

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, 2012-2020. Versão de 23 de novembro de 2020.

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 , $i \in [0, k)$.

$$N = \sum_{i=0}^{k-1} 2^i \cdot b_i \tag{6.2}$$

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 $R = (-1 - A)$?

$$\begin{array}{rcccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & -1 \\ - & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & -A \\ \hline & r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0 & R \end{array}$$

Computando a subtração bit a bit, iniciando por a_0 , temos dois casos:

(i) se $a_0 = 0$, então $r_0 = 1 - 0 = 1 = \overline{a_0}$;

e

(ii) se $a_0 = 1$, 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,

$$-A = 1 + \overline{A} \tag{6.3}$$

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.

$$X = (x_7 \cdot -2^7) + x_6 \cdot 2^6 + x_5 \cdot 2^5 + \dots + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0 \tag{6.4}$$

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.

		posição	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
valor		peso	-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.

$$\begin{array}{rcccccc}
 & \underline{0} & {}^{1+0} & {}^{1+1} & 1 & 1 & 7 \\
 + & \underline{0} & 0 & 1 & 1 & 0 & +6 \\
 \hline
 & \underline{0} & 1 & 1 & 0 & 1 & +13 \\
 & \underline{r_4} & r_3 & r_2 & r_1 & r_0 &
 \end{array}$$

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

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*.

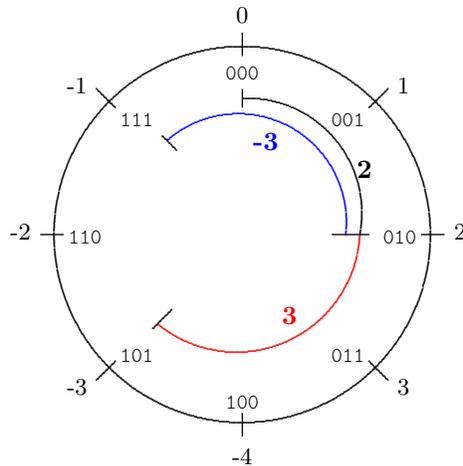


Figura 6.2: Representação em complemento de dois e *overflow*.

A representação para inteiros em complemento de dois é usada na imensa maioria dos dispositivos que efetuam computação com números inteiros. Uma representação com k bits têm as seguintes propriedades interessantes:

- o intervalo dos naturais representados é $\mathbb{N}^k : [0, 2^k - 1]$;
- o intervalo dos inteiros representados é $\mathbb{Z}^k : [-2^{k-1}, 2^{k-1} - 1]$;
- uma única representação para zero, com todos os k bits em 0;
- na representação de inteiros, o bit mais significativo é o *bit de sinal*: se $n_{k-1} = 1$, o número é negativo;
- o mesmo circuito efetua adições de naturais e de inteiros; e
- $-A = \overline{A} + 1$.

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.

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. ◁

Tabela 6.2: Exemplos de representação de frações em 3+5 bits.

valor	posição								
	<i>b</i> ₇	<i>b</i> ₆	<i>b</i> ₅	.	<i>b</i> ₄	<i>b</i> ₃	<i>b</i> ₂	<i>b</i> ₁	<i>b</i> ₀
	peso	–4	2	1	2 ^{–1}	2 ^{–2}	2 ^{–3}	2 ^{–4}	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 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.

$$\begin{aligned} s &= a \oplus b \oplus vem \\ vai &= (a \wedge b) \vee (a \wedge vem) \vee (b \wedge vem) \end{aligned} \tag{6.5}$$

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.

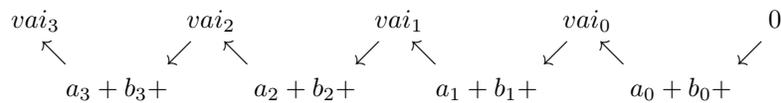


Figura 6.3: Adição de dois números representados em 4 bits.

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.

