

Capítulo 1

Organização de um Computador

It is possible to invent a single machine which can be used to compute any computable sequence.

Alan Turing

Iniciemos com um exame, necessariamente superficial, da organização de um computador disponível em lojas de varejo em 2020¹. A Figura 1.1 mostra um diagrama com os componentes principais no nosso computador. Vejamos a função de cada componente, e então um exemplo da operação conjunta de todos eles.

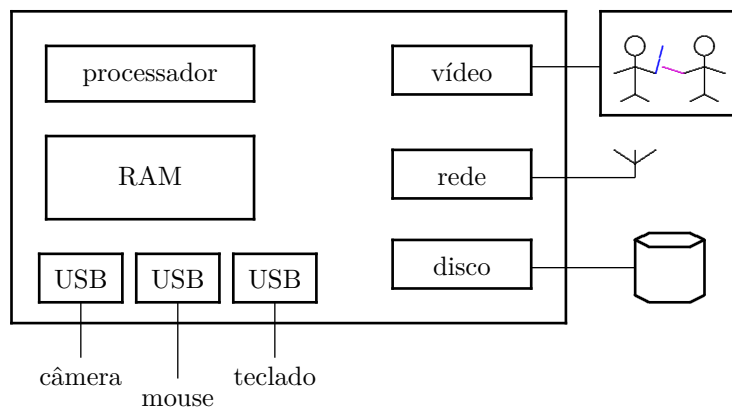


Figura 1.1: Organização de um computador pessoal.

Um pouco de notação Empregamos a notação de intervalos para denotar subconjuntos dos Inteiros, dos Naturais ou dos Reais. Um colchete fechado – '[' ou ']' – indica que o elemento adjacente ao colchete pertence ao conjunto, ou que ele faz parte do intervalo; por exemplo, o intervalo $[3, 7] \in \mathbb{N}$ inclui os Inteiros 3, 4, 5, 6, 7. Um parêntese – '(' ou ')' – indica que o elemento adjacente *não* pertence ao intervalo; por exemplo, o intervalo $(0, 1] \in \mathbb{R}$ exclui o zero mas inclui todos os reais maiores que zero e menores ou iguais a 1.

Um *byte* é um octeto de dígitos binários e armazena um número natural no intervalo $[0, 255]$, ou um inteiro no intervalo $[-128, 127]$. Os *números mágicos* são $255 = (2^8 - 1)$, $-128 = -(2^7)$ e $+127 = (2^7 - 1)$. Essa tal *magia* é desmistificada no Capítulo 2.

¹© Roberto André Hexsel, 2020. Versão de 9 de novembro de 2020.

Processador O *processador* é o componente principal de um computador pois é este quem interpreta e executa as *instruções* do programa que está a executar. As instruções representam comandos simples que o processador deve executar, tal como “adicione os conteúdos de x e y e armazene a soma em z ”, ou “armazene o valor de k na posição p da memória”. Para consumo humano estas duas instruções são representadas como

$z \leftarrow x + y$
e
 $M(p) \leftarrow k.$

No programa que o processador executa estas instruções são representadas por sequências de 32 dígitos binários (bits) tais como

11001010111100101110011010010001.

Evidentemente, a primeira representação é mais cômoda do que a segunda. A flecha que aponta para a esquerda representa a *atribuição* de um valor ($x+y$) a uma unidade de armazenamento (z), e o novo valor sobrescreve o conteúdo daquela unidade de armazenamento, que fora computado anteriormente.

Na disciplina Circuitos Digitais, estudamos a programação e a organização de um processador simples, que é descrito no Capítulo 9. Na disciplina de Projetos Digitais (Capítulos ?? e ??) veremos o projeto e a implementação de um processador realista. O projeto de computadores é objeto da disciplina Arquitetura de Computadores.

RAM Em Português, a memória RAM se chama *memória de escrita e de leitura*, embora o autor prefira a abreviatura do nome em Inglês: *Random Access Memory*. A razão para o *random* é histórica – nas décadas de 1940-50 a memória disponível era *sequencial* e para acessar a posição 19 da memória era necessário iniciar a busca pela posição 0, depois 1, 2, 3, ... 19. A possibilidade de acessos a quaisquer endereços, aleatoriamente, foi um grande progresso.

Do ponto de vista lógico, a RAM é uma função bijetora, os elementos do domínio são chamados de *endereços* e os elementos da imagem são o *conteúdo* armazenado nos respectivos endereços. Se a unidade de acesso é um byte (oito bits), então a memória é definida pela função

$$RAM : [0, C] \mapsto [0, 255]$$

sendo C a capacidade da memória, que é o número de bytes endereçáveis, e a cada endereço corresponde um valor representável em 8 bits, que é um número natural na faixa de 0 a 255.

Quando a memória é RAM, seus conteúdos podem ser alterados; quando a memória é ROM, ou *Read Only Memory*, seus conteúdos não podem ser alterados após o processo de fabricação ter sido concluído. Existem memórias que se comportam como ROM na maior parte do tempo, embora possibilitem que seu conteúdo seja alterado com o circuito em operação.

A organização interna de memórias ROM e RAM é introduzida na Seção 4.3.4. Voltaremos a esse assunto na disciplina de Projetos Digitais (Seções 5.5 e 5.6).

Interface de vídeo Talvez um nome mais apropriado do que *interface de vídeo* seja “tela de computador”. Considere uma tela quadrada com 1024×1024 pontos, ou *pixels*, com imagens em tons de cinza, de tal forma que cada pixel seja representado por um byte. Nesse caso, 0 representa um ponto branco, 255 representa um ponto preto, e 127 representa um ponto de tonalidade cinza-médio.

É necessária uma RAM com 1024×1024 bytes, ou 1 Mbytes, para armazenar uma cópia do que é exibido na tela. O controlador de vídeo varre esta memória e ilumina cada ponto da tela de acordo com o valor do byte que lhe corresponde. Para alterar a imagem, o processador altera o conteúdo da *memória de vídeo*.

Para uma tela colorida, são necessários 4 bytes por pixel, um para cada uma das cores fundamentais (vermelho, verde e azul) e mais um para a intensidade da iluminação do ponto. A representação mais rica envolve um aumento de um fator de quatro na capacidade de memória para manter a cópia da imagem em memória. Além de mais memória, o processamento de imagens coloridas é computacionalmente mais custoso do que para tons de cinza.

Interface de Rede Considere uma *interface de rede* que opere com mensagens com capacidade de 1 a 1024 bytes. No lado da recepção, as mensagens chegam a qualquer instante e por isso a interface deve conter memória para acomodar algumas mensagens, caso o processador não esteja disponível para tratá-las no instante em que são recebidas. No lado da transmissão, bastaria espaço para acomodar uma mensagem de tamanho máximo; por razões de eficiência, também aloca-se espaço para algumas mensagens no circuito de transmissão.

Como deve ser gerenciado o fluxo de mensagens para possibilitar a transferência de um arquivo grande? Como a ordem das mensagens é preservada? É desejável que ocorram várias transferências simultaneamente? Como os fluxos distintos são mantidos separados? Como os computadores ligados à Internet são endereçados? O que fazer quando da ocorrência de erros? Quais seriam os erros? Respostas precisas para estas perguntas serão obtidas ao cursar a(s) disciplina(s) de Redes de Computadores.

Disco Magnético como Memória Secundária A RAM é chamada de *memória primária* porque é essa memória que o processador acessa diretamente para executar programas e ler ou atualizar dados. Contudo, a capacidade da RAM é menor do que o conjunto de dados com que trabalhamos, e assim a *memória secundária* é empregada para armazenar arquivos tais como livros, filmes, etc. Em 2020, a memória secundária é tipicamente implementada com discos magnéticos. Como o nome indica, o ‘disco’ é coberto com óxido ferroso e porções diminutas da superfície do disco são magnetizadas para codificar 0s e 1s.

Cada superfície de um disco magnético é organizada em trilhas concêntricas, cada trilha dividida em setores. Um setor armazena de 1024 a 4096 bytes. Uma unidade de disco magnético pode ter mais de uma superfície. A posição de um determinado byte no disco é dada pela tripla $\langle \text{setor, trilha, superfície} \rangle$.

Por uma série de razões que serão discutidas mais adiante no curso, é conveniente trabalhar com um *disco lógico*, que é endereçado somente pelo número do bloco desejado. Em geral, um *bloco lógico* tem o mesmo tamanho de um setor, ou um *bloco físico*.

O controlador do disco faz a conversão de endereços físicos de 3 dimensões para endereços lógicos de uma dimensão

$$\langle \text{setor, trilha, superfície} \rangle \mapsto \langle \text{número do bloco} \rangle.$$

O endereço de um byte no disco lógico pode ser determinado pelas funções

$$\text{disco} : [0, N) \mapsto \text{bloco}, \quad \text{bloco} : [0, B) \mapsto [0, 255]$$

sendo N o número de blocos, e B o tamanho de um bloco.

Dispositivos de armazenamento com memória de estado sólido (*Solid State Disks* ou SSDs) estão se tornando populares pelo seu tempo de acesso, que é cerca de 100 vezes mais curto do que o dos discos magnéticos. É provável que em breve, os discos magnéticos sejam suplantados por SSDs. SSDs são acessados como discos magnéticos, e indexados pelo número do bloco. Em tese, estes dispositivos poderiam ser acessados como RAM, byte a byte ao invés de “bloco depois byte”.

As memórias primária e secundária são estudadas nas disciplinas Software Básico e Sistemas Operacionais. A tradução de programas escritos em Pascal ou C para linguagem de máquina é estudada aqui, em Projetos Digitais, Software Básico e em Construção de Compiladores.

Exemplo 1.1 Vejamos como é o fluxo dos bytes quando obtemos uma cópia de um arquivo a partir de um servidor de conteúdos da Internet. A Figura 1.2 mostra uma versão ligeiramente expandida do computador da Figura 1.1.

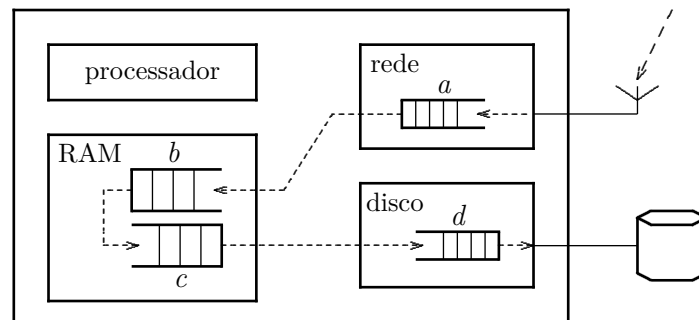


Figura 1.2: Fluxo de bytes entre interface de rede e memória secundária.

As mensagens são recebidas do servidor remoto em tempos imprevisíveis e indeterminados, e por isso emprega-se uma fila (*a*) para acomodar as diferenças na taxa em que as mensagens são recebidas, e a taxa com que elas são tratadas pelo processador. Esta fila tem a mesma função que a fila de um banco, aonde as pessoas (mensagens) esperam pelo atendimento por um dos caixas (processador). No caso da interface de rede, a fila é necessária para desacoplar as velocidades de chegada na antena e de atendimento pelo processador. A fila *a* é gerenciada pelo controlador da interface de rede, que é, ele próprio, um ‘computador’ dedicado a tratar da recepção e envio de mensagens através da interface de rede.

A fila *b* é usada para garantir que todas as mensagens que deveriam ter sido recebidas foram recebidas, e mais ainda, que a ordem dos bytes no arquivo original foi preservada durante a transmissão. Essa fila é gerenciada pelo sistema operacional (SO) do computador, e é acessada pelo processador para inserir novos elementos na cauda, e retirar os elementos mais antigos da cabeça da fila. Esta fila é parte do componente do SO que gerencia a transmissão e recepção de mensagens.

Uma vez que o SO verificou que o segmento do arquivo que está na fila *b* foi recebido corretamente, este segmento é removido da fila *b* e inserido na fila *c*. Esta fila é parte do componente do SO que gerencia o sistema de arquivos, e tem uma finalidade semelhante à da fila *b*, exceto que, ao invés de mensagens, os elementos desta fila são blocos lógicos que serão gravados no disco. Em geral, o tamanho dos blocos lógicos é distinto da capacidade máxima das mensagens, e é por isso que mensagens e blocos devem ser mantidos em filas separadas.

A fila *d* é gerenciada pelo controlador do disco, e seus elementos são blocos lógicos, esperando para ser gravados nos respectivos blocos físicos. O tempo de acesso de um disco magnético é longo e uma

escrita em disco demora o tempo equivalente à execução de 10^5 a 10^6 instruções pelo processador. Para todos os fins práticos, o tempo de acesso aos blocos físicos é imprevisível, e a fila d acomoda as diferenças de velocidade com que blocos lógicos são nela inseridos pelo processador, e blocos físicos são removidos e gravados na superfície magnetizada pelo controlador do disco. Assim como o controlador da interface de rede, o controlador de disco é um ‘computador’ dedicado a controlar as transferências entre memória RAM e a superfície magnetizada. ◁

Exercícios

Ex. 1.1 Quais as diferenças, do ponto de vista operacional, da aplicação da vacina tríplice em bebês e da vacina – quando esta ficar disponível – para a COVID-19?

Ex. 1.2 Considere um posto de saúde que atende a 100.000 pessoas, e que a vacina para COVID-19 está, finalmente, disponível. No posto existem quatro filas de aplicação de vacina, e cada aplicação demora 5 minutos. Supondo um horário de atendimento de oito horas, quantos dias são necessários para vacinar todas as 10^5 pessoas?

Ex. 1.3 Considerando os dados do Ex. 1.2, e que um lote de 5.000 doses é entregue ao posto a cada 3 dias, como isso altera o tempo de vacinação das 10^5 pessoas? Compensaria alterar o número de filas de aplicação? Se sim, como?

Ex. 1.4 Considerando os dados do Ex. 1.3, e que a fábrica de vacinas produz 1.000.000 de doses por semana, e supondo que todos os postos de saúde sejam como aquele do Ex. 1.2, quanto tempo levaria para vacinar todos os 220 milhões de brasileiros?

Ex. 1.5 Considerando os dados dos Ex. 1.2 a 1.4, compensaria alterar a logística de vacinação? Se sim, como?

Ex. 1.6 Podem ser necessárias duas doses para a imunização efetiva e eficaz. Como isso alteraria sua resposta ao Ex. 1.5?

Por mais estranho que possa parecer, estas questões são muito similares àquelas que devem ser respondidas por um projetista de computadores. As questões são abertas, e a resposta a uma questão posterior frequentemente impacta e altera as respostas para questões anteriores.

Capítulo 2

Representação de Informação

Lord Polonius: What do you read, my lord?

Hamlet: Words, words, words.

Lord Polonius: What is the matter, my lord?

Hamlet: Between who?

Lord Polonius: I mean, the matter that you read, my lord.

William Shakespeare, Hamlet, II,2.

Qual é a informação contida na frase “a Terra é redonda”? Para alguém que cursa Ciência da Computação, o conteúdo desta frase é trivial porque *qualquer* pessoa que tenha cursado o primeiro ano do Ensino Fundamental sabe que a Terra é (aproximadamente) esférica. Por “conteúdo trivial” entenda-se que não há novidade na frase, logo aprendemos muito pouco ao ouvi-la, pois a frase não traz nenhuma informação nova para alguém que tenha concluído o primeiro ano do Fundamental.

Do ponto de vista da Computação, *informação* é algo que nos ajuda a compreender melhor o mundo em que vivemos, num determinado instante. Por exemplo, a frase “a Lua é um icosaedro”, caso o satélite tivesse sofrido uma transformação extraordinária nos últimos dez minutos, conteria uma quantidade enorme de informação. Se, contudo, o satélite permanece como sempre o vemos no céu, a frase tem conteúdo informacional nulo.

A cena da peça citada acima é uma das minhas favoritas porque Hamlet exprime um fato – ele está a ler palavras no livro – mas Polônio espera por outro fato – Hamlet está a ler ‘algo’ no livro. Os dois estão corretos, exceto pelos distintos significados atribuídos à mesma informação, Hamlet num contexto micro – palavras soltas – e Polônio num contexto macro – texto. Para além disso, Hamlet está a fazer troça de Polônio ao interpretar a frase “*what is the matter?*” com um sentido diferente daquele pretendido por Polônio.

Se a frase “a Terra é redonda” for representada em caracteres Kanji, em cirílico, ou em sânscrito, teremos mais ou menos dificuldade em apreender seu significado. Para a imensa maioria dos habitantes de Angola, Brasil, Moçambique e Portugal, o significado da frase é um só. Para a imensa maioria dos habitantes da China, a frase não faz sentido algum, porque a interpretação dos símbolos com que a frase é desenhada não faz parte do dia a dia da maioria dos chineses. Além do consenso sobre o significado das palavras, é necessário um consenso sobre a forma de *representar* as palavras.

Para que a informação seja transmitida entre fonte e destino, é necessário um *alfabeto* – um *conjunto de símbolos* – mais uma gramática – um *conjunto de regras* – que define como os símbolos podem ser combinados. Além disso, um *conjunto de significados* é necessário para que informação seja transferida de uma entidade para outra. Por exemplo, a frase “a Lua é feita de queijo” somente é aceitável como exprimindo um fato verdadeiro no contexto das extraordinárias aventuras de Wallace & Gromit.

Vejamos um alfabeto e uma gramática que nos permitam exprimir informação sobre quantidades, ou que nos permitam *representar* quantidades.

2.1 Uma Representação para Números Inteiros

O alfabeto

$$A_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \_ \} \quad (2.1)$$

nos permite exprimir, de forma simbólica, quantidades a que chamamos de *números inteiros*. A gramática é relativamente simples: uma quantidade maior que o número de dedos das mãos de um humano pode ser representada pela justaposição dos símbolos do alfabeto. Duas quantidades distintas são separadas por um espaço em branco, que também é um símbolo do nosso alfabeto. O espaço é indicado como ‘ $\_$ ’.

A regra de formação dos números emprega a *representação posicional*: cada posição da quantidade representada corresponde a uma potência de dez. Um elemento ‘solteiro’ do alfabeto é multiplicado por 10^0 . Com dois elementos justapostos, o da esquerda é multiplicado por 10^1 e o da direita por 10^0 :

$$27 = 2 \times 10^1 + 7 \times 10^0 = 2 \times 10 + 7 \times 1.$$

Nesta *representação posicional* cada posição corresponde a uma potência de 10:

<i>posição</i> $\rightarrow$	10^6	10^5	10^4	10^3	10^2	10^1	10^0
4.259.301	4	2	5	9	3	0	1

Para facilitar a leitura, acrescentamos o símbolo ‘.’ ao nosso alfabeto, para separar os milhares.

Para uma representação com k dígitos, o valor do número natural N representado pela justaposição de k dígitos é obtido pela Equação 2.2, e é a soma dos dígitos d_i , $i \in [0, k)$ multiplicados pela base b elevada à i -ésima potência. Na Equação 2.2 temos $b = 10$ e $d \in A_{10}$.

$$N = \sum_{i=0}^{k-1} d_i \times b^i = d_0 \times b^0 + d_1 \times b^1 + d_2 \times b^2 + \dots + d_{k-1} \times b^{k-1} \quad (2.2)$$

Se quisermos representar números reais, nosso alfabeto deve ser ampliado com uma ‘,’ para separar as potências positivas das potências negativas, e um ‘-’ para indicar números negativos. Numa representação com m dígitos para as potências positivas e n dígitos para as negativas, o número R é obtido da Equação 2.3, com $i \in [0, m)$ e $j \in [-n, 0)$. Números negativos são prefixados por ‘-’.

$$R = \left(\sum_{i=0}^{m-1} d_i \times 10^i \right), \left(\sum_{j=-n}^{-1} d_j \times 10^j \right) \quad (2.3)$$

Com nosso alfabeto estendido, a palavra “2.408,753” significa

$$2 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 8 \times 10^0 + 7 \times 10^{-1} + 5 \times 10^{-2} + 3 \times 10^{-3}.$$

2.2 Quatro Dedos em Cada Mão

Se tivéssemos somente quatro dedos em cada mão, nosso alfabeto estendido para representar números naturais seria

$$A_8 = \{0, 1, 2, 3, 4, 5, 6, 7, \_, .\} \quad (2.4)$$

e a sequência de contagem com oito dedos seria

$$0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, \dots$$

O sistema de numeração com base oito, chamado de *octal*, era usado em computação até meados da década de 1980, e felizmente caiu em desuso. Mãos com 8 dedos tornaram-se mais comuns na década de 1990 porque $8 + 8 = 16$ dedos propiciam representações mais compactas do que com $4 + 4 = 8$ dedos.

Para uma representação com k dígitos, o valor do número natural O representado pela justaposição de k dígitos é obtido pela Equação 2.5, e é a soma dos dígitos d_i , $i \in [0, k)$ multiplicados pelas potências de oito.

$$O = \sum_{i=0}^{k-1} d_i \times 8^i, \quad d \in A_8 \quad (2.5)$$

Quando usamos representações com mais de uma base, é usual apensar um subscrito ao número, indicando a sua base. A quantidade 254_{10} é representada na base 10, enquanto que 744_8 é representada na base 8. Geralmente, números na base 10 são representados sem o subscrito.

A conversão de uma quantidade representada em octal para a sua equivalente na base 10 é obtida pela aplicação direta da Equação 2.5: cada dígito é multiplicado pela potência de 8 que lhe corresponde e o valor representado é a soma das parcelas.

A conversão de uma quantidade representada na base 10 para sua equivalente em octal é um pouco mais complexa porque as bases são distintas, e dígitos maiores do que 7 são representados por dois dígitos na base 8, mas a correspondência não é direta porque $9_{10} = 7_8 + 2_8$.

O *módulo* de um número, com relação a uma base, é o resto da divisão inteira do número pela base. O operador módulo é representado pelo símbolo $\%$. Com decimais, $5 \% 10 = 5$, $10 \% 10 = 0$, $12 \% 10 = 2$, e $12 \% 8 = 4$.

A conversão de uma quantidade D representada na base 10 para a sua equivalente na base 8 é obtida pela aplicação do *algoritmo de conversão de bases*:

Algoritmo de conversão de bases

- (1) divida D por 8;
- (2) guarde o quociente da divisão inteira para a próxima iteração;
- (3) mantenha o módulo da divisão, que é o “dígito da vez”;
- (4) repita os passos (1) a (3) até que o quociente seja zero.

Esta sequência de passos produz os dígitos do valor convertido do menos significativo para o mais significativo.

Exemplo 2.1 Façamos a conversão de 1987_{10} para sua representação em octal. A conversão é mostrada na Tabela 2.1, e o valor convertido é 3703_8 . O dígito menos significativo é obtido do módulo

de 1987 por 8, o segundo dígito é o módulo de 248 por 8, o terceiro é o módulo de 31 por 8, e o quarto dígito – o quociente da divisão da parte inteira é zero – é o módulo de 3 por 8. Para calcular o módulo, multiplicamos a parte fracionária do quociente por 8.

Na medida em que ‘descemos’ na tabela, o número original é dividido por uma potência cada vez maior da base. Na quarta linha, D já foi dividido por 8 três vezes: $8 \times 8 \times 8 = 8^3$. $\triangleleft$

Tabela 2.1: Conversão de 1987 para octal.

quociente	módulo	dígito	potência
$1987 \div 8 = 248,375$	$1987 \% 8 = 0,375 \times 8 = 3$	$d_0 = 3$	8^0
$248 \div 8 = 31,000$	$248 \% 8 = 0,000 \times 8 = 0$	$d_1 = 0$	8^1
$31 \div 8 = 3,875$	$31 \% 8 = 0,875 \times 8 = 7$	$d_2 = 7$	8^2
$3 \div 8 = 0,375$	$3 \% 8 = 0,375 \times 8 = 3$	$d_3 = 3$	8^3

2.3 Oito Dedos em Cada Mão

Se tivéssemos oito dedos em cada mão, nosso alfabeto estendido seria

$$A_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, \_, .\} \quad (2.6)$$

e a sequência de contagem seria

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, \dots$$

O sistema de numeração com base dezesseis, chamado de *hexadecimal*, substituiu o octal porque as representações em hexadecimal são mais compactas. O alfabeto para expressar quantidades na base 10 foi estendido com mais seis símbolos, que por conveniência são as seis primeiras letras do alfabeto usado para representar texto.

Para uma representação com k dígitos, o valor do número natural H representado pela justaposição de k dígitos é obtido pela Equação 2.7, e é a soma dos dígitos d_i , $i \in [0, k)$ multiplicados pelas correspondentes potências de dezesseis.

$$H = \sum_{i=0}^{k-1} d_i \times 16^i, \quad d \in A_{16} \quad (2.7)$$

A conversão de um número na base 10 para um na base 16 emprega o mesmo algoritmo que na conversão de decimal para octal, trocando-se a base.

Exemplo 2.2 Façamos a conversão de 1987_{10} para sua representação em hexadecimal. A conversão é mostrada na Tabela 2.2. Na medida em que descemos na tabela, o número original é dividido por uma potência cada vez maior da base 16.

A representação de 1987 necessita de quatro dígitos tanto em decimal (1987_{10}) quanto em octal (3703_8), mas somente três dígitos em hexadecimal ($7C3_{16}$). $\triangleleft$

Exemplo 2.3 Façamos a conversão de $DC49_{16}$ (56.393_{10}) para sua representação em octal. A conversão é mostrada na Tabela 2.3. A representação em octal para $DC49_{16}$ é 156111_8 , e esta emprega dois dígitos adicionais para representar a mesma quantidade. $\triangleleft$

Tabela 2.2: Conversão de 1987 para hexadecimal.

quociente	módulo	dígito	potência
$1987 \div 16 = 124,1875$	$1987 \% 16 = 0,1875 \times 16 = 3$	$d_0 = 3$	16^0
$124 \div 16 = 7,7500$	$124 \% 16 = 0,7500 \times 16 = C$	$d_1 = C$	16^1
$7 \div 16 = 0,4375$	$7 \% 16 = 0,4375 \times 16 = 7$	$d_2 = 7$	16^2

Tabela 2.3: Conversão de DC49 para octal.

quociente	módulo	dígito	potência
$DC49 \div 8 = 1B89$	$DC49 \% 8 = 9 \% 8 = 1$	$d_0 = 1$	8^0
$1B89 \div 8 = 371$	$1B89 \% 8 = 9 \% 8 = 1$	$d_1 = 1$	8^1
$371 \div 8 = 6E$	$371 \% 8 = 1 \% 8 = 1$	$d_2 = 1$	8^2
$6E \div 8 = D$	$6E \% 8 = 14_{10} \% 8 = 6$	$d_3 = 6$	8^3
$D \div 8 = 1$	$D \% 8 = 13_{10} \% 8 = 5$	$d_4 = 5$	8^4
$1 \div 8 = 0$	$1 \% 8 = 1 \% 8 = 1$	$d_5 = 1$	8^5

Na conversão do Exemplo 2.3 usamos uma propriedade do módulo. Como $16 = 2 \times 8$, ao computar o módulo de 8, pode-se usar somente o último dígito da representação hexadecimal para computar o resto da divisão inteira. Os dígitos para além do primeiro são múltiplos de 16, e portanto são também múltiplos de 8. O dígito relevante é mostrado em negrito na Tabela 2.3.

2.4 Um Dedo em Duas Mãos

Até agora vimos três representações para números naturais que são usadas porque facilitam o trabalho dos humanos mas que não são usadas internamente em sistemas digitais e nem nos circuitos de computadores. Os circuitos que empregamos para construir sistemas digitais representam números na base 2, ou na *representação binária*.

Na representação binária o alfabeto estendido é

$$A_2 = \{0, 1, \_, .\} \quad (2.8)$$

e a sequência de contagem é

$$0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, \dots$$

Números representados na base 2 não são nada compactos e é por isso que são usadas as representações octal ou hexadecimal. A Tabela 2.4 mostra as representações em binário, hexadecimal e octal dos primeiros 16 naturais.

A expressão *dígito binário* é abreviada para *bit*, do Inglês *binary digit*. Cada dígito hexadecimal é representado em 4 bits, e cada dígito octal em 3 bits. Na Tabela 2.4, por causa da reduzida faixa de valores, o dígito mais significativo da representação octal corresponde a um único bit.

Para uma representação com k dígitos, o valor do número natural B representado pela justaposição de k bits é obtido pela Equação 2.9, e é a soma dos dígitos d_i , $i \in [0, k)$ multiplicados pelas correspondentes potências de dois.

Tabela 2.4: Representação em decimal, binário, hexadecimal e octal.

decimal	binário	hexa.	octal
0	0000	0	00
1	0001	1	01
2	0010	2	02
3	0011	3	03
4	0100	4	04
5	0101	5	05
6	0110	6	06
7	0111	7	07
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

$$B = \sum_{i=0}^{k-1} d_i \times 2^i, \quad d \in A_2 \quad (2.9)$$

A conversão de um número na base 10 para um na base 2 emprega o algoritmo de conversão das seções anteriores. A sequência de divisões é algo mais longa porque a base é 2.

Exemplo 2.4 *Façamos a conversão de 123_{10} para sua representação binária. A conversão é mostrada na Tabela 2.5, e a representação na base 2 é 111.1011_2 . Sempre que o número por dividir for ímpar, o bit correspondente é 1.* $\triangleleft$

Tabela 2.5: Conversão de 123 para binário.

quociente	módulo	dígito	potência
$123 \div 2 = 61,5$	$123 \% 2 = 0,5 \times 2 = 1$	$d_0 = 1$	2^0
$61 \div 2 = 30,5$	$61 \% 2 = 0,5 \times 2 = 1$	$d_1 = 1$	2^1
$30 \div 2 = 15,0$	$30 \% 2 = 0,0 \times 2 = 0$	$d_2 = 0$	2^2
$15 \div 2 = 7,5$	$15 \% 2 = 0,5 \times 2 = 1$	$d_3 = 1$	2^3
$7 \div 2 = 3,5$	$7 \% 2 = 0,5 \times 2 = 1$	$d_4 = 1$	2^4
$3 \div 2 = 1,5$	$3 \% 2 = 0,5 \times 2 = 1$	$d_5 = 1$	2^5
$1 \div 2 = 0,5$	$1 \% 2 = 0,5 \times 2 = 1$	$d_6 = 1$	2^6

Para facilitar a leitura de números com muitos dígitos, os bits são agrupados em quartetos, de forma similar ao agrupamento de milhares.

As conversões entre binário e hexadecimal são triviais. Ao converter para hexadecimal, basta agrupar os bits de quatro em quatro e então converter cada quarteto segundo a Tabela 2.4. Ao converter para binário, cada dígito hexadecimal deve ser expandido para o quarteto que lhe corresponde.

Exemplo 2.5 *Façamos a conversão de $F1C7_{16}$ para sua representação binária. Ao consultar a Tabela 2.4 vemos que*

*$F \mapsto 1111$, $1 \mapsto 0001$, $C \mapsto 1100$, $7 \mapsto 0111$,
e $F1C7_{16} = 1111.0001.1100.0111_2$.*

<

Exemplo 2.6 *Façamos a conversão de $10111010011010010101101_2$ para hexadecimal. É necessário agrupar os quartetos, iniciando pela direita*

$101.1101.0011.0100.1010.1101_2$

e então consultar a Tabela 2.4:

$0101 \mapsto 5$, $1101 \mapsto D$, $0011 \mapsto 3$, $0100 \mapsto 4$, $1010 \mapsto A$, $1101 \mapsto D$.

Assim, $10111010011010010101101_2 = 5D.34AD_{16}$.

<

2.5 Uma Representação para Números Reais na Base 2

A Equação 2.3 define a representação posicional de números reais na base 10. A representação na base 2 é definida pela Equação 2.10, com m dígitos na parte inteira, e n na parte fracionária.

$$R = \left(\sum_{i=0}^{m-1} d_i \times 2^i \right), \left(\sum_{j=n}^{-1} d_j \times 2^j \right) \quad (2.10)$$

As quatro primeiras potências negativas de 2 são

$$2^{-1} = 0,5 \quad 2^{-2} = 0,25 \quad 2^{-3} = 0,125 \quad 2^{-4} = 0,0625.$$

O número $5,3125_{10}$ é representado na base dois por $101,0101_2$

$$5,3125_{10} = 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-2} + 1 \times 2^{-4} = 4 + 1 + 0,25 + 0,0625.$$

O algoritmo de conversão de base para a parte inteira é aquele da página 18. O algoritmo para converter a parte fracionária D da base 10 para a base 2 é definido abaixo.

Algoritmo de conversão de base de frações

- (1) multiplique D por 2;
- (2) guarde a parte fracionária para a próxima iteração;
- (3) se $D \times 2 \geq 1$, então o “dígito da vez” é 1, do contrário é 0;
- (4) repita os passos (1) a (3) até que a parte fracionária seja zero.

Esta sequência produz os dígitos do valor convertido do mais significativo para o menos significativo. As partes inteira e fracionária devem ser convertidas separadamente.

Exemplo 2.7 Vejamos a conversão de $0,123_{10}$ para binário empregando somente 8 bits na representação da fração. A Tabela 2.6 mostra os passos da conversão, para oito dígitos à direita da vírgula.

Da mesma forma que um computador emprega um número finito de bits para representar a parte inteira, o número de bits empregado para representar a fração também é finito. Neste exemplo, a conversão foi truncada no oitavo bit. Quanto maior a largura do campo usado para representar a parte fracionária dos números, maior é a precisão da representação.

Tabela 2.6: Conversão de 0,123 para binário.

produto	$D \times 2 \geq 1$	dígito	potência
$0,123 \times 2 = 0,246$	0	$d_{-1} = 0$	2^{-1}
$0,246 \times 2 = 0,492$	0	$d_{-2} = 0$	2^{-2}
$0,492 \times 2 = 0,984$	0	$d_{-3} = 0$	2^{-3}
$0,984 \times 2 = 1,968$	1	$d_{-4} = 1$	2^{-4}
$0,968 \times 2 = 1,936$	1	$d_{-5} = 1$	2^{-5}
$0,936 \times 2 = 1,872$	1	$d_{-6} = 1$	2^{-6}
$0,872 \times 2 = 1,744$	1	$d_{-7} = 1$	2^{-7}
$0,744 \times 2 = 1,488$	1	$d_{-8} = 1$	2^{-8}

O número $0,123_{10}$ representado em 8 bits é $0,0001.1111_2$. Este número convertido novamente para um número real é $0,12109375$, com um erro de representação que é $0,123 - 0,12109375 = 0,00190625$, ou de aproximadamente dois milésimos. Este erro é chamado de erro absoluto porque é a diferença entre o valor real e o valor representado. O erro relativo é a razão entre o erro de representação e o valor representado, e é uma medida da proporção entre o erro e o valor real. Neste exemplo, o erro relativo é $0,00190625/0,123 = 0,0155$, ou de 1,55%. $\triangleleft$

Considere uma representação com 4 bits na fração. Dos 16 valores possíveis ($\langle 0000 \rangle$ a $\langle 1111 \rangle$), o diagrama na Figura 2.1 evidencia cinco deles.

$$\left\{0, \frac{1}{2}, \left(\frac{1}{2} + \frac{1}{4}\right), \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8}\right), \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}\right)\right\}$$

As representações indicadas são exatas, e portanto o erro de representação para esses cinco valores é zero. O erro máximo de representação no intervalo $[0, 1)$ é $\approx \frac{1}{16}$ e ocorre entre cada par de valores representáveis. O erro entre o maior valor representável ($0,9375$) e o fim do intervalo é indicado pela faixa hachurada no diagrama.

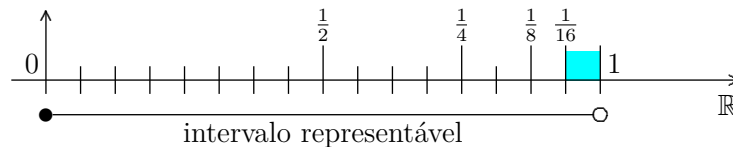


Figura 2.1: Valores representáveis com quatro bits na fração.

Os limites inferiores dos intervalos são representáveis exatamente $(0, \frac{1}{16}, \frac{2}{16}, \frac{3}{16}, \dots, \frac{15}{16})$ mas os limites superiores não o são porque os intervalos são abertos no limite superior: $[\frac{k}{16}, \frac{k+1}{16})$, $k \in [0, 15]$. Assim, o erro é uma aproximação assintótica a $\frac{1}{16}$. O erro máximo com uma representação com n bits é $\approx \frac{1}{2^n}$. O limite superior não faz parte do intervalo porque ele é o primeiro elemento do intervalo seguinte, aquele com a parte inteira que é 1 maior do que a ‘deste’ intervalo. O início do ‘próximo’ intervalo é 1 na Figura 2.1.

A representação com 4 bits é obviamente incompleta e insuficiente porque somente 16 valores podem ser representados exatamente; todos os demais devem ser aproximados para o valor representável mais próximo. Por exemplo, pode-se aproximar para baixo, quando se emprega o valor menor mais próximo de um valor representável, ou pode-se aproximar para cima e escolhe-se o valor próximo maior do que aquele por representar.

Exemplo 2.8 A constante e é chamada de número de Neper, ou de número de Euler, e é a base dos logaritmos naturais. Esta constante pode ser representada pela somatória dos inversos dos fatoriais:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots = \frac{1}{1} + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \cdots$$

A cada termo adicional, a soma fica mais próxima do valor de e . A representação de e com 15 dígitos após a vírgula é

$$2,718.281.828.459.045.$$

Se representarmos o número transcendental somente com os quatro primeiros termos da soma, obtemos

$$e_4 = \frac{1}{1} + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} = 2,66666 \cdots$$

e esta representação produz um erro de $0,05161 \cdots$ com relação à representação com 15 dígitos. Se melhorarmos a representação, empregando um termo a mais, obtemos

$$e_5 = \frac{1}{1} + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} = 2,708333 \cdots$$

com um erro de representação de $0,009948 \cdots$.

O termo erro de representação é usado para explicitar a influência que a forma de representar o número, com mais ou menos bits na parte fracionária, tem na precisão da representação. Quanto mais detalhado for o desenho de uma flor, melhor é a representação da flor real.

Qual é o limite para a quantidade de bits numa representação? Evidentemente, quanto maior o número de bits, melhor a qualidade dos valores representados. Um número infinito de bits permite representações perfeitas; contudo, a preços de 2020, pouquíssimas aplicações teriam custo razoável com tal representação. O Exemplo 2.9 ilustra as escolhas preço versus qualidade de representação. ◀

Exemplo 2.9 Você foi contratada para escrever o programa do controle remoto de um aparelho de ar condicionado residencial. O processador opera com inteiros de 8 bits e você deve projetar uma representação que permita exibir décimos de grau. A faixa de temperaturas de aplicação do aparelho é de -10°C a $+50^\circ\text{C}$, e o termômetro fornece medidas com precisão de um centésimo de centígrado.

A faixa de valores de temperatura é de 60 graus – ou 60 valores inteiros – e para representá-la são necessários 6 bits ($2^6=64$), logo sobram dois bits para representar a fração. Estes 2 bits não são suficientes para representar décimos de grau, conforme a especificação – com dois bits podem ser representados somente as frações 0, 0,25, 0,50 e 0,75.

Como outra possibilidade de representação pode-se usar um byte para representar a parte inteira e outro byte para a parte fracionária. Neste caso, sobram valores para as temperaturas inteiras ($256 \gg 60$) e as frações podem ser representadas com precisão de $1/256$, o que é um tanto melhor do que $1/10$. Esta representação é mais custosa mas atende ao especificado. ◀

Representação *versus* realidade Essa é uma questão fundamental e que está a dar nós na cabeça de vários dentre os leitores: uma coisa é a quantidade representada, outra coisa diferente é a representação da quantidade. $\frac{1}{3}$ e $0,33333\ldots$ são dois *desenhos* distintos que representam a mesma quantidade. O desenho pode ser em numerais romanos, indo-arábicos, Klingon, binário, hexadecimal ou decimal; essas todas podem ser usadas para representar uma mesma quantidade. A base com que se representa um número é só um *desenho*, uma representação, daquele número e não a *quantidade* ela própria.

Exercícios

Ex. 2.1 Escreva, em binário, todos os naturais entre zero e 31, representando-os todos com cinco bits.

Ex. 2.2 Supondo que os Klingons têm cinco dedos em cada mão, e considerando que a versão Klingon do alfabeto da Equação 2.1, estendido com o separador de milhares e o separador de frações, seja $\{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, *, \circ\}$, codifique as quantidades 123, 5.483,99 e π no alfabeto Klingon.

Ex. 2.3 Repita o Ex. 2.2 usando a representação empregada na Roma antiga.

Ex. 2.4 Some, e depois multiplique, 345 e 78, representados em numerais romanos.

Ex. 2.5 Converta 100, 1.000, 10.000 e 100.000 para octal e para hexadecimal.

Ex. 2.6 Converta 100_b , 1.000_b , 10.000_b e 100.000_b para decimal, com b valendo 8 e depois 16.

Ex. 2.7 Converta 100, 1.000, 10.000 e 100.000 para binário.

Ex. 2.8 Converta os números 133 e 1033 para binário e para hexadecimal.

Ex. 2.9 Encontre as potências de 2 (2^n) mais próximas de 10^m , $m \in \{3, 6, 9\}$.

Ex. 2.10 Estime o erro percentual entre as potências de 10 e as respectivas potências de 2 do Ex. 2.9.

Ex. 2.11 Converta $1/3$, $1/5$, $1/7$, $1/11$ e π para binário, empregando 4 bits na parte fracionária. Qual a magnitude dos erros de representação, arredondando para baixo? Considere números reais com 12 dígitos decimais após a vírgula.

Ex. 2.12 Repita o Ex. 2.11 empregando 8 bits na parte fracionária. Qual a magnitude dos erros de representação, arredondando para baixo?

Ex. 2.13 Repita o Ex. 2.11 empregando 12 bits na parte fracionária. Qual a magnitude dos erros de representação, arredondando para baixo?

Ex. 2.14 Estime o erro máximo de representação para reais representados com n bits na parte fracionária.

Ex. 2.15 Estime o erro mínimo de representação para reais representados com n bits na

parte fracionária.

Ex. 2.16 Considere o sistema do Exemplo 2.9. Uma representação com 12 bits atenderia à especificação? Qual é o número mínimo de bits necessário? Justifique sua resposta. Neste tipo de projeto, frequentemente é uma boa ideia empregar uma representação interna – que fica escondida do usuário – que seja melhor do que a representação externa – aquela que fica visível ao usuário.

Ex. 2.17 Ainda com relação ao Exemplo 2.9, suponha que o aparelho será empregado numa espaçonave que se dirige a Mercúrio. O controle deverá operar na faixa de -220°C a $+420^{\circ}\text{C}$, e manter a precisão de um décimo de grau. Quantos bits são necessários para representar a parte inteira e quantos para a parte fracionária? Justifique sua resposta.

Referências Bibliográficas

- [Ash08] Peter J Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 3rd edition, 2008. ISBN 978-0-12-088785-9.
- [BJ97] Gerrit A Blaauw and Frederick P Brooks Jr. *Computer Architecture: Concepts and Evolution*. Addison-Wesley, 1997. ISBN 0201105578.
- [Bob87] Leonard S Bobrow. *Elementary Linear Circuit Analysis*. Holt, Rinehart & Winston, 1987. ISBN 0030072980.
- [Bro04] Stuart Brorson. *Circuit simulation using gEDA and SPICE – HOWTO*, 2004. <<http://www.brorson.com/gEDA/SPICE/intro.html>>, 8/5/2012.
- [Car03] Nicholas Carter. *Arquitetura de Computadores*. Coleção Schaum. Bookman, 2003. ISBN 853630250x.
- [Cla80] Wesley A Clark. From electron mobility to logical structure: A view of integrated circuits. *ACM Computing Surveys*, 12(3):325–356, Set 1980.
- [Fle97] William I Fletcher. *An Engineering Approach to Digital Design*. Prentice Hall, 1997. ISBN 9780132776998.
- [Hex17] Roberto A Hexsel. Why it is so hard to write "system software"? To be submitted, 2017.
- [HJS00] M D Hill, N P Jouppi, and G S Sohi. *Readings in Computer Architecture*. Morgan Kaufmann, 2000. ISBN 1558605398.
- [HP90] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1st edition, 1990. ISBN 1558600698.
- [HP12] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.
- [HU79] John E Hopcroft and Jeffrey D Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. ISBN 020102988X.
- [Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993. ISBN 0070316228.
- [JNW08] B L Jacob, S W Ng, and D T Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008. ISBN 0123797513.
- [Kat94] Randy H Katz. *Contemporary Logic Design*. Benjamin-Cummings, 1994. ISBN 0805327037.
- [KB04] Randy H Katz and Gaetano Borriello. *Contemporary Logic Design*. Prentice Hall, 2004. ISBN 978-0201308570.
- [KL96] Sung-Mo Kang and Yusuf Leblebici. *CMOS Digital Integrated Circuits: Analysis and Design*. McGraw-Hill, 1996. ISBN 0070380465.
- [Koh78] Zvi Kohavi. *Switching and Finite Automata Theory*. Tata McGraw-Hill, 2nd edition, 1978. ISBN 0070993874.
- [Kor01] Israel Koren. *Computer Arithmetic Algorithms*. A K Peters, 2nd edition, 2001. ISBN 1568811608.
- [Lam84] Butler W Lampson. Hints for computer system design. *IEEE Software*, pages 11–28, Jan 1984.
- [Lip64] Seymour Lipschutz. *Set Theory and Related Topics*. Schaum Publishing, 1964.
- [Man02] M Morris Mano. *Digital Design*. Prentice Hall, 3rd edition, 2002. ISBN 01306211218.
- [MIP05a] MIPS. *MIPS32 Architecture for Programmers, Volume I: Introduction to the MIPS32 Architecture*, 2005.

- [MIP05b] MIPS. *MIPS32 Architecture for Programmers, Volume II: The MIPS32 Instruction Set*, 2005.
- [MIP05c] MIPS. *MIPS32 Architecture for Programmers, Volume III: The MIPS32 Privileged Resource Architecture*, 2005.
- [MK00] M Morris Mano and Charles R Kime. *Logic and Computer Design Fundamentals*. Prentice Hall, 2nd edition, 2000. ISBN 0130124680.
- [Mou01] Arnaldo V Moura. *Especificações em Z*. Editora da Unicamp, 2001. ISBN 8526805754.
- [PH94] David A Patterson and John L Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994. ISBN 155860281X.
- [PH14] David A Patterson and John L Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 5th edition, 2014.
- [PP03] Yale N Patt and Sanjay J Patel. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. McGraw-Hill, 2nd edition, 2003.
- [RCN03] Jan M Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits – A Design Perspective*. Prentice Hall, 2nd edition, 2003. ISBN 0130909963.
- [San90] Jeff W Sanders. Lectures on the foundations of hardware design. Lecture notes, Programming Research Group, Oxford University Computing Laboratory, 1990.
- [Sco65] Ronald E Scott. *Elements of Linear Circuits*. Addison-Wesley, 1965. ISBN 0201068427.
- [Spi89] J M Spivey. *The Z Notation*. Prentice Hall, 1989. ISBN 013983768X.
- [SS90] Adel S Sedra and Kenneth C Smith. *Microeletronic Circuits*. Holt, Rinehart & Winston, 3rd edition, 1990. ISBN 003051648X.
- [Tau82] Herbert Taub. *Digital Circuits and Microprocessors*. McGraw-Hill, 1982. ISBN 0070629455.
- [TS89] Herbert Taub and Donald Schilling. *Digital Integrated Electronics*. McGraw-Hill, 1989. ISBN 007Y857881.
- [WE82] Jim Welsh and John Elder. *Introduction to Pascal*. Prentice-Hall, 2nd edition, 1982. ISBN 0134915496.
- [WH10] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 4th edition, 2010. ISBN 0321547748.
- [Wol05] Wayne Wolf. *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann, 2005. ISBN 0123694590.

Índice Remissivo

Símbolos

$\%$, 18–20
 $\bar{a}$, *veja* $\neg$
 π , 25
 e , 24

A

alfabeto, 17
algoritmo,
 conversão de base, 18
 conversão de base de frações, 22
and, *veja* $\wedge$
aproximação, 24
assembly, *veja* ling. de montagem
atraso, *veja* tempo de contaminação
atribuição, 12

B

binário, 20
bit, 20
byte, 11

C

capture FF, *veja* *flip flop*, destino
clk, *veja* *clock*
Column Address Strobe, *veja* CAS
Complementary Metal-Oxide Semiconductor, *veja*

CMOS

complemento, *veja* $\neg$
comportamento transitório, *veja* transitório
condicional, *veja* $\triangleleft \triangleright$
conjunção, *veja* $\wedge$
conversão de base, 18

D

datapath, *veja* circuito de dados
decimal, 17
design unit, *veja* VHDL, unidade de projeto
disjunção, *veja* $\vee$

E

enviesado, relógio, *veja* *skew*
equivalência, *veja* $\Leftrightarrow$
erro,
 de representação, 23

F

Field Effect Transistor, *veja* FET
Field Programmable Gate Array, *veja* FPGA
flip-flop,

 modelo VHDL, *veja* VHDL, *flip-flop*
 um por estado, *veja* um FF por estado
frações, *veja* ponto fixo
frequência máxima, *veja* relógio
função, tipo (op. infixo), *veja* $\mapsto$

G

glitch, *veja* transitório
gramática, 17

H

hexadecimal, 19

I

implicação, *veja* $\Rightarrow$
informação, 16
Instrução,
 busca, *veja* busca
instrução, 12
 busca, *veja* busca
 decodificação, *veja* decodificação
 execução, *veja* execução
 resultado, *veja* resultado
interface,
 de rede, 13
 de vídeo, 12

L

latch, *veja* basculó
latch FF, *veja* *flip flop*, destino
launch FF, *veja* *flip flop*, fonte
linguagem,
 assembly, *veja* ling. de montagem

M

Máquina de Mealy, *veja* máq. de estados
Máquina de Moore, *veja* máq. de estados
Mealy, *veja* máq. de estados
memória,
 de vídeo, 13
 primária, 13
 secundária, 13
memória dinâmica, *veja* DRAM
memória estática, *veja* SRAM
módulo, *veja* $\%$, *mod*
Moore, *veja* máq. de estados
multiply-add, *veja* MADD

N

número,

de Euler, 24
 negação, veja $\neg$
 not, veja $\neg$

O

octal, 18
operation code, veja *opcode*
 or, veja $\vee$
 ou exclusivo, veja $\oplus$
 ou inclusivo, veja $\vee$

P

período mínimo, veja relógio
pipelining, veja segmentação
 piso, veja $\lfloor v \rfloor$
 porta lógica,
 carga, veja *fan-out*
precisão,
 representação, 23
 processador, 12
 programa de testes, veja VHDL, *testbench*
 pulso,
 espúrio, veja transitório

R

RAM, 12
Random Access Memory, veja RAM
Read Only Memory, veja ROM
Register Transfer Language, veja RTL
 registrador de deslocamento,
 modelo VHDL, veja VHDL, registrador
 relógio,
 enviesado, veja *skew*
 representação,
 binária, 20
 decimal, 17
 hexadecimal, 19
 octal, 18
 posicional, 17
 precisão, 23
 ROM, 12
Row Address Strobe, veja RAS

S

semântica, 17
síntese, veja VHDL, *síntese*
Solid State Disk, veja SSD
 soma, veja somador
 SSD, 14

T

tamanho, veja $|N|$
testbench, veja VHDL, *testbench*
 teto, veja $\lceil r \rceil$
three-state, veja *terceiro estado*
Tipo I, veja *formato*
Tipo J, veja *formato*
Tipo R, veja *formato*
transferência entre registradores, veja RTL
Transistor-Transistor Logic, veja TTL

transmission gate, veja *porta de transmissão*
 tupla, veja $\langle \rangle$

U

Unidade de Lógica e Aritmética, veja ULA

V

vetor de bits, veja $\langle \rangle$
 VHDL,
 design unit, veja VHDL, *unidade de projeto*

W

write back, veja *resultado*

X

xor, veja $\oplus$

Capítulo 3

Um Cálculo Axiomático para Bits

Los locos no están locos si uno acepta su razonamiento.

Gabriel García Márquez

Chamamos de *sinais* os valores transportados pelas conexões entre os componentes dos circuitos. Nos interessam os sinais que podem assumir um de dois possíveis valores, 0 e 1, e que variam no tempo. Tais sinais são chamados de *sinais digitais*, em contraste com os *sinais analógicos*, que podem assumir quaisquer níveis de um intervalo limitado por valores máximo e mínimo. Voltagem e temperatura são sinais analógicos enquanto que o estado de uma lâmpada (de LEDs) – acesa ou apagada – ou de uma fechadura – aberta ou fechada – é digital.

Este capítulo¹ define uma abstração para *sinais digitais* que permite tratá-los como se fossem entidades matemáticas simples. A abstração é baseada na Álgebra de Boole e é relevante para projetistas de circuitos porque simplifica enormemente a compreensão dos conceitos, bem como a manipulação algébrica dos operadores que relacionam logicamente os sinais.

A abstração de sinais elétricos como *bits*, para representar o comportamento de circuitos digitais, é útil se encarada como uma simplificação e idealização do comportamento dos sinais e das relações entre eles nos circuitos que implementam as funções lógicas. A abordagem empregada neste texto é inspirada em Sanders [San90] e o formalismo para especificar circuitos é baseado na linguagem formal Z [Spi89, Mou01].

Empregamos os símbolos ‘ $\wedge$ ’ para representar a conjunção e ‘ $\vee$ ’ para representar a disjunção para não sobrecarregar os operadores ‘.’ e ‘+’, e mais importante, para diferenciar operações sobre valores lógicos de operações aritméticas sobre valores numéricos.

3.1 Abstração de Sinais como Bits

A Figura 3.1 mostra quatro possíveis representações para um mesmo sinal que varia com o tempo. A *representação concreta* tenta reproduzir o comportamento elétrico de um sinal, como seria observado diretamente num circuito em operação. O sinal analógico varia continuamente entre V_M e V_m , e esta forma de onda aparenta estar contaminada por ruído gerado pelos circuitos próximos e pelo comportamento elétrico real do circuito que gera o sinal.

¹© Roberto André Hexsel, 2012-2020. Versão de 7 de dezembro de 2020.

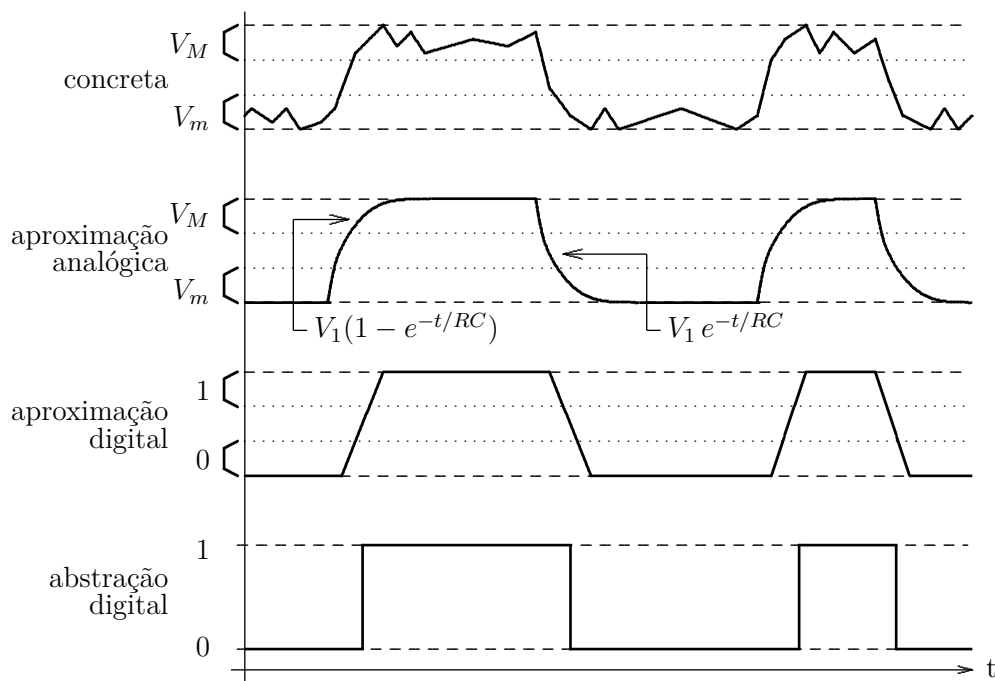


Figura 3.1: Quatro representações para o comportamento de um sinal.

Se o circuito que gera o sinal for simulado num “simulador analógico”, como o programa *spice*, o sinal teria uma forma de onda como a mostrada na segunda curva, que é uma *aproximação analógica* para o comportamento real. Neste tipo de simulação, equações diferenciais modelam correntes e tensões no circuito, e as mudanças de nível são descritas por funções do tempo com exponenciais, tais como $V_1(t) = V_1(1 - e^{-t/RC})$, para constantes R e C relacionadas aos parâmetros elétricos de resistência e capacitância dos circuitos nos quais a tensão é avaliada. Voltaremos a este assunto no Capítulo 5.

As duas últimas curvas são idealizações, ou *representações abstratas* para o comportamento real. Na chamada *aproximação digital*, as exponenciais são aproximadas por retas que indicam que a mudança do nível mínimo, que representa 0, para o nível máximo, que representa 1, não é instantânea – o instante do início não é o mesmo do final da transição. Na *abstração digital*, as transições entre 0 e 1 são instantâneas e não há ambiguidade na interpretação dos níveis lógicos do sinal.

Nas representações concreta e analógica, entre as faixas de tensão que correspondem aos níveis lógicos 1 e 0, há uma faixa de transição em que o sinal é indeterminado, porque não é nem 1 nem 0. Os projetistas dos circuitos tentam fazer com que a duração dos intervalos em que os sinais estão indeterminados seja a menor possível.

A aproximação digital é usada em alguns dos diagramas de tempo mostrados ao longo do texto para lembrar ao leitor de que as mudanças de nível dos sinais não são instantâneas. A representação idealizada e abstrata, na qual os sinais elétricos são representados por *bits*, é empregada para descrever o comportamento dos circuitos, quando aqueles estão em estado de repouso, após cessarem os efeitos de todas as transições nos sinais.

3.2 Bits, Operações e Propriedades

Definição 3.1 (Bits 0 e 1) *O conjunto $\mathbb{B}$ tem dois elementos distintos 0 e 1 chamados de bits, $\mathbb{B} = \{0, 1\}$.*

Definição 3.2 (Tupla) *Uma tupla de k bits é um produto cartesiano com k parcelas:*

$$\mathbb{B}^k = \overbrace{\mathbb{B} \times \mathbb{B} \times \cdots \times \mathbb{B}}^{k \text{ parcelas}}$$

Uma tupla de bits pode ser representada pela mera justaposição de bits, como 1 ou 1010, ou com constantes ou variáveis agrupadas por colchetes angulares como $\langle a, b, c \rangle$.

Definição 3.3 (Tipo de Função) *O tipo de uma função com domínio $\mathcal{D}$ e imagem $\mathcal{I}$ é denotado por*

$$\mathcal{D} \mapsto \mathcal{I}$$

e o tipo de uma função f com domínio $\mathcal{D}$ e imagem $\mathcal{I}$ é especificado por

$$f : \mathcal{D} \mapsto \mathcal{I}$$

que é lido “ f é uma função que mapeia elementos de $\mathcal{D}$ em $\mathcal{I}$ ”. O operador $\mapsto$ indica o mapeamento, ou a associação, entre o conjunto dos argumentos e o conjunto dos valores da função.

Definição 3.4 (Operador Unário) *Um operador f_1 sobre bits é chamado de unário se tem um bit como argumento e produz um bit como resultado:*

$$f_1 : \mathbb{B} \mapsto \mathbb{B}$$

Definição 3.5 (Operador Binário) *Um operador f_2 sobre bits é chamado de binário se tem dois bits como argumento e produz um bit como resultado:*

$$f_2 : \mathbb{B}^2 \mapsto \mathbb{B}$$

O subscrito no nome da função indica a *aridade* de um operador, ou o “número de entradas”, é geralmente omitido porque tal informação pode ser facilmente depreendida do contexto no qual o operador é usado.

Definição 3.6 (Conjunção, Disjunção, Complemento) Três operadores sobre bits, chamados de conjunção, disjunção e complemento, são definidos na Tabela 3.1, e são denotados por $\wedge$, $\vee$ e $\neg$, respectivamente.

Tabela 3.1: Definição dos operadores conjunção, disjunção e complemento.

conjunção			disjunção			complemento	
a	b	$a \wedge b$	c	d	$c \vee d$	e	$\neg e$
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

O resultado de uma conjunção é 1 se *todos* os argumentos são 1. O resultado de uma disjunção é 1 se *algum* dos argumentos é 1. A disjunção é também chamada de *ou-inclusivo* porque além da condição “um mas não o outro”, a condição “um mais o outro” produz também resultado 1. A negação é também representada por $\bar{a} = \neg a$. Os operadores *conjunção*, *disjunção* e *complemento* são funções.

A tabela para o complemento mostra *todas* as combinações dos valores possíveis para o argumento $e \in \mathbb{B}$, quais sejam 0 e 1. As tabelas para a conjunção e a disjunção contêm uma linha para cada uma de *todas* as combinações possíveis dos dois argumentos. Para a conjunção, as combinações possíveis para o par $a, b \in \mathbb{B}$ são $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$.

Definição 3.7 (Variável em $\mathbb{B}$) Uma variável é um símbolo, ou um nome, que representa um valor em $\mathbb{B}$.

Definição 3.8 (Expressão em $\mathbb{B}$) Uma expressão representa um valor em $\mathbb{B}$ e consiste de variáveis relacionadas por operadores.

Uma variável pode conter um valor que é 1 ou 0, sendo ela própria uma expressão. A expressão que representa a informação “ p é 1 se ambos q e r são também 1” pode ser escrita como $p = q \wedge r$ e é lida como “ p é a conjunção de q e de r ”, ou ainda “ p é 1 se todos q e r são 1”.

Definição 3.9 (Precedência) A precedência da aplicação dos operadores é $\neg$, $\wedge$ e $\vee$, e esta ordem pode ser alterada com o uso de parênteses.

Definição 3.10 (Valor de uma Função) Quando valores em $\mathbb{B}$ são atribuídos a todas as variáveis de uma expressão $\mathcal{E}$, o resultado da avaliação desta expressão é chamado de o valor da função $\mathcal{E}$.

Definição 3.11 (Igualdade) Duas expressões $\mathcal{E}(a, b, \dots, z)$ e $\mathcal{F}(a, b, \dots, z)$ são ditas iguais se, ao atribuir todas as combinações de valores em $\mathbb{B}$ às variáveis $a, b, \dots, z$, então para todas as combinações das entradas obtém-se $\mathcal{E}(a, b, \dots, z) = \mathcal{F}(a, b, \dots, z)$.

Propriedades dos operadores Os operadores sobre bits exibem as propriedades listadas abaixo. Como definidos na Tabela 3.1, os operadores conjunção e disjunção são funções binárias; o operador complemento é uma função unária.

Identidade (i) A *identidade* com relação a $\vee$ é chamada de 0:

$$\exists 0 \in \mathbb{B} \bullet \forall a \in \mathbb{B} \bullet a \vee 0 = a$$

(ii) A *identidade* com relação a $\wedge$ é chamada de 1:

$$\exists 1 \in \mathbb{B} \bullet \forall a \in \mathbb{B} \bullet a \wedge 1 = a$$

Máximo e Mínimo O *elemento máximo* de $\mathbb{B}$ é 1, e o *elemento mínimo* de $\mathbb{B}$ é 0:

$$\forall a \in \mathbb{B} \bullet a \vee 1 = 1, \\ a \wedge 0 = 0$$

Complemento A conjunção (disjunção) de a com seu complemento resulta no elemento mínimo (máximo) de $\mathbb{B}$:

$$\forall a \in \mathbb{B} \bullet \exists \neg a \in \mathbb{B} \bullet a \wedge \neg a = 0, \\ a \vee \neg a = 1$$

Fechamento As operações sobre bits resultam em bits:

$$\forall a, b \in \mathbb{B} \bullet (a \wedge b) \in \mathbb{B}, (a \vee b) \in \mathbb{B}, \neg a \in \mathbb{B}$$

Involução O complemento do complemento de $a \in \mathbb{B}$ é o próprio a :

$$\forall a \in \mathbb{B} \bullet \neg(\neg a) = a$$

Idempotência A conjunção, e a disjunção, de a com a é o próprio a :

$$\forall a \in \mathbb{B} \bullet a \wedge a = a, \\ a \vee a = a$$

Comutatividade A ordem dos operandos não altera o resultado:

$$\forall a, b \in \mathbb{B} \bullet a \wedge b = b \wedge a, \\ a \vee b = b \vee a$$

Associatividade A ordem de aplicação de um mesmo operador não altera o resultado:

$$\forall a, b, c \in \mathbb{B} \bullet (a \wedge b) \wedge c = a \wedge (b \wedge c), \\ (a \vee b) \vee c = a \vee (b \vee c)$$

Distributividade O operador conjunção distribui sobre a operação de disjunção, e o operador disjunção distribui sobre a operação de conjunção:

$$\forall a, b, c \in \mathbb{B} \bullet a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c), \\ a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

Teorema de DeMorgan O complemento de uma conjunção é a disjunção das entradas complementadas, e o complemento de uma disjunção é a conjunção das entradas complementadas.

$$\forall a, b \in \mathbb{B} \bullet \neg(a \wedge b) = (\neg a \vee \neg b) \\ \neg(a \vee b) = (\neg a \wedge \neg b)$$

Definição 3.12 (Dual) Define-se o dual de uma expressão sobre bits como a expressão que é obtida pela troca simultânea de todas as ocorrências de $\wedge$ por $\vee$, de todas as ocorrências de $\vee$ por $\wedge$, e de todas as ocorrências das constantes 0 por 1, e de 1 por 0.

Definição 3.13 (Princípio da Dualidade) O dual de uma propriedade é também uma propriedade.

O resultado da aplicação de um operador binário a um par de vetores de bits é determinado pela Definição 3.15.

Definição 3.14 (Elemento de Tupla) Seja uma tupla A com n elementos de $\mathbb{B}$; o i -ésimo elemento da tupla é denotado por a_i , para $0 \leq i < n$.

Definição 3.15 (Aplicação Bit a Bit) A aplicação de uma função binária $f_2 : \mathbb{B}^2 \mapsto \mathbb{B}$, bit a bit, aos elementos de um par de tuplas $A, B \in \mathbb{B}^k$ produz a tupla $C \in \mathbb{B}^k$, tal que

$$\forall i, 0 \leq i < k \bullet c_i = f_2(a_i, b_i)$$

Exemplo 3.1 Considere o vetor $A = \langle 0, 0, 0, 1, 1, 1, 0, 0 \rangle$, representado mais concisamente por $A = 0001.1100$, e o vetor $B = \langle 0, 1, 0, 1, 1, 0, 0, 1 \rangle$, ou $B = 0101.1001$. O “ponto de milhar” é usado para agrupar quartetos de bits.

O vetor $C = A \wedge B$ resulta da aplicação, bit a bit, do operador $\wedge$

$$c_7 = a_7 \wedge b_7, c_6 = a_6 \wedge b_6, c_5 = a_5 \wedge b_5, \dots, c_1 = a_1 \wedge b_1, c_0 = a_0 \wedge b_0$$

ou

índice	7	6	5	4	3	2	1	0
A	0	0	0	1	1	1	0	0
	$\wedge$	$\wedge$	$\wedge$	$\wedge$	$\wedge$	$\wedge$	$\wedge$	$\wedge$
B	0	1	0	1	1	0	0	1
$A \wedge B = C$	0	0	0	1	1	0	0	0

e portanto $C = 0001.1000$. Frequentemente, operações como esta são usadas para isolar um segmento do vetor de bits. Neste caso, $A \wedge B$ isola a tripla $\langle b_4, b_3, b_2 \rangle$ ao fazer com que os demais elementos do vetor B sejam todos convertidos para 0 na atribuição à C . O vetor A é chamado de ‘máscara’ porque os bits em 0 do vetor A ‘mascaram’ os valores de B nas posições cujos valores são 0. ◀

A Definição 3.16 permite estender um operador binário para aplicação sobre tuplas com mais do que dois argumentos.

Definição 3.16 (Redução) Se $f_2 : \mathbb{B}^2 \mapsto \mathbb{B}$ é associativa, então $f_k : \mathbb{B}^k \mapsto \mathbb{B}$ é a extensão de f_2 para k entradas, e é obtida por repetidas aplicações de f_2

$$f_k(b_0, b_1, \dots, b_{k-1}) = f_2(b_0, f_{k-1}(b_1, \dots, b_{k-1}))$$

O resultado da aplicação de f_k à tupla $B = \langle b_0, \dots, b_{k-1} \rangle$ é chamado de *redução de B por f* . O somatório de uma série de números S é a redução por soma de S . O produtório dos elementos de S é a redução pelo produto.

Exemplo 3.2 Suponha que um vetor de bits é usado para contar a população de uma certa região: se o i -ésimo elemento do vetor é 1, então a i -ésima área está ocupada, do contrário aquela área está despovoada. Podemos usar uma redução para descobrir se a região está vazia: $v = 0$ somente se nenhuma das k áreas estiver ocupada: $v = \bigvee_k(a_0, a_1, \dots, a_{k-1})$. $\triangleleft$

Definição 3.17 (Tabela Verdade) A tabela verdade da expressão $\mathcal{E}(a, b, \dots, z)$ é construída pela atribuição de valores em $\mathbb{B}$ a todas combinações das variáveis e pela avaliação dos operadores em $\mathcal{E}$.

Para uma expressão com n variáveis as linhas da tabela são enumeradas de 0 a $2^n - 1$. Por convenção, as variáveis são dispostas em ordem alfabética, de tal forma que a letra mais próxima de ‘a’ tenha peso de 2^{n-1} , e a mais próxima de ‘z’ tenha peso 2^0 . Para a expressão $\mathcal{E}(a, b, c)$, a vale 2^2 , b vale 2^1 e c vale 2^0 . Para atribuir valores às combinações de variáveis, quando uma variável está complementada ela é considerada como 0, do contrário ela é considerada como 1. As linhas de número 0, 3 e 5 correspondem às triplas $\langle \bar{a}\bar{b}\bar{c} \rangle = \langle 000 \rangle$, $\langle \bar{a}bc \rangle = \langle 011 \rangle$ e $\langle a\bar{b}c \rangle = \langle 101 \rangle$, respectivamente.

A Tabela 3.1 define o resultado das operações binárias conjunção, disjunção e complemento para todas as combinações possíveis dos argumentos. Uma *tabela verdade* pode ser usada para comprovar as propriedades dos operadores. Vejamos como comprovar a comutatividade da conjunção. A Tabela 3.2 mostra os três passos necessários para a comprovação. A última linha da tabela indica em qual passo os valores de cada coluna foram gerados; a coluna do último passo, com o maior número, contém a comprovação da igualdade, se for toda preenchida com 1, ou sua refutação, se algum dos valores for 0. A tabela verdade de uma expressão com n variáveis tem 2^n linhas.

Tabela 3.2: Construção da tabela verdade para $a \wedge b = b \wedge a$.

(1) atribuição dos valores às variáveis

a	b	a	$\wedge$	b	$=$	b	$\wedge$	a
0	0	0		0		0		0
0	1	0		1		1		0
1	0	1		0		0		1
1	1	1		1		1		1
<i>passo</i>		1		1		1		1

(2) cômputo das conjunções

a	b	a	$\wedge$	b	$=$	b	$\wedge$	a
0	0	0	0	0		0	0	0
0	1	0	0	1		1	0	0
1	0	1	0	0		0	0	1
1	1	1	1	1		1	1	1
<i>passo</i>		1	2	1		1	2	1

(3) verificação da igualdade

a	b	a	$\wedge$	b	$=$	b	$\wedge$	a
0	0	0			1	0		
0	1	0			1	0		
1	0	0			1	0		
1	1	1			1	1		
<i>passo</i>		1	2	1	3	1	2	1

Os valores das variáveis são atribuídos aos argumentos dos operadores no passo (1); no passo (2) os resultados das conjunções são inseridos na coluna dos operadores $\wedge$ (mostrados em negrito). No passo (3), os resultados das conjunções são comparados, linha a linha. Se iguais, **1** é inserido na coluna da igualdade, ou **0** caso sejam diferentes. Uma coluna de igualdade preenchida somente com **1**s comprova a propriedade – se os dois lados da igualdade são idênticos em todas as linhas, então a igualdade é válida para todas as combinações de valores das variáveis. Os valores de a e b foram omitidos no último passo para facilitar a visualização.

Exemplo 3.3 Comprovemos a distributividade: $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$. A Tabela 3.3 mostra os três primeiros passos da comprovação. Como a propriedade envolve três variáveis, a tabela tem $2^3 = 8$ linhas.

No passo (2), são computados os valores das operações dentro dos parênteses, e no passo (3) são efetuadas as operações fora dos parênteses. Na terceira tabela, os valores desnecessários foram omitidos para facilitar a visualização. Para comprovar a propriedade, no quarto passo basta verificar que as duas colunas geradas no passo 3 são idênticas. $\triangleleft$

Tabela 3.3: Tabela verdade para $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$.

(1) atribuição dos valores às variáveis									
a	b	c	$a \vee (b \wedge c)$	$=$	$(a \vee b) \wedge (a \vee c)$				
0	0	0	0		0	0	0	0	
0	0	1	0		0	0	0	1	
0	1	0	0		0	1	0	0	
0	1	1	0		0	1	0	1	
1	0	0	1		1	0	1	0	
1	0	1	1		1	0	1	1	
1	1	0	1		1	1	1	0	
1	1	1	1		1	1	1	1	
<i>passo</i>			1		1	1	1	1	

(2) operações dentro dos parênteses									
a	b	c	$a \vee (b \wedge c)$	$=$	$(a \vee b) \wedge (a \vee c)$				
0	0	0	0		0	0	0	0	
0	0	1	0		0	0	0	1	
0	1	0	0		0	1	0	0	
0	1	1	0		0	1	1	1	
1	0	0	1		1	0	1	0	
1	0	1	1		1	0	1	1	
1	1	0	1		1	1	1	0	
1	1	1	1		1	1	1	1	
<i>passo</i>			1		1	2	1	1	

(3) operações fora dos parênteses									
a	b	c	$a \vee (b \wedge c)$	$=$	$(a \vee b) \wedge (a \vee c)$				
0	0	0	0		0	0	0		
0	0	1	0		0	0	1		
0	1	0	0		1	0	0		
0	1	1	0		1	1	1		
1	0	0	1		1	0	1		
1	0	1	1		1	0	1		
1	1	0	1		1	1	1		
1	1	1	1		1	1	1		
<i>passo</i>			1		1	3	1	2	

(4) verificação de igualdade									
a	b	c	$a \vee (b \wedge c)$	$=$	$(a \vee b) \wedge (a \vee c)$				
0	0	0	0		1	0			
0	0	1	0		1	0			
0	1	0	0		1	0			
0	1	1	1		1	1			
1	0	0	1		1	1			
1	0	1	1		1	1			
1	1	0	1		1	1			
1	1	1	1		1	1			
<i>passo</i>			1		1	3	1	2	

Definição 3.18 (Operadores Adicionais) *Expressões podem ser combinadas pelo uso dos operadores lógicos listados nas Tabelas 3.4 e 3.5.*

Tabela 3.4: Operadores Lógicos sobre bits.

símbolo	operador	descrição
$\equiv$	“pode ser reescrito como”	(<i>meta símbolo</i>)
$=$	igualdade	
$\neg$	negação	
$\wedge$	conjunção	
$\vee$	disjunção	
$\oplus$	ou-exclusivo	$(a \oplus b \equiv \neg a \wedge b \vee a \wedge \neg b)$
$\Rightarrow$	implicação	$(a \Rightarrow b \equiv \neg a \vee b)$
$\Leftrightarrow$	equivalência	$(a \Leftrightarrow b \equiv a \Rightarrow b \wedge b \Rightarrow a)$
$\triangleleft \triangleright$	condicional	$(a \triangleleft c \triangleright b \equiv c \wedge a \vee \neg c \wedge b)$

Tabela 3.5: Definição dos operadores ou-exclusivo, implicação e equivalência.

ou-exclusivo			implicação			equivalência		
a	b	$a \oplus b$	c	d	$c \Rightarrow d$	e	f	$e \Leftrightarrow f$
0	0	0	0	0	1	0	0	1
0	1	1	0	1	1	0	1	0
1	0	1	1	0	0	1	0	0
1	1	0	1	1	1	1	1	1

Exemplo 3.4 O operador ou-exclusivo é usado numa série de aplicações interessantes. Em especial, ele é usado no circuito que efetua somas. Uma propriedade útil deste operador é

$$a \oplus \bar{b} = \overline{a \oplus b}$$

e a Tabela 3.6 a comprova. A operação ou-exclusivo produz 1 se os dois operandos são distintos. A negação do ou-exclusivo testa se os operandos são iguais, e esta é uma operação deveras popular.

Para verificar a propriedade, a coluna 2 do lado esquerdo deve ser comparada com a coluna 3 do lado direito. $\triangleleft$

Tabela 3.6: Tabela verdade para $a \oplus \bar{b} = \overline{a \oplus b}$.

a	b	$a \oplus \bar{b}$	$=$	$\neg (a \oplus b)$
0	0	0 1 1	1	1 0 0 0
0	1	0 0 0	1	0 0 1 1
1	0	1 0 1	1	0 1 1 0
1	1	1 1 0	1	1 1 0 1
<i>passo</i>		1 2 1	4	3 1 2 1

Exemplo 3.5 Na expressão $a \Rightarrow b$, dizemos que a é o *antecedente* e b é o *consequente*, e ela é interpretada como “se o antecedente a é verdade, então o consequente b é verdade”. A expressão $a \Rightarrow (b \Rightarrow c)$ é interpretada como “se a é verdade, então se b é verdade, então c é verdade”. Essa afirmação equivale a “se ambos a e b são verdade, então c é verdade”. A equivalência entre as duas

$$a \Rightarrow (b \Rightarrow c) \Leftrightarrow (a \wedge b) \Rightarrow c$$

é demonstrada na Tabela 3.7; deve-se comparar a coluna 3 nos dois lados da equivalência. $\triangleleft$

Tabela 3.7: Tabela verdade para $[a \Rightarrow (b \Rightarrow c)] \Leftrightarrow (a \wedge b) \Rightarrow c$.

a	b	c	a	$\Rightarrow$	$(b \Rightarrow c)$	$\Leftrightarrow$	$(a \wedge b) \Rightarrow c$
0	0	0	0	1	0	1	0
0	0	1	0	1	1	1	0
0	1	0	0	1	0	1	0
0	1	1	0	1	1	1	0
1	0	0	1	1	0	1	1
1	0	1	1	1	1	1	1
1	1	0	1	0	0	1	0
1	1	1	1	1	1	1	1
<i>passo</i>			1	3	1	2	1

Exemplo 3.6 Vejamos como comprovar a *Lei do Silogismo*

$$[(a \Rightarrow b) \wedge (b \Rightarrow c)] \Rightarrow (a \Rightarrow c) \quad (3.1)$$

que é empregada em cadeias de deduções: “se a faz com que b seja verdade, e b faz com que c seja verdade, então a faz com que c seja verdade”. A Tabela 3.8 a comprova: a coluna 3 do lado esquerdo deve ser comparada à coluna 2 do lado direito.

A Lei do Silogismo nos permite concluir absurdos a partir de premissas corretas, em argumentos ditos *non sequitur*, porque a conclusão não segue das premissas. Por exemplo, (i) *se uma pessoa não usa máscara, então ela contrai COVID-19; e (ii) se uma pessoa tem COVID-19, então ela morre; (iii) logo, quem não usa máscara morre*. Este tipo de raciocínio é encontrado com dolorosa frequência em discussões nas tais “redes sociais”. Os argumentos individuais são ‘corretos’ mas a conclusão não decorre do encadeamento dos antecedentes. $\triangleleft$

Tabela 3.8: Tabela verdade para Lei do Silogismo.

a	b	c	$(a \Rightarrow b) \wedge (b \Rightarrow c)$	$\Rightarrow$	$(a \Rightarrow c)$
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	1	1	1
<i>passo</i>			1	2	1

3.2.1 Simplificação de Expressões

Uma função com variáveis que pertencem a $\mathbb{B}$ é representada da forma usual, como na Equação 3.2. O valor de uma função f de n variáveis pode ser computado, e este valor pode ser atribuído a uma variável.

$$a, b, \dots, y, z \in \mathbb{B}, f_n : \mathbb{B}^n \mapsto \mathbb{B} \bullet z = f_n(\overbrace{a, b, \dots, y}^n) \quad (3.2)$$

O resultado da avaliação pode ser atribuído a uma variável que não aparece na expressão, para uso em outras funções ou expressões. A restrição de que a variável a que é atribuído o valor da função não pode estar no corpo da função é uma restrição importante e que somente será levantada no Capítulo 8. O método para a atribuição de valores às variáveis, para o cálculo do valor da função, é aquele usado para construir tabelas verdade.

Definição 3.19 (Literal) *Um literal numa expressão corresponde a uma variável ou o ao seu complemento.*

Definição 3.20 (Simplificação) *A expressão $\mathcal{E}'(a, \dots, z)$ para $a, \dots, z \in \mathbb{B}$ é uma versão simplificada da expressão $\mathcal{E}(a, \dots, z)$ se, para todas as combinações possíveis dos valores de $a, \dots, z$, a avaliação de $\mathcal{E}'$ produz os mesmos valores que a avaliação de $\mathcal{E}$, e $\mathcal{E}'$ tem um menor número de operadores e/ou de literais do que $\mathcal{E}$.*

O objetivo da simplificação de expressões é obter uma expressão mais simples, possivelmente com menos literais, e com menor número de operadores, mas que representa a mesma função dos argumentos que a expressão original. A implementação da função simplificada resulta num circuito mais simples, de menor custo, e possivelmente com maior velocidade de operação.

Exemplo 3.7 Simplifiquemos a expressão abaixo, empregando as propriedades dos operadores em $\mathbb{B}$. A cada passo da simplificação, a coluna da direita indica a propriedade que o justifica.

$$\begin{array}{ll} (a \vee b) \wedge \neg a & \text{distributiva} \\ \Leftrightarrow (a \wedge \neg a) \vee (b \wedge \neg a) & \text{complemento } \wedge \\ \Leftrightarrow 0 \vee (b \wedge \neg a) & \text{identidade } \vee \\ \Leftrightarrow b \wedge \neg a & \end{array}$$

A expressão original é expandida pela distributividade para que então a subexpressão $(a \wedge \neg a)$ possa ser eliminada. Isso é possível porque $(a \wedge \neg a) = 0$ e $(0 \vee x) = x$. A expressão original tem três literais e três operadores enquanto a simplificada tem dois literais e dois operadores. $\triangleleft$

Exemplo 3.8 Simplifiquemos a expressão abaixo.

$$\begin{aligned}
 & a \vee (a \wedge b) && \text{identidade } \wedge \\
 \Leftrightarrow & (a \wedge 1) \vee (a \wedge b) && \text{distributiva} \\
 \Leftrightarrow & a \wedge (1 \vee b) && \text{elem. máximo } \vee \\
 \Leftrightarrow & a \wedge 1 && \text{identidade } \wedge \\
 \Leftrightarrow & a &&
 \end{aligned}$$

A expressão original é expandida com a identidade do $\wedge$ ($a \wedge 1 = a$) para então aplicarmos a distributividade, de onde obtemos a subexpressão $(1 \vee b) = 1$. A subexpressão $(a \wedge 1)$ pode ser simplificada com outra aplicação da identidade do $\wedge$. A expressão original tem três literais e dois operandos; a simplificada é somente o literal a . $\triangleleft$

Exemplo 3.9 Simplifiquemos a expressão abaixo, eliminando todos os parênteses desnecessários.

$$\begin{aligned}
 & (((\neg a) \wedge b) \vee (c \wedge (\neg d)) \vee (b \wedge (\neg d)) \vee (a \wedge (\neg b))) && \text{precedência } \neg \\
 \Leftrightarrow & ((\neg a \wedge b) \vee (c \wedge \neg d) \vee (b \wedge \neg d) \vee (a \wedge \neg b)) && \text{precedência } \wedge \\
 \Leftrightarrow & \neg a \wedge b \vee c \wedge \neg d \vee b \wedge \neg d \vee a \wedge \neg b &&
 \end{aligned}$$

A primeira regra elimina os parênteses à volta das negações porque o operador $\neg$ é o que se liga mais fortemente às variáveis ou subexpressões. A segunda regra elimina os parênteses à volta das conjunções porque o operador $\wedge$ liga mais fortemente variáveis ou subexpressões do que o operador $\vee$. Em que pese a simplificação, a segunda linha parece ser mais gentil aos (meus) olhos do que a terceira. $\triangleleft$

Exemplo 3.10 Simplifiquemos a expressão abaixo usando as propriedades das operações sobre os elementos de $\mathbb{B}$.

$$\begin{aligned}
 & \bar{a} \wedge b \vee c \wedge \bar{d} \vee b \wedge \bar{d} \vee a \wedge b && \text{precedência } \wedge \\
 \Leftrightarrow & (\bar{a} \wedge b) \vee (c \wedge \bar{d}) \vee (b \wedge \bar{d}) \vee (a \wedge b) && \text{comutativa} \\
 \Leftrightarrow & [(\bar{a} \wedge b) \vee (a \wedge b)] \vee (c \wedge \bar{d}) \vee (b \wedge \bar{d}) && \text{distributiva (b)} \\
 \Leftrightarrow & [(\bar{a} \vee a) \wedge b] \vee (c \wedge \bar{d}) \vee (b \wedge \bar{d}) && \text{complemento } (a \vee \bar{a}) \\
 \Leftrightarrow & [1 \wedge b] \vee (c \wedge \bar{d}) \vee (b \wedge \bar{d}) && \text{identidade } \wedge, \text{comutativa} \\
 \Leftrightarrow & b \vee (b \wedge \bar{d}) \vee (c \wedge \bar{d}) && \text{identidade } \wedge \\
 \Leftrightarrow & [(b \wedge 1) \vee (b \wedge \bar{d})] \vee (c \wedge \bar{d}) && \text{distributiva (b)} \\
 \Leftrightarrow & [b \wedge (1 \vee \bar{d})] \vee (c \wedge \bar{d}) && \text{identidade } \vee \\
 \Leftrightarrow & [b \wedge 1] \vee (c \wedge \bar{d}) && \text{identidade } \wedge \\
 \Leftrightarrow & b \vee (c \wedge \bar{d}) &&
 \end{aligned}$$

A primeira alteração agrupa as subexpressões, em função da precedência dos operadores, para simplificar as próximas manipulações. A aplicação da comutatividade aproxima as duas subexpressões em que aparecem os literais a e b . A distributividade separa b e faz $a \wedge \bar{a}$, o que permite eliminar os literais em a . A identidade $b = b \wedge 1$ é introduzida para permitir a eliminação do literal $\bar{d}$. A expressão da primeira linha tem dez operadores, enquanto a da última tem somente três. O número de literais cai de cinco para três. $\triangleleft$

A uma primeira vista, a escolha de quais regras aplicar pode parecer feitiçaria, e em alguma medida o é. Há esperança e é possível acelerar o aprendizado. Tente repetir os exemplos, copiando somente a expressão por simplificar e a coluna das propriedades, e então efetue as transformações indicadas.

Exemplo 3.11 Simplifiquemos a expressão abaixo.

$$\begin{aligned}
& \neg a \wedge b \wedge \neg c \vee \neg a \wedge \neg b \wedge \neg c \vee a \wedge b \wedge \neg c \vee a \wedge \neg b \wedge \neg c \\
\Leftrightarrow & (\neg a \wedge b \wedge \neg c) \vee (\neg a \wedge \neg b \wedge \neg c) \vee (a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) && \text{precedência } \wedge \\
\Leftrightarrow & (\neg a \wedge \neg c \wedge b) \vee (\neg a \wedge \neg c \wedge \neg b) \vee (a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) && \text{comutativa } 2\times \\
\Leftrightarrow & [(\neg a \wedge \neg c) \wedge b] \vee [(\neg a \wedge \neg c) \wedge \neg b] \vee (a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) && \text{associativa } 2\times \\
\Leftrightarrow & [(\neg a \wedge \neg c) \wedge (b \vee \neg b)] \vee (a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) && \text{distributiva } (\neg a \wedge \neg c) \\
\Leftrightarrow & [\neg a \wedge \neg c] \vee (a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) && \text{compl } \vee, \text{ ident } \wedge \\
\Leftrightarrow & [\neg a \wedge \neg c] \vee [(a \wedge \neg c) \wedge b] \vee [(a \wedge \neg c) \wedge \neg b] && \text{comut } 2\times, \text{ assoc } 2\times \\
\Leftrightarrow & [\neg a \wedge \neg c] \vee [(a \wedge \neg c) \wedge (b \vee \neg b)] && \text{distributiva } (a \wedge \neg c) \\
\Leftrightarrow & [\neg a \wedge \neg c] \vee [a \wedge \neg c] && \text{compl } \vee, \text{ ident } \wedge \\
\Leftrightarrow & (\neg a \vee a) \wedge \neg c && \text{distributiva } \neg c \\
\Leftrightarrow & \neg c && \text{compl } \vee, \text{ ident } \wedge
\end{aligned}$$

A expressão contém ocorrências de a e $\neg a$, e de b e $\neg b$. A estratégia para simplificação é agrupar os literais com seus complementos e então eliminá-los. Veremos adiante o Teorema 3.21, que permite reduzir substancialmente o número de linhas desta simplificação. $\triangleleft$

Exemplo 3.12 Simplifiquemos a expressão abaixo usando as propriedades das operações sobre os elementos de $\mathbb{B}$.

$$\begin{aligned}
& \neg(a \Rightarrow b) \vee \neg(\neg b \Rightarrow \neg a) && \text{definição } \Rightarrow \\
\Leftrightarrow & \neg(\neg a \vee b) \vee \neg(\neg(\neg b) \vee \neg a) && \text{involução} \\
\Leftrightarrow & \neg(\neg a \vee b) \vee \neg(b \vee \neg a) && \text{DeMorgan } 2\times \\
\Leftrightarrow & [\neg(\neg a) \wedge \neg b] \vee [\neg b \wedge \neg(\neg a)] && \text{involução } 2\times \\
\Leftrightarrow & a \wedge \neg b \vee \neg b \wedge a && \text{comutativa } \wedge \\
\Leftrightarrow & a \wedge \neg b \vee a \wedge \neg b && \text{idempotência } \vee \\
\Leftrightarrow & a \wedge \neg b
\end{aligned}$$

A expansão da definição de $\Rightarrow$ nos permite aplicar as propriedades do complemento e reduzir o número de literais e de operadores. $\triangleleft$

Além da simplificação de expressões, podemos usar a manipulação algébrica para provar propriedades dos operadores sobre bits.

Exemplo 3.13 Vejamos como provar a propriedade chamada de *contra-positiva*

$$(a \Rightarrow b) \Leftrightarrow (\bar{b} \Rightarrow \bar{a}). \quad (3.3)$$

A prova é construída da mesma forma que numa simplificação e cada passo da cadeia de transformações é justificado por uma propriedade válida.

$$\begin{aligned} a \Rightarrow b &\Leftrightarrow (\bar{a} \vee b) && \text{definição de } \Rightarrow \\ &\Leftrightarrow \overline{\bar{a} \vee b} && \text{involução} \\ &\Leftrightarrow (\bar{a} \vee \bar{b}) && \text{DeMorgan} \\ &\Leftrightarrow \overline{(\bar{a} \wedge \bar{b})} && \text{involução} \\ &\Leftrightarrow (a \wedge b) && \text{DeMorgan} \\ &\Leftrightarrow \bar{a} \vee \bar{b} && \text{comutativa} \\ &\Leftrightarrow \bar{b} \vee \bar{a} && \text{definição de } \Rightarrow \\ &\Leftrightarrow \bar{b} \Rightarrow \bar{a} \end{aligned}$$

A estratégia de prova é, a partir da *forma* do lado esquerdo (LE) da equivalência, tentar aplicar as propriedades que transformem a expressão do LE para que ela fique com a *forma* da expressão do lado direito (LD). Como o LD da equivalência contém os complementos dos literais do lado esquerdo, a aplicação da involução e DeMorgan nos ajudam a comprovar a equivalência. $\triangleleft$

Exemplo 3.14 O Teorema de DeMorgan é excepcionalmente útil e foi definido para duas variáveis na Página 32. Vejamos como provar a sua validade para três variáveis.

$$\begin{aligned} \overline{a \wedge b \wedge c} &\Leftrightarrow \overline{a \wedge k} && \text{substitui } (b \wedge c) \text{ por } k \\ &\Leftrightarrow \bar{a} \vee \bar{k} && \text{DeMorgan} \\ &\Leftrightarrow \bar{a} \vee \overline{(b \wedge c)} && \text{substitui } k \text{ por } (b \wedge c) \\ &\Leftrightarrow \bar{a} \vee \overline{(b \wedge c)} && \text{DeMorgan em } \overline{(b \wedge c)} \\ &\Leftrightarrow \bar{a} \vee (\bar{b} \vee \bar{c}) && \text{associatividade} \\ &\Leftrightarrow \bar{a} \vee \bar{b} \vee \bar{c} \end{aligned}$$

A propriedade do fechamento nos garante que o resultado da conjunção de duas variáveis em $\mathbb{B}$ também pertence a $\mathbb{B}$. O primeiro passo da prova substitui a conjunção $b \wedge c$ pela variável $k \in \mathbb{B}$, para então aplicar DeMorgan nas duas variáveis. O terceiro passo desfaz a substituição e mais uma aplicação de DeMorgan nos leva ao resultado que desejávamos provar. A dualidade nos garante que

$$\overline{a \vee b \vee c} \Leftrightarrow \bar{a} \wedge \bar{b} \wedge \bar{c}$$

é também uma propriedade. Este teorema terá sua utilidade comprovada no Capítulo 5. $\triangleleft$

3.3 Tipos e Especificações

A ideia de *tipos* não é novidade para aqueles que tiveram a sorte de encontrar uma competente professora – ou professor – de física no Ensino Médio. Estes os terão ensinado a *nunca* misturar voltagem com temperatura, nem comprimento com velocidade. A definição popular para tipos é “laranjas com laranjas, maçãs com maçãs”.

A maioria das especificações e descrições de circuitos neste texto indicam o *tipo* dos operandos e operadores. Os tipos básicos são definidos na Tabela 3.9. O uso de tipos para os sinais é útil porque obriga o projetista a atentar para as interfaces entre os vários componentes de um circuito. Quando um sinal de tipo $\mathcal{T}_1$ é ligado a uma entrada com tipo $\mathcal{T}_2$, com $\mathcal{T}_1 \neq \mathcal{T}_2$, é muito provável que haja um erro de projeto ou de concepção. A linguagem de descrição de circuitos VHDL, introduzida no Capítulo 10, é dita *fortemente tipada* porque o compilador VHDL verifica os tipos de todos os sinais e indica como um erro a ligação entre sinais de tipos distintos e/ou incompatíveis. Em VHDL, dá-se o nome de *sinal* a uma variável lógica ou a uma tupla de variáveis lógicas. Usaremos este nome no que se segue.

Tabela 3.9: Tipos Básicos.

tipo	descrição
$\mathbb{B}$	Conjunto dos Bits
$\mathbb{N}$	Conjunto dos Números Naturais
$\mathbb{Z}$	Conjunto dos Números Inteiros
$\mathbb{R}$	Conjunto dos Números Reais
$\mathbb{B}^n$	Tupla, ou Vetor de Bits com largura n
$(\mathbb{B} \times \cdots \times \mathbb{B})$	Tupla de valores do tipo $\mathbb{B}$
$\Delta \mapsto \Theta$	Função com domínio em Δ e imagem em Θ

A Equação 3.4 contém a definição completa do operador condicional, que é a versão compacta e infixada do comando **if** c **then** a **else** b com operandos a , b e c de tipo $\mathbb{B}$.

$$\begin{aligned}
 a, b, c &: \mathbb{B} \\
 \triangleleft \triangleright &: \mathbb{B} \times (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B} \\
 a \triangleleft c \triangleright b &\equiv (c \wedge a) \vee (\neg c \wedge b)
 \end{aligned} \tag{3.4}$$

Esta equação pode ser usada para *especificar* o comportamento de um circuito, ou para *documentar* uma implementação. Nos dois casos, o comportamento é definido com precisão e sem ambiguidade, e pode ser usado como um contrato entre quem especifica o circuito e quem o implementa, ou entre quem implementa e quem usa o circuito.

O tipo dos três operandos a, b, c é declarado, explicitando que estes são do tipo *bit*.

$$a, b, c : \mathbb{B}$$

O operador condicional $\triangleleft \triangleright$ tem tipo

$$\triangleleft \triangleright : \mathbb{B} \times (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B}$$

que é lido como “o condicional é uma *função* com dois argumentos, o primeiro do tipo $\mathbb{B}$, e o segundo é um *par de operandos* $(\mathbb{B} \times \mathbb{B})$, e produz um *resultado* do tipo $\mathbb{B}$ ”. Finalmente, a expressão

$$a \triangleleft c \triangleright b \equiv (c \wedge a) \vee (\neg c \wedge b)$$

define as relações válidas entre as variáveis declaradas anteriormente. Veremos adiante que o circuito que implementa o operador condicional é tão útil quanto o comando `if then else`.

3.4 Tuplas de Bits Que Representam Números

Uma tupla com k bits pode representar 2^k padrões distintos. Por exemplo, uma tupla com $k = 1$ representa o conjunto de padrões $\{\langle 0 \rangle, \langle 1 \rangle\}$; uma tupla com $k = 2$ representa o conjunto $\{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle\}$; para $k = 3$, o conjunto é

$$\{\langle 0, 0, 0 \rangle, \langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 0, 1, 1 \rangle, \langle 1, 0, 0 \rangle, \langle 1, 0, 1 \rangle, \langle 1, 1, 0 \rangle, \langle 1, 1, 1 \rangle\}.$$

As potências de 2 são números deveras populares em Computação. A n -ésima potência de 2 é definida por

$$2^n = \overbrace{2 \times 2 \times \cdots \times 2}^{n \text{ vezes}}.$$

Por exemplo, $2^2 = 4 = 2 \times 2$, enquanto que $2^5 = 32 = 2 \times 2 \times 2 \times 2 \times 2$.

Tuplas de bits podem ser usadas para representar números na base 2. O número N é representado pelo vetor de bits $\langle n_{k-1}, n_{k-2}, \dots, n_1, n_0 \rangle$ e seu valor é definido, em notação posicional, pela soma de potências de 2 da Equação 3.5.

$$N = \sum_{i=0}^{k-1} n_i \times 2^i \quad (3.5)$$

Se o j -ésimo elemento da tupla é zero, a parcela correspondente da soma é zero. O número cinco é representado por $101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1$. O subscrito em ‘ 101_2 ’ indica que o número está representado na base 2 e não na base 10.

A operação inversa da exponenciação é o logaritmo, e estamos interessados no logaritmo da base 2. Para um número N , o *logaritmo de N na base 2* é

$$2^{\log N} = N.$$

O logaritmo de um número indica a largura mínima da tupla de bits necessária para representar aquele número. Se $M = 2^m$, então uma tupla de m bits pode representar todos os números entre 0 e $2^m - 1$. Considere $M = 16$; temos que $\log 16 = 4$ porque $16 = 2 \times 2 \times 2 \times 2$, e na base 2, $16 = 1.0000_2$. É necessária uma tupla de quatro bits para representar todos os dezesseis padrões de bits entre $0000_2 = 0$ e $1111_2 = 15$. Para facilitar a leitura, o ponto agrupador de milhares é usado para agrupar quartetos de bits, e por isso escrevemos 1.0000_2 ao invés de 10.000_2 .

No que se segue estão especificadas formalmente a função que converte uma tupla de bits para o número natural que aquela tupla representa, e o circuito que efetua a divisão inteira de dois números inteiros. A especificação não contém *nenhuma* indicação de como o circuito deva ser implementado, e esta “liberdade de implementação” é uma das liberdades fundamentais da, ou do, projetista.

Exemplo 3.15 A Equação 3.6 define a função que converte um vetor de bits para o número natural correspondente. A função *num* recebe como argumento um vetor V de n bits, de tipo $\mathbb{B}^n$, e retorna um inteiro N , que é o valor representado pelo vetor V . O i -ésimo elemento de V é denotado por v_i .

$$\begin{aligned} V &: \mathbb{B}^n \\ N &: \mathbb{N} \\ num &: \mathbb{B}^n \mapsto \mathbb{N} \\ num(V) = N &\equiv N = \sum_{i=0}^{n-1} v_i \cdot 2^i \end{aligned} \tag{3.6}$$

O valor representado pelo vetor de bits é a soma dos produtos do valor do bit v_i pela i -ésima potência de dois, que é a formulação da representação posicional para a base-2. $\triangleleft$

Exemplo 3.16 A Equação 3.7 especifica um circuito que efetua a divisão inteira de dois números representados em 8 bits. O circuito tem três entradas e três saídas: o sinal *inicia* carrega os valores do dividendo e do divisor nas entradas *ddndo* e *dvsor*, respectivamente, e dispara a computação do resultado; a saída *pronto* indica que as saídas *quoc* e *resto* contêm os valores do quociente e do resto da divisão inteira.

$$\begin{aligned} inicia, pronto &: \mathbb{B} \\ ddndo, dvsor &: \mathbb{B}^8 \\ quoc, resto &: \mathbb{B}^8 \\ div-8 &: \mathbb{B} \times (\mathbb{B}^8 \times \mathbb{B}^8) \mapsto (\mathbb{B}^8 \times \mathbb{B}^8) \times \mathbb{B} \\ div-8(inicia, ddndo, dvsor, quoc, resto, pronto) &\equiv \\ &inicia \Rightarrow [pronto \Rightarrow \\ &\quad (num(ddndo) = (num(dvsor) \cdot num(quoc)) + num(resto))] \end{aligned} \tag{3.7}$$

Note que a especificação não determina a maneira de implementar o circuito mas somente define o resultado a ser produzido em função das entradas que lhe sejam apresentadas. Esta forma de especificar o comportamento desejado para um circuito, indicando-se apenas sua funcionalidade, é empregada na *modelagem funcional* de circuitos com VHDL. O material deste e dos próximos capítulos contém vários exemplos de especificações funcionais e de implementações que as satisfazem. $\triangleleft$

Especificação com Pré-condições O uso de implicação lógica na Equação 3.7 merece uma explicação. Intuitivamente, a expressão $a \Rightarrow b$ significa que “se $a = 1$ então $b = 1$ ”. A definição do operador $\Rightarrow$ é

$$a \Rightarrow b \equiv \neg a \vee b$$

e contra intuitivamente significa também que, “se $a = 0$ então não se pode afirmar nada sobre o valor de b ”. Esta interpretação é usada nas especificações para garantir que as condições necessárias para a correta operação do circuito são válidas: se as entradas estão dentro do esperado, então o circuito operará como o desejado.

Uma especificação é um contrato entre vendedor e comprador do circuito. Como parte do contrato, consta que se as entradas não são válidas, então nada se pode dizer sobre o comportamento do circuito. No caso do circuito divisor, se o circuito externo ao divisor não disponibilizar os valores de dividendo e divisor, e então ativar o sinal *inicia*, não se pode esperar uma resposta correta. Ademais, no contrato consta que, se as entradas estão corretas, então o resultado está disponível somente se o sinal *pronto* estiver ativo.

3.5 Tabela Verdade e Soma de Produtos

Expressões complexas podem ser construídas pela combinação de expressões simples, e o valor da expressão complexa pode ser derivado dos valores das expressões simples. Uma expressão pode ser avaliada através de uma tabela verdade, na qual são listadas todas as combinações dos valores das variáveis da expressão.

A tabela verdade de uma expressão com n variáveis contém 2^n linhas, uma linha para cada uma das combinações das variáveis. Considere a função

$$p = q \wedge (r \vee s).$$

Sua tabela verdade (TV) tem 8 linhas para cobrir todas as 2^3 combinações para os valores de q , r e s , e é mostrada na Tabela 3.10. O *índice* das linhas da tabela, que normalmente é omitido, é o número representado pela atribuição de valores às variáveis em cada linha na tabela: $\text{num}(\langle q, r, s \rangle) = \text{índice}$. Os operadores foram propositalmente omitidos e nos interessa mostrar somente o valor de p para todas as combinações das entradas.

Tabela 3.10: Tabela verdade da função $p = q \wedge (r \vee s)$.

índice	q	r	s	p
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

Qualquer função em $\mathbb{B}^n$ pode ser representada por uma *soma de produtos*. A *soma* é uma disjunção de termos, e cada termo corresponde a uma linha da tabela verdade cujo valor seja 1. Cada *produto* é a conjunção de variáveis, ou de seus complementos – a própria variável se na linha da tabela seu valor é 1, ou seu complemento, se 0. A soma de produtos de p , segundo a Tabela 3.10 é

$$p = (q \wedge \bar{r} \wedge s) \vee (q \wedge r \wedge \bar{s}) \vee (q \wedge r \wedge s),$$

porque os termos de p que são 1 correspondem às linhas 5 ($(q \wedge \bar{r} \wedge s) \leftrightarrow 101$), 6 ($(q \wedge r \wedge \bar{s}) \leftrightarrow 110$) e 7 ($(q \wedge r \wedge s) \leftrightarrow 111$).

Uma soma de produtos *completa* é aquela em que todas as variáveis da função, ou seus complementos, aparecem em todos os termos da soma. Uma soma de produtos completa é representada por uma *disjunção de conjunções*, com uma conjunção para cada linha da TV na qual a função seja 1. Cada conjunção é chamada de *mintermo* e cada mintermo deve conter todas as variáveis da função ou seus complementos.

Um mintermo é a conjunção de todas as variáveis da função, ou seus complementos, e o i -ésimo mintermo é representado por m_i . Uma função $\mathcal{F}$ pode ser representada pela soma dos mintermos para os quais $\mathcal{F} = 1$. A representação com mintermos de $p(q, r, s)$ é

$$p(q, r, s) = m_5 \vee m_6 \vee m_7.$$

Dependendo da função, pode ser mais interessante empregarmos uma representação alternativa, que é um *produto de somas*, ou uma *conjunção de disjunções*, porque esta representação pode ser menor do que a soma de produtos que lhe corresponde. A disjunção de *maxterms* deve conter uma disjunção para cada linha da TV na qual a função é 0.

O *maxtermo* M_i é o complemento do mintermo m_i , no qual todas as variáveis são os respectivos complementos daquelas do mintermo. Por exemplo, ao mintermo $m_3 = \bar{a} \wedge b \wedge c$ corresponde o maxtermo $M_3 = a \vee \bar{b} \vee \bar{c}$. Assim como para os mintermos, cada maxtermo deve ser completo e conter todas as variáveis da função ou seus complementos. A representação $\overline{p(q, r, s)}$ com maxtermos de é a conjunção dos maxtermos que fazem $p = 0$

$$\overline{p(q, r, s)} = M_0 \wedge M_1 \wedge M_2 \wedge M_3 \wedge M_4.$$

A Tabela 3.11 mostra todos os mintermos e maxtermos para uma função de três variáveis.

Tabela 3.11: Mintermos e maxtermos para uma função de três variáveis.

índice	a	b	c	mintermos		maxtermos	
				<i>produto</i>	<i>repr.</i>	<i>soma</i>	<i>repr.</i>
0	0	0	0	$\bar{a} \wedge \bar{b} \wedge \bar{c}$	m_0	$a \vee b \vee c$	M_0
1	0	0	1	$\bar{a} \wedge \bar{b} \wedge c$	m_1	$a \vee b \vee \bar{c}$	M_1
2	0	1	0	$\bar{a} \wedge b \wedge \bar{c}$	m_2	$a \vee \bar{b} \vee c$	M_2
3	0	1	1	$\bar{a} \wedge b \wedge c$	m_3	$a \vee \bar{b} \vee \bar{c}$	M_3
4	1	0	0	$a \wedge \bar{b} \wedge \bar{c}$	m_4	$\bar{a} \vee b \vee c$	M_4
5	1	0	1	$a \wedge \bar{b} \wedge c$	m_5	$\bar{a} \vee b \vee \bar{c}$	M_5
6	1	1	0	$a \wedge b \wedge \bar{c}$	m_6	$\bar{a} \vee \bar{b} \vee c$	M_6
7	1	1	1	$a \wedge b \wedge c$	m_7	$\bar{a} \vee \bar{b} \vee \bar{c}$	M_7

Exemplo 3.17 Vejamos como obter uma representação em maxtermos para a função

$$\bar{f} = (a \vee \bar{b} \vee c) \wedge (\bar{b} \vee c) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (a \vee c).$$

Como a expressão para $\overline{f(a, b, c)}$ não é completa, devemos iniciar pela obtenção da forma completa. O termo $(\bar{b} \vee c)$ deve ser completado pela inclusão da variável a , e o termo $(a \vee c)$ deve ser completado com a variável b :

$$\begin{aligned} \bar{b} \vee c &\Leftrightarrow (\bar{b} \vee c) \wedge (a \vee \bar{a}) && a \vee a = 1, 1 \wedge x = x \\ &\Leftrightarrow (a \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee c) && \text{distributiva, comutativa} \\ \\ a \vee c &\Leftrightarrow (a \vee c) \wedge (b \vee \bar{b}) && a \vee a = 1, 1 \wedge x = x \\ &\Leftrightarrow (a \vee b \vee c) \wedge (a \vee \bar{b} \vee c) && \text{distributiva, comutativa} \end{aligned}$$

Ao substituir aqueles termos, obtemos a representação completa para $\overline{f(a, b, c)}$,

$$\bar{f} = (a \vee \bar{b} \vee c) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (a \vee b \vee c) \wedge (a \vee \bar{b} \vee c).$$

Consultando a Tabela 3.11, obtemos a representação de $\overline{f(a, b, c)}$ com maxtermos

$$\begin{aligned} \bar{f} &= M_2 \wedge M_2 \wedge M_6 \wedge M_3 \wedge M_0 \wedge M_2 \\ &= M_0 \wedge M_2 \wedge M_3 \wedge M_6. \end{aligned}$$

A tabela verdade de $f(a, b, c)$ é mostrada na Tabela 3.12. Os maxtermos correspondem às linhas da TV nas quais $f(a, b, c) = 0$. A TV está construída de forma a minimizar o trabalho de preenchimento; quando um termo é completamente determinado por uma variável ($0 \wedge x$ ou $1 \vee y$), as colunas das outras variáveis não estão preenchidas por serem redundantes. Somente uma coluna das conjunções ou disjunções com 2, 3 ou 4 variáveis é preenchida.

Tabela 3.12: Tabela verdade para $f(a, b, c)$.

a	b	c	$a \vee \bar{b} \vee c$	$\wedge$	$\bar{b} \vee c$	$\wedge$	$a \vee \bar{b} \vee \bar{c}$	$\wedge$	$a \vee c$		
0	0	0	1	1		1	1	0	1	1	M_0
0	0	1	1	1		1	1	1	1	1	m_1
0	1	0	0		0	0		0	1		M_2
0	1	1	1		1	1		0	0		M_3
1	0	0	1	1		1	1	1	1	1	m_4
1	0	1	1	1		1	1	1	1	1	m_5
1	1	0	1	1		0	0	0	1	1	M_6
1	1	1	1	1		1	1	1	1	1	m_7
<i>passo</i>			1	3	2	1			2	3	1

Consultando novamente a Tabela 3.11, obtemos $f(a, b, c)$ representada com mintermos

$$f = m_1 \vee m_4 \vee m_5 \vee m_7.$$

A representação com mintermos é o complemento daquela com maxtermos porque a função com maxtermos representa os termos em que $f = 0$, enquanto a representação com mintermos representa os termos com $f = 1$.

Neste exemplo, as duas representações, com maxtermos e com mintermos, têm complexidade similar, cada uma com quatro termos de três variáveis. A verificação de qual forma simplificada é mais simples fica como um exercício. $\triangleleft$

A soma (disjunção) dos mintermos é uma representação compacta para a tabela verdade de qualquer função $\mathcal{G}$. Empregamos três variáveis para simplificar a representação.

$$\mathcal{G}(a, b, c) = m_i \vee m_j \vee \dots \vee m_k, \quad i, j, k \in [0, 7]. \quad (3.8)$$

A representação em termos de produto (conjunção) de maxtermos é

$$\overline{\mathcal{G}(a, b, c)} = M_l \wedge M_m \wedge \dots \wedge M_n, \quad l, n, m \in [0, 7]. \quad (3.9)$$

Podemos também fazer uso de uma abreviação similar ao somatório e ao produtório para representar funções nas formas *soma de produtos* e o *produto de somas*. Empregando os símbolos $\bigvee$ e $\bigwedge$ para a disjunção e conjunção prefixadas, podemos representar a soma de produtos como

$$\mathcal{G}(a, b, c) = \bigvee(i, j, \dots, k), \quad i, j, k \in [0, 7]. \quad (3.10)$$

e o produto de somas como

$$\overline{\mathcal{G}(a, b, c)} = \bigwedge(l, m, \dots, n), \quad l, m, n \in [0, 7]. \quad (3.11)$$

Estas representações para funções em termos de soma de produtos completas e produto de somas completos são chamadas de *formas canônicas* das funções.

Exemplo 3.18 Considere a função h , representada por

$$h(a, b, c) = \bigvee(1, 3, 6, 7) = m_1 \vee m_3 \vee m_6 \vee m_7.$$

Seu complemento pode ser facilmente obtido com a Tabela 3.11

$$\overline{h(a, b, c)} = \bigwedge(0, 2, 4, 5) = M_0 \wedge M_2 \wedge M_4 \wedge M_5.$$

A Tabela 3.11 mostra que o maxtermo M_i é o complemento do mintermo m_i , e assim obtemos facilmente o produto de somas que representa $\overline{g(a, b, c)}$. $\triangleleft$

Considere uma função com k variáveis e m mintermos. Para converter a representação de soma de produtos para produto de somas basta trocar os operadores $\bigvee$ para $\bigwedge$ e listar os $2^k - m$ maxtermos que não estão listados na soma de produtos. A conversão no sentido inverso é similar.

Exemplo 3.19 Para exemplificar a conversão, as funções, $s(a, b, c)$ e $v(a, b, c)$, são representadas nas suas formas canônicas.

$$\begin{aligned} s(a, b, c) &= \bigvee(1, 2, 4, 7) = m_1 \vee m_2 \vee m_4 \vee m_7 \\ \overline{s(a, b, c)} &= \bigwedge(0, 3, 5, 6) = M_0 \wedge M_3 \wedge M_5 \wedge M_6, \\ v(a, b, c) &= \bigvee(3, 7) = m_3 \vee m_7 \\ \overline{v(a, b, c)} &= \bigwedge(0, 1, 2, 4, 5, 6) = M_0 \wedge M_1 \wedge M_2 \wedge M_4 \wedge M_5 \wedge M_6. \end{aligned}$$

Esta conversão é um tanto mais simples do que sucessivas aplicações de DeMorgan. A representação em mintermos de $v(a, b, c)$ é algo mais econômica do que com maxtermos. $\triangleleft$

Vejamos três exemplos de funções de tipo $\mathbb{B}^3 \mapsto \mathbb{B}$, definidas na Tabela 3.13.

Exemplo 3.20 A função $t(a, b, c)$ indica se a tripla de bits representa um número que é múltiplo de três. A função pode ser representada na forma de soma de produtos, ou de forma mais concisa como uma conjunção de mintermos:

$$t = (\bar{a} \wedge b \wedge c) \vee (a \wedge b \wedge \bar{c}) = m_3 \vee m_6.$$

Da tabela verdade vemos que só duas linhas fazem $t = 1$, as linhas de índice 3 e 6, que são aquelas das conjunções $(\bar{a} \wedge b \wedge c)$ e $(a \wedge b \wedge \bar{c})$. Isso porque a função $t(a, b, c)$ é 1 somente quando os argumentos são tais que fazem aquelas conjunções iguais a 1, a saber $\langle 0, 1, 1 \rangle$ e $\langle 1, 1, 0 \rangle$. $\triangleleft$

Tabela 3.13: Tabelas verdade para múltiplos de três, paridade par e ímpar.

índice	a	b	c	t	j	k	l	i	r	s	t	p
0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	1	1	0	0	1	0
2	0	1	0	0	0	1	0	1	0	1	0	0
3	0	1	1	1	0	1	1	0	0	1	1	1
4	1	0	0	0	1	0	0	1	1	0	0	0
5	1	0	1	0	1	0	1	0	1	0	1	1
6	1	1	0	1	1	1	0	0	1	1	0	1
7	1	1	1	0	1	1	1	1	1	1	1	0

Exemplo 3.21 O segundo exemplo de função $\mathbb{B}^3 \mapsto \mathbb{B}$ é a função que indica se a contagem de bits em 1 na tripla é ímpar. Esta função é tão útil que recebe nome próprio: *paridade ímpar*. A função $i(j, k, l)$ é 1 quando a quantidade de 1s na tripla de argumentos é ímpar. $i(j, k, l)$ é representada em forma de soma de produtos por

$$i = (\bar{j} \wedge \bar{k} \wedge l) \vee (\bar{j} \wedge k \wedge \bar{l}) \vee (j \wedge \bar{k} \wedge \bar{l}) \vee (j \wedge k \wedge l) = m_1 \vee m_2 \vee m_4 \vee m_7$$

Na Tabela 3.13 a função $i()$ está dividida entre as metades em que $j = 0$ e $j = 1$. Na metade de cima, temos que $i = k \oplus l$ e na metade de baixo temos que $i = \overline{k \oplus l}$. Podemos usar esta informação para simplificar $i(j, k, l)$:

$$\begin{aligned}
i &\Leftrightarrow \bar{j} \wedge (k \oplus l) \vee j \wedge \overline{(k \oplus l)} && \text{substitui } (k \oplus l) \text{ por } y \\
&\Leftrightarrow \bar{j} \wedge y \vee j \wedge \bar{y} && \text{definição de } \oplus \\
&\Leftrightarrow j \oplus y && \text{substitui } y \text{ por } (k \oplus l) \\
&\Leftrightarrow j \oplus (k \oplus l) && \text{Exercício 3.7} \\
&\Leftrightarrow j \oplus k \oplus l
\end{aligned}$$

A paridade ímpar de n bits é seu ou-exclusivo – veja o Exercício 3.7

◁

Exemplo 3.22 O terceiro exemplo de função $\mathbb{B}^3 \mapsto \mathbb{B}$ é a função que indica se a contagem de bits em 1 na tripla é par. Esta função é chamada de *paridade par*. A função $p(r, s, t)$ é 1 quando a quantidade de 1s na tripla de argumentos é par. $p(r, s, t)$ é representada em forma de soma de produtos por

$$p = (\bar{r} \wedge \bar{s} \wedge \bar{t}) \vee (\bar{r} \wedge s \wedge t) \vee (r \wedge \bar{s} \wedge t) \vee (r \wedge s \wedge \bar{t}) = r \oplus s \oplus \bar{t}$$

A forma com $\oplus$ é obtida se usarmos o raciocínio do Exemplo 3.21 mais o Exemplo 3.4.

◁

3.6 Mapas de Karnaugh

Um teorema útil no tratamento de expressões com operadores e elementos de $\mathbb{B}$ é o Teorema 3.21, conhecido como o *Teorema da Simplificação*. Este teorema permite a eliminação de variáveis em expressões. As equivalências (a) permitem a construção de Mapas de Karnaugh, que são apresentados no que se segue. O Teorema da Simplificação para duas variáveis pode ser provado, facilmente, com três tabelas verdade e Dualidade. O Teorema 3.21 (a) também é chamado de *Teorema da Absorção*.

Definição 3.21 (Teorema da Simplificação)

$$\begin{aligned}
(x \wedge y) \vee (x \wedge \neg y) &= x & (x \vee y) \wedge (x \vee \neg y) &= x & (a) \\
x \vee (x \wedge y) &= x & x \wedge (x \vee y) &= x & (b) \\
(x \vee \neg y) \wedge y &= x \wedge y & (x \wedge \neg y) \vee y &= x \vee y & (c)
\end{aligned}$$

Exemplo 3.23 Simplifiquemos a expressão abaixo usando as propriedades dos operadores sobre $\mathbb{B}$, mais o Teorema da Simplificação.

$$\begin{aligned}
& [w \wedge (x \vee y)] \vee (\neg w \wedge \neg x) \vee y && \text{distributiva} \\
\Leftrightarrow & (w \wedge x) \vee (w \wedge y) \vee (\neg w \wedge \neg x) \vee y && \text{comutativa} \\
\Leftrightarrow & (w \wedge x) \vee (\neg w \wedge \neg x) \vee [(w \wedge y) \vee y] && \text{simplificação (b)} \\
\Leftrightarrow & [(w \wedge x) \vee (\neg w \wedge \neg x)] \vee y && \text{definição } \oplus \\
\Leftrightarrow & (w \oplus \neg x) \vee y
\end{aligned}$$

A primeira regra expande a expressão original para permitir a simplificação. A segunda aproxima os termos em w e x , e a terceira elimina a subexpressão $(w \wedge y)$. A quarta regra reescreve os dois termos com a definição do ou-exclusivo. $\triangleleft$

O Teorema da Simplificação é empregado implicitamente na construção de *Mapas de Karnaugh* para permitir a simplificação de expressões pelo agrupamento de células. A Figura 3.2 mostra o mapa de Karnaugh para a expressão $p = q \wedge (r \vee s)$. Cada célula no mapa é identificada por um número no canto superior direito, que é justamente o índice na tabela verdade correspondente à célula. Os pares de números acima do mapa indicam os valores que correspondem às células abaixo do par, para as variáveis r e s . As colunas em que $r = 1$ são aquelas ‘cobertas’ pela linha horizontal, acima do mapa, marcada com r , e as colunas com $s = 1$ são ‘cobertas’ pela linha marcada com s . A linha vertical marcada com q indica a linha com $q = 1$.

índice	p	q	r	s
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	1	1	0	1
6	1	1	1	0
7	1	1	1	1

		r			
		s			
	p	00	01	11	10
q	0	0	0	0	0
	1	0	1	1	1

Figura 3.2: Mapa de Karnaugh para $p = q \wedge (r \vee s)$.

A i -ésima célula do mapa é preenchida com o valor para p da linha da tabela verdade de índice i . Assim, a célula de número 0 corresponde à primeira linha da tabela, com $\langle q, r, s \rangle = 000_2$, enquanto que a célula de número 5 corresponde à linha com $\langle q, r, s \rangle = 101_2$.

A disposição das células é tal que a representação em binário dos índices de duas células vizinhas difere exatamente em uma posição. Considere duas células vizinhas i e j . O mapa é organizado para que os índices destas células sejam tais que somente um dos bits na representação binária dos dois índices difere, o que nos permite usar a primeira versão do Teorema da Simplificação, e eliminar a variável que muda. Isso permite agregar e simplificar os mintermos m_i e m_j na soma de produtos.

O mapa é desenhado como um retângulo, mas seu formato é o de um toroide² porque as células dos lados menores são adjacentes – as células 0 e 2 são vizinhas – assim como as células dos lados maiores, embora isso seja mais fácil de perceber num mapa para quatro variáveis.

São possíveis duas simplificações no mapa de $p = q \wedge (r \vee s)$, e estes são indicados pelas elipses na Figura 3.3. A elipse da esquerda representa o agrupamento $q \wedge s$ e a elipse da direita representa o agrupamento $q \wedge r$.

p	$r\ s$			
	00	01	11	10
q	0	1	3	2
0	0	0	0	0
1	0	1	1	1

Figura 3.3: Mapa de Karnaugh com os agrupamentos de $p = q \wedge (r \vee s)$.

A elipse da esquerda elimina r porque as células 5 e 7 representam os mintermos m_5 (índice $\langle q, r, s \rangle = 101$) com $r = 0$, e m_7 (índice $\langle q, r, s \rangle = 111$) com $r = 1$. A elipse da direita elimina s porque as células 6 e 7 representam os mintermos m_6 (índice $\langle q, r, s \rangle = 110$) com $s = 0$ e m_7 com $s = 1$.

A expressão, na forma de *soma de produtos*, resultante da simplificação é

$$p' = (q \wedge s) \vee (q \wedge r),$$

que é uma expressão ‘maior’ que aquela de p . A expressão original pode ser obtida de p' pela aplicação de distributividade.

Exemplo 3.24 Vejamos como preencher o Mapa de Karnaugh para $g(a, b, c) = \vee(0, 1, 2, 3, 5, 7)$, e então deduzir uma simplificação para g .

O preenchimento é direto: as células com os números dos mintermos são preenchidas com 1, e as demais com 0.

O mapa na Figura 3.4 mostra os dois agrupamentos que resultam na máxima simplificação. O agrupamento da linha de cima simplifica as variáveis b e c porque as células adjacentes de ambas trocam de valor na linha. O termo simplificado corresponde a $\bar{a}$.

O agrupamento no centro do mapa corresponde a c porque há troca de valor tanto nas células em que $a = 1$, quanto nas em que $b = 1$. A função simplificada é portanto $g' = \bar{a} \vee c$.

Outra possibilidade seria aproveitar que o agrupamento do centro cobre os mintermos m_1, m_3, m_5 e m_7 , e agrupar os mintermos m_0 e m_2 , que são vizinhos “por fora” do mapa. Neste caso, a função (menos) simplificada é $g'' = (\bar{a} \wedge \bar{c}) \vee c$. $\triangleleft$

Exemplo 3.25 A Figura 3.5 mostra a tabela verdade e o Mapa de Karnaugh para uma função de quatro variáveis, $u(x, y, z, w) = \vee(1, 4, 6, 9, 12, 14)$. Neste caso, os dois agrupamentos possíveis são nas bordas do mapa. As células 1 e 9 são adjacentes porque x é a única variável que muda entre as duas células, e este agrupamento corresponde ao termo $\bar{y} \wedge \bar{z} \wedge w$. As células 4 e 6 são adjacentes porque z muda, assim como as células 12 e 14. Este agrupamento resulta no termo $y \wedge \bar{w}$. A função simplificada é portanto $u(x, y, z, w) = \bar{y} \wedge \bar{z} \wedge w \vee y \wedge \bar{w}$. $\triangleleft$

²Mesmo formato que um pneu de automóvel.

g	bc			
	00	01	11	10
a	0	1	3	2
	1	1	1	1
	4	5	7	6
1	0	1	1	0

Figura 3.4: Mapa de Karnaugh para $g(a, b, c) = \bigvee(0, 1, 2, 3, 5, 7)$.

u	zw			
	00	01	11	10
xy	0	1	3	2
	0	1	0	0
	4	5	7	6
01	1	0	0	1
	12	13	15	14
11	1	0	0	1
	8	9	11	10
10	0	1	0	0

Figura 3.5: Mapa de Karnaugh para $u(x, y, z, w) = \bigvee(1, 4, 6, 9, 12, 14)$.

Exemplo 3.26 A Figura 3.6 mostra a tabela verdade e o Mapa de Karnaugh para uma função de quatro variáveis. Para esta função em particular, não há como agrupar células para obter alguma simplificação de t . As células preenchidas com 1 formam diagonais no mapa, e isso indica que a função é o ou-exclusivo das entradas – a tabela verdade mostra que a função t é 1 somente nas linhas com um número ímpar de 1's nos valores de $\langle x, y, z, w \rangle$, e portanto $t = x \oplus y \oplus z \oplus w$. $\triangleleft$

índice	t	x	y	z	w
0	0	0	0	0	0
1	1	0	0	0	1
2	1	0	0	1	0
3	0	0	0	1	1
4	1	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	1	0	1	1	1
8	1	1	0	0	0
9	0	1	0	0	1
10	0	1	0	1	0
11	1	1	0	1	1
12	0	1	1	0	0
13	1	1	1	0	1
14	1	1	1	1	0
15	0	1	1	1	1

t	zw			
	00	01	11	10
xy	0	1	3	2
	0	1	0	1
	4	5	7	6
01	1	0	1	0
	12	13	15	14
11	0	1	0	1
	8	9	11	10
10	1	0	1	0

Figura 3.6: Tabela verdade e mapa de Karnaugh para $t(x, y, z, w)$.

O mapa de quatro variáveis das Figuras 3.5 e 3.6 é desenhado como um quadrado apenas para facilitar o desenho em duas dimensões. Uma representação mais precisa é aquela de um toroide – as células da primeira linha da figura são adjacentes às da última linha (0 e 8, 1 e 9, 3 e 11, e 2 e 10), e as células da esquerda são adjacentes às da direita (0 e 2, 4 e 6, 12 e 14, e 8 e 10). Estas células são adjacentes porque há somente a troca de x para $\bar{x}$ nas linhas do topo e base, e de z para $\bar{z}$ nas linhas da esquerda e da direita, e portanto podem ocorrer simplificações que envolvem células em “margens opostas” do mapa.

Exemplo 3.27 A Figura 3.7 mostra a tabela verdade e a função simplificada para uma função que identifica números primos codificados em 4 bits – a função $p(a, b, c, d)$ é 1 sempre que o número representado pelos argumentos for primo.

A figura também mostra a soma dos seis produtos e a expressão simplificada através de cinco aplicações do teorema da simplificação (a). Para tal, basta olhar na tabela, nas linhas em que a função é 1, e verificar em quais pares de linhas ocorre uma variável e seu complemento, desde que os outros literais sejam idênticos. O termo marcado com uma estrela, é redundante e pode ser eliminado. $\triangleleft$

índice	a	b	c	d	p	
0	0	0	0	0	0	$p = (\bar{a} \wedge \bar{b} \wedge c \wedge \bar{d}) \vee m_2$
1	0	0	0	1	0	$(\bar{a} \wedge \bar{b} \wedge c \wedge d) \vee m_3$
2	0	0	1	0	1	$(\bar{a} \wedge b \wedge \bar{c} \wedge d) \vee m_5$
3	0	0	1	1	1	$(\bar{a} \wedge b \wedge c \wedge d) \vee m_7$
4	0	1	0	0	0	$(a \wedge \bar{b} \wedge c \wedge d) \vee m_{11}$
5	0	1	0	1	1	$(a \wedge b \wedge \bar{c} \wedge d) m_{13}$
6	0	1	1	0	0	elimina d de m_2 e m_3
7	0	1	1	1	1	elimina b de m_3 e m_7
8	1	0	0	0	0	$\Leftrightarrow$ elimina c de m_5 e m_7 *
9	1	0	0	1	0	elimina a de m_5 e m_{13}
10	1	0	1	0	0	elimina a de m_3 e m_{11}
11	1	0	1	1	1	$(\bar{a} \wedge \bar{b} \wedge c) \vee$
12	1	1	0	0	0	$(\bar{a} \wedge c \wedge d) \vee$
13	1	1	0	1	1	$(\bar{a} \wedge b \wedge d) \vee *$
14	1	1	1	0	0	$(\bar{b} \wedge c \wedge d) \vee$
15	1	1	1	1	0	$(b \wedge \bar{c} \wedge d)$

Figura 3.7: Tabela verdade para a função primos em 4 bits.

Exercícios

Ex. 3.1 Prove que o operador $\wedge$ de três entradas $\wedge(a, b, c)$ é equivalente a duas aplicações do operador $\wedge$ de duas entradas: $\wedge(a, b, c) \equiv (a \wedge b) \wedge c$.

Ex. 3.2 Estenda o resultado do Ex. 3.1 para qualquer número de entradas.

Ex. 3.3 Repita o Ex. 3.1 para os operadores $\vee$ and $\vee$.

Ex. 3.4 Estenda o resultado do Ex. 3.3 para qualquer número de entradas.

Ex. 3.5 Com base nos resultados dos Ex. 3.2 e 3.4, defina os operadores $\wedge$ e $\vee$ de uma maneira análoga à definição de somatório para inteiros.

Ex. 3.6 Com base no Ex. 3.5, mostre que $\wedge$ equivale ao quantificador universal $\forall$, e que $\vee$ equivale ao quantificador existencial $\exists$.

Ex. 3.7 Prove que o operador $\oplus$ é associativo. Prove que t , na Figura 3.6, é $t = \oplus(x, y, z, w)$. *Pista:* inicie provando que $\oplus(x, y, z)$ é associativo.

Ex. 3.8 Prove que o Teorema de DeMorgan é válido para qualquer número de variáveis.

Use as expressões abaixo para resolver os Exercícios 3.9 a 3.12.

$$a, b, c, d, e, p, q, r, s, t, u : \mathbb{B}$$

$$\begin{aligned} b \vee (c \wedge (d \vee e)) & \quad (i) \\ (q \wedge (\neg t \vee u)) \vee ((q \vee s) \wedge \neg(t \vee \neg s)) & \quad (ii) \\ a \vee \neg b \vee (\neg c \wedge d \wedge (\neg b \vee \neg a)) & \quad (iii) \\ (\neg p \wedge \neg q \wedge r) \vee \neg(s \wedge (p \vee r \vee \neg q)) & \quad (iv) \end{aligned}$$

Ex. 3.9 Preencha as tabelas verdade das expressões (i) a (iv). Enumere os mintermos com as variáveis na ordem alfabética.

Ex. 3.10 Use as propriedades dos bits para simplificar algebricamente as expressões (i) a (iv).

Ex. 3.11 Simplifique as expressões (i) a (iv) com mapas de Karnaugh.

Ex. 3.12 Escreva as duas formas canônicas para as expressões (i) a (iv).

Ex. 3.13 Com base na especificação do divisor (Eq. 3.7), especifique um circuito que efetua a multiplicação inteira.

Ex. 3.14 Defina a inversa da função num (Eq. 3.6). A função num^{-1} tem como argumento um número natural e produz uma tupla de bits que representa o argumento na base dois.

Índice Remissivo

Símbolos

%, 18–20
 $\Rightarrow$, 37
 $\bigvee$, 47
 $\bigwedge$, 47
 $\equiv$, 36
 $\wedge$, 30, 36
 $\triangleleft \triangleright$, 36, 42
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 41, 44
 $\neg$, 30, 36
 $\vee$, 30, 36
 $\oplus$, 36, 52
 $\mapsto$, 42
 $\bar{a}$, veja $\neg$
 π , 25
 e , 24
 $\mathbb{B}$, 29–33, 38, 42
 $\mathbb{Z}$, 42
 $\mathbb{N}$, 42, 43
 num , 44, 54
 num^{-1} , 54
 $\mathbb{R}$, 42
 $\langle \rangle$, 29, 42, 43

A

abstração, 27
 bits como sinais, 27–33
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo,
 conversão de base, 18
 conversão de base de frações, 22
 and, veja $\wedge$
 aproximação, 24
assembly, veja ling. de montagem
 associatividade, 31
 atraso, veja tempo de contaminação
 atribuição, 12

B

binário, 20
 bit, 20
 bits, 27–37
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 variável, 30
 byte, 11

C

capture FF, veja *flip flop*, destino
 clk, veja *clock*
Column Address Strobe, veja CAS
Complementary Metal-Oxide Semiconductor, veja CMOS
 complemento, veja $\neg$
 complemento, propriedade, 31
 comportamento transitório, veja transitório
 comutatividade, 31
 condicional, veja $\triangleleft \triangleright$
 conjunção, veja $\wedge$
 contra-positiva, 41
 conversão de base, 18

D

datapath, veja circuito de dados
 decimal, 17
design unit, veja VHDL, unidade de projeto
 disjunção, veja $\vee$
 distributividade, 31, 34, 51
 divisão inteira, 44
 dual, 32
 dualidade, 32

E

enviesado, relógio, veja *skew*
 equivalência, veja $\Leftrightarrow$
 erro,
 de representação, 23
 especificação, 42
 expressões, 36

F

fechamento, 31
Field Effect Transistor, veja FET
Field Programmable Gate Array, veja FPGA
flip-flop,
 modelo VHDL, veja VHDL, *flip-flop*
 um por estado, veja um FF por estado
 forma canônica, 48
 frações, veja ponto fixo
 frequência máxima, veja relógio
 função, 30
 tipo, 29, 42
 função, aplicação bit a bit, 32
 função, tipo (op. infix), veja $\mapsto$

G

glitch, veja transitório

gramática, 17

H

hexadecimal, 19

I

idempotência, 31

identidade, 31

igualdade, 30

implementação, 42

implicação, *veja* $\Rightarrow$

informação, 16

Instrução,

busca, *veja* busca

instrução, 12

busca, *veja* busca

decodificação, *veja* decodificação

execução, *veja* execução

resultado, *veja* resultado

interface,

de rede, 13

de vídeo, 12

involução, 31

L

latch, *veja* basculo

latch FF, *veja flip flop*, destino

launch FF, *veja flip flop*, fonte

linguagem,

assembly, *veja* ling. de montagem

Z, 27

literal, 38

logaritmo, 43

M

Mapa de Karnaugh, 49

Máquina de Mealy, *veja* máq. de estados

Máquina de Moore, *veja* máq. de estados

máscara, 32

máximo e mínimo, 31

maxtermo, 46

Mealy, *veja* máq. de estados

memória,

de vídeo, 13

primária, 13

secundária, 13

memória dinâmica, *veja* DRAM

memória estática, *veja* SRAM

mintermo, 45

modelo,

funcional, 44

módulo, *veja* %, *mod*

Moore, *veja* máq. de estados

multiply-add, *veja* MADD

N

número,

de Euler, 24

negação, *veja* $\neg$

nível lógico,

0 e 1, 28

indeterminado, 28

non sequitur, 37

not, *veja* $\neg$

número primo, 53

O

octal, 18

operação,

binária, 29

bit a bit, 32

infixada, 42

prefixada, 47

unária, 29

operações sobre bits, 29–33

operador,

binário, 29

lógico, 36

unário, 29

operation code, *veja opcode*

or, *veja* $\vee$

ou exclusivo, *veja* $\oplus$

ou inclusivo, *veja* $\vee$

P

paridade,

ímpar, 49

par, 49

período mínimo, *veja* relógio

pipelining, *veja* segmentação

piso, *veja* [v]

porta lógica,

carga, *veja fan-out*

potenciação, 43

precedência, 30

precisão,

representação, 23

processador, 12

produtório, 33

produto de somas, 46

programa de testes, *veja* VHDL, *testbench*

propriedades, operações em $\mathbb{B}$, 31

prova de equivalência, 40–41

pulso,

espúrio, *veja transitório*

R

RAM, 12

Random Access Memory, *veja* RAM

Read Only Memory, *veja* ROM

redução, 33

Register Transfer Language, *veja* RTL

registrador de deslocamento,

modelo VHDL, *veja* VHDL, *registrador*

relógio,

enviesado, *veja skew*

representação,

abstrata, 28

binária, 20

concreta, 27

decimal, 17
hexadecimal, 19
octal, 18
posicional, 17
precisão, 23
ROM, 12
Row Address Strobe, veja *RAS*

S

semântica, 17
silogismo, 37
simplificação de expressões, 38–40
sinal, 27, 42
 analógico, 27
 digital, 27, 28
síntese, veja *VHDL*, *síntese*
Solid State Disk, veja *SSD*
soma, veja *somador*
soma de produtos, 45, 51
 completa, 45
somatório, 33
spice, 28
SSD, 14

T

tabela verdade, 33–35, 45
tamanho, veja $|N|$
Teorema,
 Absorção, 49
 DeMorgan, 32, 41, 48, 54
 Simplificação, 49
testbench, veja *VHDL*, *testbench*
teto, veja $\lceil r \rceil$
three-state, veja *terceiro estado*
tipo,
 função, 29
tipo de sinal, 42
Tipo I, veja *formato*
Tipo J, veja *formato*
Tipo R, veja *formato*
transferência entre registradores, veja *RTL*
Transistor-Transistor Logic, veja *TTL*
transmission gate, veja *porta de transmissão*
tupla, veja $\langle \rangle$
 elemento, 32
 largura, 43

U

Unidade de Lógica e Aritmética, veja *ULA*

V

valor da função, 30
vetor de bits, veja $\langle \rangle$, 32
 largura, 43
VHDL,
 design unit, veja *VHDL*, *unidade de projeto*
 tipos, 42

W

write back, veja *resultado*

X

xor, veja $\oplus$

Z

Z, *linguagem*, 27

Capítulo 4

Circuitos Combinacionais

*Palavra puxa palavra, uma ideia traz outra, e assim se faz
um livro, um governo, ou uma revolução; alguns dizem
mesmo que assim é que a natureza compôs as suas espécies.
Machado de Assis, Primas de Sapucaia!*

Um *circuito combinacional* produz saídas que dependem exclusivamente dos valores nas suas entradas, e para um mesmo conjunto de entradas o circuito produz sempre o mesmo conjunto de saídas. Circuitos combinacionais não têm *estado* nem *memória*.

Este capítulo¹ define os *circuitos combinacionais* que as empregam². Com essas definições postas, são investigados os circuitos que implementam algumas funções lógicas que são comuns no projeto de sistemas digitais. A Seção 4.2 descreve os circuitos que implementam os operadores lógicos básicos, chamados de *portas lógicas*. A implementação destes circuitos é discutida no Capítulo 5. A Seção 4.3 especifica o comportamento de multiplexadores, demultiplexadores e decodificadores.

Os nomes dos circuitos que implementam os operadores lógicos estão propositalmente em Inglês para evitar confusão com as palavras ‘e’ e ‘ou’ do Português. Assim, a porta lógica com a função de conjunção, ou *e-lógico*, é chamada de porta *and*. A porta com a função de disjunção, ou *ou-lógico*, é chamada de porta *or*.

Espaço em branco proposital.

¹© Roberto André Hexsel, 2012-2021. Versão de 13 de fevereiro de 2021.

²A definição de *circuito combinacional* apresentada na Seção 4.1 toma emprestado muito dos conteúdos da disciplina 6.004 – *Computation Structures* do MIT, versão de 2013.

4.1 O Que É Um Circuito Combinacional?

Com a abstração para os sinais elétricos definida em termos de *bits*, ou *sinais digitais* que podem assumir um dentre os valores discretos em $\mathbb{B}$, é possível definir o que se entende por *dispositivo combinacional* e por *circuito combinacional*.

Definição 4.1 (Dispositivo Combinacional) *Um Dispositivo Combinacional é um dispositivo com as seguintes propriedades:*

1. *uma ou mais entradas digitais;*
2. *uma ou mais saídas digitais;*
3. *uma especificação funcional que determina o valor lógico de cada saída para cada uma das combinações das entradas; e*
4. *uma especificação temporal que determina, pelo menos, um limite superior para o tempo de propagação dos sinais que atravessam o dispositivo.*

A especificação funcional do dispositivo pode ser um conjunto de somas de produtos que relacionam todas as combinações de valores nas entradas com valores bem definidos nas saídas, ou ainda uma tabela verdade para cada saída que relaciona entradas com os valores de saída. Uma *especificação funcional* é um contrato entre o projetista de um dispositivo e o usuário deste dispositivo, e garante que *entradas válidas implicam em saídas válidas*.

O comportamento temporal dos dispositivos pode ser mensurado nos terminais do dispositivo, ou computado através de simulações, e o que nos interessa agora é o *tempo de propagação* dos sinais através do dispositivo, ou o *atraso (delay)* na propagação dos sinais introduzido pelo dispositivo. O tempo de propagação é o tempo necessário para que o circuito produza uma saída válida estável, a partir do instante em que suas entradas estabilizaram. Voltaremos a este assunto na Seção 5.3.

Definição 4.2 (Circuito Combinacional) *Um circuito combinacional pode ser construído pela interligação de dispositivos combinacionais que atendem às seguintes restrições:*

5. *cada componente é um dispositivo combinacional;*
6. *cada uma das entradas está ligada a exatamente uma saída ou às constantes 0 ou 1;*
7. *o circuito não contém circularidade nas ligações ($\lambda \rightsquigarrow \sigma \rightsquigarrow \pi \not\rightsquigarrow \lambda$).*

A restrição 6 proíbe a ligação de duas saídas, o que causa curto-circuitos, e também proíbe entradas desligadas (circuitos abertos). A restrição 7 impede laços que liguem as saídas do componente Λ às entradas do componente Π , que direta ou indiretamente formam um laço que alimenta as entradas de Λ .

Considere o circuito combinacional mostrado na Figura 4.1, que consiste de um circuito com três entradas (p, q, r) e dois dispositivos combinacionais, cada um com sua especificação funcional e temporal. Para nos certificarmos de que o circuito na figura é combinacional basta observar que o sinal interno y torna-se válido e bem definido 2ns depois que as entradas p e q estabilizam. A saída s torna-se válida e bem definida $2 + 1$ ns depois que as entradas p , q e r estabilizam. O sinal y está ligado a exatamente uma única saída, e não há circularidade porque não existe ligação entre as saídas y e s com as entradas p , q ou r .

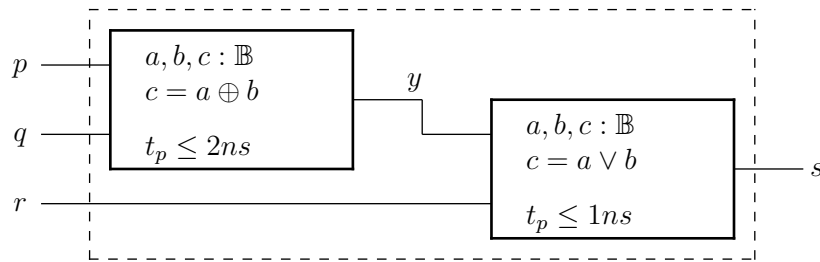


Figura 4.1: Circuito combinacional.

O valor da saída s , seja ele 0 ou 1, pode ser facilmente deduzido pela composição das funções dos dois componentes. Este circuito tem um tempo de propagação de, no máximo, $3ns$, e sua saída é uma função dos sinais de entrada, sendo portanto um circuito combinacional. Esta forma de verificação pode ser aplicada a circuitos com um número arbitrário de componentes.

Como é computado o tempo de propagação através de um circuito combinacional? Para que se possa usar um determinado dispositivo com a confiança de que seu tempo de propagação é um limite superior para o intervalo entre o instante em que as entradas ficam estáveis e o instante no qual a saída fica estável, devemos computar o *tempo de propagação de pior caso*. Para cada caminho entre as entradas e a saída computa-se o atraso cumulativo naquele caminho, que é a soma dos atrasos impostos pelos componentes no caminho. O tempo de propagação do dispositivo é o mais longo dentre os tempos de propagação de todos os caminhos. A Seção 5.3 explora este tema em mais detalhe.

Exemplo 4.1 Ignorando sua função, qual é o tempo de propagação do circuito na Figura 4.2?

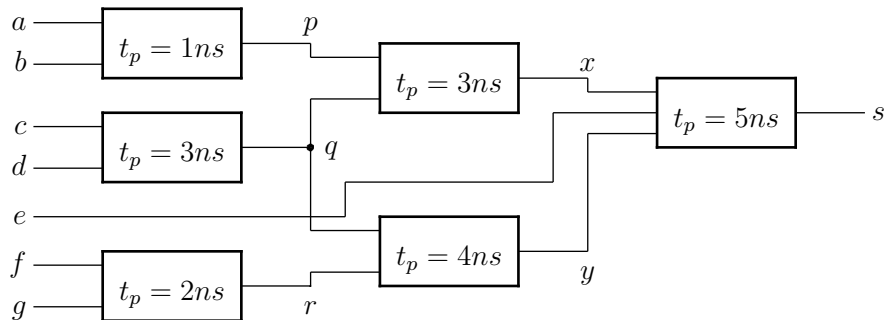


Figura 4.2: Circuito combinacional com atrasos.

1. O caminho mais curto é o da entrada e para a saída, que é de $5ns$;
2. o caminho $\{a, b\} \rightsquigarrow p \rightsquigarrow x \rightsquigarrow s$ tem custo de $1 + 3 + 5 = 9ns$;
3. o caminho $\{c, d\} \rightsquigarrow q \rightsquigarrow x \rightsquigarrow s$ tem custo de $3 + 3 + 5 = 11ns$;
4. o caminho $\{c, d\} \rightsquigarrow q \rightsquigarrow y \rightsquigarrow s$ tem custo de $3 + 4 + 5 = 12ns$; e
5. o caminho $\{f, g\} \rightsquigarrow r \rightsquigarrow y \rightsquigarrow s$ tem custo de $2 + 4 + 5 = 11ns$.

O tempo de propagação do circuito é o pior caso, que é $12ns$.

◀

4.2 Portas Lógicas

A abstração de sinais como bits, definida na Seção 3.1, é um modelo abstrato para o comportamento dos circuitos eletrônicos que são usados para implementar sistemas digitais. Se o que ocorre durante as transições entre os dois níveis lógicos pode ser ignorado, então o modelo abstrato é uma descrição perfeitamente razoável para o comportamento da implementação das portas lógicas na tecnologia CMOS, que são o objeto do Capítulo 5.

Os operadores conjunção ($\wedge$), disjunção ($\vee$) e complemento ($\neg$), podem ser representados pelos símbolos gráficos das portas lógicas *and*, *or* e *not*, respectivamente. Os símbolos destes circuitos são mostrados na Figura 4.3, e suas funções são definidas na Equação 4.1. As portas *and* e *or* são definidas para duas entradas; estas definições podem ser entendidas para qualquer número de entradas, empregando-se a Definição 3.16.

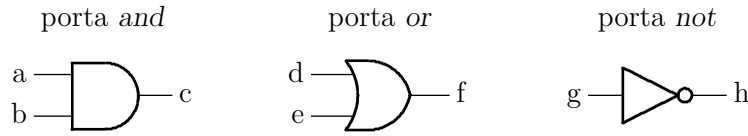


Figura 4.3: Símbolos das portas lógicas *and*, *or* e *not*.

$$a, b, c, d, e, f, g, h : \mathbb{B}$$

$$\text{and-2} : (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B}$$

$$\text{and-2}(a, b, c) \equiv c = a \wedge b$$

$$\text{or-2} : (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B}$$

$$\text{or-2}(d, e, f) \equiv f = d \vee e$$

$$\text{not} : \mathbb{B} \mapsto \mathbb{B}$$

$$\text{not}(g, h) \equiv h = \neg g$$

(4.1)

O comportamento das portas lógicas também pode ser descrito por *tabelas verdade*, e a Tabela 4.1 define o comportamento dos operadores $\wedge$, $\vee$ e $\neg$, e também das portas *and*, *or* e *not*. Note que as Tabelas 3.1 e 4.1 são idênticas.

Tabela 4.1: Tabelas Verdade das portas *and*, *or* e *not*.

<i>and</i>			<i>or</i>			<i>not</i>	
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

O comportamento da porta *or* difere do senso comum porque normalmente a palavra ‘ou’ é entendida como “ou exclusivo” e portanto uma alternativa exclui a outra, enquanto que a função $\vee$ define um “ou inclusivo” que admite uma das alternativas, além de ambas as alternativas. O operador “ou exclusivo”, que somente admite uma das alternativas é definido pelo operador $\oplus$ e implementado pela porta lógica *xor*.

Como indicado na Seção 3.2 e nos Exercícios 4.2 e 4.4, as funções $\wedge$ e $\vee$ são associativas. Se a implementação destas funções é correta, então construção de portas com um número arbitrário de entradas é trivial.

As portas lógicas *and* e *or* correspondem às funções lógicas $\wedge$ e $\vee$. Do ponto de vista de implementação de sistemas digitais nas tecnologias disponíveis em 2021, os circuitos menores, portanto mais rápidos e que dissipam menos energia, são aqueles que implementam as portas lógicas *nand* e *nor*. A estrutura interna destas portas lógicas na tecnologia CMOS é apresentada no Capítulo 5. A Equação 4.2 e a Tabela 4.2 definem seus comportamentos, e a Figura 4.4 mostra os símbolos das portas *nand*, *nor* e *xor*.

$$\begin{aligned}
 &a, b, c, d, e, f, g, h, i : \mathbb{B} \\
 &nand-2 : (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B} \\
 &nand-2(a, b, c) \equiv c = \neg(a \wedge b) \\
 &nor-2 : (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B} \\
 &nor-2(d, e, f) \equiv f = \neg(d \vee e) \\
 &xor-2 : (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B} \\
 &xor-2(g, h, i) \equiv i = \neg g \wedge h \vee g \wedge \neg h
 \end{aligned} \tag{4.2}$$

Tabela 4.2: Tabelas Verdade das portas *nand*, *nor* e *xor*.

<i>nand</i>			<i>nor</i>			<i>xor</i>		
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>
0	0	1	0	0	1	0	0	0
0	1	1	0	1	0	0	1	1
1	0	1	1	0	0	1	0	1
1	1	0	1	1	0	1	1	0

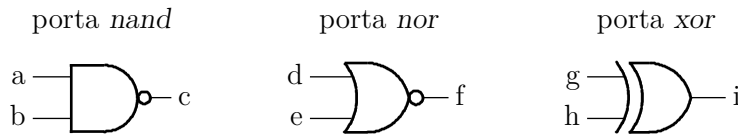


Figura 4.4: Símbolos das portas lógicas *nand*, *nor* e *xor*.

Um ponto importante e que merece ênfase é o seguinte: as definições da Equação 4.2 e da Tabela 4.2 são rigorosamente equivalentes e expressam a mesma informação sobre o comportamento dos circuitos. Da mesma forma, os símbolos gráficos das portas lógicas são apenas uma outra forma de representar aquela informação.

O *Teorema de DeMorgan* é útil na implementação e simplificação de circuitos, e é reescrito na Equação 4.3. Este teorema permite implementar funções lógicas mesmo que as portas lógicas disponíveis não sejam as ‘necessárias’ – por ‘necessárias’ entenda-se aquelas portas que correspondem fielmente às equações que descrevem o comportamento do circuito.

$$\overline{x \wedge y} = \overline{x} \vee \overline{y} \qquad \overline{x \vee y} = \overline{x} \wedge \overline{y} \tag{4.3}$$

DeMorgan nos permite expressar as funções das portas lógicas de mais de uma maneira, e estas equivalências podem ser muito úteis se a tecnologia de implementação impõe restrições quanto à variedades das portas disponíveis, como veremos na Seção 5.2.2. A Figura 4.5 mostra as equivalências entre as funções e as portas lógicas que as implementam.

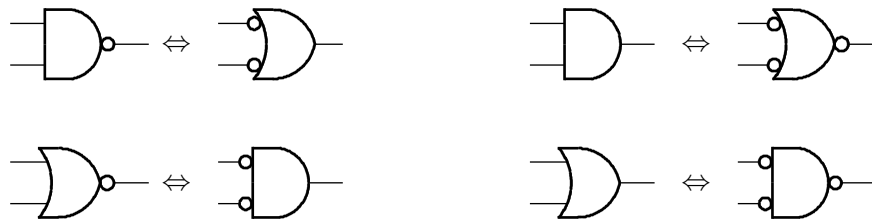


Figura 4.5: Equivalências entre funções lógicas ao aplicar DeMorgan.

As duas equivalências à esquerda são aplicações diretas da Equação 4.3: a porta *nand* equivale a um *or* com as entradas negadas; a porta *nor* equivale a um *and* com as entradas negadas. As equivalências à direita têm uma negação adicional nas saídas.

A Figura 4.6 mostra a implementação de um circuito multiplexador com quatro portas *nand*. A porta N1 é usada como inversor e a porta N4 é usada como *or*. As inversões nas saídas das portas N2 e N3 cancelam as inversões nas entradas da porta N4.

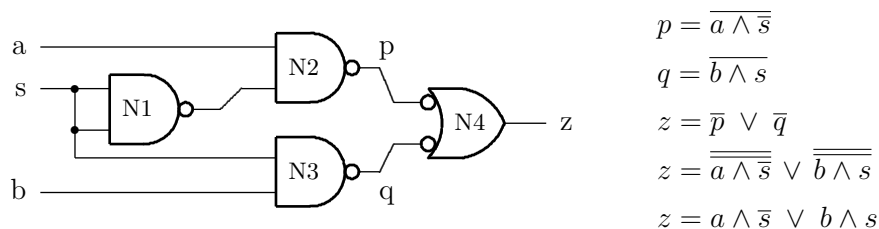


Figura 4.6: Multiplexador implementado com quatro portas *nand*.

Uma aplicação importante, do ponto de vista prático, do Teorema de DeMorgan é facilitar a compreensão da função dos circuitos. A funcionalidade do circuito da Figura 4.6 é facilmente apreendida porque a porta lógica mais à direita, que tem a função de disjunção, está desenhada como uma porta *or*, com suas entradas invertidas. As duas portas no centro do circuito têm a função de conjunção, e estão desenhadas como portas *and* com as saídas invertidas. Como mencionado acima, as inversões nas saídas são canceladas pelas inversões nas entradas, pela propriedade da involução.

Exemplo 4.2 Supondo que o tempo de propagação de uma porta *nand* seja de 2ns (2×10^{-9} s), qual é o tempo de propagação do circuito da Figura 4.6?

O caminho mais longo atravessa as portas $N1 \rightsquigarrow N2 \rightsquigarrow N4$, e os sinais demoram 2ns para atravessar cada porta. Se uma das três entradas se altera, de 1 para 0 ou de 0 para 1, essa alteração será refletida de forma estável na saída após 6ns.

A palavra ‘*estável*’ na frase anterior é importante: se todas as três entradas mudarem no mesmo instante, é possível que a saída z apresente um valor transitório após 4ns ($N2 \rightsquigarrow N4$ ou $N3 \rightsquigarrow N4$), que depois de mais 2ns sofreria nova alteração, para então permanecer estável. A saída do multiplexador somente pode ser usada com a confiança de que ela reflete os valores *estáveis* nas suas entradas depois de decorrido o tempo de propagação de 6ns. ◀

Exemplo 4.3 Frequentemente é necessário verificar se dois números são iguais, e para tanto emprega-se um *comparador de igualdade*. Iniciemos com números de dois bits; a Figura 4.7 mostra a tabela verdade para um comparador de números de dois bits, $A = \langle a_1 a_0 \rangle$ e $B = \langle b_1 b_0 \rangle$.

índice	$a_1 a_0$	$b_1 b_0$	ig
0	0 0	0 0	1
1	0 0	0 1	0
2	0 0	1 0	0
3	0 0	1 1	0
4	0 1	0 0	0
5	0 1	0 1	1
6	0 1	1 0	0
7	0 1	1 1	0
8	1 0	0 0	0
9	1 0	0 1	0
10	1 0	1 0	1
11	1 0	1 1	0
12	1 1	0 0	0
13	1 1	0 1	0
14	1 1	1 0	0
15	1 1	1 1	1

ig	$b_1 b_0$			
	00	01	11	10
$a_1 a_0$				
00	0 1	1 0	3 0	2 0
01	4 0	5 1	7 0	6 0
11	12 0	13 0	15 1	14 0
10	8 0	9 0	11 0	10 1

Figura 4.7: Tabela verdade e Mapa de Karnaugh para o comparador.

Não é possível simplificar esta função por agrupamentos no Mapa de Karnaugh, mas é possível simplificá-la algebricamente.

$$\begin{aligned}
ig &= (\overline{a_1} \wedge \overline{a_0} \wedge \overline{b_1} \wedge \overline{b_0}) \vee (\overline{a_1} \wedge a_0 \wedge \overline{b_1} \wedge b_0) \vee \\
&\quad (a_1 \wedge \overline{a_0} \wedge b_1 \wedge \overline{b_0}) \vee (a_1 \wedge a_0 \wedge b_1 \wedge b_0) \\
&\Leftrightarrow \text{associatividade, } 2\times \\
&\quad \overline{a_1} \wedge (\overline{a_0} \wedge \overline{b_1} \wedge \overline{b_0}) \vee \overline{a_1} \wedge (a_0 \wedge \overline{b_1} \wedge b_0) \vee \\
&\quad a_1 \wedge (\overline{a_0} \wedge b_1 \wedge \overline{b_0}) \vee a_1 \wedge (a_0 \wedge b_1 \wedge b_0) \\
&\Leftrightarrow \text{distributividade, } 2\times \\
&\quad \overline{a_1} \wedge [(\overline{a_0} \wedge \overline{b_1} \wedge \overline{b_0}) \vee (a_0 \wedge \overline{b_1} \wedge b_0)] \vee \\
&\quad a_1 \wedge [(\overline{a_0} \wedge b_1 \wedge \overline{b_0}) \vee (a_0 \wedge b_1 \wedge b_0)] \\
&\Leftrightarrow \text{associatividade, } 2\times \\
&\quad \overline{a_1} \wedge [\overline{b_1} \wedge (\overline{a_0} \wedge \overline{b_0}) \vee \overline{b_1} \wedge (a_0 \wedge b_0)] \vee \\
&\quad a_1 \wedge [b_1 \wedge (\overline{a_0} \wedge \overline{b_0}) \vee b_1 \wedge (a_0 \wedge b_0)] \\
&\Leftrightarrow \text{distributividade, } 2\times \\
&\quad \overline{a_1} \wedge [\overline{b_1} \wedge [(\overline{a_0} \wedge \overline{b_0}) \vee (a_0 \wedge b_0)]] \vee \\
&\quad a_1 \wedge [b_1 \wedge [(\overline{a_0} \wedge \overline{b_0}) \vee (a_0 \wedge b_0)]] \\
&\Leftrightarrow \text{definição } \oplus, 2\times \\
&\quad \overline{a_1} \wedge [\overline{b_1} \wedge (a_0 \oplus \overline{b_0})] \vee a_1 \wedge [b_1 \wedge (a_0 \oplus \overline{b_0})] \\
&\Leftrightarrow \text{associatividade, } 4\times \\
&\quad (\overline{a_1} \wedge \overline{b_1}) \wedge (a_0 \oplus \overline{b_0}) \vee (a_1 \wedge b_1) \wedge (a_0 \oplus \overline{b_0}) \\
&\Leftrightarrow \text{distributividade} \\
&\quad (a_0 \oplus \overline{b_0}) \wedge [(\overline{a_1} \wedge \overline{b_1}) \vee (a_1 \wedge b_1)] \\
&\Leftrightarrow \text{definição } \oplus \\
&\quad (a_0 \oplus \overline{b_0}) \wedge (a_1 \oplus \overline{b_1}) \\
&\Leftrightarrow \text{definição } \oplus \\
&\quad \overline{(a_0 \oplus b_0)} \wedge \overline{(a_1 \oplus b_1)}
\end{aligned}$$

Esta função pode ser implementada com duas portas *xnor* e a uma porta *and*, e o circuito é mostrado na Figura 4.8. Este circuito pode ser estendido da maneira óbvia para a comparação de vetores de bits com qualquer largura. ◀

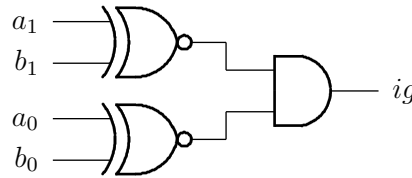


Figura 4.8: Comparador de igualdade para números de dois bits.

Exemplo 4.4 Os *Códigos de Gray* são usados em sistemas que devem operar de forma segura, para alguma definição de ‘seguro’. Considere, por exemplo, o eixo que controla a posição vertical de um canhão. A medida da posição angular do eixo é indicada por uma série de furos ao longo do raio de um disco que é rigidamente fixado ao eixo. A cada intervalo, um conjunto radial de furos indica o ângulo do eixo com relação à horizontal, de forma que cada combinação de furos indica o ângulo absoluto do eixo. O disco do medidor de posição angular é mostrado na Figura 4.9.

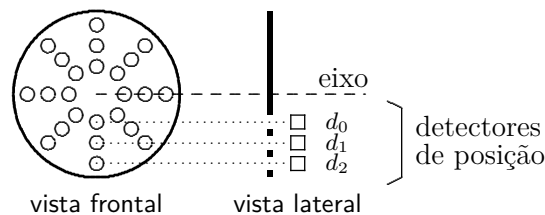


Figura 4.9: Disco do detector de posição angular.

Suponha ainda que o sistema de numeração binária com três bits é usado para marcar os ângulos. O que pode ocorrer quando o eixo está mudando de posição, de 011 para 100? Por problemas de construção, ou desgaste, o ângulo poderia ser medido pelos detectores na sequência 011 → 001 → 000 → 100, cuja correspondente em decimal é 3 → 1 → 0 → 4. Esta sequência de ângulos certamente provocaria uma pane no acionador vertical do canhão.

Uma solução para este problema consiste em usar uma codificação na qual só ocorre uma única troca de bit entre cada dois vizinhos na sequência – só uma posição troca de 1 para 0 ou de 0 para 1. A Tabela 4.3 mostra a geração de um Código de Gray em três bits.

Tabela 4.3: Geração de um Código de Gray em três bits.

1 bit	refl.	2 bits	refl.	3 bits
0	0	0 0	00	0 00
1	1	0 1	01	0 01
	1	1 1	11	0 11
	0	1 0	10	0 10
			10	1 10
			11	1 11
			01	1 01
			00	1 00

O código de um bit consiste de dois casos, 1 e 0, e é mostrado na coluna da esquerda. A construção do código de dois bits se dá em dois passos: (i) o código de um bit é copiado de forma refletida, como mostra a segunda coluna; (ii) à metade original é acrescentado um 0 à esquerda, e à parte refletida é acrescentado 1 à esquerda. As duas colunas da direita repetem estes dois passos na construção do código de três bits. Note que o 100 da última linha é adjacente ao 000 da primeira linha.

A Tabela 4.4 associa o código de três bits aos índices da Tabela 4.3. Esta é também a tabela verdade para as três funções, g_2, g_1, g_0 , que geram o dígito apropriado a partir das entradas $\langle d_2 d_1 d_0 \rangle$.

Tabela 4.4: Tabela verdade para o Código de Gray em três bits.

índice	$d_2 d_1 d_0$	$g_2 g_1 g_0$
0	0 0 0	0 0 0
1	0 0 1	0 0 1
2	0 1 0	0 1 1
3	0 1 1	0 1 0
4	1 0 0	1 1 0
5	1 0 1	1 1 1
6	1 1 0	1 0 1
7	1 1 1	1 0 0

Ao inspecionar a Tabela 4.4, percebe-se que $g_2 = d_2$, e a implementação de g_2 é apenas um fio ligado a d_2 . Da tabela verdade obtemos

$$g_2 = d_2, \quad g_1 = d_2 \wedge \overline{d_1} \vee \overline{d_2} \wedge d_1 \quad \text{e} \quad g_0 = \overline{d_1} \wedge d_0 \vee d_1 \wedge \overline{d_0}.$$

As funções g_1 e g_0 podem ser implementadas com portas *xor*. ◁

Exemplo 4.5 Supondo que os tempos de propagação de portas *and* e *or* sejam de 2ns, e o de um inversor de 1ns, qual é o tempo de propagação do circuito que gera Código de Gray do Exemplo 4.4?

O caminho mais longo atravessa um inversor, uma porta *and* e uma porta *or*. Logo, o tempo de propagação, no caminho mais longo, é $1 + 2 + 2 = 5\text{ns}$. ◁

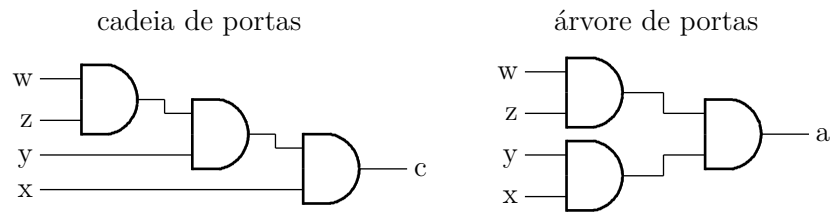
Exemplo 4.6 Considere a função $\wedge$ de 4 entradas, implementada com três portas de duas entradas *and*-2: $c = \wedge(x, y, z, w) = x \wedge (y \wedge (z \wedge w))$. Este circuito é mostrado no lado esquerdo da Figura 4.10 e seu tempo de propagação é dado pelo caminho que atravessa as três portas *and* encadeadas: $T_{\text{cadeia}} = 3 \times T_A$.

Outra implementação para $\wedge$ de 4 entradas é obtida se tirarmos proveito da associatividade da conjunção: $a = \wedge(x, y, z, w) = (x \wedge y) \wedge (z \wedge w)$. Este circuito é mostrado à direita na Figura 4.10, e seu tempo de propagação é menor do que o da cadeia de *ands*: $T_{\text{árvore}} = 2 \times T_A$.

Se um operador é associativo, o número de entradas N é uma potência de dois e $N \geq 4$, então a implementação em árvore tem um tempo de propagação menor do que o da cadeia de portas:

$$(N - 1)T_P = T_{\text{cadeia}} > T_{\text{árvore}} = (\log_2 N)T_P.$$

O número de portas de duas entradas é igual nos dois casos. ◁

Figura 4.10: Duas implementações para *and-4* com *and-2*.

Exemplo 4.7 Qual o tempo de propagação de um circuito que computa a conjunção de oito sinais, construído com duas árvores como as do Exemplo 4.6?

A implementação do circuito, com duas árvores de altura dois, é

$$a = \bigwedge(x, y, z, w, p, q, r, s) = \bigwedge(x, y, z, w) \wedge \bigwedge(p, q, r, s).$$

Supondo que o tempo de propagação de uma porta *and-2* seja de 2ns, cada ramo da árvore tem tempo de propagação de 4ns, enquanto que o tempo da árvore completa é de 6ns. Como indicado no Exemplo 4.6, $T_{\text{árvore}} = (\log_2 8) \times 2 = 6\text{ns}$. Para uma cadeia com sete portas *and-2*, o tempo de propagação é de 14ns. Compare as curvas para $y = x$ e $z = \log_2(x)$. $\triangleleft$

4.2.1 Conjunto Mínimo de Operadores

Do ponto de vista de tecnologia de fabricação de circuitos integrados, uma questão importante é relacionada ao conjunto mínimo de operadores que deve ser oferecido aos projetistas para permitir a implementação de qualquer função lógica. Pode-se facilmente provar que, com exatamente um dentre os seguintes conjuntos de operadores, é possível implementar qualquer função em $\mathbb{B}$: $\{\neg, \wedge\}$, $\{\neg, \vee\}$, $\{\neg, \Rightarrow\}$. Note que os dois primeiros são as funções *nand* e *nor*, que são as portas lógicas básicas da tecnologia CMOS.

Tabela 4.5: Funções de duas variáveis.

ab	0	$\wedge$	f_2	a	f_4	b	$\neq$	$\vee$	<i>nor</i>	$=$	$\neg b$	$\Rightarrow$	$\neg a$	$\Leftarrow$	<i>nand</i>	1
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	1	1	0	1	1
10	0	0	1	1	0	0	1	1	0	0	1	0	0	1	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

A Tabela 4.5 contém as 16 funções de duas variáveis em $\mathbb{B}$, e indica as funções mais conhecidas. Note que as funções $\neq$ e $=$ também são conhecidas como *xor* ou *ou-exclusivo*, e *xnor*, respectivamente. As funções $\Rightarrow$ e $\Leftarrow$ são implicação lógica: $a \Rightarrow b$, e $b \Rightarrow a$. Quais são as funções identificadas por f_2 e f_4 ?

4.3 Circuitos Combinacionais Básicos

Esta seção contém a especificação e implementações de três circuitos combinacionais básicos que permitem escolher um dentre um conjunto de valores. Os circuitos são chamados de multiplexador, demultiplexador e decodificador. Tais circuitos são relevantes porque são empregados de inúmeras maneiras como blocos básicos na construção de circuitos mais complexos.

4.3.1 Multiplexador

Como vimos na Seção 3.3, o operador condicional equivale ao comando `if then else`. O circuito que implementa o condicional é chamado de *multiplexador*, e é representado em diagramas por um trapézio, como mostra a Figura 4.11. No lado mais largo ficam as entradas, no lado estreito a saída, e no topo ou na base fica o sinal de controle, ou o sinal de seleção. A Figura 4.11 mostra três ifs e os circuitos equivalentes. No terceiro circuito o sinal *c2* é compartilhado pelos dois multiplexadores da esquerda.

