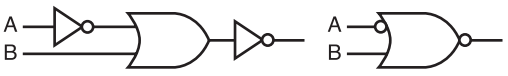


FIGURE C.2.2 Implementação de portas lógicas de $\overline{\overline{A} + B}$ usando inversores explícitos à esquerda e usando as entradas e a saída em bolha à direita. Esta função lógica pode ser simplificada para $A \cdot \overline{B}$ ou em Verilog, A & ~ B.

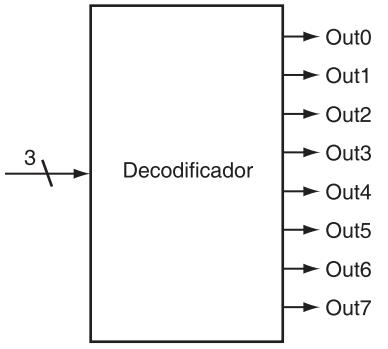
C.3 Lógica combinacional

Nesta seção, examinamos alguns dos maiores blocos de montagem lógicos mais utilizados e discutimos o projeto da lógica estruturada que pode ser implementado automaticamente a partir de uma equação lógica ou tabela verdade por um programa de tradução. Por fim, discutimos a noção de um array de blocos lógicos.

Decodificadores

Um bloco lógico que usaremos na montagem de componentes maiores é um **decodificador**. O tipo mais comum de decodificador possui uma entrada de n bits e 2^n saídas, onde somente uma saída é ativada para cada combinação de entradas. Esse decodificador traduz a entrada de n bits para um sinal que corresponde ao valor binário da entrada de n bits. As saídas, portanto, são numeradas como, digamos, Out0, Out1, ..., Out2^n-1. Se o valor da entrada for i, então Out$i$ será verdadeiro e todas as outras saídas serão falsas. A Figura C.3.1 mostra um decodificador de 3 bits e a tabela verdade. Esse decodificador é chamado *decodificador 3 para 8*, pois existem 3 entradas e 8 (2^3) saídas. Há também um elemento lógico chamado de *codificador*, que realiza a função inversa de um decodificador, exigindo 2^n entradas e produzindo uma saída de n bits.

decodificador Um bloco lógico que possui uma entrada de n bits e 2^n saídas, onde somente uma saída é ativada para cada combinação de entradas.



a.Um decodificador 3-bit

Entradas			Saídas							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. A tabela verdade para um decodificador 3-bit

FIGURA C.3.1 Decodificador de 3 bits possui 3 entradas, chamadas 12, 11 e 10, e 8 (23) saídas, chamadas de Out0 a Out7. Somente a saída correspondente ao valor binário da entrada é verdadeira, como mostra a tabela verdade. O rótulo 3 na entrada do decodificador diz que o sinal de entrada possui 3 bits de largura.

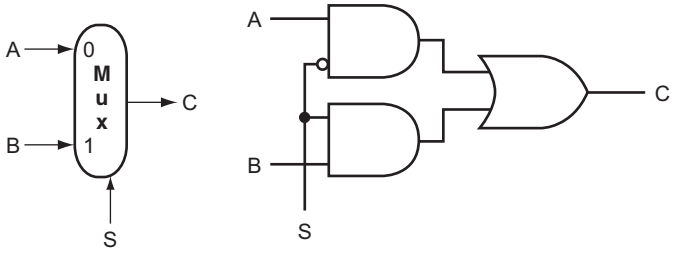


FIGURA C.3.2 Um multiplexador de duas entradas, à esquerda, e sua implementação com portas lógicas, à direita. O multiplexador tem duas entradas de dados (A e B), que são rotuladas com *0* e *1*, e uma entrada seletora (S), além de uma saída C. A implementação de multiplexadores em Verilog exige um pouco mais de trabalho, especialmente quando eles possuem mais de duas entradas. Mostramos como fazer isso a partir da página C.13.

Multiplexadores

valor seletor Também chamado **valor de controle**. O sinal de controle usado para selecionar um dos valores de entrada de um multiplexador como a saída do multiplexador.

Uma função lógica básica que usamos com muita frequência nos Capítulos 5 e 6 é o *multiplexador*. Um multiplexador poderia ser mais corretamente chamado de *seletor*, pois sua saída é uma das entradas selecionada por um controle. Considere o multiplexador de duas entradas. O lado esquerdo da Figura C.3.2 mostra que esse multiplexador tem três entradas: dois valores de dados e um **valor seletor** (ou de **controle**). O valor seletor determina qual das entradas se torna a saída. Podemos representar a função lógica calculada por um multiplexador de duas entradas, mostrado em forma de portas lógicas no lado direito da Figura C.3.2, como $C = (A \,.\, S) + (B \,.\, S)$.

Os multiplexadores podem ser criados com qualquer quantidade de entradas de dados. Quando existem apenas duas entradas, o seletor é um único sinal que seleciona uma das entradas se ela for verdadeira (1) e a outra se ela for falsa (0). Se houver n entradas de dados, terá de haver $\log_2 n$ entradas seletoras. Nesse caso, o multiplexador basicamente consiste em três partes:

1. Um decodificador que gera n sinais, cada um indicando um valor de entrada diferente.
2. Um array de n portas AND, cada uma combinando com uma das entradas com um sinal do decodificador.
3. Uma única porta OR grande, que incorpora as saídas das portas AND.

Para associar as entradas com valores do seletor, rotulamos as entradas de dados numericamente (ou seja, 0, 1, 2, 3, ..., $n - 1$) e interpretamos as entradas do seletor de dados como um número binário. Às vezes, utilizamos um multiplexador com sinais de seletor não decodificados.

Os multiplexadores são representados combinacionalmente em Verilog usando expressões *if*. Para multiplexadores maiores, instruções *case* são mais convenientes, mas deve-se ter cuidado ao sintetizar a lógica combinacional.

Lógica de dois níveis e PLAs

soma de produtos Uma forma de representação lógica que emprega uma soma lógica (OR) de produtos (termos unidos usando o operador AND).

Conforme indicado na seção anterior, qualquer função lógica pode ser implementada apenas com as funções AND, OR e NOT. Na verdade, um resultado muito mais forte é verdadeiro. Qualquer função lógica pode ser escrita em um formato canônico, no qual cada entrada é verdadeira ou uma variável complementada e existem apenas dois níveis de portas – um sendo AND e o outro OR –, com uma possível inversão na saída final. Essa representação é chamada de *representação de dois níveis* e existem duas formas, chamadas **soma de produtos** e *produto de somas*. Uma representação da soma de produtos é uma soma lógica (OR) de produtos (termos usando o operador AND); um produto de somas é exatamente o oposto. Em nosso exemplo anterior, tínhamos duas equações para a saída *E:*

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

e

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

Esta segunda equação está na forma de soma de produtos: ela possui dois níveis de lógica e as únicas inversões estão em variáveis individuais. A primeira equação possui três níveis de lógica.

Detalhamento: também podemos escrever *E* como um produto de somas:

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

Para derivar esse formato, você precisa usar os *teoremas de DeMorgan*, discutidos nos exercícios.

Neste texto, usamos o formato da soma de produtos. É fácil ver que qualquer função lógica pode ser representada como uma soma de produtos, construindo tal representação a partir da tabela verdade para a função. Cada entrada da tabela verdade para a qual a função é verdadeira corresponde a um termo do produto. O termo do produto consiste em um produto lógico de todas as entradas ou os complementos das entradas, dependendo se a entrada na tabela verdade possui um 0 ou 1 correspondente a essa variável. A função lógica é a soma lógica dos termos do produto onde a função é verdadeira. Isso pode ser visto mais facilmente com um exemplo.

EXEMPLO

Soma de produtos

Mostre a representação da soma dos produtos para a seguinte tabela verdade para *D*.

Entradas			Saída
A	***B***	***C***	***D***
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

RESPOSTA

Existem quatro termos no produto, pois a função é verdadeira (1) para quatro combinações de entrada diferentes. São estes

$$\overline{A} \cdot \overline{B} \cdot C$$
$$\overline{A} \cdot B \cdot \overline{C}$$
$$A \cdot \overline{B} \cdot \overline{C}$$
$$A \cdot B \cdot C$$

Assim, podemos escrever a função para *D* como a soma destes termos:

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

Observe que somente as entradas da tabela verdade para as quais a função é verdadeira geram os termos na equação.

Podemos usar esse relacionamento entre uma tabela verdade e uma representação bidimensional para gerar uma implementação no nível de portas lógicas de qualquer conjunto de funções lógicas. Um conjunto de funções lógicas corresponde a uma tabela verdade com várias colunas de saída, como vimos no exemplo da C-4. Cada coluna de saída representa uma função lógica diferente, que pode ser construída diretamente a partir da tabela verdade.

A representação da soma dos produtos corresponde a uma implementação lógica estruturada comum, chamada **array lógico programável** (**PLA – Programmable Logic Array**). Uma PLA possui um conjunto de entradas e complementos de entrada correspondentes (que podem ser implementados com um conjunto de inversores) e dois estágios de lógica. O primeiro estágio é um array de portas AND que formam um conjunto de **termos do produto** (às vezes chamados **mintermos**); cada termo do produto pode consistir em qualquer uma das entradas ou seus complementos. O segundo estágio é um array de portas OR, cada uma das quais formando uma soma lógica de qualquer quantidade de termos do produto. A Figura C.3.3 mostra a forma básica de uma PLA.

array lógico programável (PLA) Um elemento lógico estruturado composto de um conjunto de entradas e complementos de entrada correspondentes e dois estágios de lógica: o primeiro gerando termos do produto das entradas e complementos da entrada e o segundo gerando termos da soma dos termos do produto. Logo, PLAs implementam funções lógicas como uma soma de produtos.

mintermos Também chamados **termos do produto**. Um conjunto de entradas lógicas unidas por conjunção (operações AND); os termos do produto formam o primeiro estágio lógico do array lógico programável (PLA).

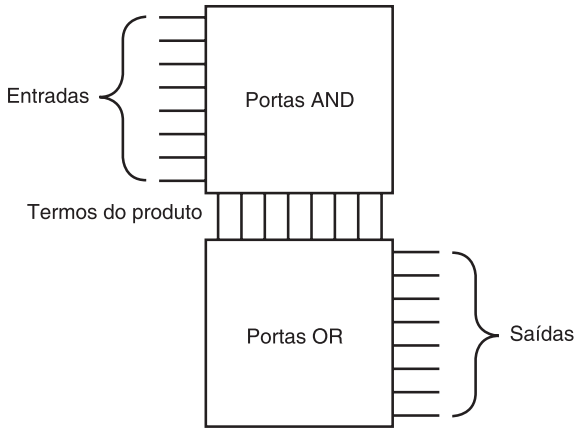


FIGURA C.3.3 O formato básico de uma PLA consiste em um array de portas AND seguido por um array de portas OR. Cada entrada no array de portas AND é um termo do produto consistindo em qualquer quantidade de entradas ou entradas invertidas. Cada entrada no array de portas OR é um termo da soma consistindo em qualquer quantidade desses termos do produto.

Uma PLA pode implementar diretamente a tabela verdade de um conjunto de funções lógicas com várias entradas e saídas. Como cada entrada onde a tabela verdade é verdadeira exige um termo do produto, haverá uma linha correspondente na PLA. Cada saída corresponde a uma linha em potencial das portas OR no segundo estágio. O número de portas OR corresponde ao número de entradas da tabela verdade para as quais a saída é verdadeira. O tamanho total de uma PLA, como aquela mostrada na Figura C.3.3, é igual à soma do tamanho do array de portas AND (chamado *plano AND*) e o tamanho do array de portas OR (chamado *plano OR*). Examinando a Figura C.3.3, podemos ver que o tamanho do array de portas AND é igual ao número de entradas vezes o número de termos do produto diferentes, e o tamanho do array de portas OR é o número de saídas vezes o número de termos do produto.

Uma PLA possui duas características que a ajudam a se tornar um meio eficiente de implementar um conjunto de funções lógicas. Primeiro, somente as entradas da tabela verdade que produzem um valor verdadeiro para pelo menos uma saída possuem quaisquer portas lógicas associadas a elas. Segundo, cada termo do produto diferente terá apenas uma entrada na PLA, mesmo que o termo do produto seja usado em várias saídas. Vamos examinar um exemplo.

EXEMPLO

PLAs

Considere o conjunto de funções lógicas definido no exemplo da página C-4. Mostre uma implementação em PLA desse exemplo para *D*, *E* e *F*.

RESPOSTA

Aqui está a tabela verdade construída anteriormente:

Entradas			Saídas		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Como existem sete termos do produto exclusivos com pelo menos um valor verdadeiro na seção de saída, haverá sete colunas no plano AND. O número de linhas no plano AND é três (pois existem três entradas) e também haverá três linhas no plano OR (pois existem três saídas). A Figura C.3.4 mostra a PLA resultante, com os termos do produto correspondendo às entradas da tabela verdade de cima para baixo.

Em vez de desenhar todas as portas, como fizemos na Figura C.3.4, os projetistas normalmente mostram apenas a posição das portas AND ou das portas OR. Os pontos são usados na interseção de uma linha de sinal do termo do produto e uma linha de entrada ou uma linha de saída quando uma porta AND ou OR correspondente é exigida. A Figura C.3.5 mostra como a PLA da Figura C.3.4 ficaria quando desenhada dessa maneira. O conteúdo de uma PLA é fixo quando a PLA é criado, embora também existam formas de estruturas tipo PLA, chamadas *PALs*, que podem ser programadas eletronicamente quando um projetista está pronto para usá-las.

ROMs

memória somente de leitura (ROM) Uma memória cujo conteúdo é definido no momento da criação, após o qual o conteúdo só pode ser lido. A ROM é usada como lógica estruturada para implementar um conjunto de funções lógicas usando os termos das funções lógicas como entradas de endereço e as saídas como bits em cada word da memória.

ROM programável (PROM) Uma forma de memória somente de leitura que pode ser programada quando um projetista conhece seu conteúdo

Outra forma de lógica estruturada que pode ser usada para implementar um conjunto de funções lógicas é uma **memória somente de leitura (ROM – Read-Only Memory)**. Uma ROM é chamada de memória porque possui um conjunto de locais que podem ser lidos; porém, o conteúdo desses locais é fixo, normalmente no momento em que a ROM é fabricada. Há também **ROMs programáveis (PROMs – Programmable ROMs)**, que podem ser programadas eletronicamente, quando um projetista conhece seu conteúdo. Há também PROMs apagáveis; esses dispositivos exigem um processo de apagamento lento usando luz ultravioleta, e assim são usados como memórias somente de leitura, exceto durante o processo de projeto e depuração.

Uma ROM possui um conjunto de linhas de endereço de entrada e um conjunto de saídas. O número de entradas endereçáveis na ROM determina o número de linhas de endereço: se a ROM contém 2^m entradas endereçáveis, chamadas de *altura*, então existem m linhas de entrada. O número de bits em cada entrada endereçável é igual ao número de bits de saída e às vezes é chamado de *largura* da ROM. O número total de bits na ROM é igual à altura vezes a largura. A altura e a largura às vezes são chamadas coletivamente como o *formato* da ROM.

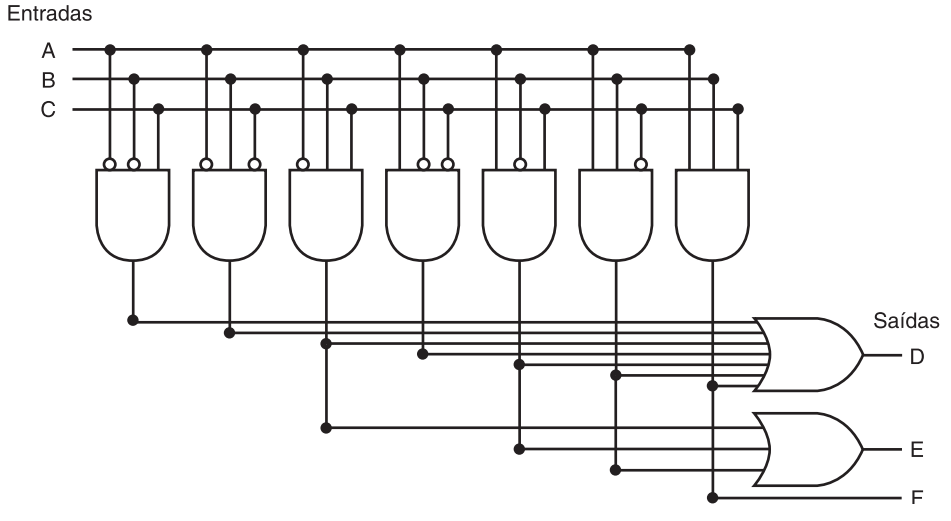


FIGURA C.3.4 **A PLA para implementar a função lógica descrita anteriormente.**

Uma ROM pode codificar uma coleção de funções lógicas diretamente a partir da tabela verdade. Por exemplo, se houver n funções com m entradas, precisamos de uma ROM com m linhas de endereço (e 2^m entradas), com cada entrada sendo de n bits de largura. O conteúdo na parte de entrada da tabela verdade representa os endereços do conteúdo na ROM, enquanto o conteúdo da parte de saída da tabela verdade constitui o conteúdo da ROM. Se a tabela verdade for organizada de modo que a sequência de entradas na parte da entrada constitua uma sequência de números binários (como todas as tabelas verdade mostradas até aqui), então a parte de saída também indica o conteúdo da ROM em ordem. No exemplo anterior, a partir da página C-10, havia três entradas e três saídas. Isso equivale a uma ROM com $2^3 = 8$ entradas, cada uma com 3 bits de largura. O conteúdo dessas entradas em ordem crescente por endereço é dado diretamente pela parte de saída da tabela verdade que aparece na C-10.

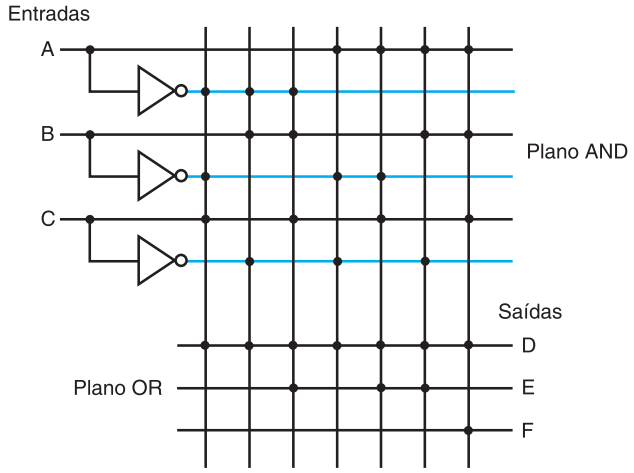


FIGURA C.3.5 Uma PLA desenhada usando pontos para indicar os componentes dos termos do produto e os termos da soma no array. Em vez de usar inversores nas portas lógicas, normalmente todas as entradas percorrem a largura do plano AND nas formas original e complemento. Um ponto no plano AND indica que a entrada, ou seu inverso, ocorre no termo do produto. Um ponto no plano OR indica que o termo do produto correspondente aparece na saída correspondente.

ROMs e PLAs estão bastante relacionadas. Uma ROM é totalmente decodificada: ela contém uma word de saída completa para cada combinação de entrada possível. Uma PLA é decodificada apenas parcialmente. Isso significa que uma ROM sempre terá mais conteúdo. Para a tabela verdade anterior, na C-10, a ROM contém itens para todas as oito entradas possíveis, enquanto a PLA contém apenas os sete termos de produto ativos. À medida que o número de entradas cresce, o número de entradas na ROM cresce exponencialmente. Ao contrário, para a maioria das funções lógicas reais, o número de termos de produto cresce muito mais lentamente (ver os exemplos no Apêndice D). Essa diferença torna as PLAs geralmente mais eficientes para implementar as funções lógicas combinacionais. As ROMs possuem a vantagem de serem capazes de implementar qualquer função lógica com o seu número de entradas e saídas. Essa vantagem facilita mudar o conteúdo da ROM se a função lógica mudar, pois o tamanho da ROM não precisa mudar.

Além de ROMs e PLAs, os sistemas de síntese de lógica modernos também traduzirão pequenos blocos de lógica combinacional em uma coleção de portas que podem ser colocadas e ligadas automaticamente. Embora algumas pequenas coleções de portas não façam uso eficiente da área, para pequenas funções lógicas elas possuem menos overhead do que a estrutura rígida de uma ROM ou PLA e, por isso, são preferidas.

Para projetar a lógica fora de um circuito integrado personalizado ou semipersonalizado, uma opção comum é um dispositivo programável em campo; descrevemos esses dispositivos na Seção C.12.

Don't Cares

Normalmente, na implementação de alguma lógica combinacional, existem situações em que não nos importamos com o valor de alguma saída, seja porque outra saída é verdadeira ou porque um subconjunto de combinações de entrada determina os valores das saídas. Essas situações são conhecidas como *don't cares*. Don't cares são importantes porque facilitam a otimização da implementação de uma função lógica.

Existem dois tipos de don't cares: don't cares de saída e don't cares de entrada, ambos podendo ser representados em uma tabela verdade. Os *don't cares de saída* surgem quando não nos importamos com o valor de uma saída para alguma combinação de entrada. Eles aparecem como X na parte de saída de uma tabela verdade. Quando uma saída é um don't care para alguma combinação de entrada, o projetista ou o programa de otimização da lógica é livre para tornar a saída verdadeira ou falsa para essa combinação de entrada. *Don't cares de entrada* surgem quando uma saída depende apenas de algumas das entradas, e também são mostradas como X, embora na parte de entrada da tabela verdade.

Don't Cares

EXEMPLO

Considere uma função lógica com entradas *A, B* e *C* definidas da seguinte maneira:

- Se *A* ou *C* é verdadeira, então a saída *D* é verdadeira, qualquer que seja o valor de *B*.
- Se *A* ou *B* é verdadeira, então a saída *E* é verdadeira, qualquer que seja o valor de *C*.
- A saída *F* é verdadeira se exatamente uma das entradas for verdadeira, embora não nos importemos com o valor de *F*, sempre que *D* e *E* são verdadeiras.

Mostre a tabela verdade completa para essa função e a tabela verdade usando don't cares. Quantos termos do produto são exigidos em uma PLA para cada uma delas?

RESPOSTA

Aqui está a tabela verdade completa, sem os don't cares:

Entradas			Saídas		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

Isso exige sete termos do produto sem otimização. A tabela verdade escrita com don't cares de saída se parece com esta:

Entradas			Saídas		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

Se também usarmos os don't cares de entrada, essa tabela verdade pode ser simplificada ainda mais, para gerar:

Entradas			Saídas		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

Essa tabela verdade simplificada exige uma PLA com quatro mintermos ou pode ser implementada em portas discretas com uma porta AND de duas entradas e três portas OR (duas com três entradas e uma com duas entradas). Isso confronta a tabela verdade original, que tinha sete mintermos e exigiria quatro portas AND.

A minimização lógica é crítica para conseguir implementações eficientes. Uma ferramenta útil para a minimização manual da lógica aleatória são os *mapas de Karnaugh*. Os mapas de Karnaugh representam a tabela verdade graficamente, de modo que os termos do produto que podem ser combinados são facilmente vistos. Apesar disso, a otimização manual das funções lógicas significativas usando os mapas de Karnaugh não é prática, tanto devido ao tamanho dos mapas quanto pela sua complexidade. Felizmente, o processo de minimização lógica é muito mecânico e pode ser realizado por ferramentas de projeto. No processo de minimização, as ferramentas tiram proveito dos don't cares, de modo que sua especificação é importante. As referências do livro-texto ao final deste apêndice oferecem uma discussão mais profunda sobre minimização lógica, mapas de Karnaugh e a teoria por trás de tais algoritmos de minimização.

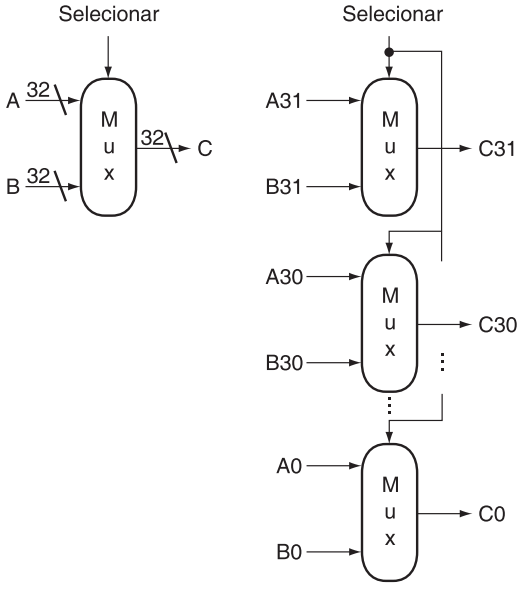
Arrays de elementos lógicos

Muitas das operações combinacionais a serem realizadas sobre os dados precisam ser feitas em uma word inteira (32 bits) de dados. Assim, normalmente queremos montar um array de elementos lógicos, que podemos representar apenas mostrando que determinada operação acontecerá a uma cole- ção inteira de entradas. Por exemplo, vimos na página C-7 a aparência de um multiplexador de 1 bit, mas, dentro de uma máquina, em grande parte do tempo queremos selecionar entre um par de *barramentos*. Um **barramento** é uma coleção de linhas de dados tratada em conjunto como um único sinal lógico. (O termo *barramento* também é usado para indicar uma coleção compartilhada de linhas com várias fontes e usos, especialmente no Capítulo 6, onde discutimos sobre os barramentos de E/S.)

barramento No projeto lógico, uma coleção de linhas de dados que é tratada em conjunto como um único sinal lógico; também é uma coleção compartilhada de linhas com várias fontes e usos.

Por exemplo, no conjunto de instruções MIPS, o resultado de uma instrução escrita em um registrador pode vir de uma dentre duas fontes. Um multiplexador é usado para escolher qual dos dois barramentos (cada um com 32 bits de largura) será escrito no registrador destino. O multiplexador de 1 bit, que mostramos anteriormente, precisará ser replicado 32 vezes.

Indicamos que um sinal é um barramento em vez de uma única linha de 1 bit representando-o com uma linha mais grossa em uma figura. A maioria dos barramentos possui 32 bits de largura; os que não possuem são rotulados explicitamente com sua largura. Quando mostramos uma unidade lógica cujas entradas e saídas são barramentos, isso significa que a unidade precisa ser replicada por um número de vezes suficiente para acomodar a largura da entrada. A Figura C.3.6 mostra como desenhamos um multiplexador que seleciona entre um par de barramentos de 32 bits e como isso se expande em termos de multiplexadores de 1 bit de largura. Às vezes, precisamos construir um array de elementos lógicos onde as entradas para alguns elementos no array são saídas de elementos anteriores. Por exemplo, é assim que é construída uma ALU de múltiplos bits de largura. Nesses casos, temos de mostrar explicitamente como criar arrays mais largos, pois os elementos individuais do array não são mais independentes, como acontece no caso de um multiplexador de 32 bits de largura.



a.Um multiplexador 2 para 1 de 32 bits de largura

b.Um multiplexador de 32 bits de largura é atualmente um array de 32 multiplexadores de 1 bit

FIGURA C.3.6 Um multiplexador é duplicado 32 vezes para realizar uma seleção entre duas entradas de 32 bits. Observe que ainda existe apenas um sinal de seleção de dados usado para todos os 32 multiplexadores de 1 bit.