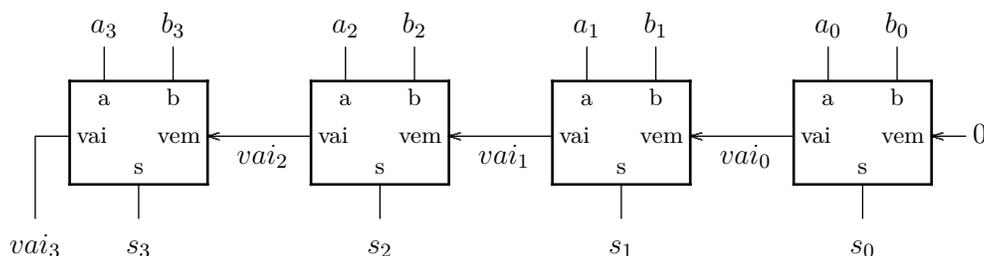


Figura 6.4: Circuito somador de quatro bits.

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 $t_0 + 8$. Para outras entradas, o tempo de propagação pode ser diferente. ◀

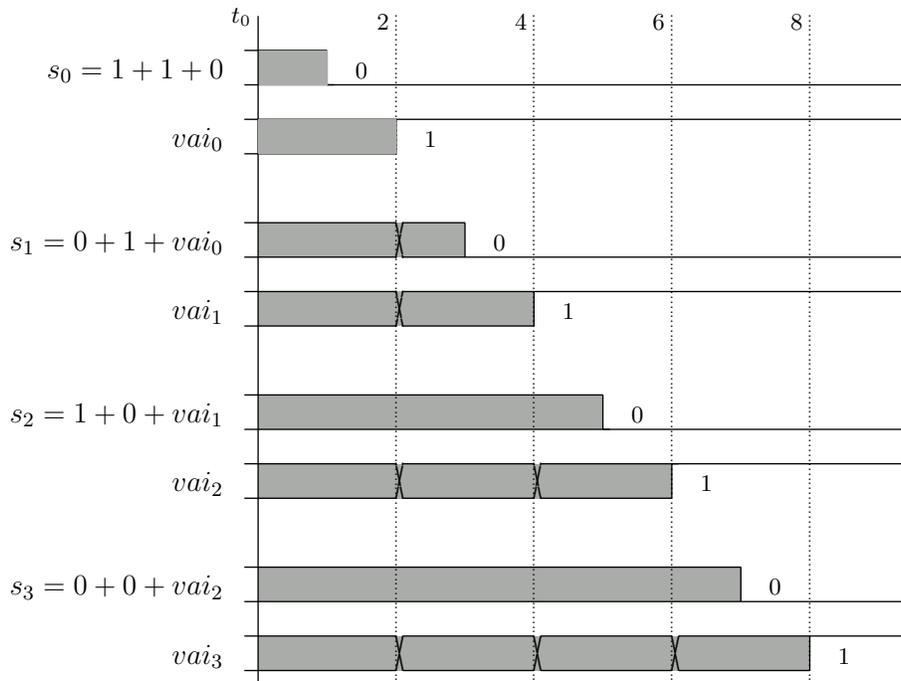


Figura 6.5: Temporização do somador de quatro bits para $0101+0011+0$.

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 $F = 1$.

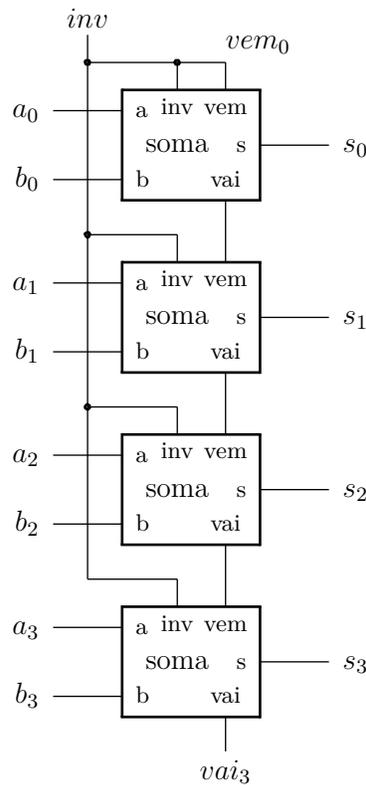


Figura 6.6: Circuito que efetua somas e subtrações.