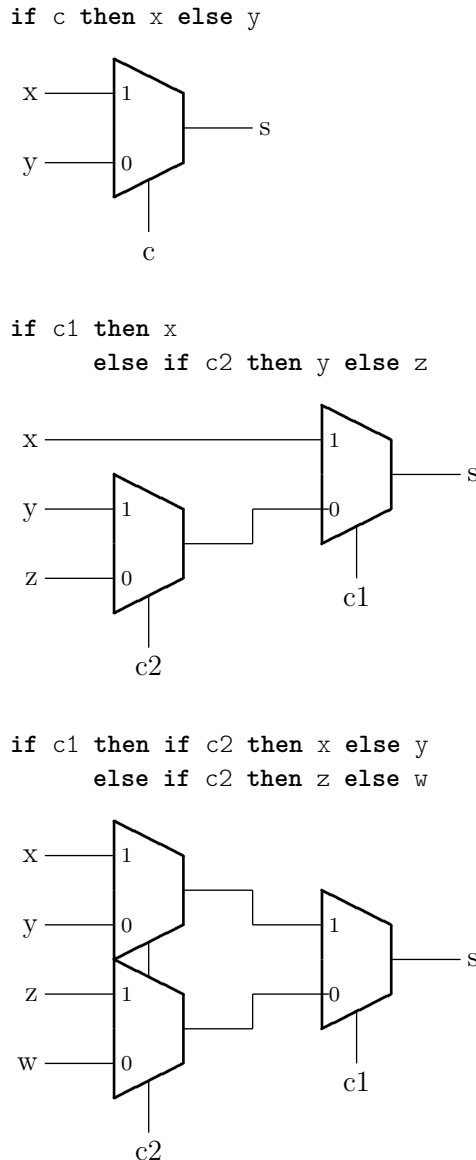


Figura 4.11: Seleção com multiplexadores de duas entradas.

Um *multiplexador de duas entradas* é um circuito com duas entradas *a, b*, uma entrada de controle *s* e uma saída *z*. A entrada de controle *s* permite escolher qual dentre as entradas *a* e *b* será apresentada na saída, conforme especificado pela Equação 4.4. A Figura 4.12 mostra o circuito e o símbolo do multiplexador de duas entradas.

$$\begin{aligned}
a, b, s, z &: \mathbb{B} \\
\text{mux-2} &: \mathbb{B} \times (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B} \\
\text{mux-2}(a, b, s, z) &\equiv z = b \triangleleft s \triangleright a
\end{aligned} \tag{4.4}$$

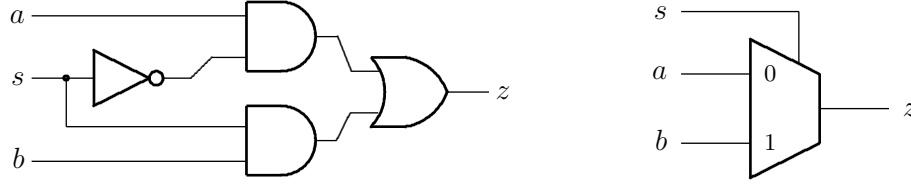


Figura 4.12: Multiplexador de duas entradas.

A expressão “ $\text{mux-2} : \mathbb{B} \times (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B}$ ” define o *tipo* do circuito. O multiplexador possui uma entrada de controle do tipo $\mathbb{B}$, uma entrada que é um par de $\mathbb{B}$, com tipo $(\mathbb{B} \times \mathbb{B})$, e produz uma saída de tipo $\mathbb{B}$.

Multiplexadores com maior número de entradas podem ser construídos pela composição de multiplexadores de duas entradas. A Equação 4.5 define um multiplexador de 4 entradas – preste atenção aos índices das variáveis de controle que estão representados em base-2.

$$\begin{aligned}
S &: \mathbb{B}^2 \\
A &: \mathbb{B}^4 \\
\text{mux-4} &: (\mathbb{B} \times \mathbb{B}) \times (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B} \\
\text{mux-4}(a_{0..3}, s_{1..0}, z) &\equiv z = (a_{11} \triangleleft s_0 \triangleright a_{10}) \triangleleft s_1 \triangleright (a_{01} \triangleleft s_0 \triangleright a_{00})
\end{aligned} \tag{4.5}$$

Como definido na Equação 4.5, o mux-4 é uma árvore de mux-2 , e esta árvore tem ‘altura’ dois, contando-se o número de multiplexadores entre a saída e as entradas. Este processo pode ser levado adiante na construção de multiplexadores de qualquer número de entradas, como indica a Equação 4.6.

$$\begin{aligned}
S &: \mathbb{B}^n \\
A &: \mathbb{B}^{2^n} \\
\text{mux-2}^n &: (\Pi_n \mathbb{B}) \times (\Pi_{2^n} \mathbb{B}) \mapsto \mathbb{B} \\
\text{mux-2}^n(a_0 \cdots a_{2^n-1}, s_{n-1} \cdots s_0, z) &\equiv \\
z &= ((a_{n-1} \triangleleft s_0 \triangleright a_{n-2}) \cdots) \triangleleft s_{n-1} \triangleright (\cdots (a_1 \triangleleft s_0 \triangleright a_0))
\end{aligned} \tag{4.6}$$

A Figura 4.13 mostra uma implementação para o circuito do multiplexador de 2^n entradas (a_0 a a_{2^n-1}) por uma árvore de mux-2 . Esta árvore tem altura de um mais a árvore de multiplexadores de 2^{n-1} entradas.

Um multiplexador de 2^n entradas é um circuito que apresenta em sua saída o valor da entrada a_k , quando o número representado pelos bits da entrada de seleção S é $k = \text{num}(S)$. Esta formulação útil é dada na Equação 4.7.

$$\begin{aligned}
S &: \mathbb{B}^n \\
A &: \mathbb{B}^{2^n} \\
\text{mux-2}^n &: (\mathbb{B}^n \times \mathbb{B}^{2^n}) \mapsto \mathbb{B} \\
\text{mux-2}^n(a_0 \cdots a_{2^n-1}, s_{n-1} \cdots s_0, z) &\equiv z = a_{\text{num}(S)}
\end{aligned} \tag{4.7}$$

Se, ao invés de usarmos mux-2 como elemento base na construção de multiplexadores, usarmos um mux-4 , então um mux-16 pode ser construído com 4 mux-4 no primeiro nível da árvore, e um mux-4 no segundo nível. Esta árvore tem altura 2 porque $\log_4 16 = 2$.

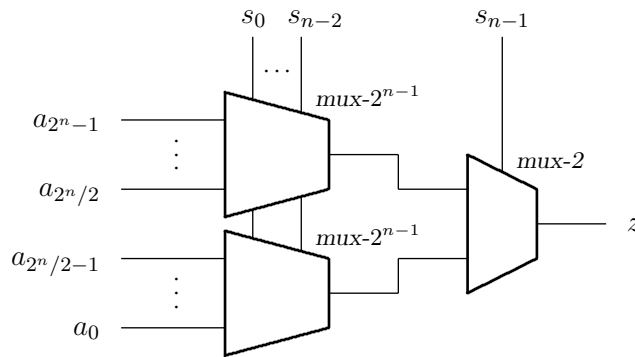


Figura 4.13: Multiplexador de 2^n entradas: um mux-2 e dois mux-2^{n-1} .

Exemplo 4.8 Considere a função de três variáveis lógicas $t(p, q, r)$ definida na Figura 4.14. A função t pode ser implementada trivialmente com um multiplexador 8 entradas, bastando ligar à entrada i do multiplexador o valor de t na linha da tabela verdade com índice i . A Figura 4.14 mostra a implementação de t com um mux-8 , que é a mera reprodução da tabela verdade. $\triangleleft$



Figura 4.14: Implementação de função de três variáveis com um mux-8 .

Exemplo 4.9 A função t do Exemplo 4.8 pode ser implementada com um multiplexador de 4 entradas. A tabela verdade de t é repetida na Figura 4.15, com as linhas agrupadas de duas em duas, para permitir a elisão da entrada r . A figura mostra a implementação com o mux-8 na qual r substitui alguns dos valores fixos em 0 ou 1 nas entradas, e a implementação com um mux-4 . A aplicação desta técnica, em geral, permite usar um multiplexador com “metade da altura”, mais um inversor. $\triangleleft$

A otimização da função t equivale a uma redução no tamanho da tabela que define a função, de uma tabela com oito linhas (t) para uma de quatro linhas (t'), sendo que em algumas linhas da imagem da função, esta contém a variável r ao invés de 1 ou 0. A função t' equivale a uma tabela de 4 linhas, com uma das variáveis da função “incluída na linha”. As linhas de t têm as seguintes correspondências em t' :

$$(p \wedge q) \Rightarrow (t' = 1), \quad (p \wedge \bar{q}) \Rightarrow (t' = r) \quad \text{e} \quad (\bar{p} \wedge q) \Rightarrow (t' = r).$$

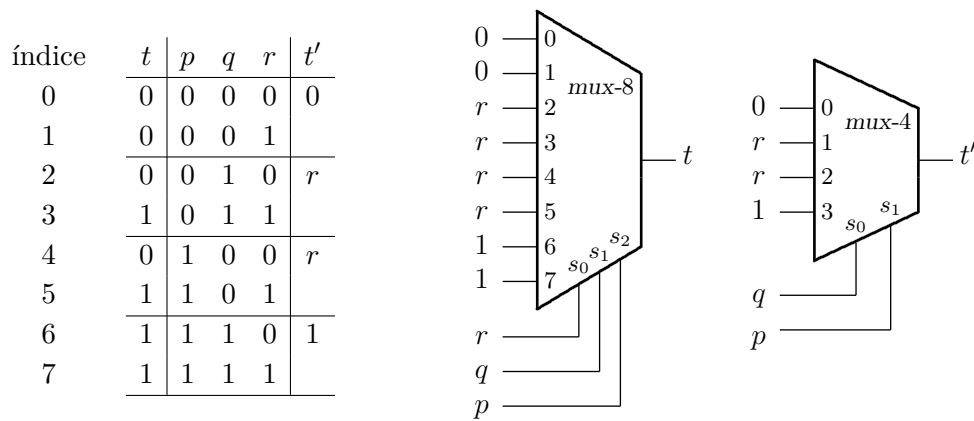


Figura 4.15: Implementação de função de três variáveis com um *mux-4*.

Exemplo 4.10 Vejamos como implementar a função $c(p, q, r, s)$ do Exercício 3.9:

$$c = (\neg p \wedge \neg q \wedge r) \vee \neg(s \wedge (p \vee r \vee \neg q))$$

A tabela verdade de c é mostrada na Figura 4.16, e a função c' resulta do agrupamento das linhas de duas em duas, para a elidir a entrada s . A função c pode ser implementada diretamente um multiplexador de 16 entradas com uma entrada associada a cada linha da tabela verdade. A figura mostra também a implementação com um *mux-8*, com $\bar{s}$ ligado nas entradas que lhe correspondem em c' . Um bom exercício para a aluna aplicada é a tentativa de implementar c'' com um *mux-4*. ◁

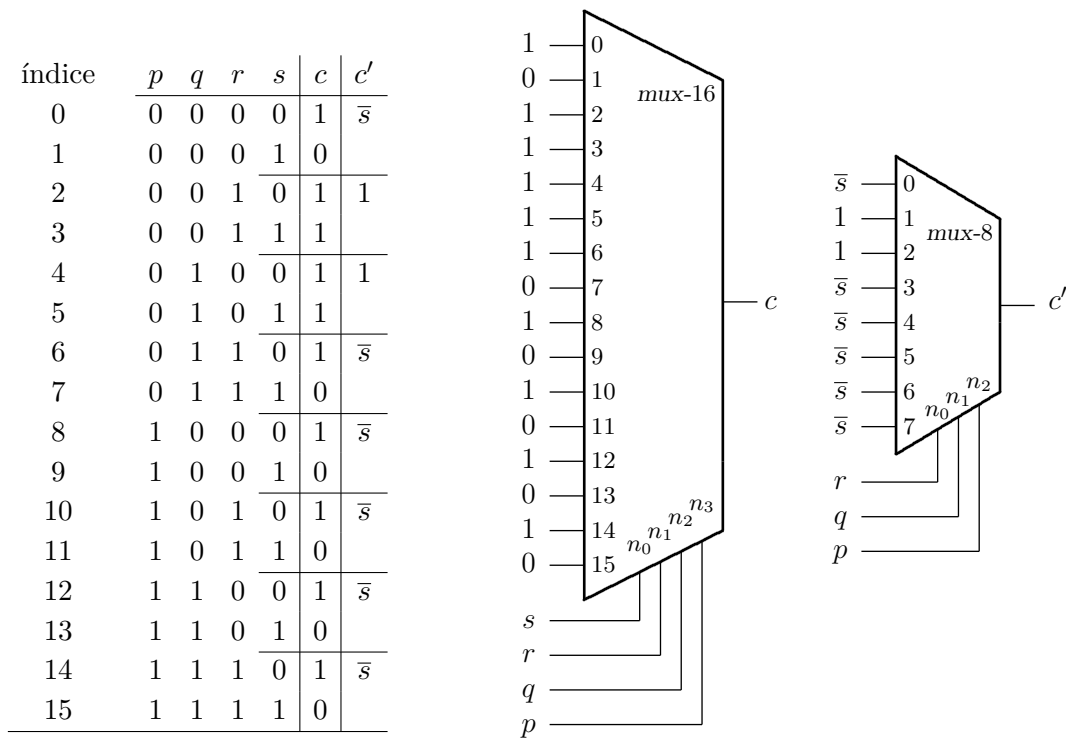


Figura 4.16: Implementação de função de quatro variáveis com um *mux-8*.

Exemplo 4.11 Para somar dois números representados em *notação científica*, ou em *ponto flutuante*, deve-se deslocar o expoente do menor número para ‘emparelhar’ as vírgulas e então efetuar a soma. Vejamos o projeto do circuito que descobre de quantas posições a parte fracionária deve ser deslocada.

Considere um circuito que tem como entrada um vetor de quatro bits $B = \langle b_3, b_2, b_1, b_0 \rangle$, e duas saídas: a saída V é 1 sempre que os quatro bits de B são zero, e a saída $N = \langle n_1, n_0 \rangle$ é o número (índice) da posição do bit mais à esquerda (mais significativo) que *não é zero* – o número N indica qual é a posição do ‘primeiro’ bit em 1.

A Tabela 4.6 mostra as combinações de entradas e saídas para a detecção do bit mais significativo de um quarteto de bits. Quando $B = \langle 0,0,0,0 \rangle$, então $V = 1$ e N pode ser ignorado porque não há nenhum bit em 1. Quando $V = 0$, N deve indicar qual é o bit mais significativo que não é zero. Se $b_k = 1$, então o valor dos $b_i, i < k$ é irrelevante, e isso é denotado com ‘X’ na tabela verdade.

Da tabela obtemos $n_1 = b_3 \vee b_2$, $n_0 = b_3 \vee (\overline{b_3} \wedge \overline{b_2} \wedge b_1)$ e $V = \overline{b_3} \wedge \overline{b_2} \wedge \overline{b_1} \wedge \overline{b_0}$. ◁

Tabela 4.6: Tabela verdade do bit mais significativo de um quarteto.

b_3	b_2	b_1	b_0	n_1	n_0	V
0	0	0	0	0	0	1
0	0	0	1	0	0	0
0	0	1	X	0	1	0
0	1	X	X	1	0	0
1	X	X	X	1	1	0

Exemplo 4.12 Vejamos como usar duas cópias do circuito do Exemplo 4.11 para projetar um circuito com uma entrada de 8 bits $C = \langle c_7, c_6, \dots, c_2, c_1, c_0 \rangle$, e duas saídas: a saída V é 1 sempre que os oito bits de C são zero, e a saída $M = \langle m_2, m_1, m_0 \rangle$ é o número (índice) da posição do bit mais à esquerda (mais significativo) que não é zero.

A Tabela 4.7 mostra as combinações de entradas e saídas para a detecção do bit mais significativo de um quarteto de bits. Da tabela observamos que se $V = v_1 \wedge v_0 = 1$ então M pode ser ignorado porque não há nenhum bit em 1.

Na metade superior da tabela, quando $v_1 = 1$, $m_2 = 0$ e $\langle m_1, m_0 \rangle$ podem ser obtidos com o circuito do Exemplo 4.11 aplicado aos bits $\langle c_3, c_2, c_1, c_0 \rangle$.

Na metade inferior da tabela (em itálico), quando $v_1 = 0$, $m_2 = 1$ e $\langle m_1, m_0 \rangle$ podem ser obtidos com o circuito do Ex. 4.11 aplicado aos bits $\langle c_7, c_6, c_5, c_4 \rangle$.

O bit v_1 pode ser usado para escolher um dentre dois pares de sinais. O circuito completo é mostrado na Figura 4.17. As funções são $m_2 = \overline{v_1}$, $m_1 = s_1 \triangleleft v_1 \triangleright r_1$, $m_0 = s_0 \triangleleft v_1 \triangleright r_0$ e $V = v_1 \wedge v_0$. ◁

Tabela 4.7: Tabela verdade do bit mais significativo de um octeto.

c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0	m_2	m_1	m_0	v_1	v_0
0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	1	X	0	0	1	1	0
0	0	0	0	0	1	X	X	0	1	0	1	0
0	0	0	0	1	X	X	X	0	1	1	1	0
0	0	0	1	X	X	X	X	1	0	0	0	X
0	0	1	X	X	X	X	X	1	0	1	0	X
0	1	X	X	X	X	X	X	1	1	0	0	X
1	X	X	X	X	X	X	X	1	1	1	0	X

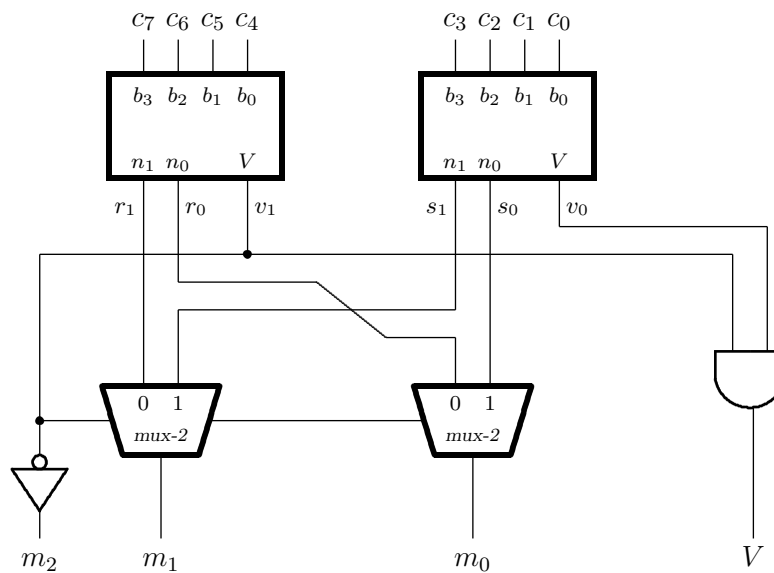


Figura 4.17: Circuito que identifica o bit mais significativo de um octeto.

4.3.2 Decodificador

Um *decodificador de duas saídas* é um circuito com uma entrada de controle s e duas saídas y_0, y_1 . A entrada de controle s permite escolher qual das duas saídas será ativada, conforme definido pela Equação 4.8 e na tabela verdade na Figura 4.18. A figura mostra também o circuito que implementa o decodificador de duas saídas e o seu símbolo.

$$\begin{aligned}
 s &: \mathbb{B} \\
 Y &: \mathbb{B}^2 \\
 \text{decod-2} &: \mathbb{B} \mapsto (\mathbb{B} \times \mathbb{B}) \\
 \text{decod-2}(s, y_0, y_1) &\equiv \begin{cases} y_0 = 0 \triangleleft s \triangleright 1 \\ y_1 = 1 \triangleleft s \triangleright 0 \end{cases}
 \end{aligned} \tag{4.8}$$

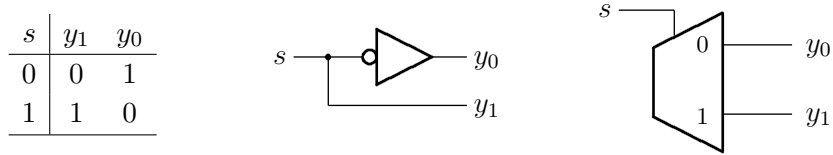


Figura 4.18: Decodificador de duas saídas.

No caso geral, um decodificador com n bits de entrada seleciona e ativa exatamente uma dentre as 2^n saídas. Posto de outra forma, este circuito *decodifica* o número codificado na entrada de controle: se $S = K$, a saída $\text{num}(K)$ é ativada, ficando todas as demais inativas. Um decodificador de 2^n saídas é especificado pela Equação 4.9.

$$\begin{aligned}
 S &: \mathbb{B}^n \\
 Y &: \mathbb{B}^{2^n} \\
 \text{decod-}2^n &: \mathbb{B} \mapsto (\Pi_n \mathbb{B}) \mapsto (\Pi_{2^n} \mathbb{B}) \\
 \text{decod-}2^n(s_{n-1} \cdots s_0, y_0 \cdots y_{2^n-1}) &\equiv y_i = 1 \triangleleft (\text{num}(S) = i) \triangleright 0
 \end{aligned} \tag{4.9}$$

A Figura 4.19 mostra a tabela verdade e o circuito de um decodificador de 4 saídas. A saída y_0 é ativada quando $S = \langle 0, 0 \rangle = 0$; a saída y_2 é ativada quando $S = \langle 1, 0 \rangle = 2$.

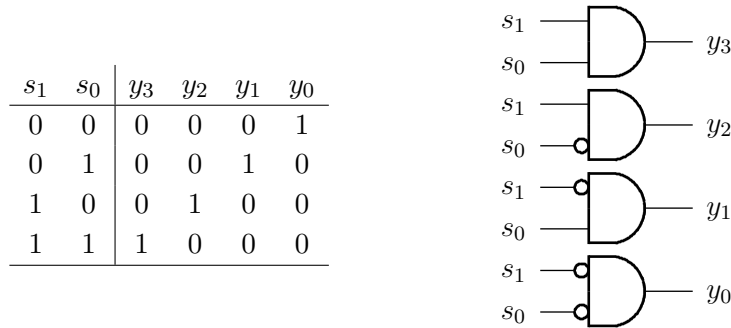


Figura 4.19: Decodificador de quatro saídas.

Decodificadores são também chamados de *seletores* porque, ao decodificar um dentre os n possíveis valores na entrada, este circuito seleciona uma das 2^n saídas.

Exemplo 4.13 Um decodificador pode ser usado para computar a *função inversa* de um multiplexador: a partir de uma entrada e , um sinal de controle com n bits direciona e para uma das 2^n saídas. A Figura 4.20 mostra um circuito que de-multiplexa, implementado com um seletor de duas saídas. $\triangleleft$

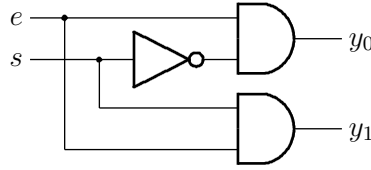


Figura 4.20: Circuito que computa a função inversa de um *mux-2*.

Exemplo 4.14 Considerando que os tempos de propagação das portas lógicas são aqueles indicados na Tabela 4.8, quais são os tempos de propagação do decodificador da Figura 4.18, e os decodificadores da Figura 4.21?

O tempo de propagação do *decod-2* é o de um inversor: $T_{\text{dec-2}} = 2\text{ns}$. O tempo de propagação do decodificador de 4 saídas é a soma dos tempos das portas no caminho mais longo, que é um inversor mais uma porta *and-2*: $T_{\text{dec-4}} = 2 + 4 = 6\text{ns}$. Com oito saídas, $T_{\text{dec-8}} = 2 + 6 = 8\text{ns}$. $\triangleleft$

Tabela 4.8: Tempo de propagação de portas lógicas.

porta	T_p	porta	T_p
<i>not</i>	2	<i>xor-2</i>	6
<i>and-2</i>	4	<i>or-2</i>	4
<i>and-3</i>	6	<i>or-3</i>	6
<i>and-4</i>	8	<i>or-4</i>	8
<i>and-5</i>	12	<i>or-5</i>	12
<i>and-6</i>	16	<i>or-6</i>	16

Tempos em nanosegundos.

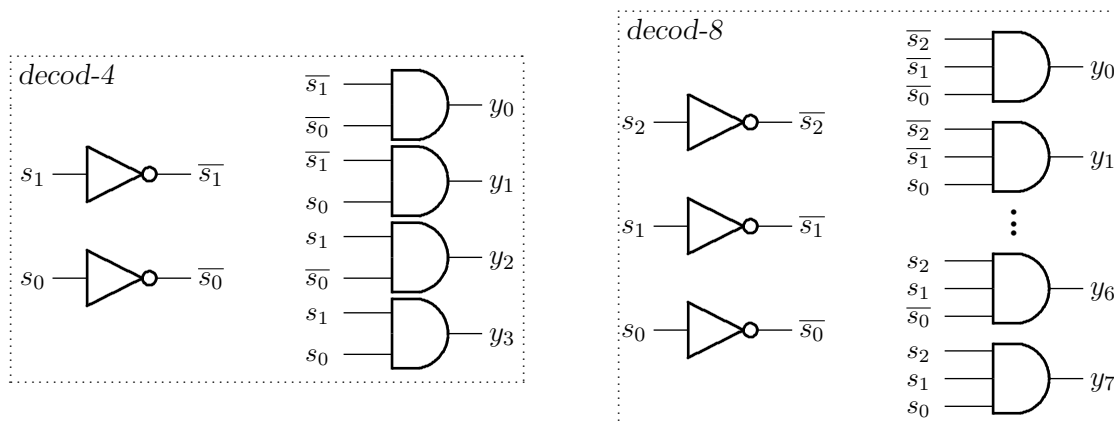


Figura 4.21: Decodificadores de quatro e de oito saídas.

Exemplo 4.15 Um decodificador, assim como um multiplexador, pode ser usado para implementar uma função definida por uma tabela verdade de maneira simples e eficiente e que é, em espírito,

similar àquela discutida na Seção 4.3.1. As saídas do decodificador geram os termos com os produtos da função, e uma porta *or* efetua a soma dos produtos. A Figura 4.22 mostra a implementação da função t com um decodificador e uma porta *or* de quatro entradas – a porta está desenhada com “asas de morcego” para acomodar todas as entradas.

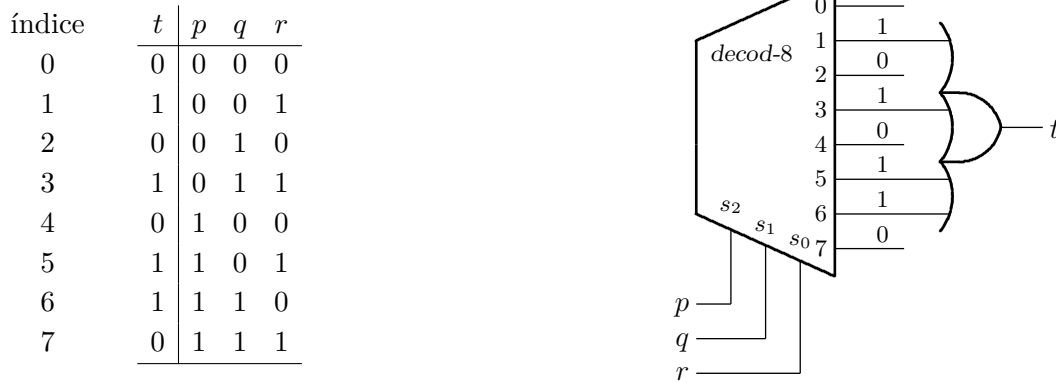


Figura 4.22: Implementação de função de três variáveis com um *decod-8*.

Exemplo 4.16 Considere um computador em que oito dispositivos disputam pela atenção do processador. Um dispositivo solicita a atenção do processador ao ativar um sinal de requisição de serviço. Quando o processador atende ao pedido do dispositivo, aquele remove o pedido de atenção.

O projetista do computador, conhecendo as especificidades de cada dispositivo escolhe e atribui distintas prioridades a cada dispositivo. O dispositivo mais importante – talvez porque produza/consuma o maior volume de dados – recebe a prioridade mais alta, que é 7 no nosso exemplo, e o dispositivo menos importante – talvez porque seja lento com relação aos demais – recebe a prioridade mais baixa, que é 0.

Para evitar que um dispositivo de baixa prioridade impeça o processador de atender a um dispositivo de prioridade mais alta, emprega-se um *decodificador de prioridades* que informa ao processador qual é o dispositivo de maior prioridade que está solicitando atenção num determinado instante. Se os dispositivos 5, 4 e 1 estão solicitando atenção, o decodificador indica que o pedido mais prioritário é o do dispositivo 5.

O decodificador tem oito entradas e quatro saídas. Às entradas $I = \langle i_7 \dots i_0 \rangle$ são ligados os sinais de requisição, e as três saídas $A = \langle a_2, a_1, a_0 \rangle$ são ligadas ao processador e indicam o nível de prioridade da requisição pendente. A quarta saída, p , indica se há alguma requisição ativa: $p \wedge (A = 0)$ indica que somente a requisição de nível 0 está ativa. A Figura 4.23 mostra o circuito que implementa as funções a_2 , a_1 e a_0 . A função p não é mostrada.

O bit a_2 fica em 1 sempre que uma dentre i_7, i_6, i_5, i_4 estiver ativa.

O bit a_1 fica em 1 sempre que i_7 ou i_6 estejam ativas, ou se ambas i_5 e i_4 estejam inativas e i_3 ou i_2 estejam ativas.

O bit a_0 fica em 1 sempre que i_7 ou i_5 estejam ativas e i_6 inativa, ou i_3 ativa com i_6 e i_4 inativas, ou i_1 ativa com i_6, i_4 e i_2 inativas.

A saída p é a disjunção das oito entradas.

A implementação mostrada neste exemplo é similar ao circuito integrado³ 74148.

◁

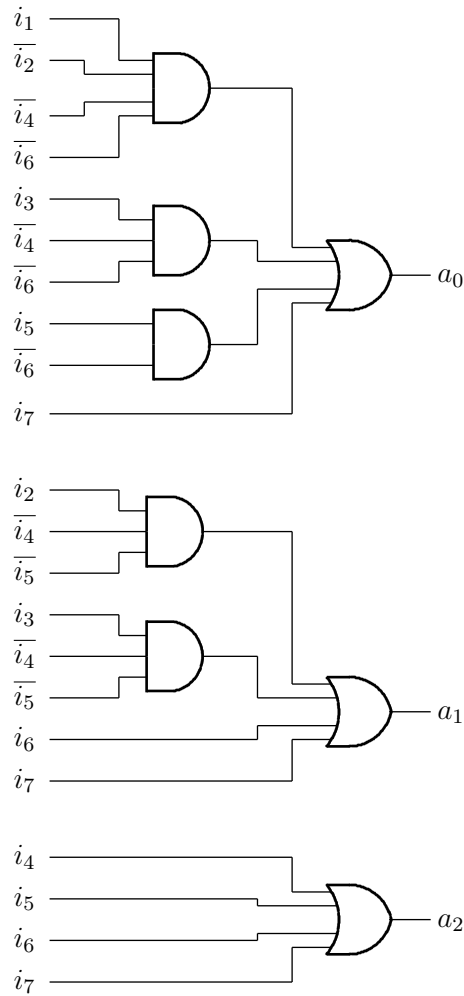


Figura 4.23: Decodificador de prioridades $8 \mapsto 3$.

4.3.3 Demultiplexador

Um *demultiplexador de duas saídas* é um circuito com uma entrada a , um sinal de controle s e duas saídas y_0, y_1 . A entrada de controle s determina em qual das duas saídas é apresentada a entrada, conforme define a Equação 4.10. A Figura 4.24 mostra o circuito e o símbolo do demultiplexador de duas saídas, que é aquele do Exemplo 4.13. Numa analogia aquática, o multiplexador se assemelha a um funil, enquanto que o demultiplexador lembra um regador.

$$\begin{aligned}
 &a, s : \mathbb{B} \\
 &Y : \mathbb{B}^2 \\
 &\text{demux-2} : \mathbb{B} \mapsto \mathbb{B} \mapsto (\mathbb{B} \times \mathbb{B}) \\
 &\text{demux-2}(a, s, y_0, y_1) \equiv \begin{cases} y_0 &= 0 \triangleleft s \triangleright a \\ y_1 &= a \triangleleft s \triangleright 0 \end{cases}
 \end{aligned} \tag{4.10}$$

³A especificação do 74148 pode ser obtida de <http://www.ti.com/product/SN74LS148>.

O *tipo* do demultiplexador é definido por suas duas entradas do tipo $\mathbb{B}$ ($\mathbb{B} \mapsto \mathbb{B}$) e pela saída que tem tipo par de $\mathbb{B}$ ($\mathbb{B} \times \mathbb{B}$).

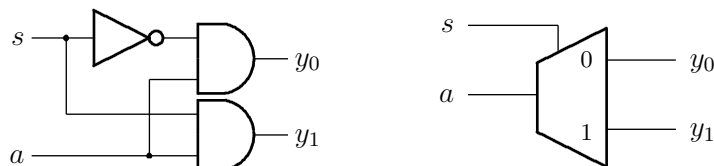


Figura 4.24: Demultiplexador de duas saídas.

Demultiplexadores com maior número de entradas podem ser construídos pela composição de demultiplexadores de duas saídas.

$$\begin{aligned}
a &: \mathbb{B} \\
S &: \mathbb{B}^2 \\
Y &: \mathbb{B}^4 \\
\text{demux-4} &: \mathbb{B} \mapsto (\mathbb{B} \times \mathbb{B}) \mapsto (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}) \\
\text{demux-4}(a, s_1, s_0, y_0, y_1, y_2, y_3) &\equiv \begin{cases} y_0 &= 0 \triangleleft s_1 \triangleright (0 \triangleleft s_0 \triangleright a) \\ y_1 &= 0 \triangleleft s_1 \triangleright (a \triangleleft s_0 \triangleright 0) \\ y_2 &= (0 \triangleleft s_0 \triangleright a) \triangleleft s_1 \triangleright 0 \\ y_3 &= (a \triangleleft s_0 \triangleright 0) \triangleleft s_1 \triangleright 0 \end{cases}
\end{aligned} \tag{4.11}$$

Como no caso dos multiplexadores, este processo pode ser levado adiante na construção de demultiplexadores de qualquer número de saídas. A Figura 4.25 mostra um *demux-8* construído com sete *demux-2*, pela extensão óbvia da Equação 4.11.

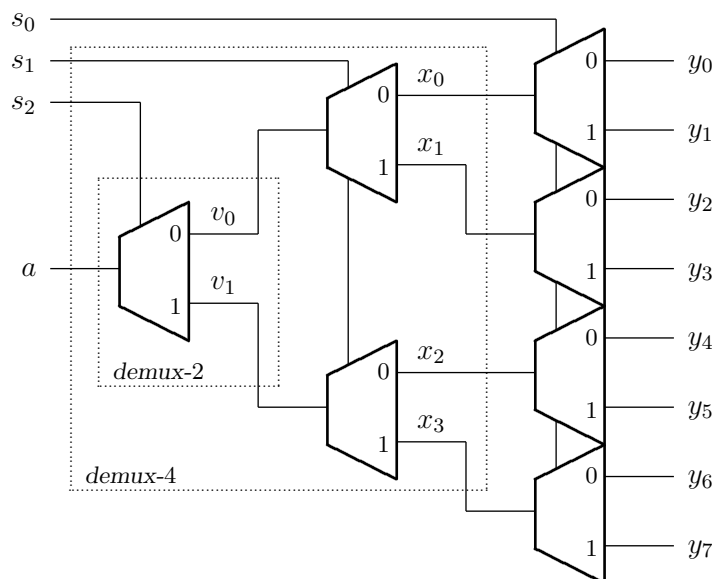


Figura 4.25: Demultiplexador de oito saídas.

As saídas do demultiplexador são determinadas pelo número binário representado pelas entradas de seleção. No comportamento definido na Equação 4.11, a saída y_2 apresenta o valor da entrada a quando $s_1s_0 = 2$. Assim, o comportamento do demultiplexador de 2^n saídas é aquele especificado pela Equação 4.12.

$$\begin{aligned}
a &: \mathbb{B} \\
S &: \mathbb{B}^n \\
Y &: \mathbb{B}^{2^n} \\
\text{demux-2}^n : \mathbb{B} &\mapsto (\Pi_n \mathbb{B}) \mapsto (\Pi_{2^n} \mathbb{B}) \\
\text{demux-2}^n(a, s_{n-1} \cdots s_0, y_0 \cdots y_{2^n-1}) &\equiv y_i = a \triangleleft (\text{num}(S) = i) \triangleright 0
\end{aligned} \tag{4.12}$$

Exemplo 4.17 Considerando que os tempos de propagação das portas lógicas são aqueles indicados na Tabela 4.8, quais são os tempos de propagação do demultiplexador da Figura 4.24, e o da Figura 4.25?

O tempo de propagação do *demux-2* é o de um inversor mais uma porta *and-2*: $T_{\text{dmx-2}} = 2 + 4 = 6\text{ns}$. Para o *demux-8*, o caminho mais longo atravessa um inversor no primeiro nível, mais uma porta *and-2* em cada um dos três níveis da árvore: $T_{\text{dmx-8}} = 2 + 4 + 4 + 4 = 14\text{ns}$. Os sinais s_1 e s_0 *não* estão no caminho crítico – se todas as entradas se alteram ao mesmo tempo, então as perturbações causadas pelos sinais s_1 e s_0 estabilizam *antes* que a influência de s_2 se propague até as portas *and-2* do segundo e terceiro níveis. $\triangleleft$

4.3.4 Aplicação: Circuitos de Memória

Vejamos duas aplicações importantes dos circuitos combinacionais básicos, que são as memórias ROM (*Read-Only Memory*), somente de leitura, e as memórias RAM (*Random Access Memory*), que permitem a leitura e a atualização de conteúdos. Aqui, veremos uma forma simples de organizar estas memórias; estes tópicos são aprofundados nas Seções 5.5 e 5.6.

Como vimos no Capítulo 1, uma memória somente de leitura é uma função cujos domínio e imagem são subconjuntos dos Naturais. Considere uma memória somente de leitura com capacidade para 256 bits; seu domínio é o intervalo $[0, 256)$, e sua imagem são 256 valores escolhidos do conjunto $\mathbb{B}$. Os valores desta função ROM são fixados na construção da memória e não podem ser alterados. Para obter o valor de um dos 256 bits, basta apresentar seu *endereço* à ROM e o valor endereçado é apresentado na saída de dados.

Uma memória RAM, por outro lado, nos permite alterar o conteúdo da função. Considere uma RAM com capacidade para armazenar 256 letras gregas. Se desejamos alterar o conteúdo da posição 120, que é π , para λ , basta apresentarmos o novo par $(120, \lambda)$ à RAM e ativar o sinal de atualização, porque então o novo valor sobre-escreverá o valor anterior. Seja R a função que descreve o conteúdo da RAM num dado instante

$$R = \{(0, \kappa), (1, \omega), (2, \alpha), (3, \sigma) \dots (120, \pi), \dots (254, \tau), (255, \theta)\}.$$

Para atualizar a posição 120 da RAM, é necessário subtrair ($\setminus$) o par $(120, \pi)$ e adicionar o par $(120, \lambda)$, o que resulta na função R'

$$R' = [R \setminus \{(120, \pi)\}] \cup \{(120, \lambda)\}.$$

Os circuitos que estudamos neste texto são fabricados na superfície de um circuito integrado, em dispostos em estruturas retangulares ou quadradas, para melhor aproveitamento da superfície. Uma memória de 256 bits, como a descrita acima, se implementada de forma simplória, resulta numa estrutura estreita e longa – um vetor – e geralmente este formato usa o espaço

bidimensional de maneira ineficiente. Se esta memória for implementada como uma matriz retangular, então o espaço pode ser melhor aproveitado. Para simplificar a utilização das memórias, a interface com o mundo externo deve ser a de um vetor, mesmo que sua implementação seja uma matriz quadrada ou retangular.

Considere um dispositivo de três terminais que é uma chave controlada, similar ao interruptor de luz da sala em que você se encontra. O terminal de controle determina a abertura (apaga-se a luz) ou o fechamento da chave (a luz acende). O símbolo para este dispositivo é mostrado na Figura 4.26. Quando o terminal de controle c está em 0, os terminais a e b estão desconectados (chave aberta), e quando c está em 1, então a chave fecha e os terminais a e b estão conectados.

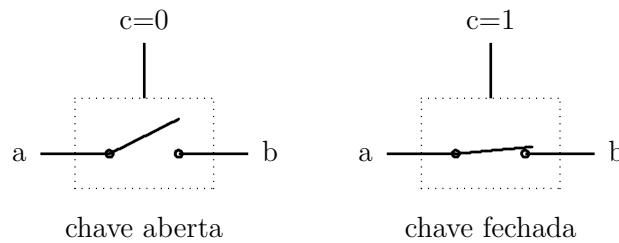


Figura 4.26: Símbolo para a chave digital.

Memória ROM

O conteúdo de uma memória ROM é definido durante a fabricação, e não pode ser alterado posteriormente. Vejamos como organizar uma ROM com 4 bits de altura e um de largura, identificada como uma ROM 4×1 . Empregamos um decodificador de quatro saídas para selecionar um dentre os quatro bits, e o bit selecionado é apresentado na saída da memória. O lado esquerdo da Figura 4.27 mostra o símbolo de uma memória ROM 4×1 , e o lado direito a sua implementação.

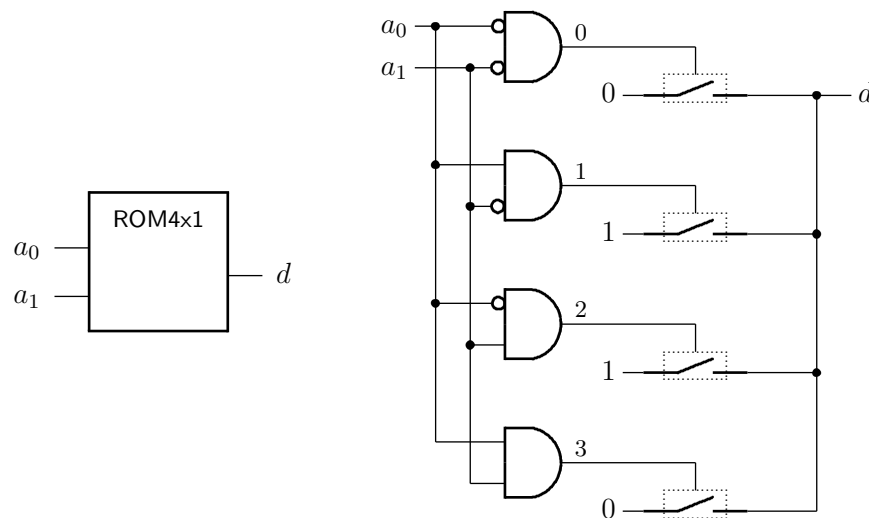


Figura 4.27: Símbolo e implementação de uma ROM 4×1 .

As duas *linhas de endereçamento* $\langle a_1, a_0 \rangle$ selecionam um dentre os quatro bits, que é exibido no sinal d . Esta memória ROM tem um comportamento puramente combinacional: decorrido

o tempo de propagação do circuito interno, o bit na saída d é aquele selecionado pelos sinais nas entradas de endereçamento.

O decodificador de duas entradas para quatro saídas escolhe uma das chaves, e o bit correspondente, 0 ou 1, é ligado à saída d . Esta ROM implementa a mesma função que uma porta *xor* de duas entradas. Os bits à esquerda das chaves são programados durante a fabricação e não podem ser alterados.

Para construir uma ROM 8×1 , basta acrescentar mais uma linha de endereçamento e duplicar as portas *and* no decodificador. Para construir uma ROM de oito bits, mas com três bits de largura (ROM 4×3) deve-se acrescentar duas colunas adicionais com chaves, como mostra a Figura 4.28. Quando uma das quatro linhas é endereçada por $\langle a_1, a_0 \rangle$, os três bits da linha são apresentados simultaneamente nas saídas $\langle d_2, d_1, d_0 \rangle$.

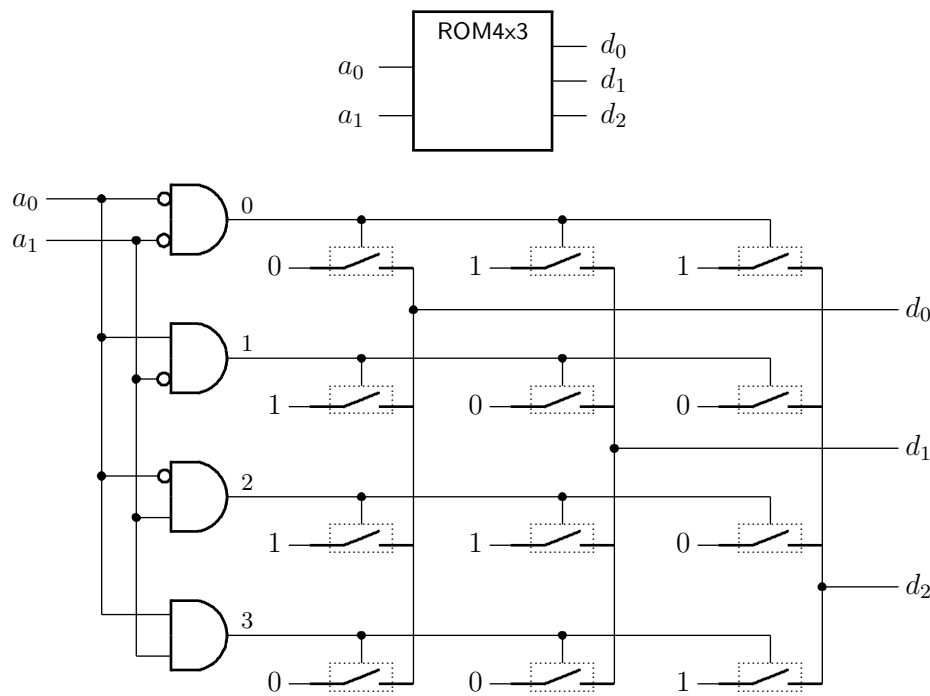


Figura 4.28: Circuito de uma ROM 4×3 .

Exemplo 4.18 Quais são as três funções de duas entradas (a_1, a_0) implementadas na ROM da Figura 4.28? As três funções (d_2, d_1, d_0) são indicadas na Tabela 4.9. $\triangleleft$

Tabela 4.9: Três funções de duas variáveis implementadas com uma ROM 4×3 .

índice	a_1	a_0	d_2	d_1	d_0
0	0	0	1	1	0
1	0	1	0	0	1
2	1	0	0	1	1
3	1	1	1	0	0

Normalmente, memórias têm de milhares a milhões de bits, e a organização em uma única linha vertical não é viável, por causa da geometria do componente, que fica longa e estreita. Por razões de construção é necessário construir circuitos mais parecidos com um quadrado, e por isso empregam-se várias linhas verticais. A Figura 4.29 mostra uma memória ROM com 16 bits, organizada como uma matriz de quatro linhas, com quatro bits em cada linha.

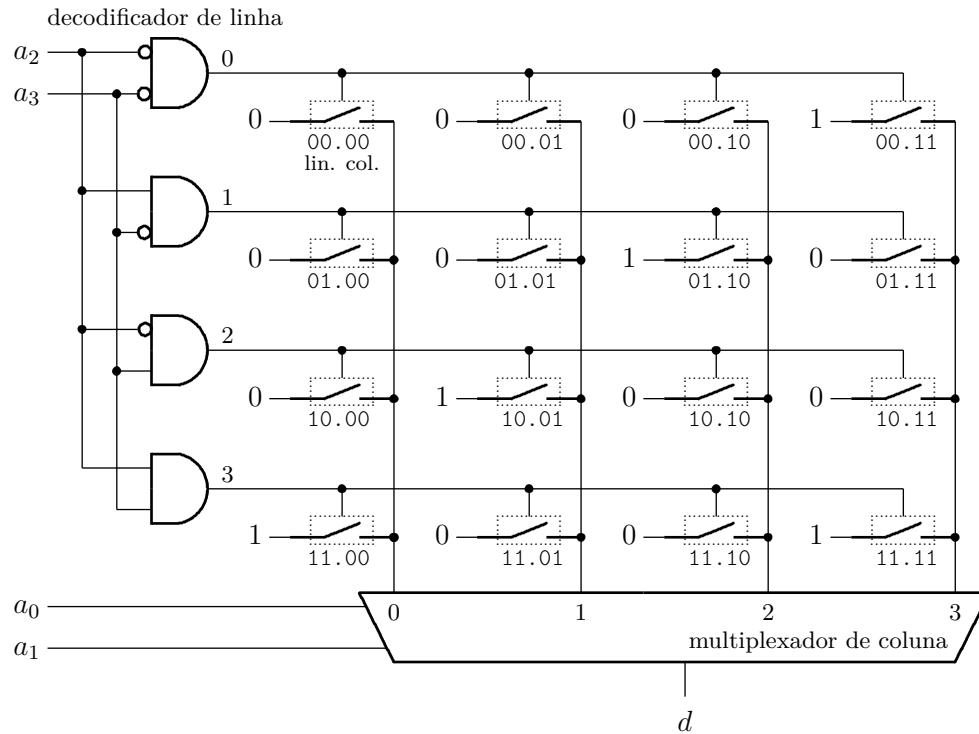


Figura 4.29: Circuito de uma ROM 16×1.

A Figura 4.29 mostra uma memória ROM com um bit de largura e capacidade de 16 bits. Internamente, ela é organizada como uma matriz quadrada, com 4 palavras, de 4 bits cada palavra. O *decodificador de linha* seleciona uma dentre as quatro linhas, ou palavras, enquanto que o *multiplexador de coluna* seleciona um dos quatro bits da palavra. Ao custo adicional de um multiplexador, é possível construir memórias com matrizes de 1024×1024 bits obtendo-se uma geometria razoável.

Há uma diferença importante entre a organização interna da memória – matriz de N linhas por M colunas – e sua interface externa – memória com P palavras de B bits de largura. A organização interna é otimizada para obter-se a maior capacidade possível, e ao mesmo tempo reduzir o tempo de acesso, enquanto que a interface externa, que é essencialmente a largura da porta de dados, depende de restrições comerciais – as larguras usuais são 8, 16 ou 32 bits.

Memória RAM

Uma memória RAM permite que seu conteúdo seja alterado pelo circuito do qual ela faz parte. Para tanto, circuitos de RAM possuem sinais adicionais para controlar a atualização dos conteúdos. Como um mínimo, é necessário um sinal de escrita, tipicamente chamado de *write* (wr), que quando ativado, atualiza os conteúdos da memória. A Figura 4.30 mostra o diagrama de blocos de uma memória RAM simples, com quatro linhas de um bit em cada linha.

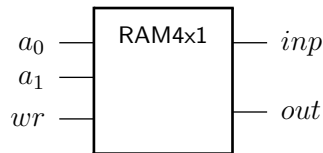


Figura 4.30: Símbolo de uma RAM 4×1.

Ao contrário da ROM, na qual o valor de cada bit é determinado na fabricação, um bit de RAM deve ser um circuito que mantém seu valor inalterado até que ocorra a próxima atualização. Este circuito é mostrado na Figura 4.31. O terminal wr , quando em 1, permite que o sinal de entrada (inp) seja aplicado à entrada do laço. O sinal out permite observar o valor armazenado no bit. Este circuito é chamado de *célula de RAM*, ou célula de memória.

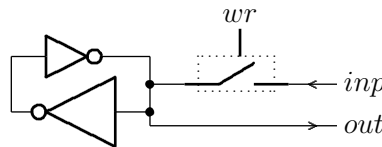


Figura 4.31: Circuito que armazena um bit de memória RAM.

Este circuito *viola* a definição de circuito combinacional por causa do laço de realimentação – há um ciclo que liga a saída do circuito a sua entrada. Suponha que $out = 0$ e $wr = 0$; então saída do inversor maior é 1, a saída do inversor menor é 0, o que mantém a saída do inversor maior em 1.

É justamente o laço de realimentação – a saída de um inversor ligado na entrada do outro – que faz com que este circuito armazene um bit. Enquanto a alimentação dos inversores for mantida, este laço mantém o último valor gravado na célula.

Quando $wr = 1$, o sinal apresentado em inp se sobrepõe à saída do inversor pequeno, e altera a entrada do inversor maior, o que altera a saída do inversor menor, assim fechando o laço de realimentação. Os dois inversores são construídos de tal forma que o sinal produzido pelo inversor pequeno é “mais fraco” do que os sinais apresentados na saída do inversor grande e em inp . Voltaremos ao tema de “sinais fortes” e “sinais fracos” no Capítulo 5.

Quando uma linha é selecionada por $\langle a_1, a_0 \rangle$, se $wr = 0$ então o valor do bit selecionado é exibido em out . Se $wr = 1$ então a célula selecionada por $\langle a_1, a_0 \rangle$ é atualizada com o (novo) valor apresentado em inp , e assim permanece até a próxima escrita. O amplificador ligado à out isola, eletricamente, o circuito interno da RAM dos circuitos externos.

A Figura 4.32 mostra a interface e o circuito de uma (RAM 4 × 1). Uma RAM com 4 bytes (RAM 4 × 8) pode ser facilmente construída ao replicar mais sete vezes a coluna das células e a linha $inp-out$. Uma memória com capacidade para 64 bytes pode ser construída com um seletor de 6 para 64 linhas, e esta RAM 64 × 8 é acessada com seis linhas de endereço.

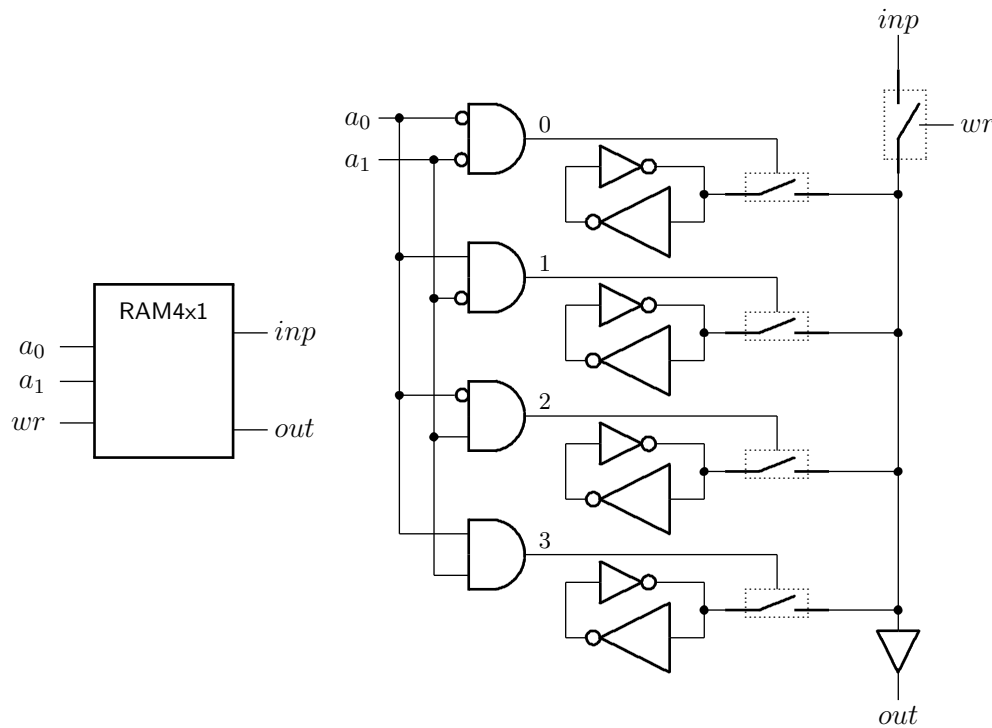


Figura 4.32: Circuito de uma RAM 4×1.

Exemplo 4.19 Vejamos como projetar um decodificador de 6 endereços para 64 linhas. Uma possibilidade é estender os circuitos do Exemplo 4.14 e empregar 64 portas *and* de 6 entradas e seis inversores. Este circuito pode ser considerado uma árvore larga e baixa.

No outro extremo temos uma árvore alta e estreita, similar ao demultiplexador da Figura 4.25, com um *decod-2* ao invés de um *demux-2* no primeiro nível. Considera-se esta árvore com altura 6 porque $\log_2 64 = 6$.

Ainda outra implementação é possível com componentes de 4 saídas, como o circuito da Figura 4.33. Esta é uma árvore de altura 3 porque $\log_4 64 = 3$.

Qual destas implementações é *a melhor*? A resposta clássica é *depende*. O Exercício 4.12 pede uma comparação dos tempos de propagação das três versões propostas aqui. Outra questão relacionada ao tempo de propagação é a influência do número de entradas conectadas à saída de uma porta lógica, ou o *fan-out* de cada porta lógica no seletor. Este problema é investigado na Seção 5.4. ◁

Exercícios

Ex. 4.1 Prove que $(a \vee b) \wedge \overline{(a \wedge b)} = (a \wedge \bar{b}) \vee (\bar{a} \wedge b)$.

Ex. 4.2 Enuncie a lei de formação de um *mux-N* a partir de *mux-(N-1)*.

Ex. 4.3 Enuncie a lei de formação de um *demux-N* a partir de *demux-(N-1)*.

Ex. 4.4 Enuncie a lei de formação de um *decod-N* a partir de *decod-(N-1)*.

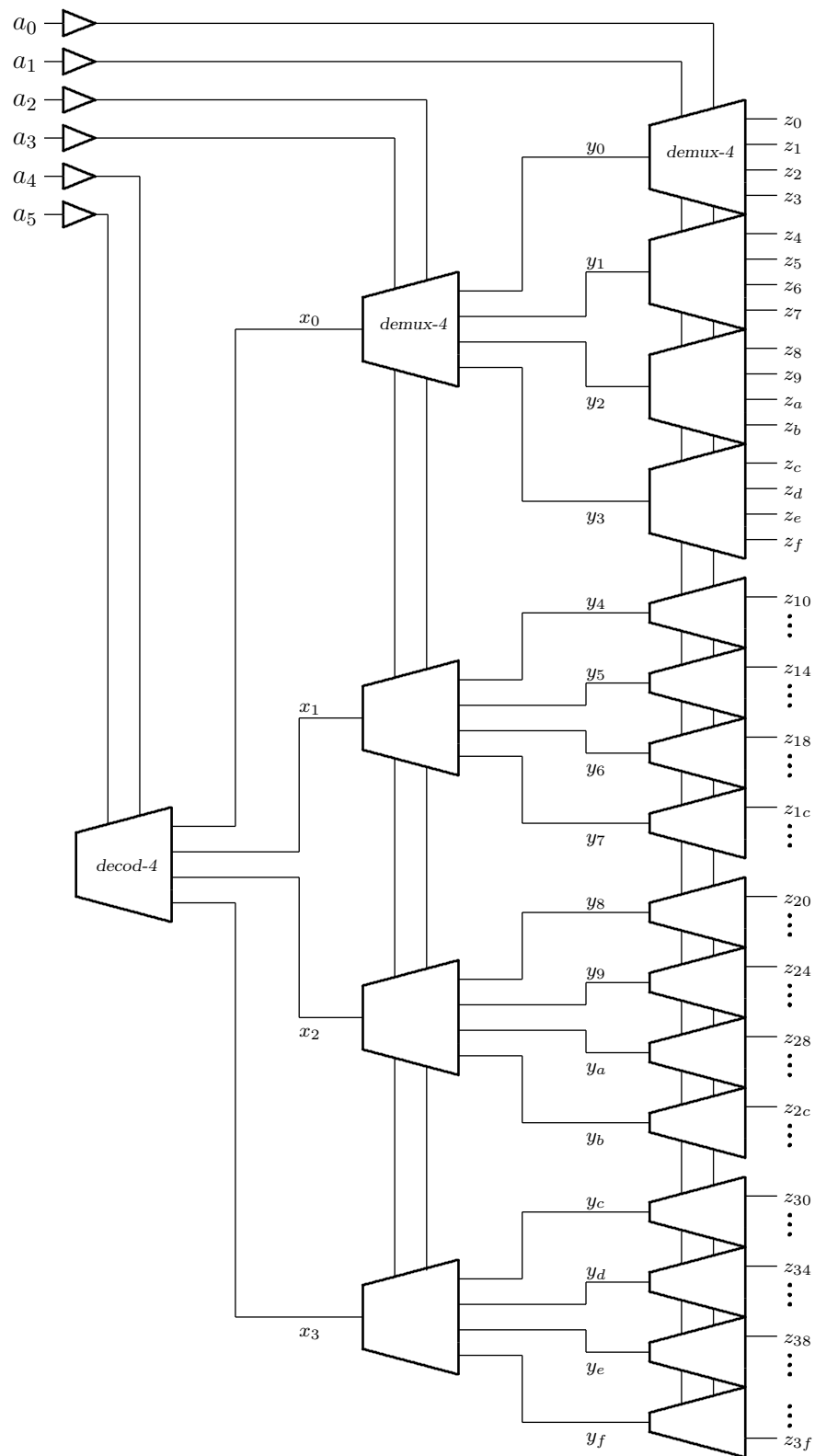


Figura 4.33: Decodificador 6 entradas para 64 saídas.

Ex. 4.5 Mostre como implementar um circuito *demux-2* com 4 portas *nor*.

Ex. 4.6 Prove que um *mux-2* implementa as funções lógicas $\wedge$, $\vee$, $\neg$ e $\oplus$. Por que isso poderia ser relevante?

Ex. 4.7 Com base na definição do multiplexador, mostre que um *mux-N* pode-se implementar qualquer função lógica de $\log_2 N$ variáveis.

Ex. 4.8 Com base na definição do multiplexador, mostre que com um *mux-N* e inversores, pode-se implementar qualquer função de $\log_2(N + 1)$ variáveis.

Ex. 4.9 Projete e implemente um comparador de magnitude ($m = A > B$) para números de dois bits. Escreva a tabela verdade e simplifique a função m com um Mapa de Karnaugh.

Ex. 4.10 Estenda a solução do Ex. 4.9 para números de três bits. *Pista*: aproveite a solução do Ex. 4.9; se os bits a_2 e b_2 são iguais, o que determina se $A > B$? Se os bits a_2 e b_2 são diferentes, o que determina se $A > B$?

Ex. 4.11 Estenda a solução do Ex. 4.10 para números de quatro bits.

Ex. 4.12 Calcule o tempo de propagação dos três seletores de 64 saídas propostos no Exemplo 4.19. Empregue os tempos de propagação da Tabela 4.8.

Ex. 4.13 Repita o Ex. 4.12 levando em conta a *carga* ligada em cada porta. A carga é o número de entradas alimentadas pela saída da porta lógica. Empregue os tempos de propagação da Tabela 4.8 e considere que o tempo de propagação de uma porta piora em 10% para cada entrada que ela alimenta além da primeira. Para cada ligação adicional, além da primeira, o tempo de propagação aumenta em 10%. Considere: (i) uma taxa composta, se uma porta κ alimenta 4 saídas, então seu tempo de propagação é $T_\kappa \times (1, 1)^3$; (ii) que cada linha de endereço a_i é gerada por uma porta lógica (um amplificador ou *buffer*), e (iii) que cada saída z_i alimenta um inversor. Compare a carga nas linhas a_0 e a_5 .

Ex. 4.14 Mostre como um decodificador de 2^n saídas pode ser construído com um decodificador de duas saídas e dois demultiplexadores de 2^{n-1} saídas.

Ex. 4.15 Uma possível implementação para multiplexadores de 2^n entradas consiste em usar um decodificador de n para 2^n , 2^n portas *and* de 2 entradas, e uma porta *or* de 2^n entradas. O sinal de entrada é ligado a todas as portas *and*, cada saída do decodificador é ligada a outra entrada da porta *and*. As saídas de todas as *and* são ligadas às respectivas 2^n entradas da porta *or*. As n entradas de seleção são ligadas às n entradas do decodificador. Desenhe o circuito de multiplexador de oito entradas implementado como descrito aqui.

Ex. 4.16 Mostre como implementar uma memória ROM 64×1 : 64 palavras de 1 bit.

Ex. 4.17 Mostre como implementar uma memória RAM 64×1 : 64 palavras de 1 bit. Lembre da atualização.

Ex. 4.18 Mostre como implementar uma memória ROM 64×4 : 64 palavras de 4 bits.

Ex. 4.19 Mostre como implementar uma memória RAM 64×1 : 64 palavras de 4 bits. Lembre da atualização.

Índice Remissivo

Símbolos

T_A , 64
 $\%$, 18–20
 $\Rightarrow$, 37
 $\bigvee$, 47
 $\bigwedge$, 47
 $\equiv$, 36
 $\wedge$, 30, 36
 $\triangleleft \triangleright$, 36, 42, 66
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 41, 44
 $\neg$, 30, 36
 $\vee$, 30, 36
 $\oplus$, 36, 53, 65
 $\mapsto$, 42
 $\bar{a}$, *veja* $\neg$
 π , 25
 $\backslash$, 77
decod, 72
demux, 75
 e , 24
mux, 67
 $\mathbb{B}$, 29–33, 38, 42
 $\mathbb{Z}$, 42
 $\mathbb{N}$, 42, 43
num, 44, 55, 72, 77
 num^{-1} , 55
 $\mathbb{R}$, 42
 $\langle \rangle$, 29, 42, 43

Números

74148, 75

A

abstração, 27
 bits como sinais, 27–33, 57
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo,
 conversão de base, 18
 conversão de base de frações, 22
 and, *veja* $\wedge$
 aproximação, 24
 árvore, 64
assembly, *veja* ling. de montagem
 associatividade, 31
 atraso, *veja* tempo de propagação, 57
 atribuição, 12

B

binário, 20
 bit, 20
 bits, 27–37, 65
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 variável, 30
branch, *veja* desvio condicional
 byte, 11

C

cadeia,
 de portas, 64
capture FF, *veja flip flop*, destino
 célula de RAM, 81
 chave,
 normalmente fechada, 78
 ciclo,
 combinacional, 57
 violação, 81
 circuito,
 combinacional, 57, 65
 circuito aberto, 57
 clk, *veja* relógio
 clock, *veja* relógio
 clock skew, *veja skew*
 CMOS, 59, 65
 código,
 Gray, 63
Column Address Strobe, *veja* CAS
 combinacional,
 ciclo, 57
 circuito, 57
 dispositivo, 57
 comparador,
 de igualdade, 62
Complementary Metal-Oxide Semiconductor, *veja*
 CMOS
 complemento, *veja* $\neg$
 complemento, propriedade, 31
 comportamento transitório, *veja* transitório
 comutatividade, 31
 condicional, *veja* $\triangleleft \triangleright$
 conjunção, *veja* $\wedge$
 conjunto mínimo de operadores, 65
 contra-positiva, 41
 conversão de base, 18

curto-circuito, 57

D

datapath, veja circuito de dados
 decimal, 17
 decodificador, 72, 78, 81–82, 84
 de prioridades, 74
delay, 57
 demultiplexador, 75
design unit, veja VHDL, unidade de projeto
 disjunção, veja $\vee$
 dispositivo,
 combinacional, 57
 distributividade, 31, 34, 51
 divisão inteira, 44
don't care, 70
 dual, 32
 dualidade, 32

E

endereço, 77
 enviesado, relógio, veja *skew*
 equivalência, veja $\Leftrightarrow$
 erro,
 de representação, 23
 especificação, 42
 expressões, 36

F

fan-out, 82, 84
 fechamento, 31
Field Effect Transistor, veja FET
Field Programmable Gate Array, veja FPGA
flip-flop,
 modelo VHDL, veja VHDL, *flip-flop*
 um por estado, veja um FF por estado
 forma canônica, 48
 frações, veja ponto fixo
 frequência máxima, veja relógio
 função, 30
 tipo, 29, 42
 função, aplicação bit a bit, 32
 função, tipo (op. infixo), veja $\mapsto$

G

glitch, veja transitório
 gramática, 17

H

hexadecimal, 19

I

idempotência, 31
 identidade, 31
 igualdade, 30
 implementação, 42
 implicação, veja $\Rightarrow$
 informação, 16
 Instrução,
 busca, veja busca

instrução, 12

 busca, veja busca
 decodificação, veja decodificação
 execução, veja execução
 resultado, veja resultado

interface,
 de rede, 13
 de vídeo, 12
 involução, 31, 61

J

jump, veja salto incondicional

L

latch, veja basculado
latch FF, veja *flip flop*, destino
launch FF, veja *flip flop*, fonte
 linguagem,
 assembly, veja ling. de montagem
 C, veja C
 Pascal, veja Pascal
 VHDL, veja VHDL
 Z, 27
 linha de endereçamento, 78
 literal, 38
 logaritmo, 43

M

Mapa de Karnaugh, 49
 Máquina de Mealy, veja máq. de estados
 Máquina de Moore, veja máq. de estados
 máscara, 32
 máximo e mínimo, 31
 maxtermo, 46
 Mealy, veja máq. de estados
 memória,
 atualização, 77
 de vídeo, 13
 decodificador de linha, 80
 endereço, 77
 multiplexador de coluna, 80
 primária, 13
 RAM, 81
 ROM, 78
 secundária, 13
 memória dinâmica, veja DRAM
 memória estática, veja SRAM
 mintermo, 45
 modelo,
 funcional, 44
 módulo, veja %, *mod*
 Moore, veja máq. de estados
 multiplexador, 61, 66–70, 80
multiply-add, veja MADD

N

número,
 de Euler, 24
 negação, veja $\neg$
 nível lógico,

0 e 1, 28
 indeterminado, 28
non sequitur, 37
 not, *veja* $\neg$
 número primo, 53

O

octal, 18
 operação,
 binária, 29
 bit a bit, 32
 infixada, 42
 prefixada, 47
 unária, 29
 operações sobre bits, 29–33
 operador,
 binário, 29
 lógico, 36
 unário, 29
operation code, *veja* *opcode*
 or, *veja* $\vee$
 ou exclusivo, *veja* $\oplus$
 ou inclusivo, *veja* $\vee$

P

paridade,
 ímpar, 49
 par, 49
 período mínimo, *veja* relógio
pipelining, *veja* segmentação
 piso, *veja* [v]
 ponto flutuante, 70
 porta lógica, 65
 and, 59
 carga, *veja* *fan-out*
 nand, 60
 nor, 60
 not, 59
 or, 59
 xor, 60, 65
 potenciação, 43
 precedência, 30
 precisão,
 representação, 23
 prioridade,
 decodificador, 74
 processador, 12
 produtivo, 33
 produto de somas, 46
 programa de testes, *veja* VHDL, *testbench*
 propriedades, operações em $\mathbb{B}$, 31
 prova de equivalência, 40–41
 pulso,
 espúrio, *veja* transitório

R

RAM, 12, 77, 81
 célula, 81
Random Access Memory, *veja* RAM
Read Only Memory, *veja* ROM

realimentação, 81
 redução, 33
Register Transfer Language, *veja* RTL
 registrador de deslocamento,
 modelo VHDL, *veja* VHDL, registrador
 relógio,
 enviesado, *veja* *skew*
 representação,
 abstrata, 28
 binária, 20
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 posicional, 17
 precisão, 23
 ROM, 12, 77–80
Row Address Strobe, *veja* RAS

S

seletor, 72
 semântica, 17
 silogismo, 37
 simplificação de expressões, 38–40
 sinal, 27, 42
 analógico, 27
 digital, 27, 28
 síntese, *veja* VHDL, síntese
Solid State Disk, *veja* SSD
 soma, *veja* somador
 soma de produtos, 45, 51
 completa, 45
 somatório, 33
 spice, 28
 SSD, 14

T

tabela verdade, 33–35, 45
 tamanho, *veja* $|N|$
 tempo,
 de propagação, 57–58, 61, 64–65, 73, 77, 84
 Teorema,
 Absorção, 49
 DeMorgan, 32, 41, 48, 54, 60, 61
 Simplificação, 49
testbench, *veja* VHDL, *testbench*
 teto, *veja* [r]
three-state, *veja* terceiro estado
 tipo,
 de sinal, 42
 função, 29
 Tipo I, *veja* formato
 Tipo J, *veja* formato
 Tipo R, *veja* formato
 transferência entre registradores, *veja* RTL
Transistor-Transistor Logic, *veja* TTL
transmission gate, *veja* porta de transmissão
 TTL,
 74148, 75

tupla, *veja* $\langle \rangle$
 elemento, 32
 largura, 43

U

Unidade de Lógica e Aritmética, *veja* ULA

V

valor da função, 30
vetor de bits, *veja* $\langle \rangle$, 32
 largura, 43
VHDL,
 design unit, *veja* VHDL, unidade de projeto
 tipos, 42

W

write back, *veja* resultado

X

xor, *veja* $\oplus$

Z

Z, linguagem, 27

Capítulo 5

A Tecnologia de Circuitos CMOS

*Everything should be made as simple as possible,
but no simpler.*

Albert Einstein.

Este capítulo¹ contém uma breve introdução à tecnologia de circuitos digitais CMOS (pronuncia-se “cê-mós”), sigla para *Complementary Metal-Oxide Semiconductor*, que é usada na implementação de circuitos digitais. O método de fabricação e o comportamento dos dois dispositivos básicos da tecnologia – transistores do tipo P e do tipo N – são introduzidos na Seção 5.1. Com estes dispositivos podem ser implementados o inversor e as portas lógicas *nand* e *nor*, bem como circuitos com um terceiro “nível lógico” para além dos níveis 0 e 1. Tais circuitos são estudados na Seção 5.2. A Seção 5.3 introduz o importante tópico “tempo de propagação de sinais” e discute algumas das razões para os atrasos nos sinais introduzidos pelas portas lógicas.

Para mais detalhes veja Weste & Harris [WH10], Rabaey [RCN03], ou Sedra & Smith [SS90], que são textos com ênfase em Eletrônica. A Seção 5.1 é baseada no texto de Clark [Cla80]; as definições de *tempo de propagação* e *tempo de contaminação*, na Seção 5.4 são baseadas em material da disciplina 6.004 – *Computation Structures* do MIT, de 2013.

5.1 Semicondutores e Transistores CMOS

Um *condutor* sólido é tipicamente um metal, no qual os elétrons da camada de valência são fracamente atraídos pelo núcleo e se movem livremente entre átomos vizinhos. A cada elétron corresponde um próton, e portanto um condutor em estado natural é eletricamente neutro. Um *isolante* sólido é um material no qual os elétrons da camada de valência estão firmemente ligados ao núcleo. No que se segue, ignoramos a condução em fluídos (íons na água ou ar) ou plasma (centelhas), e isolantes líquidos ou gasosos.

Materiais *semicondutores* diferem dos condutores e dos isolantes porque em algumas circunstâncias se comportam como isolantes e em outras como condutores. Estes materiais são empregados na construção de circuitos digitais para implementar ‘chaves’ que podem ser controladas por sinais digitais. Tais chaves tem dois estados, ou estão fechadas – conduzindo corrente elétrica – ou estão abertas – impedindo a passagem de corrente.

¹© Roberto André Hexsel, 2012-2020. Versão de 18 de novembro de 2020.

O que segue é uma descrição muito simplificada da fabricação e do funcionamento de circuitos CMOS.

5.1.1 Materiais e Fabricação

O material mais frequentemente empregado na fabricação de circuitos CMOS é o silício (Si). Um bastão de silício, tipicamente com 20 a 30cm de diâmetro e um a dois metros de comprimento, é serrado em discos com 0,3 a 1 mm de espessura. O bastão, que é um monocristal de silício, é fabricado com elevadíssimo nível de pureza. O disco de silício é polido até que na sua superfície ocorra menos do que um defeito por centímetro quadrado. Por ‘defeito’ entende-se uma cavidade na qual faltam poucas dezenas de átomos, ou a existência, no monocristal, de átomos que não sejam de silício.

Durante a fabricação de circuitos integrados são acrescentados impurezas ao substrato de silício que agregam elétrons ao cristal – tornando-o eletricamente negativo – ou que subtraem elétrons – tornando-o eletricamente positivo. Estas impurezas, chamadas de *dopantes*, são incorporadas ao retículo cristalino pela exposição, em alta temperatura, a uma elevada concentração do dopante. Este processo é conhecido como *dopagem por difusão*.

A superfície do disco é coberta com um polímero, com certas áreas mantidas sem a cobertura. Uma ‘máscara’ é usada para definir quais áreas são cobertas e quais são expostas. Tais máscaras são semelhantes ao negativo de uma fotografia em preto e branco; as partes claras na máscara são expostas, e as partes escuras são encobertas durante o processo de difusão.

O disco é levado a um forno com temperatura da ordem de 800 a 1000 C, no qual é insuflado gás de fósforo (P), arsênio (As), ou de boro (B). Átomos destes elementos se difundem através do cristal de silício e se alojam na estrutura cristalina. Após a difusão, a concentração de dopantes nas áreas expostas pela máscara é da ordem de 1 átomo de dopante para 10^7 átomos de silício.

Um transistor CMOS é produzido por uma sequência de passos de fabricação. Tipicamente um passo se inicia com a exposição do polímero para formar a máscara que é usada para selecionar as regiões do disco que serão expostas, seguido de difusão com o tipo de dopante e a concentração adequados, e concluído pela remoção da máscara. Dependendo do processo de fabricação, são necessários de 100 a 400 passos de processamento, o que toma 2 a 3 semanas.

O átomo de silício possui 4 elétrons na sua camada de valência. Num retículo cristalino composto somente de silício, cada átomo se liga a exatamente quatro vizinhos e o cristal tem carga elétrica líquida igual a zero porque todos os elétrons das camadas de valência estão ligados aos seus átomos.

O átomo de fósforo possui 5 elétrons na sua camada de valência, e quando se liga a quatro átomos no retículo cristalino de silício, o quinto elétron fica fracamente ligado ao átomo de fósforo. Cada átomo de fósforo contribui com um elétron adicional ao material e a região dopada com fósforo é chamada de *semicondutor tipo N* porque a região possui um excesso de portadores de carga negativa. Fósforo é chamado de *doador* porque acrescenta elétrons ao material.

O átomo de boro possui 3 elétrons na sua camada de valência, e quando se liga ao retículo cristalino, uma das quatro ligações fica faltando. Esta ligação resulta no *buraco* deixado pelo elétron faltante, e o buraco é portanto um portador de carga positiva. Boro é chamado de *receptor* porque cada átomo de boro ‘aceita’ um elétron, contribuindo com uma carga positiva

ao material. A região dopada com boro é chamada de *semicondutor de tipo P* porque possui excesso de portadores de carga positiva.

Os elétrons da camada de valência de um material condutor tem alta mobilidade porque são fracamente ligados aos átomos, enquanto que num isolante os elétrons são fortemente ligados aos seus átomos e sua mobilidade é pequena. Num material semicondutor, a mobilidade dos elétrons ou dos buracos depende da concentração dos dopantes, e da aplicação de um campo elétrico para aumentar a energia dos portadores de carga. Como nos condutores, se uma diferença de potencial é aplicada sobre o material, elétrons se moverão lentamente na direção do potencial mais positivo, e buracos se moverão, ainda mais lentamente, na direção do potencial mais negativo. A mobilidade dos elétrons é aproximadamente o triplo daquela dos buracos. Removido o campo elétrico, o material volta a se comportar como um isolante.

A velocidade de propagação da luz no vácuo (c) é de 300.000 km por segundo. A velocidade com que o campo elétrico se propaga através de um meio físico, tal como um fio de cobre ou fibra ótica, é de $0,6$ a $0,7c$, que é da ordem de 20 cm por nanosegundo. Quando o meio físico sólido é submetido a um campo elétrico, a velocidade de propagação de um único elétron é de alguns milímetros por segundo. Esta é uma distinção importante, e talvez contraintuitiva: o campo eletromagnético se propaga a 20 cm/ns enquanto que os portadores de carga individuais se movem a 10 mm/s.

5.1.2 Operação dos Transistores

O termo *transistor* é uma abreviatura para *transfer-resistor*, ou “resistor de transferência”. Este termo define dispositivos de três terminais, nos quais a resistência entre um par de terminais pode ser controlada pelo terceiro terminal. Dependendo dos potenciais elétricos relativos entre os terminais do transistor, este se comporta como uma baixa resistência – um condutor – ou como uma alta resistência – um isolante. É este comportamento que permite que transistores sejam usados como chaves.

A Figura 5.1 mostra um corte transversal de um transistor do tipo N. O substrato contém material do tipo P e os dois poços são dopados com material do tipo N. A cada poço são ligados os terminais com a *fonte* e o *dreno* de portadores de carga do transistor. O terceiro terminal, chamado de *gate*, é isolado do substrato por uma fina camada de óxido de silício, que é um excelente isolante. Esta camada isolante tem espessura de $20\text{\AA}$, que é a espessura de 6 ou 7 átomos empilhados. O *gate* consiste de uma camada de metal ou de polisilício². Entre os dois poços, abaixo do *gate*, se estabelece o *canal de condução* do transistor.

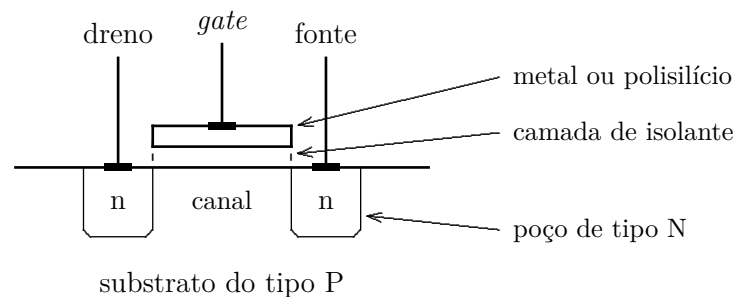


Figura 5.1: Estrutura física de um transistor CMOS tipo N.

²O termo ‘gate’ não tem relacionamento com “porta lógica”, ou “logical gate”.

O substrato é mantido no potencial elétrico correspondente ao nível lógico 0. Se o potencial correspondente ao nível lógico 1 é aplicado no *gate*, como mostrado no lado esquerdo da Figura 5.2, este terminal fica com carga líquida positiva. No substrato, sob o *gate*, acumulam-se cargas elétricas negativas atraídas da vizinhança pelo campo elétrico provocado pela carga positiva no *gate*. O acúmulo de cargas sob o *gate* transforma o canal num condutor, e se uma diferença de potencial for aplicada entre fonte e dreno, por causa da baixa resistência no canal, uma corrente se estabelece da fonte para o dreno, como indica a seta no diagrama do lado direito da Figura 5.2. Valores típicos para o potencial que representa o nível lógico 1 são de 1,25 a 2,5V.

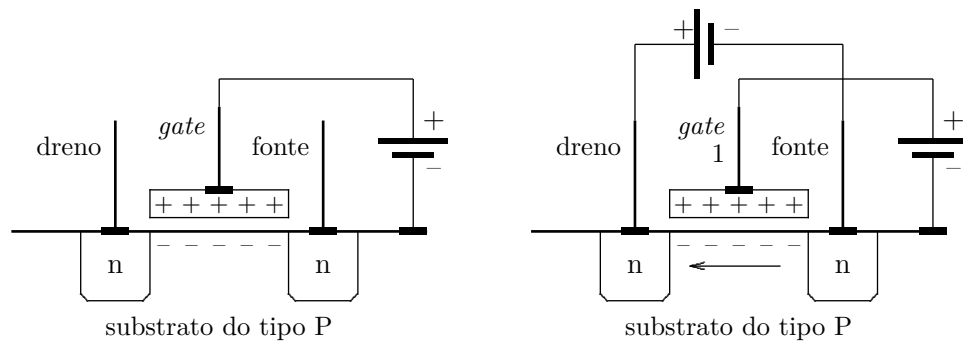


Figura 5.2: Modo de operação de um transistor CMOS tipo N.

Se o potencial do *gate* for reduzido para 0V, a corrente se interrompe porque os elétrons se afastarão do canal retornando aos seus íons de origem, aumentando a resistência do canal. Quando a tensão no *gate* corresponde ao nível lógico 1, o transistor conduz e se comporta como uma chave fechada porque há um caminho de baixa resistência entre fonte e dreno. Quando a tensão no *gate* corresponde a 0, o transistor se comporta como uma chave aberta e os terminais fonte e dreno ficam isolados.

Este transistor é chamado de *transistor tipo N* porque os portadores de carga são elétrons, com carga negativa portanto. Num *transistor tipo P*, as polaridades se invertem e os portadores de carga são buracos.

A Figura 5.3 mostra a estrutura e indica o modo de funcionamento de um transistor tipo P. O substrato é do tipo N, e os dois poços são do tipo P. O substrato é mantido no mesmo potencial que a fonte de alimentação. Se o potencial no *gate* corresponde ao nível lógico 0, este fica com carga líquida negativa, e esta carga afasta os elétrons sob o *gate* e expõe buracos que reduzem a resistência no canal. Se um potencial positivo é aplicado entre fonte e dreno, se estabelecerá um fluxo de buracos da fonte para o dreno.

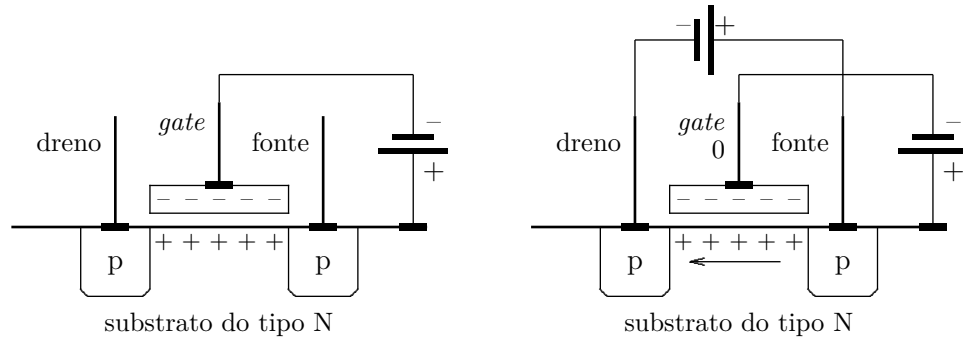


Figura 5.3: Modo de operação de um transistor CMOS tipo P.

Se a fonte de um transistor N for ligada ao nível lógico 1, no dreno o sinal elétrico será equivalente a um nível 1 ‘fraco’ porque os portadores de carga são elétrons, e não buracos. Efeito similar ocorre com transistores P: se a fonte for ligada a um nível lógico 0, o dreno apresentará uma versão fraca de 0 porque seus portadores de carga são buracos, e não elétrons. A ‘força’, ou a intensidade, de um sinal é relacionada à capacidade de transportar corrente elétrica em quantidade suficiente para a correta operação dos circuitos ligados àquele sinal. No que concerne à ‘força’ dos sinais, transistores são simétricos e fonte e dreno se comportam da mesma forma.

Os transistores descritos acima são chamados de FETs, ou *Field Effect Transistors* (Transistor de Efeito de Campo), porque o canal somente se estabelece pelo o efeito do campo elétrico provocado pelas cargas acumuladas no *gate*. O nome “por extenso” destes transistores é MOSFETs, abreviatura para *Metal Oxide Semiconductor, Field Effect Transistor*, e o *Metal-Oxide-Semiconductor* indica a construção do transistor com um *gate* metálico sobre a camada de óxido de silício, por sua vez sobre um substrato de semiconductor.

5.2 Implementação de Portas Lógicas

Esta seção mostra como construir portas lógicas com transistores CMOS. A Seção 5.2.1 emprega chaves controladas como uma primeira abstração para os transistores e descreve os circuitos do inversor e das portas *nand* e *nor*. Na Seção 5.2.2, as chaves são substituídas por modelos lógicos para os transistores, modelos estes que permitem que se empregue a abstração de bits para descrever o comportamento dos circuitos com transistores CMOS.

5.2.1 Circuitos com chaves

Os transistores descritos na Seção 5.1.2 se comportam como chaves e por isso é interessante examinarmos a implementação de funções lógicas com chaves, antes de abordar a implementação com transistores.

Uma *chave digital* é um dispositivo de três terminais, um terminal de controle que abre ou fecha o contato entre os outros dois terminais. Uma chave *normalmente aberta*, ou tipo N, é mostrada na Figura 5.4. A chave é “normalmente aberta” porque o sinal de controle deve ser ativado – colocado no nível lógico 1 – para fechar seus contatos.

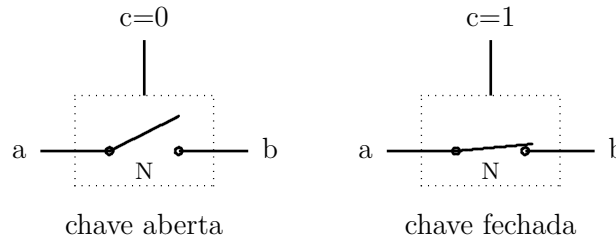


Figura 5.4: Símbolos para as chaves digitais de tipo N.

Se o terminal de controle está em 1, então os contatos se fecham, do contrário, os contatos abrem. Seu comportamento é descrito pela Equação 5.1.

$$\text{chave tipo N : } (c = 1) \Rightarrow (a = b) \quad (5.1)$$

Se o sinal de controle está ativo ($c = 1$), então os níveis lógicos em a e b são iguais. Do contrário, nada se pode afirmar a respeito de seus valores.

Note que não há contato entre o terminal de controle e os terminais a e b . Chaves são uma abstração para o comportamento dos transistores: o terminal de controle c é o *gate* do transistor e os terminais a e b , ligados ao contato da chave, são a fonte e o dreno no transistor.

É conveniente usarmos chaves nas quais o sinal de controle é ativo em 0. Uma chave *normalmente fechada*, tipo P, é mostrada na Figura 5.5. Com o sinal de controle inativo – colocado no nível lógico 0 – os contatos ficam fechados. O círculo no terminal de controle indica a inversão, e seu comportamento é descrito pela Equação 5.2.

$$\text{chave tipo P : } (z = 0) \Rightarrow (x = y) \quad (5.2)$$

Se o sinal de controle está ativo ($z = 0$), então os níveis lógicos em x e y são iguais. Do contrário, nada se pode afirmar a respeito de seus valores.

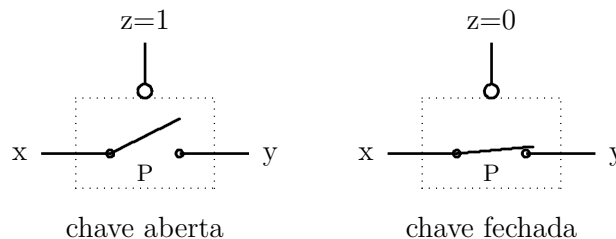


Figura 5.5: Símbolos para as chaves digitais de tipo P.

Inversor O circuito com chaves que implementa um inversor é mostrado na Figura 5.6. As linhas horizontais no topo e na base representam as ligações à fonte de alimentação; a linha no topo do diagrama é, por convenção, a ligação ao potencial mais alto (VCC) que representa uma fonte inesgotável de nível lógico 1, enquanto que a linha na base é a ligação à referência de tensão (GND, de *ground*), que representa uma fonte inesgotável de nível lógico 0. Do ponto de vista elétrico, estas são ligações a um nível lógico 1 ‘forte’, e a um nível lógico 0 ‘forte’.

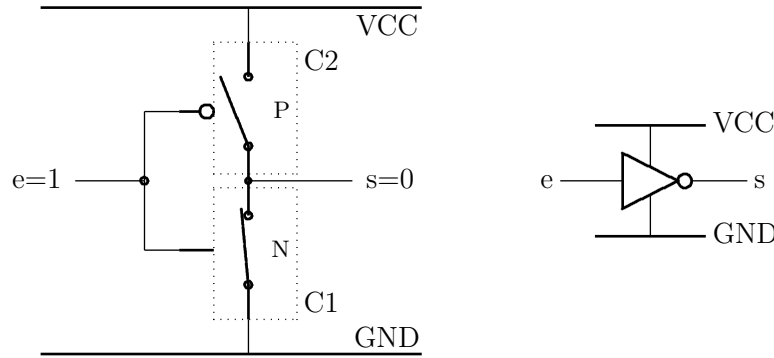


Figura 5.6: Inversor implementado com chaves.

O circuito, com duas chaves, mantém a saída s ligada a VCC (1) ou a GND (0). Se a entrada e está em 1, a chave C1 fica fechada, ligando s a 0. Se a entrada está em 0, a chave C2 fica fechada, ligando s a 1. Os sinais de controle de C1 e de C2 são complementares: o terminal de controle de C2 é ativo em 0, e o de C1 é ativo em 1. Se as duas chaves forem ativadas simultaneamente, ocorrerá um curto circuito na fonte de alimentação, através de C1 e de C2, o que pode danificar o dispositivo ou provocar o seu mau funcionamento, sendo as duas ocorrências altamente indesejáveis.

Ligação em Série A ligação de duas chaves *em série* equivale à conjunção de dois sinais, como indicado na Figura 5.7 – o sinal x é ligado ao sinal y , se ambos os terminais de controle, p e q estão em 1. Se p ou q estiver em 0, nada se pode dizer sobre os níveis de x e y . A conjunção dos sinais de controle pode ser descrita sucintamente por $[(p = 1) \wedge (q = 1)] \Rightarrow (x = y)$.

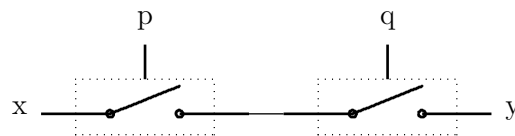


Figura 5.7: Ligação em série.

Ligação em Paralelo A ligação de duas chaves *em paralelo*, como mostra a Figura 5.8, equivale à disjunção de dois sinais. Se qualquer um dentre p ou q estiver em 1, então uma das chaves liga x a y . Se ambos p e q estiverem em 1, as duas chaves ligam x a y ; se ambos estiverem em 0, as duas chaves ficam abertas e nada se pode dizer sobre os níveis de x e y . A disjunção dos sinais de controle pode ser descrita por $[(p = 1) \vee (q = 1)] \Rightarrow (x = y)$.

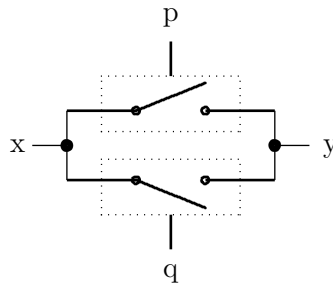
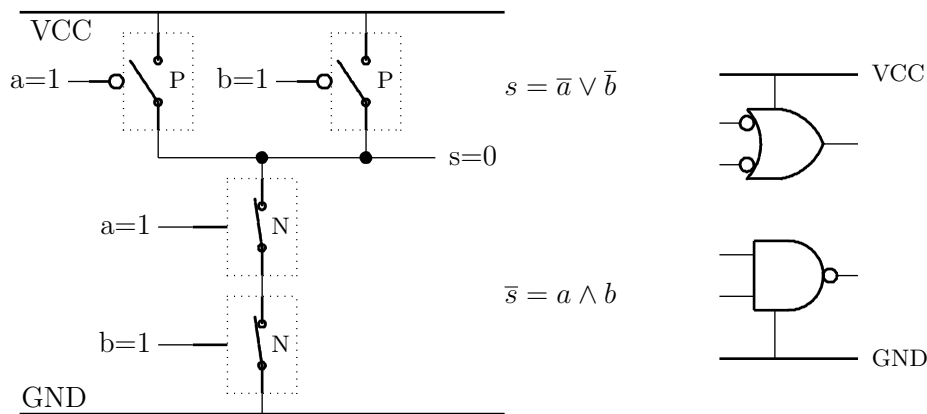


Figura 5.8: Ligação em paralelo.

Porta *nand* O circuito que implementa uma porta *nand* é mostrado na Figura 5.9. A saída s fica em 0 exatamente quando a e b estão em 1, ou $\bar{s} = a \wedge b$. Para tanto, a saída deve ser ligada a 0 por um circuito com duas chaves em série, controladas por a e b . A saída fica em 1 quando, no mínimo, um dentre a ou b é 0, ou $s = \bar{a} \vee \bar{b}$. A saída deve ser conectada a 1 pela ligação em paralelo de duas chaves controladas por $\bar{a}$ e por $\bar{b}$. O circuito da porta *nand* é composto por duas redes, uma que liga a saída à VCC, e outra que liga a saída à GND, e as equações que definem o comportamento dessas redes são equivalentes segundo o DeMorgan. Na Figura 5.9 as entradas a e b estão em 1, e portanto a saída s é 0.

Figura 5.9: Porta *nand* implementada com chaves.

Os dois pontos escuros na linha do sinal s representam ligações nos sinais, como um ponto de solda que une dois ou mais fios. Se duas linhas se cruzam sem o “ponto de solda”, então não há ligação entre elas.

Porta *nor* Uma porta *nor* pode ser implementada com chaves aplicando-se a dualidade ao projeto da porta *nand*: cada rede série é substituída por uma rede paralela, e cada rede paralela é substituída por uma rede série. A Figura 5.10 mostra uma porta *nor* implementada a partir do circuito da porta *nand*. A rede ligada em série que conecta a saída a 0 é substituída por uma rede ligada em paralelo; a rede ligada em paralelo que conecta a saída a 1 é substituída por uma rede ligada em série. Tal como na porta *nand*, as equações das duas redes são equivalentes segundo DeMorgan. Na Figura 5.10 a entrada *a* é 0 e *b* é 1, e portanto a saída *s* é 0.

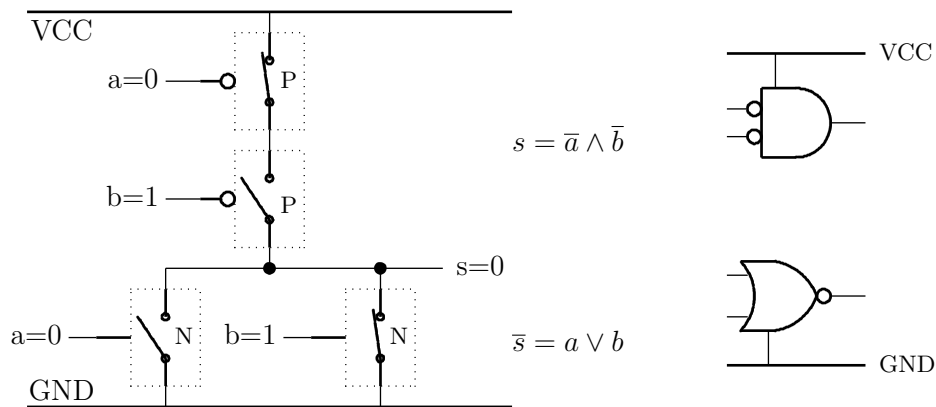


Figura 5.10: Porta *nor* implementada com chaves.

Não é coincidência que as chaves que ligam a saída à VCC tenham o terminal de controle com uma negação. A razão para tal é esclarecida na Seção 5.2.2.

5.2.2 Portas Lógicas CMOS

A Figura 5.11 mostra os símbolos usuais para os dois transistores CMOS, transistores tipo P e tipo N. Estes transistores se comportam como chaves e quando o terminal *g* (*gate*) está ligado ao nível lógico adequado, o nível lógico dos outros dois terminais *f* e *d* é equivalente, porque a chave está fechada.



Figura 5.11: Transistores CMOS.

Quando o *gate* está no outro nível, a chave fica aberta e nada se pode dizer quanto aos valores em *f* e em *d*, porque estes dependerão dos circuitos aos quais aqueles terminais estão ligados. O terminal *f* é chamado de fonte (*source*) porque este terminal é ligado à fonte de cargas elétricas, enquanto que o terminal *d* é chamado de dreno (*drain*) porque ele é ligado ao ‘dreno’ por onde as cargas elétricas escoam.

Num transistor P, os portadores da carga no canal entre os terminais fonte e dreno são os ‘buracos’ que resultam da falta de elétrons, sendo portanto cargas elétricas positivas. Os portadores de carga no canal entre fonte e dreno de um transistor tipo N são elétrons, com carga elétrica negativa. É por causa dos dois tipos de transistores, que são normalmente usados como “pares complementares”, que o nome desta tecnologia é *Complementary-MOS*.

O círculo no *gate* do transistor tipo P indica que o nível lógico 0 reduz a resistência entre os terminais fonte e dreno, como uma chave fechada. Complementarmente, o nível lógico 1 no *gate* faz o transistor de tipo N conduzir. A Equação 5.3 define o comportamento dos transistores.

$$\begin{array}{ll} \text{transistor P} & (g = 0) \Rightarrow (f = d) \\ \text{transistor N} & (g = 1) \Rightarrow (f = d) \end{array} \quad (5.3)$$

Os circuitos CMOS são tipicamente constituídos de duas redes, uma rede que “puxa a saída para cima” (para 1, ou *pull-up*) e outra rede que “puxa a saída para baixo” (para 0, ou *pull-down*). A rede que puxa para cima é composta somente de transistores tipo P porque estes conduzem bem sinais de nível lógico 1, e por conta disso, nesta rede há uma conexão com a fonte de alimentação (VCC), que é a fonte de nível lógico 1. A rede que puxa para baixo é composta somente de transistores do tipo N porque estes conduzem bem sinais de nível 0, e esta rede é ligada à referência de tensão (GND), que é a fonte de nível lógico 0.

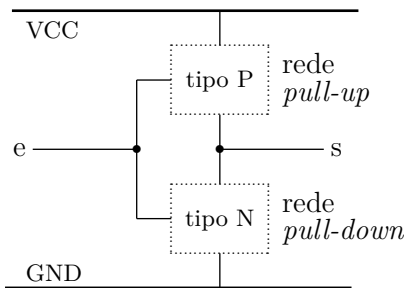


Figura 5.12: Modelo de porta CMOS com redes *pull-up* e *pull-down*.

Como o nome indica, uma *rede* é composta por fontes de alimentação (VCC e GND), por transistores e pelos fios que os interligam. Um *nó* da rede é uma ligação entre dois ou mais de seus componentes. Para simplificar a análise destes circuitos, emprega-se uma abstração para o comportamento dos fios que interligam os componentes: os fios, e portanto os nós do circuito, se comportam como *superfícies equipotenciais*. Isso significa que uma mudança de tensão em qualquer ponto de um fio se propaga instantaneamente por todos os pontos daquele fio, ou por toda a superfície coberta por aquele nó equipotencial.

Inversor

A técnica de projeto para portas lógicas com transistores CMOS é descrita empregando o inversor como exemplo. A Figura 5.13 mostra o Mapa de Karnaugh do inversor com entrada *e* e saída *s*.

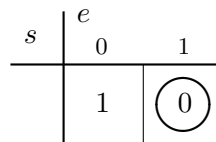


Figura 5.13: Mapa de Karnaugh para o inversor.

Para projetar a rede que puxa para baixo (*pull-down*), a função lógica que define esta rede é obtida da(s) célula(s) do mapa que estão preenchidas com 0, como indicado pelo agrupamento

da célula 1. Neste caso, a rede contém um transistor N cujos terminais fonte e dreno são GND e s respectivamente, e o *gate* é e . A rede que puxa para cima (*pull-up*) é o dual da função *pull-down*, e neste caso, $pull-up = pull-down = \bar{e}$. Esta rede consiste de um transistor P cuja fonte é ligada a VCC, o dreno à s , e o *gate* é controlado por e . Note que a inversão no *gate* do transistor P corresponde à negação de e e portanto não é necessário inverter o sinal ligado ao terminal *gate*. A implementação de um inversor é mostrada na Figura 5.14. Normalmente, as ligações à fonte de alimentação são omitidas dos esquemáticos, embora estas ligações devam sempre existir.

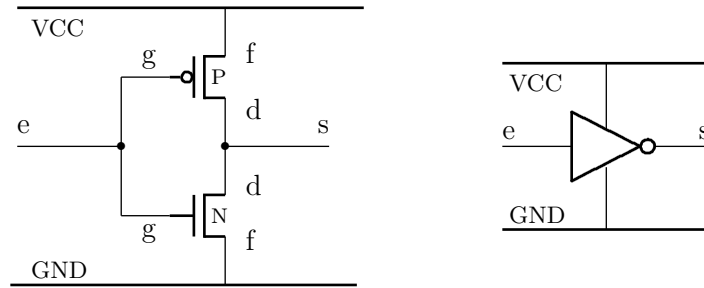


Figura 5.14: Inversor CMOS.

O sinal de entrada e é ligado ao *gate* dos dois transistores e a saída s é ligada aos seus drenos. Quando a entrada está em 1, o transistor P está ‘aberto’ (circuito aberto) e com uma ligação de alta resistência entre sua fonte e a saída. O transistor N está ‘fechado’ (circuito fechado) e portanto há um caminho de baixa resistência entre a saída e GND, fazendo $s = 0$. Quando a entrada está em 0, o transistor P é uma chave fechada, ligando a saída à VCC, enquanto que o transistor N é uma chave aberta. A Figura 5.15 mostra o inversor nas duas situações. O transistor que está ‘aberto’ é mostrado com linhas pontilhadas, indicando que aquele transistor se comporta como um isolante.

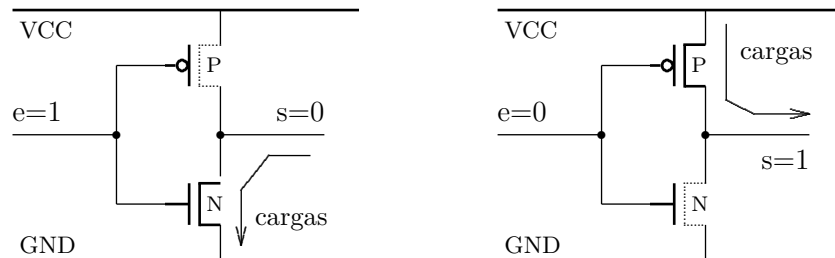


Figura 5.15: Operação do circuito inversor CMOS.

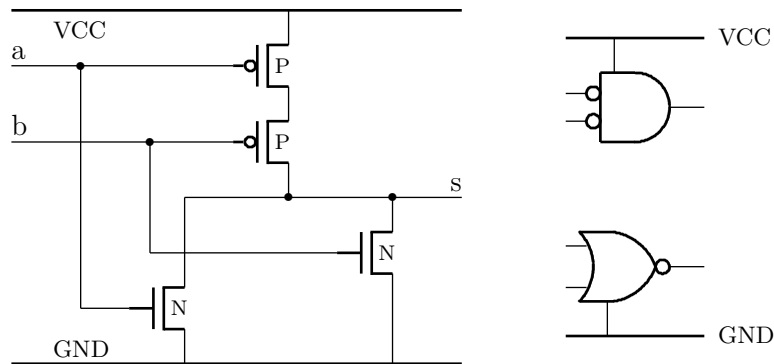
Porta *nor*

O Mapa de Karnaugh da porta *nor* é mostrado na Figura 5.16. As células preenchidas com 0 estão agrupadas e estas determinam a rede que puxa a saída para baixo: $pull-down = b \vee a$. Esta rede é implementada com uma ligação em paralelo de dois transistores N, com os sinais a e b ligados aos seus *gates*, como mostra a Figura 5.17. A rede que puxa para cima é o dual da rede $pull-down$: $pull-up = \overline{pull-down} = \bar{b} \wedge \bar{a}$. Esta rede é implementada pela ligação em série de dois transistores P, com seus *gates* ligados a a e b . Lembre que as inversões da função $pull-up$ estão implícitas no comportamento dos transistores P.

		b	
		0	1
a	0	1	0
	1	0	0

Figura 5.16: Mapa de Karnaugh para a porta *nor*.

A implementação da porta *nor* é mostrada na Figura 5.17. A rede que puxa para 1 comporta-se como uma porta *and* com as entradas complementadas, enquanto que a rede que puxa para 0 comporta-se como uma porta *nor*. O Teorema de DeMorgan garante que as “portas lógicas” destas duas redes têm comportamento equivalente. A saída é 0 se qualquer das entradas estiver em 1 – a ligação em paralelo dos transistores N equivale a $a \vee b$. A saída é 1 somente se as duas entradas forem 0 porque a ligação em série dos transistores P equivale a $\bar{a} \wedge \bar{b}$. As duas redes são necessárias por causa das características de condução dos transistores: aqueles do tipo P conduzem bem cargas positivas, enquanto que transistores do tipo N conduzem bem cargas negativas.

**Figura 5.17:** Porta *nor* implementada em CMOS.

Porta *nand*

O Mapa de Karnaugh da porta *nand* é mostrado na Figura 5.18. A rede *pull-down* é $a \wedge b$, e é portanto implementada pela ligação em série de dois transistores N. A rede que puxa a saída para cima é $pull-up = \overline{pull-down} = \bar{b} \vee \bar{a}$, implementada pela ligação em paralelo de dois transistores P. Lembre que as negações em *pull-up* são implícitas ao comportamento dos transistores P.

s	b	
	0	1
a	0	1
	1	1

Figura 5.18: Mapa de Karnaugh para a porta *nand*.

A Figura 5.19 mostra a implementação da porta *nand*. Os circuitos das portas *nand* e *nor* são chamados de *duals* por causa da dualidade nas ligações entre os transistores P e N. Neste caso, *dualidade* significa que uma ligação em série ($\wedge$) é substituída por uma ligação em paralelo ($\vee$), e que uma ligação em paralelo ($\vee$) é substituída por uma ligação em série ($\wedge$). Conforme a Definição 3.12, um circuito Γ é *dual* de um circuito Δ se para todas as ligações em série no circuito Γ existem ligações em paralelo correspondentes no circuito Δ , e se para as ligações em paralelo em Γ existem ligações em série em Δ .

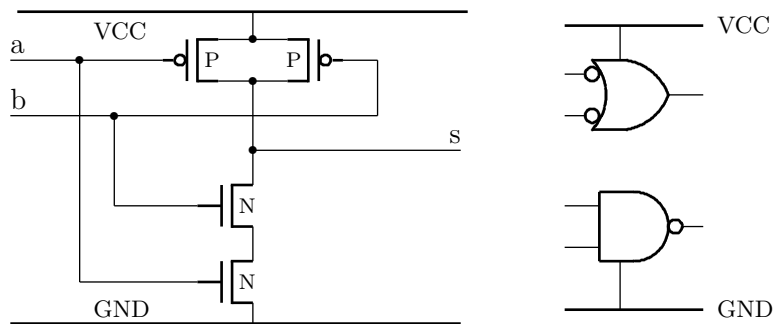


Figura 5.19: Porta *nand* implementada em CMOS.

Aspectos tecnológicos dos circuitos CMOS

A topologia de implementação de circuitos CMOS com redes *pull-up* e *pull-down* implica em que as portas CMOS sejam *sempre* inversoras. Como a rede *pull-down* é composta somente por transistores do tipo N, quando as entradas estão em 1, a saída é puxada para 0. Da mesma forma, quando as entradas da rede *pull-up* são 0, os transistores do tipo P puxam a saída para 1. Isso não chega a ser um problema porque pode-se aplicar DeMorgan para simplificar as inversões, como no circuito da Figura 4.6 (pág. 58).

A implementação de inversores, portas *nand* e portas *nor* emprega circuitos com um número mínimo de transistores. A implementação de portas *and* e *or* em CMOS implica, necessariamente, na ligação de inversores adicionais nas saídas destas portas – ou às suas entradas.

A dualidade das redes *pull-up* e *pull-down* tem um efeito colateral assaz importante: uma vez que as entradas estejam em repouso por tempo suficiente para que todos os nós da rede tenham completado as transições de 1 para 0 e de 0 para 1, uma das duas redes interrompe o fluxo de corrente entre VCC e GND, e portanto o circuito não dissipa energia. Isso significa que circuitos CMOS em repouso não dissipam energia porque não circula corrente através das resistências entre fonte e dreno dos transistores. Na Seção 5.3.3 veremos que esta afirmação não é completamente verdadeira. Se, por um erro de projeto as redes *pull-up* e *pull-down* conduzirem ao mesmo tempo, a corrente entre VCC e GND pode danificar o dispositivo, ou no mínimo, drenar a bateria mais rapidamente do que o desejável.

O comportamento dinâmico dos circuitos CMOS faz com que, durante um breve intervalo de tempo, as duas redes conduzam e portanto o circuito dissipa energia na vizinhança das transições. Considere a transição de 0 para 1 na entrada de um inversor; a transição não é instantânea e a tensão no *gate* dos transistores fica indeterminada enquanto sobe de 0V até VCC, o que faz com que o *pull-up* deixe de conduzir – sua resistência aumenta – ao mesmo tempo em que o *pull-down* começa a conduzir – sua resistência diminui. Durante a breve transição na entrada, os dois transistores conduzem e o circuito dissipa energia. Finda a transição, a dissipação tende rapidamente a zero. Voltaremos a esse assunto na Seção 5.3.3.

5.2.3 Portas Complexas e Células

Um dos grandes atrativos da tecnologia CMOS advém da possibilidade de projetar funções lógicas algo mais complexas do que portas lógicas ao interligar redes com vários transistores. Funções como a soma de dois bits e vem-um, ou o multiplexador de duas entradas, podem ser implementadas com um número de transistores que é menor do que se fossem empregadas portas lógicas individuais.

Estes circuitos são chamados de *células*, e uma vez que uma célula tenha sido otimizada para velocidade e/ou tamanho, ela pode ser replicada em circuitos ainda mais complexos. Por exemplo, um circuito que efetua a soma em 16 bits consiste de 16 réplicas da célula que computa a soma e o vai-um de três bits.

Chamamos de *portas complexas* àqueles circuitos que são algo mais complexos do que uma porta lógica. Tais circuitos são projetados da mesma forma que as portas lógicas: a partir da especificação lógica, derivamos a rede *pull-down*, e então derivamos seu dual, que é a rede *pull-up*. Vejamos alguns exemplos.

Exemplo 5.1 Considere a função

$$r = (a \wedge b) \vee (c \wedge d).$$

A rede *pull-down* é derivada a partir do complemento de r , que são as células com zeros no Mapa de Karnaugh. A Equação 5.4 contém a derivação de *pull-down*.

$$\begin{aligned} \text{pull-down: } \bar{r} &= \overline{(a \wedge b) \vee (c \wedge d)} && \text{DeMorgan} \\ &= \overline{(a \wedge b)} \wedge \overline{(c \wedge d)} && \text{DeMorgan} \\ &= (\bar{a} \vee \bar{b}) \wedge (\bar{c} \vee \bar{d}) \end{aligned} \quad (5.4)$$

O circuito que puxa a saída para baixo contém duas ligações em paralelo $(\bar{a} \vee \bar{b})$ e $(\bar{c} \vee \bar{d})$, e estas duas são ligadas em série por causa do $\wedge$, como mostra a parte de baixo da Figura 5.20.

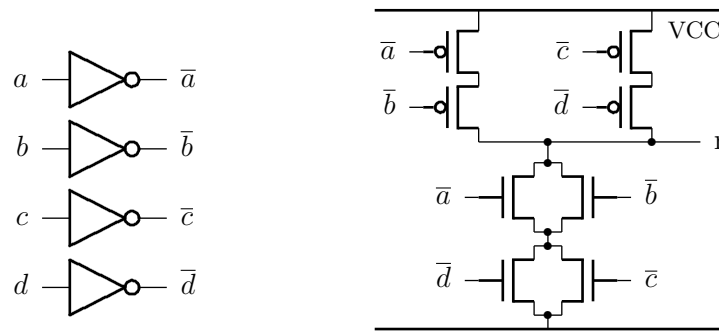


Figura 5.20: Implementação de $r = (a \wedge b) \vee (c \wedge d)$.

O circuito que puxa a saída para cima contém duas ligações em série $(\bar{a} \wedge \bar{b})$ e $(\bar{c} \wedge \bar{d})$, e estas são ligadas em paralelo ($\vee$), como mostra a parte de cima da Figura 5.20. Note que o nível lógico dos sinais nos gates dos transistores tipo P não é diferente daquele dos gates dos transistores tipo N.

A porta complexa que implementa a função r é um multiplexador de duas entradas quando $b = \bar{d}$. Dependendo do circuito à que esta célula será ligada, além dos 8 transistores, podem ser necessários quatro inversores para complementar as entradas. ◀

Exemplo 5.2 O multiplexador da Figura 5.20 emprega oito transistores, mais os oito transistores para obter os complementos dos sinais das entradas. Se iniciarmos o projeto com o complemento de r

$$\bar{r} = \overline{(a \wedge b) \vee (c \wedge d)},$$

a rede *pull-down* usa variáveis de entrada sem complementar, o que resulta num circuito mais compacto. A Equação 5.5 contém a derivação da rede *pull-down*.

$$\begin{aligned} \text{pull-down: } \bar{r} &= \overline{\overline{(a \wedge b) \vee (c \wedge d)}} && \text{involução} \\ &= (a \wedge b) \vee (c \wedge d) \end{aligned} \quad (5.5)$$

O circuito que puxa a saída para baixo contém duas ligações série $(a \wedge b)$ e $(c \wedge d)$, e estas duas são ligadas em paralelo por causa do $\vee$, como mostra a parte de baixo da Figura 5.21.

É necessário um inversor para complementar a saída $\bar{r} = s$, e esta implementação economiza seis transistores, com relação à da Figura 5.20. Por outro lado, as portas lógicas CMOS são naturalmente inversoras e é provável que os complementos dos sinais já estejam disponíveis no circuito. Qual das duas é a solução mais econômica depende do circuito à volta do multiplexador. ◀

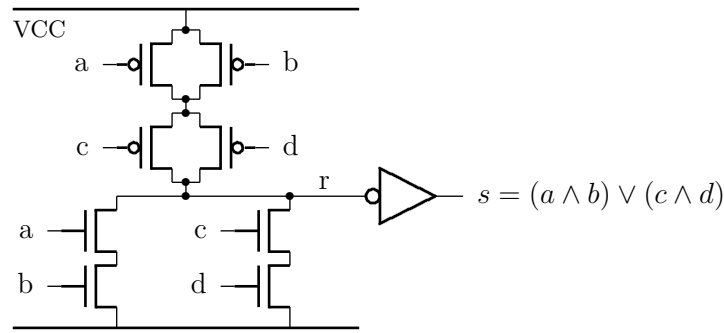


Figura 5.21: Implementação de $s = \bar{r} = (a \wedge b) \vee (c \wedge d)$.

São necessários 14 transistores para implementar um multiplexador com portas lógicas, quatro a mais do que no circuito da Figura 5.21. O ganho não está somente no número de transistores; esta célula pode ser implementado de forma mais compacta do que aquela com portas lógicas individuais – circuitos compactos implicam numa melhor utilização da superfície do circuito integrado, e com a redução na área, geralmente se obtém uma redução no tempo de propagação do circuito.

Exemplo 5.3 Implementemos a função

$$t = \overline{(a \vee b) \wedge c}.$$

As funções *pull-down* e *pull-up* estão indicadas na Equação 5.6. A rede que puxa a saída para baixo é composta uma ligação em paralelo $(a \vee b)$, que é ligada em série com c ($\wedge c$). A rede que puxa a saída para cima é uma ligação em paralelo, na qual um ramo é $\bar{c}$ e o outro a ligação em série de $\bar{a}$ com $\bar{b}$ ($\bar{a} \wedge \bar{b}$). A Figura 5.22 mostra o circuito que implementa t . ◁

$$\begin{aligned}
 \text{pull-down: } \bar{t} &= \overline{(a \vee b) \wedge c} && \text{involução} \\
 &= (a \vee b) \wedge c \\
 \text{pull-up: } \overline{\text{pull-down}} &= \overline{(a \vee b) \wedge c} && \text{DeMorgan} \\
 &= \overline{(a \vee b)} \vee \bar{c} && \text{DeMorgan} \\
 &= (\bar{a} \wedge \bar{b}) \vee \bar{c}
 \end{aligned} \tag{5.6}$$

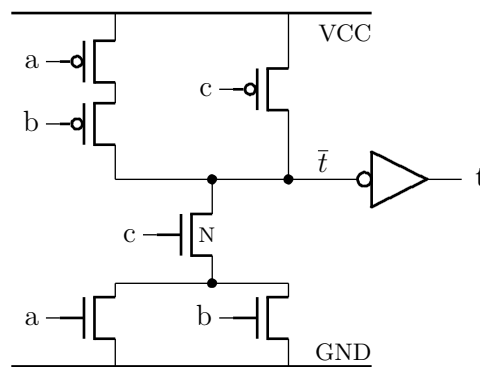


Figura 5.22: Implementação de $t = \overline{(a \vee b) \wedge c}$.

Exemplo 5.4 Implementemos uma função algo mais complexa:

$$x = \bar{a} \wedge b \wedge c \vee a \wedge \bar{b} \wedge c \vee a \wedge b \wedge \bar{c}$$

A Equação 5.7 mostra a derivação da rede *pull-down*.

$$\begin{aligned} x &= (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c}) \\ \bar{x} &= \overline{(\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})} && \text{involução} \\ \bar{x} &= (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c}) && \text{pull-down} \end{aligned} \quad (5.7)$$

A rede *pull-down* é a ligação em paralelo de três redes (R_1, R_2, R_3), cada uma destas a ligação em série de três transistores: $R_1 = (\bar{a} \wedge b \wedge c)$, $R_2 = (a \wedge \bar{b} \wedge c)$, e $R_3 = (a \wedge b \wedge \bar{c})$. A rede *pull-up* é uma série de três redes, cada uma destas a ligação em paralelo de três transistores. Por exemplo, $R'_1 = (\bar{a} \vee b \vee c)$. A Figura 5.23 mostra o circuito completo. $\triangleleft$

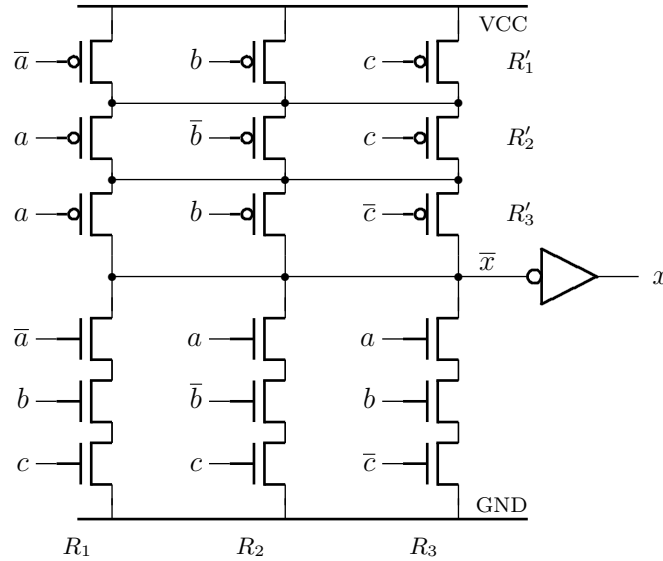


Figura 5.23: Implementação de $x = \bar{a} \wedge b \wedge c \vee a \wedge \bar{b} \wedge c \vee a \wedge b \wedge \bar{c}$.

Exercícios

Ex. 5.1 Projete as portas complexas que implementam as seguintes funções:

- $m = \bar{a} \wedge \bar{b} \wedge c$;
- $n = \bar{a} \vee \bar{b} \vee c$;
- $p = \bar{a} \wedge \bar{b} \wedge c \wedge \bar{d}$;
- $q = \bar{a} \vee \bar{b} \vee c \vee \bar{d}$;
- $x = a \wedge \bar{b} \vee \bar{a} \wedge b$; ($\oplus$)
- $e = a \wedge b \vee \bar{a} \wedge \bar{b}$; ($\oplus$)
- $y = (a \wedge s) \vee (b \wedge \bar{s}) \vee (a \wedge b)$ (multiplexador bem comportado).

Ex. 5.2 Um *somador parcial* é um circuito combinacional de entradas a e b e saídas s (soma) e v (vai-um), definidas na Equação 5.8. Mostre como implementar um somador parcial em CMOS.

$$s = a \oplus b, \quad v = a \wedge b \quad \text{somador parcial} \quad (5.8)$$

Ex. 5.3 Um *somador completo* é um circuito combinacional com entradas a , b e vem (vem-um), e saídas s (soma) e vai (vai-um), que implementa as funções s (soma) e vai (vai-um) definidas na Equação 5.9. Mostre como implementar um somador completo em CMOS. *Pista*: a operação $\oplus$ é associativa.

$$\left. \begin{aligned} s &= a \oplus b \oplus vem \\ vai &= a \wedge b \vee a \wedge vem \vee b \wedge vem \end{aligned} \right\} \text{ somador completo} \quad (5.9)$$

Ex. 5.4 Um *somador completo* pode ser implementado com dois somadores parciais e uma porta *or*, como mostra a Figura 5.24. (i) Verifique que este circuito implementa a especificação para o somador completo especificado na Equação 5.9; (ii) implemente esta versão do somador completo em CMOS; e (iii) qual das versões do somador completo emprega menos transistores? *Pista*: não esqueça de que as portas lógicas CMOS são inversoras.

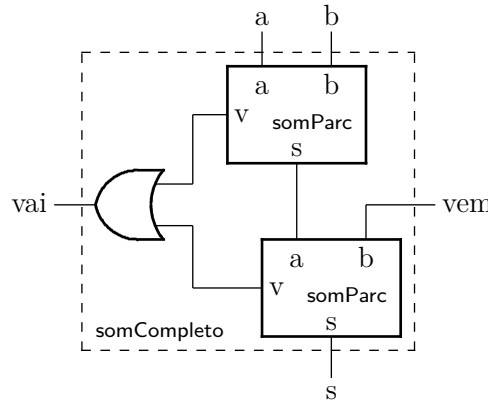


Figura 5.24: Somador completo com dois somadores parciais.

5.3 O Tempo de Propagação É Maior que Zero

Ao empregar bits como um modelo para sinais analógicos, uma característica importante foi escondida pela abstração: os sinais se propagam através dos circuitos em um tempo que não é infinitamente pequeno. Como um mínimo, no vácuo as ondas eletromagnéticas se propagam na velocidade da luz, que é de $c = 0,3 \cdot 10^9$ m/s, ou numa escala mais próxima da prática em sistemas digitais, 30 cm por nanosegundo. Em meios guiados, como em condutores metálicos, os sinais elétricos se propagam com velocidade entre $0,6$ e $0,7c$, que é de aproximadamente “um palmo por nanosegundo”. Num sistema com relógio de 3,3 GHz, a mudança no valor de um bit percorre cerca de 7cm durante um ciclo do relógio.

Circuitos CMOS acrescentam atrasos significativos ao tempo de propagação de sinais, e estes atrasos decorrem da construção dos transistores, quando se considera somente os efeitos de primeira ordem, que é o caso deste texto. As Seções 5.3.1 e 5.3.2 contêm uma breve introdução ao comportamento físico dos dispositivos. A Seção 5.4 apresenta um modelo para o tempo de propagação dos circuitos. A Seção 5.3.3 discute a dissipação de energia em circuitos CMOS.

5.3.1 Um Breve Passeio Pelo Reino da Física

Todos os materiais condutores apresentam alguma resistência à passagem da corrente elétrica. A Lei de Ohm nos informa que a corrente i através de um resistor R é proporcional à tensão v aplicada aos seus terminais, como indicado na Equação 5.10, para o diagrama do circuito na Figura 5.25.

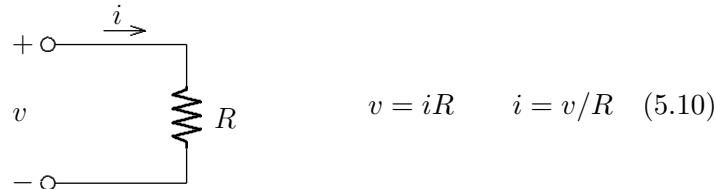


Figura 5.25: Lei de Ohm, tensão e corrente num resistor.

Duas superfícies metálicas isoladas, que estejam próximas uma da outra, armazenam energia no campo elétrico que existe entre as cargas elétricas opostas nas superfícies. Ao escovar o cabelo com uma escova sintética, os fios de cabelo eriçados se comportam como “placas metálicas” isoladas e carregadas: os fios ficam carregados com os elétrons removidos da escova, e como estes têm carga de mesmo sinal, os fios se repelem deixando o “cabelo em pé”.

Um *capacitor* é um dispositivo que armazena energia no campo elétrico que se forma entre duas placas metálicas isoladas. A *capacitância* é medida em Farads e é a razão entre a carga líquida Q numa das placas e a tensão v entre as placas: $C = Q/v$. Se área A das placas é grande, grande é a quantidade de carga líquida para uma dada tensão; se distância D entre as placas é pequena, grande é a força entre as cargas e também a energia armazenada por unidade de carga. Portanto: $C \propto A/D$.

A corrente elétrica é a variação da carga com o tempo: $i = dQ/dt$. A carga líquida numa das placas de um capacitor é, ao longo do tempo, a integral da corrente que ‘atravessa’ o dispositivo:

$$Q = \int_{-\infty}^t i(t) dt . \quad (5.11)$$

Como as placas são isoladas, não há fluxo de cargas através do dispositivo; o que ocorre é que há ingresso de carga líquida por um terminal, e egresso da mesma quantidade de carga, com polaridade oposta, pelo outro terminal. A corrente ‘através’ de um capacitor é na verdade o deslocamento das cargas entra suas placas através do circuito externo ao capacitor.

A diferença de potencial, ou tensão, entre as placas de um capacitor é proporcional à corrente ao longo do tempo:

$$v = \frac{1}{C} \int_{-\infty}^t i(t) dt = \frac{1}{C} Q . \quad (5.12)$$

A corrente ‘através’ do capacitor é proporcional à variação de tensão entre os seus terminais:

$$i = C \frac{dv}{dt} . \quad (5.13)$$

A corrente “através do capacitor” pode ser medida em um dos seus terminais porque as placas são isoladas. Com a aplicação da diferença de potencial, a carga líquida Q é removida de uma placa e acrescida à outra, através do circuito externo.

A Figura 5.26 mostra o símbolo do capacitor e a relação entre tensão e corrente nos seus terminais, na Equação 5.14.

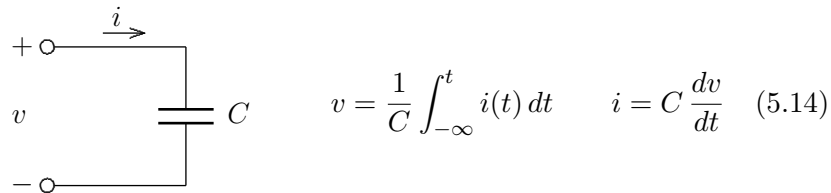


Figura 5.26: Relação entre tensão e corrente num capacitor.

Mais do que os dispositivos em si, nos interessa o comportamento dinâmico de circuitos com resistores (fios) e capacitores (*gates*) de transistores. Em breve retornaremos aos transistores.

O circuito na Figura 5.27 mostra uma fonte de alimentação de tensão V ligada a um resistor R e a um capacitor C através de uma chave de duas posições. A chave esteve na posição α por um tempo longo o bastante para que a tensão no capacitor seja igual à da fonte ($v_C = V$) e portanto não circula corrente no circuito ($i = 0$).

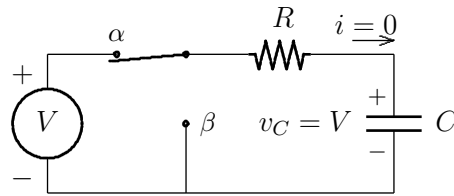


Figura 5.27: Circuito RC em repouso.

No instante t_0 , a chave é movida instantaneamente para a posição β e a energia acumulada no capacitor passa a se dissipar no resistor, na forma de calor. Como este circuito é uma malha fechada, a Lei de Kirchhoff garante que a soma das tensões ao longo da malha deve ser zero e portanto a tensão nos terminais do capacitor v_C é a mesma que nos terminais do resistor v_R , *viz* $v_R = v_C$, e

$$v_R + v_C = 0 \quad \text{e} \quad R i(t) + \frac{1}{C} \int_{t_0}^{\infty} i(t) dt = 0.$$

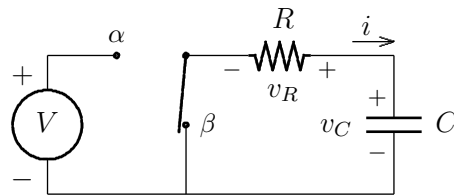


Figura 5.28: Resposta dinâmica do circuito RC.

Derivando-se esta equação com relação ao tempo, para resolver a integral da corrente, se obtém a equação diferencial

$$R \frac{di}{dt} + \frac{i(t)}{C} = 0$$

cujas solução, considerando-se que a tensão no capacitor é V_0 em t_0 , é dada abaixo, para a

tensão e a corrente nos terminais do capacitor.

$$i_C(t) = \frac{V_0}{R} e^{-t/RC} \quad \text{e} \quad v_C(t) = V_0 e^{-t/RC} \quad (5.15)$$

A Figura 5.29 mostra a curva da tensão nos terminais do capacitor (v_C) ao longo do tempo. Antes de t_0 , a tensão no capacitor é a mesma da fonte. Quando a chave muda de posição, esta tensão é aplicada sobre o resistor que passa a dissipar a energia acumulada no capacitor, a uma taxa determinada pelas constantes R e C .

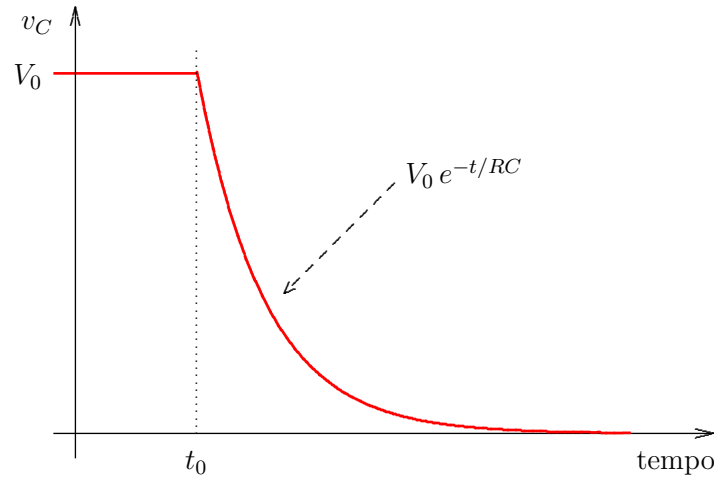


Figura 5.29: Tensão nos terminais do capacitor v_C na descarga do capacitor.

Se o produto RC é muito pequeno ($RC \rightarrow 0$), o decaimento da curva é rápido – este seria o caso de um curto-circuito nos terminais do capacitor. Se o produto RC é grande ($RC \rightarrow \infty$), então o decaimento da curva é lento – este seria o caso de uma resistência muito alta ligada aos terminais do capacitor. Com um circuito aberto, o que equivale a uma resistência infinita, um capacitor ideal manteria sua carga indefinidamente.

5.3.2 Comportamento Dinâmico de Circuitos CMOS

Considere o transistor tipo N mostrado na Figura 5.30. Para se compreender o comportamento dinâmico dos transistores, o comportamento elétrico da região que compreende *gate*, isolante e substrato pode ser aproximado àquele de um capacitor.

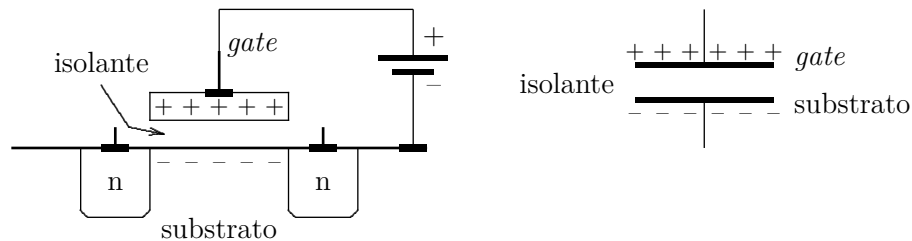


Figura 5.30: Capacitância entre *gate* e substrato.

As placas do capacitor são o *gate* e o substrato, separadas pela camada isolante. Quando há uma diferença de potencial entre as placas, o campo elétrico armazena energia, que não pode ser removida instantaneamente do dispositivo.

Num circuito CMOS típico cada entrada é ligada, no mínimo, a dois transistores, um do tipo N e outro do tipo P, e cada um dos dois *gates* contribui com uma fração da capacitância ligada à saída do circuito que produz o sinal. A Figura 5.31 mostra um circuito com dois inversores, e o circuito equivalente ao inversor I_2 , como ‘visto’ pela saída do inversor I_1 . Lembre que não circula corrente da entrada para a saída de circuitos similares ao inversor.

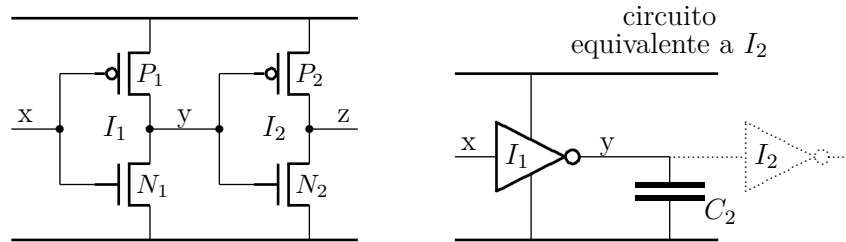


Figura 5.31: Circuito equivalente à entrada de um inversor.

Quando ocorre uma transição na entrada do inversor I_1 , o capacitor ligado a sua saída deve ser carregado, com carga líquida final numa das placas ≥ 0 , ou descarregado, com carga líquida final $= 0$, dependendo do valor anterior na entrada de I_1 . A Figura 5.32 mostra os circuitos equivalentes nas transições de 1 para 0 e de 0 para 1.

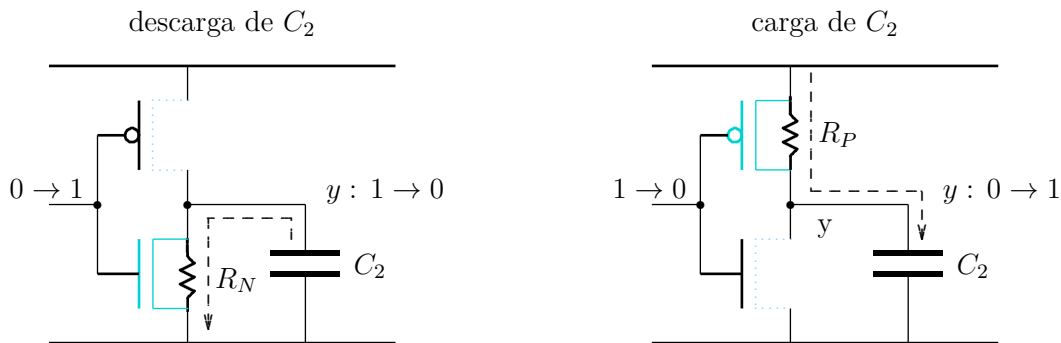


Figura 5.32: Circuitos na carga e na descarga da capacitância de saída C_2 .

Antes da transição $0 \rightarrow 1$ na entrada, o capacitor estava carregado ($v_y = VCC$) porque a saída do inversor estava em 1. Logo após a transição, o transistor P fica aberto, e o transistor N conduz com uma resistência R_N . O modelo deste circuito é um resistor R_N em série com o capacitor C_2 e a equação que descreve o comportamento dinâmico da tensão no ponto y é

$$v_y(t) = (1 - e^{-t/R_N C_2})VCC. \quad (5.16)$$

A variação com o tempo t da tensão sobre os terminais do capacitor, na descarga, é mostrada no lado esquerdo da Figura 5.33. Na medida em que o tempo passa, a exponencial cresce e a tensão em y cai de VCC até GND . A declividade da curva é determinada pelo produto $R_N C_2$.

Na transição da 1 para 0 na entrada, o capacitor que estava inicialmente descarregado, é carregado através da resistência R_P , e a tensão em y sobe de GND a VCC com

$$v_y(t) = e^{-t/R_P C_2}VCC. \quad (5.17)$$

Este comportamento é mostrado no lado direito da Figura 5.33.

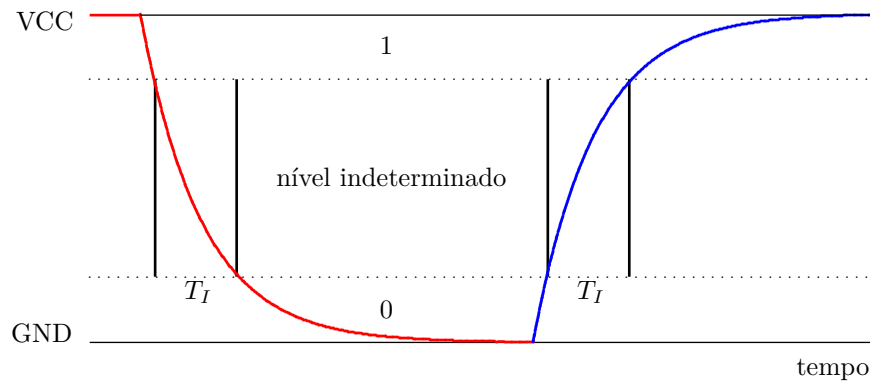


Figura 5.33: Tempo de carga e de descarga da circuito de saída.

A Figura 5.33 está dividida em três faixas horizontais: no topo está a região na qual os sinais são considerados como sendo 1, enquanto que na base está a região na qual os sinais são considerados como 0. Na região entre 0 e 1, os sinais tem um nível lógico indeterminado, que não é 0 nem 1. Exceto em situações específicas em que se deseja manter o sinal em um “terceiro estado”, o nível indeterminado pode causar problemas porque a indeterminação tende a se propagar pelo circuito. Um sinal num nível indeterminado pode colocar os dois transistores de um inversor em estado de condução, fazendo com que sua saída também seja indeterminada.

O que se tenta garantir é que os sinais permaneçam indeterminados o menor tempo possível. No diagrama da Figura 5.33, o intervalo em que os sinais estão indeterminados, delimitados pelas linhas verticais, é definido como o *tempo de propagação* do inversor, denotado por T_I .

A resistência através dos transistores, R_P e R_N , não é idêntica e em geral $R_P > R_N$ e portanto a declividade das transições de $1 \rightarrow 0$ e de $0 \rightarrow 1$ é distinta. De acordo com a definição de tempo de propagação, deve-se escolher a mais longa das transições como sendo T_I .

Fan-in e fan-out O tempo de propagação depende da resistência entre VCC (ou GND) e a saída da porta – num *nor* de j entradas, são j transistores P em série entre VCC e a saída. O tempo de propagação também depende do número de entradas alimentadas pela saída – se a saída de uma porta lógica alimenta as entradas de três outras portas, a capacitância total é aproximadamente o triplo daquela de uma única porta, porque a capacitância de capacitores ligados em paralelo é a soma de suas capacitâncias.

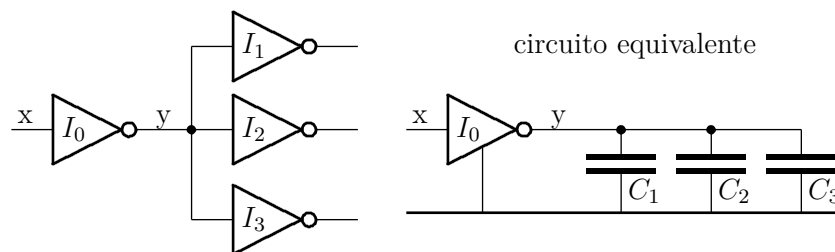


Figura 5.34: Fan-out e a carga capacitiva na saída de um inversor.

A quantidade de portas lógicas alimentadas pela saída de uma porta lógica é chamada de *fan-out*. O tempo de propagação de uma porta lógica aumenta com a carga capacitiva na sua

saída, que é proporcional ao seu *fan-out*. A Figura 5.34 mostra um inversor ligado a três outros inversores. O *fan-out* de I_0 é três e a carga capacitiva na sua saída é $C_y = C_1 + C_2 + C_3$.

O número de entradas de uma porta lógica é seu *fan-in*. Um inversor tem *fan-in* de 1, enquanto que uma porta de três entradas tem *fan-in* 3. A Figura 5.35 mostra o circuito equivalente para uma porta *nand* de duas entradas.

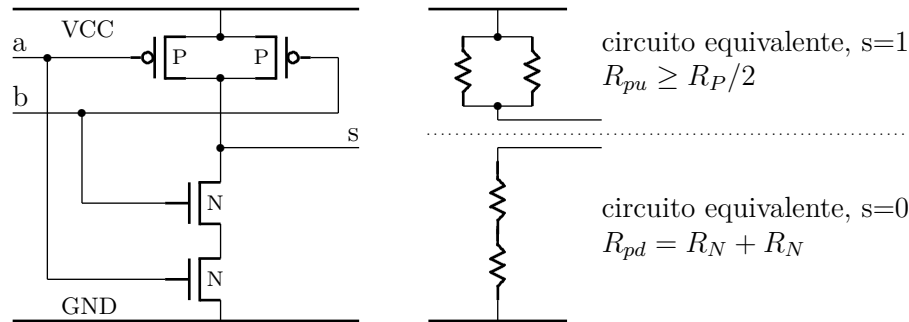


Figura 5.35: *Fan-in* e a carga resistiva na saída de uma porta *nand*.

Quando a saída é 1, a resistência na rede *pull-up*, entre VCC e a carga capacitiva da porta é $R_{pu} = R_P/2$ se as duas entradas estão em 0, ou é $R_{pu} = R_P$ se uma entrada está em 0. Com a saída em 0, as duas entradas estão em 1 e a resistência na rede *pull-down*, entre a carga capacitiva e GND, é $R_{pd} = 2R_N$.

A declividade da exponencial $e^{-t/RC}$, e portanto o tempo de propagação da porta, são afetados pelo *fan-out* ($\propto C$) e pelo *fan-in* ($\propto R$). Na medida em que R e/ou C crescem, mais longo fica o tempo de propagação da porta lógica.

A Figura 5.36 mostra uma porta *nand* de j entradas ligada a k inversores, e os circuitos equivalentes para efeitos de tempo de propagação. Nos interessa o *pior caso* do tempo de propagação, e neste circuito isso ocorre numa transição $1 \rightarrow 0$ na saída do *nand* porque então todas as j entradas estão em 1 e a resistência equivalente, da carga para GND é $R_{eq} = j \times R_N$. A capacitância equivalente, ‘vista’ desde a saída do *nand* é a soma da capacitância de todas portas ligadas ao fio y , e neste caso é $C_{eq} = k \times C$. Para este circuito, o tempo de propagação é determinado por

$$e^{-t/R_{eq}C_{eq}}. \quad (5.18)$$

Para uma porta *nor*, o tempo de propagação é determinado pelos transistores de tipo P, na rede que puxa a saída para 1.

O número de transistores em série é proporcional ao número de entradas numa porta lógica *nand* ou *nor*. O tempo de propagação de uma porta lógica deve ser determinado, de forma conservadora, pela maior resistência equivalente dentre todas as combinações possíveis de entradas, e por isso, o número de entradas de uma porta lógica raramente ultrapassa 4 ou 5. Funções com mais entradas são implementadas com associações de portas de até 5 entradas.

Temos abstraído o comportamento elétrico dos fios, usando superfícies equipotenciais para interligar os componentes. As ligações físicas entre componentes, se curtas, são através de polisilício, e se longas, através de fios em metal. Por ‘fio’ entenda-se que as ligações físicas se parecem mais com chapas retangulares longas e estreitas do que com fios cilíndricos. Por causa da geometria e da proximidade com o substrato, estes fios exibem uma capacitância que é distribuída ao longo de todo o seu comprimento. Além disso, as ligações metálicas tem resistência ôhmica que é proporcional ao comprimento e inversamente proporcional à seção

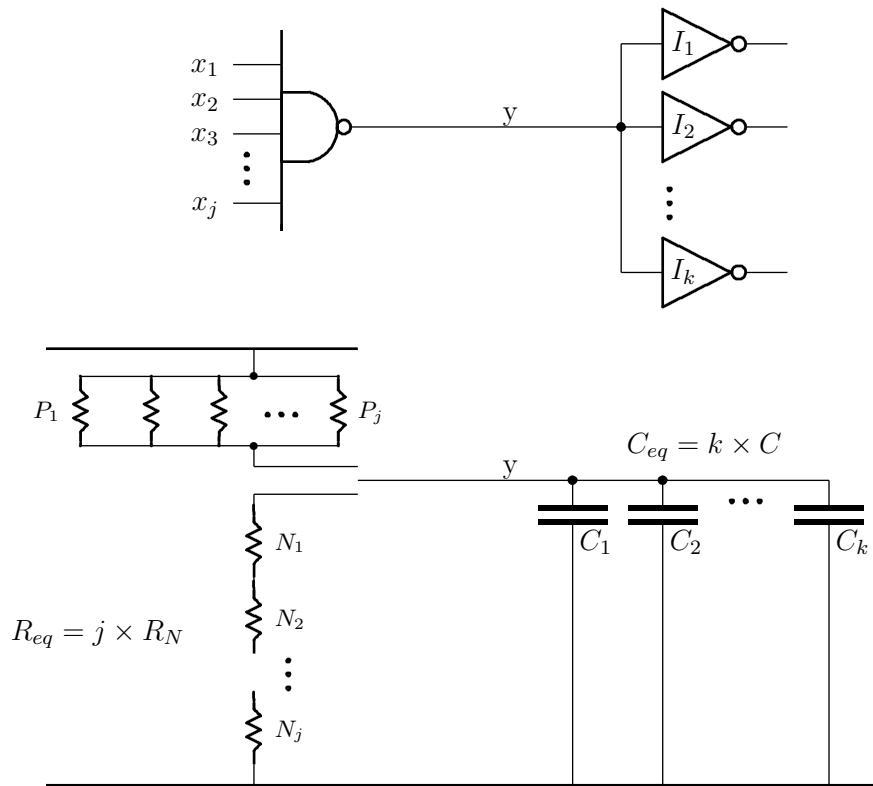


Figura 5.36: *Fan-in e fan-out de uma porta nand.*

reta transversal do condutor. Tanto a capacitância quanto a resistência dos condutores pioram o tempo de propagação dos sinais, porque elas entram como fatores aditivos no R e no C da exponencial $e^{-t/RC}$. Para minimizar estes efeitos, fios longos são segmentados e inversores são colocados ao longo do caminho para restaurar os níveis lógicos dos sinais, bem como para encurtar o tempo de propagação.

Exemplo 5.5 Vejamos a influência do *fan-out* no decodificador de 6 para 64 mostrado da Figura 4.33 (pág. 80), e no Exemplo 4.19. Nos interessa o efeito da carga nas saídas das portas que geram os sinais de endereço do decodificador.

Usando o modelo simplificado do Exercício 4.7, no qual cada entrada adicional ligada à porta piora o tempo de propagação em 10%, temos: (i) as linhas de endereço a_5 e a_4 tem um *fan-out* de 4 porque cada uma delas alimenta dois inversores e duas portas and. Os sinais x_i sofrem um atraso de um fator de $(1,1)^3 = 1,33$; (ii) as linhas de endereço a_3 e a_2 tem um *fan-out* de 4×4 porque cada uma delas alimenta quatro demux-4, e são quatro entradas de seleção em cada demux-4. Os sinais y_i sofrem um atraso de um fator de $(1,1)^{15} = 4,2$; e (iii) as linhas de endereço a_1 e a_0 tem um *fan-out* de 4×16 porque estas alimentam 16 demux-4 e seu fator de atraso é $(1,1)^{63} = 405$.

Um fator de atraso de 4,6 não é bom mas pode se aceitável, dependendo da aplicação. Um fator de 405 é obviamente inaceitável. A solução empregada nos dois casos é acrescentar inversores nestes caminhos longos para reduzir o *fan-out* dos sinais. Por exemplo, as linhas a_3 e a_2 poderiam ser segmentadas em dois e os segmentos ligados por inversores. Nesse caso, seu *fan-out* cairia para a metade e o fator de atraso se reduziria para $(1,1)^7 = 1,95$. As linhas a_0 e a_1 devem ser segmentadas algumas vezes e possivelmente distribuídas aos seus demux-4 como uma árvore, com mais de um nível de inversores

alimentando os demux-4. Sem usar a distribuição em árvore, o último segmento teria um atraso ainda maior como consequência de todos os inversores intercalados justamente para reduzir o atraso. $\triangleleft$

Exemplo 5.6 Ainda considerando o decodificador de 6 para 64 do Exemplo 4.19, vejamos a influência do *fan-in*. Consideremos um modelo igualmente simples para a influência do *fan-in* no qual cada entrada adicional numa porta lógica piora seu tempo de propagação de um fator de 30% se comparado com um inversor.

O primeiro projeto emprega 64 portas and-6 mais seis inversores. Ignorando-se os efeitos do *fan-out* nos inversores, o tempo de propagação de cada and-6 seria $(1, 3)^5 = 3,7$ vezes o de um inversor.

O segundo projeto do decodificador emprega uma árvore de decodificadores 1 para 2 e portanto as portas de duas entradas seriam $(1, 3)^1 = 1,3$, ou 30% mais lentas do que o inversor.

O terceiro projeto emprega uma árvore de decodificadores 2 para 4 e portanto as portas and-3 de três entradas seriam $(1, 3)^2 = 1,7$ vezes mais lentas do que o inversor.

O projeto com portas de largura 6 é uma árvore de um nível com um *fan-out* elevado nas linhas de endereço porque todas as 6 linhas (e seus complementos) são ligadas a 32 entradas. Uma árvore de distribuição dos endereços é fundamental neste caso.

O terceiro projeto é uma árvore de altura 3 e parece ser um compromisso razoável entre os custos associados à *fan-in* e *fan-out*. Uma árvore de distribuição dos endereços para reduzir os atrasos deve ser considerada se este decodificador estiver em algum caminho crítico do circuito.

O que se pode deduzir com estes modelos simplificados, poder-se-ia dizer simplórios, para as influências de *fan-in* e *fan-out*, é que o *fan-out* demanda mais atenção e cuidado na implementação de qualquer circuito que não seja trivial. $\triangleleft$

5.3.3 Energia e Potência

A *potência* é a taxa na qual a energia é dissipada (ou transferida) num circuito elétrico e sua unidade é o *Watt*, que expressa a taxa de trabalho por unidade de tempo, *viz.* 1 watt equivale a 1 joule por segundo ($W = J/s$).

A energia elétrica é *dissipada* num resistor na forma de calor, tal como na resistência de um chuveiro elétrico. A potência de um chuveiro é um bom indicador da temperatura na qual a água sairá do regador; mantida a temperatura ambiente, quanto maior a potência do chuveiro, mais quente será a água no regador. Para uma dada tensão da rede elétrica (127 ou 220V), quanto menor a resistência, maior a corrente que a atravessa, portanto maior a potência dissipada para aquecer a água.

A potência, ou a energia dissipada, num resistor é o produto da tensão aplicada nos seus terminais e pela corrente que circula entre eles:

$$P_R = v \times i = v \times \frac{Q}{t} = R \times i^2 = \frac{v^2}{R}, \quad (5.19)$$

para tensão v , corrente $i = dQ/dt$, carga elétrica Q , resistência R e tempo t .

Quando uma tensão v_F é aplicada às placas de um capacitor, por um tempo suficientemente longo, se diz que o capacitor “está carregado” quando a tensão entre as placas atinge v_F , e a corrente ‘através’ do capacitor é $i_C = 0$. O campo elétrico entre as placas *armazena* uma

quantidade de energia E_C determinada pela Lei de Joule, que é

$$E_C = \int_{-\infty}^t v \times i \, dt = \int_{-\infty}^t v \times C \frac{dv}{dt} \, dt \quad (5.20)$$

A Figura 5.32 mostra os circuitos equivalentes de dois inversores, quando as entradas transicionam de 1 para 0 e de 0 para 1. A Figura 5.37 mostra versões simplificadas daqueles circuitos, considerando apenas o resistor equivalente ao transistor do tipo P na carga do capacitor, e o resistor equivalente ao transistor do tipo N na descarga.

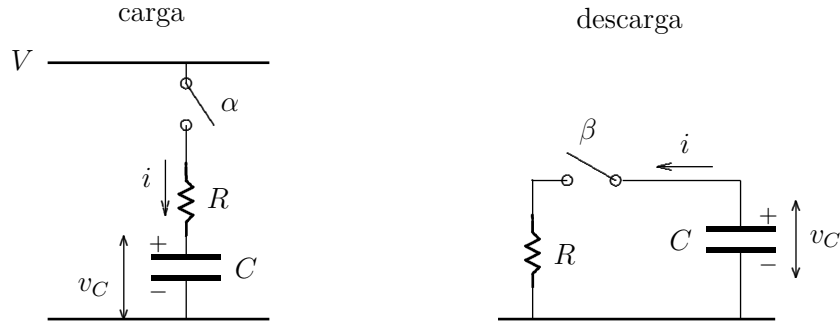


Figura 5.37: Energia acumulada num capacitor.

Supondo que o capacitor está inicialmente descarregado, e que no instante t_0 a chave α é fechada, a tensão $v_C(0)$ é 0 e a corrente através do resistor é $i(0) = V/R$. Decorrido um intervalo suficientemente longo, no instante t_1 a tensão no capacitor se iguala a da fonte de alimentação, $v_C(t_1) = V$, quando então não circula corrente através do circuito e $i(t_1) = 0$.

A energia fornecida pela fonte de alimentação E_F no processo de carga do capacitor é

$$E_F = \int v \times i \, dt = V \int_{-\infty}^{t_1} i \frac{dv}{dt} \, dt = VQ = CV^2 \quad (5.21)$$

sendo Q a carga transferida da fonte para o capacitor. Da Equação 5.12 temos que $Q = CV$ e portanto a energia fornecida pela fonte de alimentação é CV^2 .

Se considerarmos que a tensão é 0 quando $t = -\infty$ e que a tensão é V no instante t_1 , podemos fazer uma troca de variável, reescrevendo a Equação 5.20 como

$$E_C = \int_0^V C \times v \, dv = \frac{1}{2} CV^2 \quad (5.22)$$

A energia dissipada como calor no resistor é $1/2 CV^2$, pois a fonte forneceu CV^2 joules e o capacitor armazenou $1/2 CV^2$ joules.

Com o capacitor inicialmente carregado, $v_C = V$, e a chave β sendo fechada no instante t_2 , inicialmente a corrente através do resistor será $i = V/R$, decrescendo exponencialmente até 0. Se a energia inicialmente armazenado no capacitor é $1/2 CV^2$, então essa mesma quantidade de energia é dissipada no resistor.

Dissipação nas transições Até aqui usamos uma simplificação ao presumir que a resistência no canal de condução dos transistores seja ou zero ou infinita. Considere um transistor do tipo N: quando o transistor conduz, se a tensão no seu *gate* é próxima do valor do nível

lógico 1, a resistência entre fonte e dreno é baixa e o transistor está na *zona de saturação* – a corrente I_{fd} entre os terminais fonte e dreno independe da tensão v_g no *gate*. Se v_g está próxima de 0, então a resistência é elevada – o transistor está na *zona de corte* e a corrente I_{fd} é desprezível.

Durante a transição entre níveis lógicos válidos, a resistência do transistor varia de quase infinita para quase zero, e a corrente I_{fd} é proporcional a v_g . Nesta zona de operação o transistor opera como um amplificador: uma pequena variação na tensão no *gate* provoca uma grande variação na corrente I_{fd} . O efeito da resistência variável se intensifica com transições lentas nas entradas das portas lógicas e por isso os transistores são dimensionados para produzir transições rápidas e assim manter curtos os períodos em modo de amplificação.

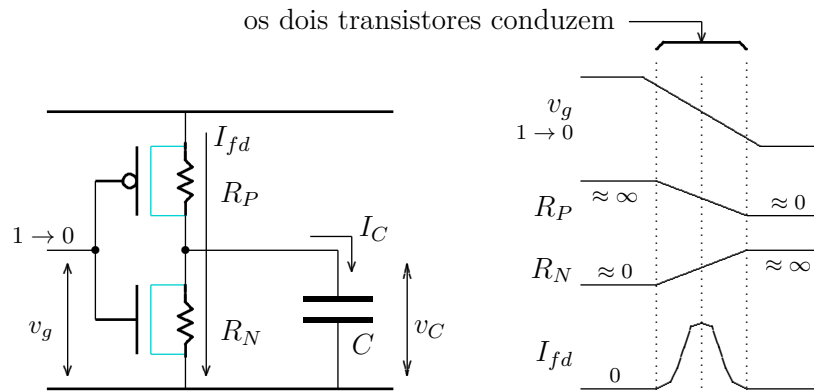


Figura 5.38: Variação dinâmica da corrente num inversor.

A Figura 5.38 mostra um inversor com sua carga capacitiva durante uma transição de 1 para 0 na sua entrada, e quatro curvas: (i) com o comportamento aproximado do sinal na entrada – tensão v_g nos *gates* dos dois transistores; (ii) com a variação na resistência do transistor P, que decresce de um valor muito grande para aproximadamente 0; (iii) com a variação na resistência do transistor N, que cresce de aproximadamente 0 para um valor muito grande; e (iv) a corrente através dos dois transistores. A dissipação de energia $(R_N + R_P)(I_{fd})^2$ ocorre durante esse pico de corrente que circula da fonte para terra.

Potência dinâmica Numa transição $0 \rightarrow 1$, um inversor dissipa $CV^2/2$ joules. A mesma quantidade de energia é dissipada numa transição $1 \rightarrow 0$. Logo, um ciclo completo na saída do inversor ($1 \rightarrow 0 \rightarrow 1$) dissipa CV^2 joules. Se a saída de um inversor é comutada com uma frequência f então a *potência dinâmica* P_{din} dissipada é

$$P_{din} = f C_L V^2 \quad (5.23)$$

para uma capacitância de carga C_L e tensão de alimentação V . Note que a potência dissipada é proporcional à capacitância de carga (*fan-out*) e à velocidade de operação, e é proporcional ao quadrado da tensão de alimentação. É por isso que a tensão de alimentação foi reduzida de 5V nos anos 1980 para 1V nos anos 2010, a frequência de operação aumentou 1000 vezes, enquanto que a potência dissipada aumentou somente 30 vezes.

Nesta discussão estamos ignorando o pico de corrente (I_{fd}) durante as transições – a energia dissipada nas transições deve ser considerada em adição àquela armazenada nos capacitores, e no caso geral, a dissipação nas transições é uma ordem de magnitude menor do que aquela.

A energia dissipada por um circuito deve ser fornecida pela fonte de alimentação ou bateria – um dos maiores componentes do custo de operação de um circuito é a conta de *energia* elétrica. A energia dissipada para efetuar a computação $\mathcal{K}$ depende da quantidade de transições ($1 \rightarrow 0$ e $0 \rightarrow 1$) nos sinais do circuito que efetua $\mathcal{K}$; mantidas as demais condições, esta energia independe da frequência de operação – se a computação demorar um segundo ou uma hora, a energia fornecida pela fonte ou bateria é a mesma.

Lembre que potência é a energia dissipada por unidade de tempo: $P = E/s$ e que a frequência f de um evento é o número de ocorrências por unidade de tempo. Considerando que

$$P_{\text{din}} = f C_L V^2 \quad (5.24)$$

e que C_L e V são fixos, se empregarmos sinais lentos – sinais com frequência baixa, logo f pequeno – para efetuar a computação $\mathcal{K}$, então a potência a ser dissipada é menor, embora tenhamos que esperar mais tempo para obter o resultado desejado. Se empregarmos sinais rápidos, com f grande, então obteremos o resultado em menos tempo mas com um circuito que dissipa mais energia e portanto trabalha mais quente. Assim como num chuveiro, a potência dissipada como calor deve ser removida de alguma forma, por convecção ou resfriamento forçado. Se é necessário resfriamento forçado, então a conta de luz será maior por causa da energia dispendida pelo equipamento adicional.

Potência estática Circuitos CMOS dissipam energia mesmo quando não há transições nas saídas dos circuitos. Por causa das dimensões reduzidas, os isolantes não são perfeitos e portanto em praticamente todos os transistores ocorrem *correntes de fuga*. Para um determinado circuito integrado, a potência estática dissipada é o produto da tensão de alimentação pela corrente suprida pela fonte de alimentação se o circuito estivesse inoperante:

$$P_{\text{est}} = VCC \times I_{\text{fuga}}. \quad (5.25)$$

É esperado que a *potência estática* de um circuito integrado seja algo perto de 20 a 25% da sua potência dinâmica. Em circuitos de alto desempenho, com muitos transistores, e transistores pequenos, a potência estática pode chegar a 50% da potência dinâmica. Um processador que dissipa 100W quando trabalha “a plena carga” pode dissipar 50W quando estiver com carga mínima.

5.4 Modelo de Temporização

Circuitos combinacionais são definidos na Seção 4.1 como sendo circuitos (i) com entradas digitais; (ii) com saídas digitais; (iii) suas saídas produzem os valores determinados pela especificação funcional; e (iv) com as entradas estáveis, decorrido o tempo de propagação do circuito, as saídas estabilizam.

Agora que temos uma melhor compreensão do comportamento dinâmico de circuitos CMOS podemos definir a abstração para o *tempo de propagação* – intervalo desde que as entradas ficaram estáveis e válidas até que a saída fique estável – e para o *tempo de contaminação* – intervalo desde que as entradas tornaram-se inválidas até que a saída torne-se também inválida. Com essa abstração podemos projetar circuitos combinacionais e determinar, ou definir, seus parâmetros de comportamento temporal, sem nos preocuparmos com a enorme complexidade da temporização exata dos circuitos.

Antes de definirmos as abstrações, vejamos uma versão ligeiramente idealizada do comportamento real de um circuito com dois inversores. A Figura 5.39 mostra dois inversores ligados em série e o diagrama de tempo de uma transição de 1 para 0 na entrada do inversor I_1 . O sinal na entrada de I_1 tem a mesma forma de onda que o sinal na sua saída, embora as constantes RC das exponenciais sejam distintas.

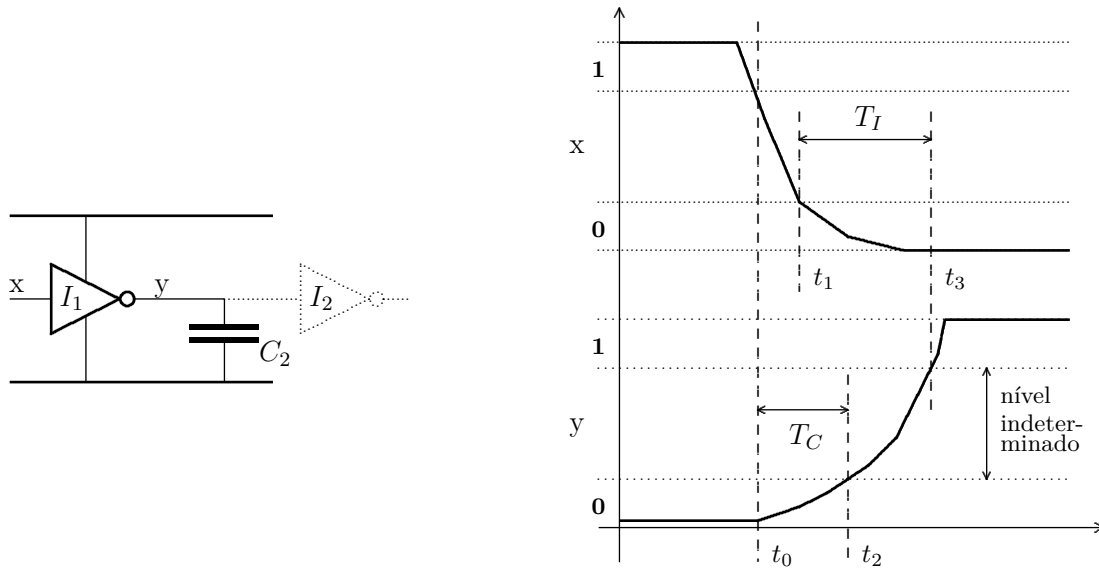


Figura 5.39: Tempo de Propagação (T_I) e Tempo de Contaminação (T_C).

O diagrama mostra, nas linhas pontilhadas horizontais, as faixas de tensão nas quais são reconhecidos os níveis lógicos 0 e 1. Após o início da transição em x , decorre um intervalo $T_C = [t_0, t_2]$ até que y atinja a tensão limite do nível 0. Este intervalo é o *tempo de contaminação* porque a saída y permanece no nível lógico anterior à transição em x – a saída permanece “contaminada pelo passado”.

O intervalo desde que a tensão em x atinge o limite do nível lógico 0 até que o sinal y atravessa a faixa de nível indeterminado e ultrapassa o limite do nível lógico 1 é o *tempo de propagação* do inversor $T_I = [t_1, t_3]$. Como vimos na Seção 5.3.2, os intervalos T_I e T_C dependem do número de entradas do circuito, 1 para um inversor, (*fan-in*) e da capacitância na saída, que é C_2 neste caso (*fan-out*).

5.4.1 Tempo de propagação

Agora que entendemos o comportamento dinâmico de circuitos CMOS é possível definirmos precisamente o que entendemos pelo *tempo de propagação* de um circuito combinacional:

o tempo de propagação de um circuito combinacional é o maior intervalo entre o instante em que todas as entradas ficam estáveis, e o instante em que sua saída fica válida e estável.

A Figura 5.40 mostra um diagrama com o comportamento temporal de um inversor. A saída y fica estável, no valor determinado pela especificação funcional do inversor, somente depois de decorrido o intervalo T_I , que é o tempo de propagação do inversor. Durante o intervalo T_I ,

entre a mudança na entrada e seu reflexo na saída, o nível lógico em y é indeterminado e nada se pode afirmar quanto a y durante este intervalo. A indeterminação é indicada no diagrama pelas regiões hachuradas.

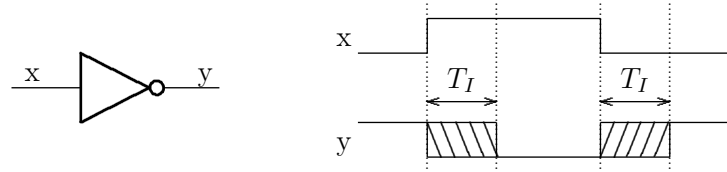


Figura 5.40: Tempo de propagação do inversor.

Lembre que o tempo de propagação de um circuito com vários dispositivos é o máximo dentre o tempo de propagação cumulativo através de todos os caminhos entre as entradas e a saída. A Figura 5.41 mostra a ligação em série de dois inversores e o diagrama de tempos dos sinais nas suas entradas e saídas.

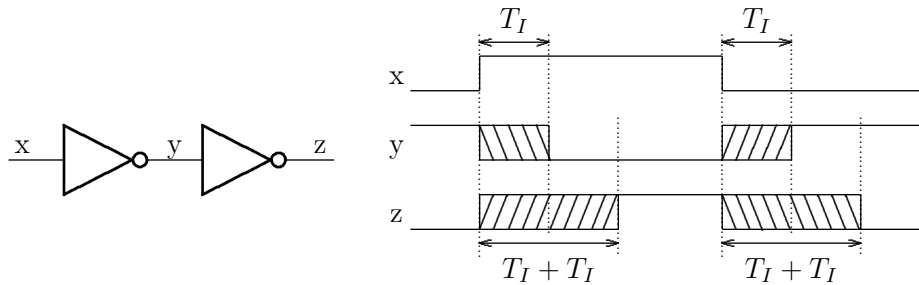


Figura 5.41: Tempo de propagação cumulativo de dois inversores.

O sinal z é uma cópia de x deslocada no tempo, e o deslocamento é de duas vezes T_I , porque o tempo de propagação do segundo inversor somente é contado a partir do instante em que o sinal y fica estável. Assim que a entrada do segundo inversor fica indefinida, sua saída fica indefinida, porque com entradas inválidas, *nada* se pode dizer sobre as saídas de circuitos combinacionais.

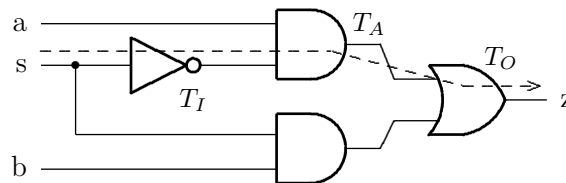


Figura 5.42: Tempo de propagação do multiplexador.

Exemplo 5.7 O tempo de propagação de um multiplexador de duas entradas é determinado pelo caminho mais longo das entradas para a saída, e este caminho é mostrado pela linha tracejada na Figura 5.42.