Verifique você mesmo

A paridade é uma função na qual a saída depende do número de 1s na entrada. Para uma função de paridade par, a saída é 1 se a entrada tiver um número par de uns. Suponha que uma ROM seja usada para implementar uma função de paridade par com uma entrada de 4 bits. Qual dentre A, B, C ou D representa o conteúdo da ROM?

Endereço	A	B	C	D
0	0	1	0	1
1	0	1	1	0
2	0	1	0	1
3	0	1	1	0
4	0	1	0	1
5	0	1	1	0
6	0	1	0	1
7	0	1	1	0
8	1	0	0	1
9	1	0	1	0
10	1	0	0	1
11	1	0	1	0
12	1	0	0	1
13	1	0	1	0
14	1	0	0	1
15	1	0	1	0

C.4 Usando uma linguagem de descrição de hardware

linguagem de descrição de hardware Uma linguagem de programação para descrever o hardware utilizado para gerar simulações de um projeto de hardware e também como entrada para ferramentas de síntese que podem gerar hardware real.

Hoje, a maior parte do projeto digital dos processadores e sistemas de hardware relacionados é feita por meio de uma **linguagem de descrição de hardware**. Essa linguagem tem duas finalidades. Primeiro, ela oferece uma descrição abstrata do hardware para simular e depurar o projeto. Segundo, com o uso da síntese lógica e ferramentas de compilação de hardware, essa descrição pode ser compilada para a implementação do hardware. Nesta

seção, introduzimos a linguagem de descrição de hardware Verilog e mostramos como ela pode ser usada para o projeto combinacional. No restante do apêndice, expandimos o uso da Verilog para incluir o projeto da lógica sequencial. Em seções opcionais do Capítulo 4, que aparece no CD, usamos Verilog do sistema para descrever implementações do controlador de cache. A Verilog do sistema acrescenta estruturas e alguns recursos úteis à Verilog.

Verilog Uma das duas linguagens de descrição de hardware mais comuns.

VHDL Uma das duas linguagens de descrição de hardware mais comuns.

Verilog é uma das duas principais linguagens de descrição de hardware; a outra é **VHDL**. Verilog é um pouco mais utilizada no setor e é baseada em C, ao contrário de VHDL, que é baseada em Ada. O leitor um pouco mais familiarizado com C achará mais fácil acompanhar os fundamentos da Verilog, que utilizamos neste apêndice. Os leitores já acostumados com VHDL deverão achar os conceitos simples, desde que já conheçam um pouco da sintaxe da linguagem C.

especificação comportamental Descreve como um sistema digital opera funcionalmente.

especificação estrutural Descreve como um sistema digital é organizado em termos de uma conexão hierárquica de elementos.

Verilog pode especificar uma definição comportamental e uma estrutural de um sistema digital. Uma **especificação comportamental** descreve como um sistema digital opera funcionalmente. Uma **especificação estrutural** descreve a organização detalhada de um sistema digital normalmente utilizando uma descrição hierárquica. Uma especificação estrutural pode ser usada para descrever um sistema de hardware em termos de uma hierarquia de elementos básicos, como portas lógicas e chaves. Assim, poderíamos usar a Verilog para descrever o conteúdo exato das tabelas verdade e o caminho de dados da seção anterior.

ferramentas de síntese de hardware Software de projeto auxiliado por computador que pode gerar um projeto no nível de portas lógicas baseado em descrições comportamentais de um sistema digital.

Com o surgimento das ferramentas de **síntese de hardware**, a maioria dos projetistas agora utiliza Verilog ou VHDL para descrever estruturalmente apenas o caminho de dados, contando com a síntese lógica para gerar o controle a partir da descrição comportamental. Além disso, a maioria dos sistemas de CAD oferece grandes bibliotecas de peças padronizadas, como ALUs, multiplexadores, bancos de registradores, memórias, blocos lógicos programáveis, além de portas básicas.

A obtenção de um resultado aceitável usando bibliotecas e síntese de lógica exige que a especificação seja escrita vigiando a síntese eventual e o resultado desejado. Para nossos projetos simples, isso significa principalmente deixar claro o que esperamos que seja implementado na lógica combinacional e o que esperamos exigir da lógica sequencial. Na maior parte dos exemplos que usamos nesta seção, e no restante deste apêndice, escrevemos em Verilog visando à síntese eventual.

Tipos de dados e operadores em Verilog

Existem dois tipos de dados principais em Verilog:

wire Em Verilog, especifica um sinal combinacional.

reg Em Verilog, um registrador.

1. Um **wire** especifica um sinal combinacional.
2. Um **reg** (registrador) mantém um valor, que pode variar com o tempo. Um reg não precisa corresponder necessariamente a um registrador real em uma implementação, embora isso normalmente aconteça.

Um registrador ou wire, chamado X, que possui 32 bits de largura, é declarado como um array: `reg [31:0] X` ou `wire [31:0] X`, que também define o índice de 0 para designar o bit menos significativo do registrador. Como normalmente queremos acessar um subcampo de um registrador ou wire, podemos nos referir ao conjunto contíguo de bits de um registrador ou wire com a notação [`bit inicial: bit final`], onde os dois índices devem ser valores constantes.

Um array de registradores é usado para uma estrutura como um banco de registradores ou memória. Assim, a declaração

```
reg [31:0] registerfile[0:31]
```

especifica uma variável register file que é equivalente a um banco de registradores MIPS, onde o registrador 0 é o primeiro. Ao acessar um array, podemos nos referir a um único elemento, como em C, usando a notação `registerfile[numreg]`.

Os valores possíveis para um registrador ou wire em Verilog são:

- 0 ou 1, representando o falso ou verdadeiro lógico.
- x, representando desconhecido, o valor inicial dado a todos os registradores e a qualquer wire não conectado a algo.
- z, representando o estado de impedância alta para portas tristate, que não discutiremos neste apêndice.

Valores constantes podem ser especificados como números decimais e também como binário, octal ou hexadecimal. Normalmente, queremos dizer o tamanho de um campo constante em bits. Isso é feito prefixando o valor com um número decimal que especifica seu tamanho em bits. Por exemplo:

- `4'b0100` especifica uma constante binária de 4 bits com o valor 4, assim como `4'd4`.
- `– 8 'h4` especifica uma constante de 8 bits com o valor –4 (na representação complemento a dois).

Os valores também podem ser concatenados colocando-os dentro de { } separados por vírgulas. A notação `{x {bit field}}` replica `bit field x` vezes. Por exemplo:

- `{16{2'b01}}` cria um valor de 32 bits com o padrão 0101 ... 01.
- `{A[31:16],B[15:0]}` cria um valor cujos 16 bits mais significativos vêm de `A` e cujos 16 bits menos significativos vêm de `B`.

Verilog oferece o conjunto completo de operadores unários e binários de C, incluindo os operadores aritméticos (+, –, *, /), os operadores lógicos (&, |, ~), os operadores de comparação (==, !=, >, <, <=, >=), os operadores de deslocamento (<<, >>) e o operador condicional de C (?, que é usado na forma `condição ? expr1 :expr2` e retorna `expr1` se a condição for verdadeira e `expr2` se ela for falsa). Verilog acrescenta um conjunto de operadores unários de redução lógica (&, |, ^) que geram um único bit aplicando o operador lógico a todos os bits de um operando. Por exemplo, `&A` retorna o valor obtido pelo AND de todos os bits de `A`, e `^A` retorna a redução obtida pelo uso do OR exclusivo em todos os bits de `A`.

Verifique você mesmo

Quais dos seguintes itens definem exatamente o mesmo valor?

1. `8'b11110000`
2. `8'hF0`
3. `8'd240`
4. `{{4{1'b1}},{4{1'b0}}}`
5. `{4'b1,4'b0}`

Estrutura de um programa em Verilog

Um programa em Verilog é estruturado como um conjunto de módulos, que podem representar qualquer coisa desde uma coleção de portas lógicas até um sistema completo. Os módulos são semelhantes às classes em C++, embora não tão poderosas. Um módulo especifica suas portas de entrada e saída, que descrevem as conexões de entrada e saída de um módulo. Um módulo também pode declarar variáveis adicionais. O corpo de um módulo consiste em:

- Construções `initial`, que podem inicializar variáveis `reg`.
- Atribuições contínuas, que definem apenas lógica combinacional.
- Construções `always`, que podem definir a lógica sequencial ou combinacional.
- Instâncias de outros módulos, usadas para implementar o módulo sendo definido.

Representando lógica combinacional complexa em Verilog

Uma atribuição contínua, indicada com a palavra-chave `assign`, atua como uma função lógica combinacional: a saída é atribuída continuamente ao valor, e uma mudança nos valores de entrada é refletida imediatamente no valor da saída. Os wires só podem receber valores com atribuições contínuas. Usando a atribuição contínua, podemos definir um módulo que implementa um meio-somador, como mostra a Figura C.4.1.

As instruções de atribuição são um modo seguro de escrever Verilog que gera lógica combinacional. Entretanto, para estruturas mais complexas, as instruções de atribuição podem ser esquisitas ou tediosas de usar. Também é possível usar o bloco `always` de um módulo para descrever um elemento lógico combinacional, embora com muito cuidado. O uso de um bloco `always` permite a inclusão de construções de controle da Verilog, como *if-then-else*, instruções *case*, instruções *for* e instruções *repeat*. Essas instruções são semelhantes às que existem em C, com pequenas mudanças.

```
module half_adder (A,B,Sum,Carry);
   input A,B; //two 1-bit inputs
   output Sum, Carry; //two 1-bit outputs
   assign Sum = A ^ B; //sum is A xor B
   assign Carry = A & B; //Carry is A and B
endmodule
```

FIGURA C.4.1 **Um módulo em Verilog que define um meio-somador usando atribuições contínuas.**

lista de sensitividade A lista de sinais que especifica quando um bloco always deve ser reavaliado.

Um bloco `always` especifica uma lista opcional de sinais aos quais o bloco é sensitivo (em uma lista começando com @). O bloco always é reavaliado se qualquer um dos sinais listados mudar de valor; se a lista for omitida, o bloco `always` é constantemente reavaliado. Quando um bloco `always` está especificando a lógica combinacional, a **lista de sensitividade** deverá incluir todos os sinais de entrada. Se houver várias instruções Verilog a serem executadas em um bloco `always`, elas estão cercadas pelas palavras-chave `begin` e `end`, que tomam o lugar de { e } em C. Um bloco `always`, portanto, se parece com

```
always @(lista de sinais que causam reavaliação) begin
   Instruções Verilog incluindo atribuições e outras instruções de controle
end
```

atribuição bloqueante Em Verilog, uma atribuição que completa antes da execução da próxima instrução.

atribuição não bloqueante Uma atribuição que continua após a avaliação do lado direito, atribuindo o valor ao lado esquerdo somente depois que todo o lado direito for avaliado.

Variáveis `reg` só podem ser atribuídas dentro de um bloco `always`, usando uma instrução de atribuição procedural (distinguida da atribuição contínua vista anteriormente). Contudo, existem dois tipos diferentes de atribuições procedurais. O operador de atribuição = é executado como em C; o lado direito é avaliado e o lado esquerdo recebe o valor. Além do mais, ele é executado como uma instrução de atribuição C normal: ou seja, é completado antes que a próxima instrução seja executada. Logo, o operador de atribuição = tem o nome **atribuição bloqueante**. Esse bloqueio pode ser útil na geração da lógica sequencial, e voltaremos a esse assunto em breve. A outra forma de atribuição (**não bloqueante**) é indicada por <=. Na atribuição não bloqueante, todo o lado direito das atribuições em um grupo `always` é avaliado, e as atribuições são feitas simultaneamente. Como um primeiro exemplo da lógica combinacional implementada usando um bloco `always`, a Figura C.4.2 mostra a implementação de um multiplexador 4-para-1, que usa uma construção `case` para facilitar a escrita. A construção `case` se parece com uma instrução `switch` do C. A Figura C.4.3 mostra uma definição de uma ALU MIPS, que também usa uma instrução `case`.

Como apenas variáveis reg podem ser atribuídas dentro de blocos `always`, quando queremos descrever a lógica combinacional usando um bloco `always`, devemos ter o cuidado de garantir que o reg não seja sintetizado como um registrador. Diversas armadilhas são descritas na Seção "Detalhamento", a seguir.

Detalhamento: instruções de atribuição contínua sempre geram lógica combinacional, mas outras estruturas Verilog, mesmo quando em blocos `always`, podem gerar resultados inesperados durante a síntese lógica. O problema mais comum é a criação de lógica sequencial implicando a existência de um latch ou registrador, o que resulta em uma implementação mais lenta e mais dispendiosa do que talvez pretendido. Para garantir que a lógica que deverá ser combinacional seja sintetizada dessa maneira, faça o seguinte:

1. Coloque toda a lógica combinacional em uma atribuição contínua ou em um bloco `always`.
2. Verifique se todos os sinais usados como entradas aparecem na lista de sensitividade de um bloco `always`.
3. Garanta que cada caminho dentro de um bloco `always` atribui um valor ao mesmo conjunto exato de bits.

O último deles é o mais fácil de se deixar de lado; examine o exemplo da Figura C.5.15 para convencer-se de que essa propriedade foi respeitada.

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
   input [31:0] In1, In2, In3, In4; /quatro entradas de 32 bits
   input [1:0] Sel; //sinal seletor
   output reg [31:0] Out;// saída de 32 bits
   always @(In1, In2, In3, In4, Sel)
   case (Sel) //um 4->1 mutiplexador
      0: Out <= In1;
      1: Out <= In2;
      2: Out <= In3;
      default: Out <= In4;
   endcase
endmodule
```

FIGURA C.4.2 Uma definição Verilog de um multiplexador 4-para-1 com entradas de 32 bits, usando uma instrução `case`. Uma instrução `case` atua como uma instrução `switch` do C, exceto que, em Verilog, somente o código associado ao case selecionado é executado (como se cada estado de case tivesse um break no final) e não existe passagem direta para a instrução seguinte.

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero é verdadeiro se ALUOut é 0; vai em qualquer lugar
   always @(ALUctl, A, B) //reavaliar se este mudar
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A – B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // resultado é nor
         default: ALUOut <= 0; //padrão para 0, não deve acontecer;
      endcase
endmodule
```

FIGURA C.4.3 Uma definição comportamental em Verilog de uma ALU MIPS. Isso poderia ser sintetizado por meio de uma biblioteca de módulos contendo operações aritméticas e lógicas básicas.

Verifique você mesmo

Supondo que todos os valores sejam inicialmente zero, quais são os valores de A e B depois de executar este código Verilog dentro de um bloco always?

```
C=1;
A <= C;
B = C;
```

*ALU n. [**A**rthritic **L**ogic **U**nit ou (raro) **A**rithmetic **L**ogic **U**nit] Um gerador de números aleatórios fornecido por padrão com todos os sistemas computacionais.*

Stan Kelly-Bootle, The *Devil's DP Dictionary,* 1981

C.5 Construindo uma unidade lógica e aritmética

A **unidade lógica e aritmética** (**ALU** – Arithmetic Logic Unit) é o músculo do computador, o dispositivo que realiza as operações aritméticas, como adição e subtração, ou as operações lógicas, como AND e OR. Esta seção constrói uma ALU a partir de quatro blocos de montagem do hardware (portas AND e OR, inversores e multiplexadores) e ilustra como funciona a lógica combinacional. Na próxima seção, veremos como a adição pode ser agilizada por meio de projetos mais inteligentes.

Como a word do MIP tem 32 bits de largura, precisamos de uma ALU de 32 bits. Vamos supor que iremos conectar 32 ALUs de 1 bit para criar a ALU desejada. Portanto, vamos começar construindo uma ALU de 1 bit.

Uma ALU de 1 bit

As operações lógicas são as mais fáceis, pois são mapeadas diretamente nos componentes de hardware da Figura C.2.1.

A unidade lógica de 1 bit para AND e OR se parece com a Figura C.5.1. O multiplexador à direita, então, seleciona *a* AND *b* ou *a* OR *b*, dependendo se o valor de *Operação* é 0 ou 1. A linha que controla o multiplexador aparece em destaque para distingui-la das linhas com dados. Observe que renomeamos as linhas de controle e a saída do multiplexador para lhes dar nomes que refletem a função da ALU.

A próxima função a incluir é a adição. Um somador precisa ter duas entradas para os operandos e uma saída de único bit para a soma. É preciso haver uma segunda saída para o carry, chamada *CarryOut*. Como o CarryOut do somador vizinho precisa ser incluído como uma entrada, precisamos de uma terceira entrada. Essa entrada é chamada *CarryIn*. A Figura C.5.2 mostra as entradas e as saídas de um somador de 1 bit. Como sabemos o que a adição precisa fazer, podemos especificar as saídas dessa "caixa preta" com base em suas entradas, como a Figura C.5.3 demonstra.

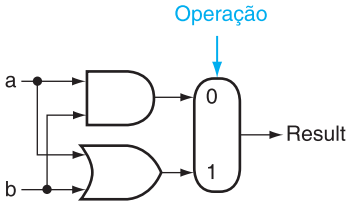


FIGURA C.5.1 **A unidade lógica de 1 bit para AND e OR.**

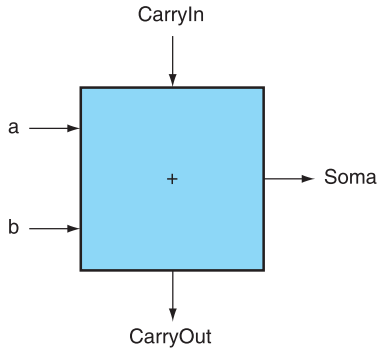


FIGURA C.5.2 Um somador de 1 bit. Esse somador é chamado de somador completo; ele também é chamado somador (3,2), pois tem 3 entradas e 2 saídas. Um somador com apenas as entradas a e b é chamado somador (2,2) ou meio somador.

Entradas			Saídas		
a	B	CarryIn	CarryOut	Soma	Comentários
0	0	0	0	0	$0 + 0 + 0 = 00_{bin}$
0	0	1	0	1	$0 + 0 + 1 = 01_{bin}$
0	1	0	0	1	$0 + 1 + 0 = 01_{bin}$
0	1	1	1	0	$0 + 1 + 1 = 10_{bin}$
1	0	0	0	1	$1 + 0 + 0 = 01_{bin}$
1	0	1	1	0	$1 + 0 + 1 = 10_{bin}$
1	1	0	1	0	$1 + 1 + 0 = 10_{bin}$
1	1	1	1	1	$1 + 1 + 1 = 11_{bin}$

FIGURA C.5.3 Especificação de entrada e saída para um somador de 1 bit.

Podemos expressar as funções de saída CarryOut e Soma como equações lógicas, e essas equações, por sua vez, podem ser implementadas com portas lógicas. Vamos realizar um CarryOut. A Figura C.5.4 mostra os valores das entradas quando CarryOut é 1.

Podemos transformar essa tabela verdade em uma equação lógica.

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

Entradas		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

FIGURA C.5.4 Valores das entradas quando CarryOut é 1.

Se a · b · CarryIn for verdadeiro, então todos os outros três termos também precisam ser verdadeiros, de modo que podemos omitir esse último termo correspondente à quarta linha da tabela. Assim, podemos simplificar a equação para

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

A Figura C.5.5 mostra que o hardware dentro da caixa preta do somador para CarryOut consiste em três portas AND e uma porta OR. As três portas AND correspondem exatamente aos três termos entre parênteses da fórmula anterior para CarryOut, e a porta OR soma os três termos.

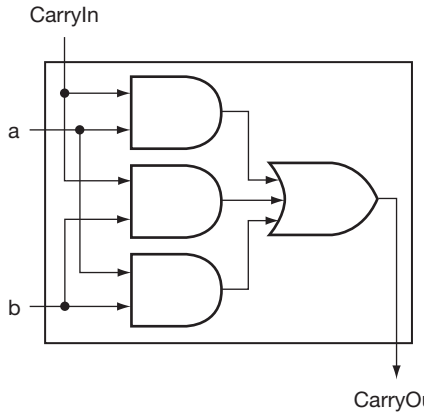


FIGURA C.5.5 Hardware do somador para o sinal CarryOut. O restante do hardware do somador é a lógica para a saída de Soma dada na equação da C-23.

O bit Soma é ligado quando exatamente uma entrada é 1 ou quando todas as três entradas são 1. A Soma resulta em uma equação Booleana complexa (lembre-se de que $\overline{a}$ significa NOT a):

$$\text{Soma} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

O desenho da lógica para o bit Soma na caixa preta do somador fica como um exercício.

A Figura C.5.6 mostra uma ALU e 1 bit derivada da combinação do somador com os componentes anteriores. Às vezes, os projetistas também querem que a ALU realize mais algumas operações simples, como gerar 0. O modo mais fácil de somar uma operação é expandir o multiplexador controlado pela linha Operação e, para este exemplo, conectar 0 diretamente à nova entrada desse multiplexador expandido.

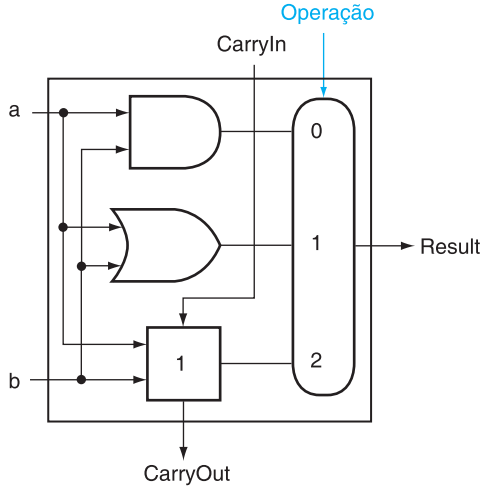


FIGURA C.5.6 Uma ALU de 1 bit realiza AND, OR e adição (ver Figura C.5.5).

Uma ALU de 32 bits

Agora que completamos a ALU de 1 bit, a ALU completa de 32 bits é criada conectando "caixas pretas" adjacentes. Usando *xi* para indicar o *i-ésimo* bit de *x*, a Figura C.5.7 mostra uma ALU de 32 bits. Assim como uma única pedra pode causar ondulações partindo da

costa de um lago tranquilo, um único carry do bit menos significativo (Result0) pode causar ondulações por todo o somador, levando a uma carry do bit mais significativo (Result31). Logo, o somador criado ligando diretamente os carries de somadores de 1 bit é chamado de somador de *carry por ondulação*. Veremos um modo rápido de conectar os somadores de 1 bit a partir da C-30.

A subtração é o mesmo que a adição da versão negativa de um operando, e é assim que os somadores realizam a subtração. Lembre-se de que o atalho para negar um número em complemento a dois é inverter cada bit (às vezes chamado de *complemento a um*) e depois somar 1. Para inverter cada bit, simplesmente acrescentamos um multiplexador 2:1 que escolhe entre b e $\overline{b}$, como mostra a Figura C.5.8.

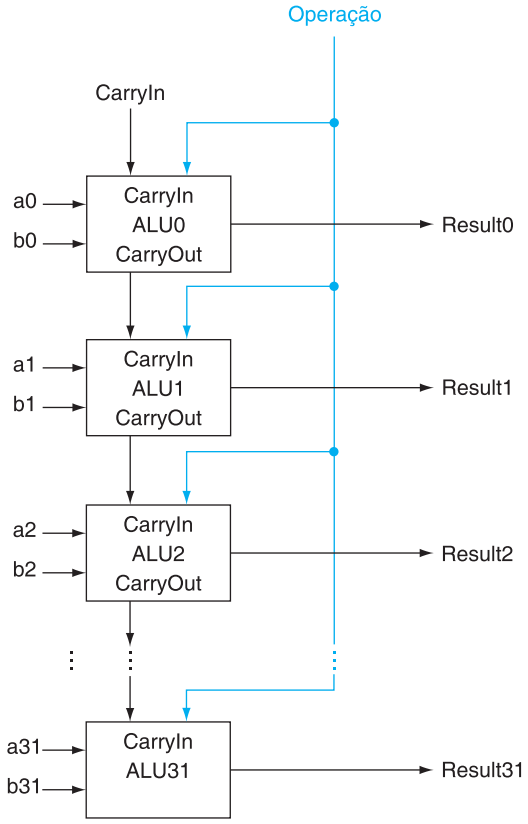


FIGURA C.5.7 Uma ALU de 32 bits construída a partir de 32 ALUs de 1 bit. O CarryOut de um bit é conectado ao CarryIn do próximo bit mais significativo. Essa organização é chamada de carry por ondulação.

Suponha que conectemos 32 dessas ALUs de 1 bit, como fizemos na Figura C.5.7. O multiplexador adicionado dá a opção de b ou seu valor invertido, dependendo de Binvert, mas essa é apenas uma etapa na negação de um número em complemento a dois. Observe que o bit menos significativo ainda possui um sinal CarryIn, embora seja desnecessário para a adição. O que acontece se definirmos esse CarryIn como 1 em vez de 0? O somador, então, calculará a + b + 1. Selecionando a versão invertida de b, obtemos exatamente o que queremos:

$$a + \overline{b} + 1 = a + (\overline{b} + 1) = a + (-b) = a - b$$

A simplicidade do projeto de hardware de um somador em complemento a dois ajuda a explicar por que a representação em complemento a dois tornou-se um padrão universal para a aritmética computacional com inteiros.

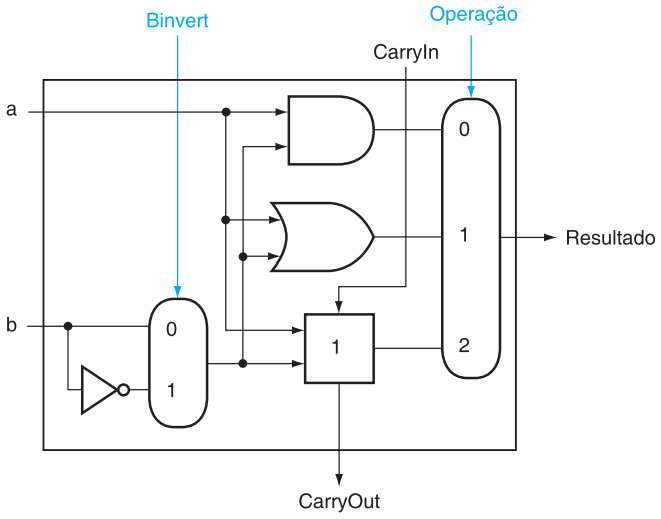


FIGURA C.5.8 Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou a e $\overline{b}$. Selecionando $\overline{b}$ (Binvert = 1) e definindo CarryIn como 1 no bit menos significativo da ALU, obtemos a subtração em complemento a dois de b a partir de a, em vez da adição de b e a.