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

$$\begin{aligned}
 s_i &= (a_i \oplus \bar{a}_i) \oplus vem_i && \text{definição } \oplus \\
 &= 1 \oplus vem_i && \text{definição } \oplus \\
 &= \overline{vem_i} \\
 vai_i &= (a_i \wedge \bar{a}_i) \vee (a_i \wedge vem_i) \vee (\bar{a}_i \wedge vem_i) && \text{complemento } \wedge \\
 &= 0 \vee (a_i \wedge vem_i) \vee (\bar{a}_i \wedge vem_i) && \text{identidade } \vee \\
 &= (a_i \wedge vem_i) \vee (\bar{a}_i \wedge vem_i) && \text{distributiva} \\
 &= (a_i \vee \bar{a}_i) \wedge vem_i && \text{complemento } \vee \\
 &= 1 \wedge vem_i && \text{identidade } \wedge \\
 &= vem_i
 \end{aligned} \tag{6.6}$$

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.

$$\begin{array}{rcl} 0110 & & \text{seis} \\ 0110 \gg 1 & = & 0011 \quad \text{três} \\ 0110 \ll 1 & = & 1100 \quad \text{doze} \end{array}$$

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 $S = B \times 2$.

$$\begin{array}{l} n : \mathbb{N} \\ B : \mathbb{B}^n \\ S : \mathbb{B}^{n+1} \\ d : \mathbb{B} \\ \text{desl} : (\mathbb{B} \times \mathbb{B}^n) \mapsto \mathbb{B}^{n+1} \end{array} \quad (6.8)$$

$$\text{desl}(n, d, B, S) \equiv \begin{cases} s_0 = (0 \triangleleft d \triangleright b_0) \\ s_i = (b_{i-1} \triangleleft d \triangleright b_i) & 1 \leq i < n \\ s_n = (b_{n-1} \triangleleft d \triangleright 0) \end{cases}$$

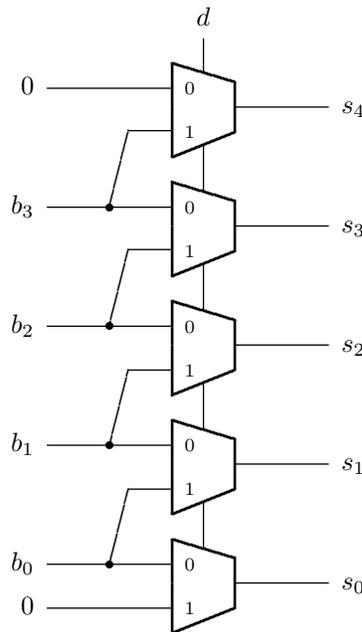


Figura 6.7: Deslocador de uma posição, com quatro bits de entrada.

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: $C = (2^1 \times B) \triangleleft d_2 \triangleright (2^0 \times B)$.

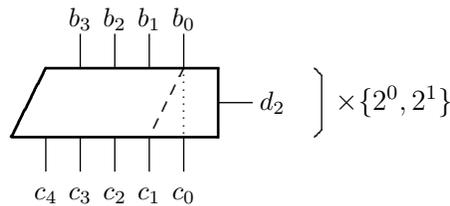


Figura 6.8: Deslocador de 4 bits, uma posição.

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 $C = 2^D \times B$.