O tempo de propagação do mux-2 é a soma dos tempos de propagação dos componentes no caminho mais longo, ou no caminho crítico, que neste exemplo é dado por $T_M \geq T_I + T_A + T_O$. $\triangleleft$

5.4.2 Tempo de contaminação

Além do tempo de propagação, o *tempo de contaminação* pode ser usado para especificar o comportamento temporal de um dispositivo.

O tempo de contaminação é o menor intervalo no qual uma saída permanece válida após a entrada tornar-se inválida.

A ‘contaminação’ é o intervalo que decorre entre a alteração numa entrada até que a saída torne-se inválida, e essa demora resulta da carga e/ou descarga das capacitâncias nos nós internos e externos do circuito. Uma vez que uma saída é contaminada, e atinge um nível indeterminado, todos os sinais a ela ligados devem também ser considerados inválidos.

A Figura 5.43 mostra o diagrama de tempo de um inversor e indica o intervalo de contaminação e o tempo de propagação. Assim que o sinal x inicia a transição de 0 para 1, o sinal cruza a região indeterminada, tornando-se portanto inválido, mas a saída do inversor permanece válida durante T_C .

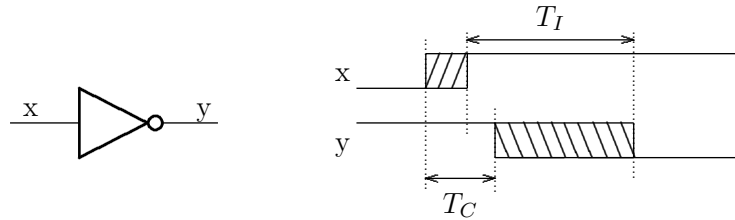


Figura 5.43: Tempo de contaminação T_C e tempo de propagação T_I .

Lembre que o tempo de propagação é o intervalo mais longo entre a entrada válida e a saída também válida, enquanto que o tempo de contaminação é o intervalo mais curto entre a entrada inválida e a saída ainda válida.

Para computar o tempo de propagação de um circuito com vários componentes escolhemos o caminho mais longo, que é aquele que acumula o *maior* atraso das entradas para a saída, porque o tempo de propagação é um limite superior para o intervalo entre entradas válidas e saídas válidas.

Para computar o tempo de contaminação de um circuito escolhemos o caminho com o *menor* valor acumulado, porque o tempo de contaminação é definido como um limite inferior na validade de uma saída com entradas inválidas.

Denotamos o tempo de propagação do circuito X como T_X , enquanto que seu tempo de contaminação é denotado por $T_{C,X}$.

Exemplo 5.8 Considere que o tempo de contaminação das portas *and* nos circuitos da Figura 4.10 (pág. 62) seja $T_{C,A} = 50$ ps. O tempo de contaminação da cadeia de portas é 50ps, determinado pelo caminho $x \leadsto c$, porque este caminho manteria a saída válida por 50ps após as entradas tornarem-se inválidas: $T_{C,\text{cadeia}} \leq T_{C,A}$.

O tempo de contaminação da árvore é de 100ps porque todos os caminhos entre entrada e saída atravessam duas portas lógicas: $T_{C,\text{árvore}} \leq 2T_A$. ◁

Exemplo 5.9 O tempo de contaminação do multiplexador de duas entradas é determinado pelo

caminho mais curto das entradas para a saída, e estes caminhos são mostrados pelas linhas tracejadas na Figura 5.44. O tempo de contaminação do mux-2 é dado por $T_{C,M} \leq T_{C,A} + T_{C,O}$, porque uma alteração nas entradas só se manifesta na saída após a mudança propagar-se através de uma das portas and e da porta or. $\triangleleft$

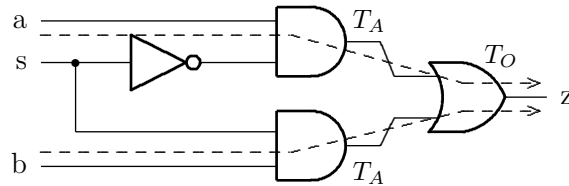


Figura 5.44: Tempo de contaminação do multiplexador.

Exemplo 5.10 Suponha que o tempo de propagação das portas and e or sejam de 20ps, e o de um inversor, 15ps. Com estes valores, podemos computar o tempo de propagação de um multiplexador de 8 entradas, construído com sete multiplexadores de duas entradas.

O tempo de propagação de um mux-2 é de 55ps, e o do mux-8 é de 135ps, e o caminho de propagação mais longo é indicado pela linha pontilhada na figura. Somente o inversor do mux-2 selecionado por s_0 está no caminho crítico: se as entradas mudam ao mesmo tempo então as influências de s_1 e s_2 estabilizam antes do que a perturbação causada por s_0 .

O tempo de contaminação $T_{C,M}$ é indicado pela linha tracejada e é de 40ps. Os sinais s_0 e s_1 não estão no caminho de contaminação: se as entradas mudam ao mesmo tempo, então as perturbações causadas por s_0 e s_1 se manifestam na saída depois que as perturbações causadas por s_2 no terceiro nível. $\triangleleft$

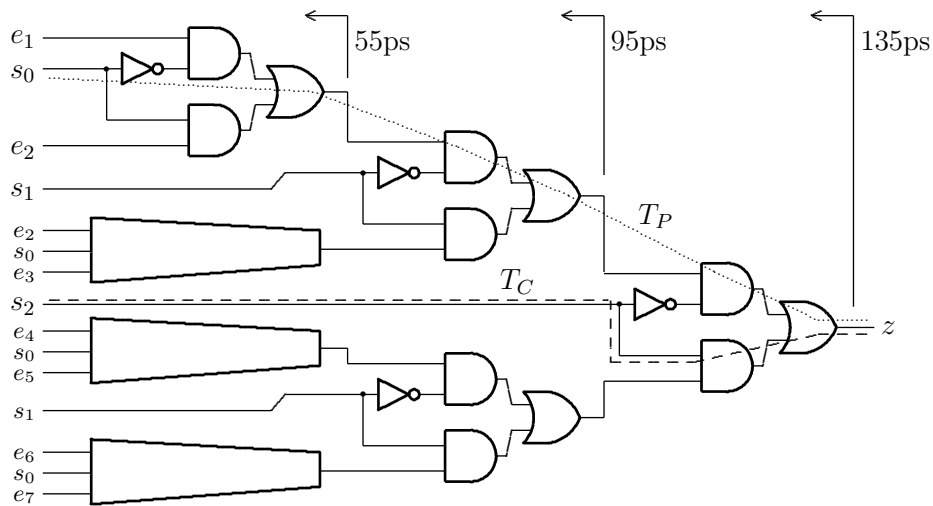


Figura 5.45: Temporização de um multiplexador de 8 entradas.

Exemplo 5.11 Com base nos dados de temporização do Exemplo 5.10, calculemos o tempo de propagação e de contaminação do demux-8 da Figura 4.25. Para a temporização do demux-2, considere o circuito da Figura 4.24.

O caminho mais longo atravessa o inversor do primeiro nível e o and de cada um dos três níveis, portanto o tempo de propagação é $T_D \geq T_I + 3 \times T_A = 75\text{ps}$.

O tempo de contaminação é determinado pelo tempo de contaminação do and no terceiro nível, que é o caminho mais curto da entrada s_0 para as saídas y_i : $T_{C,D} \leq T_{C,A}$. $\triangleleft$

Exemplo 5.12 Consideremos em mais detalhe o comportamento temporal de portas lógicas CMOS. O circuito da Figura 5.46 contém uma porta nand, seguida de um inversor, e a saída do inversor é ligada a duas portas, uma nand e uma nor. Nos interessa o comportamento dos sinais, ou nós da rede, a , c e d .

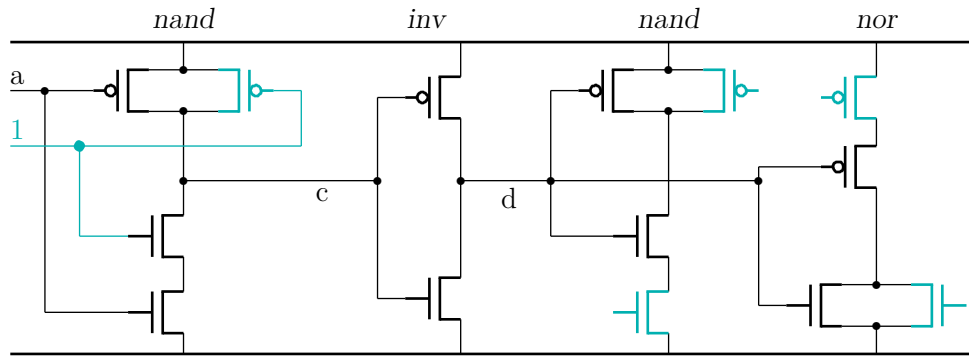


Figura 5.46: Circuito com quatro portas lógicas.

O fan-out do primeiro nand é 1 porque sua saída é ligada somente ao inversor (nó c), enquanto que o fan-out do inversor é 2, porque sua saída é ligada a duas portas (nó d). As transições na saída do inversor são mais lentas porque a capacitância ‘vista’ pelo inversor é o dobro daquela ‘vista’ pelo primeiro nand – o inversor deve drenar a carga de quatro gates, enquanto que o nand drena somente a carga de dois gates.

A Figura 5.47 mostra um diagrama de tempos com os sinais a , c e d . A escala de tempo está um tanto exagerada para facilitar a visualização, e as exponenciais $e^{-t/RC}$ foram aproximadas por retas. Os intervalos em que os níveis ficam indeterminados estão assinalados pelos retângulos de cor cinza.

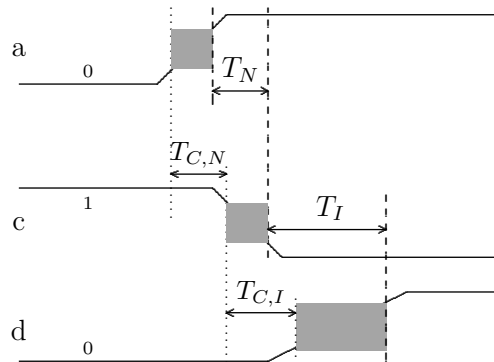


Figura 5.47: Tempo de propagação e de contaminação de quatro portas.

O intervalo de contaminação ($T_{C,N}$) do primeiro nand inicia assim que sua entrada a fica indeterminada, e persiste até que a saída do inversor (c) fique também indeterminada. O intervalo de contaminação do inversor inicia quando o sinal c fica indeterminado, e termina quando o nível em d fica indeterminado. Por causa do *fan-out*, $T_{C,I} > T_{C,N}$.

O tempo de propagação do primeiro nand (T_N) é contado desde que a fica determinado até que c estabiliza. O tempo de propagação do inversor (T_I) é contado desde que c estabiliza até que d estabilize. Também por causa do *fan-out*, $T_I > T_N$. $\triangleleft$

Nos diagramas de tempo mostrados neste texto usamos duas convenções: (i) as bordas de subida e de descida são mostradas como verticais, e (ii) a não ser que mencionado explicitamente, ignoramos os efeitos de *fan-in* e *fan-out* na modelagem de temporização dos circuitos.

O diagrama de tempos mostrado no Exemplo 5.12 pode ser considerado otimista porque as mudanças nas saídas das portas lógicas ocorrem *após* os níveis dos sinais nas entradas ficarem estáveis. Esta é uma escolha arbitrária porque, em circuitos reais, a temporização depende de uma série de fatores, tais como a geometria de cada transistor e de cada fio. As capacitâncias e resistências são determinadas pelas dimensões dos dispositivos, tais como a área dos *gates* dos transistores, o comprimento e espessura dos fios e a espessura dos isolantes entre as partes condutoras dos componentes e o substrato.

As simulações detalhadas das correntes e tensões nos circuitos tomam em conta todos estes fatores, e obviamente têm custo computacional elevado porque os modelos são complexos e o número de componentes em circuitos relevantes varia de milhares a milhões. A análise dos resultados também é custosa porque envolve grandezas contínuas – tensões e correntes – que devem ser discretizadas para que uma interpretação ‘digital’ seja possível, e para verificar se o circuito atende à sua especificação funcional e de temporização.

5.4.3 Comportamento transitório

O tempo de contaminação é uma espécie de memória de curto prazo e este comportamento será explorado adiante. O tempo de contaminação ajuda a explicar certos comportamentos inesperados. Nos circuitos mostrados na Figura 5.48, quando se considera seu comportamento estático, o sinal c é *sempre* 0, enquanto que o sinal r é *sempre* 1, se todos os dispositivos se comportam de acordo com nossa definição para circuitos combinacionais.

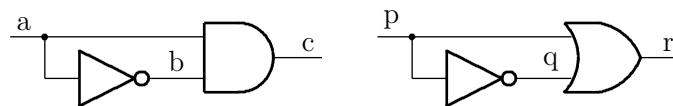


Figura 5.48: Circuitos com comportamento estático bem definido.

Contudo, se considerarmos o comportamento dinâmico destes circuitos, as afirmações da frase anterior são falsas durante um intervalo relativamente curto após as transições nas entradas. Vejamos o circuito com a porta *and*. Para simplificar, suponha que o tempo de contaminação da porta *and* e do inversor são idênticos. A Figura 5.49 mostra um diagrama de tempo que considera somente os efeitos do tempo de contaminação, e que $T_I = T_A = 0$.

No intervalo α , a entrada b é zero e a saída é aquela esperada. No intervalo β , o atraso de contaminação mantém a saída do inversor válida após a alteração na sua entrada, as duas entradas da porta *and* são 0 e sua saída permanece em 0. No intervalo γ a saída do inversor

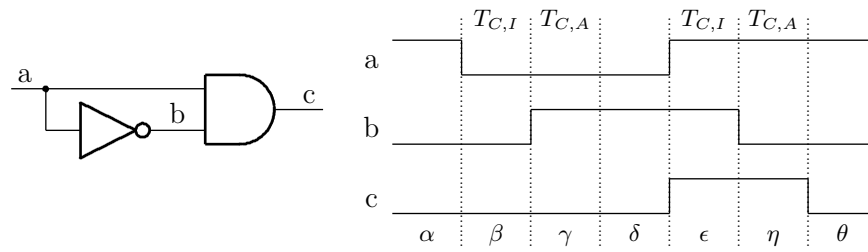


Figura 5.49: Circuito com comportamento dinâmico distinto do estático.

muda para 1 e c permanece em 0. No intervalo δ os sinais ficam estáveis. No intervalo ϵ o atraso de contaminação mantém a saída do inversor em 1, enquanto a entrada a muda para 1 – com as duas entradas da porta *and* em 1, sua saída muda para 1 porque estamos considerando que $T_I = T_A = 0$. No intervalo η , a contaminação da porta *and* mantém sua saída em 1 durante $T_{C,A}$. No intervalo θ , a saída do *and* reflete sua entrada b em 0. O pulso espúrio em c dura $T_{C,I} + T_{C,A}$.

Exemplo 5.13 O autor recomenda enfaticamente ao leitor que desenhe, sem demora, um diagrama de tempo similar ao da Figura 5.49 para o circuito com a porta *or*. <

A definição de circuito combinacional que adotamos considera o seu comportamento correto após o decurso do seu tempo de propagação, caso sua especificação funcional seja satisfeita pelas saídas. Isso significa que o comportamento descrito no diagrama de tempos da Figura 5.49 é uma ocorrência observável e indesejada, mas quando se considera o tempo de propagação das portas lógicas, aquele circuito se enquadra na definição de circuito combinacional.

A Figura 5.50 expande o diagrama da Figura 5.49 com a inclusão do tempo de propagação, que é o intervalo mais longo necessário para a saída estabilizar, após as entradas estabilizarem. Os efeitos da contaminação estão indicados por traços espessos, e o tempo de propagação por regiões hachuradas.

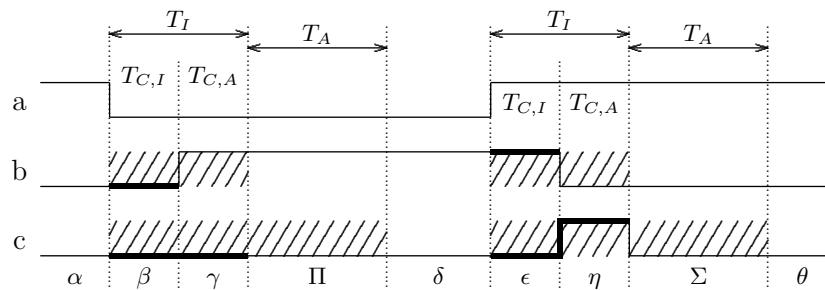


Figura 5.50: Comportamento dinâmico considerando o tempo de propagação.

Nos intervalos β e γ que se estendem por T_I segundos, a saída c é inválida porque a entrada b do *and* é inválida. No intervalo Π , b fica válido mas a saída c é inválida durante o intervalo T_A , que é o tempo de propagação da porta *and*. No intervalo δ a saída é aquela determinada pelas especificações do inversor e da porta *and*. Nos intervalos ϵ e η a saída c é inválida porque b é inválido, em que pese a ocorrência do pulso em c durante η . Decorrido o segundo T_A (Σ), a saída fica válida e correta no intervalo θ .

O sinal b sofre contaminação por causa de a nos intervalos β e ϵ . O sinal c sofre contaminação em β por causa de b e em γ por causa de a e de b . A contaminação em c , no intervalo ϵ decorre do nível em a , e no intervalo η decorre do nível (contaminado) em b .

Exemplo 5.14 Considere o circuito da Figura 5.51, que detecta tanto as bordas de subida quanto as bordas de descida na sua entrada. A cada borda no sinal e , o xor produz um pulso em sua saída porque suas entradas estão distintas. Desenhe um diagrama de tempos como o da Figura 5.49 – somente considerando o tempo de contaminação – para certificar-se de que o comportamento é o esperado. $\triangleleft$

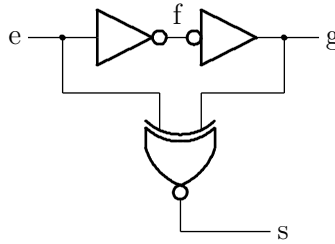


Figura 5.51: Detector de bordas com porta xor.

Exemplo 5.15 Vejamos uma aplicação para o comportamento dinâmico do circuito com a porta or e dois inversores ou um buffer, como o mostrado na Figura 5.52. Para simplificar a discussão, suponha que o tempo de propagação da porta or é zero.

Considere um pulso de curta duração, ou um espículo (glitch), de $1 \rightsquigarrow 0 \rightsquigarrow 1$, que dura uma fração do tempo de propagação do buffer T_B . O buffer atrasa o espículo no sinal q o suficiente para que a entrada p do or estabilize, de tal forma que a saída r não apresenta o espículo, e assim o pulso é ‘filtrado’ pelo circuito. Isso é o que mostra o lado esquerdo do diagrama de tempos. Contudo, o lado direito do diagrama apresenta dois espículos, um reproduzido a partir do sinal original e um segundo, que é a versão atrasada pelo buffer. Melhora um lado, piora o outro. $\triangleleft$

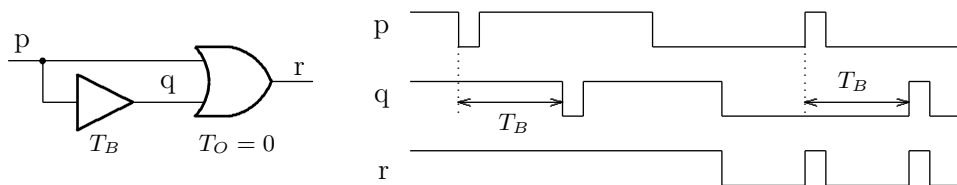


Figura 5.52: Filtro de espículos com porta or.

5.4.4 Implicativos Primários, Implicativos Essenciais e Temporização

O multiplexador mostrado na Figura 5.53 exibe um comportamento similar ao circuito com a porta or da Figura 5.48. Quando as entradas a e b estão ambas em 1, os sinais m e n refletem, a menos do tempo de propagação das portas and, os valores de $\bar{s}$ e s . Dependendo de como este multiplexador é empregado, o comportamento do sistema pode ser errático por causa do pulso espúrio causado pela ‘corrida’ entre os sinais derivados de s nas entradas da porta or – a versão atrasada de s em n contra a versão atrasada de $\bar{s}$ em m . A detecção destas corridas é problemática porque, durante a depuração, pode ser difícil de reproduzir as condições que provocam o pulso espúrio na saída do multiplexador.

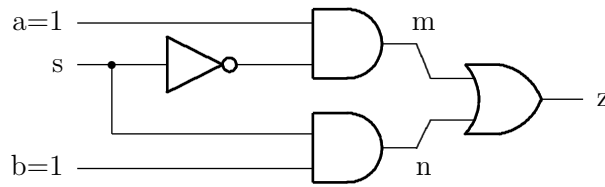


Figura 5.53: Multiplexador com pulso espúrio na saída.

A Figura 5.54 mostra o Mapa de Karnaugh para o multiplexador de duas entradas. O pulso espúrio na saída do multiplexador ocorre quando as entradas passam da configuração correspondente à célula 7 ($\langle s, a, b \rangle = 111$) para a configuração da célula 3 ($\langle s, a, b \rangle = 011$). Se um agrupamento adicional for incluído no mapa, cobrindo as células 3 e 7, o pulso indesejado não ocorrerá. Este agrupamento corresponde a $a \wedge b$ e mantém a saída em 1, independentemente do que ocorra com s . O circuito é mais complexo, com a adição de uma porta *and* e de uma entrada adicional no *or*, mas seu comportamento dinâmico é exatamente o desejado, e independe dos efeitos transitórios causados pelo tempo de contaminação.

		$a\ b$			
		00	01	11	10
s	0	1	1	1	1
	1	0	1	1	0

Figura 5.54: Mapa de Karnaugh do multiplexador bem comportado.

A saída do multiplexador de duas entradas pode ser descrita em termos de uma soma de produtos com os termos

$$z(s, a, b) = (\bar{s} \wedge \bar{b} \wedge a) \vee (\bar{s} \wedge b \wedge a) \vee (s \wedge b \wedge \bar{a}) \vee (s \wedge b \wedge a) \quad (5.26)$$

Do ponto de vista lógico, cada um dos termos da soma implica $z = 1$, embora somente a disjunção dos quatro seja equivalente a z . Assim,

$$(\bar{s} \wedge \bar{b} \wedge a) \Rightarrow (z = 1)$$

O mesmo vale para a versão simplificada, como a do mapa da Figura 5.54:

$$(\bar{s} \wedge a) \Rightarrow (z = 1)$$

Os termos individuais da soma de produtos são chamados de *implicativos* porque, quando as variáveis fazem o termo igual a 1, esta mesma combinação de variáveis implica em que o resultado da soma de produtos seja 1.

Todos os quatro termos da Equação 5.26 são termos implicativos. Os três termos simplificados obtidos do Mapa de Karnaugh $\langle (\bar{s} \wedge a), (s \wedge b), (a \wedge b) \rangle$ são chamados de *implicativos primários*, porque não podem ser eliminados por simplificação. Um termo da soma de produtos que não pode ser agrupado a nenhum outro, sendo portanto imprescindível para representar uma função é chamado de *implicativo essencial*. Para uma função com um certo número de mintermos, pode haver mais de um conjunto de implicativos primários que simplificam aquela função.

O mapa da Figura 5.55 mostra três implicativos essenciais que não podem ser eliminados por simplificação: um que cobre o mintermo 0, um que cobre os mintermos 3 e 7, e o que cobre os mintermos 12, 13, 14 e 15. O implicativo primário – mostrado com o traço fino – que cobre os mintermos 7 e 15 não é essencial. Este implicativo primário deve ser implementado caso a aplicação do circuito especifique um comportamento dinâmico sem corridas, que neste exemplo são transições $1 \rightsquigarrow 0$ em x . No caso geral, a saída de um circuito implementado com todos os implicativos primários é livre dos pulsos espúrios provocados por corridas.

		$z w$			
		00	01	11	10
$x y$	00	0 ⁰ 1	1 ¹ 0	3 ³ 1	2 ² 0
	01	4 ⁴ 0	5 ⁵ 0	7 ⁷ 1	6 ⁶ 0
	11	12 ¹² 1	13 ¹³ 1	15 ¹⁵ 1	14 ¹⁴ 1
	10	8 ⁸ 0	9 ⁹ 0	11 ¹¹ 0	10 ¹⁰ 0

Figura 5.55: Função de quatro variáveis e seus implicativos.

5.4.5 Lógica Restauradora

O comportamento dinâmico destes circuitos é distinto daqueles previstos pela nossa abstração para dispositivos combinacionais. Nosso modelo abstrai particularidades dos circuitos CMOS que se manifestam em circuitos reais. Circuitos CMOS são projetados para ‘melhorar’ os sinais em suas saídas, com relação aos sinais nas suas entradas. Os níveis lógicos nas saídas das portas lógicas são “mais fortes” do que aqueles presentes em suas entradas porque as redes *pull-up* e *pull-down* puxam as saídas para VCC e para GND, que são 1 e 0 fortes. Por causa deste comportamento os circuitos CMOS são chamados de *restauradores* (*restoring logic*) porque entradas fracas resultam em saídas fortes. Veja a Figura 5.33: assim que um sinal de entrada sai da região indeterminada, a saída é puxada para 1 ou para 0, porque os transistores funcionam como amplificadores: um nível elétrico fraco numa entrada é amplificado até os níveis das fontes de 0 ou de 1.

Exercícios

Ex. 5.5 Desenhe um diagrama de tempos similar ao do Exemplo 5.15 para um filtro com uma porta *and*.

Ex. 5.6 Ainda com referência ao Exemplo 5.15, desenhe outro diagrama de tempos para um filtro que combina uma porta *or* e uma porta *and*.

Ex. 5.7 Qual é a largura máxima dos pulsos que podem ser filtrados pelos circuitos do Exemplo 5.15 e dos Exercícios 5.5 e 5.6?

Ex. 5.8 Mostre como implementar portas lógicas *and*, *or*, *xor*, *nand* e *nor* de 16 entradas empregando somente portas *nand* e *nor* de 4 ou menos entradas e inversores. Sua implementação deve minimizar o tempo de propagação. *Pistas*: (i) quais funções são associativas? (ii) árvores tem altura $\propto \log(N)$ enquanto que vetores tem comprimento $\propto N$.

5.5 ROM – *Read-Only Memory*

Memórias somente de leitura (*Read-Only Memory*, ROM) são memórias que podem ser lidas mas não podem ser alteradas enquanto o circuito está em operação normal, ao contrário das memórias de leitura e escrita (*Random Access Memory*, RAM) – cujos conteúdos podem ser alterados durante a operação normal do circuito.

Qualquer função $F : \mathbb{B}^n \mapsto \mathbb{B}$ pode ser representada como uma soma de produtos

$$F(e_0, e_1, \dots, e_n) = \bigvee [\bigwedge (e_0, e_1, \dots, e_N)].$$

Uma função de n variáveis pode ser implementada com um multiplexador de $N = 2^n$ entradas. Cada uma das N entradas é qualificada pela conjunção de n literais – a k -ésima entrada é qualificada pela conjunção dos literais que representam o número k . A Figura 5.56 mostra a implementação da função F com um multiplexador de n entradas. Cada porta *and* seleciona, e qualifica, um dos mintermos da soma de produtos.

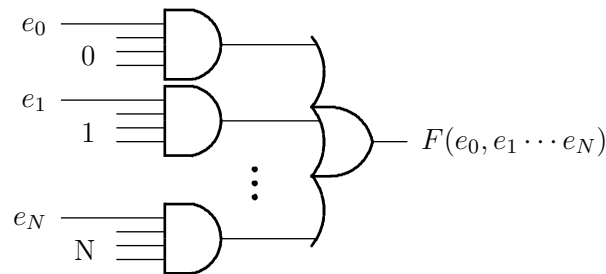


Figura 5.56: Função implementada com um multiplexador de n entradas.

Um circuito com a mesma organização de um multiplexador é usado para implementar uma memória ROM. Um *decodificador de linha*, similar ao conjunto de portas *and* do multiplexador, é usado para selecionar uma das linhas da matriz. Uma porta *or* é utilizada para ‘coletar’ os bits em 1 em cada coluna da matriz. A próxima seção mostra como implementar um decodificador de linha.

5.5.1 Decodificador de Linha

Vejamos como uma porta *or* com várias entradas pode ser implementada usando somente transistores do tipo N. Neste circuito violaremos uma das regras de projeto e usaremos um transistor do tipo N ligado à VCC – como estes transistores conduzem mal o nível lógico 1, dizemos que o sinal do dreno deste transistor é um *sinal fraco*. Quando um sinal fraco é ligado a um sinal forte, o sinal forte sempre prevalece e se sobrepõe ao nível do sinal fraco.

O circuito da Figura 5.57 mostra uma porta *or* com 4 entradas. O transistor no topo do circuito é chamado de *pull-up* porque sua função é puxar, fracamente, o sinal r para o nível

lógico 1. Se uma ou mais das entradas é 1, então o sinal r é puxado para 0 pelo(s) transistor(es) tipo N e a saída s é 1. Se todas as entradas forem 0, então o *pull-up* puxa r para 1 e portanto a saída s fica em 0.

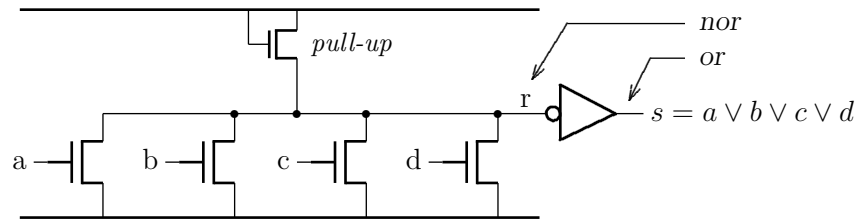


Figura 5.57: Função *or* de quatro entradas.

Necessitamos de uma porta *and* para implementar um decodificador de linhas. Se a memória é uma matriz com L linhas, então o decodificador necessita $\ell = \log_2(L)$ bits de endereço, e cada uma das linhas é selecionada pela conjunção dos bits apropriados do endereço. O circuito da Figura 5.58 mostra a função *and* implementada com a porta *nor*. O circuito dentro das linhas pontilhadas é a porta *nor* da Figura 5.57.

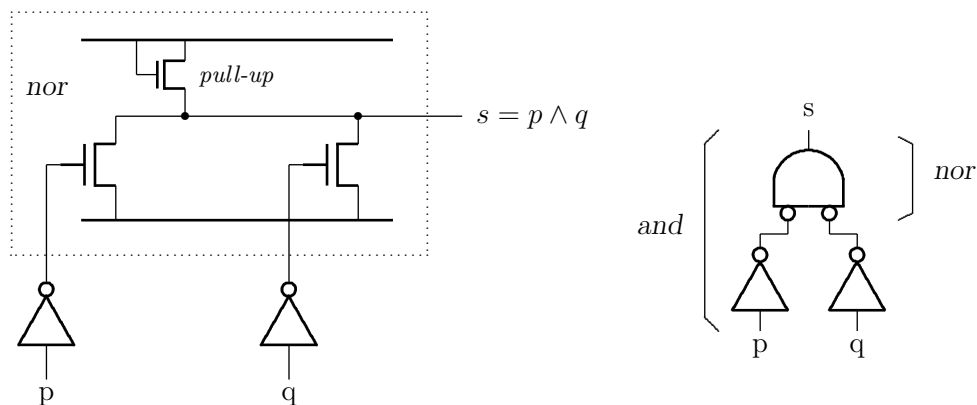


Figura 5.58: Função *and* de duas entradas.

Um decodificador de 2 bits de endereço $\langle a_1, a_0 \rangle$ para 4 linhas $\langle l_3, l_2, l_1, l_0 \rangle$ é mostrado na Figura 5.59. Nesta figura VCC é indicado como uma barra horizontal, e GND como um triângulo achatado. Cada linha do circuito corresponde a uma porta *and* de duas entradas.

A caixa pontilhada mostra a representação para diagramas de bloco. Este decodificador é uma coleção de *ands* e por isso é chamado de *and array*. Cada uma das L linhas da matriz corresponde a um mintermo e qualquer função de até ℓ variáveis pode ser implementada pela disjunção dos mintermos apropriados, como veremos no que se segue.

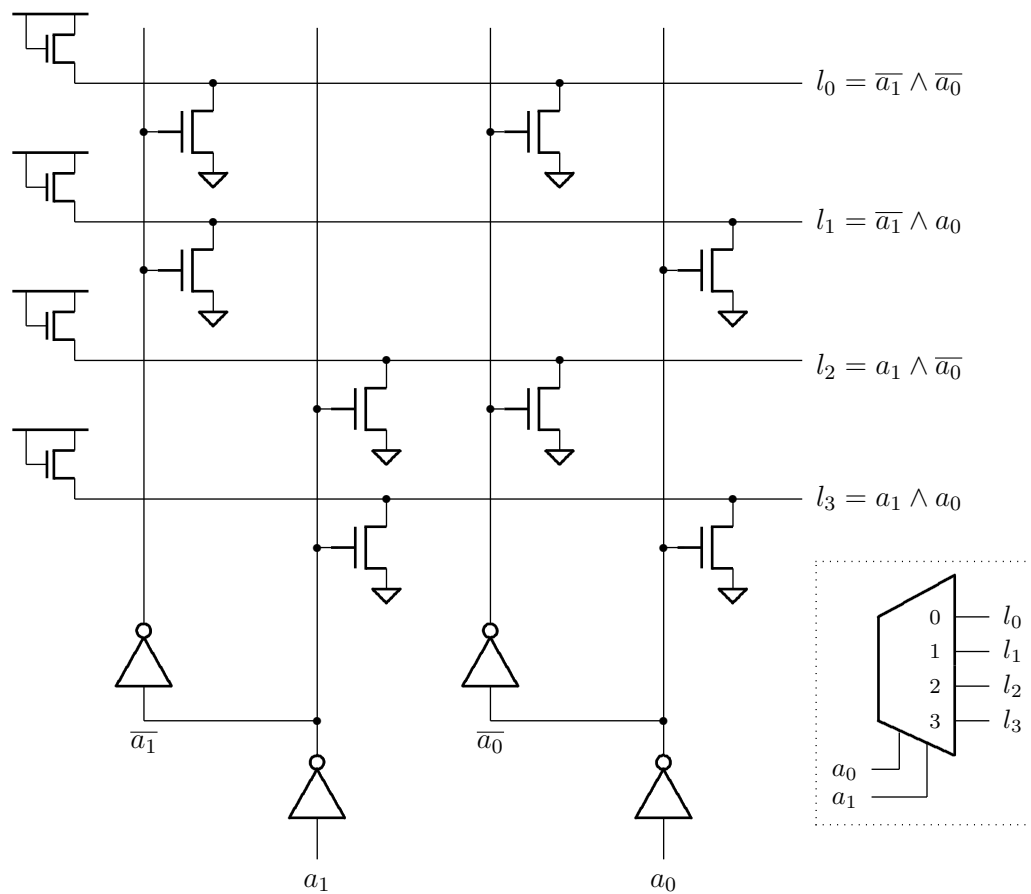


Figura 5.59: Decodificador de 2 bits para 4 linhas.

A Figura 5.60 mostra o decodificador com as entradas $\langle a_1, a_0 \rangle = 2$, seleccionando a saída 2. Os transistores que não conduzem são mostrados com o canal como um linha pontilhada, e os que conduzem são mostrados com linhas contínuas. Como os dois transistores da linha l_2 estão com seus *gates* em 0, e portanto não conduzem, o *pull-up* desta linha ‘puxa’ l_2 para 1. Nas outras linhas há pelo menos um transistor puxando o sinal no fio horizontal para 0, como indicam as setas tracejadas.

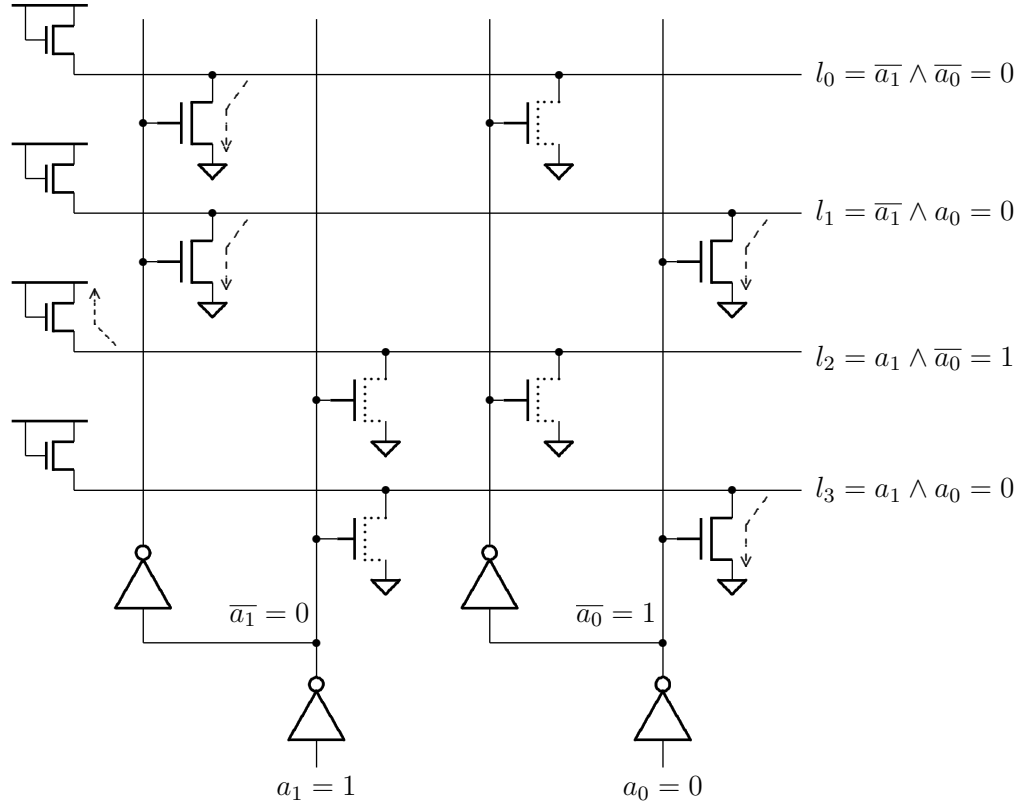


Figura 5.60: Seleção da saída 2 no *and-array* (decod-4).

5.5.2 Matriz de Dados da ROM

Uma memória ROM é construída como uma matriz de L linhas por C colunas, e em cada intersecção há um transistor para indicar que o bit correspondente à posição (i, j) é 1; a ausência de um transistor na intersecção (m, n) indica que aquele bit é 0.

Uma função *or* similar à da Figura 5.57 pode ser usada para implementar a disjunção dos mintermos gerados pelo decodificador. Com várias disjunções obtemos um *or array*, que combinado com um *and array*, resulta numa coleção de somas de produtos, que é justamente a matriz de memória M que estamos a construir. A Figura 5.61 mostra uma memória com $n + 1$ bits de endereço e $w + 1$ bits de dados tais que o valor de cada um dos bits de dados r_k resulta da disjunção de um certo número de mintermos

$$r_k = \bigvee l_j, j \in [0, m], k \in [0, w] \quad (5.27)$$

e cada mintermo l_j é uma conjunção de endereços ou de seus complementos

$$l_j = \bigwedge \dot{a}_i, i \in [0, n]. \quad (5.28)$$

Na Equação 5.28, $\dot{a}_i$ representa um literal que pode ser a_i ou $\overline{a_i}$.

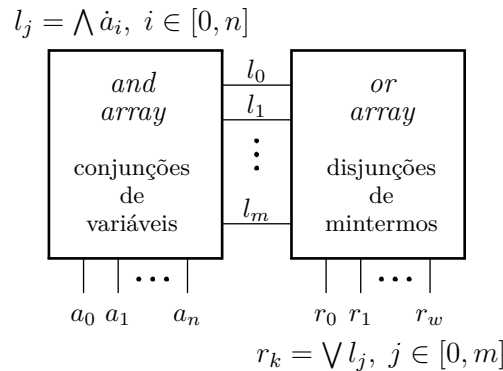


Figura 5.61: Memória ROM como soma de produtos.

A Figura 5.62 mostra o circuito de uma memória ROM com 4 bits de altura e 1 bit de largura. O decodificador ativa exatamente uma das quatro entradas do *or* – estas saídas são as linhas da matriz 4×1 . Se na intersecção de uma linha com a coluna houver um transistor, quando aquela linha é seleccionada o transistor conduz, r fica em 0 e a saída s em 1. Se não há um transistor na intersecção, quando a linha é seleccionada, o *pull-up* mantém o sinal r em 1 e a saída s fica em 0. O inversor na saída restaura e reforça o sinal na saída da ROM.

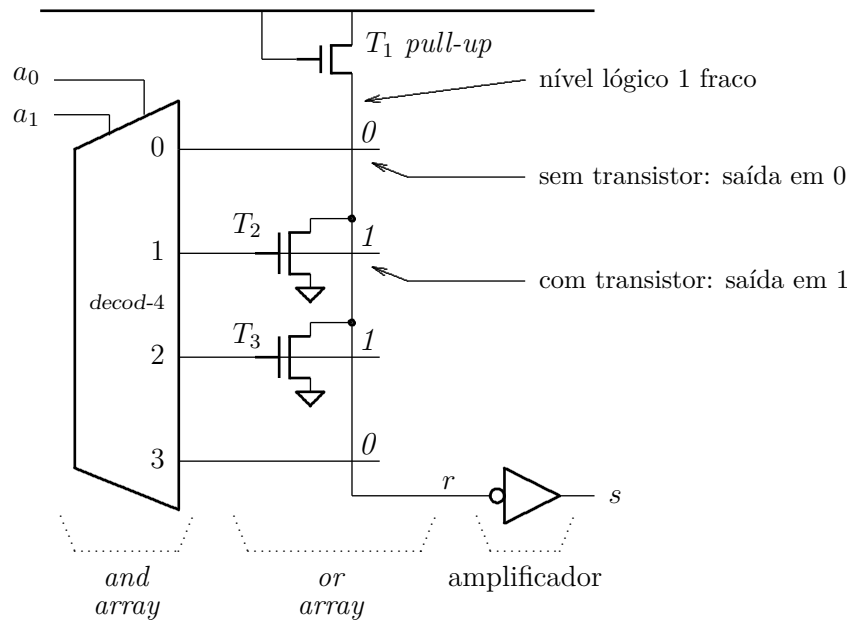


Figura 5.62: ROM 4x1.

O circuito da Figura 5.62 é uma memória ROM com 4x1 bits – *quatro palavras de um bit*. Esta nomenclatura é a usual para descrever a organização externa de memórias: altura×largura, sendo a ‘altura’ o número de palavras da memória, e a ‘largura’ o número de bits de cada palavra. Para escolher uma das quatro palavras são necessárias duas *linhas de endereço*, que são os sinais a_0 e a_1 . A saída da ROM apresenta a palavra seleccionada, e neste exemplo é o valor do bit seleccionado por $\langle a_1, a_0 \rangle$. Qual é a função de duas variáveis, $f(a_0, a_1) = s$, implementada nesta memória ROM?

No topo da figura, o transistor T_1 é um transistor do tipo N e seus terminais *gate* e fonte estão ligados à VCC, e seu terminal dreno ligado à porta *or*. Transistores tipo N não conduzem bem o nível lógico 1 e quando não há um transistor tipo N puxando a linha de bit para zero (T_2 ou T_3), T_1 mantém um nível lógico 1 fraco. O inversor na saída garante níveis lógicos fortes para os circuitos que usam a saída da ROM. Da forma como desenhada, uma linha vertical da matriz, que é uma disjunção, é chamada de *linha de bit* (*bit line*), enquanto que as saídas do decodificador são chamadas de *linhas de palavra* (*word line*) porque selecionam uma palavra da memória. Lembre que esta matriz tem somente uma coluna.

Nas linhas 0 e 3, o cruzamento da linha de seleção com a linha de bit significa que não há contato elétrico entre estes dois sinais. A Figura 5.63 (a) mostra uma versão simplificada do diagrama da ROM 4x1. Quando a linha 0 é selecionada, não há ligação entre a saída do decodificador e a linha de bit, e portanto a saída da ROM é o complemento do 1 fraco no dreno de T_1 .

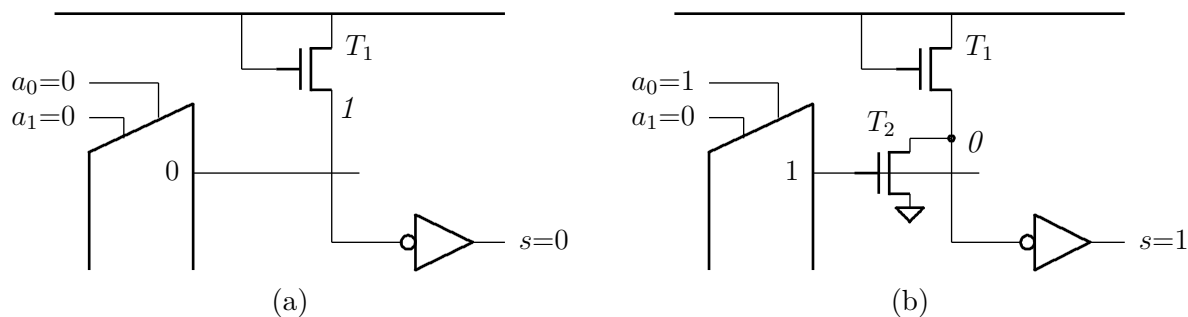


Figura 5.63: ROM: (a) bit em 0; (b) bit em 1.

Nas linhas 1 e 2, há um transistor tipo N, com seu *gate* ligado à saída do decodificador – com o dreno ligado à linha de bit, e a fonte ligada à GND. Quando a linha 1 é selecionada, o *gate* de T_2 fica em 1 e a linha de bit é ligada à GND. Esta ligação produz um 0 ‘forte’ na linha de bit, que se sobrepõe ao sinal fraco produzido por T_1 . A Figura 5.63 (b) mostra o transistor na linha 1.

Memórias ROM permitem implementações compactas de tabelas verdade, de uma forma similar àquela apresentada na Seção 4.3.1. O circuito da Figura 5.62 implementa a tabela verdade da função *xor*: $s = a_0 \oplus a_1$.

Exemplo 5.16 Vejamos como implementar um somador completo com uma memória ROM. A Figura 5.64 mostra a tabela verdade de um somador completo e a implementação das funções soma e vai-um com uma memória ROM de oito linhas de dois bits cada, ou uma memória ROM 8x2. Há uma correspondência exata entre as linhas da tabela e as ligações nas intersecções. Os oito produtos são implementados no decodificador, enquanto que as somas (disjunções) são implementada nas colunas.

Considere a soma de produtos da função vai-um. O decodificador gera os oito mintermos, um por linha, enquanto que o *or* da coluna soma os produtos distintos de 0, que são as linhas com transistores nas intersecções. ◁

Memórias como estas permitem a implementação rápida e eficiente de funções lógicas com muitas entradas. A implementação de uma função de oito entradas com Mapas de Karnaugh e portas lógicas é um processo tedioso e complexo, enquanto que a geração de uma ROM para esta função consiste em ‘copiar’ a tabela verdade da função para a ROM. Veja o Exercício 5.13.

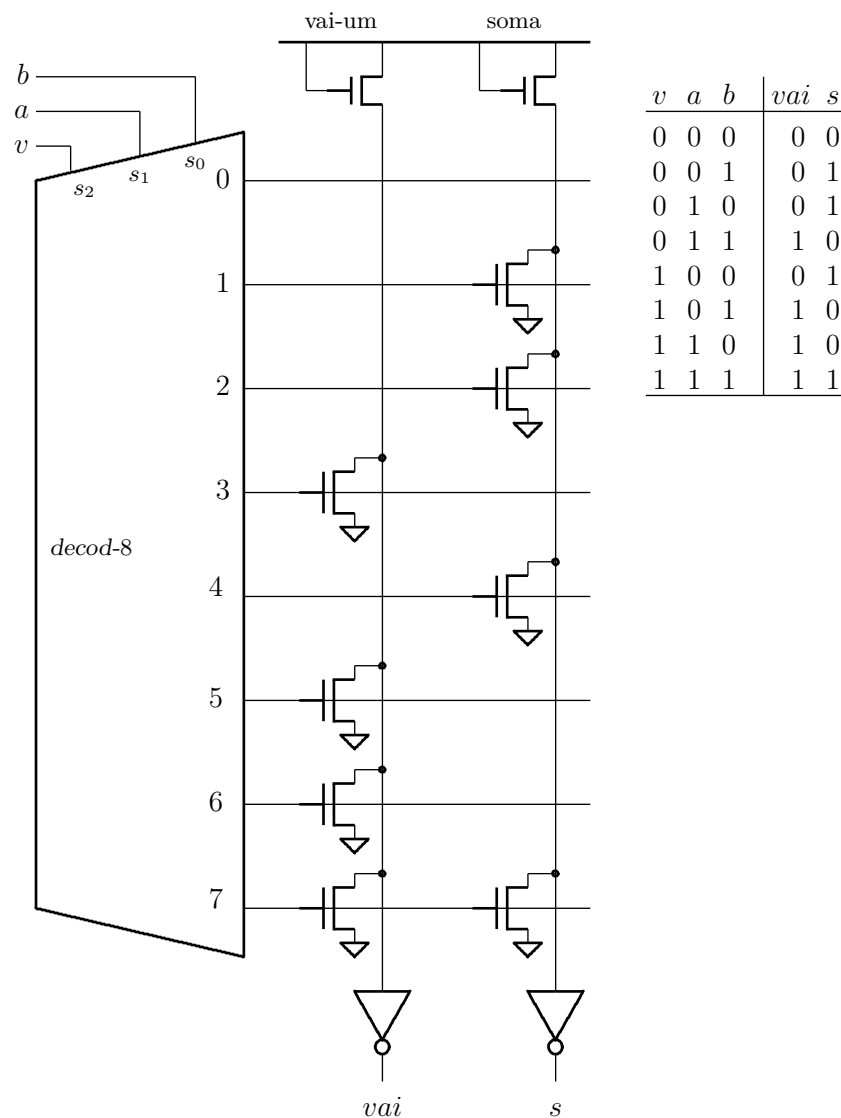


Figura 5.64: Somador completo implementado com uma ROM 8x2.

Na medida do possível, tenta-se projetar a memória para que ela seja mais ‘quadrada’ do que “alta e estreita”, porque linhas longas incorrem em tempo de propagação também longo. Veja o Exercício 5.11.

Memórias ROM podem ser gravadas uma única vez, durante sua fabricação, no que é chamado de “programação por máscara”. Todas as intersecções entre linhas e colunas possuem transistores mas somente aquelas posições com bits em 1 tem seus *gates* ligados à linha de seleção. Isso é obtido num dos passos finais do processo de fabricação, no qual uma das máscaras da metalização só contém conexões para os *gates* que correspondem a 1s na ROM.

Memórias PROM são *Programmable ROM* se os transistores nas intersecções podem ser programados eletricamente, ao invés de somente durante a fabricação. Memórias EPROM são *Erasable Programmable ROM* e depois de programadas eletricamente podem ser apagadas, tipicamente pela exposição à luz ultravioleta. Memórias EEPROM são *Electrically-Erasable Programmable ROM* e podem ser apagadas eletricamente. Estas últimas são extremamente

úteis porque permitem a reprogramação sem que seja necessário remover o circuito integrado de memória para sua exposição a uma fonte de luz. Memórias FLASH são um tipo de EEPROM.

Exercícios

Ex. 5.9 Projete um decodificador de 3 para 8 linhas tomando por base o circuito da Figura 5.59. Compare o número de transistores da sua implementação com o de uma implementação que usa portas *and*-3 e inversores.

Ex. 5.10 Projete uma memória ROM que implementa as funções *and*, *nand*, *or*, *nor* com 4 entradas. Compare o número de transistores necessários para a implementação das portas lógicas como visto na Seção 5.2.2, com o número de transistores na ROM. Não esqueça de contar os transistores no decodificador.

Ex. 5.11 Converta o projeto da Figura 5.64 para uma memória de altura 4, usando dois multiplexadores de duas entradas.

Ex. 5.12 Discuta as vantagens e desvantagens de implementar uma memória ROM 1024x1024, como um retângulo bem mais alto do que largo, como um retângulo bem mais largo do que alto, e aproximadamente retangular. Considere todos os circuitos envolvidos, o decodificador de linha, a matriz de dados e o multiplexador de coluna.

Ex. 5.13 Escreva um programa que, dada uma função de V variáveis, produz os números das linhas da ROM nas quais devem ser ligados os transistores.

5.6 RAM – *Random Access Memory*

Memórias que permitem leitura e escrita são chamadas de *Random Access Memory*, ou RAM³. Estas memórias são usadas para a armazenagem de dados, e tipicamente são construídas com matrizes de armazenagem retangulares ou quadradas. O *random* é em contraposição à “memória de acesso sequencial”, que era a tecnologia disponível na época em que as RAMs foram desenvolvidas.

Memórias RAM são construídas com duas tecnologias distintas. *Memórias Dinâmicas* são construídas com grande capacidade de armazenamento – CIs de 4 a 16Gbits estão disponíveis em 2016 – e baixo custo. A outra tecnologia é a das *memórias estáticas*, construídas para tempo de acesso curto, em detrimento da capacidade. As próximas seções apresentam estas tecnologias.

5.6.1 DRAM – Memória Dinâmica

Memórias RAM dinâmicas (DRAM) empregam células com somente dois transistores, donde advém a grande densidade – muitas células por unidade de área – e a grande capacidade. O ‘dinamismo’ destas memórias é um efeito colateral da célula minúscula: um transistor com a geometria apropriada é usado como um capacitor, e a energia contida neste capacitor representa

³Não custa insistir: a tradução *correta* para o Português é “memória de escrita e leitura” e não o ridículo “memória de acesso randômico”.

o valor do bit armazenado na célula. Como o circuito de uma célula é pequeno, com diâmetro da ordem de 10^{-8}m , o isolamento elétrico dos componentes é frágil, a carga armazenada no capacitor escapa para a vizinhança, e o valor armazenado no capacitor se perde. Por isso, o conteúdo destas memórias deve ser ‘refrescado’ periodicamente. Durante o “ciclo de *refresh*”, o conteúdo das células é lido e então reescrito, garantindo-se assim que os dados armazenados sejam preservados. Tipicamente, um ciclo de *refresh* deve ocorrer a cada 50 ou 60ms.

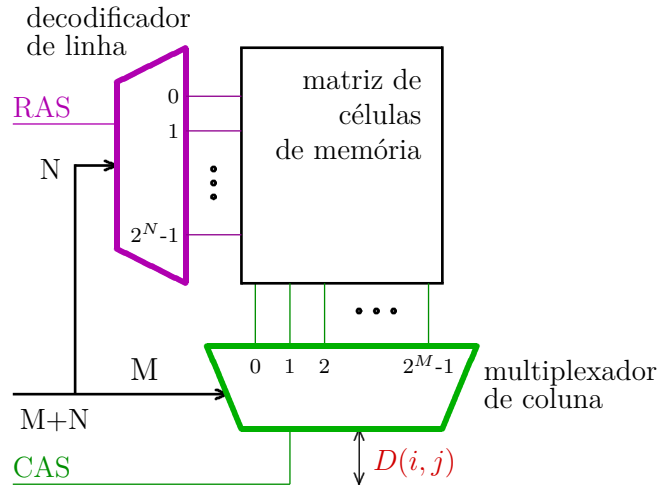


Figura 5.65: Organização de memória.

A Figura 5.65 mostra a organização de uma matriz de memória. Para uma matriz com 2^N linhas e 2^M colunas, são necessárias $N + M$ linhas de endereço para selecionar a célula de memória apontada pelo endereço indicado nos sinais N e M . Geralmente, os bits mais significativos do endereço selecionam uma das linhas da matriz.

O sinal RAS, ou *Row Address Strobe*, fica ativo para selecionar, ou ativar, uma das linhas da matriz. O sinal CAS, ou *Column Address Strobe*, fica ativo para selecionar uma das colunas.

Uma matriz de memória dinâmica é mostrada na Figura 5.66. Cada célula se liga à matriz por dois sinais, uma *linha de bit* (*bit line*), e uma *linha de palavra* (*word line*). Estes sinais selecionam uma célula e ao mesmo tempo permitem observar ou alterar seu conteúdo. Quando se usa o jargão de memória, as linhas da matriz são chamadas de *páginas*. Cuidado com o uso da palavra ‘linha’: esta palavra é usada (i) para os sinais de acesso às células de memória, e (ii) para a linha da matriz de dados, que é uma “página da DRAM”.

Aplicando-se os números da página i e da coluna j aos endereços, se obtém acesso ao conteúdo da célula da memória com o bit $D(i, j)$. A entrada do decodificador de linha, com o número da linha, contém um registrador que armazena todos os N bits com o número da linha, e este número é capturado quando o sinal RAS fica ativo. Uma vez que o endereço da página foi armazenado, o sinal CAS é usado para selecionar uma das M colunas da matriz. Um acesso à matriz de dados ocorre em três fases: (i) o número da página é armazenado, quando o sinal RAS fica ativo; (ii) uma coluna é selecionada quando o sinal CAS fica ativo; e (iii) o conteúdo da célula é então lido da saída do multiplexador de coluna. O *controlador de memória* é o circuito que ativa os sinais certos na hora e na sequência corretas. Parece complicado, e de fato é. Em breve retornaremos ao acesso à uma célula da matriz de dados.

Uma célula de memória dinâmica consiste de um *transistor de passagem* e de um capacitor, e uma versão simplificada é mostrada na Figura 5.67. O capacitor é o *gate* de um segundo

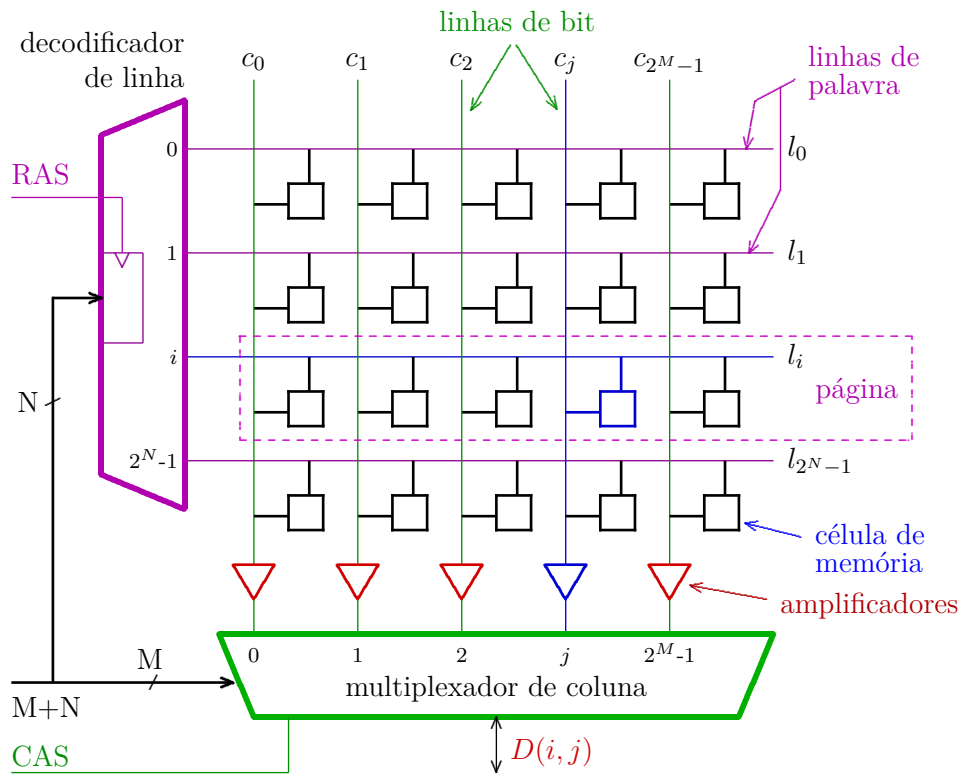


Figura 5.66: Memória DRAM com capacidade de $N \times M$ bits.

transistor, e a tensão entre as placas deste capacitor representa o bit armazenado. Quando uma página da matriz é selecionada, a linha de palavra é ativada, e o transistor de passagem conecta o terminal do capacitor à linha de bit – ($g = 1$) $\Rightarrow$ ($s = d$) – a variação de tensão na linha de bit é amostrada e amplificada, permitindo a recuperação do bit armazenado.

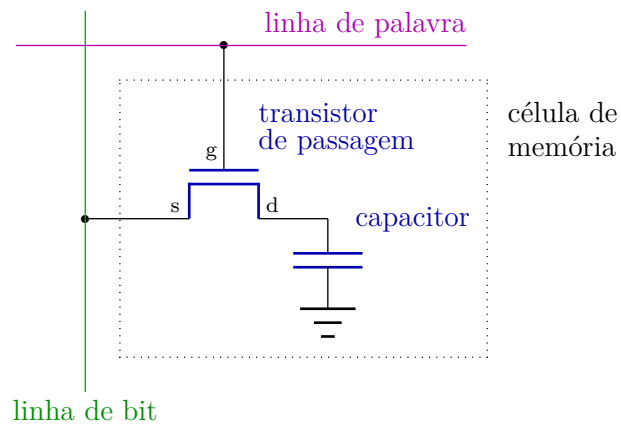


Figura 5.67: Célula de memória dinâmica.

Escrita

Numa escrita, a linha de bit é colocada no valor do bit por escrever, e então a linha de palavra é colocada em 1. O capacitor, se está carregado com uma tensão próxima de VCC, armazena o nível lógico 1, e quando está descarregado, com 0V entre as placas, armazena o nível 0. A Figura 5.68 mostra a sequência de eventos de uma escrita. O amplificador na linha de bit é usado nas leituras para restaurar o valor armazenado na célula.

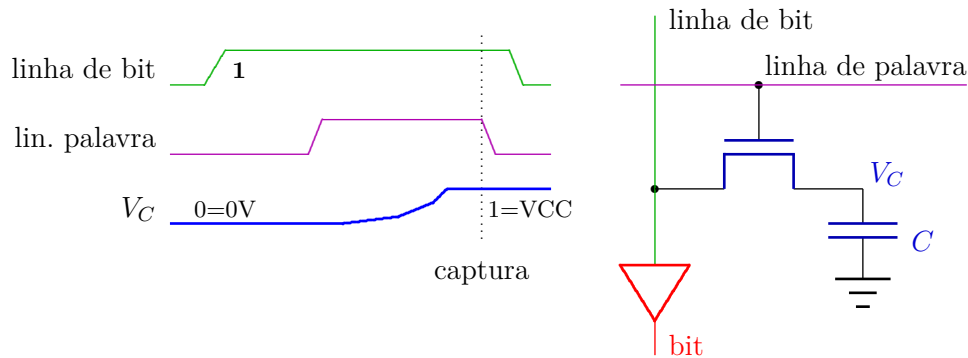


Figura 5.68: Escrita numa célula de DRAM.

Suponha que desejamos escrever 1 na célula de memória, e que o valor armazenado anteriormente era 0, e portanto o capacitor está descarregado. A linha de bit é colocada em 1, e quando a linha de palavra fica em 1, o capacitor, que estava com 0V de tensão, se carrega até VCC. Quando a linha de palavra é desativada, o novo valor é capturado porque a energia armazenada no capacitor não tem por onde escapar. Contudo, sendo frágil o isolamento, a carga escapa a energia se dissipa lentamente como calor.

Se o novo valor é igual ao anterior, a tensão armazenada sofre um reforço porque o sinal na linha de bit é mais forte do que o sinal no terminal do capacitor. Se o bit por armazenar é 0 e o valor anterior era 1, quando a linha de palavra é ativada, a tensão no capacitor cai para 0V, e assim permanece depois que a linha de palavra for desativada.

Leitura

Leituras são algo mais complexas do que escritas porque a carga armazenada nos capacitores é minúscula, da ordem de 10^6 elétrons. Na preparação para a leitura, a tensão na linha de bit é colocada exatamente em $VCC/2$. Quando a linha de palavra é ativada e o terminal do capacitor é ligado à linha de bit, a tensão entre as placas do capacitor gera uma pequena perturbação na tensão da linha de bit. Essa perturbação é mostrada com um traço mais espesso na linha de seleção de bit. A perturbação é amplificada e o valor original é recuperado da saída do amplificador mostrado na Figura 5.68, no sinal de nome *bit*.

As leituras são destrutivas. Para obter o valor armazenado, o capacitor é descarregado para $VCC/2$ se estava em 1, ou é carregado para $VCC/2$ se estava em 0. A descarga ocorre, e é mostrada, na fase de leitura. $VCC/2$ é um nível lógico indeterminado e pode contaminar o restante do circuito com valores indeterminados. Quando uma célula é lida, o valor recuperado deve ser gravado novamente, com seu nível lógico restaurado. A restauração do valor degradado é indicada como a *fase de restauração*.

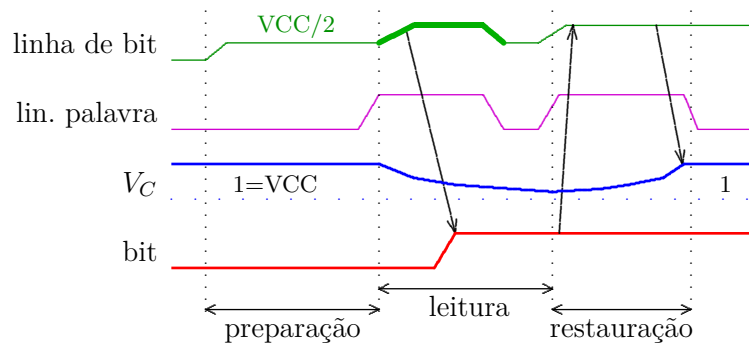


Figura 5.69: Preparação, leitura e restauração de uma célula de DRAM.

Refresh

O refrescamento da memória é efetuado com se fosse uma leitura. Veja, na Figura 5.66, que todas as células de uma linha da matriz podem ser refrescadas simultaneamente. Num ciclo de *refresh*, ao contrário de uma escrita ou leitura, todas as células de uma linha são lidas, e então gravadas novamente. O controlador de memória dinâmica deve varrer todas as linhas da DRAM, e refrescar cada linha dentro de um intervalo de 50 a 60ms.

5.6.2 SRAM – Memória Estática

Memórias estáticas empregam células maiores e mais complexas que aquelas de DRAM, sendo estas memórias menos densas e com menor capacidade. Contudo, o parâmetro de projeto mais importante para memórias estáticas é o tempo de acesso, que é um tanto menor do que nas DRAMs. Em geral, memórias estáticas são implementadas junto aos processadores, no mesmo circuito integrado, e portanto seu tempo de acesso tende a ser tão curto quanto o possível, para uma dada capacidade e tecnologia de circuitos.

Uma célula de memória estática é mostrada na Figura 5.70. Esta célula emprega 6 transistores ao invés dos dois de uma célula dinâmica. A ligação circular dos dois inversores viola a nossa definição de circuito combinacional, mas os apresentamos nesta seção para manter a continuidade do assunto.

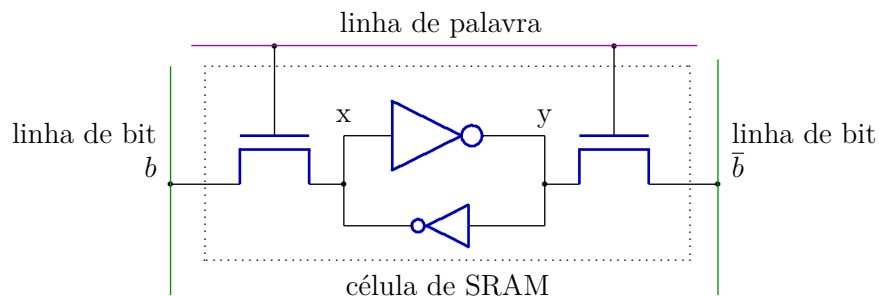


Figura 5.70: Célula de memória estática.

Os dois inversores formam um *básculo*, que é um circuito que se mantém num determinado estado se as condições externas não se alterarem e a alimentação do circuito for mantida. Quando a linha de palavra está em zero, se o sinal x está em 1, então o sinal y está em 0.

Se y está em 0, então x está em 1, e o básico mantém um valor estável. Este valor pode ser alterado se a linha de palavra estiver em 1, e os sinais complementares em b e $\bar{b}$ forem diferentes dos valores armazenados. Os inversores têm tamanhos distintos para que, quando o valor armazenado for trocado, o inversor mais fraco perca a disputa com o inversor mais forte.

Os inversores do básico são projetados para gerar sinais mais fracos do que os circuitos que geram os sinais b e $\bar{b}$. Numa escrita, as linhas de bit complementares devem ser colocadas nos níveis apropriados e então a linha de palavra é ativada. Se o conteúdo da célula se mantém, os valores em x e y não se alteram. Se o conteúdo da célula deve ser alterado, os valores em b e $\bar{b}$ se sobrepõem aos sinais em x e y , que acabam trocando de nível. Quando a linha de palavra é desativada, o novo valor é capturado pelo básico.

O diagrama de blocos da memória DRAM não mostra o circuito de escrita. A Figura 5.71 mostra um diagrama de blocos de uma SRAM com mais detalhe do que aquele da DRAM. Os circuitos de decodificação de linha e de multiplexação de coluna são similares aos da DRAM. O topo do diagrama mostra o *seletor de escrita*, que escolhe qual a coluna por atualizar. Se o sinal de escrita (wr) estiver ativo, então o valor de entrada $D(i, j)$ é aplicado ao circuito de escrita da coluna selecionada.

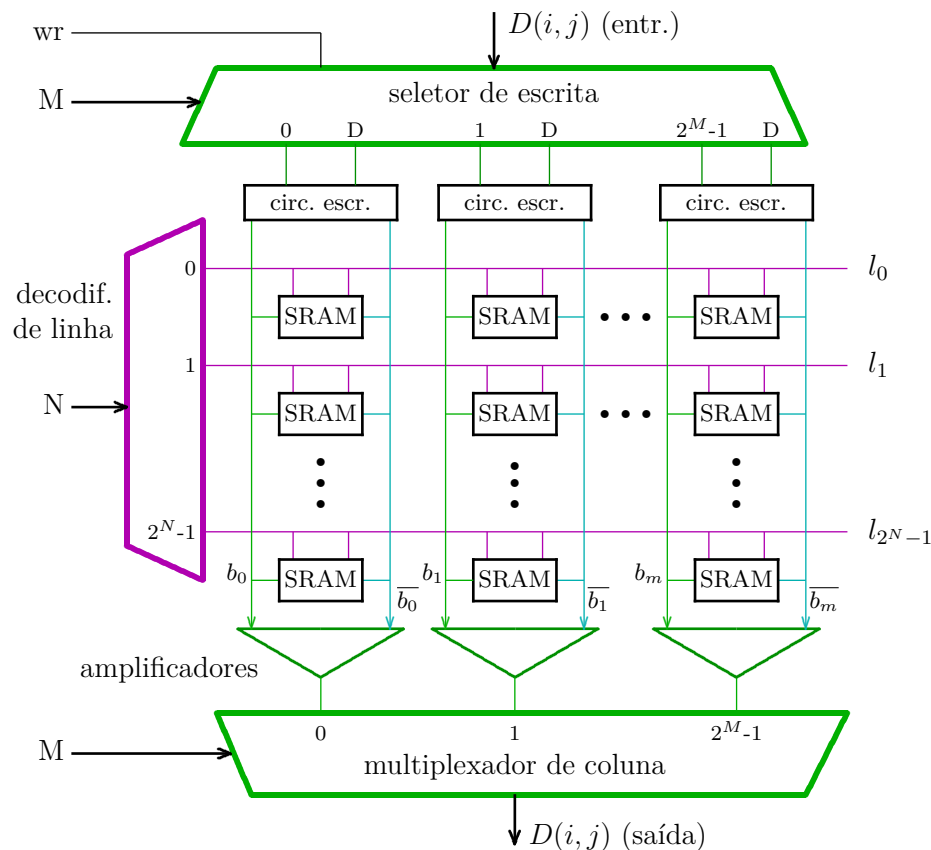


Figura 5.71: Memória SRAM com capacidade de $N \times M$ bits.

Na base do diagrama estão os *amplificadores diferenciais* que amplificam a diferença entre b e $\bar{b}$ para recuperar o valor armazenado na célula $D(i, j)$.

Numa leitura, a tensão nas linhas de bit deve ser colocada em $VCC/2$. Quando a linha de palavra é ativada, as saídas dos inversores provocam uma perturbação nas linhas de bit, e

o amplificador diferencial detecta a *diferença* entre as tensões em b e $\bar{b}$ e recupera o valor armazenado na célula.

Não é necessário refrescar as células após a leitura porque os búsculos mantêm o valor armazenado. Para acelerar o acesso ao conteúdo das células, o número da linha e o da coluna são fornecidos simultaneamente à matriz. Numa leitura, decorrido o tempo de acesso do decodificador de linha, mais o tempo para acessar o conteúdo da célula, este valor é apresentado na saída após o tempo de propagação do multiplexador de coluna.

O tempo de acesso à SRAM é menor do que o de uma DRAM porque a capacidade da memória estática é, em geral, uma fração daquela da DRAM, e portanto o tempo para decodificar a linha e multiplexar a coluna é mais curto porque estes circuitos são menores. Além disso, não há *refresh* nas SRAM, o que também reduz seu tempo médio de acesso.

5.7 Bits que não pertencem à $\mathbb{B}$

Neste capítulo introduzimos novos valores para os bits, valores estes que não pertencem à $\mathbb{B}$, e passamos a usar sinais ‘fortes’ e sinais ‘fracos’. Como é possível conciliar a nossa lógica quase-Booleana com estes novos valores, sem aumentar exageradamente a complexidade dos modelos que empregamos para nossos circuitos?

Uma solução, relativamente simples, é adotada na linguagem VHDL, quando se empregam sinais da chamada *lógica padrão*, de tipo `std\_logic`. Sinais deste tipo podem estar num de nove níveis: não-inicializado, desconhecido-forte, 0-forte, 1-forte, alta-impedância, desconhecido-fraco, 0-fraco, 1-fraco e não-importa.

Se um sinal fraco é ligado a um sinal forte, o nível do sinal forte predomina. Um curto circuito entre 1-forte e 0-forte resulta em desconhecido-forte. Um sinal desconhecido-forte contamina todos os circuitos aos quais esteja ligado.

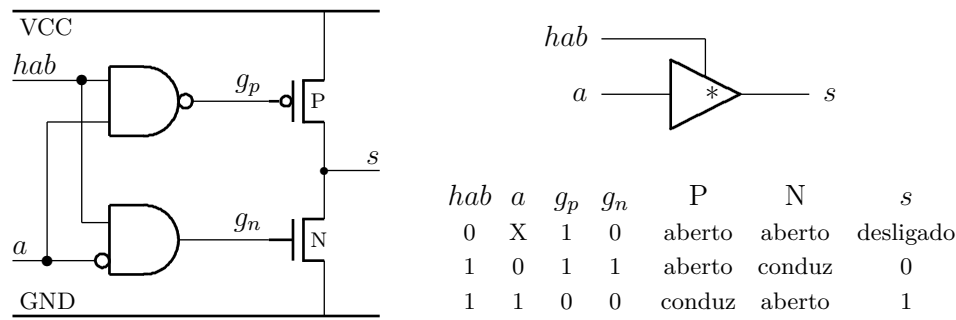
Para fins de simulação, sinais não-inicializados equivalem a desconhecido-forte porque a máquina de simulação é incapaz de adivinhar os valores iniciais desejados para os sinais.

Vejamos os circuitos que operam com um *terceiro estado* que não é 0 e nem é 1, e ainda *multiplexadores analógicos* construídos com dois transistores.

5.7.1 Terceiro Estado

É possível projetar circuitos digitais com um *terceiro estado*, além dos estados 0 e 1. No terceiro estado, a saída do circuito forma um caminho que oferece alta resistência à passagem de corrente, desligando-a dos circuitos aos quais está conectada. Circuitos que operam com os três estados são chamados de *three-state* ou *tri-state*.

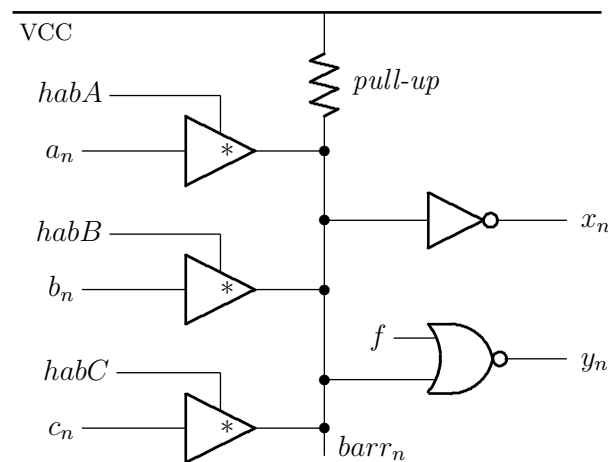
Estes circuitos possuem um sinal de controle que permite colocar a saída no terceiro estado com o efeito de desligá-la dos circuitos à que está conectada. A Figura 5.72 mostra o circuito interno de um *buffer three-state* – um *buffer* funciona como um amplificador que não altera o nível lógico do sinal, apenas torna-o “mais forte”. O símbolo do circuito é mostrado no lado direito da figura. O asterisco junto à saída é uma indicação visual de que esta saída pode operar no terceiro estado.

Figura 5.72: *Buffer three-state CMOS.*

Quando o sinal de habilitação está inativo ($hab = 0$) os níveis nos *gates* dos dois transistores ligados à saída fazem com que eles fiquem abertos, de forma a que não haja nenhum caminho de baixa resistência que ligue a saída à VCC ou à GND. Um sinal ligado a uma saída *three-state* é indeterminado, e se diz que está *flutuando* se não há um circuito que puxe o nível lógico neste sinal para 0 ou para 1.

Circuitos *three-state* são usadas para a ligação de várias saídas a um mesmo fio, formando um *barramento*, como mostra a Figura 5.73. Os circuitos que geram os sinais a_n , b_n e c_n têm suas saídas ligadas ao fio $barr_n$, e este conduz o sinal que é entrada para outros circuitos que produzem os sinais x_n e y_n .

O circuito de controle deve ser projetado para garantir que no máximo um dentre $habA$, $habB$, ou $habC$, esteja ativo a qualquer momento. Se mais de uma saída estiver ligada simultaneamente, o nível de sinal resultante em $barr_n$ pode assumir um nível lógico indeterminado, se um dos sinais estiver em 1 e o outro em 0. Note que este circuito implementa um multiplexador de três entradas.

Figura 5.73: *Ligação de saídas three-state.*

Quando todas as saídas estão no terceiro estado, o nível lógico do sinal é indeterminado porque nada se pode dizer sobre ele. Para garantir que este sinal fique em um nível lógico determinado, um resistor é ligado à VCC. Este resistor é chamado de *pull-up* porque sua função é puxar o nível lógico do sinal para 1 se todas as saídas estão no terceiro estado. Quando uma das saídas é ativada e está em 0, o transistor N desta saída puxa o nível lógico para 0, sobrepondo-se ao 1-fraco do *pull-up*. O *pull-up* é chamado de “circuito passivo” porque somente atua quando

nenhum dos circuitos ativos – portas lógicas ou *buffers* – está puxando o nível lógico do sinal para 0 ou para 1.

Em geral, circuitos *three-state* são usados para interligar circuitos integrados, e raramente são usados internamente aos CIs. A temporização destes circuitos é complexa porque deve-se garantir que somente uma fonte de sinal esteja ativa a cada instante. Multiplexadores são usados internamente aos CIs porque sua construção impede curto-circuitos entre as várias fontes de sinal.

5.7.2 Portas de Transmissão

O terceiro estado é empregado por causa das características dos transistores CMOS que permitem o desligamento do circuito de saída. Outro circuito útil, também dependente do comportamento elétrico dos transistores é a *porta de transmissão* (*transmission gate*). A Equação 5.3 indica que se o *gate* de um transistor CMOS está no nível lógico ativo, então seus terminais fonte e dreno estão conectados (chave fechada). Lembre que transistores do tipo P transmitem bem o nível lógico 1 e transmitem mal o nível 0. O mesmo vale para transistores do tipo N, com os níveis trocados – o nível 0 é transmitido bem, e o nível 1 fracamente.

Uma porta de transmissão pode ser implementada com dois transistores e sinais de controle adequados, como indica a Figura 5.74. Os dois transistores são necessários para garantir que sinais dos dois níveis são transmitidos entre os terminais “de dados”. Como os transistores são simétricos com relação aos níveis nos terminais fonte e dreno, este circuito permite a circulação de corrente da entrada para a saída, e da saída para a entrada, ao contrário das outras portas lógicas que estudamos. Por esta razão, estes circuitos são também chamados de *chaves analógicas*.

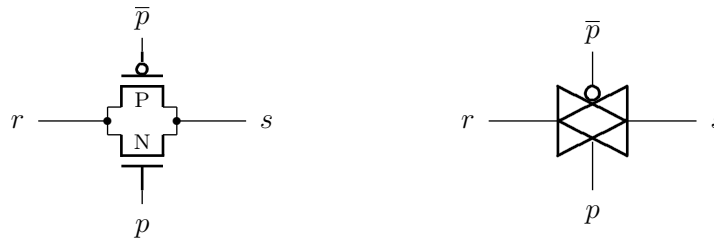


Figura 5.74: Porta de transmissão.

Quando os sinais de passagem estão ativos, $p = 1$ e $\bar{p} = 0$, os terminais r e s ficam interligados e no mesmo nível lógico. O símbolo desta porta parece com dois *buffers*, um em cada direção, justamente para indicar a capacidade de transmissão nos dois sentidos.

Portas de transmissão podem ser usadas para implementar multiplexadores e circuitos de deslocamento e rotação. A Figura 5.75 mostra um multiplexador de duas entradas para sinais com 3 bits de largura. Quando $selA = 1$, as portas de transmissão ligadas ao sinal $A = \langle a_0, a_1, a_2 \rangle$ transmitem, enquanto que as portas ligadas ao sinal $B = \langle b_0, b_1, b_2 \rangle$ são desligadas. Quando $selA = 0$, a situação se inverte e o sinal B é ligado a C .

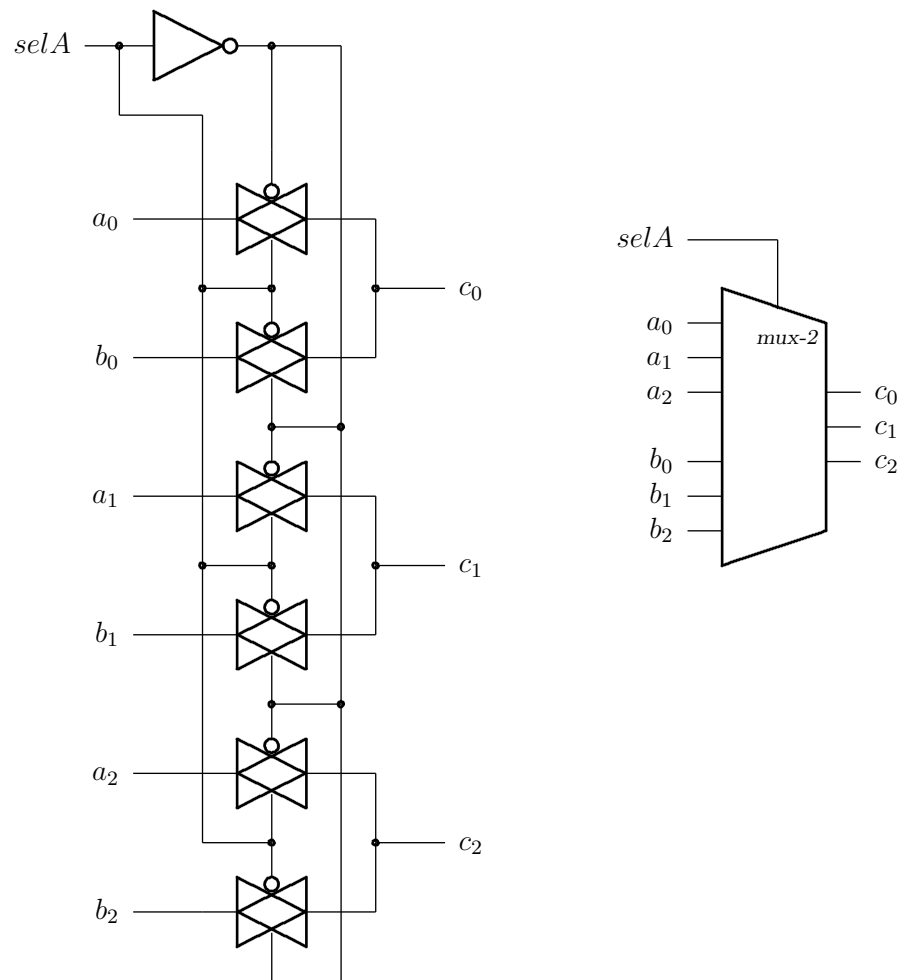


Figura 5.75: Multiplexador com três bits de largura.

Exercícios

Ex. 5.14 É possível implementar um *buffer three-state* com menos de 10 transistores? Não use portas de transmissão.

Ex. 5.15 Mostre como implementar a função lógica $\vee$ com *buffers three-state*.

Ex. 5.16 Usando a técnica da Figura 4.13, mostre como implementar um multiplexador de 8 entradas com portas de transmissão.

Ex. 5.17 Mostre como implementar um deslocador de um bit com portas de transmissão.

Índice Remissivo

Símbolos

T_A , 61
 T_D , 115
 T_I , 104, 111
 T_M , 112, 114
 T_P , 111
 $T_{C,x}$, 113–115
 $\%$, 18–20
 $\bigvee$, 45
 $\bigwedge$, 45
 $\equiv$, 36
 $\wedge$, 30, 36
 $\triangleleft \triangleright$, 36, 42, 63
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 40, 44
 $\neg$, 30, 36
 $\vee$, 30, 36
 $\oplus$, 36, 50, 62
 $\mapsto$, 41
 $\bar{a}$, *veja* $\neg$
 π , 25
 $\backslash$, 74
decod, 69
demux, 72
 e , 24
mux, 64
 $\mathbb{B}$, 29–33, 38, 41
 $\mathbb{Z}$, 41
 $\mathbb{N}$, 41, 42
num, 43, 52, 69, 74
 num^{-1} , 52
 $\mathbb{R}$, 41
 $\langle \rangle$, 29, 41, 42

A

abstração, 27
 bits como sinais, 27–33, 54
 tempo discretizado, 111, 113
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo,
 conversão de base, 18
 conversão de base de frações, 22
 amplificador, 120, 130
 diferencial, 133
 and, *veja* $\wedge$
and array, 122
 aproximação, 24
 árvore, 61, 113

assembly, *veja* ling. de montagem
 associatividade, 31
 atraso, *veja* tempo de contaminação, 54
 atribuição, 12

B

barramento, 135
 básculo, 132
 binário, 20
 bit, 20
 bits, 27–37, 62
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 variável, 30
buffer, 118
buffer three-state, 134
 buraco, 83
 byte, 11

C

cadeia,
 de portas, 61, 113
 caminho crítico, 112
 capacitor, 100, 102, 128, 129
capture FF, *veja flip flop*, destino
 CAS, 129
 célula de RAM, 78
 célula, 95
 chave,
 analógica, 136
 digital, 87
 normalmente aberta, 87
 normalmente fechada, 75, 87
 ciclo,
 combinacional, 54
 violação, 78
 circuito,
 combinacional, 54, 62
 dual, 94
 circuito aberto, 54
 clk, *veja clock*
 CMOS, 56, 62, 82–137
 buffer three-state, 134
 célula, 95
 inversor, 91
 nand, 94
 nor, 93
 porta de transmissão, 136

portas inversoras, 94
 sinal restaurado, 120
 código,
 Gray, 60
Column Address Strobe, veja CAS
 combinacional,
 ciclo, 54
 circuito, 54
 dispositivo, 54
 comparador,
 de igualdade, 59
Complementary Metal-Oxide Semiconductor, veja CMOS
 complemento, veja $\neg$
 complemento, propriedade, 31
 comportamento transitório, veja transitório
 comutatividade, 31
 condicional, veja $\triangleleft \triangleright$
 condutor, 82
 conjunção, veja $\wedge$
 conjunto mínimo de operadores, 62
 contra-positiva, 40
 controlador,
 de memória, 129
 conversão de base, 18
 corrente, 82, 100, 102, 107, 109
 de fuga, 110
 corrida, 118, 120
 curto-circuito, 54

D

datapath, veja circuito de dados
 decimal, 17
 decodificador, 69, 75, 79–81, 121
 de linha, 121, 129
 de prioridades, 71
delay, 54
 demultiplexador, 72, 115
design unit, veja VHDL, unidade de projeto
 detecção de bordas, 118
 disjunção, veja $\vee$
 dispositivo, 82
 combinacional, 54
 distributividade, 31, 34, 49
 divisão inteira, 43
 doador, 83
don't care, 67
 dopagem por difusão, 83
 dopante, 83
 DRAM, 128–132
 controlador, 129
 fase de restauração, 131
 linha,
 de palavra, 129
 linha de bit, 129
 linha de palavra, 130
 página, 129
 refresh, 129
 dual, 32, 90, 94

dualidade, 32

E

EEPROM, 127
 endereço, 74
 energia, 95, 100, 107–110
 enviesado, relógio, veja skew
 EPROM, 127
 equivalência, veja $\Leftrightarrow$
 erro,
 de representação, 23
 especificação, 42
 expressões, 36

F

fan-in, 104–107, 111
fan-out, 79, 81, 104–107, 111, 115
 fechamento, 31
 FET, 86
Field Effect Transistor, veja FET
Field Programmable Gate Array, veja FPGA
flip-flop,
 modelo VHDL, veja VHDL, *flip-flop*
 um por estado, veja um FF por estado
 forma canônica, 46
 frações, veja ponto fixo
 frequência máxima, veja relógio
 função, 30
 tipo, 29, 41
 função, aplicação bit a bit, 32
 função, tipo (op. infix), veja $\mapsto$

G

glitch, veja transitório
 GND, 88
 gramática, 17

H

hexadecimal, 19

I

idempotência, 31
 identidade, 31
 igualdade, 30
 implementação, 42
 implicação, veja $\Rightarrow$
 informação, 16
 Instrução,
 busca, veja busca
 instrução, 12
 busca, veja busca
 decodificação, veja decodificação
 execução, veja execução
 resultado, veja resultado
 interface,
 de rede, 13
 de vídeo, 12
 inversor, 91
 tempo de propagação, 104
 involução, 31, 58

isolante, 82

J

Joule, 107

L

latch, veja *básculo*

latch FF, veja *flip flop*, destino

launch FF, veja *flip flop*, fonte

Lei de Joule, 108

Lei de Kirchoff, 101

Lei de Ohm, 100, 101

ligação,

 barramento, 135

 em paralelo, 88, 94

 em série, 88, 94

linguagem,

assembly, veja *ling. de montagem*

 Z, 27

linha de endereçamento, 75

literal, 38

logaritmo, 43

lógica restauradora, 120

M

Mapa de Karnaugh, 47, 119

Máquina de Mealy, veja *máq. de estados*

Máquina de Moore, veja *máq. de estados*

máscara, 32

máximo e mínimo, 31

maxtermo, 45

Mealy, veja *máq. de estados*

memória,

 atualização, 74

 de vídeo, 13

 decodificador de linha, 77

 endereço, 74

 FLASH, 128

 matriz, 124, 129

 multiplexador de coluna, 77

 primária, 13

 RAM, 78, 128

 ROM, 75, 121

 secundária, 13

memória dinâmica, veja *DRAM*

memória estática, veja *SRAM*

mintermo, 45, 119, 121

modelo,

 funcional, 43

 porta lógica, 91

 temporização, 110

módulo, veja *%*, *mod*

Moore, veja *máq. de estados*

MOSFET, 86

multiplexador, 58, 63–67, 77, 96, 112–114, 118–

 119, 121, 129, 135, 136

multiply-add, veja *MADD*

N

número,

 de Euler, 24

 negação, veja $\neg$

 nível lógico,

 0 e 1, 28

 indeterminado, 28, 104, 111, 135

 terceiro estado, 134

 nó, 91

 not, veja $\neg$

 número primo, 51

O

octal, 18

operação,

 binária, 29

 bit a bit, 32

 infixada, 42

 prefixada, 45

 unária, 29

operações sobre bits, 29–33

operador,

 binário, 29

 lógico, 36

 unário, 29

operation code, veja *opcode*

or, veja $\vee$

or array, 124

ou exclusivo, veja $\oplus$

ou inclusivo, veja $\vee$

P

paridade,

 ímpar, 47

 par, 47

período mínimo, veja *relógio*

pipelining, veja *segmentação*

piso, veja $\lfloor v \rfloor$

ponto flutuante, 67

porta lógica, 62

 and, 56

 carga, veja *fan-out*

de transmissão, 136

nand, 57, 94

nor, 57, 93

not, 56, 91

or, 56

xor, 57, 62

portas complexas, 95

potência, 107–110

dinâmica, 109

estática, 110

potenciação, 42

precedência, 30

precisão,

representação, 23

prioridade,

decodificador, 71

processador, 12

produtório, 33

programa de testes, veja *VHDL*, *testbench*

PROM, 127
 propriedades, operações em $\mathbb{B}$, 31
 prova de equivalência, 40–41
 pull-down, 91
 pull-up, 91, 121, 135
 pulso, 117, 118
 espúrio, veja *transitório*

R

RAM, 12, 74, 78, 128–134
 célula, 78
 dinâmica, 128
 Random Access Memory, veja *RAM*
 RAS, 129
 Read Only Memory, veja *ROM*
 realimentação, 78
 receptor, 83
 rede, 91
 redução, 33
 refresh, 132, 134
 Register Transfer Language, veja *RTL*
 registrador de deslocamento,
 modelo VHDL, veja *VHDL*, *registrador*
 relógio,
 enviesado, veja *skew*
 representação,
 abstrata, 28
 binária, 20
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 posicional, 17
 precisão, 23
 resistência, 84, 95, 100
 ROM, 12, 74–77, 121–128
 Row Address Strobe, veja *RAS*

S

seletor, 69
 semântica, 17
 semicondutor, 82
 tipo N, 83
 tipo P, 84
 silogismo, 37
 simplificação de expressões, 38–40
 sinal, 27, 41
 analógico, 27
 digital, 27, 28
 fraco, 120, 121, 133, 136
 intensidade, 86, 134
 restaurado, 120, 134
 síntese, veja *VHDL*, *síntese*
 Solid State Disk, veja *SSD*
 soma, veja *somador*
 soma de produtos, 45, 49, 121
 somador,
 completo, 99
 parcial, 98

somatório, 33
 spice, 28
 SRAM, 132–134
 SSD, 14
 superfície equipotencial, 91, 105

T

tabela verdade, 33–35, 44
 tamanho, veja $|N|$
 tempo,
 de contaminação, 110, 113–116
 de propagação, 54–55, 58, 61–62, 70, 74, 80, 97,
 99, 104, 110–112
 temporização, 99–121
 tensão, 100
 Teorema,
 DeMorgan, 32, 41, 46, 52, 57, 58, 89, 93
 Dualidade, 94
 Simplificação, 47
 terceiro estado, 134–136
 testbench, veja *VHDL*, *testbench*
 teto, veja $\lceil r \rceil$
 three-state, veja *terceiro estado*
 tipo,
 função, 29
 tipo de sinal, 41
 Tipo I, veja *formato*
 Tipo J, veja *formato*
 Tipo R, veja *formato*
 transferência entre registradores, veja *RTL*
 transistor, 84–86, 90–91
 CMOS, 90
 corte, 109
 gate, 84
 saturação, 109
 sinal fraco, 86
 tipo N, 85
 tipo P, 86
 Transistor-Transistor Logic, veja *TTL*
 transitório, 116–118
 transmission gate, veja *porta de transmissão*
 TTL,
 74148, 71
 tupla, veja $\langle \rangle$
 elemento, 32
 largura, 43

U

Unidade de Lógica e Aritmética, veja *ULA*

V

valor da função, 30
 VCC, 88
 vetor de bits, veja $\langle \rangle$, 32
 largura, 43
 VHDL,
 design unit, veja *VHDL*, *unidade de projeto*
 std_logic, 134
 tipos, 41

W*Watt*, 107*write back*, veja *resultado***X***xor*, veja $\oplus$ **Z***Z*, *linguagem*, 27

Capítulo 6

Aritmética

You cannot ask us to take sides against arithmetic.

Winston Churchill

Computadores foram inventados, e são usados, para efetuar operações aritméticas sobre números em aplicações tão diversas como o processamento da folha de pagamentos de uma empresa, na simulação de trajetórias de planetas e de partículas subatômicas, na simulação de paisagens em jogos, no cálculo do preço a pagar pelo etanol num posto de combustíveis.

Neste capítulo¹ estendemos nossa abstração para bits, permitindo agora que tuplas de bits representem números naturais e números inteiros. Circuitos de aritmética normalmente empregam a representação para inteiros chamada de *complemento de dois* porque esta é simples e eficaz, e resulta em circuitos também simples, portanto eficazes. Esta representação é definida na Seção 6.1.

Neste capítulo estudamos os circuitos usados para computar a soma, a diferença e o produto de dois números representados por sequências de bits. Os circuitos que efetuam estas operações são descritos nas Seções 6.2 e 6.3.

Deslocamentos são operações interessantes: um deslocamento para a esquerda equivale a uma multiplicação por uma potência de dois, enquanto um deslocamento para a direita, tomados os cuidados necessários, equivale a uma divisão por uma potência de dois. Estes circuitos são apresentados na Seção 6.4.

O componente de um processador em que são efetuadas somas e subtrações, mais as operações lógicas sobre vetores de bits, é chamado de Unidade de Lógica e Aritmética, e sua implementação é descrita na Seção 6.5.

O algoritmo que usamos para somar dois números implica na propagação do vai-um, potencialmente através de todos os dígitos dos operandos. Num somador com operandos de 32 ou 64 bits, o tempo necessário para computar o resultado pode ser excessivo. Nas Seções 6.6 e 6.7 veremos dois somadores que minimizam o tempo de propagação do vai-um.

Na Seção 6.8 são apresentados três circuitos combinacionais que efetuam a multiplicação. Na multiplicação, como na soma, a cadeia de propagação do vai-um é problemática e seus efeitos no tempo de propagação devem ser minimizados.

¹© Roberto André Hexsel, 2012-2021. Versão de 25 de fevereiro de 2021.

6.1 Sequências de Bits Que Representam Números

Números devem ser representados internamente ao computador por sequências de bits, uma vez que os bits são as menores unidades de informação que podem ser usadas num computador digital. Vejamos então como números naturais e números inteiros podem ser representados como sequências de bits. Iniciamos a exploração pela adição, que é operação aritmética mais simples, e a mais popular nos programas que escrevemos.

6.1.1 Adição na base 2

A soma s de dois números a e b , representados em um único dígito binário é:

a	$+$	b	$=$	s	$\mathbb{N}$
0	+	0	=	0	0
0	+	1	=	1	1
1	+	0	=	1	1
1	+	1	=	?	2

A coluna da direita mostra o número natural que representa o resultado. Na quarta linha, a soma $1 + 1 = 2$ não é representável por um único dígito, e portanto a tabela verdade da soma deve ser aumentada com uma coluna para o vai-um.

Se incluímos o vai-um no resultado, que incluamos também o vem-um nas parcelas. O resultado da soma é representado pelo número de dois bits $\langle vai, s \rangle$:

vem	$+$	a	$+$	b	$=$	vai	s	$\mathbb{N}$
0	+	0	+	0	=	0	0	0
0	+	0	+	1	=	0	1	1
0	+	1	+	0	=	0	1	1
0	+	1	+	1	=	1	0	2
1	+	0	+	0	=	0	1	1
1	+	0	+	1	=	1	0	2
1	+	1	+	0	=	1	0	2
1	+	1	+	1	=	1	1	3

(6.1)

Como o resultado da adição de três bits ($1+1+1 = 3$) é representável em dois dígitos ($3 = 11_2$), todas as oito combinações das entradas produzem os resultados esperados.

6.1.2 Sequências de Bits Que Representam Inteiros

Qual seria então uma representação para números, usando bits? Uma primeira restrição, que deriva de custo e da tecnologia disponível, é que os números sejam representados por sequências de bits com tamanho fixo, cujas larguras típicas são de 8, 16, 32 ou 64 bits. Isso é necessário porque os circuitos que efetuam operações aritméticas devem ter tamanho finito, assim como deve ser finita a memória que armazena os números.

Felizmente, a forma mais óbvia é também a mais simples e eficaz: emprega-se a mesma notação posicional que usamos com números na base 10, na qual cada dígito, ou posição, é multiplicada por uma potência de 10. Como a representação é com bits, a base da representação é 2.

Considere uma representação para números naturais ($\mathbb{N}$) com um campo fixo com 8 bits de largura, como a mostrada abaixo. A posição de cada dígito binário corresponde a uma potência de dois; o dígito à direita tem peso $2^0 = 1$ enquanto o dígito mais significativo tem peso $2^7 = 128$. A faixa de valores representáveis é de 0 a 255. O número 99 é representado em binário, num campo de 8 bits, por $0110.0011_2 = 2^6 + 2^5 + 2^1 + 2^0 = 64 + 32 + 2 + 1$.

$$\begin{array}{rcccccccc} \textit{peso} \rightarrow & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 99 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Para uma representação com k bits de largura, o valor do número natural N representado por uma sequência de k bits é obtido com a Equação 6.2, e é a soma ponderada dos bits b_i , $i \in [0, k)$.

$$N = \sum_{i=0}^{k-1} 2^i \cdot b_i \quad (6.2)$$

Qual seria uma representação igualmente eficiente para números inteiros ($\mathbb{Z}$)? Dada a conveniência da representação para os naturais, expressa na Equação 6.2, a representação para zero, em 8 bits, deve ser

$$0000.0000.$$

Assim como para zero, a representação óbvia para o número $+1$ é

$$0000.0001.$$

Qual seria então a representação para -1 ? Esta representação deve ser tal que $-1 + 1 = 0$. Se somarmos $+1$ ao número M que representa -1 , obtemos:

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ + & m_7 & m_6 & m_5 & m_4 & m_3 & m_2 & m_1 & m_0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Da Equação 6.1 temos que $1 + 1 = 0$ e vai um para a próxima posição. A soma $1 + m_0 = 0$ somente se $m_0 = 1$. A soma de $m_0 = 1$ com 1 é zero e *vai-um*. Para o segundo dígito, a soma *vem-um* + $0 + m_1 = 0$ somente se $m_1 = 1$. Da mesma forma para o terceiro dígito, com $m_1 = 1$, obtemos $m_2 = 1$. Continuando até m_7 , descobrimos que a representação em 8 bits para -1 é 1111.1111 .

Qual seria a representação para os demais números negativos, de tal forma que $A + (-A) = 0$?

$$\begin{aligned} (+A) + (-A) &= 0 \\ -A &= 0 - A \\ &= (1 - 1) - A \\ &= 1 + (-1 - A) \end{aligned}$$

Considerando somente o termo $(-1 - A)$, qual o valor de $R = (-1 - A)$?

$$\begin{array}{cccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & -1 \\ - & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & -A \\ \hline & r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0 & R \end{array}$$

Computando a subtração bit a bit, iniciando por a_0 , temos dois casos:

$$(i) \quad \text{se } a_0 = 0, \text{ então } r_0 = 1 - 0 = 1 = \overline{a_0};$$

e

$$(ii) \quad \text{se } a_0 = 1, \text{ então } r_0 = 1 - 1 = 0 = \overline{a_0}.$$

Se aplicarmos o mesmo raciocínio aos demais bits, descobre-se que $R = (-1 - A)$ é o complemento bit a bit de A , e é representado por $R = \overline{A}$. Assim,

$$-A = 1 + \overline{A} \quad (6.3)$$

que é a maneira simples, embora tediosa, de obter a representação de números negativos: basta complementar bit a bit o número positivo e somar 1 ao valor complementado.

Esta representação é chamada de *complemento de dois*, e nela o dígito mais significativo é sempre multiplicado por -2^m para um m apropriado. Em números representados em k bits, o bit b_{k-1} é multiplicado por -2^{k-1} . Se $k = 4$, então o número $+3$ é representado por

$$+3 = 0011_2 = (0 \cdot -2^3) + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0,$$

sendo -3 representado por

$$-3 = 1101_2 = (1 \cdot -2^3) + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

O conjunto de números inteiros representados em k bits é $[-2^{k-1}, 2^{k-1} - 1]$. Por causa da representação para zero, as 2^k representações distintas são divididos em 2^{k-1} números negativos e $2^{k-1} - 1$ números positivos.

A Equação 6.4 mostra os pesos dos dígitos de um número representado em 8 bits. O dígito mais significativo é multiplicado por -2^7 , e os demais dígitos são multiplicados por potências decrescentes positivas de 2.

$$X = (x_7 \cdot -2^7) + x_6 \cdot 2^6 + x_5 \cdot 2^5 + \cdots + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0 \quad (6.4)$$

Se o dígito mais significativo é zero, então o número representado é positivo, do contrário é negativo. Logo, esse dígito é chamado de *bit de sinal*.

Exemplo 6.1 Alguns exemplos de números representados em complemento de dois, em 8 bits são mostrados na Tabela 6.1. A posição mais significativa é multiplicada por 1 nos números negativos e portanto esta parcela vale -128 do valor representado. Para representar um número maior do que -128 , outras parcelas positivas devem ser adicionadas para descontar a diferença do valor desejado para -128 . Como mostra a segunda linha da tabela, $-1 = -128 + 127$. $\triangleleft$

Tabela 6.1: Exemplos de representação em complemento de dois.

		<i>posição</i>	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
<i>valor</i>		<i>peso</i>	-128	64	32	16	8	4	2	1
$+1$	$=$	$+0 + 1$	0	0	0	0	0	0	0	1
-1	$=$	$-128 + 127$	1	1	1	1	1	1	1	1
-125	$=$	$-128 + 3$	1	0	0	0	0	0	1	1
-4	$=$	$-128 + 124$	1	1	1	1	1	1	0	0

Considerando que num circuito digital as operações são efetuadas em circuitos com largura fixa e finita, o que ocorre quando somamos dois números grandes? Se somarmos o maior número representável a si próprio, ele dobra de tamanho:

$$N + N = 2N, \quad \log_2(N) = k, \quad \log_2(2N) = k + 1.$$

A soma de dois números de k bits tem $k + 1$ bits por causa da possibilidade de ocorrer vai-um no dígito mais significativo. Isso significa que o resultado de uma soma pode ser maior do que o maior número que pode ser representado em k bits. Esta situação indica a ocorrência de *overflow* porque um bit do resultado ‘transborda’ para além do limite máximo da representação.

Felizmente, o algoritmo para a detecção de *overflow* é simples. Para números representados em k bits e operação $A + B = R$:

1. replique o dígito mais significativo das duas parcelas, representando os operandos em $k + 1$ bits ($a_k = a_{k-1}$ e $b_k = b_{k-1}$);
2. adicione os dois operandos de $k + 1$ bits e ignore o vai-um para a posição r_{k+1} ;
3. ocorre *overflow* se, e somente se, os dois bits mais significativos do resultado diferem ($r_{k-1} \neq r_k$); do contrário, a resposta é o número representado corretamente em k bits.

Exemplo 6.2 Pode o resultado de $7 + 6$ ser representado corretamente em 4 bits?

Na operação abaixo, o dígito replicado está sublinhado, e o vai-um é mostrado à esquerda do dígito que o recebe.

$$\begin{array}{rcccccc}
 & \underline{0} & {}^{1+0} & {}^{1+1} & 1 & 1 & 7 \\
 + & \underline{0} & 0 & 1 & 1 & 0 & +6 \\
 \hline
 & \underline{0} & 1 & 1 & 0 & 1 & \\
 & \underline{r_4} & r_3 & r_2 & r_1 & r_0 & +13
 \end{array}$$

Os bits r_3 e r_4 diferem e portanto $7 + 6$ não pode ser representado em 4 bits porque o maior número representado corretamente com $k = 4$ é $2^{4-1} - 1 = 7$. $\triangleleft$

A Figura 6.1 mostra o círculo da representação em complemento de dois para campos de três bits. Os números binários 000_2 , 001_2 , 010_2 , e 011_2 representam os inteiros 0, 1, 2 e 3, respectivamente, enquanto os binários 111_2 , 110_2 , 101_2 , e 100_2 representam os inteiros -1, -2, -3, e -4, respectivamente.

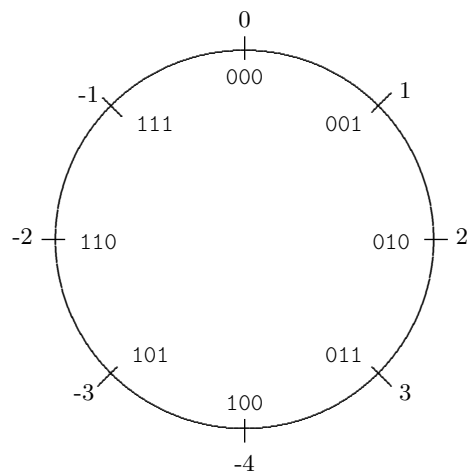


Figura 6.1: Círculo da representação em complemento de dois.

Duas somas são mostradas na Figura 6.2. A primeira, no arco externo, é a soma de +2 com +3, que resulta em +5 e portanto não é representável em complemento de dois com três bits, porque o binário 101_2 representa o inteiro -3. Na segunda soma, no arco interno, +2 + -3 produz o resultado correto que é -1. Daqui se extrai uma regra: *a soma de dois inteiros positivos pode produzir resultado incorreto, enquanto a soma de dois inteiros de sinais trocados produz resultados corretos*. A regra equivalente para a subtração é: *a diferença de números com sinais diferentes pode produzir resultados incorretos, enquanto a diferença de números com sinais iguais produz resultados corretos*.

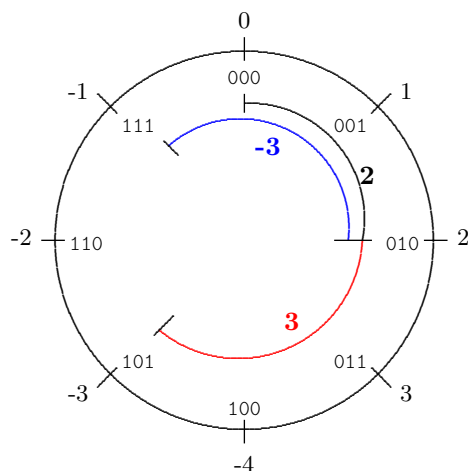


Figura 6.2: Representação em complemento de dois e *overflow*.

Exercícios

Ex. 6.1 Efetue as operações abaixo. Os números estão representados em complemento de dois, em 4 bits. Os resultados são representados corretamente?

- (a) 0100+0011 (b) 0110+1010 (c) 1001+0110
(d) 1010+1110 (e) 1101+0111 (f) 1111+0111

Ex. 6.2 Efetue as operações abaixo. Os números estão representados em complemento de dois, em 5 bits. Os resultados são representados corretamente?

- (a) 11011–01101 (b) 11010–01110 (c) 10110+01010 (d) 10100–10110

6.1.3 Sequências de Bits Que Representam Frações

As cinco primeiras potências negativas de 2 são

$$\begin{aligned} 2^{-1} &= 0,5 &= 1/2 \\ 2^{-2} &= 0,25 &= 1/4 \\ 2^{-3} &= 0,125 &= 1/8 \\ 2^{-4} &= 0,0625 &= 1/16 \\ 2^{-5} &= 0,03125 &= 1/32. \end{aligned}$$

Para representar quantidades que não são inteiras, empregamos uma *representação posicional* que estende aquela usada para inteiros, e é exemplificada abaixo. Essa representação é chamada de *ponto fixo*, em contraste com a representação em *ponto flutuante*, que não é discutida neste texto. As potências que multiplicam os dígitos da parte fracionária são negativas. Na base 10 temos

$$34,567 = 3 \cdot 10^1 + 4 \cdot 10^0 + 5 \cdot 10^{-1} + 6 \cdot 10^{-2} + 7 \cdot 10^{-3}$$

enquanto o número 3,3125 é representado na base dois por 11,01010₂

$$3,3125_{10} = 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-2} + 1 \times 2^{-4} = 2 + 1 + 0,25 + 0,0625.$$

Exemplo 6.5 Considerando uma representação em oito bits, com três bits para a parte inteira, e cinco bits para a fração, vejamos como representar números em *ponto fixo*. A Tabela 6.2 mostra a representação de cinco números fracionários, em complemento de dois.

Tabela 6.2: Exemplos de representação de frações em 3+5 bits.

valor	posição peso	b_7 –4	b_6 2	b_5 1	.	b_4 2^{-1}	b_3 2^{-2}	b_2 2^{-3}	b_1 2^{-4}	b_0 2^{-5}
+0,03125 = +0 + 0,03125		0	0	0	.	0	0	0	0	1
+0,0625 = +0 + 0,0625		0	0	0	.	0	0	0	1	0
–3 = –4 + 1		1	0	1	.	0	0	0	0	0
–1,78125 = –4 + 2 + 2^{-3} + 2^{-4} + 2^{-5}		1	1	0	.	0	0	1	1	1
–0,03125 = –4 + 2 + 1 + 2^{-1} + ... + 2^{-5}		1	1	1	.	1	1	1	1	1

A posição mais significativa é multiplicada por 1 nos números negativos e portanto esta parcela contribui com –4 ao valor representado. Para representar um número maior do que –4, parcelas positivas devem ser adicionadas para ‘descontar’ a diferença do valor desejado para –4. <

A precisão da representação depende do número de bits à direita do ponto. Para f bits de fração, a separação entre dois valores contíguos na representação é de 2^{-f} . Quanto maior o número de bits na fração, melhor a precisão dos valores representados.

6.2 Soma

O circuito que efetua a soma de dois bits mais o vem-um é chamado de *somador completo de um bit*, e tem entradas a , b e vem-um (vem), e saídas s e vai-um (vai). O relacionamento entre estes sinais é definido na Tabela 6.3. A coluna intitulada N mostra o valor da soma na base 10. As equações para computar s e vai são facilmente derivadas da Tabela 6.3, e são mostradas na Equação 6.5. Estas equações definem o circuito do bloco *soma* na Figura 6.13.

$$\begin{aligned} s &= a \oplus b \oplus vem \\ vai &= (a \wedge b) \vee (a \wedge vem) \vee (b \wedge vem) \end{aligned} \quad (6.5)$$

Tabela 6.3: Tabela verdade do somador completo de um bit.

a	b	vem	vai	s	N
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	0	1	0	2
0	0	1	0	1	1
0	1	1	1	0	2
1	0	1	1	0	2
1	1	1	1	1	3

O circuito que efetua a soma de números representados em mais de um bit replica o algoritmo usado nas operações que efetuamos manualmente, que é indicado na Figura 6.3. Iniciando da direita, na posição menos significativa, um dígito de cada parcela é adicionado, e o vai-um desta adição é incluído na soma do próximo par de dígitos. Isso se repete até o dígito mais significativo dos operandos.

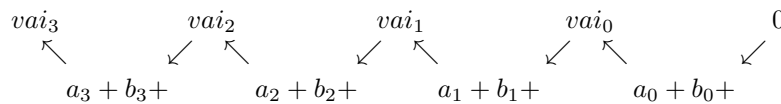


Figura 6.3: Adição de dois números representados em 4 bits.

Para somar dois números de quatro bits basta encadear quatro somadores como aqueles definidos na Equação 6.5. O circuito do somador de quatro bits de largura é mostrado na Figura 6.4. O sinal vai_i de um somador é ligado à entrada vem_{i+1} do somador do próximo bit mais significativo. O sinal vem_0 é ligado a 0 nas somas.

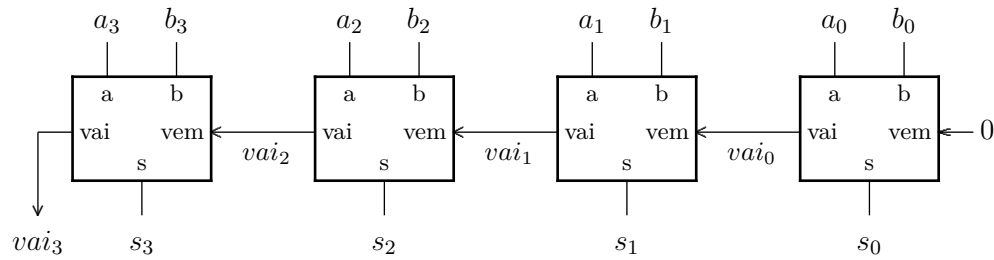
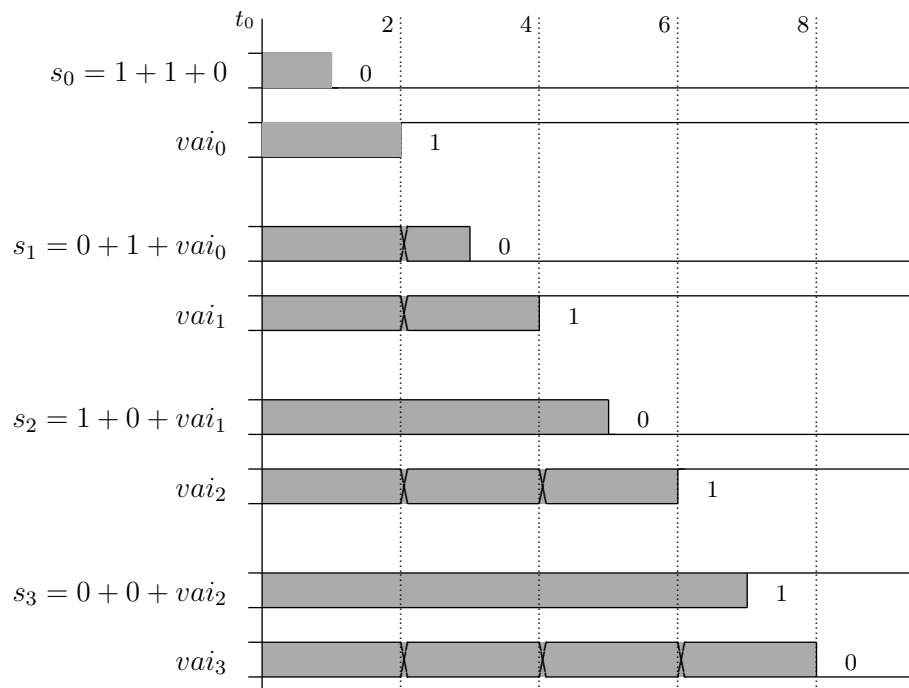


Figura 6.4: Circuito somador de quatro bits.

Exemplo 6.6 Vejamos como é o comportamento temporal do somador de 4 bits. A Figura 6.5 mostra o diagrama de tempos da adição $0101 + 0011$, com $vem_0 = 0$. Para simplificar o diagrama, a soma estabiliza após uma unidade de tempo, e o vai-um após duas unidades.

Os sinais de entrada (A e B) estabilizam no instante t_0 . Depois de uma unidade de tempo o valor em s_0 estabiliza e vai_0 estabiliza uma unidade mais tarde. Em cada um dos somadores, a saída s_i estabiliza depois que o respectivo vai_{i-1} também estabilize. As partes hachuradas indicam os intervalos em que os sinais estão indeterminados, e portanto inválidos. Para **esta combinação de entradas**, o resultado pode ser usado após $t_0 + 8$. O tempo de propagação pode ser diferente para outras entradas. $\triangleleft$

Figura 6.5: Temporização do somador de quatro bits para $0101+0011+0=0.1000$.

6.3 Subtração

O complemento de dois de um número, e portanto o negativo daquele número, é obtido complementando-se todos os bits do número e somando-se 1 ao complemento: $-B = \overline{B} + 1$. O circuito para efetuar a subtração é o mesmo circuito usado para efetuar a soma, como o da Figura 6.4, desde que a entrada B seja complementada, e seja adicionado 1 à $\overline{B}$:

$$A - B = A + (\overline{B} + 1).$$

O circuito para complementar B é controlado pelo sinal inv : $\overline{B} \triangleleft inv \triangleright B$. A adição do 1 é obtida ligando-se 1 à entrada de vem-um do bit menos significativo, vem_0 .

O circuito que efetua somas e subtrações é mostrado na Figura 6.6. Repare que o sinal inv é também ligado à entrada vem_0 : se a operação é uma soma, então $inv = vem_0 = 0$ e o circuito computa $A + B + 0$; se é uma subtração, então $inv = vem_0 = 1$ e o circuito computa $A + \overline{B} + 1$. No circuito da ULA, na Figura 6.13, o sinal inv deve ser ativado quando $F = 1$.

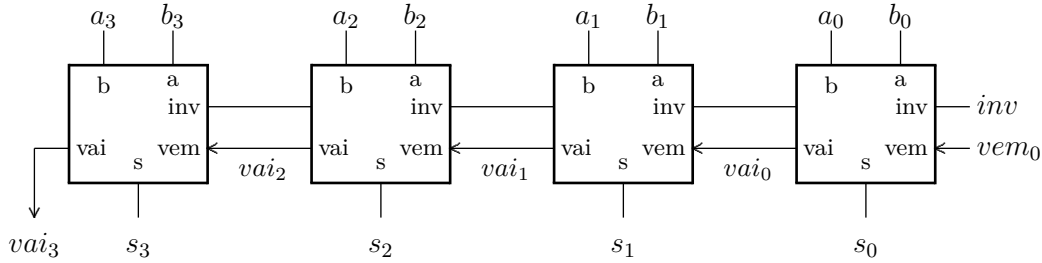


Figura 6.6: Circuito que efetua somas e subtrações.

O circuito que subtrai tem uma ligeira idiossincrasia: se efetuarmos $A - A$, o resultado é 0 com $vai_3 = 1$. Isso decorre da implementação do circuito da soma, conforme a Equação 6.5. Ao efetuar $A - A$ obtemos

$$\begin{aligned}
 s_i &= (a_i \oplus \overline{a_i}) \oplus vem_i && \text{definição } \oplus \\
 &= 1 \oplus vem_i && \text{definição } \oplus \\
 &= \overline{vem_i} \\
 vai_i &= (a_i \wedge \overline{a_i}) \vee (a_i \wedge vem_i) \vee (\overline{a_i} \wedge vem_i) && \text{complemento } \wedge \\
 &= 0 \vee (a_i \wedge vem_i) \vee (\overline{a_i} \wedge vem_i) && \text{identidade } \vee \\
 &= (a_i \wedge vem_i) \vee (\overline{a_i} \wedge vem_i) && \text{distributiva} \\
 &= (a_i \vee \overline{a_i}) \wedge vem_i && \text{complemento } \vee \\
 &= 1 \wedge vem_i && \text{identidade } \wedge \\
 &= vem_i
 \end{aligned} \tag{6.6}$$

A Equação 6.6 mostra que a cadeia dos vai-um repassa, sem alteração, o bit $vem_0 = 1$ para vai_3 , enquanto todos os bits da diferença são gerados corretamente: $s_i = \overline{vem_i} = \overline{vem_0} = 0$. Para que o circuito produza o resultado esperado, que é $A - A = 0$ e $vai_3 = 0$, o bit vai_3 deve ser complementado na subtração. Essa complementação não pode ser ignorada na detecção de *overflow* nas subtrações.

6.4 Deslocamentos

Quando uma tupla de bits é interpretada como um número, segundo a notação posicional, um deslocamento pode ser utilizado para efetuar uma multiplicação ou divisão por uma potência de dois. Considere um número binário representado em quatro bits. Se este número é deslocado de uma casa para a direita, o resultado é o quociente da divisão inteira por dois. Se o número é deslocado para a esquerda, e a posição menos significativa preenchida com 0, o resultado é o número original multiplicado por dois, como mostrado abaixo. Os operadores $\gg$ e $\ll$ são aqueles da linguagem C: o operando à esquerda é deslocado do número de posições indicado pelo operando à direita.

$$\begin{array}{rcll} & 0110 & & \text{seis} \\ 0110 \gg 1 & = & 0011 & \text{três} \\ 0110 \ll 1 & = & 1100 & \text{doze} \end{array}$$

Um deslocamento de n posições à esquerda corresponde a uma multiplicação por 2^n ao passo que um deslocamento à direita corresponde a uma divisão por 2^n :

$$P \ll n = P \times 2^n, \quad Q \gg n = \frac{Q}{2^n}. \quad (6.7)$$

Para representação em complemento de dois, o deslocamento para a direita de números negativos deve garantir que o número permaneça negativo. Para tanto, o bit mais à esquerda, que é o bit de sinal, deve ser replicado. Se o deslocamento é para a esquerda, um número positivo pode tornar-se negativo. Por conta da diferença no tratamento de números com e sem sinal empregam-se os termos *deslocamento lógico* e *deslocamento aritmético*. No primeiro, o bit de sinal é ignorado e os bits inseridos são sempre zero, enquanto no deslocamento aritmético o bit de sinal é replicado nos deslocamentos à direita.

Exemplo 6.7 Vejamos dois exemplos, com números de complemento de dois representados em 4 bits. Um deslocamento para a direita deve preservar o bit de sinal, e produzir resultado negativo com operando negativo: $-2/2 = -1$.

$$1110 \gg 1 = 1111 \quad -2 \gg 1 = -1 \quad \text{certo}$$

Um deslocamento para a esquerda pode alterar o sinal de um operando positivo, resultando em erro: $6 \times 2 = 12 \neq -4$.

$$0110 \ll 1 = 1100 \quad +6 \ll 1 = -4 \quad \text{errado}$$

O deslocamento inverte o sinal e produz resultado errado: $6 \times 2 \neq -4$.

◁

A Figura 6.7 mostra um circuito, cuja saída tem cinco bits de largura, que efetua o deslocamento de uma posição para a esquerda. O comportamento deste deslocador é generalizado na Equação 6.8. Se $d = 0$ então $S = B$, enquanto se $d = 1$ então $S = B \times 2$.

$$\begin{aligned}
n &: \mathbb{N} \\
B &: \mathbb{B}^n \\
S &: \mathbb{B}^{n+1} \\
d &: \mathbb{B} \\
desl : (\mathbb{B} \times \mathbb{B}^n) &\mapsto \mathbb{B}^{n+1} \\
desl(n, d, B, S) &\equiv \begin{cases} s_0 = (0 \triangleleft d \triangleright b_0) \\ s_i = (b_{i-1} \triangleleft d \triangleright b_i) & 1 \leq i < n \\ s_n = (b_{n-1} \triangleleft d \triangleright 0) \end{cases}
\end{aligned} \tag{6.8}$$

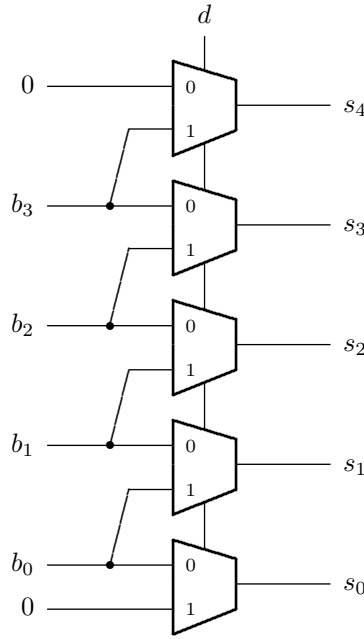


Figura 6.7: Deslocador de uma posição, com quatro bits de entrada.

6.4.1 Deslocador Exponencial

Deslocamentos de uma posição são úteis, mas deslocamentos de um número arbitrário de posições são um tanto mais úteis. Vejamos como a combinação apropriada de deslocadores simples nos permite deslocar um vetor de bits de até $N - 1$ posições, quando $N = 2^n$.

Iniciemos com um deslocador com 4 bits de largura, similar ao circuito da Figura 6.7. O circuito da Figura 6.8 desloca sua entrada B de uma posição se $d_2 = 1$, ou sua saída é igual à entrada: $C = (2^1 \times B) \triangleleft d_2 \triangleright (2^0 \times B)$.

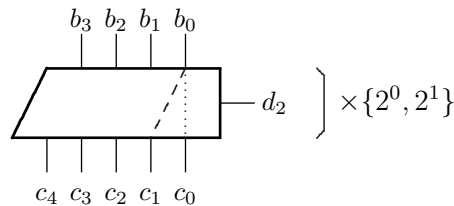


Figura 6.8: Deslocador de 4 bits, uma posição.

A composição de um deslocador de uma posição, com um deslocador de duas posições nos permite deslocar nenhuma, uma, duas, ou três posições, como mostrado na Figura 6.9, e se $D = \text{num}(d_4 d_2)$, então $C = 2^D \times B$.

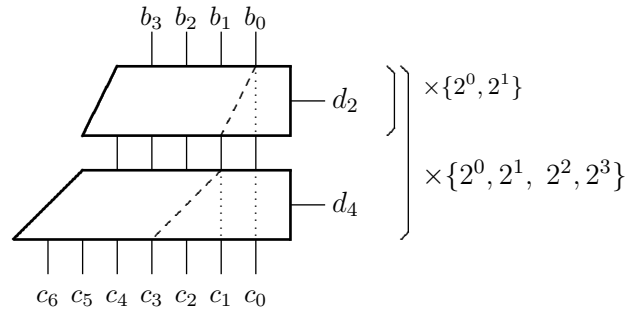


Figura 6.9: Deslocador de 4 bits, três posições.

Adicionando mais um deslocador de quatro posições, como mostra a Figura 6.10, o que se obtém é um deslocador idêntico ao de três posições quando $d_{16} = 0$, ou um deslocador que desloca de $4 + \{0, 1, 2, 3\}$ posições quando $d_{16} = 1$. Se $D = \text{num}(d_{16} d_4 d_2)$, então $C = 2^D \times B$.

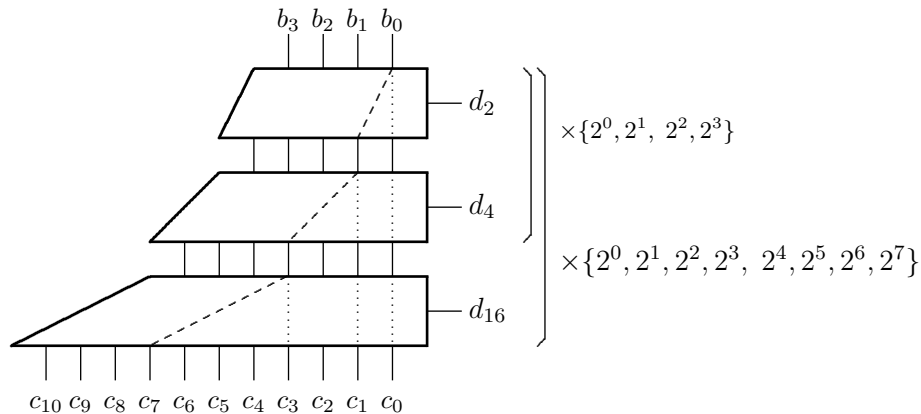


Figura 6.10: Deslocador de 4 bits, sete posições.

Se adicionarmos um deslocador de oito posições ao circuito da Figura 6.10, o resultado é idêntico ao do deslocador de sete posições quando $d_{256} = 0$, ou um resultado deslocado de $8 + \{0, 1, 2, 3, 4, 5, 6, 7\}$ posições quando $d_{256} = 1$. Neste caso, o circuito efetua deslocamentos de qualquer número de posições no intervalo $[0, 15]$. O deslocador de 15 posições é mostrado na Figura 6.11.

Adiante teremos uso para deslocadores de até 31 posições. Tal circuito é a extensão, agora já óbvia do deslocador de 15 posições: adicionamos um deslocador de 16 posições ao projeto da Figura 6.11, e assim é possível efetuar qualquer deslocamento no intervalo $[0, 31]$.

Tabela 6.5: Comportamento de um deslocador rotacional de quatro bits.

d_1d_0	s_3	s_2	s_1	s_0
00	b_3	b_2	b_1	b_0
01	b_2	b_1	b_0	b_3
10	b_1	b_0	b_3	b_2
11	b_0	b_3	b_2	b_1

Exercícios

Ex. 6.3 Mostre como implementar um deslocador exponencial com entrada de 4 bits, que desloca até quatro posições, empregando as portas de transmissão da Seção 5.7.2.

Ex. 6.4 Estenda o circuito da Figura 6.7 para que ele permita deslocamentos para a esquerda e para a direita. Atenção com os dois extremos s_0 e s_n .

Ex. 6.5 Estenda o circuito obtido no exercício anterior para que ele mantenha o sinal correto de números representados em complemento de dois.

Ex. 6.6 Estenda o circuito obtido no exercício anterior para que ele possa ser usado com números com e sem sinal. É necessária a adição de um sinal de controle para definir o tratamento do sinal.

Ex. 6.7 Altere o circuito do deslocador exponencial na Figura 6.7 para transformá-lo num deslocador rotacional.

Ex. 6.8 Implemente um deslocador rotacional de 8 bits usando multiplexadores com o número apropriado de entradas.

Ex. 6.9 Projete um circuito combinacional que produz em sua saída uma versão deslocada da entrada, conforme a especificação na Equação 6.10. Você deve determinar o valor de N .

$$\begin{aligned}
E &: \mathbb{B}^4 \\
S &: \mathbb{B}^N \\
d &: \mathbb{B}^3 \\
\text{circ} &: (\mathbb{B}^4 \times \mathbb{B}^3) \mapsto \mathbb{B}^N \\
\text{circ}(E, S, d) &\equiv S = E \times 2^d, \quad 2^d \in \{1, 4, 16, 64, 256, 1024, 4096, 16384\}
\end{aligned} \tag{6.10}$$

Ex. 6.10 Projete um circuito combinacional que produz em sua saída uma versão deslocada da entrada, conforme a especificação na Equação 6.11. Você deve determinar o valor de N . Qual a posição de E em S para $d = 0$?

$$\begin{aligned}
E &: \mathbb{B}^8 \\
S &: \mathbb{B}^N \\
d &: \mathbb{B}^3 \\
m &: \mathbb{B} \\
\text{circ} &: (\mathbb{B} \times \mathbb{B}^8 \times \mathbb{B}^3) \mapsto \mathbb{B}^N \\
\text{circ}(m, E, S, d) &\equiv [S = E \times 2^d] \triangleleft m \triangleright [S = E \div 2^d]
\end{aligned} \tag{6.11}$$

Ex. 6.11 Seu circuito contém dois deslocadores exponenciais com 32 bits de largura, um que desloca para a esquerda, e outro que desloca para a direita. Mostre como mascarar (isolar) os oito bits do centro ($b_4 \dots b_{11}$) de um vetor de 16 bits com dois deslocamentos. Por *mascarar* entenda-se que todos os bits que não pertencem ao octeto central devem ser zero após ao final das operações.

Ex. 6.12 Considere que N e M são vetores de 32 bits e representam inteiros sem sinal. Explique os valores atribuídos aos sinais P, Q, R, S como resultado da avaliação dos comandos da linguagem C listados abaixo.

```
unsigned int L, M, N, P, Q, R;
P = (L >> 6) << 12;
Q = (M & 0xffffffffc0) * 4096;
R = N & (16 - 1);
S = N & (1024 - 1);
```

6.5 Unidade de Lógica e Aritmética

Em seu curso de Programação já devem ter sido usadas as operações de soma, subtração, multiplicação, divisão, comparação de igualdade e de magnitude. Estas são ditas *operações de aritmética*. Algo como

```
x := x+1;

if (a = b) then ...

r1 := -b/4*a + squareroot(b*b - 4*a*c);
```

As *operações de lógica*, (**AND**, **OR**, **XOR**, **NOT**) serão empregadas mais adiante no curso.

As operações de lógica e aritmética são efetuadas no componente do processador chamado de *Unidade de Lógica e Aritmética* (ULA).

Uma vez que os operandos estejam disponíveis, o circuito da ULA efetua *todas* operações ao mesmo tempo, porque esta é a maneira mais simples e barata de construir o circuito. Dependendo do comando em C ou Pascal, somente uma dentre as possíveis operações (+, -, *, /, **AND**, **OR**, **XOR**, **NOT**) é escolhida pelo programador.

O circuito, em função do comando, escolhe um dos possíveis resultados – todos estão disponíveis, mas o programador escolhe um, e o circuito da ULA apresenta o resultado em função do comando em C ou Pascal.

Um programa em C ou Pascal é traduzido, pelo compilador, para uma sequência de operações simples, que estão disponíveis em *hardware* na ULA. Quando o programador escreve

```
x := x+1;
```

o programa traduzido pelo compilador para *linguagem de máquina* inclui a *instrução*

```
ADD x, x, 1
```

e o circuito da ULA apresenta em sua saída o resultado da soma dos dois operandos.

A *Unidade de Lógica e Aritmética* (ULA) é a unidade funcional de um processador na qual as operações sobre os dados são efetuadas. A ULA é um circuito combinacional com duas entradas de dados, uma de controle, e uma ou duas saídas de resultados. A Equação 6.12 define seu

comportamento e a Figura 6.12 mostra o símbolo usado para representá-la. A operação é selecionada pelo sinal F e, neste caso, pode ser uma dentre soma, subtração, deslocamento de uma posição, complemento, conjunção, ou-exclusivo ou disjunção.

$$\begin{aligned}
 &A, B, R : \mathbb{B}^4 \\
 &F, T : \mathbb{B}^3 \\
 &ULA-4x8 : [(\mathbb{B}^4 \times \mathbb{B}^4) \times \mathbb{B}^3] \mapsto (\mathbb{B}^4 \times \mathbb{B}^3) \\
 &ULA-4x8(A, B, F, R, T) \equiv (R, T) = \Omega_F(A, B) \\
 &\Omega \in \{+, -, \ll, \gg, \neg, \wedge, \oplus, \vee\}
 \end{aligned} \tag{6.12}$$

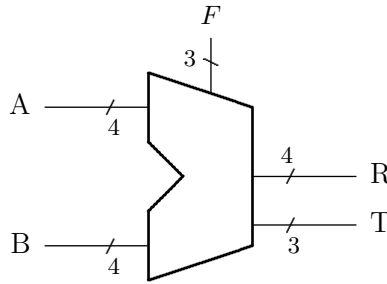


Figura 6.12: Unidade de Lógica e Aritmética.

Os operandos A e B são apresentados às entradas e o resultado R é obtido na saída da ULA. A saída T (*sTatus*) contém bits com informação sobre o resultado, indicando se aquele é zero ou negativo, por exemplo. O sinal F define qual é a operação efetuada sobre os operandos. As entradas A e B e a saída R são representadas por sinais de 4 bits; a escolha da função F e o *status* do resultado T são representados por sinais de 3 bits. Os operandos têm apenas 4 bits para simplificar a descrição do circuito. A Tabela 6.6 mostra os resultados das oito operações da ULA sobre o par de operandos $A = 0011$ e $B = 1100$; o significado das três colunas da direita (*status*) é explicitado adiante.

Tabela 6.6: Operações de lógica e aritmética sobre $A = 0011$ e $B = 1100$.

F	operação	resultado	status		
			n	z	v
0	$+$ $R = A + B$	$0011 + 1100 = 1111$	1	0	0
1	$-$ $R = A - B$	$0011 - 1100 = 0111$	0	0	0
2	$\ll$ $R = A \ll 1$	$0011 \ll 1 = 0110$	0	0	0
3	$\gg$ $R = A \gg 1$	$0011 \gg 1 = 0001$	0	0	0
4	$\neg$ $R = \overline{A}$	$\overline{0011} = 1100$	1	0	0
5	$\wedge$ $R = A \wedge B$	$0011 \wedge 1100 = 0000$	0	1	0
6	$\oplus$ $R = A \oplus B$	$0011 \oplus 1100 = 1111$	1	0	0
7	$\vee$ $R = A \vee B$	$0011 \vee 1100 = 1111$	1	0	0

Uma ‘fatia’ da ULA com largura de um bit é mostrada na Figura 6.13. Os circuitos para as outras fatias são idênticos; para construir uma ULA com N bits de largura, emprega-se N cópias da fatia de um bit. As extremidades são especiais por causa dos deslocamentos. O

sinal F escolhe uma das oito entradas do multiplexador, que é a operação a realizar sobre as entradas a_i e b_i .

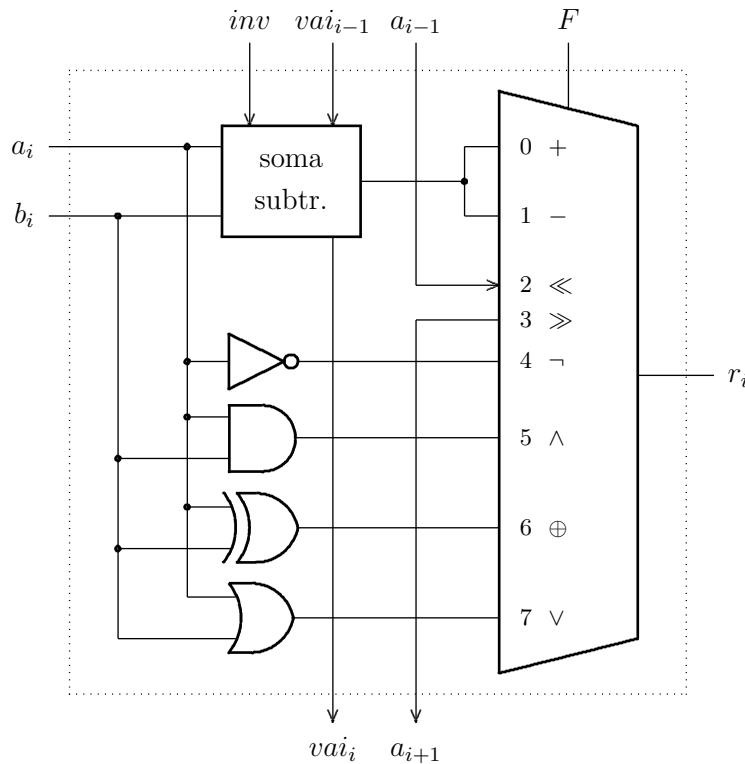


Figura 6.13: Fatia para o i -ésimo bit da ULA.

O circuito computa *todas as oito funções* simultaneamente e o sinal F determina qual resultado é apresentado à saída r_i . Nos deslocamentos, as entradas são os bits vizinhos, da esquerda e da direita, e $a_0 = 0$ no deslocamento à esquerda, e $a_3 = 0$ no deslocamento à direita.

6.5.1 Status

O circuito completo da ULA de 4 bits é obtido pela justaposição de quatro cópias do circuito da Figura 6.13, com as ligações apropriadas. O *status* da operação da ULA é obtido do sinal T , que indica se o resultado é negativo ($t_2 = 1$), se é zero ($t_1 = 1$), ou se ocorreu um vai-um do bit mais significativo ($t_0 = vai_3$). O uso de um mnemônico facilita lembrar o significado dos bits de *status*: $T = \langle t_2, t_1, t_0 \rangle = \langle n, z, v \rangle$, para ‘negativo’, ‘zero’ e ‘vai-um’. A implementação dos circuitos dos bits de *status* é discutida nos Exercícios 6.15 a 6.17.

6.5.2 Detecção de *overflow*

Considerando o circuito da Figura 6.6, que efetua somas e subtrações, devemos projetar um circuito que detecte a ocorrência de *overflow* tanto nas somas como nas subtrações. A Tabela 6.7 contém todas as combinações de sinais dos operandos – que podem ser positivos ou negativos – com operações – soma com $inv = 0$ e subtração com $inv = 1$.

Com base na Tabela 6.7, podemos derivar um sinal que indica a possibilidade de ocorrência de *overflow* somente com base nos sinais dos operandos e na operação. A possibilidade de que ocorra *overflow* é indicada por

$$poss\_ovfl = \neg(a_3 \oplus inv \oplus b_3). \quad (6.13)$$

Tabela 6.7: Possibilidade de *overflow* em circuito que soma e subtrai.

a_3	inv	b_3	<i>overflow</i>	razão
0	0	0	possível	soma com sinais iguais
0	0	1	não	soma com sinais distintos
0	1	0	não	subtração com sinais iguais
0	1	1	possível	subtração com sinais distintos
1	0	0	não	soma com sinais distintos
1	0	1	possível	soma com sinais iguais
1	1	0	possível	subtração com sinais distintos
1	1	1	não	subtração com sinais iguais

A técnica para detecção de *overflow*, indicada na Seção 6.1.2, não é muito prática porque envolve uma soma adicional no caminho mais longo do tempo de propagação de um somador – a ocorrência de *overflow* implicaria em duas linhas adicionais no diagrama da Figura 6.5, com o resultado disponível após 10 unidades de tempo.

Um método mais simples compara o vai-um das duas posições mais significativas: se eles são diferentes, então o resultado da soma não está representado corretamente. No nosso exemplo de 4 bits, para a soma temos

$$ovfl\_soma = vai_3 \oplus vai_2. \quad (6.14)$$

Por outro lado, numa subtração a representação é errada se os vai-um são iguais

$$ovfl\_subtr = \neg(vai_3 \oplus vai_2). \quad (6.15)$$

Combinando essas duas equações e lembrando que o sinal $inv = 0$ indica uma soma, temos

$$overflow = vai_3 \oplus vai_2 \oplus inv. \quad (6.16)$$

O circuito da Equação 6.16 provê o resultado em 9 unidades de tempo, se consideramos os parâmetros da Figura 6.5.

O código gerado pelo compilador da linguagem Pascal sinaliza a ocorrência de *overflow* nas operações com inteiros. O compilador para C ignora esses eventos e é responsabilidade da programadora detectar e corrigir os erros de representação. A Equação 6.13 indica que o resultado de uma soma ou subtração deva ser verificado e talvez corrigido.

Exercícios

Ex. 6.13 Desenhe um diagrama com o circuito completo da ULA com as 4 fatias e todas as ligações entre elas. Use uma folha A3 para que os circuitos dos exercícios abaixo possam ser incluídos no mesmo diagrama.

Ex. 6.14 Mostre como o circuito para inverter a entrada B do somador ($\overline{B} \triangleleft inv \triangleright B$) pode ser implementado com *uma* porta lógica de duas entradas.

Ex. 6.15 Projete os circuitos que computam os dois sinais de *status* da ULA: (i) um circuito para verificar se o resultado é negativo; (ii) um circuito para verificar se o resultado é zero.

Ex. 6.16 Estenda o somador para detectar a ocorrência de *overflow* e acrescente um bit de *status* à ULA. *Pista:* veja a Seção 6.5.2.

Ex. 6.17 Mostre como implementar comparações de igualdade ($A=B$) e de magnitude ($A<B$). *Pista:* use subtrações.

Ex. 6.18 Adicione os circuitos e ligações adicionais à ULA para permitir rotações de uma posição, além de deslocamentos de uma posição.

Ex. 6.19 Modifique o circuito do Exercício 6.18 para permitir deslocamentos lógicos e aritméticos. Note que estas adições implicam em aumentar o número de sinais de controle fazendo com que $|F| > 3$.

6.6 Somador com Adiantamento de Vai-um

A soma é uma operação deveras popular, e desde a década de 1940 projetistas se dedicam a inventar circuitos com tempos de propagação cada vez mais curtos. O caminho crítico de um somador é a cadeia de propagação do vai-um, e esse é o ponto de partida óbvio.

Considere o somador de 4 bits mostrado na Figura 6.4. Supondo que as entradas A e B e vem_0 estabilizem no instante t , o sinal vai_3 somente estabilizará após a propagação dos sinais através dos quatro somadores de um bit:

$$s_3 \longleftarrow s_2 \longleftarrow s_1 \longleftarrow s_0.$$

Supondo também que o tempo de propagação de uma porta lógica seja de uma unidade de tempo, então cada somador contribui com 2 unidades de tempo porque o sinal de vai-um é definido por

$$vai = a \wedge b \vee a \wedge vem \vee b \wedge vem,$$

e implementado com uma porta *or* de três entradas, mais três portas *and* de duas entradas. Dessa forma, para um somador de quatro bits, o sinal vai_3 estabiliza em 8 unidades de tempo, como mostra a Figura 6.5. Nessas condições, para um somador de 16 bits, o sinal vai_{15} estabilizaria após 32 unidades de tempo.

A Tabela 6.8 contém a tabela verdade de um somador completo de dois bits. A sexta coluna ($\mathbb{N}$) mostra o valor da soma na base 10. Dos mintermos 3 e 7 vemos que $vai = 1$ independentemente do valor de vem , donde

$$(a \wedge b) \Rightarrow (vai = 1). \quad (6.17)$$

Dos mintermos 1, 2, 5 e 6 vemos que, se $a \vee b$ então $vai = vem$, logo

$$(a \vee b) \Rightarrow (vai = vem). \quad (6.18)$$

Esses casos estão mostrados em negrito na Tabela 6.8.

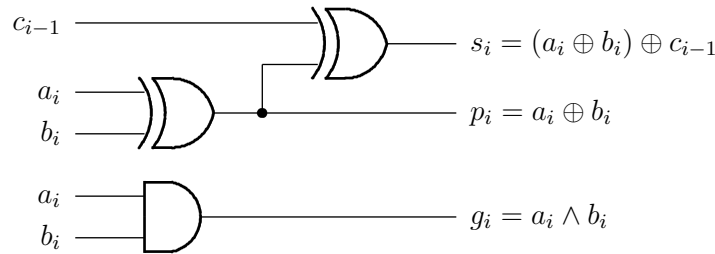
Tabela 6.8: Soma de dois bits.

<i>vem</i>	<i>a</i>	<i>b</i>	<i>vai</i>	<i>s</i>	$\mathbb{N}$	<i>g</i>	<i>p</i>
0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	1
0	1	0	0	1	1	0	1
0	1	1	1	0	2	1	1
1	0	0	0	1	1	0	0
1	0	1	1	0	2	0	1
1	1	0	1	0	2	0	1
1	1	1	1	1	3	1	1

As colunas *g* e *p* são, respectivamente, as funções *gera vai-um*: $g(a, b)$; e *propaga vai-um*: $p(a, b)$. A função $g(a, b) = 1$ sempre que o vai-um é 1. A função $p(a, b) = 1$ sempre que o vem-um deve ser propagado como vai-um. Com base nas Equações 6.17 e 6.18, a função que computa o vai-um pode ser escrita como

$$vai = g \vee (p \wedge vem). \quad (6.19)$$

O nome em Inglês para o ‘vai-um’ é *carry-out*, e por isso, no que segue, a letra *c* é usada para denotar os bits de vai-um nas cadeias de propagação. Um somador que computa os sinais de *gera vai-um* e *propaga vai-um* é mostrado na Figura 6.14. O sinal *p* pode ser implementado com um ou-exclusivo, ao invés de ou-inclusivo, porque quando as entradas *a* e *b* são 1, ambos *p* e *g* são 1.

**Figura 6.14: Circuito otimizado do somador completo.**

Um quarteto É possível reduzir o tempo de propagação da cadeia de vai-um quando se observa que todos os bits das entradas estão disponíveis ao mesmo tempo, e portanto todos os bits de vai-um podem ser computados simultaneamente: o c_3 depende das entradas a_3 e b_3 e de c_2 , que por sua vez depende de a_2 e b_2 e de c_1 , e assim sucessivamente. O que se quer é um circuito computa em paralelo, e então adianta, o valor de todos bits intermediários da cadeia de vai-um. Esta otimização é chamada de *adiantamento de vai-um*.

Considerando um somador de 4 bits, o vai-um dos bits menos significativo é:

$$c_0 = g_0 \vee (p_0 \wedge vem). \quad (6.20)$$

Temos dois casos:

- (i) se $a_0 \wedge b_0 = g_0 = 1$, então o vai-um é gerado pelos bits da posição 0;
- e
- (ii) $g_0 = 0$, e se $a_0 \vee b_0 = p_0 = 1$, então o vem-um é repassado para o somador da próxima posição.

Para o vai-um dos bits da próxima posição, $c_1(a_1, b_1, c_0)$, substituindo a expressão para c_0 obtemos

$$\begin{aligned} c_1 &= g_1 \vee (p_1 \wedge c_0) \\ &= g_1 \vee (p_1 \wedge [g_0 \vee (p_0 \wedge vem)]) \\ &= g_1 \vee (p_1 \wedge g_0) \vee (p_1 \wedge p_0 \wedge vem) \end{aligned} \quad (6.21)$$

São três as possibilidades: (i) ou o vai-um é gerado localmente (g_1), (ii) ou é gerado no somador vizinho e propagado ($p_1 \wedge g_0$), (iii) ou é gerado no vizinho do vizinho (vem) e propagado pelo vizinho ($p_1 \wedge p_0 \wedge vem$). Se parece ser um trava-língua é porque o é.

Se continuarmos com as substituições na derivação das equações para os dois bits de vai-um faltantes, obtemos as Equações 6.22 e 6.23. Estas equações refletem a intuição sobre o comportamento da cadeia de propagação do vai-um: ou o vai-um é gerado localmente, ou ele é propagado desde uma posição menos significativa.

$$c_2 = q_2 \vee (p_2 \wedge q_1) \vee (p_2 \wedge p_1 \wedge q_0) \vee (p_2 \wedge p_1 \wedge p_0 \wedge vem) \quad (6.22)$$

$$c_3 = g_3 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge g_1) \vee (p_3 \wedge p_2 \wedge p_1 \wedge g_0) \vee (p_3 \wedge p_2 \wedge p_1 \wedge p_0 \wedge vem) \quad (6.23)$$

Um somador de 4 bits com cadeia de adiantamento de vai-um é mostrado na Figura 6.15. Os sinais de vai-um adiantados são gerados nos trapézios.

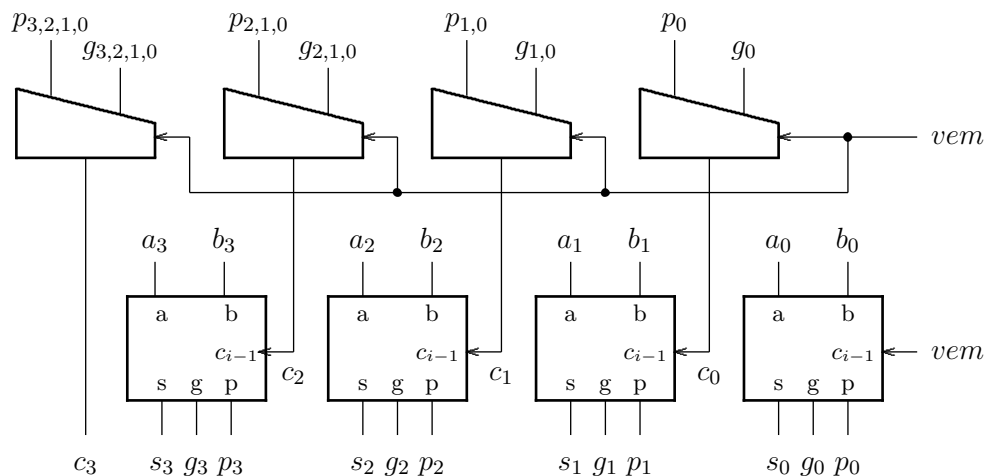


Figura 6.15: Somador de quatro bits com adiantamento de vai-um.

Se consideramos o circuito da Figura 6.14 e que o tempo de propagação de todas as portas lógicas é de uma unidade de tempo, então o valor da soma fica estável após 2 unidades de tempo e os sinais p e g estabilizam após 1 unidade. Se as entradas A, B, vem estabilizam no instante t , então podemos computar o tempo de propagação do somador de 4 bits otimizado:

s_0 estabiliza em $t + 2$ porque se propaga somente através de dois xor;

s_1 estabiliza em $t + 4$ porque se propaga através de portas *and* e *or* que geram p e g , mais um *and* seguido de um *or* para computar c_0 , e finalmente o xor de c_0 e $a_1 \oplus b_1$;

c_0, c_1, c_2, c_3 estabilizam em $t + 3$ porque se propagam através de uma soma de produtos – dois ou mais *and* seguidos de um *or*;

s_2, s_3 estabilizam em $t + 4$ porque se propagam através de um xor de c_{i-1} e $a_i \oplus b_i$.

Comparando-se com o somador original, o somador de 4 bits com adiantamento de vai-um produz resultados na metade do tempo. Este ganho de desempenho é (ligeiramente) superestimado porque o tempo de propagação das portas é proporcional ao número de entradas – o seu *fan-in*. Assim, o tempo de propagação de c_3 é algo mais longo do que o de c_1 . Uma estimativa precisa do tempo de propagação depende dos detalhes da implementação, e de temporização, das portas lógicas. Para circuitos de tamanho realista, emprega-se simulações detalhadas para determinar o tempo de propagação.

Resultados de simulação com tempos de propagação realistas são mostrados na Figura 6.16, quando se efetua a soma $0001 + 1111$. De cima para baixo são listadas as entradas $\text{inpa}[15:0]$ e $\text{inpb}[15:0]$, o resultado com um somador sem adiantamento $\text{rescadeia}[15:0]$, e os sinais individuais de soma e vai-um para os quatro somadores completos, de s_0 a s_3 . O resultado, determinado pelo sinal vai_3 , estabiliza após 190ps.

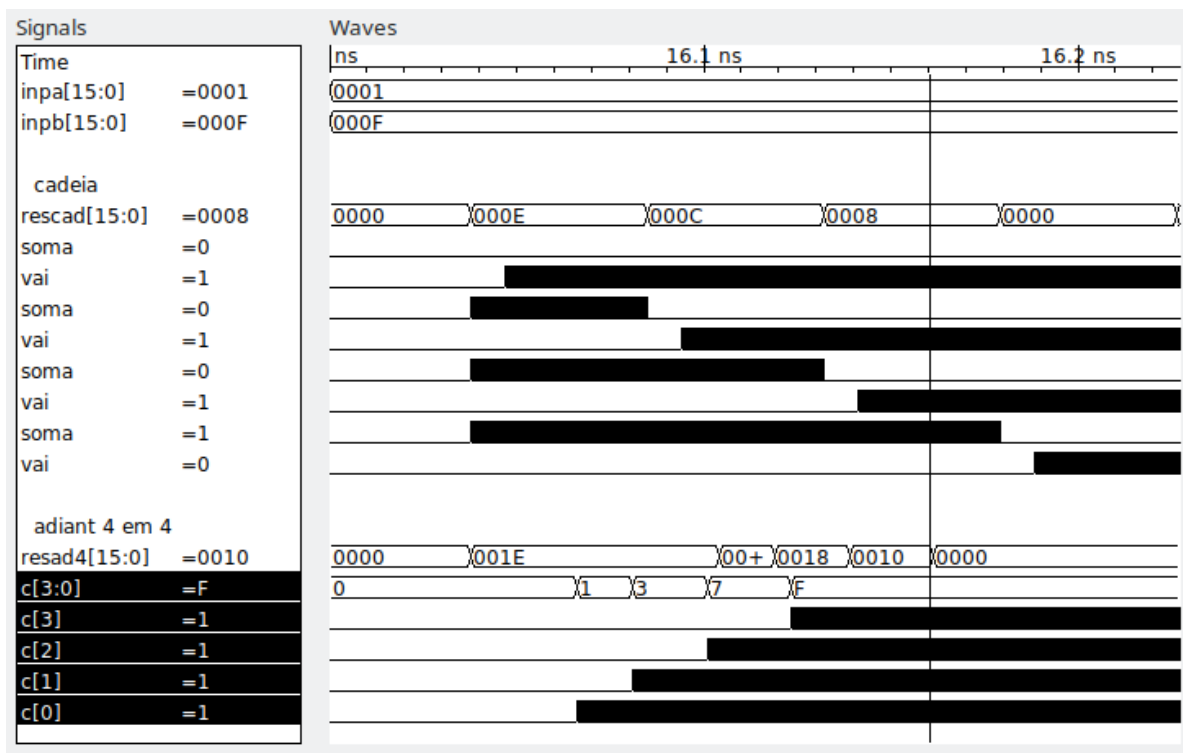


Figura 6.16: Simulação de somadores de 4 bits com adiantamento de vai-um.

O resultado para o somador com adiantamento de vai-um é $\text{resad4}[15:0]$; os bits de adiantamento são mostrados como um vetor de 4 bits, $c[3:0]$ e individualmente $c[3]$ a $c[0]$. A linha vertical indica o instante em que a soma estabiliza, o que ocorre após 160ps. Os sinais de Vai-um, os c_i , estabilizam antes que os sinais da soma s_i , que não é o que ocorre com o somador sem adiantamento.

O ganho obtido com o circuito mais complexo, que efetua a soma em $160/190 = 0,84$ do tempo do somador simples, pode não parecer suficiente em face da complexidade adicional. Contudo, a promessa de ganhos da ordem de $1/2$ se realiza quando aplicamos o adiantamento em somadores mais largos, como aqueles de 16 bits.

Quatro quartetos A mesma ideia do adiantamento de 4 bits pode ser empregada para adiantar o vai-um em somadores de 16 bits, considerando-se agora quatro grupos de 4 bits. As Equações 6.24 e 6.25 definem a propagação do vem-um através de um quarteto, e a geração do vai-um em qualquer posição do quarteto, respectivamente.

A propagação através de um quarteto pode ‘pular’ o quarteto se todos os seus quatro somadores propagariam o vem-um e isso é codificado na Equação 6.24, expandida para 16 bits.

$$\begin{aligned} P_0 &= p_3 \wedge p_2 \wedge p_1 \wedge p_0 \\ P_1 &= p_7 \wedge p_6 \wedge p_5 \wedge p_4 \\ P_2 &= p_{11} \wedge p_{10} \wedge p_9 \wedge p_8 \\ P_3 &= p_{15} \wedge p_{14} \wedge p_{13} \wedge p_{12} \end{aligned} \quad (6.24)$$

Ocorre vai-um no quarteto se: (i) ocorre vai-um do quarto somador; ou (ii) ocorre vai-um no terceiro somador e é propagado pelo quarto; ou (iii) ocorre vai-um no segundo somador e é propagado pelo terceiro e pelo quarto; ou (iv) ocorre vai-um no primeiro somador e é propagado pelos outros três. Essa mesma informação está codificado na Equação 6.23. A Equação 6.25 estende a computação de vai-um para os quatro quartetos de somadores.

$$\begin{aligned} G_0 &= g_3 \vee p_3 \wedge g_2 \vee p_3 \wedge p_2 \wedge g_1 \vee p_3 \wedge p_2 \wedge p_1 \wedge g_0 \\ G_1 &= g_7 \vee p_7 \wedge g_6 \vee p_7 \wedge p_6 \wedge g_5 \vee p_7 \wedge p_6 \wedge p_5 \wedge g_4 \\ G_2 &= g_{11} \vee p_{11} \wedge g_{10} \vee p_{11} \wedge p_{10} \wedge g_9 \vee p_{11} \wedge p_{10} \wedge p_9 \wedge g_8 \\ G_3 &= g_{15} \vee p_{15} \wedge g_{14} \vee p_{15} \wedge p_{14} \wedge g_{13} \vee p_{15} \wedge p_{14} \wedge p_{13} \wedge g_{12} \end{aligned} \quad (6.25)$$

A Equação 6.26 combina os sinais de geração e propagação de vai-um dos quartetos. C_0 depende unicamente do quarteto menos significativo (s_3 a s_0) e de vem . C_1 depende das suas entradas e de G_0 e P_0 . C_2 depende das suas entradas e de seus vizinhos $\langle G_0, P_0 \rangle$ e $\langle G_1, P_1 \rangle$. C_3 depende das suas entradas e de seus vizinhos $\langle G_0, P_0 \rangle$, $\langle G_1, P_1 \rangle$ e $\langle G_2, P_2 \rangle$. O sinal C_3 é o vai-um adiantado do somador na posição mais significativa: $C_3 = c_{15} = vai$.

$$\begin{aligned} C_0 &= G_0 \vee P_0 \wedge vem \\ C_1 &= G_1 \vee P_1 \wedge G_0 \vee P_1 \wedge P_0 \wedge vem \\ C_2 &= G_2 \vee P_2 \wedge G_1 \vee P_2 \wedge P_1 \wedge G_0 \vee P_2 \wedge P_1 \wedge P_0 \wedge vem \\ C_3 &= G_3 \vee P_3 \wedge G_2 \vee P_3 \wedge P_2 \wedge G_1 \vee P_3 \wedge P_2 \wedge P_1 \wedge G_0 \\ &\quad \vee P_3 \wedge P_2 \wedge P_1 \wedge P_0 \wedge vem \end{aligned} \quad (6.26)$$

O circuito do somador de 16 bits com adiantamento de vai-um é mostrado na Figura 6.17. Os somadores são agrupados em blocos de quatro, e o vai-um de cada quarteto é computado pelo circuito de adiantamento de vai-um, mostrado como um trapézio na figura. Cada quarteto implementa as Equações 6.20 a 6.23 e os sinais C_0 , C_1 , C_2 , e C_3 são gerados simultaneamente pelos quatro circuitos de adiantamento nos trapézios, com circuitos que implementam as Equações 6.26.

Exemplo 6.8 Façamos uma primeira estimativa para o tempo de propagação dos sinais das Equações 6.24, 6.25, e 6.26. Iniciemos pelo mais simples: o tempo de propagação dos P_i é determinado pela ligação dos p_i (t_{x2}) às quatro entradas de um *and-4* (t_{a4}) e portanto

$$T(P_i) = t_{a4} + t_{x2}. \quad (6.27)$$

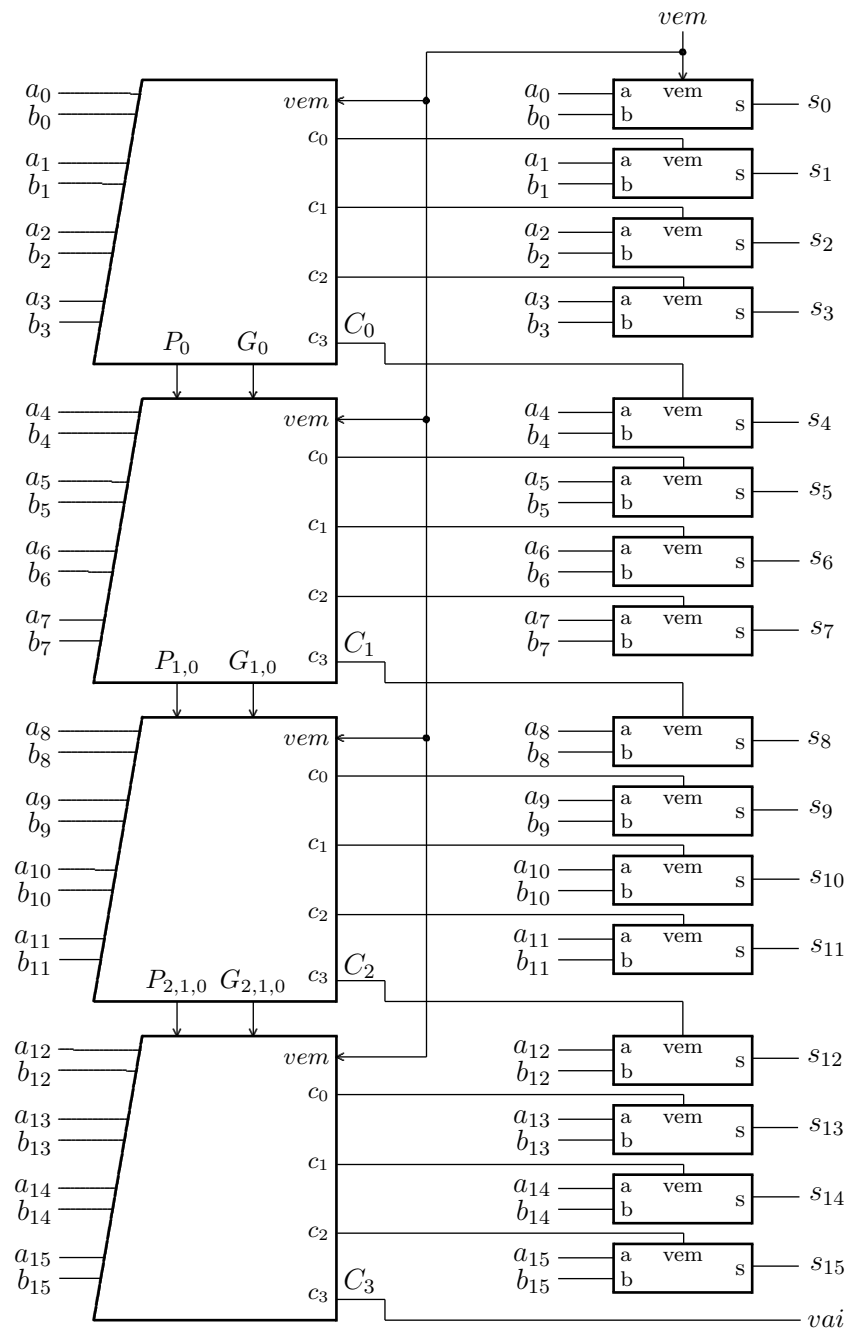


Figura 6.17: Somador de 16 bits com adiantamento de 4 quartetos.

As Equações 6.25 para os G_i consistem de um or -4 (t_{o4}) ligado a um and -4 (t_{a4}), que por sua vez é ligado aos and -2 (t_{a2}) e xor -2 de g_i e p_i

$$T(G_i) = t_{o4} + t_{a4} + \max(t_{x2}, t_{a2}). \quad (6.28)$$

Da Equação 6.26, temos para C_0

$$T(C_0) = t_{o2} + \max(T(G_0), t_{a2} + T(P_0)) \quad (6.29)$$

O tempo de propagação de C_3 é determinado pelos caminhos mais longos:

$$T(C_3) = t_{o5} + \max\{ [(t_{a4} + \max(T(P_i), T(G_i)))], [t_{a5} + T(P_i)] \}. \quad (6.30)$$

Aqui, para estimar o tempo de propagação supusemos que o tempo de propagação é dominado pelas portas com maior *fan-in*, e ignoramos seus *fan-out*'s. $\triangleleft$

6.7 Somador com Seleção de Vai-um

Uma alternativa para a implementação de somadores rápidos é indicada na Figura 6.18, que mostra um somador de 8 bits construído com três somadores de 4 bits. Este circuito é chamado de “somador com seleção de vai-um” (*carry select adder*) porque a metade mais significativa do resultado é *seleccionada* pelo vai-um da metade menos significativa.

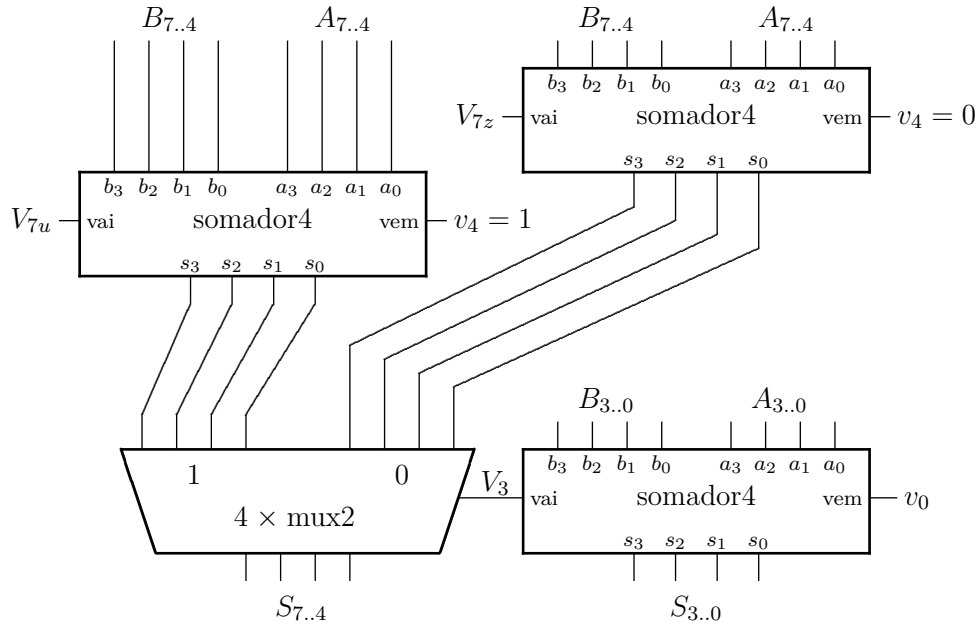


Figura 6.18: Somador com seleção de vai-um.