Uma ALU no MIPS também precisa de uma função NOR. Em vez de acrescentar uma porta separada para NOR, podemos reutilizar grande parte do hardware já existente na ALU, como fizemos para a subtração. A ideia vem da seguinte tabela verdade sobre NOR:

$$\overline{(a + b)} = \overline{a} \cdot \overline{b}$$

Ou seja, NOT (a OR b) é equivalente a NOT a AND NOT b. Esse fato é chamado de teorema de DeMorgan e é explorado nos exercícios com mais profundidade.

Como temos AND e NOT b, só precisamos acrescentar NOT a à ALU. A Figura C.5.9 mostra essa mudança.

Ajustando a ALU de 32 bits ao MIPS

Essas quatro operações – adição, subtração, AND, OR –, são encontradas na ALU de quase todo computador, e as operações da maioria das instruções MIPS podem ser realizadas por essa ALU. Mas o projeto da ALU está incompleto.

Uma instrução que ainda precisa de suporte é a instrução "set on less than" (`slt`). Lembre-se de que a operação produz 1 se rs < rt, ou 0 em caso contrário. Consequentemente, `slt` colocará todos os bits, menos o bit menos significativo, em 0, com o valor do bit menos significativo definido de acordo com a comparação. Para a ALU realizar `slt`, primeiro precisamos expandir o multiplexador de três entradas da Figura C.5.8 para acrescentar uma entrada para o resultado de `slt`. Chamamos essa nova entrada de *Less* e a usamos apenas para `slt`.

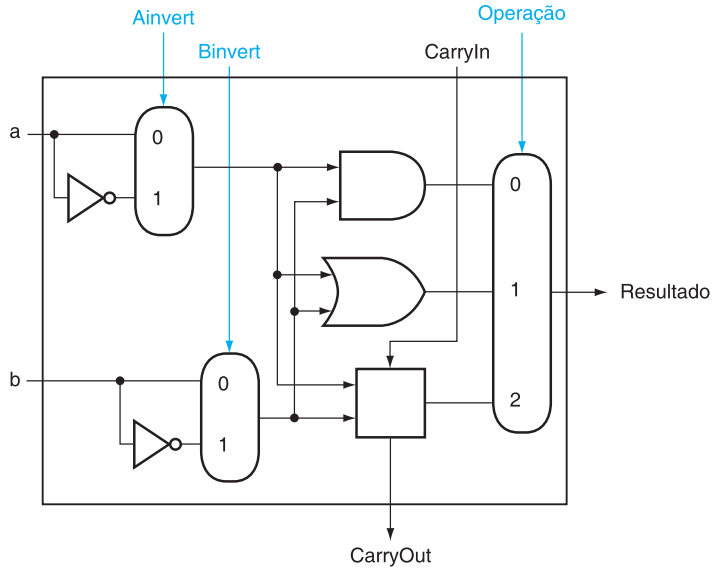


FIGURA C.5.9 Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou $\bar{a}$ e $\bar{b}$. Selecionando $\bar{a}$ (Ainvert = 1) e $\bar{b}$ (Binvert = 1), obtemos a NOR b, em vez de a AND b.

O desenho superior da Figura C.5.10 mostra a nova ALU de 1 bit com o multiplexador expandido. A partir da descrição de `slt` anterior, temos de conectar 0 à entrada Less para os 31 bits superiores da ALU, pois esses bits sempre serão 0. O que falta considerar é como comparar e definir o valor do *bit menos significativo* para instruções "set on less than".

O que acontece se subtrairmos b de a? Se a diferença for negativa, então a < b, pois

$$(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b)$$
$$\Rightarrow a < b$$

Queremos que o bit menos significativo de uma operação "set on less than" seja 1 se a < b; ou seja, 1 se a – b for negativo e 0 se for positivo. Esse resultado desejado corresponde exatamente aos valores do bit de sinal: 1 significa negativo e 0 significa positivo. Seguindo essa linha de argumento, só precisamos conectar o bit de sinal da saída do somador ao bit menos significativo para obter "set on less than".

Infelizmente, a saída Result do bit da ALU mais significativo no topo da Figura C.5.10 para a operação `slt` *não* é a saída do somador; a saída da ALU para a operação `slt` é obviamente o valor de entrada Less.

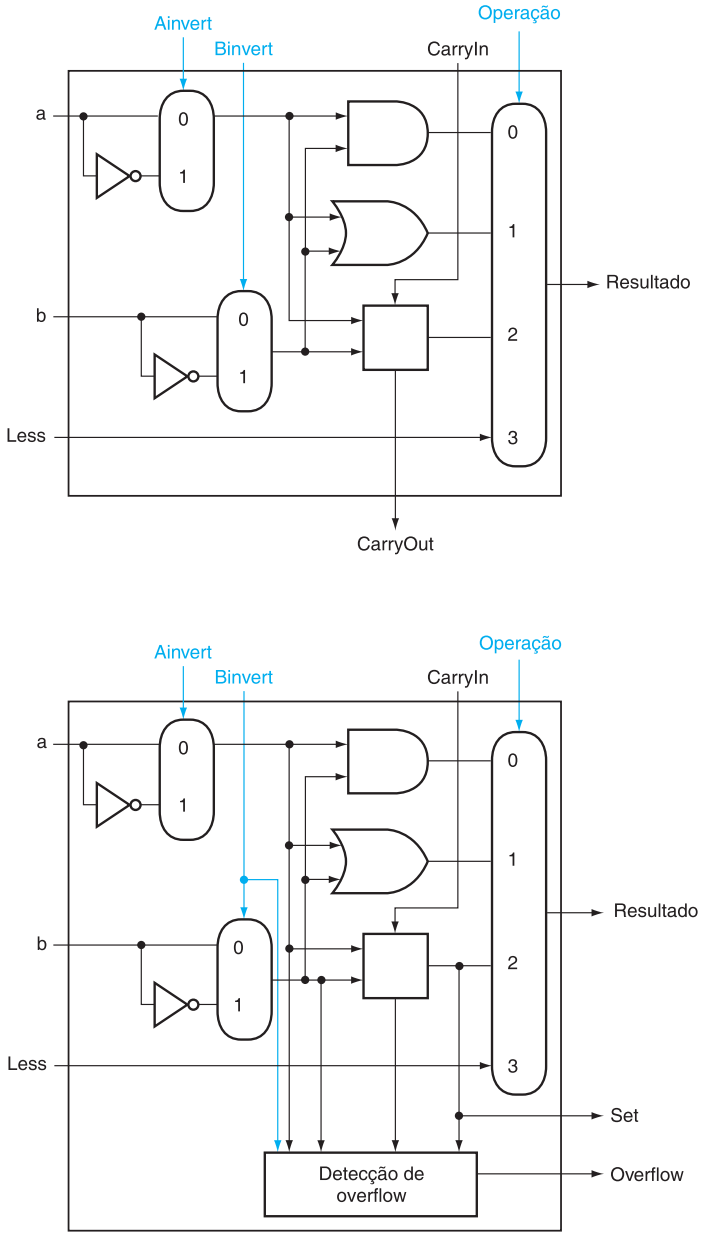


FIGURA C.5.10 **(Superior) Uma ALU de 1 bit que realiza AND, OR e adição entre a e b ou $\overline{b}$, e (inferior) uma ALU de 1 bit para o bit mais significativo**. O desenho superior inclui uma entrada direta que está conectada para realizar a operação "set on less than" (ver Figura C.5.11); o desenho inferior possui uma saída direta do somador para a comparação "less than", chamada Set. (Veja, no Exercício 3.24, como calcular o overflow com menos entradas.)

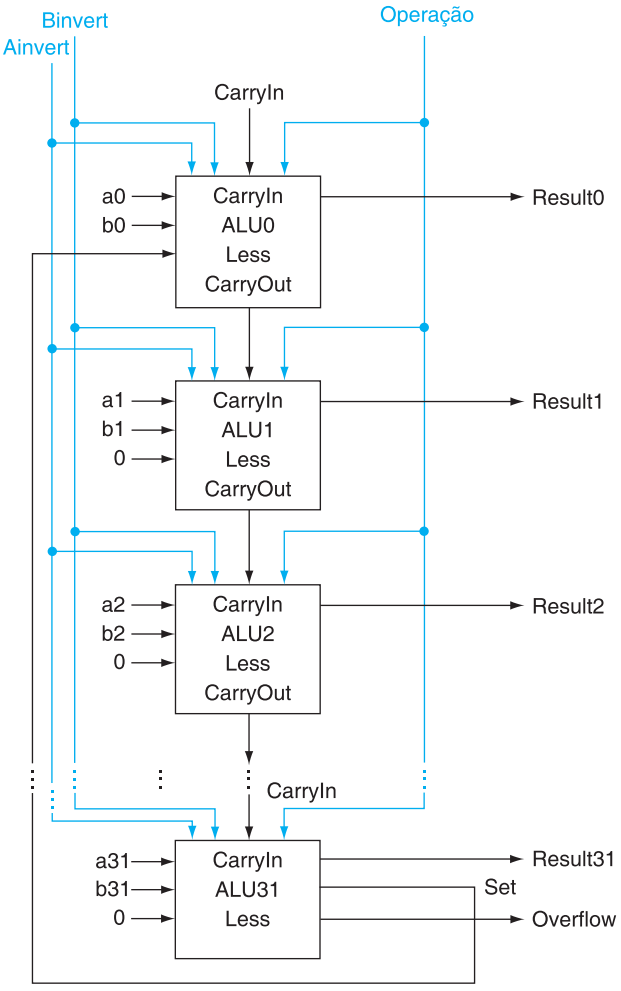


FIGURA C.5.11 Uma ALU de 32 bits construída a partir de 31 cópias da ALU de 1 bit na parte superior da Figura C.5.10 e uma ALU de 1 bit na parte inferior dessa figura. As entradas Less são conectadas a 0, exceto para o bit menos significativo, que está conectado à saída Set do bit mais significativo. Se a ALU realizar a – b e selecionarmos a entrada 3 no multiplexador da Figura C.5.10, então Result = 0 ... 001 se a < b, e Result = 0 ... 000 caso contrário.

Assim, precisamos de uma nova ALU de 1 bit, para o bit mais significativo, a qual possui um bit de saída extra: a saída do somador. O desenho inferior da Figura C.5.10 mostra o projeto, com essa nova linha de saída do somador chamada *Set*, e usada apenas para `slt`. Como precisamos de uma ALU especial para o bit mais significativo, acrescentamos a lógica de detecção de overflow, pois também está associada a esse bit.

Infelizmente, o teste de "less than" é um pouco mais complicado do que acabamos de descrever, devido ao overflow, conforme exploramos nos exercícios. A Figura C.5.11 mostra a ALU de 32 bits.

Observe que toda vez que quisermos que a ALU subtraia, colocamos CarryIn e Binvert em 1. Para adições ou operações lógicas, queremos que as duas linhas de controle sejam 0. Portanto, podemos simplificar o controle da ALU combinando CarryIn e Binvert a uma única linha de controle, chamada *Bnegate*.

Para ajustar ainda mais a ALU ao conjunto de instruções do MIPS, temos de dar suporte a instruções de desvio condicional. Essas instruções desviam se dois registradores forem iguais ou se forem diferentes. O modo mais fácil de testar a igualdade com a ALU é subtrair b de a e depois testar se o resultado é zero, pois

$$(a - b = 0) \Rightarrow a = b$$

Assim, se acrescentarmos hardware para testar se o resultado é 0, podemos testar a igualdade. O modo mais simples é realizar um OR de todas as saídas juntas e depois enviar esse sinal por um inversor:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

A Figura C.5.12 mostra a ALU de 32 bits revisada. Podemos pensar na combinação da linha Ainvert de 1 bit, a linha Binvert de 1 bit, e as linhas de Operação de 2 bits como linhas de controle de 4 bits para a ALU, pedindo que realize soma, subtração, AND, OR ou "set on less than". A Figura C.5.13 mostra as linhas de controle da ALU e a operação ALU correspondente.

Finalmente, agora que vimos o que há dentro de uma ALU de 32 bits, usaremos o símbolo universal para uma ALU completa, como mostra a Figura C.5.14.

Definindo a ALU MIPS em Verilog

A Figura C.5.15 mostra como uma ALU combinacional do MIPS poderia ser especificada em Verilog; essa especificação provavelmente seria compilada com uma biblioteca de partes padrão, que oferecesse um somador, que poderia ser instanciado. Para completar, mostramos o controle da ALU para o MIPS na Figura C.5.16, que usamos no Capítulo 4, onde montamos uma versão Verilog do caminho de dados do MIPS.

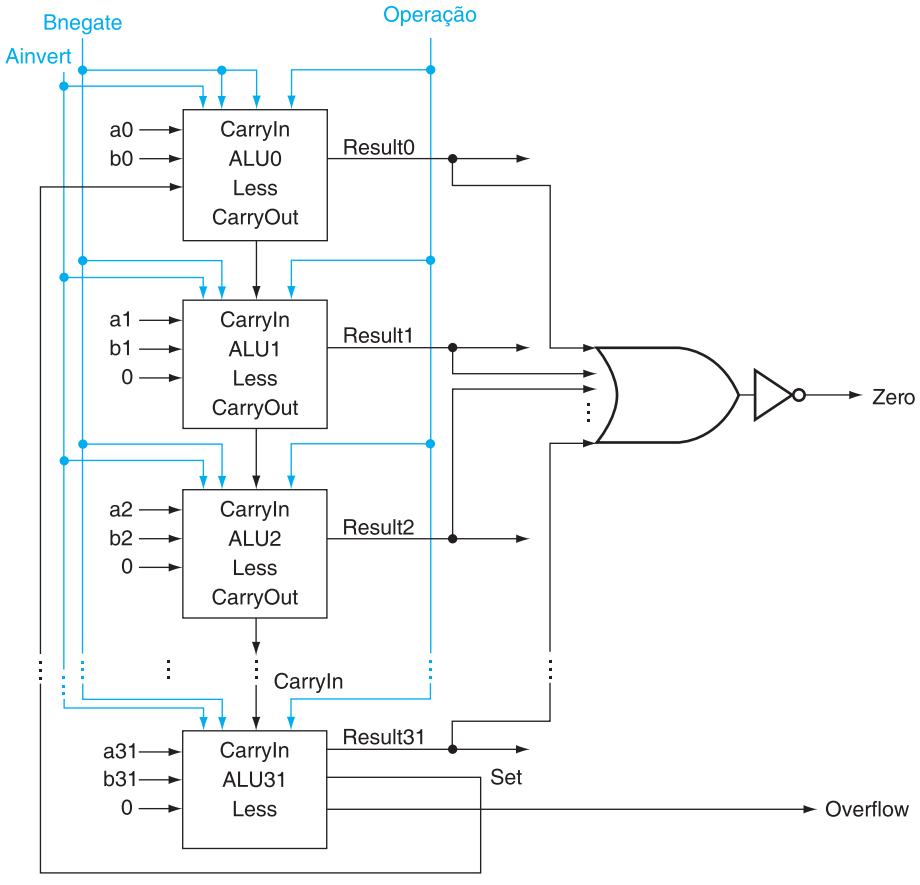


FIGURA C.5.12 A ALU final de 32 bits. Isso acrescenta um detetor de zero à Figura C.5.11.

Linhas de controle ALU	Função
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

FIGURA C.5.13 Os valores das três linhas de controle ALU Bnegate e Operação e as operações ALU correspondentes.

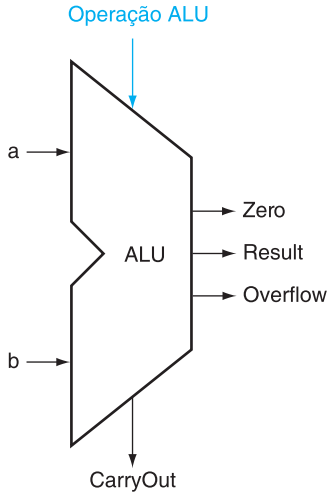


FIGURA C.5.14 **O símbolo normalmente usado para representar uma ALU, como mostra a Figura C.5.12.** Esse símbolo também é usado para representar um somador, de modo que normalmente é rotulado com ALU ou Adder (somador).

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;

   assign Zero = (ALUOut==0); // Zero é true se ALUOut é 0
   always @(ALUctl, A, B) begin // reavalia se isso mudar
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A – B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // resultado é nor
         default: ALUOut <= 0;
      endcase
   end
endmodule
```

FIGURA C.5.15 **Definição comportamental em Verilog de uma ALU MIPS.**

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;
   always case (FuncCode)
   32: ALUOp<=2;  // soma
   34: ALUOp<=6;  // subtrai
   36: ALUOP<=0;  // and
   37: ALUOp<=1;  // or
   39: ALUOp<=12; // nor
   42: ALUOp<=7;  // slt
   default: ALUOp<=15; // não deverá acontecer
   endcase
endmodule
```

FIGURA C.5.16 **O controle ALU do MIPS: um peça simples da lógica de controle combinacional.**

A próxima pergunta é: com que rapidez essa ALU pode somar dois operandos de 32 bits? Podemos determinar as entradas a e b, mas a entrada CarryIn depende da operação no somador de 1 bit adjacente. Se traçarmos todo o caminho pela cadeia de dependências, conectamos o bit mais significativo ao bit menos significativo, de modo que o bit mais significativo da soma precisa esperar pela avaliação *sequencial* de todos os 32 somadores de 1 bit. Essa reação em cadeia sequencial é muito lenta para ser usada no hardware de tempo crítico. A próxima seção explora como agilizar a adição. Esse assunto não é fundamental para a compreensão do restante do apêndice e pode ser pulado.

Verifique você mesmo

Suponha que você queira acrescentar a operação NOT (a AND b), chamada NAND. Como a ALU poderia mudar para dar suporte a ela?

1. Nenhuma mudança. Você pode calcular NAND rapidamente usando a ALU atual, pois $(\overline{a \cdot b}) = (\overline{a} + \overline{b})$ e já temos NOT a, NOT b, e OR.
2. Você precisa expandir o multiplexador grande para acrescentar outra entrada e depois acrescentar nova lógica para calcular NAND.

C.6 Adição mais rápida: Carry Lookahead

A chave para agilizar a adição é determinar o "carry in" para os bits mais significativos mais cedo. Existem diversos esquemas para antecipar o carry, de modo que o cenário de pior caso é uma função do $\log_2$ do número de bits do somador. Esses sinais antecipatórios são mais rápidos porque passam por menos portas em sequência, mas são necessárias muito mais portas para antecipar o carry apropriado.

Uma chave para entender os esquemas de carry rápido é lembrar que, diferente do software, o hardware executa em paralelo sempre que as entradas mudam.

Carry rápido usando hardware "infinito"

Conforme mencionamos anteriormente, qualquer equação pode ser representada em dois níveis de lógica. Como as únicas entradas externas são os dois operandos e o CarryIn para o bit menos significativo do somador, em teoria, poderíamos calcular os valores de CarryIn para todos os bits restantes do somador com apenas dois níveis de lógica.

Por exemplo, o CarryIn para o bit 2 do somador é exatamente o CarryOut do bit 1, de modo que a fórmula é

$$\text{CarryIn2} = (b1 \cdot \text{CarryIn1}) + (a1 \cdot \text{CarryIn1}) + (a1 \cdot b1)$$

De modo semelhante, CarryIn1 é definido como

$$\text{CarryIn1} = (b0 \cdot \text{CarryIn0}) + (a0 \cdot \text{CarryIn0}) + (a0 \cdot b0)$$

Usando a abreviação mais curta e mais tradicional de *ci* para CarryIn*i*, podemos reescrever as fórmulas como

$$c2 = (b1 \cdot c1) + (a1 \cdot c1) + (a1 \cdot b1)$$
$$c1 = (b0 \cdot c0) + (a0 \cdot c0) + (a0 \cdot b0)$$

Substituindo a definição de c1 para a primeira equação, o resultado é esta fórmula:

$$c2 = (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot c0) + (a1 \cdot b0 \cdot c0)$$
$$+ (b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot c0) + (b1 \cdot b0 \cdot c0) + (a1 \cdot b1)$$

Você pode imaginar como a equação se expande à medida que chegamos a bits mais significativos do somador; ela cresce rapidamente com o número de bits. Essa complexidade é refletida no custo do hardware para o carry rápido, tornando esse esquema simples tremendamente dispendioso para somadores largos.

Carry rápido usando o primeiro nível de abstração: propagar e gerar

A maior parte dos esquemas de carry rápido limita a complexidade das equações para simplificar o hardware, enquanto ainda causa melhorias de velocidade substanciais em relação ao carry por ondulação. Um esquema desse tipo é um *somador carry lookahead*. No Capítulo 1, dissemos que os sistemas computacionais enfrentam a complexidade usando níveis de abstração. Um somador carry lookahead conta com níveis de abstração em sua implementação.

Vamos fatorar a equação original como uma primeira etapa:

$$\begin{aligned} c_{i+1} &= (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i) \\ &= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i \end{aligned}$$

Se tivéssemos de reescrever a equação para c2 usando essa fórmula, veríamos alguns padrões repetidos:

$$c2 = (a1 \cdot b1) + (a1 + b1) \cdot ((a0 \cdot b0) + (a0 + b0) \cdot c0$$

Observe o surgimento repetido de (a_i . b_i) e ($a_i + b_i$) na fórmula anterior. Esses dois fatores importantes são tradicionalmente chamados *gerar* (g_i) e *propagar* (p_i):

$$g_i = a_i \cdot b_i$$
$$p_i = a_i + b_i$$

Usando-os para definir c_{i+1}, obtemos

$$c_{i+1} = g_i + p_i \cdot c_i$$

Para ver de onde os sinais recebem seus nomes, suponha que g_i seja 1. Então

$$c_{i+1} = g_i + p_i \cdot c_i = 1 + p_i \cdot c_i = 1$$

Ou seja, o somador *gera* um CarryOut (c_{i+1}) independente do valor de CarryIn (c_i). Agora, suponha que g_i seja 0 e p_i seja 1. Então

$$c_{i+1} = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i$$

Ou seja, o somador *propaga* CarryIn para um CarryOut. Juntando os dois, CarryIn*i+1* é 1 se g*i* for 1 ou se p*i* for 1 e CarryIn*i* for 1.

Por analogia, imagine uma fileira de dominós encostados um no outro. O dominó da outra ponta pode ser empurrado para mais longe desde que não existam intervalos entre eles. De modo semelhante, um carry out pode se tornar verdadeiro por uma geração distante desde que todas as propagações entre eles sejam verdadeiras.

Contando com as definições de propagar e gerar como nosso primeiro nível de abstração, podemos expressar o sinal CarryIn de forma mais econômica. Vamos mostrá-lo para 4 bits:

$$
\begin{aligned}
c1 &= g0 + (p0 \cdot c0)\\
c2 &= g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)\\
c3 &= g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)\\
c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)\\
&\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)
\end{aligned}
$$

Essas equações representam apenas o bom senso: CarryIn*i* é 1 se algum somador anterior gerar um carry e todos os somadores intermediários propagarem um carry. A Figura C.6.1 usa um encanamento para tentar explicar o carry lookahead.

Até mesmo essa forma simplificada leva a grandes equações e, portanto, uma lógica considerável, mesmo para um somador de 16 bits. Vamos tentar prosseguir para dois níveis de abstração.

Carry rápido usando o segundo nível de abstração

Primeiro, consideramos esse somador de 4 bits com sua lógica de carry lookahead como um único bloco de montagem. Se os conectarmos no padrão de carry por ondulação para formar um somador de 16 bits, a soma será mais rápida do que a original, com um pouco mais de hardware.

Para ir mais rápido, precisaremos do carry lookahead em um nível mais alto. Para realizar o carry lookahead para somadores de 4 bits, precisamos propagar e gerar sinais nesse nível mais alto. Aqui, eles são para os quatro blocos somadores de 4 bits:

$$
\begin{aligned}
P0 &= p3 \cdot p2 \cdot p1 \cdot p0\\
P1 &= p7 \cdot p6 \cdot p5 \cdot p4\\
P2 &= p11 \cdot p10 \cdot p9 \cdot p8\\
P3 &= p15 \cdot p14 \cdot p13 \cdot p12
\end{aligned}
$$

Ou seja, o sinal de "super" propagação para a abstração de 4 bits (P*i*) é verdadeiro somente se cada um desses bits no grupo propagar um carry.

Para o sinal de "super" geração (G*i*), nos importamos apenas se houver um carry out do bit mais significativo do grupo de 4 bits. Isso obviamente ocorre se a geração for verdadeira para esse bit mais significativo; ela também ocorre se uma geração anterior for verdadeira *e* todas as propagações intermediárias, incluindo aquela do bit mais significativo, também forem verdadeiras:

$$
\begin{aligned}
G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)\\
G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)\\
G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)\\
G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)
\end{aligned}
$$

A Figura C.6.2 atualiza nossa analogia de encanamento para mostrar P0 e G0.

Então, as equações nesse nível de abstração mais alto para o carry in para cada grupo de 4 bits do somador de 16 bits (C1, C2, C3, C4 na Figura C.6.3) são muito semelhantes às equações de carry out para cada bit do somador de 4 bits (c1, c2, c3, c4) na página C-32:

$$C1 = G0 + (P0 \cdot c0)$$
$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$
$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$
$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

A Figura C.6.3 mostra somadores de 4 bits conectados com tal unidade de carry-lookahead. Os exercícios exploram as diferenças de velocidade entre esses esquemas de carry, diferentes notações para a propagação e geração de sinais em múltiplos bits, e o projeto de um somador de 64 bits.