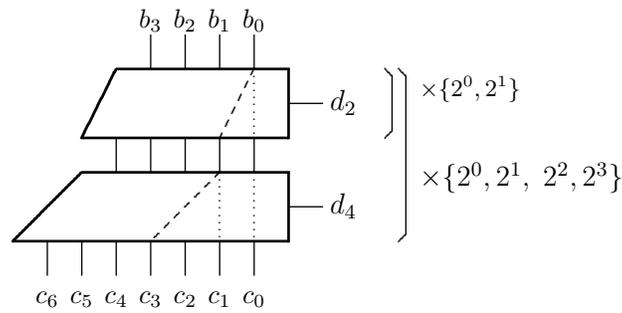


Figura 6.9: Deslocador de 4 bits, três posições.

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 $D = \text{num}(d_{16}d_4d_2)$, então $C = 2^D \times B$.

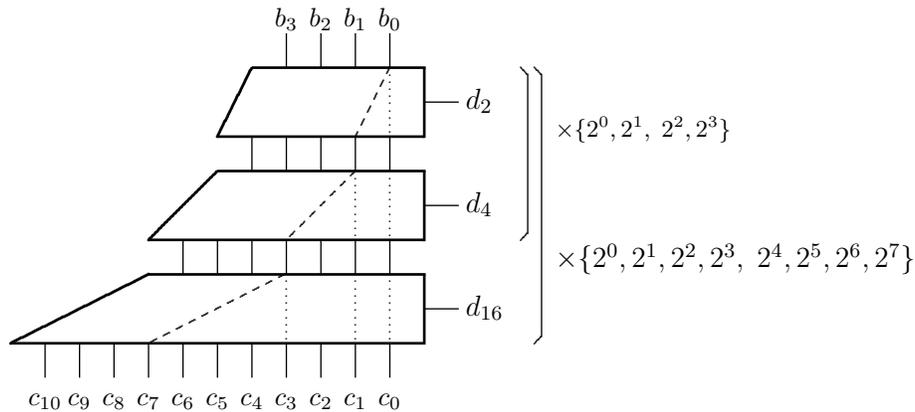


Figura 6.10: Deslocador de 4 bits, sete posições.

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 $[0, 31]$.

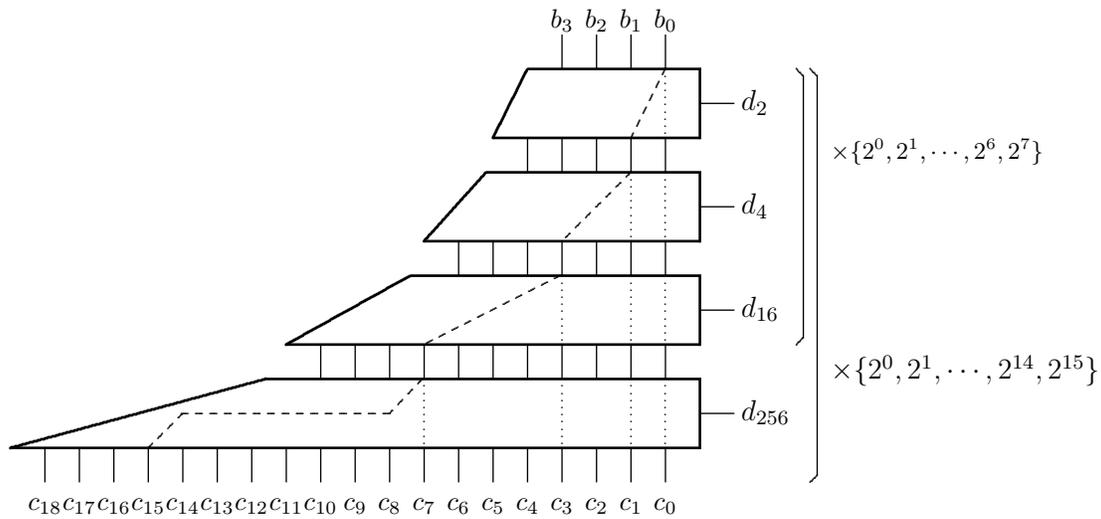


Figura 6.11: Deslocador de 4 bits, 15 posições.