A parte menos significativa do resultado é obtida pela soma dos dois operandos e do vem-um:

$$\langle V_3 S_{3..0} \rangle = A_{3..0} + B_{3..0} + v_0.$$

A parte mais significativa da soma é obtida em paralelo com a parte menos significativa, computando-se simultaneamente, e de forma especulativa, duas alternativas para o vai-um da parte menos significativa, uma com $v_4 = 0$, e a outra com $v_4 = 1$.

$$\langle V_7 S_{7..4} \rangle = A_{7..4} + B_{7..4} + v_4.$$

Quando os sinais se propagam através do somador da parte menos significativa, e o valor de V_3 fica estável, este é usado para escolher um dos dois resultados parciais, através do multiplexador de 4 bits de largura.

Um diagrama de tempos com a soma de dois pares de valores é mostrado na Figura 6.19. A parte menos significativa da soma, $S_{3..0} = \alpha_{3..0} + \beta_{3..0} + v_0$ é computada pelo somador na parte de baixo da Figura 6.18, e este valor fica disponível após o tempo de propagação do somador T_{s4} .

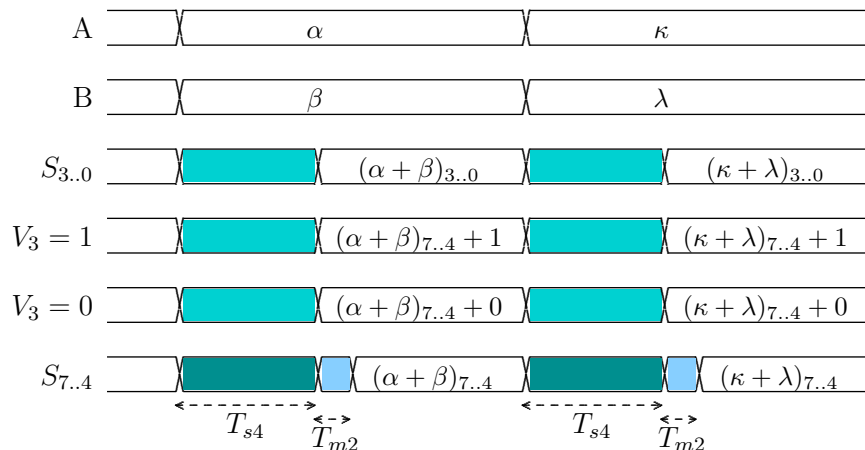


Figura 6.19: Temporização do somador com seleção de vai-um.

Simultaneamente, as duas versões da parte mais significativa da soma, $\alpha_{7..4} + \beta_{7..4} + 1$ e $\alpha_{7..4} + \beta_{7..4} + 0$ são computadas pelos somadores na parte de cima da Figura 6.18, e estes dois valores também ficam disponíveis após T_{s4} .

O valor final para a parte mais significativa da soma é escolhido pelo vai-um do somador dos bits menos significativos (V_3), e o valor $S_{7..4}$ fica definido e estável após os sinais se propagarem através do multiplexador (T_{m2}). Assim, o tempo de propagação deste somador é $T_{s4} + T_{m2}$, que é um tanto menor do que o tempo de propagação de dois somadores ligados em série: $2T_{s4}$.

O vai-um da soma em 8 bits depende do valor de V_3 , e é dado por $V_7 = (V_{7z} \vee V_3) \wedge V_{7u}$.

Infelizmente o tempo de propagação do somador de 8 bits é um tanto maior do que o de um somador de 4 bits por causa do multiplexador de duas entradas. O sinal V_3 , que seleciona o valor final da saída, é um sinal com um *fan-out* elevado. Quanto maior a largura do somador, maior é o *fan-out* de V_3 , e mais demorada a seleção da metade mais significativa do resultado.

Se o número de bits do somador da parte mais significativa for maior do que o da parte menos significativa, então o tempo de propagação desse somador pode compensar a propagação através do multiplexador. Por exemplo, se implementarmos o somador da parte menos significativa com 3 bits e o da parte mais significativa com 5 bits, então o tempo de propagação desse novo projeto seria mais curto que aquele do circuito da Figura 6.18.

Exercícios

Pista: Uma planilha pode ser útil para resolver estes problemas.

Ex. 6.20 Estime o tempo de propagação do somador de 16 bits com o adiantamento de vai-um definido na Equação 6.26, considerando que cada entrada além da segunda adiciona ao tempo de propagação da porta lógica 20% do seu tempo original. Considere que $T_p = 1$ para todas portas de duas entradas.

Ex. 6.21 Uma vez que você tenha resolvido o Ex. 6.20, considere uma implementação das Equações 6.24, 6.25 e 6.26, que emprega somente portas de duas entradas – isso é possível (e razoável) porque várias subexpressões aparecem repetidas nas equações. Compare o tempo de propagação desta implementação com aquela do Ex. 6.20.

Ex. 6.22 Estime o tempo de propagação de um somador de 32 bits com seleção de vai-um, implementado com três somadores de 16 bits iguais ao que você avaliou para responder ao Ex. 6.20. Considere que o *fan-out* não influencia o tempo de propagação das portas lógicas. Considere que $T_p = 1$ para todas portas lógicas de duas entradas.

Ex. 6.23 Repita o Ex. 6.22, agora considerando que cada saída adicional, além da primeira, piora o tempo de propagação da porta lógica em 10% do seu tempo original – cada incremento no *fan-out* da porta aumenta o seu tempo de propagação em 10%. Considere que $T_p = 1$ para todas portas lógicas de duas entradas, quando ligadas a uma única saída – com *fan-out* de 1.

6.8 Multiplicação

Efetue a multiplicação indicada abaixo *sem* nenhuma das simplificações que fazemos automaticamente ao multiplicar a mão.

$$\begin{array}{r} 1011 \\ \times 1001 \\ \hline \end{array}$$

Espaço em branco proposital.

A multiplicação, sem simplificações, é mostrada na Figura 6.20. A coluna da direita indica o dígito do multiplicador que multiplica cada parcela do resultado parcial. As parcelas multiplicadas por 0 são mostradas em *itálico*, e normalmente as ignoramos ao somar as parcelas do produto parcial.

$$\begin{array}{rcccccccc}
 & & 11 & & & & 1 & 0 & 1 & 1 \\
 & & \times 9 & & & \times & 1 & 0 & 0 & 1 \\
 \hline
 & & & & & & 1 & 0 & 1 & 1 & \times 1 \\
 & + & & & 1^1 & 0 & 1 & 1 & & & \times 0 \\
 & + & 1^1 & 0 & 1 & 1 & & & & & \times 0 \\
 & + & 1 & 0 & 1 & 1 & & & & & \times 1 \\
 \hline
 99 & & 1 & 1 & 0 & 0 & 0 & 1 & 1 & &
 \end{array}$$

Figura 6.20: Exemplo de multiplicação: 11×9 .

O resultado da multiplicação da Figura 6.20 é o mesmo, se os números representam valores na base 10, ou na base 2. Se consideramos a base 2, então $11 \times 9 = 99$ é o resultado, como mostra a Figura 6.20.

Se multiplicarmos dois números de n bits, o produto será representado em, no máximo, $2n$ bits porque $2^n \times 2^n = 2^{2n}$. Números com n bits representam valores no intervalo $[0, 2^n - 1]$, e

$$2^{2n-1} < (2^n - 1)^2 < 2^{2n}.$$

O produto de dois números de n bits é estritamente menor que 2^{2n} , e portanto a multiplicação não provoca *overflow*.

6.8.1 Acumulação de Produtos Parciais – Versão 1

Nossa primeira tentativa de implementar um multiplicador combinacional é uma reprodução exata do algoritmo da Figura 6.20, para um multiplicando X , multiplicador Y , e produto Z .

Cada parcela da soma é computada por um bloco que efetua o produto do multiplicando X pelo dígito correspondente àquela parcela do multiplicador, y_i . O bloco contém um somador que acrescenta o produto $X \times y_i$ à soma que acumula as parcelas ‘anteriores’ do produto.

Este bloco é especificado pela Equação 6.31. A soma dos dois operandos de 4 bits produz um resultado em 5 bits. Se $s = 0$, então o produto parcial desta parcela é zero; como a saída R tem 5 bits, um 0 é concatenado à esquerda da entrada A – o ‘&’ denota a concatenação. Do contrário, $s = 1$, e à soma das parcelas anteriores (A) é acrescentado mais um multiplicando (B).

$$\begin{aligned}
 s &: \mathbb{B} \\
 A, B &: \mathbb{B}^4 \\
 R &: \mathbb{B}^5 \\
 mp1 &: \mathbb{B} \times (\mathbb{B}^4 \times \mathbb{B}^4) \mapsto \mathbb{B}^5 \\
 mp1(s, A, B, R) &\equiv num(R) = [num(A) + num(B)] \triangleleft s \triangleright [0 \& A]
 \end{aligned} \tag{6.31}$$

A entrada A do primeiro bloco $mp1$ é zero porque a primeira parcela é, no máximo, $y_0 \times X$. De cada uma das parcelas extrai-se um bit definitivo do produto: da primeira resulta o bit z_0 , da segunda z_1 , da terceira z_2 , e da quarta z_3 – reveja a Figura 6.20. Da quarta parcela resultam

ainda os quatro bits mais significativos do produto, $\langle z_7, z_6, z_5, z_4 \rangle$. A Figura 6.21 mostra o multiplicador de 4×4 bits, com os multiplicando X e multiplicador Y no topo, e o produto Z na base.

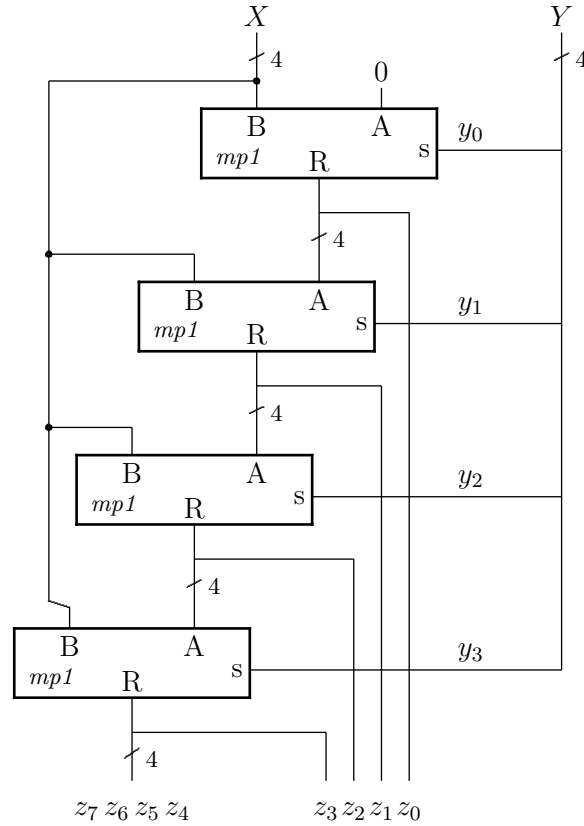


Figura 6.21: Primeira implementação da multiplicação em 4×4 bits.

O bloco que computa a primeira parcela consiste somente de quatro *mux-2*, que selecionam 0 se $y_0 = 0$, ou o multiplicando se $y_0 = 1$. Os demais blocos contém um somador seguido de um multiplexador. Se o tempo de propagação de um somador completo é T_S , e o de um multiplexador T_x , então o tempo de propagação deste circuito é de

$$T_{m1} = 3 \times 4 \times T_S + 4 \times T_x$$

porque a saída de cada *mp1* atravessa um *mux-2* e o valor do bit z_7 depende da propagação do vai-um através dos quatro somadores completos de cada uma das três últimas parcelas.

A Equação 6.31 parece indicar um multiplexador para selecionar um dentre adicionar uma nova parcela ($s_i = 1$), ou adicionar zero ($s_i = 0$). A leitora atenta certamente percebeu a otimização: ao invés de um multiplexador basta uma porta *and* para efetuar a multiplicação por 1, ou por 0, e com isso removemos duas portas lógicas do caminho crítico de cada parcela.

Para operandos com n bits, o tempo de propagação deste circuito é

$$T_{m1} \propto n \times (n - 1)$$

enquanto seu custo é

$$C_{m1} \propto n \times (n - 1) \approx n^2$$

porque são necessários $n \times (n - 1)$ somadores e n^2 portas *and*.

6.8.2 Acumulação de Produtos Parciais – Versão 2

Circuitos multiplicadores combinacionais que implementam diretamente o algoritmo da Figura 6.20 são chamados de *multiplicadores com acumulação de produtos parciais*. A tabela abaixo mostra a multiplicação de 1×1 bits, e esta função é implementada pela função $\wedge$.

$$\begin{array}{c|cc} \times & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array} \quad (6.32)$$

A Figura 6.22 explicita as posições dos dígitos dos operandos para que possamos construir um circuito multiplicador. O bit menos significativo do produto, p_0 é obtido diretamente da conjunção dos bits menos significativos dos operandos, a_0 e b_0 – para simplificar a figura, a conjunção é indicada pela justaposição de dois bits. O bit p_1 é a soma de duas conjunções e o vai-um desta soma é usado para computar o valor de p_2 . O valor do bit p_7 depende da soma de todas as parcelas, exceto p_0 .

				a_3	a_2	a_1	a_0	
				b_3	b_2	b_1	b_0	
				a_3b_0	a_2b_0	a_1b_0	a_0b_0	$A \times b_0$
+			a_3b_1	a_2b_1	a_1b_1	a_0b_1		$A \times b_1$
+		a_3b_2	a_2b_2	a_1b_2	a_0b_2			$A \times b_2$
+	a_3b_3	a_2b_3	a_1b_3	a_0b_3				$A \times b_3$
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Figura 6.22: Multiplicação de operandos de 4 bits.

Uma implementação direta da multiplicação é mostrada na Figura 6.23. Os blocos ‘ss’ são somadores simples, e os blocos ‘sc’ são somadores completos. A conjunção é indicada pela justaposição dos bits. Somadores simples são usados nas somas de dois bits, e completos nas somas de três. O caminho mais longo, e que portanto determina o tempo de propagação deste circuito, é o somador com seis bits de largura, que computa p_7 .

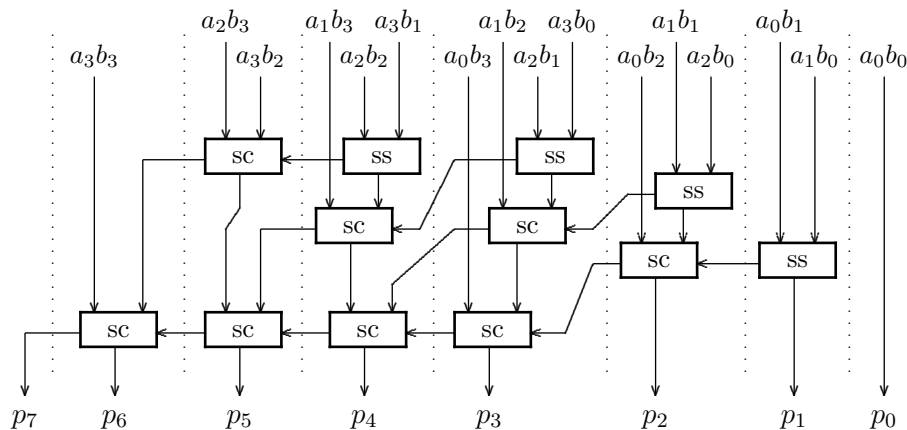


Figura 6.23: Implementação irregular da multiplicação em 4×4 bits.

Esta implementação ingênua da multiplicação é um circuito correto porém irregular – sua implementação num circuito integrado causaria desperdício de espaço porque o leiaute do

multiplicador não se presta facilmente a uma geometria regular e retangular. Uma consequência da falta de regularidade é que a extensão deste circuito para operandos de 8 bits é intuitiva, embora nada da versão de 4 bits possa ser reaproveitado. Vejamos como a reorganização da soma nos ajuda a projetar um circuito com leiaute regular.

6.8.3 Acumulação de Produtos Parciais – Versão 3

Vejamos uma segunda implementação de um circuito multiplicador com acumulação de produtos parciais. Este circuito combinacional é eficiente em termos de forma e área, e seu tempo de propagação é proporcional a $2n$ para operandos de n bits.

Sejamos um pouco mais ambiciosos e tentemos multiplicar de dois em dois bits, como mostrado abaixo para $3 \times 2 = 6$.

$$\begin{array}{r} 3 \\ \times 2 \\ \hline 6 \end{array} \quad \begin{array}{r} 1 \ 1 \\ \times 1 \ 0 \\ \hline 1 \ 1 \\ + 1 \ 1 \\ \hline 1 \ 1 \ 0 \end{array} \begin{array}{l} \times 0 \\ \times 1 \end{array}$$

Se chamamos o multiplicando de A e o multiplicador de B , e indicamos os bits mais e menos significativos pelos subscritos H e L respectivamente, podemos reescrever as quatro multiplicações separadamente:

$$\begin{array}{r} A_L \quad 1 \\ B_L \quad \times 0 \\ \hline 00 \end{array} \quad \begin{array}{r} A_H \quad 1 \\ B_L \quad \times 0 \\ \hline 00 \end{array} \quad \begin{array}{r} A_L \quad 1 \\ B_H \quad \times 1 \\ \hline 01 \end{array} \quad \begin{array}{r} A_H \quad 1 \\ B_H \quad \times 1 \\ \hline 01 \end{array}$$

Repetindo a soma das parcelas, agora usando os resultados dos pares de bits, obtemos o mesmo resultado, 6:

$$\begin{array}{r} 0 \ 0 \quad A_L B_L \\ 0 \ 0 \quad A_H B_L \\ 0 \ 1 \quad A_L B_H \\ 0 \ 1 \quad A_H B_H \\ \hline 0 \ 1 \ 1 \ 0 \end{array}$$

Agora o “pulo do gato”: se reordenarmos as parcelas desta soma, obtemos um bloco funcional que pode ser replicado na implementação de multiplicadores com operandos mais largos do que dois bits, e este bloco é mostrado na Figura 6.24.

Vejamos como se dá o próximo passo na construção do multiplicador. Tentemos multiplicar dois números de 4 bits, da mesma forma que fizemos com operandos de 2 bits. Efetuemos $8 \times 9 = 72$, agora usando $A_{\{L,H\}}$ e $B_{\{L,H\}}$ para representar *pares* de bits:

$$8_{10} = 1000_2, \quad A_H = 10, \quad A_L = 00$$

$$9_{10} = 1001_2, \quad B_H = 10, \quad B_L = 01$$

$$\begin{array}{r} A_L \quad 00 \\ B_L \quad \times 01 \\ \hline 0000 \end{array} \quad \begin{array}{r} A_H \quad 10 \\ B_L \quad \times 01 \\ \hline 0010 \end{array} \quad \begin{array}{r} A_L \quad 00 \\ B_H \quad \times 10 \\ \hline 0000 \end{array} \quad \begin{array}{r} A_H \quad 10 \\ B_H \quad \times 10 \\ \hline 0100 \end{array}$$

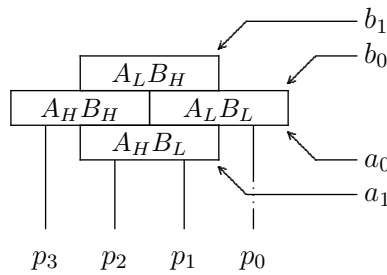


Figura 6.24: Bloco que multiplica dois operandos de 2 bits.

Cada um destes produtos pode ser quebrado em quatro produtos de pares de bits, cada novo quarteto reordenado como na Figura 6.24, e os quatro blocos agrupados como indicado na Figura 6.25. O conteúdo de cada bloco indica os dígitos dos operandos de A e de B .

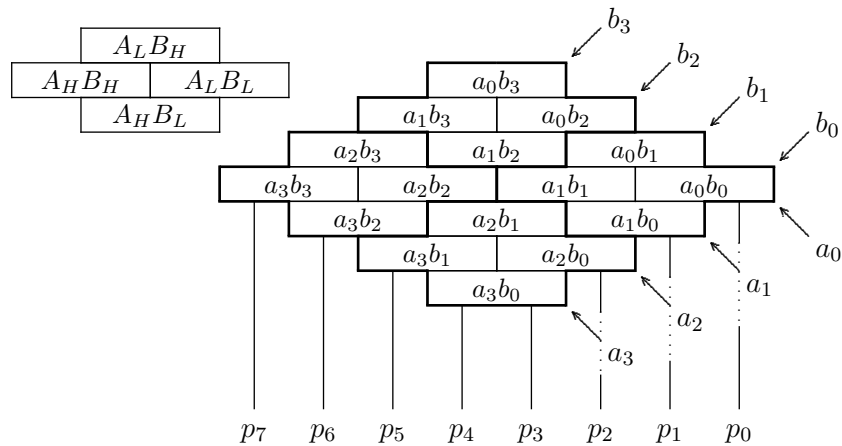


Figura 6.25: Implementação eficiente da multiplicação em 4×4 bits.

Os bits dos operandos atravessam o circuito na diagonal, enquanto os dígitos do produto são computados nas verticais.

Falta-nos projetar o bloco que efetua o produto de um par de bits e o soma com o produto parcial. Este bloco é mostrado na Figura 6.26. Os dois somadores devem efetuar a soma do produto $a_j \wedge b_i$ com o *vem-um* do bloco à direita (v_k), dois bits do produto parcial da parcela de cima ($s_{m,k+1}$) e ($s_{m,k}$), e produzem três bits do resultado parcial: ($s_{n,k+1}$) e ($s_{n,k}$) são encaminhados para a parcela de baixo, e v_{k+2} é o vai-um para o próximo bloco.

Com operandos com n bits, são necessários n^2 portas *and* para efetuar as multiplicações, mais $2n^2$ somadores, e portanto o custo C desse circuito é

$$C_{m3} \propto n^2.$$

Os somadores são distintos: ‘sc’ é um somador completo e ‘ss’ é um somador simples. Algumas otimizações são possíveis nas bordas do multiplicador: os blocos na borda direita não fazem uso da entrada v_k e o somador completo pode ser simplificado nestes blocos. O bit p_0 do produto não usa o somador completo. Note ainda que as maiores colunas (p_3 e p_4) têm somente sete parcelas.

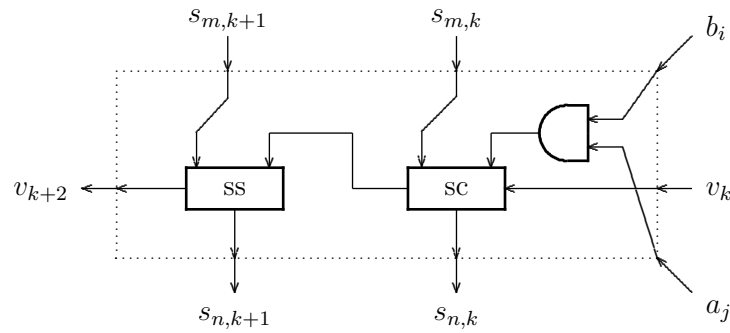


Figura 6.26: Bloco que multiplica e acumula os produtos parciais.

Quanto ao tempo de propagação, o maior somador tem largura $2n$. Para qualquer bit do produto, os sinais se propagam através de até n blocos na horizontal, seguidos de até n blocos na vertical, e portanto o tempo de propagação deste circuito é proporcional a $2n$:

$$T_{m3} \propto 2n.$$

6.9 Teste de Circuitos Aritméticos

É necessário um trabalho de detetive para localizar erros em circuitos que implementam as operações aritméticas. Usaremos somadores para exemplificar, mas as mesmas técnicas podem ser usadas para testar multiplicadores.

No que segue, usaremos a notação de VHDL para a base dos números: `b"0101"` indica que a sequência é representada em binário (prefixo `b`), `x"89AB"` indica que a sequência é representada em hexadecimal (prefixo `x`).

A Figura 6.27 mostra um somador de 4 bits. Quais padrões de bits, nas entradas A e B são necessários para evidenciar erros na implementação do circuito, ou no modelo que descreve o circuito?

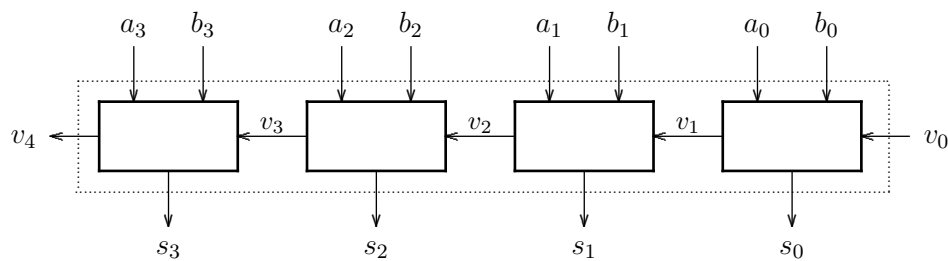


Figura 6.27: Circuito de testes do somador.

São necessários, ao menos, três tipos de testes: (i) testes para verificar se somas ‘simples’ produzem resultados corretos – quais os resultados esperados para a soma das parcelas `b"0001"+b"0000"` e `b"0010"+b"0000"`? (ii) testes para verificar a propagação do vai-um – quais os resultados esperados para a soma das parcelas `b"0001"+b"0001"` e `b"0010"+b"0010"`? (iii) testes para verificar se o modelo exibe as propriedades aritméticas da soma, no caso a comutatividade, operações com o elemento neutro, e operações que provocam *overflow* – reveja o Exemplo 6.4.

A operação do circuito nos extremos das faixas de valores deve ser examinada com cuidado: operações com valores pequenos e valores grandes devem produzir os resultados corretos.

Vejamos alguns padrões de teste para somadores de 16 bits. Como diria Orwell, alguns números são mais úteis do que outros. Que tal x"1111" com x"1111"?

$$\begin{array}{r} 1111 \\ +1111 \\ \hline 2222 \end{array} \quad \begin{array}{r} 0001\ 0001\ 0001\ 0001 \\ +0001\ 0001\ 0001\ 0001 \\ \hline 0010\ 0010\ 0010\ 0010 \end{array}$$

Na coluna da esquerda os números estão representados em hexadecimal, e na da direita em binário. Além deste par, outros pares de operandos são úteis: x"2222", x"4444", x"8888". Quais erros estes quatro padrões evidenciam?

O que se descobre somando x"5555" com x"5555"?

$$\begin{array}{r} 5555 \\ +5555 \\ \hline AAAA \end{array} \quad \begin{array}{r} 0101\ 0101\ 0101\ 0101 \\ +0101\ 0101\ 0101\ 0101 \\ \hline 1010\ 1010\ 1010\ 1010 \end{array}$$

O que se descobre somando x"AAAA" com x"AAAA"?

$$\begin{array}{r} AAAA \\ +AAAA \\ \hline 1\ 5554 \end{array} \quad \begin{array}{r} 1010\ 1010\ 1010\ 1010 \\ +1010\ 1010\ 1010\ 1010 \\ \hline 1\ 0101\ 0101\ 0101\ 0100 \end{array}$$

Quais erros estes dois padrões (x"5555" e x"AAAA") evidenciam, que não são possíveis descobrir com os padrões anteriores (x"1111" a x"8888")?

O que se descobre com x"FFFF"+x"0001" e x"0001"+x"FFFF"?

Esses operandos nos ajudam a localizar a posição dos bits em que ocorrem os erros; uma vez localizados, examina-se o modelo para encontrar o erro, ou empregam-se novos testes até que o problema seja encontrado. O que se tenta com esta estratégia é “cercar o erro”.

Cada padrão usado nos testes *deve* ser o mais simples possível para evidenciar um determinado tipo de erro. Nos exemplos mostrados acima, somente porções bem delimitadas do somador são exercitadas em cada um dos testes.

A operação do circuito nos extremos das faixas de valores deve ser examinada com cuidado: operações com valores pequenos e valores grandes devem produzir os resultados corretos e, se for o caso, *overflow* deve ser sinalizado, conforme reza a Seção 6.5.2.

Uma vez que acreditemos que o circuito, ou seu modelo, esteja correto, os testes devem ser repetidos para garantir que *todos* os erros foram sanados – sempre existe o risco de que nossas correções possam ter introduzido novos erros.

Para testar exaustivamente um somador de 16 bits são necessários $2^{16} \times 2^{16} \times 2 > 8,5$ bilhões de padrões distintos. Se o somador tem operandos de 32 bits, ou de 64-bits, testes exaustivos são obviamente inviáveis. Um fator importantíssimo na geração de casos de teste é gerar tuplas que elucidem *todos* os possíveis erros, sem que o tempo necessário para conduzir² os testes seja impraticável.

²Reza a lenda que o time de projeto do processador Pentium IV da Intel consistia de 250 pessoas enquanto o time de verificação consistia de 750 pessoas.

Exercícios

Ex. 6.24 Escreva um conjunto de tuplas $\langle A, B, vem, S, vai \rangle$ para capturar erros na implementação de somadores para operandos de 16 bits. Sua lista deve ser a mais curta possível – tempo é dinheiro – ao mesmo tempo em que a cobertura – variedade de erros descobertos – seja máxima.

Ex. 6.25 Escreva um conjunto de tuplas $\langle A, B, R \rangle$ para capturar erros na implementação de multiplicadores para operandos de 16 bits. Sua lista deve ser a mais curta possível ao mesmo tempo em que sua cobertura seja máxima.

Ex. 6.26 Escreva um conjunto de tuplas $\langle A, B, F, R, T \rangle$ para capturar erros na implementação da ULA de 4 bits da Seção 6.5. Sua lista deve ser a mais curta possível ao mesmo tempo em que sua cobertura seja máxima.

Ex. 6.27 Complete o circuito da Figura 6.25, acrescentando todas as ligações que faltam na periferia. *Pista:* veja o bloco da Figura 6.26.

Ex. 6.28 Efetua as otimizações possíveis no circuito da sua resposta ao Ex. 6.27.

Índice Remissivo

Símbolos

T_A , 64
 T_D , 120
 T_I , 109, 116
 T_M , 117, 119
 T_P , 116
 T_p , 171
 $T_{C,x}$, 118–120
 $\%$, 18–20
 $\Rightarrow$, 37
 $\bigvee$, 47
 $\bigwedge$, 47
 $\equiv$, 36
 $\gg$, 155
 $\wedge$, 30, 36
 $\triangleleft \triangleright$, 36, 42, 66
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 41, 44
 $\ll$, 155
 $\neg$, 30, 36, 154
 $\vee$, 30, 36
 $\oplus$, 36, 53, 65
 $\mapsto$, 42
 $\mathbb{N}$, 146
 $\bar{a}$, veja $\neg$
 π , 25
 $\backslash$, 77
decod, 72
demux, 75
e, 24
mod, 158
mux, 67
 $\mathbb{B}$, 29–33, 38, 42
 $\mathbb{Z}$, 42, 149
 $\mathbb{N}$, 42, 43, 149
num, 44, 55, 72, 77
 num^{-1} , 55
 $\mathbb{R}$, 42
 $\langle \rangle$, 29, 42, 43, 180

Números

74148, 75

A

abstração, 27
 bits como sinais, 27–33, 57
 tempo discretizado, 116, 118
 acumulação de produtos parciais, 173–178
 adição, 152
 adiantamento de vai-um, 164–172

alfabeto, 17
 Álgebra de Boole, 27
 algoritmo,
 conversão de base, 18
 conversão de base de frações, 22
 amplificador, 125, 136
 diferencial, 138
 and, veja $\wedge$
and array, 128
 aproximação, 24
 árvore, 64, 118
assembly, veja ling. de montagem
 associatividade, 31
 atraso, veja tempo de propagação, 57
 atribuição, 12

B

barramento, 140
barrel shifter, 158
 básculo, 138
 binário, 20
 bit, 20
 de sinal, 147
 bits, 27–37, 65
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 variável, 30
branch, veja desvio condicional
buffer, 123
buffer three-state, 140
 buraco, 87
 byte, 11

C

C,
 deslocamento, 160
 overflow, 163
 cadeia,
 de portas, 64, 118
 de somadores, 152
 caminho crítico, 117
 capacitor, 105, 107, 135, 136
capture FF, veja *flip flop*, destino
 CAS, 134
 célula de RAM, 81
 célula, 100
 chave,
 analógica, 142

digital, 92
 normalmente aberta, 92
 normalmente fechada, 78, 92
 ciclo,
 combinacional, 57
 violação, 81
 circuito,
 combinacional, 57, 65
 dual, 99
 circuito aberto, 57
 clk, *veja* relógio
 clock, *veja* relógio
 clock skew, *veja* skew
 CMOS, 59, 65, 85–142
 buffer three-state, 140
 célula, 100
 inversor, 96
 nand, 99
 nor, 98
 porta de transmissão, 141
 portas inversoras, 99
 sinal restaurado, 125
 código,
 Gray, 63
 Column Address Strobe, *veja* CAS
 combinacional,
 ciclo, 57
 circuito, 57
 dispositivo, 57
 comparador,
 de igualdade, 62, 164
 de magnitude, 164
 Complementary Metal-Oxide Semiconductor, *veja* CMOS
 complemento, *veja* $\neg$
 complemento de dois, 145–151, 154–164
 complemento, propriedade, 31
 comportamento transitório, *veja* transitório
 comutatividade, 31
 condicional, *veja* $\triangleleft \triangleright$
 condutor, 85
 conjunção, *veja* $\wedge$
 conjunto mínimo de operadores, 65
 contra-positiva, 41
 controlador,
 de memória, 136
 conversão de base, 18
 corrente, 85, 105, 106, 112, 113
 de fuga, 115
 corrida, 123, 125
 curto-circuito, 57

D

datapath, *veja* circuito de dados
 decimal, 17
 decodificador, 72, 78, 81–82, 84, 126
 de linha, 126, 135
 de prioridades, 74
 delay, 57

demultiplexador, 75, 120
 design unit, *veja* VHDL, unidade de projeto
 deslocador exponencial, 156
 deslocamento, 155–158
 aritmético, 155, 158, 164
 exponencial, 159
 lógico, 155, 162
 rotação, 158
 detecção de bordas, 123
 disjunção, *veja* $\vee$
 dispositivo, 85
 combinacional, 57
 distributividade, 31, 34, 51
 divisão inteira, 44
 doador, 87
 don't care, 70
 dopagem por difusão, 86
 dopante, 86
 DRAM, 134–137
 controlador, 136
 fase de restauração, 137
 linha,
 de palavra, 135
 linha de bit, 135
 linha de palavra, 136
 página, 135
 refresh, 135
 dual, 32, 95, 99
 dualidade, 32

E

EEPROM, 133
 endereço, 77
 energia, 100, 105, 112–115
 enviesado, relógio, *veja* skew
 EPROM, 133
 equivalência, *veja* $\Leftrightarrow$
 erro,
 de representação, 23
 especificação, 42
 expressões, 36

F

fan-in, 109–112, 116, 167, 170
 fan-out, 82, 84, 109–112, 116, 120, 170–172
 fechamento, 31
 FET, 91
 Field Effect Transistor, *veja* FET
 Field Programmable Gate Array, *veja* FPGA
 flip-flop,
 modelo VHDL, *veja* VHDL, *flip-flop*
 um por estado, *veja* um FF por estado
 forma canônica, 48
 frações, *veja* ponto fixo
 frequência máxima, *veja* relógio
 função, 30
 tipo, 29, 42
 função, aplicação bit a bit, 32
 função, tipo (op. infixo), *veja* $\mapsto$

G

glitch, *veja* transitório
 GND, 93
 gramática, 17

H

hexadecimal, 19

I

idempotência, 31
 identidade, 31
 igualdade, 30
 implementação, 42
 implicação, *veja* $\Rightarrow$
 informação, 16
 Instrução,
 busca, *veja* busca
 instrução, 12
 busca, *veja* busca
 decodificação, *veja* decodificação
 execução, *veja* execução
 resultado, *veja* resultado
 interface,
 de rede, 13
 de vídeo, 12
 inversor, 96
 tempo de propagação, 109
 involução, 31, 61
 isolante, 85

J

Joule, 112
jump, *veja* salto incondicional

L

latch, *veja* basculado
latch FF, *veja flip flop*, destino
launch FF, *veja flip flop*, fonte
 Lei de Joule, 112
 Lei de Kirchhoff, 106
 Lei de Ohm, 105, 106
 ligação,
 barramento, 140
 em paralelo, 93, 99
 em série, 93, 99
 linguagem,
 assembly, *veja* ling. de montagem
 C, *veja* C
 Pascal, *veja* Pascal
 VHDL, *veja* VHDL
 Z, 27
 linha de endereçamento, 78
 literal, 38
 logaritmo, 43
 lógica restauradora, 125

M

Mapa de Karnaugh, 49, 124
 Máquina de Mealy, *veja* máq. de estados
 Máquina de Moore, *veja* máq. de estados

máscara, 32
 máximo e mínimo, 31
 maxtermo, 46
 Mealy, *veja* máq. de estados
 memória,
 atualização, 77
 de vídeo, 13
 decodificador de linha, 80
 endereço, 77
 FLASH, 133
 matriz, 129, 132, 134
 multiplexador de coluna, 80
 primária, 13
 RAM, 81, 134
 ROM, 78, 126
 secundária, 13
 memória dinâmica, *veja* DRAM
 memória estática, *veja* SRAM
 mintermo, 45, 124, 126
 modelo,
 funcional, 44
 porta lógica, 96
 temporização, 115
 módulo, *veja* %, *mod*
 Moore, *veja* máq. de estados
 MOSFET, 91
 multiplexador, 61, 66–70, 80, 101, 117, 119, 123–
 124, 126, 141, 142
 de coluna, 132, 136
 multiplicação, 172–178
 acumulação de produtos parciais, 173–178
multiply-add, *veja* MADD

N

número,
 de Euler, 24
 negação, *veja* $\neg$
 nível lógico,
 0 e 1, 28
 indeterminado, 28, 109, 116, 141
 terceiro estado, 140
 nó, 96
non sequitur, 37
 not, *veja* $\neg$
 número primo, 53

O

octal, 18
 operação,
 binária, 29
 bit a bit, 32
 infixada, 42
 prefixada, 47
 unária, 29
 operações sobre bits, 29–33
 operador,
 binário, 29
 lógico, 36
 unário, 29

operation code, veja *opcode*
 or, veja $\vee$
 or array, 129
 ou exclusivo, veja $\oplus$
 ou inclusivo, veja $\vee$
 overflow, 148–150, 154, 162–164, 173, 178

P

paridade,
 ímpar, 49
 par, 49
 período mínimo, veja relógio
pipelining, veja segmentação
 piso, veja $\lfloor \cdot \rfloor$
 ponto fixo, 151–152
 ponto flutuante, 70
 porta lógica, 65
 and, 59
 carga, veja *fan-out*
 de transmissão, 141
 nand, 60, 99
 nor, 60, 98
 not, 59, 96
 or, 59
 xor, 60, 65
 portas complexas, 100
 potência, 112–115
 dinâmica, 114
 estática, 115
 potenciação, 43
 precedência, 30
 precisão,
 representação, 23
 prioridade,
 decodificador, 74
 processador, 12
 produtivo, 33
 produto de somas, 46
 programa de testes, veja VHDL, *testbench*
 PROM, 133
 propriedades, operações em $\mathbb{B}$, 31
 prova de equivalência, 40–41
pull-down, 96
pull-up, 96, 126, 141
 pulso, 122, 123
 espúrio, veja transitório

R

RAM, 12, 77, 81, 134–139
 célula, 81
 dinâmica, 135
Random Access Memory, veja RAM
 RAS, 134
Read Only Memory, veja ROM
 realimentação, 81
 receptor, 87
 rede, 96
 redução, 33
refresh, 137, 139

Register Transfer Language, veja RTL
 registrador de deslocamento,
 modelo VHDL, veja VHDL, registrador
 relógio,
 enviesado, veja *skew*
 representação,
 abstrata, 28
 binária, 20
 complemento de dois, 147
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 ponto fixo, 151
 posicional, 17
 precisão, 23
 resistência, 87, 100, 105
 ROM, 12, 77–80, 126–133
 rotação, 158, 164
Row Address Strobe, veja RAS

S

seletor, 72
 semântica, 17
 semicondutor, 85
 tipo N, 87
 tipo P, 87
 silogismo, 37
 simplificação de expressões, 38–40
 sinal, 27, 42
 analógico, 27
 digital, 27, 28
 fraco, 125, 126, 138, 141
 intensidade, 90, 139
 restaurado, 125, 139
 síntese, veja VHDL, síntese
Solid State Disk, veja SSD
 soma, veja somador
 soma de produtos, 45, 51, 126
 completa, 45
 somador, 145, 152–153
 adiantamento de vai-um, 165
 cadeia, 152
 completo, 104, 152
 overflow, 163
 parcial, 103
 seleção de vai-um, 170–172
 teste, 178
 somatório, 33
 spice, 28
 SRAM, 138–139
 SSD, 14
status, 162
 subtração, 154
 superfície equipotencial, 96, 110

T

tabela verdade, 33–35, 45
 tamanho, veja $|N|$

- tempo,
 de contaminação, 115, 118–121
 de propagação, 57–58, 61, 64–65, 73, 77, 84, 102,
 104, 109, 115–117
temporização, 104–126
tensão, 105
Teorema,
 Absorção, 49
 DeMorgan, 32, 41, 48, 54, 60, 61, 94, 98
 Dualidade, 99
 Simplificação, 49
terceiro estado, 140–141
testbench, veja VHDL, *testbench*
teste,
 cobertura, 179
 de corretude, 178
teto, veja [r]
three-state, veja terceiro estado
tipo,
 de sinal, 42
 função, 29
Tipo I, veja formato
Tipo J, veja formato
Tipo R, veja formato
transferência entre registradores, veja RTL
transistor, 87–91, 95–96
 CMOS, 95
 corte, 113
 gate, 88
 saturação, 113
 sinal fraco, 90
 tipo N, 88
 tipo P, 89
Transistor-Transistor Logic, veja TTL
transitório, 121–123
transmission gate, veja porta de transmissão
TTL,
 74148, 75
tupla, veja $\langle \rangle$
 elemento, 32
 largura, 43
- U**
- ULA, 160–164, 180
 status, 162
Unidade de Lógica e Aritmética, veja ULA
- V**
- valor da função, 30
VCC, 93
vetor de bits, veja $\langle \rangle$, 32
 largura, 43
VHDL,
 design unit, veja VHDL, unidade de projeto
 std_logic, 140
 tipos, 42
- W**
- Watt, 112
write back, veja resultado
- X**
- xor, veja $\oplus$
- Z**
- Z, linguagem, 27

Capítulo 7

Básculos e *Flip-Flops*

We will perform in measure, time and place.

William Shakespeare, Macbeth, V/6.

Até aqui vimos circuitos que são ditos *combinacionais* porque os valores nos sinais em suas saídas resultam de combinações dos valores nos sinais de suas entradas. Para um conjunto de entradas $\mathcal{E}$, as saídas do circuito $\mathcal{S}$ serão *sempre* $\mathcal{S} = \mathcal{F}(\mathcal{E})$, com valores determinados pelas funções lógicas $\mathcal{F}$ que definem o comportamento de cada membro de $\mathcal{S}$. Circuitos combinacionais somente implementam funções simples, tais como tabelas verdade.

Na Seção 4.3.4, e com mais detalhes na Seção 5.6.2, vimos um circuito simples que tem a capacidade de memorizar um bit e portanto pode permanecer no estado 1 ou no estado 0. Circuitos com memória são ditos *sequenciais* porque o comportamento das saídas depende tanto dos valores nas entradas quanto da *sequência* em que os valores foram armazenados na memória. Circuitos sequenciais são um tanto mais úteis porque estes nos permitem implementar algoritmos cujos comportamentos dependem das sequências de valores apresentados as suas entradas.

Neste capítulo¹ retornamos ao tema *memória*, estudando dois circuitos com memória, os básculos e os *flip-flops*. Fazendo uso de circuitos combinacionais e de circuitos de memória podemos implementar desde máquinas simples como sinaleiros de trânsito até o processador em que o autor digita este texto.

Iniciamos, na Seção 7.3, com circuitos que, independentemente do que ocorre em suas entradas, mantêm o valor de um bit enquanto a fonte de alimentação fornecer energia. A Seção 7.4 introduz os *básculos*², que são circuitos que podem capturar um bit em sua entrada e memorizá-lo, define dois novos parâmetros para o comportamento temporal de circuitos com memória, e com base neles, define uma abstração para *tempo discreto*. A Seção 7.5 apresenta um circuito composto de dois básculos, chamado de *flip-flop*, cuja temporização é mais simples do que a de básculos. A Seção 7.6 define circuitos sequenciais síncronos (CSS) e o que se entende por “comportamento apropriado”.

¹© Roberto André Hexsel, 2012-2021. Versão de 1 de março de 2021.

²O termo em Inglês, *latch*, denota uma trava de porta que pode, ou estar aberta, ou estar fechada. Segundo Houaiss, a tradução correta é ‘basculador’, um cacófono assaz deselegante. O autor é um admirador do *über*-dicionarista mas prefere empregar, desavergonhadamente, a forma incorreta ‘básculo’. Atravessemos o Rubicão sem mais delongas.

7.1 Estado e Tempo Discretizado

Neste texto empregamos uma noção estreita e estrita de *estado*, que é distinta do conceito empregado nas Ciências Humanas. Vejamos a noção de estado num jogo de xadrez. Num dado instante, o *estado* de uma partida é a configuração das peças no tabuleiro e qual o jogador com direito a mover uma peça. A partida evolui – troca de estado – de acordo com as regras de movimentação para cada peça – um cavalo se move “em L”, um bispo na diagonal, *et cetera* – e da alternância entre os jogadores. O *estado atual* do jogo é completamente definido pelo par tabuleiro–jogador e as *mudanças de estado* são rigidamente definidas pelas regras de movimentação das peças. Uma partida evolui ao longo de uma sequência de estados, que se inicia com 16 pedras brancas e 16 pedras pretas no tabuleiro, e que termina com um único rei “de pé”. Cada um dos estados da sequência é atingido após a movimentação de uma única peça. Esta é uma ideia central para o que se segue: *os estados são discretos e representados univocamente por uma configuração de peças no tabuleiro*.

Encontramos *estados discretos* em jogos de baralho, na maioria dos jogos de tabuleiro e nas variáveis de um programa. Em todos esses exemplos, existe uma maneira de descrever a configuração atual do jogo ou programa, e é possível apontar qual a regra que deve ser aplicada para atingir uma nova configuração modificando-se a atual.

Recordemos a abstração de voltagem discretizada:

1. os níveis de tensão nos circuitos são separados em três faixas, uma chamada de 0 (zero), outra chamada de 1 (um), e valores na terceira faixa são ditos indefinidos. Os valores válidos definem o conjunto $\mathbb{B} = \{0, 1\}$;
2. emprega-se um conjunto fechado de operadores $\mathbb{B}^n \mapsto \mathbb{B}$;
3. um *dispositivo combinacional* é definido por uma função $\mathbb{B}^n \mapsto \mathbb{B}$ e pelo seu tempo de propagação;
4. um *circuito combinacional* é uma composição de dispositivos combinacionais, com entradas e saídas de tipo $\mathbb{B}^m$, não há curto-circuitos, entradas abertas e nem ciclos.

Essa abstração simplifica enormemente a especificação, projeto, implementação e verificação de circuitos *sem estado*.

Neste capítulo são definidos os *circuitos com estado* e uma abstração que *discretiza o tempo* de tal forma que as mudanças de estado ocorram somente em instantes determinados. Estes instantes são definidos por um *signal de relógio* que sincroniza as mudanças de estado em todos os componentes *com estado*. O comportamento dos circuitos no intervalo entre os instantes de sincronização é indefinido. A Figura 7.1 identifica as duas abstrações.

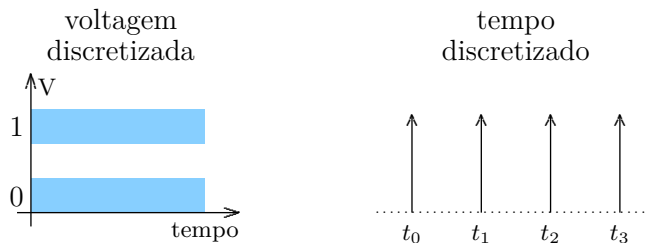


Figura 7.1: Abstrações: voltagem e tempo discretizados.

7.2 Circuitos Úteis que Violam a Abstração Combinacional

Vimos no Capítulo 5 que a tecnologia CMOS, na maior parte do tempo, se submete docilmente às imposturas da abstração para circuitos combinacionais. O comportamento temporal destes circuitos é deveras selvagem, especialmente no que concerne ao tempo de propagação dos sinais. Felizmente, estes podem ser amestrados para que possamos usar abstrações simples e úteis para a propagação dos sinais através de circuitos CMOS.

Pode ser interessante examinarmos um pequeno bestiário de circuitos cujo comportamento analógico viola nossa abstração combinacional e ainda assim, são circuitos excepcionalmente úteis.

Oscilador Livre Um inversor CMOS é um amplificador com um elevado fator de amplificação, e que inverte a fase da entrada para a saída – a saída fica 180° deslocada com relação à entrada. Se a saída for ligada à entrada, então obtemos um circuito que não é combinacional mas que é útil. Um oscilador composto de um inversor é mostrado na Figura 7.2.

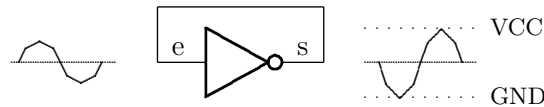


Figura 7.2: Oscilador com um inversor.

Enquanto a alimentação for mantida, esse circuito oscila com um período que é proporcional ao tempo de propagação do próprio inversor. A amplitude da saída é limitada pela tensão da fonte de alimentação. Este circuito é dito “oscilador livre” porque sua frequência normal de operação independe de qualquer dispositivo externo.

Se, ao invés de um único inversor, o laço contiver três inversores, o circuito oscila a uma frequência que depende do tempo de propagação dos três inversores, e obviamente tem período mais longo do que o oscilador com um único inversor. Tal circuito é mostrado na Figura 7.3.

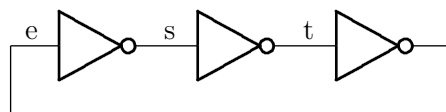


Figura 7.3: Oscilador com três inversores.

Oscilador com cristal de quartzo Estes circuitos não são muito confiáveis porque o período depende da soma dos tempos de propagação de cada um dos inversores e estes por sua vez, dependem da tensão e da temperatura de operação. Se ligarmos um cristal de quartzo ao inversor, este cortado com as dimensões apropriadas, então o circuito passa a oscilar a uma frequência que é determinada pelas dimensões do cristal. A frequência é estável com relação à temperatura, tensão de alimentação e envelhecimento. Por conta dessas características, osciladores a cristal são amplamente empregados como geradores de base de tempo, ou geradores de “relógio” em sistemas digitais. O dispositivo é indicado pelo seu apelido, XTAL, na Figura 7.4. O amplificador é necessário para gerar uma versão “quadrada”, ou digital, da forma de onda analógica que se estabelece no laço *e-s*.

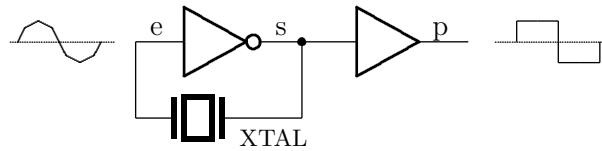


Figura 7.4: Oscilador com um inversor.

7.3 Circuitos com Memória

O tempo de contaminação das portas lógicas é, em essência, uma forma de memória: enquanto as alterações provocadas pelos novos valores nas entradas de um circuito não se propagarem através de todos os nós internos, sua saída ‘memoriza’ o valor anterior.

O circuito da Figura 7.5 mantém um valor estável e definido no sinal a' enquanto a fonte de alimentação estiver ligada. Esse comportamento decorre do tempo de contaminação dos inversores e da realimentação da saída de um inversor para a entrada do outro. O tempo de contaminação finito dos inversores possibilita que os níveis lógicos nos dois nós do laço, a e b , estabilizem, e que um nível lógico constante se mantenha nas saídas dos inversores.

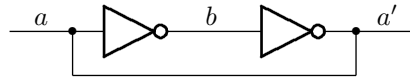


Figura 7.5: Circuito com memória.

Esse circuito tem três modos de operação. No primeiro, quando a fica em 0 durante um intervalo maior que $2 \times T_I$, para que os sinais se propaguem através dos dois inversores, b fica em 1, e a saída do segundo inversor a' fica em 0, que por sua vez realimenta o sinal a com 0, ‘capturando’ o bit em 0 no circuito. No segundo modo de operação, quando a fica em 1 por um intervalo maior do que $2 \times T_I$, a' manterá o valor 1.

Esses dois modos de operação são *estáveis* porque o laço de realimentação no caminho a - b - a' mantém os valores nas entradas dos inversores estáveis, se a entrada se mantiver estável por $2 \times T_I$.

O terceiro modo de operação é dito *metaestável*, e a metaestabilidade pode ocorrer se o nível em a ficar indeterminado por, ao menos, T_I . Neste caso, o sinal em b também fica indeterminado, e o valor armazenado no laço pode não ser nem 0 e nem 1, ou pode oscilar entre 0 e 1. Em qualquer destes casos, a abstração de sinais como bits é violada.

Não há como determinar a duração do intervalo metaestável: (i) os inversores podem regenerar os sinais imediatamente, ou após o decurso de um intervalo arbitrariamente longo; (ii) ruído elétrico na vizinhança pode interferir positiva ou negativamente para a regeneração do sinal; e (iii) as tensões de saída dos inversores podem oscilar durante um intervalo arbitrariamente longo em níveis abaixo dos limiares para 0 ou para 1.

Não existe um parâmetro similar a T_I para definir a duração da metaestabilidade; o que se pode fazer é tentar evitá-la, ao impedir que a condição se estabeleça. Veremos na Seção 7.5 como evitar que transições nos sinais de controle do circuito de memória o levem à metaestabilidade.

Da forma como está desenhado na Figura 7.5, pode não ser fácil alterar o valor memorizado. A Figura 7.6 mostra o circuito de memória com duas chaves que permitem alterar o valor armazenado. Na esquerda, o laço de realimentação está fechado e assim mantém o valor do

bit; no lado direito, o laço está aberto e um novo valor pode ser propagado de a para a' através dos dois inversores. A posição das duas chaves deve ser trocada simultaneamente para que o circuito se comporte como o desejado.

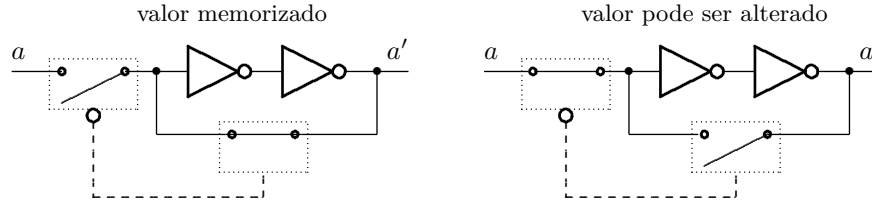


Figura 7.6: Circuito com memória programável.

Com circuitos CMOS, as chaves são implementadas com portas de transmissão ou transistores de passagem, como vimos na Seção 5.6. Circuitos com memória podem ser implementados com portas lógicas ao invés de inversores e portas de transmissão. Estes circuitos são chamados de *bascúlos* porque sua saída normalmente está em um de dois estados possíveis. A Figura 7.7 mostra duas formas de desenhar o circuito de um bascúlo que é implementado com portas *nore* chamado de *bascúlo SR*. Normalmente as entradas r e s estão em 0. Se a entrada s (*set*) mudar para 1, a saída q muda para 1. Se a entrada r (*reset*) mudar para 1, a saída q muda para 0. Se as duas entradas mudarem de 1 para 0 simultaneamente, o bascúlo pode entrar em metaestabilidade. Bascúlos podem ser implementados com um par de portas *nand*, e nesse caso o nível ativo das entradas *set* e *reset* é 0.

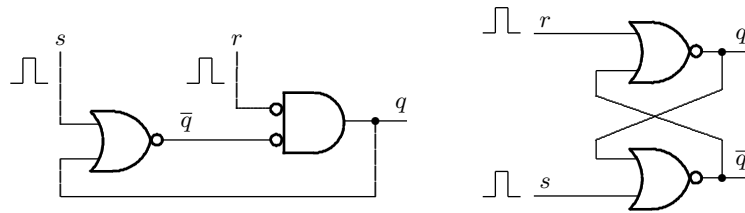


Figura 7.7: Bascúlo SR com portas *nor*.

O diagrama no lado esquerdo da Figura 7.7 mostra o bascúlo de forma similar ao laço com dois inversores, e o do lado direito mostra a forma convencional de representação. Os pulsos (0-1-0) nas entradas s e r indicam os níveis lógicos necessários para mudar o estado do bascúlo: um pulso em s coloca a saída q em 1 (*set*), e um pulso em r coloca a saída q em 0 (*reset*).

No que se segue, empregaremos bascúlos ligeiramente mais complexos do que aqueles com um par de portas lógicas, embora sejam funcionalmente equivalentes. Tal escolha favorece o aprendizado sobre o realismo na implementação dos circuitos³.

7.4 Bascúlos

Como vimos na Seção 5.4, o comportamento dinâmico de um circuito combinacional pode ser distinto daquele determinado por sua especificação lógica, por causa da contaminação dos

³Grande parte do material deste capítulo é baseado em conteúdos da disciplina 6.004 – *Computation Structures* do MIT, versão de 2013, que é, com uma larga vantagem, a melhor explanação sobre bascúlos de que o autor tem conhecimento. Material acessado em setembro de 2015.

sinais em sua saída. Para minimizar as perturbações causadas por pulsos espúrios, usaremos um multiplexador bem comportado para implementar circuitos que podem armazenar um bit, tal como aquele mostrado na Figura 7.8.

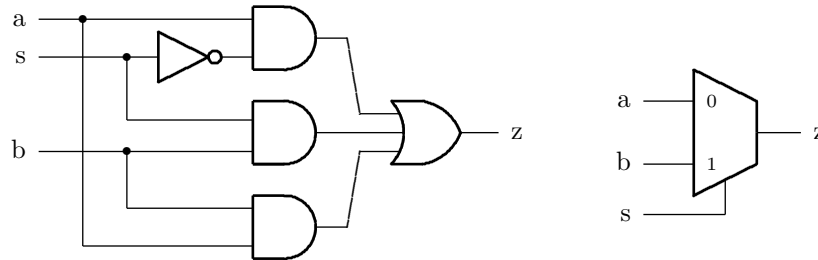


Figura 7.8: Multiplexador com saída bem comportada.

Um *básculo* (*latch*) é um circuito que armazena um bit e o valor do bit pode ser facilmente alterado ativando-se um sinal de controle, que permite carregar o básico com um novo valor, ou manter o valor carregado previamente.

O básico mostrado na Figura 7.9 tem três terminais: uma entrada D (*datum*), uma entrada Q' que é também a saída Q (*quiescent*), e uma entrada de ‘gatilho’ G . A entrada D determina o valor por armazenar quando $G = 1$. Quando $G = 0$, o básico mantém o valor armazenado no terminal Q . O terminal de controle é chamado de ‘gatilho’ pois a transição de 1 para 0 em G é o que permite capturar um novo valor para armazená-lo. Note que o caminho com realimentação $Q-Q'$ é similar àquele do circuito com os dois inversores. Observe na Figura 7.8 que a saída do or mantém o sinal na entrada a do and, cuja saída é uma das entradas do or.

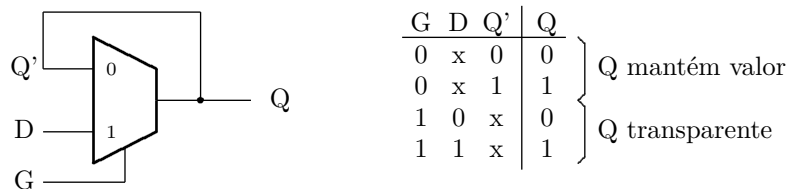


Figura 7.9: Básculo implementado com um multiplexador.

O multiplexador implementado com dois implicativos primários – sem o implicativo $a \wedge b$ – produz um pulso espúrio em sua saída quando o sinal de controle passa de 1 para 0. Este pulso contaminaria o caminho de realimentação, e com alta probabilidade alteraria o bit armazenado em $Q-Q'$. Usamos o multiplexador bem comportado para evitar a contaminação da saída.

Como no caso dos dois inversores, no básico é necessário usar um circuito que viola a restrição aos ciclos em circuitos combinacionais. A violação é necessária para obtermos a função de memória. Como no caso da abstração que faz uso de elementos de $\mathbb{B}$ para representar sinais elétricos contínuos, aqui usaremos uma nova abstração para discretizar o tempo contínuo.

Nossa abstração para *tempo discreto* é uma sequência discreta de instantes, determinada por um sinal periódico que alterna entre 1 e 0. Aqui é necessária alguma paciência porque convém estudarmos o comportamento dinâmico do básico com mais atenção antes que possamos definir “tempo discreto”.

A especificação temporal para o básico é determinada pelo tempo de propagação T_M do multiplexador. O diagrama de tempos na Figura 7.10 mostra o comportamento dinâmico do básico. Quando a entrada $G = 1$, o básico está em modo transparente e a saída é uma cópia

da entrada, a menos do atraso causado pelo tempo de propagação. Quando $G = 0$, a saída do basculo mantém seu valor enquanto G for 0.

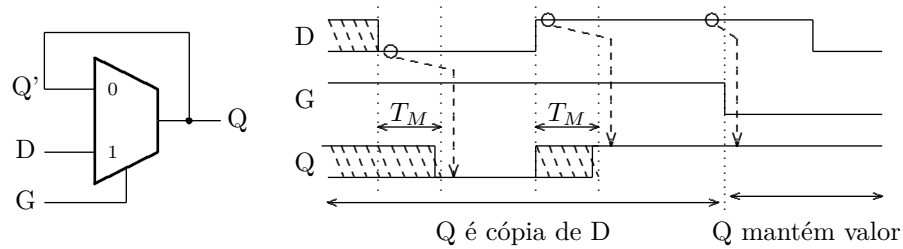


Figura 7.10: Comportamento dinâmico do basculo.

Considere as três combinações de entradas que definem o comportamento dinâmico do basculo, mostradas na Figura 7.11. Aqui, t é um intervalo arbitrário, e $\beta \in \mathbb{B}$.

- (1) *Cópia de um novo valor.* Se $G = 1$, então Q é uma cópia de D . A entrada D é selecionada e se o gatilho G e a entrada $D = \beta$ permanecem estáveis por $t \geq T_M$ então Q assume o valor β independentemente do que ocorra com Q' .
- (2) *Captura.* Alteração em G sem contaminar a saída Q . Se as entradas $D = Q' = \beta$ durante $t \geq T_M$, então $Q = \beta$ independentemente da alteração em G porque a escolha entre dois valores iguais nas entradas não altera a saída.
- (3) *Q mantém seu valor enquanto $G = 0$.* Se $G = 0$ e $Q' = \beta$ por $t \geq T_M$, então $Q = \beta$ independentemente do valor em D porque Q' foi selecionado como a entrada do multiplexador por $G = 0$.

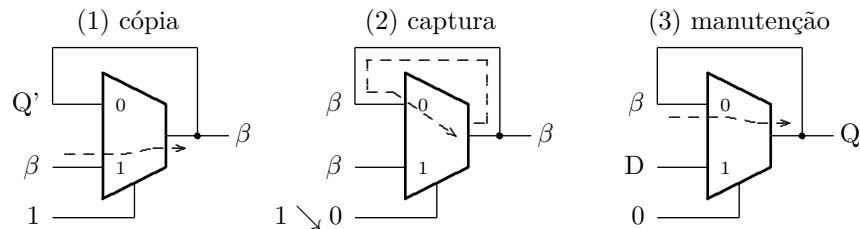


Figura 7.11: Combinações de entradas do basculo.

Antes da mudança de 1 para 0 em G , a entrada D deve ficar estável por $2 \times T_M$ para garantir que β seja capturado de forma confiável. O primeiro T_M garante a estabilização dos nós no caminho $D \leadsto Q$, enquanto que o segundo T_M garante a estabilização no caminho $Q' \leadsto Q$. Estes caminhos estão mostrados como linhas tracejadas na Figura 7.12.

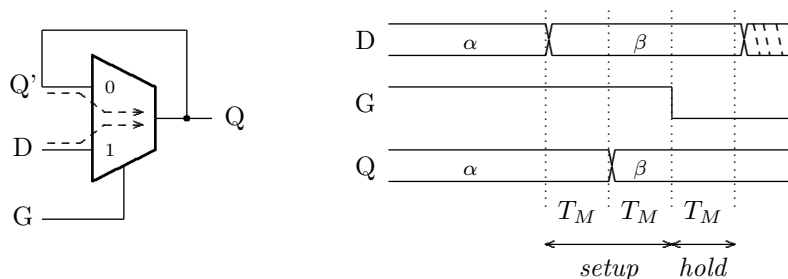


Figura 7.12: Comportamento dinâmico do basculo – transições em G .

A entrada D deve permanecer estável por $2 \times T_M$ antes de uma alteração em G de 1 para 0. Este intervalo é necessário para garantir a captura de um novo valor antes da borda descendente no gatilho. O intervalo de $2 \times T_M$ antes da borda em G é chamado de *tempo de estabilização das entradas* (*setup time*), e é denotado por T_{su} .

A entrada D deve permanecer estável por T_M após a borda de descida em G para garantir que o valor capturado no laço de realimentação não seja contaminado por uma mudança em D . O intervalo T_M após a borda no gatilho é o *tempo de manutenção das entradas* (*hold time*), e é denotado por T_h .

Se os tempos de estabilização, ou de manutenção, do basculo são violados, seu comportamento pode tornar-se metaestável e a saída do basculo pode manter um valor indefinido por um tempo arbitrariamente longo.

7.4.1 Abstração do Tempo Discretizado

Para que possamos discretizar o tempo, e assim abstrair a complexidade do comportamento temporal dos basculos, é necessário definirmos limites para os intervalos de tempo de estabilidade dos sinais imediatamente antes e imediatamente depois das bordas descendentes no sinal de gatilho. Estes limites são usados como a especificação temporal do comportamento de basculos, em adição aos tempos de propagação e de contaminação:

- o tempo de estabilização das entradas (setup time) é o intervalo em que as entradas estão válidas e estáveis antes da transição no sinal de gatilho; e*
- o tempo de manutenção das entradas (hold time) é o intervalo em que as entradas estão válidas e estáveis após a transição no sinal de gatilho.*

Nos resta um pequeno porém incômodo problema: nossa definição para circuitos combinacionais proíbe ciclos, tais como os caminhos de realimentação nos basculos. O problema decorre de que, em circuitos com ciclos, os sinais percorrem o circuito combinacional e computam um novo valor, e por causa do ciclo, o novo valor percorre o circuito e computa mais um novo valor, *ad aeternum*. Dependendo da complexidade da parte combinacional do circuito, os sinais se propagam a taxas incontroláveis, e não há como prever se e quando estabilizarão. A parte ‘pequena’ do problema é que a sua solução é relativamente simples: empregam-se dois basculos para quebrar os ciclos.

7.5 Flip-Flops

Basculos possuem características temporais que tornam os projetos complexos porque sua temporização tem uma certa tendência à metaestabilidade. A combinação de dois basculos elimina o problema causado pelos ciclos combinacionais e ainda permite armazenar bits com uma especificação temporal que é relativamente simples de satisfazer e implementar. O circuito com dois basculos é chamado de *flip-flop*.

Os instantes em que os dados são capturados pelos *flip-flops* são determinados pelas bordas ascendentes do *sinal de relógio*, que é um sinal periódico que alterna entre 0 e 1. Este sinal é, tipicamente, uma *onda quadrada*, na qual os intervalos em 0 têm a mesma duração que os intervalos em 1. No que se segue, o sinal de relógio é chamado de *clock* e frequentemente abreviado como *clk*.

Se usarmos dois bástulos ligados em série, com seus gatilhos em níveis distintos do sinal de relógio, temos o que se chama de *flip-flop mestre-escravo*. A Figura 7.13 mostra o *flip-flop* e um diagrama de tempo que relaciona entradas com a saída Q . Note que as entradas de gatilho tem efeitos complementares no mestre e no escravo: quando $clock = 0$, o mestre fica transparente e o escravo mantém seu valor.

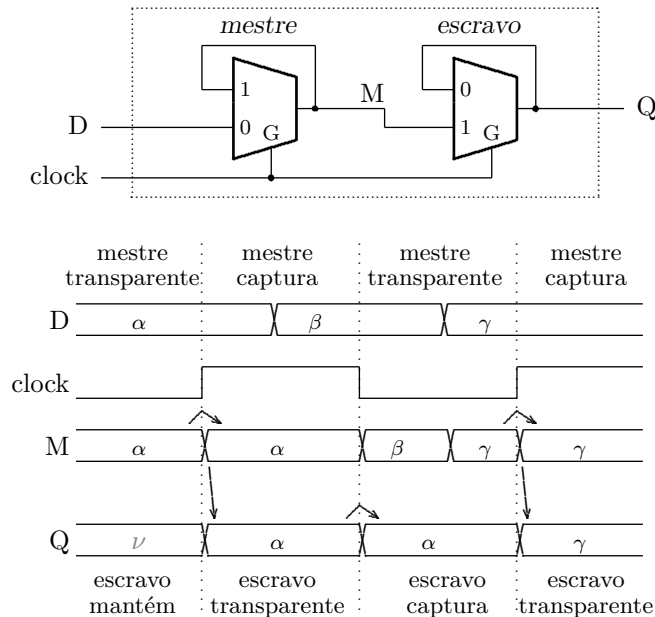


Figura 7.13: *Flip-flop* mestre-escravo.

No *borda ascendente*, na qual $clock$ muda de 0 para 1, o mestre captura a entrada $D = \alpha$, enquanto que o escravo fica ‘transparente’ porque sua saída segue $M = \alpha$ e portanto $Q = \alpha$.

No *borda descendente*, na qual $clock$ muda de 1 para 0, o escravo captura a entrada $M = \alpha$, e assim a saída se mantém inalterada, em $Q = \alpha$. Neste semiciclo o mestre está transparente, e sua saída segue $D = \beta$ e $D = \gamma$.

No primeiro semiciclo, o mestre captura a entrada D , e o escravo, que está transparente, transfere sua entrada para a saída Q . No segundo semiciclo, o escravo captura a saída do mestre e a transfere para sua saída Q , que permanece inalterada. Neste semiciclo o mestre está transparente e transfere a entrada que será capturada pelo escravo na próxima borda ascendente. O comportamento do *flip-flop* mestre-escravo é o desejado: a cada borda ascendente do relógio, a entrada é capturada e permanece estável até a próxima borda ascendente.

A Figura 7.14 mostra os símbolos que representam um bástulo e um *flip-flop*. A diferença entre os dois símbolos é o triângulo no *flip-flop*, que representa a entrada de relógio que é sensível à borda ascendente, enquanto que a entrada *gate* do bástulo é sensível ao nível do sinal de sincronismo.

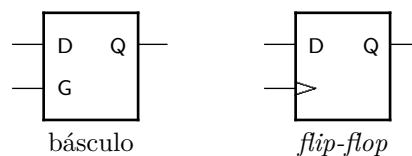


Figura 7.14: Símbolos para bástulos e *flip-flops*.

A diferença de comportamento entre basculos e *flip-flops* é explicitada no diagrama de tempo da Figura 7.15. No semiciclo em que o sinal de sincronismo é 1, o basculo fica transparente e a saída Q segue a entrada D , e isso é indicado pela faixa hachurada no sinal de entrada. Antes do início da observação dos sinais, no ciclo 1, os valores em B e em FF são desconhecidos. No ciclo 1, basculo e *flip-flop* têm o mesmo comportamento porque a entrada permanece estável ao longo do primeiro semiciclo.

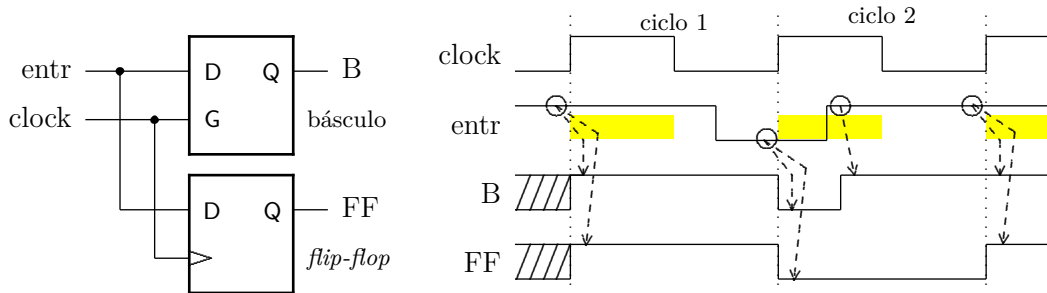


Figura 7.15: Comportamento de basculos e *flip-flops*.

No ciclo 2, a entrada muda no primeiro semiciclo e a saída do basculo segue a entrada. O valor na entrada do *flip-flop* é capturado na borda do sinal de relógio e este valor não se altera até a próxima borda.

Exemplo 7.1 Vejamos uma aplicação simples porém importante de *flip-flops*. Suponha que a saída de um circuito combinacional seja ‘ruidosa’, porque contenha muitos pulsos espúrios, o que pode prejudicar a operação do(s) circuito(s) que fazem uso deste sinal. O circuito da Figura 7.16 mostra uma forma de filtrar os pulsos indesejados: o sinal ruidoso R é ligado à entrada de um *flip-flop* ($Q1$), e a saída deste é ligada à entrada de um segundo *flip-flop* ($Q2$).

Antes do ciclo 1, o estado de $Q1$ (sinal X) é indeterminado, e antes do ciclo 2 o estado de $Q2$ (Y) ainda é indeterminado. No ciclo 1 o valor de R é capturado por $Q1$ e esse valor é copiado para $Q2$ no ciclo 2. Os pulsos em R não se refletem na saída de $Q1$, e a entrada de $Q2$ recebe uma versão ‘limpa’, porém atrasada, do sinal R .

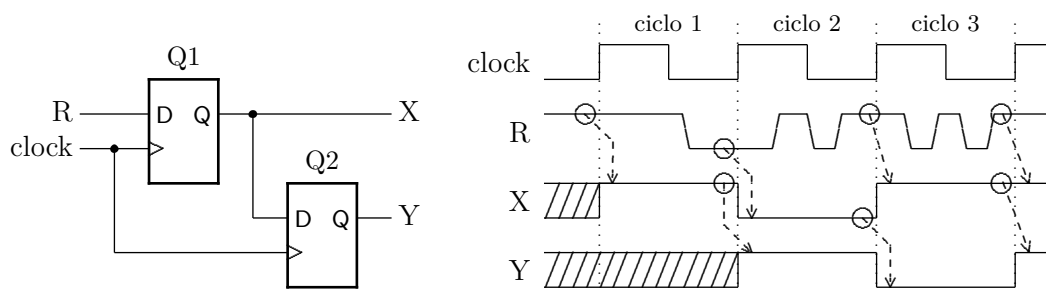


Figura 7.16: Filtro de ruído com dois *flip-flops*.

O valor de Y está dois períodos do relógio atrasado em relação a R , e este é um ponto importante: ao final do ciclo 1, o nível 0 em R é capturado por $Q1$, e $X=0$ ao longo do ciclo 2. Somente *após* a borda do ciclo 3 é que $Q2$ passa ao nível 0 que R exibira no ciclo 1. ◀

Exemplo 7.2 O circuito do Ex. 7.1 tem mais uma utilidade, que é a de eliminar o comportamento metaestável que pode ocorrer em *flip-flops*. O *flip-flop* Q1 é suficiente para filtrar o ruído no sinal R, embora Q1 possa incorrer em metaestabilidade caso uma transição em R ocorra suficientemente perto da borda do relógio. Isso é mostrado no diagrama de tempos da Figura 7.17, com a transição no sinal R junto à borda do ciclo 2, o que faz com que a saída de Q1 fique metaestável durante quase todo aquele ciclo.

A probabilidade de que a saída de Q1 ainda esteja metaestável na próxima borda é pequena. O *flip-flop* Q2 reduz significativamente a probabilidade de que um sinal metaestável se propague a partir do sinal Y. Nesse caso, ambos Q1 e Q2 teriam que permanecer metaestáveis durante dois ciclos completos, o que é possível, mas altamente improvável. Comportamento metaestável pode ser eliminado se nossos projetos aderem rigidamente à abstração de tempo discretizado. ◁

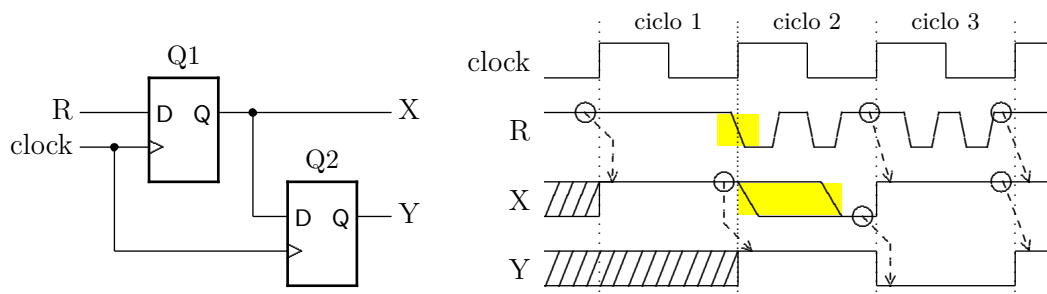


Figura 7.17: Defesa contra metaestabilidade com dois *flip-flops*.

Exemplo 7.3 O circuito da Figura 7.18 mostra um *flip-flop* cuja saída se inverte ($1 \rightarrow 0$ ou $0 \rightarrow 1$) sempre que sua entrada for 1. Pode não ser óbvio, mas este circuito conta, em módulo-2, o número de vezes em que o sinal de entrada trocou de nível. Voltamos a este assunto na Seção 8.2.1.

Como mostra a tabela verdade do *xor*, se $T = 0$ então $R = S$, e mais ainda, se $T = 1$ então $R = \bar{S}$. A porta *xor* se comporta como um inversor controlado pelo sinal T . Suponha que, inicialmente, $S = 1$. Com $T = 0$ a saída do *flip-flop* é uma cópia da sua entrada e assim, no ciclo 2 temos $S = 1$. Quando $T = 1$, R se inverte, e no ciclo 3 S muda para 0. Enquanto $T = 1$, na próxima borda do relógio, o valor de S muda. Este circuito é chamado de *flip-flop toggle* porque sua saída se inverte sempre que a entrada de controle for 1. ◁

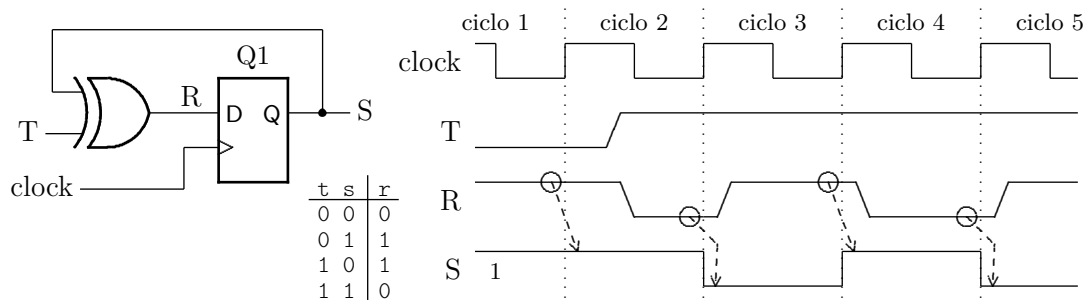


Figura 7.18: *Flip-flop* que troca de estado, ao invés de mantê-lo.

Exemplo 7.4 O circuito da Figura 7.19 é um *gerador de relógio*, que emprega um cristal que quartzo como base de tempo, tal como o da Figura 7.4. O sinal periódico é ligado ao terminal de relógio do *flip-flop*, que por sua vez gera uma onda quadrada – a duração do intervalo em 1 é igual à duração do intervalo em 0 – cuja frequência é a metade da frequência de oscilação do cristal. O inversor tem o mesmo efeito que a porta *xor* da Figura 7.18 e provoca trocas periódicas na saída Q. ◁

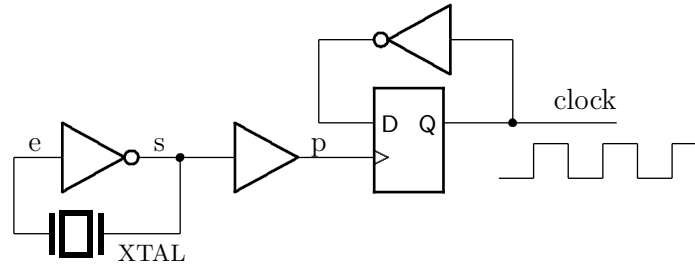


Figura 7.19: Gerador de relógio.

7.5.1 *Flip-flops* e Tempo Discretizado

Para garantir que a abstração de tempo discretizado seja válida, e que o comportamento dinâmico do *flip-flop* é o desejado, sua especificação temporal deve ser respeitada. O diagrama de tempo com as restrições na temporização da entrada de dados, e a especificação do comportamento da saída é mostrada na Figura 7.20.

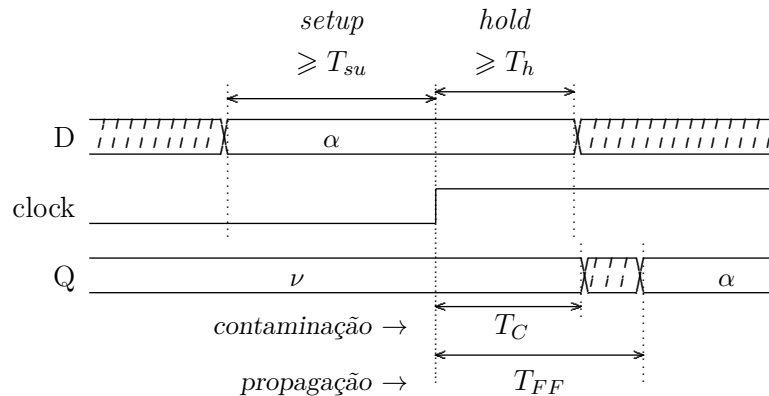


Figura 7.20: Temporização do *flip-flop*.

Para *flip-flops*, os tempos de estabilização, de manutenção, de propagação e de contaminação são *sempre* definidos com relação à borda de subida do relógio. Frequentemente são empregadas as expressões *clock-to-Q* ou *clock-to-output* para explicitar que o tempo de propagação do *flip-flop* é contado a partir da borda do sinal de relógio.

A entrada *D* deve permanecer estável *antes* da borda do relógio, por um intervalo maior do que o tempo de estabilização T_{su} . A entrada deve permanecer estável, *após* a borda do relógio, por um intervalo maior do que o tempo de manutenção T_h . Se as restrições quanto à entrada forem respeitadas, então a saída *Q* do *flip-flop* mantém seu valor durante o tempo de contaminação $T_{C,F}$ após a borda do relógio, e a saída se torna definida e válida após o tempo de propagação T_{FF} .

O tempo de propagação T_{FF} é a soma do tempo de propagação dos dois bascúlos – após a borda do relógio, um novo valor é capturado pelo mestre e se propaga através do escravo até a saída $T_{FF} = 2 \times T_B$. O tempo de contaminação T_C do *flip-flop* é determinado pelo bascúlo escravo – ao ficar transparente, o escravo mantém a saída do *flip-flop* como estava durante $T_{C,E}$. Os tempos de estabilização T_{su} e de manutenção T_h são determinados pelo bascúlo mestre – que mantém a entrada do *flip-flop* após a borda de subida do relógio.

O leitor atento deve ter percebido um possível problema com o tempo de manutenção do escravo. Se o tempo de contaminação $T_{C,M}$ do mestre for menor do que o tempo de manutenção T_h do escravo, então o escravo pode capturar um valor incorreto na borda de descida do relógio. Isso pode ser contornado pelo acréscimo de um *buffer* entre a saída do mestre e a entrada do escravo, para alongar o $T_{C,M}$ do mestre.

7.5.2 Tipos de *Flip-Flops*

Além do *flip-flop* tipo D (D de ‘direto’ ou ‘data’), outros *flip-flops* são úteis. A Figura 7.21 mostra os símbolos para os *flip-flops* tipo D, tipo T (de ‘toggle’) e JK. Estes *flip-flops* podem ser implementados com um par de saídas complementares, o que é indicado pela negação na segunda saída, que é chamada de $\overline{Q}$.

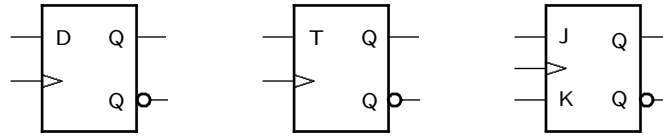


Figura 7.21: *Flip-flops* tipo D, T e JK.

O comportamento desses *flip-flops* (FFs) é definido pelas suas *equações características* mostradas na Equação 7.1, ou alternativamente, pelas suas tabelas de excitação, na Tabela 7.1. Nesta tabela, Q^+ é o estado do *flip-flop* após a borda do relógio. Na Equação 7.1 o símbolo $\leftarrow$ denota a atribuição do novo estado ao FF. A equação característica do FF T pode ser reescrita como $Q \leftarrow T \oplus Q$.

$$\begin{aligned}
 D, T, J, K, Q &: \mathbb{B} \\
 \text{FF D: } Q &\leftarrow D \\
 \text{FF T: } Q &\leftarrow T\overline{Q} \vee \overline{T}Q \\
 \text{FF JK: } Q &\leftarrow J\overline{Q} \vee \overline{K}Q
 \end{aligned} \tag{7.1}$$

Tabela 7.1: Tabela de excitação dos *flip-flops* JK, T e D.

Q	Q^+	JK	T	D
0	0	0X	0	0
0	1	1X	1	1
1	0	X1	1	0
1	1	X0	0	1

Existem situações nas quais é necessário inicializar os *flip-flops* com um estado definido. Por exemplo, frequentemente se deseja inicializar uma contagem em zero, e para isso todos os

flip-flops do circuito devem iniciar no estado $Q = 0$. O sinal de inicialização é tipicamente chamado de *reset*, e quando este sinal está ativo, é atribuído um estado definido aos *flip-flops*.

A equação característica do FF D com sinais de inicialização *set* ($Q \leftarrow 1$) e *reset* ($Q \leftarrow 0$) é mostrada na Equação 7.2. O estado do *flip-flop* é indeterminado quando ambos *set* e *reset* são ativados ao mesmo tempo.

$$\text{FF D: } \begin{cases} \text{set} \wedge \text{reset} & \Rightarrow Q \leftarrow D \\ \overline{\text{set}} & \Rightarrow Q \leftarrow 1 \\ \overline{\text{reset}} & \Rightarrow Q \leftarrow 0 \end{cases} \quad (7.2)$$

O símbolo para o *flip-flop* D ‘completo’ é mostrado na Figura 7.22. Quando ativo, o sinal *set* coloca imediatamente o FF em 1, e o sinal *reset* força o estado em 0. Note que os sinais *set* e *reset* são, geralmente, ativos em 0, o que é indicado pelo círculo de negação junto à entrada. Esses dois sinais são assíncronos e atuam independentemente do sinal de relógio.

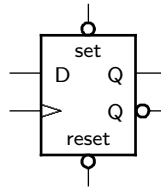


Figura 7.22: *Flip-flop* tipo D com entradas *set* e *reset*.

Os *flip-flops* tipo T e JK também podem ser implementados com as entradas assíncronas *set* e *reset*, que têm os mesmos efeitos que no FF D. Quando essas entradas são desnecessárias, elas são omitidas dos diagramas, embora não devam ficar desligadas em nenhuma implementação.

Exemplo 7.5 Quando se emprega dispositivos com lógica programável (*Field Programmable Gate Arrays*, ou FPGA), o compilador de VHDL impõe uma série de restrições aos tipos de construções que podem ser empregadas nos modelos dos circuitos. Por exemplo, o compilador de um dos grandes fabricantes de FPGAs proíbe a utilização de bástulos porque estes tornam a temporização do circuito imprevisível por causa dos ciclos.

Contudo, bástulos são úteis e necessários em muitas aplicações. O circuito da Figura 7.23 contém um *flip-flop*, e seu comportamento imita aquele de um bástulo SR. Um pulso em 1 na entrada *set* coloca a saída Q em 1. Um pulso em 0, na entrada *clr* (*clear*), coloca a saída em 0.

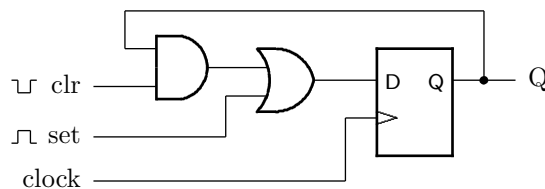


Figura 7.23: Circuito com *flip-flop* que se comporta como bástulo SR.

O comportamento do circuito da Figura 7.23 é mostrado no diagrama de tempo da Figura 7.24, para as quatro combinações de *clr* e *set*. Existe alguma combinação das entradas que torna o circuito metaestável? Considere que todos os parâmetros de temporização sejam atendidos com folga.

Se as entradas forem $clr=0$ e $set=1$, Q ficará num “estado ‘preferencial’”, quer dizer, uma das entradas tem mais prioridade que a outra. Altere o circuito para inverter a prioridade das entradas. O novo circuito é imune à metaestabilidade? ◀

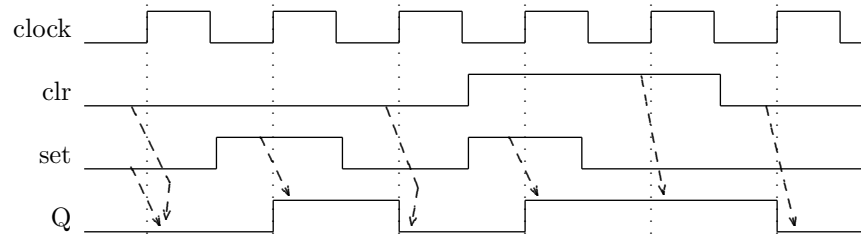


Figura 7.24: Temporização do circuito que se comporta como básico SR.

7.5.3 Registradores

Um *registrador* é um circuito com um certo número de *flip-flops* que compartilham o sinal de relógio, como mostrado na Figura 7.25. Tipicamente todos os *flip-flops* são atualizados simultaneamente e o registrador armazena o equivalente a uma variável numa linguagem de programação de alto nível. Apesar da sua simplicidade, esse componente é usado extensivamente no projeto de sistemas digitais, e em breve veremos vários exemplos de seu uso.

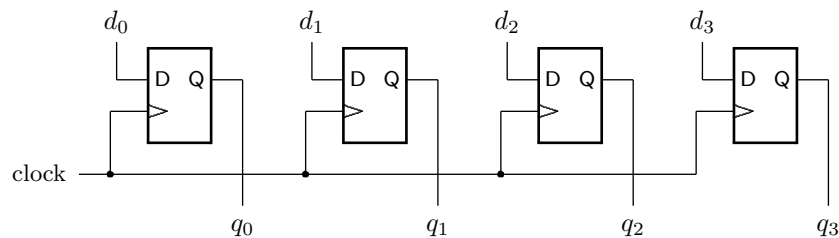


Figura 7.25: Registrador simples de quatro bits.

Um *registrador simples*, que é aquele da Figura 7.25, é atualizado a cada borda ascendente do relógio. Há aplicações nas quais deseja-se atualizar o conteúdo do registrador somente em determinados instantes, mantendo seu conteúdo inalterado no restante do tempo. O registrador da Figura 7.26 permite que um novo valor seja carregado na borda do relógio em que o sinal *carga* esteja ativo ($carga=1$), enquanto que seu valor é preservado quando *carga* está inativo. Este registrador é chamado *registrador com carga paralela*.

Os diagramas das Figuras 7.25 e 7.26 são demasiado detalhados para uso em desenhos/diagramas de circuitos complexos. As versões simplificadas dos símbolos para registradores com n bits de largura – n *flip-flops* – são mostradas na Figura 7.27.

Em geral, emprega-se um sinal de *reset* em circuitos complexos para inicializar todos os *flip-flops* e registradores para um estado conhecido. Um circuito dedicado ativa o *reset* por um intervalo suficiente para garantir que todos os registradores tenham sido inicializados. Depois disso, esse sinal permanece inativo até que o circuito seja desligado ou reinicializado.

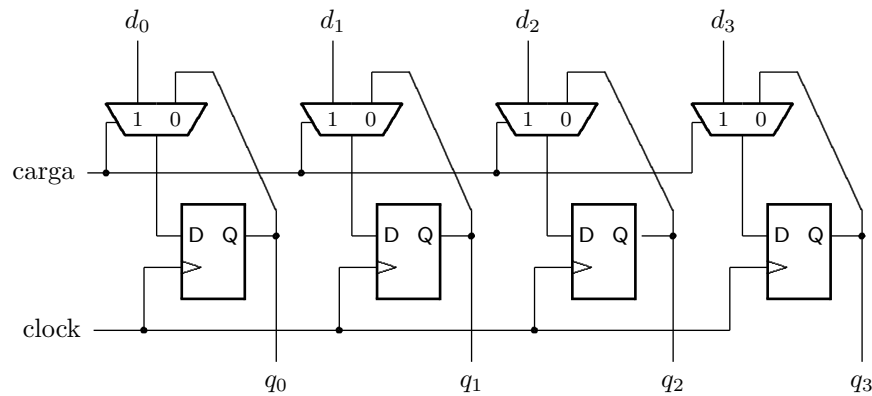
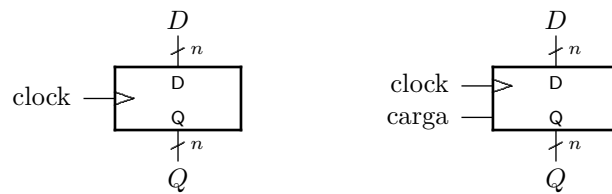


Figura 7.26: Registrador com carga paralela.

Figura 7.27: Símbolos para registradores de n bits.

7.6 Circuitos Sequenciais Síncronos

Agora que fomos apresentados a todos os componentes interessantes, podemos definir o que se entende por *circuito sequencial síncrono*⁴ (CSS). Esses são circuitos compostos por um componente com memória para armazenar seu *estado atual*, além de lógica combinacional que computa o *próximo estado* como uma função do estado atual e das entradas, bem como uma função para computar as *saídas* do circuito. O ‘síncrono’ indica que as mudanças de estado ocorrem sincronicamente ao sinal de relógio.

Um *circuito sequencial síncrono* é composto por:

1. um sinal de relógio que sincroniza a operação do CSS;
2. zero ou mais *entradas* digitais;
3. uma ou mais *saídas* digitais;
4. um *registrador de estado*;
5. uma *função de próximo estado*; e
6. uma *função de saída*.

A Figura 7.28 mostra um modelo para um circuito sequencial síncrono. O *registrador de estado* (RE) tem ϵ bits de largura e portanto o CSS pode ter até 2^ϵ estados. O CSS tem λ bits de entrada, e o circuito combinacional que implementa a *função de próximo estado* deve computar o próximo estado PE com a função f

$$PE = f(\text{estado atual}, \text{entradas}).$$

⁴A definição de *circuito sequencial síncrono* desta seção é baseada em conteúdos da disciplina 6.004 – *Computation Structures* do MIT, versão de 2013. Material acessado em setembro de 2015.

A saída S tem σ bits e é computada pela *função de saída* g

$$S = g(\text{estado atual}, \text{entradas}).$$

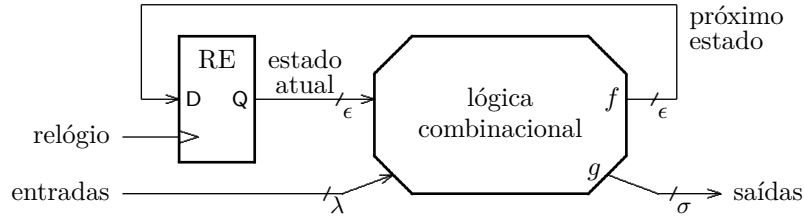


Figura 7.28: Circuito sequencial síncrono.

Se a máxima “tempo é dinheiro” pode ser considerada válida, então nos interessa computar a velocidade máxima de operação dos nossos CSSs. Esta é determinada pela frequência máxima do relógio na qual o circuito pode operar e produzir os resultados desejados.

7.6.1 Operação Apropriada – *Setup* e *Hold*

Neste contexto é desejável e conveniente introduzir o conceito de *operação apropriada*, em contraste ao conceito de *operação correta*. Um circuito opera corretamente se seu funcionamento satisfaz à sua especificação: se uma sequência de valores aceitáveis é apresentada nas suas entradas então o circuito produz uma sequência de valores nas suas saídas que são exatamente aqueles valores determinados pela sua especificação.

Um circuito opera apropriadamente se nenhuma das restrições impostas pela tecnologia de implementação é violada. Por exemplo, a fonte de alimentação mantém um fluxo de corrente estável enquanto opera na sua tensão nominal; as restrições de temperatura e umidade são atendidas. O que nos interessa aqui é verificar se as restrições de *setup* e *hold* dos *flip-flops* são atendidas pelo projeto do CSS.

Se, e somente se, as restrições de *setup* e *hold* são atendidas, então podemos fazer uso da abstração de *tempo discretizado*.

A *operação apropriada* de um CSS é definida como:

para cada evento de sincronização, todos os flip-flops controlados pelo mesmo sinal de relógio examinam suas entradas e determinam seus próximos estados. Para tanto, duas condições devem ser satisfeitas:

- (1) *as entradas ficam estáveis e determinadas antes do evento de sincronização;*
- e
- (2) *nenhum flip-flop pode mudar de estado mais do que uma vez em cada evento de sincronização.*

Se o circuito sequencial é uma máquina de estados implementada segundo o modelo da Figura 7.28, o registrador de estado pode ser encarado como se fosse dividido em dois, no qual as saídas – registrador fonte – são consideradas separadamente das entradas – registrador destino.

Setup ou Tempo de Estabilização

A primeira condição é satisfeita se o tempo de estabilização (*setup*) dos *flip-flops* é respeitado. O tempo de estabilização é uma especificação temporal definida em função de dois eventos consecutivos de sincronização, ou de duas bordas consecutivas do relógio.

O relacionamento entre as bordas do relógio e a especificação para *setup* é mostrada na Figura 7.29. Dois *flip-flops* são ditos *adjacentes* se estes compartilham o sinal de relógio e são interligados somente por lógica combinacional. A temporização é definida entre dois *flip-flops* adjacentes, sendo FF1 o *flip-flop* fonte do valor e FF2 o destino. No circuito da Figura 7.28, ‘fonte’ e ‘destino’ são a saída e a entrada do mesmo *flip-flop*.

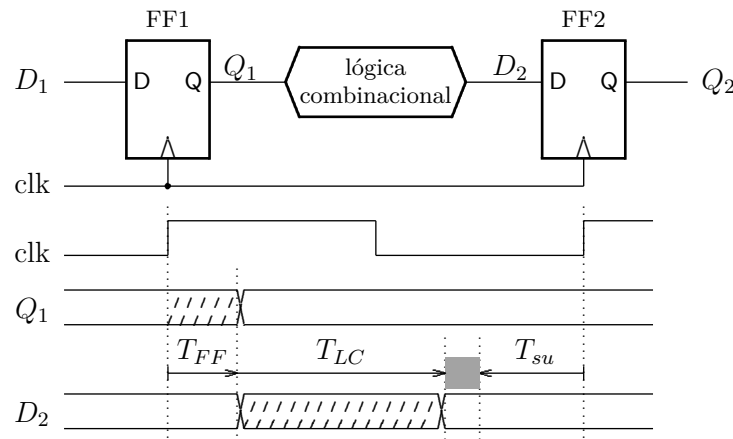


Figura 7.29: Operação apropriada: relação entre borda do relógio e *setup*.

A especificação de *setup* envolve duas bordas consecutivas do relógio: a soma dos tempos de propagação do *flip-flop* T_{FF} e do circuito combinacional T_{LC} , mais o *setup* (T_{su}), devem ser menores do que o intervalo entre duas bordas, que é o período do relógio T_{clk} . A ‘folga’ para a estabilização na entrada do FF2 é indicada pela faixa cinza no diagrama.

Hold ou Tempo de Manutenção

A segunda condição é satisfeita se o tempo de manutenção (*hold*) dos *flip-flops* é respeitado. O tempo de manutenção é definido em função de um único evento de sincronização que é observado em dois locais diferentes: a saída de um *flip-flop* que produz o valor α e o mantém estável por T_C segundos, e a entrada do *flip-flop* adjacente que deve observar aquele mesmo valor α , estável durante T_h segundos.

A Figura 7.30. mostra o relacionamento entre a borda do relógio e a especificação para *hold* entre dois *flip-flops* adjacentes. A especificação de *hold* é satisfeita quando a soma dos tempos de contaminação do *flip-flop* $T_{C,F}$ e do circuito combinacional $T_{C,C}$ são maiores do que o tempo de manutenção T_h . A ‘folga’ para o *hold* é indicada pela faixa cinza no diagrama de tempos.

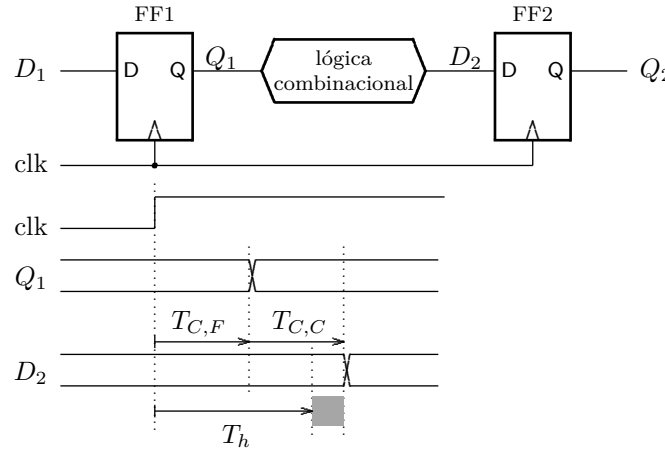


Figura 7.30: Operação apropriada: relação entre borda do relógio e *hold*.

Período Mínimo, *Setup* e *Hold*

Para que um CSS opere adequadamente, sua frequência máxima de operação é computada em função do período mínimo do relógio

$$T_{min} = 1/f_{max} \quad (7.3)$$

que é determinado pelo tempo de propagação do *flip-flop* ou registrador (T_{FF}), pelo tempo de propagação do circuito combinacional (T_{LC}) e pelo tempo de estabilização do *flip-flop* (T_{su})

$$T_{min} \geq T_{FF} + T_{LC} + T_{su} . \quad (7.4)$$

O cálculo do período mínimo do relógio atende à primeira condição para a operação apropriada se o intervalo T_{su} (*setup*) dos *flip-flops* é respeitado.

Estritamente falando, a Equação 7.4 é uma igualdade; o autor emprega ‘ $\geq$ ’ para enfatizar a necessidade da folga entre o computado e o que deve ser implementado. As razões para a folga são apresentadas no que se segue.

O tempo de manutenção deve ser respeitado para que a segunda condição seja atendida. A soma do tempo de contaminação do *flip-flop* ou registrador ($T_{C,F}$) e do circuito combinacional ($T_{C,C}$) deve ser maior do que o tempo de manutenção (T_h)

$$T_{C,F} + T_{C,C} \geq T_h . \quad (7.5)$$

As entradas de um CSS devem satisfazer à especificação do tempo de estabilização dos *flip-flops*

$$T_{s,E} \geq T_{LC} + T_{su} . \quad (7.6)$$

Mudanças nas entradas devem se propagar através do circuito combinacional com alguma folga para respeitar o *setup* dos *flip-flops*.

As entradas de um CSS devem também satisfazer à especificação do tempo de manutenção dos *flip-flops*

$$T_{h,E} \geq T_h - T_{C,C} . \quad (7.7)$$

As entradas devem permanecer estáveis após a borda do relógio para que seja possível garantir que as consequências da mudança sejam capturadas pelo registrador de estado.

As Equações 7.4 a 7.7 definem o período mínimo do relógio e os intervalos de *setup* e *hold* como desigualdades. Os valores exatos, tais como

$$T_{min} = T_{FF} + T_{LC} + T_{su} \quad (7.8)$$

definem um período mínimo em condições ideais, que dificilmente são aquelas atingidas na fabricação em larga escala de circuitos integrados. Por isso, os engenheiros de fabricação projetam os circuitos de temporização com alguma folga, para acomodar as diferenças de construção de cada cópia do circuito.

Exemplo 7.6 Considere o circuito da Figura 7.30. A especificação temporal do registrador de estado é: $T_{FF} = 2 \text{ ns}$, $T_{C,F} = 0,5 \text{ ns}$, $T_{su} = 2 \text{ ns}$, $T_h = 1 \text{ ns}$. A especificação temporal do circuito combinacional é: $T_{LC} = 10 \text{ ns}$, $T_{C,C} = 2 \text{ ns}$.

Qual é o período mínimo do relógio?

$$T_{min} = T_{FF} + T_{LC} + T_{su} = 2 + 10 + 2 = 14 \text{ ns}.$$

Há folga no tempo de manutenção do registrador?

$$T_{C,F} + T_{C,C} = 0,5 + 2 = 2,5 \text{ ns} \geq 1 \text{ ns} = T_h.$$

A folga é de 1,5 ns. Qual deve ser o *setup* mínimo nas entradas?

$$T_{s,E} \geq T_{LC} + T_{su} = 10 + 2 = 12 \text{ ns}.$$

Qual deve ser o *hold* mínimo nas entradas?

$$T_{h,E} \geq T_h - T_{C,C} = 1 - 2 = -1 \text{ ns}.$$

Há uma folga de 1 ns para acomodar a especificação de *hold* porque o tempo de contaminação do circuito combinacional $T_{C,C}$ é maior do que T_h . $\triangleleft$

Exemplo 7.7 Considere o circuito da Figura 7.30. A especificação temporal do registrador de estado é: $T_{FF} = 3 \text{ ns}$, $T_{C,F} = 1 \text{ ns}$, $T_{su} = 2 \text{ ns}$, $T_h = 2 \text{ ns}$. A especificação temporal do circuito combinacional é: $T_{LC} = 5 \text{ ns}$.

Qual deve ser o tempo mínimo de contaminação do circuito combinacional?

$$T_{C,F} + T_{C,C} = 1 \text{ ns} + T_{C,C} \geq T_h = 2 \text{ ns}; \text{ portanto } T_{C,C} \geq 1 \text{ ns}.$$

Qual é o período mínimo do relógio?

$$T_{min} \geq T_{FF} + T_{LC} + T_{su} = 3 + 5 + 2 = 10 \text{ ns}.$$

Qual deve ser o *setup* mínimo nas entradas?

$$T_{s,E} \geq T_{LC} + T_{su} = 5 + 2 = 7 \text{ ns}.$$

Qual deve ser o *hold* mínimo nas entradas, considerando-se o $T_{C,C}$ computado?

$$T_{h,E} \geq T_h - T_{C,C} = 2 - 1 = 1 \text{ ns}.$$

As entradas devem permanecer definidas e estáveis 1 ns após a borda do relógio. $\triangleleft$

Exemplo 7.8 Considere o circuito que efetua a operação ternária *multiply-add* (MADD), que multiplica dois valores e adiciona o produto a um terceiro valor. Esta operação é usada em vários algoritmos de processamento digital de sinais. O circuito da Figura 7.31 mostra uma possível implementação: o operando B é multiplicado por 2^D , e o produto é adicionado ao operando A . A multiplicação por 2^D é obtida com um deslocador exponencial como aquele descrito na Seção 6.4.1, e neste exemplo B pode ser multiplicado por 1, 2, 4, ou 8. O produto de um número de 4 bits (B) por um número de 3 bits ($\ll 3$) produz um número de 7 bits. Na entrada α do somador, A é estendido para um número de 7 bits pela concatenação de três zeros nas posições mais significativas. A soma de dois números de 7 bits é representada em 8 bits.

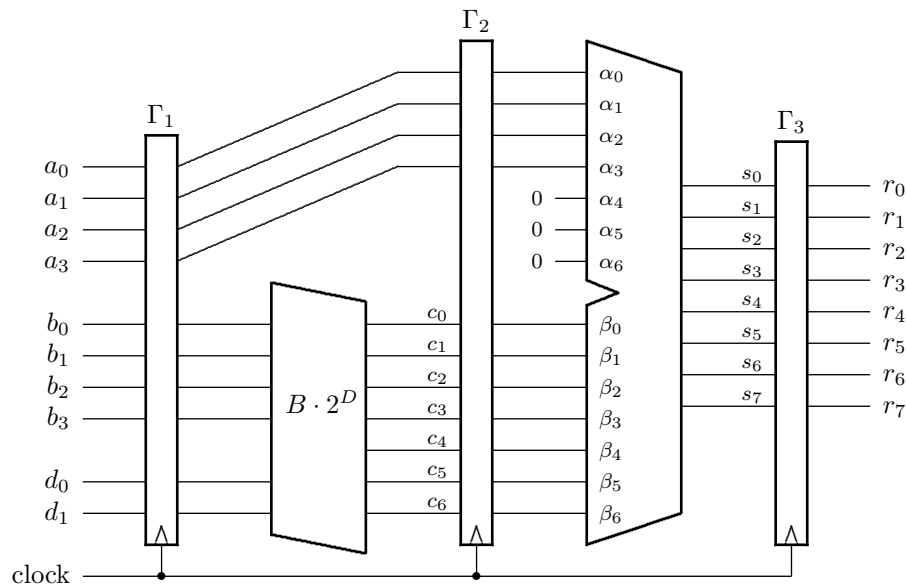


Figura 7.31: Implementação do circuito MADD.

O circuito é segmentado em quatro estágios pelos três registradores Γ_1 , Γ_2 e Γ_3 . Há um circuito ligado ao registrador Γ_1 que alimenta o MADD com operandos, e um outro ligado a Γ_3 , que consome os resultados R . A cada ciclo, uma nova tripla $\langle A, B, D \rangle$ é transferida para o segmento com o deslocador, entre Γ_1 e Γ_2 , e no ciclo seguinte a dupla $\langle A, C \rangle$ é transferida para o segmento com o somador. No terceiro ciclo, o resultado R é apresentado nas saídas do registrador Γ_3 . Um novo resultado R é produzido a cada ciclo, embora decorram dois ciclos desde a captura de uma nova tripla em Γ_1 até que o resultado para aqueles valores seja capturado em Γ_3 .

A sequência de eventos descrita acima é mostrada no diagrama de tempos da Figura 7.32. As partes hachuradas nos sinais C e S correspondem aos intervalos em que os sinais nas saídas do deslocador e do somador estão instáveis porque seus nós internos ainda não estabilizaram, e os X são valores irrelevantes.

Um diagrama de tempos detalhado para os sinais C e S é mostrado na Figura 7.33. No ciclo 1, os tempos de propagação do registrador e do deslocador são mostrados como T_{r+d} e este é ligeiramente menor que um semiciclo do relógio. No ciclo 2, os tempos de propagação do registrador e somador (T_{r+a}) são tais que há um curto intervalo entre a estabilização da soma e o *setup*.

A Equação 7.4 é uma desigualdade: o período deve ser maior ou igual que a soma dos tempos de propagação do registrador, do circuito combinacional, mais o tempo de estabilização. O período do relógio está portanto bem dimensionado para este circuito. Alguma tolerância é necessária para acomodar variações que ocorram na fabricação e/ou pequenos desvios na frequência do relógio. <

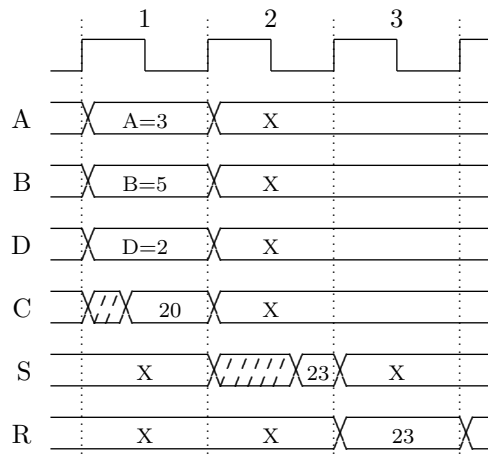


Figura 7.32: Temporização do circuito MADD.

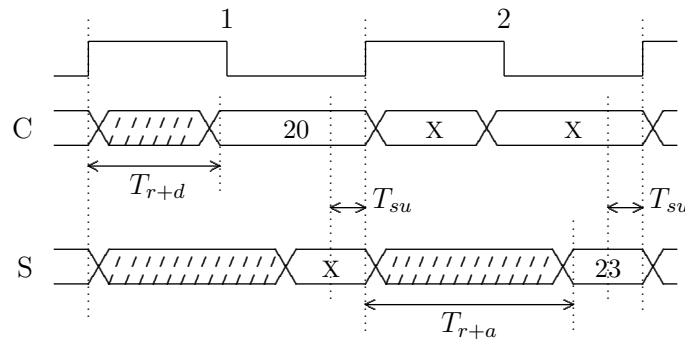


Figura 7.33: Limite na frequência máxima do relógio do MADD.

A segmentação de circuitos combinacionais é uma técnica amplamente usada para aumentar a vazão de circuitos que efetuam cálculos repetitivos. Em particular, a segmentação de processadores é uma técnica muito eficaz para aumentar a capacidade de processamento. Essa técnica é conhecida como *pipelining* porque os registradores de segmento lembram as flanges que interligam os segmentos de tubulações num reator químico.

O circuito do Exemplo 7.8 – com blocos combinacionais separados por registradores – pode ser modelado como $R \leftarrow f(A, B)$, sendo a função $f()$ implementada por um bloco combinacional com entradas A e B , com o valor da função atribuído ao registrador R no final do ciclo de relógio. Esses circuitos é que dão origem ao que se chama de “linguagem de transferência entre registradores” (*register transfer language*, ou RTL), porque os valores computados pelos circuitos combinacionais nos estágios são transferidos de registrador em registrador.

7.6.2 Operação Adequada – *Skew*

Quando jogamos uma pedra num lago, as frentes de onda se afastam do local onde a pedra afundou em círculos concêntricos. Se traçarmos uma reta que corta estes círculos, como mostra a Figura 7.34, percebe-se que uma mesma onda atinge a reta em instantes distintos (a, a'), bem como que mais de uma onda corta a reta ao mesmo tempo, como nos pontos a, b e c .

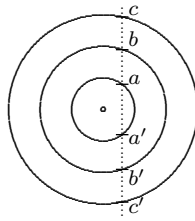


Figura 7.34: Propagação de uma frente de onda.

A analogia da onda no lago se aplica à propagação do sinal de relógio num sistema digital. O local de onde as ondas se propagam é a fonte do sinal de relógio, e as frentes de onda são as bordas do relógio, avançando com uma velocidade entre $0,6c$ e $0,7c$, que é aproximadamente um palmo por nanosegundo. Infelizmente, os efeitos das bordas de sinais digitais viajam a velocidades um tanto menores que $0,7c$ porque, embora a frente da onda eletromagnética se desloque a $0,7c$, a determinação dos níveis lógicos depende da resistência e da capacitância distribuída ao longo do trajeto dos sinais, o que acentua o problema. Se a forma das ondas for aproximada por losangos, elas avançam de forma enviesada pelo circuito, e esse comportamento é batizado de *clock skew*, por causa do viés na propagação do sinal de relógio.

O circuito da Figura 7.35 explicita os atrasos a_1 e a_2 nos sinais de relógio dos dois *flip-flops* clk_1 e clk_2 , com relação ao sinal original de relógio.

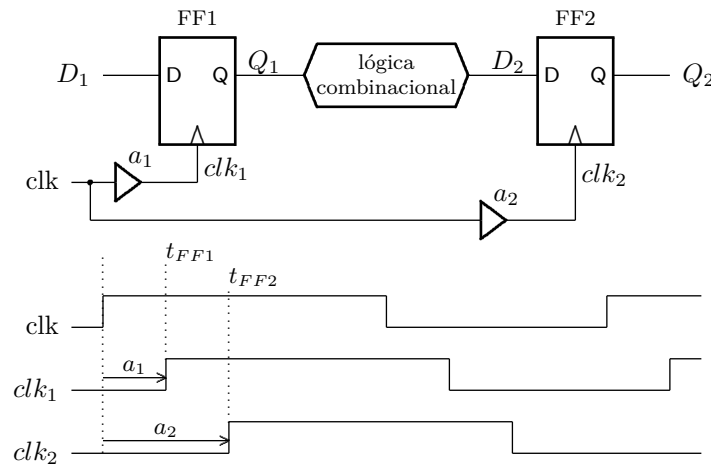


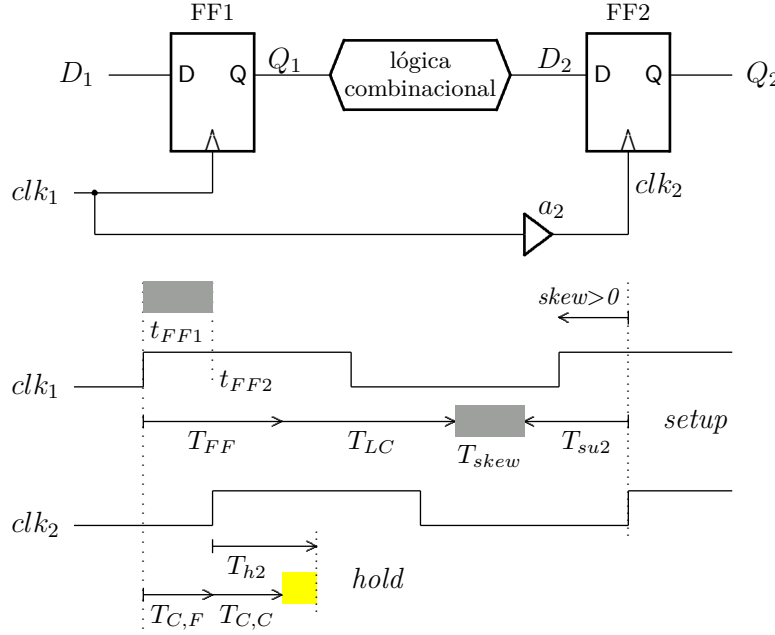
Figura 7.35: Enviesamento entre os sinais de relógio.

Para cada par de *flip-flops* adjacentes, nos interessa o atraso relativo entre os dois, e não o atraso com relação ao relógio original – a uma certa distância do centro do lago, num setor com mesmo raio, todos os pontos daquela região estão mais ou menos próximos de uma mesma frente de onda. O intervalo de enviesamento (*skew*) $T_{skew,1,2}$ entre os *flip-flops* FF1 e FF2 é dado pela diferença entre os instantes em que o sinal de relógio atinge o FF2 e o FF1, como define a Equação 7.9.

$$T_{skew,1,2} = t_{FF2} - t_{FF1} \quad (7.9)$$

Skew Positivo

Se a diferença entre os tempos de chegada das bordas de relógio é positiva, $t_{FF2} - t_{FF1} > 0$, então o *skew* é positivo, e neste caso as restrições quanto a *setup* melhoram, enquanto que as do *hold* pioram, como indica a Figura 7.36.

**Figura 7.36: Skew positivo.**

A folga no *setup* decorre do atraso da borda de subida em clk_2 com relação a clk_1 , o que aumenta o intervalo para a propagação através do *flip-flop* (T_{FF}) e da lógica combinacional (T_{LC}) mais o *setup* no *flip-flop* destino (T_{su2}).

Por outro lado, o atraso na borda do relógio em FF2 faz aumentar a exigência de contaminação no sinal D_2 , reduzindo a folga no *hold*. A soma da contaminação do FF1 ($T_{C,F}$) com a contaminação do circuito combinacional ($T_{C,C}$), descontando-se o *skew*, pode tornar-se menor do que o intervalo de manutenção no FF2 (T_{h2}). Essas condições estão explicitadas na Equação 7.10.

$$\begin{aligned} \text{Folga } setup &= T_{clk} - T_{FF} - T_{LC} - T_{su2} + T_{skew} \\ \text{Folga } hold &= T_{C,F} + T_{C,C} - T_{h2} - T_{skew} \end{aligned} \quad (7.10)$$

A contaminação das saídas é gerada no FF1 pela borda em clk_1 e é observada no FF2 na borda em clk_2 .

Skew Negativo

Se a diferença entre os tempos de chegada das bordas de relógio é negativa, $t_{FF2} - t_{FF1} < 0$, então o *skew* é negativo. No exemplo mostrado na Figura 7.37, a restrição quanto a *setup* é violada porque o período do relógio é curto demais para acomodar a propagação dos sinais e o intervalo de *setup*:

$$T_{min} < T_{FF} + T_{LC} + T_{su2}. \quad (7.11)$$

A restrição do *hold* melhora porque a contaminação inicia com atraso, o que aumenta o intervalo de estabilidade no sinal D_2 antes da borda em clk_2 .

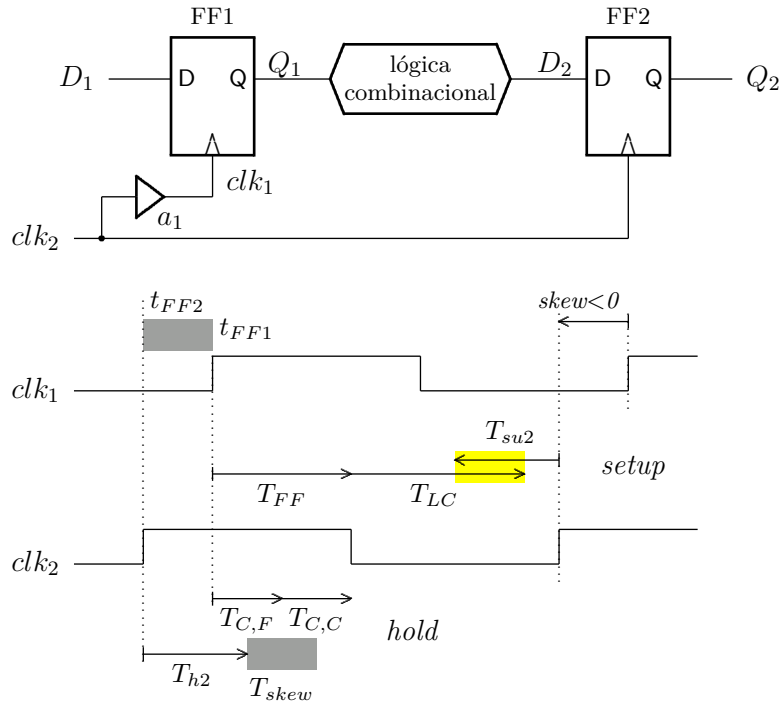


Figura 7.37: Skew negativo.

Por causa do atraso na borda de clk_1 , o intervalo entre as bordas do relógio nos *flip-flops* fonte e destino diminui – as bordas se aproximam – o que reduz a folga no cálculo do *setup*. O atraso na chegada do relógio em FF1 diminui a exigência de contaminação no sinal D_2 , melhorando a folga para a restrição de *hold*. Estas condições são as mesmas que na Equação 7.10, exceto que $T_{skew} < 0$.

Período Mínimo e Tempo de Manutenção na Presença de *Skew*

Quando há enviesamento nos sinais de relógio, a equação que computa o período mínimo do relógio deve ser adaptada para acomodar as diferenças entre as bordas dos sinais de relógio. O *skew* deve ser subtraído ao período mínimo T_{min} , enquanto que o *skew* é adicionado no cálculo da contaminação, como mostram as Equações 7.12 e 7.13. Lembre que $T_{skew} = t_{FF2} - t_{FF1}$.

$$T_{min} \geq T_{FF} + T_{LC} + T_{su} - T_{skew} \quad (7.12)$$

$$T_{C,F} + T_{C,C} - T_{skew} \geq T_h \quad (7.13)$$

Exemplo 7.9 Considere o circuito da Figura 7.30, na página 199. A especificação temporal dos *flip-flops* é: $T_{FF} = 10$ ns, $T_{C,F} = 5$ ns, $T_{su} = 4$ ns, $T_h = 2$ ns. A especificação temporal do circuito combinacional é: $T_{LC} = 6$ ns, $T_{C,C} = 2$ ns. O *skew* é $T_{skew} = -5$ ns.

Qual é o período mínimo do relógio?

$$T_{min} = T_{FF} + T_{LC} + T_{su} - T_{skew} = 10 + 6 + 4 - (-5) = 25 \text{ ns.}$$

Há folga no tempo de manutenção do FF2?

$$\text{Folga } hold = T_{C,F} + T_{C,C} - T_{h2} - T_{skew} = 5 + 2 - 2 - (-5) = 10 \text{ ns.}$$

Há folga no tempo de estabilização do FF2?

$$\text{Folga } setup = T_{clk} - T_{FF} - T_C - T_{su2} + T_{skew} = 25 - 10 - 6 - 4 - 5 = 0$$

Neste caso não há folga porque empregou-se $T_{clk} = T_{min}$.

◁

Exemplo 7.10 Considere o circuito da Figura 7.30, com a seguinte especificação temporal: *flip-flops* com $T_{FF} = 6$ ns, $T_h = 3$ ns, $T_{su} = 6$ ns, $T_{C,F} = 1$ ns, lógica combinacional com $T_{C,C} = 4$ ns e $T_{LC} = 25$ ns. O *skew* é $T_{skew} = +1$ ns.

Qual é o período mínimo do relógio?

$$T_{min} = T_{FF} + T_{LC} + T_{su} - T_{skew} = 6 + 25 + 6 - 1 = 36 \text{ ns.}$$

Há folga no tempo de manutenção do FF2?

$$T_h = 3 \text{ ns} \leq T_{C,F} + T_{C,C} - T_{skew} = 1 + 4 - 1 = 4 \text{ ns.}$$

Há folga de 1ns, e portanto o circuito está completo.

◁

7.6.3 Definição de Tempo de Propagação Revisitada

Finalmente estamos em condições de explicitar uma premissa escondida na definição de *tempo de propagação*. A definição da Seção 5.4 reza que o intervalo de propagação se inicia quando *todas as entradas ficam estáveis*. O que está escondido na definição é que o modelo de circuito que se emprega, na grande maioria dos casos, é aquele da Figura 7.29, no qual as entradas do circuito combinacional provêm de um registrador, e suas saídas alimentam outro registrador. Essa metodologia de projeto é a usada no circuito do Exemplo 7.8.

Se todas as entradas do circuito combinacional provêm da saída de um ou mais registradores, então estes sinais estabilizam aproximadamente ao mesmo tempo. Todas as entradas são atualizadas na mesma borda do relógio, e o circuito que conduz o sinal de relógio é otimizado para minimizar o *skew* entre todos os registradores fonte e todos os registradores destino.

7.6.4 “Não Se Faz Lógica Com O Relógio”

A máxima de projeto “não se faz lógica com o relógio” pretende defender o projetista contra uma tentação comum, que é de acrescentar lógica (combinacional) ao sinal de relógio⁵. Uma porta lógica aqui e outra acolá reduzem dramaticamente qualquer possibilidade de que o *skew* seja eliminado do sinal de relógio. Por “lógica com o relógio” não me refiro aos atrasos introduzidos para compensar as diferenças no tempo de propagação do sinal pelo circuito, mas sim a tentativas preguiçosas de inserir ou remover bordas no sinal de relógio.

Se o que se deseja é controlar a atualização do conteúdo de *flip-flops* ou registradores, deve-se empregar registradores como aquele da Figura 7.26, no qual a lógica de decisão é inserida exclusivamente no caminho de dados, e nunca no caminho de sincronização. Além do problema com *skew*, o circuito combinacional pode introduzir transitórios na entrada de relógio, que causam erros que são particularmente difíceis de detectar.

Sistemas de síntese são compiladores que produzem uma descrição do circuito a partir de código escrito numa linguagem de descrição de hardware, tais como VHDL ou Verilog. Estes compiladores proíbem lógica com o sinal de relógio porque isso inviabiliza a otimização do *skew* na rede de distribuição daquele sinal.

Exercícios

Ex. 7.1 Desenhe um diagrama de tempos similar ao da Figura 7.32 para a sequência de entradas $\langle 1, 2, 0 \rangle$, $\langle 4, 5, 1 \rangle$, $\langle 7, 9, 2 \rangle$, $\langle 15, 15, 3 \rangle$. Considerando que a cada ciclo uma das triplas na sequência é amostrada, quantos ciclos são necessários para computar os quatro resultados?

Espaço em branco proposital.

⁵O autor escutou essa frase, pela primeira vez, dita pelo Professor Anatólio Laschuck, na UFRGS, no início dos anos 1980.

Ex. 7.2 Considere o circuito da Figura 7.38. A especificação temporal dos *flip-flops* é: $T_{FF} = 3$ ns, $T_{C,F} = 1$ ns, $T_{su} = 2$ ns, $T_h = 2$ ns, e o *skew* é zero. A especificação temporal do circuito combinacional é: $T_{LC} = 10$ ns, $T_{C,C} = 3$ ns. (a) Qual é o período mínimo do relógio? (b) Há folga no tempo de *hold* do FF2? (c) Qual deve ser o *setup* mínimo nas entradas? (d) Qual deve ser o *hold* mínimo nas entradas?

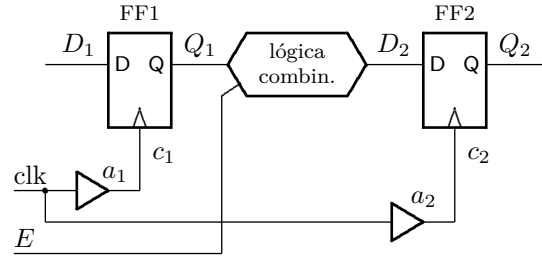


Figura 7.38: Circuito com entradas e *skew*.

Ex. 7.3 Considere o circuito da Figura 7.38. A especificação temporal dos *flip-flops* é: $T_{FF} = 4$ ns, $T_{C,F} = 1$ ns, $T_{su} = 2$ ns, $T_h = 2$ ns, $T_{skew} = -3$ ns. A especificação temporal do circuito combinacional é: $T_{LC} = 12$ ns, $T_{C,C} = 4$ ns. (a) Qual é o período mínimo do relógio? (b) Há folga no tempo de *hold* do FF2? (c) Qual deve ser o *setup* mínimo nas entradas? (d) Qual deve ser o *hold* mínimo nas entradas?

Índice Remissivo

Símbolos

T_A , 64
 T_D , 120
 T_I , 109, 116
 T_M , 117, 119, 186
 T_P , 116
 T_h , 188, 192, 199, 204
 T_p , 171
 $T_{C,F}$, 192
 $T_{C,x}$, 118–120, 199, 204, 206
 T_{FF} , 192
 T_{min} , 199, 206
 T_{skew} , 203
 T_{su} , 188, 192, 199, 204
 $\%$, 18–20
 $\Rightarrow$, 37
 $\bigvee$, 47
 $\bigwedge$, 47
 $\equiv$, 36
 $\gg$, 155
 $\wedge$, 30, 36
 $\triangleleft \triangle$, 36, 42, 66
 $\leftarrow$, 193
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 41, 44
 $\ll$, 155
 $\neg$, 30, 36, 154
 $\vee$, 30, 36
 $\oplus$, 36, 53, 65
 $\mapsto$, 42
 $\mathbb{N}$, 146
 $\bar{a}$, *veja* $\neg$
 π , 25
 $\backslash$, 77
decod, 72
demux, 75
e, 24
mod, 158
mux, 67
 $\mathbb{B}$, 29–33, 38, 42
 $\mathbb{Z}$, 42, 149
 $\mathbb{N}$, 42, 43, 149
num, 44, 55, 72, 77
 num^{-1} , 55
 $\mathbb{R}$, 42
 $\langle \rangle$, 29, 42, 43, 180, 201

Números

74148, 75

A

abstração, 27
 bits como sinais, 27–33, 57, 182, 184
 tempo discretizado, 116, 118, 182, 186–188, 191–193, 197
 acumulação de produtos parciais, 173–178
 adição, 152
 adiantamento de vai-um, 164–172
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo,
 conversão de base, 18
 conversão de base de frações, 22
 amplificador, 125, 136
 diferencial, 138
 and, *veja* $\wedge$
and array, 128
 aproximação, 24
 árvore, 64, 118
assembly, *veja* ling. de montagem
 associatividade, 31
 atraso, *veja* tempo de propagação, 57
 atribuição, 12

B

barramento, 140
barrel shifter, 158
 básculo, 138, 185–189
 SR, 185, 194
 binário, 20
 bit, 20
 de sinal, 147
 bits, 27–37, 65
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 variável, 30
 borda, *veja* relógio
branch, *veja* desvio condicional
buffer, 123
buffer three-state, 140
 buraco, 87
 byte, 11

C

C,
 deslocamento, 160
 overflow, 163
 cadeia,

de portas, 64, 118
 de somadores, 152
 caminho crítico, 117
 capacitor, 105, 107, 135, 136
capture FF, veja *flip flop*, destino
 CAS, 134
 célula de RAM, 81
 célula, 100
 chave,
 analógica, 142
 digital, 92
 normalmente aberta, 92
 normalmente fechada, 78, 92
 ciclo,
 combinacional, 57
 violação, 81, 184, 186, 188
 circuito,
 combinacional, 57, 65
 dual, 99
 segmentado, 202
 sequencial síncrono, 196
 circuito aberto, 57
clear, 194
clk, veja relógio
clock, veja relógio
clock skew, veja *skew*
clock-to-output, 192
clock-to-Q, 192
 CMOS, 59, 65, 85–142
 buffer three-state, 140
 célula, 100
 inversor, 96
 nand, 99
 nor, 98
 porta de transmissão, 141
 portas inversoras, 99
 sinal restaurado, 125
 código,
 Gray, 63
Column Address Strobe, veja CAS
 combinacional,
 ciclo, 57
 circuito, 57
 dispositivo, 57
 comparador,
 de igualdade, 62, 164
 de magnitude, 164
Complementary Metal-Oxide Semiconductor, veja CMOS
 complemento, veja $\neg$
 complemento de dois, 145–151, 154–164
 complemento, propriedade, 31
 comportamento transitório, veja transitório
 comutatividade, 31
 condicional, veja $\triangleleft \triangleright$
 condutor, 85
 conjunção, veja $\wedge$
 conjunto mínimo de operadores, 65
 contra-positiva, 41

controlador,
 de memória, 136
 conversão de base, 18
 corrente, 85, 105, 106, 112, 113
 de fuga, 115
 corrida, 123, 125
 CSS, 196–197
 curto-circuito, 57

D

datapath, veja circuito de dados
 decimal, 17
 decodificador, 72, 78, 81–82, 84, 126
 de linha, 126, 135
 de prioridades, 74
delay, 57
 demultiplexador, 75, 120
design unit, veja VHDL, unidade de projeto
 deslocador exponencial, 156
 deslocamento, 155–158
 aritmético, 155, 158, 164
 exponencial, 159, 201
 lógico, 155, 162
 rotação, 158
 detecção de bordas, 123
 disjunção, veja $\vee$
 dispositivo, 85
 combinacional, 57
 distributividade, 31, 34, 51
 divisão inteira, 44
 doador, 87
don't care, 70
 dopagem por difusão, 86
 dopante, 86
 DRAM, 134–137
 controlador, 136
 fase de restauração, 137
 linha,
 de palavra, 135
 linha de bit, 135
 linha de palavra, 136
 página, 135
 refresh, 135
 dual, 32, 95, 99
 dualidade, 32

E

EEPROM, 133
 endereço, 77
 energia, 100, 105, 112–115
 enviesado, relógio, veja *skew*
 EPROM, 133
 equação característica do FF, 193
 equivalência, veja $\Leftrightarrow$
 erro,
 de representação, 23
 especificação, 42
 estado, 182
 estado atual, 196

expressões, 36

F

fan-in, 109–112, 116, 167, 170

fan-out, 82, 84, 109–112, 116, 120, 170–172

fechamento, 31

FET, 91

Field Effect Transistor, veja FET

Field Programmable Gate Array, veja FPGA

filtro digital de ruído, 190

flip-flop, 188–195

adjacentes, 198

comportamento, 193

destino, 198

fonte, 197, 198

mestre-escravo, 189

modelo VHDL, veja VHDL, *flip-flop*

temporização, 192

tipo JK, 193

tipo T, 191, 193

um por estado, veja um FF por estado

folga de *hold*, 198

folga de *setup*, 198

forma canônica, 48

FPGA, 194

frações, veja ponto fixo

frequência máxima, veja relógio

função, 30

tipo, 29, 42

função, aplicação bit a bit, 32

função, tipo (op. infixo), veja $\mapsto$

G

gerador de relógio, 192

glitch, veja transitório

GND, 93

gramática, 17

H

hexadecimal, 19

hold time, 188, 197–202

folga, 198, 204, 205

I

idempotência, 31

identidade, 31

igualdade, 30

implementação, 42

implicação, veja $\Rightarrow$

informação, 16

inicialização,

flip-flop, 194

Instrução,

busca, veja busca

instrução, 12

busca, veja busca

decodificação, veja decodificação

execução, veja execução

resultado, veja resultado

interface,

de rede, 13

de vídeo, 12

inversor, 96

tempo de propagação, 109

involução, 31, 61

isolante, 85

J

Joule, 112

jump, veja salto incondicional

L

latch, veja búsculo

latch FF, veja *flip flop*, destino

launch FF, veja *flip flop*, fonte

Lei de Joule, 112

Lei de Kirchhoff, 106

Lei de Ohm, 105, 106

ligação,

barramento, 140

em paralelo, 93, 99

em série, 93, 99

linguagem,

assembly, veja ling. de montagem

C, veja C

Pascal, veja Pascal

VHDL, veja VHDL

Z, 27

linha de endereçamento, 78

literal, 38

logaritmo, 43

lógica restauradora, 125

M

MADD, 201

Mapa de Karnaugh, 49, 124

Máquina de Mealy, veja máq. de estados

Máquina de Moore, veja máq. de estados

máscara, 32

máximo e mínimo, 31

maxtermo, 46

Mealy, veja máq. de estados

memória, 181

atualização, 77

bit, 184

de vídeo, 13

decodificador de linha, 80

endereço, 77

FLASH, 133

matriz, 129, 132, 134

multiplexador de coluna, 80

primária, 13

RAM, 81, 134

ROM, 78, 126

secundária, 13

memória dinâmica, veja DRAM

memória estática, veja SRAM

metaestabilidade, 184, 188, 191, 194

defesa contra artes das trevas, 191

minitermo, 45, 124, 126

modelo,
 funcional, 44
 porta lógica, 96
 temporização, 115
 módulo, *veja* %, *mod*
 Moore, *veja* máq. de estados
 MOSFET, 91
 multiplexador, 61, 66–70, 80, 101, 117, 119, 123–
 124, 126, 141, 142
 de coluna, 132, 136
 multiplicação, 172–178
 acumulação de produtos parciais, 173–178
multiply-add, *veja* MADD

N

número,
 de Euler, 24
 negação, *veja* $\neg$
 nível lógico,
 0 e 1, 28
 indeterminado, 28, 109, 116, 141
 terceiro estado, 140
 nó, 96
non sequitur, 37
 not, *veja* $\neg$
 número primo, 53

O

octal, 18
 onda quadrada, 188
 operação,
 binária, 29
 bit a bit, 32
 infixada, 42
 MADD, 201
 prefixada, 47
 unária, 29
 operação apropriada, 197
 operações sobre bits, 29–33
 operador,
 binário, 29
 lógico, 36
 unário, 29
operation code, *veja* opcode
 or, *veja* $\vee$
or array, 129
 ou exclusivo, *veja* $\oplus$
 ou inclusivo, *veja* $\vee$
overflow, 148–150, 154, 162–164, 173, 178

P

paridade,
 ímpar, 49
 par, 49
 período mínimo, *veja* relógio
pipelining, *veja* segmentação, 202
 piso, *veja* [v]
 ponto fixo, 151–152
 ponto flutuante, 70
 porta lógica, 65

and, 59
 carga, *veja fan-out*
 de transmissão, 141, 185
 nand, 60, 99
 nor, 60, 98
 not, 59, 96
 or, 59
 xor, 60, 65, 191
 portas complexas, 100
 potência, 112–115
 dinâmica, 114
 estática, 115
 potenciação, 43
 precedência, 30
 precisão,
 representação, 23
preset, 194
 prioridade,
 decodificador, 74
 processador, 12
 produtivo, 33
 produto de somas, 46
 programa de testes, *veja* VHDL, *testbench*
 PROM, 133
 propriedades, operações em $\mathbb{B}$, 31
 prova de equivalência, 40–41
 próximo estado, 196
pull-down, 96
pull-up, 96, 126, 141
 pulso, 122, 123, 185, 194
 espúrio, *veja* transitório

R

RAM, 12, 77, 81, 134–139
 célula, 81
 dinâmica, 135
Random Access Memory, *veja* RAM
 RAS, 134
Read Only Memory, *veja* ROM
 realimentação, 81
 receptor, 87
 rede, 96
 redução, 33
refresh, 137, 139
Register Transfer Language, *veja* RTL
 registrador, 195
 carga paralela, 195
 de segmento, 201
 simples, 195
 registrador de deslocamento,
 modelo VHDL, *veja* VHDL, registrador
 registrador de estado, 196
 relógio, 186, 188, 192
 bordas,
 ascendente, 189
 descendente, 189
 enviesado, *veja skew*
 frequência máxima, 199
 período mínimo, 199

representação,
 abstrata, 28
 binária, 20
 complemento de dois, 147
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 ponto fixo, 151
 posicional, 17
 precisão, 23
reset, 185, 194, 195
 resistência, 87, 100, 105
 ROM, 12, 77–80, 126–133
 rotação, 158, 164
Row Address Strobe, *veja* RAS
 RTL, 202

S

segmentação, 201
 seletor, 72
 semântica, 17
 semicondutor, 85
 tipo N, 87
 tipo P, 87
set, 185, 194
setup time, 188, 197–202
 folga, 198, 204, 205
 silogismo, 37
 simplificação de expressões, 38–40
 sinal, 27, 42
 analógico, 27
 digital, 27, 28
 fraco, 125, 126, 138, 141
 intensidade, 90, 139
 restaurado, 125, 139
 síntese, *veja* VHDL, síntese
skew, 202–207
Solid State Disk, *veja* SSD
 soma, *veja* somador
 soma de produtos, 45, 51, 126
 completa, 45
 somador, 145, 152–153
 adiantamento de vai-um, 165
 cadeia, 152
 completo, 104, 152
 overflow, 163
 parcial, 103
 seleção de vai-um, 170–172
 temporização, 201
 teste, 178
 somatório, 33
 spice, 28
 SRAM, 138–139
 SSD, 14
 status, 162
 subtração, 154
 superfície equipotencial, 96, 110

T

tabela,
 de excitação dos FFs, 193
 tabela verdade, 33–35, 45
 tamanho, *veja* |N|
 tempo,
 de contaminação, 115, 118–121, 184, 192
 de estabilização, 188, 192
 de manutenção, 188, 192
 de propagação, 57–58, 61, 64–65, 73, 77, 84, 102,
 104, 109, 115–117, 192, 207
 discretizado, 186, 188, 192, 197
 temporização, 104–126
 tensão, 105
 Teorema,
 Absorção, 49
 DeMorgan, 32, 41, 48, 54, 60, 61, 94, 98
 Dualidade, 99
 Simplificação, 49
 terceiro estado, 140–141
testbench, *veja* VHDL, *testbench*
 teste,
 cobertura, 179
 de corretude, 178
 teto, *veja* [r]
three-state, *veja* terceiro estado
 tipo,
 de sinal, 42
 função, 29
 Tipo I, *veja* formato
 Tipo J, *veja* formato
 Tipo R, *veja* formato
 transferência entre registradores, *veja* RTL
 transistor, 87–91, 95–96
 CMOS, 95
 corte, 113
 gate, 88
 saturação, 113
 sinal fraco, 90
 tipo N, 88
 tipo P, 89
Transistor-Transistor Logic, *veja* TTL
 transitório, 121–123, 186
transmission gate, *veja* porta de transmissão
 TTL,
 74148, 75
 tupla, *veja* ⟨ ⟩
 elemento, 32
 largura, 43

U

ULA, 160–164, 180
 status, 162
 Unidade de Lógica e Aritmética, *veja* ULA

V

valor da função, 30
 VCC, 93
 vetor de bits, *veja* ⟨ ⟩, 32

largura, 43
VHDL, 194
 design unit, veja VHDL, unidade de projeto
 síntese, 207
 std_logic, 140
 tipos, 42

W

Watt, 112
write back, veja resultado

X

xor, veja $\oplus$

Z

Z, linguagem, 27

Capítulo 8

Circuitos Sequenciais Síncronos

*Cuatro son las historias. Durante el tiempo que nos queda
seguiremos narrándolas, transformadas.*

Jorge Luis Borges, El oro de los tigres.

Um circuito combinacional é um circuito com entradas e saídas digitais, cujo comportamento funcional é definido por uma função que especifica os valores das saídas para cada uma das combinações de entradas. Seu comportamento temporal é determinado pelo tempo de propagação, que determina um limite superior para o intervalo entre o instante em que as entradas estabilizaram e as saídas estabilizem nos valores determinados pela especificação funcional. Circuitos combinacionais não podem conter ciclos, curto-circuitos entre saídas e tampouco entradas desconectadas.

Um *circuito sequencial* produz saídas que dependem da sequência com que os valores são apresentados às suas entradas. Num dado instante, a saída de um circuito sequencial depende da sequência de todos os valores apresentados às suas entradas. Posto de outra forma, o *estado atual* de um circuito sequencial depende das entradas e da sequência de todos os estados anteriores. Circuitos sequenciais empregam, necessariamente, alguma forma de memória para armazenar o estado atual.

Circuitos sequenciais são extremamente úteis porque seu comportamento pode ser definido como uma *sequência de estados*, que é determinada pela sequência de entradas. Um computador nada mais é do que um circuito sequencial de grande complexidade, cuja sequência de estados é determinada pelo programa que está sendo executado e pelos dados de entrada. Este capítulo¹ define o que se entende por um *circuito sequencial síncrono* (CSS) e apresenta um conjunto de técnicas de projeto destes circuitos.

Dependendo de como são interligados os *flip-flops* de um registrador, sua sequência de estados pode representar um subconjunto dos números naturais, conforme mostra a Seção 8.2, que trata de *contadores*. Contadores podem produzir sinais de sincronização particularmente úteis e estes são discutidos na Seção 8.2.10.

A Seção 8.3 apresenta uma outra maneira de interligar os *flip-flops* para formar *registradores de deslocamento*, que permitem deslocar um certo número de bits, de uma ou mais posições para a direita, ou para a esquerda. Estes registradores são os componentes básicos de qualquer circuito de comunicação serial, nos quais os bits são enviados ou recebidos um a um através

¹© Roberto André Hexsel, 2012-2020. Versão de 2 de dezembro de 2020.

de um fio ou através de um par de antenas.

A Seção 8.4 discute a metodologia para o projeto de uma classe particularmente útil de CSSs, que são as *máquinas de estados finitas*. Estas máquinas de estados (MEs) podem ser consideradas uma generalização de circuitos contadores.

A Seção 8.5 introduz uma metodologia de projeto para máquinas de estados complexas. Esta metodologia permite o projeto rápido e eficaz de máquinas complexas, chamadas de *sequenciadores* ou de *controladores*. Os *microcontroladores* são máquinas de estado cujo comportamento é determinado por uma forma rudimentar de programa.

Quando há abundância de recursos, especialmente *flip-flops*, podemos implementar máquinas de estado através de uma ‘tradução’ direta do *diagrama de estados*. Esta forma de implementação de MEs, com *um flip-flop por estado* é discutida na Seção 8.6.

A Seção 8.7 apresenta exemplos de circuitos sequenciais relativamente complexos, obtidos da composição de blocos simples como registradores, memórias, e circuitos combinacionais como somadores e multiplicadores. A operação de tais circuitos pode ser coordenada por máquinas de estado ou por microcontroladores.

As quatro ‘histórias’ referidas na citação a Borges são (i) os contadores, (ii) os registradores de deslocamento, (iii) as máquinas de estado finitas, especialmente os microcontroladores, e (iv) os processadores, que são uma classe de microcontroladores cuja sequência de controle é o programa executado no processador. A quarta ‘história’ merece um capítulo inteiro, que é o Capítulo 9.

8.1 Uma Rápida Olhada no Relógio

Os circuitos que são o objeto deste capítulo são qualificados como “síncronos” porque as mudanças de estado acontecem sincronicamente a um sinal que define os instantes em que as transições podem ocorrer. Estes sinais são chamados de *relógio* por conta da periodicidade e regularidade das transições.

Um sinal contínuo no tempo e que apresenta comportamento cíclico pode ser descrito pelas suas *amplitude* e *frequência*. A amplitude é a distância entre seus valores máximos e mínimos, e em circuitos digitais a amplitude dos sinais é tal que varia entre valores modelados como 0 e 1 – isto é uma idealização, embora na prática a amplitude dos sinais permaneça (quase) sempre entre os limites de tensão das faixas aceitáveis como nível lógico 0 e o nível lógico 1 para uma dada tecnologia de circuitos integrados.

A frequência f de um sinal periódico é o número de vezes por segundo em que o ciclo do sinal se repete, e a unidade da frequência é Hertz, com unidade [Hz]. O período T de um sinal é a duração de um ciclo, e é medido em segundos [s]. Frequência e período são relacionados pela Equação 8.1.

$$T = 1/f \quad (8.1)$$

A Figura 8.1 mostra um período de uma *onda quadrada* que poderia ser usada como sinal de relógio. Esta forma de onda é chamada de ‘quadrada’ porque o semiciclo em 0 tem a mesma duração do semiciclo em 1.

A faixa de frequências empregadas em circuitos digitais é ampla, desde frequências abaixo de um Hertz ($f \propto 0,25Hz$), em sinais de trânsito, até uma dezena de gigahertz, em circuitos de

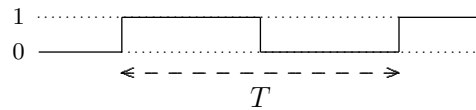


Figura 8.1: Período de um sinal periódico.

alto desempenho ($f \propto 10^{10} Hz$). Analogamente, a faixa dos períodos destes sinais também é ampla, desde uns poucos segundos até picossegundos ($T \propto 10^{-12} s$). A Tabela 8.1 relaciona as potências de 10 e os nomes usados para qualificar as frequências e períodos. Em se tratando de tempo ou frequência, são usadas potências de 10 e não potências de 2. Isto se deve ao desenvolvimento das telecomunicações ter ocorrido antes do desenvolvimento da computação automática e a nomenclatura empregada pelos engenheiros de telecomunicações se manteve em uso.

Tabela 8.1: Tabela de potências de 10.

potência	nome	símbolo
-12	pico	p
-9	nano	n
-6	micro	μ
-3	mili	m
3	kilo	k
6	mega	M
9	giga	G
12	tera	T

8.2 Contadores

Contadores são circuitos sequenciais síncronos com um comportamento cíclico, e cuja sequência de estados pouco depende de outros estímulos externos além do sinal de relógio – em sua forma mais simples, um contador é um circuito que *conta pulsos do relógio*. Esta seção apresenta vários exemplos de contadores sequenciais síncronos. A Figura 8.2 mostra o modelo para um circuito sequencial sem nenhuma entrada. O sinal de relógio deve estar sempre presente mas não é contado como sendo uma entrada do CSS.

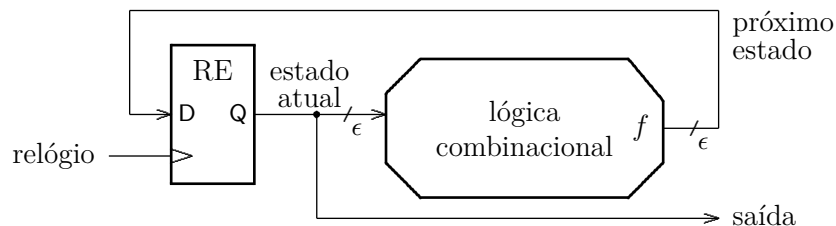


Figura 8.2: Modelo de contador como um circuito sequencial síncrono.

Quando falamos de contadores, usamos tuplas de bits para representar números naturais ou inteiros, como vimos na Seção 6.1.

8.2.1 Contador Módulo-2

Nosso primeiro circuito contador segue uma sequência de contagem de tamanho dois, e é chamado de *contador módulo-2*:

$$\text{módulo-2: } 0 \mapsto 1 \mapsto 0 \mapsto 1 \dots$$

e sua tabela verdade é mostrada na Figura 8.2. O lado esquerdo da tabela mostra o estado atual (EA), que é o valor da contagem no ciclo c , e sua representação como um número natural. O lado direito da tabela mostra o próximo estado (PE), que é o valor da contagem no ciclo $c+1$, e sua representação como um número natural. O sinal d_0 é a entrada do *flip-flop*, e o sinal q_0 é sua saída.

Tabela 8.2: Sequência de contagem módulo-2.

	EA	PE	
N	q_0	d_0	N
0	0	1	1
1	1	0	0

Ao inspecionar a Tabela 8.2, percebe-se que a função que determina o próximo estado, em função do estado atual, é um inversor. O circuito do contador módulo-2 é mostrado na Figura 8.3.

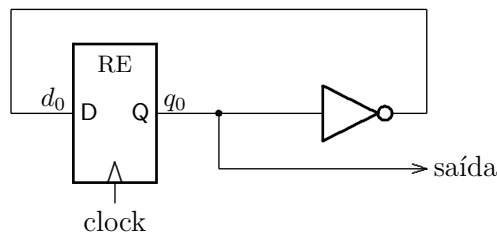


Figura 8.3: Contador módulo-2.

8.2.2 Contador Módulo-4

Um *contador módulo-4* conta numa sequência com quatro itens:

$$\text{módulo-4: } 00 \mapsto 01 \mapsto 10 \mapsto 11 \mapsto 00 \mapsto 01 \mapsto 10 \dots$$

e a Tabela 8.3 define esta sequência para um circuito com dois *flip-flops*.

Tabela 8.3: Sequência de contagem módulo-4.

	EA	PE	
N	$q_1 q_0$	$d_1 d_0$	N
0	0 0	0 1	1
1	0 1	1 0	2
2	1 0	1 1	3
3	1 1	0 0	0

Inspecionando a tabela, percebe-se que o próximo estado de q_0 é sempre o seu complemento, e a função de próximo estado para este *flip-flop* é $d_0 = \overline{q_0}$. O próximo estado de q_1 é 1 somente quando os dois bits do estado atual $\langle q_1, q_0 \rangle$ são distintos e portanto a função de próximo estado do *flip-flop* q_1 é $d_1 = q_1 \oplus q_0$. O circuito do contador módulo-4 é mostrada na Figura 8.4.

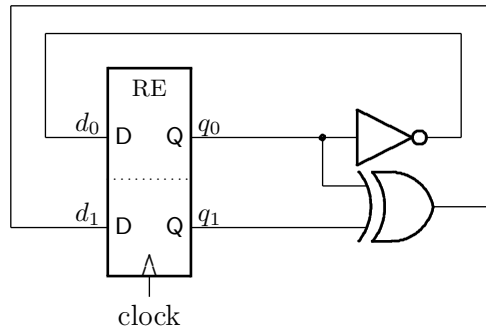


Figura 8.4: Contador módulo-4.

Embora pareçam um tanto inúteis, estes circuitos executam uma função importante: eles dividem a frequência do relógio por 2^n . A cada borda ascendente, a saída q_0 troca de estado, e o período do sinal em q_0 é o dobro daquele de *clock*. A cada segunda borda ascendente, a saída q_1 troca de estado, e o período do sinal em q_1 é o quádruplo daquele de *clock*. A frequência de q_0 é 1/2 da de *clock*, e a de q_1 é 1/4 da de *clock*.

8.2.3 Contador Módulo-8

Um *contador módulo-8* conta numa sequência com oito itens:

$$\text{módulo-8: } 000 \mapsto 001 \mapsto 010 \mapsto \dots \mapsto 110 \mapsto 111 \mapsto 000 \mapsto 001 \dots$$

e a Tabela 8.4 define a função de próximo estado deste contador. Da mesma forma que nos outros contadores, a função de próximo estado para o *flip-flop* q_0 é $d_0 = \overline{q_0}$ porque o bit menos significativo sempre inverte a cada novo elemento da sequência: par $\mapsto$ ímpar $\mapsto$ par $\mapsto$ ímpar...

Tabela 8.4: Sequência de contagem módulo-8.

N	EA			PE			N
	q_2	q_1	q_0	d_2	d_1	d_0	
0	0	0	0	0	0	1	1
1	0	0	1	0	1	0	2
2	0	1	0	0	1	1	3
3	0	1	1	1	0	0	4
4	1	0	0	1	0	1	5
5	1	0	1	1	1	0	6
6	1	1	0	1	1	1	7
7	1	1	1	0	0	0	0

A metade de cima da tabela, ignorando-se a metade com $q_2 = 1$, é exatamente a mesma função do contador módulo-4 e portanto $d_1 = q_1 \oplus q_0$. O mesmo vale para a metade de baixo da tabela, com $q_2 = 1$.

A função de próximo estado d_2 pode ser obtida da inspeção da tabela verdade: q_2 se altera, e inverte, somente quando q_1 e q_0 são 1. Assim, $d_2 = q_2 \oplus (q_1 \wedge q_0)$. A Figura 8.5 mostra o circuito do contador módulo-8.

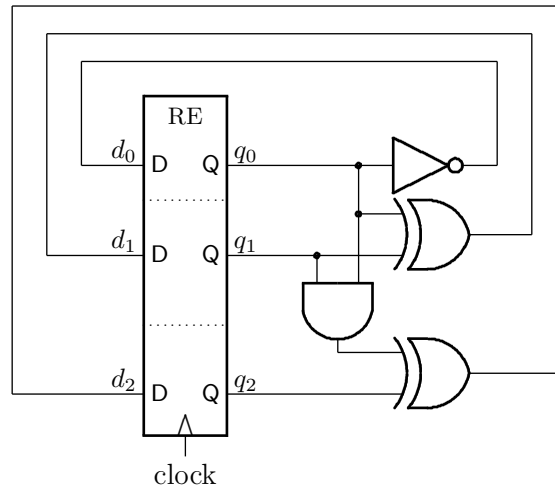


Figura 8.5: Contador módulo-8.

Intuitivamente, o bit n do contador inverte somente quando

$$1 + \langle q_{n-1}q_{n-2} \dots q_1q_0 \rangle$$

produz vai-um, e isso ocorre somente se os bits $q_{n-1} \dots q_0$ são todos 1.

O diagrama de tempo do contador módulo-8 é mostrado na Figura 8.6. A cada borda ascendente do sinal de relógio, o sinal q_0 troca de valor, e seu período é o dobro daquele do sinal de relógio. Em todas as bordas em que $q_0 = 1$, q_1 troca de valor, e portanto seu período é o quádruplo daquele do relógio. Quando ambos, q_0 e q_1 são 1, q_2 troca de estado na próxima borda.

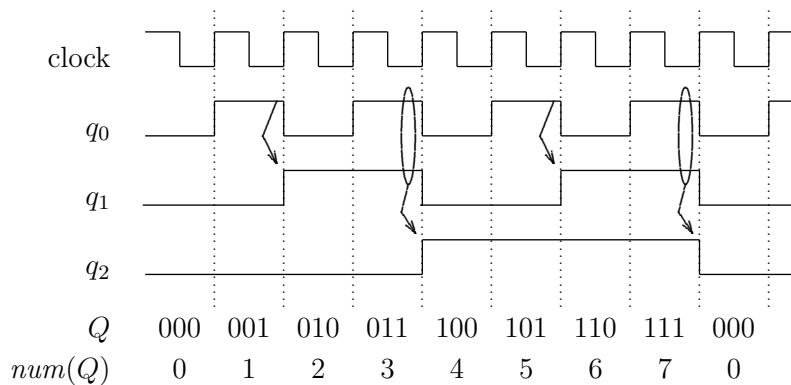


Figura 8.6: Comportamento do contador módulo-8.

A Figura 8.6 mostra também a sequência da contagem como uma sequência de números naturais. A cada borda ascendente do sinal de relógio, o estado da máquina muda para o estado que representa o próximo número na sequência de contagem módulo-8.

8.2.4 Contador Módulo-16

Um contador que conta na sequência de 0 a 15 necessita de $\log_2 16 = 4$ bits para armazenar o estado atual da contagem, e sua função de próximo estado é mostrada na Tabela 8.5.

Tabela 8.5: Sequência de contagem módulo-16.

N	EA				PE				N
	q_3	q_2	q_1	q_0	d_3	d_2	d_1	d_0	
0	0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	0	1	0	2
2	0	0	1	0	0	0	1	1	3
3	0	0	1	1	0	1	0	0	4
4	0	1	0	0	0	1	0	1	5
5	0	1	0	1	0	1	1	0	6
6	0	1	1	0	0	1	1	1	7
7	0	1	1	1	1	0	0	0	8
8	1	0	0	0	1	0	0	1	9
9	1	0	0	1	1	0	1	0	10
10	1	0	1	0	1	0	1	1	11
11	1	0	1	1	1	1	0	0	12
12	1	1	0	0	1	1	0	1	13
13	1	1	0	1	1	1	1	0	14
14	1	1	1	0	1	1	1	1	15
15	1	1	1	1	0	0	0	0	0

As funções d_0 , d_1 e d_2 são aquelas derivadas para os contadores de módulo menor que 16. A função de próximo estado d_3 também pode ser obtida ao inspecionar a tabela ou através de simplificação – quais são as condições para que q_3 troque de estado?

A Equação 8.2 mostra as funções de próximo estado para os quatro bits do contador módulo-16.

$$\begin{aligned}
 d_0 &= q_0 \oplus 1 = \overline{q_0} \\
 d_1 &= q_1 \oplus q_0 \\
 d_2 &= q_2 \oplus (q_1 \wedge q_0) \\
 d_3 &= q_3 \oplus (q_2 \wedge q_1 \wedge q_0)
 \end{aligned} \tag{8.2}$$

A Figura 8.7 mostra a implementação do contador módulo-16. As funções de próximo estado para os bits d_2 e d_3 são parecidas e isso indica uma forma de implementar contadores para qualquer módulo que seja uma potência de dois.

Como uma alternativa ao circuito com portas lógicas, a função de próximo estado poderia ser implementada com uma memória ROM com 16 linhas de 4 bits. O conteúdo da ROM deve ser a coluna de próximo estado da Tabela 8.5, e os quatro bits do estado atual são ligados nas entradas de endereço. O conteúdo da linha indexada é entrada de cada um dos bits do registrador de estado.

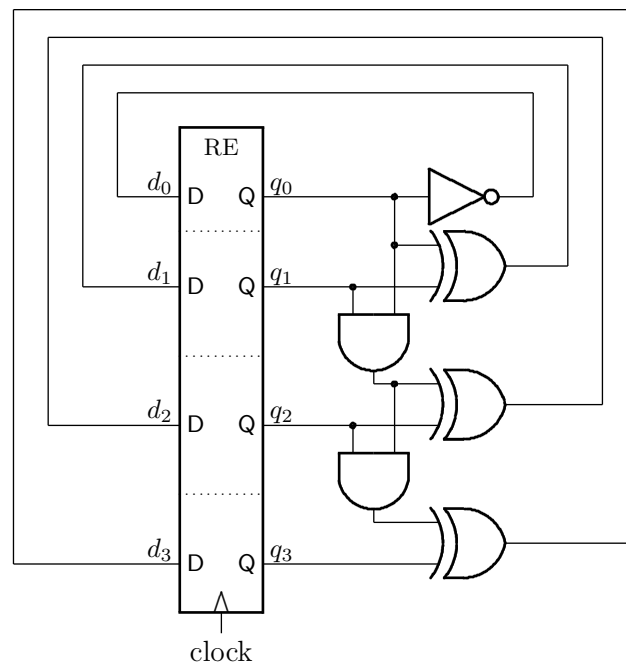


Figura 8.7: Contador módulo-16.

8.2.5 Divisão de Frequência

Um circuito contador pode efetuar a *divisão de frequência* do sinal de relógio. Considerando a frequência f do sinal de relógio, a saída do primeiro *flip-flop* do contador (q_0) produz um sinal com frequência $f/2$, a saída do segundo (q_1) tem frequência $f/4$, e assim sucessivamente. A cada divisão na frequência corresponde uma duplicação no período, como mostra a Figura 8.8.

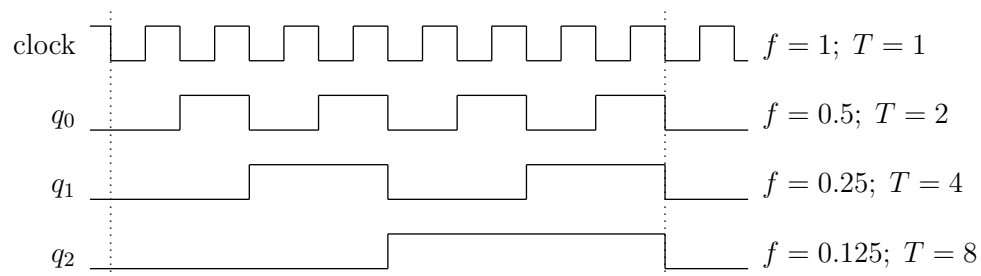


Figura 8.8: Divisão de frequência.

Exemplo 8.1 Vejamos como implementar um contador de três bits que conta na sequência de um Código de Gray, mostrada na Tabela 8.6.

Os Mapas de Karnaugh para as funções d_0 , d_1 , e d_2 são mostrados na Figura 8.9. As funções de próximo estado são definidas na Equação 8.3. O segundo termo de d_1 pode ser aproveitado em d_2 . $\triangleleft$

Tabela 8.6: Sequência de contagem Gray com três bits.

EA			PE		
q_2	q_1	q_0	d_2	d_1	d_0
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	0	1	0
0	1	0	1	1	0
1	1	0	1	1	1
1	1	1	1	0	1
1	0	1	1	0	0
1	0	0	0	0	0

d_0	$q_1 q_0$			
	00	01	11	10
q_2	0	1	3	2
	0	1	0	0
	4	5	7	6
	1	0	1	1

d_1	$q_1 q_0$			
	00	01	11	10
q_2	0	1	3	2
	0	1	1	1
	4	5	7	6
	1	0	0	1

d_2	$q_1 q_0$			
	00	01	11	10
q_2	0	1	3	2
	0	0	0	1
	4	5	7	6
	1	1	1	1

Figura 8.9: Mapas de Karnaugh para o contador Gray de três bits.

$$\begin{aligned}
 d_0 &= \overline{q_2} \wedge \overline{q_1} \vee q_2 \wedge q_1 = q_2 \oplus \overline{q_1} \\
 d_1 &= \overline{q_2} \wedge q_0 \vee q_1 \wedge \overline{q_0} \\
 d_2 &= q_2 \wedge q_0 \vee q_1 \wedge \overline{q_0}
 \end{aligned} \tag{8.3}$$

Os contadores que vimos até aqui são CSS sem entradas, e as saídas são o próprio estado do contador – sua função de saída é a função identidade. Vejamos alguns exemplos de contadores com sinais de controle que aumentam a sua funcionalidade.

Exemplo 8.2 O contador 74163, da família TTL, é um contador módulo-16, síncrono, com entradas de carga (ld) e inicialização (clr) síncronas. Se as entradas p e t estão ativas ($p = t = 1$), e ld e clr estão inativas ($\overline{ld} = \overline{clr} = 1$) então a contagem é incrementada na borda do relógio. Quando a contagem chega em quinze, o sinal rco fica em 1. ‘rco’ é a abreviatura do Inglês para *ripple carry out*, que é o vai-um do incremento da contagem. A Figura 8.10 mostra o símbolo deste contador e a Equação 8.4 define seu comportamento. O operador mod é o resto da divisão inteira. $\triangleleft$

$$\begin{aligned}
 & p, pt, ld, clr, rco : \mathbb{B} \\
 & E, Q : \mathbb{B}^4 \\
 & 163 : (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^4) \mapsto [(\mathbb{B}^4 \mapsto \mathbb{B}^4) \times \mathbb{B}] \\
 & (p \wedge t \wedge ld \wedge clr) \Rightarrow Q \leftarrow num^{-1}[(num(Q) + 1) \bmod 16] \\
 & \quad \overline{clr} \Rightarrow Q \leftarrow 0 \\
 & (p \wedge t \wedge \overline{ld} \wedge clr) \Rightarrow Q \leftarrow E \\
 & (\overline{p} \vee \overline{t}) \wedge ld \wedge clr \Rightarrow Q \leftarrow Q \\
 & num(Q) = 15 \Rightarrow rco = 1
 \end{aligned} \tag{8.4}$$

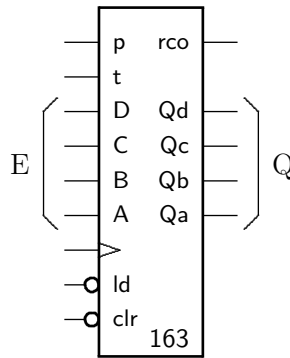


Figura 8.10: Símbolo e sinais do 74163.

Exemplo 8.3 A Figura 8.11 mostra um contador de 12 bits implementado com três 74163, e o diagrama de tempo deste contador indicando as mudanças de estado que causam incremento em mais de um dos contadores é mostrada na Figura 8.12. Note que a saída rco é combinacional e portanto este sinal é um pouco atrasado com relação às mudanças de estado do contador. Quando a contagem do primeiro contador passa pelo estado 15, sua saída rcoA fica em 1 durante o tempo suficiente para habilitar o incremento do segundo contador, de 0 para 1. Após 255 pulsos de relógio, quando o primeiro e o segundo contadores estão ambos em 15, os sinais rcoA e rcoB ficam em 1, habilitando o incremento do terceiro contador.

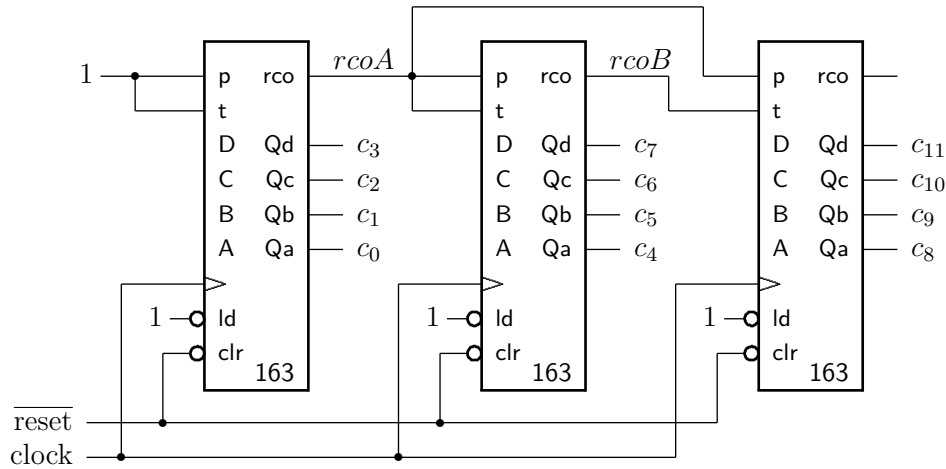


Figura 8.11: Contador síncrono de 12 bits com 74163.