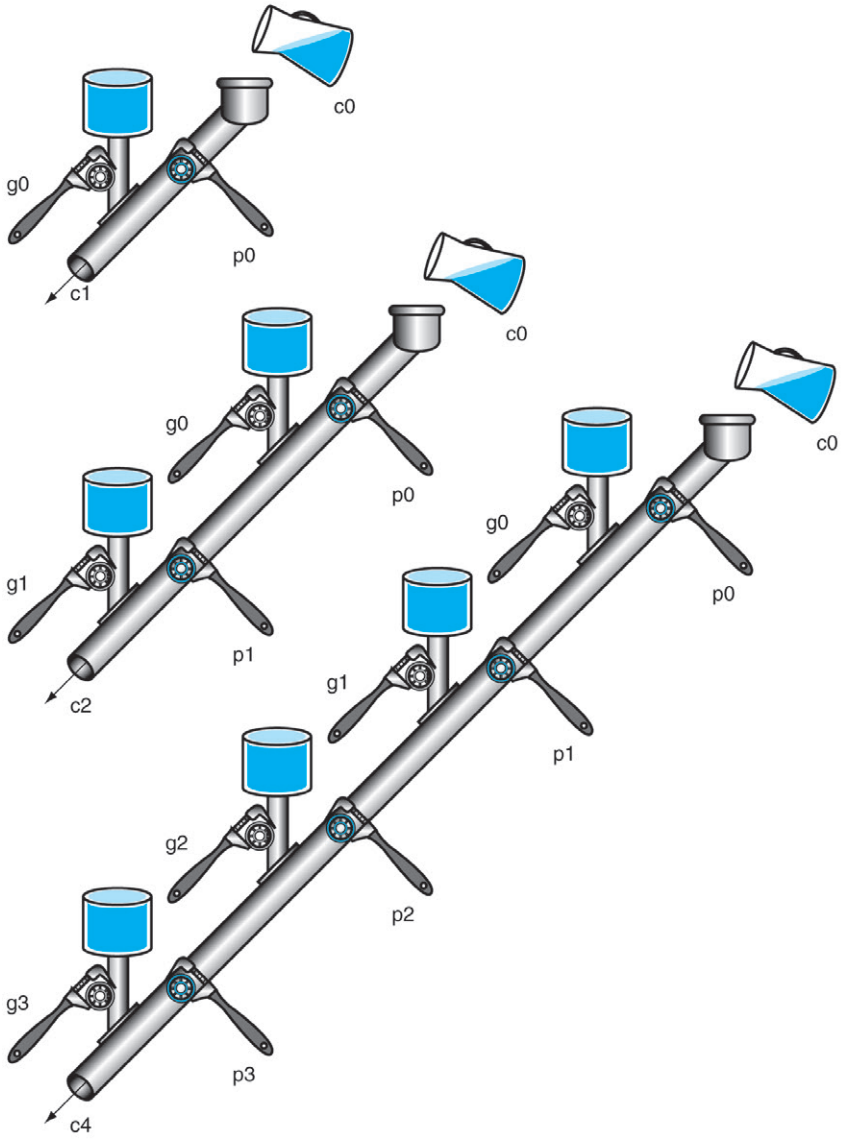


FIGURA C.6.1 Uma analogia de encanamento para o carry lookahead para 1 bit, 2 bits e 4 bits, usando canos d'água e registros. As chaves são viradas para abrir e fechar os registros. A água aparece em destaque. A saída do encanamento (*ci + 1*) estará completa se o valor gerado mais próximo (*gi*) estiver aberto ou se o valor de propagação *i* (*pi*) estiver aberto e houver fluxo de água acima, seja de um gerador anterior, ou propagado com água por trás dele. O CarryIn (c0) pode resultar em um carry out sem a ajuda de quaisquer gerações, mas com a ajuda de todas as propagações.

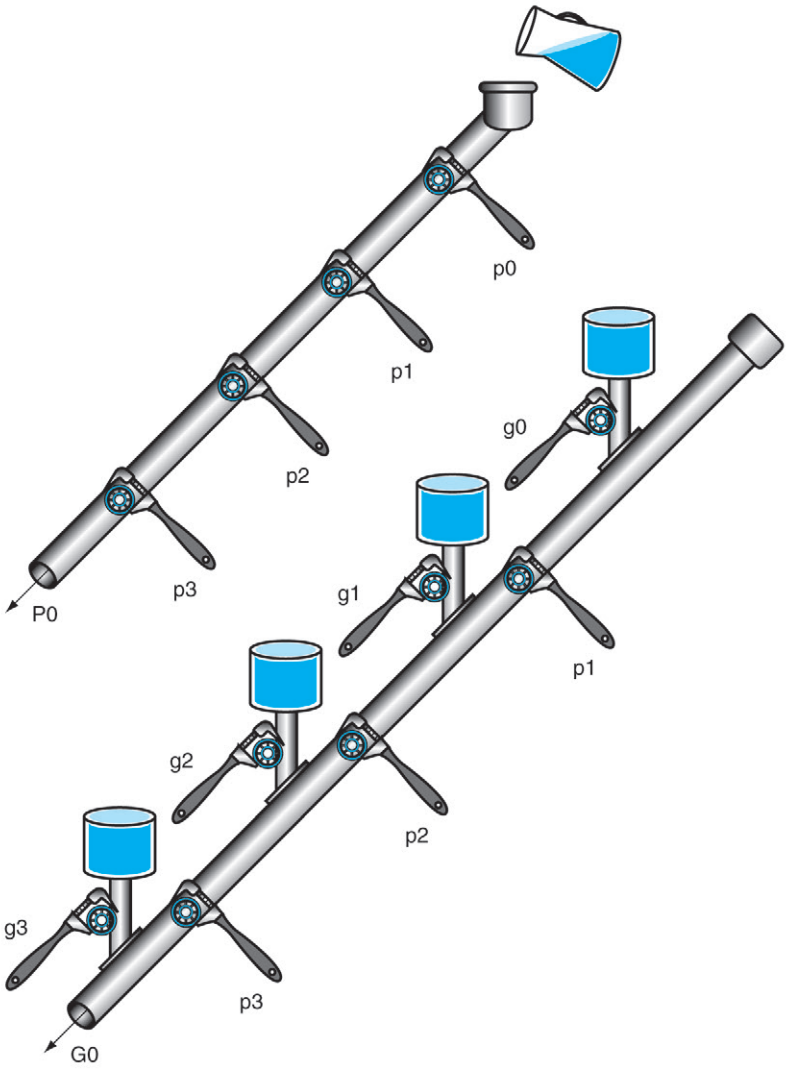


FIGURA C.6.2 Uma analogia de encanamento para os sinais de carry lookahead de próximo nível P0 e G0. P0 é aberto apenas se todas as quatro propagações (p*i*) estiverem abertas, enquanto a água flui em G0 somente se pelo menos uma geração (g*i*) estiver aberta e todas as propagações de fluxo abaixo, a partir dessa geração, estiverem abertas.

Níveis de propagação e geração

EXEMPLO

Determine os valores de g*i*, p*i*, P*i* e G*i* destes dois números de 16 bits:

```
a:      0001 1010 0011 0011bin
b:      1110 0101 1110 1011bin
```

Além disso, qual é o CarryOut15 (C4)?

RESPOSTA

O alinhamento dos bits facilita ver os valores de geração g*i* (a*i* . b*i*) e propagação p*i* (a*i* + b*i*):

```
a:      0001 1010 0011 0011
b:      1110 0101 1110 1011
gi:     0000 0000 0010 0011
pi:     1111 1111 1111 1011
```

onde os bits são numerados de 15 a 0, da esquerda para a direita. Em seguida, as "super" propagações (P3, P2, P1, P0) são simplesmente o AND das propagações de nível inferior:

$$P3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$

Os "super" geradores são mais complexos; portanto, use as seguintes equações:

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0$$
$$G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1$$
$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0$$
$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0$$

Finalmente, CarryOut15 é

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0)$$
$$+ (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0)$$
$$= 0 + 0 + 1 + 0 + 0 = 1$$

Logo, existe um carry out quando se somam esses dois números de 16 bits.

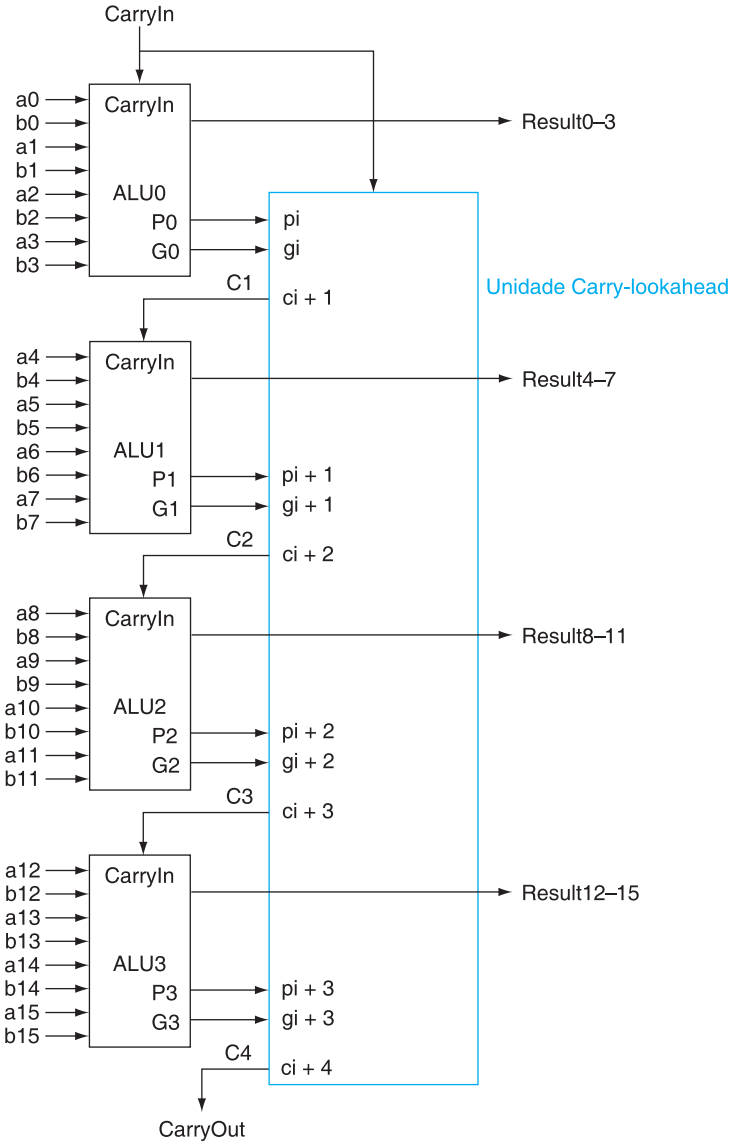


FIGURA C.6.3 Quatro ALUs de 4 bits usando carry lookahead para formar um somador de 16 bits. Observe que os carries vêm da unidade de carry lookahead, e não das ALUs de 4 bits.

O motivo pelo qual o carry lookahead pode tornar os carries mais rápidos é que toda a lógica começa a avaliação no momento em que o ciclo de clock começa, e o resultado não mudará quando a saída de cada porta deixar de mudar. Tomando um atalho de passar por menos portas para enviar o carry no sinal, a saída das portas deixará de mudar mais cedo, e, por isso, o tempo para o somador pode ser menor.

Para apreciar a importância do carry lookahead, precisamos calcular o desempenho relativo entre ele e os somadores de carry por ondulação.

Velocidade do carry por ondulação versus carry lookahead

EXEMPLO

Um modo simples de modelar o tempo para a lógica é considerar que cada porta AND ou OR leva o mesmo tempo para um sinal passar por ela. O tempo é estimado simplesmente contando-se o número de portas ao longo do caminho por uma parte da lógica.
Compare o número de *atrasos de porta* para os caminhos de dois somadores de 16 bits, um usando o carry por ondulação e um usando o carry lookahead em dois níveis.

RESPOSTA

A Figura C.5.5 mostra que o sinal carry out utiliza dois atrasos de porta por bit. Depois, o número de atrasos de porta entre um carry in para o bit menos significativo e o carry out do mais significativo é $16 \times 2 = 32$.

Para o carry lookahead, o carry out do bit mais significativo é apenas C4, definido no exemplo. São necessários dois níveis de lógica para especificar C4 em termos de P*i* e G*i* (o OR de vários termos AND). P*i* é especificado em um nível de lógica (AND) usando p*i*, e G*i* é especificado em dois níveis usando p*i* e g*i*, de modo que o pior caso para esse próximo nível de abstração são dois níveis de lógica. p*i* e g*i* são um nível de lógica cada um, definido em termos de a*i* e b*i*. Se assumirmos que um atraso de porta para cada nível de lógica nessas equações, o pior caso é $2 + 2 + 1 = 5$ atrasos de porta.

Logo, para o caminho de carry in para carry out, a adição de 16 bits por um somador carry lookahead é seis vezes mais rápida, usando essa estimativa muito simples da velocidade do hardware.

Resumo

O carry lookahead oferece um caminho mais rápido do que esperar que os carries ondulem por todos os 32 somadores de 1 bit. Esse caminho mais rápido é pavimentado por dois sinais, gerar e propagar. O primeiro cria um carry independente da entrada do carry, e o outro passa um carry adiante. O carry lookahead também oferece outro exemplo de como a abstração é importante no projeto de computadores para lidar com a complexidade.

Verifique você mesmo

Usando a estimativa simples da velocidade de hardware (que acabamos de ver) com atrasos de porta, qual é o desempenho relativo de uma adição de 8 bits com carry por ondulação *versus* uma adição de 64 bits usando a lógica de carry lookahead?

1. Um somador de 64 bits com carry lookahead é três vezes mais rápido: adições de 8 bits possuem atrasos de 16 portas, e adições de 64 bits possuem atrasos de 7 portas.
2. Elas têm praticamente a mesma velocidade, pois adições de 64 bits precisam de mais níveis de lógica no somador de 16 bits.
3. Adições de 8 bits são mais rápidas do que 64 bits, mesmo com carry lookahead.

Detalhamento: agora, consideramos todas menos uma das operações lógicas e aritmética para o conjunto de instruções principal do MIPS: a ALU na Figura C.5.14 omite o suporte às instruções de deslocamento. Seria possível ampliar o multiplexador da ALU para incluir um deslocamento à esquerda de 1 bit ou um deslocamento à direita de 1 bit. Mas os projetistas de hardware criaram um circuito chamado *barrel shifter*, que pode deslocar de 1 a 32 bits não em mais tempo do que é preciso para somar dois números de 32 bits, de modo que o deslocamento normalmente é feito fora da ALU.

Detalhamento: a equação lógica para a saída de Sum do somador completo na página C-16 pode ser expressa de forma mais simples usando uma porta mais poderosa do que AND e OR. Uma porta *OR exclusiva* é verdadeira se os dois operandos divergirem, ou seja,

$$x \neq y \Rightarrow 1 \text{ and } x == y \Rightarrow 0$$

Em algumas tecnologias, o OR exclusivo é mais eficiente do que dois níveis de portas AND e OR. Usando o símbolo para representar o OR exclusivo, aqui está a nova equação:

$$\text{Sum} = a \oplus b \oplus \text{CarryIn}$$

Além disso, desenhamos a ALU da maneira tradicional, usando portas. Os computadores são projetados hoje em transistores CMOS, que são basicamente chaves. A ALU CMOS e os barrel shifters tiram proveito dessas chaves e podem ter menos multiplexadores do que aparece em nossos projetos, mas os princípios de projeto são semelhantes.

Detalhamento: usar minúsculas e maiúsculas para distinguir a hierarquia de símbolos de geração e propagação não funciona quando você tem mais de dois níveis. Uma notação alternativa que funciona é $g_{i..j}$ e $p_{i..j}$ para os sinais de geração e propagação para os bits i a j. Assim, $g_{1..1}$ é a geração para o bit 1, $g_{4..1}$ é a geração para os bits de 4 a 1, e $g_{16..1}$ é para os bits de 16 a 1.

C.7 Clocks

Antes de discutirmos sobre elementos de memória e lógica sequencial, é útil discutir brevemente o tópico de clocks. Esta seção curta introduz o assunto e é semelhante à discussão encontrada na Seção 4.2. Outros detalhes sobre metodologias de clock e temporização são apresentados na Seção C.11.

clocking acionado por transição Um esquema de clocking em que todas as mudanças de estado ocorrem em uma transição do clock.

metodologia de clocking A técnica usada para determinar quando os dados são válidos e estáveis em relação ao clock.

Clocks são necessários na lógica sequencial para decidir quando um elemento que contém estado deve ser atualizado. Um clock é simplesmente um sinal de execução livre com um *tempo de ciclo* fixo; a *frequência de clock* é simplesmente o inverso do tempo de ciclo. Como vemos na Figura C.7.1, o *tempo de ciclo de clock* ou *período de clock* é dividido em duas partes: quando o clock é alto e quando o clock é baixo. Neste texto, usamos apenas o **clocking acionado por transição**. Isso significa que todas as mudanças de estado ocorrem em uma transição do clock. Usamos uma metodologia acionada por transição porque é mais simples de explicar. Dependendo da tecnologia, essa pode ou não ser a melhor escolha para uma **metodologia de clocking**.

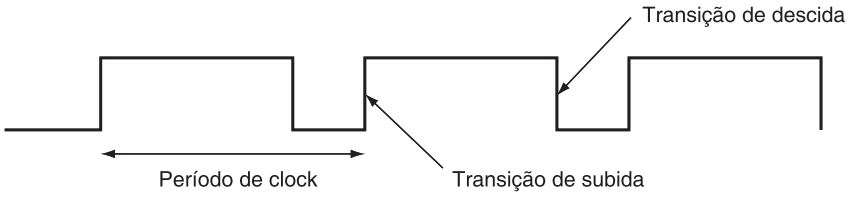


FIGURA C.7.1 Um sinal de clock oscila entre valores alto e baixo. O período de clock é o tempo para um ciclo completo. Em um projeto acionado por transição, a transição de subida ou descida do clock é ativa e faz com que o estado seja alterado.

Em uma metodologia acionada por transição, a transição de subida ou a transição de descida do clock é *ativa* e faz com que haja mudanças de estado. Como veremos na próxima seção, os **elementos de estado** em um projeto acionado por transição são implementados de modo que o conteúdo dos elementos de estado só mudem na transição de clock ativa. A escolha da transição que será ativa é influenciada pela tecnologia de implementação e não afeta os conceitos envolvidos no projeto da lógica.

elementos de estado Um elemento de memória.

A transição do clock atua como uma amostra do sinal, fazendo com que o valor da entrada de dados para um elemento de estado seja amostrado e armazenado no elemento de estado. O uso de um acionador por transição significa que o processo de amostragem é essencialmente instantâneo, eliminando problemas que poderiam ocorrer se os sinais fossem amostrados em momentos ligeiramente diferentes.

A principal restrição em um sistema com clock, também chamado de **sistema síncrono**, é que os sinais escritos nos elementos de estado precisam ser *válidos* quando ocorre a transição de clock ativa. Um sinal é válido se ele for estável (ou seja, não muda), e o valor não mudará novamente até as entradas mudarem. Como os circuitos combinacionais não podem ter feedback, se as entradas de uma unidade lógica combinacional não forem alteradas, as saídas por fim se tornarão válidas.

sistema síncrono Um sistema de memória que emprega clocks e onde os sinais de dados são lidos apenas quando o clock indicar que os valores de sinal são estáveis.

A Figura C.7.2 mostra o relacionamento entre os elementos de estado e os blocos lógicos combinacionais em um projeto lógico síncrono, sequencial. Os elementos de estado, cujas saídas mudam apenas depois da transição do clock, oferecem entradas válidas ao bloco lógico combinacional. Para garantir que os valores escritos nos elementos de estado na transição de clock ativa sejam válidos, o clock precisa ter um período longo o suficiente para que todos os sinais no bloco lógico combinacional se estabilizem, depois a transição do clock pega esses valores para armazenar nos elementos de estado. Essa restrição define um limite inferior sobre o tamanho do período de clock, que precisa ser longo o suficiente para que todas as entradas de elementos de estado sejam válidas.

No restante desse apêndice, bem como no Capítulo 4, normalmente omitimos o sinal de clock, pois estamos supondo que todos os elementos de estado são atualizados na mesma transição de clock. Alguns elementos de estado serão escritos em cada transição de clock, enquanto outros serão escritos apenas sob certas condições (como um registrador sendo atualizado). Nesses casos, teremos um sinal de escrita explícito para esse elemento de estado. O sinal de escrita ainda precisa ser disparado com o clock para que a atualização ocorra apenas na transição do clock se o sinal de escrita estiver ativo.

Uma outra vantagem de uma metodologia acionada por transição é que é possível ter um elemento de estado utilizado como entrada e saída para o mesmo bloco lógico combinacional, como vemos na Figura C.7.3. Na prática, deve-se ter o cuidado de impedir corridas em tais situações e garantir que o período de clock seja longo o suficiente; esse tópico é discutido ainda mais na Seção C.11.

Agora que discutimos como o clocking é utilizado para atualizar elementos de estado, podemos discutir como construir os elementos de estado.

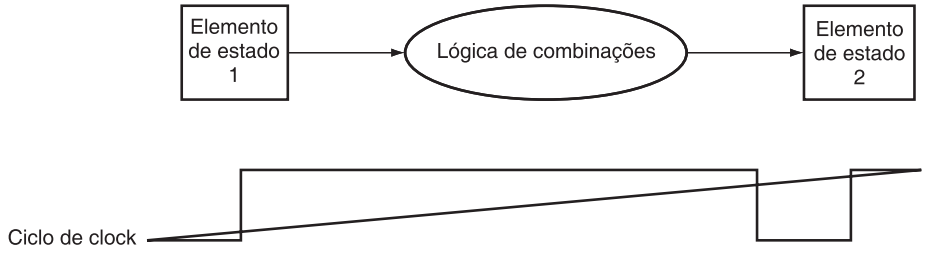


FIGURA C.7.2 As entradas de um bloco lógico combinacional vêm de um elemento de estado, e as saídas são escritas em um elemento de estado. A transição do clock determina quando o conteúdo dos elementos de estado são atualizados.

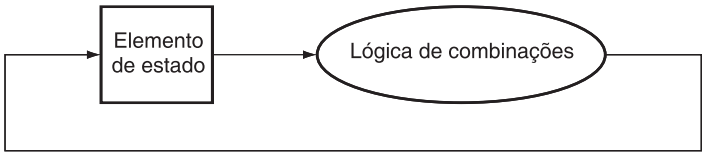


FIGURA C.7.3 Uma metodologia acionada por transição permite que um elemento de estado seja lido e escrito no mesmo ciclo de clock sem criar uma corrida que pudesse levar a valores indeterminados nos dados. Naturalmente, o ciclo de clock ainda precisa ser grande o suficiente para que os valores de entrada sejam estáveis quando ocorrer a transição de clock ativa.

Detalhamento: ocasionalmente, os projetistas acham útil ter uma pequena quantidade de elementos de estado que mudam na transição de clock oposta a partir da maioria dos elementos de estado. Isso exige cuidado extremo, pois tal técnica tem efeitos sobre as entradas e as saídas do elemento de estado. Por que, então, os projetistas fariam isso? Considere o caso em que a quantidade de lógica combinacional antes e depois de um elemento de estado é pequena o suficiente, de modo que cada uma poderia operar em meio ciclo de clock, em vez de um ciclo de clock completo, que é mais comum. Então, o elemento de estado pode ser escrito na transição de clock correspondente a meio ciclo de clock, pois as entradas e saídas serão utilizáveis após meio ciclo de clock. Um lugar comum onde essa técnica é usada é em **bancos de registradores**, onde a simples leitura ou escrita do banco de registradores normalmente pode ser feita em metade do ciclo de clock normal. O Capítulo 6 utiliza essa ideia para reduzir o overhead da técnica de pipelining.

banco de registradores Um elemento de estado que consiste em um conjunto de registradores que podem ser lidos e escritos fornecendo um número de registrador a ser acessado.

C.8 Elementos de memória: flip-flops, latches e registradores

Nesta seção e na seguinte, discutimos os princípios básicos por trás dos elementos de memória, começando com flip-flops e latches, passando para bancos de registradores e finalmente para as memórias. Todos os elementos de memória armazenam estado: a saída de qualquer elemento da memória depende das entradas e do valor armazenado dentro do elemento da memória. Assim, todos os blocos lógicos com um elemento da memória contêm estado e são sequenciais.

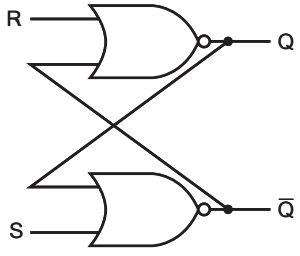


FIGURA C.8.1 Um par de portas NOR cruzadas pode armazenar um valor interno. O valor armazenado no Q de saída é reciclado invertendo-o para obter $\overline{Q}$ e depois invertendo $\overline{Q}$ para obter Q. Se R ou $\overline{Q}$ estiverem ativos, Q será desativado e vice-versa.

Os tipos mais simples de elementos de memória são *sem clock*; ou seja, eles não possuem qualquer entrada de clock. Embora só usemos elementos de memória com clock neste texto, um latch sem clock é o elemento de memória mais simples, de modo que veremos esse circuito primeiro. A Figura C.8.1 mostra um *latch S-R* (latch set-reset), construído a partir de um par de portas NOR (portas OR com saídas invertidas). As saídas Q e $\overline{Q}$ representam o valor do estado armazenado e seu complemento. Quando nem S nem R estão ativos, as portas NOR cruzadas atuam como inversores e armazenam os valores anteriores de Q e $\overline{Q}$.

Por exemplo, se a saída, Q, for verdadeira, então o inversor de baixo produz uma saída falsa (que é $\overline{Q}$), que se torna a entrada para o inversor de cima, que produz uma saída verdadeira, que é Q, e assim por diante. Se S estiver ativa, então a saída Q estará ativa e $\overline{Q}$ estará inativa. Quando S e R estiverem inativas, os últimos valores de Q e $\overline{Q}$ continuarão a ser armazenados na estrutura cruzada. A ativação de S e R simultaneamente pode levar a uma operação incorreta: dependendo de como S e R são desativadas, o latch pode oscilar ou tornar-se metaestável (isso é descrito com mais detalhes na Seção C.11).

Essa estrutura cruzada é a base para elementos de memória mais complexos, que nos permitem armazenar sinais de dados. Esses elementos contêm portas adicionais, usadas para armazenar valores de sinal e fazem com que o estado seja atualizado apenas em conjunto com um clock. A próxima seção mostra como esses elementos são criados.

Flip-flops e latches

Flip-flops e **latches** são os elementos de memória mais simples. Em flip-flops e latches, a saída é igual ao valor do estado armazenado dentro do elemento. Além do mais, diferente do latch S-R descrito anteriormente, todos os latches e flip-flops que usaremos deste ponto em diante são com clock, o que significa que eles têm uma entrada de clock e a mudança de estado é acionada por esse clock. A diferença entre um flip-flop e um latch é o ponto em que o clock faz com que o estado realmente mude. Em um latch com clock, o estado é alterado sempre que as entradas apropriadas mudam e o clock está ativo, enquanto, em um flip-flop, o estado é trocado somente em uma transição de clock. Como em todo este texto usamos uma metodologia de temporização acionada por transição, onde o estado é atualizado apenas em transições de clock, só precisamos usar flip-flops. Os flip-flops normalmente são criados a partir de latches, de modo que começamos descrevendo a operação de um latch com clock simples e depois discutimos a operação de um flip-flop construído a partir desse latch.

flip-flop Um elemento da memória para o qual a saída é igual ao valor do estado armazenado dentro do elemento e para o qual o estado interno é alterado apenas em uma transição do clock.

latch Um elemento da memória em que a saída é igual ao valor do estado armazenado dentro do elemento e o estado é alterado sempre que as entradas apropriadas mudarem e o clock estiver ativo.