O deslocamento da saída é definido pelas entradas de controle dos deslocadores intermediários: em cada deslocador, ou a entrada não se altera, se $d_n=0$, ou é deslocada de $\log_2 n$ posições, se $d_n=1$. A Figura 6.11 mostra cinco das 16 possibilidades de deslocamento para o bit b_0 , e estas são indicadas na Tabela 6.4.

Tabela 6.4: Deslocamentos no deslocador exponencial.

posições	D	ligação	S
0	0000	$b_0 \mapsto c_0$	$B \times 2^0$
1	0001	$b_0 \mapsto c_1$	$B \times 2^1$
3	0011	$b_0 \mapsto c_3$	$B \times 2^{2+1}$
7	0111	$b_0 \mapsto c_7$	$B \times 2^{4+2+1}$
15	1111	$b_0 \mapsto c_{15}$	$B \times 2^{8+4+2+1}$

Em *deslocamentos lógicos*, os bits que são inseridos nos extremos são sempre 0. Em *deslocamentos aritméticos*, o sinal do número deve ser preservado nos deslocamentos à direita. Esta diferença é explorada no Exercício 6.19.

6.4.2 Rotação

Um deslocador rotacional (*barrel shifter*) é um circuito similar ao deslocador exponencial, mas a saída do deslocador rotacional é sua entrada com uma rotação de d posições. Após uma rotação de uma posição para a esquerda, $s_i = b_{i-1}$ se $i \in [1, n)$, e $s_0 = b_{n-1}$. A Tabela 6.5 e a Equação 6.9 especificam o comportamento de um deslocador rotacional com quatro bits de largura. O operador *mod* computa o resto da divisão inteira.

$$\forall i, d \in \{0, 1, 2, 3\} \bullet s_i = b_{(i-d) \bmod 4} \tag{6.9}$$

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 & 0xffffffc0) * 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.

$$\begin{aligned}
 &A, B, R : \mathbb{B}^4 \\
 &F, T : \mathbb{B}^3 \\
 &ULA-4x8 : [(\mathbb{B}^4 \times \mathbb{B}^4) \times \mathbb{B}^3] \mapsto (\mathbb{B}^4 \times \mathbb{B}^3) \\
 &ULA-4x8(A, B, F, R, T) \equiv (R, T) = \Omega_F(A, B) \\
 &\Omega \in \{+, -, \ll, \gg, \neg, \wedge, \oplus, \vee\}
 \end{aligned} \tag{6.12}$$

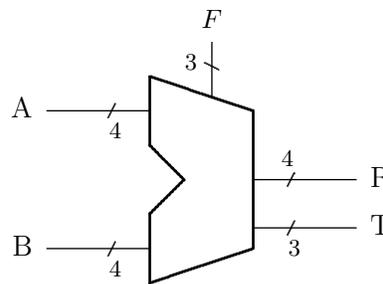


Figura 6.12: Unidade de Lógica e Aritmética.

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 $R = \bar{A}$	$\overline{0011} = 1100$	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 .