A Figura 8.12 mostra o diagrama de tempo do contador de 12 bits, para as transições na contagem de 15 para 16, e de 255 para 256. Na transição de 15 para 16, o sinal rcoA fica ativo quando a saída do contador menos significativo (c_3, c_2, c_1, c_0) atinge 15 e isso permite que o segundo contador (c_7, c_6, c_5, c_4) seja incrementado. Quando a contagem atinge 255, os sinais rcoA e rcoB estão ambos ativos e isso permite que o contador mais significativo (c_{11}, c_{10}, c_9, c_8) seja incrementado. Note que é o sinal rcoA quem define o instante em que o terceiro contador é incrementado, e que o pulso em rcoA dura aproximadamente um ciclo de relógio.

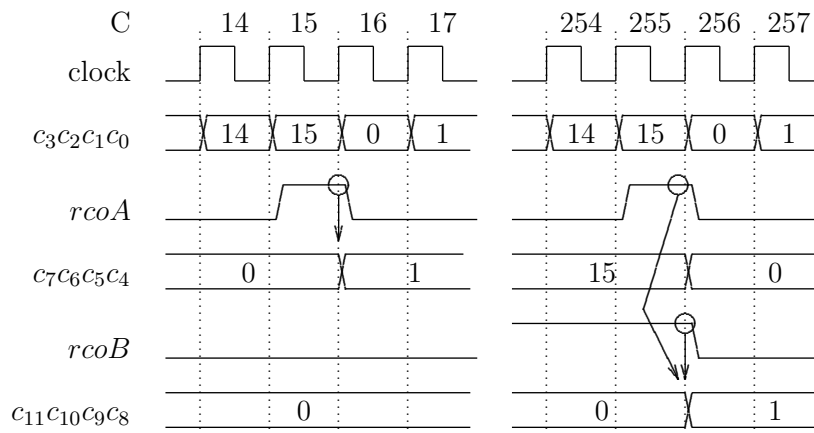


Figura 8.12: Diagrama de tempo do contador de 12 bits com 74163.

Exemplo 8.4 A Figura 8.13 mostra mais duas aplicações do 74163. O circuito (a) conta na sequência entre 5 e 15: quando a contagem chega em 15, o sinal rco faz com que a tupla que representa o número 5, ligada às entradas D-A, seja carregado sincronicamente. Nos ciclos subsequentes do relógio a contagem prossegue de 6 até 15. O sinal reset inicializa o contador em 5 para garantir que a primeira sequência seja correta. O circuito (b) é um contador módulo-13: toda vez que a contagem chegar em 12, a saída da porta and reinicializa a sequência sincronicamente. O desenho da porta foi ajustado para acomodar as quatro entradas, duas delas com inversões.

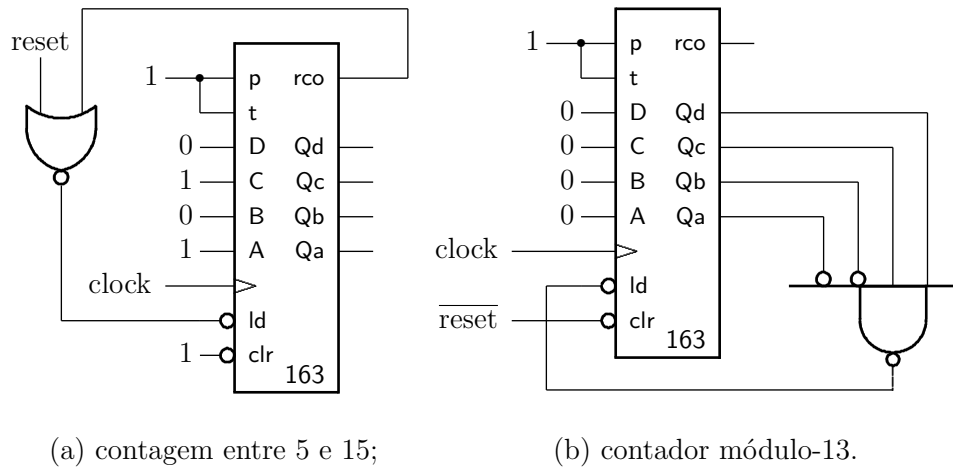


Figura 8.13: Aplicações do 74163.

8.2.6 Contador Assíncrono – *Ripple Counter*

Se observarmos o diagrama de tempos da Figura 8.6, veremos que o estado do *flip-flop* q_{n+1} se inverte quando há uma transição de 1 para 0 no *flip-flop* q_n . Podemos tirar proveito deste fato para simplificar o projeto do contador módulo-8, e para tanto necessitamos de *flip-flops* com as saídas q e $\bar{q}$. Tal circuito é mostrado na Figura 8.14, e emprega três *flip-flops* tipo D. A saída complementada é ligada à entrada de relógio do próximo *flip-flop* porque, quando há uma borda de descida em q_n , há uma borda de subida em $\bar{q}_n$.

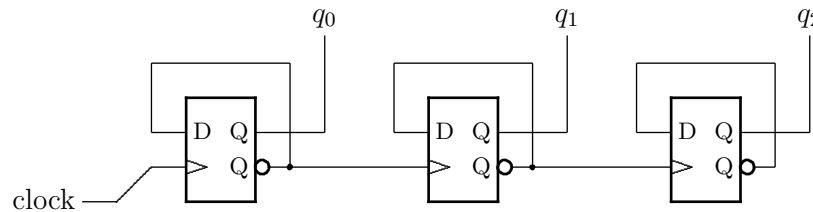


Figura 8.14: *Ripple counter* implementado com FFs tipo D.

O valor de contagem C para um contador módulo-8 é obtido do estado dos *flip-flops*, e é expresso na Equação 8.5. Nesta equação os valores dos bits q_0 , q_1 e q_2 são interpretados como números naturais 0 ou 1. A Equação 8.5 indica que C não é exatamente a sequência desejada; as razões para tal são discutidas adiante.

$$\begin{aligned}
Q &: \mathbb{B}^3 \\
C &: \mathbb{N} \\
\text{ripple} &: \mathbb{B}^3 \mapsto \mathbb{B}^3 \\
C = \text{num}(Q) &\approx q_2 \cdot 2^2 + q_1 \cdot 2^1 + q_0 \cdot 2^0
\end{aligned} \tag{8.5}$$

Este contador é chamado de *ripple* porque os sinais de relógio dos vários *flip-flops* se propagam como uma onda, da saída $\bar{q}$ de um *flip-flop* para a entrada de relógio do próximo *flip-flop*.

As mudanças de estado nos *flip-flops* ocorrem na borda ascendente do sinal de relógio, e a sequência de contagem é mostrada na Figura 8.15. Suponha que inicialmente os três *flip-flops* estão com a saída $\langle q_2, q_1, q_0 \rangle = 000$ e portanto $C = 0$. Na primeira borda ascendente do sinal de relógio, o *flip-flop* q_0 troca de estado e $C = 1$ (001). Na primeira borda descendente em q_0 ($\bar{q}_0 \nearrow$) o *flip-flop* q_1 troca de estado e $C = 2$ (010). Quando a contagem atinge sete, os três *flip-flops* trocam de estado e a sequência reinicia de zero. Este o comportamento esperado de um contador módulo-8.

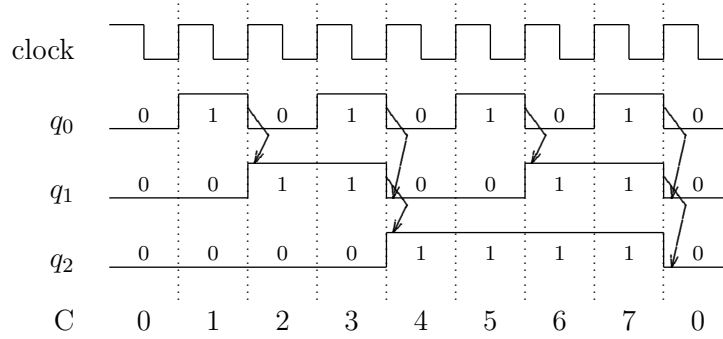


Figura 8.15: Comportamento idealizado do *ripple counter*.

Os sinais não se propagam instantaneamente através de *flip-flops* e isso implica em que a sequência de contagem deste contador não é exatamente aquela mostrada na Figura 8.15. Dependendo do período do sinal de relógio, é mais provável que a sequência de estados se pareça com a mostrada na Figura 8.16. Neste diagrama, a escala de tempo é tal que o tempo de propagação dos *flip-flops* é próximo da duração de um ciclo do relógio.

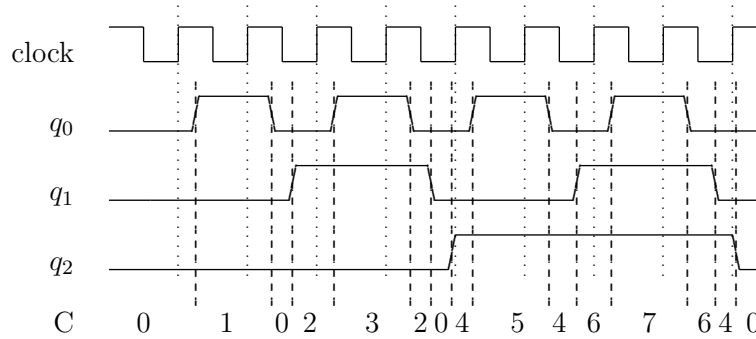


Figura 8.16: Sequência de contagem do *ripple counter*.

Este contador é dito de *assíncrono* porque a saída de um *flip-flop* é usada como sinal de relógio do *flip-flop* seguinte, e o tempo de propagação dos sinais faz com que os *flip-flops* mudem de estado em instantes diferentes, e estas mudanças de estado se propaguem feito uma onda

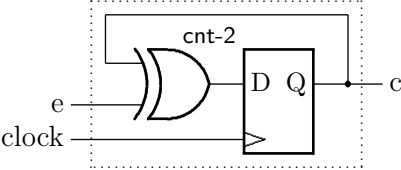
(*ripple*). Tal comportamento é altamente indesejável e frequentemente perigoso, dependendo da aplicação². Existem técnicas de projeto que permitem tolerar a assincronia entre sinais, mas o projeto de circuitos assíncronos está fora do escopo deste texto.

É impossível garantir a perfeita sincronia nos eventos num circuito que não seja trivial por causa dos tempos de propagação dos diversos tipos de portas lógicas e da propagação de sinais através de fios com comprimentos diferentes. Contudo, as técnicas de projeto de circuitos síncronos reduzem os problemas potenciais causados pela assincronia porque tais técnicas garantem que todas as trocas de estado ocorrem durante janelas de tempo muito estreitas, e na vizinhança das transições no sinal de relógio. Se as condições definidas nas Seções 7.5.1 e 7.5.2 para *setup*, *hold* e *skew* forem respeitadas, então circuitos sequencias síncronos podem ser projetados com relativa facilidade por causa da abstração de tempo discretizado.

8.2.7 Contadores Síncronos

Os contadores vistos nas Seções 8.2.1 a 8.2.4 são ditos *síncronos* porque as transições em todos os *flip-flops* são disparadas por eventos simultâneos no sinal de relógio, que são as bordas ascendentes. Ao contrário do especificação na Equação 8.5, para estes contadores, a igualdade é válida e portanto o ‘ $\approx$ ’ deve ser substituído por um ‘=’, quando se emprega a abstração para tempo discreto.

Exemplo 8.5 Vejamos como implementar um contador síncrono de 5 bits, projetado para operar na máxima frequência possível. Você dispõe de um contador módulo-2 (cnt-2) e das portas lógicas especificadas na Figura 8.17. Os tempos estão em nanossegundos, e são dados os tempos de propagação (T_{prop}), de contaminação (T_{cont}), de estabilização (T_{su}) e de manutenção (T_h). O circuito completo é mostrado na Figura 8.18. O bit q_{i+1} inverte quando $\langle q_i..q_0 \rangle$ são todos 1, o que é garantido pela conjunção $q_i \wedge \dots \wedge q_0$. No circuito da Figura 8.18, emprega-se a associatividade do and para computar a conjunção das saídas dos cnt-2 menos significativos. ◀



	T_{prop}	T_{cont}	T_{su}	T_h
FF	6	2	4	2
xor-2	3	1		
and-2	3	1		
and-3	4	2		
and-4	5	3		

Figura 8.17: Especificação dos componentes do contador de 5 bits.

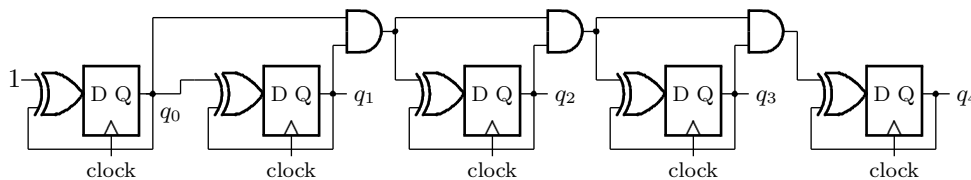


Figura 8.18: Primeira versão do contador de 5 bits.

²O Ministério dos Projetos adverte: circuitos assíncronos fazem mal à saúde e à nota.

Exemplo 8.6 Iniciemos pelo comportamento temporal do contador de 1 bit da Figura 8.17. O circuito combinacional é somente a porta xor; o período mínimo e a restrição de hold são:

$$\text{cnt-2} : T_{min} = T_{FF} + T_x + T_{su} = 6 + 3 + 4 = 13 \text{ ns},$$

$$\text{cnt-2} : T_{C,F} + T_{C,x} = 2 + 1 = 3 > 2 = T_h,$$

e portanto este circuito opera apropriadamente. $\triangleleft$

Exemplo 8.7 Vejamos agora o comportamento temporal do contador de 5 bits da Figura 8.18. O caminho combinacional mais longo é aquele através das três portas and e mais um xor, da saída q_0 até a entrada d_4 . Esse caminho atravessa 3 portas and e o xor na entrada d_4 .

$$\text{cnt-32 v1} : T_{min} = T_{FF} + 3T_a + T_x + T_{su} = 6 + 3 \times 3 + 3 + 4 = 22 \text{ ns}.$$

O tempo de manutenção de cnt-32 é o mesmo de cnt-2 porque o caminho de contaminação mais curto é o laço de realimentação $q_i \rightsquigarrow \text{xor} \rightsquigarrow d_i$. $\triangleleft$

Exemplo 8.8 Se trocarmos a cadeia de ands, por portas com maior número de entradas, mas com tempo de propagação menor do que a cadeia de ands, o período mínimo pode ser reduzido. Este circuito é mostrado na Figura 8.19.

O período mínimo de relógio é determinado pelo caminho mais longo, e este é na entrada do flip-flop Q_4 por causa da porta and-4.

$$\text{cnt-32 v2} : T_{min} = T_{FF} + T_{a4} + T_x + T_{su} = 6 + 5 + 3 + 4 = 18 \text{ ns}.$$

A segunda versão é $22\text{ns}/18\text{ns} = 1,22$ vezes mais rápida do que a primeira. $\triangleleft$

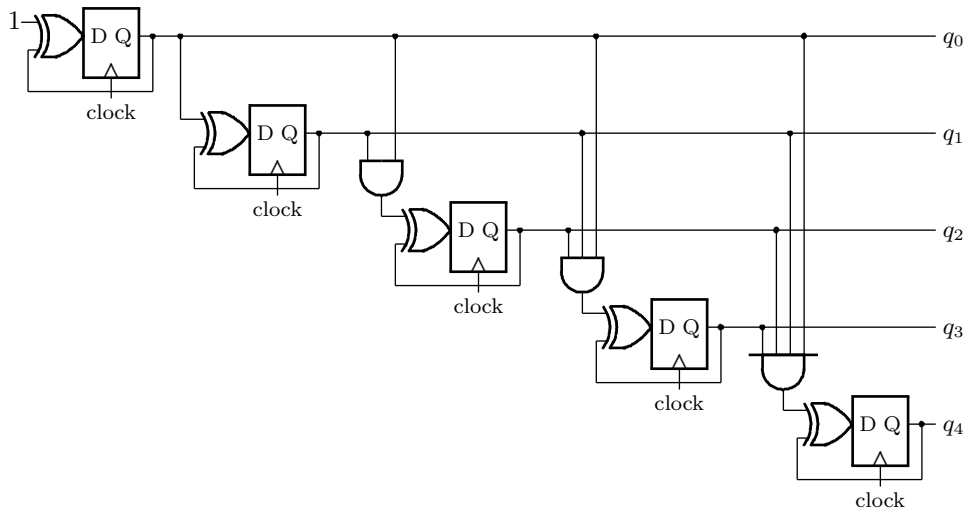


Figura 8.19: Segunda versão do contador de 5 bits.

Exercícios

Ex. 8.1 Qual seria a sequência de contagem do contador da Figura 8.14 se os FFs forem sensíveis à borda ascendente do relógio?

Ex. 8.2 O que se pode dizer quanto ao comportamento temporal de um contador *ripple* com 16 FFs? E quanto a um contador com 64 FFs?

Ex. 8.3 Re-compute o período mínimo de relógio para os circuitos dos Exemplos 8.5 e 8.8 considerando que o *skew* de pior caso é -3ns . Defina com cuidado o que seja o *skew de pior caso*. Qual dos dois circuitos é o mais rápido com *skew* negativo?

Ex. 8.4 O que se pode dizer quanto ao comportamento temporal de um contador síncrono com 16 FFs? Idem para 64 FFs?

Ex. 8.5 Especifique um contador de três bits que incrementa, ou decrementa, dependendo do estado da entrada Id : se $Id = 1$, a contagem incrementa (módulo-8); se $Id = 0$, a contagem decrementa (módulo-8).

Ex. 8.6 Implemente a especificação do Exercício 8.5, que é o circuito *inc-dec*.

Ex. 8.7 Estenda a especificação do Exercício 8.5 para 4 bits e a implemente.

8.2.8 Contador em Anel

Um *contador em anel* se comporta como um contador de N bits cujo conjunto de estados é menor que 2^N estados. Um contador em anel pode ser implementado como aquele mostrado na Figura 8.20. A saída de cada um dos FFs é ligada à entrada do FF seguinte, de forma tal que os valores iniciais dos FFs circulem pelo anel. A sequência de contagem depende do valor inicial dos FFs, e a contagem nunca se alteraria nos casos triviais com $\langle q_3q_2q_1q_0 \rangle = \{0000, 1111\}$. O circuito da Figura 8.20 é inicializado em $\langle q_3q_2q_1q_0 \rangle = 0001$.

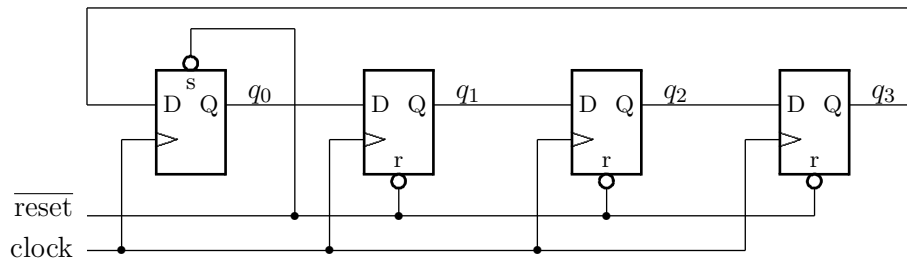


Figura 8.20: Contador em anel.

O contador em anel da Figura 8.20 tem $N=4$ bits e possui um conjunto de estados E com $n < 2^N$ estados, que são $E = \{0001, 0010, 0100, 1000\}$, e a sequência de contagem é $1, 2, 4, 8, 1, 2, 4, \dots$, como mostra a Figura 8.21. No diagrama, os q_i são mostrados como ϕ_i , que significa *fase- i* , e cuja utilidade descobriremos em breve.

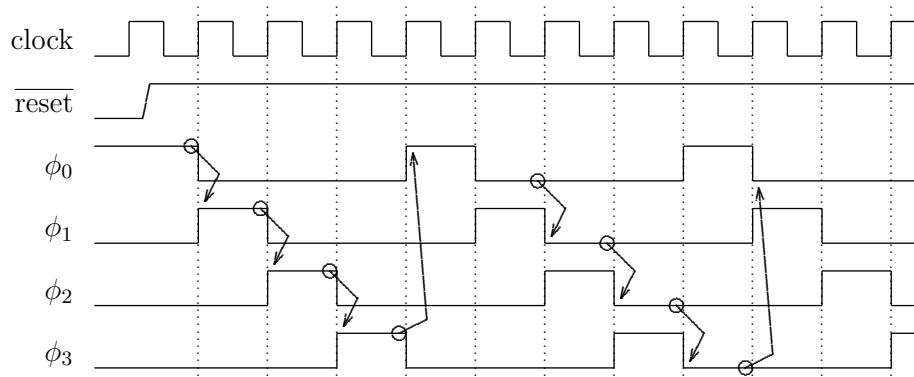


Figura 8.21: Comportamento do contador em anel.

8.2.9 Ciclo de Trabalho

A saída de um dos FFs do contador em anel produz um sinal periódico *assimétrico* porque os intervalos em que o sinal permanece em 1 e em 0 são diferentes. Por exemplo, o diagrama de tempo mostrado na Figura 8.21 mostra que os sinais ϕ_i permanecem em 1 durante 1/4 do período e em 0 por 3/4 do período. O *ciclo de trabalho* (*duty cycle*) de um sinal periódico é a relação entre os intervalos em que o sinal permanece em 1 e o período – tempo em 1 mais o tempo em 0. A Figura 8.22 mostra sinais com ciclos de trabalho de 25, 50 e 75%.

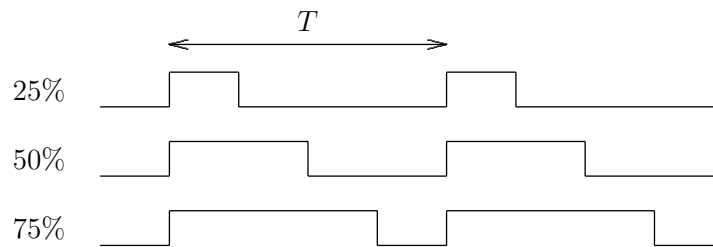


Figura 8.22: Ciclo de trabalho.

8.2.10 Relógio Multi-fase

Uma das aplicações para contadores em anel é a geração de sinal de *relógio multi-fase* de tal forma que cada um dos FFs gera uma dentre as quatro fases do sinal de relógio. Geralmente, as fases são chamadas de $\phi_0, \phi_1, \phi_2, \phi_3$, e a cada instante somente uma das quatro fases está ativa, como mostra a Figura 8.21.

Suponha que seja necessário reduzir o custo de um circuito, e para tal são empregados básicos simples, ao invés de *flip-flops* mestre-escravo. A Figura 8.23 mostra um circuito com três blocos combinacionais separados por quatro registradores compostos por básicos simples. Esta implementação é problemática porque, quando o sinal de relógio estiver em 1, o sinal d_1 contamina todo o circuito pois os quatro básicos ficam transparentes.

Se somente um par de saídas do contador em anel for usado para gerar os sinais de relógio dos básicos, *viz.* $\langle \phi_0, \phi_2 \rangle$, o comportamento do circuito é aquele mostrado na Figura 8.24.

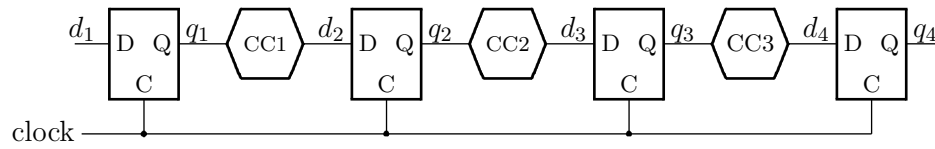


Figura 8.23: Basculos simples com contaminação de todas as entradas.

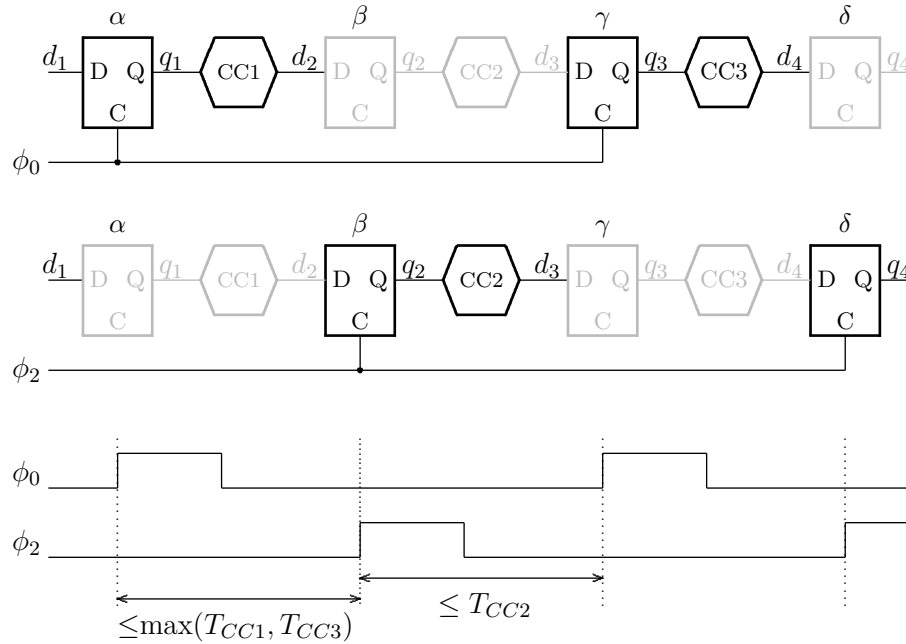


Figura 8.24: Relógio com duas fases elimina a contaminação.

Quando ϕ_0 está ativo, os b́asculos α e γ ficam transparentes, enquanto que os b́asculos β e δ mantêm seus valores q_2 e q_4 inalterados. Quando ϕ_2 está ativo, a situação se inverte e os b́asculos β e δ ficam transparentes, enquanto que α e γ mantêm seus valores. Os b́asculos que mantêm seus valores são mostrados em cinza porque estes, por estar inativos, impedem a contaminação de suas saídas pelos sinais dos b́asculos adjacentes ligados as suas entradas.

A Figura 8.24 indica que os intervalos entre ϕ_0 e ϕ_2 ativos devem ser mais longos que os tempos de propagação dos circuitos combinacionais, mais a propagação através dos b́asculos fonte e o *setup* dos b́asculos destino. No diagrama estão indicados somente os tempos de propagação dos circuitos combinacionais.

Se o tempo de propagação dos circuitos combinacionais entre os b́asculos for mais longo do que o intervalo nos quais os sinais de relógio ficam inativos, – ϕ_1 a ϕ_3 na Figura 8.21 – estes intervalos inativos podem ser alongados. Para tanto basta acrescentar *flip-flops* ao contador em anel para afastar as bordas entre as fases ativas dos sinais de relógio.

Relógios multi-fase são usados em circuitos sequenciais implementados com b́asculos simples para garantir que seu estado seja alterado somente após os sinais nas suas entradas estabilizarem. Normalmente, uma fase é usada para a alteração do estado dos b́asculos (ϕ_i), e a fase seguinte (ϕ_{i+1}) é usada para garantir que os sinais se propaguem através da parte combinacional do circuito e estabilize imediatamente antes da próxima fase de alteração de estado (ϕ_{i+2}).

8.2.11 Contador Johnson

Um *contador Johnson* é similar a um contador em anel exceto que a saída do último FF é complementada. Um contador Johnson de N bits possui um conjunto de estados E cujo tamanho é $|E| = 2N$ estados. A Figura 8.25 mostra o diagrama de tempos de um contador Johnson com 4 FFs.

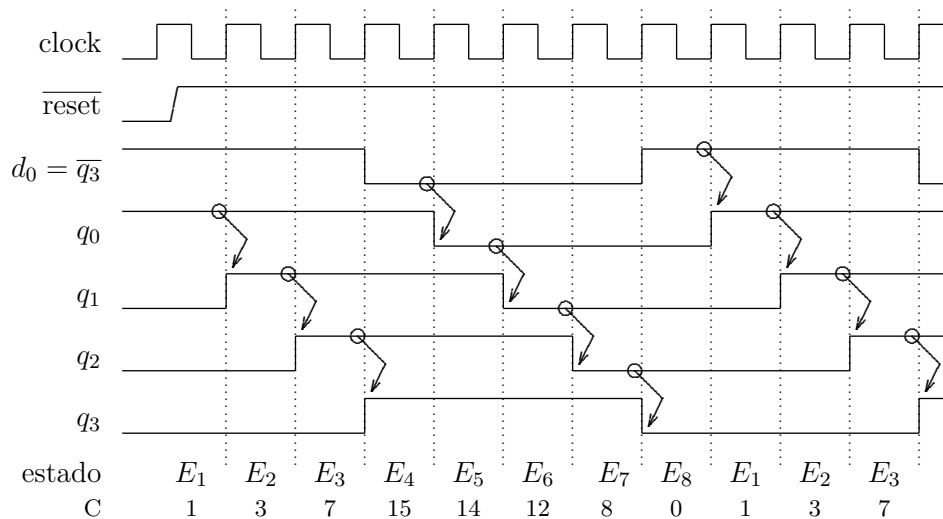


Figura 8.25: Comportamento do contador Johnson.

Note que na sequência de contagem de um contador Johnson apenas uma das saídas se altera a cada ciclo, o que torna esta sequência de estados um código de Gray. Veja o Exercício 8.20 para outra implementação de um contador Gray.

8.3 Registradores de Deslocamento

Um *registrador de deslocamento* permite deslocar seu conteúdo de tal forma que, a cada borda do relógio, todos os bits armazenados nos *flip-flops* deslocam-se simultaneamente de uma posição. Os registradores de deslocamento básicos são de dois tipos: carga em série e leitura em paralelo, ou carga em paralelo e leitura em série.

8.3.1 Registrador de Deslocamento Série-Paralelo

A Figura 8.26 mostra um *registrador de deslocamento série-paralelo* que é carregado com um bit através da entrada d_3 a cada ciclo do relógio. As saídas $\langle q_3, q_2, q_1, q_0 \rangle$ de todos os *flip-flops* podem ser observadas ao mesmo tempo.

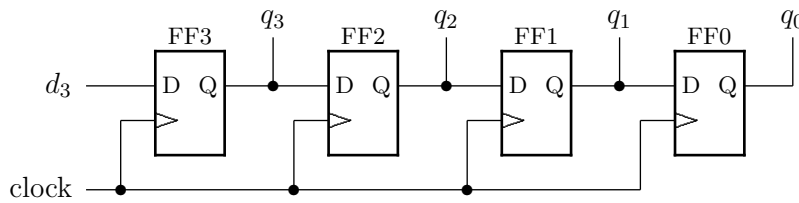


Figura 8.26: Registrador de deslocamento série-paralelo.

Na borda ascendente do relógio a entrada de cada *flip-flop* é copiada para sua saída, deslocando assim o conteúdo do registrador de uma posição. A Tabela 8.7 mostra uma sequência de estados deste registrador. Inicialmente, os *flip-flops* $\langle q_3, q_2, q_1, q_0 \rangle$ contêm os bits $\langle x, y, z, w \rangle$, e à entrada d_3 são apresentados os bits $\langle a, b, c, d \rangle$, um a cada ciclo. Após 4 ciclos, os *flip-flops* estão no estado $\langle q_3, q_2, q_1, q_0 \rangle = \langle d, c, b, a \rangle$.

Tabela 8.7: Sequência de deslocamento de estados.

estado	d_3	q_3	q_2	q_1	q_0
0	—	x	y	z	w
1	a	a	x	y	z
2	b	b	a	x	y
3	c	c	b	a	x
4	d	d	c	b	a

O registrador da Figura 8.26 é chamado de *conversor série-paralelo* porque este circuito converte entradas que são amostradas em série – um bit a cada ciclo – em saída que é apresentada em paralelo – todos os bits ao mesmo tempo.

O diagrama da Figura 8.26 identifica cada um dos *flip-flops*, de FF3 a FF0. Esta convenção é usada ao longo do texto, e para simplificar os diagramas, o nome do *flip-flop* é omitido, mas o seu número/nome é o mesmo do índice dos sinais ligados à saída Q. Assim, a saída do FF3 é chamada de q_3 .

8.3.2 Registrador de Deslocamento Paralelo-Série

Todos os *flip-flops* de um *registrador de deslocamento paralelo-série* podem ser carregados simultaneamente, para que então seus conteúdos sejam deslocados, um bit a cada ciclo de relógio. Os quatro *flip-flops* do circuito da Figura 8.27 são carregados em paralelo se o sinal $carga=1$ numa borda ascendente do relógio. Do contrário, $carga=0$, os bits armazenados nos *flip-flops* são deslocados a cada borda do sinal de relógio.

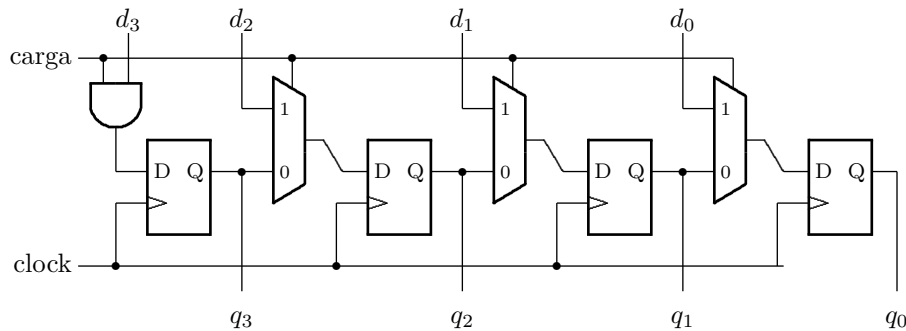


Figura 8.27: Registrador de deslocamento paralelo-série.

Um registrador que permite a carga dos valores nas entradas em paralelo, e apresente na saída um bit de cada vez, é chamado de *conversor paralelo-série*.

Exemplo 8.9 Considere o registrador de deslocamento da Figura 8.27, sendo as entradas carregadas com $\langle d_3, d_2, d_1, d_0 \rangle = \langle 1, 0, 1, 0 \rangle$. O diagrama de tempos da Figura 8.28 mostra a sequência de estados nas saídas do registrador.

No ciclo α , os quatro *flip-flops* são carregados com os valores em $d_{3..0}$, e no ciclo β os deslocamentos se iniciam. Por causa da porta and, a partir do ciclo β , $d_3 = 0$ e este 0 se propaga ao longo do registrador, um bit a cada borda do relógio. No ciclo δ , o bit inicialmente carregado no *flip-flop* mais à esquerda é transferido para q_0 , e a partir de então, a saída q_0 é sempre 0. $\triangleleft$

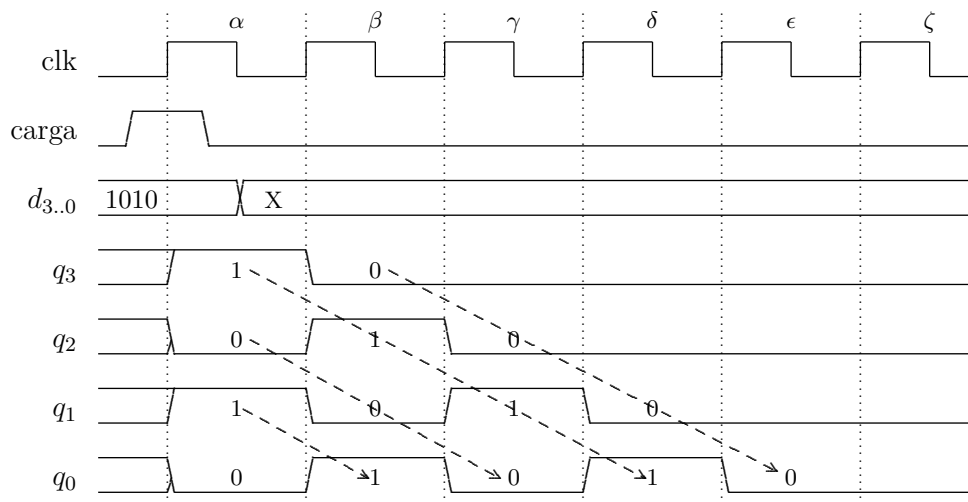


Figura 8.28: Diagrama de tempos do registrador paralelo-série.

A Figura 8.29 mostra os símbolos dos dois tipos de conversores, um registrador série-paralelo e um registrador paralelo-série.

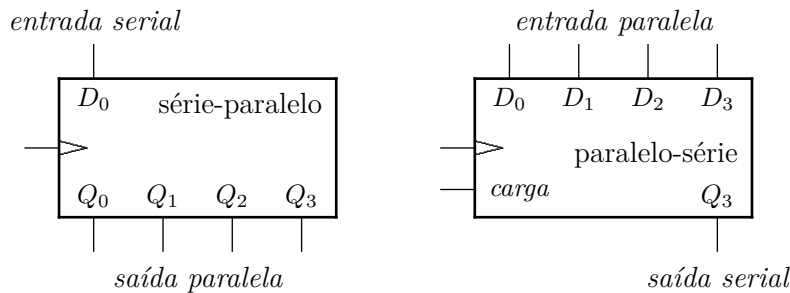


Figura 8.29: Conversores série-paralelo e paralelo-série.

Exemplo 8.10 Vejamos uma primeira versão, assaz primitiva, de um circuito de transmissão serial. Este circuito recebe uma sequência de quatro bits, os repassa sem alteração da entrada para a saída, e então insere um quinto bit, este com a paridade do quarteto recém enviado.

O valor do bit de “paridade par” para 4 bits deve ser tal que o número de bits em 1, considerando os 4 bits de dados mais o bit de paridade, seja par. Para os bits 0000, o bit de paridade par é 0. Para 1111, o bit de paridade também é 0. Para 0001, o bit de paridade é 1, pois o bit de dados em 1 mais o bit de paridade em 1, garantem que o número de bits em 1 seja par.

A solução emprega um flip-flop T para computar a paridade, e é mostrada na Figura 8.30. A cada bit transmitido, a paridade é computada no flip-flop T. No intervalo correspondente ao quinto bit (sel=1), o valor armazenado no flip-flop T é a paridade dos quatro bits anteriores. Quando se inicia um novo quarteto, o flip-flop T deve ser reinicializado em 0, fazendo-se novo=1.

O diagrama de tempo da operação do circuito é mostrado na Figura 8.31. As quatro entradas são copiadas da entrada para a saída, e então o bit de paridade π é inserido na sequência. ◁

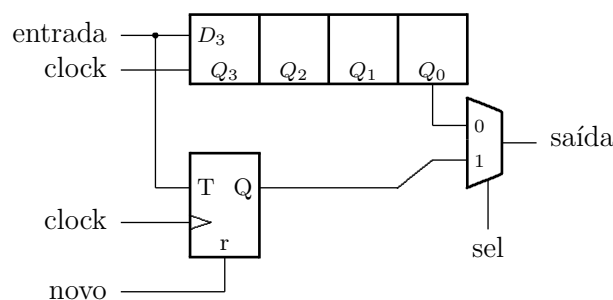


Figura 8.30: Paridade ‘sequencial’ para 4 bits.

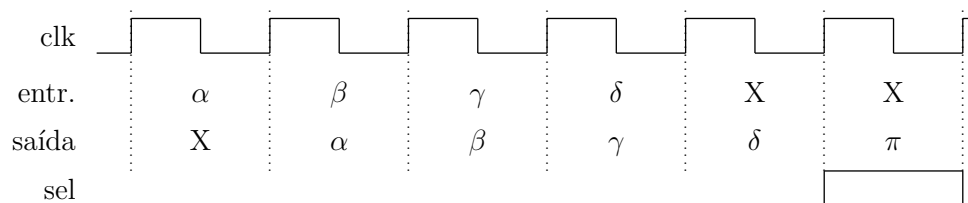


Figura 8.31: Temporização do circuito da paridade para 4 bits.

8.3.3 Registrador de Deslocamento Universal

Um *registrador de deslocamento universal* é um registrador de deslocamento que efetua as duas formas de conversão (série-paralelo e paralelo-série), e permite deslocamentos para a esquerda e para a direita, carga paralela, e manutenção dos bits armazenados. O comportamento deste registrador é especificado na Tabela 8.8, e a Figura 8.32 mostra uma parte do seu circuito.

Tabela 8.8: Especificação do registrador de deslocamento universal.

	$s_1 s_0$	d_3	d_2	d_1	d_0	q_3	q_2	q_1	q_0
carga	1 1	a	b	c	d	a	b	c	d
$\gg 1$	1 0	v	X	X	X	v	q_3	q_2	q_1
$\ll 1$	0 1	X	X	X	v	q_2	q_1	q_0	v
mantém	0 0	X	X	X	X	q_3	q_2	q_1	q_0

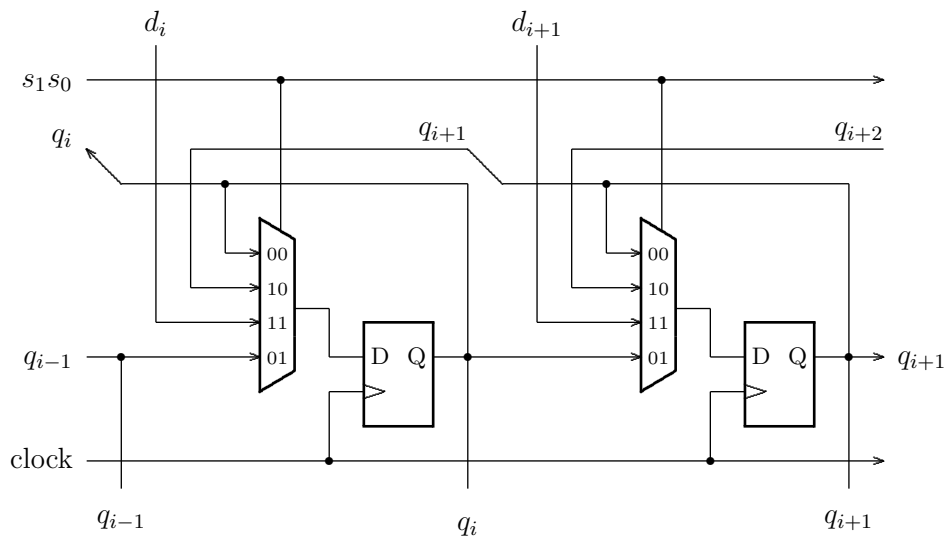


Figura 8.32: Registrador de deslocamento universal.

8.3.4 Somador Serial

Existem aplicações nas quais consumo de energia reduzido é mais importante do que velocidade de execução. Nestas aplicações pode ser conveniente empregar circuitos que operam serialmente, ao invés de paralelamente. Por exemplo, um somador serial para números inteiros representados em n bits pode ser implementado com três registradores de deslocamento para conter os operandos e o resultado, e um somador completo para efetuar a soma de um par de bits a cada ciclo do relógio, como mostrado na Figura 8.33. O circuito opera de forma similar ao cálculo manual da soma de dois números com muitos dígitos: efetua-se a soma da direita para a esquerda, considerando-se um par de dígitos mais o vem-um do par anterior, e lembrando do vai-um para o próximo par de dígitos. Este algoritmo está formalizado na Equação 8.6, na qual são computados os dois bits $\langle c_i s_i \rangle$ que resultam da soma dos dois operandos (a_i, b_i) e do vem-um (c_{i-1}).

$$\begin{aligned} c_{-1} &= 0 \\ c_i s_i &= a_i + b_i + c_{i-1} \end{aligned} \quad (8.6)$$

O bloco que efetua a soma dos três bits contém um somador completo e um *flip-flop*, para armazenar o valor do vai-um para a soma do próximo par de bits.

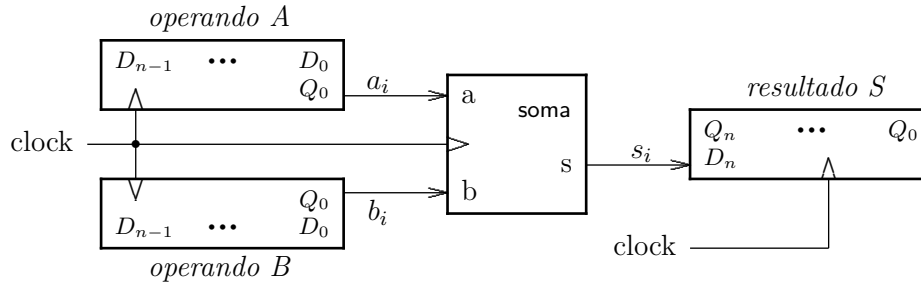


Figura 8.33: Somador serial.

Exercícios

Ex. 8.8 Complemente o diagrama de blocos da Figura 8.33, e explicita como devem ser inicializados o registradores com os operandos, o de resultado, e o *flip-flop* do vai-um.

Ex. 8.9 Tomando o circuito mostrado na Figura 8.18 como base, projete um contador de 32 bits. Quantas portas lógicas *and* são ligadas em série num contador com N bits?

Ex. 8.10 Escreva uma expressão para o limite de velocidade do circuito do Ex. 8.9 como uma função do número de bits do contador.

Ex. 8.11 Usando uma técnica similar a do somador com adiantamento de vai-um (*carry look-ahead*), modifique o contador de 32 bits do Ex. 8.10 para melhorar seu desempenho e reescreva a equação do limite de velocidade para refletir a modificação. Qual o ganho em desempenho? *Pista:* veja a Seção 6.6.

Ex. 8.12 O fabricante do registrador 74374, com 8 FFs tipo D, informa que o tempo de propagação médio é de 20ns, e o máximo de 30ns, o *setup time* é no mínimo 20ns, e o *hold time* de 3ns. Suponha que este registrador é usado em um circuito no qual o tempo de propagação da parte combinacional é de 130ns no caminho crítico. Qual a frequência máxima do relógio para este circuito?

Ex. 8.13 Mostre como um contador em anel (Seção 8.2.8) pode ser usado para reduzir o impacto na frequência do relógio de um circuito com um componente combinacional cujo tempo de propagação é muito mais longo do que qualquer outra parte do circuito.

Ex. 8.14 Adapte a Equação 7.3 para uso num circuito com relógio de duas fases, gerado por um contador em anel, como aquele mostrado na Figura 8.21.

8.4 Máquinas de Estados Finitas

Um *algoritmo* pode ser definido como uma sequência de operações que são aplicadas sobre um conjunto de dados para produzir um determinado resultado. No nosso contexto, uma das possíveis implementações de um algoritmo consiste de: (i) um certo número de circuitos combinacionais que efetuem as operações requeridas; (ii) registradores (ou memória) para manter os resultados intermediários; e (iii) um circuito de controle para gerenciar o fluxo de dados através dos circuitos combinacionais e registradores. Veremos adiante que o circuito de controle é geralmente implementado como uma *máquina de estados finita*.

As seções anteriores tratam de contadores e registradores de deslocamento, que são exemplos relativamente simples de máquinas de estados finitas. Esta seção formaliza a definição destas máquinas e discute técnicas de projeto que se aplicam a uma classe mais ampla de circuitos sequenciais.

Formalmente, uma *máquina de estados finita* M é definida pela sêxtupla da Equação 8.7. Q é um conjunto finito de estados, q_0 é o estado inicial pós-*reset*, I é um conjunto finito de símbolos de entrada, f é a função de transição ou *função de próximo estado*, S é um conjunto finito de símbolos de saída, e g é a função de saída.

$$\begin{array}{ll}
 Q : \mathcal{Q} & \text{conjunto de estados, de tipo } \mathcal{Q} \\
 I : \mathcal{I} & \text{alfabeto de entrada, de tipo } \mathcal{I} \\
 S : \mathcal{S} & \text{alfabeto de saída, de tipo } \mathcal{S} \\
 q_0 \in Q & \text{estado inicial} \\
 f : (Q \times \mathcal{I}) \mapsto Q & \text{função de próximo estado} \\
 g : (Q \times \mathcal{I}) \mapsto S & \text{função de saída}
 \end{array} \tag{8.7}$$

$$M = (Q, I, f, q_0, g, S)$$

A entrada para uma máquina de estados finita é uma sequência de símbolos em I e quando ocorre uma mudança de estado o símbolo associado àquela transição é consumido da sequência de entradas.

Exemplo 8.11 A especificação do contador módulo-4 pode ser formalizada pela máquina de estados definida na Equação 8.8.

$$\begin{array}{ll}
 Q = \{00, 01, 10, 11\} \\
 I = \emptyset \\
 f = \{(00, 01), (01, 10), (10, 11), (11, 00)\} \\
 q_0 = 00 \\
 g = \text{identidade} \\
 S = Q
 \end{array} \tag{8.8}$$

$$M_{\text{mod-4}} = (Q, \emptyset, f, 00, =, Q)$$

O conjunto de quatro estados é representado em binário, o conjunto de entrada é vazio porque as transições ocorrem independentemente de qualquer sinal externo, a função de próximo estado é definida como um conjunto de pares ordenados, o estado inicial é o estado 00, a função de saída é a identidade porque a saída do contador é o próprio estado, e o conjunto de saída é o conjunto de estados. $\triangleleft$

8.4.1 Diagrama de Estados

Usamos *diagramas de estado* para representar os estados de uma máquina de estado (ME), as transições entre os estados em função das entradas, e as saídas produzidas pela ME.

Um estado é representado por um círculo, e as transições de estado por setas, e a cada seta é associada à condição que possibilita a mudança de estado. O nome do estado é indicado na parte superior do círculo, e a saída produzida no estado é indicada na parte inferior. O estado inicial é indicado por uma seta que não provém de nenhum outro estado, e este é o estado para o qual a máquina vai quando o sinal de *reset* é ativado. A Figura 8.34 mostra estas convenções.

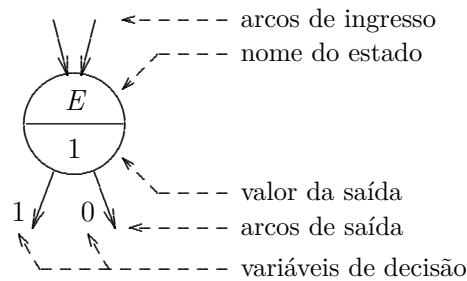


Figura 8.34: Convenção para desenhar diagramas de estados.

Exemplo 8.12 O comportamento de um contador módulo-4 pode ser descrito pela sequência de estados que é percorrida ao longo do tempo:

$$\text{reset} \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow 1 \dots$$

Este comportamento é representado por um diagrama de estados, como o mostrado na Figura 8.35. No diagrama, não são indicadas as saídas porque o estado dos *flip-flops* é a própria saída. As condições para as transições de estado não estão marcadas porque num contador como este sempre ocorre uma mudança de estado a cada ciclo do relógio. ◁

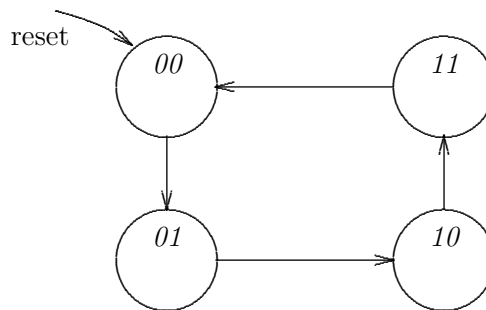


Figura 8.35: Diagrama de estados de um contador módulo-4.

Exemplo 8.13 Considere uma máquina de estados que produz um pulso na sua saída toda vez que sua entrada amostrar a sequência 01110. Dito de outra forma, esta máquina é capaz de reconhecer a sequência 01110. No modelo da Figura 7.24, a entrada e a saída têm largura de um bit ($\lambda = 1, \sigma = 1$).

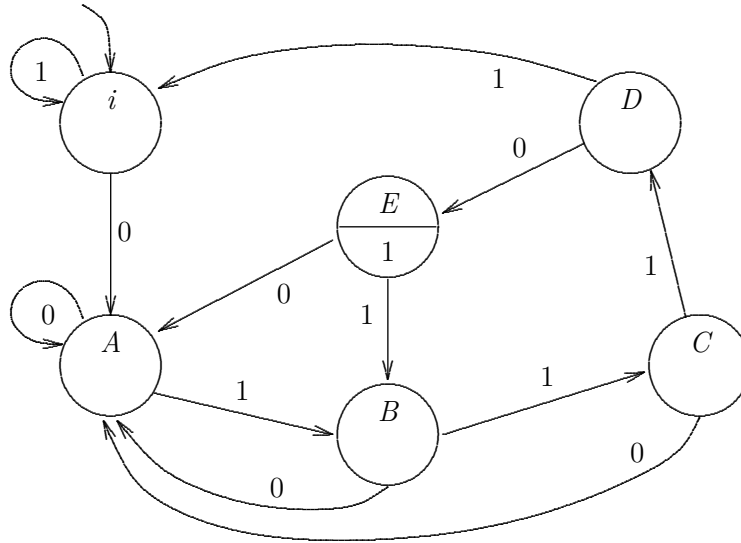


Figura 8.36: Diagrama de estados do reconhecedor de 01110.

As mudanças de estado ocorrem a cada ciclo do relógio, caso a sequência de entradas assim o permita. Como há somente uma entrada binária, os estados apresentam duas possíveis transições, uma para entrada 0 e outra para entrada 1.

Uma decisão de projeto pode determinar que, para a sequência 011101110, a máquina produza dois pulsos em sua saída. A Figura 8.36 mostra o diagrama de estados desta ME.

Após a inicialização, enquanto a entrada for 1, a máquina permanece no estado i . A sequência de estados que produz um pulso, com a duração de um período do relógio, é $i A B C D E$ para entradas 01110. Para entradas 011101110, a sequência de estados é $i A B C D E B C D E$. Para simplificar o diagrama, somente no estado E a saída é indicada como 1, e subentende-se que os demais produzem saída 0. ◁

Exemplo 8.14 O comportamento da ME que detecta a sequência 01110, do Exemplo 8.13, é formalizado na Equação 8.9.

$$\begin{aligned}
 Q &= \{i, A, B, C, D, E\} \\
 I &= \{0, 1\} \\
 f &= \{(i, 0, i), (i, 1, A), (A, 0, A), (A, 1, B), \\
 &\quad (B, 0, A), \dots, (E, 0, A), (E, 1, B)\} \\
 q_0 &= i \\
 g &= \{(i, 0), (A, 0), \dots, (D, 0), (E, 1)\} \\
 S &= I
 \end{aligned} \tag{8.9}$$

$$M_{01110} = (Q, I, f, i, g, I)$$

A função de próximo estado é definida por triplas $\langle \text{estado atual}, \text{entrada}, \text{próximo estado} \rangle$ porque, em todos os estados, a decisão de qual é o próximo estado depende do valor da entrada naquele estado. ◁

Dos dois exemplos de diagramas de estado discutidos até agora, o contador não depende de nenhuma entrada externa porque as transições ocorrem a cada ciclo do relógio, e o reconhecedor da sequência 01110 depende de uma única entrada binária. Em geral, para uma máquina de estado com n entradas, em cada estado podem ocorrer até 2^n transições distintas para próximo(s) estado(s).

No projeto de MEs há três restrições importantes que não podem ser ignoradas:

1. os diagramas de estado devem permitir a implementação de máquinas de estado *determinísticas* e portanto não pode haver mais de um próximo estado para cada um dos possíveis valores das entradas;
2. dentre todas as transições para outros estados, *ao menos uma, e somente uma*, deve ser possível; e
3. todos os estados devem ser o destino de ao menos uma transição.

Para garantir que todos os estados sejam alcançáveis, alguma transição para cada um dos estados deve ser possível, para ao menos uma dentre todas as combinações possíveis das entradas e estados.

Os diagramas de estados são uma representação para *máquinas de estado finitas*, que por sua vez pertencem ao conjunto dos *autômatos finitos*. Estes últimos são estudados na disciplina Teoria da Computação [Koh78, HU79].

Exercícios

Ex. 8.15 Desenhe os diagramas de estados completos de um contador em anel (Seção 8.2.8) e de um contador Johnson (Seção 8.2.11), ambos com 4 FFs.

Ex. 8.16 Formalize o projeto do contador em anel da Seção 8.2.8. Especifique a função de próximo estado através de uma tabela com os campos (i) *estado atual* e (ii) *próximo estado*.

Ex. 8.17 Especifique formalmente o contador Johnson da Seção 8.2.11. Especifique a função de próximo estado através de uma tabela.

Ex. 8.18 Especifique a função de próximo estado do detector de sequências da Figura 8.36 através de uma tabela com três colunas, (i) *estado atual*, (ii) *entrada*, e (iii) *próximo estado*.

Ex. 8.19 Modifique o diagrama de estados do detector de sequências de Figura 8.36 para evitar que a sequência 011101110 produza saída no terceiro 0. O circuito deve reconhecer somente a primeira sequência em **011101110** e ignorar a segunda. A sequência 0111001110 produz dois pulsos na saída.

Ex. 8.20 Projete e implemente um contador Gray, de 3 bits cuja sequência de contagem é aquela definida no Exercício 4.4.

Ex. 8.21 Projete um circuito sequencial síncrono que produz em sua saída a sequência mostrada abaixo. Seu projeto deve empregar quatro FFs tipo D.
0000 $\mapsto$ 0011 $\mapsto$ 0110 $\mapsto$ 1001 $\mapsto$ 1100 $\mapsto$ 1111 $\mapsto$ 0000, e repete.

8.4.2 Máquinas de Mealy e Máquinas de Moore

Dependendo da função de saída, podem ser definidos dois tipos de máquinas de estado finitas, as Máquinas de Moore e as Máquinas de Mealy.

Máquina de Moore

Numa Máquina de Moore a saída depende apenas do estado atual, como definido na Equação 8.10.

$$g : Q \mapsto S \quad \text{Máquina de Moore} \quad (8.10)$$

A saída produzida por uma Máquina de Moore, em resposta a uma sequência de entrada $e_1, e_2 \dots e_n$, $n \geq 0$ é

$$g(q_0), g(q_1) \dots g(q_n)$$

quando $q_0, q_1 \dots q_n$ é a sequência de estados tal que $f(q_{i-1}, e_i) = q_i$, $1 \leq i \leq n$.

A Figura 8.37 mostra um circuito genérico que pode ser usado para implementar Máquinas de Moore. Estas máquinas consistem de um *registrador de estado*, de uma *função de próximo estado*, e de uma *função de saída*. O próximo estado (*PE*) da máquina depende do estado atual (*EA*) e das entradas (*E*): $PE = f(E, EA)$.

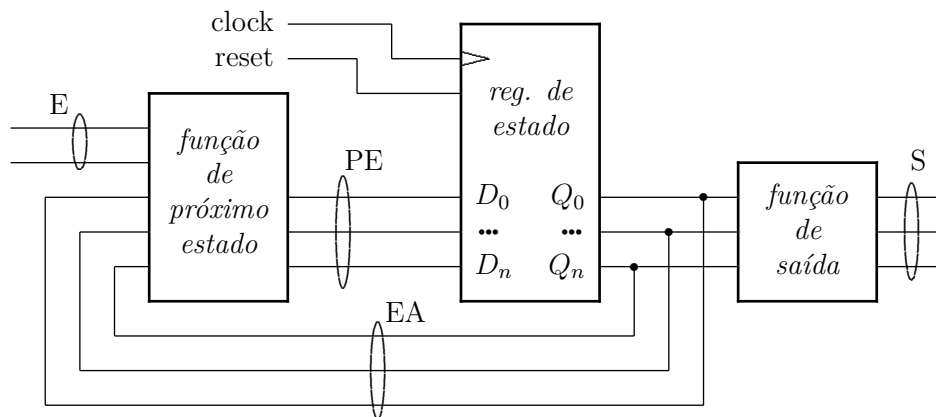


Figura 8.37: Máquina de Moore.

Exemplo 8.15 Vejamos um exemplo de uma Máquina de Moore que detecta sequências de dois ou mais 1s em sua entrada. O diagrama de estados da ME é mostrado no lado esquerdo da Figura 8.38, e um diagrama de tempo para a sequência de entrada **0010111** é mostrado à direita.

O sinal reset coloca a ME no estado A. Enquanto a entrada for **0**, a ME permanece no estado A. Na recepção do primeiro **1** a ME muda para o estado B, e retorna para A na recepção do terceiro **0**. No segundo **1** a máquina muda para o estado B, e na recepção do terceiro **1** a máquina vai para o estado C, quando então a saída é ativada porque uma sequência de dois 1s foi detectada.

Note que a mudança de estado ocorre depois da borda do relógio. Evidentemente, os parâmetros de temporização da entrada devem ser atendidos. ◁

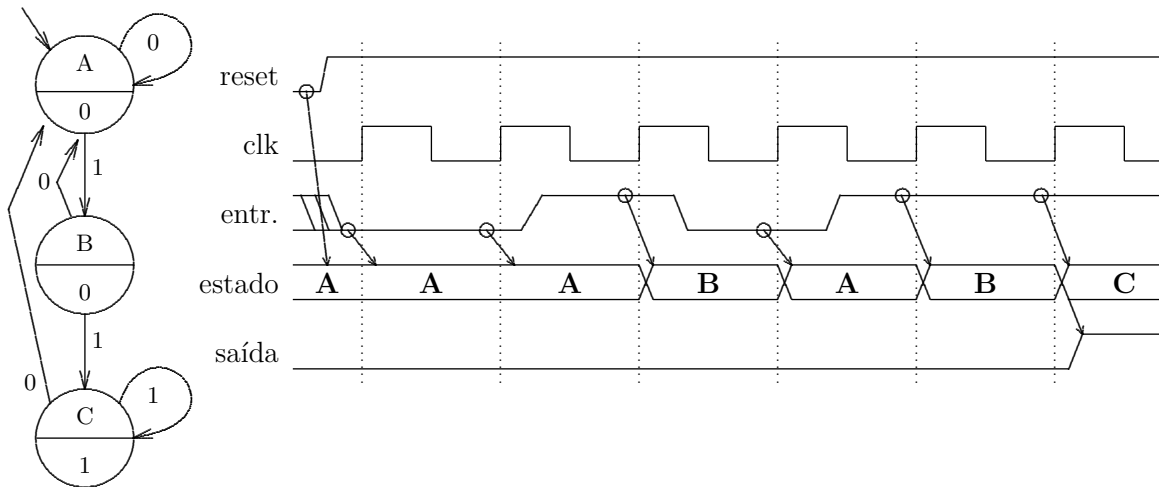


Figura 8.38: Máquina de Moore que reconhece a sequência 011^+ .

Máquina de Mealy

Numa máquina de Mealy a saída depende do estado atual e das entradas, como definido na Equação 8.11.

$$g : (\mathcal{Q} \times \mathcal{I}) \mapsto \mathcal{S} \quad \text{Máquina de Mealy} \quad (8.11)$$

Em resposta a uma sequência de entrada $e_1, e_2 \dots e_n$, $n \geq 0$, a saída produzida por uma Máquina de Mealy é

$$g(q_0, e_1), g(q_1, e_2) \dots g(q_{n-1}, e_n)$$

quando $q_0, q_1 \dots q_n$ é a sequência de estados tal que $f(q_{i-1}, e_i) = q_i$, $1 \leq i \leq n$. Note que esta sequência de estados tem um estado a menos que a sequência correspondente produzida por uma Máquina de Moore.

Numa Máquina de Mealy, como aquela mostrada Figura 8.39, as saídas dependem do estado atual e das entradas: $S = g(E, EA)$.

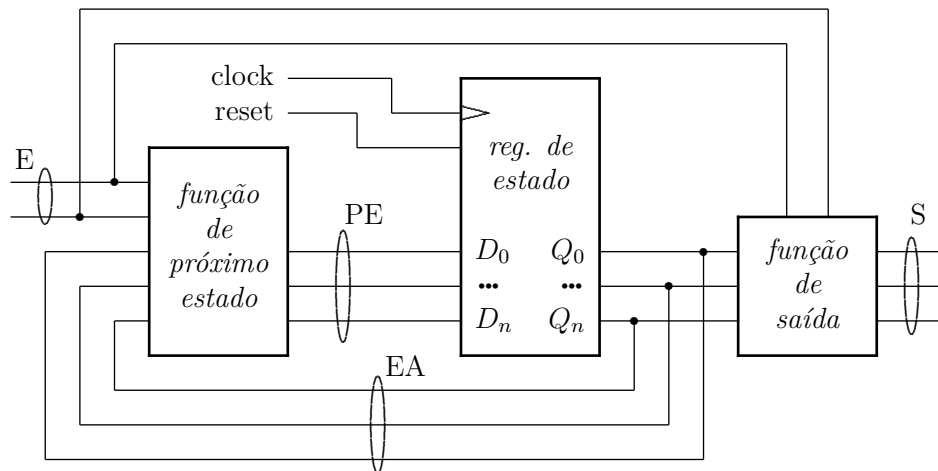


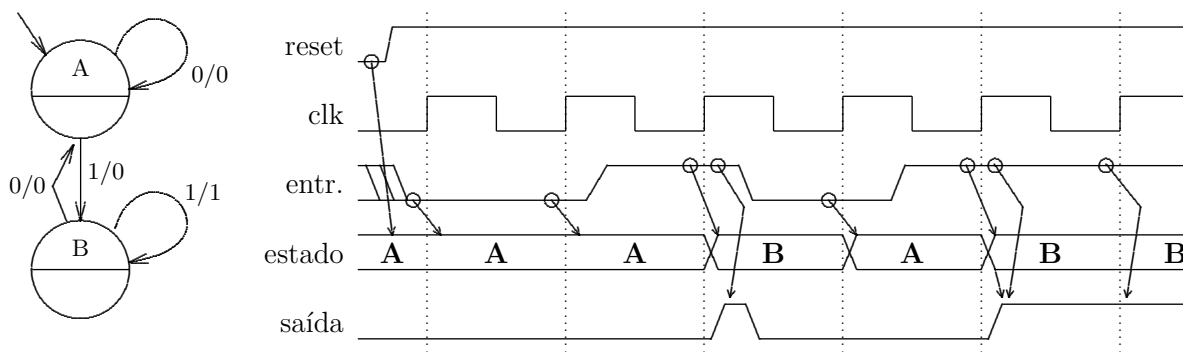
Figura 8.39: Máquina de Mealy.

Exemplo 8.16 Vejamos a Máquina de Mealy que detecta sequências de dois ou mais **1s** em sua entrada. O diagrama de estados da ME é mostrado no lado esquerdo da Figura 8.40, e um diagrama de tempo para a sequência de entrada **0010111** é mostrado à direita.

O sinal reset coloca a ME no estado A. Enquanto a entrada for **0**, a ME permanece no estado A. Na recepção do primeiro **1** a ME muda para o estado B, e retorna para A na recepção do terceiro **0**.

No segundo **1** a máquina muda para o estado B, e a saída é ativada porque uma sequência de dois **1s** foi detectada. Enquanto a ME estiver no estado B, a saída segue a entrada, e nesse exemplo ocorre um pulso espúrio na saída quando o primeiro **1** é amostrado. Contudo, imediatamente antes da borda do relógio, a saída é **0**, como desejado.

Esta é uma diferença importante entre os dois modelos de ME: numa Máquina de Mealy, a saída geralmente é “mais suja”, ou mais ruidosa, do que numa Máquina de Moore. ◁

Figura 8.40: Máquina de Mealy que reconhece a sequência 011⁺.

8.4.3 Implementação de Máquinas de Estado

A metodologia de projeto de máquinas de estado compreende seis passos:

1. compreensão e formalização do problema;
2. atribuição de estados;
3. cálculo da função de próximo estado;
4. cálculo da função de saída;
5. implementação; e
6. projeto dos testes.

Exemplo 8.17 Implementemos a ME do Exemplo 8.13, que reconhece a sequência 01110.

O passo de compreensão e formalização está descrito nos Exemplos 8.13 e 8.14.

São seis os estados e portanto são necessários $\lceil \log_2 6 \rceil = 3$ FFs. A atribuição dos estados pode ser, iniciando-se do estado de reset: $i \mapsto 0$, $A \mapsto 1$, $B \mapsto 2$, $C \mapsto 3$, $D \mapsto 4$, $E \mapsto 7$. A atribuição de 7 ao estado E simplifica a função de saída.

A função de próximo estado é mostrada na Tabela 8.9. Para os dois estados sem atribuição, estados 5 e 6, a escolha tenta ser segura: se a ME acidentalmente entrar num destes estados, no próximo ciclo a ME volta ao estado inicial.

A função de saída é a conjunção das saídas dos FFs porque $\text{num}(\langle q_2, q_1, q_0 \rangle) = 7$.

Para implementar as funções de próximo estado pode-se empregar três multiplexadores, como mostra a Figura 8.41. Como temos uma entrada e três FFs, são necessárias quatro variáveis de controle para selecionar o próximo estado. As funções de próximo estado foram simplificadas – conforme a coluna ‘8’ da Tabela 8.9 – para a implementação com três mux-8 ao invés da implementação direta com três mux-16.

Tabela 8.9: Tabela de PE da máquina que reconhece a sequência 01110.

EA	8	$q_2 q_1 q_0$	e	PE	$d_2 d_1 d_0$
i	0	0 0 0	0	A	0 0 1
		0 0 0	1	i	0 0 0
A	1	0 0 1	0	A	0 0 1
		0 0 1	1	B	0 1 0
B	2	0 1 0	0	A	0 0 1
		0 1 0	1	C	0 1 1
C	3	0 1 1	0	A	0 0 1
		0 1 1	1	D	1 0 0
D	4	1 0 0	0	E	1 1 1
		1 0 0	1	i	0 0 0
ϕ	5	1 0 1	X	i	0 0 0
ψ	6	1 1 0	X	i	0 0 0
E	7	1 1 1	0	A	0 0 1
		1 1 1	1	B	0 1 0

Quanto aos testes, obviamente deve-se empregar seqüências adequadas para garantir que a saída é aquela esperada quando a seqüência 01110 é observada na entrada. A seqüência 011101110 também deve fazer parte do conjunto de vetores de teste. Seqüências de 0s (indicadas como 0^+) e de 1s (1^+) devem ser testadas, assim como $\{0, 1\}^+ 01110 \{0, 1\}^+$. $\triangleleft$

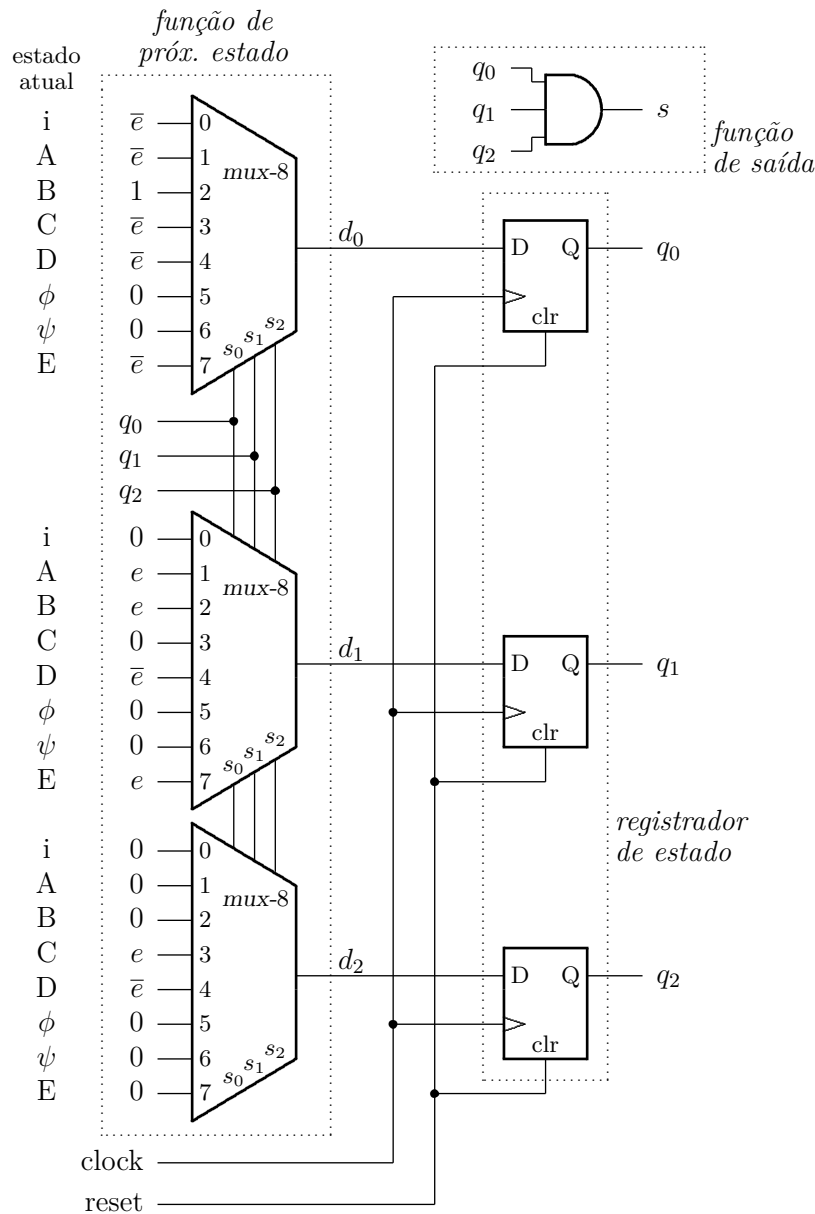


Figura 8.41: Implementação da ME que reconhece a seqüência 01110.

Exemplo 8.18 Considere o projeto de um contador módulo-10: para implementar este contador são necessários, no mínimo, quatro flip-flops: $\lceil \log_2 10 \rceil = 4$. A atribuição de estados mais simples é associar a cada estado de um contador binário o número que lhe corresponde. A função de próximo estado deve garantir que a seqüência normal de contagem seja obedecida, e ainda que quando o contador atinja o estado correspondente ao número 9 a seqüência reinicie de 0. A função de saída é trivial porque a própria atribuição de estados a definiu implicitamente. Quanto aos testes, a seqüência de contagem deve ser observada ao longo de, pelo menos, dois ciclos completos ($0 \mapsto 9, 0 \mapsto 9$). $\triangleleft$

8.4.4 Máquina de Vender Chocolates

Suponha que o controlador de uma máquina de vender chocolates deva ser implementado com uma máquina de estados. A máquina possui uma entrada para a inserção de moedas, uma saída para a devolução de moedas, e uma saída para entregar os chocolates vendidos. Para simplificar o problema, suponha que a máquina não devolve troco – se as moedas recebidas excedem o preço do chocolate, todas as moedas são devolvidas.

Cada chocolate custa \$1,00, e a máquina deve aceitar apenas moedas de \$0,50 e de \$0,25. O detector de moedas é muito bem construído e aceita e informa o valor de uma moeda de cada vez, e é capaz de rejeitar moedas de valores fora da faixa aceitável – moedas rejeitadas são desviadas automaticamente para a saída de moedas. O controlador digital possui duas entradas correspondentes a moedas de \$0,50 e de \$0,25, e duas saídas, *choc* que aciona o dispensador de chocolates, e *dev* que aciona a devolução de moedas. A Figura 8.42 mostra um diagrama de blocos com os componentes da máquina de vender chocolates.

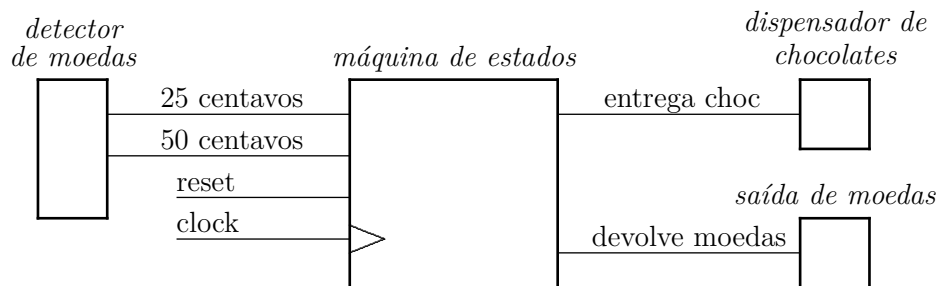


Figura 8.42: Diagrama de blocos da máquina de vender chocolates.

Formalização do problema A Figura 8.43 contém o diagrama de estados com a especificação do comportamento da máquina de vender chocolates. Os nomes dos estados correspondem ao valor acumulado até aquele estado. Se as moedas inseridas ultrapassam o valor de um chocolate, no estado *erro* todas as moedas são devolvidas. O cliente recebe seu chocolate após inserir \$1,00.

Atribuição de estados O diagrama de estados contém seis estados, sendo portanto necessários $\lceil \log_2 6 \rceil = 3$ FFs para implementá-lo. Uma das possíveis atribuições de estados é mostrada na Tabela 8.10.

Tabela 8.10: Atribuição de estados de máquina de vender chocolates.

estado	$q_2q_1q_0$
0c	0 0 0
25c	0 0 1
50c	0 1 0
75c	0 1 1
100c	1 0 0
erro	1 1 1

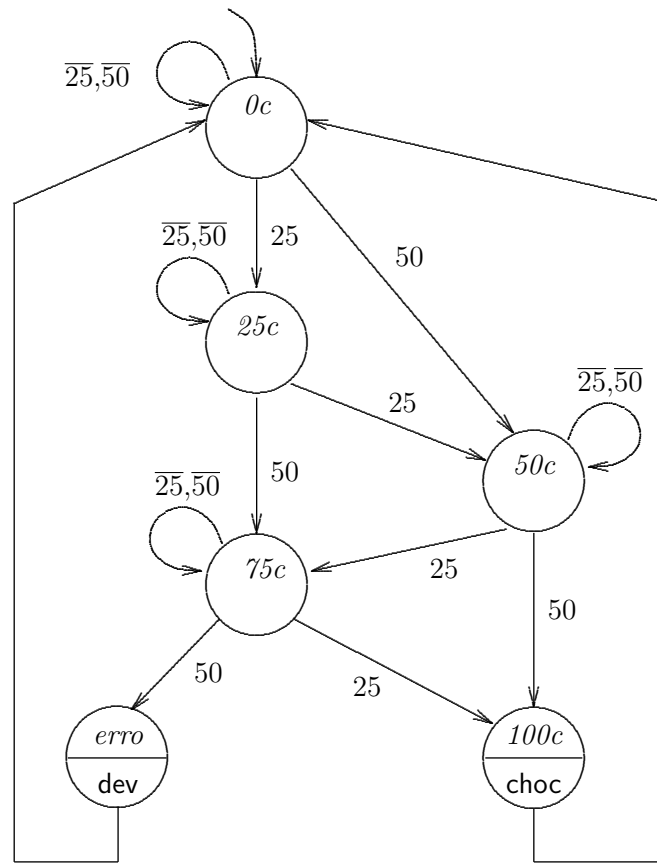


Figura 8.43: Especificação da máquina de vender chocolates.

Função de próximo estado Tomando por base a atribuição de estados da Tabela 8.10 e a especificação do comportamento do controlador da Figura 8.43, a função de próximo estado pode ser projetada, e a tabela de próximo estado é mostrada na Tabela 8.11. Para reduzir o tamanho da tabela, as colunas marcadas com X representam o par de valores $\{0, 1\}$ porque as saídas dos estados *devem* estar definidas para as quatro possibilidades das entradas: $\langle 0, 0 \rangle$, $\langle 0, 1 \rangle$, $\langle 1, 0 \rangle$ e $\langle 1, 1 \rangle$.

Duas decisões de projeto estão implícitas na tabela de próximo estado. A primeira, sinalizada por ‘†’, indica uma preferência por moedas de \$0,25 – se existem duas moedas disponíveis, o controlador consome a moeda de \$0,25. A segunda decisão de projeto, sinalizada por ‘‡’, define o comportamento do controlador se estados inválidos são atingidos pelo controlador – as moedas são todas devolvidas e a máquina fica a esperar por uma nova transação.

Função de saída A função de saída para uma Máquina de Moore é mostrada na Tabela 8.12. A saída *choc* é ativada no estado *100c* para entregar um chocolate ao cliente, e a saída *dev* é ativada no estado *erro* para devolver as moedas em caso de erro ou excesso de moedas.

Tabela 8.11: Tabela de próximo estado de máquina de vender chocolates.

estado	EA	entr 25 50	PE		
	$q_2 q_1 q_0$		$d_2 d_1 d_0$		
0c	0 0 0	0 0	0 0 0		
		0 1	0 1 0		
		1 0	0 0 1	†	
		1 1	0 0 1	†	
25c	0 0 1	0 0	0 0 1		
		0 1	0 1 1		
		1 0	0 1 0	†	
		1 1	0 1 0	†	
50c	0 1 0	0 0	0 1 0		
		0 1	1 0 0		
		1 0	0 1 1	†	
		1 1	0 1 1	†	
75c	0 1 1	0 0	0 1 1		
		0 1	1 1 1		
		1 0	1 0 0	†	
		1 1	1 0 0	†	
100c	1 0 0	X X	0 0 0		
—	1 0 1	X X	1 1 1	‡	
—	1 1 0	X X	1 1 1	‡	
erro	1 1 1	X X	0 0 0		

Tabela 8.12: Função de saída de máquina de vender chocolates.

estado	EA	saída
100c	100	choc
erro	111	dev

Implementação A função de próximo estado mostrada na Tabela 8.11 deve ser desmembrada em três funções, uma para cada um dos FFs (d_2, d_1, d_0). Cada função tem 5 variáveis – três FFs de estado (q_2, q_1, q_0) e duas entradas (25,50). As saídas dependem da mera decodificação dos estados: $\text{choc} = q_2 \wedge \overline{q_1} \wedge \overline{q_0}$, $\text{dev} = q_2 \wedge q_1 \wedge q_0$.

Exercícios

Ex. 8.22 Desenhe outro diagrama de estados para a máquina de vender chocolates (MdVC) considerando que a implementação será com uma Máquina de Mealy.

Ex. 8.23 Estenda o projeto da MdVC adicionando um botão que permita ao comprador abortar a operação a qualquer instante, causando a devolução das moedas.

Ex. 8.24 Estenda o projeto da MdVC adicionando um sinal para que o controlador solicite uma nova moeda ao detector de moedas.

8.5 Microcontroladores

Vejamos uma técnica mais simples e eficiente para o projeto de máquinas de estados finitas. Esta técnica se baseia na implementação das funções de próximo estado e de saída com memórias ROM. Considere a Máquina de Moore da Figura 8.44. A função de próximo estado e a função de saída podem ser implementadas como uma memória ROM indexada pelos bits de estado atual (EA) e pelas entradas (E). Algumas das saídas da ROM contém a função de próximo estado (PE) e outras a função de saída (S). A codificação das tabelas de próximo estado e de saída são gravadas diretamente na ROM, de tal forma que o conteúdo da n -ésima linha da tabela é gravado na n -ésima posição da ROM.

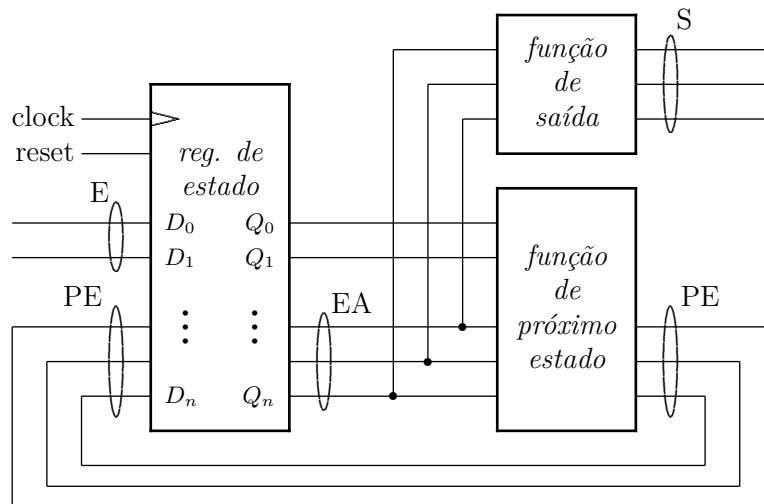


Figura 8.44: Máquina de Moore.

A utilidade desta técnica fica óbvia quando se considera o projeto de máquinas de estado complexas. Considere uma ME com 18 estados e 4 entradas; para codificar os 18 estados são necessários $\lceil \log_2 18 \rceil = 5$ *flip-flops*. A tabela de próximo estado possui $2^4 \times 2^5 = 2^9 = 512$ linhas. A simplificação das funções de PE dos 5 *flip-flops* é uma tarefa tediosa e que exige paciência e obstinação digna de orientais.

Se a tabela de próximo estado for copiada direta e simplesmente para uma memória ROM, com 512 linhas, o esforço de projeto é dramaticamente reduzido, diminui a possibilidade de introdução de erros, bem como o esforço para verificar a corretude da ME resultante. A geração do conteúdo da ROM pode ser facilmente automatizada, o que elimina quase todo o trabalho tedioso do projeto de MEs.

Na Figura 8.44, as entradas são sincronizadas no registrador de estado. Isso é necessário se as entradas são geradas por circuitos que operam de forma assíncrona ao relógio da ME. Se as entradas forem geradas externamente, mas de forma a garantir os requisitos de *setup* e *hold* do registrador de estado, então não é necessário sincronizar as entradas.

Algumas máquinas de estado complexas são chamadas de *microcontroladores*. O nome *microcontrolador*, como usado aqui, refere-se à técnica de implementação de máquinas de estado com dezenas ou centenas de estados e com fluxos de controle não triviais. Tais máquinas de estado são empregadas para controlar sistemas digitais complexos, tipicamente com milhares de portas lógicas e centenas de registradores.

Exemplo 8.19 Considere uma ME que detecta uma sequência de dois ou mais 1s, seguidos de 0. Um diagrama de blocos desta ME é mostrado na Figura 8.45, com uma entrada e e uma saída s , que fica em 1 somente quando a sequência é observada na entrada e . A figura também mostra o diagrama de estados do detector de sequência.

A Tabela 8.13 contém a codificação do diagrama de estados bem como a codificação da ROM que controla a ME. O lado esquerdo da tabela com a codificação da ROM mostra os bits de endereço da ROM, que são os dois bits do registrador de estado $\langle q_1 q_0 \rangle$ mais o bit da entrada e . Note que o sinal de entrada e é ligado ao endereço menos significativo da ROM.

O lado direito da tabela de codificação mostra o conteúdo de cada linha da ROM, com os dois bits de próximo estado $\langle d_1 d_0 \rangle$ e o bit de saída s . ◀

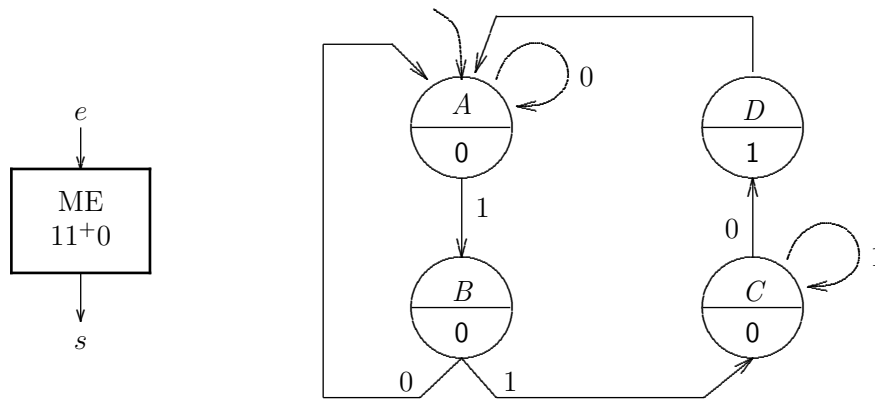


Figura 8.45: Máquina de estados que detecta a sequência 11+0.

Tabela 8.13: Codificação da ROM da ME que detecta a sequência 11+0.

Tabela de próximo estado						Codificação da ROM					
EA			PE			$\langle EA, e \rangle$			$\langle PE, s \rangle$		
$q_1 q_0$			$d_1 d_0$			$q_1 q_0 e$			$d_1 d_0 s$		
A	00	0	A	00	0	0	0	0	0	0	0
		1	B	01		0	0	1	0	1	0
B	01	0	A	00	0	0	1	0	0	0	0
		1	C	10		0	1	1	1	0	0
C	10	0	D	11	0	1	0	0	1	1	0
		1	C	10		1	0	1	1	0	0
D	11	0	A	00	1	1	1	0	0	0	1
		1	A	00		1	1	1	0	0	1

Exemplo 8.20 Considere um microcontrolador que pode seguir uma de várias sequências distintas. Uma entrada de 3 bits de largura determina qual das 8 possíveis sequências será seguida. Suponha que cada sequência possui no máximo 32 microinstruções. Ao final de cada sequência, o estado atual é reinicializado com o endereço da primeira microinstrução. O controlador possui 4 saídas.

A Figura 8.46 mostra um diagrama de blocos do controlador. As 3 entradas do controlador são ligadas, através do registrador, aos 3 bits mais significativos do endereço da memória ROM, $\langle e_7, e_6, e_5 \rangle$. Note que, ao contrário do circuito do Exemplo 8.19, as entradas ligadas aos bits mais significativos dividem

a ROM em oito faixas de endereços, e cada uma delas corresponde a uma sequência de $2^5 = 32$ microinstruções.

Os cinco bits menos significativos do endereço da ROM $\langle e_4 \dots e_0 \rangle$ são ligados ao registrador de estado, e o conteúdo de cada posição da ROM aponta o próximo endereço na sequência de microinstruções. Este projeto não usa um contador para percorrer as sequências, mas o próprio conteúdo da ROM determina qual a próxima microinstrução será executada. Cada palavra da ROM é dividida em 2 campos, o primeiro com os bits d_0 a d_4 aponta para a próxima microinstrução $\langle p_4, p_3, p_2, p_1, p_0 \rangle$, e o segundo campo, com os bits d_5 a d_8 , com as quatro saídas $\langle s_0, s_1, s_2, s_3 \rangle$. ◀

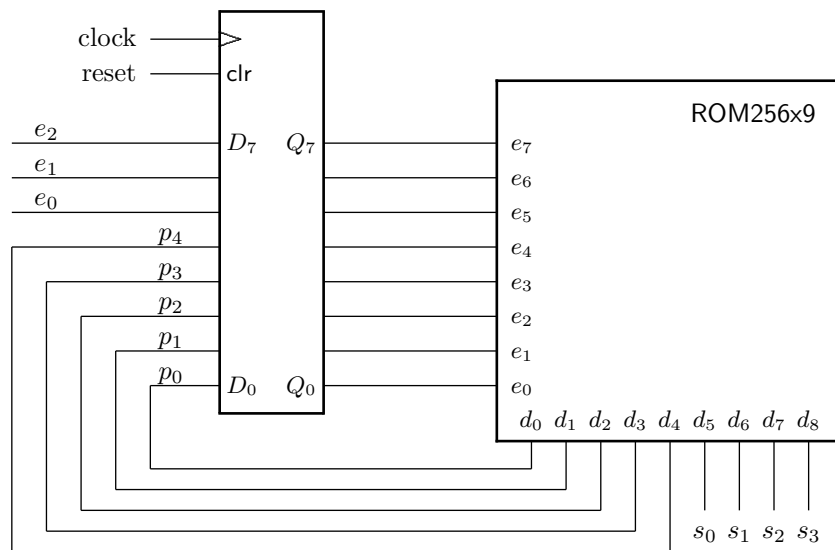


Figura 8.46: Microcontrolador com três entradas.

Exemplo 8.21 Suponha que você foi contratada para projetar a ME que controla uma bomba de (posto de) gasolina. A bomba fornece diesel, gasolina, etanol e querosene. O processo de venda para os quatro combustíveis é o mesmo, o que muda é o preço, que é definido pelo volume de líquido para cada Real de pagamento.

Uma possibilidade para o projeto da ME seria usar o mesmo sensor de volume, e fazer a multiplicação do preço com uma micro-rotina diferente para cada combustível. Os quatro combustíveis são coisas parecidas, mas não iguais. Portanto, sua ME emprega uma rotina (similar a uma função em Pascal) diferente para cada combustível.

Na implementação da tabela de próximo estado, cada micro-rotina corresponde a uma faixa de endereços, que contém as operações – sequência de sinais de controle – na ordem certa para cada rotina (diesel, etanol, etc.). Para simplificar o projeto, as faixas de endereço das microrotinas são do mesmo tamanho (2^n), mesmo que alguns endereços não sejam usados, e assim desperdiçados. Em geral, uns tantos bits de memória são muito menos custosos do que o tempo de projeto, implementação e testes.

Necessariamente, um sinal de dois bits identifica qual dos quatro combustíveis está sendo vendido, e este número é usado para escolher a microrotina apropriada. ◀

A época de glória dos microcontroladores foi nas décadas de 1960-80, quando eram usados como unidades de controle dos computadores de médio e grande porte da Digital (VAX11-780) e IBM (360, 370), por exemplo [BJ97]. Unidades de controle com microcontroladores também foram empregadas em microprocessadores das décadas de 1980-90, como a família 68000 da Motorola

e 80x86 da Intel, e são usadas correntemente em microprocessadores da AMD e da Intel para a interpretação das instruções complexas da arquitetura 80x86 [HP12]. Microcontroladores são também usados em multiprocessadores, para controlar as transações de memória que é fisicamente distribuída na máquina [JNW08]. Em geral, empregam-se microcontroladores em sistemas cuja velocidade de operação é tal que as funções de controle não podem ser desempenhadas por um microprocessador executando um programa de controle.

8.5.1 Máquina de Vender Chocolates – Versão 2

Considere uma segunda implementação do circuito de controle da máquina de vender chocolates. A função de próximo estado é repetida na Tabela 8.14, com as quatro combinações possíveis das entradas 25 e 50.

Tabela 8.14: Função de próximo estado da máquina de vender chocolates.

estado	EA	entr		PE
	$q_2q_1q_0$	25	50	$d_2d_1d_0$
0c	0 0 0		0 0	0 0 0
			0 1	0 1 0
			1 0	0 0 1
			1 1	0 0 1
25c	0 0 1		0 0	0 0 1
			0 1	0 1 1
			1 0	0 1 0
			1 1	0 1 0
50c	0 1 0		0 0	0 1 0
			0 1	1 0 0
			1 0	0 1 1
			1 1	0 1 1
75c	0 1 1		0 0	0 1 1
			0 1	1 1 1
			1 0	1 0 0
			1 1	1 0 0
100c	1 0 0	X X		0 0 0
—	1 0 1	X X		1 1 1
—	1 1 0	X X		1 1 1
erro	1 1 1	X X		0 0 0

Espaço em branco proposital.

Esta implementação do controlador da máquina de vender chocolates emprega uma memória ROM para implementar a função de próximo estado. A memória ROM é preenchida diretamente com a função de próximo estado de tal forma que o sequenciamento é gravado na ROM, conforme mostra a Tabela 8.15. A coluna *EA* corresponde ao estado atual, a coluna ‘25 50’ corresponde às duas entradas, a coluna *PE* corresponde ao próximo estado, e as saídas são mostradas nas duas colunas à direita.

Tabela 8.15: Codificação da ROM da máquina de vender chocolates.

estado	endereço		conteúdo		
	<i>EA</i>	25 50	<i>PE</i>	choc	dev
	$e_4e_3e_2$	e_1e_0	$r_4r_3r_2$	r_1	r_0
0 <i>c</i>	0 0 0	0 0	0 0 0	0	0
	0 0 0	0 1	0 1 0	0	0
	0 0 0	1 0	0 0 1	0	0
	0 0 0	1 1	0 0 1	0	0
25 <i>c</i>	0 0 1	0 0	0 0 1	0	0
	0 0 1	0 1	0 1 1	0	0
	0 0 1	1 0	0 1 0	0	0
	0 0 1	1 1	0 1 0	0	0
50 <i>c</i>	0 1 0	0 0	0 1 0	0	0
	0 1 0	0 1	1 0 0	0	0
	0 1 0	1 0	0 1 1	0	0
	0 1 0	1 1	0 1 1	0	0
75 <i>c</i>	0 1 1	0 0	0 1 1	0	0
	0 1 1	0 1	1 1 1	0	0
	0 1 1	1 0	1 0 0	0	0
	0 1 1	1 1	1 0 0	0	0
100 <i>c</i>	1 0 0	0 0	0 0 0	1	0
	1 0 0	0 1	0 0 0	1	0
	1 0 0	1 0	0 0 0	1	0
	1 0 0	1 1	0 0 0	1	0
inv1	1 0 1	0 0	1 1 1	0	0
	1 0 1	0 1	1 1 1	0	0
	1 0 1	1 0	1 1 1	0	0
	1 0 1	1 1	1 1 1	0	0
inv2	1 1 0	0 0	1 1 1	0	0
	1 1 0	0 1	1 1 1	0	0
	1 1 0	1 0	1 1 1	0	0
	1 1 0	1 1	1 1 1	0	0
erro	1 1 1	0 0	0 0 0	0	1
	1 1 1	0 1	0 0 0	0	1
	1 1 1	1 0	0 0 0	0	1
	1 1 1	1 1	0 0 0	0	1

A ROM é logicamente dividida em oito faixas de endereços, com quatro posições em cada faixa. O estado atual é determinado pelos 3 bits mais significativos do endereço $\langle e_4, e_3, e_2 \rangle$. As entradas são ligadas aos 2 bits menos significativos do endereço ($50 \leftrightarrow e_0$ e $25 \leftrightarrow e_1$). Dependendo das combinações de estado atual e entradas, o próximo estado é determinado pelos bits $\langle r_4, r_3, r_2 \rangle$ da ROM.

A Figura 8.47 mostra a implementação do controlador. A cada ciclo do relógio o registrador é atualizado em função do estado atual e das entradas. Note que as entradas são sincronizadas pelo relógio.

As duas principais vantagens desta implementação com relação àquela discutida na Seção 8.4.4 são a simplicidade do projeto e a possibilidade de se alterar a lógica do controlador sem grandes mudanças no circuito – basta ajustar o conteúdo da ROM para que este corresponda ao novo diagrama de estados. O circuito de controle é composto por apenas dois blocos, o registrador de estado e a memória ROM, o que simplifica muito a sua depuração e testes.

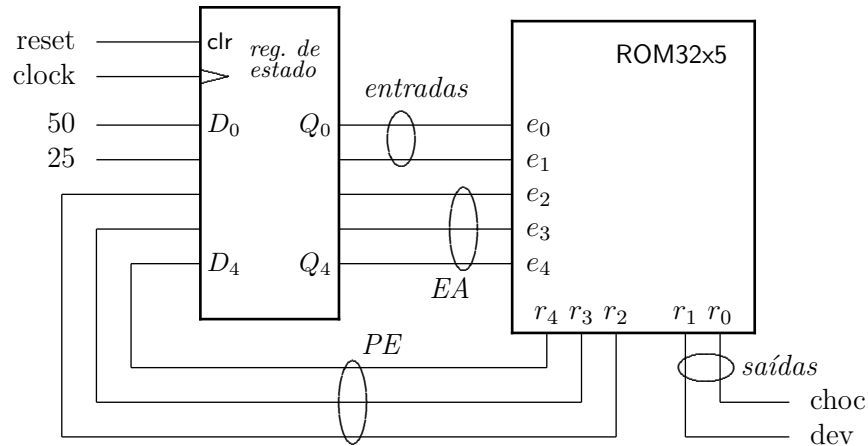


Figura 8.47: Controlador da máquina de vender chocolates.

8.5.2 Otimização da ROM com o Microprograma

Considere um microcontrolador com E entradas e Q bits de estado: a altura da sua ROM é de 2^{E+Q} linhas. Suponha que dentre as entradas, os valores de e_i e e_j nunca são amostrados simultaneamente num mesmo estado. Se este é o caso, então a altura da ROM pode ser reduzida à metade se um multiplexador for usado para escolher uma dentre e_i ou e_j , e neste caso, $E' = E/2$. Isso é obtido à custa de um bit de saída adicional, que é usado para escolher qual das entradas é amostrada em cada estado. Em geral, esta otimização é vantajosa se, (1) é fácil acrescentar um multiplexador ao projeto, e (2) o número de bits da ROM diminui. O número de bits da ROM é $A \times L$ para altura A e largura L . A otimização é vantajosa se $A/2 \times (L + 1) < A \times L$. Geralmente, uma ROM “mais quadrada” ocupa menos área do que uma ROM “mais retangular” e é mais rápida. Esta otimização é mostrada na Figura 8.48.

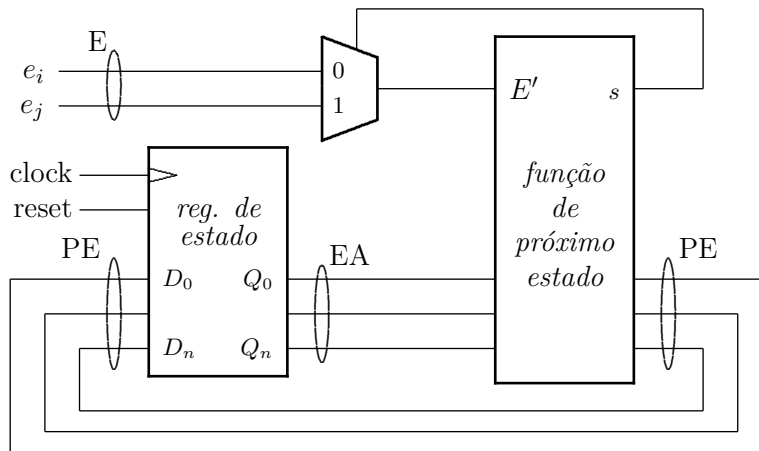


Figura 8.48: Microcontrolador com seleção de entradas.

Caso seja possível aumentar a largura da ROM, pode-se duplicar o campo de próximo estado – todo ele ou somente alguns dos bits. O próximo estado pode ser selecionado diretamente por uma ou mais entradas. Essa otimização reduz a altura da ROM porque uma das suas entradas é eliminada – e_k no diagrama da Figura 8.49 – enquanto que aumenta a largura porque alguns bits são acrescentados para a escolha do próximo estado.

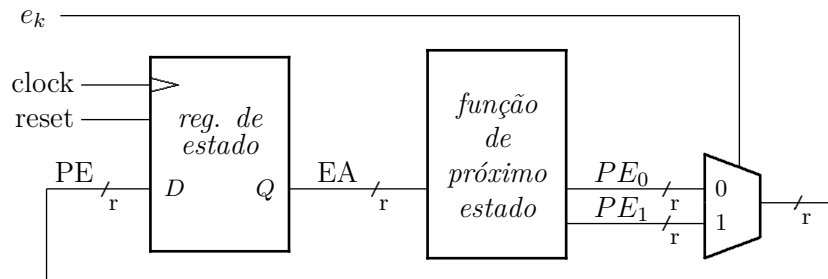


Figura 8.49: Microcontrolador com seleção do próximo estado.

Exercícios

Ex. 8.25 No circuito do Exemplo 8.20 podem ocorrer algumas sequências nas quais a porção final é comum a todas elas. Esta porção comum é chamada de *microrrotina*. Projete o circuito que permite desvios para a microrrotina. Suponha que uma microrrotina possui no máximo 32 microinstruções.

Ex. 8.26 Estenda o projeto do exercício anterior para permitir o desvio condicional para uma microrrotina.

Ex. 8.27 Estenda o projeto do exercício anterior para permitir o desvio condicional para uma de várias microrrotinas. Suponha que existem até oito microrrotinas distintas.

Ex. 8.28 Modifique a Tabela 8.15 associando as entradas 25 e 50 aos bits mais significativos $\langle e_3, e_4 \rangle$. Esta troca é vantajosa? Justifique.

Ex. 8.29 Considere a possibilidade de usar um contador no lugar do registrador de estado. Em que isso altera o nível de complexidade do projeto do microcontrolador?

8.6 Um *Flip-flop* por Estado

As formas de implementação de máquinas de estados que vimos até agora tentam minimizar o número de *flip-flops* que mantêm o estado da ME. Uma alternativa mais perdulária é usar um *flip-flop* para cada estado, e neste caso, o que se tenta otimizar é o tempo do projetista, ou a facilidade de gerar automaticamente uma implementação da ME. Esta implementação emprega um *flip-flop* por estado e o projeto deve garantir que, a qualquer instante, o estado da ME é representado por exatamente *um flip-flop* com sua saída em 1, enquanto que todos os demais estados/*flip-flops* mantêm suas saídas em 0.

Usaremos como exemplo a ME que detecta sequências de dois ou mais 1s seguidos de 0, descrita no Exemplo 8.19. O diagrama de estados desta ME é mostrado na Figura 8.50, juntamente com os *circuitos de entrada* dos estados à esquerda do diagrama, e os *circuitos de saída* dos estados, à direita. A flecha ' $\leadsto$ ' indica o estado associado à saída de cada circuito de entrada ou de saída.

O circuito de entrada do estado *A* é uma porta *or* de três entradas porque são três os estados que antecedem o estado *A*: o próprio estado *A* ou os estados *B* ou *D*. O circuito de entrada do estado *C* é uma porta *or* de duas entradas porque os estados que antecedem aquele estado são os estados *B* ou *C*.

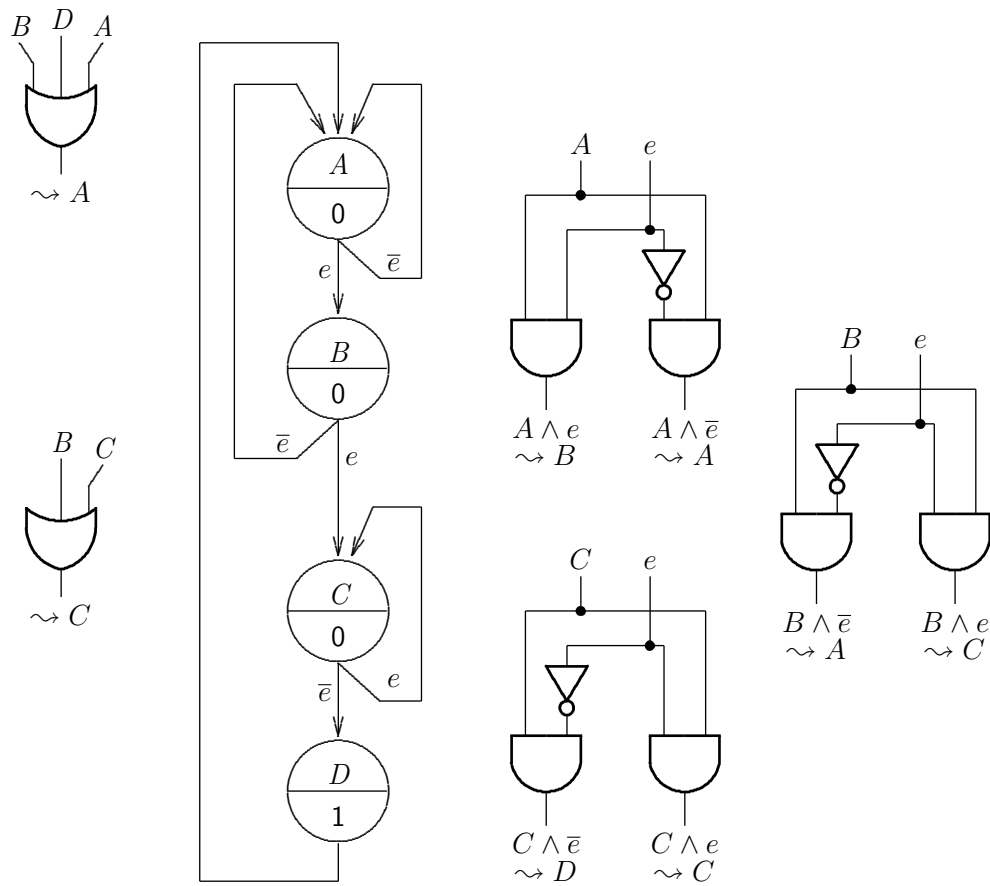


Figura 8.50: Transformação da ME que detecta a sequência 11^+0 .

O circuito de saída do estado A depende da entrada e : se $e = 0$ então a ME permanece a esperar por uma entrada 1, senão, $e = 1$ e o próximo estado é B . Os circuitos de saída dos estados B e C são similares. Os circuitos de saída são demultiplexadores cuja saída depende da combinação das entradas da ME.

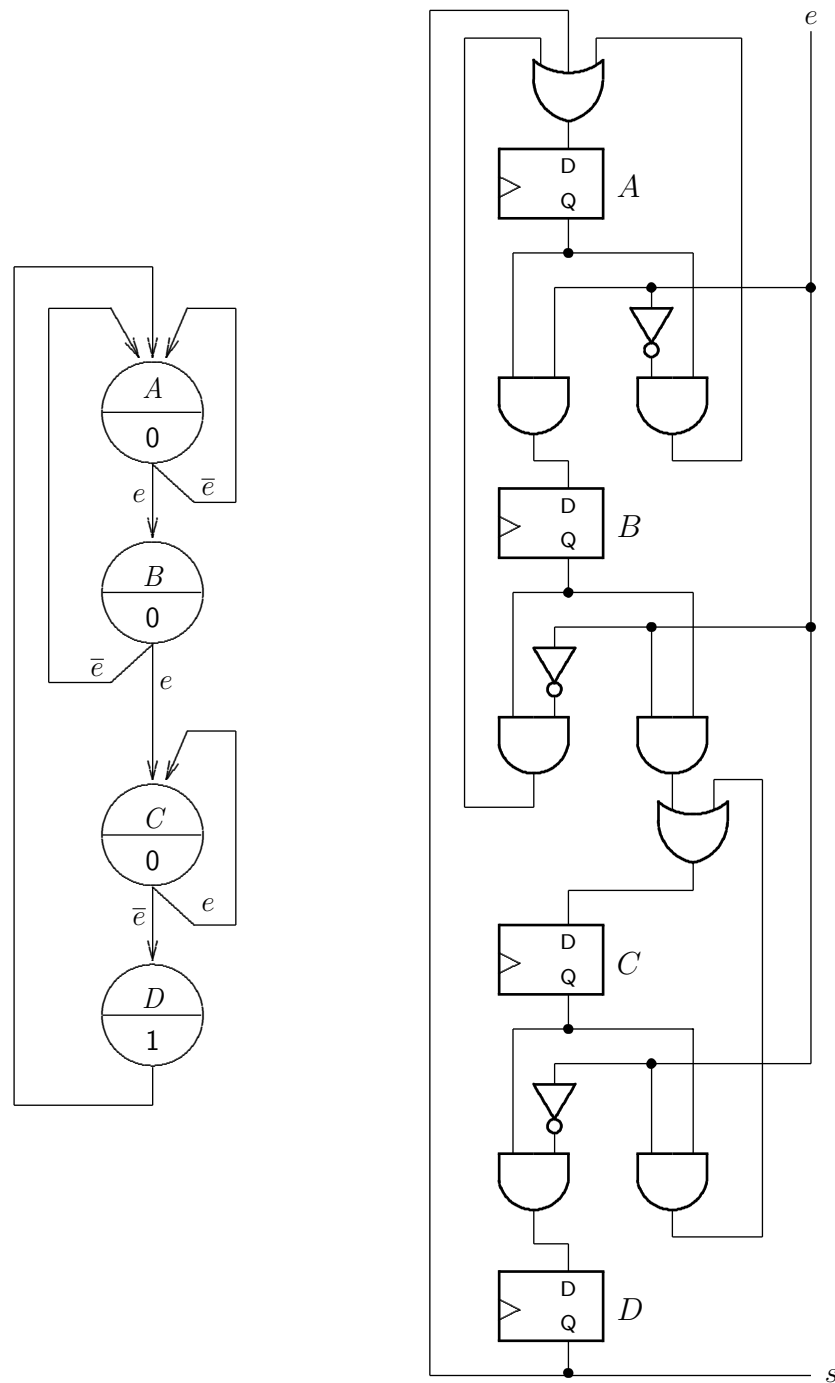


Figura 8.51: Implementação da ME que detecta a sequência 11^+0 .

Vejamos a implementação completa da ME com um *flip-flop* por estado, que é mostrada na Figura 8.51. O estado que corresponde ao *flip-flop* está indicado junto ao *flip-flop*. O circuito completo é a junção dos circuitos de entrada e de saída indicados na Figuras 8.50, mais os quatro *flip-flops* para os quatro estados.

No diagrama da figura não está indicado o circuito de inicialização da ME. Se os *flip-flops* não possuem entrada para o sinal de *reset* e nem de *set*, as portas lógicas dos circuitos de entrada

e de saída devem ser usadas para garantir que a ME é inicializada corretamente. No nosso exemplo, a porta *or* na entrada do *flip-flop-A* deve ter uma entrada adicional para que este *flip-flop* seja inicializado em 1. Os circuitos de entrada dos demais *flip-flops* devem empregar portas *and* de três entradas, com o sinal de *reset* ligado a todas as portas *and*, assim garantindo que todos os demais *flip-flops* sejam inicializados em 0.

Como dito inicialmente, esta implementação pode ser perdulária, por geralmente empregar um número maior de *flip-flops* e de portas lógicas do que uma implementação que minimiza o número de *flip-flops*. Simplificar a função de próximo estado de MEs pequenas como as que vimos até agora, com uns poucos estados e umas poucas variáveis, é factível. Esse não é o caso para MEs realistas com mais de 10 estados e/ou de 5 variáveis.

Uma vez compreendida a técnica de projeto com um *flip-flop* por estado, a implementação pode ser derivada pela mera inspeção do diagrama de estados da ME. A entrada de cada *flip-flop*/estado é uma porta *or*, e a saída de cada estado é um demultiplexador, que escolhe, em função das variáveis de entrada relevantes, qual será o próximo estado. Estes componentes são mostrados na Figura 8.52.

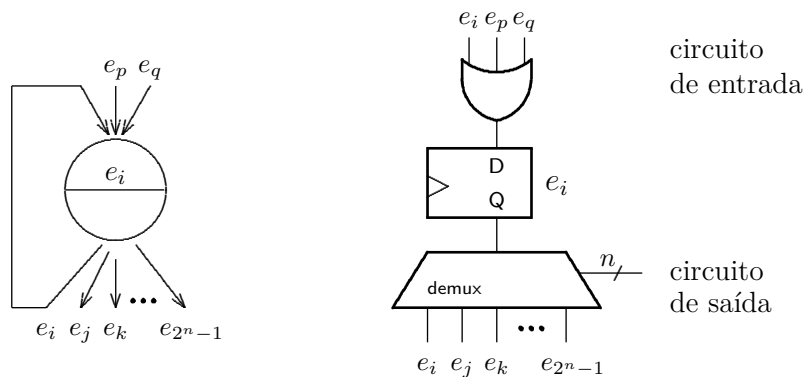


Figura 8.52: Modelo para implementação com um *flip-flop* por estado.

Nos falta implementar a função de saída da ME. No exemplo da ME que identifica a sequência 11^+0 , a saída que indica que a sequência foi encontrada é emitida no estado D , uma vez que esta é uma Máquina de Moore. Logo, a saída s é a própria saída Q do *flip-flop*. Numa Máquina de Mealy, cada saída é a conjunção do sinal de estado com a(s) entrada(s) apropriada(s).

8.7 Circuitos Complexos

Esta seção contém exemplos de circuitos sequenciais relativamente complexos que podem ser implementados pela composição de componentes simples tais como memória, contadores, registradores e máquinas de estado. Os controladores dos circuitos desta seção podem ser implementados como Máquinas de Mealy ou de Moore, com um *flip-flop* por estado, ou com microcontroladores. A escolha depende dos requisitos de projeto da aplicação, tais como custo, tempo de projeto, tempo de depuração, e velocidade mínima de operação. Os circuitos foram escolhidos por serem componentes de dispositivos sofisticados e para exemplificar as técnicas de projeto apresentadas nas seções anteriores.

8.7.1 Circuito de Dados Mais Controlador

Os circuitos nos exemplos desta seção consistem de duas partes. Um *circuito de dados* (*data-path*), com somadores, multiplexadores, deslocadores, unidades de lógica e aritmética e registradores, e um *circuito de controle*.

O circuito de controle é uma máquina de estados que ativa os sinais de controle dos circuitos combinacionais nos momentos apropriados. Os dados são processados repetidamente nos circuitos combinacionais, e os resultados parciais são mantidos nos registradores. Ao final de uns tantos ciclos, obtém-se o(s) resultado(s) desejado(s).

A Figura 8.53 mostra um modelo destes circuitos. A linha pontilhada indica a fronteira entre um circuito externo ao projeto (mestre), que fornece ao circuito interno (escravo) as entradas (ε_i e ε_j). Quando disponibilizadas, o mestre captura as saídas produzidas (σ_m e σ_n). No diagrama, as entradas são sinais com larguras i e j , e os sinais de saída têm largura m e n .

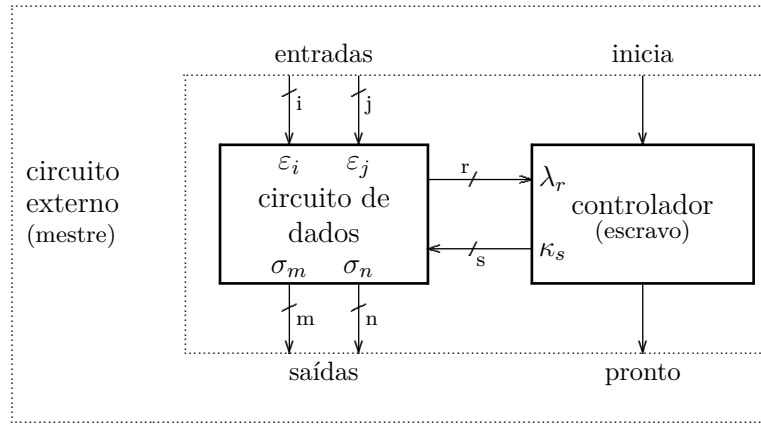


Figura 8.53: Modelo de circuito de dados e controlador.

Quando necessita de uma computação, o circuito externo apresenta as entradas e ativa o sinal *inicia*, indicando que os sinais de entrada estão válidos e estáveis. O controlador observa o estado do circuito de dados (λ_r), e atua sobre os componentes do circuito de dados ativando um ou mais dos sinais de controle (κ_s). Uma vez que os resultados estejam disponíveis, o controlador ativa o sinal *pronto*, indicando esta condição ao circuito externo. Uma vez que detecta o sinal *pronto*=1, o circuito externo faz *inicia*=0.

Considere que, inicialmente, o circuito está em repouso e os sinais *inicia* e *pronto* estão inativos. A sequência de sinalização

$$inicia \rightsquigarrow espera \rightsquigarrow pronto \rightsquigarrow \overline{inicia} \rightsquigarrow \overline{pronto}$$

sincroniza o mestre com o escravo, e ainda garante que o circuito interno somente observe entradas válidas, e que o circuito externo somente observe respostas válidas. Este é um *protocolo de sincronização* simples e extremamente útil. O termo em Inglês para este tipo de sincronização é *handshake*, que é uma forma de interação com regras bem definidas.

A Figura 8.54 mostra as máquinas de estado do mestre e do escravo que efetuam a sincronização. O mestre inicia no estado Γ e nele permanece até que necessite que o escravo efetue uma computação. Quando necessita de atenção, o mestre transiciona para o estado α e verifica se o escravo está livre, ao testar o sinal *pronto*. Se o escravo está livre (*pronto* inativo), então o mestre transiciona para o estado β e nele permanece até que o escravo complete sua tarefa.

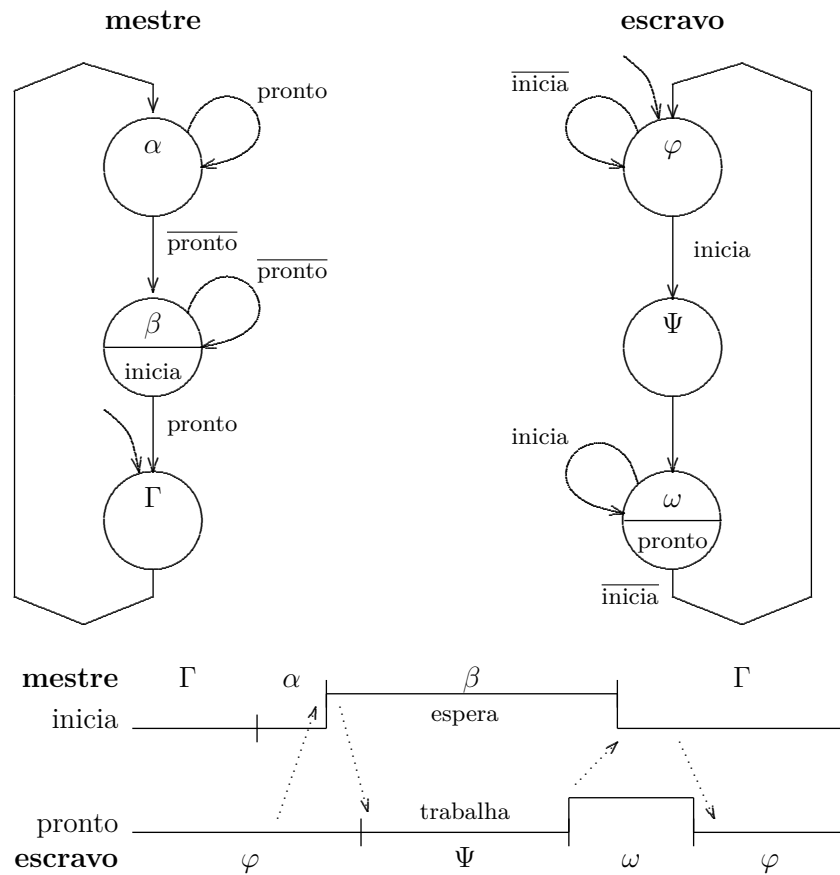


Figura 8.54: Handshake entre mestre e escravo.

O escravo inicia no estado φ , e quando o sinal *inicia* é ativado pelo mestre, o escravo transiciona para o estado Ψ e nele permanece até completar a tarefa. Quando isso ocorre, o escravo muda para o estado ω e ativa o sinal *pronto*. O mestre informa ao escravo que observou o resultado desativando o sinal *inicia*. A Figura 8.54 mostra também um diagrama de tempo com a interação entre as duas máquinas de estado. As quatro interações indicadas pelas linhas pontilhadas são bastante e suficientes para garantir a sincronização entre mestre e escravo.

Para simplificar o diagrama, os estados Γ no mestre, e Ψ no escravo, são indicados como um único estado. Na prática, estes ‘estados’ podem demorar de dezenas a milhares de ciclos do relógio, dependendo da complexidade dos circuitos do mestre e do escravo.

Falta vermos uma questão importante quanto ao modelo da Figura 8.53, que é a sincronização das transferências de dados do mestre para o escravo. Dependendo dos detalhes de cada projeto, é necessário ter em mente a temporização das trocas de dados entre mestre e escravo. Supondo que estes operam com um mesmo relógio, ou o mestre armazena os dados em registradores e os mantém estáveis até que *pronto* seja ativado, ou o escravo armazena suas entradas em registradores assim que *inicia* for ativado. O mesmo vale para as saídas: o escravo pode manter os resultados num registrador até que *inicia* seja ativado novamente, ou o mestre registra os resultados assim que *pronto* seja ativado. Seria assaz problemático se nem mestre nem escravo registrassem entradas e/ou saídas.

Exemplo 8.22 Vejamos como projetar a máquina de estados que controla a operação do somador serial da Seção 8.3.4, considerando operandos de 16 bits.

O circuito de dados é mostrado na Figura 8.55. Os dois registradores do lado esquerdo são registradores paralelo-série (com carga em paralelo); o registrador da direita é um registrador série-paralelo – desloca se o sinal de controle $\text{desloc}=1$.

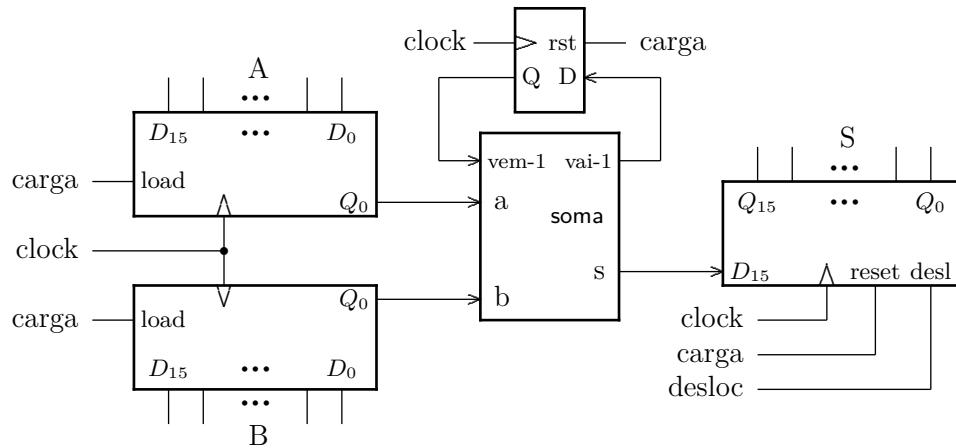


Figura 8.55: Somador serial de 16 bits – circuito de dados.

Depois que os operandos A e B estejam estáveis nas entradas dos registradores, o circuito externo dispara uma operação de soma ativando o sinal *inicia*. Quando a operação completar, a máquina de estados que controla o somador deve ativar o sinal de pronto, e manter a saída S estável até que *inicia* seja ativado novamente.

O diagrama de estados do controlador é mostrado na Figura 8.56. Para simplificar o desenho, as saídas ativas em cada estado são mostradas à direita do diagrama. Os estados *start* e *wait* correspondem ao estado Ψ da Figura 8.54.

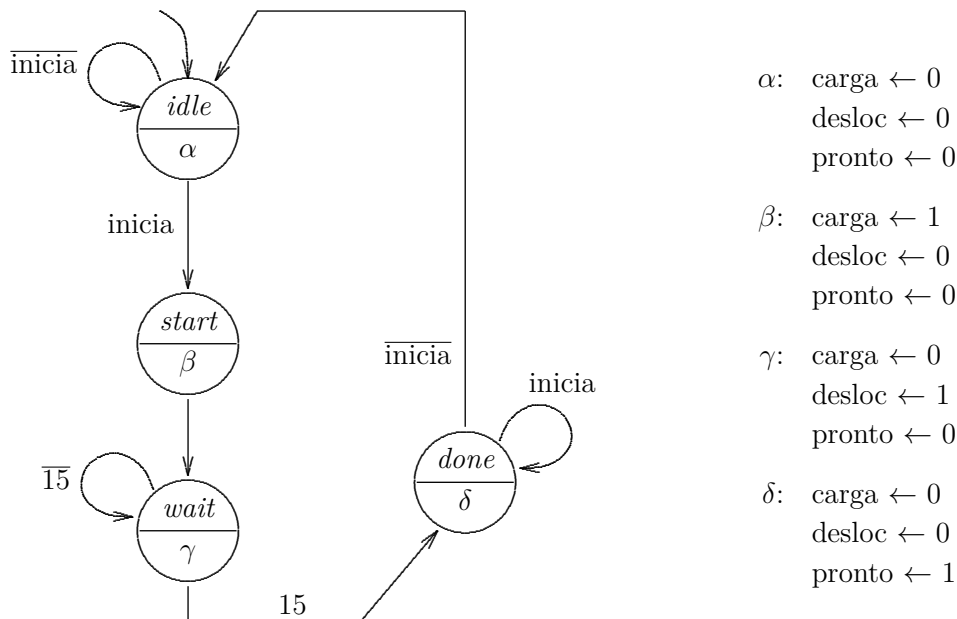


Figura 8.56: Somador serial de 16 bits – controlador.

O projeto do controlador pressupõe a existência de um contador módulo-16, que gera o sinal 15 quando a contagem chega em 15. Tanto os operandos quanto o resultados são mantidos nos registradores do escravo.

No estado *idle*, a ME espera pelo sinal externo inicia e nenhum dos sinais de controle está ativo.

No estado *start*, os registradores da entrada são carregados com os operandos A e B , o flip-flop do vai-um é inicializado em zero, e o registrador da direita é inicializado em zero ($S = 0$). O sinal inicia pode ser removido a qualquer tempo, desde que o seja antes da contagem chegar em 15.

No estado *wait*, a ME espera durante 16 ciclos – até que o sinal 15 seja ativado – a cada ciclo deslocando o resultado parcial no registrador S .

No estado *done*, a ME avisa ao circuito externo que o registrador S contém a soma $A + B$, fazendo *pronto*=1, e este registrador não é mais deslocado até que *inicia*=1.

Esta ME pode ser implementada com um flip-flop por estado, com dois flip-flops mais multiplexadores ou uma memória ROM. Uma alternativa ao contador seria incluir a contagem das 16 somas na própria ME, ao invés de usar um contador externo.

O diagrama de tempo da Figura 8.57 mostra o desenrolar da computação de uma soma. ◀

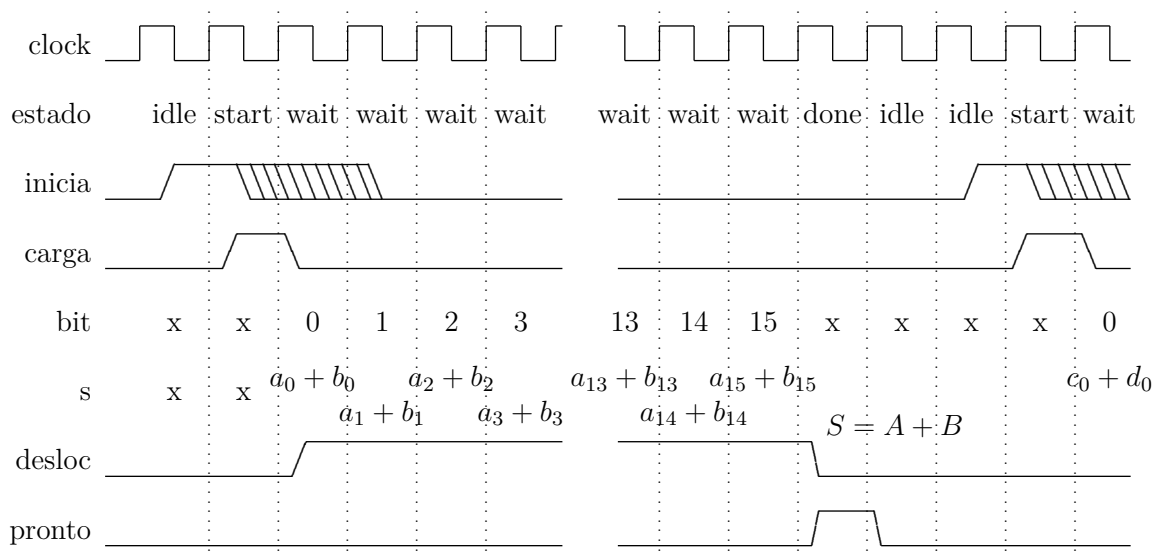


Figura 8.57: Somador serial de 16 bits – diagrama de tempos.

Exemplo 8.23 Você deve projetar um circuito sequencial síncrono que aceita, em sua entrada, uma sequência E de números positivos de 8 bits. A cada pulso do relógio, um novo valor E_i deve ser amostrado, e então as duas saídas MAX e min mostram os valores máximos e mínimos observados até o i -ésimo ciclo na sequência de entrada. O circuito é especificado pela Equação 8.12. A função com tipo $\mathbb{N} \mapsto \mathbb{B}^8$ corresponde a uma sequência de bytes; a cada ciclo do relógio $(i, i + 1, i + 2, \dots)$, um novo valor é apresentado à entrada do circuito $(E(i), E(i + 1), E(i + 2), \dots)$. Considere que os elementos da sequência são números naturais.

$$\begin{aligned}
 & \text{inicia} : \mathbb{B} \\
 & E : \mathbb{N} \mapsto \mathbb{B}^8 \\
 & MAX, min : \mathbb{B}^8 \\
 & MAXmin : \mathbb{B} \times (\mathbb{N} \mapsto \mathbb{B}^8) \mapsto (\mathbb{B}^8 \times \mathbb{B}^8) \\
 & MAXmin(\text{inicia}, E, MAX, min) \equiv \\
 & \quad \text{inicia} \Rightarrow \forall t \bullet [\text{num}(MAX) \geq \text{num}(E(t)) \wedge \\
 & \quad \quad \text{num}(min) \leq \text{num}(E(t))]
 \end{aligned} \tag{8.12}$$

Uma possível implementação para o circuito de dados é mostrada na Figura 8.58.

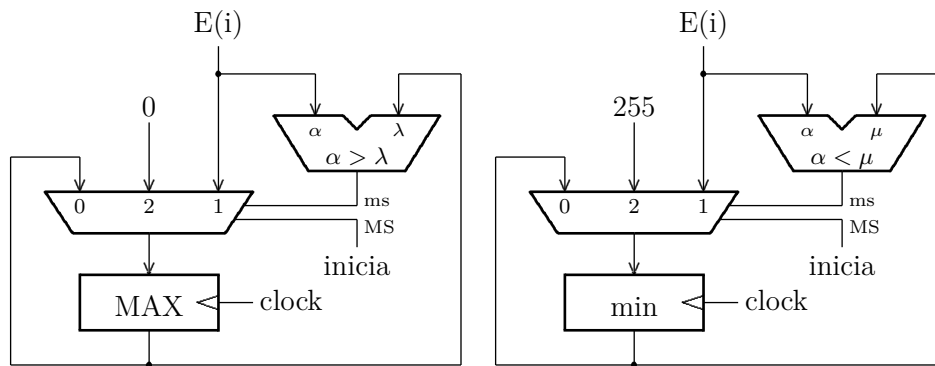


Figura 8.58: Circuito de dados do MAXmin.

O sinal *inicia* é ligado ao bit mais significativo da seleção dos multiplexadores e assim inicializa o registrador MAX com o menor valor possível, e o registrador min com o maior valor possível. O resultado das comparações é ligado ao bit menos significativo da seleção dos multiplexadores, e atualiza, se for o caso, o conteúdo dos registradores. O circuito de controle, neste exemplo, é trivial. ◁

Para computar o período mínimo do relógio de nossos projetos, usaremos a especificação de temporização da Tabela 8.16.

Tabela 8.16: Especificação temporal dos componentes dos exemplos.

Componente	[ns]	T_{prop}	T_{cont}	T_{setup}	T_{hold}
registrador		6	1	4	2
somador 8 bits		13	3	—	—
somador 16 bits		25	3	—	—
comparador de magnitude 8 bits		15	4	—	—
multiplexador		5	1	—	—
comparador com zero		8	1	—	—

Exemplo 8.24 Considerando a especificação temporal dos componentes da Tabela 8.16, qual é o período mínimo do relógio do circuito MAXmin?

O caminho crítico nos dois circuitos é composto do comparador de magnitude, do multiplexador e do registrador MAX/min, e o caminho mais longo é através do comparador de magnitude.

$$T_{min} \geq T_{reg} + T_{cMag} + T_{mux} + T_{su} = 6 + 15 + 5 + 4 = 30 \text{ ns}$$

$$T_h = 2ns \geq T_{C,reg} + T_{C,mux} = 1 + 1 = 2 \text{ ns}$$

Na Figura 8.58, o caminho mais curto para o T_h é mostrado à esquerda dos registradores. O resultado para T_h é marginal, mas ainda dentro do especificado. $\triangleleft$

Exemplo 8.25 Vejamos o projeto de um circuito que efetua a multiplicação de dois números positivos de 8 bits, através de adições repetidas. Seu comportamento é especificado pela Equação 8.13. Se os sinais inicia e pronto estão ambos ativos, então o registrador prod contém o resultado da multiplicação.

$$\begin{aligned} & \text{inicia, pronto} : \mathbb{B} \\ & \text{mndo, mdor} : \mathbb{B}^8 \\ & \text{prod} : \mathbb{B}^{16} \\ & \text{prod8} : \mathbb{B} \times (\mathbb{B}^8 \times \mathbb{B}^8) \mapsto \mathbb{B}^{16} \times \mathbb{B} \\ & \text{prod8}(\text{inicia}, \text{mndo}, \text{mdor}, \text{prod}, \text{pronto}) \equiv \\ & \quad \text{inicia} \Rightarrow \text{pronto} \Rightarrow \\ & \quad [\text{num}(\text{prod}) = \text{num}(\text{mndo}) \cdot \text{num}(\text{mdor})] \end{aligned} \tag{8.13}$$

A implementação do multiplicador é separada em duas partes, um circuito de dados, e a máquina de estados que o controla. A ME que controla o multiplicador é mostrada na Figura 8.59. A interface externa do multiplicador (prod8) aceita os operandos multiplicando (mndo) e multiplicador (mdor). O sinal de controle inicia faz com que os operandos sejam carregados nos registradores mndo e mdor, e dispara a computação do produto. Quando o resultado estiver disponível no registrador prod, o sinal pronto é ativado. Os estados start e wait correspondem ao estado Ψ da Figura 8.54.

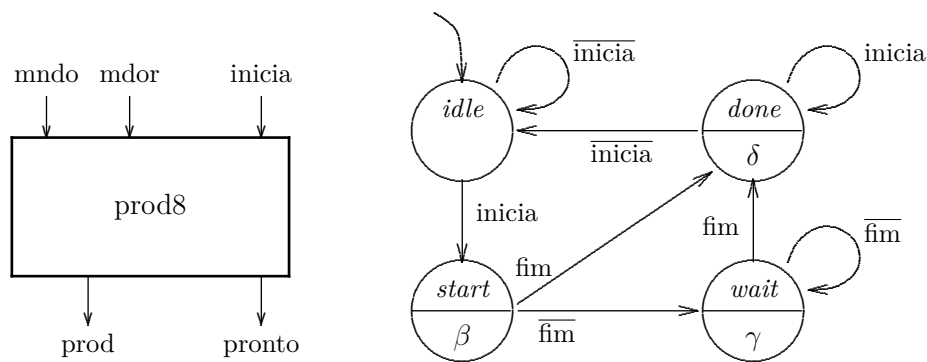


Figura 8.59: Interface e ME que controla o multiplicador.

Numa multiplicação por zero ($N \times 0$), o sinal pronto deve ser ativado sem que nenhuma parcela seja adicionada ao registrador prod. Se o multiplicador for diferente de zero, então a ME permanecerá no estado wait até que todas as parcelas sejam adicionadas.

O circuito de dados é mostrado na Figura 8.60. O multiplicador é carregado no registrador cnt, que atua como o contador de parcelas por adicionar. A cada parcela adicionada, cnt é decrementado; quando cnt for zero, o registrador prod contém o produto. O registrador prod é inicializado com zero, e a

Tabela 8.17: Sinais de controle do multiplicador.

estado	carMN	carC	selP	selC	carP	carC	pronto
β	1	1	1	1	1	1	0
γ	0	0	0	0	1	1	0
δ	0	0	0	0	0	0	1

cada ciclo do relógio, uma nova parcela – o multiplicando – é adicionada ao resultado parcial. A Tabela 8.17 contém os sinais ativos em cada estado. Quais sinais podem ser agrupados para simplificar a implementação?

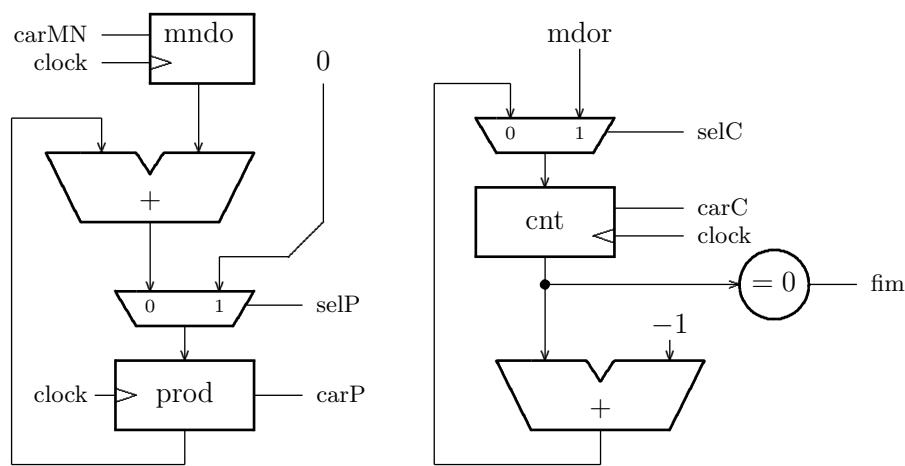


Figura 8.60: Circuito de dados do multiplicador.

A Figura 8.61 mostra um diagrama de tempo com duas operações; a primeira é uma multiplicação por 0, e a segunda uma multiplicação por 1.

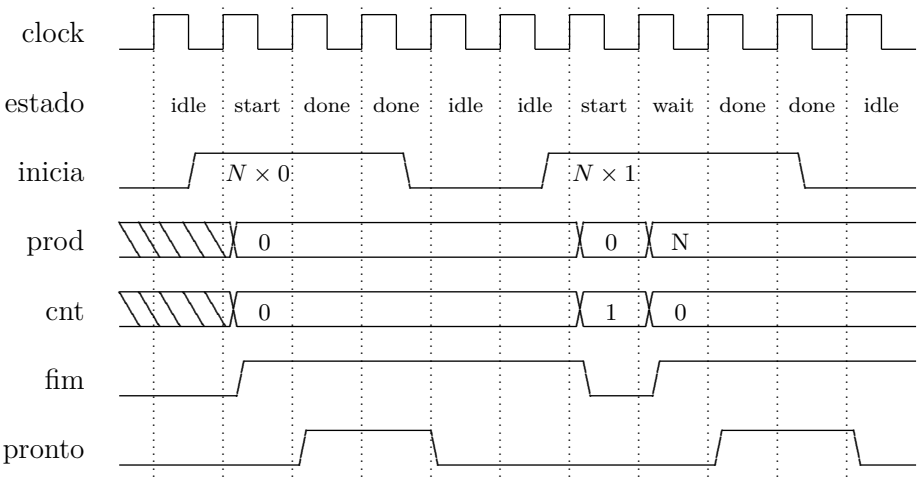


Figura 8.61: Diagrama de tempos de $N \times 0$ e $N \times 1$.

Exemplo 8.26 Considerando a especificação temporal dos componentes da Tabela 8.16, qual é o período mínimo do relógio do multiplicador?

O circuito com maior tempo de propagação é o que computa o produto, porque o tempo de propagação do somador de 16 bits (25ns) é maior do que aquele do circuito com o somador de 8 bits mais o comparador (13+8ns).

$$T_{min} \geq T_{prod} + T_{som16} + T_{mux} + T_{su} = 6 + 25 + 5 + 4 = 40 \text{ ns}$$

O tempo de propagação dos circuitos nas duas entradas do somador é idêntico porque as duas são providas pelos registradores mndo e prod. O setup no caminho crítico é o do registrador com o produto.

$$T_h = 2ns \leq T_{C,prod} + T_{C,som16} + T_{C,mux} = 1 + 3 + 1 = 5 \text{ ns}$$

A contaminação no circuito é maior do que T_h . ◁

Exemplo 8.27 Uma pequena alteração no circuito do Ex. 8.25 nos permite computar a exponenciação de inteiros. Se o somador que adiciona as parcelas do multiplicando for substituído por um multiplicador combinacional, o conteúdo do registrador prod será $prod = mndo^{mdor}$. O circuito da exponenciação é especificado pela Equação 8.14.

$$\begin{aligned} & inicia, pronto : \mathbb{B} \\ & valor, exp : \mathbb{B}^8 \\ & result : \mathbb{B}^{32} \\ & expon8 : \mathbb{B} \times (\mathbb{B}^8 \times \mathbb{B}^8) \mapsto \mathbb{B}^{32} \times \mathbb{B} \\ & expon8(inicia, valor, exp, result, pronto) \equiv \\ & \quad inicia \Rightarrow pronto \Rightarrow \\ & \quad [num(result) = num(valor)^{num(exp)}] \end{aligned} \tag{8.14}$$

Os dois operandos de 8 bits devem ser carregados nos registradores valor (mndo) e exp (mdor). O sinal de entrada da interface externa ao multiplicador inicia carrega os operandos e dispara a operação. O resultado, representado em 32 bits, é disponibilizado no registrador result (prod) quando o sinal pronto for ativado pela máquina de estados que controla o circuito. ◁

8.7.2 Bloco de Registradores

O *bloco de registradores* é um dos componentes mais importantes do circuito de dados de um processador. Os *registradores de uso geral* do processador são o nível mais elevado da hierarquia de armazenamento de dados de um computador – os demais níveis são memória cache, memória principal (RAM) e memória secundária (SSD ou disco). Na sua versão mais simples, um bloco de registradores contém duas portas de saída ou de leitura (A e B) e uma porta de escrita, ou de *destino* do resultado (D), além de sinais de controle.

A Figura 8.62 mostra uma parte do circuito de dados de um processador com a Unidade de Lógica e Aritmética (ULA) e um bloco de registradores com quatro registradores de 8 bits cada. As duas portas de saída A e B permitem a leitura do conteúdo dos registradores apontados por *selA* e *selB* respectivamente, enquanto a porta D permite a escrita síncrona do registrador apontado por *selD*, quando o sinal *escr* está ativo. Note que a leitura dos registradores apontados por *selA* e *selB* depende da operação de um circuito puramente combinacional – os multiplexadores que selecionam os conteúdos que serão exibidos nas *portas de leitura* A e B.

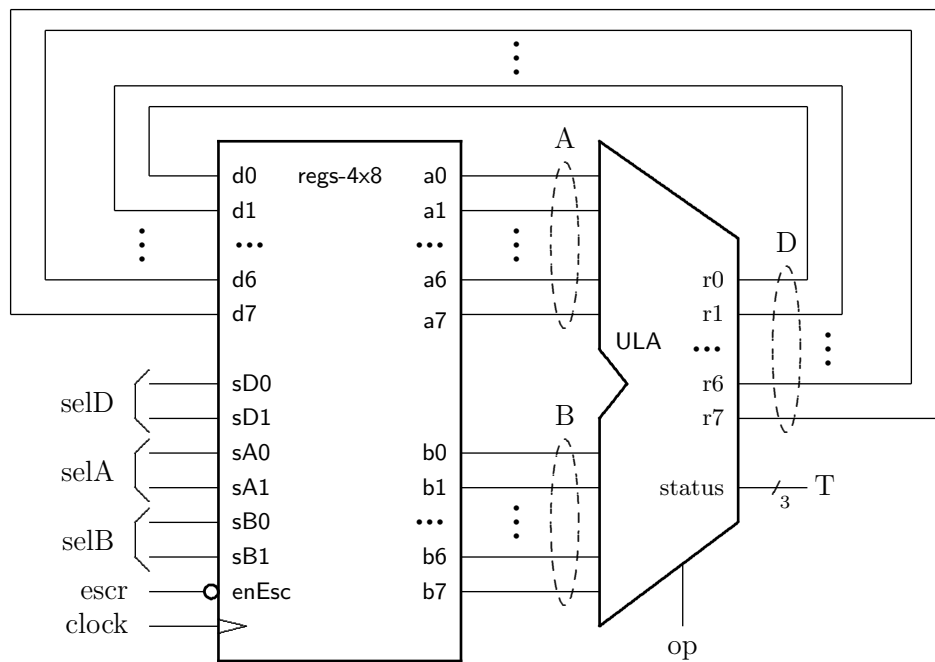


Figura 8.62: Bloco de registradores e Unidade de Lógica e Aritmética.

A atualização do registrador apontado por *selD* é síncrona e só ocorre na borda ascendente do sinal de relógio. A conjunto de sinais *selD*, *escr* e *D* é chamado de *porta de escrita*.

Os sinais de controle *escr*, *selA*, *selB* e *selD* são gerados pelo controlador do processador, e os valores armazenados nos registradores são computados na medida em que as instruções do programa são executadas pelo processador. A operação a ser efetuada pela ULA é determinada pelo circuito de controle, quando aquele decodifica as instruções de lógica e aritmética.

Um diagrama de blocos do bloco de registradores é mostrado na Figura 8.63.

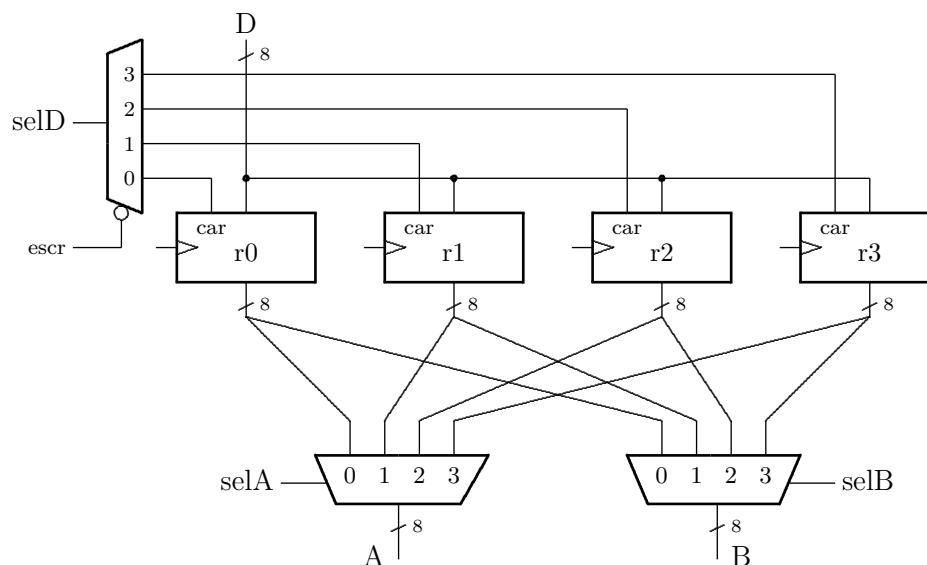


Figura 8.63: Bloco de registradores.

O bloco de registradores é um vetor de quatro elementos de 8 bits cada, e os sinais de seleção têm 2 bits para escolher um dentre os quatro registradores. Quando um par de endereços é aplicado às entradas de seleção, o par de valores armazenados nos registradores selecionados é apresentado nas saídas A e B , base do diagrama. Quando o sinal de escrita está ativo, na borda ascendente do relógio, o conteúdo do registrador apontado por $selD$ é atualizado com o valor na entrada D , no topo do diagrama.

O bloco de registradores implementa uma função memória que mapeia endereços em valores: endereço $\mapsto$ valor. A atualização dos registradores corresponde a uma modificação da imagem da função memória, que é atualizada com o novo valor (valor').

A Equação 8.15 especifica o bloco de registradores da Figura 8.63. As funções mem e mem' são os vetores de 4 bytes; a função mem' é a memória imediatamente após uma atualização.

$$\begin{aligned}
 & mem, mem' : \mathbb{B}^2 \mapsto \mathbb{B}^8 \\
 & A, B, D : \mathbb{B}^8 \\
 & selA, selB, selD : \mathbb{B}^2 \\
 & escr : \mathbb{B} \\
 \\
 & LER : \overbrace{(\mathbb{B}^2 \times \mathbb{B}^2)}^{selA, selB} \mapsto \overbrace{(\mathbb{B}^8 \times \mathbb{B}^8)}^{A, B} \\
 \\
 & ESC : \left(\overbrace{\mathbb{B}^8}^D \times \overbrace{\mathbb{B}}^{escr} \times \overbrace{\mathbb{B}^2}^{selD} \times \overbrace{(\mathbb{B}^2 \mapsto \mathbb{B}^8)}^{mem} \right) \mapsto \overbrace{(\mathbb{B}^2 \mapsto \mathbb{B}^8)}^{mem'} \\
 \\
 & \mathcal{R} : (LER \times ESC) \\
 \\
 & \mathcal{R}(selA, selB, A, B, escr, selD, D) \equiv \begin{cases} (A, B) = LER(selA, selB) \\ mem' = ESC(D, escr, selD, mem) \end{cases}
 \end{aligned} \tag{8.15}$$

A função LER especifica as portas de leitura. Esta operação é combinacional: uma vez que os endereços dos registradores estabilizem em $selA$ e $selB$, e decorrido o tempo de propagação através dos registradores e multiplexadores, os conteúdos dos registradores endereçados são exibidos nas portas de leitura A e B .

A função ESC especifica a porta de escrita. Esta operação é sequencial: a função memória é atualizada na borda do relógio se o sinal $escr$ estiver ativo. O conteúdo do endereço apontado por $selD$ é sobrescrito com o valor na entrada D .

A Equação 8.15 especifica os tipos das operações do bloco de registradores mas não define, concretamente, o comportamento dos circuitos. Esta forma de especificação abstrata deixa em aberto os detalhes da implementação, que devem ser fornecidos pelo projetista do bloco de registradores. O circuito da Figura 8.63 é uma dentre as possíveis implementações para o bloco de registradores.

A Figura 8.64 mostra parte da implementação do bloco de registradores. Para simplificar o diagrama, é mostrado somente o plano correspondente ao bit 0 do bloco de registradores. Os outros sete planos, correspondentes aos bits 1 a 7, são idênticos ao plano do bit 0.

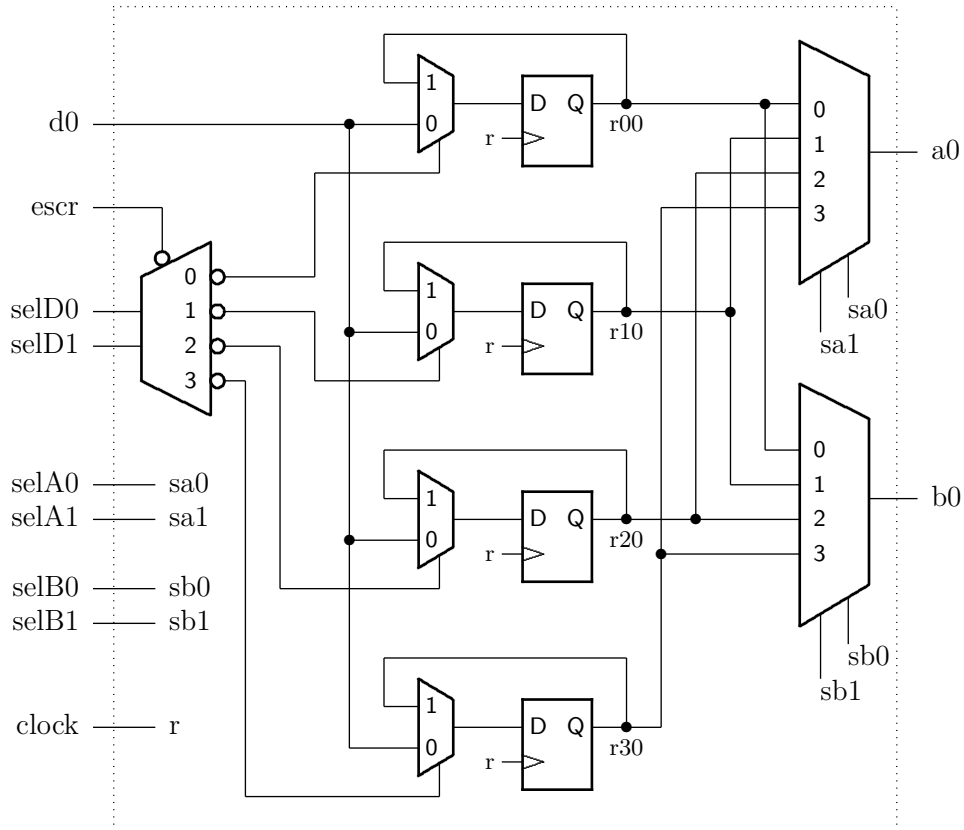


Figura 8.64: Plano do bit 0 do bloco de registradores.

A cada ciclo do relógio os valores dos registradores são atualizados. Se o sinal *escr* estiver inativo, todos os seletores nas entradas dos FFs escolhem a saída do FF e o conteúdo dos registradores não se altera. Se o sinal *escr* estiver ativo, o registrador selecionado por *selD* será atualizado com o valor presente na porta *D*. Os sinais *selA* e *selB* selecionam os registradores cujos conteúdos serão apresentados nas portas *A* e *B*, respectivamente. O sinal de *status* não é armazenado no bloco de registradores.

8.7.3 Circuito Integrado com Memória RAM

Esta seção contém uma brevíssima descrição da interface de circuitos integrados de memória RAM porque estes componentes são usados nos exemplos de circuitos sequenciais discutidos a seguir.

O comportamento de um circuito integrado de memória RAM é similar ao da memória ROM mas existe a possibilidade de que o conteúdo de cada posição endereçável seja modificado, além de ser examinado. A Figura 8.65 mostra o símbolo para uma memória RAM de 16 palavras, cada palavra com 8 bits de largura. Esta memória é chamada de RAM16x8: 16 palavras (de altura) com 8 bits (de largura).

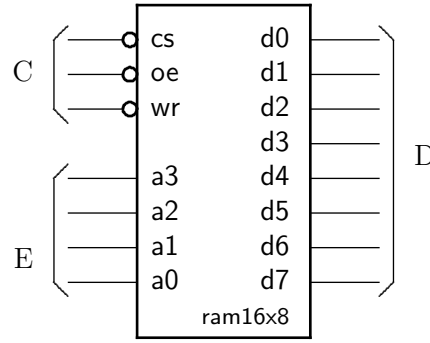


Figura 8.65: Símbolo da memória RAM 16x8.

Memórias RAM geralmente possuem três sinais de controle: *cs*, *oe* e *wr*. O sinal *cs*, ou *chip select*, habilita a operação do circuito – se inativo, o CI de memória não reage a nenhum dos outros sinais de controle. O sinal *oe*, ou *output enable*, habilita a saída de dados – se *oe* está inativo, a saída fica em *three-state* e o conteúdo da posição endereçada não é acessível para leitura. O sinal *wr*, *write*, define o tipo da referência: *wr*=0 indica escrita e a posição endereçada é atualizada com o conteúdo das linhas de dados, e *wr*=1 indica leitura e a posição endereçada é exibida nas linhas de dados quando *oe*=0. As relações entre os sinais de controle são definidas na Equação 8.16.

$$\begin{aligned}
 D &: \mathbb{B}^8 \\
 E &: \mathbb{B}^4 \\
 cs, oe, wr &: \mathbb{B} \\
 ram16x8 &: (\mathbb{B} \times \mathbb{B} \times \mathbb{B}) \times \mathbb{B}^4 \times \mathbb{B}^8 \\
 &\mapsto [(\mathbb{B}^4 \mapsto \mathbb{B}^8) \mapsto (\mathbb{B}^4 \mapsto \mathbb{B}^8)]
 \end{aligned} \tag{8.16}$$

$$\begin{aligned}
 \overline{cs} \wedge \overline{oe} \wedge wr &\Rightarrow D = ram16x8(E) \\
 \overline{cs} \wedge oe \wedge \overline{wr} &\Rightarrow ram16x8(E) \leftarrow D \\
 cs &: \text{ inativo}
 \end{aligned}$$

8.7.4 Pilha

Uma *pilha* é uma estrutura de dados que suporta, como um mínimo, três operações: (i) inicialização, (ii) inserção, e (iii) retirada. Elementos são inseridos na pilha através da operação *push*, e removidos da pilha através da operação *pop*. A inicialização deixa a pilha no estado *vazio*. Novos elementos são inseridos no topo da pilha, e o elemento inserido com último *push* é removido pelo próximo *pop*. Se a pilha está vazia um elemento não pode ser retirado, e se a pilha está cheia um novo elemento não pode ser inserido. Esta disciplina é conhecida como *LIFO* por conta do nome em Inglês: *Last In, First Out* – o último que entra é o primeiro a sair, que é o que ocorre numa pilha de cartas ou de pratos.

A Figura 8.66 mostra uma pilha com capacidade para 16 elementos, e com 14 posições já preenchidas. O *topo* da pilha é o endereço da última posição preenchida. O *apontador de pilha* aponta para a posição do topo. Na pilha da Figura 8.66, após a próxima inserção o topo estará na posição 14 e somente mais uma inserção é possível. Se o topo da pilha está na posição 1, somente uma retirada é possível porque esta última esvaziará a pilha. A operação

push armazena o novo valor no endereço imediatamente acima do topo e então incrementa o apontador. A operação *pop* decrementa o apontador e então mostra o valor que é o novo topo da pilha.

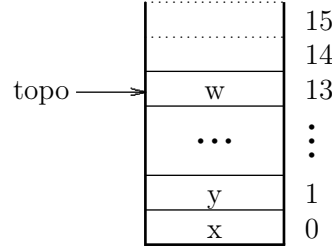


Figura 8.66: Pilha com capacidade para 16 elementos.

Uma pilha como esta pode ser implementada com um circuito sequencial. Os valores mantidos na pilha devem ser armazenados em uma memória que permita tanto a leitura como a atualização de seu conteúdo, como a apresentada na Seção 8.7.3. A implementação da pilha descrita a seguir possui três sinais de controle: *reset*, *push*, *pop*, quatro sinais de *status vazia*, *cheia*, *válido* e *erro*, e um barramento de dados D através do qual os elementos da pilha são inseridos e removidos.

As operações *push* e *pop* são mutuamente exclusivas e sua ativação simultânea é sinalizada como *erro*. O controlador da pilha não responde até que um dos dois sinais seja desativado, resolvendo assim a ambiguidade.

O comportamento da pilha é especificado pela Equação 8.17.

$$\begin{aligned}
 D &: \mathbb{B}^8 \\
 topo &: \mathbb{B}^4 \\
 reset, push, pop, vazia, cheia, erro &: \mathbb{B} \\
 pilha16x8 &: (\mathbb{B} \times \mathbb{B} \times \mathbb{B}) \times \mathbb{B}^8 \\
 &\mapsto \{ [(\mathbb{B}^4 \mapsto \mathbb{B}^8) \mapsto (\mathbb{B}^4 \mapsto \mathbb{B}^8)] \times (\mathbb{B} \times \mathbb{B} \times \mathbb{B}) \} \\
 \overline{reset} &\Rightarrow vazia = 1, cheia = 0, topo = 0, erro = 0 \\
 push \wedge \overline{pop} &\Rightarrow pilha16x8(topo) \leftarrow D; topo \leftarrow topo + 1 \\
 pop \wedge \overline{push} &\Rightarrow topo \leftarrow topo - 1; D = pilha16x8(topo) \\
 push \wedge pop &\Rightarrow erro = 1 \\
 repouso &: [cheia \triangleleft topo = 15 \triangleright \overline{cheia}] \wedge [vazia \triangleleft topo = 0 \triangleright \overline{vazia}]
 \end{aligned} \tag{8.17}$$

A Figura 8.67 mostra os componentes principais do circuito de dados da pilha. O contador *inc-dec* é o apontador da pilha, e aponta sempre para a próxima posição disponível. Quando *reset* é ativado, o valor na sua entrada é carregado e aquele endereço a primeira posição da memória, inicializando a pilha, vazia.

Numa inserção, o valor a ser inserido na pilha é gravado na memória e então o contador é incrementado, apontando a próxima posição vazia. Numa remoção, o contador é decrementado, apontando a última posição que foi preenchida, e então o conteúdo daquele endereço é exibido no barramento de dados. Os sinais *cheia* e *vazia* informam ao circuito externo sobre o estado da pilha. O sinal *válido* indica que o valor no barramento é válido e pode ser lido.

A Figura 8.68 mostra a máquina de Moore que determina o comportamento da pilha conforme especificado acima. Os sinais ativos em cada estado são mostrados à direita. Normalmente o

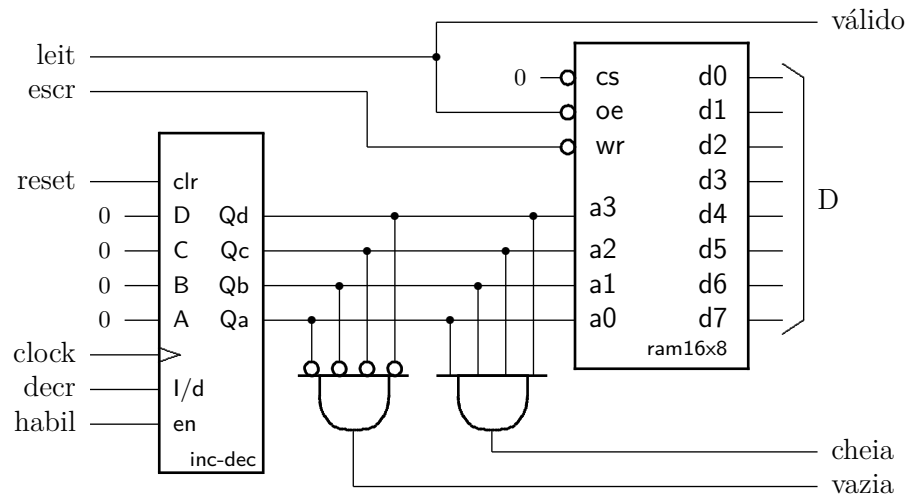


Figura 8.67: Circuito de dados da pilha.

barramento D está em *three-state* ($leit=1$) e o contador está desabilitado. Uma operação de remoção ($pop=1$) faz com que o contador seja decrementado no estado *rem* ($habil=0, decr=1$) e então o valor no topo da pilha é exibido no estado *val* ($leit=0$). Uma operação de inserção ($push=1$) causa uma escrita na memória ($escr=0$) e então o incremento no contador ($habil=0, decr=0$). Note que a escrita na posição vazia ocorre ao longo de todo o ciclo de relógio do estado *ins* e que o contador somente é incrementado na borda do relógio ao final do ciclo/estado.

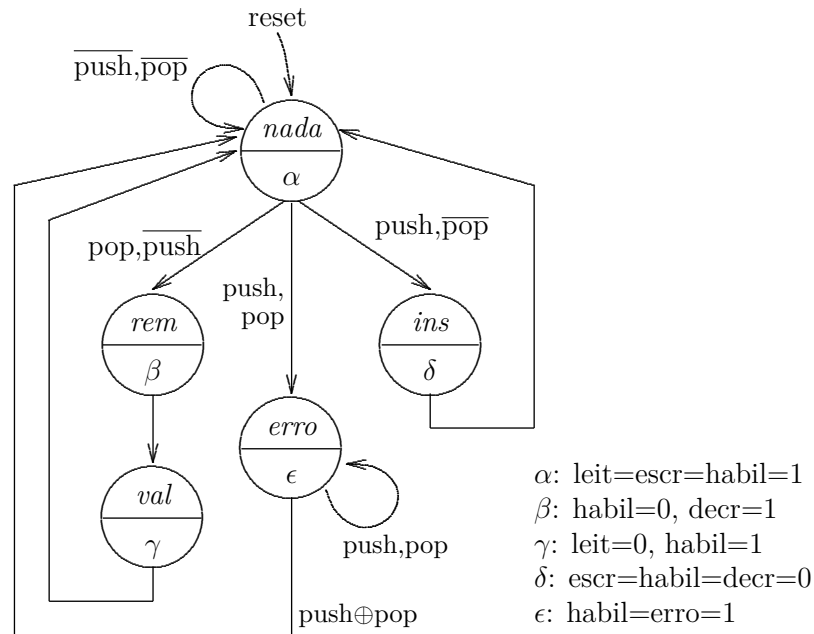


Figura 8.68: Máquina de estados da pilha.

O projetista circuito que faz uso desta pilha deve garantir que: (i) os sinais da interface são acionados/observados corretamente; (ii) o barramento contém dados válidos durante uma inserção; (iii) o valor do topo da pilha é copiado enquanto o sinal *válido* está ativo; e (iv) o período do relógio é suficientemente longo para a correta operação do contador e da memória.

8.7.5 Fila Circular

Uma fila circular pode ser implementada com um circuito sequencial similar ao da pilha vista na Seção 8.7.4. Ao contrário da pilha, inserções ocorrem no *fim* da fila enquanto que remoções ocorrem no *início* da fila, ou na sua *cabeça*. A fila de que trata esta seção é chamada de circular porque os apontadores de início e de fim da fila são implementados com contadores módulo N . Quando a contagem ultrapassa o módulo, a posição apontada pelo contador “fecha o círculo”, como indica a Figura 8.69. A disciplina de fila é chamada também de FIFO, de *First In, First Out* – o primeiro a entrar é o primeiro a sair, como numa fila de supermercado.

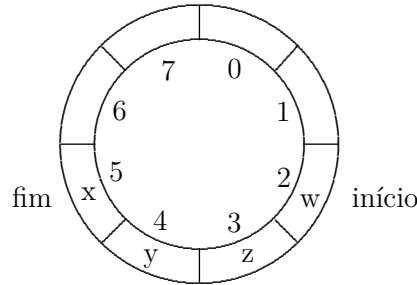


Figura 8.69: Fila circular com capacidade para oito elementos.

Estas filas são geralmente empregadas para amortecer diferenças entre as velocidades de dois componentes, um produtor de dados e um consumidor de dados. O tamanho da fila deve ser tal que o produtor possa produzir dados a uma taxa momentaneamente diferente daquela em que eles são consumidos, embora, no longo prazo, as taxas de produção e consumo devam ser iguais. A fila deve suportar, no mínimo, três operações: (i) inicialização; (ii) inserção; e (iii) retirada. Elementos são inseridos no final da fila com a operação *insere*, e removidos da cabeça da fila através da operação *remove*. A inicialização deixa a fila *vazia*. Se a fila está vazia a retirada de um elemento não é possível, e se a fila está cheia um novo elemento não pode ser inserido. Estes dois casos devem ser sinalizados ao circuito do qual a fila é um componente.

Para implementar uma fila são necessários uma memória RAM e dois contadores, um que aponta para o endereço do início da fila, e outro que aponta para o final da fila. A diferença entre os dois endereços é o tamanho da fila – o cálculo do endereço pode ser problemático porque os dois contadores têm módulo finito. Assim, um terceiro contador pode ser usado para manter o número de elementos na fila. A contagem é incrementada na inserção e decrementada na remoção. O comportamento da fila é especificado pelas Equações 8.18. Os incrementos nos apontadores são módulo-16.

$$\begin{aligned}
 D &: \mathbb{B}^8 \\
 ini, fim, tam &: \mathbb{B}^4 \\
 reset, ins, rem, vazia, cheia &: \mathbb{B} \\
 fila16x8 &: (\mathbb{B} \times \mathbb{B} \times \mathbb{B}) \times \mathbb{B}^8 \\
 &\mapsto \{ [(\mathbb{B}^4 \mapsto \mathbb{B}^8) \mapsto (\mathbb{B}^4 \mapsto \mathbb{B}^8)] \times (\mathbb{B} \times \mathbb{B} \times \mathbb{B}^4) \}
 \end{aligned} \tag{8.18}$$

$$\begin{aligned}
 \overline{reset} &\Rightarrow vazia = 1, cheia = 0, tam \leftarrow 0 \\
 ins &\Rightarrow fila(fim) \leftarrow D; fim \leftarrow fim + 1 \\
 rem &\Rightarrow D = fila(ini); ini \leftarrow ini + 1 \\
 \text{repouso} &: [cheia \triangleleft tam = 15 \triangleright cheia] \wedge [vazia \triangleleft tam = 0 \triangleright vazia]
 \end{aligned}$$

A Figura 8.70 mostra os componentes principais do circuito de dados da fila. A fila possui três sinais de controle *reset*, *ins*, e *rem*, dois sinais de *status vazia* e *cheia* e um barramento de dados *D* através do qual novos valores são inseridos e removidos. O sinal *válido* indica que o valor no barramento é válido e pode ser capturado pelo circuito externo. O contador *inc-dec*, – veja o Exercício 8.7 – mantém o número de elementos na fila. Os dois 74163 apontam um para o início e o outro para o fim da fila, e o seletor de endereços determina qual dos dois apontadores é usado para indexar a memória. Quando *reset* é ativado, os três contadores são inicializados em zero, esvaziando assim a fila.