Para aplicações computacionais, a função de flip-flops e latches é armazenar um sinal. Um *latch D* ou **flip-flop D** armazena o valor de seu sinal de entrada de dados na memória interna. Embora existam muitos outros tipos de latches e flip-flops, o tipo D é o único bloco de montagem básico de que precisaremos. Um latch D possui duas entradas e duas saídas. As entradas são o valor de dados a ser armazenado (chamado D) e um sinal de clock (chamado C) que indica quando o latch deve ler o valor na entrada D e armazená-lo. As saídas são simplesmente o valor do estado interno (Q) e seu complemento ($\overline{Q}$). Quando a entrada de clock C está ativa, o latch é considerado *aberto*, e o valor da saída (Q) torna-se o valor da entrada D. Quando a entrada do clock C está inativa, o latch é considerado *fechado*, e o valor da saída (Q) é qualquer valor que tenha sido armazenado pela última vez em que o latch foi aberto.

flip-flop D Um flip-flop com uma entrada de dados que armazena o valor desse sinal de entrada na memória interna quando a transição do clock ocorre.

A Figura C.8.2 mostra como um latch D pode ser implementado com duas portas adicionais acrescentadas às portas NOR cruzadas. Visto que, quando o latch é aberto, o valor de Q muda quando D muda, essa estrutura às vezes é chamada de *latch transparente*. A Figura C.8.3 mostra como esse latch D funciona, supondo que a saída Q seja inicialmente falsa e que D mude primeiro.

Como já dissemos, usamos flip-flops como bloco de montagem básico, no lugar de latches. Os flip-flops não são transparentes: suas saídas *só* mudam na transição do clock. Um flip-flop pode ser construído de modo que seja acionado na subida (positivo) ou descida (negativo) da transição do clock; para nossos projetos, podemos usar qualquer tipo. A Figura C.8.4 mostra como um flip-flop D de transição de descida é construído a partir de um par de latches D. Em um flip-flop D, a saída é armazenada quando ocorre a transição do clock. A Figura C.8.5 mostra como opera esse flip-flop.

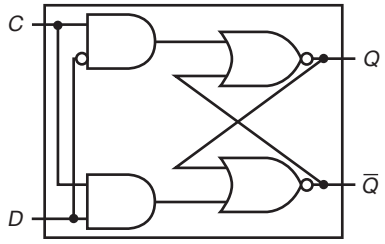


FIGURA C.8.2 Um latch D implementado com portas NOR. Uma porta NOR atua como um inversor se a outra entrada for 0. Assim, o par cruzado de portas NOR atua para armazenar o valor de estado, a menos que a entrada do clock, *C*, esteja ativa, quando o valor da entrada *D* substitui o valor de *Q* e é armazenado. O valor da entrada *D* precisa ser estável quando o sinal de clock *C* mudar de ativo para inativo.

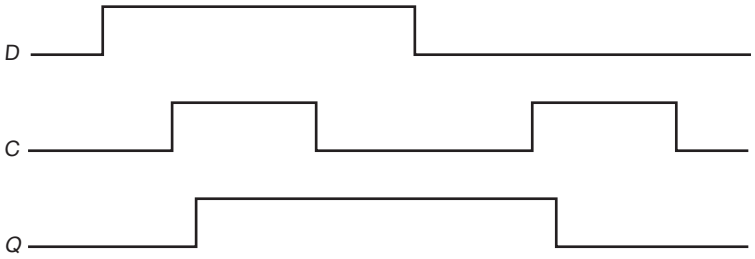


FIGURA C.8.3 Operação de um latch D, assumindo que a saída está inicialmente inativa. Quando o clock, *C*, está ativo, o latch é aberto e a saída *Q* imediatamente assume o valor da entrada *D*.

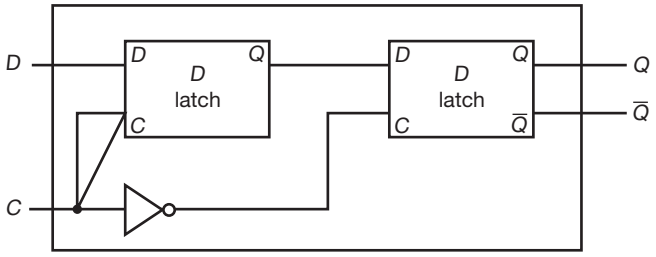


FIGURA C.8.4 Um flip-flop D acionado na transição de descida. O primeiro latch, chamado de mestre, é aberto e acompanha a entrada *D* quando a entrada de clock, *C*, é ativada. Quando a entrada de clock *C* cai, o primeiro latch é fechado, mas o segundo latch, chamado de escravo, é aberto e recebe sua entrada da saída do latch mestre.

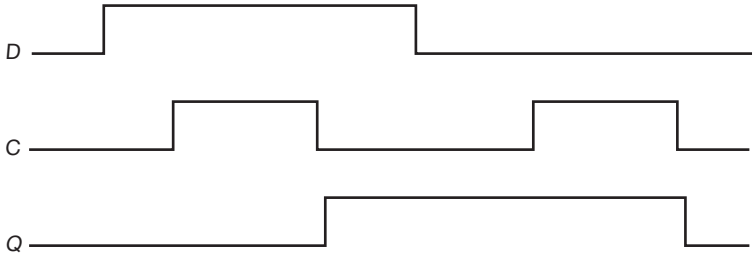


FIGURA C.8.5 Operação de um flip-flop D acionado na transição de descida, assumindo que a saída está inicialmente inativa. Quando a entrada de clock (*C*) muda de ativa para inativa, a saída *Q* armazena o valor da entrada *D*. Compare esse comportamento com o do latch D com clock, mostrado na Figura C.8.3. Em um latch com clock, o valor armazenado e a saída, *Q*, mudam sempre que *C* está alto, ao contrário de somente quando C realiza transições.

Aqui está uma descrição em Verilog de um módulo para um flip-flop D na transição de subida, assumindo que C é a entrada de clock e D é a entrada de dados:

```
module DFF(clock,D,Q,Qbar);
   input clock, D;
   output reg Q; // Q é um reg, pois é atribuído em um bloco always
   output Qbar;
   assign Qbar = ~ Q; // Qbar é sempre exatamente o inverso de Q
   always @(posedge clock) // realiza ações sempre que o clock levanta
      Q = D;
endmodule
```

Como a entrada *D* é pega na transição do clock, ela precisa ser válida por um período imediatamente antes e depois da transição do clock. O tempo mínimo que a entrada precisa ser válida antes da transição do clock é chamado **tempo de preparação**; o tempo mínimo durante o qual ela precisa ser válida após a transição do clock é chamado **tempo de suspensão**. Assim, as entradas de qualquer flip-flop (ou qualquer coisa construída usando flip-flops) precisam ser válidas durante uma janela que começa no tempo de preparação antes da transição do clock e termina no tempo de suspensão após a transição do clock, como mostra a Figura C.8.6. A Seção C.11 fala sobre as restrições de clocking e temporização, incluindo o atraso de propagação por um flip-flop, com mais detalhes.

tempo de preparação O tempo mínimo que a entrada para um dispositivo de memória precisa ser válida antes da transição do clock.

tempo de suspensão O tempo mínimo durante o qual a entrada precisa ser válida após a transição do clock.

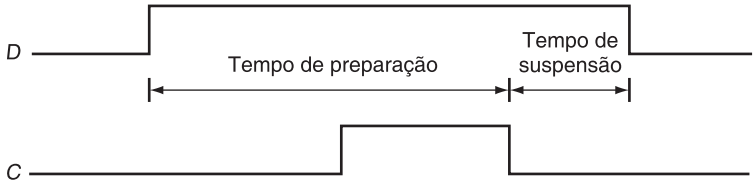


FIGURA C.8.6 Requisitos de tempo de preparação e suspensão para um flip-flop D acionado na transição de descida. A entrada precisa ser estável por um período antes da transição do clock, bem como após a transição do clock. O tempo mínimo que o sinal precisa estar estável antes da transição de clock é chamado de tempo de preparação, enquanto o tempo mínimo que o sinal precisa estar estável após o clock é chamado de tempo de suspensão. Deixar de atender a esses requisitos mínimos pode resultar em uma situação em que a saída do flip-flop pode nem sequer ser previsível, conforme descrevemos na Seção C.11. Os tempos de suspensão normalmente são 0 ou muito pequenos e, portanto, não causam preocupação.

Podemos usar um array de flip-flops D para construir um registrador que possa manter um dado de múltiplos bits, como um byte ou uma word. Usamos registradores por todos os nossos caminhos de dados no Capítulo 4.

Bancos de registradores

Uma estrutura central no nosso caminho de dados é um *banco de registradores.* Consiste em um conjunto de registradores que podem ser lidos e escritos fornecendo um número de registrador a ser acessado. Um banco de registradores pode ser implementado com um decodificador para cada porta de leitura ou escrita e um array de registradores construídos a partir de flip-flops D. Como a leitura de um registrador não muda qualquer estado, só precisamos fornecer um número de registrador como entrada, e a única saída será os dados contidos nesse registrador. Para escrever em um registrador, precisaremos de três entradas: um número de registrador, os dados a escrever e um clock que controla a escrita no registrador. No Capítulo 4, usamos um banco de registradores com duas portas de leitura e uma porta de escrita. Ele é desenhado como mostra a Figura C.8.7. As portas de leitura podem ser implementadas com um par de multiplexadores, cada um tão largo quanto o número de bits em cada registrador do banco de registradores. A Figura C.8.8 mostra a implementação de duas portas de leitura de registrador para um banco de registradores de 32 bits.

A implementação da porta de escrita é ligeiramente mais complexa, pois só podemos mudar o conteúdo do registrador designado. Podemos fazer isso usando um decodificador para gerar um sinal que possa ser usado para determinar qual registrador escrever. A Figura C.8.9 mostra como implementar a porta de escrita para um banco de registradores. É importante lembrar que o flip-flop só muda de estado na transição do clock. No Capítulo 4, conectamos sinais de escrita para o banco de registradores explicitamente e assumimos que o clock mostrado na Figura C.8.9 está anexado implicitamente.

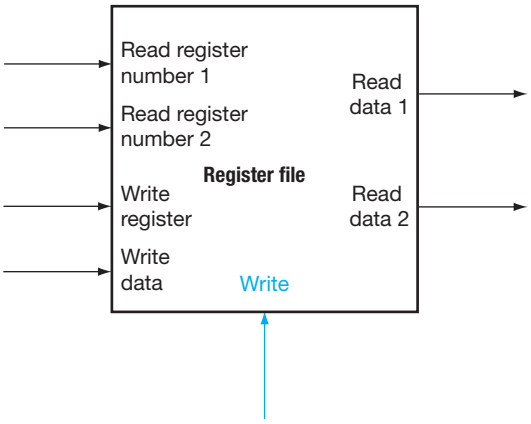


FIGURA C.8.7 Um banco de registradores com duas portas de leitura e uma porta de escrita possui cinco entradas e duas saídas. A entrada de controle Write aparece em destaque na parte inferior.

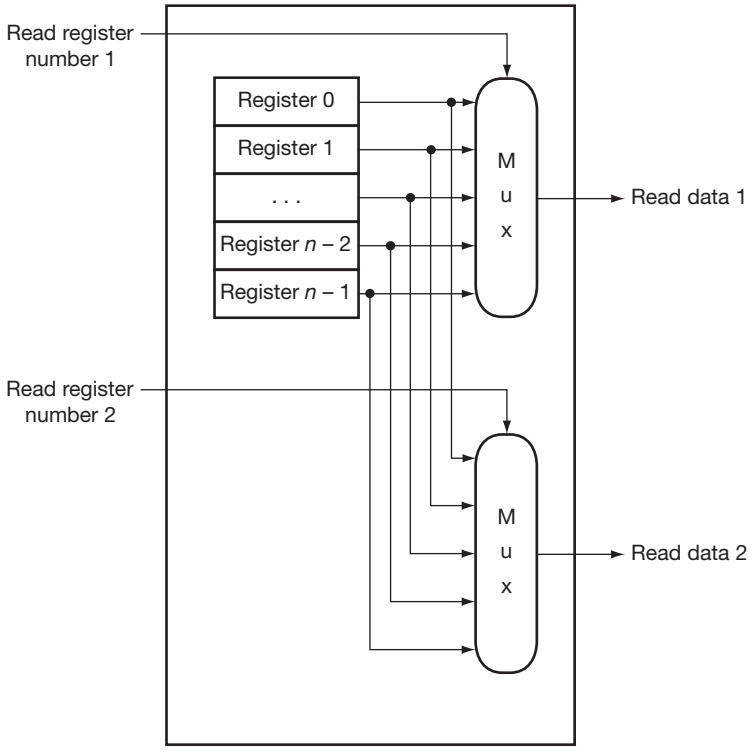


FIGURA C.8.8 A implementação de duas portas de leitura para um banco de registradores com *n* registradores pode ser feita com um par de multiplexadores *n*-para-1, cada um com 32 bits de largura. O sinal do número do registrador de leitura é usado como sinal seletor do multiplexador. A Figura C.8.9 mostra como a porta de escrita é implementada.

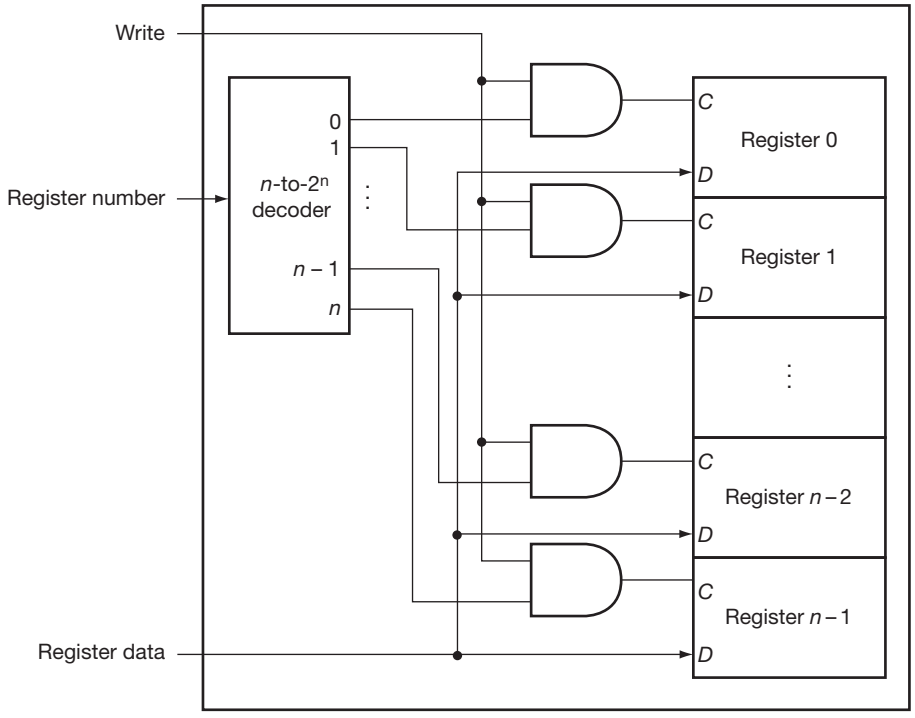


FIGURA C.8.9 A porta de escrita para um banco de registradores é implementada com um decodificador que é usado com o sinal de escrita (Write) para gerar a entrada *C* dos registradores. Todas as três entradas (o número do registrador, os dados e o sinal de escrita) terão restrições de tempo de preparação e suspensão que garantem que os dados corretos são escritos no banco de registradores.

O que acontece se o mesmo registrador for lido e escrito durante um ciclo de clock? Como a escrita no banco de registradores ocorre na transição do clock, o registrador será válido durante o tempo em que for lido, como vimos anteriormente na Figura C.7.2. O valor retornado será o valor escrito em um ciclo de clock anterior. Se quisermos que uma leitura retorne o valor atualmente sendo escrito, uma lógica adicional no banco de registradores ou fora dele será necessária. O Capítulo 4 utiliza muito dessa lógica.

Especificando a lógica sequencial em Verilog

Para especificar a lógica sequencial em Verilog, temos de entender como gerar um clock, como descrever quando um valor é escrito em um registrador e como especificar o controle sequencial. Vamos começar especificando um clock. Um clock não é um objeto predefinido em Verilog; em vez disso, geramos um clock usando a notação `#n` da Verilog antes de uma instrução; isso causa um atraso de `n` etapas de tempo de simulação antes da execução da instrução. Na maioria dos simuladores Verilog, também é possível gerar um clock como uma entrada externa, permitindo que o usuário especifique em tempo de simulação o número de ciclos de clock para executar uma simulação.

O código na Figura C.8.10 implementa um clock simples que é alto ou baixo para uma unidade de simulação e depois muda de estado. Usamos a capacidade de atraso e atribuição bloqueante para implementar o clock.

```
reg clock; // clock é um registrador
always
   #1 clock = 1; #1 clock = 0;
```

FIGURA C.8.10 Uma especificação de um clock.

Em seguida, precisamos ser capazes de especificar a operação de um registrador acionado por transição. Em Verilog, isso é feito usando a lista de sensitividade em um bloco `always` e especificando como acionador a transição positiva ou negativa de uma variável binária com a notação `posedge` ou `negedge`, respectivamente. Logo, o código Verilog a seguir faz com que o registrador A seja escrito com o valor b na transição positiva do clock:

```
reg [31:0] A;
wire [31:0] b;

always @(posedge clock) A <= b;
```

No decorrer deste capítulo e nas seções Verilog do Capítulo 4, consideraremos um projeto acionado por transição positiva. A Figura C.8.11 mostra uma especificação Verilog de um banco de registradores MIPS que considera duas leituras e uma escrita, com apenas a escrita possuindo clock.

```
module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,Data1,Data2,clock);

   input [5:0] Read1,Read2,WriteReg; // os números dos registradores para leitura
e escrita
   input [31:0] WriteData; // dados para escrever
   input RegWrite, // O controle de escrita
     clock; // o clock para acionar a escrita
   output [31:0] Data1, Data2; // os valores dos registradores lidos
   reg [31:0] RF [31:0]; // 32 registradores cada um com 32 bits

   assign Data1 = RF[Read1];
   assign Data2 = RF[Read2];

   always begin
      // escreve no registrador o novo valor se Regwrite for alto
      @(posedge clock) if (RegWrite) RF[WriteReg] <= WriteData;
   end
endmodule
```

FIGURA C.8.11 Um banco de registradores MIPS escrito em Verilog comportamental. Esse banco de registradores escreve na transição de subida do clock.

Verifique você mesmo

No código Verilog para o banco de registradores da Figura C.8.11, as portas de saída correspondentes aos registradores lidos são atribuídas por meio de uma atribuição contínua, mas o registrador sendo escrito é atribuído em um bloco `always`. Qual dos seguintes é o motivo?

a. Não existe um motivo especial. Isso simplesmente foi conveniente.

b. Porque Data1 e Data2 são portas de saída e WriteData é uma porta de entrada.

c. Porque a leitura é um evento combinacional, enquanto a escrita é um evento sequencial.

C.9 Elementos de memória: SRAMs e DRAMs

SRAM (Static Random Access Memory) Uma memória na qual os dados são armazenados estaticamente (como nos flip-flops), e não dinamicamente (como na DRAM). SRAMs são mais rápidas do que as DRAMs, mas menos densas e mais caras por bit.

Registradores e bancos de registradores oferecem os blocos de montagem básicos para pequenas memórias, mas quantidades maiores de memória são construídas usando **SRAMs (Static Random Access Memories)** ou *DRAMs* (Dynamic Random Access Memories). Primeiro, discutimos sobre SRAMs, que são mais simples, e depois passamos para DRAMs.

SRAMs

SRAMs são simplesmente circuitos integrados que são arrays de memória com (normalmente) uma única porta de acesso que pode ser de leitura ou escrita. SRAMs possuem um tempo de acesso fixo a qualquer dado, embora as características de leitura e escrita sejam diferentes. Um chip de SRAM possui uma configuração específica em termos do número de locais endereçáveis, bem como a largura de cada local endereçável. Por exemplo, uma SRAM de 4M × 8 oferece 4M entradas, cada uma com 8 bits. Assim, ela terá 22 linhas de entrada (pois $4M = 2^{22}$), uma linha de saída de dados de 8 bits, e uma única linha de entrada de dados de 8 bits. Assim como as ROMs, o número de locais endereçáveis é chamado de *altura*, com o número de bits por unidade chamado de *largura*. Por diversos motivos técnicos, as SRAMs mais novas e mais rápidas normalmente estão disponíveis em configurações estreitas: × 1 e × 4. A Figura C.9.1 mostra os sinais de entrada e saída para uma SRAM de 2M × 16.

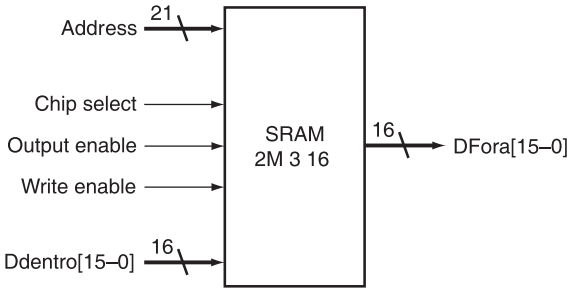


FIGURA C.9.1 **Uma SRAM de 2M x 16 mostrando as 21 linhas de endereço (2MK = 221) e 16 entradas de dados, as três linhas de controle e as 16 saídas de dados.**

Para iniciar um acesso de leitura ou escrita, o sinal Chip select precisa ser ativado. Para leituras, também temos de ativar o sinal Output enable que controla se o dado selecionado pelo endereço foi levado para os pinos. O Output enable é útil para conectar várias memórias a um barramento de saída único e usar Output enable para determinar qual memória é levada ao barramento. O tempo de acesso de leitura da SRAM costuma ser especificado como o espaço de tempo entre o Output enable ser verdadeiro e as linhas de endereço estarem válidas até o momento em que os dados estão nas linhas de saída. Os tempos de acesso de leitura típicos para SRAMs em 2004 variavam desde 2-4ns para as partes mais rápidas da CMOS, que costumam ser um pouco menores e mais estreitas, até 8-20ns para as partes maiores típicas, que em 2004 tinham mais de 32 milhões de bits de dados. A demanda por SRAMs de menor potência para produtos eletrônicos e aparelhos digitais cresceu bastante nos últimos cinco anos; essas SRAMs possuem potências de stand-by e acesso muito menores, mas normalmente são de 5-10 vezes mais lentas. Mais recentemente, as SRAMs síncronas – semelhantes às DRAMs síncronas, que discutimos na próxima seção –, também têm sido desenvolvidas.

Para a escrita, temos de fornecer os dados a serem escritos e o endereço, além dos próprios sinais que causarão a escrita. Quando Write enable e Chip select são verdadeiros, os dados nas linhas de entrada de dados são escritos na célula especificada pelo endereço. Existem requisitos de tempo de preparação e tempo de suspensão para as linhas de endereço e dados, assim como existiam para flip-flops D e latches. Além disso, o sinal Write enable não é uma transição de clock, mas um pulso com um requisito de largura mínima. O tempo para concluir uma escrita é especificado pela combinação entre tempos de preparação, tempos de suspensão e a largura do pulso Write enable.

SRAMs grandes não podem ser construídas da mesma maneira como construímos um banco de registradores porque, diferente de um banco de registradores, no qual um multiplexador 32 para 1 poderia ser prático, o multiplexador de 64K para 1 que seria necessário

para uma SRAM de 64K × 1 seria totalmente inviável. Em vez de usar um multiplexador gigante, memórias grandes são implementadas com uma linha de saída compartilhada, chamada *linha de bit*, que múltiplas células de memória no array de memória podem ativar. Para permitir que múltiplas origens possam colocar um sinal em uma única linha, é utilizado um *buffer tristate*. Um buffer tristate possui duas entradas – um sinal de dados e um Output enable –, e uma única saída que está em um dos três estados: ativa, inativa ou alta impedância. A saída de um buffer tristate é igual ao sinal de entrada de dados, ativo ou inativo, se o Output enable estiver ativo; caso contrário, está em um *estado de alta impedância*, permitindo que outro buffer tristate cujo Output enable esteja ativo determine o valor de uma saída compartilhada.

A Figura C.9.2 mostra um conjunto de buffers tristate ligados para formar um multiplexador com uma entrada decodificada. É fundamental que o Output enable de no máximo um dos buffers tristate esteja ativo; caso contrário, os buffers tristate podem tentar colocar valores diferentes na linha de saída. Usando buffers tristate nas células individuais da SRAM, cada célula que corresponde a uma saída em particular pode compartilhar a mesma linha de saída. O uso de um conjunto de buffers tristate é incorporado nos flip-flops que formam as células básicas da SRAM. A Figura C.9.3 mostra como uma SRAM pequena de 4 × 2 poderia ser criada, usando latches D com uma entrada chamada Enable, que controla a saída tristate.

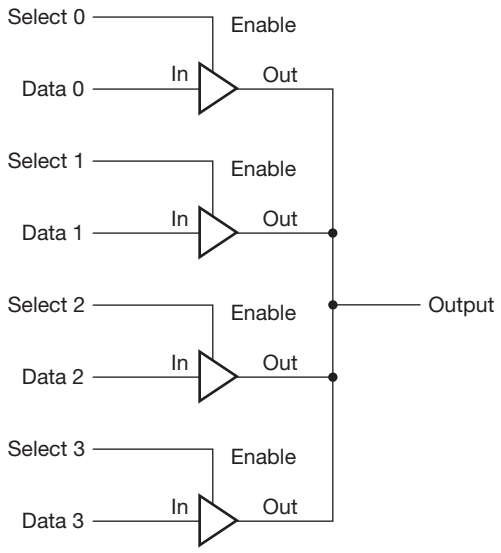


FIGURA C.9.2 Quatro buffers tristate são usados para formar um multiplexador. Somente uma das quatro entradas Select pode estar ativa. Um buffer tristate com um Output enable inativo possui uma saída de alta impedância que permite um buffer tristate, cujo Output enable está ativo, ativar a linha de saída compartilhada.