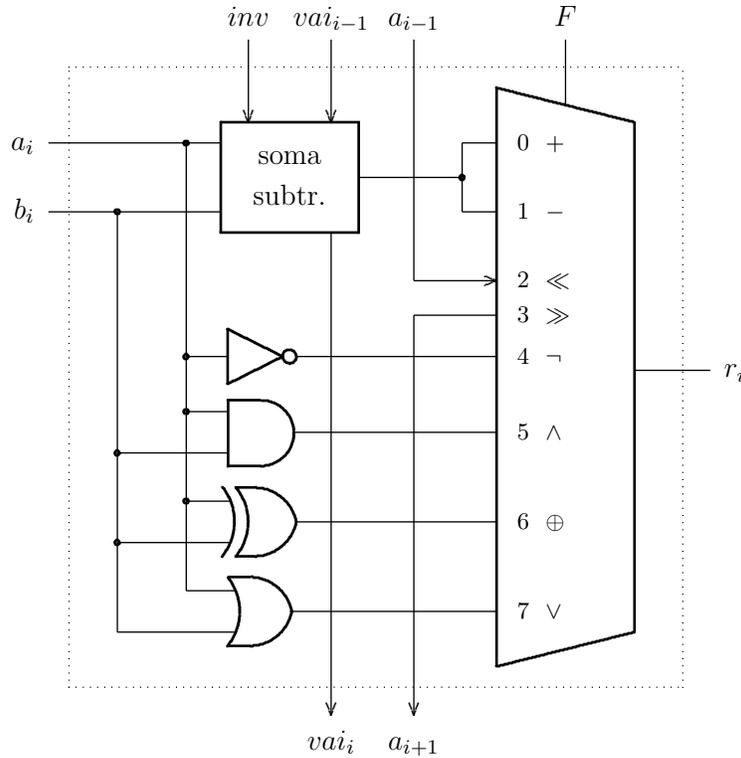


Figura 6.13: Fatia para o i -ésimo bit da ULA.

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.

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.

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 ($\overline{B} \leftarrow inv \triangleright B$) 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*: cuidado com a subtração.

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

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:

$$s_3 \leftarrow s_2 \leftarrow s_1 \leftarrow s_0.$$

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. Para um somador de 16 bits, o sinal vai_{15} estabiliza após 32 unidades de tempo.

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 é possível computar todos os bits de vai-um em paralelo: o vai_3 depende das entradas a_3 e b_3 e de vai_2 , que por sua vez depende de a_2 e b_2 e de vai_1 , e assim sucessivamente.

Uma otimização que reduz substancialmente o tempo de propagação da cadeia de vai-um consiste em computar em paralelo, e então adiantar, o valor de todos bits intermediários da cadeia de vai-um. Esta otimização é chamada de *adiantamento de vai-um*.

A Tabela 6.7 mostra a tabela verdade de um somador completo de dois bits. A sexta coluna (\mathbb{N}) mostra o valor da soma na base 10. As colunas g e p mostram 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, independentemente do vem-um. A função $p(a, b) = 1$ sempre que o valor do vem-um deve ser propagado para o vai-um. O nome em Inglês para o ‘vai-um’ é *carry-out*, e por isso, no que segue, c_i é usado para os bits de vai-um nas cadeias de propagação.

Tabela 6.7: Soma de dois bits.

a	b	vem	vai	s	\mathbb{N}	g	p
0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	0	2	1	1
0	0	1	0	1	1	0	0
0	1	1	1	0	2	0	1
1	0	1	1	0	2	0	1
1	1	1	1	1	3	1	1

O vai-um do bit i pode ser computado com as funções g_i e p_i para um somador de 4 bits:

$$c_i = g_i \vee (p_i \wedge c_{i-1}). \tag{6.13}$$

Temos dois casos:

- (i) se $a_i \wedge b_i = g_i = 1$ então o vai-um $c_i = 1$ é gerado pelos bits da posição i ;
- e
- (ii) se $a_i \vee b_i = p_i = 1$ então c_i é propagado desde a posição $i - 1$.

A Equação 6.14 define a cadeia de geração de vai-um para um somador de quatro bits, na qual a conjunção é denotada por um ponto, e o vem-um da posição menos significativa (c_0) é denotado por vem . Esta equação reflete 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.

$$\begin{aligned}
 c_0 &= g_0 \vee (p_0 \cdot vem) \\
 c_1 &= g_1 \vee (p_1 \cdot g_0) \vee (p_1 \cdot p_0 \cdot vem) \\
 c_2 &= g_2 \vee (p_2 \cdot g_1) \vee (p_2 \cdot p_1 \cdot g_0) \vee (p_2 \cdot p_1 \cdot p_0 \cdot vem) \\
 c_3 &= g_3 \vee (p_3 \cdot g_2) \vee (p_3 \cdot p_2 \cdot g_1) \vee (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\
 &\quad \vee (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot vem)
 \end{aligned} \tag{6.14}$$

Um somador de 4 bits com cadeia de adiantamento de vai-um é mostrado na Figura 6.14. Os sinais de vai-um adiantados são gerados nos trapézios.