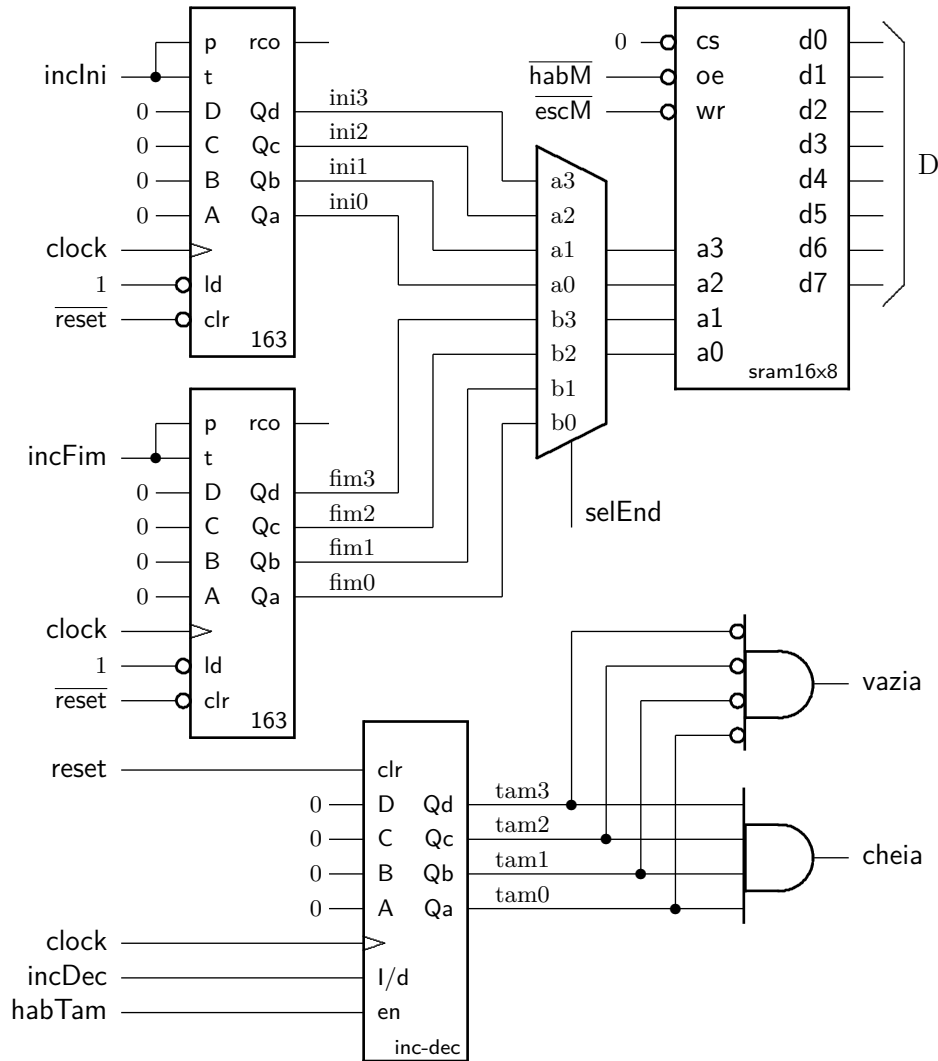


Figura 8.70: Circuito de dados da fila.

Numa inserção, o valor a ser inserido é gravado no endereço apontado pelo contador de fim da fila, e este é incrementado para que aponte para uma posição vazia. Numa remoção, o conteúdo do endereço apontado pelo contador de início de fila é exibido no barramento, e então este contador é incrementado para que aponte para nova cabeça da fila. Os sinais *cheia* e *vazia* informam ao circuito externo sobre o estado da fila.

Da mesma forma que no projeto da pilha, o usuário deste circuito deve garantir que: (i) o barramento contém dados válidos durante uma inserção; (ii) que o valor da cabeça da fila é

copiado enquanto o sinal *válido* está ativo; e (iii) o período do relógio é tão longo quanto o necessário para a correta operação dos contadores e da memória.

Exercícios

Ex. 8.30 Implemente o controlador da máquina de vender chocolates com um FF por estado. *Pista:* percorra o diagrama de estados de trás para a frente para definir as funções de próximo estado de cada FF.

Ex. 8.31 Desenhe os diagramas de tempo para o controlador da pilha, com os sinais de controle do contador e da memória, e certifique-se de que os sinais gerados pela máquina de estados da Figura 8.68 produzem os efeitos esperados.

Ex. 8.32 É possível simplificar a máquina de estados da pilha? *Pista:* projete uma Máquina de Mealy. Repita o Exercício 8.31 para seu novo controlador.

Ex. 8.33 Desenhe diagramas de tempo com os sinais de controle dos três contadores e da memória mostrados na Figura 8.70. Com base nos diagramas de tempo projete a máquina de estados que controla as operações da fila.

Ex. 8.34 Altere a especificação da fila para que ela se comporte como uma fila com capacidade para até 15 elementos. Se a capacidade for excedida, o sinal *cheia* é ativado; se a fila estiver vazia, o sinal *vazia* é ativado.

Ex. 8.35 A máquina de estados que controla o multiplicador do Exemplo 8.25 é uma Máquina de Moore. Projete uma Máquina de Mealy com funcionalidade equivalente. Talvez seja necessário alterar o circuito de dados.

Ex. 8.36 Indique no diagrama de tempos da Figura 8.61 todos os eventos do protocolo de sincronização entre mestre e escravo.

Ex. 8.37 Altere o circuito do Exemplo 8.25 para que o resultado seja o fatorial da entrada.

Ex. 8.38 Projete um circuito que efetua a divisão inteira de dois números positivos de 8 bits, como especificado na Equação 8.19. O circuito deve implementar a divisão por subtrações repetidas. Seu projeto deve conter o circuito de dados e a máquina de estados que o controla. Os dois operandos são carregados nos registradores *ddndo* e *dvsor* quando o sinal de entrada da interface externa *inicia* é ativado. Os valores finais do quociente e do resto são carregados nos registradores *quoc* e *resto* quando o sinal *pronto* for ativado pela máquina de estados do controlador.

$$\begin{aligned}
 & inicia, pronto : \mathbb{B} \\
 & ddndo, dvsor : \mathbb{B}^8 \\
 & quoc, resto : \mathbb{B}^8 \\
 & div8 : \mathbb{B} \times (\mathbb{B}^8 \times \mathbb{B}^8) \mapsto (\mathbb{B}^8 \times \mathbb{B}^8) \times \mathbb{B} \\
 & div8(inicia, ddndo, dvsor, quoc, resto, pronto) \equiv \\
 & \quad inicia \Rightarrow pronto \Rightarrow \\
 & \quad [num(ddndo) = num(dvsor) \cdot num(quoc) + num(resto)]
 \end{aligned} \tag{8.19}$$

Ex. 8.39 Altere o sua resposta ao Ex. 8.38 para que ele compute uma aproximação para a raiz quadrada. Este exercício é similar ao Exemplo 8.27.

Ex. 8.40 Projete um circuito combinacional que efetua a comparação de magnitude de dois números positivos de 16 bits. Este circuito tem duas entradas P e Q , de 16 bits cada, e três saídas que são *menor*, *igual* e *maior*, definidas na Equação 8.20.

$$\begin{aligned}
 P, Q &: \mathbb{B}^{16} \\
 menor, igual, maior &: \mathbb{B} \\
 menor &= num(P) < num(Q) \\
 igual &= num(P) = num(Q) \\
 maior &= num(P) > num(Q)
 \end{aligned} \tag{8.20}$$

Ex. 8.41 Um circuito captura dados emitidos por um sensor e os armazena numa memória com 1024 palavras de 16 bits. Os dados são gravados e mantidos em ordem crescente. Especifique e projete um circuito que encontra a posição na memória na qual um valor desejado se encontra, ou indica que o valor não existe. A implementação do circuito deve ser eficiente do ponto de vista do tempo de operação. Seu projeto deve conter o circuito de dados e a máquina de estados que o controla. *Pista:* por ‘eficiente’ entenda-se um algoritmo de busca com tempo de execução melhor do que “proporcional a N ”.

Índice Remissivo

Símbolos

T_A , 61
 T_D , 115
 T_I , 104, 111
 T_M , 112, 114, 175
 T_P , 111
 T_h , 177, 181, 188, 193
 T_p , 162
 $T_{C,F}$, 181
 $T_{C,x}$, 113–115, 188, 192, 194
 T_{FF} , 181
 T_{min} , 187, 194
 T_{skew} , 191
 T_{su} , 177, 181, 188, 193
 $\%$, 18–20
 $\bigvee$, 45
 $\bigwedge$, 45
 $\equiv$, 36
 $\gg$, 148
 $\wedge$, 30, 36
 $\triangleleft \triangleright$, 36, 42, 63
 $\leftarrow$, 182
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 40, 44
 $\ll$, 148
 $\neg$, 30, 36, 147
 $\vee$, 30, 36
 $\oplus$, 36, 50, 62
 $\mapsto$, 41
 $\mathbb{N}$, 140
 $\bar{a}$, *veja* $\neg$
 π , 25
 $\backslash$, 74
decod, 69
demux, 72
 e , 24
mod, 152, 206
mux, 64
 $\mathbb{B}$, 29–33, 38, 41
 $\mathbb{Z}$, 41, 144
 $[r]$, 229, 230
 $\mathbb{N}$, 41, 42, 144
num, 43, 52, 69, 74, 248, 249, 251, 261
 num^{-1} , 52
 $\mathbb{R}$, 41
 $|N|$, 215
 $\langle \rangle$, 29, 41, 42, 170, 190, 213

Números

74163, 206, 259
 74374, 220

A

abstração, 27
 bits como sinais, 27–33, 54, 172, 173
 tempo discretizado, 111, 113, 172, 175–177, 180–181, 185–186, 210
 acumulação de produtos parciais, 163–168
 adição, 146
 adiantamento de vai-um, 157–161
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo, 221
 conversão de base, 18
 conversão de base de frações, 22
 amplificador, 120, 130
 diferencial, 133
 amplitude, 198
 and, *veja* $\wedge$
and array, 122
 apontador de pilha, 255
 aproximação, 24
 árvore, 61, 113
assembly, *veja* ling. de montagem
 associatividade, 31
 atraso, *veja* tempo de contaminação, 54
 atribuição, 12
 autômatos finitos, 224

B

barramento, 135
barrel shifter, 152
 básculo, 132, 173–178
 relógio multi-fase, 214
 SR, 174, 183
 binário, 20
 bit, 20
 de sinal, 141
 bits, 27–37, 62
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 variável, 30
 bloco de registradores, 251
 borda, *veja* relógio
buffer, 118
buffer three-state, 134

buraco, 83
 busca binária, 261
 byte, 11

C

C,
 deslocamento, 153
 C, linguagem, 153
 cadeia,
 de portas, 61, 113
 de somadores, 146
 caminho crítico, 112
 capacitor, 100, 102, 128, 129
capture FF, veja *flip flop*, destino
 CAS, 129
 célula de RAM, 78
 célula, 95
 chave,
 analógica, 136
 digital, 87
 normalmente aberta, 87
 normalmente fechada, 75, 87
chip select, 255
 ciclo,
 combinacional, 54
 violação, 78, 173, 175, 177
 de trabalho, 213
 circuito,
 combinacional, 54, 62
 de controle, 244
 de dados, 244
 dual, 94
 segmentado, 191
 sequencial síncrono, 185, 197
 circuito aberto, 54
clear, 183
 clk, veja *clock*
clock, 177, 198–213
 período mínimo, 187
 skew, 191
clock-to-output, 181
clock-to-Q, 181
 CMOS, 56, 62, 82–137
 buffer three-state, 134
 célula, 95
 inversor, 91
 nand, 94
 nor, 93
 porta de transmissão, 136
 portas inversoras, 94
 sinal restaurado, 120
 código,
 Gray, 60, 205, 215, 224
Column Address Strobe, veja CAS
 combinacional,
 ciclo, 54
 circuito, 54
 dispositivo, 54
 comparador,

 de igualdade, 59, 156
 de magnitude, 156, 248, 261
Complementary Metal-Oxide Semiconductor, veja CMOS
 complemento, veja $\neg$
 complemento de dois, 139–145, 147–156
 complemento, propriedade, 31
 comportamento transitório, veja transitório
 comutatividade, 31
 condicional, veja $\triangleleft \triangleright$
 condutor, 82
 conjunção, veja $\wedge$
 conjunto mínimo de operadores, 62
 contador, 199–211
 74163, 206
 assíncrono, 208
 em anel, 212, 213, 220
 inc-dec, 212, 259
 Johnson, 215
 módulo-16, 203, 206
 módulo-2, 200, 211
 módulo-32, 211
 módulo-4, 200
 módulo-4096, 207
 módulo-8, 201, 209
 ripple, 208, 210
 síncrono, 210
 contra-positiva, 40
 controlador, 230
 de memória, 129
 conversão de base, 18
 conversor,
 paralelo-série, 217
 série-paralelo, 216
 corrente, 82, 100, 102, 107, 109
 de fuga, 110
 corrida, 118, 120
 CSS, 185, 197
 curto-circuito, 54

D

datapath, veja circuito de dados
 decimal, 17
 decodificador, 69, 75, 79–81, 121
 de linha, 121, 129
 de prioridades, 71
delay, 54
 demultiplexador, 72, 115
design unit, veja VHDL, unidade de projeto
 deslocador exponencial, 150
 deslocamento, 148–152
 aritmético, 149, 152, 156
 exponencial, 152, 189
 lógico, 149, 155
 rotação, 152
 detecção de bordas, 118
 diagrama de estado, 222
 restrições, 224
 disjunção, veja $\vee$

dispositivo, 82
 combinacional, 54
 distributividade, 31, 34, 49
 divisão de frequência, 201, 204
 divisão inteira, 43, 260
 doador, 83
don't care, 67
 dopagem por difusão, 83
 dopante, 83
 DRAM, 128–132
 controlador, 129
 fase de restauração, 131
 linha,
 de palavra, 129
 linha de bit, 129
 linha de palavra, 130
 página, 129
 refresh, 129
 dual, 32, 90, 94
 dualidade, 32
duty cycle, 213

E

EEPROM, 127
 endereço, 74
 energia, 95, 100, 107–110
 enviesado, relógio, *veja skew*
 EPROM, 127
 equação característica do FF, 182
 equivalência, *veja* $\Leftrightarrow$
 erro,
 de representação, 23
 especificação, 42
 estado, 172
 estado atual, 185, 197, 200
 exponenciação, 251
 expressões, 36

F

fan-in, 104–107, 111, 159
fan-out, 79, 81, 104–107, 111, 115, 161, 162
 fatorial, 260
 fechamento, 31
 FET, 86
Field Effect Transistor, *veja* FET
Field Programmable Gate Array, *veja* FPGA
 FIFO, 258
 fila, 257–260
 circular, 258
 filtro digital de ruído, 179
flip-flop, 177–183
 adjacentes, 186
 comportamento, 182
 destino, 186
 fonte, 186, 195
 mestre-escravo, 178
 modelo VHDL, *veja* VHDL, *flip-flop*
 temporização, 181
 tipo JK, 182

 tipo T, 180, 182
 um por estado, *veja* um FF por estado
 folga de *hold*, 187
 folga de *setup*, 187
 forma canônica, 46
 FPGA, 183
 frações, *veja* ponto fixo
 frequência, 198
 frequência máxima, *veja* relógio
 função, 30
 tipo, 29, 41
 função de próximo estado, 221, 228
 função de saída, 221, 228
 função, aplicação bit a bit, 32
 função, tipo (op. infixo), *veja* $\mapsto$

G

glitch, *veja* transitório
 GND, 88
 gramática, 17

H

handshake, 244
 hexadecimal, 19
hold time, 177, 185–191, 233
 folga, 187, 192, 193

I

idempotência, 31
 identidade, 31
 igualdade, 30
 implementação, 42
 implicação, *veja* $\Rightarrow$
 informação, 16
 inicialização,
 flip-flop, 182
 Instrução,
 busca, *veja* busca
 instrução, 12
 busca, *veja* busca
 decodificação, *veja* decodificação
 execução, *veja* execução
 resultado, *veja* resultado
 interface,
 de rede, 13
 de vídeo, 12
 inversor, 91
 tempo de propagação, 104
 involução, 31, 58
 isolante, 82

J

Joule, 107

L

latch, *veja* basculo
latch FF, *veja flip flop*, destino
launch FF, *veja flip flop*, fonte
 Lei de Joule, 108
 Lei de Kirchhoff, 101

Lei de Ohm, 100, 101
 LIFO, 255
 ligação,
 barramento, 135
 em paralelo, 88, 94
 em série, 88, 94
 linguagem,
 assembly, veja ling. de montagem
 Z, 27
 linha de endereçamento, 75
 literal, 38
 logaritmo, 43
 lógica restauradora, 120

M

MADD, 189
 Mapa de Karnaugh, 47, 119
 máquina de estados, 221, 224
 Mealy, 226, 232, 243, 260
 Moore, 225, 231, 243, 256
 projeto, 228
 Máquina de Mealy, veja máq. de estados
 Máquina de Moore, veja máq. de estados
 máscara, 32
 máximo e mínimo, 31
 maxtermo, 45
 Mealy, veja máq. de estados
 memória, 171
 atualização, 74
 bit, 173
 de vídeo, 13
 decodificador de linha, 77
 endereço, 74
 FLASH, 128
 matriz, 124, 129
 multiplexador de coluna, 77
 primária, 13
 RAM, 78, 128, 254
 ROM, 75, 121, 233
 secundária, 13
 memória dinâmica, veja DRAM
 memória estática, veja SRAM
 metaestabilidade, 173, 177, 180, 183
 defesa contra as artes das trevas, 180
 microcontrolador, 232–240
 microrrotina, 240
 mintermo, 45, 119, 121
 modelo,
 funcional, 43
 porta lógica, 91
 temporização, 110
 módulo de contagem, 209
 módulo, veja %, mod
 Moore, veja máq. de estados
 MOSFET, 86
 multiplexador, 58, 63–67, 77, 96, 112–114, 118–
 119, 121, 129, 135, 136
 multiplicação, 162–168, 249
 acumulação de produtos parciais, 163–168

multiplicador,
 somas repetidas, 249, 260
multiply-add, veja MADD

N

número,
 de Euler, 24
 negação, veja $\neg$
 nível lógico,
 0 e 1, 28
 indeterminado, 28, 104, 111, 135
 terceiro estado, 134
 nó, 91
 not, veja $\neg$
 número primo, 51

O

octal, 18
 onda quadrada, 177, 198
 operação,
 binária, 29
 bit a bit, 32
 infixada, 42
 MADD, 189
 prefixada, 45
 unária, 29
 operação apropriada, 186
 operações sobre bits, 29–33
 operador,
 binário, 29
 lógico, 36
 unário, 29
operation code, veja opcode
 or, veja $\vee$
or array, 124
 ou exclusivo, veja $\oplus$
 ou inclusivo, veja $\vee$
output enable, 255
overflow, 142–144, 148, 156, 163, 168

P

paridade,
 ímpar, 47
 par, 47
 período mínimo, veja relógio
 pilha, 255–257, 260
pipelining, veja segmentação, 191
 piso, veja [v]
 ponto fixo, 145
 ponto flutuante, 67
pop, 255
 porta,
 de escrita, 252
 de leitura, 251
 porta lógica, 62
 and, 56
 carga, veja fan-out
 de transmissão, 136, 173
 nand, 57, 94
 nor, 57, 93

not, 56, 91
or, 56
xor, 57, 62, 180
 portas complexas, 95
 potência, 107–110
 dinâmica, 109
 estática, 110
 potenciação, 42
 precedência, 30
 precisão,
 representação, 23
 preset, 182
 prioridade,
 decodificador, 71
 processador, 12
 produtivo, 33
 programa de testes, veja VHDL, testbench
 PROM, 127
 propriedades, operações em $\mathbb{B}$, 31
 protocolo,
 de sincronização, 244, 260
 prova de equivalência, 40–41
 próximo estado, 185, 200
 pull-down, 91
 pull-up, 91, 121, 135
 pulso, 117, 118, 174, 183, 199, 207, 223, 248
 espúrio, veja transitório
 push, 255

R

raiz quadrada, 261
 RAM, 12, 74, 78, 128–134
 célula, 78
 dinâmica, 128
 Random Access Memory, veja RAM
 RAS, 129
 Read Only Memory, veja ROM
 realimentação, 78
 receptor, 83
 rede, 91
 redução, 33
 refresh, 132, 134
 Register Transfer Language, veja RTL
 registrador, 183, 220, 238, 251
 carga paralela, 184
 de segmento, 190
 simples, 184
 registrador de deslocamento, 216–220
 modelo VHDL, veja VHDL, registrador
 paralelo-série, 217
 série-paralelo, 216
 universal, 219
 registrador de estado, 185
 relógio, 175, 177, 198–213
 bordas,
 ascendente, 178
 descendente, 178
 ciclo de trabalho, 213
 enviesado, veja skew

 multi-fase, 213
 período mínimo, 187
 representação,
 abstrata, 28
 binária, 20
 complemento de dois, 141
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 ponto fixo, 145
 posicional, 17
 precisão, 23
 reset, 174, 182, 184
 resistência, 84, 95, 100
 ROM, 12, 74–77, 121–128
 rotação, 152, 156
 Row Address Strobe, veja RAS
 RTL, 191

S

segmentação, 189
 seletor, 69
 semântica, 17
 semicondutor, 82
 tipo N, 83
 tipo P, 84
 set, 174, 182
 setup time, 177, 185–191, 233
 folga, 187, 192, 193
 silogismo, 37
 simplificação de expressões, 38–40
 sinal, 27, 41
 analógico, 27
 digital, 27, 28
 fraco, 120, 121, 133, 136
 intensidade, 86, 134
 restaurado, 120, 134
 síntese, veja VHDL, síntese
 skew, 191–195
 Solid State Disk, veja SSD
 soma, veja somador
 soma de produtos, 45, 49, 121
 somador, 139, 146–147
 adiantamento de vai-um, 157, 220
 cadeia, 146
 completo, 99, 146
 parcial, 98
 seleção de vai-um, 161
 serial, 219
 temporização, 190
 teste, 168
 somatório, 33
 spice, 28
 SRAM, 132–134
 SSD, 14
 status, 155
 subtração, 147–148
 superfície equipotencial, 91, 105

T

tabela,
 de excitação dos FFs, 182
 tabela verdade, 33–35, 44
 tamanho, veja $|N|$
 tempo,
 de contaminação, 110, 113–116, 173, 181
 de estabilização, 177, 181
 de manutenção, 177, 181
 de propagação, 54–55, 58, 61–62, 70, 74, 80, 97,
 99, 104, 110–112, 181, 195, 210, 211
 discretizado, 175, 177, 181, 186, 210
 temporização, 99–121
 tensão, 100
 Teorema,
 DeMorgan, 32, 41, 46, 52, 57, 58, 89, 93
 Dualidade, 94
 Simplificação, 47
 terceiro estado, 134–136
 testbench, veja VHDL, testbench
 teste,
 cobertura, 169
 de corretude, 168
 teto, veja $\lceil r \rceil$
 three-state, veja terceiro estado
 tipo,
 função, 29
 tipo de sinal, 41
 Tipo I, veja formato
 Tipo J, veja formato
 Tipo R, veja formato
 transferência entre registradores, veja RTL
 transistor, 84–86, 90–91
 CMOS, 90
 corte, 109
 gate, 84
 saturação, 109
 sinal fraco, 86
 tipo N, 85
 tipo P, 86
 Transistor-Transistor Logic, veja TTL
 transitório, 116–118, 174, 175, 227
 transmissão,
 serial, 218
 transmission gate, veja porta de transmissão
 TTL,
 74148, 71
 74163, 206
 74374, 220
 tupla, veja $\langle \rangle$
 elemento, 32
 largura, 43

U

ULA, 154–156, 170, 251
 status, 155
 um FF por estado, 240–243
 Unidade de Lógica e Aritmética, veja ULA

V

valor da função, 30
 VCC, 88
 vetor de bits, veja $\langle \rangle$, 32
 largura, 43
 VHDL, 183
 design unit, veja VHDL, unidade de projeto
 síntese, 195
 std_logic, 134
 tipos, 41

W

Watt, 107
 write back, veja resultado

X

xor, veja $\oplus$

Z

Z, linguagem, 27

Capítulo 9

Uma Breve Introdução à VHDL

The limits of my language means the limits of my world.
Ludwig Wittgenstein.

Minha pátria é minha língua.
Língua, Caetano Veloso.

A linguagem VHDL foi desenvolvida no início da década de 1980 como parte de um programa de pesquisa do Departamento de Defesa dos EUA, chamado de *Very High Speed Integrated Circuits*. Este programa visava o desenvolvimento de circuitos eletrônicos de alta velocidade e um de seus resultados foi a especificação, padronização, e posterior implementação da linguagem *Very High Speed Integrated Circuit Hardware Description Language* ou VHSIC-HDL, ou o mais palatável VHDL. Este capítulo¹ contém uma breve introdução aos elementos da linguagem necessários para a compreensão dos circuitos modelados com VHDL.

Ao contrário de linguagens de programação como C ou *assembly*, nas quais os comandos/instruções são executados na ordem em que aparecem no código fonte, em VHDL o cômputo das expressões e as atribuições de valores ocorrem em paralelo, com um efeito que emula a propagação de sinais num circuito. Neste sentido, o código VHDL é declarativo, ao invés de imperativo, como em C. A diferença entre estes estilos, imperativo e declarativo, ficará mais clara no que se segue.

As ambições do autor, com este capítulo, são particularmente modestas. O que se pertence é apresentar um subconjunto pequeno de VHDL, que é o mínimo necessário para permitir a modelagem dos circuitos estudados neste texto, visando a simulação destes circuitos em atividades de laboratório. Para quem deseja se aprofundar na linguagem, o texto mais completo e o com melhor didática de que o autor tem ciência é o de Ashenden [Ash08].

A leitura deste capítulo deve acompanhar a execução das tarefas de laboratório que correspondem ao material apresentado aqui. Os exemplos no texto podem ser insuficientes para a compreensão dos conceitos e os laboratórios foram projetados para complementar as leituras. Para efetuar as tarefas dos laboratórios é necessário um compilador de VHDL e um visualizador de formas de onda. Com nossas turmas empregamos o compilador *ghdl* e o visualizador *gtkwave*. Existem sistemas comerciais que oferecem a mesma funcionalidade, em versões disponíveis para os sistemas operacionais mais populares.

¹© Roberto André Hexsel, 2012-2021. Versão de 1 de março de 2021.

Ao contrário de C, que não tem um óbvio competidor em seu nicho de aplicação, *Verilog* é outra linguagem de descrição de *hardware* com amplo emprego na indústria. O código fonte Verilog se parece com o da linguagem C ao passo que VHDL se parece mais com Pascal ou Ada. A expressividade e a funcionalidade de Verilog são similares àsquelas de VHDL e a escolha dentre as duas me parece ser uma questão de gosto e/ou de fé religiosa.

9.1 A linguagem VHDL

VHDL é uma sigla que contém uma sigla: *VHSIC Hardware Description Language*, sendo VHSIC abreviatura para *Very High Speed Integrated Circuits*. Como o nome diz, VHDL é uma linguagem de **descrição** de *hardware*. Guarde bem essa sigla, *HDL*, que abrevia *Hardware Description Language*.

Diferentemente de C, um programa em VHDL *descreve* uma estrutura, ao invés de *definir* ou *prescrever* uma computação. Neste sentido, um programa em C é uma “receita de bolo” com ingredientes (variáveis) e modo de preparar (algoritmo). Em VHDL um modelo estrutural pode descrever a interligação de componentes, ou pode especificar sua função sem mencionar uma implementação daquela função.

A partir da estrutura, um compilador como ghdl gera um simulador para o circuito descrito pelo código fonte. VHDL também permite descrever a função ou o comportamento de um circuito, sem que seja necessário definir sua estrutura.

Um sintetizador é um compilador que gera uma descrição de baixo nível – portas lógicas ou transistores – para que um circuito integrado seja fabricado a partir da descrição.

9.1.1 Descrição gráfica de circuitos

Examine o diagrama da Figura 9.1 e liste, de forma sucinta, as convenções empregadas para que ele possa ser interpretado por alguém que tenha estudado Sistemas Digitais. Relembre DeMorgan.

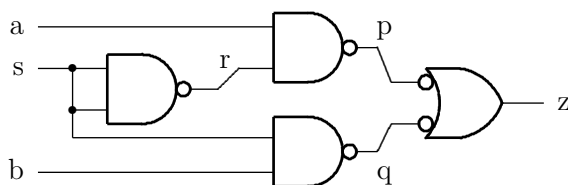


Figura 9.1: Diagrama de um circuito combinacional.

Sua lista deve conter, pelo menos:

- (i) uma convenção para representar as portas lógicas;
- (ii) uma convenção para representar as ligações;
- (iii) uma convenção para representar as entradas;
- (iv) uma convenção para representar as saídas; e
- (v) uma convenção para nomear os valores lógicos transportados pelas ligações.

Até agora temos usado várias *convenções*:

- * portas lógicas tem formatos distintos (*not*, *and*, *or*, *xor*);
- * multiplexadores e decodificadores são trapézios;
- * somadores são trapézios com um recorte/chanfro;
- * registradores são retângulos com o triângulo do relógio;
- * circuitos complexos são caixas retangulares;
- * ligações (fios, sinais) são linhas;
- * sinais fluem da esquerda para a direita;
- * entradas ficam no lado esquerdo (ou acima);
- * saídas ficam no lado direito (ou abaixo);
- * se não é óbvio, setas indicam o fluxo dos sinais; e
- * um traço curto com um número indica largura do sinal, se maior do que 1.

O diagrama na Figura 9.2 ilustra estas convenções: são quatro os registradores, chamados de *r1*, *r2*, *r3* e *STAT*, os sinais fluem da esquerda para a direita, o sinal *clock* tem um bit de largura, o sinal *oper* e a entrada do registrador *STAT* tem 4 bits de largura, e os demais sinais tem largura de 16 bits. O circuito no centro é uma unidade de lógica e aritmética. O comentário acima do diagrama, é uma indicação da função do circuito, num idioma chamado de *linguagem de transferência entre registradores* (*register transfer language* ou *RTL*). O comentário indica que, a cada borda do relógio, o resultado da soma dos conteúdos de *r1* e *r2* é atribuído (transferido) para *r3*.

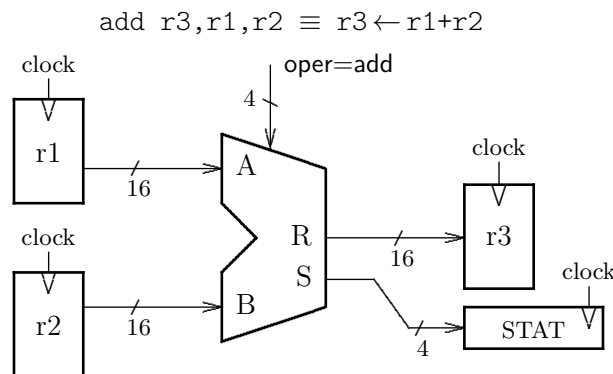


Figura 9.2: Exemplo das convenções usadas em diagramas de circuitos.

9.1.2 E se a descrição for textual?

A interpretação automática de desenhos é uma arte, mesmo que os desenhos sejam diagramas tão simples como nossos circuitos. Para facilitar a interpretação automática de representações para circuitos emprega-se texto.

Circuitos são representados em VHDL por *sinais* (fios) e *componentes* (*design units*). Um componente é descrito por duas construções da linguagem, uma *entidade* – que descreve sua interface com o mundo externo – e uma *arquitetura* – que descreve seu comportamento através das construções da linguagem.

No que segue, empregaremos as seguintes convenções de tipografia para o código VHDL: palavras reservadas são grafadas em negrito (**entity**), nomes de sinais e componentes são grafados sem serifa (`meuSinal`), e comentários iniciam com ‘`--`’ e terminam no final da linha e são grafados com *-- caracteres inclinados*.

Uma entidade (**entity**) descreve as ligações externas do componente:

- (i) entradas são **in**
- (ii) saídas são **out**
- (iii) ligações são representadas por *sinais*, no nosso caso bits (`bit`) ou vetores de bits (`bit_vectors`).

Uma arquitetura (**architecture**) descreve as ligações internas do componente. A arquitetura declara quais componentes são usados na implementação, que sinais internos são necessários para interligar os componentes internos, e de que forma os componentes são interligados entre si e com os sinais da entidade.

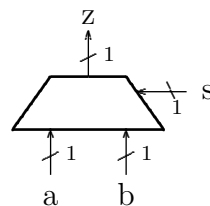
Uma certa entidade pode ser modelada em VHDL de mais de uma forma, como veremos em breve – para uma entidade (interface), podem ser projetadas mais de uma versão da arquitetura (circuito).

MUX2 – entidade e arquitetura

O multiplexador da Figura 9.1 é descrito em VHDL pelas suas entidade e arquitetura. A entidade, mostrada no Programa 9.1, é chamada `mux2` e descreve a *interface* deste componente, com três sinais de entrada e um de saída, todos com a largura de um bit.

Programa 9.1: Exemplo de entidade.

```
entity mux2 is
  port (a, b : in    bit;
        s : in    bit;
        z : out   bit);
end mux2;
```



A entidade tem nome `mux2`. A associação entre uma entidade (interface) e sua arquitetura (implementação) se dá no comando de definição da arquitetura. No trecho de código abaixo, a entidade tem nome `mux2` e a implementação é chamada de *estrutural* – que é a versão *estrutural* da entidade `mux2`.

```
entity mux2 is
  ...
end mux2;

architecture estrutural of mux2 is
  ...
begin
  ...
end architecture estrutural;
```

O código com a arquitetura do *mux-2* é mostrado no Programa 9.2. A arquitetura, chamada *estrutural*, define a *estrutura* do circuito modelado – ela descreve como os componentes são interligados. Entre as palavras chave **architecture** e **begin** devem ser declarados os componentes usados na implementação do multiplexador – neste caso inversor e porta *nand* – e os sinais internos necessários para ligar os componentes e a interface, que são os sinais *r*, *p* e *q*.

Entre o **begin** e o **end architecture** são definidas as ligações entre os componentes da implementação. Como mostra o diagrama ao lado do código, há uma correspondência direta entre o circuito e o código VHDL que o representa. Infelizmente, nas descrições textuais *todos* os sinais devem ser explicitamente nomeados.

A construção **port map** faz a ligação entre os sinais internos à arquitetura e os sinais declarados para cada componente individual usado no modelo. O **port map** é quem liga sinais em componentes ao mapear os sinais internos declarados na arquitetura aos sinais declarados nas entidades dos componentes. A instanciação dos componentes com o mapeamento das portas é similar a um operador aritmético prefixado; ao invés da forma usual, que é infixada ($c = a + b$), emprega-se a forma prefixada $+(c, a, b)$.

Programa 9.2: Exemplo de arquitetura.

```
architecture estrutural of mux2 is

  — declaração de componentes
  component inv is
    port(A : in bit; S : out bit);
  end component inv;

  component nand2 is
    port(A,B : in bit; S : out bit);
  end component nand2;

  — declaração de sinais internos
  signal r, p, q : bit;

begin

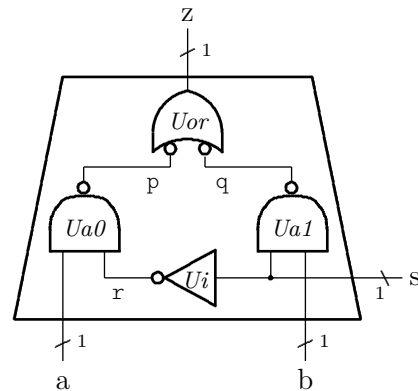
  — instanciação e
  — interligação dos componentes
  Ui: inv port map(s, r);

  Ua0: nand2 port map(a, r, p);

  Ua1: nand2 port map(b, s, q);

  Uor: nand2 port map(p, q, z);

end architecture estrutural;
```



Entre **architecture** e **begin**, os componentes e os sinais são *declarados*. Entre o **begin** e o **end architecture**, os sinais e componentes são *instanciados* e conectados através dos sinais internos e do mapeamento dos sinais nas portas dos entidades com o **port map**. O **port map** associa, ou ‘gruda’, os sinais nos terminais do componente.

9.1.3 Projeto *top-down*

O projeto de sistemas digitais tende a ser complexo por envolver uma quantidade enorme de informações a respeito dos componentes do sistema, sobre as interconexões entre aqueles, bem como sobre os detalhes da implementação de cada componente. Considere um microprocessador comercial de 32 bits que seja relativamente sofisticado – tais circuitos são implementados com cerca de um milhão de transistores [HP12]. Supondo, para simplificar o cálculo, que cada porta lógica e cada *flip-flop* seja implementado com 5 transistores, este projeto empregaria algo como 200 mil componentes. É impossível a um humano lidar com tanta informação e portanto empregam-se diferentes níveis de abstração no projeto de sistemas tão complexos. Por exemplo, no nível de abstração que trata das unidades funcionais do circuito de dados de um processador, tais como somadores, ULA, e bloco de registradores, todos os detalhes sobre a implementação destas unidades no nível de portas lógicas são abstraídos e escondidos do projetista da CPU. O projetista das unidades funcionais preocupa-se com as portas lógicas, mas pode abstrair, e em larga medida ignorar, os detalhes da implementação das portas lógicas no nível dos transistores.

Um dos mecanismos mais eficazes para permitir o uso das abstrações mencionadas no parágrafo anterior é o uso de modelos para descrever os subsistemas e suas interfaces. Neste contexto, um *modelo* consiste de alguma forma de codificação do nosso conhecimento sobre o sistema modelado. Se os modelos descrevem os sistemas formalmente, como as especificações dos circuitos dos Capítulos 4 e 8, então estas podem ser usadas como um contrato entre projetista de um componente e seus usuários. Neste caso, o contrato define as interfaces e o comportamento do componente. Se o modelo for descrito utilizando-se de um formalismo adequado, então este pode ser usado como documentação do projeto. Se o formalismo permitir escrever modelos que sejam executáveis em simuladores, então o modelo executável pode ser usado para testar a validade da especificação, e o mapeamento entre versões nos distintos níveis de abstração, bem como para verificar se as descrições nos diferentes níveis de abstração satisfazem às mesmas especificações. Um modelo executável, ao menos em teoria, pode ser traduzido por um compilador adequado para então permitir a síntese de um circuito integrado. A linguagem VHDL foi projetada para ser empregada em todas estas formas de modelagem.

Uma forma de descrever o “ciclo de vida” do desenvolvimento de um sistema é com a chamada *espiral de projeto*, mostrada na Figura 9.3. Tipicamente, o processo se inicia com uma ideia, e a partir dela escreve-se uma *especificação* (abstrata) para o sistema. Esta especificação é *refinada* para um projeto, que essencialmente é uma versão mais detalhada e menos abstrata da especificação. A *implementação* do projeto resulta numa versão concreta da especificação, e esta pode então ser *verificada* quanto a sua corretude e avaliada quanto ao seu desempenho. Se a implementação não satisfaz à especificação, deve-se efetuar um novo ciclo da espiral, com tantas repetições quantas necessárias até que os requisitos de projeto sejam atendidos.

Da forma como VHDL é empregada nesse texto, (i) a *ideia* é a especificação em Português de um circuito, (ii) o *projeto* é o primeiro refinamento da especificação, que é obtido com trabalho intelectual, lápis e papel; (iii) a implementação é a *codificação* em VHDL, e (iv) a verificação é a confirmação de que a implementação satisfaz ao especificado, através de *simulação*.

Um projeto pode ser refinado de mais de uma forma, na sua fase inicial, e a cada refinamento, o resultado é mais concreto e próximo da implementação. Por exemplo, partindo-se de uma especificação (abstrata) do comportamento desejado, esta sofre um refinamento que resulta num *diagrama de blocos*, que ao ser refinado produz o *projeto dos blocos*, o refinamento dos

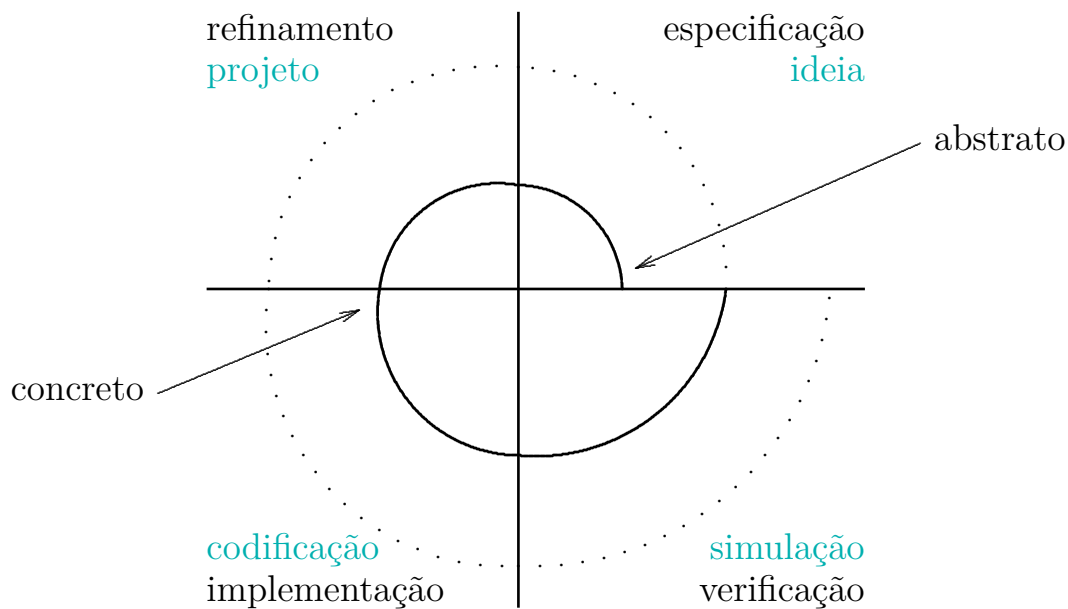


Figura 9.3: Espiral de projeto.

blocos resulta na *formalização da especificação*, que é a modelagem *concreta* dos componentes do projeto em VHDL, gerando-se código VHDL para os *componentes* do projeto.

9.2 Modelagem de circuitos combinacionais em VHDL

VHDL é uma linguagem extremamente versátil e é usada para

- * a modelagem e simulação de circuitos;
- * especificação, projeto e síntese de circuitos;
- * descrever listas de componentes e de ligações; e
- * verificação de corretude: se especificação $\Rightarrow$ implementação.

A linguagem é padronizada pelo IEEE, no padrão IEEE 1076-1987 V[HSIC] H D L, ou *Very High Speed Integrated Circuit Hardware Description Language*. O padrão IEEE Std 1164 define pacote `std_logic`, e o padrão IEEE Std 1076.3 VHDL Numeric Standard define uma biblioteca de funções numéricas. Há mais, muito mais, nos bons textos do ramo tais como [Ash08, PT96].

9.2.1 Tipos de dados

Alguns dos tipos básicos de dados disponíveis em VHDL são mostrados na Tabela 9.1. O tipo `bit` tem dois valores, '0' e '1' – note as aspas simples. Um vetor de bits, ou um `bit_vector`, é representado por mais de um bit, enfeitado com aspas duplas: "1001". As regras para caracteres (aspas simples) e cadeias (aspas duplas) são similares àquelas para bits e vetores de bits. Constantes booleanas, números inteiros, números reais e constantes de tempo são representadas sem aspas.

A atribuição a um sinal é representada pela *flecha dupla* ' $\leq$ ' e a concatenação de vetores de bits, ou de caracteres, é representada pelo '&', como mostra o exemplo da concatenação das

TIPO	VALOR	EXEMPLO DE USO
bit	'1', '0'	D <= '1';
bit_vector	vetor de bits	Dout <= "0011";
boolean	TRUE, FALSE	start <= TRUE;
integer	-2,-1,0,1,1	Cnt <= Cnt + 17;
real	2.3, 2.3E-4	med <= sum/16.0;
time	100 ps, 3 ns	sig <= '1' after 12 ns;
character	'a','1','@'	c <= 'k';
string	vetor de char	msg <= "error: " & "badAddr";

Tabela 9.1: Subconjunto pequeno dos tipos de dados em VHDL.

cadeias de caracteres.

Atenção porque isso é importante: VHDL **não** é *case sensitive* – maiúsculas e minúsculas são consideradas iguais.

Sinais e Vetores de Sinais

Os *sinais* em VHDL correspondem aos fios que transportam os bits no circuito e a linguagem define sinais de um bit, e vetores de sinais com muitos bits.

O código abaixo declara um sinal *x* com um bit de largura, e este é inicializado em '1'. O sinal *vb* é um vetor com 4 bits de largura e a posição de cada um dos bits é aquela em que o bit mais significativo está à esquerda, e o menos significativo à direita. Esta é a declaração adequada para vetores que representam números.

```
— declaração de sinais do tipo bit
signal x: bit := '1'; — inicializado em '1'
signal vb: bit_vector(3 downto 0) := "1100"; — inicializa em 12
```

O sinal *bv* é declarado com a direção contrária àquela do sinal *vb*, e o bit mais significativo deste vetor é aquele na direita.

```
signal bv : bit_vector(0 to 3); — bit 0 é o Mais Significativo
```

As atribuições abaixo produzem resultados que talvez não sejam intuitivos a uma primeira vista.

```
signal bH : bit_vector(7 downto 0); — bit 7 é Mais Significativo
signal bL : bit_vector(0 to 7); — bit 0 é Mais Significativo
```

```
bH <= b"11000000"; — bH se torna 0xc0 = 192
bL <= b"11000000"; — bL se torna 0x03 = 3
```

É possível selecionar um subconjunto dos bits de um sinal. Os comandos abaixo declaram 4 sinais: um com 1 bit, um com 8 bits, e dois sinais com 4 bits de largura. A primeira atribuição seleciona o bit mais significativo do sinal *v8* e o atribui a *x*. As atribuições a *v4* e *t4* selecionam o quarteto central de *v8*, e armazenam esses quartetos na *ordem numérica* em *v4*, e na *ordem invertida* em *t4*.

```
signal x: bit; — um bit
signal v8: bit_vector(7 downto 0); — vetor de 8 bits
```



```

signal v4: bit_vector(3 downto 0); — vetor de 4 bits
signal t4: bit_vector(3 downto 0); — vetor de 4 bits
...
x <= v8(7);           — atribuição do bit mais significativo
v4 <= v8(5 downto 2); — atribuição do quarteto central de v8
t4 <= v8(2 to 5);     — mesmo quarteto, na ordem inversa

```

O operador que seleciona um ou mais bits de um vetor é um par de parenteses (`v8(7)`), ao invés de um par de colchetes como em C ou Pascal. Em VHDL é possível selecionar um subconjunto com vários elementos de um vetor (`v8(2 to 5)`).

Vetores de bits podem ser representados nas bases hexadecimal (com prefixo `x`), octal (prefixo `o`) e binária (prefixo `b`). A representação binária implícita é a normal: se nada for dito em contrário, o vetor é um vetor de bits.

```

hexadecimal <= x"C0";      — 1100 0000 OITO bits
octal        <= o"300";    — 011 000 000 NOVE bits
binaria_expl <= b"11000000"; — 11000000 base-2 explícita
binaria_impl <= "11000000"; — 11000000 base-2 implícita

```

9.2.2 Abstração em VHDL: *Packages*

O Programa 9.3 mostra o código VHDL que define uma série de nomes para sinais com mais de um bit. Este conjunto de definições é chamado de *pacote*, ou **package** e pode ser armazenado em um arquivo que é compilado separadamente dos demais arquivos com código fonte. Uma vez que o pacote seja compilado, suas definições podem ser usadas em todas as demais unidades de projeto. Pacotes têm função similar aos arquivos `.h` da linguagem C.

Programa 9.3: Um pacote com abreviaturas.

```

package p_WIRES is — abreviaturas para
                  — novos tipos e agrupamentos de sinais

  subtype natur8  is integer range 0 to 255; — natural de 8 bits
  subtype natur16 is integer range 0 to 65535; — natural de 16 bits

  subtype reg2   is bit_vector(1 downto 0); — par de bits
  subtype reg4   is bit_vector(3 downto 0);
  subtype reg8   is bit_vector(7 downto 0);
  subtype reg15  is bit_vector(14 downto 0);
  subtype reg16  is bit_vector(15 downto 0); — barramento de 16 bits

package body p_WIRES is

  — o corpo deste pacote é vazio

end package p_WIRES;

```

Um tipo restringido de dados, tal como um inteiro de 8 bits pode ser declarado como sendo um subtipo de `integer`, com a declaração **subtype** mais a faixa de valores da restrição. Por exemplo, a declaração

```

subtype natur8 is integer range 0 to 255;

```

define um novo tipo, chamado de `natur8`, que é um número natural com valores no intervalo $[0, 255]$. Sinais de tipo `natur8` podem ser usadas nos modelos e o compilador verifica se os valores atribuídos a eles estão dentro do intervalo correto. A atribuição do valor 300 a um sinal de tipo `natur8` resulta num erro de compilação ou de simulação.

Outra aplicação dos tipos restringidos é na definição de abreviaturas para tipos de sinais. Por exemplo, os subtipos `reg2` a `reg16` definem nomes curtos para vetores de bits com larguras 2 a 16. O nome curto `reg16` pode ser usado no lugar do verboso `bit_vector(15 downto 0)` para declarar sinais com 16 bits de largura.

O corpo do pacote, entre o **package body** e o **end package** é vazio. A Seção 9.4.13 mostra como definir funções no corpo do pacote.

Para que as definições contidas no pacote possam ser usadas, a declaração da entidade deve ser precedida do nome do pacote, como mostra o Programa 9.4. O conteúdo do pacote é compilado e depositado na biblioteca chamada `work`. A declaração

```
use work.p_wires.all;
```

indica que a entidade fará uso das definições do pacote `p_wires`, e mais que isso, todas (`.all`) as definições devem estar disponíveis: `p_wires.all`. A alternativa a *tudo* é explicitar *uma* definição para ser usada na unidade de projeto, tal como os naturais de 16 bits `p_wires.natur16`.

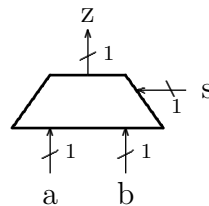
9.2.3 Modelo estrutural do multiplexador

Um *modelo estrutural* descreve a estrutura de um circuito, do ponto de vista das interconexões entre seus componentes. Dependendo das ferramentas disponíveis, um circuito pode ser sintetizado a partir da sua descrição no nível RTL ou do nível estrutural, porque estas descrições são suficientemente concretas e detalhadas para permitir a tradução automática e eficiente da descrição em VHDL para uma lista de componentes e ligações entre aqueles, que é então usada para produzir o circuito integrado. Esta lista de ligações e componentes é chamada de *net list*.

A *entidade* do multiplexador de duas entradas é mostrada abaixo. O circuito tem duas entradas `a` e `b`, uma entrada de seleção `s` e uma saída `z`, todas de um bit. Um diagrama convencional da interface do `mux-2` é mostrado ao lado da entidade.

Programa 9.4: Entidade do `mux2`.

```
use work.p_wires.all;
entity mux2 is
  port(a,b : in  bit;
        s : in  bit;
        z : out bit);
end mux2;
```



Um *modelo estrutural* do `mux2` é mostrado no Programa 9.5. Um diagrama convencional da interligação dos componentes do `mux-2` é mostrado ao lado da arquitetura. Numa implementação óbvia do `mux-2` são necessários três sinais internos, `r` com o complemento de `s`, e `p`, `q` para interligar as portas `and` e a porta `or`.

No Programa 9.5, os componentes e os sinais internos são *declarados* entre o **architecture** e o **begin**. Os componentes são *instanciados* entre o **begin** e o **end architecture**.

Na instanciação, cada componente tem um *label* (opcional) – `Uxx`: que significa *design Unit*

xx – e o mapeamento das portas do componente com os sinais da interface (declarados na entidade) e os sinais internos (declarados na arquitetura). Por exemplo, na instanciação

```
Ua0: nand2 port map(a,r,p)
```

a porta A do componente *nand2* é ligada ao sinal de interface *a*, a porta B é ligada ao sinal interno *r*, e a porta de saída *S* é ligada ao sinal interno *p*. Este mapeamento é chamado de *mapeamento posicional* porque os sinais da arquitetura são associados às portas do componente na ordem – na mesma posição – em que as portas são declaradas.

Programa 9.5: Arquitetura do *mux2*.

```
architecture estrutural of mux2 is
```

```
— declaração dos componentes
```

```
component inv is
```

```
port(A: in bit; S: out bit);
```

```
end component inv;
```

```
component nand2 is
```

```
port(A,B: in bit; S: out bit);
```

```
end component nand2;
```

```
— declaração sinais internos
```

```
signal r, p, q: bit;
```

```
begin
```

```
— início da área concorrente
```

```
— instanciação dos componentes
```

```
Ui: inv port map (s, r);
```

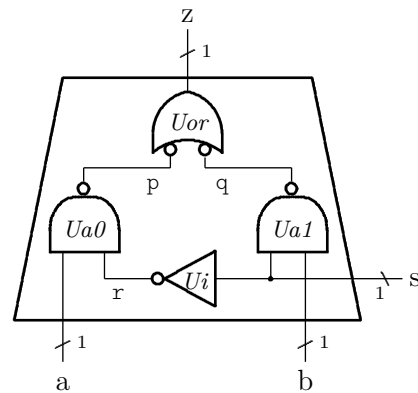
```
Ua0: nand2 port map (a, r, p);
```

```
Ua1: nand2 port map (b, s, q);
```

```
Uor: nand2 port map (p, q, z);
```

```
— fim da área concorrente
```

```
end architecture estrutural;
```



A região entre o **begin** e o **end architecture** é chamada de *área concorrente* e os comandos nesta área são executados concorrentemente. Isso significa que o código dos componentes instanciados é simulado (executado) em paralelo, imitando o comportamento de um circuito real. Os quatro componentes do *mux-2* reagem a eventos nas suas entradas e possivelmente provocam eventos nas suas saídas.

Por isso que VHDL é dita *declarativa*: as quatro instâncias dos componentes (*Ui*, *Ua0*, *Ua1* e *Uor*) são declaradas numa ordem arbitrária no código fonte enquanto que a ocorrência de eventos nas saídas dos componentes depende somente da sequência de eventos nos sinais que os interligam. Um evento na saída de um componente pode provocar a execução simultânea de vários componentes. Este é o comportamento dos sinais num circuito físico.

Outra diferença entre VHDL e linguagens como C ou Pascal, que também é relacionada com VHDL ser declarativa, é que a instanciação de um componente representa uma *cópia* daquele componente. A instanciação não equivale a uma chamada de função numa linguagem imperativa como C.

VHDL provê outro mecanismo para a conexão de sinais declarados na entidade (nomes locais) a sinais da unidade de projeto externa (nomes externos). Esta forma de ligação é chamada de *associação nominal* e seu uso pode ser mais conveniente do que a associação posicional. Com associação nominal, as ligações entre os sinais internos e externos podem ser codificadas em qualquer ordem, ao invés da ordem estrita em que os sinais estão declarados na entidade do componente externo. O Programa 9.6 mostra um exemplo de associação nominal na instanciação de um componente com uma entrada e uma saída. No exemplo, a ordem dos sinais, de entrada e de saída, está invertida na instanciação da `unidExterna`. O operador da associação nominal é uma *flecha invertida*: `'=>'`.

Programa 9.6: Associação nominal de portas.

```
architecture exemplo of assocNominal

    component unidExterna is
        port(INPext : in bit; OUText : out bit);
    end component unidExterna;

    signal ENTRlocal, SAIDAllocal : bit;

begin
    ...
    U: unidExterna port map (SAIDAllocal => OUText, ENTRlocal => INPext);
    ...
end architecture exemplo;
```

9.2.4 Processo de compilação de VHDL

A compilação de código VHDL se dá em três fases: análise, elaboração e simulação. No nosso caso, o resultado da compilação é um programa em C gerado por `ghdl`, que simula o comportamento do modelo escrito em VHDL. Usaremos `gtkwave` para verificar o comportamento dos modelos dos circuitos.

Análise Na fase de análise, o compilador VHDL verifica a corretude da sintaxe e a semântica dos comandos. Na análise sintática, são sinalizados erros de grafia das palavras reservadas, sinais que não foram declarados, etc. Na análise da semântica, são verificados os tipos dos sinais, e se os operadores são aplicados a operandos com os tipos apropriados – somar dois números faz sentido, somar dois `ifs` não.

É possível efetuar a análise somente em partes de um modelo. Por exemplo, um par entidade-arquitetura (uma *design unit*) pode ser analisado em separado, e se nenhum erro é encontrado, o resultado da análise é armazenado numa biblioteca, para uso futuro. Com `ghdl`, essa biblioteca é mantida no arquivo `work-obj93.cf`, no diretório em que ocorre a compilação.

Elaboração Na fase de elaboração, o compilador constrói um modelo completo de toda a hierarquia do projeto. Reveja o código da arquitetura do `mux2` no Programa 9.5 – naquele trecho de código, os dois níveis da hierarquia são (i) a arquitetura do `mux2`, e (ii) as declarações dos componentes `inv` e `nand2`.

O compilador examina a arquitetura de mais alto nível da hierarquia e expande todas as declarações dos componentes instanciados naquela arquitetura. Para cada componente, o compilador examina sua arquitetura e expande todas as declarações lá instanciadas. A elaboração termina quando todas as declarações forem substituídas por atribuições – para ser mais preciso, a elaboração termina quando só restam processos e os sinais que os interligam. De forma muito simplificada, um *processo* é uma generalização para uma atribuição. Voltaremos a falar de processos na Seção 9.4.3.

Simulação Ao final da elaboração, o compilador `ghdl` traduz a estrutura de processos e os sinais que os interligam para um programa em C que, depois de compilado, permite simular a execução do modelo completo. Veja, na Seção 9.2.5, como ocorre a simulação.

A saída da simulação pode ser mostrada na tela, pode ser gravada em arquivos, ou pode gerar um conjunto de dados que representa um diagrama de tempo para ser exibido por um programa como `gtkwave`.

9.2.5 Mecanismo de simulação de VHDL

A simulação de modelos escritos em VHDL emprega a *Simulação de Eventos Discretos* – o tempo simulado progride em função de eventos que ocorrem em instantes determinados pela própria simulação.

Um *processo* é uma construção básica da linguagem e que estudaremos adiante. No trecho de código abaixo, cada uma das atribuições é um processo, embora o uso do nome ‘processo’ seja reservado para um comando específico, que veremos em breve.

Uma atribuição pode especificar o instante em que transação ocorrerá:

```
A <= 5;           — atribuição válida no próximo delta
X <= Y after 10 ns; — atribuição válida depois de 10ns
```

A simulação consiste de duas fases: uma fase de inicialização, seguida da fase com os passos de simulação.

Na *fase de inicialização* (i) todos os sinais são inicializados com os valores declarados, ou com os menores valores possíveis para os respectivos tipos dos sinais; (ii) o tempo simulado é inicializado em zero; e (iii) todos os processos (atribuições) são executados exatamente uma vez.

A atribuição a um sinal causa uma *transação*, e a atribuição será efetivada no próximo *delta* (Δt). Se o sinal muda de estado, então esse sinal sofre um *evento*. Um evento no sinal S causa a execução de processos que dependem de S .

A Figura 9.4 mostra uma representação da linha de tempo como uma sequência de intervalos δ_n . No δ_0 ocorre a atribuição de 5 para A, e em δ_1 o sinal A já carrega o valor 5. Em δ_0 a atribuição de X em Y é enfileirada e escalonada para ocorrer no instante $\delta_0 + 10ns$.

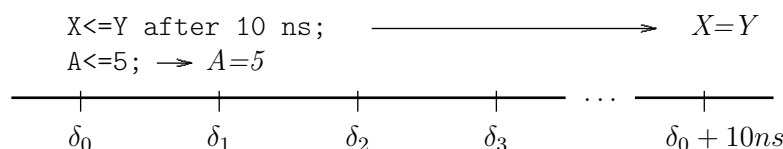


Figura 9.4: Linha de tempo como uma sequência de deltas.

Durante um *passo de simulação* (i) o tempo simulado avança até o próximo instante (δ) em que uma transação está programada; (ii) todas as transações programadas para aquele δ são executadas; (iii) estas transações podem provocar eventos em sinais; (iv) processos que dependem destes eventos são então executados; e (v) depois que todos os processos executam, a simulação avança até o δ em que a próxima transação está programada para ocorrer. Então repete-se o passo de simulação.

O tempo simulado avança em função de eventos nos sinais; a cada mudança no estado de um sinal corresponde um evento. Se ocorre um evento num sinal no lado direito de uma atribuição, a expressão do lado direito é reavaliada, e se houver mudança no estado do sinal representado pela expressão, o novo estado é atribuído ao sinal do lado esquerdo – causando um evento neste sinal. Vejamos um exemplo.

```
A <= 5;
B <= 4;
C <= A + B;
D <= C * 2;
```

As atribuições a A e a B ocorrem em δ_0 e provocam eventos em A e B. Em δ_1 , o lado direito de $C \leq A + B$; é avaliado e a alteração de valor em C provoca um evento naquele sinal. Em δ_2 , o lado direito de $D \leq C * 2$; é avaliado e a alteração provoca um evento em D. O sinal D valerá 18 em δ_3 ,

Note que, ao contrário de C, todas as atribuições que podem ocorrer num δ ocorrem ‘simultaneamente’, e *não na ordem em que aparecem no código fonte*. Como num circuito de verdade, todos os sinais que podem mudar num determinado instante, mudam naquele instante, e as consequências destas mudanças se propagam pelo resto do circuito.

Sinais e expressões são avaliados neste delta, mas eventos têm efeito no próximo delta – enquanto houver eventos no delta corrente, estes são avaliados e eventos resultantes são escalonados para o próximo delta. Eventos podem ser escalonados para instantes futuros com a cláusula **after**.

Considere o modelo de um somador completo, com modelagem da propagação dos sinais através do circuito. Um evento em A em δ_i pode causar um evento em Cout em δ_{i+1} , e em Sum em $\delta_i + 10ns$.

```
Sum <= A xor B xor Cin after 10 ns;
Cout <= (A and B) or (A and Cin) or (B and Cin);
```

9.2.6 O modelo está pronto, e agora? Testar, testar e testar

Os modelos escritos em VHDL, por si só, não podem ser verificados quanto à sua corretude. O programador dos modelos deve escrever um ou mais *programas de teste* (*testbenches*) para excitar os modelos e observar o seu comportamento. Estes programas são chamados de *testbenches* em Inglês porque são o análogo de uma bancada de testes equipada com geradores de sinais e osciloscópios. Em geral, *testbenches* são programas relativamente complexos porque: (i) se assemelham ao circuito externo àquele que está sob teste; (ii) devem gerar os estímulos apropriados para cobrir todos os casos de teste relevantes; e (iii) devem permitir a observação das respostas para que estas possam ser verificadas. Não é incomum que um programa de teste seja mais complexo do que o modelo que se deseja testar.

Iniciemos o projeto de um *mux-8* pela tabela verdade ao lado. São três as entradas e portanto as $2^3 = 8$ combinações de entradas devem ser verificadas, para garantir que a saída do circuito é a esperada.

O projetista de hardware tem duas obrigações distintas porém inseparáveis: (i) deve projetar um circuito que atenda ao especificado; e (ii) deve prover a garantia de que seu circuito atende à especificação.

Com base na tabela verdade – este não é o único método – o projetista gera um *vetor de testes* para exercitar o circuito, e confirmar que seu comportamento é o correto, segundo a especificação.

A especificação, junto com o vetor de testes, são o *contrato de compra e venda* do circuito, que garante ao *comprador* que o circuito cumpre o que foi prometido pelo *vendedor*.

s	a	b	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

9.2.7 Testbenches

Tipicamente, um arquivo com um *testbench* (TB, ou programa de teste)) contém código para verificar a corretude de modelos. Para simplificar a depuração do seu código VHDL, é uma boa ideia verificar cada novo modelo assim que seu código for completado. Um componente com N entradas tem uma tabela verdade com 2^N linhas; quanto menor for N , mais simples é o conjunto de testes necessário para garantir a corretude do modelo.

A entidade `tb_estrut`, mostrada no Programa 9.7 é vazia porque o programa de testes é auto-contido e não tem interfaces com nenhum outro componente. O TB é o “mundo externo” do componente por testar.

Programa 9.7: Entidade do *testbench* do *mux-2*.

```
use work.p_wires.all;
entity tb_estrut is
  — entidade vazia, não se comunica com "mundo externo"
end tb_estrut;
```

O código da arquitetura do *testbench* é demasiado longo e está dividido nos Programas 9.8, 9.9 e 9.10. O Programa 9.8 mostra a declaração do componente por testar, que é o *mux-2*.

Programa 9.8: Arquitetura do *testbench*, declaração dos componentes.

```
architecture TB of tb_estrut is
  — declaração dos componentes por testar
  component mux2 is
    port(A,B : in bit; S : in bit; Z : out bit);
  end component mux2;
```

Um *vetor de testes* é um conjunto bem definido de valores, que quando apresentados às entradas do modelo, produzem um conjunto bem definido de saídas. A tabela verdade da página 289 é um vetor de testes que define completamente o comportamento do *mux-2*.

A seguir descrevemos uma das formas de codificar e de aplicar vetores de teste na verificação de corretude do modelo do *mux-2*. A arquitetura do TB, no Programa 9.9, define um **record** que contém os valores necessários para excitar o modelo. Um **record** nada mais é do que uma

tupla com o número adequado de componentes. O registro `test_record_mux` possui três campos e os valores destes campos devem ser atribuídos de forma a gerar todas as combinações de entradas necessárias para garantir a corretude do modelo. O vetor de testes `test_array_mux` contém os oito elementos necessários para excitar e verificar o `mux-2`.

No `test_record_mux`, os quatro campos de tipo bit ('0') correspondem aos sinais da declaração do componente. Um novo tipo, `test_array_mux` é definido para denotar um vetor de `test_record_mux`. O tamanho do vetor não é definido *a priori* por causa do `positive range <>` – a faixa de valores é definida como sendo o número de elementos de cada instância de vetores de tipo `test_array_mux`. O vetor `test_vectors_mux` é declarado como uma constante e seu tamanho não é fixo mas depende de quantos elementos o projetista julgar necessários durante a execução dos testes.

Programa 9.9: Arquitetura do *testbench*, vetor de testes.

```

type test_record_mux is record
  s   : bit;      — entrada de seleção
  a,b : bit;      — entradas
  z   : bit;      — saída esperada
end record;

type test_array_mux is array(positive range <>) of test_record_mux;

constant test_vectors_mux : test_array_mux := (
  —s,   a,   b,   z
  ('0','0','0','0'), — linhas da tabela verdade
  ('0','0','1','0'),
  ('0','1','0','1'),
  ('0','1','1','1'),
  ('1','0','0','0'),
  ('1','0','1','1'),
  ('1','1','0','0'),
  ('1','1','1','1') );

```

O diagrama na Figura 9.5 mostra as ligações entre o modelo e o processo que percorre o vetor de testes e gera as sequências de *entradas de teste* e *saídas de teste*. As entradas de teste (a, b e s) excitam o modelo, que produz saídas de acordo com sua especificação. O comportamento esperado, segundo a especificação, é a coluna z de `test_vectors_mux` e este é comparado com os valores produzidos pelo modelo.

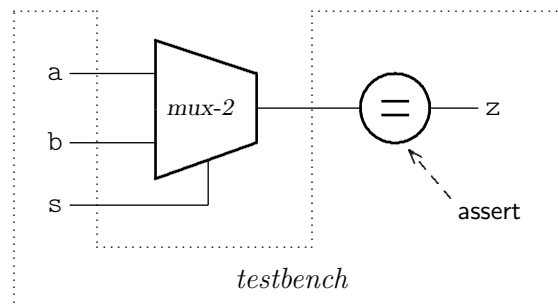


Figura 9.5: Ligações entre o *testbench* e o modelo do *mux-2*.

A sequência de valores de entrada para os testes do modelo é gerada pelo processo `U_testValues`, com o laço **for ... loop**. A variável de indução `i` itera no espaço definido pelo número de elementos do vetor de testes (`test_vectors` **'range'**) – o atributo **'range'** representa a faixa de valores do índice do vetor. Se mais elementos forem acrescentados ao vetor, o laço executará mais iterações. Processos e o comando **for** são discutidos nas Seções 9.4.3 e 9.4.11, respectivamente.

O `i`-ésimo elemento do vetor é atribuído à variável `v` e todos os campos do vetor são então atribuídos aos sinais que excitam o modelo. O processo `U_testValues` executa concorrentemente com o modelo do `mux-2` e quando os sinais de teste são atribuídos no laço, estes provocam alterações nos sinais do modelo.

Programa 9.10: Arquitetura do *testbench*, laço de testes.

```
architecture TB of tb_estrut is
  — declaração do componente por testar
  component mux2 is
    port(A,B : in bit; S : in bit; Z : out bit);
  end component mux2;

  — sinais do testbench
  signal sel, ea, eb, saida, esperada : bit;
begin

  U_mux2: mux2 port map(ea,eb,sel,saida); — componente por testar

  U_testValues: process — este componente efetua os testes
    variable v : test_record_mux;
  begin

    for i in test_vectors_mux'range loop
      v      := test_vectors_mux(i); — valores de teste
      sel    <= v.s; — bit de seleção
      ea     <= v.a; — entrada a
      eb     <= v.b; — entrada b
      esperada <= v.z; — saída cfe tabela verdade

      wait for 200 ps; — duração do passo de simulação

      assert (saida = esperada)
        report LF & "mux2:␣saida␣errada␣sel=" & B2STR(sel) &
          "␣saiu=" & B2STR(saida) & "␣esperada=" & B2STR(esperada)
          severity error;
    end loop;

    wait; — encerra a simulação

  end process U_testValues;

end architecture TB;
```

Depois que os valores do `i`-ésimo vetor de teste são atribuídos aos sinais, o simulador deve esperar para que os sinais estabilizem nas saídas do modelo. Neste exemplo, para simplificar a visualização dos sinais em um diagrama de tempo, o simulador deve esperar 200ps entre cada

mudança nos sinais nas entradas do modelo. Isso é obtido com o comando **wait for** 200 ps; que faz com que o tempo simulado progrida por 200ps antes da próxima atribuição à *v*. O uso do comando **wait** é discutido na Seção 9.4.4.

O comando **assert** é similar a um `printf()` em C e pode ser usado para exibir o valor de sinais ao longo de uma simulação. Este comando tem três cláusulas, a saber

- (i) **assert** condição – a *condição* deve ser **falsa** para que o simulador imprima a *string*;
- (ii) **report** *string* – texto informativo por imprimir; e
- (iii) **severity** nível – a severidade pode ser um de quatro *níveis*, *note*, *warning*, *error*, *failure*, e esta última aborta a simulação.

O **assert** no Programa 9.11 verifica se a saída observada no multiplexador é igual à saída esperada. Se os valores forem iguais, o comportamento é o esperado, e portanto correto *com relação aos vetores de teste que o projetista escreveu*. Se o projetista escolher valores de teste inadequados, ou errados, pode ser difícil diagnosticar problemas no modelo.

Programa 9.11: Mensagem de verificação de comportamento.

```
assert (saida = esperada)
  report "mux2:␣saida␣errada␣sel=" & B2STR(sel) &
    "␣saiu=" & B2STR(saida) & "␣esperada=" & B2STR(esperada)
  severity error;
```

Se os valores nos sinais *saida* e *esperada* diferem, a mensagem no Programa 9.11 é emitida no terminal, indicando o erro:

```
mux2: saída errada   sel=1   saiu=0   esperada=1.
```

A função `B2STR` converte um bit em uma *string* para que o valor do bit seja emitido na tela. O operador ‘&’ concatena duas *strings*. É obrigação do projetista determinar a saída esperada para cada combinação de entradas no vetor de teste.

Ao final do laço, a simulação termina no comando **wait**; este faz com que a execução do processo `U_testValues` se encerre silenciosamente.

A Seção 9.2.9 mostra um exemplo completo de modelagem, desde a especificação de um somador de 16 bits, o código VHDL do modelo e do *testbench*, e o resultado da simulação.

9.2.8 Abstração em VHDL: uma entidade, muitas arquiteturas

VHDL provê várias construções que facilitam o emprego de *abstração*. Vimos, na Seção 9.2.2, como definições podem ser agrupadas em **packages** e estas importadas pelas unidades de projeto que delas fazem uso. Outra construção útil é a possibilidade de definir mais de uma arquitetura para uma entidade.

Lembre da espiral de projeto: a cada avanço na espiral, a realização da especificação se torna mais concreta e mais próxima da implementação final.

Uma *entidade* define a interface de um componente com o mundo exterior àquele componente. Uma *arquitetura* é um refinamento, e uma das possíveis implementações, para a especificação daquele componente.

Esta forma de trabalhar é capturada de uma maneira simples por VHDL: uma entidade pode ser implementada por mais de uma arquitetura. Em geral, a primeira arquitetura desenvolvida é a mais abstrata e pode ser uma especificação executável; as intermediárias são mais detalhadas e contêm mais detalhes da implementação; a última é um modelo concreto que pode ser sintetizado como um circuito.

Do ponto de vista do código VHDL, caso haja mais do que uma arquitetura, o compilador emprega aquela última ‘vista’, que é a que está “mais longe” da definição da entidade no código fonte, e ignora a(s) arquitetura(s) definida(s) ‘antes’. O Programa 9.12 mostra um exemplo.

Programa 9.12: Duas arquiteturas para uma entidade.

```
entity meuProjeto is
  port( ... )
end meuProjeto;

architecture especificacao of meuProjeto is
begin
  ...
end architecture especificacao;

architecture concreta of meuProjeto is
begin
  ...
end architecture concreta;
```

A entidade meuProjeto define a interface do componente. A arquitetura especificacao define a função e o comportamento desejado de meuProjeto. A arquitetura concreta é uma versão detalhada e que pode(ria) ser sintetizada diretamente pelo compilador VHDL. Na medida em que o projeto é refinado, cada arquitetura mais concreta é inserida no código abaixo da menos concreta, para que a última seja usada pelo compilador. Dessa forma, é fácil comparar diferentes versões de um mesmo modelo.

9.2.9 Um exemplo completo: somador de 16 bits

Vejamos um exemplo completo de modelagem estrutural em VHDL. Nosso objetivo é desenvolver um somador de 16 bits a partir de 16 cópias de um somador completo. Nosso projeto é uma extensão direta do somador de 4 bits apresentado na Seção 6.2.

Especificação Tipicamente, na fase inicial de um projeto que emprega VHDL como ferramenta de modelagem, um componente é especificado informalmente num elevado nível de abstração, como: *somador de 16 bits*. Esta especificação abstrata é então codificada num *modelo funcional*, similar à Equação 9.1, que define as interfaces e o efeito das operações de uma forma abstrata e que omite os detalhes da implementação.

$$\begin{aligned}
& vem : \mathbb{B} \\
& A, B : \mathbb{B}^{16} \\
& S : \mathbb{B}^{16} \\
& vai : \mathbb{B} \\
& soma16 : (\mathbb{B} \times \mathbb{B}^{16} \times \mathbb{B}^{16}) \mapsto (\mathbb{B}^{16} \times \mathbb{B}) \\
& soma16(vem, A, B, S, vai) \equiv (S, vai) = num(vem + A + B)
\end{aligned} \tag{9.1}$$

A especificação do somador define que os dois operandos A e B e o resultado S são representados em 16 bits e que a entrada de vem-um e a saída de vai-um são bits. O par (S, vai) é o resultado em 17 bits da soma de A , B e do vem-um.

O modelo funcional é então traduzido para um modelo mais concreto em que alguns dos detalhes do projeto são explicitados. Modelos neste nível são chamados de *modelo de fluxo de dados* (*dataflow*) ou *modelos RTL*. RTL é a abreviatura para *Register Transfer Language*, empregada para descrever o fluxo dos dados através de um circuito, no qual valores são transformados por circuitos combinacionais que são delimitados por pares de registradores.

Um *modelo estrutural* descreve a estrutura de um circuito, do ponto de vista das interconexões entre seus componentes. Dependendo das ferramentas disponíveis, um circuito pode ser sintetizado a partir da sua descrição no nível RTL ou do nível estrutural, porque estas descrições são suficientemente concretas e detalhadas para permitir a tradução automática e eficiente da descrição em VHDL para uma lista de componentes e ligações entre aqueles, que é então usada para produzir o circuito integrado. Esta lista de ligações e componentes é chamada de *net list*.

Modelo estrutural do somador completo Nosso projeto faz uso do pacote de definições do Programa 9.3. Definimos o comportamento das portas lógicas necessárias para implementar o somador completo da Seção 6.2 e salvamos o código VHDL num arquivo que chamamos de `aux.vhd`.

VHDL define as operações lógicas **and**, **or**, **xor** para operandos de tipo bit. O Programa 9.13 define o comportamento das portas lógicas *and2*, *or3* e *xor3*. À primeira vista pode parecer que o código é extremamente verboso – são nove linhas de código para cada porta. Essas definições são coletadas em uma biblioteca para uso posterior, ou como é o caso aqui, mantidas num arquivo separado daquele em que está a parte principal do projeto. Uma vez que os modelos dos componentes estejam corretos, eles podem ser usados de forma mais econômica nos modelos mais complexos, mantidos em bibliotecas ou em arquivos fonte separados.

As três arquiteturas têm o mesmo nome (`comport`) porque são modelos comportamentais das respectivas portas lógicas. Os modelos são diferenciados pelos nomes de suas entidades, a saber *and2*, *xor3* e *or3*.

Programa 9.13: Modelo funcional das portas lógicas.

```

entity and2 is
  port(A,B : in bit;
        S   : out bit);
end and2;

  architecture comport of and2 is
begin
  S <= A and B;
end architecture comport;

entity xor3 is
  port(A,B,C : in bit;
        S     : out bit);
end xor3;

  architecture comport of xor3 is
begin
  S <= A xor B xor C;
end architecture comport;

entity or3 is
  port(A,B,C : in bit;
        S     : out bit);
end or3;

  architecture comport of or3 is
begin
  S <= A or B or C;
end architecture comport;

```

Definidos os componentes básicos, podemos escrever o *modelo funcional* do somador completo, como definido pela Equação 6.5. O bit com a soma é o ou-exclusivo das três entradas

$$s = a \oplus b \oplus vem$$

e o bit de vai-um é a soma de produtos das três entradas, tomadas duas a duas

$$vai = (a \wedge b) \vee (a \wedge vem) \vee (b \wedge vem).$$

Estas equações são o corpo do modelo funcional do somador completo, como mostrado nas linhas entre o **begin** e o **end architecture** do Programa 9.14.

O *modelo estrutural* do somador completo também é mostrado no Programa 9.14. Como vimos na Seção 9.2.8, caso haja mais de uma, o compilador VHDL examina as várias arquiteturas e produz uma implementação para a última, na ordem do código fonte. Isso permite que o programador use a arquitetura abstrata (anterior) como a especificação para a arquitetura mais concreta (posterior). Note que as duas arquiteturas devem ter nomes distintos, que neste caso são funcional e estrutural.

O modelo estrutural é uma versão textual do diagrama do circuito com portas lógicas. São necessários três sinais internos ao modelo para interligar as saídas dos *and2* às entradas do

or3. Estes sinais devem ser declarados entre o **architecture** e o **begin**. Além dos sinais internos, os componentes empregados no modelo também podem ser declarados explicitamente² **architecture** e o **begin**.

Programa 9.14: Modelo estrutural do somador completo.

```

use work.p_wires.all;
entity addBit is
    port(bitA, bitB, vem : in bit;    — entradas A,B,vem—um
        soma, vai      : out bit); — saida C,vai—um
end addBit;

— modelo funcional —————
architecture funcional of addBit is
begin
    soma = bitA xor bitB xor vem;
    vai  = (bitA and bitB) or (bitA and vem) or (vem and bitB);
end architecture funcional;

— modelo estrutural —————
architecture estrutural of addBit is
    component and2 is
        port (A,B: in bit; S: out bit);
    end component and2;

    component or3 is
        port (A,B,C: in bit; S: out bit);
    end component or3;

    component xor3 is
        port (A,B,C: in bit; S: out bit);
    end component xor3;

    signal a1,a2,a3, : bit;
begin
    U_xor: xor3 port map (bitA, bitB, vem, soma);
    U_and1: and2 port map (bitA, bitB, a1);
    U_and2: and2 port map (bitA, vem, a2);
    U_and3: and2 port map (bitB, vem, a3);
    U_or: or3 port map (a1, a2, a3, vai);
end architecture estrutural;

```

Espaço em branco proposital.

²O autor considera uma boa prática declarar os explicitamente componentes, embora isso seja uma questão de estilo, ou de gosto.

Modelo estrutural do somador de 16 bits A Figura 9.6 mostra o circuito de um somador de 16 bits composto por 16 réplicas do somador completo, e é a extensão óbvia para 16 bits do somador mostrado na Figura 6.4.

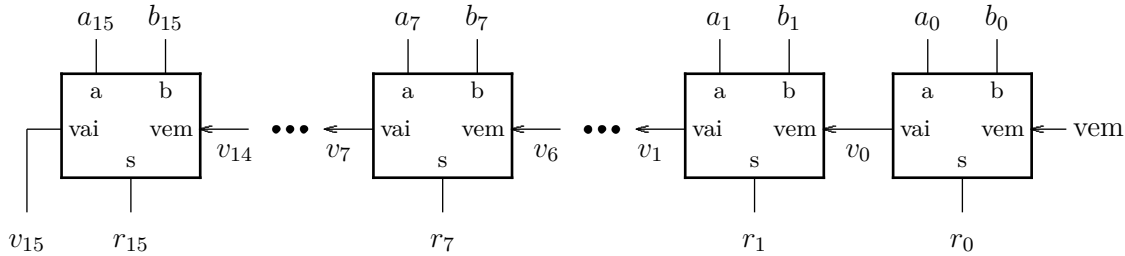


Figura 9.6: Somador composto de 16 cópias de *addBit*.

O Programa 9.15 contém a entidade *adder16*, com a interface especificada pela Equação 9.1: uma entrada de *vem-um*, dois operandos de 16 bits, a soma em 16 bits e o bit de *vai-um*.

Programa 9.15: Modelo estrutural do somador de 16 bits.

```

use work.p_wires.all;
entity adder16 is
    port(vem : in bit;      — vem-um
          A, B : in reg16; — entradas de 16 bits
          S : out reg16; — saída em 16 bits
          vai : out bit); — vai-um
end adder16;

architecture estrutural of adder16 is
    component addBit port(bitA, bitB, vem : in bit;
                          soma, vai : out bit);
    end component addBit;

    signal v : reg16; — vetor de bits para a cadeia de vai-um

begin
    U_b0: addBit port map ( A(0), B(0), vem, S(0), v(0) );
    U_b1: addBit port map ( A(1), B(1), v(0), S(1), v(1) );
    U_b2: addBit port map ( A(2), B(2), v(1), S(2), v(2) );
    U_b3: addBit port map ( A(3), B(3), v(2), S(3), v(3) );
    U_b4: addBit port map ( A(4), B(4), v(3), S(4), v(4) );
    U_b5: addBit port map ( A(5), B(5), v(4), S(5), v(5) );
    U_b6: addBit port map ( A(6), B(6), v(5), S(6), v(6) );
    U_b7: addBit port map ( A(7), B(7), v(6), S(7), v(7) );
    U_b8: addBit port map ( A(8), B(8), v(7), S(8), v(8) );
    U_b9: addBit port map ( A(9), B(9), v(8), S(9), v(9) );
    U_ba: addBit port map ( A(10), B(10), v(9), S(10), v(10) );
    U_bb: addBit port map ( A(11), B(11), v(10), S(11), v(11) );
    U_bc: addBit port map ( A(12), B(12), v(11), S(12), v(12) );
    U_bd: addBit port map ( A(13), B(13), v(12), S(13), v(13) );
    U_be: addBit port map ( A(14), B(14), v(13), S(14), v(14) );
    U_bf: addBit port map ( A(15), B(15), v(14), S(15), vai );
end architecture estrutural;

```

O Programa 9.15 mostra o código da arquitetura *estrutural* da entidade *adder16*. A arquitetura faz uso de um único componente, que é o somador completo – as portas lógicas ficam escondidas no modelo *addBit* – e um vetor de bits para transportar os bits de vai-um entre cada par de somadores – o bit $v(i)$ é gerado pelo somador do i -ésimo bit do resultado.

Entre o **architecture** e o **begin**, o modelo estrutural declara um único componente (*addBit*) e o vetor da cadeia de vai-um *v* de tipo *reg16*. No corpo da arquitetura são instanciadas 16 cópias desse componente, cada uma com seu *label* *U_bi*. Note que o mapeamento de portas é posicional e que a posição de cada sinal no modelo que usa o componente deve ser exatamente a mesma que na definição do componente. Ao sinal de interface *vai* é atribuído o valor do bit de vai-um produzido pelo somador dos bits mais significativos.

Este modelo, com o acréscimo de informação sobre temporização, é utilizado nos laboratórios, e uma versão mais sofisticada é empregada no processador do projeto/trabalho semestral.

Testbench O Programa 9.16 mostra o código de teste para o somador de 16 bits, e este é similar ao *testbench* para o modelo do *mux-2*, mostrado no Programa 9.10.

Por razões de espaço, o vetor de testes *test_array* contém somente três elementos. Teoricamente, seriam necessários $2^1 \times 2^{16} \times 2^{16}$ casos para testar *todas* as combinações para as entradas. A Seção 6.9 indica algumas das possibilidades para reduzir o número impraticável de vetores de testes sem sacrificar a confiança nos resultados.

O vetor de testes não inclui um campo para o vem-um, somente para as entradas *a* e *b* do somador, e dois campos para os valores esperados para a soma e o vai-um. Por isso o componente *adderEstrut16* é instanciado com a constante '0' no campo correspondente ao vem-um. O acréscimo do vem-um no vetor de testes e no corpo do *testbench* fica como um exercício para a leitora e o leitor dedicados.

Programa 9.16: Testbench para o modelo estrutural do somador de 16 bits.

```

use work.p_wires.all;
entity tb_somador is
    — entidade vazia
end tb_somador;

architecture TB of tb_somador is

    component adderEstrut16 is port(vem : in bit;
                                     A, B : in bit_vector;
                                     C   : out bit_vector;
                                     vai : out bit);

    end component adderEstrut16;

    type test_record is record
        a : reg16; b : reg16; — entradas
        c : reg16;          — saída esperada
        v : bit;            — vai-um esperado
    end record;

    type test_array is array(positive range <>) of test_record;
    constant test_vectors : test_array := (
        — a,          b,          c,          vai-um
        (x"1111",x"1111",x"2222",'0'), — testes insuficientes
        (x"2222",x"2222",x"4444",'0'),
        (x"4444",x"4444",x"8888",'0'));

    signal inpA,inpB,res,esp_res : reg16; — entradas de teste
    signal vai, esp_vai : bit;

begin — TB
    U_addCad: adderEstrut16 port map('0', inpA, inpB, res, vai);

    U_testValues: process — testa o circuito
        variable v : test_record;
    begin
        for i in test_vectors'range loop
            v := test_vectors(i); — valores desse teste
            inpA <= v.a;
            inpB <= v.b;
            esp_res <= v.c;
            esp_vai <= v.v;
            assert ((res = esp_res) and (vai = esp_vai))
                report LF & "somador_estrutural: saída errada" &
                LF & "uuuuA=" & BV2STR(inpA) & LF & "uuuuB=" & BV2STR(inpB) &
                LF & "usoma=" & BV2STR(res) & LF & "uuvai=" & B2STR(vai)
                severity error;
            wait for 2 ns; — duração do passo de simulação
        end loop;
        wait; — encerra processo
    end process;
end TB;

```

Em caso de erro, se o resultado ou se vai-um produzido(s) pelo circuito é diferente do(s) valore(s) esperado(s), então o simulador emite a mensagem de erro na tela:

```
somador estrutural: saida errada
A=2222  B=2222  soma=4444  vai=1.
```

A Figura 9.7 mostra uma parte da tela do gtkwave com o resultado da simulação do *testbench* no Programa 9.16. Os sinais *inpa* (*input A*) e *inpb* (*input B*) são as entradas, o resultado esperado é *esp_res*, e a saída do somador são os sinais *rescad* (*resultado cadeia*) e *vaicad* (*vai-um cadeia*). O diagrama de tempos mostra os resultados para a soma dos pares de operandos definidos pelos elementos do registro *test_array*, com a soma de um par de x"0000"s no início e no final da simulação.

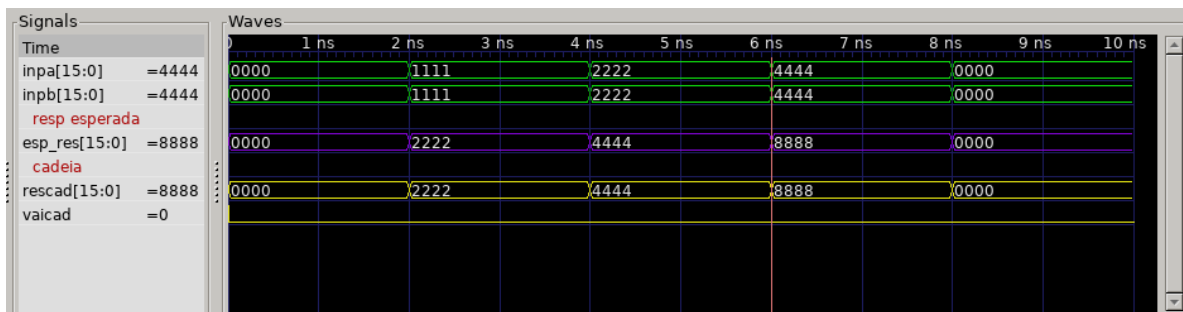


Figura 9.7: Resultado da simulação com o *testbench* do somador de 16 bits.

Os sinais na entrada do somador se alteram a cada 2ns, o que facilita a visualização da sequência de vetores de testes e resultados. Para todas os pares de entradas, a soma produzida pelo modelo é igual ao esperado, logo, para o conjunto *limitadíssimo* de valores empregados neste teste, o modelo do somador está correto. A Seção 6.9 discute a geração de vetores de teste mais completos e eficazes.

9.2.10 Síntese para *Field Programmable Gate Arrays*

O resultado da compilação para *síntese* pode ser a programação de um *Field Programmable Gate Array*, ou ainda pode ser um conjunto de máscaras para fabricar um *Application Specific Integrated Circuit* (ASIC).

Um *Field Programmable Gate Array* (FPGA) é um circuito integrado programável composto por uma matriz com milhares (10^4 a 10^5) de células programáveis. Por célula programável entenda-se algo como o mostrado na Figura 9.8, que é um modelo altamente simplificado.

Cada célula contém um *flip-flop*, duas *look-up tables* (LUT) e multiplexadores para interligar sinais. Uma LUT nada mais é do que um multiplexador com 8 ou 16 entradas, e estas entradas podem ser ligadas para implementar qualquer função lógica de 3 ou 4 variáveis. Tanto as LUTs quando os multiplexadores na entrada do *flip-flop* são acionados pelos bits da *memória de programação*.

A memória de programação é uma RAM que não pode ser acessada diretamente e que é preenchida quando o FPGA é programado com o ‘programa’ gerado pelo compilador/sintetizador. Quando o FPGA é inicializado, esta memória é carregada com o padrão de bits necessário para colocar os multiplexadores nas posições apropriadas e implementar as funções das LUTs.

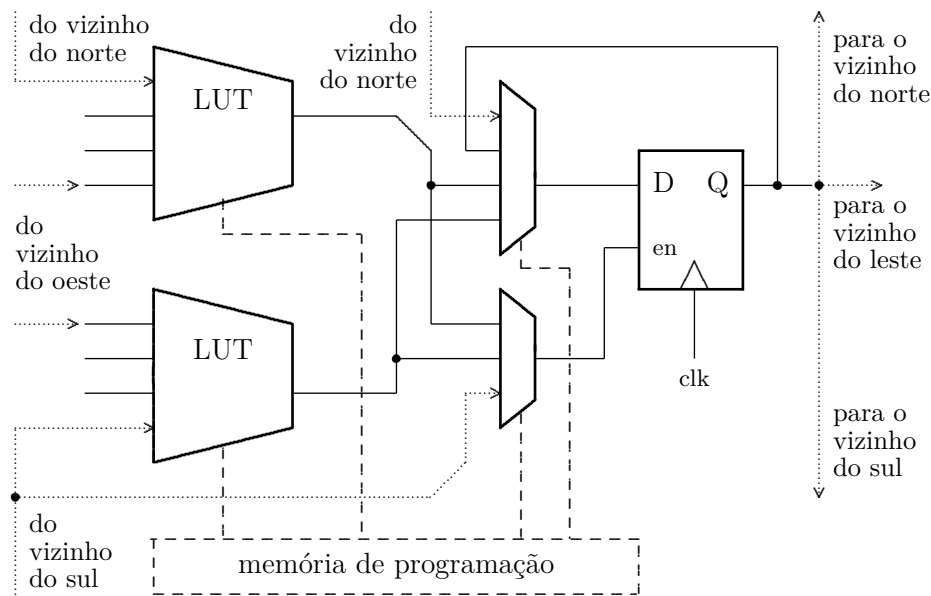


Figura 9.8: Modelo simplificado de uma célula de um FPGA.

Quando se compila VHDL para *synthesize* um FPGA, o que o compilador produz é o conteúdo para a memória de programação, de forma a configurar o FPGA para que se comporte como definido no código fonte.

O circuito da Figura 9.8 mostra, nas linhas pontilhadas, as ligações para os *flip-flops* e LUTs das células vizinhas, o que permite implementar contadores, registradores, deslocadores ou máquinas de estado.

As células são organizadas em matrizes, separadas por caminhos largos que interligam as matrizes entre si e com células de interface para a periferia. Estas células de interface são circuitos especializados para interligar o FPGA a outros dispositivos ou circuitos integrados.

9.2.11 Comandos concorrentes de seleção: *when-else* e *with-select*

Vejamos o que VHDL nos oferece de mais abstrato do que representações textuais para projetos com portas lógicas. Sendo uma linguagem assaz versátil, VHDL nos permite trabalhar com baixo nível de abstração – interligação de portas lógicas – e também com código um tanto mais abstrato. Vejamos as duas maneiras de modelar seleção.

A *área concorrente* de uma arquitetura é a região de código entre o **begin** e o **end architecture**. O trecho de código no Programa 9.17 mostra a arquitetura chamada *exemplo* da entidade *area_concorrente*. Na área concorrente estão dois comandos, ou dois processos, que são executados concorrentemente: a disjunção e a conjunção de dois sinais. Num circuito, esta arquitetura seria implementada com duas portas lógicas que constantemente amostram suas entradas e produzem suas saídas em função daquelas. O modelo VHDL deve ter o mesmo comportamento e portanto os dois processos executam concorrentemente: qualquer alteração em *a* e/ou em *b* provocará a avaliação do lado direito das atribuições, e se for o caso, eventos nos sinais *r* e *s*.

Programa 9.17: Área concorrente num arquitetura.

```

architecture exemplo of area_concorrente is
begin
    — inicio da área concorrente

    U_or:  r <= a or b;
    U_and: s <= a and b;

    — final da área concorrente
end architecture exemplo;

```

A linguagem nos oferece dois *comandos concorrentes* de seleção que podem ser usados na área concorrente: o comando **when-else** e o comando **with-select**.

when-else A forma mais simples do comando de atribuição condicional **when-else** é

```
a when cond else b;
```

Se a condição *cond* é verdadeira, então o comando tem o valor de *a*, do contrário o comando tem o valor de *b*. Quando usado numa atribuição, este comando se comporta como um comando de seleção de *C*: $s = (\text{cond} ? a : b)$;

```
s <= a when cond else b;
```

O comando pode ser encadeado com várias cláusulas **else**:

```

s <= a when cond_1 else
      b when cond_2 else
      c when cond_3 else
      d;

```

As condições deste comando são avaliadas na ordem do texto: primeiro *cond_1*, se esta é falsa então *cond_2* é avaliada, e se esta é falsa então *cond_3* é avaliada. Se a última cláusula é falsa, então o valor atribuído é o que segue o último **else**.

O comando **when-else** pode ser usado para implementar um *mux-4*:

```

z <= a when (s = "00") else
      b when (s = "01") else
      c when (s = "10") else
      d;

```

Um modelo completo é mostrado no Programa 9.18, para entradas e saídas que são vetores de 8 bits – o multiplexador tem 4 entradas de 8 bits cada. Este modelo é um tanto mais abstrato do que um modelo estrutural com a mesma função.

Programa 9.18: Modelo do mux4x8 com when-else.

```

entity mux4x8 is
  port (a,b,c,d: in bit_vector(7 downto 0);
        s:       in bit_vector(1 downto 0);
        z:       out bit_vector(7 downto 0));
end mux4x8;

architecture when_else of mux4x8 is
begin
  z <= a when (s = "00") else — atribuição condicional
    b when (s = "01") else
    c when (s = "10") else
    d; — todos os demais casos
end architecture when_else;

```

with-select O comando concorrente de atribuição selecionada **with-select** se parece com um switch da linguagem C. Para um sinal de seleção *s* que pode assumir um dentre dois valores (*v1,v2*), o valor do comando **with-select** corresponde ao valor selecionado. Vejamos um exemplo.

```

with s select — s pode ser v1 OU v2
  r <= a when v1,
    b when v2;

```

Uma cláusula **others** pode ser usada para capturar todos os demais casos:

```

with s select — s pode ser v1 OU v2
  r <= a when v1,
    b when others; — todos os demais casos

```

Vejamos como fica o nosso mux-4, com um comando de atribuição selecionada.

```

with s select
  z <= a when "00",
    b when "01",
    c when "10",
    d when others; — todos os demais casos

```

Ao contrário do **when-else**, todas as cláusulas do **with-select** tem a mesma precedência e geralmente este comando resulta em circuitos menores e mais rápidos. O Programa 9.19 mostra a arquitetura do mux-4x8 com **with-select**.

Programa 9.19: Modelo do mux4x8 com with-select.

```

architecture with_select of mux4x8 is
begin
  with s select — atribuição selecionada
    r <= a when "00",
      b when "01",
      c when "10",
      d when others; — todos os demais casos
end architecture with_select;

```

9.3 Modelagem da temporização de circuitos combinacionais

VHDL permite que informação de tempo seja agregada às descrições dos circuitos, possibilitando assim a simulação temporal dos modelos. Tipicamente, estas simulações são usadas em fases iniciais do projeto porque a “anotação ‘manual’ da temporização dos circuitos permite prever o comportamento dinâmico e a velocidade máxima de operação. As ferramentas de síntese produzem modelos temporais detalhados e com melhor acurácia do que modelos produzidos manualmente e servem para confirmar as previsões das fases iniciais do projeto.

9.3.1 Modelo funcional *versus* modelo temporal

Nas Seções 9.2.3 e 9.2.9 empregamos *modelos funcionais* para portas lógicas, e com aqueles construímos modelos para circuitos combinacionais compostos de várias portas lógicas. Aqueles modelos são chamados de *funcionais* porque representam somente o aspecto *função lógica* dos circuitos, e ignoram informações de tempo.

Nesta seção mostramos como enriquecer os modelos funcionais com informação de tempo, criando modelos mais sofisticados para o comportamento dos circuitos.

Modelos temporais incorporam a informação de tempo de propagação dos componentes e permitem simulações mais detalhadas e realistas dos circuitos. Evidentemente, simulações mais detalhadas são mais custosas em termos de tempo de programação, tempo de simulação e de memória.

Os modelos que usamos nesta seção permitem a visualização do comportamento dos circuitos combinacionais ao longo do tempo, e a medição do seu tempo de propagação. Em breve estudaremos a propagação de sinais em somadores e nestes circuitos os problemas são algo mais severos do que veremos aqui. Cada coisa a seu tempo.

Antes de prosseguirmos, reveja a definição de *circuito combinacional* da Seção 4.1.

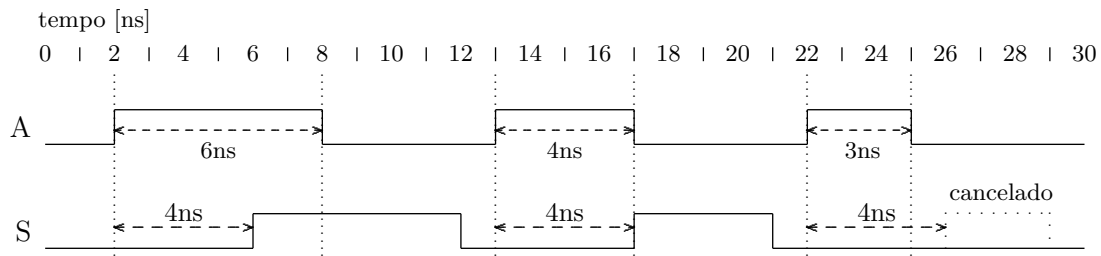
9.3.2 Descrição de temporização em VHDL

A atribuição a um sinal pode ser postergada para modelar o comportamento de circuitos reais. Por exemplo, o modelo para o comportamento de um inversor pode incluir o tempo de propagação com o atributo **after**:

```
S <= inertial (not A) after 4 ns;
```

O complemento de A é atribuído a S após o tempo de propagação de 4 nanosegundos. O atributo **after** faz com que o evento no sinal S seja postergado para ocorrer 4ns no futuro, contados desde o tempo corrente. O qualificador **inertial** é optativo e pode ser omitido.

VHDL possibilita a modelagem de dois tipos de atrasos, atraso inercial e atraso de transporte. O *atraso inercial* modela o tempo de propagação de portas lógicas. Com atraso inercial, pulsos com duração menor que o intervalo especificado pelo **after** não são transferidos, por causa da *inércia* natural dos sinais – um pulso na entrada de um circuito é propagado para sua saída caso dure o suficiente para alterar os valores em todos os nós internos ao circuito. A Figura 9.9 mostra um diagrama de tempo de uma simulação na qual ocorrem três pulsos no sinal A e o efeito do comando acima no sinal S. Os dois pulsos iniciais têm duração maior ou igual a 4ns e são transferidos para S; o terceiro pulso tem uma duração de 3ns, menor do que a indicada pelo **after**, não é transferido para S e portanto ignorado.

Figura 9.9: Atraso inercial com *after*.

O *atraso de transporte*, indicado pelo atributo **transport**, modela a propagação através de um fio e portanto pulsos de qualquer duração são propagados. O comando abaixo corresponde a um fio longo, com um atraso na propagação de sinais de 2ns.

```
T <= transport A after 2 ns;
```

A Figura 9.10 mostra o diagrama de tempo da simulação da propagação através do fio (longo) que conecta os nós A e T.

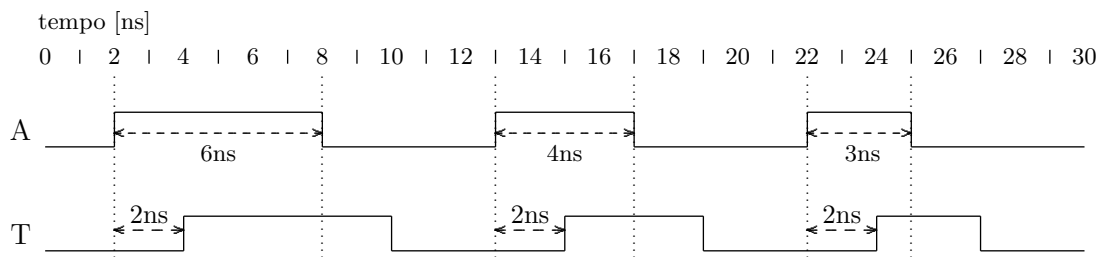


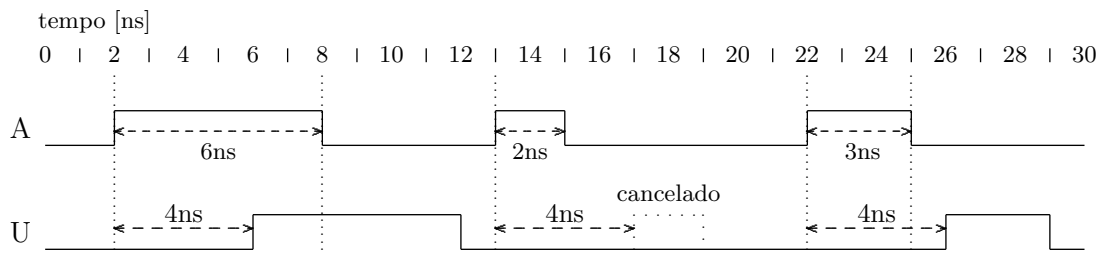
Figura 9.10: Atraso de transporte.

A modelagem do tempo de propagação pode ser refinada para evitar que pulsos com largura menor do que o especificado por **after** sejam rejeitados. A cláusula **reject** faz com que pulsos de largura menor ou igual a seu argumento sejam eliminados; pulsos mais largos que o argumento de **reject** são transferidos. Se o argumento de **after** é t_a , então o argumento de **reject** deve estar no intervalo $[0, t_a]$.

No comando

```
U <= reject 2.5 ns inertial A after 4 ns;
```

a atribuição a *U*, por conta da cláusula **reject**, rejeita pulsos em A com até 2,5ns de duração, mas propaga pulsos com duração maior que o limite de rejeição. O qualificador **inertial** não pode ser omitido quando se emprega a cláusula **reject**.

Figura 9.11: Atraso inercial com *reject*.

Não há uma correspondência direta entre o modelo de temporização definido na Seção 5.4 e os três modelos para simular temporização disponibilizados por VHDL. Quanto a contaminação, o modelo para a simulação de tempo com VHDL considera que, (i) não existe contaminação e todos os pulsos são transferidos – o intervalo de rejeição é especificado como zero; ou (ii) a contaminação dura tanto quanto a propagação, e para tanto basta especificar o tempo de propagação com **after**.

A capacitância interna aos nós do circuito é considerada apenas em termos da inércia na transmissão de sinais entre as entradas e as saídas. Se a capacitância no caminho de um sinal é baixa, então o circuito responde a pulsos estreitos – para isso empregamos uma especificação com ambos **reject** e **after**

```
U <= reject t_rej ns inertial A after t_prop;
```

Pulsos com duração menor ou igual a t_rej não são transferidos para a saída e portanto a saída do circuito permanece imutável – como se estivesse contaminada – durante o intervalo de rejeição. Pulsos com duração maior do que t_rej são transferidos com um atraso de t_prop .

A Figura 9.12 mostra um diagrama de tempo para o modelo de temporização para atrasos inerciais com rejeição de pulsos estreitos. As entradas do modelo inicialmente estão na configuração α e sua saída é $f(\alpha)$. Ocorre uma mudança de α para β , possivelmente por causa de alguma corrida de sinais nos circuitos que alimentam o modelo, e esta alteração nas entradas não é refletida para a saída porque sua duração é menor do que o tempo de contaminação T_C do modelo. A nova configuração de entradas γ persiste por mais do que T_C , e decorrido o tempo de propagação T_P , a saída torna-se $f(\gamma)$.

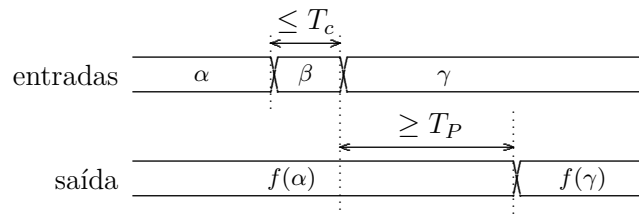


Figura 9.12: Modelagem da temporização em VHDL.

O modelo de temporização de VHDL é similar ao modelo definido na Seção 7.6. O modelo para o tempo de propagação tem a mesma definição: uma vez que as entradas estejam válidas e estáveis, as saídas se tornam válidas e estáveis após T_P . A rejeição de pulsos estreitos tem uma interpretação similar àquela do tempo de contaminação, no sentido de que a saída não é alterada a menos que a alteração nas entradas tenha duração suficiente para provocar uma alteração na saída que perdure até que o nível na saída estabilize.

9.3.3 Definição dos parâmetros de temporização

Há quatro formas de codificar os parâmetros de temporização das portas lógicas, que também são usadas para definir quaisquer constantes empregadas nos modelos. Estas quatro formas satisfazem regras óbvias de escopo, de forma que as constantes fiquem disponíveis: (i) localmente na arquitetura; (ii) em todas as arquiteturas de uma entidade; (iii) nas entidades que importam o conteúdo de um pacote; e (iv) através de parâmetros genéricos.

Na primeira forma, as constantes são definidas na arquitetura, como mostra o Programa 9.20. Esta forma é menos flexível do que a anterior porque as constantes ficam visíveis somente no escopo da arquitetura em que foram definidas.

Programa 9.20: Parâmetros de temporização definidos na arquitetura.

```
entity or2time is
  port(A,B : in bit;
        S   : out bit);
end or2time;

architecture temporiz of or2time is
  constant p_or2 : time := 20.0 ps;
  constant r_or2 : time := 5.0 ps;
begin
  S <= reject r_or2 inertial (A or B) after p_or2;
end architecture temporiz;
```

A segunda maneira de definir as constantes de temporização é defini-las na entidade, como mostra o Programa 9.21. As constantes ficam assim disponíveis para quaisquer das arquiteturas associadas àquela entidade.

Programa 9.21: Parâmetros de temporização definidos na entidade.

```
entity and2time is
  port(A,B : in bit;
        S   : out bit);
  constant p_and2 : time := 20.0 ps;
  constant r_and2 : time := 5.0 ps;
end and2time;

architecture temporiz of and2time is
begin
  S <= reject r_and2 inertial (A and B) after p_and2;
end architecture temporiz;
```

A terceira forma é colocar as definições num pacote. Por exemplo, os tempos de propagação das portas lógicas (p_inv, p_and2, p_or2), assim como os intervalos de rejeição (r_inv, r_and2 e r_or2), são constantes declaradas na segunda versão do pacote p_wires, mostrada no Programa 9.22. Assim, a informação quanto aos parâmetros de temporização das portas lógicas fica concentrada num único lugar/arquivo.

Programa 9.22: Abreviaturas, definições e temporização.

package p_WIRES **is** — *versão com temporização*

```

constant p_inv   : time := 15.0 ps;
constant r_inv   : time :=  4.0 ps;

constant p_and2  : time := 20.0 ps;
constant r_and2  : time :=  5.0 ps;

constant p_or2   : time := 22.0 ps;
constant r_or2   : time :=  6.0 ps;

subtype reg2    is bit_vector(1 downto 0);
subtype reg4    is bit_vector(3 downto 0);
subtype reg8    is bit_vector(7 downto 0);
subtype reg16   is bit_vector(15 downto 0);

```

```

...
package body p_WIRES is
...
end package p_WIRES;

```

A quarta maneira emprega *interfaces genéricas*, que permitem redefinir as constantes a cada vez que um componente é instanciado. A entidade define um valor *default* e os parâmetros definidos como *genéricos* podem ser alterados em cada instanciamento. Vejamos como o modelo da porta *and2* do Programa 9.21 pode ser adaptado para suportar constantes genéricas para temporização. O Programa 9.23 mostra a unidade de projeto para a porta *and2*. As constantes p_and2 e r_and2 são importadas do pacote p_wires. Os parâmetros genéricos são declarados na cláusula **generic**, antes da declaração dos sinais de interface do componente.

Programa 9.23: Parâmetros de temporização genéricos da porta *pand2*.

```

use work.p_wires.all;
entity and2gen is
  generic (prop : time := p_and2, reje : time := r_and2);
  port(A,B : in bit;
       S   : out bit);
end and2gen;

architecture generica of and2gen is
begin
  S <= reject reje inertial (A and B) after prop;
end architecture generica;

```

Veremos em seguida, na Seção 9.3.4, o modo de emprego dos parâmetros genéricos.

9.3.4 Modelo temporizado do multiplexador

A entidade e a arquitetura do multiplexador que vimos na Seção 9.2.3 são estendidas para fazer uso dos parâmetros genéricos de temporização. Os modelos do inversor e da porta *or2* são similares ao mostrado no Programa 9.23. A entidade do *mux-2* não sofre nenhuma alteração; contudo, os componentes na arquitetura são declarados e instanciados com a informação de tempo.

Note que a entidade *mux2* não importa a biblioteca do IEEE porque nenhuma função daquela biblioteca é empregada nesta unidade de projeto. As funções da biblioteca do IEEE ficam escondidas nos modelos dos componentes. O modelo do *mux-2* faz uso e importa a biblioteca de trabalho.

Entre o **architecture** e o **begin** estão declarados os componentes, com as interfaces genéricas. No corpo da arquitetura, entre o **begin** e o **end architecture**, os componentes são instanciados, cada um com os parâmetros de tempo adequados. O inversor *Ui* tem o seu tempo de propagação acrescido de 5ps enquanto que o parâmetro da rejeição é inalterado – o **open** indica que o valor *default* declarado³ na unidade de projeto do inversor é que deve ser usado.

Espaço em branco proposital.

³Aparentemente, o compilador *ghdl* não aceita o **open** no lugar de um valor explícito, embora [Ash08] assim o preconize. Este autor prefere o padrão à implementação.

Programa 9.24: Modelo do *mux-2* com informação de temporização.

```

use work.p_wires.all;
entity mux2 is
    port(a,b : in bit;
          s : in bit;
          z : out bit);
end mux2;

architecture temporiz of mux2 is
    — declaração dos componentes
    component inv is
        generic (prop : time; reje : time); — propagação, rejeição
        port(A : in bit; S : out bit);
    end component inv;

    component and2 is
        generic (prop : time; reje : time); — propagação, rejeição
        port(A,B : in bit; S : out bit);
    end component and2;

    component or2 is
        generic (prop : time; reje : time); — propagação, rejeição
        port(A,B : in bit; S : out bit);
    end component or2;

    — sinais internos ao mux2
    signal r, p, q : bit;

begin
    — instanciação dos componentes
    Ui: inv generic map (10 ps, 2 ps) port map (s, r);
    Ua0: and2 generic map (16 ps, 4 ps) port map (a, r, p);
    Ua1: and2 generic map (16 ps, 1 ps) port map (b, s, q);
    Uor: or2 generic map ( 9 ps, 3 ps) port map (p, q, z);
end architecture temporiz;

```

As portas *and2* Ua0 e Ua1 usam o tempo de propagação definido em *packageWires.vhd*, Ua1 altera o tempo de rejeição para 1ps. O *or2* (Uor) altera seu tempo de propagação *default* em 9ps e redefine a rejeição para 3ps.

Esta forma de modelagem *esconde toda a informação que pode ser escondida (information hiding)*: **toda** a informação sobre tempo está escondida na arquitetura do *mux-2*, e sua entidade não contém nenhuma informação sobre a temporização. Quando usarmos o *mux-2* em outros modelos, tal como para implementar um *mux-4*, nenhuma informação sobre tempo de propagação do *mux-2* é transferida explicitamente para o *mux-4* porque tal informação está escondida na arquitetura do *mux-2*, e fica portanto invisível ao usuário do modelo do *mux-4*. O usuário dos modelos baseados no *mux-2* não mais se ocupa com definir a temporização dos circuitos.

A Figura 9.13 mostra o digrama de tempo, produzido com gtkwave, da simulação do mux-2. As entradas estão no vetor de bits `entr_2`, o sinal de seleção é `s0` e a saída simulada é o sinal `saidamux2`. O sinal `esperadamux` é a saída esperada para um modelo correto *de acordo com a especificação funcional*, tal como aquela no Programa 9.9.

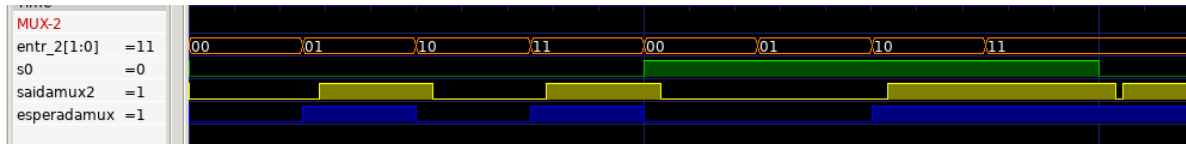


Figura 9.13: Pulsos transitórios na saída do multiplexador.

A menos do atraso de propagação – o sinal `saidamux2` está atrasado com relação à `esperadamux` – o comportamento do modelo é idêntico ao especificado. *Sempre?* No lado direito, após o último valor de teste (`entr_2="11"`), há um pulso transitório em `saidamux2`, que seguramente não deveria estar ali. Reveja a Seção 5.4.3 para a causa deste pulso e para um acréscimo ao circuito do multiplexador que elimina o problema.

Note ainda que as entradas permanecem como `entr_2="11"` após o último vetor de teste. Se as entradas, ou `esperadamux`, tivessem retornado para zero após o último vetor de testes da sequência (`entr_2="11"` para `entr_2="00"`) então é possível que o pulso espúrio passasse despercebido e o modelo estaria errado. Ao gerar vetores de testes é importante, mais que isso, é *fundamental* considerar casos limite que podem esconder ou elucidar justamente o comportamento transitório que viola a especificação do circuito.

9.3.5 Modelo temporizado do somador de 16 bits

Nesta seção veremos a codificação em VHDL dos modelos para os somadores apresentados na Seção 6.6, um com adiantamento de vai-um de quatro em quatro bits, e outro com adiantamento de 16 em 16 bits. Iniciemos com o circuito para o adiantamento de 4 em 4 bits.

Primeiramente, vejamos um comando assaz interessante que é o **generate**. Este é um tipo especial de laço que, ao invés de iterar durante a simulação do modelo, ele itera durante a síntese (compilação) e replica as linhas de código em seu corpo ajustando os índices de acordo com o espaço de iteração. Ao invés de a repetição ocorrer durante a execução, ela se dá sobre o código fonte e permite escrever descrições concisas de estruturas cuja forma se repete de uma maneira relativamente simples.

O trecho de código abaixo replica a linha em seu corpo – que equivale a uma porta `and2` – com o índice *i* variando de 3 até 0 e produzindo quatro réplicas de uma porta `and2` com saída `c(i)` e entradas `a(i)` e `b(i)`.

```
gen: for i in 3 downto 0 generate
  c(i) <= (a(i) and b(i)) after p_and2;
end generate gen;
```

É possível adicionar informação de temporização aos sinais gerados, como o tempo de propagação indicado acima. No que segue, empregamos o modelo mais completo de temporização, com tempos de propagação (**after**) e contaminação (**reject**):

```
gen: for i in 3 downto 0 generate
  c(i) <= reject (r_and2) inertial (a(i) and b(i)) after p_and2;
end generate gen;
```

Adiantamento de 4 em 4 bits

O Programa 9.25 mostra o modelo do circuito de adiantamento de vai-um para 4 bits. O código é uma transcrição literal das Equações 6.20 a 6.23, com informação de tempo de propagação associada aos vários sinais. Para modelar a temporização dos sinais, por exemplo, o atraso do sinal `vai(3)` é a soma do atraso dominante, que é aquele na porta `or5` mais o atraso na porta `and5`. Este modelo não é estrutural mas funcional: ao invés da estrutura de interligação das portas lógicas, o código é uma transcrição para VHDL das Equações 6.20 a 6.23.

Programa 9.25: Adiantamento de vai-um para quatro bits.

```

use work.p_wires.all;
entity adianta4 is
  port(a,b : in reg4;           — entradas A(i),B(i)
        vem : in bit;          — vem-um
        vai : out reg4);       — vai(i)
end entity adianta4;

architecture funcional of adianta4 is
  signal p,g : reg4;
begin
  gen_g_p: for i in 3 downto 0 generate
    g(i) <= reject (r_and2) inertial (a(i) and b(i)) after p_and2;
    p(i) <= reject (r_or2)  inertial (a(i) or b(i)) after p_or2;
  end generate gen_g_p;

  vai(0) <= g(0) or (p(0) and vem) after (p_and2+p_or2);
  vai(1) <= g(1) or (p(1) and g(0)) or (p(1) and p(0) and vem)
    after (p_and3+p_or3);
  vai(2) <= g(2) or (p(2) and g(1)) or (p(2) and p(1) and g(0)) or
    (p(2) and p(1) and p(0) and vem) after (p_and4+p_or4);
  vai(3) <= g(3) or (p(3) and g(2)) or (p(3) and p(2) and g(1)) or
    (p(3) and p(2) and p(1) and g(0)) or
    (p(3) and p(2) and p(1) and p(0) and vem)
    after (p_and5+p_or5);
end architecture funcional;

```

Para simplificar o código, considera-se somente o tempo de propagação para a geração dos sinais `vai(i)`. Para modelar a contaminação, o código seria

```

vai(0) <= reject (r_or2) inertial (g(0) or (p(0) and vem))
  after (p_and2+p_or2);

```

sem esquecer que o intervalo de rejeição deve ser o mínimo dentre todos os caminhos. Neste caso, o caminho mais curto entre as entradas e a saída é `g(0) or (...)`.

O somador de 16 bits, com os circuitos de adiantamento de 4 em 4 bits é mostrado na Figura 6.17 e seu modelo no Programa 9.26, que é um modelo estrutural.

A cadeia de adiantamento não é usada porque os bits de vai-um de cada quarteto são gerados pelo adiantador correspondente. A cada quatro bits, o bit de vai-um (`vai(3)` no Programa 9.25) produzido pelo circuito adiantador é inserido na entrada de vem-um dos bits nas posições 4, 8 e 12. O bit de vai-um produzido pelo quarto adiantador (`c(15)`) é o próprio vai-um do somador de 16 bits.

Os sinais de vai-um gerados em cada somador completo (addBit) são ignorados e por isso estão marcados como **open**.

Programa 9.26: Somador de 16 bits com adiantamento de 4 em 4 bits.

```

architecture adderAdianta4 of adder16 is
  component addBit port (bitA, bitB, vem : in bit;
                        soma, vai      : out bit);
  end component addBit;

  component adianta4 port (a,b : in reg4; vem : in bit;
                        vai: out reg4);
  end component adianta4;

  signal c : reg16;           — cadeia de adiantamento de vai-um
begin
  U_a0_3: adianta4 port map — adianta vai(0)..vai(3)
    (A(3 downto 0),B(3 downto 0),vem,c(3 downto 0));

  U_b0: addBit port map ( A(0),B(0),vem, S(0),open );
  U_b1: addBit port map ( A(1),B(1),c(0),S(1),open );
  U_b2: addBit port map ( A(2),B(2),c(1),S(2),open );
  U_b3: addBit port map ( A(3),B(3),c(2),S(3),open );

  U_a4_7: adianta4 port map — adianta vai(4)..vai(7)
    (A(7 downto 4),B(7 downto 4),c(3),c(7 downto 4));

  U_b4: addBit port map ( A(4),B(4),c(3),S(4),open );
  U_b5: addBit port map ( A(5),B(5),c(4),S(5),open );
  U_b6: addBit port map ( A(6),B(6),c(5),S(6),open );
  U_b7: addBit port map ( A(7),B(7),c(6),S(7),open );

  U_a8_b: adianta4 port map — adianta vai(8)..vai(11)
    (A(11 downto 8),B(11 downto 8),c(7),c(11 downto 8));

  U_b8: addBit port map ( A(8), B(8), c(7), S(8), open );
  U_b9: addBit port map ( A(9), B(9), c(8), S(9), open );
  U_ba: addBit port map ( A(10),B(10),c(9), S(10),open );
  U_bb: addBit port map ( A(11),B(11),c(10),S(11),open );

  U_a12_15: adianta4 port map — adianta vai(12)..vai(15)
    (A(15 downto 12),B(15 downto 12),c(11),c(15 downto 12));

  U_bc: addBit port map ( A(12),B(12),c(11),S(12),open );
  U_bd: addBit port map ( A(13),B(13),c(12),S(13),open );
  U_be: addBit port map ( A(14),B(14),c(13),S(14),open );
  U_bf: addBit port map ( A(15),B(15),c(14),S(15),open );

  vai <= c(15);           — vai-um, sinal da interface
end architecture adderAdianta4;

```

Adiantamento de 16 bits

O modelo do circuito de adiantamento de 16 bits é mostrado no Programa 9.27. O código é uma transcrição literal das Equações 6.24, 6.25 e 6.26. Como VHDL ignora maiúsculas e minúsculas, no modelo os sinais P_i e G_i são chamados de pp(i) e gg(i), respectivamente. Os comandos que geram os sinais pp(i) e gg(i) poderiam ser substituídas por comandos **generate**, com expressões apropriadas para os índices.

Programa 9.27: Adiantamento de 16 em 16 bits.

```

use work.p_wires.all;
entity adianta16 is
    port(a,b : in reg16;           — entradas A(i),B(i)
          vem : in bit;            — vem-um
          vai : out reg4);         — vai(i), de 4 em 4 bits
end adianta16;

architecture funcional of adianta16 is
    signal p,g : reg16;
    signal pp,gg : reg4;
begin
    gen: for i in 15 downto 0 generate
        g(i) <= reject (r_and2) inertial (a(i) and b(i)) after p_and2;
        p(i) <= reject (r_or2)  inertial (a(i) or  b(i)) after p_or2;
    end generate gen;

    pp(0) <= p(3) and p(2) and p(1) and p(0) after p_and4;
    pp(1) <= p(7) and p(6) and p(5) and p(4) after p_and4;
    pp(2) <= p(11) and p(10) and p(9) and p(8) after p_and4;
    pp(3) <= p(15) and p(14) and p(13) and p(12) after p_and4;

    gg(0) <= g(3) or (p(3) and g(2)) or (p(3) and p(2) and g(1)) or
              (p(3) and p(2) and p(1) and g(0)) after p_or4+p_and4;
    gg(1) <= g(7) or (p(7) and g(6)) or (p(7) and p(6) and g(5)) or
              (p(7) and p(6) and p(5) and g(4)) after p_or4+p_and4;
    gg(2) <= g(11) or (p(11) and g(10)) or (p(11) and p(10) and g(9)) or
              (p(11) and p(10) and p(9) and g(8)) after p_or4+p_and4;
    gg(3) <= g(15) or (p(15) and g(14)) or (p(15) and p(14) and g(13)) or
              (p(15) and p(14) and p(13) and g(12)) after p_or4+p_and4;

    vai(0) <= gg(0) or (pp(0) and vem) after p_or2+p_and2;
    vai(1) <= gg(1) or (pp(1) and gg(0)) or (pp(1) and pp(0) and vem)
              after p_or3+p_and3;
    vai(2) <= gg(2) or (pp(2) and gg(1)) or (pp(2) and pp(1) and gg(0)) or
              (pp(2) and pp(1) and pp(0) and vem) after p_or4+p_and4;
    vai(3) <= gg(3) or (pp(3) and gg(2)) or (pp(3) and pp(2) and gg(1)) or
              (pp(3) and pp(2) and pp(1) and gg(0)) or
              (pp(3) and pp(2) and pp(1) and pp(0) and vem)
              after p_or5+p_and5;
end architecture funcional;

```

O modelo estrutural do somador de 16 bits com adiantamento de vai-um otimizado é mostrado no Programa 9.28. O modelo é similar àquele do Programa 9.26, exceto pela instanciação do

circuito de adiantamento de 16 bits, e que os sinais de vem-um dos circuitos de adiantamento de 4 em 4 bits são gerados pelo adiantador de 16 bits, nas posições 4, 8 e 12. O sinal de vai-um do somador é o bit mais significativo do adiantador de 16 bits, $cc(3)$.

Programa 9.28: Somador de 16 bits com adiantamento de 16 bits.

```

architecture adderAdianta16 of adderAdianta16 is
  component addBit port( ... ) end component addBit;
  component adianta4 port( ... ) end component adianta4;
  component adianta16 port( ... ) end component adianta16;

  signal v : reg16;      — adiantamento de 4 em 4 bits
  signal cc : reg4;      — adiantamento de 16 bits
begin
  U_a15_0: adianta16 port map (A,B,vem,cc);

  U_a3_0: adianta4 port map
    (A(3 downto 0),B(3 downto 0),vem,v(3 downto 0));

  U_b0: addBit port map ( A(0),B(0),vem, S(0),open );
  U_b1: addBit port map ( A(1),B(1),v(0),S(1),open );
  U_b2: addBit port map ( A(2),B(2),v(1),S(2),open );
  U_b3: addBit port map ( A(3),B(3),v(2),S(3),open );

  U_a4_7: adianta4 port map
    (A(7 downto 4),B(7 downto 4),cc(0),v(7 downto 4));

  U_b4: addBit port map ( A(4),B(4),cc(0),S(4),open );
  U_b5: addBit port map ( A(5),B(5), v(4),S(5),open );
  U_b6: addBit port map ( A(6),B(6), v(5),S(6),open );
  U_b7: addBit port map ( A(7),B(7), v(6),S(7),open );

  U_a8_11: adianta4 port map
    (A(11 downto 8),B(11 downto 8),cc(1),v(11 downto 8));

  U_b8: addBit port map ( A(8), B(8), cc(1), S(8),open );
  U_b9: addBit port map ( A(9), B(9), v(8), S(9),open );
  U_ba: addBit port map ( A(10),B(10), v(9),S(10),open );
  U_bb: addBit port map ( A(11),B(11),v(10),S(11),open );

  U_a12_15: adianta4 port map
    (A(15 downto 12),B(15 downto 12),cc(2),v(15 downto 12));

  U_bc: addBit port map ( A(12),B(12),cc(2),S(12),open );
  U_bd: addBit port map ( A(13),B(13),v(12),S(13),open );
  U_be: addBit port map ( A(14),B(14),v(13),S(14),open );
  U_bf: addBit port map ( A(15),B(15),v(14),S(15),open );

  vai <= cc(3);
end architecture adderAdianta16;

```

Comparação de desempenho

Finalmente estamos numa posição que nos permite comparar o desempenho dos três modelos de somadores vistos até aqui, aquele sem adiantamento de vai-um (Prog. 9.15), aquele com adiantamento de 4 em 4 bits (Prog. 9.26), e o somador com adiantamento de 16 bits (Prog. 9.28). Os valores empregados para os tempos de propagação e de contaminação das portas lógicas são consistentes para os quatro modelos e servem como medidas quantitativas, ao invés de medidas absolutas de velocidade para cada um dos circuitos. O *testbench* usado para excitar os somadores é similar àquele do Programa 9.16.

O diagrama de tempos na Figura 9.14 mostra a evolução dos resultados ao longo do tempo. As entradas são os sinais *inpa* e *inpb*, e são mostrados o resultado (*rescad*) e o sinal de vai-um (*vaicad*), para o somador sem adiantamento de vai-um, *rescad4* e *vaiad4* para o somador com adiantamento de 4 em 4 bits, e *rescad16* e *vaiad16* para o somador com adiantamento de 16 bits.

Para as entradas, mostrados em hexadecimal como 0001 e FFFF, o somador sem adiantamento produz resultado estável após 800ns, quando então o sinal *vaicad* fica em 1 e o resultado estabiliza em 0000: $0001 + \text{FFFF} = 1.0000$. O diagrama mostra a sequência de 15 alterações do resultado enquanto os somadores de um bit computam seus resultados bit a bit – a cada alteração é computado um bit definitivo do resultado: $\text{FFFE} \rightarrow \text{FFFC} \rightarrow \text{FFF8} \rightarrow \text{FFF0} \dots 8000 \rightarrow 0000$. O sinal *vaicad* estabiliza 20ns depois que o resultado estabiliza em 0000.

O somador com adiantamento de 4 em 4 bits produz resultado estável após 433ns, e o sinal *vaicad4* se define 10ns antes do resultado estabilizar em 0000. A sequência de valores é a mesma que para o primeiro somador, a menos do intervalo mais curto em cada alteração. Uma vez que o adiantador tenha computado os bits de vai-um, os resultados do quarteto estabilizam rapidamente.

O somador com adiantamento de 16 bits produz resultado estável após 292ns e a sequência de valores é mais curta, com 10 valores intermediários, ao invés de 15, como nos outros dois somadores. O sinal *vaicad16* estabiliza após 200ns, e uma vez que todos os bits intermediários de vai-um são computados, os bits do resultado final são obtidos de dois em dois, nas quatro últimas alterações do resultado – a escala horizontal do diagrama esconde estas variações.

Do ponto de vista de desempenho global, e com o modelo de temporização relativamente simples empregado aqui, o somador com adiantamento de 4 em 4 bits é $800\text{ns}/433\text{ns} = 1,85$ vezes mais rápido que o modelo sem adiantamento, enquanto que o modelo com adiantamento de 16 bits é $800\text{ns}/292\text{ns} = 2,74$ vezes mais rápido. Os ganhos de desempenho são consideráveis, e são obtidos à custa de acréscimos não triviais no número de portas lógicas, de maior complexidade do projeto, e no tempo adicional para depuração e testes.

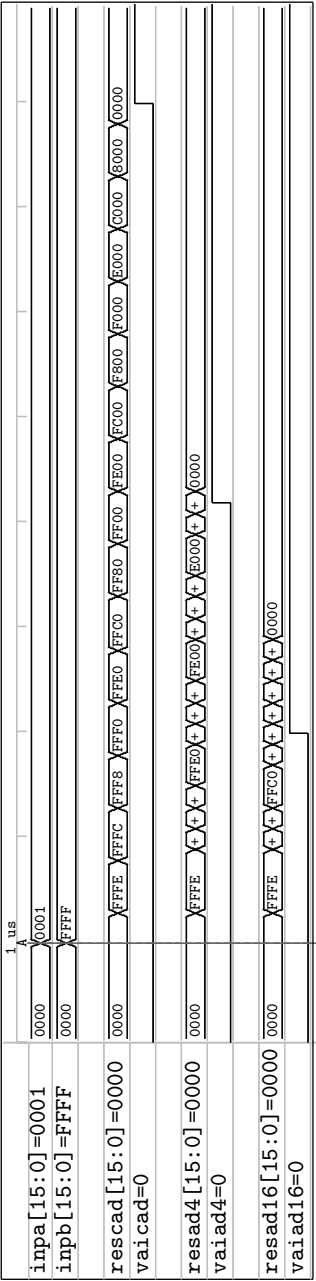


Figura 9.14: Comparação do tempo de propagação dos três somadores para 0001+FFFF.

9.4 Modelagem de circuitos sequenciais em VHDL

Reveja a definição de *circuitos sequenciais síncronos* da Seção 7.6.

9.4.1 Revisão: como funciona o mecanismo de simulação

VHDL emprega *simulação de eventos discretos*: os eventos ocorrem nos instantes determinados pela própria simulação. Por enquanto, consideremos que um *processo* é tão simples quanto uma atribuição; veremos em breve processos ‘completos’. Uma simulação consiste de:

fase de inicialização – todos os sinais são inicializados com os valores declarados, ou com os menores valores do tipo de cada sinal; tempo simulado $\leftarrow 0$; todos os processos são executados exatamente uma vez;

passo de simulação – primeira etapa – o tempo simulado avança até o próximo instante em que uma *transação* está programada; todas as transações programadas para aquele instante são executadas; estas transações podem provocar *eventos* em sinais;

passo de simulação – segunda etapa – processos que dependem dos eventos disparados na primeira etapa são então executados; depois que todos os processos executam, a simulação avança até o instante em que a próxima transação está programada.

Uma *transação* decorre da atribuição a um sinal;

- a atribuição será efetivada no próximo *delta* (Δt);
- se o sinal muda de estado, então este sinal sofre um *evento*, na primeira etapa;
- um evento no sinal S causa a execução dos processos que dependem de S , na segunda etapa.

Em nas simulações geradas a partir do compilador ghdl, o femtossegundo ($1\text{fs} = 10^{-15}\text{s}$) é a menor unidade de tempo simulado, e ocorrem $1.000 \Delta t$ em cada femtossegundo.

Em resumo:

- (1) O tempo simulado avança em função de *eventos* nos sinais;
- (2) a cada mudança no estado de um sinal corresponde um *evento*;
- (3) em função de um evento num sinal que aparece no lado direito de uma atribuição, a expressão é reavaliada, e se houver mudança no valor da expressão, o novo estado é atribuído ao sinal do lado esquerdo (primeira etapa);
- (4) todos os comandos são avaliados e os eventos disparados passarão a ter efeito no próximo *delta*;
- (5) enquanto houver eventos no *delta* corrente, estes são avaliados e eventos resultantes são programados para o próximo *delta*;
- (6) processos que dependem dos eventos disparados neste *delta* são executados (segunda etapa);
- (7) eventos causados pela execução dos processos são programados para o próximo *delta*; e
- (8) eventos podem ser programados para ocorrer no futuro com **after**.

9.4.2 Comandos Sequenciais

Até agora empregamos somente os *comandos concorrentes* de VHDL. Estes comandos nos permitem escrever e verificar modelos para circuitos combinacionais pela interligação de componentes que são, eles próprios modelados como circuitos combinacionais.

Para que possamos escrever modelos para *flip-flops* e registradores temos que empregar os *comandos sequenciais* da linguagem. Estes comandos descrevem comportamento em termos de sequências de eventos e sua execução depende de ordem em que aparecem no código fonte. Nesse sentido, comandos sequenciais se assemelham àqueles da linguagem C.

Comandos sequenciais são ‘executados’ em tempo zero numa simulação, e *só podem ser usados em processos*. Comandos sequenciais podem, também, ser usados para descrever lógica combinacional.

9.4.3 Processos

O *processo* é a construção de VHDL que encapsula um ou mais comandos sequenciais, que modelam uma determinada funcionalidade, tais como um *flip-flop*, um registrador ou uma máquina de estados.

Processos são definidos dentro de arquiteturas, e numa arquitetura, todos os processos devem ter nomes distintos.

Exemplo 9.1 Vejamos um exemplo simples de processo, para que o mecanismo de simulação fique mais claro. Nosso primeiro exemplo é um *gerador de sinal de relógio*. Desejamos um período de 50ns e que o sinal seja uma onda quadrada.. Usaremos uma constante para definir o período:

```
constant CLOCK_PER : time := 50 ns;
```

O código do gerador de relógio é mostrado no Programa 9.29. O processo U_clock é executado na fase de inicialização: o sinal clk é inicializado em '0' e a execução do processo é suspensa no primeiro **wait**, e este processo é escalonado para ser executado depois de 25ns de tempo simulado.

Programa 9.29: Processo gerador de relógio.

```
U_clock: process
begin
  clk <= '0';           — executa
  wait for CLOCK_PER / 2; — espera meio ciclo
  clk <= '1';           — volta a executar
  wait for CLOCK_PER / 2; — espera meio ciclo e volta ao topo
end process U_clock;
```

Decorrido o primeiro semiciclo de 25ns, o processo volta a executar a partir do comando que segue o **wait** que causou a suspensão: ao sinal clk é atribuído '1' e o processo se suspende no segundo **wait**, novamente a esperar por 25ns de tempo simulado.

Decorrido este segundo intervalo de 25ns, o processo volta a executar do seu início, e clk torna-se '0' novamente. Assim, o processo executa periodicamente, e a cada execução o nível lógico do sinal de relógio é invertido.

9.4.4 Processos *sem* lista de sensibilidade – *wait*

Processos podem ser usados sem uma *lista de sensibilidade*. Estes processos devem conter um ou mais comandos que suspendem temporariamente a execução do processo, tal como o comando **wait**. Quando um destes comandos é executado, o processo fica suspenso até que a condição de espera seja resolvida. No código abaixo, quando o comando **wait on** é executado e a condição for falsa, a execução do processo é suspensa até que a condição de espera se torne verdadeira, quando então a execução é retomada no comando após o **wait**.

```
nomeDoProcesso: process    — sem lista de sensibilidade
  declarações;
begin
  ...
  wait on (condição) ;    — comando que suspende execução
  x <= ...;                — reinicia neste ponto
end process nomeDoProcesso;
```

Processos podem *ou* conter uma lista de sensibilidade *ou* conter comandos **wait**, mas nunca os dois num mesmo processo.

O Exemplo 9.1 mostra um processo sem lista de sensibilidade que produz um sinal periódico, que pode ser usado como um sinal de relógio.

Exemplo 9.2 O processo U_reset, no Programa 9.30, gera um sinal aperiódico que pode ser usado na inicialização de um circuito sequencial síncrono. Na fase de inicialização, ao sinal reset é atribuído '0' na linha 3 e então o processo se suspende por 3/4 de ciclo na linha 4. Decorrido este intervalo, o processo volta a executar e o sinal reset muda para '1' e o processo se suspende no **wait** da linha 6 e nunca mais é executado. ◀

Programa 9.30: Gerador de reset.

```
U_reset: process                — sem lista de sensibilidade
begin
  reset <= '0';                 — executa e
  wait for CLOCK_PER * 0.75;    — espera por 37,5ns
  reset <= '1';                 — volta a executar e
  wait;                        — se suspende para sempre
end process U_reset;
```

Existem quatro tipos de condição de parada com **wait**:

- (i) **wait for** intervalo; a execução suspende por um certo tempo (**wait for** 10 ns);
- (ii) **wait until** condição; suspende até que a expressão booleana da condição se torne verdadeira (**wait until** (received = true));
- (iii) **wait on** (lista de sinais); suspende até a ocorrência de um evento em algum dos sinais na lista de sinais (**wait on** (clk, reset, received));
- (iv) **wait**; sem condição, espera para sempre, finalizando a execução do processo.

Um comando **wait** pode combinar mais de uma cláusula de espera, como por exemplo

```
wait on (reset, kill) for 100~ns;
```

O processo fica a esperar por um evento nos sinais reset ou kill ou o decurso de 100ns.

9.4.5 Processos *com* lista de sensibilidade

Processos podem conter uma *lista de sensibilidade*, como mostrado no trecho de código abaixo.

```
nomeDoProcesso: process ( listaDeSensibilidade )
  declarações;
begin
  comando_sequencial_1;
  comando_sequencial_2;
  comando_sequencial_3;
end process nomeDoProcesso;
```

Os sinais que são atribuídos em um processo retêm seus valores até a próxima execução daquele processo. A execução de um processo pode ser encarada como ocorrendo em tempo zero, e os processos executam paralelamente com demais comandos concorrentes no corpo da arquitetura. Pode-se dizer que o processo inteiro é um comando concorrente, mas os comandos do corpo do processo são executados sequencialmente. Um evento em qualquer das variáveis na lista de sensibilidade de um processo dispara a sua execução, quando então os comandos no corpo do processo são executados.

Na inicialização de uma simulação, todos os processos são executados exatamente uma vez. Nos passos de simulação, quando ocorre um evento num sinal da *lista de sensibilidade* o processo é escalonado para execução na segunda etapa do delta corrente e os comandos sequenciais são executados na ordem do código fonte.

Se a execução dos comandos provoca eventos em sinais, estes eventos são escalonados para o próximo delta, ou para um instante futuro se a cláusula **after** for usada.

Quando a execução chega ao final do corpo do processo, ela é retomada do primeiro comando sequencial, no topo do processo. O corpo do processo se assemelha a um laço infinito, que só é executado quando ocorrer um evento em sinais da sua lista de sensibilidade. Esta é uma descrição inexata, mas ela será retificada em breve.

O atributo `event`, aplicado a um sinal (`x'event`) tem o valor TRUE se ocorreu um evento em `x` no delta corrente.

Exemplo 9.3 O Programa 9.31 mostra um exemplo de processo com lista de sensibilidade. Sempre que ocorrer um evento no sinal `clk`, o corpo do processo é executado *em tempo zero*.

O atributo `clk'event` indica que ocorreu um evento no sinal `clk` e este evento provocou a execução do processo porque `clk` está na lista de sensibilidade. Se o evento colocou `clk` em '1' então ocorreu uma borda ascendente em `clk` – o sinal se alterou (evento) e a alteração fez com que o sinal ficasse em '1'. Se ocorreu uma borda ascendente em `clk`, então o valor do sinal `D` é atribuído ao sinal `Q`. Se nenhuma outra atribuição é feita ao sinal `Q`, então ele preserva seu valor até a próxima borda ascendente. Uma função que detecta bordas, chamada `rising_edge()`, é definida na Seção 9.4.13 <

Programa 9.31: Primeira versão de um *flip-flop D*.

```
FF0: process ( clk ) — somente clk na lista de sensibilidade
begin
  if ( clk'event and clk = '1' ) then — clk mudou para '1'
    Q <= D;
  end if;
end process FF0;
```

9.4.6 O que é um *flip-flop*?

Consultando as fontes em papel que estão disponíveis na minha estante, e tomado de alguma surpresa, passo a desvelar à diletta leitora que conceitos que emprego há umas tantas décadas são definidos em dicionários de alta qualidade como sendo:

flip-flop circuit *noun*: an electronic circuit with two permanently stable conditions (as when one electron tube is conducting while the other is cut off) so that conduction is switched from one to the other by successive pulses, Webster's Third New International Dictionary;

flip-flop *verb*: to change your opinion about something, especially when you then hold the opposite opinion, Oxford Advanced Learner's Dictionary;

latch *noun*: to catch or fasten by means of a latch, Webster's Third New International Dictionary;

basculador (1) ELETRÔN circuito flip-flop, Dic. Houaiss da Língua Portuguesa⁴; e

básculo (1) ponte levadiça com mecanismo de contrapeso, (2) peça móvel de metal ou ferro, que gira apoiada num pino, destinada a abrir ou fechar ferrolhos de portas, janelas, etc, Dic. Houaiss da Língua Portuguesa.

9.4.7 *Flip-flops* em VHDL

Em se tratando de sistemas digitais, um *flip-flop* é um circuito cuja saída, em condições normais de operação, pode estar em um de dois possíveis estados: ou a saída está no estado *flip*, ou está no estado *flop* – um *flip-flop* é um circuito *biestável* cujo comportamento é definido para somente uma de duas condições: ou sua saída é estável em 1, ou é estável em 0. Um *flip-flop* é um modelo para uma ponte levadiça ou um ferrolho: ou estão abertos, ou estão fechados.

Exemplo 9.4 *Flip-flop* com wait Vejamos o código que modela um *flip-flop*, empregando um processo sem lista de sensibilidade. A entidade FF0 descreve a interface de um *flip-flop* tipo D simples, com entrada D, relógio, e saída Q, e é mostrada no Programa 9.32.

A arquitetura waitSimple modela um *flip-flop* tipo D. O atributo 'event' indica ocorrência de evento no sinal clk. O processo executa e fica bloqueado no **wait** a esperar por um evento no sinal clk, e além disso, que o evento faça clk='1'. Estas duas condições equivalem a uma borda ascendente no sinal clk: ocorreu uma mudança, e a mudança foi para 1. Após a detecção da borda, a entrada D é copiada para o sinal Q, que mantém seu estado até a próxima borda em clk.

Quando o processo é desbloqueado no **wait**, a execução prossegue até o fim do processo e é retomada no seu topo. Nesse exemplo, após o desbloqueio acontece a atribuição (Q<=D;) e a execução do processo é logo interrompida novamente no **wait**. <

⁴Este me parece um cacófato assaz desagradável. O autor, que é adepto do *uber*-dicionarista, admite que emprega, desavergonhadamente, a forma incorreta 'básculo' ao invés do horrendo 'basculador'. Atravessemos o Rubicão sem mais delongas.

Programa 9.32: *Flip-flop com wait.*

```

1  entity FF0 is
2      port(D:    in  bit;
3            clk: in  bit;
4            Q:    out bit);
5  end FF0;
6
7  architecture waitSimples of FF0 is
8  begin
9      FF: process
10     begin
11         wait for ( clk'event and clk = '1' ); — borda em clk
12         Q <= D;
13     end process FF;
14 end architecture waitSimples;

```

Exemplo 9.5 *Flip-flop com lista de sensibilidade* A arquitetura `listaSimples`, no Programa 9.33, emprega uma lista de sensibilidade ao invés de um comando `wait` no corpo do processo. A lista de sensibilidade junto à palavra reservada `process` contém somente o sinal `clk`, e sempre que ocorrer um evento neste sinal, o processo `FF` é executado. O corpo de um processo com lista de sensibilidade é executado do início ao final sempre que ocorrer um evento num dos sinais da lista.

Programa 9.33: *Flip-flop com lista de sensibilidade.*

```

1  architecture listaSimples of FF0 is
2  begin
3      FF: process (clk) — eventos em clk causam execução
4      begin
5          if clk = '1' then — valor pós-evento = '1'
6              Q <= D;
7          end if;
8      end process FF;
9  end architecture listaSimples;

```

O corpo do processo, que é o comando `if`, só é executado se ocorrer um evento em `clk`. Se, após o evento `clk='1'`, então ocorreu uma borda ascendente naquele sinal e a entrada deve ser registrada; se `clk` tem qualquer outro valor após o evento, não ocorreu uma borda ascendente, e o valor em `Q` deve permanecer como estava. O corpo do processo é ‘executado’ *em tempo zero*, somente após eventos no sinal `clk`. <

Exemplo 9.6 Flip-flop com reset assíncrono Vejamos como modelar um *flip-flop* com uma entrada de *reset* assíncrona. A entidade FF1, no Programa 9.34, declara duas entradas de controle, *clk* e *rst*. O processo na arquitetura *rstAssincrono* tem dois sinais na sua lista de sensibilidade, *clk* e *rst*. O **if** faz *Q*='0' sempre que ocorrer um evento em *rst*, e o evento fizer *rst*=0. O **elsif** testa a ocorrência de uma borda ascendente em *clk*, e se este for o caso, a entrada *D* é registrada. Em qualquer outra combinação nas entradas, o sinal *Q* mantém o seu valor.

A execução do processo pode ocorrer tanto por eventos em *rst* quanto em *clk*. É por isso que a cláusula **elsif** verifica se ocorreu um evento em *clk*, e mais ainda, se *clk*=1 após o evento. ◁

Programa 9.34: Flip-flop com reset assíncrono.

```

1  entity FF1 is
2      port(D:          in  bit;
3            clk, rst: in  bit;
4            Q:          out bit);
5  end FF1;
6
7  architecture rstAssincrono of FF1 is
8  begin
9      FF: process (clk, rst) — lista de sensibilidade
10     begin
11         if rst = '0' then
12             Q <= '0';
13         elsif clk'event and clk = '1' then — borda em clk
14             Q <= D;
15         end if;
16     end process FF;
17 end architecture rstAssincrono;
```

Espaço em branco proposital.

Exemplo 9.7 *Flip-flop com set e reset assíncronos* A entidade FF2, no Programa 9.35, declara três entradas de controle, clk, clr e set.

O processo na arquitetura setclr tem três sinais na sua lista de sensibilidade. O comando **if** coloca a saída Q em 0 caso rst='0'; o primeiro **elsif** a coloca em 1 caso set='0', ou registra a entrada D caso ocorra uma borda ascendente em clk. Em qualquer outra combinação de entradas, Q mantém o seu valor.

Este exemplo evidencia a distinção entre a lista de sensibilidade e o uso de **waits**: com a lista de sensibilidade, caso ocorram eventos nos três sinais no mesmo delta, o código do processo decide a ordem de avaliação e as consequências dos eventos. Se **waits** forem usados, a lógica das condições de espera pode ficar mais complexa, e o código confuso e de difícil compreensão. ◁

Programa 9.35: *Flip-flop com set e reset assíncronos.*

```

1  entity FF2 is
2      port(D:           in  bit;
3            clk, rst, set: in  bit;
4            Q:          out bit);
5  end FF2;
6
7  architecture setclr of FF2 is
8  begin
9      FF: process (clk, rst, set) — lista de sensibilidade
10     begin
11         if rst = '0' then          — reset tem precedência
12             Q <= '0';
13         elsif set = '0' then
14             Q <= '1';
15         elsif clk'event and clk = '1' then — borda em clk
16             Q <= D;
17         end if;
18     end process FF;
19 end architecture setclr;
```

Num *flip-flop* real, a ocorrência de rst='0' e set='0' é uma condição proibida e que viola a especificação temporal daquele circuito. Caso esta situação ocorra, o valor no sinal Q é indeterminado, e por causa da metaestabilidade, pode permanecer indeterminado por um tempo também indeterminado. O comportamento modelado na arquitetura setClr é uma versão idealizada do circuito e que abstrai a possibilidade de metaestabilidade para simplificar a modelagem e a simulação.

Equivalência entre *lista de sensibilidade* e *wait*

O Programa 9.36 mostra dois processos com comportamento equivalente, o primeiro com lista de sensibilidade e o segundo com um comando **wait**.

Programa 9.36: Processos com lista de sensibilidade e com wait.

```

architecture doisProcessos of SisSeqSin is
begin

  FFlista: process (clk, rst) — COM lista de sensibilidade
  begin
    if rst = '0' then
      Q <= '0';
    elsif clk'event and clk = '1' then
      Q <= D;
    end if;
  end process FFlista;

  FFwait: process — SEM lista de sensibilidade
  begin
    if rst = '0' then
      Q <= '0';
    elsif clk'event and clk = '1' then
      Q <= D;
    end if;
    wait on (clk, rst); — espera eventos em clk, rst
  end process FFwait;

end architecture doisProcessos;

```

9.4.8 Sinais e Variáveis

Enquanto falávamos da modelagem de circuitos combinacionais, sinais eram os equivalentes aos fios de um circuito combinacional – a cada delta, o valor atribuído a um sinal é o resultado da avaliação do lado direito da expressão – RHS ou *Right Hand Side*, ou o *rvalue*, no jargão de compiladores – que lhe atribui valor.

Quando modelamos circuitos sequenciais, alguns sinais “adquirem memória” e mantêm seus valores entre as execuções dos processos que os manipulam. Os valores transportados por estes sinais são alterados pela combinação de eventos nas listas de sensibilidade dos processos, e do código sequencial que determina seus valores em função dos eventos apropriados. Sinais somente são atualizados no próximo delta, ou no evento “próxima borda do sinal de relógio” em circuitos sequenciais síncronos.

Frequentemente é necessário empregar ‘fios’ que transportam informação instantaneamente entre os comandos de um processo e estes ‘fios’ não podem ser sinais, porque estes somente são atualizados em um delta futuro, e não no delta corrente. Quando é necessária a comunicação instantânea de valores entre os comandos de um processo, deve-se empregar variáveis. Uma *variável* transporta valores entre comandos sequenciais em processos porque um novo valor é atribuído à variável imediatamente, e não num delta futuro.

Variáveis só podem ser declaradas dentro de processos, e a atribuição a uma variável é denotada por `:=`, como `v := '1'`.

As portas de saída de uma entidade não podem ser lidas numa arquitetura ou processo. Para que seja possível ler valores de estado que são atribuídos às saídas, devem ser empregadas variáveis para manter o estado do modelo, e estas variáveis são atribuídas às portas de saída da entidade.

Veremos exemplos em breve.

Atributos de sinais e de vetores

Atributos fornecem informação adicional sobre vários tipos de objetos em VHDL. Dentre muitos, nos interessam os *atributos de sinais* e os *atributos de vetores*.

Para um sinal `S`, nos interessam os atributos:

`S'event` é *true* se ocorreu evento no sinal neste delta;

`S'active` é *true* se alguma transação ocorreu no sinal neste delta;

`S'last_value` retorna o valor de `S`, antes do último evento.

Já empregamos a atributo `'event` nos modelos dos *flip-flops*. A condição para a detecção da borda pode ser ainda mais estrita, ao se verificar que o valor no delta anterior era mesmo `'0'` e não um valor inválido, por exemplo.

```
if( clk'event and clk='1' and clk'last_value='0' ) then
    ...
end if;
```

Para um vetor `V`, nos interessam os atributos:

`V'length` retorna o número de elementos de um vetor;

`V'range` retorna a faixa de índices de um vetor, ou se o vetor é um tipo restringido (**subtype**), retorna o conjunto de valores do subtipo.

Por exemplo, para os dois vetores de 12 elementos, as suas faixas são aquelas declaradas, *viz.*:

```
V: bit_vector(11 downto 0);
W: bit_vector(0 to 11);
...
... V'range ... — faixa de 11 a 0
...
... W'range ... — faixa de 0 a 11
...
```

Espaço em branco proposital.

9.4.9 Processos na modelagem de lógica combinacional

Processos podem ser usados para modelar lógica combinacional. O código de um “processo combinacional” deve evitar que sinais mantenham o mesmo estado entre duas execuções do processo – o processo não pode ter memória. Para tanto, o código do processo deve obedecer à três regras:

- Regra 1 a lista de sensibilidade contém *todos* os sinais de entrada (sinais lidos no RHS, ou *rvalues*) usados pelo processo;
- Regra 2 as atribuições às saídas do processo cobrem todas as combinações possíveis das entradas do processo;
- Regra 3 a todas as variáveis no processo deve ser atribuído um valor antes que elas sejam usadas como entrada no RHS (*rvalue*).

Exemplo 9.8 Um multiplexador de quatro entradas, que é um circuito combinacional, pode ser modelado com um processo, como mostra o Programa 9.37. O processo mux satisfaz à Regra 1 porque a lista de sensibilidade contém *todos* os sinais que são lidos pelo processo, tanto o sinal Sel que escolhe uma das entradas, quanto as quatro entradas (A,B,C,D). A Regra 2 é obedecida porque, nas quatro combinações da entrada Sel, ocorre uma atribuição ao sinal Y. O processo não emprega variáveis e portanto a Regra 3 é trivialmente satisfeita.

Esta não é a maneira mais eficiente de se codificar um multiplexador porque as entradas são avaliadas em sequência, e na síntese este código resulta num circuito mais complexo do que o necessário. Como está, o circuito resultante da síntese envolve uma cadeia de quatro *ands*. O comando **if** é descrito na Seção 9.4.11.

Em breve veremos um comando VHDL que modela um multiplexador que é sintetizado eficientemente. A *síntese* de um modelo é o processo de compilação que produz uma descrição de baixo nível um circuito – lista de conexões – ao invés de um simulador. ◁

Programa 9.37: Multiplexador modelado com um processo.

```

1  entity simpleMux is
2    port (Sel:           in  bit_vector(0 to 1);
3          A, B, C, D:    in  bit;
4          Y:            out bit);
5  end simpleMux;
6
7  architecture ineficiente of simpleMux is
8  begin
9    mux: process (Sel,A,B,C,D) — lista de sensibilidade
10   begin
11     if Sel = "00" then Y <= A; — todas as 4
12     elsif Sel = "01" then Y <= B; — combinações
13     elsif Sel = "10" then Y <= C; — para 2 bits
14     elsif Sel = "11" then Y <= D;
15     end if;
16   end process mux;
17 end architecture ineficiente;
```

9.4.10 Processos na modelagem de lógica sequencial

Para descrever lógica sequencial, portanto com memória, processos devem obedecer à três regras:

- Regra 1 a lista de sensibilidade não inclui todos os sinais no RHS (*rvalue*) das atribuições;
- Regra 2 a lógica **if—then—elsif** é incompletamente especificada, indicando que um ou mais sinais/variáveis devem manter seus valores entre execuções do processo;
- Regra 3 uma ou mais variáveis mantêm seus valores entre execuções do processo — variáveis podem ser lidas (RHS) antes que um valor lhes tenha sido atribuído.

Quando um processo atende às Regras 1 e 2, e opcionalmente à 3, durante a síntese o compilador gera um circuito com *flip-flops* e registradores para manter os valores entre as execuções do processo, que tipicamente ocorrem na borda do sinal de relógio, ou quando sinais de controle assíncronos (*reset*) são ativados.

Programa 9.38: Modelo para um registrador de deslocamento.

```

1  entity shiftReg is
2    port (clk, rst, load: in bit;
3          serInp:         in bit;
4          data:           in bit_vector(0 to 7);
5          Q:              out bit_vector(0 to 7));
6  end shiftReg;
7
8  architecture correta of shiftReg is
9    begin
10     reg: process (rst, clk)
11       variable Qvar: bit_vector(0 to 7);
12     begin
13       if rst = '1' then           — inicialização assíncrona
14         Qvar := "00000000";
15       elsif (clk'event and clk = '1') then
16         if load = '1' then       — carga síncrona
17           Qvar := data;
18         else                     — rotação síncrona
19           Qvar := Qvar(1 to 7) & serInp;
20         end if;
21       end if;
22       Q <= Qvar;   — atribui estado ao sinal da interface
23     end process;
24  end architecture correta;
```

Exemplo 9.9 A entidade *shiftReg*, no Programa 9.38 modela um registrador de deslocamento com 8 bits, que desloca seu conteúdo a cada ciclo do relógio.

Quando o sinal *load* é ativado, o registrador é carregado com o valor em sua entrada *data*. O processo *reg* declara a variável *Qvar* para manter o conteúdo do registrador entre os eventos no sinal de relógio.

Se o sinal `rst` é ativado, o registrador é carregado com zero. A cada borda do relógio, se o sinal `load` está ativo, um novo valor é atribuído à `Qvar`.

Do contrário, o conteúdo de `Qvar` é deslocado de uma posição – o bit `serInp` é inserido na posição do bit `Q0`, e os demais deslocam-se de uma posição.

Ao final da execução do processo, o conteúdo da variável `Qvar` é atribuído ao sinal da interface `Q`. O valor de `Qvar`, que mantém o estado do registrador, é lido na linha 18 – é usado no RHS – e portanto o sinal da interface `Q` não pode ser usado para manter o conteúdo do registrador.

A Regra 1 é atendida porque o sinal `load` não está na lista de sensibilidade. A Regra 2 é atendida porque os `ifs` não cobrem todas as oito combinações possíveis dos três sinais de controle. A Regra 3 é atendida porque, dependendo das combinações das variáveis de controle (`rst=1` e `load=0`), o valor em `Qvar` pode ser lido antes de que algum valor tenha sido atribuído àquela variável. ◁

Há uma diferença sutil porém importante entre variáveis e sinais: variáveis são atualizadas instantaneamente enquanto que sinais somente são atualizados no próximo delta, ou em momento futuro. No Programa 9.39, a arquitetura `ERRADA` da entidade `shiftReg` é praticamente a mesma que a arquitetura correta – a diferença é que, em `ERRADA`, o valor memorizado em `Qsig` é mantido num sinal e não numa variável. Suponha que não ocorram eventos no sinal `rst`, e portanto o processo `reg` só é executado nas bordas do relógio. O comportamento do modelo é aquele esperado pelo programador?

Programa 9.39: Modelo *errado* para o registrador de deslocamento.

```

1  architecture ERRADA of shiftReg is
2  begin
3      reg: process (rst, clk)
4          signal Qsig: bit_vector(0 to 7);          -- SIGNAL
5          begin
6              if rst = '1' then
7                  Qsig <= "00000000";
8              elsif (clk = '1' and clk'event) then
9                  if load = '1' then
10                     Qsig <= data;
11                 else
12                     Qsig <= Qsig(1 to 7) & serInp;
13                 end if;
14             end if;
15             Q <= Qsig;    -- atribui estado ao sinal da interface
16         end process;
17     end architecture ERRADA;
```

O comportamento do modelo não é o esperado pelo programador porque a atualização do sinal da interface `Q` só terá efeito visível na próxima execução do processo `reg`, embora os dois `ifs` computem o novo valor de `Qsig` no delta corrente. Este sinal, `Qsig`, é atualizado no delta corrente mas a atribuição à `Q` somente acontecerá na próxima execução do processo, no delta em que ocorrer um evento em `clk`, portanto com um ciclo de atraso em relação ao que se deseja.

9.4.11 Comandos Sequenciais

Vejamos quais comandos podem ser empregados em processos. Estes são chamados de *comandos sequenciais* porque são executados na ordem do código fonte, de forma similar aos comandos da linguagem C.

Comando Sequencial IF-THEN-ELSE

Já vimos vários usos do comando **if** nos modelos dos *flip-flops*. O trecho abaixo mostra o comando **if** em toda sua glória. Se condição1 é verdadeira, então os comandos em comandos1 são executados; do contrário, condição2 é avaliada, e se verdadeira, comandos2 são executados; do contrário as duas condições são falsas e comandos3 são executados.

```
if condição1 then           — condição deve ter tipo Boolean
    comandos1;
elsif condição2 then      — cláusula opcional
    comandos2;
else                     — cláusula opcional
    comandos3;
end if;
```

As cláusulas **elsif** e **else** são opcionais, e **ifs** aninham da forma óbvia.

```
if cond_externa then — if 's aninhados
    comandos_externos;
else
    if cond_interna then
        comandos_internos;
    end if;
end if;
```

Comando Sequencial CASE

O comando sequencial **case** é similar ao *switch* da linguagem C. A expressão de escolha é avaliada, e a cláusula que ‘casar’ com o resultado da avaliação é executada.

No exemplo abaixo a expressão controle é avaliada e se uma das cláusulas teste_i for selecionada, o respectivo comando_i é executado. A palavra reservada **others** ‘casa’ todos os valores que não estão explicitamente listados e deve ser a última cláusula do comando. Uma cláusula pode conter mais de um valor, sendo estes separados por uma barra vertical.

```
case controle is
    when teste1 =>
        comando1;
    when teste2 | teste3 => — dois casos por testar
        comando2;
    when others =>         — todos os casos não cobertos acima
        comandoOthers;
end case;
```

Os testes devem ser mutuamente exclusivos, e todas os possíveis valores de controle devem ser cobertos, nem que seja por **others**.

Ao contrario do **if–then–else**, no **case** não há prioridade na avaliação das cláusulas e portanto este comando é mais adequado para a modelagem de circuitos tais como o multiplexador, como mostra a arquitetura eficiente para o multiplexador de quatro entradas no Programa 9.40. Esse comando é sintetizado como um multiplexador de quatro entradas, e não como uma sequência de **ifs** encadeados.

Programa 9.40: Modelo eficiente para processo do multiplexador.

```

1  architecture eficiente of simpleMux is
2  begin
3      mux: process (Sel,A,B,C,D) — lista de sensibilidade
4      begin
5          case Sel is
6              when "00" => Y <= A; — todas as quatro
7              when "01" => Y <= B; — combinações
8              when "10" => Y <= C; — estão cobertas
9              when "11" => Y <= D;
10         end case;
11     end process mux;
12 end architecture eficiente;
```

Verifique se este processo obedece às três regras para processos que modelam circuitos combinacionais.

Comando Sequencial FOR

O comando **for** de VHDL é muito mais expressivo do que seu insípido primo da linguagem C. Infelizmente, este comando não é sintetizável e é usado apenas em *testbenches* para gerar sequências de valores de teste.

A variável de indução de um **for** não precisa ser explicitamente declarada, tem escopo somente no corpo do laço, e encobre outra variável com mesmo nome que esteja declarada em escopo externo ao **for**. O tipo da variável de indução é inferido pelo compilador e pode iterar sobre um conjunto de inteiros, sobre os elementos de um tipo, ou sobre o tamanho de um vetor, por exemplo.

```

for i in faixa loop
    comando;
end loop;
```

O valor de *i* cobre todos os valores em faixa e portanto *i* pode iterar sobre um subconjunto dos inteiros, ou sobre elementos de um tipo, ou sobre o conjunto de índices de um vetor. O Programa 9.16, na página 299, contém um exemplo de laço que itera sobre os elementos do vetor de tuplas de teste do somador de 16 bits.

Vejamos um exemplo em que a faixa de valores de laços pode ser determinada de três formas diferentes: as duas últimas empregam atributos de vetores de bits para determinar a faixa de valores das variáveis de indução *j* e *k* – o tamanho do vetor pode ser alterado pelo programador e o compilador recomputará a nova faixa de valores para indexar o vetor. Os três laços determinam a paridade do vetor de bits D e são mostrados no Programa 9.41. O terceiro formato, com **D'range**, é usado nos *testbenches* para facilitar a indexação dos vetores de testes – a faixa de valores válidos do índice é computada pelo compilador ao determinar o tamanho do vetor D.

Programa 9.41: Exemplos de indexação de laço for.

```

1  signal D: bit_vector(0 to 9);
2
3  paridade3: process(D)
4      variable otmp: Boolean := FALSE;
5  begin
6      ...
7      for i in 0 to 9 loop — faixa definida explicitamente
8          if D(i) /= '1' then otmp := not otmp; end if;
9      end loop;
10     ...
11     for j in 0 to (D'length - 1) loop — comprimento do vetor
12         if D(j) /= '1' then otmp := not otmp; end if;
13     end loop;
14     ...
15     for k in D'range loop — faixa de valores do índice
16         if D(k) /= '1' then otmp := not otmp; end if;
17     end loop;
18     ...
19 end process paridade3;

```

Comando Sequencial WHILE

O comando **while** é similar ao da linguagem C: enquanto a condição for verdadeira, os comandos no corpo do laço são executados. Este comando não é sintetizável e é usado para gerar sequências de valores para testes, ou para ler os caracteres de um arquivo de texto, por exemplo.

```

while condição loop
    comandos;
end loop;

```

Comando Sequencial LOOP

O comando sequencial **loop** é usado para efetuar tarefas repetitivas e tampouco é sintetizável. O laço pode ser terminado com uma cláusula **exit when**; se a condição avaliar como verdadeira, o laço termina.

```

nomeDoLoop: loop
    ...
    exit when condição;
    ...
end loop nomeDoLoop;

```

Este laço tem um *label* que pode ser usado para interromper a execução de laços aninhados; e o *label* do laço que se deseja interromper é usado no **exit**. No exemplo abaixo, quando a condição é verdadeira, a execução dos dois laços é interrompida.

```

Externo: loop
    ...
    Interno: loop
        ...
        if condição then
            exit Externo; — salta para fora dos dois laços
        end if;
        ...
    end loop Interno;
    ...
end loop Externo;

```

9.4.12 Modelagem de máquinas de estado

Empregaremos o estilo de modelagem de Máquinas de Estado (MEs) descrito em [Ash08]: são necessários dois processos, um para o registrador de estado, e um processo combinacional para as funções de próximo estado e de saída. Dependendo da complexidade da ME, pode ser conveniente separar a função de saída da função de próximo estado.

O Programa 9.42 é um protótipo para as MEs que empregaremos. Um tipo (*states*) é definido para os estados, com um elemento do tipo para identificar cada um dos estados. Dois sinais do tipo *states* são declarados para armazenar o estado atual e o próximo estado.

Programa 9.42: Modelo para MEs.

```

type states is (s0, s1, s2, s3); — novo tipo para estados
signal curr_st, next_st : states; — sinais do novo tipo

U_state_reg: process(reset, clk) — registrador de estado
begin — lógica sequencial
    if reset = '0' then
        curr_st <= s0; — estado inicial
    elsif rising_edge(clk) then
        curr_st <= next_st; — troca de estado na borda
    end if;
end process U_state_reg;

U_st_transitions: process(curr_st) — função de próx estado
begin — lógica combinacional
    case curr_st is
        when s0 => next_st <= s1;
        when s1 => next_st <= s2;
        when s2 => next_st <= s3;
        when s3 => next_st <= s0;
    end case;
end process U_state_transitions;

```

O processo *U_state_reg* mantém o estado atual, e nas bordas do relógio, faz com que o próximo estado torne-se o estado atual. Quando *reset=0*, a ME é colocada em seu estado inicial.

O processo *U_st_transitions* computa o próximo estado em função do estado atual.

A cada evento no sinal `curr_st`, disparado por uma borda no relógio no processo `U_state_reg`, o novo estado é computado em função do estado atual, e na próxima borda o estado atual progredirá para o próximo estado. Este processo modela lógica combinacional.

Exemplo 9.10 Máquina de Moore Vejamos um exemplo completo: uma *Máquina de Moore* que detecta sequências de dois ou mais **1s** em sua entrada. O diagrama de estados da ME é mostrado ao lado do código no Programa 9.43.

A linha 1 declara o tipo para os estados e a linha 2 declara os sinais para o estado atual e o próximo estado. O sinal `found` é a saída da ME. O processo `U_state_reg` mantém o estado atual e o atualiza nas bordas do relógio. O estado inicial é o estado A. O processo `U_st_trans` computa o próximo estado em função do estado atual e das entradas, e define a saída da ME estado a estado.

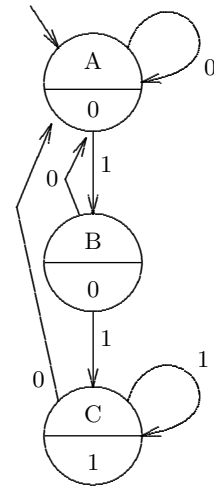
Verifique se o código do processo `U_st_trans` corresponde à função de próximo estado codificada no diagrama de estados. Verifique também se os dois processos desta ME satisfazem às condições para modelar processos sequenciais e combinacionais. ◀

Programa 9.43: Máquina de Moore para 11⁺.

```

1  type states is (A, B, C);
2  signal curr_st, next_st : states;
3  signal found : bit;
4
5  U_state_reg: process(reset, clk)
6  begin
7      if reset = '0' then
8          curr_st <= A;
9      elsif rising_edge(clk) then
10         curr_st <= next_st;
11     end if;
12 end process U_state_reg;
13
14 U_st_trans: process(curr_st, entr)
15 begin           — Máquina de Moore
16     case curr_st is
17         when A =>
18             if entr = '0' then
19                 next_st <= A;
20             else
21                 next_st <= B;
22             end if;
23             found <= '0';
24         when B =>
25             if entr = '0' then
26                 next_st <= A;
27             else
28                 next_st <= C;
29             end if;
30             found <= '0';
31         when C =>
32             if entr = '0' then
33                 next_st <= A;
34             else
35                 next_st <= C;
36             end if;
37             found <= '1';
38     end case;
39 end process U_st_trans;

```



Exemplo 9.11 Máquina de Mealy O Programa 9.44 mostra a implementação para uma *Máquina de Mealy* que detecta sequências de dois ou mais 1s em sua entrada.