O projeto na Figura C.9.3 elimina a necessidade de um multiplexador enorme; porém, ainda exige um decodificador muito grande e um número correspondentemente grande de linhas de words. Por exemplo, em uma SRAM 4M × 8, precisaríamos de um decodificador de 2 para 4M e 4M linhas de words (que são as linhas usadas para habilitar os flip-flops individuais)! Para contornar esse problema, as memórias grandes são organizadas como arrays retangulares e utilizam um processo de decodificação em duas etapas. A Figura C.9.4 mostra como uma SRAM de 4M × 8 poderia ser organizada internamente usando uma decodificação em duas etapas. Conforme veremos, o processo de decodificação em dois níveis é muito importante para entender a operação das DRAMs.

Recentemente, vimos o desenvolvimento de SRAMs síncronas (SSRAMs) e DRAMs síncronas (SDRAMs). A principal capacidade oferecida pelas RAMs síncronas é a capacidade de transferir uma *rajada* de dados a partir de uma série de endereços sequenciais dentro

de um array ou linha. A rajada é definida por um endereço inicial, fornecido no padrão normal, e um tamanho de rajada. A vantagem de velocidade das RAMs síncronas vem da capacidade de transferir os bits na rajada sem ter de especificar bits de endereço adicionais. Em vez disso, um clock é usado para transferir os bits sucessivos na rajada. A eliminação da necessidade de especificar o endereço para as transferências dentro da rajada melhora bastante a taxa para transferir o bloco de dados. Devido a essa capacidade, as SRAMs e DRAMs síncronas estão rapidamente se tornando as RAMs preferidas para a criação de sistemas de memória nos computadores. Discutimos o uso de DRAMs síncronas em um sistema de memória com mais detalhes na próxima seção e no Capítulo 5.

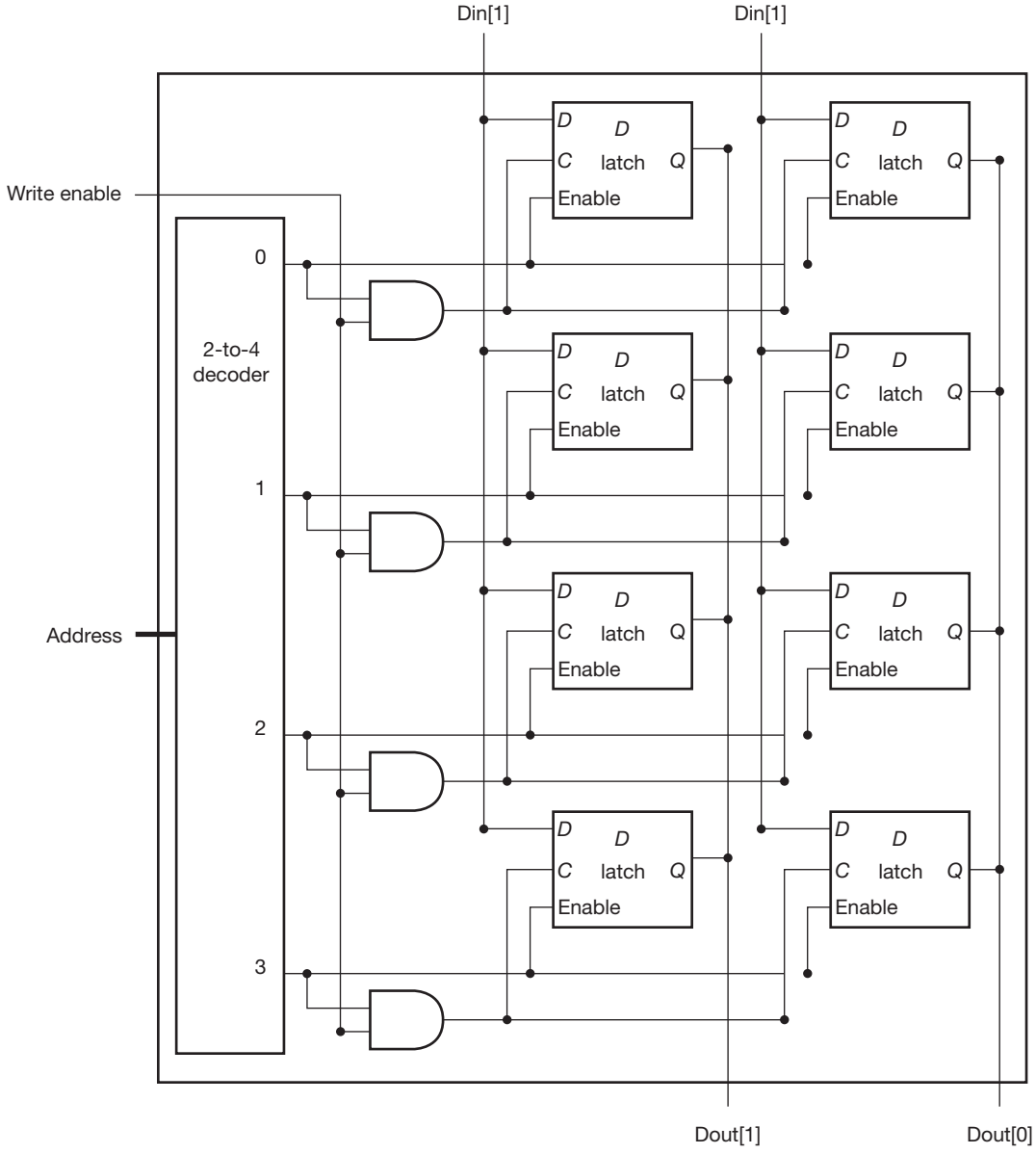


FIGURA C.9.3 A estrutura básica de uma SRAM 4 × 2 consiste em um decodificador que seleciona qual par de células ativar. As células ativadas utilizam uma saída tristate conectada às linhas de bit verticais que fornecem os dados requisitados. O endereço que seleciona a célula é enviado em um de um conjunto de linhas de endereço horizontais, chamadas linhas de words. Para simplificar, os sinais Output enable e Chip select foram omitidos, mas facilmente poderiam ser incluídos com algumas portas AND.

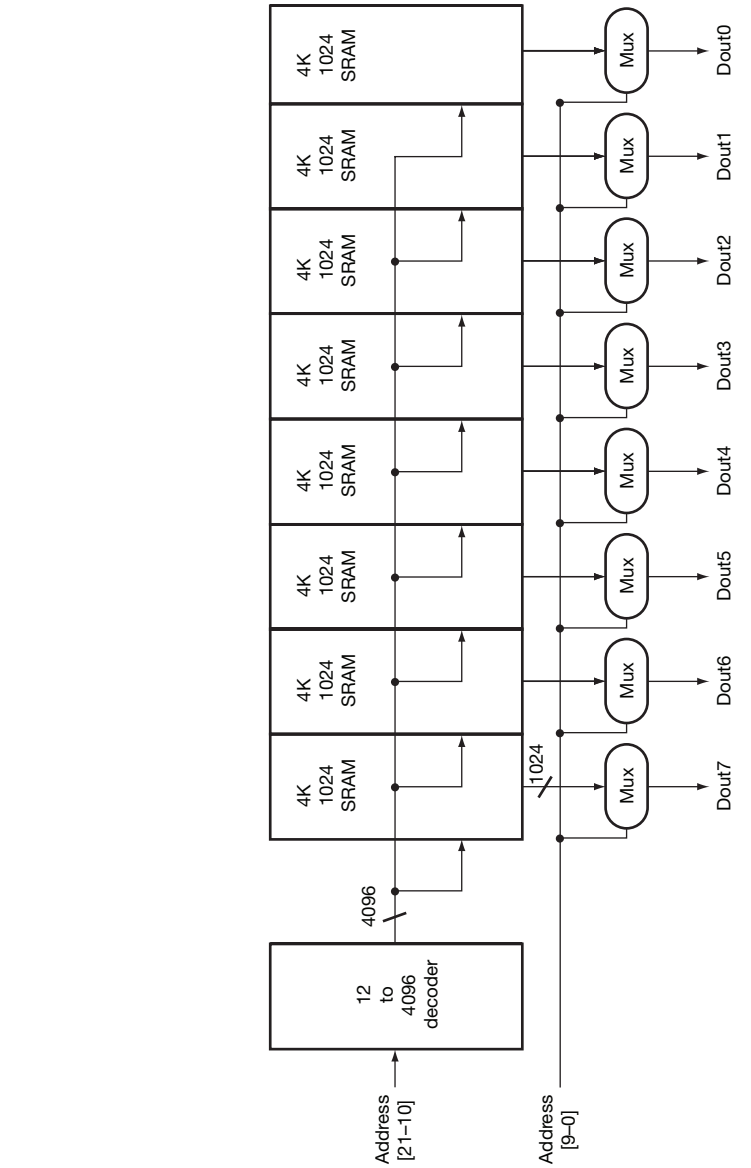


FIGURA C.9.4 Organização típica de uma SRAM 4M × 8 como um array de arrays 4K × 1024. O primeiro decodificador gera os endereços para oito arrays 4K × 1.024; depois, um conjunto de multiplexadores é utilizado para selecionar 1 bit de cada array de 1.024 bits de largura. Esse é um projeto muito mais fácil do que um decodificador de único nível, que precisaria de um decodificador enorme ou de um multiplexador gigantesco. Na prática, uma SRAM moderna desse tamanho provavelmente usaria um número ainda maior de blocos, cada um deles um pouco menor.

DRAMs

Em uma RAM estática (SRAM), o valor armazenado em uma célula é mantido em um par de portas inversoras, e desde que haja energia sendo aplicada, o valor pode ser mantido indefinidamente. Em uma RAM dinâmica (DRAM), o valor mantido em uma célula é armazenado como uma carga em um capacitor. Um único transistor é utilizado para acessar essa carga armazenada, ou para ler o valor ou para escrever sobre a carga armazenada lá. Como as DRAMs utilizam apenas um único transistor por bit de armazenamento, eles são muito mais densos e mais baratos por bit. Em comparação, as SRAMs exigem quatro a seis transistores por bit. Nas DRAMs, a carga é armazenada em um capacitor, de modo que não pode ser mantida indefinidamente e precisa sofrer *refresh* periodicamente. É por isso que essa estrutura de memória é chamada de *dinâmica*, ao contrário do armazenamento estático em uma célula de SRAM.

Para renovar a célula, lemos seu conteúdo e o escrevemos de volta. A carga pode ser mantida por vários milissegundos, o que poderia corresponder a algo perto de um milhão de ciclos de clock. Hoje, controladores de memória de único chip normalmente tratam da função de refresh de modo independente do processador. Se cada bit tivesse de ser lido da DRAM e depois escrito de volta individualmente, com as DRAMs grandes contendo vários megabytes, estaríamos sempre realizando refresh na DRAM, sem deixar tempo para acessá-la. Felizmente, as DRAMs também usam uma estrutura de decodificação de dois níveis, e isso nos permite realizar refresh em uma linha inteira (que compartilha uma linha de words) com um ciclo de leitura seguido por um ciclo de escrita. Em geral, as operações de refresh consomem 1% a 2% dos ciclos ativos da DRAM, deixando os 98% a 99% restantes dos ciclos disponíveis para leitura e escrita de dados.

Detalhamento: como uma DRAM lê e escreve o sinal armazenado em uma célula? O transistor dentro da célula é uma chave, chamada *transistor de passagem*, que permite que o valor armazenado no capacitor seja acessado para leitura ou escrita. A Figura C.9.5 mostra como é a célula de único transistor. O transistor de passagem atua como uma chave: quando o sinal na linha de words está ativo, a chave é fechada, conectando o capacitor à linha de bits. Se a operação for de escrita, então o valor a ser escrito é colocado na linha de bits. Se o valor for 1, o capacitor será carregado. Se o valor for 0, então o capacitor será descarregado. A leitura é um pouco mais complexa, pois a DRAM precisa detectar uma carga muito pequena armazenada no capacitor. Antes de ativar a linha de words para uma leitura, a linha de bits é carregada com uma voltagem que é a metade entre a voltagem baixa e alta. Depois, ativando a linha de words, a carga no capacitor é lida na linha de bits. Isso faz com que a linha de bits mude ligeiramente para a direção alta ou baixa, e essa mudança é detectada com um amplificador de sensibilidade, que pode detectar pequenas mudanças na voltagem.

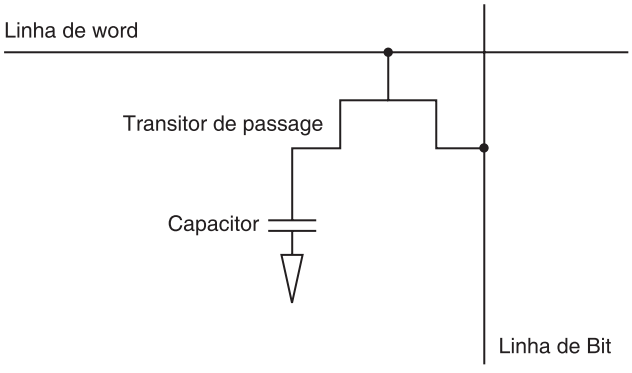


FIGURA C.9.5 Uma célula de DRAM com único transistor contém um capacitor que armazena o conteúdo da célula e um transistor usado para acessar a célula.

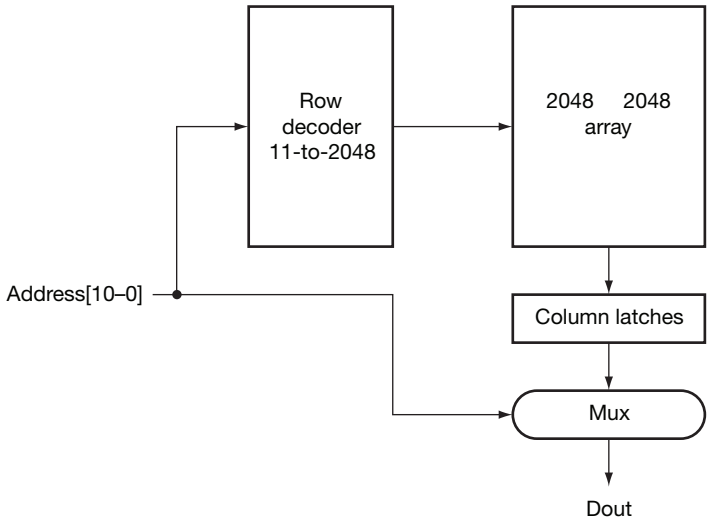


FIGURA C.9.6 Uma DRAM de 4M × 1 é criada com um array de 2048 × 2048. O acesso à linha usa 11 bits para selecionar uma linha, que é então guardada em 2048 latches de 1 bit. Um multiplexador escolhe o bit de saída a partir desses 2048 latches. Os sinais RAS e CAS controlam se as linhas de endereço são enviadas ao decodificador de linhas ou multiplexador de colunas.

As DRAMs utilizam um decodificador de dois níveis consistindo em um *acesso de linha* seguido por um *acesso de coluna*, como mostra a Figura C.9.6. O acesso de linha escolhe uma dentre diversas linhas e ativa a linha de words correspondente. O conteúdo de todas as colunas na linha ativa é armazenado em um conjunto de latches. O acesso à coluna, então, seleciona os dados dos latches de coluna. Para economizar pinos e reduzir o custo do pacote, as mesmas linhas de endereço são usadas para o endereço de linha e coluna; um par de sinais, chamados RAS (Row Access Strobe) e CAS (Column Access Strobe), é utilizado para sinalizar a DRAM que um endereço de linha ou coluna está sendo fornecido. O refresh é realizado simplesmente lendo as colunas nos latches de coluna e depois escrevendo os mesmos valores de volta. Assim, uma linha inteira sofre refresh em um ciclo. O esquema de endereçamento de dois níveis, combinado com o circuito interno, torna os tempos de acesso típicos da DRAM muito maiores (por um fator de 5-10) do que os tempos de acesso da SRAM. Em 2004, os tempos de acesso da DRAM típicos variavam de 45 a 65ns; DRAMs de 256Mbits estão em plena produção, e as primeiras amostras ao cliente das DRAMs de 1GB estavam disponíveis no primeiro trimestre de 2004. Um custo muito menor por bit torna a DRAM a melhor escolha para a memória principal, enquanto o tempo de acesso mais rápido torna a SRAM a melhor escolha para as caches.

Você poderia observar que uma DRAM de 64M × 4 na realidade acessa 8Kbits em cada acesso de linha e depois joga fora tudo menos 4 deles durante um acesso de coluna. Os projetistas da DRAM utilizaram uma estrutura interna da DRAM como um meio de oferecer maior largura de banda a partir de uma DRAM. Isso é feito permitindo que o endereço de coluna mude sem mudar o endereço de linha, resultando em um acesso a outros bits nos latches de coluna. Para tornar esse processo mais rápido e mais preciso, as entradas de endereço utilizam clocks, levando à forma dominante de DRAM em uso atualmente: DRAM ou SDRAM síncrona.

Desde 1999, mais ou menos, as SDRAMs são o chip de memória preferido da maioria dos sistemas de memória principal com cache. As SDRAMs oferecem acesso rápido a uma série de bits dentro de uma linha, transferindo sequencialmente todos os bits em uma rajada sob o controle de um sinal de clock. Em 2004, as DDRRAMs (Double Data Rate RAMs), chamadas "double data rate" porque transferem dados nas transições de subida e descida de um clock fornecido externamente, eram a forma mais utilizada das SDRAMs. Conforme discutimos no Capítulo 5, essas transferências de alta velocidade podem ser usadas para aumentar a largura de banda disponível a partir da memória principal para corresponder às necessidades do processador e das caches.

Correção de erros

Devido ao potencial para adulteração de dados em memórias grandes, a maioria dos sistemas computacionais utiliza algum tipo de código de verificação de erro para detectar a possível adulteração de dados. Um código simples bastante utilizado é o *código de paridade*. Em um código de paridade, o número de 1s em uma word é contado; a word tem paridade ímpar se o número de 1s for ímpar; caso contrário, a paridade é par. Quando uma word é escrita na memória, o bit de paridade também é escrito (1 para ímpar, 0 para par). Depois, quando a word é lida, o bit de paridade é lido e verificado. Se a paridade da word da memória e o bit de paridade armazenado não combinarem, então houve um erro.

Um esquema de paridade de 1 bit pode detectar no máximo 1 bit de erro em um item de dados; se houve 2 bits de erro, então um esquema de paridade de 1 bit não detectará qualquer erro, pois a paridade não combinará os dados com dois erros. (Na realidade, um esquema de paridade de 1 bit pode detectar qualquer número ímpar de erros; porém, a probabilidade de ter três erros é muito menor do que a probabilidade de ter dois; portanto, na prática, um código de paridade de 1 bit é limitado a detectar um único bit de erro.) Naturalmente, um código de paridade não pode dizer qual bit em um item de dados está errado.

Um esquema de paridade de 1 bit é um **código de detecção de erro**; há também *códigos de correção de erro* (ECC) que detectarão e permitirão a correção de um erro. Para grandes memórias principais, muitos sistemas utilizam um código que permite a detecção de até 2 bits de erro e a correção de um único bit de erro. Esses códigos funcionam usando mais bits para codificar os dados; por exemplo, os códigos típicos utilizados para as memórias principais exigem 7 ou 8 bits para cada 128 bits de dados.

código de detecção de erro Um código que permite a detecção de um erro nos dados, mas não o local exato, e portanto nem a correção do erro.

Detalhamento: um código de paridade de 1 bit é um *código de distância 2*, o que significa que, se olharmos para os dados mais o bit de paridade, nenhuma mudança de 1 bit é suficiente para gerar outra combinação válida dos dados mais a paridade. Por exemplo, se mudarmos um bit nos dados, a paridade estará errada, e vice-versa. Naturalmente, se mudarmos 2 bits (ou 2 bits de dados ou 1 bit de dados e o bit de paridade), a paridade corresponderá aos dados e o erro não poderá ser detectado. Logo, existe uma distância de dois entre as combinações válidas da paridade e dos dados.

Para detectar mais de um erro ou corrigir um erro, precisamos de um *código de distância 3*, que tem a propriedade de que qualquer combinação válida dos bits no código de correção de erro e os dados ter pelo menos 3 bits diferindo de qualquer outra combinação. Suponha que tenhamos tal código e que tenhamos um erro nos dados. Nesse caso, o código mais os dados ficará 1 bit de distância de uma combinação válida e podemos corrigir os dados para essa combinação válida. Se tivermos dois erros, podemos reconhecer que existe um erro, mas não podemos corrigir os erros. Vejamos um exemplo. Aqui estão as words de dados e um código de correção de erros de distância 3 para um item de dados de 4 bits.

Dados	Bits de código	Dados	Bits de código
0000	000	1000	111
0001	011	1001	100
0010	101	1010	010
0011	110	1011	001
0100	110	1100	001
0101	101	1101	010
0110	011	1110	100
0111	000	1111	111

Para entender como isso funciona, vamos escolher uma word de dados, digamos, 0110, cujo código de correção de erro é 011. Aqui estão as quatro possibilidades de erro de 1 bit para esses dados: 1110, 0010, 0100 e 0111. Agora veja o item de dados com o mesmo código (011), que é a entrada com o valor 0001. Se o decodificador de correção de erro recebesse uma das quatro words de dados possíveis com erro, ele teria de escolher entre corrigir para 0110 ou 0001. Embora essas quatro words com erro tenham apenas 1 bit trocado do padrão correto, 0110, elas têm 2 bits diferentes da correção alternativa, 0001. Logo, o mecanismo de correção de erro pode facilmente escolher a correção para 0110, pois um único erro é uma probabilidade muito maior. Para ver se

dois erros podem ser detectados, basta observar que todas as combinações com 2 bits trocados possuem um código diferente. A única reutilização do mesmo código é com 3 bits diferentes, mas, se corrigirmos um erro de 2 bits, corrigiremos para o valor errado, pois o decodificador irá supor que ocorreu apenas um único erro. Se quisermos corrigir erros de 1 bit e detectar, mas não corrigir erroneamente os erros de 2 bits, precisamos de um código com distância 4.

Embora distinguíssemos entre o código e os dados em nossa explicação, na verdade, um código de correção de erro trata a combinação de código e dados como uma única word em um código maior (7 bits neste exemplo). Assim, ele lida com erros nos bits de código da mesma forma como os erros nos bits de dados.

Embora o exemplo anterior exija $n - 1$ bits para n bits de dados, o número de bits exigido cresce lentamente, de modo que, para um código de distância 3, uma word de 64 bits precisa de 7 bits e uma word de 128 bits precisa de 8. Esse tipo de código é chamado de *código de Hamming*, devido a R. Hamming, que descreveu um método para a criação de tal código.

C.10 Máquinas de estados finitos

máquina de estados finitos Uma função lógica sequencial consistindo em um conjunto de entradas e saídas, uma função de próximo estado que mapeia o estado atual e as entradas para um novo estado, e uma função de saída que mapeia o estado atual e possivelmente as entradas para um conjunto de saídas ativas.

função de próximo estado Uma função combinacional que, dadas as entradas e o estado atual, determina o próximo estado de uma máquina de estados finitos.

Como vimos anteriormente, os sistemas lógicos digitais podem ser classificados como combinacionais ou sequenciais. Os sistemas sequenciais contêm estado armazenado em elementos da memória internos ao sistema. Seu comportamento depende do conjunto de entradas fornecidas e do conteúdo da memória interna, ou do estado do sistema. Assim, um sistema sequencial não pode ser descrito com uma tabela verdade. Em vez disso, um sistema sequencial é descrito como uma **máquina de estados finitos** (ou, normalmente, apenas *máquina de estados*). Uma máquina de estados finitos possui um conjunto de estados e duas funções, chamadas **função de próximo estado** e a *função de saída*. O conjunto de estados corresponde a todos os valores possíveis do armazenamento interno. Assim, se houver n bits de armazenamento, haverá 2^n estados. A função de próximo estado é uma função combinacional que, dadas as entradas e o estado atual, determina o próximo estado do sistema. A função de saída produz um conjunto de saídas a partir do estado atual e suas entradas. A Figura C.10.1 mostra isso em forma de diagrama.

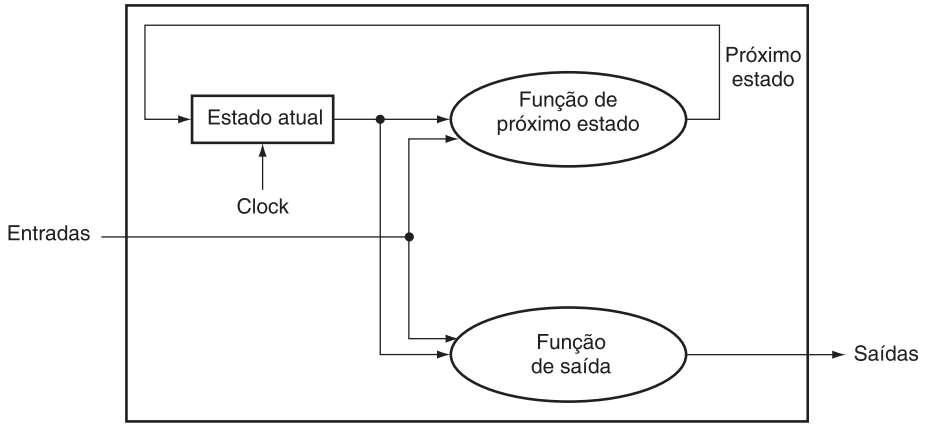


FIGURA C.10.1 Uma máquina de estados consiste em armazenamento interno que contém o estado e duas funções combinacionais: a função de próximo estado e a função de saída. Normalmente, a função de saída é restrita a usar apenas o estado atual como sua entrada; isso não muda a capacidade de uma máquina sequencial, mas afeta seu funcionamento interno.

As máquinas de estados que discutimos aqui e no Capítulo 4 são *síncronas*. Isso significa que o estado muda junto com o ciclo de clock, e um novo estado é calculado uma vez a cada clock. Assim, os elementos de estado são atualizados apenas na transição do clock.

Usamos essa metodologia nesta seção e no Capítulo 4, e em geral não mostramos o clock explicitamente. Usamos máquinas de estados no Capítulo 4 para controlar a execução do processador e as ações do caminho de dados.

Para ilustrar como uma máquina de estados finitos opera e é projetada, vejamos um exemplo simples e clássico: controlar um sinal de trânsito. (Os Capítulos 4 e 5 contêm exemplos mais detalhados do uso de máquinas de estados finitos para controlar a execução do processador.) Quando uma máquina de estados finitos é usada como controlador, a função de saída normalmente é restrita a depender apenas do estado atual. Essa máquina de estados finitos é chamada de *máquina de Moore*. Esse é o tipo de máquina de estados finitos usado em todo este livro. Se a função de saída puder depender do estado atual e da entrada atual, a máquina é chamada de *máquina de Mealy*. Essas duas máquinas são equivalentes em suas capacidades, e uma pode ser transformada na outra mecanicamente. A vantagem básica de uma máquina de Moore é que ela pode ser mais rápida, enquanto uma máquina de Mealy pode ser menor, pois pode precisar de menos estados do que uma máquina de Moore. No Capítulo 5, discutimos as diferenças com mais detalhes e mostramos uma versão Verilog do controle de estados finitos usando uma máquina de Mealy.