Os modelos Moore e Mealy são semelhantes, exceto que na Máquina de Mealy são necessários apenas dois estados e a saída depende tanto do estado atual quanto da entrada: o sinal found é atualizado nas quatro possíveis combinações de estado e entrada.

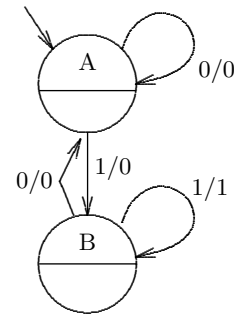
Verifique se o código do processo U_st_trans corresponde à função de próximo estado do diagrama de estados, e se os dois processos desta ME satisfazem às condições para modelar processos sequenciais e combinacionais. ◁

Programa 9.44: Máquina de Mealy para 11⁺.

```

1  type states is (A, B);
2  signal curr_st, next_st : states;
3  signal found : bit;
4
5  U_state_reg: process(reset, clk)
6  begin
7      if reset = '0' then
8          curr_st <= A;
9      elsif rising_edge(clk) then
10         curr_st <= next_st;
11     end if;
12 end process U_state_reg;
13
14 U_st_trans: process(curr_st, entr)
15 begin
16     -- Máquina de Mealy
17     case curr_st is
18         when A =>
19             if entr = '0' then
20                 next_st <= A;
21                 found <= '0';
22             else
23                 -- entr='1'
24                 next_st <= B;
25                 found <= '0';
26             end if;
27         when B =>
28             if entr = '0' then
29                 next_st <= A;
30                 found <= '0';
31             else
32                 -- entr='1'
33                 next_st <= B;
34                 found <= '1';
35             end if;
36     end case;
37 end process U_st_trans;

```



9.4.13 Funções

VHDL nos permite definir *funções* que computam valores, e que podem ser usadas no lado direito de expressões. Funções têm argumentos de tipo entrada (**in**) e um valor de retorno, e podem ser usadas, por exemplo, na conversão de tipos, tais como a conversão de inteiros para vetores de bits, e a conversão de vetores de bits para cadeias de caracteres.

Exemplo 9.12 Função para detecção de borda A função `rising_edge()`, no Programa 9.45, pode ser usada para testar a ocorrência de uma borda de subida num sinal. Seu argumento é um sinal do tipo `bit` e seu valor é um Booleano. Os registradores de estado nas MEs da Seção 9.4.12 empregam esta função para detectar bordas no sinal de relógio.

Se esta função estiver declarada na entidade de um *flip-flop*, ela pode ser empregada em todas as arquiteturas para aquela entidade. Se a função estiver declarada numa biblioteca, então ela pode ser empregada em todas as entidades que façam uso da biblioteca. ◁

Programa 9.45: Função que detecta bordas ascendentes.

```
function rising_edge(signal S: bit)
  return boolean is
begin
  if (S'event) and           — ocorreu evento em S
     (S = '1') and          — e valor atual é '1'
     (S'last_value = '0') — e valor anterior era '0'
  then
    return TRUE;
  else
    return FALSE;
  end if;
end rising_edge;
```

Espaço em branco proposital.

Exemplo 9.13 Conversões entre Boolean e Bit O Programa 9.46 mostra duas funções de conversão de tipos: (i) `bool2bit()` converte um valor do tipo `boolean` para o tipo `bit`; (ii) `bit2bool()` faz a conversão na direção oposta. ◁

Programa 9.46: Funções de conversão entre boolean e bit.

```
function bool2bit(B: in boolean) return bit is
  variable S : bit;
begin
  case B is
    when TRUE   => S := '1';
    when others => S := '0';
  end case;
  return S;
end bool2bit;

function bit2bool(S: in bit) return boolean is
  variable B : boolean;
begin
  case S is
    when '1'   => B := TRUE;
    when others => B := FALSE;
  end case;
  return B;
end bit2bool;
```

Lembre que VHDL é uma linguagem com tipos fortes e estas conversões, aparentemente inúteis, ao um preço módico a se pagar, comparando-se aos benefícios advindos dos tipos fortes. Dentre os benefícios está a detecção, durante a compilação, de erros de programação, tais como a combinação de tipos incompatíveis – soma de bit com inteiro – ou a aplicação de um operador incompatível – conjunção de dois inteiros.

Funções, podem ser definidas em pacotes e usadas da mesma forma que os novos tipos. O tipo da função é declarado no cabeçalho do **package** e o corpo da função é ser definido no corpo (**body**) do pacote. O Programa 9.47 contém a declaração e a definição de uma função que detecta bordas ascendentes em sinais.

Programa 9.47: Um pacote com abreviaturas e funções.

```
package p_WIRES is — abreviaturas para barramentos e sinais
```

```
    subtype int8   is integer range 0 to 255;
    subtype int16  is integer range 0 to 65535;
```

```
    subtype reg2   is bit_vector(1 downto 0);
    subtype reg4   is bit_vector(3 downto 0);
    subtype reg8   is bit_vector(7 downto 0);
    subtype reg15  is bit_vector(14 downto 0);
    subtype reg16  is bit_vector(15 downto 0);
```

```
    function rising_edge(S: in bit) return boolean;
    function bit2bool(S: in bit) return boolean;
    function bool2bit(B: in boolean) return bit;
```

```
package body p_WIRES is
```

```
    function rising_edge(signal S: bit) return boolean is
    begin
        if (S'event) and           — ocorreu evento em S
            (S = '1') and         — e valor atual é '1'
            (S'last_value = '0') — e valor anterior era '0'
        then
            return TRUE;
        else
            return FALSE;
        end if;
    end rising_edge;
```

```
    function bool2bit(B: in boolean) return bit is
        variable s : bit;
    begin
        ...
    end bool2bit;
```

```
    function bit2bool(S: in bit) return boolean is
        variable b : boolean;
    begin
        ...
    end bit2bool;
```

```
end package p_WIRES;
```

Índice Remissivo

Símbolos

T_A , 64
 T_D , 120
 T_I , 109, 116
 T_M , 117, 119, 186
 T_P , 116
 T_h , 188, 192, 199, 204
 T_p , 171
 $T_{C,F}$, 192
 $T_{C,x}$, 118–120, 199, 204, 206
 T_{FF} , 192
 T_{min} , 199, 206
 T_{skew} , 203
 T_{su} , 188, 192, 199, 204
 $\%$, 18–20
 $\Rightarrow$, 37
 $\bigvee$, 47
 $\bigwedge$, 47
 $\equiv$, 36
 $\gg$, 155
 $\wedge$, 30, 36
 $\triangleleft \triangleright$, 36, 42, 66
 $\leftarrow$, 193
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 41, 44
 $\ll$, 155
 $\neg$, 30, 36, 154
 $\vee$, 30, 36
 $\oplus$, 36, 53, 65
 $\mapsto$, 42
 $\mathbb{N}$, 146
 $\bar{a}$, *veja* $\neg$
 π , 25
 $\backslash$, 77
decod, 72
demux, 75
 e , 24
mod, 158, 218
mux, 67
 $\mathbb{B}$, 29–33, 38, 42
 $\mathbb{Z}$, 42, 149
 $[r]$, 241, 242
 $\mathbb{N}$, 42, 43, 149
num, 44, 55, 72, 77, 261, 262, 264, 274
 num^{-1} , 55
 $\mathbb{R}$, 42
 $|\mathbb{N}|$, 227
 $\langle \rangle$, 29, 42, 43, 180, 201, 225

Números

74148, 75
 74163, 218, 272
 74374, 232

A

abstração, 27, 280, 283, 292, 293, 301
 bits como sinais, 27–33, 57, 182, 184
 tempo discretizado, 116, 118, 182, 186–188,
 191–193, 197, 222
 acumulação de produtos parciais, 173–178
 adição, 152
 adiantamento de vai-um, 164–172, 312–315
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo, 233
 conversão de base, 18
 conversão de base de frações, 22
 amplificador, 125, 136
 diferencial, 138
 amplitude, 210
 and, *veja* $\wedge$
and array, 128
 apontador de pilha, 268
 aproximação, 24
 árvore, 64, 118
assembly, *veja* ling. de montagem
 associatividade, 31
 atraso, *veja* tempo de propagação, 57
 de transporte, 305
 inercial, 304
 atribuição, 12
 autômatos finitos, 236
 avaliação de desempenho, 316

B

barramento, 140
barrel shifter, 158
 básculo, 138, 185–189
 relógio multi-fase, 226
 SR, 185, 194
 binário, 20
 bit, 20
 de sinal, 147
 bits, 27–37, 65
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 tipo VHDL, 281

variável, 30
 bloco de registradores, 264
 borda, *veja* relógio
branch, *veja* desvio condicional
buffer, 123
buffer three-state, 140
 buraco, 87
 busca binária, 274
 byte, 11

C

C,
 deslocamento, 160
 overflow, 163
 cadeia,
 de portas, 64, 118
 de somadores, 152
 caminho crítico, 117
 capacitor, 105, 107, 135, 136
capture FF, *veja flip flop*, destino
 CAS, 134
 célula de RAM, 81
 célula, 100
 chave,
 analógica, 142
 digital, 92
 normalmente aberta, 92
 normalmente fechada, 78, 92
chip select, 268
 ciclo,
 combinacional, 57
 violação, 81, 184, 186, 188
 de trabalho, 225
 circuito,
 combinacional, 57, 65
 de controle, 257
 de dados, 257
 dual, 99
 segmentado, 202
 sequencial síncrono, 196, 209
 circuito aberto, 57
clear, 194
 clk, *veja* relógio
clock, *veja* relógio, 319
clock skew, *veja skew*
clock-to-output, 192
clock-to-Q, 192
 CMOS, 59, 65, 85–142
 buffer three-state, 140
 célula, 100
 inversor, 96
 nand, 99
 nor, 98
 porta de transmissão, 141
 portas inversoras, 99
 sinal restaurado, 125
 código,
 Gray, 63, 217, 227, 236
Column Address Strobe, *veja* CAS

combinacional,
 ciclo, 57
 circuito, 57
 dispositivo, 57
 comparador,
 de igualdade, 62, 164
 de magnitude, 164, 261, 274
Complementary Metal-Oxide Semiconductor, *veja* CMOS
 complemento, *veja* $\neg$
 complemento de dois, 145–151, 154–164
 complemento, propriedade, 31
 comportamento transitório, *veja* transitório
 comutatividade, 31
 condicional, *veja* $\triangleleft \triangleright$
 condutor, 85
 conjunção, *veja* $\wedge$
 conjunto mínimo de operadores, 65
 contador, 211–223
 74163, 218
 assíncrono, 220
 em anel, 224, 225, 232
 inc-dec, 224, 272
 Johnson, 227
 módulo-16, 215, 218
 módulo-2, 212, 223
 módulo-32, 223
 módulo-4, 212
 módulo-4096, 219
 módulo-8, 213, 221
 ripple, 220, 222
 síncrono, 222
 contra-positiva, 41
 controlador, 242
 de memória, 136
 conversão de base, 18
 conversor,
 paralelo-série, 229
 série-paralelo, 228
 corrente, 85, 105, 106, 112, 113
 de fuga, 115
 corrida, 123, 125
 CSS, 196–197, 209
 curto-circuito, 57

D

datapath, *veja* circuito de dados
 decimal, 17
 decodificador, 72, 78, 81–82, 84, 126
 de linha, 126, 135
 de prioridades, 74
delay, 57
 DeMorgan, 276
 demultiplexador, 75, 120
design unit, *veja* VHDL, unidade de projeto, 278
 deslocador exponencial, 156
 deslocamento, 155–158
 aritmético, 155, 158, 164
 exponencial, 159, 201

lógico, 155, 162
 rotação, 158
 detecção de bordas, 123
 diagrama de estado, 234
 restrições, 236
 disjunção, *veja* $\vee$
 dispositivo, 85
 combinacional, 57
 distributividade, 31, 34, 51
 divisão de frequência, 213, 216
 divisão inteira, 44, 273
 doador, 87
don't care, 70
 dopagem por difusão, 86
 dopante, 86
 DRAM, 134–137
 controlador, 136
 fase de restauração, 137
 linha,
 de palavra, 135
 linha de bit, 135
 linha de palavra, 136
 página, 135
 refresh, 135
 dual, 32, 95, 99
 dualidade, 32
duty cycle, 225

E

EEPROM, 133
 endereço, 77
 energia, 100, 105, 112–115
 enviesado, relógio, *veja skew*
 EPROM, 133
 equação característica do FF, 193
 equivalência, *veja* $\Leftrightarrow$
 erro,
 de representação, 23
 escopo, 306
 especificação, 42, 280, 293
 estado, 182
 estado atual, 196, 209, 212
 exponenciação, 264
 expressões, 36

F

fan-in, 109–112, 116, 167, 170
fan-out, 82, 84, 109–112, 116, 120, 170–172
 fatorial, 273
 fechamento, 31
 FET, 91
Field Effect Transistor, *veja* FET
Field Programmable Gate Array, *veja* FPGA
 FIFO, 271
 fila, 270–273
 circular, 271
 filtro digital de ruído, 190
flip-flop, 188–195
 adjacentes, 198

comportamento, 193
 destino, 198
 fonte, 197, 198
 mestre-escravo, 189
 modelo VHDL, *veja* VHDL, *flip-flop*
 temporização, 192
 tipo JK, 193
 tipo T, 191, 193
 um por estado, *veja* um FF por estado
 folga de *hold*, 198
 folga de *setup*, 198
 forma canônica, 48
 FPGA, 194, 300
 frações, *veja* ponto fixo
 frequência, 210
 frequência máxima, *veja* relógio
 função, 30
 tipo, 29, 42
 função de próximo estado, 233, 240, 334
 função de saída, 233, 240
 função, aplicação bit a bit, 32
 função, tipo (op. infix), *veja* $\mapsto$

G

ganho de desempenho, 316
 gerador de relógio, 192
 ghdl, 275
glitch, *veja* transitório
 GND, 93
 gramática, 17
 gtkwave, 275, 300, 311

H

handshake, 257
 hexadecimal, 19
hold time, 188, 197–202, 246
 folga, 198, 204, 205

I

idempotência, 31
 identidade, 31
 igualdade, 30
 implementação, 42
 implicação, *veja* $\Rightarrow$
 informação, 16
 inicialização,
 flip-flop, 194
 Instrução,
 busca, *veja* busca
 instrução, 12
 busca, *veja* busca
 decodificação, *veja* decodificação
 execução, *veja* execução
 resultado, *veja* resultado
 interface,
 de rede, 13
 de vídeo, 12
 inversor, 96
 tempo de propagação, 109
 involução, 31, 61

isolante, 85

J

Joule, 112

jump, veja salto incondicional

L

label, 333

latch, veja báculo

latch FF, veja *flip flop*, destino

launch FF, veja *flip flop*, fonte

Lei de Joule, 112

Lei de Kirchhoff, 106

Lei de Ohm, 105, 106

LIFO, 268

ligação,

 barramento, 140

 em paralelo, 93, 99

 em série, 93, 99

linguagem,

assembly, veja ling. de montagem

 C, veja C

 declarativa, 275, 285

 imperativa, 285

 Pascal, veja Pascal

 Verilog, 276

 VHDL, veja VHDL

 Z, 27

linha de endereçamento, 78

literal, 38

logaritmo, 43

lógica restauradora, 125

look-up table, 300

LUT, 300

M

MADD, 201

Mapa de Karnaugh, 49, 124

máquina de estados, 233, 236

 Mealy, 238, 245, 256, 273, 337

 modelo VHDL, 334

 Moore, 237, 244, 256, 269, 335

 projeto, 240

Máquina de Mealy, veja máq. de estados

Máquina de Moore, veja máq. de estados

máscara, 32

máximo e mínimo, 31

maxtermo, 46

Mealy, veja máq. de estados

memória, 181

 atualização, 77

 bit, 184

 de programação de FPGA, 300

 de vídeo, 13

 decodificador de linha, 80

 endereço, 77

 FLASH, 133

 matriz, 129, 132, 134

 multiplexador de coluna, 80

 primária, 13

 RAM, 81, 134, 267

 ROM, 78, 126, 245

 secundária, 13

memória dinâmica, veja DRAM

memória estática, veja SRAM

metaestabilidade, 184, 188, 191, 194

 defesa contra artes das trevas, 191

microcontrolador, 245–253

microrrotina, 253

mintermo, 45, 124, 126

modelagem, 280

modelo, 280

dataflow, 294

 estrutural, 284, 294, 295

 funcional, 44, 293, 295

 porta lógica, 96

 RTL, 294

 temporal, 304

 temporização, 115

módulo de contagem, 221

módulo, veja %, *mod*

Moore, veja máq. de estados

MOSFET, 91

multiplexador, 61, 66–70, 80, 101, 117, 119, 123–

 124, 126, 141, 142, 278–279, 284–286,

 309–311

 de coluna, 132, 136

multiplicação, 172–178, 262

 acumulação de produtos parciais, 173–178

multiplicador,

 somas repetidas, 262, 273

multiply-add, veja MADD

N

número,

 de Euler, 24

negação, veja $\neg$

net list, 284, 294

níveis de abstração, veja abstração

nível lógico,

 0 e 1, 28

 indeterminado, 28, 109, 116, 141

 terceiro estado, 140

nó, 96

non sequitur, 37

not, veja $\neg$

número primo, 53

O

octal, 18

onda quadrada, 188, 210, 319

operação,

 binária, 29

 bit a bit, 32

 infixada, 42

 MADD, 201

 prefixada, 47, 279

 unária, 29

operação apropriada, 197

operações sobre bits, 29–33
 operador,
 binário, 29
 lógico, 36
 unário, 29
operation code, veja *opcode*
 or, veja $\vee$
or array, 129
 ou exclusivo, veja $\oplus$
 ou inclusivo, veja $\vee$
output enable, 268
overflow, 148–150, 154, 162–164, 173, 178

P

paridade, 332
 ímpar, 49
 par, 49
 período mínimo, veja relógio
 pilha, 268–270, 273
pipelining, veja segmentação, 202
 piso, veja $\lfloor v \rfloor$
 ponto fixo, 151–152
 ponto flutuante, 70
pop, 268
 porta,
 de escrita, 265
 de leitura, 265
 porta lógica, 65
 and, 59
 carga, veja *fan-out*
 de transmissão, 141, 185
 modelo VHDL, 295
 nand, 60, 99
 nor, 60, 98
 not, 59, 96
 or, 59
 xor, 60, 65, 191
 portas complexas, 100
 potência, 112–115
 dinâmica, 114
 estática, 115
 potenciação, 43
 precedência, 30
 precisão,
 representação, 23
preset, 194
 prioridade,
 decodificador, 74
 processador, 12
 produtivo, 33
 produto de somas, 46
 programa de testes, veja VHDL, *testbench*
 PROM, 133
 propriedades, operações em $\mathbb{B}$, 31
 protocolo,
 de sincronização, 257, 273
 prova de equivalência, 40–41
 próximo estado, 196, 212
pull-down, 96

pull-up, 96, 126, 141
 pulso, 122, 123, 185, 194, 211, 219, 235, 261
 espúrio, veja transitório
push, 268

R

raiz quadrada, 274
 RAM, 12, 77, 81, 134–139
 célula, 81
 dinâmica, 135
Random Access Memory, veja RAM
 RAS, 134
Read Only Memory, veja ROM
 realimentação, 81
 receptor, 87
 rede, 96
 redução, 33
refresh, 137, 139
Register Transfer Language, veja RTL
 registrador, 195, 232, 250, 264
 carga paralela, 195
 de segmento, 201
 simples, 195
 registrador de deslocamento, 228–232
 modelo VHDL, veja VHDL, registrador, 329
 paralelo-série, 229
 série-paralelo, 228
 universal, 231
 registrador de estado, 196, 334
 relógio, 186, 188, 192, 210–225, 319
 bordas, 321
 ascendente, 189
 descendente, 189
 ciclo de trabalho, 225
 enviesado, veja *skew*
 frequência máxima, 199
 multi-fase, 225
 período mínimo, 199
 representação,
 abstrata, 28
 binária, 20
 complemento de dois, 147
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 ponto fixo, 151
 posicional, 17
 precisão, 23
reset, 185, 194, 195, 320, 325
 resistência, 87, 100, 105
 ROM, 12, 77–80, 126–133
 rotação, 158, 164
Row Address Strobe, veja RAS
 RTL, 202, 277, 294
rvalue, 326

S

segmentação, 201

seletor, 72
 semântica, 17
 semiconductor, 85
 tipo N, 87
 tipo P, 87
set, 185, 194, 325
setup time, 188, 197–202, 246
 folga, 198, 204, 205
 silogismo, 37
 simplificação de expressões, 38–40
 simulação,
 eventos discretos, 287
 sinal, 27, 42, 282, 326
 analógico, 27
 digital, 27, 28
 fraco, 125, 126, 138, 141
 intensidade, 90, 139
 restaurado, 125, 139
 síntese, *veja* VHDL, síntese, 300
skew, 202–207
Solid State Disk, *veja* SSD
 soma, *veja* somador
 soma de produtos, 45, 51, 126
 completa, 45
 somador, 145, 152–153, 293–300
 adiantamento de vai-um, 165, 232, 311–316
 cadeia, 152
 completo, 104, 152, 295
 modelo VHDL, 295, 312, 314
 overflow, 163
 parcial, 103
 seleção de vai-um, 170–172
 serial, 231
 temporização, 201, 311–316
 teste, 178, 298
 somatório, 33
 spice, 28
 SRAM, 138–139
 SSD, 14
status, 162
 subtração, 154
 superfície equipotencial, 96, 110

T

tabela,
 de excitação dos FFs, 193
 tabela verdade, 33–35, 45
 tamanho, *veja* |N|
 tempo,
 de contaminação, 115, 118–121, 184, 192, 306
 de estabilização, 188, 192
 de manutenção, 188, 192
 de propagação, 57–58, 61, 64–65, 73, 77, 84, 102,
 104, 109, 115–117, 192, 207, 222, 223, 304
 discretizado, 186, 188, 192, 197, 222
 temporização, 104–126, 304–316
 somador, 311–316
 tensão, 105
 Teorema,

Absorção, 49
 DeMorgan, 32, 41, 48, 54, 60, 61, 94, 98
 Dualidade, 99
 Simplificação, 49
 terceiro estado, 140–141
testbench, *veja* VHDL, *testbench*
 teste,
 cobertura, 179
 de corretude, 178
 teto, *veja* [r]
three-state, *veja* terceiro estado
 tipo,
 de sinal, 42
 função, 29
 Tipo I, *veja* formato
 Tipo J, *veja* formato
 Tipo R, *veja* formato
 transferência entre registradores, *veja* RTL
 transistor, 87–91, 95–96
 CMOS, 95
 corte, 113
 gate, 88
 saturação, 113
 sinal fraco, 90
 tipo N, 88
 tipo P, 89
Transistor-Transistor Logic, *veja* TTL
 transitório, 121–123, 186, 239, 311
 transmissão,
 serial, 230
transmission gate, *veja* porta de transmissão
 TTL,
 74148, 75
 74163, 218
 74374, 232
 tupla, *veja* ⟨ ⟩
 elemento, 32
 largura, 43

U

ULA, 160–164, 180, 264
 status, 162
 um FF por estado, 253–256
 Unidade de Lógica e Aritmética, *veja* ULA
 unidade de projeto, 283

V

valor da função, 30
 VCC, 93
 Verilog, 276
 vetor,
 de testes, 289
 vetor de bits, *veja* ⟨ ⟩, 32
 largura, 43
 VHDL, 194, 275–340
 &, 281
 (i), 282
 :=, 282
 <=, 281

- =>, 286
- active, 327
- after, 304, 311
- all, 284
- architecture, 278
- área concorrente, 285
- arquitetura, 278
- array, 290
- assert, 292
- associação,
 - nominal, 286
 - posicional, 285
- atraso,
 - de transporte, 305
 - inercial, 304
- atributo, 321, 327
- bit, 278, 282, 294
- bit_vector, 278
- boolean, 282
- borda ascendente, 321, 338
- case, 331
- concatenação, 282
- delta, 287–288, 318
- design unit*, veja VHDL, unidade de projeto
- entidade, 278
- entity, 278
- event, 321, 327
- evento, 287
- exit, 333
- flip-flop*, 321–326
- for, 292, 332
- função, 338–340
- generate, 311
- generic, 308
- in, 278, 333
- inertial, 304
- inicialização de sinal, 282
- instanciação, 285
- interface genérica, 308
- label*, 333
- last_value, 327
- length, 327
- linguagem, 275
- loop, 332, 333
- mapeamento de portas, 285
- máquina de estados, 334
- modelo executável, 280
- open, 309
- others, 331
- out, 278
- package, 283
- port map, 279, 285, 286
- processo,
 - combinacional, 328
 - sequencial, 329
- processo, 287
- range, 327, 333
- record, 289
- registrador, 329

- reject, 305, 311
- report, 292
- rising_edge, 338, 339
- rvalue, 326
- síntese, 280
- seleção de elemento de vetor, 282
- severity, 292
- simulação, 287
- sinal, 282
- síntese, 207, 300, 304
- std_logic*, 140
- subtype, 283
- testbench*, 288–292, 298–300
- time, 282
- tipos, 42, 281, 286
- to, 333
- transação, 287
- transport, 305
- type, 282, 334
- unidade de projeto, 283
- use, 284
- variável, 326–330
- wait, 319, 320, 322
- wait for, 320
- wait on, 320
- wait until, 320
- when, 331
- while, 333
- work, 284

W

- Watt, 112
- write back*, veja resultado

X

- xor, veja $\oplus$

Z

- Z, linguagem, 27

Capítulo 10

O Processador Mico XII

*Se algo tem rabo de jacaré, couro de jacaré, boca de jacaré,
pé de jacaré, olho de jacaré, corpo de jacaré e cabeça de
jacaré, como é que não é jacaré?*

Leonel Brizola

O autor teve seu primeiro contato com os livros de Hennessy & Patterson em 1990, com [HP90], e vem lecionando com base nas várias edições do livro introdutório desde 1994 [PH94]. Evidentemente, o conteúdo deste capítulo, bem como o projeto do Mico XII, foram muito influenciados por aqueles textos e de suas edições posteriores. O autor é um adepto do método de estudar Arquitetura de Computadores, proposto por Hennessy & Patterson, segundo o qual a linguagem *assembly* é a especificação do processador, e isso se reflete na forma e no conteúdo deste capítulo. Referências explícitas e implícitas àqueles textos são inevitáveis após um contato, quase que diário, de três décadas.

O Mico XII é um processador extremamente simples porém completo. O conjunto de instruções do Mico permite executar programas que contenham funções e que façam acessos a uma memória de dados. Todos os processadores comerciais conhecidos pelo autor (8085, z80, Atmel, 8086, 68000, ARM, MIPS, PowerPC, SPARC) possuem instruções como as do Mico, além de muitas outras que acrescentam pouco quando o que nos interessa é compreender a funcionalidade de um processador. O processador chama-se Mico porque este é uma imitação que tenta macaquear o processador MIPS32. A versão apresentada aqui é o décimo segundo membro de uma linhagem iniciada em 1996.

Iniciamos o capítulo¹ com a programação em linguagem de máquina do Mico, e então prosseguimos com a descrição dos componentes do processador, e de como os componentes são interligados para implementar as instruções. O capítulo encerra com uma discussão sobre a metodologia de sincronização do processador.

10.1 Conjunto de Instruções do Mico XII

A Figura 10.1 mostra a ligação do processador às memórias de instruções e de dados. O programa por executar é armazenado na memória de instruções, que por ora consideraremos como uma memória ROM. Os dados do programa são armazenados (escritos) e recuperados

¹© Roberto André Hexsel, 2012-2021. Versão de 2 de fevereiro de 2021.

(lidos) de uma memória RAM. A capacidade de cada uma destas memórias é de 64K (2^{16}) palavras de 32 bits.

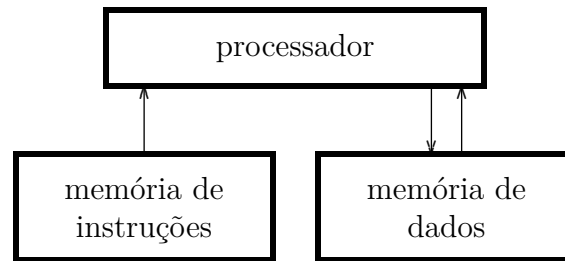


Figura 10.1: Processador e memórias de instruções e de dados.

As *instruções* são as operações indivisíveis efetuadas pelo processador, como uma soma ou uma comparação de magnitude. Um comando em linguagem de alto nível, tal como C ou Pascal, é traduzido pelo compilador para uma sequência de instruções que produz as alterações no estado da computação definidas naquele comando. Cada instrução do Mico é codificada em 32 bits, e em sua maioria, os operandos das instruções também são de 32 bits. As exceções são os endereços, que têm largura de 16 bits.

O processador, mostrado na Figura 10.2, contém quatro componentes principais: (1) um registrador que aponta a próxima instrução a ser executada e é chamado de *instruction pointer* (IP); (2) um circuito que gera os sinais de controle para os demais componentes (*cntrl*), em função da instrução que está sendo executada; (3) um *bloco de registradores* (*regs*) com 16 registradores de 32 bits, que permite observar o conteúdo de dois registradores e atualizar o conteúdo de um terceiro registrador; e (4) uma *unidade de lógica e aritmética* (ULA) na qual são efetuadas todas as operações de lógica e aritmética.

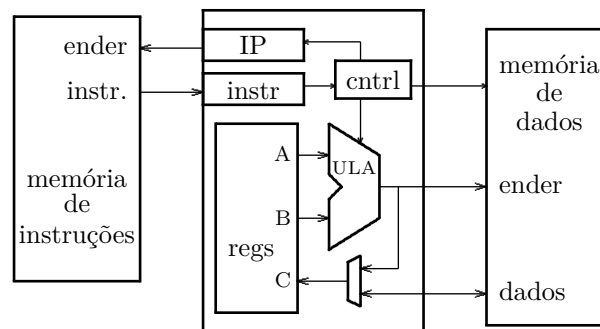
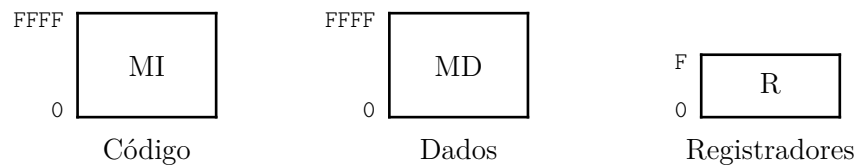


Figura 10.2: Modelo do processador.

A *arquitetura* do Mico é definida com três *espaços de endereçamento* distintos, indicados na Figura 10.3. O primeiro é o *espaço de código* ou *de instruções*, através do qual são acessadas as instruções do programa. O programador tem acesso a um conjunto de instruções que manipulam o registrador IP, e assim controlam a sequência de execução das instruções. Estas instruções são definidas na Seção 10.1.3. A faixa de endereços deste espaço é o intervalo $[0, 2^{16})$, e o registrador IP tem 16 bits de largura. As faixas de endereço são mostradas em hexadecimal na Figura 10.3.

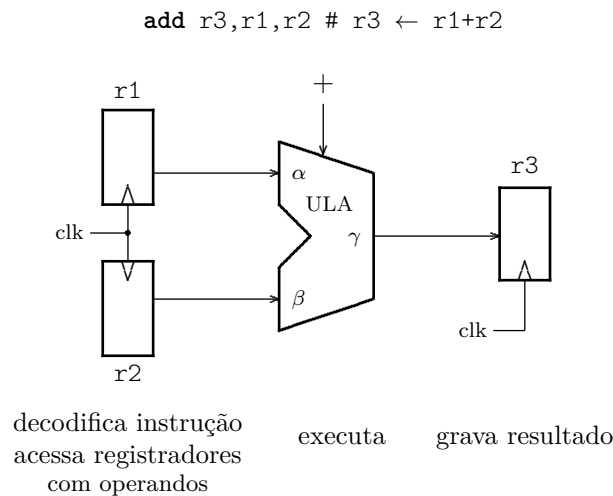
O segundo espaço é o *espaço de dados*, pelo qual são acessadas as variáveis e estruturas de dados dos programas. São duas as instruções para acessar dados, que estão definidas na Seção 10.1.2. A faixa de endereços também é o intervalo $[0, 2^{16})$, e os endereços têm 16 bits de largura.

**Figura 10.3: Espaços de endereçamento do Mico.**

O terceiro espaço de endereçamento, *espaço registradores*, é pequeno se comparado aos outros dois, sendo somente 16 os registradores, referenciados no intervalo $[0, 2^4)$. As instruções que referenciam os registradores são definidas na Seção 10.1.1. Estes 16 registradores são usados para a manipulação aritmética ou lógica das variáveis dos programas.

Notação O caractere ‘;’ significa execução sequencial; o caractere ‘,’ significa execução em paralelo dos eventos à esquerda e à direita da vírgula; ‘R(s)’ é a sintaxe de VHDL para indexar um vetor, seja um vetor de bits seja um vetor de bytes ou inteiros; ‘&’ é o operador de concatenação de VHDL, e ‘←’ representa atribuição.

A Figura 10.4 mostra o modelo de execução de uma instrução de soma. No topo da figura está a *instrução* de soma, chamada de **add**. Esta instrução tem dois operandos, que são obtidos de dois registradores, identificados por r1 e r2, e o resultado é depositado num terceiro registrador, chamado de *registrador de destino*, que é r3 neste exemplo. O caractere ‘#’ separa a instrução do comentário r3←r1+r2. O que estiver à direita do ‘#’ é ignorado pelo compilador.

**Figura 10.4: Execução de uma instrução de adição.**

O circuito de controle *decodifica* a instrução e gera os sinais de controle necessários àquela instrução. No caso de um **add**, a operação da ULA deve ser uma soma, e o registrador de destino deve ser atualizado.

Na base da Figura 10.4 estão indicadas três das fases da execução de um **add**: (i) a decodificação da instrução e acesso aos seus operandos, (ii) a execução na ULA, e (iii) a gravação do resultado no registrador de destino.

O diagrama não mostra a primeira fase, que é a *busca* da instrução na memória de instruções. A memória de instruções é indexada pelo registrador IP, e o conteúdo daquele endereço é a instrução que deve ser decodificada e executada. No caso de um **add**, enquanto a instrução grava o resultado, o IP é incrementado e a instrução seguinte é buscada, decodificada, executada e seu resultado armazenado.

Todas as instruções do Mico são codificadas em cinco campos, como mostra a Figura 10.5. O campo no quarteto mais significativo (opc) corresponde ao ‘nome’ da instrução, e é conhecido como *operation code*, ou *opcode*. Com 4 bits, pode-se enumerar até 16 instruções distintas.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	...	03	02	01	00
opc				a				b				c				const								

Figura 10.5: Formato das instruções do Mico.

Os próximos dois campos, a e b correspondem aos operandos das instruções de lógica e aritmética e contêm os números, ou os endereços, de dois registradores. O campo c indica o número/endereço do registrador que será atualizado com o resultado da operação. O campo nos 16 bits menos significativos (const) contém um endereço, ou uma constante codificada em complemento de dois. A maioria das instruções faz uso de alguns dos campos e ignora outros – a instrução **add** ignora o campo const enquanto que a instrução **j** (*jump*) ignora os campos a, b e c.

10.1.1 Instruções de Lógica e Aritmética

A ULA executa as operações de lógica e aritmética definidas para as instruções. O bloco de registradores contém 16 registradores de 32 bits, duas portas de leitura (A e B) e uma porta de escrita (C). Este bloco de registradores é similar ao da Seção 8.7.2.

As instruções de lógica e aritmética são **add**, **sub**, **mul**, **and**, **or**, **xor**, **slt** e **not**. Todas, exceto **not**, têm dois operandos provenientes de registradores, e depositam o resultado em um registrador. A instrução **not** tem um único operando. A Tabela 10.1 mostra as instruções de lógica e aritmética do Mico.

Tabela 10.1: Instruções de lógica e aritmética.

instrução	semântica	
op c, a, b	$R(c) \leftarrow R(a) \text{ op } R(b)$	$+, -, \times, \wedge, \vee, \oplus$
slt c, a, b	$R(c) \leftarrow (R(a) < R(b))$	<i>set on less than</i>
not c, a	$R(c) \leftarrow \text{not}(R(a))$	$\neg$
addi c, a, const	$R(c) \leftarrow R(a) + \text{extSin}(\text{const})$	<i>add immediate</i>

Na Tabela 10.1, a, b, c são nomes de registradores (endereços dos registradores, de 0 a 15) enquanto que A, B, C são os conteúdos dos respectivos registradores ($R(a)=A$). O registrador 0 (r_0) contém a constante zero ($R(0)=0$) e escritas neste registrador não tem nenhum efeito.

A instrução *set on less than*, **slt**, faz uma comparação de magnitude entre seus operandos, e armazena os valores TRUE=1 ou FALSE=0 no registrador de destino, dependendo do resultado da comparação.

A instrução *add immediate*, **addi**, adiciona o conteúdo de um registrador ($R(a)$) a uma constante, representada em complemento de dois, em 16 bits. A constante é estendida de 16 para 32 bits para que possa ser aplicada à entrada β da ULA – todos os operandos da ULA são de 32 bits. Esta instrução possibilita o uso de constantes positivas ou negativas. A instrução **addi** é frequentemente usada para inicializar uma variável ou registrador com uma constante. Por exemplo,

```
addi r5, r0, K    # r5 <- 0 + K
```

soma o valor de K com zero (r0) e o atribui ao registrador r5.

Vejamos três exemplos simples de tradução de código Pascal para *assembly*. Nestes exemplos usaremos a convenção de chamar os operandos das instruções com os nomes das variáveis, prefixados de 'r' para 'registrador'.

Pascal	<i>assembly</i>
a := b + c;	add ra, rb, rc

Há uma correspondência direta entre o comando (simples) em Pascal e sua tradução para *assembly*. Nosso segundo exemplo tira proveito da associatividade da soma. O resultado intermediário é acumulado no registrador ra.

Pascal	<i>assembly</i>
a := b + c + d + e;	add ra, rb, rc # ra <- rb + rc
	add ra, ra, rd # ra <- ra + rd
	add ra, ra, re # ra <- ra + re

No terceiro exemplo podemos tirar proveito da associatividade, ou então escrever código ligeiramente mais complexo para demonstrar o uso de dois registradores temporários (t0 e t1) para a computação de expressões que não sejam triviais. O código Pascal poderia ser transformado para evitar o uso dos dois registradores temporários.

Pascal	<i>assembly</i>
f := (g+h) - (i+j);	add t0, rg, rh # t0 <- rg + rh
	add t1, ri, rj # t1 <- ri + rj
	sub rf, t0, t1 # rf <- t0 - t1

Exemplo 10.1 Vejamos um exemplo completo de tradução de um programa em Pascal para o seu equivalente lógico em *assembly* do Mico XII. O programa em Pascal calcula a média aritmética de dois números que são lidos do teclado. Nosso processador não opera com frações, então computaremos a aproximação inteira da média.

O código em Pascal lê dois valores do teclado, computa sua média aritmética e escreve o resultado na tela. Espaço em memória é alocado para três variáveis de tipo inteiro (a, b e m). Os parênteses são necessários no cálculo da média porque a precedência da divisão é maior do que a da soma.

Pascal	assembly
<code>program med2(input,output)</code>	<code># segmento de dados (RAM)</code>
	<code># a em r4 = ra</code>
<code>var a, b, m : integer;</code>	<code># b em r5 = rb</code>
	<code># m em r6 = rm</code>
<code>begin</code>	
<code>read(a);</code>	<code># segmento de código (ROM)</code>
<code>read(b);</code>	<code>read ra</code>
<code>m := (a + b) / 2;</code>	<code>read rb</code>
<code>write(m);</code>	<code>add rm, ra, rb # rm <- a + b</code>
<code>end.</code>	<code>sra rm, rm, 1 # rm <- rm / 2</code>
	<code>show rm</code>
	<code>halt</code>

Este programa é tão pequeno e simples que não é necessário alocar as variáveis em memória; as variáveis são alocadas em registradores, como indica o comentário do *segmento de dados*.

No segmento de código empregamos duas instruções que não são definidas para o Mico XII: a instrução **read**, que lê um inteiro do teclado; e a instrução **sra** (*shift right arithmetic*) que desloca seu operando de *n* posições para a direita e preserva o sinal do operando.

A expressão $(a+b)/2$ é traduzida para o par **add** seguido da divisão por dois – deslocamento de uma posição para a direita. A média é exibida e o programa termina com uma instrução **halt**, que paralisa o processador. ◁

10.1.2 Instruções de Acesso à Memória

A memória de dados do Mico é um vetor de inteiros com 64K elementos, endereçados de 0 a 65.535. A memória é denotada² por $M()$ e se comporta como um circuito combinacional nas leituras ($x \leftarrow M(i)$) – a porta de leitura sempre mostra o conteúdo da posição apontada pelo endereço *i* – e como um registrador nas escritas ($M(j) \leftarrow y$) – a posição de memória apontada pelo endereço *j* é atualizada com o valor na porta de escrita na borda de subida do relógio, se a escrita estiver habilitada.

São duas as instruções de acesso à memória, uma que copia o conteúdo de um registrador para a memória (*store to memory*), e outra que copia o conteúdo de uma posição de memória para um registrador (*load from memory*). Nos dois casos, o endereço é computado pela soma do conteúdo de um registrador com uma constante. O registrador é a *base* do endereço, e a constante é um *deslocamento* com relação à base. A constante é representada em complemento de dois, em 16 bits. A Tabela 10.2 mostra as instruções de acesso à memória.

²A notação para vetores é aquela da linguagem de descrição de *hardware* VHDL. A indexação de vetores é denotada por parênteses ($V(i)$) ao invés de colchetes ($V[i]$).

Tabela 10.2: Instruções de acesso à memória.

instrução	semântica	
ld c, dsl(a)	$R(c) \leftarrow M(R(a) + \text{extSin}(\text{dsl}))$	<i>load from memory</i>
st dsl(a), b	$M(R(a) + \text{extSin}(\text{dsl})) \leftarrow R(b)$	<i>store to memory</i>

A instrução **ld** copia o conteúdo da posição indexada pela soma do registrador base com o deslocamento. O deslocamento é uma constante representada em complemento de dois, e pode ser positiva – sendo portanto o endereço maior do que a base, ou negativa – sendo o endereço menor do que a base.

A instrução **st**, copia o conteúdo de um registrador para a posição de memória indexada pela soma do registrador base com o deslocamento. O endereço é computado como num *load*.

Alocação de dados pelo compilador O resultado da compilação de um programa são três *segmentos*: (i) um com as instruções do programa, chamado de *segmento de texto*; (ii) um com os dados, chamado de *segmento de dados*; e (iii) um segmento com a pilha de chamadas de funções, o *segmento de pilha*.

Para gerar o segmento de texto, as instruções que resultam da compilação de todas as funções do programa são agrupadas, tipicamente na ordem em que as funções aparecem no código fonte. Uma vez que todas as funções tenham sido agrupadas, é possível determinar qual é o endereço da primeira instrução de cada função, e portanto o endereço do código pode ser associado ao nome simbólico da função.

Por exemplo, num programa com as funções *f()* com 1000 instruções, e *g()* com 2000 instruções, mais o programa principal com 5000 instruções, a alocação dos endereços das funções seria algo como o mostrado ao lado.

endereço	função
0000:	princ()
5000:	f()
6000:	g()
8000:	fim

O mesmo processo é aplicado na alocação das variáveis e estruturas de dados. O compilador agrupa todas as declarações de variáveis no segmento de dados, e então aloca os endereços das variáveis em função dos seus tamanhos.

Considere um programa com uma variável inteira *war*, e dois vetores de inteiros *vet[]* com 100 elementos e *get[]* com 200. A alocação dos endereços das variáveis seria algo como o mostrado ao lado.

endereço	variável
0000:	war
0001:	vet[]
0101:	get[]
0301:	fim

O segmento de pilha é vazio porque a pilha cresce dinamicamente em função da execução do programa. O conteúdo da pilha não pode ser alocado em tempo de compilação porque, em geral, não é possível prever o comportamento do programa por causa da incerteza quanto aos dados de entrada.

A Figura 10.6 mostra o cálculo do endereço para acessar o *k*-ésimo elemento de um vetor, quando o registrador base aponta para o endereço do *i*-ésimo elemento. O registrador base é *r5*, que aponta para o endereço do elemento *V[i]*. A soma do conteúdo do registrador base com o deslocamento (2) é o endereço efetivo, e o conteúdo daquela posição é copiado para o registrador de destino (*r4*). Elementos contíguos de vetores são alocados em posições contíguas da memória.

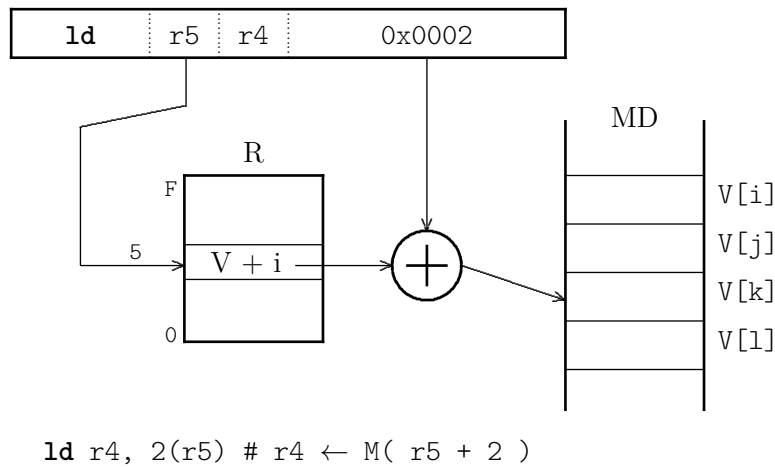


Figura 10.6: Cálculo do endereço para acessar o elemento $V[k]$.

O trecho de código abaixo mostra um exemplo com a instrução que efetua um acesso de leitura no k -ésimo elemento de um vetor de inteiros, indicada abaixo em Pascal e em *assembly*. O endereço de um vetor, na linguagem Pascal ou em *assembly*, é representado pelo nome do vetor, que é V no nosso exemplo. No código *assembly*, o endereço base do vetor (V) é adicionado ao índice i ; o k -ésimo elemento é acessado com um deslocamento de 2 com relação ao registrador base ($r5$). Para simplificar o código, os valores de V e de i são mantidos nos registradores apelidados de rV e ri .

Pascal

```
k      := i+2;
elem   := V[k];
```

assembly

```
add r5, rV, ri # r5 ← V+i
ld  r4, 2(r5)  # acessa V[i+2] = V[k]
```

No próximo exemplo, os dois operandos da soma são copiados para registradores pelos dois **lds**, e o resultado da adição é armazenado no quinto elemento do vetor com um **st**.

Pascal

```
V[4] := V[0]+V[1];
```

assembly

```
addi r15, r0, V # r15 ← 0 + base de V
ld   r1, 0(r15) # r1 ← M(V+0) = V[0]
ld   r2, 1(r15) # r2 ← M(V+1) = V[1]
add  r3, r1, r2
st   4(r15), r3 # M(V+4) ← V[0]+V[1]
```

A tradução para binário do comando $V[4] := V[0] + V[1]$; , assim como sua alocação em memória, são mostradas na Figura 10.7. A coluna da esquerda mostra os endereços na memória de programa e na memória de dados. Para simplificar a explanação, a primeira instrução é alocada no endereço 0x100 da memória de programa, e o endereço do primeiro elemento do vetor ($V[0]$) é alocado no endereço 0x300 da memória de dados. Neste exemplo, a memória de dados foi inicializada com valores aleatórios.

```
# memória de instruções
# end binário      assembly
...
0100: 800f.0300  addi r15, r0, V  # r15 <- 0 + base de V
0101: 9f01.0000  ld   r1, 0(r15)  # r1 <- M(V+0) = V[0]
0102: 9f02.0001  ld   r2, 1(r15)  # r2 <- M(V+1) = V[1]
0103: 0123.0000  add  r3, r1, r2
0104: af30.0004  st   4(r15), r3  # M(V+4) <- V[0]+V[1]
...
# memória de dados
...
0300: 0000.0020  V[0] = 0x20
0301: 0000.0033  V[1] = 0x33
0302: 0000.0504  V[2] = 0x504
0303: ffff.ffff  V[3] = -1
0304: 0000.0053  V[4] = 0x53
...
```

Figura 10.7: Tradução para binário de $V[4] := V[0] + V[1]$.

A segunda coluna mostra o código binário da instrução, e a terceira coluna mostra a instrução em *assembly* que lhe corresponde. A tradução de *assembly* para binário é deveras simples, embora tediosa. Com base na Tabela 10.3, o código binário correspondente a cada instrução é gerado e acrescentado à sequência de instruções. Para facilitar a leitura, as instruções na Figura 10.7 estão divididas por um “ponto de milhar” em dois campos de 16 bits.

Tabela 10.3: Codificação das instruções do Mico (parcial).

opc	instrução	semântica	descrição
0	add c, a, b	$R(c) \leftarrow R(a) + R(b)$	soma registradores
8	addi c, a, K	$R(c) \leftarrow R(a) + \text{extS}(K)$	soma com constante
9	ld c, D(a)	$R(c) \leftarrow M(R(a) + \text{extS}(D))$	load from memory
a	st D(a), b	$M(R(a) + \text{extS}(D)) \leftarrow R(b)$	store to memory
e	beq a, b, E	$IP \leftarrow ((R(a) == R(b)) ? E : IP+1)$	desvio condicional

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	...	03	02	01	00	
opc				a				b				c				const									

Na memória de programa, cada instrução ocupa um endereço, e as instruções são executadas na ordem em que aparecem na memória. O IP inicialmente aponta para 0x100, depois para 0x101, 0x102, e assim por diante. Veremos em breve o que ocorre na presença de decisões e laços.

O estado da memória de dados é mostrado após a execução do comando $V[4] := V[0] + V[1]$, depois que $V[4]$, armazenado em $M(0x304)$, é atualizado com a soma de $V[0]$ e $V[1]$, por sua vez armazenados em $M(0x300)$ e $M(0x301)$.

É mais fácil para humanos pensar em termos de comandos, vetores e variáveis, que são *nomes simbólicos*, enquanto que o processador trata somente de endereços e instruções simples. A tarefa do programador em *assembly*, e de quem escreve o compilador, é traduzir as construções abstratas de Pascal ou C para as sequências de instruções simples que o processador executa. Enquanto que código em linguagens de alto nível pode ser escrito em “duas dimensões”, com construções tais como **if then else**, o código em *assembly* é ‘unidimensional’ e o programador em *assembly* deve traduzir os comandos de alto nível para sequências lineares de instruções.

No comando $V[4] := V[0] + V[1]$, os índices do vetor são constantes, e o compilador computa os deslocamentos com relação à base do vetor pela mera inspeção do código fonte. Se o programador decide mudar estes índices, então o programa deve ser recompilado. Vejamos como fica o código de acesso ao vetor quando os índices variam em tempo de execução; ao invés de constantes, os índices são variáveis, cujos conteúdos se alteram ao longo da execução do programa.

Pascal

```
V[k] := V[i]+V[j];
```

assembly

```
addi r15, r0, 0300 # r15 <- base de V
add  r14, r15, ri  # V+i
ld   r1, 0(r14)    # r1 <- M(V+i)
add  r13, r15, rj  # V+j
ld   r2, 0(r13)    # r2 <- M(V+j)
add  r3, r1, r2
add  r12, r15, rk  # V+k
st   0(r12), r3    # M(V+k) <- r3
```

Para referenciar o i -ésimo elemento do vetor, o compilador deve gerar código para, explicitamente, efetuar a soma do *endereço base* do vetor com o deslocamento armazenado na variável i . As três primeiras linhas da versão em *assembly* o fazem: a base do vetor – endereço $0x0300$ – é atribuída a $r15$, e o deslocamento indicado na variável i é adicionado e atribuído a $r14$. O conteúdo de $r14$, que é o endereço resultante de *base+deslocamento*, é chamado de *endereço efetivo*, e este é usado na instrução **ld** para atribuir $V[i]$ a $r1$. O mesmo cálculo deve ser efetuado para computar os endereços de $V[j]$ e $V[k]$.

Exemplo 10.2 Vejamos um exemplo mais complexo de acesso a vetores. Neste caso, um elemento do vetor $Y[]$ é usado para indexar o vetor $X[]$. N e M são números razoáveis.

Pascal	assembly
$X[0..(N-1)] : \text{integer};$	<code>addi ry, r0, Y # ry <- base de Y</code>
$Y[0..(M-1)] : \text{integer};$	<code>addi r5, ry, 3 # r5 <- Y+3</code>
...	<code>ld r6, 0(r5) # r6 <- Y[3]</code>
$a := X[Y[3]];$	<code>addi rx, r0, X # rx <- base de X</code>
...	<code>add r7, rx, r6 # r7 <- X+Y[3]</code>
	<code>ld r8, 0(r7) # r8 <- X[Y[3]]</code>
	<code>addi r9, r0, a # r9 <- ender a</code>
	<code>st 0(r9), r8 # M[a] <- X[Y[3]]</code>

Inicialmente, o quarto elemento de $Y[]$ é carregado para o registrador $r6$ com o primeiro `ld`. Seu valor é adicionado à base de $X[]$ e o segundo `ld` carrega o conteúdo daquela posição para $r8$. Este valor é armazenado na variável a com o `st`. ◁

Os limites de um vetor podem ser ultrapassados caso o programador cometa um erro, e tal deve ser sinalizado durante a execução do programa Pascal. Suponha que se deseje atribuir valores ao vetor, sem nunca extrapolar seus limites. Isso pode ser facilmente obtido empregando o operador módulo ($\%$): $(x \% k) < k$ para qualquer x . Considere um vetor declarado como

$X[0..(N-1)] : \text{integer};$

quando N é uma potência de dois. Para garantirmos que o índice i nunca ultrapasse o limite superior, basta acessar o vetor com $X[i \text{ MOD } N]$ porque $i \% N \in [0, N)$.

O módulo é computado com uma divisão, que é uma operação deveras custosa. O módulo de uma potência de dois pode ser obtido de uma maneira mais econômica empregando-se uma conjunção. Se $N = 16$ então para qualquer i o resto da divisão por 16 é menor ou igual a 15:

$$i \% 16 \in \{0, 1, 2, \dots, 13, 14, 15\}.$$

A conjunção bit a bit de qualquer número com 15 (1111_2) é no máximo 15. Logo

$$(i \% 16) = (i \wedge 15)$$

e a extensão óbvia para qualquer potência de dois é

$$\text{se } N = 2^n \text{ então } (i \% N) = (i \wedge (N - 1)).$$

Exemplo 10.3 Vejamos um exemplo em que se garante que o índice não ultrapassa os limites em acessos ao vetor. N é uma potência de dois.

Pascal	assembly
$X[0..(N-1)] : \text{integer};$	<code>addi rx, r0, X # rx <- base de X</code>
...	<code>addi r4, r0, (N-1) # r4 <- N-1</code>
$a := X[i \text{ MOD } N];$	<code>and r5, ri, r4 # r5 <- i AND N-1</code>
...	<code>add r6, rx, r5 # r6 <- X+i</code>
	<code>ld r7, 0(r6) # r7 <- X[i MOD N]</code>
	<code>addi r9, r0, a # r9 <- ender a</code>
	<code>st 0(r9), r7 # M(a) <- X[i]</code>

A operação de módulo é efetuada com o `and` e assim o acesso com o `ld` fica garantidamente entre os limites máximo e mínimo do vetor. A constante N é definida pelo programador; o endereço inicial do vetor $X[]$ é definido pelo compilador ou pelo programador do *assembly*. ◁

10.1.3 Instruções de Controle de Fluxo de Execução

Até aqui vimos instruções que são executadas uma após a outra, na mesma ordem em que estão armazenadas em memória. As instruções que *alteram* o fluxo sequencial de execução nos permitem escrever programas que executam sequências distintas de comandos em função dos dados do programa. A Tabela 10.4 mostra as instruções de *controle de fluxo* do Mico.

Tabela 10.4: Instruções de controle de fluxo de execução.

instrução	semântica	
j ender	$IP \leftarrow \text{ender}$	<i>jump to address</i>
beq a, b, ender	if ($R(a)=R(b)$) $IP \leftarrow \text{ender}$ else $IP \leftarrow IP+1$	<i>branch if equal</i>
nop	$IP \leftarrow IP+1$	<i>no operation</i>
jal c, ender	$IP \leftarrow \text{ender}$, $R(c) \leftarrow IP+1$	<i>jump and link</i>
jr a	$IP \leftarrow R(a)$	<i>jump register</i>

A instrução *jump* (**j**) é o equivalente em *assembly* do **goto**³ e a sequência de execução é incondicionalmente desviada para o endereço indicado no operando. O endereço de destino do **j** deve ser o endereço de uma instrução, identificado por um nome simbólico, ou um *label*, que identifica o destino do salto.

A instrução *jump* é um apelido para a instrução **jal** r0, ender, de tal modo que o valor $IP+1$ é gravado no registrador r0, o que não altera seu conteúdo, e o efeito obtido é o de um salto para o endereço de destino. O nome para este tipo de apelido é *pseudoinstrução* porque o programador pode fazer uso de uma instrução que não existe mas que ainda assim tem uma função útil. Em geral, linguagens de montagem empregam uma série de pseudoinstruções porque elas facilitam a vida do programador. Por exemplo, a instrução **addi** r5, r0, K poderia ser apelidada de **copy** r5, K, porque este é o sentido desejado: *copiar a constante K para o registrador r5*. O compilador faz a tradução de pseudoinstruções para as respectivas instruções reais.

A instrução *branch-if-equal* (**beq**) faz uma comparação de igualdade, e se os operandos são iguais, então desvia a execução para o endereço indicado na instrução. Os conteúdos dos dois registradores ($R(a)$ e $R(b)$) são comparados; se iguais, a instrução apontada por ender é a próxima a ser executada; se diferentes, então a instrução apontada por $IP+1$ é executada.

A instrução *no-operation* (**nop**) não tem nenhum efeito, a não ser a passagem do tempo para executar uma instrução. Esta instrução parece inútil nas não é, como veremos adiante. As instruções **jal** e **jr** são descritas na Seção 10.1.5.

Espaço em branco proposital.

³A programação em *assembly* é o único ambiente em que **goto**'s podem ser utilizados sem causar asco no leitor do código.

Comando if Vejamos um exemplo que emprega as três instruções que vimos nesta seção. O trecho em Pascal é um comando **if** simples que compara duas variáveis e incrementa uma terceira, se a comparação de igualdade tem resultado verdadeiro. Como antes, os nomes das variáveis são tratados como nomes de registradores.

Pascal	assembly
if (a = b) then	if: beq ra, rb, then
c := c + 1;	j cont
cont:	then: addi rc, rc, 1 # c = c + 1
	cont: nop # código após o if(a=b)

A comparação do **beq** define qual é a instrução seguinte a ser executada. Se $R(a) \neq R(b)$, então a instrução seguinte ao **beq** é executada; se os conteúdos dos registradores são iguais, então a próxima instrução a ser executada é aquela apontada pelo campo *ender*. Como só temos comparação de igualdade, o código *assembly* é menos elegante do que poderia: em caso de igualdade, a próxima instrução, que é a primeira da cláusula **then**, não pode ser a seguinte ao **beq**. Neste exemplo, a instrução seguinte ao **beq** é um *jump*, que pula “por cima” do código do **then**. A instrução seguinte ao **beq** é um **j** que salta para a continuação do programa, para além do código que implementa o **then**.

Os nomes na margem esquerda do código *assembly* são chamados de *labels*, e servem para nomear o endereço de uma instrução. No exemplo acima o *label* then: identifica o endereço da instrução **addi**, enquanto que o *label* cont: identifica o endereço da instrução seguinte ao comando **if**, e neste caso, como não sabemos qual instrução seria, usamos um **nop** como um marcador de local. Adiante veremos um exemplo com a tradução de *assembly* para binário de código com saltos e desvios.

Comando if-then-else Um comando **if-then-else** garante a execução exclusiva de uma das duas cláusulas, ou a cláusula do **then**, ou a cláusula do **else**, mas nunca as duas. O que em Pascal ou C é ‘bidimensional’, em *assembly* deve ser escrito de forma ‘unidimensional’, ficando explícitos os saltos sobre a ‘outra’ cláusula, e isso é mostrado no trecho de código abaixo.

Pascal	assembly
if (a = b) then	beq ra, rb, then # a=b -> then
c := c AND d;	else: or rc, rc, rd # c <- c OR d
else	j cont # pula then
c := c OR d;	then: and rc, rc, rd # c <- c AND d
cont:	cont: nop

Aqui, podemos tirar melhor proveito da instrução **beq** porque a cláusula **else** deve ser executada, se a **then** não o for. Se $R(a) \neq R(b)$, então a disjunção (**else**) deve ser executada; do contrário, a conjunção (**then**) é executada. Se a cláusula **else** é executada, então o *jump* garante que a outra cláusula não o é.

Comando while Um laço é mostrado no trecho de código abaixo. A variável de redução (*s*) e a de controle do laço (*i*) são inicializadas. O laço **while** acumula em *s* a redução por soma dos dez primeiros elementos do vetor *V*.

Pascal	<i>assembly</i>
<i>s</i> := 0;	add <i>rs</i> , <i>r0</i> , <i>r0</i> # <i>rs</i> <- 0
<i>i</i> := 0;	add <i>ri</i> , <i>r0</i> , <i>r0</i> # <i>ri</i> <- 0
while (<i>i</i> <> 10) do	addi <i>rv</i> , <i>r0</i> , <i>V</i> # <i>rv</i> <- <i>V</i>
begin	addi <i>r10</i> , <i>r0</i> , 10 # <i>r10</i> <- 10
<i>s</i> := <i>s</i> + <i>V</i> [<i>i</i>];	while: beq <i>ri</i> , <i>r10</i> , <i>fim</i> # <i>i</i> =10 -> <i>fim</i>
<i>i</i> := <i>i</i> + 1;	add <i>r5</i> , <i>rv</i> , <i>ri</i> # <i>r5</i> <- <i>V</i> + <i>i</i>
end	ld <i>r6</i> , 0(<i>r5</i>) # <i>r6</i> <- <i>M</i> (<i>V</i> + <i>i</i>)
<i>fim</i> :	add <i>rs</i> , <i>rs</i> , <i>r6</i> # <i>s</i> <- <i>s</i> + <i>V</i> [<i>i</i>]
	addi <i>ri</i> , <i>ri</i> , 1 # <i>i</i> <- <i>i</i> +1
	j <i>while</i> # <i>repete</i>
	<i>fim:</i> nop

A tradução para *assembly* inicializa as variáveis *s* e *i*, assim como um apontador para a base do vetor. O registrador *r10* é inicializado com 10, que é 1 além do limite de *i*.

No corpo do laço, o endereço do *i*-ésimo elemento é computado em *r5*, e este endereço é usado para carregá-lo em *r6*, e então somá-lo à variável *s*. Quando *i*=10, o laço termina e a execução continua a partir do endereço *fim*.

A tradução do laço para binário é mostrada na Figura 10.8. A coluna da esquerda é o endereço da instrução, e neste exemplo o código inicia no endereço 0x0010, e o vetor *V* está alocado no endereço 0x0200 da memória de dados. A segunda coluna é a tradução para binário, e aqui os nomes das variáveis foram traduzidos para números de registradores: *s*, *i* e *v* foram alocados aos registradores *r1*, *r2*, e *r3*, respectivamente. Os *opcodes* das instruções estão listados na Tabela 10.3.

...	...
0010: 0001.0000	add <i>r1</i> , <i>r0</i> , <i>r0</i> # <i>rs</i> <- 0
0011: 0002.0000	add <i>r2</i> , <i>r0</i> , <i>r0</i> # <i>ri</i> <- 0
0012: 8003.0200	addi <i>r3</i> , <i>r0</i> , <i>V</i> # <i>rv</i> <- 0x0200
0013: 800a.000a	addi <i>r10</i> , <i>r0</i> , 10 # <i>r10</i> <- 10
0014: e2a0.001a	while: beq <i>r2</i> , <i>r10</i> , <i>fim</i> # <i>i</i> =10 -> <i>fim</i>
0015: 0325.0000	add <i>r5</i> , <i>r3</i> , <i>r2</i> # <i>r5</i> <- <i>v</i> + <i>i</i>
0016: 9506.0000	ld <i>r6</i> , 0(<i>r5</i>) # <i>r6</i> <- <i>M</i> (<i>V</i> + <i>i</i>)
0017: 0161.0000	add <i>r1</i> , <i>r1</i> , <i>r6</i> # <i>s</i> <- <i>s</i> + <i>V</i> [<i>i</i>]
0018: 8202.0001	addi <i>r2</i> , <i>r2</i> , 1 # <i>i</i> <- <i>i</i> +1
0019: c000.0014	j <i>while</i> # <i>repete</i>
001a: 0000.0000	<i>fim:</i> nop
...	...

Figura 10.8: Tradução para binário da redução do vetor.

Uma vez que todas as instruções tenham sido traduzidas e colocadas na ordem, os endereços dos *labels* *while:* e *fim:* podem ser computados. O endereço do *label* *while* é 0x0014 e o endereço que corresponde ao *label* *fim* é 0x001a. Este código traduzido pode ser gravado na memória de programa, e então executado.

Exemplo 10.4 Considere um programa que conta o número de elementos num vetor que são maiores do que zero. A contagem encerra no primeiro elemento negativo encontrado. Vejamos a sua tradução para *assembly*.

Pascal	<i>assembly</i>
V[0..(N-1)] : integer;	addi r1, r0, 0 # i <- 0
...	addi rc, r0, 0 # c <- 0
i := 0;	addi rn, r0, N # rn <- N
c := 0;	addi rv, r0, V # rv <- base de V
x := V[0];	ld rx, 0(rv) # rx <- V[0]
while ((x > 0) AND (i < N))	w: slt r1, r0, rx # r1 <- (0 < V[i])
begin	beq r1, r0, f # 0 >= V[i] -> fim
c := c + 1;	slt r1, ri, rn # r1 <- (i < N)
i := i + 1;	beq r1, r0, f # i >= N -> fim
x := V[i];	addi rc, rc, 1 # c <- c+1
end;	addi ri, ri, 1 # i <- i+1
	add r5, ri, rv # r5 <- V+i
	ld rx, 0(r5) # rx <- V[i]
	j w # repete
	f: nop

A condição contém dois testes; o primeiro compara V[i] com zero; caso V[i] não seja maior do que zero, o primeiro *beq* desvia a execução para o final do laço. O segundo teste compara i < N e se resultar em falso, o segundo *beq* desvia a execução para o final do laço. A conjunção das duas condições está implícita na sequência dos testes: o corpo do laço é executado somente se o fluxo de execução sobreviver aos dois testes. <

Exemplo 10.5 Vejamos um segundo exemplo completo de tradução de um programa em Pascal para o seu equivalente em *assembly* do Mico XII. O programa em Pascal calcula a aproximação inteira da média aritmética de 64 inteiros, que são lidos do teclado.

Para fins de exemplo, o código é ligeiramente mais complexo do que o necessário: no primeiro laço, o código em Pascal lê os valores do teclado e os armazena no vetor K[]; no segundo laço, o vetor é então percorrido e a média computada.

Espaço em memória é alocado para quatro variáveis escalares de tipo inteiro (c, v, s e m), e para um vetor de 64 inteiros. Repare que as variáveis são sempre inicializadas antes de que sejam lidas.

Pascal

```

program med64(input,output)

var c, v, s, m : integer;
var K : array(0..63)
      of integer;

begin
  c := 0;
  while (c < 64)
  begin
    read(v);
    K[c] := v;
    c := c + 1;
  end;
  c := 0;
  s := 0;
  while (c < 64)
  begin
    s := s + K[c];
    c := c + 1;
  end;
  m := s / 64;
  write(m);
end.

```

assembly

```

# segmento de dados (RAM)
# c em r4 = rc
# v em r5 = rv
# s em r6 = rs
# m em r7 = rm
# K em M(0) até M(63)
# endereço de K[] em re

# segmento de código (ROM)
addi r8, r0, 64 # limite
add rc, r0, r0 # c <- 0
add re, r0, r0 # K[0]=M(0)
w1: slt r9, rc, r8 # r9 <- (c<64)
    beq r9, r0, f1 # r9=F -> fim
    read rv
    st 0(re), rv # K[re] <- v
    addi rc, rc, 1 # c <- c+1
    addi re, re, 1 # e <- e+1
    j w1

f1: add rc, r0, r0
    add re, r0, r0
    add rs, r0, r0
w2: slt r9, rc, r8 # r9 <- (c<64)
    beq r9, r0, f2 # r9=F -> fim
    ld rv, 0(re) # v <- K[re]
    add rs, rs, rv # s <- s+v
    addi rc, rc, 1 # c <- c+1
    addi re, re, 1 # e <- e+1
    j w2

f2: sra rm, rs, 6 # rm <- rs/64
    show rm
    halt

```

As variáveis escalares são alocadas em registradores, como indica o comentário do *segmento de dados*. O vetor $K[]$ é alocado a partir do endereço zero da RAM, porque não há outras variáveis alocadas em RAM. Se $K[]$ fosse alocado em outra posição, o registrador re deveria ser inicializado com aquele endereço, ao invés de 0.

Como no Exemplo 10.1, aqui também empregamos as instruções **read** e **sra**, que não são definidas para o Mico XII. A instrução **sra** desloca seu operando de n posições para a direita e preserva o sinal do operando.

A média $(\sum_{i=0}^{63} K_i)/64$ é traduzida para o segundo laço, que percorre os elementos do vetor e acumula os resultados parciais da soma na variável s . Após as 64 iterações, o somatório é dividido por 64 com um deslocamento de seis posições para a direita. A média é exibida na tela e o programa termina com uma instrução **halt**. ◀

10.1.4 Outras Instruções

A instrução **show** mostra o conteúdo do registrador $R(a)$ e tem efeito similar ao um **writeln**. Processadores “de verdade” não suportam uma instrução tal como **show** porque a operação *escrever na tela* é relativamente complexa e que pode necessitar de centenas, ou milhares, de instruções. Para nossos fins, **show** é perfeitamente adequada.

Tabela 10.5: Outras instruções.

instrução	semântica	
show a	$\text{display} \leftarrow R(a)$	exibe valor na tela
halt		paralisa a execução

A instrução **halt** paralisa a execução, e o processador deve ser reinicializado – com a ativação do sinal **reset** – para voltar a operar. Esta instrução também é útil para interromper a simulação da execução de programas, por exemplo.

10.1.5 Suporte a Funções

Uma *função* é um trecho de código que computa um *valor de retorno* a partir de um ou mais *argumentos*. A função pode ser invocada em vários pontos do programa, em uma ordem arbitrária e que depende dos dados com que o programa executa a cada vez. Logo, o endereço para onde a função retorna só pode ser determinado em tempo de execução porque o compilador não tem como prever a sequência de invocações.

A instrução que *salta para a função* deve fazer duas coisas: (i) saltar para a primeira instrução do código da função; e (ii) registrar o endereço para onde deve retornar, depois de executar todo o código da função. A instrução que *retorna da função* simplesmente salta para o endereço de retorno. A Tabela 10.6 mostra as instruções de *suporte a funções* do Mico.

A instrução *jump and link* (**jal**) tem dois efeitos: (i) salta para o endereço indicado no operando; e (ii) armazena o endereço de retorno em $R(c)$, que é o que faz a ligação do endereço de retorno – este é o *link* para o *return address*.

A instrução *jump register* (**jr**) salta para o endereço de ligação/retorno, que foi armazenado por uma instrução **jal**.

Tabela 10.6: Instruções para suporte a funções.

instrução	semântica	
jal c, ender	$IP \leftarrow \text{ender}, R(c) \leftarrow IP+1$	<i>jump and link</i>
jr a	$IP \leftarrow R(a)$	<i>jump register</i>

Considere o trecho de programa abaixo, com uma invocação da função `f()` a partir do programa principal. Por convenção, o endereço de retorno deve ser armazenado no registrador `r15`, que é apelidado de *return address* ou *ra*. No retorno da função, a instrução `jr` usa o *ra* para retornar ao ponto de invocação.

Pascal	assembly
...	...
<code>z := fun(x);</code>	<code>jal ra, fun # IP <- fun, ra <- ret</code>
...	<code>ret: add r5, r0, r1 # retorna aqui</code>
	...
	...
	...
	<code>fun: add r4, r2, r3 # fun()</code>
	...
	<code>jr ra # IP <- ret</code>
	...

Protocolo de invocação de função

Além do mecanismo de saltar para funções, e retornar delas, o código das funções que invocam e o código das funções que são invocadas devem concordar em mais três coisas: (i) a forma de comunicação dos argumentos; (ii) a forma de comunicação do valor de retorno; e (iii) o mecanismo para controlar o aninhamento de funções. Vejamos um protocolo simplificado para resolver os dois primeiros pontos.

Um *protocolo de invocação* extremamente simples é o seguinte:

- (1) dois argumentos são transferidos através dos registradores `r2` e `r3`, que são apelidados de `a0` e `a1` ('a' de argumento);
- (2) o valor da função é retornado no registrador `r1`, apelidado de `v0` ('v' de valor); e
- (3) o endereço de retorno é mantido no registrador `r15`, apelidado de *ra* (*return address*).

Considere o trecho de programa abaixo, com uma invocação da função `f()` de dois argumentos. Os argumentos são preenchidos nas duas primeiras linhas, e o valor da função é usado na atribuição à variável `z` na quarta linha.

Pascal	assembly
...	...
<code>z := fun(x, y);</code>	<code>add a0, r0, rx # a0 <- x</code>
...	<code>add a1, r0, ry # a1 <- y</code>
	<code>jal ra, fun # IP <- fun, ra <- ret</code>
	<code>ret: add rz, r0, v0 # z <- f(x,y)</code>
	...
	...
	<code>fun: add ... # f(a0,a1)</code>
	...
	<code>add v0, r0, r5 # v0 <- f(x,y)</code>
	<code>jr ra # IP <- ret</code>
	...

Adiante veremos como empregar funções com mais do que dois argumentos. Vejamos o que fazer quanto ao aninhamento de funções.

Aninhamento de funções

Ainda nos falta definir o mecanismo de *software* para suportar o aninhamento de funções. Em tese, o par `jal-jr` nos garante que cada função retornará para o ponto de onde foi invocada. Infelizmente, essa tese é inválida.

Considere o programa abaixo: o programa principal invoca `alfa()`, que invoca `beta()`, que por sua vez invoca `gama()`. Quando a função `alfa()` é invocada, o registrador `ra` é atualizado com o endereço de retorno no programa principal, indicado pelo *label* (`r_0`) na versão em *assembly*. Quando `beta()` é invocada, `ra` é sobrescrito com o endereço de retorno em `alfa()`, indicado por `r_1`. Quando `gama()` é invocada, `ra` é sobrescrito novamente com o endereço de retorno em `beta()`, que é `r_2`. Ao longo da execução deste programa, ao registrador `ra` são atribuídos os seguintes valores: *indefinido* $\rightsquigarrow$ `r_0` $\rightsquigarrow$ `r_1` $\rightsquigarrow$ `r_2` $\rightsquigarrow$ `r_1` $\rightsquigarrow$ `r_0`.

Pascal

```
program principal(...);

  function alfa(...):integer;
  begin
    ...
    x := beta(...);
    ...
  end;

  function beta(...):integer;
  begin
    ...
    y := gama(...);
    ...
  end;

  function gama(...):integer;
  begin
    ...
  end;

begin
  ...
  z := alfa(...);
  ...
end.
```

assembly

```
alfa: addi sp, sp, -16 # aloca
      st    10(sp), ra  # ra = r_0
      ...
      jal   beta        # ra <- r_1
r_1:  add   rx, r0, r1   # x:=beta()
      ...
      ld    ra, 10(sp)   # ra <- r_0
      addi  sp, sp, +16  # devolve
      jr    ra           # IP <- r_0

beta:  addi  sp, sp, -20 # aloca
      st    18(sp), ra  # ra = r_1
      ...
      jal   gama        # ra <- r_2
r_2:  add   ry, r0, r1   # y:=gama()
      ...
      ld    ra, 18(sp)   # ra <- r_1
      addi  sp, sp, +20  # devolve
      jr    ra           # IP <- r_1

gama:  addi  sp, sp, -8  # aloca
      ...
      addi  sp, sp, +8   # devolve
      jr    ra           # IP <- r_2

principal:
      addi  sp, r0, 0xffff # inicia
      ...
      jal   alfa        # ra <- r_0
r_0:  add   rz, r0, r1   # z:=alfa()
      ...
      halt                # termina
```

Além do endereço de retorno, é necessário alocar espaço em memória para cada uma das funções, espaço que deve ser desalocado quando a função retorna. Isso é particularmente importante com funções recursivas, que são funções que invocam a si próprias⁴.

⁴Funções recursivas estão fora do escopo deste capítulo.

Uso da pilha

A estrutura de dados empregada para suportar funções é uma *pilha* porque esta estrutura reflete o comportamento natural de programas com funções, especialmente quando se emprega funções aninhadas, ou funções que invocam outras funções. Além de refletir o comportamento dos programas, a manutenção da pilha é extremamente simples e só necessita um registrador para apontar o *topo da pilha*. No nosso caso, este registrador é chamado de *stack pointer*, e o registrador empregado é o r14, apelidado de sp. Por convenção, o sp aponta para a última posição ocupada.

Tipicamente, o registrador que aponta para o topo da pilha é inicializado no endereço mais alto da memória de dados, e portanto a pilha *cresce para baixo*: a cada novo registro acrescentado à pilha, o valor do apontador é decrementado, e quando um registro é removido da pilha, o apontador é incrementado. Em breve veremos o que é um *registro*.

O trecho de programa abaixo mostra um programa Pascal com duas funções, `f()` e `g()`. O programa principal necessita de 256 variáveis e portanto aloca 256 posições na pilha, desde o endereço em que a pilha é inicializada (0xffff) descendo 0x100 posições até o endereço 0xfeff=0xffff-0x0100. Quando o código da função principal está a executar, seu *stack pointer* aponta para 0xfeff.

Pascal	uso da pilha
program ...	# <i>ender.</i> <i>função</i>
...	0xffff principal
function f(x:int): int;	0xfffe 256 posições
begin	
...	0xff00 ...
j := g(k);	0xfeff <- sp de principal
...	
f := ...;	0xfefe f()
end ;	0xfefd 32 posições
function g(x:int): int;	
begin	0xfedf ...
...	0xfede <- sp de f()
g := ...;	
end ;	0xfedd g()
begin	0xfedc 16 posições
...	
a := f(b);	0xfece ...
...	0xfecd <- sp de g()
end.	

Quando a função `f()`, que necessita 32 posições na pilha para as suas variáveis locais, é invocada, o conteúdo do sp é decrementado de 32 posições, passando a apontar para 0xfede=0xfeff-0x0020. Por sua vez, a função `g()` necessita 16 posições na pilha para suas variáveis locais, e durante a execução das instruções de `g()`, o sp aponta para o endereço 0xfecd=0xfede-0x0010.

No programa da página 355, a primeira instrução do programa principal inicializa o apontador de pilha para o endereço mais alto da RAM, que no caso do Mico XII é 0xffff ou 65.535. Quando o *hardware* é inicializado, os conteúdos dos registradores são indefinidos e portanto o programa, logo no seu início, deve atribuir os valores adequados aos registradores. No nosso caso, o mais importante é inicializar a pilha com o endereço mais alto da memória RAM.

Registro de ativação

Um *registro de ativação* (*stack frame*) é alocado para cada função. A Figura 10.9 mostra o leiaute de um registro de ativação completo. Nem sempre é necessário alocar todos os campos de um registro de ativação. Por exemplo, se a função não invoca nenhuma outra função, então não é necessário salvar o endereço de retorno. Se não é necessário preservar os valores de a0 e a1, então estes não são preservados no registro de ativação em memória.

	<i>registro da função que chamou</i>
	argumentos a mais que 2 (3,4,5...)
	registradores com argumentos 1 e 2 (a0, a1), se necessário
	endereço de retorno (ra), se necessário
	registradores preservados, se são alterados pela função
	variáveis locais, se existem
sp →	área para construir argumentos a mais que 2, se existem
	<i>registro da próxima função a ser chamada</i>

Figura 10.9: Registro de ativação no Mico.

O número de registradores do processador é limitado em 15, ou como veremos em breve, limitado a 10 se a convenção da Tabela 10.7 for observada. Isso significa que o código de uma função pode alterar os conteúdos dos registradores da função que a invocou, já que o número de registradores é limitado. Por isso o registro de ativação contém espaço para que uma função salve na pilha o conteúdo dos registradores que ela altera. Quando a função está prestes a retornar, os conteúdos originais devem ser recuperados da pilha e gravados nos respectivos registradores. Do ponto de vista da função que invoca, a função invocada não pode alterar o conteúdo de *nenhum* registrador, a menos de v0, a0 e a1. Voltaremos a este assunto quando apresentarmos a convenção de uso de registradores.

A Figura 10.10 mostra o registro de ativação da função `g()` do exemplo acima, sem detalhar o registro de ativação da função `f()`.

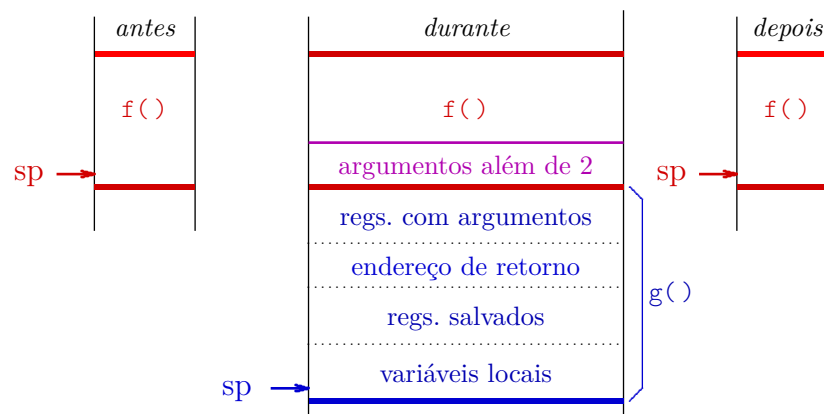


Figura 10.10: Registro de ativação das funções `f()` e `g()`.

Argumentos para além do segundo devem ser gravados no registro de ativação da função que invoca ($f()$ no exemplo da Figura 10.10), o terceiro argumento apontado por sp , o quarto argumento na posição $sp+1$, e assim por diante.

No topo do registro de ativação são preservados os dois argumentos, seguidos do endereço de retorno. Se a função $g()$ altera um ou mais registradores, seus valores originais – necessários na função $f()$ – devem ser preservados no registro de ativação. Abaixo dos registradores preservados ficam as variáveis locais da função.

O programador, ou o compilador, é responsável por definir o tamanho e a quantidade de campos no registro de ativação. Para tanto, o programador deve observar quantos e quais os tipos dos argumentos da função, quantos e quais registradores o código da função altera, e quantas e quais os tipos das variáveis locais à função. Com estes dados, é uma tarefa trivial definir o registro de ativação de uma função.

Exemplo 10.6 Vejamos um exemplo com a criação de um registro de ativação para uma função com três argumentos e que emprega três variáveis locais, mostrada no Programa 10.1. A convenção de programação de funções do Mico reserva dois registradores para passar argumentos para uma função, que são os registradores $a0=r2$ e $a1=r3$. Se uma função tem três ou mais argumentos, os argumentos para além do segundo devem ser armazenados na pilha *antes* da invocação da função. O terceiro argumento fica na posição apontada por sp *antes* do salto para a função, o quarto fica em $sp+1$, o quinto em $sp+2$, e assim por diante.

A Figura 10.11 mostra o estado da pilha durante a execução da função $h(x,y,z)$.

<

Programa 10.1: Parte do código da função $h(x,y,z)$.

```
# w = h(x,y,z);
add  a0,r0,rx      # prepara 3 argumentos
add  a1,r0,ry
st    0(sp),rz      # empilha 3o argumento
jal   ra, h         # salta, ra <- i = IP+1 (link)
i: add  rw,r0,v0     # valor de retorno w <- v0
...
...
h: addi sp,sp,-6     # espaço para 2 args + ra + 3 vars
st    4(sp),a0       # empilha a0
st    5(sp),a1       # empilha a1
st    3(sp),ra       # empilha endereço de retorno
# corpo de h()
...
ld    r5, 6(sp)      # carrega valor de z
...
add   v0,r0,r5       # valor de retorno r5 -> v0
ld    ra,3(sp)       # recompõe endereço de retorno
ld    a0,4(sp)       # recompõe a0
ld    a1,5(sp)       # recompõe a1
addi  sp,sp,6        # devolve espaço na pilha
jr    ra             # retorna para endereço i
```

sp anterior	z	sp + 6
	a1 = y	sp + 5
	a0 = x	sp + 4
	ra = i	sp + 3
	var loc1	sp + 2
	var loc2	sp + 1
sp →	var loc3	sp + 0

Figura 10.11: Registro de ativação do Programa 10.1.**Convenção para escrever funções**

Para escrever programas no *assembly* do Mico usaremos uma convenção similar à do processador MIPS, indicada na Tabela 10.7.

Tabela 10.7: Convenção para uso de registradores.

reg.	nome	função
0	r0	sempre zero (<i>hardware</i>)
1	v0	valor de retorno de função
2	a0	primeiro argumento de função
3	a1	segundo argumento de função
4..13	r4..r13	registradores de uso geral
14	sp	apontador de pilha
15	ra	endereço de retorno

Além de reservar alguns registradores, a convenção determina que: (i) a pilha acima do registro de ativação da função que está executando seja preservada – a função invocada *não pode* alterar as variáveis da função que a invocou; (ii) que o conteúdo do sp recebido pela função seja preservado quando do seu retorno; (iii) o mesmo se aplica ao ra; e (iv) os registradores de uso geral cujo conteúdo é destruído pela função invocada devem ser preservados para não interferir com o estado da computação da função que a invocou, a menos do resultado da execução da função. Estas condições estão explicitadas na Tabela 10.8.

Tabela 10.8: Preservação de conteúdos entre chamadas de funções.

PRESERVADOS	NÃO PRESERVADOS
ra (<i>return address</i>)	a0, a1 (argumentos)
sp (<i>stack pointer</i>)	v0 (valor de retorno)
registradores de uso geral	
pilha acima do reg. de ativação corrente	pilha abaixo do sp

O conteúdo dos registradores com argumentos a0 e a1, e o valor de retorno em v0 podem ser alterados. A pilha abaixo do registro de ativação corrente pode ser alterada – uma função pode invocar outras funções cujos registros de ativação ficarão, na pilha, *abaixo* do registro da função corrente.

O Programa 10.2 mostra o código *assembly* para a implementação do comando $z := f(x)$, tendo $f()$ um argumento inteiro e retornando um valor inteiro. A cada chamada de função encontrada num programa, o compilador deve gerar instruções para efetuar os 6 passos listados abaixo. Os números das linhas indicadas referem-se ao Programa 10.2.

1. Alocar os argumentos onde o corpo da função possa encontrá-los (linha 1);
2. transferir controle para a função e armazenar o endereço de retorno (*link*) no registrador ra (linha 2);
3. o corpo da função deve alocar o espaço necessário na pilha para computar seu resultado, que são 8 inteiros neste exemplo (linha 5);
4. executar as instruções do corpo da função (linha 6);
5. colocar o valor computado onde a função que chamou possa encontrá-lo, que é o registrador $v0$ (linha 7);
6. devolver o espaço alocado em pilha (linha 8); e
7. retornar controle ao ponto de invocação da função (linha 9).

Programa 10.2: Protocolo de invocação de função no comando $z := f(x)$.

```

1      add   a0, r0, rx # prepara argumento em a0=r2
2      jal   ra, f      # salta para f, ra <- r = link
3  r: add   rz, r0, v0 # valor da função em v0=r1
4      ...
5  f: addi  sp, sp, -8 # aloca espaço na pilha, 8 inteiros
6      ...           # computa valor de f()
7      add   v0, r0, t0 # prepara valor (v0) a retornar
8      addi  sp, sp, 8  # devolve espaço alocado na pilha
9      jr    ra        # retorna, IP <- r

```

10.1.6 Conjunto de Instruções

O processador, a cada ciclo, busca e executa uma instrução de lógica/aritmética, uma instrução de controle de fluxo (salto/desvio), ou uma instrução de acesso à memória. O conjunto de instruções do Mico está definido na Tabela 10.9.

Espaço em branco proposital.

Tabela 10.9: Modelo de programação e instruções do Mico.

Modelo de programação			
16 registradores, r0 a r15, r0=0, r1=v0 valor da função, r2=a0, r3=a1 argumentos, r4..r13 uso geral, r14=sp <i>stack pointer</i> , r15=ra <i>return address</i> , IP <i>Instruction Pointer</i>			
Memória de programa MI [0..64K], Memória de dados M[0..64K], MI e M com 32 bits de largura			
Conjunto de instruções			
opcode	instrução	semântica descrição	
0	add c, a, b	$R(c) \leftarrow R(a) + R(b)$, IP $\leftarrow$ IP+1	soma
1	sub c, a, b	$R(c) \leftarrow R(a) - R(b)$, IP $\leftarrow$ IP+1	subtração
2	mul c, a, b	$R(c) \leftarrow R(a) * R(b)$, IP $\leftarrow$ IP+1	multiplicação
3	and c, a, b	$R(c) \leftarrow R(a) \wedge R(b)$, IP $\leftarrow$ IP+1	conjunção
4	or c, a, b	$R(c) \leftarrow R(a) \vee R(b)$, IP $\leftarrow$ IP+1	disjunção
5	xor c, a, b	$R(c) \leftarrow R(a) \oplus R(b)$, IP $\leftarrow$ IP+1	ou-exclusivo
6	slt c, a, b	$R(c) \leftarrow (R(a) < R(b))$, IP $\leftarrow$ IP+1	<i>set on less than</i>
7	not c, a	$R(c) \leftarrow \text{not}(R(a))$, IP $\leftarrow$ IP+1	complemento
8	addi c, a, K	$R(c) \leftarrow R(a) + \text{extS}(K)$, IP $\leftarrow$ IP+1	soma constante aritmética
9	ld c, D(a)	$R(c) \leftarrow M(R(a) + \text{extS}(D))$, IP $\leftarrow$ IP+1	<i>load from memory</i>
a	st D(a), b	$M(R(a) + \text{extS}(D)) \leftarrow R(b)$, IP $\leftarrow$ IP+1	<i>store to memory</i>
b	show a	display $\leftarrow R(a)$, IP $\leftarrow$ IP+1	exibe valor na saída
c	j E	IP $\leftarrow$ E , R(0) $\leftarrow$ IP+1	salto incondicional
c	jal c, E	IP $\leftarrow$ E , R(c) $\leftarrow$ IP+1	<i>jump and link</i>
d	jr a	IP $\leftarrow$ R(a)	<i>jump register</i>
e	beq a, b, E	IP $\leftarrow ((R(a)=R(b)) ? E : IP+1)$	desvio condicional
f	halt	IP $\leftarrow$ IP + 0	paralisa a execução

Formato das instruções:

3130292827262524232221201918171615141312...03020100

opcode	a	b	c	const (K, D, E)

10.2 Componentes do Processador

Quais são os recursos necessários para implementar o processador?

IP	o <i>instruction pointer</i> é um registrador de 16 bits que aponta o endereço da próxima instrução a ser executada;
MI	uma memória de instruções, com capacidade de 64K palavras, para armazenar o programa por executar;
memória de controle	uma memória ROM com 16 palavras de C bits que contém a tabela com todos os C sinais de controle do processador;
somador	para incrementar o IP;
multiplexadores	para escolher uma dentre as possíveis entradas;
ULA	uma unidade de lógica e aritmética com operandos e resultados de 32 bits;
extensor de sinal	para estender o sinal de um número de 16 bits representado em complemento de dois;
R	um bloco de registradores com 16 registradores, cada um com 32 bits de largura; e
M	uma memória de dados, com capacidade de 64K palavras, para armazenar os dados do programa.

A Figura 10.12 mostra os circuitos combinacionais necessários para a implementação do processador. A *Unidade de Lógica e Aritmética* (ULA) tem como entradas dois operandos de 32 bits e produz um resultado de 32 bits. O valor na saída depende da entrada *func*, que define qual a função a ser aplicada aos operandos, que é uma dentre soma, subtração, multiplicação, conjunção, disjunção, ou-exclusivo ou complemento. O *somador* efetua a soma em 16 bits de seus operandos. O *multiplexador* (*mux*) apresenta em sua saída uma dentre as suas entradas, selecionadas pela entrada *sel*.

A Figura 10.12 também mostra um dos elementos de estado necessários, um *registrador* com atualização que depende de um sinal de controle (*habEscr*). Quando *habEscr*=1, o valor em D é capturado na borda ascendente do relógio e mantido em Q. Se *habEscr*=0, então o valor memorizado anteriormente não se altera. Estas duas situações são mostradas nos diagramas de tempo. O IP é um registrador cujo sinal de habilitação está sempre ativo.

A Figura 10.13 mostra os elementos de estado algo mais complexos do que registradores ‘solteiros’. O *bloco de registradores* (*register bank*) contém 16 registradores de 32 bits cada. Dois registradores podem ser acessados simultaneamente para leitura – são duas portas de leitura, A e B – e um registrador pode ser atualizado na borda do relógio, se o sinal de habilitação estiver ativo – que é a porta de escrita C. A Figura 10.13 mostra a interface do bloco de registradores e das memórias. O registrador 0 é *sempre zero*, e escritas neste registrador são ignoradas. A Seção 8.7.2 mostra o projeto detalhado de um bloco de registradores.

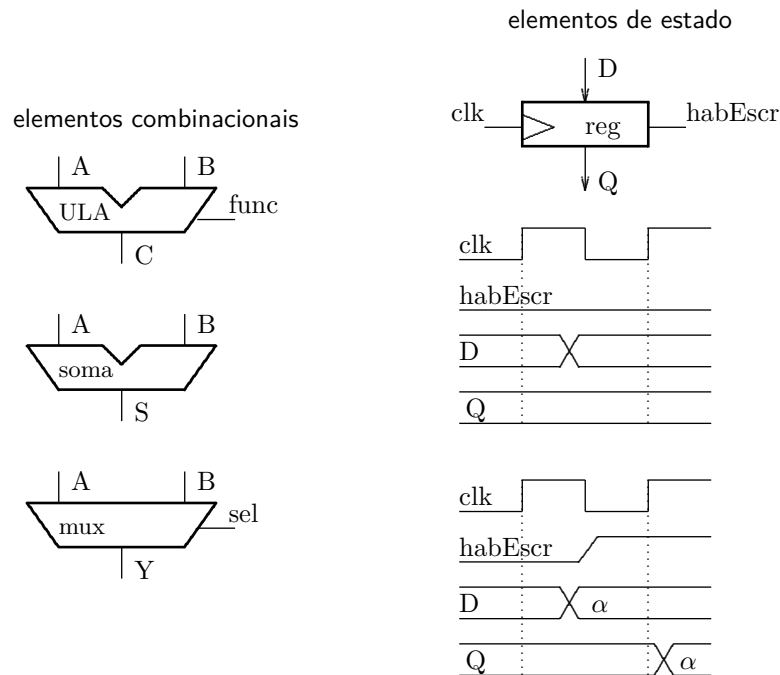


Figura 10.12: Elementos combinacionais e elementos de estado.

Os registradores cujos conteúdos são mostrados nas portas de leitura são selecionados por dois endereços de 4 bits. As portas de leitura se comportam como circuitos combinacionais e podem ser consideradas como dois vetores de inteiros independentes:

```
x <= A(a);
y <= B(b);
```

sendo *a* e *b* dois números representados em 4 bits ($\log(16) = 4$). A porta de escrita se comporta como um registrador e o novo valor só é atualizado na borda do relógio se o sinal de habilitação (*hab*=1) está ativo:

```
if ( rising_edge( clk ) and hab=1 ) then C(c) <= z end if ;
```

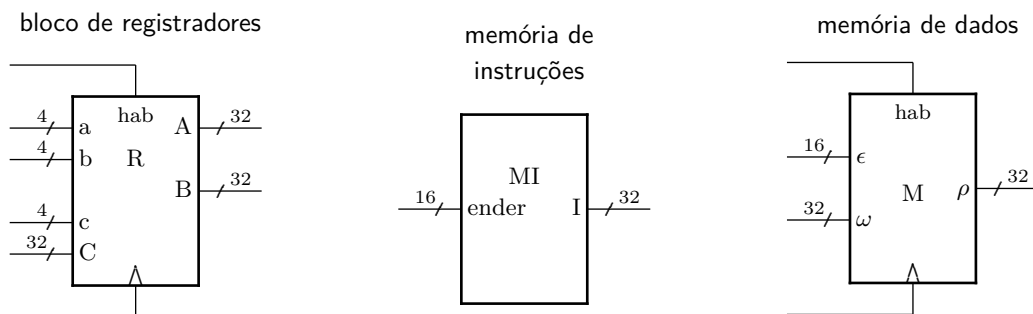


Figura 10.13: Bloco de registradores, memória de instruções e de dados.

A *memória de instruções* (MI) se comporta como um circuito combinacional – uma vez que o endereço estabiliza, passado o tempo de acesso à memória, a instrução fica estável na porta *I*:

```
I <= MI(ender);
```

A *memória de dados* (M) tem um comportamento similar ao banco de registradores. A porta de leitura ρ se comporta como um vetor e a leitura do conteúdo endereçado é combinacional – uma vez que o endereço estabiliza e decorrido o tempo de acesso, o dado fica disponível na porta ρ :

$\rho \leq M(\epsilon)$;

A atualização da palavra indexada pelo endereço ϵ só é efetivada na borda do relógio, se a escrita estiver habilitada ($hab=1$):

if (rising_edge(clk) **and** hab=1) **then** $M(\epsilon) \leq \omega$ **end if**;

A decodificação das instruções é extremamente simples, graças à codificação simples: basta indexar uma tabela de 16 elementos que contém todos os sinais de controle do processador. O *opcode* é um número de 4 bits e é o ‘nome’ da instrução. Esse número indexa a tabela de controle – implementada com uma memória ROM – e os sinais de controle de todos os componentes do processador são definidos pelos seus campos – logo veremos alguns exemplos.

10.2.1 Busca e Decodificação de Instruções

A memória de instruções (MI) mantém o código binário do programa que está sendo executado. No momento não nos interessa como o programa foi gravado nesta memória; mais adiante no curso estudaremos a carga de programas para execução. A MI é indexada pelo registrador *instruction pointer* (IP) e este registrador é incrementado de 1 a cada ciclo do relógio. A Figura 10.14 mostra um diagrama com o circuito que efetua a busca e a decodificação das instruções.

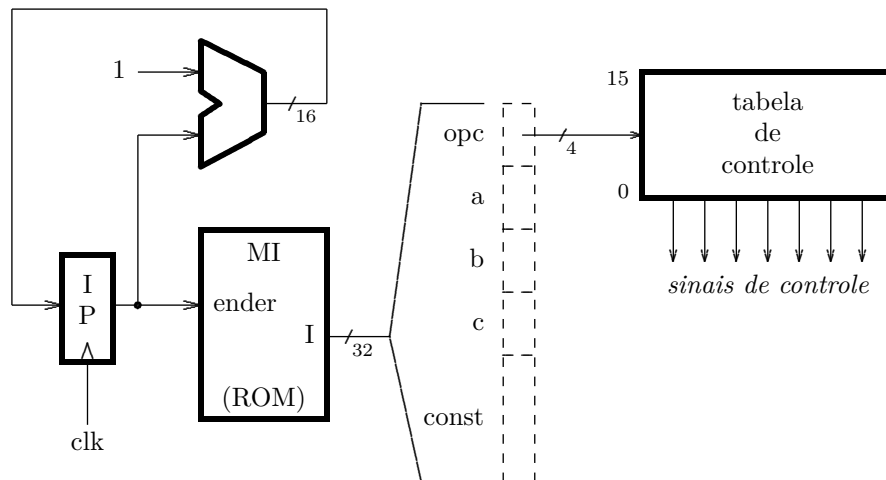


Figura 10.14: Circuito de busca e decodificação de instruções.

A saída do IP é ligada à entrada de endereços da ROM que contém as instruções do programa e a um somador, que incrementa o conteúdo do IP de 1 a cada borda do sinal de relógio (clk). Tanto o IP quanto o somador têm largura de 16 bits.

Uma vez que a saída do IP estabiliza após a borda do relógio, e decorrido o tempo de acesso da MI, uma nova instrução fica disponível para ser decodificada e executada. A *decodificação* da instrução ocorre com a indexação da tabela de controle com o *opcode* – abreviatura para *operation code* (opc), o que ativa os sinais de controle relevantes para a execução da instrução recém buscada.

A largura dos sinais é indicada no diagrama na Figura 10.14 com um traço diagonal e pelo número de bits daquele sinal. O *opcode* tem 4 bits de largura, a instrução *I* tem 32 bits, e o *IP* tem 16 bits. As setas indicam o fluxo dos sinais – de saídas para entradas.

Cada uma das instruções de 32 bits é dividida em cinco campos. O *opcode* tem 4 bits de largura e ocupa a posição mais significativa da palavra (bits 28 a 31). Os campos *a*, *b* e *c* também têm 4 bits de largura e indicam quais registradores são os operandos da instrução (*a* e *b*), e em qual registrador (*c*) o resultado deve ser armazenado –este registrador é chamado de o *destino* da instrução. O campo *constante* (*const*) armazena uma constante inteira, representada em complemento de dois em 16 bits, ou um endereço de 16 bits.

A *tabela de controle* é uma memória ROM com 16 palavras de *C* bits, sendo que cada um dos c_i bits é usado para controlar um dos componentes do processador, tais como multiplexadores e registradores com carga controlada.

O circuito de busca não é mostrado nos próximos diagramas, que descrevem os caminhos de dados das instruções.

10.2.2 Operações de Lógica e Aritmética

As instruções de lógica e aritmética usam dois registradores como fonte dos operandos e um terceiro registrador como destino para o resultado. A Figura 10.15 mostra uma instrução **add** no topo da figura e as ligações necessárias. Os campos da instrução *a* e *b* indexam os registradores e seus conteúdos são apresentados nas portas *A* e *B* do banco de registradores (*R*), e de lá para as entradas α e β da ULA.

O resultado da soma é levado da saída γ da ULA para a porta de escrita (*C*) do banco de registradores. O registrador que deve ser atualizado é indicado pelo campo *c*, e o sinal *escReg* é ativado. Na borda ascendente do relógio a soma de $R(a)$ com $R(b)$ é armazenada em $R(c)$.

$$\text{add } c, a, b \quad \# \quad R(c) \leftarrow R(a) + R(b)$$

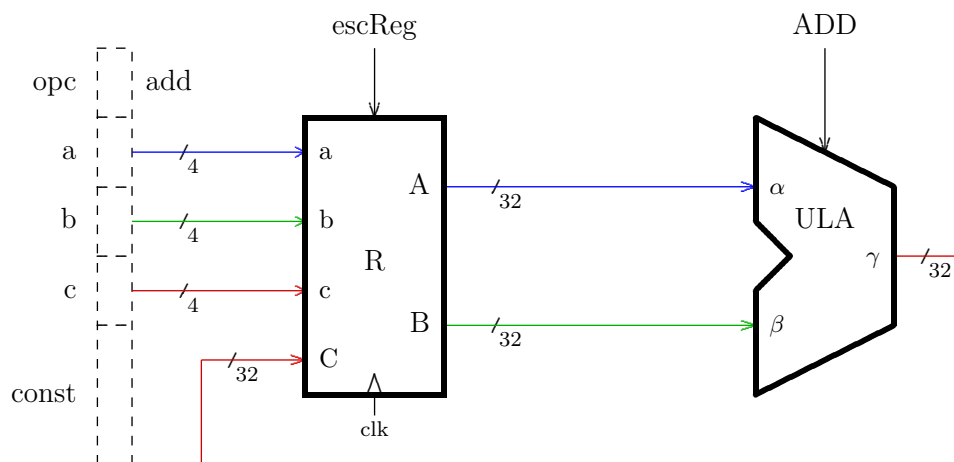


Figura 10.15: Operações de lógica e aritmética – add.

Os sinais de controle *escReg* e *ADD* são gerados/ativados na tabela de controle.

10.2.3 Operações de Lógica e Aritmética com Imediato

Há uma versão da instrução de soma que usa uma constante – que é parte da instrução – como um dos operandos, e o outro operando é o conteúdo de um registrador, apontado por *a*. Esta constante é chamada de *operando imediato*. O registrador de destino é apontado por *c*. O operando da porta B do banco de registradores não é utilizado nesta instrução e portanto as ligações de *b* e *B* são mostradas em tom desbotado. A Figura 10.16 mostra o circuito para executar a instrução de soma com um operando imediato.

A constante é codificada em 16 bits, no campo *const*, e ela deve ser estendida para 32 bits para que possa ser aplicada à entrada β da ULA. Na operação aritmética (**addi**), a constante é estendida para 32 bits pela replicação do seu bit de sinal.

`addi c, a, const    # R(c)  $\leftarrow$  R(a) + extSinal(const)`

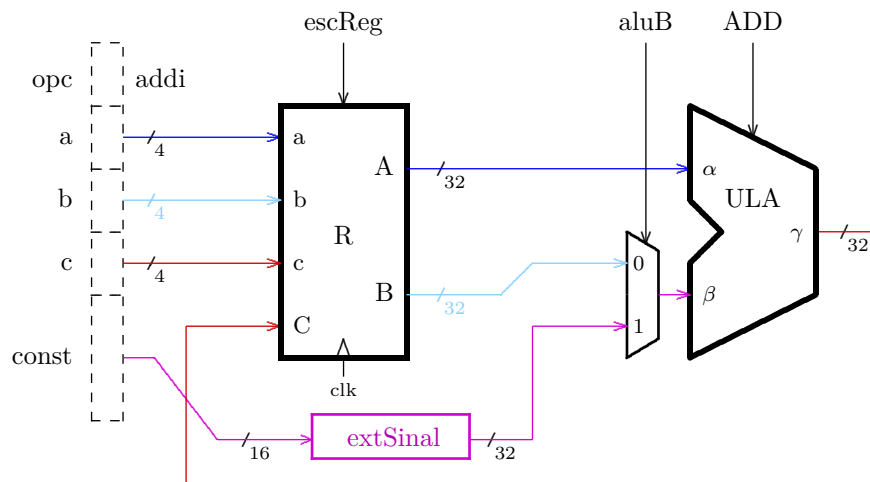


Figura 10.16: Operações de lógica e aritmética com imediato – addi.

O operando apontado por *a* é apresentado à entrada α da ULA; o segundo operando (*const*) é estendido para 32 bits e apresentado à entrada β da ULA. A operação da ULA é definida como ADD e o resultado é armazenado no registrador apontado por *c*, na borda de subida do relógio. Os sinais *escReg* e *aluB* devem ser 1, e estes sinais de controle, mais ADD, são ativados na tabela de controle.

10.2.5 Operação de Acesso à Memória – Escrita

A instrução **st** (*store-word*) copia o conteúdo do registrador apontado por **b** para a palavra indexada pela soma de um deslocamento de 16 bits (**const**) com o conteúdo do registrador apontado por **a**. A Figura 10.18 mostra o diagrama do processador com as ligações para a instrução **st**.

```
st const(a), b      # MD( extSinal(const) + R(a) ) ← R(b)
```

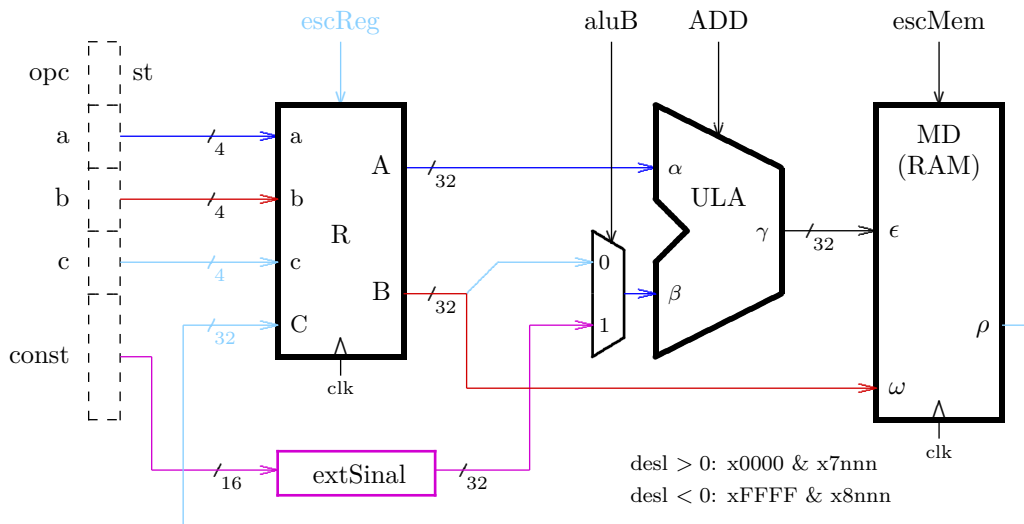


Figura 10.18: Operação de escrita em memória – st.

O endereço efetivo é aplicado à entrada de endereços ϵ da memória de dados e o conteúdo de $R(b)$ é aplicado à porta de escrita ω da memória de dados. O sinal `escMem` é ativado e na borda ascendente do relógio a posição indexada pelo endereço efetivo é atualizada.

Os sinais de controle `escMem`, `aluB` e `ADD` são ativados na memória de controle. O sinal `escReg` não é ativado porque não ocorre uma escrita no bloco de registradores, mas somente na RAM.

10.2.6 Desvio Condicional

Num desvio condicional (**beq** – *branch if equal*), o endereço da próxima instrução a ser buscada depende de uma comparação de igualdade. Para tanto, o conteúdo de dois registradores são comparados. A Figura 10.19 mostra o diagrama do processador com as ligações para a instrução **beq**. O sinal **iguais** indica o resultado da comparação.

`beq a, b, const    # if (R(a)=R(b)) IP <- const else IP <- IP+1`

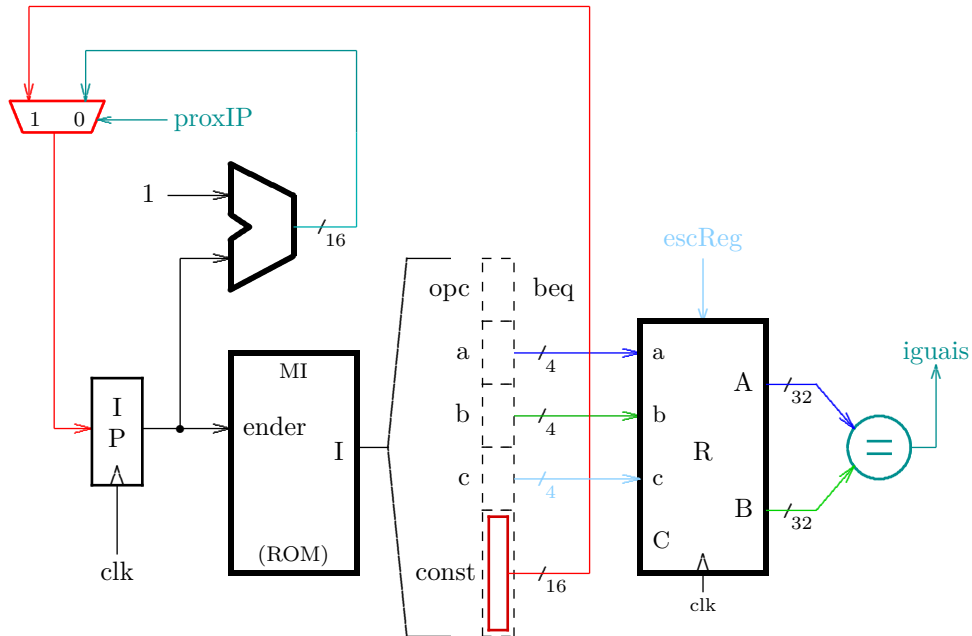


Figura 10.19: Desvio condicional – beq.

O *endereço de destino* de um desvio é a constante *const*. Se *iguais*=1 então o endereço de destino é selecionado e aplicado à entrada do IP, e a instrução apontada por aquele endereço será buscada no próximo ciclo. Do contrário, se *iguais*=0 então *IP+1* é aplicado à entrada do IP e a instrução após o **beq** será buscada. O sinal *proxIP* só é ativado se a instrução for um **beq** e *iguais*=1.

10.2.9 Circuito de Dados Completo

O circuito de dados completo é a ‘união’ dos circuitos necessários para cada classe de instruções e é mostrado na Figura 10.21. O circuito de dados consiste de uma Unidade de Lógica e Aritmética (ULA), de um Bloco de Registradores (R), de uma memória de dados (M), e circuitos auxiliares. O circuito de controle consiste de um apontador de instrução (IP), um somador, uma memória de instruções (MI), e de uma memória ROM com os sinais de controle.

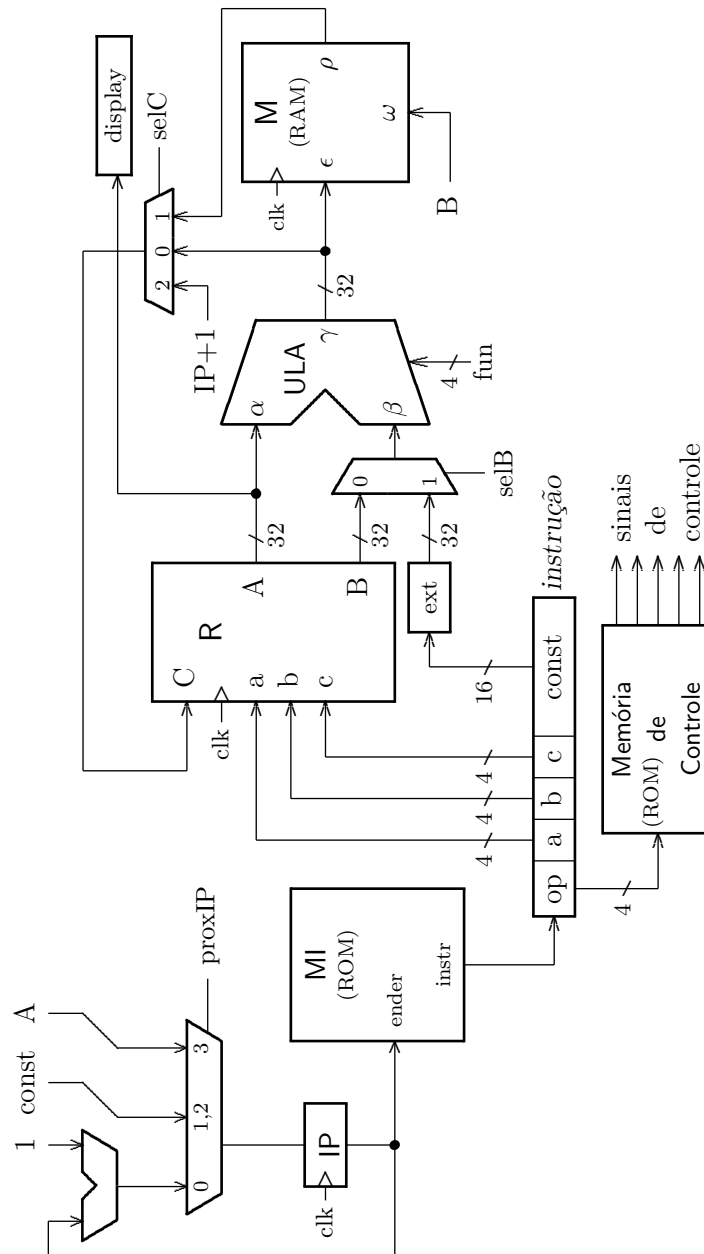


Figura 10.21: Circuito de dados e de controle do Mico.

O valor que será gravado no registrador de destino pode ser aquele na saída da ULA, ou aquele lido da memória ou $IP+1$. Um multiplexador, controlado pelo sinal $selC$, define qual dos três valores é apresentado à porta C do bloco de registradores, para ser gravado no registrador de destino.

Um dos operandos da ULA é sempre o registrador apontado por **a**, que corresponde à porta A do banco de registradores. O outro operando pode ser a porta B do banco de registradores ou a constante estendida. O multiplexador controlado pelo sinal **selB** determina o operando na porta β da ULA.

10.2.10 Tabela de Controle

O projetista do processador deve elaborar uma *tabela de controle* com todos os sinais que controlam os componentes do processador. A Figura 10.22 mostra o circuito de busca e decodificação com os sinais de controle das unidades funcionais do processador – os nomes estão abreviados no diagrama por causa do espaço. A tabela de controle é indexada pelo *opcode* da instrução recém buscada e o conteúdo do elemento indexado pelo *opcode* ativa somente os sinais de controle apropriados à instrução por executar.

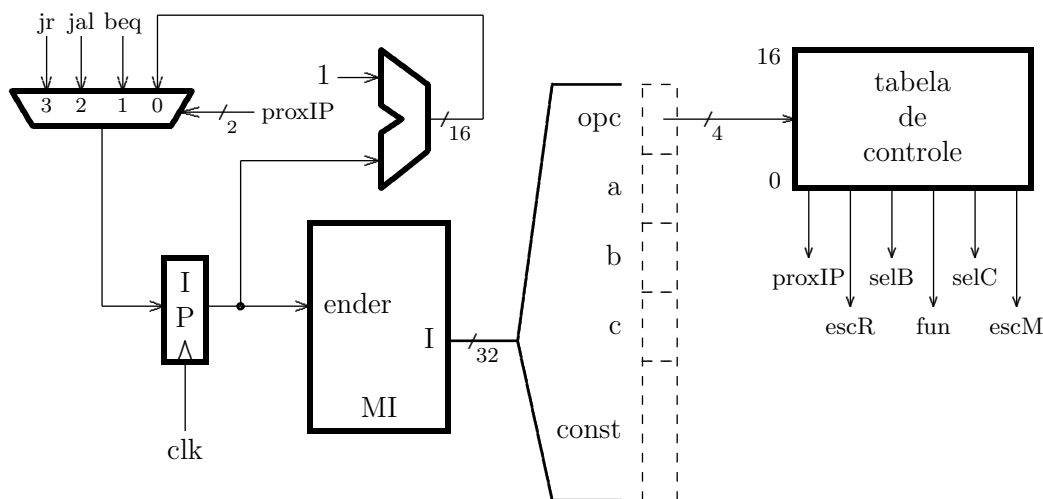


Figura 10.22: Circuito de busca e decodificação com tabela de controle.

A Tabela 10.10 mostra os sinais de controle que devem ser ativados para as instruções vistas nas seções anteriores. Sinais que prescindem de um valor determinado são indicados com um ‘X’ (*don’t care*).

Os sinais de controle que alteram o estado do processador, `proxIP`, `escR` e `escM`, devem estar *sempre* determinados porque não há nenhuma instrução que *talvez* altere a memória, ou que *talvez* altere os registradores.

Tabela 10.10: Tabela de controle para as instruções.

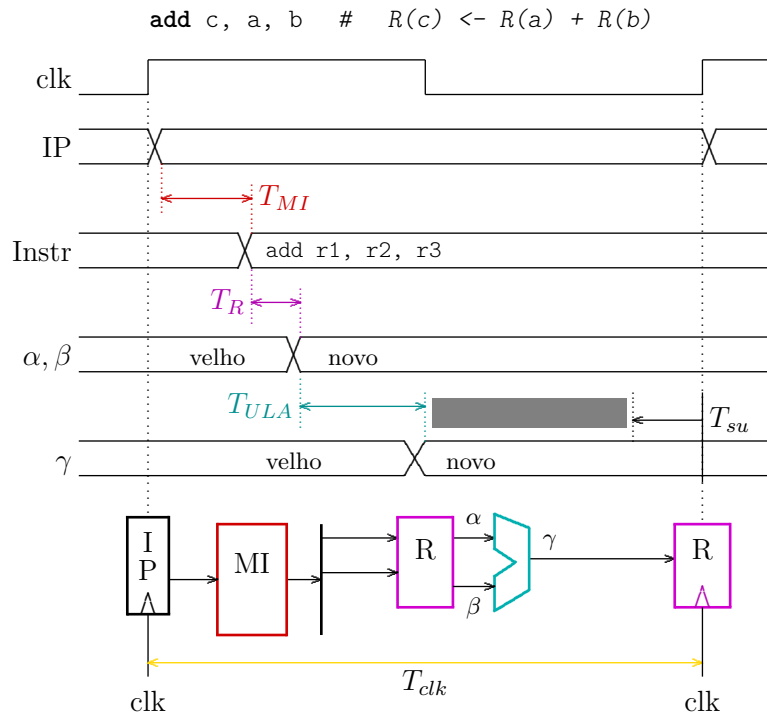
INSTRUÇÃO	proxIP	escR	selB	fun	selC	escM
ALU	0	1	0	oper.	0	0
addi	0	1	1	ADD	0	0
ld	0	1	1	ADD	1	0
st	0	0	1	ADD	X	1
beq	iguais	0	0	X	X	0
jal	2	1	X	X	2	0
jr	3	0	X	X	X	0

10.3 Metodologia de Sincronização

A metodologia de sincronização para esta implementação do conjunto de instruções Mico é chamada de *ciclo longo* porque o período do relógio é longo o bastante para executar uma instrução durante um único ciclo.

10.3.1 ADD

A Figura 10.23 mostra o diagrama de tempo da execução de um **add**. Um diagrama simplificado do processador é mostrado na base da figura, com o bloco de registradores dividido em duas partes: no centro do diagrama está a parte na qual ocorre a leitura dos operandos, e na direita a parte em que ocorre a gravação do resultado.

**Figura 10.23: Diagrama de tempo de uma operação de ULA.**

Decorrido o tempo de acesso à memória de instruções (T_{MI}), a nova instrução é decodificada paralelamente à leitura do banco de registradores (T_R). Com os operandos disponíveis, decorre o tempo de propagação através da ULA (T_{ULA}) até que o resultado esteja disponível no sinal γ . Há folga para respeitar a limitação de *setup* do banco de registradores (T_{su}), como mostra o intervalo entre a disponibilidade do novo valor em γ e a indicação do tempo de estabilização dos sinais (*setup*) – a folga é indicada pela barra de cor cinza.

10.3.2 LD

A Figura 10.24 mostra o diagrama de tempo da execução de um **ld**. O período do relógio deve ser longo o bastante para acomodar o tempo de acesso à memória de instruções (T_{MI}), a leitura do banco de registradores (T_R), o tempo de propagação através da ULA (T_{ULA}), o tempo de acesso para leitura da memória de dados (T_{MD}), e respeitar a limitação de *setup* do banco de registradores (T_{su}). A folga é mostrada pela barra de cor cinza.

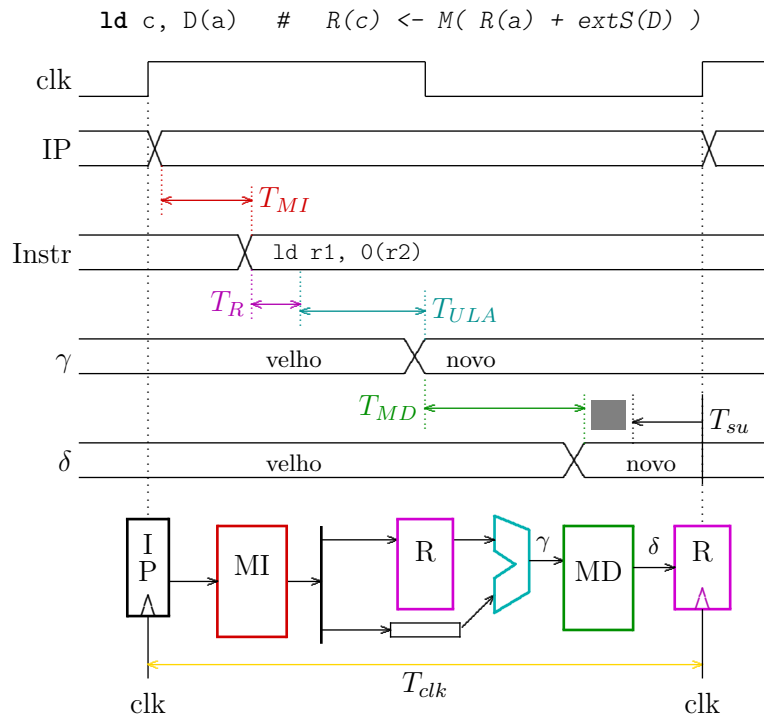


Figura 10.24: Diagrama de tempo de uma leitura em memória.

A instrução **ld** é a instrução mais demorada porque todas as unidades funcionais do processador são usadas em série, $MI \leadsto R \leadsto ULA \leadsto M \leadsto R$, e portanto o ciclo do processador é limitado pela temporização do **ld**.

10.3.3 Período Mínimo

As outras instruções não usam todas as unidades funcionais e há algum desperdício de tempo porque o ciclo do relógio deve ser fixado para atender ao pior caso, que é o **ld**. Por exemplo, na instrução **add**, as unidades funcionais utilizadas são $MI \leadsto R \leadsto ULA \leadsto R$, não há acesso à memória de dados, e o ciclo desta instrução poderia ser encurtado de T_{MD} .

Alterar o ciclo do relógio em função da instrução não é realizável e por isso o período do relógio deve ser tal que acomode a temporização da instrução mais demorada. O Capítulo 12 discute uma implementação mais eficiente do que esta com ciclo longo.

Exemplo 10.7 Considerando os dados da Tabela 10.11, vejamos como computar a duração de um ciclo de relógio do Mico. A instrução mais demorada é o `ld` porque seu fluxo de dados atravessa, em série, todas as unidades funcionais do processador. Na tabela, todos os tempos estão em picossegundos.

O caminho crítico inicia no circuito de busca de instruções, com o registrador IP seguido da memória de instruções: $T_{IP} + T_{MI}$. A instrução indexa o banco de registradores (T_R) para obter o valor de $R(a)$; ao mesmo tempo, o deslocamento deve ser estendido⁵ para 32 bits e atravessar o *mux-2* na entrada β da ULA: $T_b + T_{m2}$. O endereço efetivo é computado na ULA (T_U) e aplicado à entrada de endereços da memória de dados; o conteúdo da posição indexada, através do *mux-4*, é levado à entrada do banco de registradores: $T_M + T_{m4} + T_{su,R}$. O tempo de propagação deste caminho é

$$\begin{aligned} T_{ld} &= T_{IP} + T_{MI} + \max(T_R, T_b + T_{m2}) + T_U + T_M + T_{m4} + T_{su,R} \\ &= 40 + 250 + \max(60, 5 + 40) + 150 + 250 + 40 + 20 \\ &= 810 \text{ ps.} \end{aligned} \quad (10.1)$$

Deve-se escolher o pior caso, o máximo, dentre dois caminhos paralelos, como é o caso do acesso à $R(a)$ que ocorre em paralelo à extensão do sinal do deslocamento. $\triangleleft$

Tabela 10.11: Tempos de propagação dos componentes do Mico.

Componente	T_{prop}	T_{setup}
IP	T_{IP} 40	$T_{su,I}$ 20
somador de 16 bits	T_s 75	
Mem. Instr (MI)	T_{MI} 250	
registradores (R)	T_R 60	$T_{su,R}$ 20
<i>buffer</i> -16	T_b 5	
mux-2 de 32 bits	T_{m2} 20	
mux-4 de 16 bits	T_{m4} 40	
mux-4 de 32 bits	T_{m4} 40	
ULA (soma)	T_U 150	
Mem. Dados (M)	T_M 250	$T_{su,M}$ 40

Do ponto de vista da fabricação em grande escala, não é uma boa ideia usar como período de relógio exatamente o valor para o atraso através do caminho crítico. Qualquer desvio no processo de fabricação, ou na temperatura de operação, pode provocar comportamento incorreto por problemas com a temporização. Por isso, deve-se alocar uma folga no período, algo como 5 a 10% do valor calculado na Equação 10.1. Se considerarmos uma folga pessimista de 10%, então o período mínimo seria de $T_{ld} \times 1,1 = 891 \text{ ps} \approx 890 \text{ ps}$. Esse período corresponde a uma frequência de $\approx 1,24 \text{ GHz}$, que é relativamente elevada para aplicações embarcadas. Uma frequência tão elevada implica em grande dispêndio de energia, o que pode inviabilizar aplicações alimentadas por bateria. Para estas aplicações, a frequência de operação deveria ser uma fração dos $1,24 \text{ GHz}$. A Seção 5.3.3 trata da relação entre frequência de operação e dispêndio de energia.

⁵O bit de sinal deve ser replicado 16 vezes; para evitar carga excessiva nesse sinal, empregamos um *buffer* para reproduzir as 16 cópias. As razões para tal são discutidas na Seção 5.4.

Exemplo 10.8 Uma vez definido o período mínimo do relógio, é necessário verificar se todas as demais instruções completam com folga. A instrução **st** aparenta ter um período similar ao do **ld**.

A temporização é a mesma que para o **ld** desde a busca até o cálculo do endereço efetivo. O valor a ser gravado na memória provém de $R(b)$, que fica disponível ao mesmo tempo que $R(a)$. Uma vez computado o endereço, deve-se atender ao *setup* da memória de dados $T_{su,M}$. O tempo do **st** é

$$\begin{aligned} T_{st} &= T_{IP} + T_{MI} + \max(T_R, T_b + T_{m2}) + T_U + T_{su,M} \\ &= 40 + 250 + \max(60, 5 + 40) + 150 + 40 \\ &= 540 \text{ ps} \end{aligned} \tag{10.2}$$

O caminho crítico do **st** tem 540 ps, o que é 0,67 daquele do **ld**. Considerando o período de 890 ps, a folga é de 350 ps. ◁

10.3.4 Extensões ao Processador

O departamento de vendas avisa ao departamento de engenharia que os clientes necessitam de instruções adicionais, tais como deslocamentos, e de um desvio condicional que compare desigualdade. Essas modificações não podem alterar (demasiadamente) a codificação das instruções, para permitir que o código antigo e já compilado possa ser usado nos novos processadores. A Tabela 10.9 indica os possíveis caminhos para as adições.

No caso dos desvios a modificação solicitada pode ser implementada facilmente porque o campo para o registrador de destino (*c*) dessas instruções não é usado, o que permitiria adicionar quinze novas instruções de desvio, todas com o mesmo *opcode*. O campo *c* passa a ser o *opcode secundário* dos desvios.

Para implementar a instrução **beq** (*branch if equal*) basta computar a diferença entre os dois operandos e verificar se o resultado é zero. Para implementar a instrução **bzero** (*branch if zero*), o operando apontado por *a* é subtraído de *r0* e se o resultado é zero, então o desvio deve ser tomado. A instrução **ba** (*branch always*) equivale a um salto incondicional e é obtida com **beq** *r0,r0,dest*.

As instruções de ULA usam somente três registradores e ignoram o campo de deslocamento. As instruções de ULA podem ser alocadas no *opcode* 0, por exemplo, e alguns bits do deslocamento são usados para codificar a operação de ULA desejada. Essa modificação liberaria os *opcodes* 1 a 7 para outras instruções, tais como os deslocamentos. Se forem usados 4 bits do deslocamento, a ULA pode ser expandida para suportar 16 operações ao invés das atuais 8.

Exercícios

Ex. 10.1 Mostre como implementar as instruções abaixo. Para tanto, indique quaisquer modificações que sejam necessárias no processador da Figura 10.21 e mostre a tabela de sinais de controle ativos durante a execução da instrução.

```

# jump-register
jr ra      # IP <- R(ra)

# jump-and-link register
jalr rc, ra # IP <- R(ra) , R(rc) <- IP+1
```

Ex. 10.2 Desenhe os diagramas de tempo para as instruções do Ex. 10.1.

Ex. 10.3 Calcule o caminho crítico para as instruções do Ex. 10.1.

Ex. 10.4 Calcule o caminho crítico para as instruções de ULA.

Ex. 10.5 Calcule o caminho crítico para o **beq**. Suponha que a comparação é efetuada na ULA, através de uma subtração.

Instrução	semântica	nome
beq ra,rb,D	$IP \leftarrow D \triangleleft ra = rb \triangleright IP + 1$	<i>branch if equal</i>
bne ra,rb,D	$IP \leftarrow D \triangleleft ra \neq rb \triangleright IP + 1$	<i>branch if not equal</i>
bzero ra,D	$IP \leftarrow D \triangleleft ra = 0 \triangleright IP + 1$	<i>branch if zero</i>
bnz ra,D	$IP \leftarrow D \triangleleft ra \neq 0 \triangleright IP + 1$	<i>branch if not zero</i>
bpos ra,D	$IP \leftarrow D \triangleleft ra > 0 \triangleright IP + 1$	<i>branch if positive</i>
bneg ra,D	$IP \leftarrow D \triangleleft ra < 0 \triangleright IP + 1$	<i>branch if negative</i>
ba D	$IP \leftarrow D$	<i>branch always</i>

Tabela 10.12: Instruções adicionais: desvios condicionais

Ex. 10.6 Com base no sugerido na Seção 10.3.4, mostre como implementar as instruções da Tabela 10.12. Sua resposta deve conter a codificação das instruções, indicar como as operações de comparação são efetuadas, e os sinais de controle necessários para as novas instruções. Uma codificação cuidadosa do *opcode* secundário no campo *c* pode simplificar a lógica de controle.

Ex. 10.7 Com base no sugerido na Seção 10.3.4, mostre como implementar as modificações para as instruções de ULA. Sua resposta deve conter a codificação das instruções, e eventuais mudanças na tabela de controle do processador.

Índice Remissivo

Símbolos

T_A , 64
 T_D , 120
 T_I , 109, 116
 T_M , 117, 119, 183
 T_P , 116
 T_R , 374
 T_h , 185, 189, 196, 201
 T_p , 170
 T_s , 374
 $T_{C,F}$, 189
 $T_{C,x}$, 118–120, 196, 200, 202
 T_{FF} , 189
 T_{MD} , 374, 377
 T_{MI} , 374, 377
 T_{ULA} , 374, 377
 T_{min} , 195, 202, 374
 T_{skew} , 199
 T_{su} , 185, 189, 196, 201, 374, 377
 $\%$, 18–20, 347
 $\Rightarrow$, 37
 $\bigvee$, 47
 $\bigwedge$, 47
 $\equiv$, 36
 $\gg$, 155
 $\wedge$, 30, 36
 $\triangleleft \triangleright$, 36, 42, 66
 $\leftarrow$, 190, 339
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 41, 44
 $\ll$, 155
 $\neg$, 30, 36, 153
 $\vee$, 30, 36
 $\oplus$, 36, 53, 65
 $\mapsto$, 42
 $\mathbb{N}$, 146
 $\bar{a}$, veja $\neg$
 π , 25
 $\backslash$, 77
decod, 72
demux, 75
 e , 24
mod, 159, 214, 347
mux, 67
 $\&$, 339
 $\mathbb{B}$, 29–33, 38, 42
 $\mathbb{Z}$, 42, 151
 $[r]$, 237, 238
 $\mathbb{N}$, 42, 43, 151

num, 44, 55, 72, 77, 257, 258, 260, 270
 num^{-1} , 55
 $\mathbb{R}$, 42
 $|N|$, 223
 $\langle \rangle$, 29, 42, 43, 178, 197, 221
 $,$, 339
 $;$, 339

Números

74148, 75
 74163, 214, 268
 74374, 228

A

abstração, 27
 bits como sinais, 27–33, 57, 180, 181
 tempo discretizado, 116, 118, 180, 183–185,
 188–189, 193–194, 218
 acumulação de produtos parciais, 171–176
 adição, 153
 adiantamento de vai-um, 165–169
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo, 229
 conversão de base, 18
 conversão de base de frações, 22
 amplificador, 125, 136
 diferencial, 138
 amplitude, 206
 and, veja $\wedge$
and array, 128
 apontador de pilha, 264
 aproximação, 24
 arquitetura, 338
 árvore, 64, 118
assembly, veja ling. de montagem
 associatividade, 31
 atraso, veja tempo de propagação, 57
 atribuição, 12
 autômatos finitos, 232

B

barramento, 140
barrel shifter, 159
 báculo, 138, 181–186
 relógio multi-fase, 222
 SR, 182, 191
 binário, 20
 bit, 20
 de sinal, 147

bits, 27–37, 65
 definição, 29
 expressões e fórmulas, 30
 expressão, 30
 propriedades da abstração, 31
 variável, 30
 bloco de registradores, 260, 362, 372, 373
 borda, *veja* relógio
 branch, *veja* desvio condicional
 buffer, 123, 375
 buffer three-state, 140
 buraco, 87
 busca, 364
 busca binária, 270
 byte, 11

C

C,
 deslocamento, 160
 overflow, 164
 cadeia,
 de portas, 64, 118
 de somadores, 153
 caminho crítico, 117
 capacitor, 105, 107, 135, 136
 capture FF, *veja* flip flop, destino
 CAS, 134
 célula de RAM, 81
 célula, 100
 chave,
 analógica, 142
 digital, 92
 normalmente aberta, 92
 normalmente fechada, 78, 92
 chip select, 264
 ciclo,
 combinacional, 57
 violação, 81, 181, 183, 185
 de trabalho, 221
 ciclo longo, 373
 circuito,
 combinacional, 57, 65
 de controle, 253
 de dados, 253
 dual, 99
 segmentado, 199
 sequencial síncrono, 193, 205
 circuito aberto, 57
 clear, 191
 clk, *veja* relógio
 clock, *veja* relógio
 clock skew, *veja* skew
 clock-to-output, 189
 clock-to-Q, 189
 CMOS, 59, 65, 85–142
 buffer three-state, 140
 célula, 100
 inversor, 96
 nand, 99

nor, 98
 porta de transmissão, 141
 portas inversoras, 99
 sinal restaurado, 125
 codificação das instruções,
 Mico, 340
 código,
 Gray, 63, 213, 223, 232
 Column Address Strobe, *veja* CAS
 combinacional,
 ciclo, 57
 circuito, 57
 dispositivo, 57
 comentário, 339
 comparador,
 de igualdade, 62, 164
 de magnitude, 164, 257, 270
 Complementary Metal-Oxide Semiconductor, *veja* CMOS
 complemento, *veja* $\neg$
 complemento de dois, 145–151, 153–164
 complemento, propriedade, 31
 comportamento transitório, *veja* transitório
 comutatividade, 31
 condicional, *veja* $\triangleleft \triangleright$
 condutor, 85
 conjunção, *veja* $\wedge$
 conjunto mínimo de operadores, 65
 contador, 207–219
 74163, 214
 assíncrono, 216
 em anel, 220, 221, 228
 inc-dec, 220, 268
 Johnson, 223
 módulo-16, 211, 214
 módulo-2, 208, 219
 módulo-32, 219
 módulo-4, 208
 módulo-4096, 215
 módulo-8, 209, 217
 ripple, 216, 218
 síncrono, 218
 contra-positiva, 41
 controlador, 238
 de memória, 136
 controle de fluxo, 348–352
 conversão de base, 18
 conversor,
 paralelo-série, 225
 série-paralelo, 224
 corrente, 85, 105, 106, 112, 113
 de fuga, 115
 corrida, 123, 125
 CSS, 193, 205
 curto-circuito, 57

D

datapath, *veja* circuito de dados
 decimal, 17

decodificação, 339, 364
 decodificador, 72, 78, 82–84, 126
 de linha, 126, 135
 de prioridades, 74
delay, 57
 demultiplexador, 75, 120
design unit, *veja* VHDL, unidade de projeto
 deslocador exponencial, 156
 deslocamento, 155–159
 aritmético, 155, 159, 164
 exponencial, 159, 197
 lógico, 155, 162
 rotação, 159
 desvio condicional, 369, 377
 detecção de bordas, 123
 diagrama de estado, 230
 restrições, 232
 disjunção, *veja* $\vee$
 dispositivo, 85
 combinacional, 57
 distributividade, 31, 34, 51
 divisão de frequência, 209, 212
 divisão inteira, 44, 269
 doador, 87
don't care, 70, 372
 dopagem por difusão, 86
 dopante, 86
 DRAM, 134–137
 controlador, 136
 fase de restauração, 137
 linha,
 de palavra, 135
 linha de bit, 135
 linha de palavra, 136
 página, 135
 refresh, 135
 dual, 32, 95, 99
 dualidade, 32
duty cycle, 221

E

EEPROM, 133
 endereço, 77
 base, 346
 de destino, 348, 369, 370
 de retorno, 353, 370
 efetivo, 346, 367, 368
 energia, 100, 105, 112–115
 enviesado, relógio, *veja skew*
 EPROM, 133
 equação característica do FF, 190
 equivalência, *veja* $\Leftrightarrow$
 erro,
 de representação, 23
 espaço,
 de nomes, 338
 especificação, 42
 estado, 180
 estado atual, 193, 205, 208

execução, 365–367, 372
 paralela, 339
 sequencial, 339
 exponenciação, 260
 expressões, 36
 extensão,
 de sinal, 366

F

fan-in, 109–112, 116, 167
fan-out, 82, 84, 109–112, 116, 120, 169, 170
 fatorial, 269
 fechamento, 31
 FET, 91
Field Effect Transistor, *veja* FET
Field Programmable Gate Array, *veja* FPGA
 FIFO, 267
 fila, 266–269
 circular, 267
 filtro digital de ruído, 187
flip-flop, 185–191
 adjacentes, 194
 comportamento, 190
 destino, 194
 fonte, 194, 203
 mestre-escravo, 186
 modelo VHDL, *veja* VHDL, *flip-flop*
 temporização, 189
 tipo JK, 190
 tipo T, 188, 190
 um por estado, *veja* um FF por estado
 folga de *hold*, 195
 folga de *setup*, 195
 forma canônica, 48
 formato de instrução,
 Mico, 340
 FPGA, 191
 frações, *veja* ponto fixo
 frequência, 206
 frequência máxima, *veja* relógio
 função, 30
 aninhamento, 355
 protocolo de invocação, 354
 tipo, 29, 42
 função de próximo estado, 229, 236
 função de saída, 229, 236
 função, aplicação bit a bit, 32
 função, tipo (op. infixo), *veja* $\mapsto$
 funções, 353–360

G

glitch, *veja* transitório
 GND, 93
 gramática, 17

H

handshake, 253
 hexadecimal, 19
hold time, 185, 193–199, 242
 folga, 195, 200, 201

I

idempotência, 31
 identidade, 31
 igualdade, 30
 imediato, 366
 implementação, 42
 implicação, *veja* $\Rightarrow$
 informação, 16
 inicialização,
 flip-flop, 190
 Instrução,
 busca, *veja* busca
 instrução, 12, 338
 add, 339, 365, 373, 374
 addi, 341, 366
 beq, 348, 369, 376
 busca, *veja* busca
 com imediato, 366
 decodificação, *veja* decodificação
 execução, *veja* execução
 formato, 340
 j, 348
 jal, 353, 370
 jr, 353, 370
 lógica e aritmética, 365
 ld, 343, 367, 374, 375
 nop, 348
 resultado, *veja* resultado
 slt, 340
 st, 343, 368, 376
 interface,
 de rede, 13
 de vídeo, 12
 inversor, 96
 tempo de propagação, 109
 involução, 31, 61
 IP, 338, 362, 364
 isolante, 85

J

Joule, 112
jump, *veja* salto incondicional

L

label, 348, 349
latch, *veja* basculo
latch FF, *veja flip flop*, destino
launch FF, *veja flip flop*, fonte
 Lei de Joule, 112
 Lei de Kirchhoff, 106
 Lei de Ohm, 105, 106
 LIFO, 264
 ligação,
 barramento, 140
 em paralelo, 93, 99
 em série, 93, 99
 linguagem,
 assembly, *veja* ling. de montagem
 C, *veja* C

de montagem, 337–361
 Pascal, *veja* Pascal
 VHDL, *veja* VHDL
 Z, 27
 linha de endereçamento, 78
 literal, 38
 logaritmo, 43
 lógica restauradora, 125

M

MADD, 197
 Mapa de Karnaugh, 49, 124
 máquina de estados, 229, 232
 Mealy, 234, 241, 252, 269
 Moore, 233, 240, 252, 265
 projeto, 236
 Máquina de Mealy, *veja* máq. de estados
 Máquina de Moore, *veja* máq. de estados
 máscara, 32
 máximo e mínimo, 31
 maxtermo, 46
 MD, 364
 Mealy, *veja* máq. de estados
 memória, 179
 atualização, 77
 bit, 181
 de controle, 372
 de dados, 364
 de instruções, 363
 de vídeo, 13
 decodificador de linha, 80
 endereço, 77
 FLASH, 133
 matriz, 129, 132, 134
 multiplexador de coluna, 80
 primária, 13
 RAM, 81, 134, 263
 ROM, 78, 126, 241
 secundária, 13
 memória dinâmica, *veja* DRAM
 memória estática, *veja* SRAM
 metaestabilidade, 181, 185, 188, 191
 defesa contra artes das trevas, 188
 metodologia de sincronização, 373
 MI, 363
 Mico XII, 337
 assembly, 337–361
 processador, 362–377
 temporização, 373–376
 microcontrolador, 241–249
 microrrotina, 249
 mintermo, 45, 124, 126
 MIPS32, 337, 359
 modelo,
 funcional, 44
 porta lógica, 96
 temporização, 115
 módulo de contagem, 217
 módulo, *veja* %, *mod*

Moore, *veja* máq. de estados
 MOSFET, 91
 multiplexador, 61, 66–70, 80, 101, 117, 119, 123–
 124, 126, 141, 142, 362, 372
 de coluna, 132, 136
 multiplicação, 170–176, 258
 acumulação de produtos parciais, 171–176
 multiplicador,
 somas repetidas, 258, 269
multiply-add, *veja* MADD

N

número,
 de Euler, 24
 negação, *veja* $\neg$
 nível lógico,
 0 e 1, 28
 indeterminado, 28, 109, 116, 141
 terceiro estado, 140
 nó, 96
 nomes simbólicos, 346
non sequitur, 37
 not, *veja* $\neg$
 número primo, 53

O

octal, 18
 onda quadrada, 185, 206
opcode, 340, 364, 372
 secundário, 376
 operação,
 binária, 29
 bit a bit, 32
 infixada, 42
 MADD, 197
 prefixada, 47
 unária, 29
 operação apropriada, 194
 operações sobre bits, 29–33
 operador,
 binário, 29
 lógico, 36
 unário, 29
operation code, *veja* *opcode*
 or, *veja* $\vee$
or array, 129
 ou exclusivo, *veja* $\oplus$
 ou inclusivo, *veja* $\vee$
output enable, 264
overflow, 148–150, 155, 163–164, 171, 176

P

paridade,
 ímpar, 49
 par, 49
 Pascal, 341–360
 funções, 353–360
 if, 349
 if-then-else, 349
 while, 350

período mínimo, *veja* relógio
 pilha, 264–266, 269, 355–360
pipelining, *veja* segmentação, 199
 piso, *veja* [v]
 ponto fixo, 151–152
 ponto flutuante, 70
pop, 264
 porta,
 de escrita, 261
 de leitura, 261
 porta lógica, 65
 and, 59
 carga, *veja* fan-out
 de transmissão, 141, 181
 nand, 60, 99
 nor, 60, 98
 not, 59, 96
 or, 59
 xor, 60, 65, 188
 portas complexas, 100
 potência, 112–115
 dinâmica, 114
 estática, 115
 potenciação, 43
 precedência, 30
 precisão,
 representação, 23
preset, 190
 prioridade,
 decodificador, 74
 processador, 12, 360–377
 produtivo, 33
 produto de somas, 46
 programa de testes, *veja* VHDL, *testbench*
 PROM, 133
 propriedades, operações em $\mathbb{B}$, 31
 protocolo,
 de sincronização, 253, 269
 invocação de função, 354
 prova de equivalência, 40–41
 próximo estado, 193, 208
 pseudoinstrução, 348
pull-down, 96
pull-up, 96, 126, 141
 pulso, 122, 123, 182, 191, 207, 215, 231, 257
 espúrio, *veja* transitório
push, 264

R

raiz quadrada, 270
 RAM, 12, 77, 81, 134–139
 célula, 81
 dinâmica, 135
Random Access Memory, *veja* RAM
 RAS, 134
Read Only Memory, *veja* ROM
 realimentação, 81
 receptor, 87
 rede, 96

redução, 33, 350
refresh, 137, 139
Register Transfer Language, veja RTL
 registrador, 191, 228, 246, 260, 338, 362
 carga paralela, 192
 de segmento, 197
 destino, 372
 ra, 359
 simples, 192
 sp, 356, 359
 registrador de deslocamento, 224–228
 modelo VHDL, veja VHDL, registrador
 paralelo-série, 225
 série-paralelo, 224
 universal, 227
 registrador de estado, 193
 registro de ativação, 357
 relógio, 183, 185, 206–221, 374
 bordas,
 ascendente, 186
 descendente, 186
 ciclo de trabalho, 221
 enviesado, veja skew
 frequência máxima, 195
 multi-fase, 221
 período mínimo, 195, 374–376
 representação,
 abstrata, 28
 binária, 20
 complemento de dois, 147
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 ponto fixo, 151
 posicional, 17
 precisão, 23
reset, 182, 190, 192
 resistência, 87, 100, 105
 resultado, 339, 365, 367, 370
 retorno de função, 370
return address, 353
 ROM, 12, 77–80, 126–133
 rotação, 159, 164
Row Address Strobe, veja RAS
 RTL, 199

S

salto incondicional, 348
 salto para função, 370
 segmentação, 197
 segmento, 343
 dados, 342, 343, 352
 texto, 342, 343
 seletor, 72
 semântica, 17
 semicondutor, 85
 tipo N, 87
 tipo P, 87

set, 182, 190
setup time, 185, 193–199, 242, 374
 folga, 195, 200, 201
 silogismo, 37
 simplificação de expressões, 38–40
 sinal, 27, 42
 analógico, 27
 digital, 27, 28
 fraco, 125, 126, 138, 141
 intensidade, 90, 139
 restaurado, 125, 139
 síntese, veja VHDL, síntese
skew, 199–203
Solid State Disk, veja SSD
 soma, veja somador
 soma de produtos, 45, 51, 126
 completa, 45
 somador, 145, 152–153, 362
 adiantamento de vai-um, 165, 228
 cadeia, 153
 completo, 104, 152
 overflow, 163
 parcial, 103
 seleção de vai-um, 169
 serial, 227
 temporização, 197
 teste, 176
 somatório, 33
 spice, 28
 SRAM, 138–139
 SSD, 14
stack frame, 357
status, 162
 subtração, 153–155
 superfície equipotencial, 96, 110

T

tabela,
 de controle, 365, 372
 de excitação dos FFs, 190
 tabela verdade, 33–35, 45
 tamanho, veja $|N|$
 tempo,
 de contaminação, 115, 118–121, 181, 189
 de estabilização, 185, 189
 de manutenção, 185, 189
 de propagação, 57–58, 61, 64–65, 73, 77, 83, 102,
 104, 109, 115–117, 189, 203, 218, 219
 discretizado, 183, 185, 189, 194, 218
 temporização, 104–126, 373–376
 tensão, 105
 Teorema,
 Absorção, 49
 DeMorgan, 32, 41, 48, 54, 60, 61, 94, 98
 Dualidade, 99
 Simplificação, 49
 terceiro estado, 140–141
testbench, veja VHDL, *testbench*
 teste,

- cobertura, 177
- de corretude, 176
- teto, *veja* [r]
- three-state*, *veja* terceiro estado
- tipo,
 - de sinal, 42
 - função, 29
- Tipo I, *veja* formato
- Tipo J, *veja* formato
- Tipo R, *veja* formato
- transferência entre registradores, *veja* RTL
- transistor, 87–91, 95–96
 - CMOS, 95
 - corte, 113
 - gate, 88
 - saturação, 113
 - sinal fraco, 90
 - tipo N, 88
 - tipo P, 89
- Transistor-Transistor Logic*, *veja* TTL
- transitório, 121–123, 182, 183, 235
- transmissão,
 - serial, 226
- transmission gate*, *veja* porta de transmissão
- TTL,
 - 74148, 75
 - 74163, 214
 - 74374, 228
- tupla, *veja* $\langle \rangle$
 - elemento, 32
 - largura, 43

U

- ULA, 161–164, 178, 260, 338, 362, 366, 372
 - status*, 162
- um FF por estado, 249–252
- Unidade de Lógica e Aritmética, *veja* ULA

V

- valor da função, 30
- VCC, 93
- vetor de bits, *veja* $\langle \rangle$, 32
 - largura, 43
- VHDL, 191, 339, 342
 - design unit*, *veja* VHDL, unidade de projeto
 - síntese, 203
 - std_logic*, 140
 - tipos, 42

W

- Watt, 112
- write back*, *veja* resultado

X

- xor, *veja* $\oplus$

Z

- Z, linguagem, 27

Capítulo 12

Duas Implementações para MIPS32r2

Before you design, you must understand the components.

Yale Patt.

Esta seção¹ descreve duas implementações do conjunto de instruções MIPS32, uma que é simples e sem paralelismo na execução, e outra que tira proveito de paralelismo e portanto tem melhor desempenho. A implementação com paralelismo introduz dois artefatos que podem reduzir a eficiência desejada. Para mais detalhes quanto ao *hardware*, veja [PH14]; para detalhes quanto ao *software*, veja [Swe07].

Evidentemente, a implementação do conjunto completo de instruções não é discutida por ser demasiado complexa; o que se deseja é mostrar os princípios de funcionamento do processador. A implementação de algumas poucas instruções representativas é discutida; outras são indicadas nos exercícios. O código VHDL de uma implementação completa do conjunto de instruções MIPS32r2 pode ser encontrado em [Hex15].

12.1 Um subconjunto das instruções MIPS32r2

O conjunto de instruções a ser estudado aqui é mostrado na Tabela 12.1. Cada uma delas representa uma classe de instruções e a implementação de uma classe completa depende de pequenas variações nos circuitos que são discutidos no que se segue.

Convenção: o caractere ‘;’ significa execução sequencial; o caractere ‘,’ significa execução em paralelo dos eventos à esquerda e à direita da vírgula; ‘R(s)’ é a sintaxe de VHDL para indexar um vetor, seja um vetor de bits seja um vetor de bytes ou palavras; ‘&’ é o operador de concatenação de VHDL, e ‘←’ representa atribuição. A operação de alguns dispositivos é definida por código VHDL.

Além de uma Unidade de Lógica e Aritmética (ULA), o processador deve conter um bloco de registradores (R) para manter os resultados intermediários da computação. Um registrador que é usualmente chamado de *contador de programa* (*Program Counter* ou PC) indica qual é a próxima instrução a ser buscada da memória de instruções e então executada. Dependendo do fabricante, esse registrador pode ser chamado de *Instruction Pointer*. Empregaremos o nome mais usual que é PC.

¹© Roberto André Hexsel, 2012-2021.

Tabela 12.1: Subconjunto das instruções do MIPS

INSTRUÇÃO	DESCRIÇÃO
addu rd, rs, rt	$R(rd) \leftarrow R(rs) + R(rt)$, $PC \leftarrow PC+4$
ori rt, rs, im16	$R(rt) \leftarrow R(rs) \text{ OR } \text{extZero}(\text{im16})$, $PC \leftarrow PC+4$
lw rt, des16(rs)	$R(rt) \leftarrow M(R(rs) + \text{extSinal}(\text{des16}))$, $PC \leftarrow PC+4$
sw rt, des16(rs)	$M(R(rs) + \text{extSinal}(\text{des16})) \leftarrow R(rt)$, $PC \leftarrow PC+4$
beq rs, rt, des16	if $(R(rs) = R(rt))$ $PC \leftarrow PC+4 + \text{extSinal}(\text{des16}) \& 00$ else $PC \leftarrow PC+4$
j ender26	$PC \leftarrow PC(31:28) \& \text{ender26} \& 00$

Quais são os recursos necessários para implementar as instruções da Tabela 12.1?

Para quase todas elas é necessário um somador para incrementar o PC de 4.

Para quase todas elas é necessário um bloco de registradores que permita a leitura de um ou dois registradores e a escrita em um registrador.

Para as instruções de lógica e aritmética é necessária uma unidade de lógica e aritmética que implemente as operações usuais, tais como soma, subtração, *and*, *or*, *xor* e *not*.

Para a instrução **ori** é necessário um circuito que efetue a disjunção bit a bit – que existe na ULA – e um circuito que estenda com zeros um número de 16 bits para um número com 32 bits.

Para as instruções **lw** e **sw** é necessária uma memória que permita escritas e leituras, um somador – da ULA – e um circuito que estenda o sinal de uma constante de 16 bits, representada em complemento de dois, para 32 bits.

Para **beq** é necessário um comparador de igualdade, para comparar dois números de 32 bits.

Embora não esteja indicado na Tabela 12.1, todas as instruções devem ser buscadas da memória de instruções, para que possam ser decodificadas e então executadas. A memória de instruções é indexada pelo PC. A cada instrução que não seja um desvio ou salto, o PC é incrementado de 4; em desvios e saltos, o valor carregado no PC depende da instrução.

A Figura 12.1 mostra os circuitos combinacionais necessários para a implementação do conjunto de instruções. A *Unidade de Lógica e Aritmética* (ULA) tem como entradas dois operandos de 32 bits e produz um resultado de 32 bits. O valor na saída depende da entrada *func*, que define qual a função a ser aplicada aos operandos, que é uma dentre soma, subtração, conjunção, disjunção, ou-exclusivo ou *nor*. O *somador* efetua a soma em 32 bits de seus operandos. O *multiplexador* (*mux*) apresenta em sua saída uma dentre as suas entradas, selecionadas pela entrada *sel*.

A Figura 12.1 também mostra os elementos de estado necessários. Os mais simples são *registradores*, que nada mais são do que vários *flip-flops* tipo D que compartilham os sinais de relógio (*clk*) e habilitação (*habEscr*). Quando *habEscr*=1, na borda ascendente do relógio o valor em D é capturado e mantido em Q. Se *habEscr*=0, então o valor memorizado anteriormente não se altera. Estas duas situações são mostradas nos diagramas de tempo. O PC é um registrador cujo sinal de habilitação está sempre ativo e portanto é atualizado em todos os ciclos do relógio.

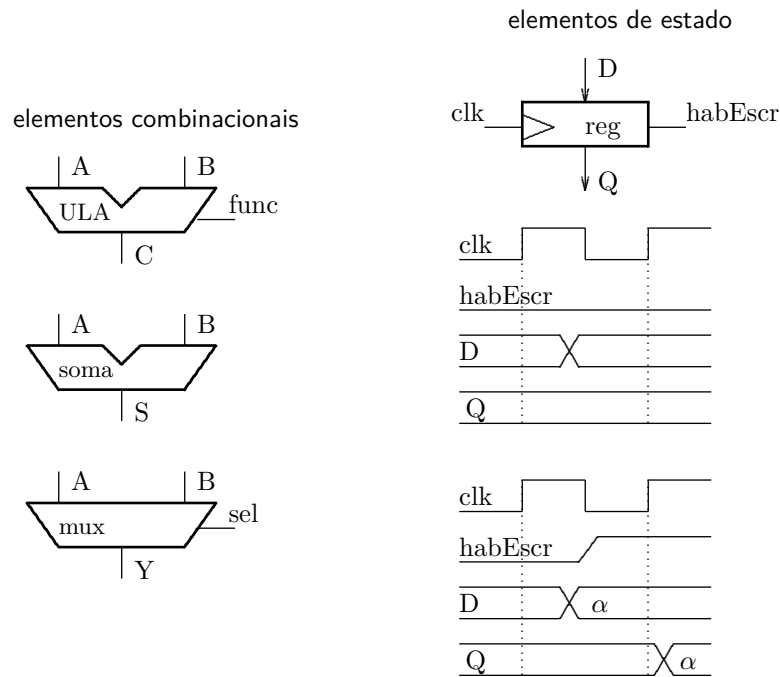


Figura 12.1: Elementos combinacionais e elementos de estado.

O *bloco de registradores* (*register bank*) contém 32 registradores de 32 bits cada. Dois registradores podem ser acessados simultaneamente para leitura – são duas portas de leitura, A e B – e um registrador pode ser atualizado na borda do relógio, se o sinal de habilitação está ativo – que é a porta de escrita C. A Figura 12.2 mostra a interface do banco de registradores e das memórias. O registrador 0 é *sempre* zero, e escritas neste registrador são ignoradas.

Os registradores cujos conteúdos são mostrados nas portas de leitura são selecionados por dois endereços de 5 bits. As portas de leitura se comportam como circuitos combinacionais e podem ser consideradas como dois vetores de inteiros independentes:

```
x <= A(a);
y <= B(b);
```

sendo *a* e *b* dois números representados em 5 bits ($\log(32) = 5$). A porta de escrita se comporta como um registrador, e o novo valor só é atualizado na borda do relógio se o sinal de habilitação (*hab*=1) está ativo:

```
if ( rising_edge( clk ) and hab=1 ) then C(c) <= z end if ;
```

A *memória de instruções* (MI) se comporta como um circuito combinacional – uma vez que o endereço estabiliza, passado o tempo de acesso à memória, a instrução fica estável na porta I:

```
instr <= MI(ender);
```

A *memória de dados* (MD) tem um comportamento similar ao banco de registradores. A porta de leitura L se comporta como um vetor e a leitura do conteúdo endereçado é combinacional – uma vez que o endereço estabiliza e decorrido o tempo de acesso, o dado fica disponível na porta L:

```
dado <= MD(ender);
```

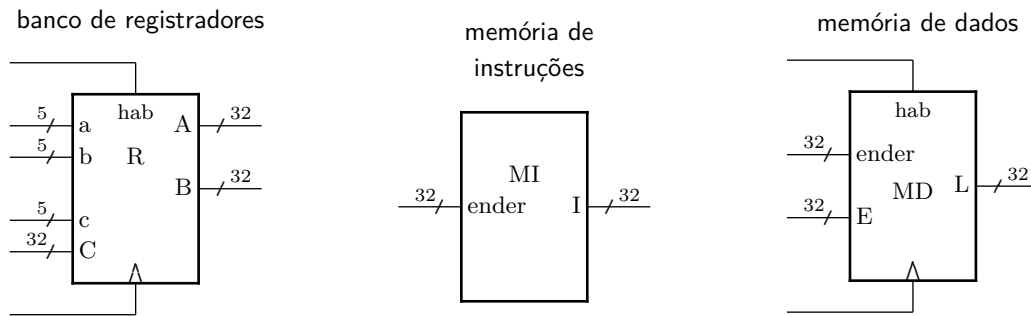


Figura 12.2: Banco de registradores, memória de instruções e de dados.

A atualização da palavra indexada pelo endereço só é efetivada na borda do relógio, se a escrita estiver habilitada ($hab=1$):

if ($rising\_edge(clk)$ **and** $hab=1$) **then** $MD(ender) \leftarrow dado$ **end if**;

12.1.1 Codificação das instruções

Antes que possamos falar da implementação do processador é necessário examinarmos a codificação das instruções do MIPS32. Todas as instruções tem 32 bits e são codificadas em um dos três formatos: Tipo R, Tipo I e Tipo J, mostrados na Figura 12.3.

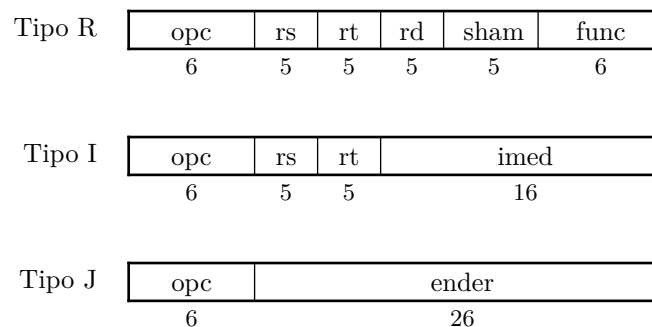


Figura 12.3: Formatos das instruções.

Nos três formatos, o campo *opcode* (opc), abreviatura para *operation code*, é o ‘nome’ da instrução, e em todas elas este campo tem 6 bits. Os campos rs (*register source*), rt (*register target*) e rd (*register destination*) endereçam os registradores com os operandos da instrução. O campo sham (*shift amount*) é um campo de 5 bits que indica o número de posições por deslocar nas instruções de deslocamento. O campo func (*function*) determina a operação da ULA nas instruções de lógica e aritmética quando $opc=0$. O campo imed (*immediate*) é uma constante de 16 bits, que é representada em complemento de dois sempre que isso faça sentido. Finalmente, ender é um endereço de 26 bits.

Na Tabela 12.1, a instrução **addu** é do Tipo R porque a instrução tem seus três operandos em registradores; as instruções **ori**, **lw**, **sw** e **beq** são do tipo I porque todas tem dois registradores como operandos e o terceiro operando é uma constante que faz parte da instrução. As instruções do tipo J são as instruções de saltos, **j** e **jal**.

Esta codificação tem uma característica muito importante que é a sua *regularidade*: todas as instruções tem o mesmo tamanho; todas as instruções tem o *opcode* no mesmo lugar e com

o mesmo tamanho (6 bits), a maioria das instruções tem seus operandos identificados nos campos *rs* e *rt*; a maioria das constantes tem 16 bits (*imed*), e pode ser interpretada como uma constante lógica ou como um inteiro.

Os projetistas do conjunto de instruções reservaram 6 bits para $2^6 = 64$ *opcodes*. Este número é suficiente para codificar todas as instruções do processador? A resposta é *não*, e para aumentar o número de *opcodes* foi usado um truque: quando *opc*=0, a função da ULA é determinada pelo campo *func*, também com 6 bits. Assim, o total de *opcodes* disponíveis é $(2^6 - 1) + 2^6 = 127$.

Como a atenta leitora pode verificar em [MIP05b], a realidade é ligeiramente mais complexa do que o exposto aqui. Por ora, nos interessa mais estudar os princípios de projeto do que as poucas idiosincrasias de um conjunto de instruções real e bem projetado.

A decodificação das instruções é extremamente simples, graças à codificação simples: basta indexar uma tabela de 64 elementos que contém todos os sinais de controle do processador. O *opcode* indexa a tabela e os sinais de controle de todos os componentes do processador são definidos pelos campos dessa tabela – logo veremos alguns exemplos.

12.2 Implementação com relógio de ciclo longo

Vamos, pois, à implementação das instruções da Tabela 12.1. No que segue, os diagramas mostrados estão incompletos para simplificar a explanação. Na Seção 12.2.8 o circuito completo do processador é apresentado.

12.2.1 Busca e decodificação de instruções

A memória de instruções (MI) mantém o código binário do programa que está sendo executado. No momento não nos interessa como o programa foi gravado nesta memória; mais adiante estudaremos a carga de programas para execução. A MI é indexada pelo *program counter* (PC) e este registrador é incrementado de 4 a cada ciclo do relógio. A Figura 12.4 mostra um diagrama com o circuito que efetua a busca e a decodificação das instruções. O PC é incrementado de 4 em 4 porque a memória é indexada byte a byte mas as instruções ocupam uma palavra de 4 bytes.

Uma vez que a saída do PC estabilize após a borda do relógio, e decorrido o tempo de acesso da memória, uma nova instrução fica disponível para ser decodificada e executada. A *decodificação* da instrução ocorre com a indexação da tabela de controle com o *opcode*, o que ativa todos os sinais de controle para a instrução recém buscada.

A largura dos sinais é indicada no diagrama na Figura 12.4 com um traço diagonal e pelo número de bits daquele sinal. O *opcode* tem 6 bits de largura, a instrução *I* e o próximo PC tem 32 bits. As setas indicam a ‘direção’ dos sinais – de saídas para entradas.

O circuito de busca não é mostrado nos próximos diagramas, que descrevem os caminhos de dados das instruções.

12.2.3 Operações de lógica e aritmética com imediato

As instruções de lógica e aritmética com um operando imediato usam a constante que é parte da instrução como um dos operandos, e o outro operando é o conteúdo de um registrador, apontado por *rs*. O registrador de destino é apontado por *rt*. O operando da porta B do banco de registradores não é utilizado nesta instrução e portanto as ligações de *b* e *B* não são mostradas. A Figura 12.6 mostra o circuito para executar as instruções com um operando imediato.

A constante é codificada em 16 bits, no campo *im16*, e ela deve ser estendida para 32 bits para que possa ser aplicada à entrada β da ULA. Se a operação é aritmética (**addi** ou **addiu**), então a constante é estendida para 32 bits pela replicação do seu bit de sinal. Se a operação é lógica (**ori** ou **andi**), a constante é estendida com 16 zeros.

`ori rt, rs, im16    # R(rt)  $\leftarrow$  R(rs) OR extZero(im16)`

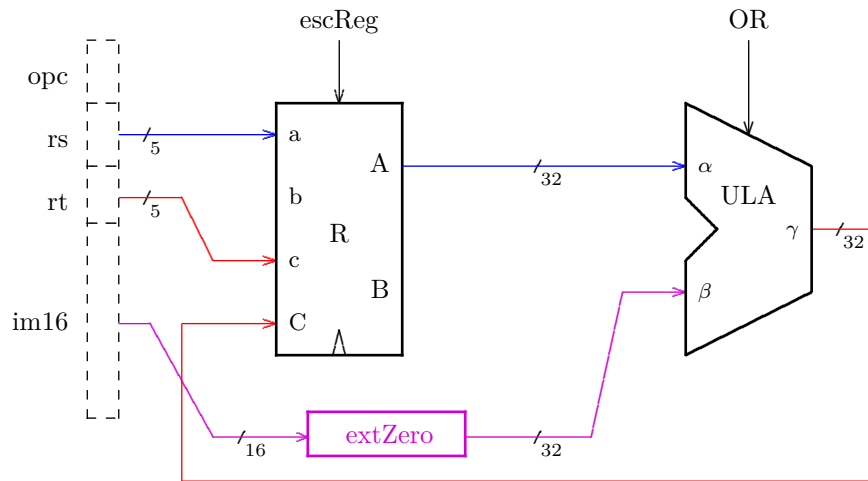


Figura 12.6: Operações de lógica e aritmética com imediato – **ori**.

O operando apontado por *rs* é apresentado à entrada α da ULA; o segundo operando (*im16*) é estendido para 32 bits e apresentado à entrada β da ULA. A operação da ULA é definida como OR e o resultado é armazenado no registrador apontado por *rt*, na borda de subida do relógio, e *escReg* deve ser 1.

As operações aritméticas são similares, exceto que a constante é estendida com sinal ao invés de com zeros.

12.2.4 Operação de acesso à memória – leitura

A instrução **lw** (*load-word*) copia, para o registrador apontado por *rt*, o conteúdo da palavra indexada pela soma de um deslocamento de 16 bits (*des16*) com o conteúdo do registrador apontado por *rs*. A soma da constante com um registrador é chamada de *endereço efetivo*.

A constante é estendida com sinal para permitir deslocamentos positivos e negativos com relação ao endereço apontado por *R(rs)*. A saída da ULA, que é o endereço efetivo, é aplicada à entrada de endereços da memória de dados. Decorrido o tempo de acesso à memória, a porta de saída *L* contém a cópia do valor lido da memória. Na borda do relógio esse valor é armazenado no registrador apontado por *rt*.

A Figura 12.7 mostra o diagrama do processador com as ligações para a instrução **lw**.

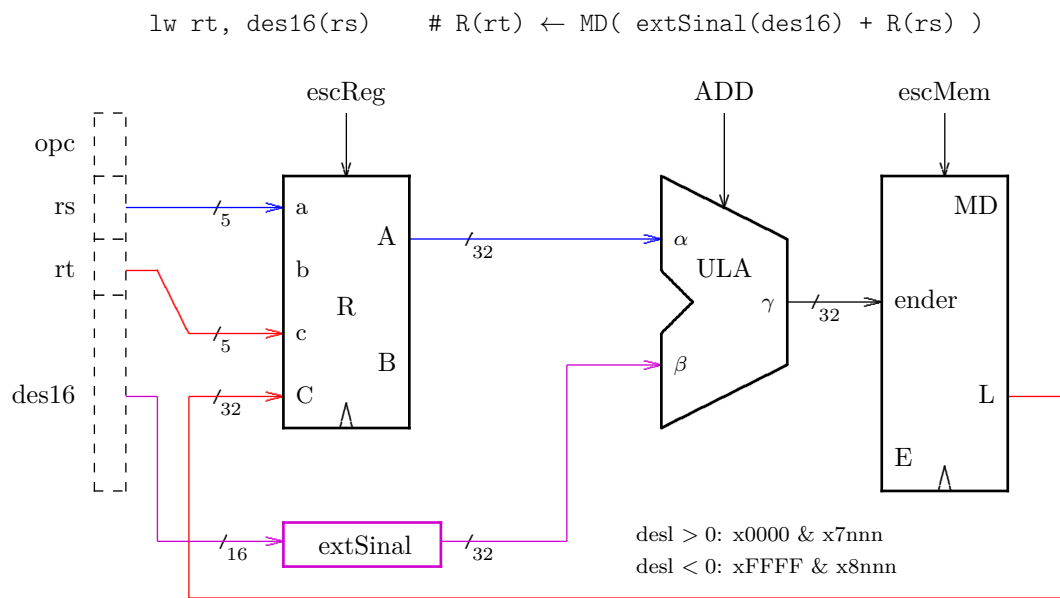


Figura 12.7: Operação de leitura da memória – lw.

12.2.6 Desvio condicional

Num desvio condicional (**beq**), o endereço da próxima instrução a ser buscada depende de uma comparação de igualdade. Para tanto, os dois valores por comparar são subtraídos e se o resultado for zero, então eles são iguais. A Figura 12.9 mostra o diagrama do processador com as ligações para a instrução **beq**. A operação na ULA é definida como uma subtração e o sinal **iguais** indica o resultado da comparação.

```
beq rs, rt, des16    # if (R(rs)=R(rt)) PC <- PC+4 + extSinal(des16)*4
```

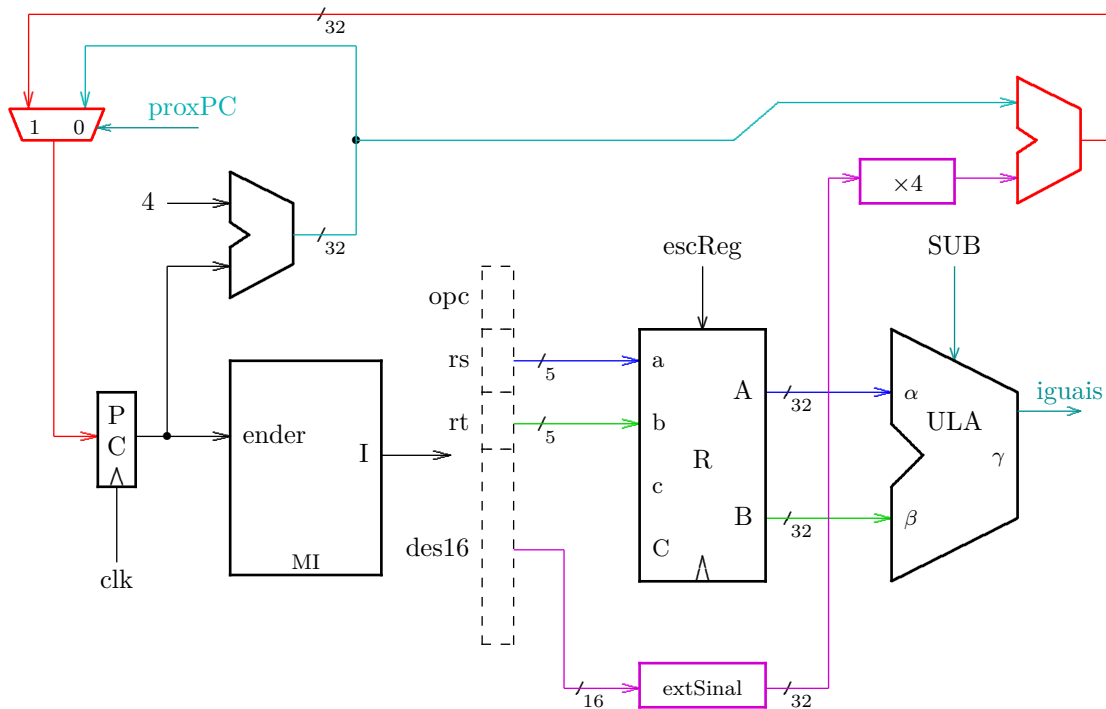


Figura 12.9: Desvio condicional – beq.

O *endereço de destino* de um desvio é obtido pela soma da constante des16, estendida para 32 bits com sinal, mais o PC incrementado de quatro. A constante estendida é multiplicada por quatro antes de ser somada ao PC+4. Isso significa que o deslocamento nos desvios é o número de instruções por pular, e não o número de bytes. A multiplicação por quatro é obtida pelo deslocamento de duas posições para a esquerda.

Se **iguais=1** então o endereço de destino é selecionado e aplicado à entrada do PC, e a instrução apontada por aquele endereço será buscada no próximo ciclo. Do contrário, **iguais=0**, PC+4 é aplicado à entrada do PC e a instrução após o **beq** será buscada.

12.2.7 Salto incondicional

Num salto incondicional (**j** para *jump*), o endereço da próxima instrução a ser buscada está codificado na própria instrução. Esta é do Tipo J, com 6 bits para o *opcode* e 26 bits para o endereço de destino.

A especificação do conjunto de instruções determina que as instruções estejam em endereços alinhados, o que significa que os dois bits menos significativos do endereço de destino sejam *sempre* 00₂. Assim, os 26 bits da instrução concatenados com os dois 0s compõem um endereço de 28 bits. A faixa de endereços alcançáveis com 28 bits é $2^{28} = 256$ Mbytes, ou 64 M instruções. O folclore nos diz que, na média, um comando em C é implementado com 10 instruções, e portanto uma instrução de salto pode cobrir até 6,4 milhões de linhas de código fonte. Convenhamos que uma ‘perna’ de `if()` com 6 milhões de instruções é assaz incomum, mas não é proibida pela definição da linguagem C e portanto o projetista do *hardware* deve garantir que um salto mais longo que 64 M instruções possa ser usado por programadores néscios.

Para permitir saltos para qualquer distância, os projetistas do MIPS decidiram por usar os 4 bits mais significativos do PC incrementado (PCinc) para completar os 32 bits do endereço. Assim, o endereço de destino de um salto incondicional é

$$\text{PCinc}(31:28) \& \text{ender26} \& 00$$

com a concatenação representada por $\&$. Em *hardware*, a concatenação é a mera justaposição dos sinais, e no circuito da Figura 12.10, isso é representado pela “integral quadrada” ($\int$), sendo que a parte ‘cortada’ representa o lado mais significativo da justaposição.

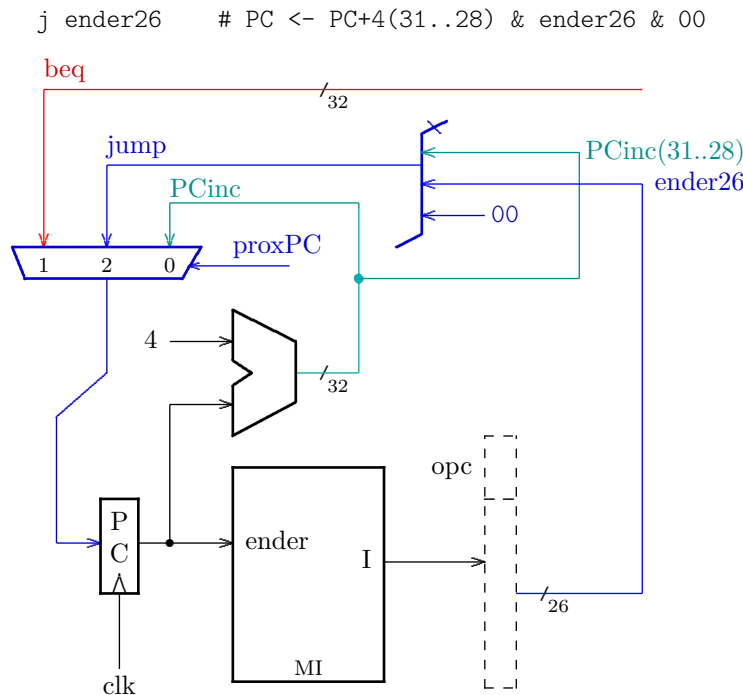


Figura 12.10: Salto incondicional – j.

12.2.9 Tabela de controle

O projetista do processador deve elaborar uma *tabela de controle* com todos os sinais que controlam os componentes do processador. A Figura 12.12 mostra o circuito de busca e decodificação com os sinais de controle das unidades funcionais do processador – os nomes estão abreviados no diagrama por causa do espaço. A tabela de controle é indexada pelo *opcode* da instrução recém buscada, e o conteúdo do elemento indexado pelo *opcode* ativa somente os sinais de controle apropriados à instrução por executar.

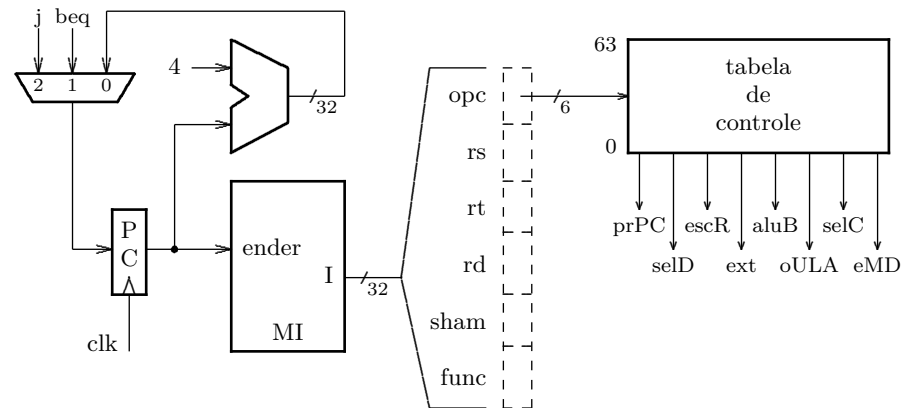


Figura 12.12: Circuito de busca e decodificação com tabela de controle.

A Tabela 12.2 mostra os sinais de controle que devem ser ativados para as instruções vistas nas seções anteriores. Sinais que prescindem de um valor determinado são indicados com um ‘X’ (*don’t care*).

Os sinais de controle que alteram o estado do processador, proxPC, escR e escMD, devem estar *sempre* determinados porque não há nenhuma instrução que *talvez* altere a memória, ou que *talvez* altere os registradores.

Tabela 12.2: Tabela de controle para as instruções.

INSTRUÇÃO	proxPC	selD	escR	ext	aluB	operULA	selC	escMD
addu	0	0	1	X	0	fun	0	0
ori	0	1	1	zero	1	OR	0	0
lw	0	1	1	sinal	1	ADD	1	0
sw	0	X	0	sinal	1	ADD	X	1
beq	iguais	X	0	sinal	0	SUB	X	0
j	2	X	0	X	X	X	X	0

12.2.10 Metodologia de sincronização

A metodologia de sincronização para esta implementação do conjunto de instruções MIPS32 é chamada de *ciclo longo* porque o período do relógio é longo o bastante para executar uma instrução durante um único ciclo.

Temporização do ADD

A Figura 12.13 mostra o diagrama de tempo da execução de um **add**. Um diagrama simplificado do processador é mostrado na base da figura, com o bloco de registradores dividido em duas partes: no centro do diagrama está a parte na qual ocorre a leitura dos operandos, e na direita a parte em que ocorre a gravação do resultado.

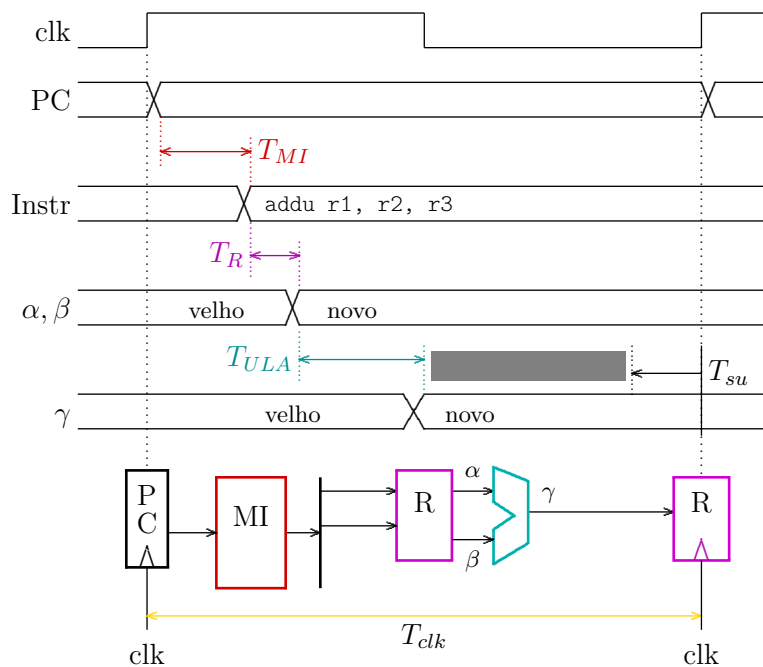


Figura 12.13: Diagrama de tempo de uma operação de ULA.

Decorrido o tempo de acesso à memória de instruções (T_{MI}), a nova instrução é decodificada paralelamente à leitura do banco de registradores (T_R). Com os operandos disponíveis, decorre o tempo de propagação através da ULA (T_{ULA}) até que o resultado esteja disponível no sinal γ . Há folga para respeitar a limitação de *setup* do banco de registradores (T_{su}), como mostra o intervalo entre a disponibilidade do novo valor em γ e a indicação do tempo de estabilização dos sinais (*setup*) – a folga é indicada pela barra de cor cinza.

Temporização do LW

A Figura 12.14 mostra o diagrama de tempo da execução de um **lw**. O período do relógio deve ser longo o bastante para acomodar o tempo de acesso à memória de instruções (T_{MI}), a leitura do banco de registradores (T_R), o tempo de propagação através da ULA (T_{ULA}), o tempo de acesso para leitura da memória de dados (T_{MD}), e respeitar a limitação de *setup* do banco de registradores (T_{su}), e a folga é mostrada pela barra de cor cinza.

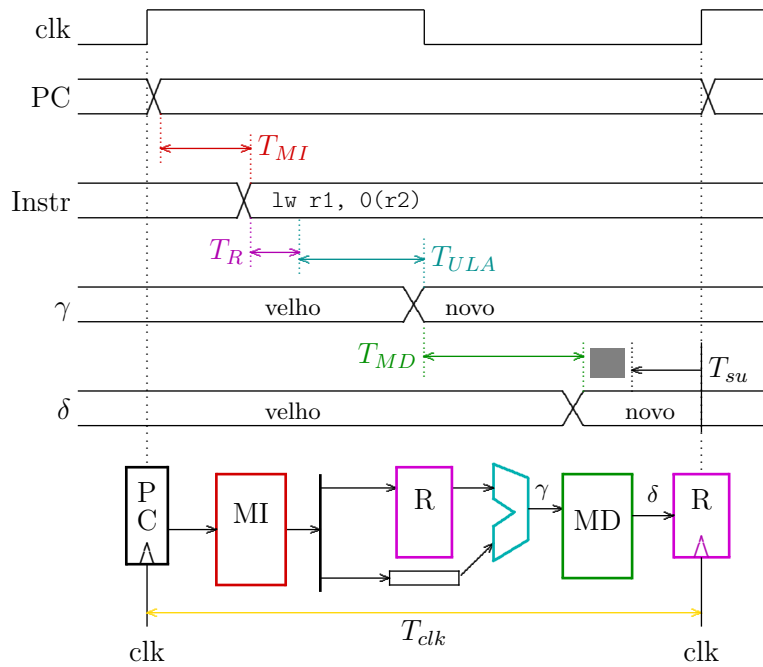


Figura 12.14: Diagrama de tempo de uma leitura em memória.

A instrução **lw** é a instrução mais demorada porque todas as unidades funcionais do processador são usadas em série, $MI \rightsquigarrow R \rightsquigarrow ULA \rightsquigarrow MD \rightsquigarrow R$, e portanto o ciclo do processador é limitado pela sua temporização.

As outras instruções não usam todas as unidades funcionais e há algum desperdício de tempo porque o ciclo do relógio deve ser fixado para atender ao pior caso, que é o **lw**. Por exemplo, na instrução **addu**, as unidades funcionais utilizadas são $MI \rightsquigarrow R \rightsquigarrow ULA \rightsquigarrow R$, não há acesso à memória de dados, e o ciclo desta instrução poderia ser encurtado de T_{MD} .

Alterar o ciclo do relógio em função da instrução não é realizável e por isso o período do relógio deve ser tal que acomode a temporização da instrução mais demorada. Na próxima seção veremos uma implementação mais eficiente do que a do ciclo longo.

Exemplo 12.1 Vejamos como deve ser modificado o processador para acrescentar o circuito de dados para a instrução **BRANCH-AND-LINK** definida abaixo. A instrução **bal** tem formato I.

bal *desl* # $R[31] \leftarrow PC+8$, $PC \leftarrow (PC+4) + (\text{ext}(\text{desl}) \ll 2)$

Esta instrução é uma mistura de uma instrução **jal** com um “desvio incondicional”: o endereço de ligação é armazenado no registrador *ra* como num **jal**: $R[31] \leftarrow PC+8$, e o endereço de destino é computado como numa instrução **beq**: $PC \leftarrow (PC+4) + (\text{ext}(\text{desl}) \ll 2)$.

Quais são as modificações necessárias ao circuito da Figura 12.11?

- é necessário acrescentar uma entrada ao multiplexador controlado pelo sinal *selD*, fixada em 31 para garantir que ao registrador 31 seja atribuído o valor de $PC+8$;
- deve-se acrescentar um somador para somar 4 ao valor $PC+4$;
- a saída deste somador deve ser ligada a uma entrada adicional no multiplexador controlado pelo sinal *selC*, para que $PC+8$ possa ser atribuído ao registrador 31; e
- o sinal $(PC+4) + (\text{ext}(\text{desl}) \ll 2)$ é computado para os desvios, e esse sinal já está ligado à entrada 1 do multiplexador controlado por *proxPC*.

A Figura 12.15 mostra o diagrama de tempos da execução de **bal**. Assim que o valor no PC estabiliza, decorrido T_{PC} , o incremento de 4 fica disponível após o tempo de propagação do somador da esquerda (T_{add}). O segundo incremento de 4 fica estável após $T_{PC} + 2 \times T_{add}$. A faixa cinza indica que a limitação de *setup* do banco de registradores é respeitada com folga. Decorrido o tempo de acesso à memória de instruções, o endereço de destino fica estável após $T_{PC} + T_{MI} + T_{add}$.

A Tabela 12.3 mostra o estado dos sinais de controle durante a execução de **bal**. Para fins de comparação, a tabela também mostra os sinais da instrução **beq**. Os sinais que controlam multiplexadores que foram alargados necessitam de dois bits de controle ao invés de um. ◀

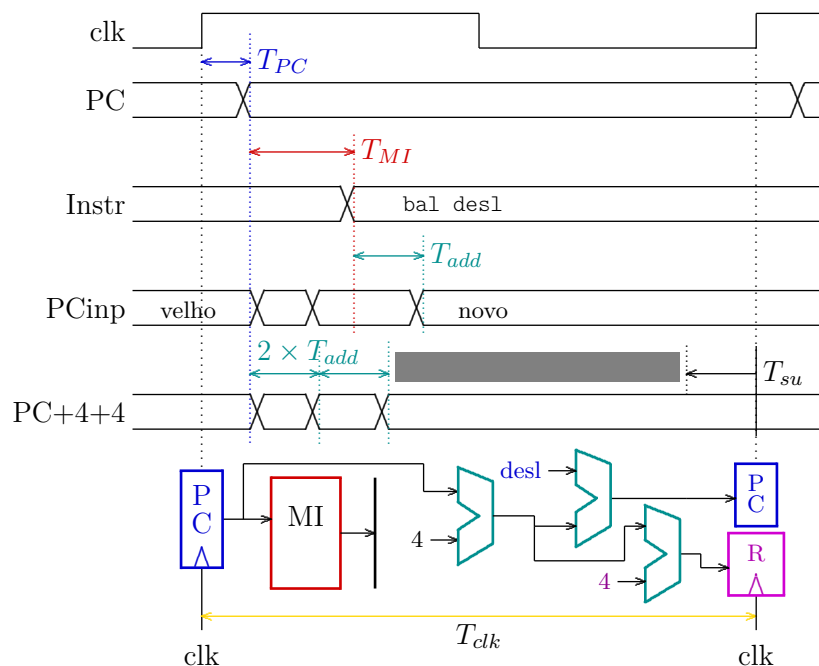


Figura 12.15: Diagrama de tempo da instrução **bal**.

Tabela 12.3: Tabela de controle para a instrução bal.

INSTRUÇÃO	proxPC	selD	escR	ext	aluB	operULA	selC	escMD
beq	iguais	X	0	sinal	0	SUB	X	0
bal	1	2	1	sinal	X	X	2	0

12.2.11 Exercícios

Ex. 12.1 Mostre como implementar as instruções abaixo. Para tanto, indique quaisquer modificações que sejam necessárias no processador da Figura 12.11 e mostre a tabela de sinais de controle ativos durante a execução da instrução. Nos comentários, a vírgula significa “execução simultânea”.

```

                                # jump-register
jr rs                          # PC <- R(rs)

                                # jump-and-link
jal ender26                    # R(31) <- PC+8 , PC <- PCinc & ender26 & 00

                                # jump-and-link-register
jalr rt,rs                    # R(rt) <- PC+8 , PC <- R(rs)

```

Espaço em branco proposital.

12.3 Implementação segmentada

A técnica de *segmentação* (*pipelining*) é usada para aumentar a taxa com que as instruções completam. Suponha que um determinado circuito combinacional tenha tempo de propagação T_p – veja a Figura 12.16. A entrada do circuito é alimentada por um registrador e sua saída é capturada por outro registrador. O período mínimo do relógio para este circuito é $T_{\min} \geq T_p + T_r + T_{su}$, que é a soma do tempo de propagação do circuito ($T_p = n$), do tempo de propagação do registrador de saída (T_r), mais o *setup* do registrador de saída (T_{su}). Para este circuito, a latência é $n + T_r + T_{su}$ segundos, e a vazão é $1/(n + T_r + T_{su})$ resultados por segundo, que pode ser aproximada por $1/n$, porque, no geral, $n \gg (T_r + T_{su})$.

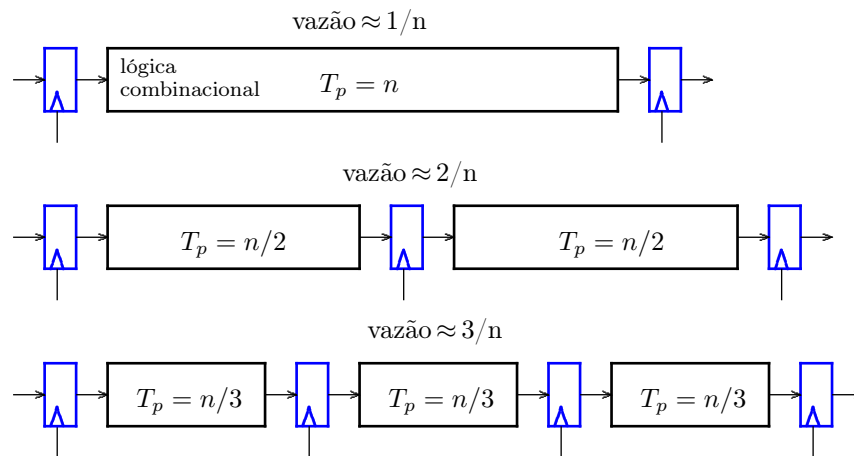


Figura 12.16: Circuito combinacional segmentado.

Se for possível particionar o circuito combinacional em duas metades iguais, o período do relógio pode ser reduzido para a metade se um registrador for inserido entre as duas partes, e neste caso $T'_{\min} \geq n/2 + T_r + T_{su}$, a vazão passa a ser $\approx 2/n$ resultados/s, e a latência permanece em $n + T_r + T_{su}$ s. Se o circuito pode ser particionado em três segmentos, $T''_{\min} \geq n/3 + T_r + T_{su}$, a vazão sobe para $\approx 3/n$ resultados/s enquanto a latência permanece a mesma.

Por “a latência permanece a mesma” entenda-se que o tempo para produzir um resultado não aumenta com a divisão do circuito em dois ou em três segmentos; o que aumenta é a taxa com que novos resultados são produzidos pelo circuito segmentado – a vazão aumenta de forma quase proporcional ao número de segmentos. O ‘quase’ decorre de que dificilmente os circuitos combinacionais possam ser particionados de forma a que o tempo de propagação dos dois segmentos seja exatamente igual, e esse desbalanceamento é a principal causa para que a vazão não seja exatamente proporcional ao número de segmentos.

A analogia mais próxima com a segmentação de um circuito é uma linha de montagem de automóveis, na qual o longo processo de montagem de um carro é subdividido em muitas tarefas simples. Um carro demora algo como 48 horas para ser produzido, mas uma linha de montagem moderna produz um veículo pronto e acabado a cada 15 minutos, dependendo do modelo e da fábrica.

12.3.1 Processador segmentado

A Figura 12.17 mostra o processador de ciclo longo já segmentado em cinco estágios, com registradores de segmento introduzidos nos locais apropriados. A execução de uma instrução é dividida em cinco estágios: (i) busca na memória de instrução e incremento do PC; (ii) decodificação e leitura dos operandos; (iii) execução na ULA; (iv) acesso à memória nos *loads* e *stores*; e (v) gravação do resultado.

Note que o banco de registradores aparece dividido no diagrama: no estágio de decodificação (**decod**), os registradores são lidos, enquanto no estágio de resultado (**result**) o registrador de destino é atualizado.

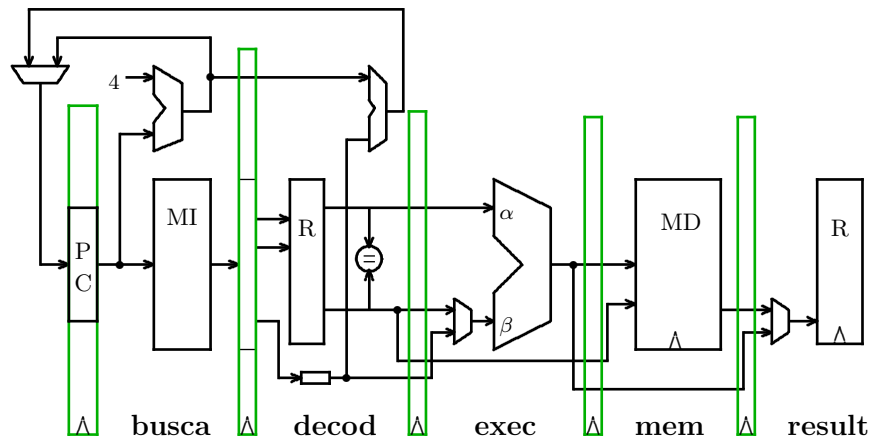


Figura 12.17: Processador segmentado em cinco estágios.

O período do relógio do processador é determinado pelo estágio mais demorado, e este é o estágio de acesso à memória. A latência de uma única instrução é aproximadamente a mesma daquela do processador de ciclo longo, enquanto a vazão, ou a taxa em que uma nova instrução é completada, é aproximadamente cinco vezes maior. A Figura 12.18 mostra dois diagramas de tempo, um do processador de ciclo longo executando duas instruções, e outro do processador segmentado executando três instruções.

Enquanto o processador de ciclo longo emprega 2 ciclos longos para executar um *load* seguido de um *store* – estes dois ciclos longos equivalem a 10 ciclos do processador segmentado, o processador segmentado executaria *seis* instruções neste mesmo intervalo. A primeira instrução completa no ciclo número 5, a segunda no ciclo 6, e a sexta no ciclo 10. Passado o intervalo para encher os segmentos com instruções, uma nova instrução completa a cada ciclo curto. O ganho potencial de desempenho, medido pela taxa de instruções executadas, é de um fator de cinco, que é o número de segmentos.

A Figura 12.19 mostra o quinto ciclo do diagrama de tempos da Figura 12.18: o valor lido da memória pelo **lw** é gravado no registrador de destino; o **sw** escreve na memória de dados; o **add** é computado na ULA; o **xor** é decodificado, r12 e r13 são obtidos do bloco de registradores; o **sub** é buscado da MI e o PC é incrementado de 4.

O ganho de desempenho advém do paralelismo no nível de instrução: cinco instruções executam ao mesmo tempo, cada uma delas num estágio distinto de execução. Os registradores de segmentação separam cada instrução da instrução que a precede, e da instrução que a sucede. A cada borda do relógio as instruções ficam mais perto de completar.

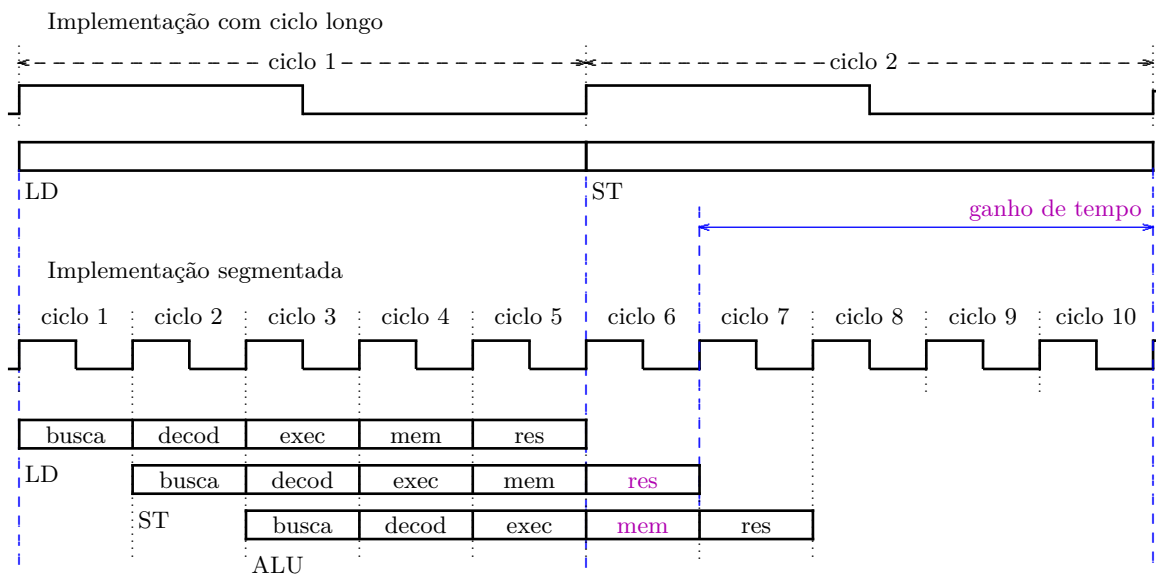


Figura 12.18: Diagrama de tempo do processador de ciclo longo e segmentado.

Compare o diagrama do processador na Figura 12.19 com aquele da Figura 12.11. No processador segmentado, o circuito necessário para as instruções de desvio está todo concentrado em **decod**. Isso é possível com a adição de um comparador de igualdade entre as portas de leitura do banco de registradores para gerar o sinal *iguais*, ao invés de empregar-se a ULA na comparação. Em breve veremos o motivo para resolver os desvios em **decod**.

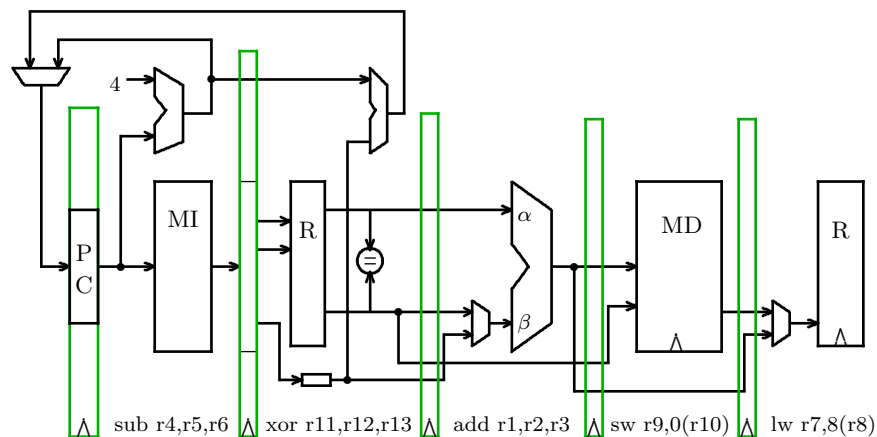


Figura 12.19: Cinco instruções em cinco estágios de execução.

Os registradores de segmento são *invisíveis* ao programador, porque estes registradores não fazem parte do estado da computação como definido pelo conjunto de instruções MIPS32r2. Os *registradores visíveis* são aqueles explicitamente usados como operandos das instruções.

Exemplo 12.2 Vejamos como se dá a execução de um `addu` no processador segmentado. A Figura 12.20 mostra a sequência de execução. O tempo decorre de cima para baixo, ao longo dos cinco ciclos de relógio necessários para completar uma instrução (C_0 a C_4), um ciclo em cada estágio. O registrador “de saída” dos estágios é mostrado em azul, e na próxima borda do relógio, o registrador de saída é carregado com os valores produzidos pelos circuitos combinacionais, alimentados pelo “registrador de entrada” do estágio.

Busca (C_0): o conteúdo do PC indexa a memória de instruções (MI), e no final do ciclo a instrução `addu` é carregada no registrador de saída; em paralelo ao acesso à memória de instruções, o PC é incrementado de 4.

Decod (C_1): a instrução é decodificada – a tabela de controle é indexada com o *opcode*, e o banco de registradores é acessado: $A \leftarrow R(rs)$ e $B \leftarrow R(rt)$. O endereço do registrador de destino ($rd=c$) é transferido para o próximo estágio.

Exec (C_2): às entradas α e β da ULA são atribuídos os conteúdos dos registradores $A = R(rs)$ e $B = R(rt)$, e o resultado é apresentado em γ . O endereço do registrador de destino ($c' \leftarrow c$) é transferido para o próximo estágio.

Mem (C_3): o resultado é transferido sem alteração para o estágio de resultado $\gamma' \leftarrow \gamma$. O endereço do registrador de destino ($c'' \leftarrow c'$) é transferido para o próximo estágio.

Result (C_4): o resultado da operação da ULA ($\gamma'' \leftarrow \gamma'$) é gravado no registrador apontado pelo registrador de destino rd ($c''' \leftarrow c''$): $R(c''') \leftarrow \gamma''$.

O resultado da operação na ULA (γ) é copiado, sem alteração, do registrador de entrada para o registrador de saída no estágio de memória ($\gamma' \leftarrow \gamma$), e este valor ($\gamma'' \leftarrow \gamma'$) é selecionado pelo multiplexador no estágio de resultado, e atribuído ao registrador de destino: $R(rd) \leftarrow (R(rs) + R(rt))$. $\triangleleft$

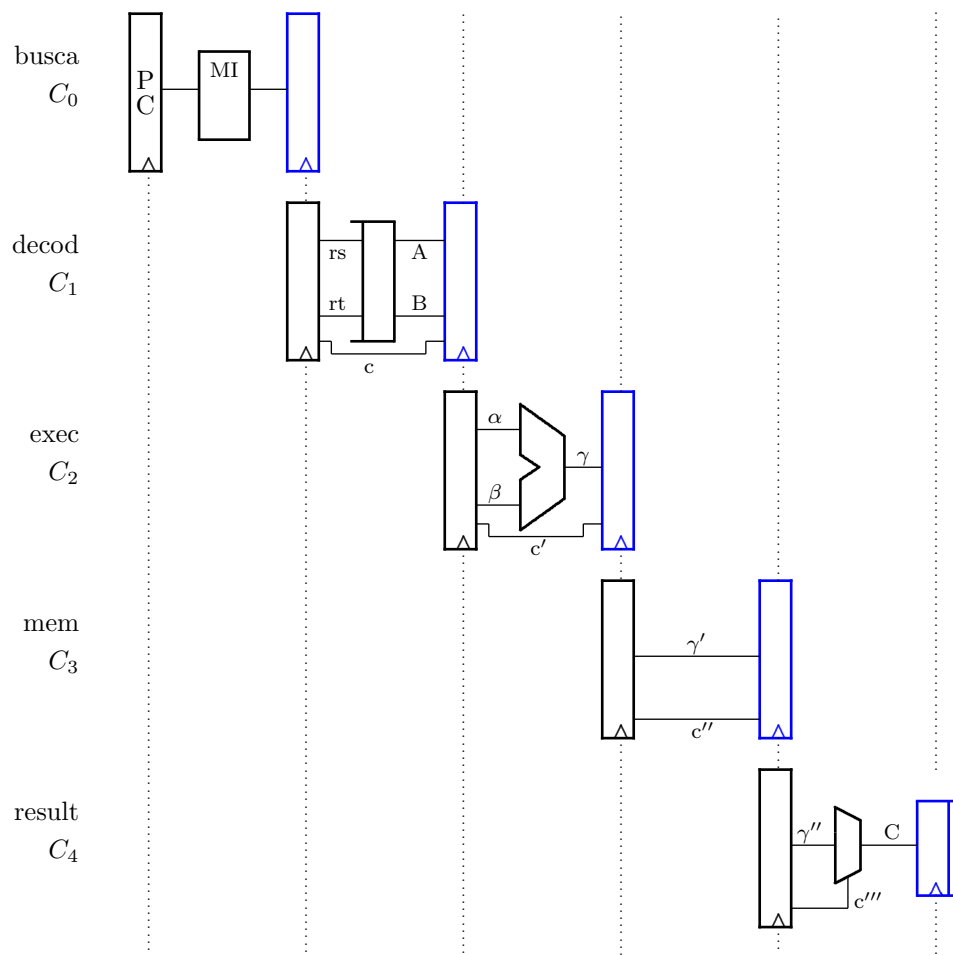


Figura 12.20: Progresso de um `addu` pelos segmentos.

Espaço em branco proposital.

Exemplo 12.3 Como um segundo exemplo, sigamos a execução de um `lw`. A Figura 12.21 mostra a sequência de execução.

Busca (C_0): o conteúdo do PC indexa a memória de instruções (MI), e no final do ciclo a instrução `lw` é carregada no registrador de saída; em paralelo ao acesso à memória de instruções, o PC é incrementado de 4.

Decod (C_1): a instrução é decodificada – a tabela de controle é indexada com o *opcode*, o banco de registradores é acessado ($A \leftarrow R(rs)$), e o deslocamento é estendido com o sinal ($D \leftarrow \text{extS}(\text{desl})$). O endereço do registrador de destino ($rt=c$) é transferido para o próximo estágio.

Exec (C_2): às entradas α e β da ULA são atribuídos os conteúdos dos registradores ($\alpha \leftarrow A$) e do deslocamento ($\beta \leftarrow D$), e o endereço efetivo é apresentado em γ . O endereço do registrador de destino ($c' \leftarrow c$) é transferido para o próximo estágio.

Mem (C_3): o endereço efetivo é aplicado à entrada de endereço da memória de dados (MD), e o conteúdo indexado é atribuído para o registrador de saída ($\delta \leftarrow MD(\gamma')$). O endereço do registrador de destino ($c'' \leftarrow c'$) é transferido para o próximo estágio.

Result (C_4): o valor obtido da memória ($\delta' \leftarrow \delta$) é gravado no registrador apontado pelo registrador de destino ($c''' \leftarrow c''$): $R(rt) \leftarrow MD(R(rs) + \text{extS}(\text{desl}))$. ◁

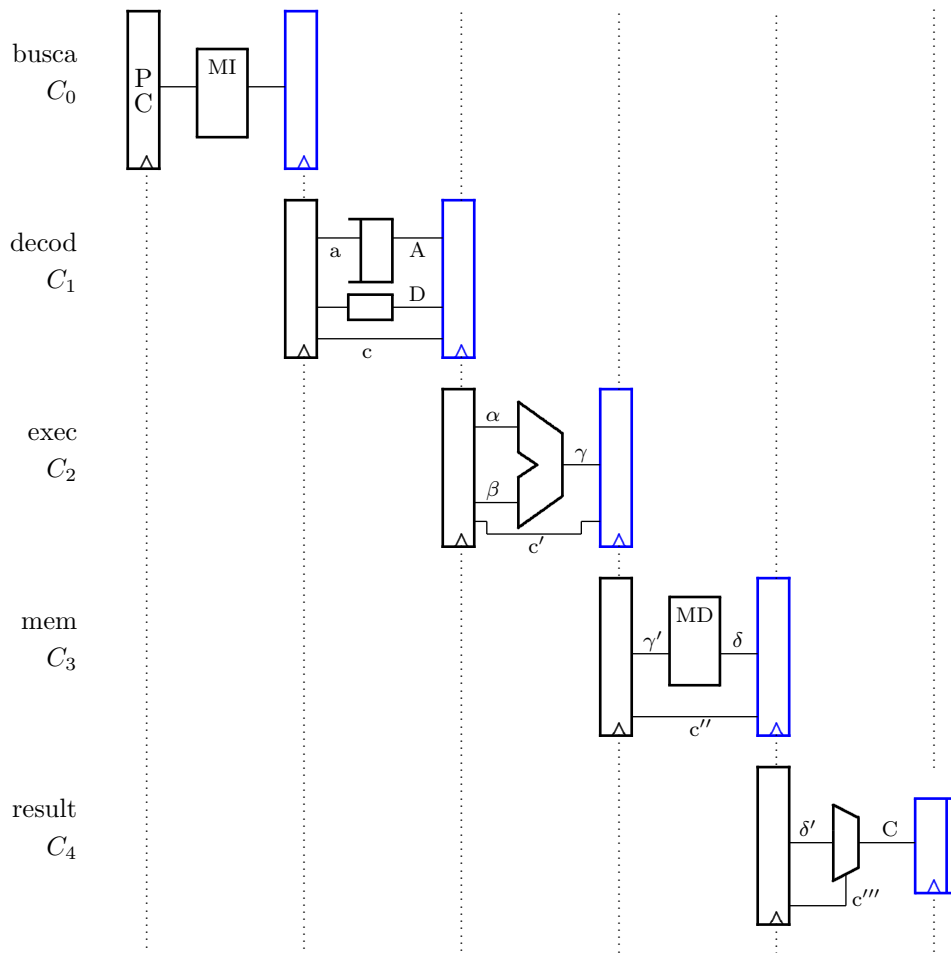


Figura 12.21: Progresso de um `lw` pelos segmentos.

Exemplo 12.4 A instrução **BRANCH-AND-LINK** é similar a um **JUMP-AND-LINK**, exceto que o modo de endereçamento é o mesmo daquele de um desvio condicional: o endereço de destino é um deslocamento com relação ao PC. A especificação da instrução **bal** é:

bal desl # R(31) $\leftarrow$ PC+8 , PC $\leftarrow$ (PC+4)+(ext(des) $\ll$ 2)

Esta instrução tem formato do Tipo I. O circuito para executar a instrução **BRANCH-AND-LINK** é mostrado na Figura 12.22 – neste diagrama o banco de registradores *não* está separado em uma seção de leitura e outra de escrita.

O endereço de destino independe de qualquer comparação, e o circuito para computá-lo é o mesmo que para as instruções **beq**: a constante é estendida para 32 bits, multiplicada por 4 e então adicionada ao PC+4.

O endereço de retorno pode ser computado no estágio de execução ((PC+4)+4), e deve ser levado até o estágio de resultado, para então ser armazenado no registrador 31. Para tanto, o multiplexador no estágio de resultado ganha uma nova entrada (PC+8), assim como multiplexador do registrador de destino ganha uma entrada fixa em 31.

O endereço de retorno deve ser PC+8 porque este é o endereço da primeira instrução que está além do *branch delay slot* – este assunto deve esperar até a Seção 12.3.2. ◁

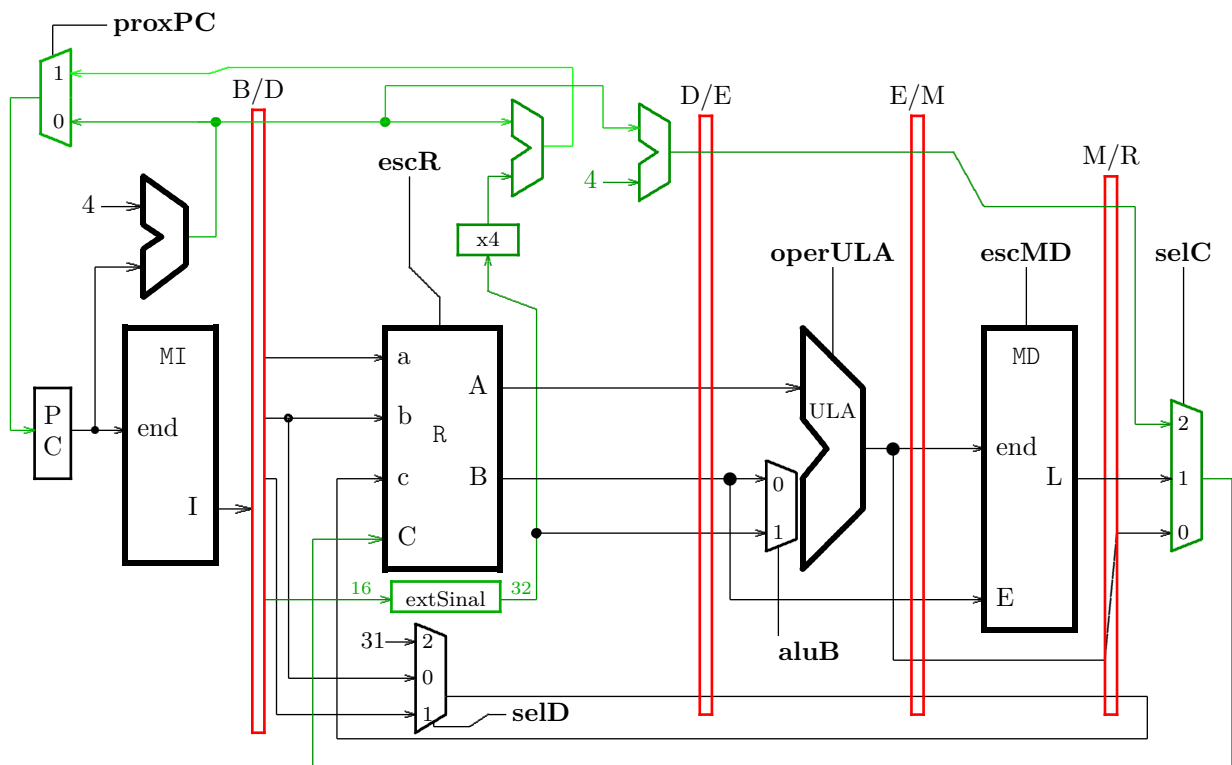


Figura 12.22: Processador com suporte à instrução **bal**.

12.3.2 O ganho é menor que o número de estágios

Ao contrário dos automóveis numa linha de montagem, as instruções de um programa dependem de outras instruções que se encontram adiante na “linha de instruções”, e isso tem impactos significativos no desempenho de processadores segmentados. Nos interessam três situações em particular: dependências de dados, desvios e *loads*.

Usaremos um diagrama de blocos simplificado do processador para examinarmos a execução das instruções através dos segmentos. A Figura 12.23 mostra o modelo simplificado do processador.

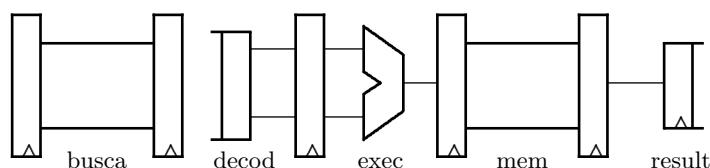


Figura 12.23: Modelo simplificado do processador.

Dependências de dados

Uma das formas de dependência entre instruções é chamada de *dependência de dados* e o nome decorre do fluxo de dados que existe entre duas ou mais instruções. Os dados fluem de uma instrução produtora para instruções consumidoras através dos registradores visíveis ao programador. As instruções consumidoras são chamadas de *instruções dependentes*.

Considere o trecho de programa abaixo, no qual o valor computado pelo **add** em r1 é usado como operando pelo **sub**. O **sub** *depende* do **add** para computar a diferença entre r2 e r1.

```
add  r1,  r6, r7
sub  r3,  r2, r1
```

O diagrama do processador na Figura 12.24 mostra o ciclo em que o **add** está no estágio de execução e o **sub** está obtendo seus operandos do banco de registradores. O valor contido em **r1**, no banco de registradores, é um valor caduco que foi produzido por alguma instrução anterior, e não a soma de **r6** com **r7**, que é o desejado pelo programador.

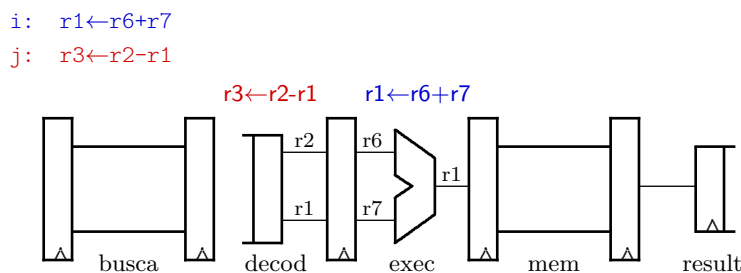


Figura 12.24: Dependência de dados.

A implementação do processador segmentado cria um *risco*: se esta sequência for executada, existe o risco – neste caso uma certeza – de o programa produzir resultados errados. Esta distinção é importante: as dependências são introduzidas no programa pelo programador; os riscos são introduzidos pelo projetista do processador. Se a implementação pode violar as

dependências entre instruções, então os projetistas do *hardware* e do *software* devem prover os meios para a execução correta dos programas.

No caso acima a solução é simples porém ineficiente: basta afastar a instrução dependente da instrução produtora para garantir que seu resultado seja depositado no banco de registradores *antes* que este valor seja lido pela instrução dependente. No diagrama da Figura 12.25, a instrução dependente está separada da produtora por dois **nops**, o que garante que o valor usado pelo **sub** será aquele produzido pelo **add**.

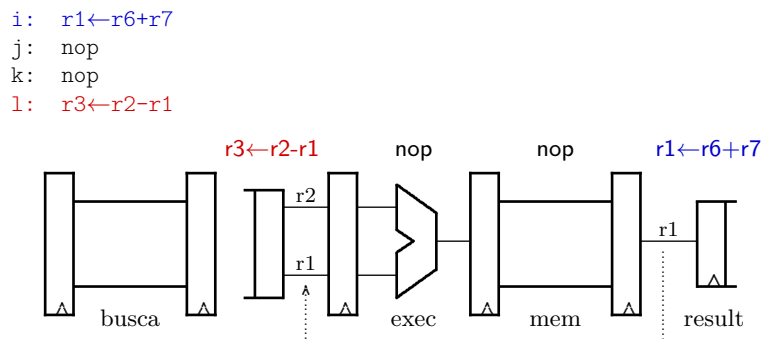


Figura 12.25: Dependência de dados resolvida com dois nops.

A corretude é obtida ao custo de dois ciclos desperdiçados por causa da dependência de dados. Se o compilador reordenar o código, é possível substituir um, ou os dois, **nops** por instruções que efetuem trabalho útil e assim minorar a queda de desempenho causada pela dependência.

Exemplo 12.5 Abaixo estão dois trechos de código que computam o mesmo resultado. No lado esquerdo, o código ignora as dependências de dados entre as instruções. No lado direito está a versão que executaria corretamente no processador da Figura 12.23. Verifique se todas as dependências de dados estão resolvidas com os **nops** que foram adicionados para garantir execução correta.

<i># riscos por resolver</i>	<i># riscos resolvidos</i>
add r5, r6, r7	add r5, r6, r7
sub r8, r9, r5 <i># r5</i>	nop
addi r2, r8, 9 <i># r8</i>	nop
xor r10, r8, r5 <i># r8, r5</i>	sub r8, r9, r5
add r5, r10, r2 <i># r10, r2</i>	nop
	nop
	addi r2, r8, 9
	xor r10, r8, r5
	nop
	add r5, r1, r2

A resolução dos riscos neste trecho de código custa cinco ciclos adicionais, o que reduz a vazão em 50% neste trecho de código. Felizmente, é possível minimizar as perdas causadas por dependências de dados. <

Existem duas técnicas de *hardware* para minimizar o risco causado pela dependência de dados. Na mais simples, o controle do processador detecta a dependência, ao comparar os números dos registradores de resultado nos estágios **exec** e **mem** com os operandos das instruções em **decod**. Uma vez detectada a dependência, a instrução que está em **decod** fica bloqueada até que a instrução produtora chegue a **result**, quando então o valor atualizado é obtido do banco

de registradores. Há desperdício de tempo, mas o compilador não é obrigado a inflar o código com a inserção dos **nops**. Os ciclos perdidos por causa dos bloqueios são chamados de *bolhas*.

Exemplo 12.6 Vejamos uma parte do circuito que computa as dependências de dados entre as instruções. Este circuito é geralmente implementado no estágio de decodificação, e é usado para decidir se a instrução que está sendo decodificada deve esperar até que seus operandos estejam disponíveis no banco de registradores.

O número do registrador de destino deve ser transportado com a instrução, na medida em que esta progride pelos estágios, para que seja possível ‘casar’ o resultado com o registrador de destino. Chamemos este registrador de *a_c*, e cada um dos registradores de segmento possui um campo *a_c*, *viz.* EXE.*a_c*, MEM.*a_c*, e RES.*a_c*.

O circuito que detecta uma dependência no primeiro operando de uma instrução, DEC.*rs* pode ser modelado em VHDL como:

```
if (DEC.rs = EXE.a_c) and
    (EXE.a_c != 0) and
    (EXE.regWR = true)
then
    segura_em_decod <= true;
end if;
```

Se a instrução no estágio de execução escreve num registrador (*regWR*) e este registrador não é o registrador *r0*, e o registrador de destino daquela instrução é o operando da instrução que está sendo decodificada, então esta instrução deve permanecer no estágio **decod** (*segura_em_decod*), para que a instrução produtora avance até o estágio **result**, quando esta gravará o seu resultado no registrador de destino.

Um teste similar deve ser efetuado para a instrução que está em **mem**:

```
if ( ( (DEC.rs = EXE.a_c) and
        (EXE.a_c != 0) and
        (EXE.regWR = true) )
    or
    ( (DEC.rs = MEM.a_c) and
      (MEM.a_c != 0) and
      (MEM.regWR = true) ) )
then
    segura_em_decod <= true;
end if;
```

Se a instrução no estágio de memória produz um operando para a instrução em **decod**, então esta deve esperar até que aquela chegue em **result**.

Estes testes, com as adaptações necessários, devem ser efetuados para o segundo operando das instruções, que é apontado pelo campo *rt*. ◁

A técnica mais eficiente, e com maior complexidade, consiste em adiantar os resultados das instruções produtoras para a instrução dependente. O *adiantamento* de resultados pode eliminar todas as dependências de dados, com excessão do uso do *load*.

Um multiplexador é adicionado a cada entrada da ULA e ligações adiantam os valores que estão disponíveis nos registradores de segmentos nos estágios adiante da instrução que está em **exec**. O circuito de detecção de dependências é usado para acionar as entradas dos multiplexadores

de forma a adiantar os valores dos registradores de segmento diretamente às entradas da ULA. A Figura 12.26 mostra o circuito de adiantamento.

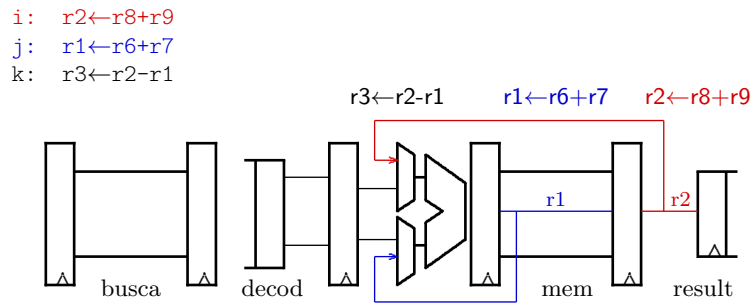


Figura 12.26: Dependência de dados resolvida com adiantamento.

Com adiantamento evitam-se as bolhas e o processador pode executar instruções na máxima taxa possível, completando uma instrução por ciclo. Infelizmente, *loads* e desvios são problemáticos, como mostra a discussão que se segue.

Uso do *load*

A Figura 12.27 mostra um diagrama do processador quando executa um *lw* seguido de um *add*. O resultado do *lw* $r1, 100(r6)$ é gravado no registrador de segmento do estágio *mem* somente no final do ciclo em que a leitura da memória é concluída, porque o circuito de memória é o componente mais lento do processador.

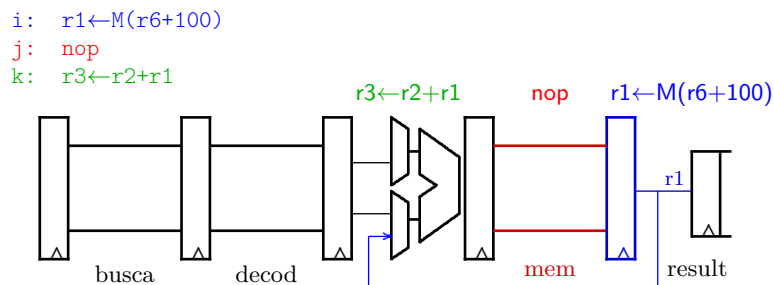


Figura 12.27: Bolha introduzida pelo *load delay-slot*.

Neste mesmo ciclo, a instrução *add* $r3, r2, r1$, que segue o *lw*, não pode usar o valor adiantado do registrador de estágio *exec* porque este valor não é o resultado obtido da leitura da memória. Para evitar o erro decorrente da leitura do valor caduco que está no registrador do segmento *exec*, o *add* deve ser mantido em *exec* por um ciclo, para garantir que o valor no registrador *mem* seja adiantado para a entrada da ULA, como indicado pela seta que adianta $r1$, na Figura 12.27.

O compilador deve inserir um *nop* entre todos os pares de instruções com um *uso do load* para garantir que a instrução dependente do *load* obtenha o valor correto para seu operando. Esta instrução *nop*, ou uma bolha, é necessária para garantir corretude mas é um desperdício de tempo. Se o compilador consegue reordenar o código de forma a preencher o *load delay-slot* com uma instrução útil, então não há desperdício de um ciclo a cada *load*. Evidentemente, a instrução que preenche o *delay-slot* não pode alterar a corretude do programa.

Exemplo 12.7 Os trechos de código abaixo não computam o mesmo resultado. No lado esquerdo, o código não considera o uso do *load*. No lado direito está a versão que executaria corretamente no processador da Figura 12.27.

<pre># riscos por resolver li r1, 0 li r2, 1024 la r3, A li r4, 0 for: lw r5, 0(r3) add r4, r4, r5 addi r3, r3, 4 addi r1, r1, 4 bne r1, r2. for fim:</pre>	<pre># riscos resolvidos li r1, 0 li r2, 1024 la r3, A li r4, 0 for: lw r5, 0(r3) nop add r4, r4, r5 addi r3, r3, 4 addi r1, r1, 4 nop nop bne r1, r2. for fim:</pre>

Há uma dependência no registrador *r1*, entre o segundo *addi* e o desvio *beq*. Esta dependência é resolvida com a introdução de dois *nops*.

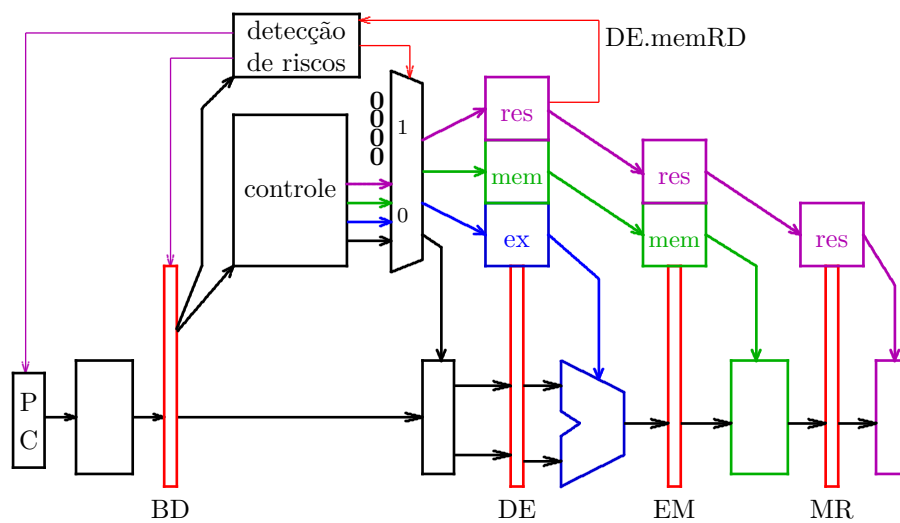
A resolução dos riscos neste trecho de código custa três ciclos adicionais a cada volta do laço, o que reduz a vazão em $9/6=1,5$. É possível reordenar as instruções para eliminar um ou mais *nops*? Se sim, quantos? A reordenação deve manter a corretude do código. ◀

Dependências de controle

As instruções de saltos e desvios causam o seu próprio *delay-slot*. A instrução de desvio só é decodificada, e o destino do desvio determinado, ao final do estágio de decodificação. Isso significa que a instrução que será executada após o desvio, caso este seja tomado, só será conhecida um ciclo após a determinação do destino. No processador segmentado, a instrução que segue o desvio é buscada antes que o destino do desvio seja conhecido. A Figura 12.28 mostra uma instrução de desvio no estágio de decodificação e a instrução que a segue, um *nop*.

Os projetistas do MIPS transformaram o limão em limonada e decretaram que a instrução que segue um desvio é sempre executada. A tarefa de preencher o *branch delay-slot* com uma instrução útil recai sobre o compilador. Se não há instrução que possa ser movida para o *delay-slot*, então este deve ser preenchido com um *nop*. Retornaremos ao tópico de reordenamento de código na Seção ??.

Exemplo 12.8 O trecho de código abaixo é uma versão reordenada daquele do Exemplo 12.7, que resolve a dependência de uso do *load* mas não elimina a bolha causada pela dependência de controle.

Figura 12.28: Bolha introduzida pelo *branch delay-slot*.

```

li    r1, 0
li    r2, 1024
la    r3, A
li    r4, 0

for:  lw    r5, 0(r3)
      addi  r1, r1, 4      # resolve uso do load
      add   r4, r4, r5
      addi  r3, r3, 4      # resolve dependência em r1
      bne   r1, r2. for
      nop                    # branch delay-slot por preencher
fim:

```

Esta versão reordenada executa com a máxima eficiência possível no processador da Figura 12.28. Este é um caso de “cobertor curto” porque falta uma instrução para preencher o *branch delay-slot*: se a instrução `add r4, r4, r5` for movida para após o `beq`, a dependência em `r1` fica exposta, e deve ser resolvida com um `nop`. ◀

Exemplo 12.9 Vejamos uma implementação da instrução `BRANCH-EQUAL`, que tira proveito do adiantamento dos valores de seus operandos. Os acréscimos ao circuito são mostrados na Figura 12.29 – neste diagrama o banco de registradores *não* está separado em uma seção de leitura e outra de escrita.

A escolha do endereço de destino depende da comparação de dois registradores, e tipicamente, um dos operandos foi computado por uma instrução que precede o `beq`, como mostrado abaixo:

```

addi  r1, r1, 4
sub    r2, r4, r5
bne    r1, r2. dest  # depende de r2 e de r1
nop                    # branch delay slot

```

Ao decodificar o `beq`, as dependências são detectadas, e o `beq` é mantido na decodificação até que o `sub` avance para o estágio de acesso à memória, quando então o valor de `r2` é adiantado da saída do registrador de segmento E/M até a entrada do multiplexador controlado por `fwdD`. O valor de `r1` é obtido da porta A do bloco de registradores.

Além do *branch delay slot*, a dependência nos operandos causa uma bolha. Esta combinação de instruções é comum, e nem sempre o compilador consegue preencher as *duas* bolhas que circundam o desvio, uma da dependência de dados, e a outra do *delay slot*. ◁

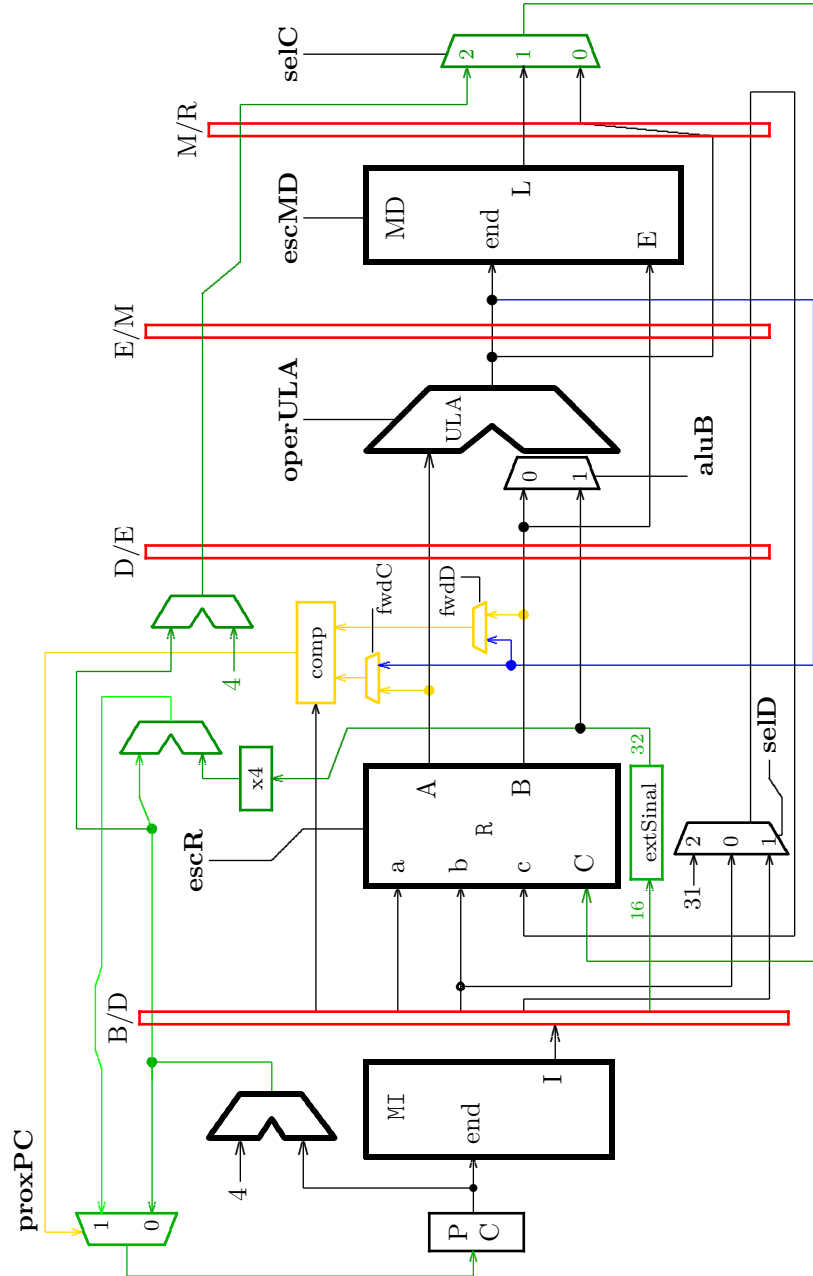


Figura 12.29: Suporte a desvios condicionais com adiantamento.

Exercícios

Ex. 12.2 Mostre como implementar a instrução JUMP-REGISTER no processador segmentado. Sua resposta deve conter: (i) uma indicação clara de quais modificações são necessárias no circuito de dados do processador; (ii) quais são os sinais de controle ativos em cada estágio; (iii) uma indicação de quais são os caminhos de adiantamento para esta instrução. O formato é R.

```
jr rs # PC <- R(rs)
```

Ex. 12.3 Repita para a instrução JUMP-AND-LINK-REGISTER, de formato R.

```
jalr rd, rs # R(rd) <- PC+8 , PC <- R(rs)
```

Ex. 12.4 Há vantagem em incrementar o PC de 8 no estágio de decodificação, na instrução BRANCH-AND-LINK? Se for vantajoso, explique quando e o por quê. Veja a Figura 12.22.

Ex. 12.5 Com base na sequência abaixo, que é deveras popular, acrescente quaisquer circuitos de adiantamento que possam ser úteis ao processador da Figura 12.29.

```
sub    r2, r4, r5
lw     r1, 0(r5)
bne    r1, r2, dest    # depende de r2 e de r1
nop                                # branch delay slot
```

Ex. 12.6 Escreva as equações para o controle de bloqueios, necessárias para eliminar os riscos de dados que você descobriu ao resolver o Exercício. 12.5. Veja o Exemplo 12.6.

Ex. 12.7 Para simplificar a exposição, todos os exemplos de código em *assembly* do Capítulo 11 ignoram os *branch delay-slots* e os *load delay-slots*. Revise todos os exemplos, e reordene as instruções para eliminar quaisquer ciclos desperdiçados.

Ex. 12.8 Para responder a esta pergunta, explicita quaisquer suposições necessárias.

Considere o programa ao lado, para ser executado no processador segmentado mais simples, sem nenhuma forma de adiantamento e nem bloqueios, mas com *branch delay-slot* e *load delay-slot*. (i) Acrescente ao código o que for necessário para garantir a execução correta deste programa. Qual o número total de ciclos necessários para armazenar a redução contida em r5? (ii) Otimize o código para reduzir o número de ciclos. Qual o novo número total de ciclos? Qual o ganho com relação ao item anterior?

```
la
r20, 0x40000000
li    r21, 800
move  r5, r0
L: lw  r10, 0(r20)
add   r5, r5, r10
addiu r20, r20, 4
addiu r21, r21, -4
bne   r21, r0, L
nop
sw
r5, -808(r20)
```

Ex. 12.9 Indique, com setas da instrução produtora para a(s) instrução(ões) dependente(s), as dependências entre as instruções do trecho de programa abaixo.

```
# int a, *b, i, j, x[N], y[M], z[P];
# a = 16 / y[ -33000 + i ];

la    ry, Y
lui   at, %hi(-33000)      ; -33.000 > 2**15
ori   rc, at, %lo(-33000)
add   r5, ri, rc
sll   r5, r5, 2            ; indice * 4
add   r5, r5, ry           ; r5 <= Y+4*(i-33000)
lw    r5, 0(r5)
li    r16, 16
div   r16, r5
mflo  r6                   ; r6 <= quoc
sw    r6, 0(ra)            ; *a <= y[i-33000]

# b = &(x[ y[ z[j]+2 ] ]);

la    rz, Z
la    ry, Y
la    rx, X
sll   r5, rj, 2            ; indice*4
add   r5, r5, rz           ; r5 <= Z+4*j
lw    r6, 0(r5)            ; r6 <= z[j]
addi  r6, r6, 2
sll   r6, r6, 2            ; indice*4
add   r6, r6, ry           ; r6 <= Y+4*([z[j]+2])
lw    r7, 0(r6)            ; r7 <= y[ z[j]+2 ]
sll   r7, r7, 2            ; indice*4
add   r7, r7, rx           ; r6 <= X+4*(y[z[j]+2])
sw    r7, 0(rb)            ; *b <= &( x[y[z[j]+2]] )
```

Índice Remissivo

Símbolos

T_A , 64
 T_D , 120
 T_I , 109, 116
 T_M , 117, 119, 186
 T_P , 116
 T_R , 379, 439, 440
 T_h , 188, 192, 199, 204
 T_p , 171
 T_s , 379, 439, 440
 $T_{C,F}$, 192
 $T_{C,x}$, 118–120, 199, 204, 206
 T_{FF} , 192
 T_{MD} , 379, 382, 440
 T_{MI} , 379, 382, 439, 440
 T_{ULA} , 379, 382, 439, 440
 T_{min} , 199, 206, 379
 T_{skew} , 203
 T_{su} , 188, 192, 199, 204, 379, 382
 $\%$, 18–20, 352, 396
 $\Rightarrow$, 37
 $\bigvee$, 47
 $\bigwedge$, 47
 $\equiv$, 36
 $\gg$, 155
 $\wedge$, 30, 36
 $\triangleleft \triangleright$, 36, 42, 66
 $\leftarrow$, 193, 344, 426
 $\Leftrightarrow$, 36
 $\Rightarrow$, 36, 37, 41, 44
 $\ll$, 155
 $\neg$, 30, 36, 154
 $\vee$, 30, 36
 $\oplus$, 36, 53, 65
 $\mapsto$, 42
 $\mathbb{N}$, 146
 $\bar{a}$, veja $\neg$
 π , 25
 $\backslash$, 77
decod, 72
demux, 75
e, 24
mod, 158, 218, 352
mux, 67
 $\&$, 344, 426
 $\mathbb{B}$, 29–33, 38, 42
 $\mathbb{Z}$, 42, 149
 $[r]$, 241, 242
 $\mathbb{N}$, 42, 43, 149

num, 44, 55, 72, 77, 261, 262, 265, 275
 num^{-1} , 55
 $\mathbb{R}$, 42
 $|N|$, 227
 $\langle \rangle$, 29, 42, 43, 180, 201, 225
 $, ,$, 344, 426
 $; ,$, 344, 426

Números

74148, 75
 74163, 218, 272
 74374, 232

A

ABI, 409
 abstração, 27, 281, 284, 293, 294, 302
 bits como sinais, 27–33, 57, 182, 184
 tempo discretizado, 116, 118, 182, 186–188,
 191–193, 197, 222
 acumulação de produtos parciais, 173–178
 adição, 152
 adiantamento, 452
 adiantamento de vai-um, 164–172, 313–316
 alfabeto, 17
 Álgebra de Boole, 27
 algoritmo, 233
 conversão de base, 18
 conversão de base de frações, 22
 amplificador, 125, 136
 diferencial, 138
 amplitude, 210
 and, veja $\wedge$
and array, 128
 apontador de pilha, 269
Application Binary Interface, veja ABI
 aproximação, 24
 arquitetura, 343, 385
 arquivo,
 objeto, 418
 árvore, 64, 118
assembler, veja montador
assembly, veja ling. de montagem
 associatividade, 31
 atraso, veja tempo de propagação, 57
 de transporte, 306
 inercial, 305
 atribuição, 12
 autómatos finitos, 236
 auto, 409
 avaliação de desempenho, 317

avaliação preguiçosa, 407

B

barramento, 140

barrel shifter, 158

básculo, 138, 185–189

relógio multi-fase, 226

SR, 185, 194

binário, 20

bit, 20

de sinal, 147

bits, 27–37, 65

definição, 29

expressões e fórmulas, 30

expressão, 30

propriedades da abstração, 31

tipo VHDL, 282

variável, 30

bloco de registradores, 265, 367, 377, 378, 427, 428, 437, 439

bloqueio, 451

bolha, 452

borda, *veja* relógio

branch, *veja* desvio condicional

branch delay slot, 449, 454

buffer, 123, 380

buffer three-state, 140

buraco, 87

busca, 369, 384, 430, 446

busca binária, 275

byte, 11

C

C, 383, 386–409

classes de armazenagem, 409

deslocamento, 160

funções, 407

if, 397, 400

if-then-else, 401

matriz, 392

overflow, 163

sizeof, 391

string, 403

vetor, 390

while, 401

cadeia,

de portas, 64, 118

de somadores, 152

caminho crítico, 117

capacitor, 105, 107, 135, 136

capture FF, *veja* *flip flop*, destino

CAS, 134

célula de RAM, 81

célula, 100

chave,

analógica, 142

digital, 92

normalmente aberta, 92

normalmente fechada, 78, 92

chip select, 269

ciclo,

combinacional, 57

violação, 81, 184, 186, 188

de trabalho, 225

ciclo longo, 378, 439, 444

circuito,

combinacional, 57, 65

de controle, 257

de dados, 257

dual, 99

segmentado, 202

sequencial síncrono, 196, 209

circuito aberto, 57

classes de armazenagem, 409

clear, 194

clk, *veja* relógio

clock, *veja* relógio, 320

clock skew, *veja* *skew*

clock-to-output, 192

clock-to-Q, 192

CMOS, 59, 65, 85–142

buffer three-state, 140

célula, 100

inversor, 96

nand, 99

nor, 98

porta de transmissão, 141

portas inversoras, 99

sinal restaurado, 125

codificação das instruções,

Mico, 345

MIPS, 429

código,

Gray, 63, 217, 227, 236

portável, 408

Column Address Strobe, *veja* CAS

combinacional,

ciclo, 57

circuito, 57

dispositivo, 57

comentário, 344, 384, 386

comparação de magnitude, 399

comparador, 427

de igualdade, 62, 164

de magnitude, 164, 261, 275

compilador,

cross, 417

nativo, 417

Complementary Metal-Oxide Semiconductor, *veja* CMOS

complemento, *veja* $\neg$

complemento de dois, 145–151, 154–164, 427, 432

complemento, propriedade, 31

comportamento transitório, *veja* transitório

comutatividade, 31

condicional, *veja* $\triangleleft \triangleright$

condutor, 85

conjunção, *veja* $\wedge$

conjunto de instruções, 385, 416, 426
 conjunto mínimo de operadores, 65
 contador, 211–223
 74163, 218
 assíncrono, 220
 em anel, 224, 225, 232
 inc-dec, 224, 272
 Johnson, 227
 módulo-16, 215, 218
 módulo-2, 212, 223
 módulo-32, 223
 módulo-4, 212
 módulo-4096, 219
 módulo-8, 213, 221
 ripple, 220, 222
 síncrono, 222
 contra-positiva, 41
 controlador, 242
 de memória, 136
 controle de fluxo, 353–357, 397–402
 conversão de base, 18
 conversor,
 paralelo-série, 229
 série-paralelo, 228
 corrente, 85, 105, 106, 112, 113
 de fuga, 115
 corrida, 123, 125
 cross-compilador, 417
 CSS, 196–197, 209
 curto-circuito, 57

D

datapath, veja circuito de dados
 decimal, 17
 decodificação, 344, 369, 384, 430, 446
 decodificador, 72, 78, 81–82, 84, 126
 de linha, 126, 135
 de prioridades, 74
delay, 57
 DeMorgan, 277
 demultiplexador, 75, 120
 dependência, 458
 de controle, 454
 de dados, 450, 454
design unit, veja VHDL, unidade de projeto, 279
 deslocador exponencial, 156
 deslocamento, 155–158, 398
 aritmético, 155, 158, 164
 exponencial, 159, 201
 lógico, 155, 162
 rotação, 158
 desvio condicional, 374, 382, 397, 435
 detecção de bordas, 123
 diagrama de estado, 234
 restrições, 236
 diretiva, 418
 disjunção, veja $\vee$
 dispositivo, 85
 combinacional, 57

distributividade, 31, 34, 51
 divisão de frequência, 213, 216
 divisão inteira, 44, 274
 doador, 87
don't care, 70, 377
 dopagem por difusão, 86
 dopante, 86
dot, 418
 DRAM, 134–137
 controlador, 136
 fase de restauração, 137
 linha,
 de palavra, 135
 linha de bit, 135
 linha de palavra, 136
 página, 135
 refresh, 135
 dual, 32, 95, 99
 dualidade, 32
duty cycle, 225

E

EEPROM, 133
 efeito colateral, 408
 endereço, 77
 alinhado, 390
 base, 351
 de destino, 353, 374, 375, 398, 435, 436, 449, 455
 de retorno, 358, 375, 449
 efetivo, 351, 372, 373, 391, 433, 434
 relativo ao PC, 398, 449
 energia, 100, 105, 112–115
 enviesado, relógio, veja *skew*
 EPROM, 133
 equação característica do FF, 193
 equivalência, veja $\Leftrightarrow$
 erro,
 de representação, 23
 escopo, 307, 409
 espaço,
 de nomes, 343, 386
 especificação, 42, 281, 294, 385
 estágio,
 busca, 446, 448
 decod, 444–446, 448, 452
 exec, 446, 448, 451
 mem, 446, 448, 452, 453
 result, 444, 446, 448
 estado, 182
 estado atual, 196, 209, 212
 estrutura de dados, 395
 excessão, 388
 execução, 370–372, 377, 384, 431–433, 437, 446
 paralela, 344, 426
 sequencial, 344, 426
 exponenciação, 264
 expressões, 36
 extensão,
 de sinal, 371, 427, 432, 435

de zeros, 427, 432
extern, 409

F

fan-in, 109–112, 116, 167, 170
fan-out, 82, 84, 109–112, 116, 120, 170–172
fatorial, 274
fechamento, 31
FET, 91
Field Effect Transistor, veja FET
Field Programmable Gate Array, veja FPGA
FIFO, 271
fila, 271–274
 circular, 271
filtro digital de ruído, 190
flip-flop, 188–195
 adjacentes, 198
 comportamento, 193
 destino, 198
 fonte, 197, 198
 mestre-escravo, 189
 modelo VHDL, veja VHDL, *flip-flop*
 temporização, 192
 tipo JK, 193
 tipo T, 191, 193
 um por estado, veja um FF por estado
folga de *hold*, 198
folga de *setup*, 198
forma canônica, 48
formato de instrução,
 I, 429, 449
 J, 429
 Mico, 345
 R, 429, 431
FPGA, 194, 301
frações, veja ponto fixo
frequência, 210
frequência máxima, veja relógio
função, 30
 aninhamento, 360
 convenção, 409
 declaração, 407
 definição, 407
 folha, 410
 protocolo de invocação, 359
 tipo, 29, 42
função de próximo estado, 233, 240, 335
função de saída, 233, 240
função, aplicação bit a bit, 32
função, tipo (op. infixo), veja $\mapsto$
funções, 358–365, 407–414

G

ganho de desempenho, 317
gas, 387
gerador de relógio, 192
ghdl, 276
glitch, veja transitório
GND, 93

gramática, 17
gtkwave, 276, 301, 312

H

half word, 386
handshake, 257
hexadecimal, 19
hold time, 188, 197–202, 246
 folga, 198, 204, 205

I

idempotência, 31
identidade, 31
igualdade, 30
imediato, 371, 429, 432, 437
implementação, 42, 385, 426
 ciclo longo, 430
 segmentada, 443
implicação, veja $\Rightarrow$
informação, 16
inicialização,
 flip-flop, 194
Instrução,
 busca, veja busca
instrução, 12, 343, 416
 add, 344, 370, 378, 379, 384
 addi, 346, 371, 432
 addu, 427, 431, 439, 440, 446
 andi, 432
 bal, 441, 449, 457
 beq, 353, 374, 381, 398, 427, 435, 455
 busca, veja busca
 com imediato, 371, 432
 corrente, 384
 decodificação, veja decodificação
 execução, veja execução
 formato, 345, 429
 j, 353, 400, 427, 436
 jal, 358, 375, 409
 jalr, 457
 jr, 358, 375, 409, 457
 lógica e aritmética, 370, 431
 ld, 348, 372, 379, 380
 lw, 427, 433, 440, 448, 453
 nop, 353, 451
 ori, 427, 432
 resultado, veja resultado
 slt, 345
 st, 348, 373, 381
 sw, 427, 434
interface,
 de rede, 13
 de vídeo, 12
inversor, 96
 tempo de propagação, 109
involução, 31, 61
IP, 343, 367, 369
isolante, 85

J

Joule, 112
jump, *veja* salto incondicional

L

laço, 401–404
 infinito, 399
 reordenar instruções, 454
label, 334, 353, 354, 386, 398
 latência, 443
latch, *veja* búsculo
latch FF, *veja flip flop*, destino
launch FF, *veja flip flop*, fonte
 Lei de Joule, 112
 Lei de Kirchoff, 106
 Lei de Ohm, 105, 106
 LIFO, 269
 ligação,
 barramento, 140
 em paralelo, 93, 99
 em série, 93, 99
 linguagem,
 assembly, *veja* ling. de montagem
 C, *veja* C
 de montagem, 342–366, 385–391, 397–402, 409–414
 declarativa, 276, 286
 imperativa, 286
 Pascal, *veja* Pascal
 Verilog, 277
 VHDL, *veja* VHDL
 Z, 27
 linha de endereçamento, 78
 literal, 38
load delay slot, 453
location counter, 418
 logaritmo, 43
 lógica restauradora, 125
look-up table, 301
 LUT, 301

M

MADD, 201
 Mapa de Karnaugh, 49, 124
 máquina de estados, 233, 236
 Mealy, 238, 245, 256, 274, 338
 modelo VHDL, 335
 Moore, 237, 244, 256, 271, 336
 projeto, 240
 Máquina de Mealy, *veja* máq. de estados
 Máquina de Moore, *veja* máq. de estados
 MARS, 402
 máscara, 32
 matriz, 392
 máximo e mínimo, 31
 maxtermo, 46
 MD, 369, 428, 448
 Mealy, *veja* máq. de estados
 memória, 181

atualização, 77
 bit, 184
 de controle, 377
 de dados, 369, 428
 de instruções, 368, 427, 428
 de programação de FPGA, 301
 de vídeo, 13
 decodificador de linha, 80
 endereço, 77
 FLASH, 133
 matriz, 129, 132, 134
 multiplexador de coluna, 80
 primária, 13
 RAM, 81, 134, 268
 ROM, 78, 126, 245
 secundária, 13
 memória dinâmica, *veja* DRAM
 memória estática, *veja* SRAM
 metaestabilidade, 184, 188, 191, 194
 defesa contra artes das trevas, 191
 metodologia de sincronização, 378, 439
 MI, 368, 428, 446, 448
 Mico XII, 342
 assembly, 342–366
 processador, 367–382
 temporização, 378–381
 microarquitetura, 385
 microcontrolador, 245–253
 microrrotina, 253
 minitermo, 45, 124, 126
 MIPS, 385
 MIPS32, 342, 364, 426
 modelagem, 281
 modelo, 281
 dataflow, 295
 de von Neumann, 383
 estrutural, 285, 295, 296
 funcional, 44, 294, 296
 porta lógica, 96
 RTL, 295
 temporal, 305
 temporização, 115
 modo de endereçamento, 385
 módulo de contagem, 221
 módulo, *veja* %, *mod*
 montador, 385, 416–424
 duas passadas, 423
 Moore, *veja* máq. de estados
 MOSFET, 91
 multiplexador, 61, 66–70, 80, 101, 117, 119, 123–124, 126, 141, 142, 279–280, 285–287, 310–312, 367, 377, 427, 437, 452
 de coluna, 132, 136
 multiplicação, 172–178, 262
 acumulação de produtos parciais, 173–178
 multiplicador,
 somas repetidas, 262, 274
multiply-add, *veja* MADD

N

número,
 de Euler, 24
 negação, *veja* $\neg$
net list, 285, 295
 níveis de abstração, *veja* abstração
 nível lógico,
 0 e 1, 28
 indeterminado, 28, 109, 116, 141
 terceiro estado, 140
 nó, 96
 nomes simbólicos, 351
non sequitur, 37
 not, *veja* $\neg$
 número primo, 53

O

octal, 18
 onda quadrada, 188, 210, 320
opcode, 345, 369, 377, 386, 429, 430, 438
 secundário, 381
 operação,
 binária, 29
 bit a bit, 32
 infixada, 42, 385
 MADD, 201
 prefixada, 47, 280, 385
 unária, 29
 operação apropriada, 197
 operações sobre bits, 29–33
 operador,
 binário, 29
 lógico, 36
 unário, 29
operation code, *veja* *opcode*
 or, *veja* $\vee$
or array, 129
 otimização de código, 451–454
 ou exclusivo, *veja* $\oplus$
 ou inclusivo, *veja* $\vee$
output enable, 269
overflow, 148–150, 154, 162–164, 173, 178, 387

P

paridade, 333
 ímpar, 49
 par, 49
 Pascal, 346–365
 funções, 358–365
 if, 354
 if-then-else, 354
 while, 355
 PC, 383, 427, 430, 435, 449
 período mínimo, *veja* relógio
 pilha, 269–271, 274, 360–365, 410
pipelining, *veja* segmentação, 202, 443
 piso, *veja* [v]
 ponto fixo, 151–152
 ponto flutuante, 70

pop, 269
 porta,
 de escrita, 266
 de leitura, 266
 porta lógica, 65
 and, 59
 carga, *veja* *fan-out*
 de transmissão, 141, 185
 modelo VHDL, 296
 nand, 60, 99
 nor, 60, 98
 not, 59, 96
 or, 59
 xor, 60, 65, 191
 portabilidade, 408
 portas complexas, 100
 potência, 112–115
 dinâmica, 114
 estática, 115
 potenciação, 43
 precedência, 30
 precisão,
 representação, 23
preset, 194
 prioridade,
 decodificador, 74
 processador, 12, 365–382, 430–456
 produtivo, 33
 produto de somas, 46
Program Counter, *veja* PC
 programa de testes, *veja* VHDL, *testbench*
 PROM, 133
 propriedades, operações em $\mathbb{B}$, 31
 protocolo,
 de sincronização, 257, 274
 invocação de função, 359
 prova de equivalência, 40–41
 próximo estado, 196, 212
 pseudoinstrução, 353, 390
pull-down, 96
pull-up, 96, 126, 141
 pulso, 122, 123, 185, 194, 211, 219, 235, 261
 espúrio, *veja* transitório
push, 269

R

raiz quadrada, 274
 RAM, 12, 77, 81, 134–139
 célula, 81
 dinâmica, 135
Random Access Memory, *veja* RAM
 RAS, 134
Read Only Memory, *veja* ROM
 realimentação, 81
 receptor, 87
 rede, 96
 redução, 33, 355, 457
refresh, 137, 139
 register, 409

Register Transfer Language, veja RTL
 registrador, 195, 232, 250, 265, 343, 367, 427
 \$zero, 387
 a0-a3, 412
 base, 391
 carga paralela, 195
 convenção de uso, 409
 de segmento, 201, 444
 destino, 377, 437
 invisível, 445
 k0,k1, 412
 ra, 364, 412, 449
 s0-s7, 412
 simples, 195
 sp, 361, 364, 412
 t0-t9, 412
 v0,v1, 412
 visível, 387, 445
 registrador de deslocamento, 228–232
 modelo VHDL, veja VHDL, registrador, 330
 paralelo-série, 229
 série-paralelo, 228
 universal, 231
 registrador de estado, 196, 335
 registro de ativação, 362, 410
 regra,
 de escopo, 409
 relógio, 186, 188, 192, 210–225, 320, 379, 440, 444
 bordas, 322
 ascendente, 189
 descendente, 189
 ciclo de trabalho, 225
 enviesado, veja skew
 frequência máxima, 199
 multi-fase, 225
 período mínimo, 199, 379–381
 reordenação de código, 451
 representação,
 abstrata, 28
 binária, 20
 complemento de dois, 147
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 ponto fixo, 151
 posicional, 17
 precisão, 23
reset, 185, 194, 195, 321, 326
 resistência, 87, 100, 105
 resultado, 344, 370, 372, 375, 384, 437, 446
 retorno de função, 375
return address, 358
 risco, 450, 451
 ROM, 12, 77–80, 126–133
 rotação, 158, 164
Row Address Strobe, veja RAS
 RTL, 202, 278, 295
rvalue, 327

S

salto incondicional, 353, 400, 436
 salto para função, 375
 seção, 418
 .bss, 418
 .data, 418
 .text, 418
 segmentação, 201, 443
 segmento, 348
 dados, 347, 348, 357
 texto, 347, 348
 seletor, 72
 semântica, 17, 388
 semicondutor, 85
 tipo N, 87
 tipo P, 87
set, 185, 194, 326
setup time, 188, 197–202, 246, 379, 439, 440
 folga, 198, 204, 205
signed, 388
 silogismo, 37
 simplificação de expressões, 38–40
 simulação,
 eventos discretos, 288
 sinal, 27, 42, 283, 327
 analógico, 27
 digital, 27, 28
 fraco, 125, 126, 138, 141
 intensidade, 90, 139
 restaurado, 125, 139
 síntese, veja VHDL, síntese, 301
 sizeof, 391
skew, 202–207
Solid State Disk, veja SSD
 soma, veja somador
 soma de produtos, 45, 51, 126
 completa, 45
 somador, 145, 152–153, 294–301, 367, 427
 adiantamento de vai-um, 165, 232, 312–317
 cadeia, 152
 completo, 104, 152, 296
 modelo VHDL, 296, 313, 315
 overflow, 163
 parcial, 103
 seleção de vai-um, 170–172
 serial, 231
 temporização, 201, 312–317
 teste, 178, 299
 somatório, 33
 spice, 28
 SRAM, 138–139
 SSD, 14
stack frame, 362, 410
 static, 409
status, 162
string, 403
 struct, 395
 subtração, 154
 superfície equipotencial, 96, 110

T

tabela,
 de controle, 370, 377, 438
 de excitação dos FFs, 193
 de símbolos, 423
 tabela verdade, 33–35, 45
 tamanho, *veja* |N|
 tempo,
 de compilação, 393
 de contaminação, 115, 118–121, 184, 192, 307
 de estabilização, 188, 192
 de execução, 394
 de manutenção, 188, 192
 de propagação, 57–58, 61, 64–65, 73, 77, 84, 102,
 104, 109, 115–117, 192, 207, 222, 223, 305
 discretizado, 186, 188, 192, 197, 222
 temporização, 104–126, 305–317, 378–381
 somador, 312–317
 tensão, 105
 Teorema,
 Absorção, 49
 DeMorgan, 32, 41, 48, 54, 60, 61, 94, 98
 Dualidade, 99
 Simplificação, 49
 terceiro estado, 140–141
testbench, *veja* VHDL, *testbench*
 teste,
 cobertura, 179
 de corretude, 178
 teto, *veja* [r]
three-state, *veja* terceiro estado
 tipo,
 de sinal, 42
 função, 29
 Tipo I, *veja* formato
 Tipo J, *veja* formato
 Tipo R, *veja* formato
 transferência entre registradores, *veja* RTL
 transistor, 87–91, 95–96
 CMOS, 95
 corte, 113
 gate, 88
 saturação, 113
 sinal fraco, 90
 tipo N, 88
 tipo P, 89
Transistor-Transistor Logic, *veja* TTL
 transitório, 121–123, 186, 239, 312
 transmissão,
 serial, 230
transmission gate, *veja* porta de transmissão
 TTL,
 74148, 75
 74163, 218
 74374, 232
 tupla, *veja* ⟨⟩
 elemento, 32
 largura, 43

U

ULA, 160–164, 180, 265, 343, 367, 371, 377, 385,
 427, 432, 437, 452
 status, 162
 um FF por estado, 253–256
 Unidade de Lógica e Aritmética, *veja* ULA
 unidade de projeto, 284
unsigned, 388
 uso do *load*, 452, 453

V

valor da função, 30
 vazão, 443
 VCC, 93
 Verilog, 277
 vetor, 390
 de testes, 290
 endereçamento, 392
 vetor de bits, *veja* ⟨⟩, 32
 largura, 43
 VHDL, 194, 276–341, 344, 347
 &, 282
 (i), 283
 :=, 283
 <=, 282
 =>, 287
 active, 328
 after, 305, 312
 all, 285
 architecture, 279
 área concorrente, 286
 arquitetura, 279
 array, 291
 assert, 293
 associação,
 nominal, 287
 posicional, 286
 atraso,
 de transporte, 306
 inercial, 305
 atributo, 322, 328
 bit, 279, 283, 295
 bit_vector, 279
 boolean, 283
 borda ascendente, 322, 339
 case, 332
 concatenação, 283
 delta, 288–289, 319
 design unit, *veja* VHDL, unidade de projeto
 entidade, 279
 entity, 279
 event, 322, 328
 evento, 288
 exit, 334
 flip-flop, 322–327
 for, 293, 333
 função, 339–341
 generate, 312
 generic, 309

in, 279, 334
 inertial, 305
 inicialização de sinal, 283
 instanciação, 286
 interface genérica, 309
label, 334
last_value, 328
length, 328
 linguagem, 276
 loop, 333, 334
 mapeamento de portas, 286
 máquina de estados, 335
 modelo executável, 281
 open, 310
 others, 332
 out, 279
 package, 284
 port map, 280, 286, 287
 processo, 288
 combinacional, 329
 sequencial, 330
 range, 328, 334
 record, 290
 registrador, 330
 reject, 306, 312
 report, 293
rising_edge, 339, 340
rvalue, 327
 síntese, 281
 seleção de elemento de vetor, 283
 severity, 293
 simulação, 288
 sinal, 283
 síntese, 207, 301, 305
std_logic, 140
 subtype, 284
testbench, 289–293, 299–301
 time, 283
 tipos, 42, 282, 287
 to, 334
 transação, 288
 transport, 306
 type, 283, 335
 unidade de projeto, 284
 use, 285
 variável, 327–331
 wait, 320, 321, 323
 wait for, 321
 wait on, 321
 wait until, 321
 when, 332
 while, 334
 work, 285

W

Watt, 112
word, 386
write back, veja resultado

X

xor, veja $\oplus$

Z

Z, linguagem, 27

13.1 Modelagem de Circuitos Combinacionais em VHDL

VHDL é uma sigla que contém uma sigla: *VHSIC Hardware Description Language*, sendo VH-SIC abreviatura para *Very High Speed Integrated Circuits*. Como o nome diz, VHDL é uma linguagem de **descrição** de *hardware* (*HDL*). Neste, e nos três próximos laboratórios, usaremos primariamente construções para descrever a *estrutura* dos circuitos. As construções para definir *comportamento* serão vistas adiante.

Diferentemente de C, um programa em VHDL *descreve* uma estrutura, ao invés de *definir* ou *prescrever* uma computação. Neste sentido, um programa em C é uma “receita de bolo” com ingredientes (variáveis) e modo de preparar (algoritmo), enquanto que em VHDL um programa descreve a interligação de componentes. A partir da estrutura, o compilador ghdl gera um simulador para o circuito descrito pelo código fonte. Adiante veremos como definir o comportamento de um circuito, sem que seja necessário definir a sua estrutura.

13.1.1 Objetivos

São três os objetivos deste laboratório: (i) efetuar a modelagem estrutural em VHDL de multiplexadores, demultiplexadores e decodificadores; (ii) verificar, através de simulação, a corretude dos modelos daqueles circuitos; e (iii) iniciar o contato com a linguagem VHDL.

O trabalho pode ser efetuado em duplas. Não há entrega(s) neste laboratório.

13.1.2 Descrição gráfica de circuitos

Veja a Seção 9.1 de `vhdl.pdf`.

13.1.3 VHDL

VHDL é uma linguagem extremamente versátil e é usada para

- * a modelagem e simulação de circuitos;
- * especificação, projeto e síntese de circuitos;
- * descrever lista de componentes e de ligações; e
- * verificação de corretude: se especificação $\Rightarrow$ implementação.

Tipos de dados

Veja a Seção 9.2.1 de `vhdl.pdf`.

Como se dá a compilação de VHDL?

Veja a Seção 9.2.4 de `vhdl.pdf`.

Mecanismo de simulação

Veja a Seção 9.2.5 de `vhdl.pdf`.

13.1.4 Da tarefa

Etapas 1 Copie para sua área de trabalho o arquivo com o código VHDL, e extraia seu conteúdo com os seguintes comandos:

- (1) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_estrutural.tgz`
- (2) expanda-o com `tar xzf l_estrutural.tgz`
- (3) o diretório `estrutural` será criado na expansão da *tarball*;
- (4) mude para aquele diretório com `cd estrutural`

O arquivo `packageWires.vhd` contém definições ou apelidos para nomes de sinais para facilitar a digitação, porque digitar `reg8` é mais econômico do que `bit_vector(7 downto 0)`. **packages** são descritas na Seção 9.2.2.

O arquivo `aux.vhd` contém os modelos das portas lógicas *not*, *and*, *or*, *xor*, que são os componentes básicos para este laboratório. Este arquivo não deve ser editado.

O arquivo `estrut.vhd` contém um modelo para um multiplexador de duas entradas, *mux-2*. Este modelo serve de base para que você escreva os modelos para os componentes *mux-4*, *mux-8*, *demux-2*, *demux-4*, *demux-8*, *decod-2*, *decod-4* e *decod-8*, que foram vistos em sala e estão definidos na Seção 4.3 de `combin.pdf`.

O *script* `run_mux.sh` compila o código VHDL e produz um simulador. Se executado sem nenhum argumento de linha de comando, `run_mux.sh` somente (re)compila o simulador e executa a simulação. Eventuais mensagens de erro são mostradas na tela. Com qualquer argumento o *script* também dispara a execução de `gtkwave`: `run_mux.sh 1 &`.

Para compilar e verificar eventuais mensagens de erro, diga, no terminal

```
prompt: ./run_mux.sh
```

para executar a simulação e ver o diagrama de tempos no `gtkwave`, diga

```
prompt: ./run_mux.sh 1 &
```

O arquivo `v_mux.sav` contém definições para o `gtkwave` tais como a escala de tempo e sinais a serem exibidos na tela para a verificação do modelo *mux-2*.

Se a tela do `gtkwave` mostra os diagramas em tamanho inadequado, mova o arquivo `gtkwaverc` para o seu `$HOME`, como um arquivo escondido (`$HOME/.gtkwaverc`) e edite as duas últimas linhas – aquelas com `fontname_signals fixed 10` e `fontname_waves terminus 9`. Os números podem ser aumentados/reduzidos para melhorar a legibilidade. `gtkwave` deve ser reinicializado com a nova configuração.

Modelo do multiplexador

Veja a Seção 9.2.3 de `vhdl.pdf`.

O modelo está pronto, e agora?

Veja a Seção 9.2.6 de `vhdl.pdf`.

Testbench para os multiplexadores

Veja a Seção 9.2.7 de `vhdl.pdf`.

Examinaremos primeiro os quatro modelos do multiplexador. O diagrama na Figura 13.15 mostra as ligações entre os componentes que você deve modelar e o processo que percorre o vetor de testes e gera as sequências de entradas-de-teste e saídas-de-teste. As entradas-de-teste (e , k , s) excitam seus modelos, que produzem saídas de acordo com suas especificações. As saídas-de-teste (mx , dm , dc) são comparadas com as saídas produzidas pelos modelos. O número de bits em e depende da largura do multiplexador, assim como o número de bits (n e m) dos sinais dm e dc .

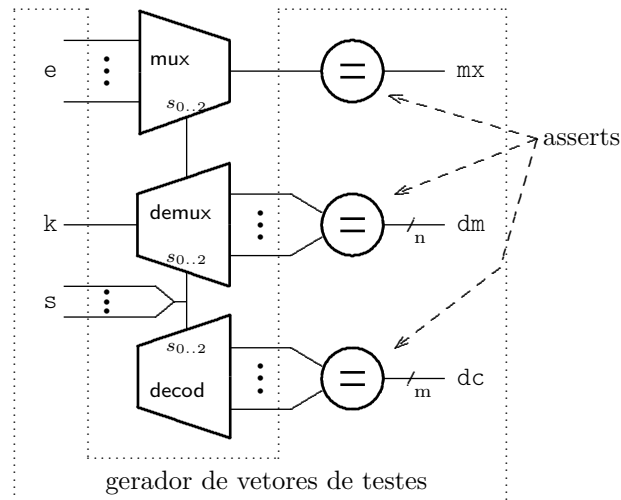


Figura 13.15: Ligações entre o *testbench* e os modelos.

O arquivo `tb_mux.vhd` contém o programa de testes (*testbench*, ou TB) para verificar a corretude dos seus modelos dos multiplexadores (*mux-4*, *mux-8* com bits e *mux-8* com vetores de bits). Para simplificar a depuração de seu código, você deve verificar cada novo modelo assim que o código VHDL for completado.

Das mensagens de erro Em caso de erro de compilação ser detectado por `ghdl`, o `script run_mux.sh` aborta a compilação, e exibe as mensagens de erro emitidas pelo compilador. Estas mensagens são a melhor indicação que o compilador é capaz de emitir para ajudá-lo a encontrar o erro, e portanto **as mensagens de erro devem ser lidas**. Os programadores do `ghdl` dispenderam um esforço considerável para emitir mensagens de erro (relativamente) úteis. Não desperdice a preciosa ajuda que lhe é oferecida.

São usados três conjuntos de vetores de teste, um para cada largura de circuito. A seguir descrevemos os vetores de teste para circuitos de largura dois. Aqueles para largura quatro e oito são similares.

A arquitetura do TB declara os componentes que serão testados e um **record** que será usado para excitar os modelos. O registro `test_record_2` possui três campos e os valores destes campos devem ser atribuídos por você de forma a gerar todas (*todas?*) as combinações de entradas necessárias para garantir a corretude do seu modelo. O vetor de testes `test_array_2` contém os oito elementos necessários para excitar e verificar o *mux-2*, na primeira tarefa deste laboratório.

No `test_record_2`, os campos `s` e `mx` são de tipo bit (`'0'`) e o campo `a` é um vetor de bits codificado em binário (`b"10"` – o prefixo `b` indica *binário*).

O campo `mx` é o bit com a saída esperada para um multiplexador quando os valores definidos em `s` e `a` são aplicados às entradas.

Programa 13.48: Vetor de valores de entrada para testar o modelo do *mux-2*.

```
% label=code:le_test]
— definicao do vetor de testes para MUX-2
type test_record_2 is record
  a : reg2;      — entrada para multiplexadores
  s : bit;       — entrada de seleção
  mx : bit;      — saída esperada do MUX
end record;
type test_array_2 is array(positive range <>) of test_record_2;

— vetor de testes
constant test_vectors_2 : test_array_2 := (
  —s,      a,      mx
  ('0',b"00", '0'), — transcricao da tabela verdade do mux-2
  ('0',b"01", '1'),
  ('0',b"10", '0'),
  ('0',b"11", '1'),
  ('1',b"00", '0'),
  ('1',b"01", '0'),
  ('1',b"10", '1'),
  ('1',b"11", '1'),
  —
  ('0',b"11", '1'), — nao alterar estes três últimos
  ('0',b"11", '1'), — estes bits indicam o final da simulação
  ('0',b"11", '1')
);

— troque a constante para FALSE para testar seus modelos
constant TST_MUX_2 : boolean := true;
```

A sequência de valores de entrada para os testes dos modelos é gerada pelo processo `U_testValues`, com um laço **for ... loop**. A variável de iteração itera no espaço definido pelo número de elementos do vetor de testes (`test_vectors'range`) – o atributo `'range` representa a faixa de valores do índice do vetor. Se mais elementos forem acrescentados ao vetor, o laço executará mais iterações. O elemento do vetor é atribuído à variável `v` e os todos os campos do vetor são então atribuídos aos sinais que excitam os modelos. Lembre que o processo `U_testValues` executa concorrentemente com o seu(s) modelo(s) e quando os sinais de teste são atribuídos no laço,

estes provocam alterações nos sinais dos modelos.

O **assert** no Programa 13.49 verifica se a saída observada no multiplexador é igual à saída esperada. Se os valores forem iguais, o comportamento é o esperado, e portanto correto COM RELAÇÃO AOS VETORES DE TESTE QUE VOCÊ ESCREVEU.

Se você escolher valores de teste inadequados, ou errados, pode ser difícil diagnosticar problemas no seu modelo.

Programa 13.49: Mensagem de verificação de comportamento.

```
assert TST_MUX_2 or (saidaMUX2 = esperadaMUX)
  report "mux2:␣saida␣errada␣sel=" & B2STR(s0) &
    "␣saiu=" & B2STR(saidaMUX2) & "␣esperada=" & B2STR(esperadaMUX)
  severity error;
```

Se os valores de saidaMUX2 e esperadaMUX diferem, a mensagem no Programa 13.49 é emitida na tela do terminal, indicando o erro:

```
tb_teste.vhd:201:7:@2ns:(assertion error):
mux2: saída errada sel=1 saiu=0 esperada=1.
```

Ao final do laço a simulação termina no comando **wait**, que faz com que a execução do processo U_testValues se encerre.

A condição de teste do **assert** é

```
TST_MUX_2 or (saidaMUX2 = esperadaMUX)
```

A constante TST_MUX_2 está definida como true logo abaixo do vetor de testes para os modelos de dois bits:

```
constant TST_MUX_2 : boolean := true;
```

Por causa do true **or** (...), o assert não efetua a comparação entre a saída e o valor esperado. A razão para isso é diminuir a poluição na tela durante os testes dos modelos.

Para testar seu(s) modelo(s), altere a constante respectiva (TST_MUX_2, TST_MUX_4 ou TST_MUX_8) para false e então verifique os resultados.

Etapa 2 Para executar a simulação sem invocar gtkwave, diga

```
prompt: ./run_mux.sh
```

Verifique se há alguma mensagem de erro; se sim, leia a mensagem e use seus miolos antes de chamar o professor.

Para executar a simulação com o gtkwave, diga

```
prompt: ./run_mux.sh 1 &
```


Diagramas de tempo do gtkwave

O script `run_mux.sh` pode disparar a execução do `gtkwave` para mostrar o diagrama de tempo dos circuitos. O diagrama de tempo mostra, de cima para baixo:

MUX-2 um agrupamento com os sinais para o *mux-2*, com

- entr_2** os dois bits de entrada para *mux-2* (laranja);
- s0** bit de seleção (azul);
- saidamux2** a saída do *mux-2* (verde);
- esperadamux** a saída esperada para os muxes (amarelo);

MUX-4 um agrupamento com os sinais para o *mux-4*, com

- entr_4** os quatro bits de entrada para *mux-2* (laranja);
- s0,s1** bits de seleção (azul);
- saidamux4** a saída do *mux-4* (verde);
- esperadamux** a saída esperada para os muxes (amarelo);

MUX-8 mesmo que para *mux-4*, com largura 8

MUX-8-vet um agrupamento com os sinais para o *mux-8*, com

- entr_8** vetor com os bits de entrada para *mux-8* (laranja);
- s** vetor com três bits de seleção (azul);
- saidamux8vet** a saída do *mux-4* (verde);
- esperadamux** a saída esperada para os muxes (amarelo);

Note que **é você quem deve ajustar a saída esperada** para cada um dos circuitos nos vetores de teste. O projetista dos modelos é responsável por escrever os vetores de teste e portanto sua tarefa é ajustar os campos *mx* (saída esperada dos *mux-N*). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **asserts** indicarão falso-positivos para erros.

Se a tela do `gtkwave` mostra os diagramas em tamanho inadequado, veja a observação na página 196.

Teste dos multiplexadores

A coluna *mx* de `test_vectors` é a saída esperada para o *mux-2* e o

```
assert TST_MUX_2 or (saidaMUX2 = esperadaMUX)
```

emitirá mensagem de erro somente se o modelo produzir saída diferente de *mx*. Se a saída esperada é a produzida, então o **assert** fica silente porque o circuito está correto, segundo o vetor de testes *que você, projetista, escreveu*.

Uma vez que você esteja certo de que o *mux-2* está correto, teste o *mux-4*. Este circuito é composto de três *mux-2*, e você garante que o *mux-2* é um circuito que atende a sua especificação. O *mux-4* tem seis entradas e sua tabela verdade tem $2^6 = 64$ linhas, sendo portanto necessários 64 testes. Certo?

Sim, é certo. Podemos usar de inteligência e descobrir qual é o número *mínimo* de testes para garantir a corretude do *mux-4*. Este número é bem menor do que 64.

Gere o vetor de testes (reduzido) para o *mux-4* e verifique sua corretude. Isso feito, troque ‘TST_MUX_4 para false e execute a simulação para testar seu modelo.

O *mux-8* é composto de dois *mux-4* – que você garante que é correto – e de um *mux-2* – que você também garante que é correto. O *mux-8* tem onze entradas e sua tabela verdade tem $2^{11} = 2048$ linhas. São necessários 2048 testes. Certo? Hmm...

Gere o vetor de testes (reduzido) para o *mux-8* e verifique sua corretude. Isso feito, troque ‘TST_MUX_8 para false e execute a simulação para testar seu modelo.

Vetor de bits na entrada do *mux8vet* Há duas entidades distintas para o *mux8*. A primeira, *mux8*, emprega oito sinais de tipo bit nas entradas, e três bits para seleção. A segunda, *mux8vet* mostrada no Programa 13.50, tem como entradas um vetor de 8 bits *entr:reg8* e outro vetor *sel:reg3* para a seleção. As duas arquiteturas são idênticas, exceto que as ligações dos sinais da interface aos componentes devem usar seleção de campos de bits.

Programa 13.50: Entidade do *mux8* com vetores de bits.

```
entity mux8vet is
  port(entr : in  reg8; — vetor com 8 bits
        sel  : in  reg3; — vetor com 3 bits
        z    : out bit); — saída de 1 bit
end mux8vet;
```

Para a entidade *mux8*, as entradas são a,b,c,d,e,f,g,h, enquanto que para a entidade *mux8vet*, as entradas são *entr(0)*, *entr(1)*, ..., *entr(6)*, *entr(7)*.

13.1.5 *Testbench* para os demultiplexadores e decodificadores

O *script* *run_estrut.sh* compila o código VHDL e gera um simulador para os modelos dos demultiplexadores e decodificadores/seletores. Este *script* é similar a *run_mux.sh*.

Etapa 3 Para esta etapa você deve:

- (1) escrever os modelos, segundo a Seção 3.4 de *combin.pdf*, para o *demux-2*. Para tanto edite o arquivo *estrut.vhd*;
- (2) repetir para o *demux-4* e *demux-8*;
- (3) acrescentar mais elementos aos respectivos vetores de teste em *tb_estrut.vhd*; e
- (4) verificar a corretude de seus modelos. Basta dizer `./run_estrut.sh` ler eventuais mensagens de erro, resolver os problemas, e então dizer `./run_estrut.sh 1 &`.

Achtung: no TB dos demultiplexadores e decodificadores, em *tb_estrut.vhd*, um único registro é usado para testar os dois circuitos de largura *N* (*demux-N* e *decod-N*) ao mesmo tempo e portanto, dependendo do teste, alguns dos campos não são relevantes *naquele teste*.

No registro com os vetores de teste, o campo *dm* é o vetor de bits com a saída esperada para um demultiplexador quando recebe as entradas definidas pelos valores em *k,s*.

O campo *dc* é o vetor de bits com a saída esperada para um decodificador cujas entradas são definidas pelos valores em *s*.

Usaremos o mesmo procedimento dos *mux-n* para testar os demultiplexadores. Em *tb_estrut.vhd*, mude *TST_DEMUX_2* para false.

A tabela verdade das saídas *z* e *w* do *demux-2* é mostrada ao lado, e o vetor de testes que lhe corresponde está em *tb_estrut.vhd*.

s	k	z	w
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

Da mesma forma que com os multiplexadores, os *demux-4* e *demux-8* são composições de *demux-2*. Uma vez que o componente ‘pequeno’ tenha sido verificado, a verificação do com-

ponente ‘grande’ necessita de um número relativamente pequeno de testes para também ser verificada.

Os **asserts** que verificam a corretude dos demultiplexadores e decodificadores estão desabilitados para reduzir a poluição na tela. Para testar seus modelos, altere a constante respectiva (TST_DEMUX_2 ou TST_DECOD_2) para `false` e então verifique os resultados. Há uma tripla de constantes para cada circuito, *viz* TST_DEMUX_2, TST_DEMUX_4 e TST_DEMUX_8.

Você é quem escreve os vetores de teste e portanto sua tarefa é ajustar os campos `dm` (saída esperada dos *demux-N*) e `dc` (saída esperada dos *decod-N*). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **asserts** indicarão falso-positivos para erros inexistentes.

Diagramas de tempo do gtkwave

O script `run_estrut.sh` pode disparar a execução do `gtkwave` para mostrar o diagrama de tempo dos circuitos. O diagrama de tempo mostra, de cima para baixo:

seleção bits de seleção, agrupados num vetor de três bits `s[2:0]` e os três bits isolados `s0,s1,s2`, todos em azul;

DEMUX-2 um agrupamento com os sinais para o *demux-2*, com

- inp** entrada para os demuxes (laranja);
- sdemux2** dois bits de saída do *demux-2* (amarelo);
- esperadademux_2** a saída esperada para o *demux-2* (verde);

DEMUX-4 mesmo que para o *demux-2*, com largura 4;

DEMUX-8 mesmo que para o *demux-2*, com largura 8;

DECOD-2 um agrupamento com os sinais para *decod-2*, similar ao do *mux-2*

DECOD-4 mesmo que para o *decod-2*, com largura 4;

DECOD-8 mesmo que para o *decod-2*, com largura 8.

Teste dos decodificadores

Você já entendeu o processo.

Etapa 4 Acrescente ao arquivo `estrut.vhd` o código VHDL para os modelos dos decodificadores de 2,4,8 saídas. Acrescente e/ou altere os elementos do vetor de testes para verificar a corretude dos seus modelos.

Você é quem deve ajustar a *saída esperada* para cada um dos circuitos nos vetores de teste. O projetista dos modelos é responsável por escrever os vetores de teste e portanto sua tarefa é ajustar os campos `dm` (saída esperada dos *demux-N*), e `dc` (saída esperada dos *decod-N*). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **asserts** indicarão falso-positivos para erros.

Histórico das Revisões:

24jun2020: remoção do material que está em vhd1.pdf, separação dos TBs para mux e demux;
02set2019: ajustes cosméticos, diagramas do gtkwaverc;
15ago2018: texto introdutório, exemplos com deltas, mux8vet, gtkwave, (true or) nos asserts;
03set2016: vetores de testes separados por tamanho de circuito;
26ago2016: exemplos de vetores de testes;
25ago2016: incluídas sugestões de Zanata, remoção de FPGA, troca seletor para decodificador;
22ago2016: compilação VHDL;
12set2015: ajustes cosméticos, diagrama com simulador;
29jul2015: primeira versão.

EOF

13.2 Temporização de Circuitos Combinacionais

Objetivos: são dois os objetivos deste laboratório: (i) verificar o comportamento temporal do multiplexador; e (ii) verificar a corretude dos modelos do multiplexador, do demultiplexador e do decodificador através de **asserts** e com diagramas de tempo.

O trabalho pode ser efetuado em duplas.

Na Seção 13.2.10 estão as questões que devem ser respondidas nesta aula e entregues ao professor.

13.2.1 O que é um circuito combinacional?

Reveja a Seção 4.1.

13.2.2 Modelo do comportamento temporal de CCs

Reveja a Seção 5.4.

13.2.3 Modelo funcional *versus* modelo temporal

No laboratório sobre descrição estrutural empregamos *modelos funcionais* para portas lógicas, e com aqueles construímos modelos para circuitos combinacionais pela composição de várias portas lógicas. Aqueles modelos são chamados de *funcionais* porque representam somente o aspecto “função lógica” dos circuitos, e não contêm informações de tempo.

Neste laboratório enriqueceremos os modelos funcionais com informação de tempo, criando modelos mais sofisticados e realistas para o comportamento daqueles circuitos.

Modelos temporais incorporam a informação de tempo de propagação dos componentes e permitem simulações mais detalhadas dos circuitos. Evidentemente, simulações mais detalhadas são mais custosas em termos de tempo de programação e de simulação.

Os modelos que usaremos nesta aula permitem a visualização do comportamento dos circuitos combinacionais ao longo do tempo, e a medição do tempo de propagação através de circuitos relativamente simples. Em breve estudaremos a propagação de sinais em somadores e nestes circuitos os problemas são um tanto mais severos do que o que veremos hoje. Cada coisa a seu tempo.

13.2.4 Modelo para temporização em VHDL

Reveja a Seção 5.3.2. Reveja a Seção 9.3 das notas de aula (`vhdl.pdf`).

Espaço em branco proposital.

13.2.5 Material disponibilizado para sua tarefa

Etapa 5 Copie para sua área de trabalho o arquivo com o código VHDL:

- (a) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_combinacionais.tgz`
- (b) expanda-o com: `tar xzf l_combinacionais.tgz`
o diretório combinacionais será criado;
- (c) mude para aquele diretório: `cd combinacionais`

O arquivo `packageWires.vhd` contém definições dos tempos de propagação das portas lógicas e abreviaturas para nomes de sinais. Este arquivo também contém as definições de temporização das portas lógicas.

O arquivo `aux.vhd` contém os modelos das portas lógicas *not*, *and*, *or* e *xor*, que são os componentes básicos para este laboratório. Este arquivo não deve ser editado.

O arquivo `combin.vhd` contém um modelo para um multiplexador de duas entradas, *mux-2*. Este modelo serve de base para os modelos dos componentes *mux-4*, *mux-8*, *demux-2*, *demux-4*, *demux-8*, *decod-2*, *decod-4* e *decod-8*, definidos na Seção 4.3, *Circuitos Combinacionais Básicos*, em `combin.pdf`.

13.2.6 Modelos de multiplexador

Reveja a Seção 5.3.2. Veja a Seção 9.3.4 de `vhdl.pdf` para a descrição do modelo temporizado do *mux-2*.

Ao invés de reescrever os modelos dos *mux-N*, utilize as arquiteturas dos modelos do laboratório anterior sem esquecer de que as declarações das portas lógicas devem incluir a especificação do tempo de propagação e de contaminação.

Para simular com tempos de propagação diferentes de zero, edite o arquivo `packageWires.vhd` e troque as constantes `simulate_time` e `simulate_rej` para 1. As constantes estão no topo do arquivo. Se as constantes são zero, então a simulação é puramente funcional.

13.2.7 *Testbench* para os multiplexadores

Reveja a Seção 5.3.2. Examinaremos primeiro os quatro modelos do multiplexador.

O *script* `run_mux.sh` compila o código VHDL e produz um simulador. Se executado sem nenhum argumento de linha de comando, `run_mux.sh` (re)compila o simulador e executa a simulação com **asserts**. Se invocado com qualquer argumento o *script* também dispara a execução de `gtkwave`: `./run_mux.sh 1 &`

Das mensagens de erro Em caso de erro de compilação ser detectado por `ghdl`, o *script* `run.sh` aborta a compilação, e exibe as mensagens de erro emitidas pelo compilador. Estas mensagens são a melhor indicação que o compilador é capaz de emitir para ajudá-lo a encontrar o erro, e portanto **as mensagens de erro devem ser lidas**. Os programadores do `ghdl` dispenderam um esforço considerável para emitir mensagens de erro (relativamente) úteis. Não desperdice a preciosa ajuda que lhe é oferecida.

Se a tela do `gtkwave` mostra os diagramas em tamanho inadequado, mova o arquivo `gtkwaverc` para o seu `$HOME`, como um arquivo escondido (`$HOME/.gtkwaverc`) e edite

as duas últimas definições – os números podem ser alterados para melhorar a legibilidade.
O `gtkwave` deve ser reinicializado para que as definições tenham efeito.

O arquivo `v_mux.sav` contém definições para o `gtkwave` tais como a escala de tempo e sinais a serem exibidos na tela para a verificação dos modelos `mux-2`, `mux-4` e `mux-8`.

O arquivo `tb_mux.vhd` contém o programa de testes (*testbench*, ou TB) para verificar a corretude dos seus modelos. A entidade `tb_mux` é vazia porque o programa de testes é autocontido e não tem interfaces com nenhum outro circuito.

São usados três conjuntos de vetores de teste, um para cada largura de circuito. A seguir descrevemos os vetores de teste para circuitos de largura dois. Aqueles para largura quatro e oito são similares.

A arquitetura do TB declara os componentes que serão testados e um **record** que será usado para excitar os modelos. O registro `test_record_2` possui três campos e os valores destes campos devem ser atribuídos por você de forma a gerar todas (*todas?*) as combinações de entradas necessárias para garantir a corretude do seu modelo. O vetor de testes `test_array_2` contém os oito elementos necessários para excitar e verificar o `mux-2`, na primeira tarefa deste laboratório.

No `test_record_2`, os campos `s` e `mx` são de tipo bit ('0') e o campo `a` é um vetor de bits codificado em binário (`b"10"` – o prefixo `b` indica *binário*).

O campo `mx` é o bit com a saída esperada para um multiplexador quando os valores definidos em `s` e `a` são aplicados às entradas.

Programa 13.51: Vetor de valores de entrada para testar modelos.

```

— definição do vetor de testes para MUX-2
type test_record_2 is record
  a  : reg2;      — entrada para multiplexadores
  s  : bit;       — entrada de seleção
  mx : bit;       — saída esperada do MUX
end record;
type test_array_2 is array(positive range <>) of test_record_2;

— vetor de testes
constant test_vectors_2 : test_array_2 := (
  —s,    a,    mx
  ('0',b"00", '0'), — transcrição da tabela verdade do mux-2
  ('0',b"01", '1'), — s e a são entradas, mx é a saída
  ('0',b"10", '0'),
  ('0',b"11", '1'),
  ('1',b"00", '0'),
  ('1',b"01", '0'),
  ('1',b"10", '1'),
  ('1',b"11", '1'),
  —
  ('0',b"11", '1'), — não alterar estes três últimos
  ('0',b"11", '1'),
  ('0',b"11", '1')
);

— troque a constante para FALSE para testar seus modelos
constant TST_MUX_2 : boolean := true;

```

A sequência de valores de entrada para os testes dos modelos é gerada pelo processo `U_testValues`, com o laço **for** ... **loop**. A variável de iteração itera no espaço definido pelo número de elementos do vetor de testes (`test_vectors` 'range') – o atributo 'range' representa a faixa de valores do índice do vetor. Se mais elementos forem acrescentados ao vetor, o laço executará mais iterações.

O *i*-ésimo elemento do vetor é atribuído à variável *v* e todos os campos do vetor são então atribuídos aos sinais que excitam os modelos. Lembre que o processo `U_testValues` executa concorrentemente com o seu(s) modelo(s) e quando os sinais de teste são atribuídos no laço, estes provocam alterações nos sinais dos modelos.

O **assert** no Programa 13.52 verifica se a saída observada no multiplexador é igual à saída esperada. Se os valores forem iguais, o comportamento é o esperado, e portanto correto COM RELAÇÃO AOS VETORES DE TESTE QUE VOCÊ ESCREVEU.

Se você escolher valores de teste inadequados, ou errados, pode ser difícil diagnosticar problemas no seu modelo.

Programa 13.52: Mensagem de verificação de comportamento.

```
assert TST_MUX_2 or (saidaMUX2 = esperadaMUX)
  report "mux2: _saida_errada_sel=" & B2STR(s0) &
    "_saiu=" & B2STR(saidaMUX2) & "_esperada=" & B2STR(esperadaMUX)
  severity error;
```

Se os valores de `saidaMUX2` e `esperadaMUX` diferem, a mensagem no Programa 13.52 é emitida no terminal, indicando o erro.

```
mux2: saída errada  sel=1  saiu=0  esperada=1
```

Ao final do laço a simulação termina no comando **wait**, que faz com que a execução do processo `U_testValues` se encerre.

A condição de teste do **assert** é

```
TST_MUX_2 or (saidaMUX2 = esperadaMUX)
```

A constante `TST_MUX_2` está definida como `true` logo abaixo do vetor de testes para os modelos de dois bits:

```
constant TST_MUX_2 : boolean := true;
```

Por causa do `true or (...)`, o **assert** não efetua a comparação entre a saída e o valor esperado. O código foi escrito assim para diminuir a poluição na tela durante os testes dos modelos.

Para testar seu modelo, altere a constante respectiva (`TST_MUX_2`, `TST_MUX_4` ou `TST_MUX_8`) para `false` e então verifique os resultados.

Etapa 6 Para executar a simulação sem invocar `gtkwave`, diga

```
prompt: ./run_mux.sh
```

Verifique se há alguma mensagem de erro; se sim, leia a mensagem e use seus miolos antes de chamar o professor.

Para executar a simulação com o `gtkwave`, diga

```
prompt: ./run_mux.sh 1 &
```

Achtung: se o diagrama de tempo é idêntico ao da simulação funcional, sem mostrar atrasos de propagação, então edite `packageWires.vhd` e altere a constante `simulate_time` para 1. Esta constante está no topo do arquivo.

13.2.8 Nem mesmo um circuito simples é bem-comportado?

Etapa 7 Reveja a Seção-5.3.2. Verifique, cuidadosamente, as combinações de entradas e a saída do modelo mux2. A entradas são entr_2 (v_2.a(0) e v_2.a(1)) e o sinal de controle é s0 (v_2.s). Se os **asserts** correspondentes ao mux2 não são impressos, então o comportamento é o esperado. *Esperado* não é o mesmo que *correto*.

Contudo, no diagrama de tempos do gtkwave, os sinais s0, entr_2(1 **downto** 0) e saidamux2 indicam que há algo de podre no reino dos multiplexadores. Para gerar o diagrama de tempos para o gtkwave execute

```
./run_mux.sh 1 &
```

Observe os instantes nos quais as entradas se alteram. Uma vez identificado o problema, qual seria a solução? Reveja a Seção-5.3.2.

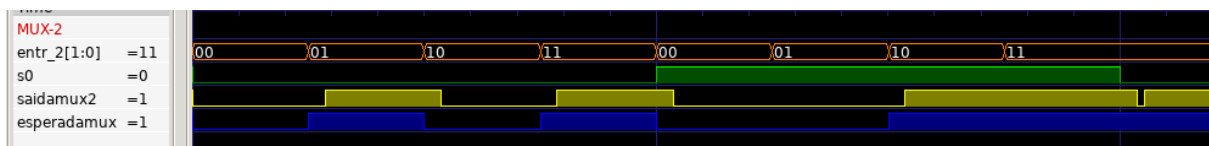


Figura 13.16: Pulsos transitórios na saída do multiplexador.

O pulso transitório – *indesejado porque viola a especificação funcional do mux-2* – ocorre no lado direito da Figura 13.16. Meça a duração deste pulso com os cursores do gtkwave, e procure em packageWires.vhd os tempos de propagação similares à duração do pulso. Por que ele ocorre?

Etapa 8 Conserte seu modelo do mux-2 para remover o pulso indesejado – o tal de *glitch* – e refaça os testes para garantir que a sua solução é mesmo correta.

Etapa 9 Uma vez que o funcionamento do mux-2 seja aquele especificado, você deve, lembrando das árvores projetadas em combin.pdf,

- (1) escrever⁵ o modelo para o mux-4. Para tanto edite o arquivo combin.vhd;
- (2) repetir para o mux-8;
- (3) repetir para o multiplexador de 8 entradas com um vetor de bits ao invés de oito escalares;
- (4) acrescentar mais elementos aos respectivos vetores de teste em tb_mux.vhd; e
- (5) verificar a corretude de seus modelos⁶.

Quantos vetores são necessários para testar exaustivamente um mux-4? São duas entradas de seleção e quatro entradas de dados, logo são necessários $2^2 \times 2^4$ testes. Use o material biológico que se encontra entre suas orelhas e veja se é mesmo necessário executar *todos* os 64 testes, ou se algo pode ser aproveitado dos testes do mux-2. Reveja a Seção-5.3.2.

Você quer mesmo escrever $2^3 \times 2^8 = 2^{11}$ vetores de teste para o mux-8? Como você convenceria o advogado do seu cliente de que um conjunto menor de testes é o *suficiente* para garantir a corretude do seu modelo?

Copie seus vetores de teste do laboratório passado (Seção 13.1), e se for o caso, acrescente novos elementos ao vetor.

⁵Você **pode** se lembrar do que fez no laboratório passado. Eu sei, o filme é uma b*st*.

⁶Para aqueles que tem memória de invertebrado: basta dizer “./run_mux.sh”, ler eventuais mensagens de erro, resolver os problemas e então dizer “./run_mux.sh 1 &”.

13.2.9 *Testbench* para os demultiplexadores e decodificadores

Reveja a Seção-5.3.2. O *script* `run_combin.sh` compila o código VHDL e gera um simulador para os modelos dos demultiplexadores e decodificadores. Este *script* é similar a `run_mux.sh`.

Etapa 10 Lembrando das árvores mostradas em `combin.pdf`, para esta etapa você deve:

- (1) escrever os modelos para o *demux-2*. Para tanto edite o arquivo `combin.vhd`;
- (2) repetir para o *demux-4* e *demux-8*;
- (3) acrescentar mais elementos aos respectivos vetores de teste em `tb_combin.vhd`; e
- (4) verificar a corretude de seus modelos⁷.

Achtung: no TB dos demultiplexadores e decodificadores, em `tb_combin.vhd`, um único registro é usado para testar os dois circuitos de largura N (*demux-N* e *decod-N*) ao mesmo tempo e portanto, dependendo do teste, alguns dos campos não são relevantes naquele teste.

No registro com os vetores de teste, o campo `dm` é o vetor de bits com a saída esperada para um demultiplexador quando recebe as entradas definidas pelos valores em `k,s`.

O campo `dc` é o vetor de bits com a saída esperada para um decodificador cujas entradas são definidas (somente) pelos valores em `s`.

Os **asserts** que verificam a corretude dos demultiplexadores e decodificadores estão comentados para reduzir a poluição na tela. Para testar seus modelos, altere a constante respectiva (`TST_DEMUX_2` ou `TST_DECOD_2`) para `false` e então verifique os resultados. Há uma tripla de constantes para cada circuito, *viz* `TST_DEMUX_2`, `TST_DEMUX_4` e `TST_DEMUX_8`.

Você é quem escreve os vetores de teste e portanto sua tarefa é ajustar os campos `dm` (saída esperada dos *demuxN*) e `dc` (saída esperada dos *decodN*). Se os valores que você atribuir àqueles campos forem incorretos para as entradas, então os **asserts** indicarão falso-positivos para erros inexistentes.

Etapa 11 Caso necessário, aplique sua solução para o problema hamletiano do *mux-2* aos modelos do *demux-2* e *decod-2*. Reveja a Seção-5.3.2.

Responda às questões na última página e as entregue ao professor até as 23:30 da terça de carnaval – envie fotos das respostas, ou PDFs.

Histórico das Revisões:

26fev21: Não esqueça da minha caloi: *Reveja a Seção-5.3.2*.
 22jun20: remoção do material que está em `vhdl.pdf`, separação dos TBs para mux e demux;
 10out16: acréscimo de **with-select**
 03set16: vetores de testes separados por tamanho de circuito;
 25ago16: sugestões de Zanata, troca seletor para decodificador;
 22sep15: modelo de temporização com T_c ;
 14sep15: definição de circuito combinacional;
 30jul15: remoção de modelagem estrutural, mais abstração;
 13ago13: inserção de `lstinline`, revisão no texto;
 20mar13: segunda versão;
 01nov12: primeira versão.

EOF

⁷Reveja a nota de rodapé número 6.

13.2.10 Questionário

Nome:

Nome:

As questões abaixo devem ser respondidas e entregues ao professor. Os problemas podem ser respondidos em dupla; os dois nomes devem estar na folha de respostas.

Etapa 12 Leia o código VHDL dos modelos e estime os tempos de propagação dos 9 circuitos, tomando como base os tempos de propagação das portas lógicas em `packageWires.vhd`. Mostre suas contas.

Etapa 13 Repita para o tempo de contaminação dos nove circuitos. Mostre suas contas.

Etapa 14 Compile os modelos e verifique seu funcionamento com `gtkwave`. Verifique as diferenças nos tempos de propagação das versões de 2, 4 e 8 entradas dos multiplexadores, demultiplexadores e decodificadores. Use os cursores do `gtkwave` para medir os tempos.

Etapa 15 Confirme, com base nos diagramas de tempo, se os tempos medidos são similares aos que você estimou. Justifique as diferenças nas suas estimativas.

Veja a Seção 13.3.4 de `vhdl.pdf` para responder às próximas perguntas. Copie o ‘miolo’ dos seus modelos na folha de respostas.

Etapa 16 Implemente um `mux-8` com o comando **when–else** e verifique seu modelo. Acrescente o novo modelo ao arquivo `combin.vhd` *após* a definição da arquitetura do `mux-8` – VHDL usa a última arquitetura que encontra após a definição da entidade.

Etapa 17 Implemente um `mux-8` com o comando **with–select** e verifique seu modelo. Acrescente o novo modelo ao arquivo `combin.vhd` *após* a definição da última arquitetura do `mux-8`.

13.3 Somadores

Objetivos: são três os objetivos deste laboratório: (i) compreender e apreender as bases do processo de depuração de modelos dos três somadores; (ii) medir e comparar seus tempos de propagação; e (iii) construir um modelo para um somador do tipo “seleção do vai-um” (*carry-select adder*), verificar sua corretude, e medir seu tempo de propagação.

O trabalho pode ser efetuado em duplas.

Preparação: leia as Seções 6.6, 6.7 e 6.9 de *aritm.pdf*.

13.3.1 Material Disponibilizado Para Sua Tarefa

Etapa 18 Copie para sua área de trabalho o arquivo com o código VHDL:

(a) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_somadores.tgz`

(b) expanda-o com: `tar xzf l_somadores.tgz`

(c) mude para o diretório recém criado: `cd somadores`

O *script* `run.sh` compila o código VHDL e produz um simulador. Se executado sem nenhum argumento de linha de comando, `run.sh` somente (re)compila o simulador e o executa; com um argumento o *script* também dispara a execução de `gtkwave`: `./run.sh 1 & .`

13.3.2 Somadores com Adiantamento de Vai-um

O arquivo `packageWires.vhd` contém definições de abreviaturas para nomes de sinais, e constantes para os tempos de propagação das portas lógicas. Neste laboratório todos os valores são do tipo `bit`, com valores em $\mathbb{B}$.

O arquivo `aux.vhd` contém os modelos das portas lógicas *inv*, *and*, *or*, *xor*, que são os componentes básicos para este laboratório. Este arquivo não deve ser editado.

O arquivo `somador.vhd` contém três modelos de somadores, com complexidade e desempenho crescentes, a saber *adderCadeia*, *adderAdiant4* e *adderAdianta16*, descritos em *aritm.pdf*.

O modelo *adderCadeia* é um modelo estrutural do somador composto por 16 instâncias do somador completo (*full-adder*), sem circuito de adiantamento de vai-um.

O modelo *adderAdianta4* estende o modelo *adderCadeia* com cadeias de adiantamento de vai-um, para grupos de 4 bits.

O modelo *adderAdianta16* estende o modelo *adderAdianta4* com uma cadeia de adiantamento de vai-um de 16 bits.

O arquivo `tb_somador.vhd` contém o programa de testes (*testbench*, ou TB) para verificar a corretude dos modelos. A arquitetura do TB declara os componentes que serão testados e um **record** que é usado para excitar os modelos. O registro `test_record`, mostrado no Programa 13.53, possui cinco campos e os valores destes campos devem ser atribuídos de forma a gerar todas (*todas?*) as combinações de entradas para garantir a corretude do modelo. O vetor de testes `test_array` contém alguns elementos para ilustrar as possibilidades.

Programa 13.53: Vetor de valores de entrada para testar os modelos.

```

type test_record is
  record
    a : reg16;      — entrada
    b : reg16;      — entrada
    f : bit;        — operação: 0=ADD, 1=SUB
    c : reg16;      — saída esperada
    v : bit;        — vai-um/empresta-um esperado
  end record;

type test_array is array(positive range <>) of test_record;

constant test_vectors : test_array := (
  — a,      b,      f, c,      vai-um
  — testes para soma
  (x"0000", x"0000", '0', x"0000", '0'),
  — acrescente novos vetores aqui, testes para soma f=0
  (x"0001", x"0001", '0', x"0002", '0'),
  (x"0010", x"0010", '0', x"0020", '0'),
  ...
  — acrescente novos vetores aqui, testes para subtração f=1
  (x"0000", x"0000", '1', x"0000", '1')
  ...
);

— troque a constante para FALSE para testar seus modelos
constant TST_CADEIA    : boolean := false;
constant TST_ADIAANT4  : boolean := true;
constant TST_ADIAANT16 : boolean := true;

```

No `test_record`, os campos `a`, `b`, `c` são vetores de bits codificados em hexadecimal (`x"0FA4"`) e contém as parcelas da soma e o valor esperado para o resultado. O campo `f` determina a função do circuito: se `f='0'`, o circuito efetua somas; se `f='1'`, o circuito efetua subtrações. O campo `v` contém o vai-um esperado. Nas subtrações o valor de `v` deve ser complementado porque este bit se comporta como empresta-um, ao invés de vai-um.

Para testar cada modelo de somador, altere a respectiva constante para `false` e então verifique os resultados. As constantes são `TST_CADEIA`, `TST_ADIAANT4` ou `TST_ADIAANT16`.

A sequência de testes é implementada no processo `U_testValues`, com um laço **for ... loop**. A variável de iteração itera no espaço definido pelo número de elementos do vetor de testes (`test_vectors'range`) – o atributo `'range` representa a faixa de valores do índice do vetor. Se mais elementos forem acrescentados ao vetor, o laço executará mais iterações. O elemento do vetor é atribuído à variável `v` e os vários campos do vetor são então atribuídos aos sinais que excitam os modelos. O processo `U_testValues` executa concorrentemente com os modelos dos somadores e, quando os sinais de teste são atribuídos no laço, estes provocam alterações nos sinais das entradas dos modelos.

O comando **assert** verifica se a saída observada do somador é igual à saída esperada. Se os valores são iguais, o comportamento é o esperado, e portanto *correto, do ponto de vista dos vetores de teste que você escreveu*. Note que se você escolher valores de teste inadequados, ou errados, o diagnóstico de eventuais problemas no modelo pode ser difícil.

Ao final do laço, a simulação termina no comando **wait**, que faz com que a execução do simulador se encerre.

O arquivo `s.vcd` contém definições para o gtkwave tais como a escala de tempo e sinais a serem exibidos na tela para a verificação dos modelos dos 3 somadores: os sinais no topo são do somador sem adiantamento de vai-um, ao centro são os sinais do somador com adiantamento de 4 em 4 bits, e embaixo os sinais do somador com adiantamento de 4 em 4, e de 16 em 16 bits.

Etapa 19 Os modelos no arquivo `somador.vhd` contém erros que você deve encontrar e corrigir. Os erros estão nos modelos dos somadores. Você deve ampliar o vetor de testes para explicitar os erros.

Acrescente ao arquivo `tb_somadores.vhd` os vetores de teste para verificar a corretude dos três somadores. Acrescente tantas tuplas quantas forem necessárias à `test_vectors` para que se possa ter um mínimo de confiança na corretude do projeto.

Cada um dos modelos contém um erro e sua tarefa é gerar vetores de teste que evidenciem os erros para então corrigir os modelos. Os erros estão (somente) nos modelos dos somadores.

Que valores usar nos testes?

Revise a Seção 6.9. Os valores escolhidos para os testes ajudam a evidenciar os locais onde, provavelmente, há um erro no modelo. É imprescindível ler com atenção as saídas de texto do simulador para descobrir a eventual inconsistência em cada um dos testes. A posição do bit em erro indica a posição no modelo que pode estar errada.

13.3.3 Temporização

Os modelos dos somadores contém informação de temporização.

No topo do arquivo `packageWires.vhd` estão as declarações das constantes com o tempo de propagação das portas lógicas definidas em `aux.vhd`. Edite `packageWires.vhd` e altere a definição da constante `simulate_time`, no topo do arquivo, de 0 para 1. Note que as constantes que definem os tempos de propagação estão multiplicadas por um número que pode ser zero ou um. Recompile os modelos e verifique seu funcionamento com gtkwave.

A Figura 13.17 mostra a tela do gtkwave com uma simulação dos três somadores. As barras de cor ciano indicam o tempo de propagação dos sinais, desde a mudança nas entradas, até que a saída estabiliza. Por “estabiliza” entenda-se que *ambos*, a soma e o vai-um estabilizaram.

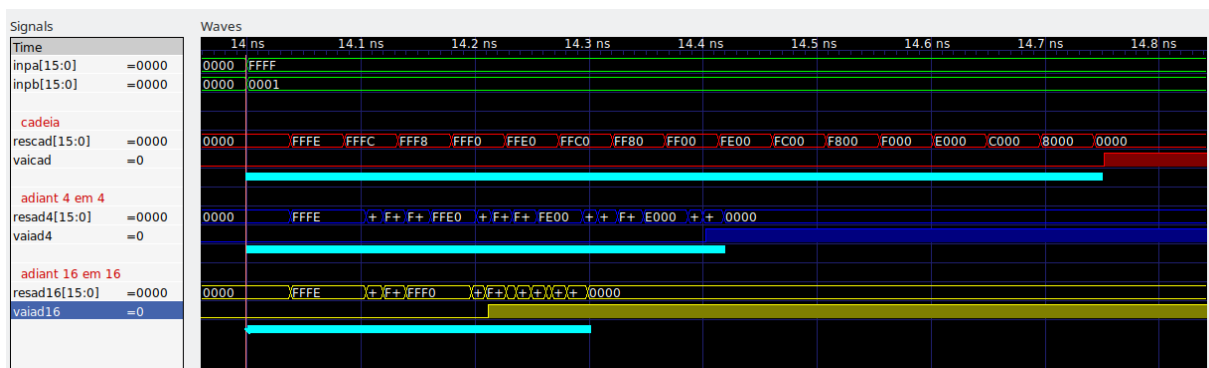


Figura 13.17: Tempo de propagação dos três somadores.

A saída do somador sem adiantamento (*cadeia*) passa por 15 estágios, e em cada estágio, um bit dos menos significativos assume o seu valor definitivo. No somador com adiantamento de 4 bits, também são 15 estágios, e em quatro deles se percebe que um quarteto inteiro assumiu seu valor definitivo. A saída do somador com adiantamento de 16 bits, por causa dos vários caminhos de adiantamento, embaralha a evolução do resultado ao longo do tempo. Ajustando-se a escala de tempo horizontal (*zoom*) é possível que todos os estágios fiquem aparentes.

Lembre que o *tempo de propagação* é aquele do caminho mais longo entre entradas e saídas. No caso de somadores, o caminho mais longo inicia no *vem-um* e termina, ou no *vai-um*, ou no bit mais significativo.

Achtung: o tempo de propagação pode mudar em função dos valores de teste; escolha valores das entradas que atravessam o caminho crítico provocando alteração em todos os bits da saída. O par (x0001, xffff) não é o que produz o pior tempo de propagação.

Etapa 20 Meça e compare os tempos de propagação dos três somadores.

Não esqueça de entregar os resultados das medições ao professor. Use os cursores do gtkwave para medir os tempos de propagação.

- (1) Quais são os valores para as entradas que explicitam os piores casos do tempo de propagação? Justifique sua resposta.
- (2) Quais os ganhos de desempenho dos modelos com adiantamento, com relação ao modelo sem adiantamento? Mostre como efetuou a comparação.

Espaço em branco proposital.

13.3.4 Somador de 32 bits – *carry select adder*

Reveja a Seção 6.7 de aritm.pdf.

Nos diagramas, os números inteiros são denotados por cadeias de bits $a_{n-1}a_{n-2}\cdots a_0$ que representam o número $a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \cdots + a_0 \cdot 2^0$.

Uma alternativa de projeto para a implementação de somadores é mostrada na Figura 13.18, que contém um somador de 8 bits construído com três somadores de 4 bits. A parte menos-significativa do resultado é obtida pela soma dos dois operandos:

$$\langle V_3 S_{3..0} \rangle = A_{3..0} + B_{3..0} + vem_0.$$

A parte mais-significativa é obtida em paralelo com a parte menos-significativa, computando-se simultaneamente as duas alternativas, uma com $vem_4 = 0$, e a outra com $vem_4 = 1$.

Quando os sinais se propagam através do somador da metade menos significativa e o valor de $V_3 = vai_3$ fica estável, este é usado para escolher uma das duas somas para a metade mais significativa, com o multiplexador de 4 bits de largura. Este circuito é chamado de “somador com seleção de vai-um” (*carry select adder*).

Verifique no modelo se a função lógica que determina o valor de vai_8 está correta.

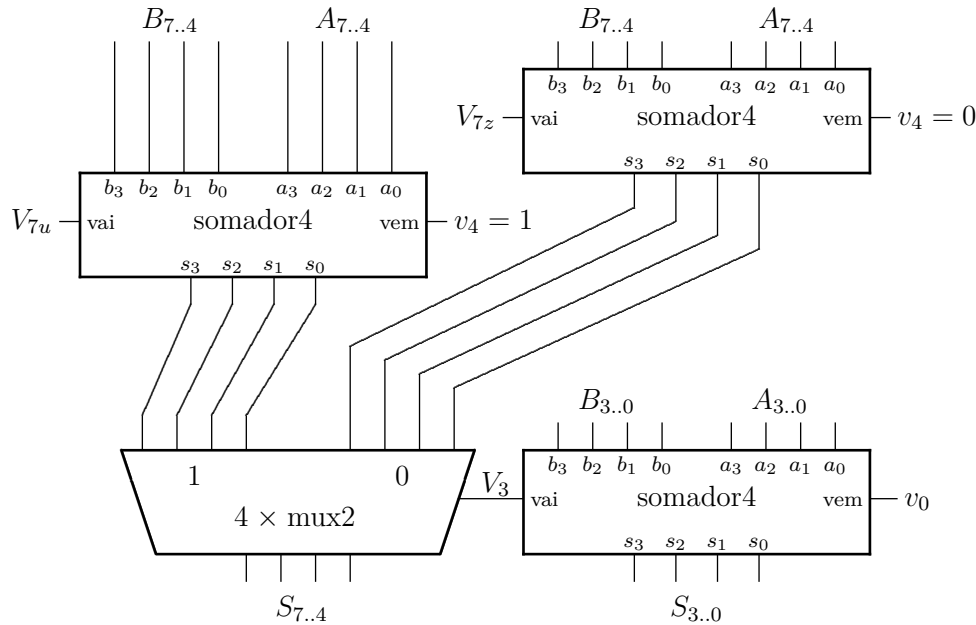


Figura 13.18: Somador com seleção de vai-um.

Etapa 21 Sua tarefa é escrever um modelo de somador com seleção de vai-um com 32 bits de largura. Acrescente seu código VHDL ao final do arquivo `somadores.vhd`. O *testbench* a ser usado nesta etapa é `tb_csa.vhd`, que já contém alguns dos vetores de teste para seu modelo. Para executar os testes, invoque o script `./runCSA.sh`, que compila e executa a simulação.

O vetor de testes `test_array` foi ajustado para os operandos de 32 bits e é mostrado no Programa 13.54. Esse modelo efetua somente somas.

Programa 13.54: Vetor de testes para o somador de 32 bits.

```

type test_record is
  record
    a : reg32;      — entrada
    b : reg32;      — entrada
    c : reg32;      — saída
    v : bit;        — vai-um
  end record;

type test_array is array(positive range <>) of test_record;

constant test_vectors : test_array := (
  — a, b, c, vai-um
  — testes para soma
  (x"00000000", x"00000000", x"00000000", '0'),
  — acrescente novos valores aqui
  (x"00000001", x"00000001", x"00000002", '0'),
  (x"00000fff", x"00000001", x"00001000", '0'),
  ...
);

```

Etapa 22 O tempo de propagação do somador com seleção de vai-um é mais curto do que um circuito com dois somadores de 16 bits ligados em série? Nesta etapa, para efetuar a comparação, suponha que dois somadores de 16 bits rápidos são ligados em série e o vai-um do primeiro (V_{15}) é ligado diretamente à entrada vem-um do segundo.

Entregar até as 07:30 do dia 24fev (quarta-feira).

Histórico das Revisões:

22fev2021: adição do diagrama de tempos;
 10fev2021: revisão no texto;
 02set2019: revisão no texto; remoção do multiplicador;
 23ago2018: pequenos ajustes c.r.aos testes;
 30ago2017: revisão no texto;
 31ago2016: adição de tb_csa.vhd e runCSA.sh;
 03set2014: melhora no texto sobre testes; mult para dever de casa;
 05mar2014: adição de texto sobre vetores de teste;
 19ago2013: revisão do texto, adição do carry-select adder;
 20mar2013: segunda versão;
 01nov2012: primeira versão.

13.4 Modelagem de Circuitos Sequenciais em VHDL

São quatro os objetivos deste laboratório: (i) apreender o conceito de *processo* em VHDL; (ii) efetuar a modelagem em VHDL de sistemas sequenciais síncronos; (iii) escrever um modelo para uma máquina de estados finita e (iv) verificar, através de simulação, a corretude do modelo. O trabalho pode ser efetuado em duplas.

A Seção 13.4.6 contém a(s) tarefa(s) por resolver nesta aula e que deve(m) ser enviada(s) ao professor.

13.4.1 Revisão: como funciona o mecanismo de simulação

Reveja a Seção 9.2.5.

13.4.2 Comandos Sequenciais

Leia a Seção 9.4.2.

13.4.3 Processos

Leia a Seção 9.4.3.

13.4.4 Comandos Sequenciais

Leia a Seção 9.4.11.

13.4.5 Modelagem de máquinas de estado

Leia a Seção 9.4.12

Espaço em branco proposital.

13.4.6 Da tarefa:

Etapa 23 Projete uma Máquina de Moore, com uma entrada de dados D , uma saída C , uma entrada de relógio rel , e uma entrada de $reset$. Após a inicialização, sempre que o circuito detectar a sequência $\dots 01111110 \dots$ (0, seis 1, 0) a saída C produz um pulso com duração de um ciclo.

D: 001110111111001111110001010

C: 000000000000010000000100000

A sequência $\dots 011111101111110 \dots$ produz um único pulso na saída C .

Etapa 24 Copie para sua área de trabalho o arquivo com o código VHDL:

(a) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_processos.tgz`

(b) expanda-o com: `tar xzf l_processos.tgz`

o diretório `processos` será criado;

(c) mude para aquele diretório: `cd processos`

Ao contrário dos outros laboratórios, neste você deve escrever seu código VHDL para a máquina de estados diretamente no arquivo `tb_me.vhd`. Você deve consertar e completar o esqueleto que está no TB, acrescentando mais estados ao tipo `states` e cláusulas ao **case** do processo `U_st_transitions`. Uma vez completado seu modelo, e gerados os casos de teste que **comprovem** sua corretude – o vetor de testes deve, pelo menos, triplicar de tamanho – envie o código do `tb_me.vhd` para o seu professor, com assunto `ci1210-ME`, e os nomes dos dois autores, até as 23:59 do dia 27fev.

EOF

Histórico das Revisões:

02set2019: remoção de VHDL;

02set2019: revisão do texto;

06set2018: revisão do texto;

19set2017: revisão do texto, formatação de exemplos de código;

07set2017: revisão do texto, acréscimo de exemplos;

16set2016: revisão do texto por Marco Zanata;

12set2016: acréscimo de exemplos, melhorias no esqueleto do código VHDL;

01out2015: revisão do texto;

19set2015: primeira versão.

13.5 Máquinas de Estado Finitas em VHDL

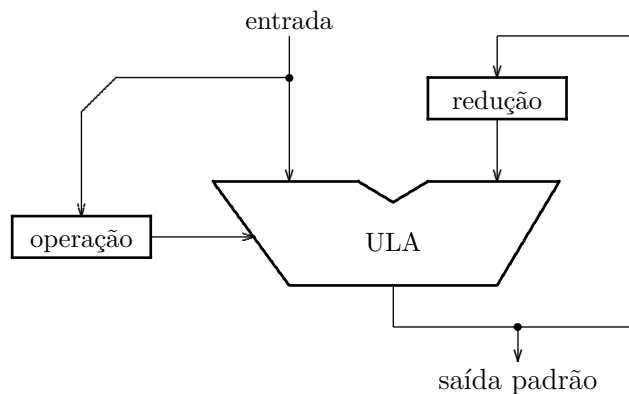
São dois os objetivos deste laboratório: (i) escrever um modelo para uma máquina de estados finita para controlar um circuito que não é trivial; e (ii) verificar, através de simulação, a correteza do seu modelo.

Sua tarefa é projetar, implementar em VHDL, e testar a ME que controla um circuito que efetua a redução de uma sequência de vetores. Por *redução* entende-se que, para a operação associativa $\Theta \in \{\oplus, \wedge, \vee, +, \times\}$, e entrada com 8 palavras de 32 bits $d_i, i \in [0, 7]$, o circuito computa

$$x = \Theta_0^7 d_i.$$

O *testbench* percorre um vetor com uma sequência de inteiros, composta por palavras de sincronismo, um comando (*oper*) e vetores de dados com 8 palavras, seguido por uma ou mais palavras de sincronismo. Após a leitura da oitava palavra, o circuito exibe a redução daquele vetor na saída padrão.

Você receberá um *testbench* com código que percorre o vetor de entradas e que escreve na saída padrão do simulador, além de um modelo completo do circuito de dados. O sistema está representado no diagrama abaixo. Sua tarefa é implementar a máquina de estados que controla o circuito de dados.



As operações para a redução, e seus respectivos códigos são:

0 XOR

1 AND

2 OR

3 SOMA

4 MULT

A palavra de sincronismo é sync=0xffff.ffff.

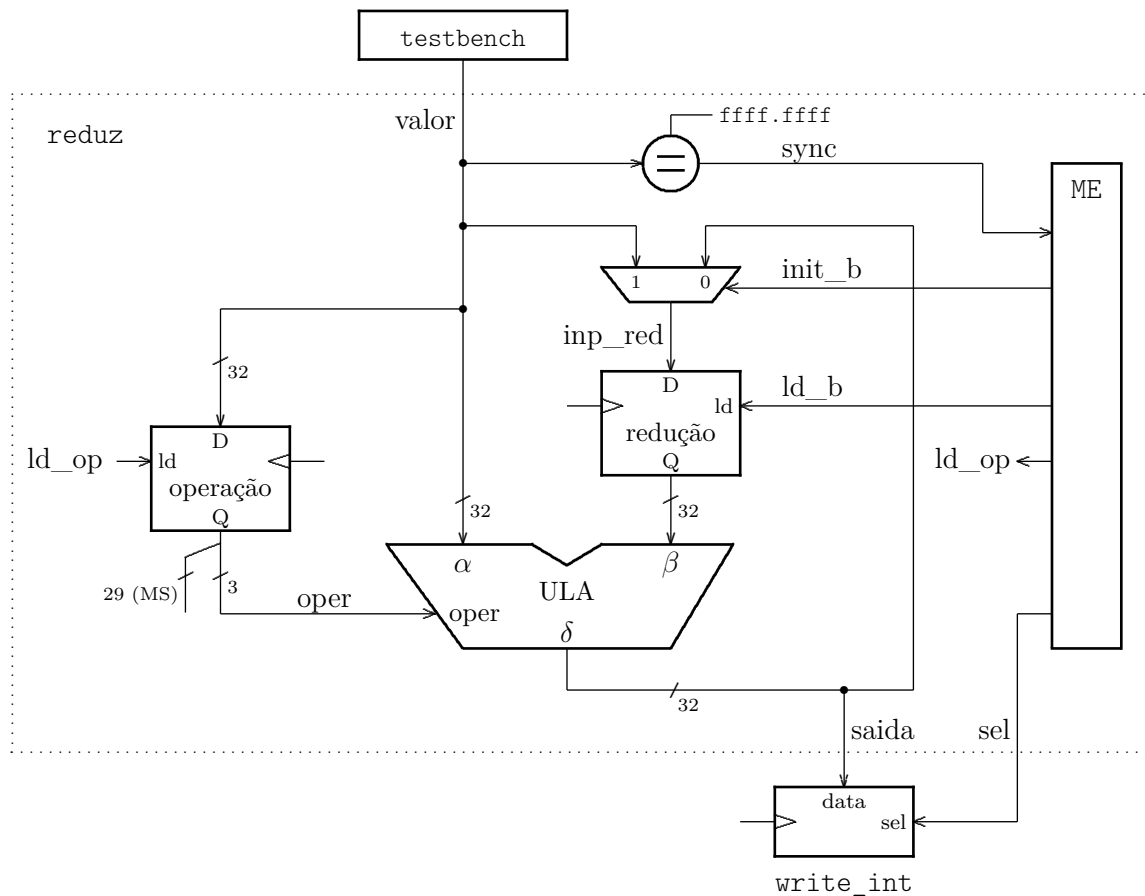
Uma sequência de entrada é algo como:

```
ffff.ffff  ffff.ffff  0000.0003  00ff.00ff  ff00.5544  ...  1122.3344  ffff.ffff  ffff.ffff
sync       sync      oper +      d0      d1      ...      d7      sync      sync
```

O valor da redução deve ser emitido durante o sync após d_7 . Se um valor ffff.ffff for observado numa sequência de dados, este deve ser tratado como dado e não como um sync.

Espaço em branco proposital.

O diagrama abaixo mostra o circuito de dados completo, com todos os sinais de controle que devem ser gerados pela sua ME.



Os arquivos `aux.vhd`, `tb_maqEst.vhd` e `packageWires.vhd` contêm os componentes necessários para o seu projeto. Você deve editar `reduz.vhd` e acrescentar o que for necessário para ‘implementar’ a máquina de estados que controla o circuito de dados.

O *script* `run.sh` compila os arquivos VHDL e dispara a simulação. Diga `./run.sh -h` para ver as opções de linha de comando.

Para o vetor de testes no TB, a saída correta da simulação é mostrada abaixo.

```
kipling:/maqEst # ./run.sh
00009d80
99999999
ffffffff
00001111
ffff1111
```

Espaço em branco proposital.

Da tarefa:

1. O trabalho pode ser efetuado em duplas; **entrega em 03mar**;
2. você deve remeter por e-mail para seu professor somente o arquivo com `reduz.vhd`;
3. copie para sua área de trabalho o arquivo com o código VHDL:
 - (a) `wget http://www.inf.ufpr.br/roberto/ci210/vhdl/l_maqEst.tgz`
 - (b) expanda-o com: `tar xzf l_maqEst.tgz`
o diretório `maqEst` será criado;
 - (c) mude para aquele diretório: `cd maqEst`
4. PLÁGIO NÃO SERÁ TOLERADO. É do interesse geral que vocês conversem sobre o projeto mas cada dupla deve escrever seu próprio código;
5. assegure-se de que entendeu a especificação antes de iniciar o projeto da ME.

EOF

13.5 Programação em *Assembly* do MIPS

Objetivos: São dois os objetivos deste laboratório: (i) aprender usar o simulador MARS; e (ii) escrever e testar um programa completo em *assembly*.

Preparação: Veja MARS_Tutorial.pdf e MARS_features.pdf em <http://www.inf.ufpr.br/roberto/ci210/assembly> .

Entrega: 12mar2021.

13.5.1 Programa que Computa a Série de Fibonacci

Copie o simulador de <https://courses.missouristate.edu/KenVollmar/MARS/> -> Download

Armazene o arquivo em /opt:

```
sudo mv ~/Downloads/Mars4_5.jar /opt
```

e ajustes as permissões:

```
sudo chmod 755 /opt/Mars4_5.jar
```

Para executar o simulador MARS diga:

```
java -jar /opt/Mars4_5.jar
```

Possivelmente, seria uma boa ideia acrescentar ao seu ~/.bashrc a função que contenha esta linha de comando. Edite ~/.bashrc e acrescente o seguinte, preferencialmente no final do arquivo:

```
function mars() { java -jar /opt/Mars4_5.jar "$@" ; }
```

Isso feito, diga `source ~/.bashrc` e então invoque o simulador dizendo apenas `mars` .

Copie <http://www.inf.ufpr.br/roberto/ci210/assembly/fibonacci.s> para a sua área de trabalho e siga as instruções em MARS_Tutorial.pdf.

Espaço em branco proposital.

13.5.2 Programa que Computa o Fatorial

Copie <http://www.inf.ufpr.br/roberto/ci210/assembly/fatorial.s> para a sua área de trabalho e verifique se o programa produz resultados corretos. Se encontrar algum erro, **corrija-o** e verifique sua solução.

```
wget http://www.inf.ufpr.br/roberto/ci210/assembly/fatorial.s
```

Programa 13.53: fatorial.s

```

1  # void main(void) { // fatorial iterativo
2
3  #    int i,j;
4      .data                # início da seção de dados
5  vi:    .space 1*4        # aloca espaço para vars globais
6  vj:    .space 1*4        #    vi e vj são apenas para exemplificar
7
8  resp:   .asciiz "fatorial de 5="      # string com '\0'
9
10
11      .text                # início da seção de código
12      .globl main          # define main como nome global
13  main:
14
15  #      j=1;
16      addi $t1, $zero, 1      # este é o endereço de main()
17
18  #      for(i=1; i <= n; i++)
19      addi $t4, $zero, 1
20
21  for:    slti $t7, $t4, 6      # n < 6    fat(5)
22          beq $t7, $zero, fimfor # t7 == false -> fimfor
23
24  #      j = j*i;
25      mult $t1, $t4            # resultado em 64 bits (HI & LO)
26      mflo $t4                # seleciona 32 bits menos sign
27
28      addi $t4, $t4, 1
29
30      j for
31
32  fimfor: addi $v0, $zero, 4      # imprime resposta
33          la $a0, resp          # syscall(4) = imprime string
34          syscall
35
36          addi $v0, $zero, 1      # imprime inteiro
37          addu $a0, $zero, $t1    # syscall(1) = imprime inteiro
38          syscall
39
40  #      return(0);
41      li $v0, 10                # termina programa
42      syscall                    # syscall(10) = termina programa
43
44  # }

```


13.5.3 Programa para Copiar *Strings*

Traduza o Programa 13.54 para *assembly* do MIPS e verifique sua corretude com MARS. Use o código que imprime inteiros em `fibonacci.s` como modelo para o `printf()`, alterando o tipo de saída de inteiro para *string*. Os códigos das *syscalls* estão definidos em

Help → MIPS → Syscalls.

A instrução **lb** *rt,desl(rs)* (*load-byte*) carrega o byte apontado por (`extSinal(desl)+rs`) no registrador *rt*, estendendo o sinal (*b₇*).

A instrução **lbu** *rt,desl(rs)* (*load-byte unsigned*) é similar à **lb** mas não estende o sinal do byte carregado; os três bytes mais significativos são preenchidos com zero.

A instrução **sb** *rt,desl(rs)* (*store-byte*) armazena o byte menos significativo em *rt* no endereço apontado por (`extSinal(desl)+rs`).

Para alocar as *strings* em memória use a linha 8 do Programa 13.53 como exemplo. A diretiva `.asciiz` aloca uma *string* incluindo o `'\0'`, enquanto que `.ascii` aloca uma *string* sem o `'\0'`. Veja o tutorial do Mars para a lista das diretivas que este provê, ou

Help → MIPS → Directives.

Note que as diretivas aceitas pelo montador do Mars são diferentes daquelas aceitas por `mips-as`.

Programa 13.54: `strcpy.c`

```

1  char fte[16]="abcd-efgh-ijkl-";
2  char dst[16]= { '\0' };           // inicializa com 16 '\0'
3
4  void main (void) {
5      int i,f,n;
6
7      i = 0;                        // inclui '\0' na contagem
8      // copia e computa tamanho da cadeia, inclusive '\0'
9      while( (dst[i] = fte[i]) != '\0' ) // atribui e então compara
10         i = i + 1;
11
12     i = i + 1;                     // inclui '\0' na contagem
13
14     // sua versão assembly de printf() deve ter um
15     // número fixo de argumentos e usar as syscalls do MARS
16     printf("fonte:_%s_\n", fte);
17     printf("dest:_%s_\n", dst);
18     printf("tam:_%d_\n", i);
19 }
```

Num programa em C, as variáveis são declaradas, tipicamente no início do programa, e então escrevemos o código das funções. Em *assembly*, o espaço em memória para as variáveis é alocado na seção `.data`, através de diretivas tais como `.space`, `.byte` e `.word`. Por exemplo, um vetor para 32 caracteres, chamado de `buf`, é declarado com

```

.data          # seção com dados
buf: .space 32  # char buf[32];
```

Seu programa em *assembly* pode referenciar este vetor com

```
.text          # seção com código
...
la    t0, buf      # t0 = &(buf[0]);
lbu   t5, 0(t0)    # t5 = *t0;
addi  t0, t0, 1    # t0++;
```

13.5.4 Mais do Fatorial

Este exercício é para aqueles que já utilizaram o Mars em outras disciplinas, e que chegaram a esta tarefa com tempo disponível. Para aqueles que utilizam Mars pela primeira vez, este exercício deve ser tentado fora do horário de aula.

Traduza o Programa 13.55 para *assembly* do MIPS e verifique sua corretude com MARS.

Programa 13.55: Duas versões do fatorial iterativo

```
1 void main (void) {
2     int i,f,n;
3
4     n = 5;
5     i = f = 1;
6
7     do {
8         f = f * i;
9         i = i + 1;
10    } while (i <= n);
11
12    printf("%d_ %d\n",n,f); // use syscalls do MARS
13
14    f = 1;
15    i = n;
16    while (i > 0) {
17        f = f * i;
18        i = i - 1;
19    }
20    printf("%d_ %d\n",n,f); // use syscalls do MARS
21
22    return(0);
23 }
```

Histórico das Revisões:

09mar21: alocação em /opt no período especial;

21nov19: .space ao invés de .size;

09nov17: correção em strcpy, defs de lb,lbu, .data .text;

15jan16: programas com números de linha;

17mar14: terceira versão, adição para alunos que já usaram MARS;

23out13: segunda versão, sem funções;

08jun13: primeira versão.

13.6 Programação em *Assembly* do MIPS – Funções

Objetivos: escrever e testar programas com funções em *assembly*.

Preparação: veja MARS_Tutorial.pdf e MARS_features.pdf em <http://www.inf.ufpr.br/roberto/ci210/assembly> .

Entrega: 15mar2021.

13.6.1 Programa para Copiar *Strings*, Versão com Função

Traduza o programa abaixo para *assembly* do MIPS e verifique sua corretude com MARS. Escreva uma função `prints()` que imprime uma *string*, e uma função `printi()` que imprime um inteiro, e use estas duas funções para “traduzir” os `printf()`s.

Programa 13.56: `strcpy.c`

```
1  char fte[16]="abcd-efgh-ijkl-";
2  char dst[16]= { '\0' };           // inicializa vetor com '\0'
3
4  // retorna tamanho da cadeia, inclusive o '\0'
5  int strcpy(char fte[], char dst[]) {
6
7      int i=0;
8
9      while ( (dst[i] = fte[i]) != '\0' ) // atribui e entao compara
10         i++;                          // inclui p '\0' na contagem
11     return(i+1);
12 }
13
14 void main (void) {
15     int j;
16
17     j = strcpy(fte,dst);
18
19     // sua versao assembly de printf() deve ter um
20     // numero fixo de argumentos
21     printf("fonte:_%s_\n", fte);
22     printf("dest:_%s_\n", dst);
23     printf("tam:_%d_\n", j);
24 }
```

13.6.2 Programa que Computa o Fatorial com Recursão

Traduza o programa abaixo para *assembly* do MIPS e verifique sua corretude com MARS.

Programa 13.57: fatorial.c

```
1  int fat(int n) {
2      if (n > 0)
3          return (n * fat(n-1));
4      else
5          return 1;
6  }
7
8  void main (void) {
9      int x,y;
10
11     y = 5;
12     x = fat(y);
13     printf("y_□%d\n", y);
14     printf("x_□%d\n", x);
15 }
```

13.6.3 Programa que Eleva um Inteiro a uma Potência Inteira

Traduza o programa abaixo para *assembly* do MIPS e verifique sua corretude com MARS.

Programa 13.58: power.c

```
1  int power(int n,int exp) {
2      if (exp > 1)
3          return (n * power(n, exp-1));
4      else
5          return (n);
6  }
7
8  void main (void) {
9      int x,y,z;
10
11     y = 5;
12     z = 3;
13     x = power(y,z);
14     printf("y_□%d\n", y);
15     printf("z_□%d\n", z);
16     printf("x_□%d\n", x);
17 }
```

Histórico das Revisões:

15jan15: revisão do texto;

23out13: primeira versão.