Nosso exemplo trata do controle de um sinal de trânsito em um cruzamento de uma rua norte-sul e uma rua leste-oeste. Para simplificar, vamos considerar apenas os sinais verde e vermelho; a inclusão de um sinal amarelo fica como um exercício. Queremos que os sinais alternem por não menos do que 30 segundos em cada direção, de modo que usaremos um clock de 0,033Hz para que a máquina alterne entre os estados com um tempo não inferior a 30 segundos. Existem dois sinais de saída:

- *NSlite:* quando esse sinal está ativo, a luz na rua norte-sul está verde; quando esse sinal está inativo, a luz na rua norte-sul está vermelha.
- *EWlite:* quando esse sinal está ativo, a luz na rua leste-oeste está verde; quando esse sinal está inativo, a luz na rua leste-oeste está vermelha.

Além disso, existem duas entradas:

- *NScar:* indica que um carro está sobre o detector colocado na rua, na frente do sinal de trânsito da rua norte-sul (indo para o norte ou para o sul).
- *EWcar:* indica que um carro está sobre o detector colocado na rua, na frente do sinal de trânsito da rua leste-oeste (indo para o leste ou para o oeste).

O sinal de trânsito deverá mudar de uma direção para a outra somente se um carro estiver esperando para seguir na outra direção; caso contrário, o sinal deverá continuar a mostrar verde na mesma direção do último carro que cruzou a interseção.

Para implementar esse sinal de trânsito simples, precisamos de dois estados:

- *NSgreen:* o sinal de trânsito é verde na direção norte-sul.
- EW*green:* o sinal de trânsito é verde na direção leste-oeste.

Também precisamos criar a função de próximo estado, que pode ser especificada com uma tabela:

Estado atual	Entradas		Estado seguinte
	NScar	EWcar	
Nsgreen	0	0	NSgreen
Nsgreen	0	1	EWgreen
Nsgreen	1	0	NSgreen
Nsgreen	1	1	EWgreen
Ewgreen	0	0	EWgreen
Ewgreen	0	1	EWgreen
Ewgreen	1	0	NSgreen
Ewgreen	1	1	NSgreen

Observe que não especificamos no algoritmo o que acontece quando um carro se aproxima vindo das duas direções. Nesse caso, a função do estado seguinte que apresentamos muda o estado para garantir que um fluxo constante de carros de uma direção não possa bloquear um carro na outra direção.

A máquina de estados finitos é concluída com a especificação da função de saída:

Estado atual	Saídas	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

Antes de examinarmos como implementar essa máquina de estados finitos, vejamos uma representação gráfica, muito utilizada para máquinas de estados finitos. Nessa representação, os nós são usados para indicar estados. Dentro do nó, colocamos uma lista das saídas ativadas para esse estado. Os arcos direcionados são usados para mostrar a função de próximo estado, com os rótulos nos arcos especificando a condição de entrada como funções lógicas. A Figura C.10.2 mostra a representação gráfica para essa máquina de estados finitos.

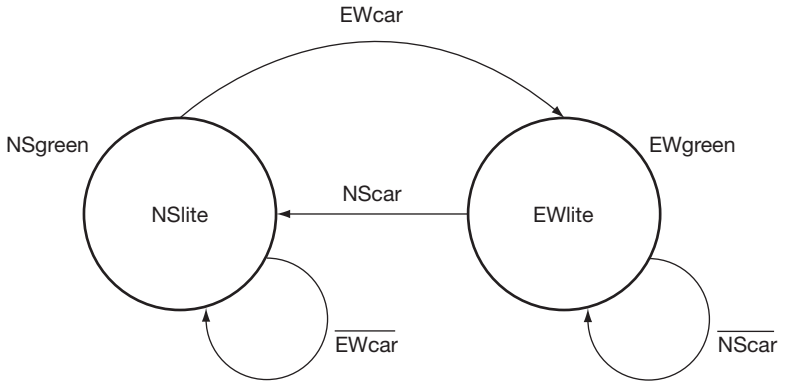


FIGURA C.10.2 A representação gráfica do controle de sinal de trânsito de dois estados. Simplificamos as funções lógicas nas transições de estado. Por exemplo, a transição de NSgreen para EWgreen na tabela de próximo estado é ($\overline{\text{NScar}}$. EWcar) + (NScar . EWcar), que é equivalente a EWcar.

Uma máquina de estados finitos pode ser implementada com um registrador para manter o estado atual e um bloco de lógica combinacional que calcula a função de próximo estado e a função de saída. A Figura C.10.3 mostra uma máquina de estados finitos com 4 bits de estado e, portanto, até 16 estados. Para implementar a máquina de estados finitos dessa maneira, primeiro temos de atribuir números de estado aos estados. Esse processo é chamado de *atribuição de estado*. Por exemplo, poderíamos atribuir NSgreen ao estado 0 e WEgreen ao estado 1. O registrador de estado teria um único bit. A função de próximo estado seria dada como

$$\text{PróximoEstado} = (\overline{\text{EstadoAtual}} \cdot \text{EWcar}) + (\text{EstadoAtual} \,.\, \overline{\text{NScar}})$$

onde EstadoAtual é o conteúdo do registrador de estado (0 ou 1) e EstadoSeguinte é a saída da função de próximo estado, que será escrita no registrador de estado ao final do ciclo de clock. A função de saída também é simples:

$$\text{NSlite} = \overline{\text{EstadoAtual}}$$
$$\text{EWlite} = \text{EstadoAtual}$$

O bloco lógico combinacional normalmente é implementado por meio de lógica estruturada, como PLA. Uma PLA pode ser construída automaticamente a partir das tabelas de função de próximo estado e saída. Na verdade, existem programas de CAD

(Computer-Aided Design) que transformam uma representação gráfica ou textual de uma máquina de estados finitos e produzem uma implementação otimizada. Nos Capítulos 4 e 5, as máquinas de estados finitos foram usadas para controlar a execução do processador. O Apêndice C discute a implementação detalhada desses controladores com PLAs e ROMs.

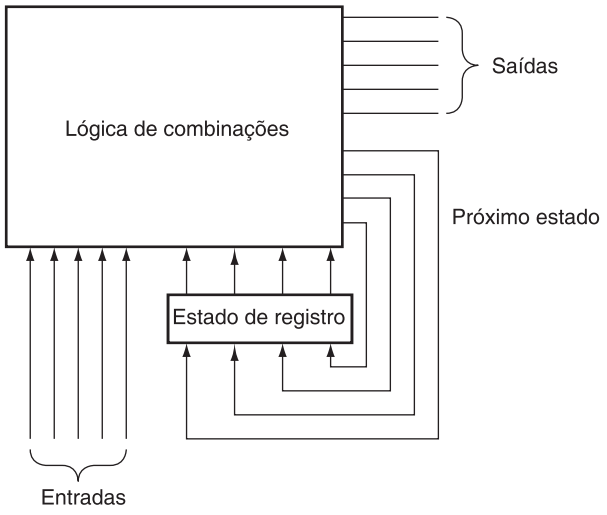


FIGURA C.10.3 Uma máquina de estados finitos é implementada com um registrador de estado que mantém o estado atual e um bloco lógico combinacional para calcular as funções de próximo estado e de saída. As duas últimas funções normalmente são separadas e implementadas com dois blocos de lógica separados, o que pode exigir menos portas.

Para mostrar como poderíamos escrever o controle em Verilog, a Figura C.10.4 mostra uma versão Verilog projetada para a síntese. Observe que, para essa função de controle simples, uma máquina de Mealy não é útil, mas esse estilo de especificação é utilizado no Capítulo 5 para implementar uma função de controle que é uma máquina de Mealy e possui menos estados do que o controlador da máquina de Moore.

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);
   input EWCar, NSCar,clock;
output EWLite,NSLite;

reg state;

initial state=0;  //definir o estado inicial

//após duas atribuições defina a saída, que se baseia
apenas na variável de estado
assign NSLite = ~ state; //em NSLite se estado = 0;
assign EWLite = state; //em EWLite se estado = 1

always @(posedge clock) // todas as atualizações de estado em um positivo
clock edge
   case (state)
      0: state = EWCar; //mude de estado apenas se EWCar

      1: state = NSCar; //mude de estado apenas se NSCar

   endcase
endmodule
```

FIGURA C.10.4 Uma versão Verilog do controlador de sinal de trânsito.

Verifique você mesmo

Qual é o menor número de estados em uma máquina de Moore para os quais uma máquina de Mealy poderia ter menos estados?

a. Dois, pois poderia haver uma máquina de Mealy de um estado que poderia fazer a mesma coisa.

b. Três, pois poderia ser uma máquina de Moore simples, que fosse para um dentre dois estados diferentes e sempre retornasse para o estado original depois disso. Para uma máquina tão simples, uma máquina de Mealy de dois estados é possível.

c. Você precisa de pelo menos quatro estados para explorar as vantagens de uma máquina de Mealy em relação a uma máquina de Moore.

C.11 Metodologias de temporização

No decorrer deste apêndice e no restante do texto, usamos uma metodologia de temporização acionada por transição. Essa metodologia de temporização tem a vantagem de ser mais simples de explicar e entender do que uma metodologia acionada por nível. Nesta seção, explicamos essa metodologia de temporização com um pouco mais de detalhe e também apresentamos o clocking sensível ao nível. Concluímos esta seção discutindo rapidamente a questão dos sinais assíncronos e sincronizadores, um problema importante para os projetistas digitais.

A finalidade desta seção é apresentar os principais conceitos da metodologia de clocking. A seção faz algumas suposições importantes para simplificar; se você estiver interessado em entender a metodologia de temporização com mais detalhes, consulte uma das referências listadas ao final deste apêndice.

Usamos a metodologia de temporização acionada por transição porque ela é mais simples de explicar e tem menos regras exigidas para exatidão. Em particular, se considerarmos que todos os clocks chegam ao mesmo tempo, temos garantias de que um sistema com registradores acionados por transição entre os blocos de lógica combinacional pode operar corretamente, sem condições de corrida, se tornarmos o clock longo o suficiente. Uma condição de *corrida* ocorre quando o conteúdo de um elemento de estado depende da velocidade relativa de elementos lógicos diferentes. Em um projeto acionado por transição, o ciclo de clock precisa ser grande o suficiente para acomodar o caminho de um flip-flop, passando pela lógica combinacional, até chegar a outro flip-flop, onde precisa satisfazer ao requisito do tempo de preparação. A Figura C.11.1 mostra esse requisito para um sistema que usa flip-flops acionados por transição de subida. Em tal sistema, o período de clock (ou tempo de ciclo) precisa ser pelo menos tão grande quanto

$$t_{\text{prop}} + t_{\text{combinacional}} + t_{\text{preparação}}$$

para os valores de pior caso desses três atrasos, que são definidos da seguinte maneira:

- t_{prop} é o tempo para um sinal se propagar por um flip-flop; às vezes, ele também é chamado de clock-para-Q.
- $t_{\text{combinacional}}$ é o maior atraso para qualquer lógica combinacional (que, por definição, é cercada por dois flip-flops).
- $t_{\text{preparação}}$ é o tempo antes da transição do clock de subida que a entrada para um flip-flop precisa ser válida.

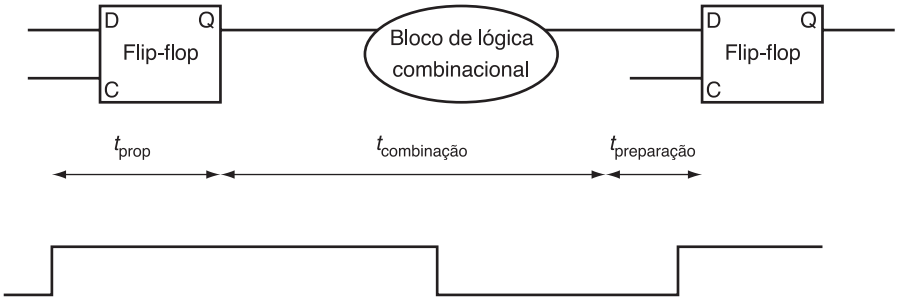


FIGURA C.11.1 No projeto acionado por transição, o clock precisa ser grande o suficiente para permitir que os sinais sejam válidos para o tempo de preparação exigido antes da próxima transição de clock. O tempo para uma entrada de flip-flop se propagar até as saídas do flip-flop é t_{prop}; o sinal leva então $t_{combinacional}$ para atravessar a lógica combinacional e precisa ser válido por $t_{preparação}$ antes da próxima transição de clock.

Fazemos uma suposição para simplificar: os requisitos do tempo de suspensão são satisfeitos, o que quase nunca é um problema com a lógica moderna.

Uma complicação adicional que precisa ser considerada nos projetos acionados por transição é a **inclinação do clock**. Inclinação do clock é a diferença em tempo absoluto entre os instantes em que dois elementos de estado vêem uma transição de clock. A inclinação do clock aumenta porque o sinal de clock normalmente usará dois caminhos diferentes, com atrasos um pouco diferentes, para alcançar dois elementos de estado diferentes. Se a inclinação do clock for grande o suficiente, pode ser possível que um elemento de estado mude e faça com que a entrada para outro flip-flop mude antes que a transição do clock seja vista pelo segundo flip-flop.

inclinação do clock A diferença em tempo absoluto entre os tempos em que dois elementos de estado vêem uma transição de clock.

A Figura C.11.2 ilustra esse problema, ignorando o tempo de preparação e o atraso de propagação do flip-flop. Para evitar operação incorreta, o período de clock é aumentado para permitir a inclinação máxima do clock. Assim, o período do clock precisa ser maior do que

$$t_{prop} + t_{combinacional} + t_{preparação} + t_{inclinação}$$

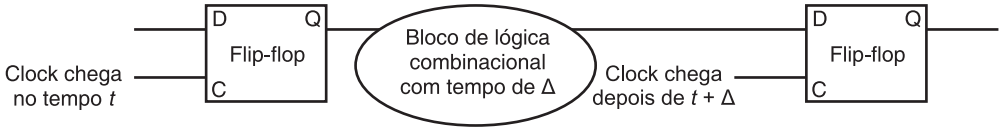


FIGURA C.11.2 Ilustração de como a inclinação do clock pode causar uma condição de corrida, levando à operação incorreta. Devido à diferença entre quando os dois flip-flops vêem o clock, o sinal armazenado no primeiro flip-flop pode se adiantar e mudar a entrada para o segundo flip-flop antes que o clock chegue no segundo flip-flop.

Com essa restrição sobre o período de clock, os dois clocks também podem chegar na ordem contrária, com o segundo clock chegando $t_{inclinação}$ mais cedo, e o circuito funcionará corretamente. Os projetistas reduzem os problemas com inclinação do clock roteando o sinal do clock para minimizar a diferença nos tempos de chegada. Além disso, projetistas inteligentes também oferecem alguma margem, tornando o clock um pouco maior do que o mínimo; isso permite a variação nos componentes e também na fonte de alimentação. Como a inclinação do clock também pode afetar os requisitos do tempo de suspensão, é importante minimizar o tamanho da inclinação do clock.

Os projetos acionados por transição possuem duas desvantagens: eles exigem uma lógica extra e às vezes podem ser mais lentos. Olhar apenas para o flip-flop D *versus* o latch sensível no nível que usamos para construir o flip-flop mostra que o projeto acionado por transição requer mais lógica. Uma alternativa é usar o **clocking sensível ao nível**. Como as mudanças de estado em uma metodologia sensível ao nível não são instantâneas, um esquema sensível ao nível é ligeiramente mais complexo e exige cuidado adicional para fazer com que opere corretamente.

clocking sensível ao nível Uma metodologia de temporização em que as mudanças de estado ocorrem em níveis de clock altos ou baixos, mas não são instantâneas como são em projetos acionados por transição.

Temporização sensível ao nível

Em uma metodologia de temporização sensível ao nível, as mudanças de estado ocorrem nos níveis alto ou baixo, mas não são instantâneas, como acontece em uma metodologia acionada por transição. Devido à mudança não instantânea no estado, condições de corrida podem ocorrer com facilidade. Para garantir que um projeto sensível ao nível também funcione corretamente se o clock for lento o suficiente, os projetistas utilizam o *clocking em duas fases*. O clocking em duas fases é um esquema que utiliza dois sinais de clock não superpostos. Como os dois clocks, normalmente chamados de ϕ_1 e ϕ_2, não são superpostos, no máximo um dos sinais de clock é alto em um momento determinado, como mostra a Figura C.11.3. Podemos usar esses clocks para criar um sistema que contenha latches sensíveis ao nível, mas que seja livre de quaisquer condições de corrida, como eram os projetos acionados por transição.

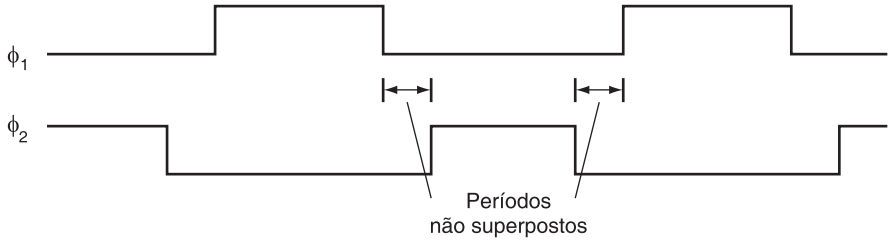


FIGURA C.11.3 Um esquema de clocking de duas fases mostrando o ciclo de cada clock e os períodos não-superpostos.

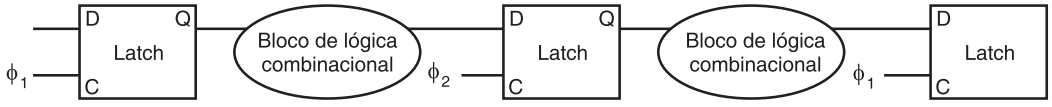


FIGURA C.11.4 Um esquema de temporização de duas fases com latches alternados, mostrando como o sistema opera nas duas fases de clock. A saída de um latch é estável na fase oposta, a partir de sua entrada C. Assim, o primeiro bloco de entradas combinacionais possui uma entrada estável durante ϕ_2, e sua saída é guardada durante ϕ_2. O segundo bloco combinacional (mais à direita) opera exatamente na forma oposta, com entradas estáveis durante ϕ_1. Assim, os atrasos pelos blocos combinacionais determinam o tempo mínimo que os respectivos clocks precisam estar ativos. O tamanho do período não superposto é determinado pela inclinação de clock máxima e pelo atraso mínimo de qualquer bloco lógico.

Um modo simples de projetar tal sistema é alternar o uso de latches abertos em ϕ_1 com os latches abertos em ϕ_2. Como os dois clocks não estão ativos ao mesmo tempo, uma condição de corrida não pode ocorrer. Se a entrada para um bloco combinacional for um clock ϕ_1, então sua saída é acionada por um clock ϕ_2, e só é aberta durante ϕ_2, quando o latch de entrada estiver fechado e, portanto, tiver uma saída válida. A Figura C.11.4 mostra como um sistema com temporização de duas fases e latches alternados é operado. Assim como em um projeto acionado por transição, temos de prestar atenção à inclinação do clock, particularmente entre as duas fases do clock. Aumentando o intervalo de não superposição entre as duas fases, podemos reduzir a margem de erro em potencial. Assim, o sistema tem garantias de operar corretamente se cada fase for grande o suficiente e houver não sobreposição grande o suficiente entre as fases.

Entradas assíncronas e sincronizadores

Usando um único clock ou um clock de duas fases, podemos eliminar condições de corrida se os problemas de inclinação de clock forem evitados. Infelizmente, não é prático fazer um sistema inteiro funcionar com um único clock e ainda reduzir a inclinação do clock. Embora a CPU possa usar um único clock, os dispositivos de E/S provavelmente terão seu próprio clock. O Capítulo 6 descreve como um dispositivo assíncrono pode se comunicar com a CPU por uma série de etapas de handshaking. Para traduzir a entrada assíncrona

para um sinal síncrono que pode ser usado para mudar o estado de um sistema, precisamos usar um *sincronizador*, cujas entradas são o sinal assíncrono e um clock e cuja saída é um sinal síncrono com o clock de entrada.

Nossa primeira tentativa de criar um sincronizador usa um flip-flop D acionado por transição, cuja entrada *D* é o sinal assíncrono, como mostra a Figura C.11.5. Como nos comunicamos com um protocolo de handshaking (conforme visto no Capítulo 6), não importa se detectamos o estado ativo do sinal assíncrono em um clock ou no seguinte, pois o sinal será mantido ativo até que seja confirmado. Assim, você poderia pensar que essa estrutura simples é suficiente para examinar o sinal com precisão, o que aconteceria, exceto por um pequeno problema.

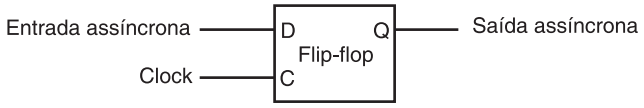


FIGURA C.11.5 Um sincronizador construído a partir de um flip-flop D é usado para examinar um sinal assíncrono para produzir uma saída que é síncrona com o clock. Esse "sincronizador" *não* funcionará corretamente!

metaestabilidade Uma situação que ocorre se um sinal for examinado quando não estiver estável pelos tempos de preparação e suspensão exigidos, possivelmente fazendo com que o valor examinado caia na região indeterminada entre um valor alto e baixo.

falha do sincronizador Uma situação em que um flip-flop entra em um estado metaestável e onde alguns blocos lógicos lendo a saída do flip-flop vêem 0 enquanto outros vêem 1.

O problema é uma situação chamada **metaestabilidade**. Suponha que o sinal assíncrono esteja intercalando entre alto e baixo quando chega a transição do clock. Certamente, não é possível saber se o sinal será mantido como alto ou baixo. Poderíamos viver com esse problema. Infelizmente, a situação é pior: quando o sinal examinado não estiver estável pelos tempos de preparação e suspensão exigidos, o flip-flop poderá entrar em um estado *metaestável*. Nesse estado, a saída não terá um valor alto ou baixo legítimo, mas estará na região indeterminada entre eles. Além do mais, o flip-flop não tem garantias de sair desse estado em qualquer período determinado. Alguns blocos lógicos que examinam a saída do flip-flop podem ver sua saída como 0, enquanto outros podem vê-la como 1. Essa situação é chamada de **falha do sincronizador**.

Em um sistema puramente síncrono, a falha do sincronizador pode ser evitada garantindo-se que os tempos de preparação e suspensão para um flip-flop ou latch sempre sejam atendidos, mas isso é impossível quando a entrada é assíncrona. Em vez disso, a única solução possível é esperar por um tempo suficiente antes de examinar a saída do flip-flop para garantir que sua saída seja estável, e que tenha saído do estado metaestável, se tiver entrado nele. Qual é o tempo suficiente? Bem, a probabilidade de que o flip-flop permaneça no estado metaestável diminui exponencialmente, de modo que, depois de um período muito curto, a probabilidade de que o flip-flop esteja no estado metaestável é muito baixa; porém, a probabilidade nunca chega a 0! Assim, os projetistas esperam por tempo suficiente para que a probabilidade de uma falha do sincronizador seja muito baixa, e o tempo entre essas falhas será de anos ou até mesmo milhares de anos.

Para a maioria dos projetos de flip-flop, esperar por um período muitas vezes maior do que o tempo de preparação torna a probabilidade de falha de sincronismo muito baixa. Se a taxa de clock for maior do que o período de metaestabilidade em potencial (o que é provável), então um sincronizador seguro poderá ser criado com dois flip-flops D, como mostra a Figura C.11.6. Se você estiver interessado em ler mais sobre esses problemas, consulte as referências.

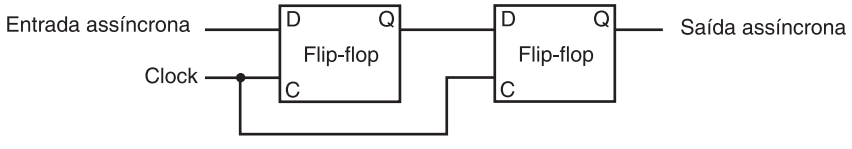


FIGURA C.11.6 Este sincronizador funcionará corretamente se o período de metaestabilidade que desejamos evitar for menor que o período de clock. Embora a saída do primeiro flip-flop possa ser metaestável, ela não será vista por qualquer outro elemento lógico até o segundo clock, quando o segundo flip-flop D examina o sinal, que nesse momento não deverá mais estar em um estado metaestável.

Verifique você mesmo

Suponha que tenhamos um projeto com uma inclinação de clock muito grande – maior do que o **tempo de propagação** do registrador. Sempre será possível que esse projeto atrase o clock por tempo suficiente para garantir que a lógica opere corretamente?

a. Sim, se o clock for lento o suficiente, os sinais sempre podem se propagar e o projeto funcionará, mesmo que a inclinação seja muito grande.

b. Não, pois é possível que dois registradores vejam a mesma transição de clock afastada o suficiente para que um registrador seja acionado, e suas saídas propagadas e vistas por um segundo registrador com a mesma transição de clock.

C.12 Dispositivos programáveis em campo

dispositivo programável em campo (FPD) Um circuito integrado contendo lógica combinacional, e possivelmente dispositivos de memória, configurável pelo usuário final.

Dentro de um chip personalizado ou semipersonalizado, os projetistas podem utilizar a flexibilidade da estrutura básica para facilmente implementar a lógica combinacional ou sequencial. Como um projetista que não quer usar um CI personalizado ou semipersonalizado implementa uma lógica complexa tirando proveito dos níveis de integração muito altos à sua disposição? Os componentes mais populares utilizados para o projeto lógico sequencial e combinacional fora de um CI personalizado ou semipersonalizado é um **dispositivo programável em campo (FPD – Field Programmable Device)**. Um FPD é um circuito integrado contendo lógica combinacional, e possivelmente dispositivos de memória, configurável pelo usuário final.

dispositivo lógico programável Um circuito integrado com lógica combinacional cuja função é configu- rada pelo usuário final.

gate array programável em campo Um circuito integrado configurável contendo blocos lógicos combinacionais e flip-flops.

dispositivo lógico programável simples (SPLD) Dispositivo lógico programável normalmente contendo uma única PAL ou PLA.

array lógico programável (PAL) Contém um plano and programável seguido por um plano or fixo.

antifuse Uma estrutura de um circuito integrado que, quando programada, faz uma conexão permanente entre dois fios.

Look-Up Tables (LUTs) Em um dispositivo programável em campo, o nome dado às células porque consistem em uma pequena quantidade de lógica e RAM.

Os FPDs em geral se encontram em dois campos: **dispositivos lógicos programáveis** (PLDs – Programmable Logic Devices), que são puramente combinacionais, e **gate arrays programáveis em campo** (FPGAs – Field Programmable Gate Arrays), que oferece lógica combinacional e flip-flops. Os PLDs consistem em duas formas: **PLDs simples** (SPLDs), que normalmente são uma PLA ou uma **lógica de array programável** (PAL – Programmable Array Logic), e PLDs complexos, que permitem mais de um bloco lógico e também interconexões configuráveis entre os blocos. Quando falamos de uma PLA em um PLD, queremos dizer uma PLA com plano and e or programável pelo usuário. Uma **PAL** é como uma PLA, exceto que o plano or é fixo.

Antes de discutirmos sobre os FPGAs, é útil falarmos sobre como os FPDs são configurados. A configuração é basicamente uma questão de onde fazer ou romper conexões. Estruturas de porta e de registrador são estáticas, mas as conexões podem ser configuradas. Observe que, configurando as conexões, um usuário determina quais funções lógicas são implementadas. Considere uma PLA configurável: determinando onde estão as conexões no plano and e no plano or, o usuário dita quais funções lógicas são computadas pela PLA. As conexões nos FPDs são permanentes ou reconfiguráveis. As conexões permanentes envolvem a criação ou a destruição de uma conexão entre dois fios. FPLDs atuais utilizam uma tecnologia **antifuse**, que permite que uma conexão seja criada no momento da programação, que será então permanente. A outra maneira de configurar FPLDs CMOS é por meio de uma SRAM. A SRAM é baixada durante a inicialização, e o conteúdo controla a configuração das chaves, que, por sua vez, determinam quais linhas de metal estão conectadas. O uso do controle da SRAM tem a vantagem de que o FPD pode ser reconfigurado alterando-se o conteúdo da SRAM. As desvantagens do controle baseado em SRAM são duas: a configuração é volátil e precisa ser recarregada a cada inicialização, e o uso de transistores ativos para as chaves aumenta um pouco a resistência de tais conexões.

FPGAs incluem dispositivos lógicos e de memória, normalmente estruturados em um array bidimensional, com os corredores dividindo as linhas e colunas usadas para a interconexão global entre as células do array. Cada célula é uma combinação de portas e flip-flops que pode ser programada para realizar alguma função específica. Por serem basicamente RAMs pequenas e programáveis, elas também são chamadas de **Look-Up Tables**. As FPGAs

contêm blocos de montagem mais sofisticados, como partes de somadores e blocos de RAM que podem ser usados para criar bancos de registradores. Algumas FPGAs grandes contêm até mesmo núcleos RISC de 32 bits!

Além de programar cada célula para realizar uma função específica, as interconexões entre as células também são programáveis, permitindo que as FPGAs modernas, com centenas de blocos e centenas de milhares de portas, sejam utilizadas para funções lógicas complexas. A interconexão é um desafio importante nos chips personalizados, e isso é ainda mais verdadeiro para FPGAs, pois as células não representam unidades naturais de decomposição para o projeto estruturado. Em muitas FPGAs, 90% da área é reservada para a interconexão e apenas 10% são blocos lógicos e de memória.

Assim como você não pode projetar um chip personalizado ou semipersonalizado sem ferramentas CAD, também precisa delas para FPDs. Foram desenvolvidas ferramentas de síntese de lógica que visam às FPGAs, permitindo a geração de um sistema usando FPGAs a partir da Verilog estrutural e comportamental.

C.13 Comentários finais

Esse apêndice introduz os fundamentos do projeto lógico. Se você tiver compreendido o material deste apêndice, estará pronto para pegar o material dos Capítulos 4 e 5, ambos usando os conceitos discutidos extensivamente neste apêndice.

Leitura adicional

Existem muitos textos bons sobre projeto lógico. Aqui estão alguns que você poderia examinar.

Ciletti, M. D. [2002] *Advanced Digital Design with the Verilog HDL*, Englewood Cliffs, NJ: Prentice-Hall.

Um livro profundo sobre projeto lógico usando Verilog.

Katz, R. H. [2004] *Modern Logic Design*, segunda edição, Reading, MA: Addison Wesley.

Um texto geral sobre projeto lógico.

Wakerly, J. F. [2000] *Digital Design: Principles and Practices*, terceira edição, Englewood Cliffs, NJ: Prentice-Hall.

Um texto geral sobre projeto lógico.

C.14 Exercícios

C.1 [10] <§C.2> Além das leis básicas que discutimos nesta seção, existem dois teoremas importantes, chamados teoremas de DeMorgan:

$$\overline{A + B} = \overline{A} \cdot \overline{B} \text{ e } \overline{A \cdot B} = \overline{A} + \overline{B}$$

Prove os teoremas de DeMorgan com uma tabela verdade no formato

A	B	$\overline{A}$	$\overline{B}$	$\overline{A + B}$	$\overline{A} \cdot \overline{B}$	$\overline{A \cdot B}$	$\overline{A} + \overline{B}$
0	0	1	1	1	1	1	1
0	1	1	0	0	0	1	1
1	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0

C.2 [15] <§C.2> Prove que as duas equações para E no exemplo a partir da página C-5 são equivalentes usando os teoremas de DeMorgan e os axiomas mostrados na página C-5.

C.3 [10] <§C.2> Mostre que existem 2^n entradas em uma tabela verdade para uma função com *n* entradas.

C.4 [10] <§C.2> Uma função lógica que é usada para diversas finalidades (incluindo dentro de somadores e para calcular paridade) é *OR exclusiva*. A saída de uma função OR exclusiva de duas entradas é verdadeira somente se exatamente uma das entradas for verdadeira. Mostre a tabela verdade para uma função OR exclusiva de duas entradas e implemente essa função usando portas AND, portas OR e inversores.

C.5 [15] <§C.2> Prove que a porta NOR é universal mostrando como montar as funções AND, OR e NOT usando uma porta NOR de duas entradas.

C.6 [15] <§C.2> Prove que a porta NAND é universal, mostrando como montar as funções AND, OR e NOT usando uma porta NAND de duas entradas.

C.7 [10] <§§C.2, C.3> Construa a tabela verdade para uma função de paridade ímpar com quatro entradas (veja na página C-53 uma descrição sobre paridade).

C.8 [10] <§§C.2, C.3> Implemente a função de paridade ímpar com quatro entradas e portas AND e OR usando entradas e saídas em bolha.

C.9 [10] <§§C.2, C.3> Implemente a função de paridade ímpar com quatro entradas com uma PLA.

C.10 [15] <§§C.2, C.3> Prove que um multiplexador de duas entradas também é universal, mostrando como montar uma porta NAND (ou NOR) usando um multiplexador.

C.11 [5] <§§4.2, C.2, C.3> Suponha que X consista em 3 bits, x2 x1 x0. Escreva quatro funções lógicas que sejam verdadeiras se e somente se:

- X tiver apenas 0.
- X contém um número par de 0s.
- X, quando interpretado como um número binário sem sinal, é menor que 4.
- X, quando interpretado como um número com sinal (complemento a dois) é negativo.

C.12 [5] <§§4.2, C.2, C.3> Implemente as quatro funções descritas no Exercício C.11 usando uma PLA

C.13 [5] <§§4.2, C.2, C.3> Suponha que X consista em 3 bits, x2 x1 x0, e Y consista em 3 bits, y2 y1 y0. Escreva funções lógicas que sejam verdadeiras se e somente se:

- X < Y , onde X e Y são considerados números binários sem sinal.
- X < Y, onde X e Y são considerados números binários sem sinal (complemento a dois).
- X = Y

Use uma técnica hierárquica que pode ser estendida para números maiores de bits. Mostre como você pode estendê-la para a comparação em 6 bits.

C.14 [5] <§§C.2, C.3> Implemente uma rede de comutação que possui duas entradas de dados (*A* e *B*), duas saídas de dados (*C* e *D*), e uma entrada de controle (*S*). Se *S* é igual a 1, a rede está no modo pass-through, e *C* deve ser igual a A, e *D* deve ser igual a *B*. Se *S* é igual a 0, a rede está no modo crossing, e *C* deve ser igual a *B*, e *D* deve ser igual a *A*.

C.15 [15] <§§C.2, C.3> Derive a representação do produto-das-somas para *E*, mostrada na página C-8, começando com a representação da soma-dos-produtos. Você precisará usar os teoremas de DeMorgan.

C.16 [30] <§§C.2, C.3> Dê um algoritmo para construir a representação da soma-dos-produtos para uma equação de lógica arbitrária consistindo em AND, OR e NOT. O algoritmo deverá ser recursivo e não deverá construir a tabela verdade no processo.

C.17 [5] <§§C.2, C.3> Mostre uma tabela verdade para um multiplexador (entradas *A*, *B* e *S*; saída *C*), usando don't cares para simplificar a tabela, onde for possível.

C.18 [5] <§C.3> Qual é a função implementada pelos seguintes módulos da Verilog:

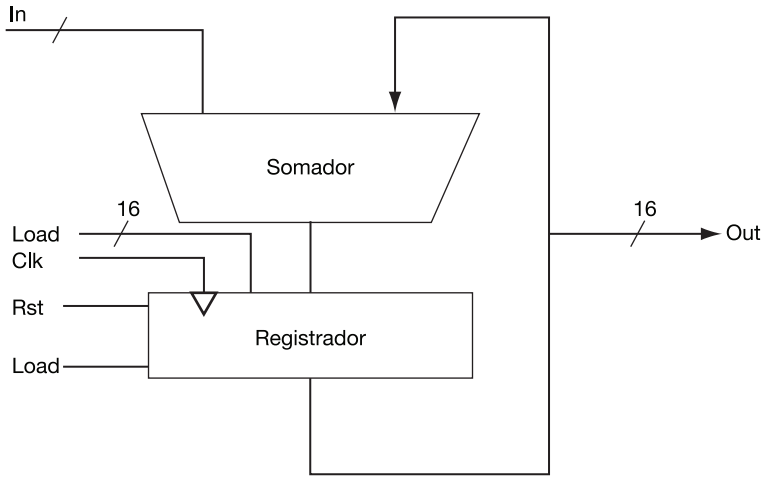
```
module FUNC1 (I0, I1, S, out);
    input I0, I1;
    input S;
    output out;
    out = S? I1: I0;
endmodule

module FUNC2 (out,ctl,clk,reset);
    output [7:0] out;
    input ctl, clk, reset;
    reg [7:0] out;
    always @(posedge clk)
    if (reset) begin
        out <= 8'b0 ;
    end
    else if (ctl) begin
        out <= out + 1;
    end
    else begin
        out <= out - 1;
    end
endmodule
```

C.19 [5] <§C.4> O código em Verilog na página C-42 é para um flip-flop D. Mostre o código em Verilog para um latch D.

C.20 [10] <§§ C.3, C.4> Escreva uma implementação de módulo em Verilog de um decodificador (e/ou codificador) 2-para-4.

C.21 [10] <§§C.3, C.4> Dado o diagrama lógico a seguir para um acumulador, escreva sua implementação do módulo em Verilog. Considere um registrador acionado por transição e Rst assíncrono.



C.22 [20] <§§C.3, C.4, C.5> A Seção 3.3 apresenta a operação básica e possíveis implementações dos multiplicados. Uma unidade básica dessas implementações é uma unidade de shift e soma. Mostre uma implementação Verilog para essa unidade. Mostre como você pode usar essa unidade para montar um multiplicador de 32 bits.

C.23 [20] <§§C.3, C.4, C.5> Repita o Exercício C.22, mas para um divisor sem sinal, ao invés de um multiplicador.

C.24 [15] <§C.5> A ALU admite set on less than (`slt`) usando apenas o bit de sinal do somador. Vamos experimentar uma operação set on less than usando os valores -7_{dec} e 6_{dec}. Para tornar mais simples acompanhar o exemplo, vamos limitar as representações binárias a 4 bits: 1001_{bin} e $0110b_{in}$.

$$1001_{bin} - 0110_{bin} = 1001_{bin} + 1010_{bin} = 0011_{bin}$$

Esse resultado sugeriria que -7 > 6, que certamente é errado. Logo, temos que considerar o overflow na decisão. Modifique a ALU de 1 bit na Figura C.5.10, na página C-27, para lidar com `slt` corretamente. Faça suas mudanças em uma fotocópia dessa figura, para ganhar tempo.

C.25 [20] <§C.6> Uma verificação simples do overflow durante a adição é ver se o CarryIn para o bit mais significativo não é igual ao CarryOut do bit mais significativo. Prove que essa verificação é a mesma da Figura 3.5, na página 190.

C.26 [5] <§C.6> Reescreva as equações da página C-35 para uma lógica carry-lookahead para um somador de 16 bits usando uma nova notação. Primeiro, use os nomes para os sinais CarryIn dos bits individuais do somador. Ou seja, use c4, c8, c12, ... ao invés de C1, C2, C3, ... Além disso, considere que $P_{i,j}$ signifique um sinal de propagação para os bits de *i* a *j*, e $G_{i,j}$ signifique um sinal de geração para os bits *i* a *j*. Por exemplo, a equação

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

pode ser escrita como

$$c8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c0)$$

Essa notação mais geral é útil na criação de somadores mais largos.

C.27 [15] <§C.6> Escreva as equações para a lógica carry-lookahead para um somador de 64 bits usando a nova notação do Exercício C.26 e usando somadores de 16 bits como blocos de montagem. Inclua um desenho semelhante à Figura C.6.3 na sua solução.

C.28 [10] <§C.6> Agora, calcule o desempenho relativo dos somadores. Suponha que o hardware correspondente a qualquer equação contendo apenas termos OR ou AND, como as equações para p*i* e g*i* na página C-32, utilize uma unidade de tempo T. As equações que consistem no OR de vários termos AND, como as equações para c1, c2, c3 e c4 na página C-32, usariam assim duas unidades de tempo, 2T. O motivo é que é necessário T para produzir os termos AND e depois um T adicional para produzir o resultado do OR. Calcule os números e a razão de desempenho para somadores de 4 bits para o carry por ondulação e carry lookahead. Se os termos nas equações forem ainda mais definidos por outras equações, então some os atrasos apropriados para essas equações intermediárias, e continue recursivamente até que os bits de entrada reais do somador sejam usados em uma equação. Inclua um desenho de cada somador rotulado com os atrasos calculados e o caminho do atraso no pior caso destacado.

C.29 [15] <§C.6> Este exercício é semelhante ao Exercício C.28, mas desta vez calcule as velocidades relativas de um somador de 16 bits usando apenas o carry por ondulação, carry por ondulação de grupos de 4 bits que usam carry lookahead, e o esquema de carry-lookahead na página C-31.

C.30 [15] <§C.6> Este exercício é semelhante aos Exercícios C.28 e C.29, mas desta vez calcule as velocidades relativas de um somador de 64 bits usando apenas o carry por ondulação, carry por ondulação de grupos de 4 bits que usam carry lookahead, carry por ondulação de grupos de 16 bits que usam carry lookahead, e o esquema de carry-lookahead na página C-21.

C.31 [10] <§C.6> Ao invés de pensar em um somador como um dispositivo que soma dois números e depois une os carries, podemos pensar no somador como um dispositivo de hardware que pode somar três entradas (a*i*, b*i*, c*i*) e produzir duas saídas (s, *ci* + 1). Ao somar dois números, há pouco que podemos fazer com essa observação. Quando estamos somando mais de dois operações, é possível reduzir o custo do carry. A ideia é formar duas somas independentes, chamadas S2 (bits de soma) e C2 (bits de carry). Ao final do processo, precisamos somar C2 e S2 usando um somador normal. Essa técnica de atrasar a propagação de carry até o final de uma soma de números é chamada *adição carry save*. O desenho em bloco no canto inferior direito da Figura C.14.1 mostra a organização, com dois níveis de somadores carry save, conectados por um único somador normal.

Calcule os atrasos para somar quatro números de 16 bits usando somadores carry-lookahead completos versus carry save com um somador carry-lookahead formando a soma final. (A unidade de tempo T no Exercício C.28 é a mesma.)

C.32 [20] <§C.6> Talvez o caso mais provável de somar muitos números ao mesmo tempo em um computador seria quando se tenta multiplicar mais rapidamente usando muitos somadores para somar muitos números em um único ciclo de clock. Comparado com o algoritmo de multiplicação do Capítulo 3, um esquema carry save com muitos somadores poderia multiplicar mais de 10 vezes mais rápido. Esse exercício estima o custo e a velocidade de um multiplicador combinatório mais de 10 vezes mais rápido. Este exercício estima o custo e a velocidade de um multiplicador combinatório para multiplicar mais de 10 vezes mais rápido. Este exercício estima o custo e a velocidade de um multiplicador combinatório para multiplicar dois números de 16 bits positivos. Suponha que você tenha 16 termos intermediários M15, M14, . . . , M0, chamados *produtos parciais*, que contêm o resultado dos bits do multiplicando AND bits multiplicadores m15, m14, ..., m0. A ideia é usar somadores carry save para reduzir os *n* operandos em 2*n*/3 em grupos paralelos de três, e fazer isso repetidamente até que você obtenha dois números grandes para somar com um somador tradicional.

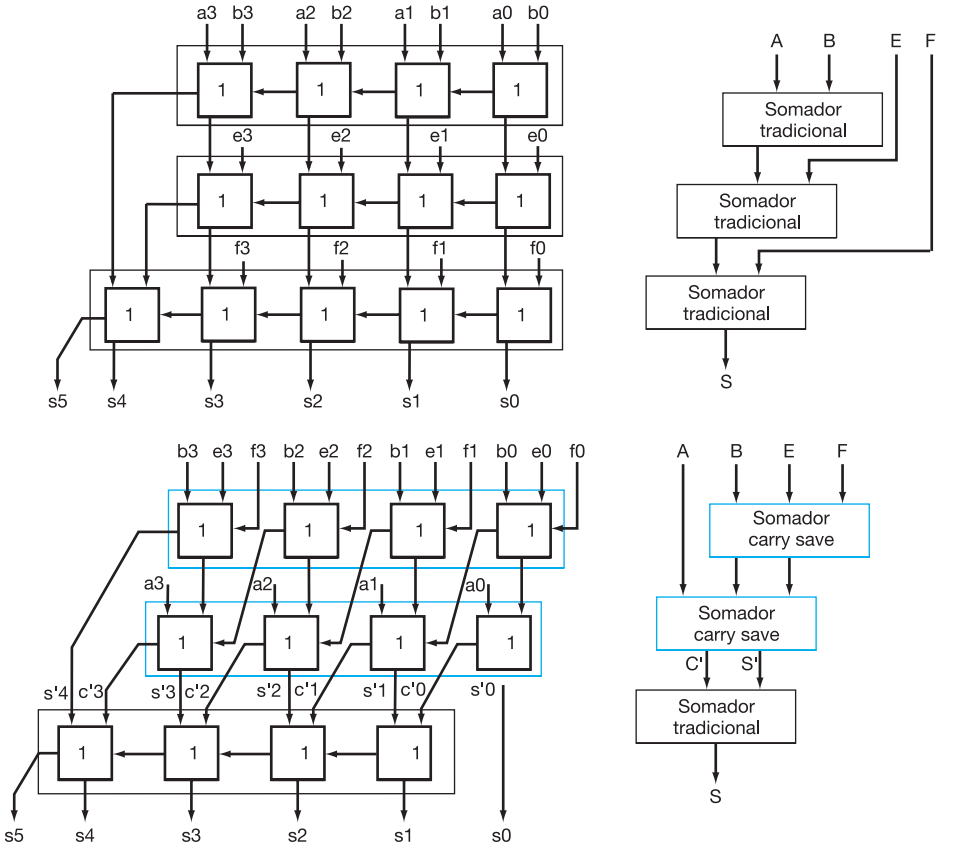


FIGURA C.14.1 Carry por ondulação tradicional e adição carry save de quatro números de 4 bits. Os detalhes aparecem à esquerda, com os sinais individuais em minúsculas, e os blocos correspondentes de nível superior estão à direita, com sinais coletivos em maiúsculas. Observe que a soma de quatro números de n bits pode gerar $n + 2$ bits.

Primeiro, mostre a organização em blocos dos somadores carry save de 16 bits para somar esses 16 termos, mostrados à direita na Figura C.14.1. Depois, calcule os atrasos para somar esses 16 números. Compare esse tempo com o esquema de multiplicação iterativo no Capítulo 3, mas considere apenas 16 iterações usando o somador de 16 bits que tem carry lookahead completo, cuja velocidade foi calculada no Exercício C.29.

C.33 [10] <§C.6> Existem ocasiões em que queremos somar uma coleção de números. Suponha que queiramos somar quatro números de 4 bits (A, B, E, F) usando somadores completos de 1 bit. Vamos ignorar o carry lookahead por enquanto. Você provavelmente conectaria os somadores de 1 bit na organização no topo da Figura C.14.1. Abaixo da organização tradicional está uma nova organização dos somadores completos. Tente somar quatro números usando as duas organizações para se convencer que você obtém a mesma resposta.

C.34 [5] <§C.6> Primeiro, mostre a organização em bloco dos somadores carry save de 16 bits para somar esses 16 termos, como mostra a Figura C.14.1. Suponha que o atraso de tempo por cada somador de 1 bit seja 2T. Calcule o tempo da soma de quatro números de 4 bits para a organização no tipo versus a organização na parte inferior da Figura C.14.1.

C.35 [5] <§C.8> Normalmente, você esperaria que, dado um diagrama de tempo contendo uma descrição das mudanças que ocorrem em uma entrada de dados D e uma entrada de clock C (como nas Figuras C.8.3 e C.8.6 nas páginas C-42 e C-43, respectivamente), haveria diferenças entre as formas de onda de saída (Q) para um latch D e flip-flop D. Em uma

sentença ou duas, descreva as circunstâncias (por exemplo, a natureza das entradas) para as quais não haveria qualquer diferença entre as duas formas de onda de saída.

C.36 [5] <§C.8> A Figura C.8.8 na página C-44 ilustra a implementação do banco de registradores para o caminho de dados do MIPS. Imagine que um novo banco de registradores deva ser montado, mas que haja somente dois registradores e apenas uma porta de leitura, e que cada registrador tenha apenas 2 bits de dados. Redesenhe a Figura C.8.8 de modo que cada fio no seu diagrama corresponda a somente 1 bit de dados (diferente do diagrama na Figura C.8.8, em que alguns fios são de 5 bits e alguns fios são de 32 bits). Redesenhe os registradores usando flip-flops D. Você não precisa mostrar como implementar um flip-flop D ou um multiplexador.

C.37 [10] <§C.10> Um amigo gostaria que você montasse um "olho eletrônico" para usar como um dispositivo de segurança falso. O dispositivo consiste em três luzes alinhadas em sequência, controladas pelas saídas Esquerda, Meio e Direita, que, se ativadas, indicam que uma luz deve ser acesa. Somente uma luz está acesa de cada vez, e a luz se "move" da esquerda para a direita e depois da direita para a esquerda, o que afugenta os ladrões, que acreditam que o dispositivo está monitorando sua atividade. Desenhe uma representação gráfica para a máquina de estados finitos usada para especificar o olho eletrônico. Observe que a velocidade do movimento do olho será controlada pela velocidade do clock (que não deverá ser muito grande) e que basicamente não existem entradas.

C.38 [10] <§C.10> {Ex. C.37} Atribua números de estado aos estados da máquina de estados finitos que você construiu para o Exercício C.37 e escreva um conjunto de equações lógicas para cada uma das saídas, incluindo os bits do estado seguinte.

C.39 [15] <§§C.2, C.8, C.10> Construa um contador de 3 bits usando três flip-flops D e uma seleção de portas. As entradas deverão consistir em um sinal que retorna o contador a 0, chamado *reset*, e um sinal para incrementar o contador, chamado *inc*. As saídas deverão ser o valor do contador. Quando o contador tiver valor 7 e for incrementado, ele deverá retornar e se tornar 0.

C.40 [20] <§C.10> Um *código cinza* é uma sequência de números binários com a propriedade de que não mais de 1 bit muda ao passar de um elemento da sequência para o outro. Por exemplo, aqui está um código cinza binário de 3 bits: 000, 001, 011, 010, 110, 111, 101 e 100. Usando três flip-flops D e uma PLA, construa um contador de código cinza de 3 bits que tem duas entradas: *reset*, que define o contador como 000, e *inc*, que faz o contador seguir para o próximo valor na sequência. Observe que o código é cíclico, de modo que o valor após 100 na sequência é 000.

C.41 [25] <§C.10> Queremos acrescentar uma luz amarela ao nosso exemplo de sinal de trânsito da página C-54. Faremos isso alterando o clock para 0,25Hz (um tempo de ciclo de clock de 4 segundos), que é a duração de uma luz amarela. Para impedir que as luzes verde e vermelha se repitam muito rapidamente, acrescentamos um temporizador de 30 segundos. O temporizador tem uma única entrada, chamada *TimerReset*, que reinicia o timer, e uma única saída, chamada *TimerSignal*, que indica que o período de 30 segundos expirou. Além disso, temos que redefinir os sinais de trânsito para incluir o amarelo. Fazemos isso definindo dois sinais de saída para cada luz: verde e amarelo. Se a saída verde NS estiver ativada, a luz verde está acesa; se a saída NSyellow estiver ativada, a luz amarela está acesa. Se os dois sinais estiverem desativados, a luz vermelha está acesa. *Não* ative os sinais verde e amarelo ao mesmo tempo, pois os motoristas certamente ficarão confusos, mesmo que os motoristas europeus entendam o que isso significa! Desenhe a representação gráfica para a máquina de estados finitos para esse controlador melhorado. Escolha nomes para os estados que sejam *diferentes* dos nomes das saídas.

C.42 [15] <§C.10> {Ex. C.41} Escreva as tabas de próximo estado e função de saída para o controlador de sinal de trânsito descrito no Exercício C.41.

C.43 [15] <§§C.2, C.10> {Ex. C.42} Atribua os números de estado aos estados no exemplo de sinal de trânsito do Exercício C.41 e use as tabelas do Exercício C.42 para escrever um conjunto de equações lógicas para cada uma das saídas, incluindo as saídas do estado seguinte.

C.44 [15] <§§C.3, C.10> {Ex. C.43} Implemente as equações lógicas do Exercício C.43 como uma PLA.

Respostas das Seções "Verifique você mesmo"

§C.2, página C-6: Não. Se $A = 1$, $C = 1$, $B = 0$, o primeiro é verdadeiro, mas o segundo é falso.

§C.3, página C-15: C.

§C.4, página C-17: Todos são exatamente iguais.

§C.4, página C-20: A = 0, B = 1.

§C.5, página C-30: 2.

§C.6, página C-37: 1.

§C.8, página C-46: c.

§C.10, página C-58: b.

§C.11, página C-62: b.