O circuito do somador pode ser simplificado porque a geração do vai-um não é efetuada como no circuito da Figura 6.4. Um somador, que computa os sinais de *gera vai-um* e *propaga vai-um* é mostrado na Figura 6.15. 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. O valor da soma fica estável após 2 unidades de tempo, e os sinais p e g estabilizam após 1 unidade.

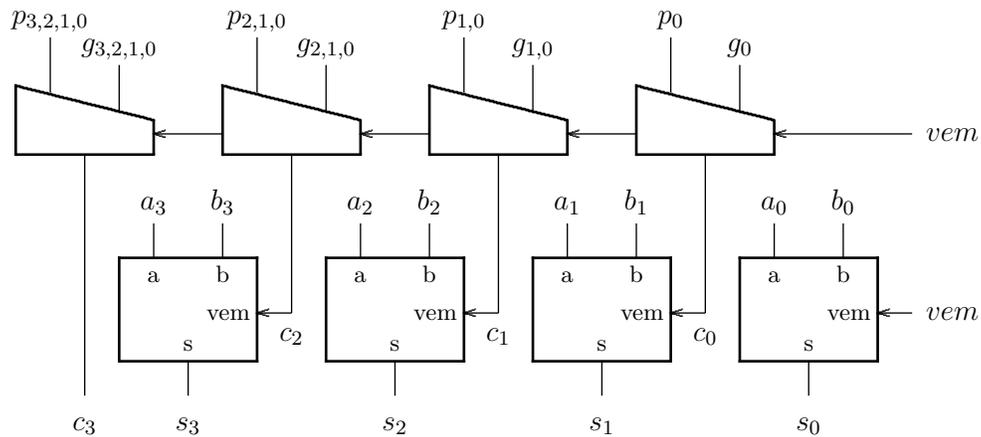


Figura 6.14: Propagação e geração na cadeia de vai-um.

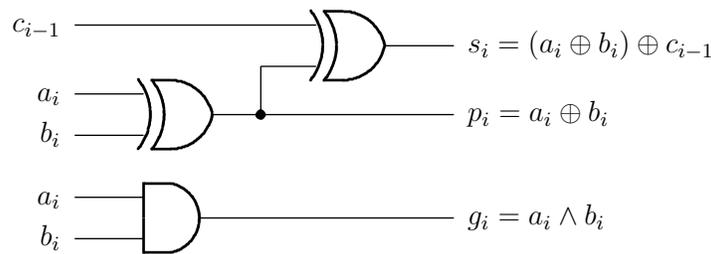


Figura 6.15: Circuito otimizado do somador para adiantamento de vai-um.

Supondo que as entradas A, B, vem estabilizam no instante t , 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.

O circuito do somador de 16 bits com adiantamento de vai-um é mostrado na Figura 6.16. Os somadores são agrupados em blocos de quatro, e o vai-um de cada bloco é computado pelo circuito de adiantamento de vai-um, mostrado como um trapézio na figura. Cada bloco de adiantamento implementa a Equação 6.14 e os sinais $vem_4 = c_3, vem_8 = c_7, vem_{12} = c_{11}$, e $vai = c_{15}$ são gerados em paralelo pelos quatro circuitos de adiantamento. Há uma cadeia de dependência entre os quatro circuitos de adiantamento: $C_3 \leftarrow C_2 \leftarrow C_1 \leftarrow C_0$.

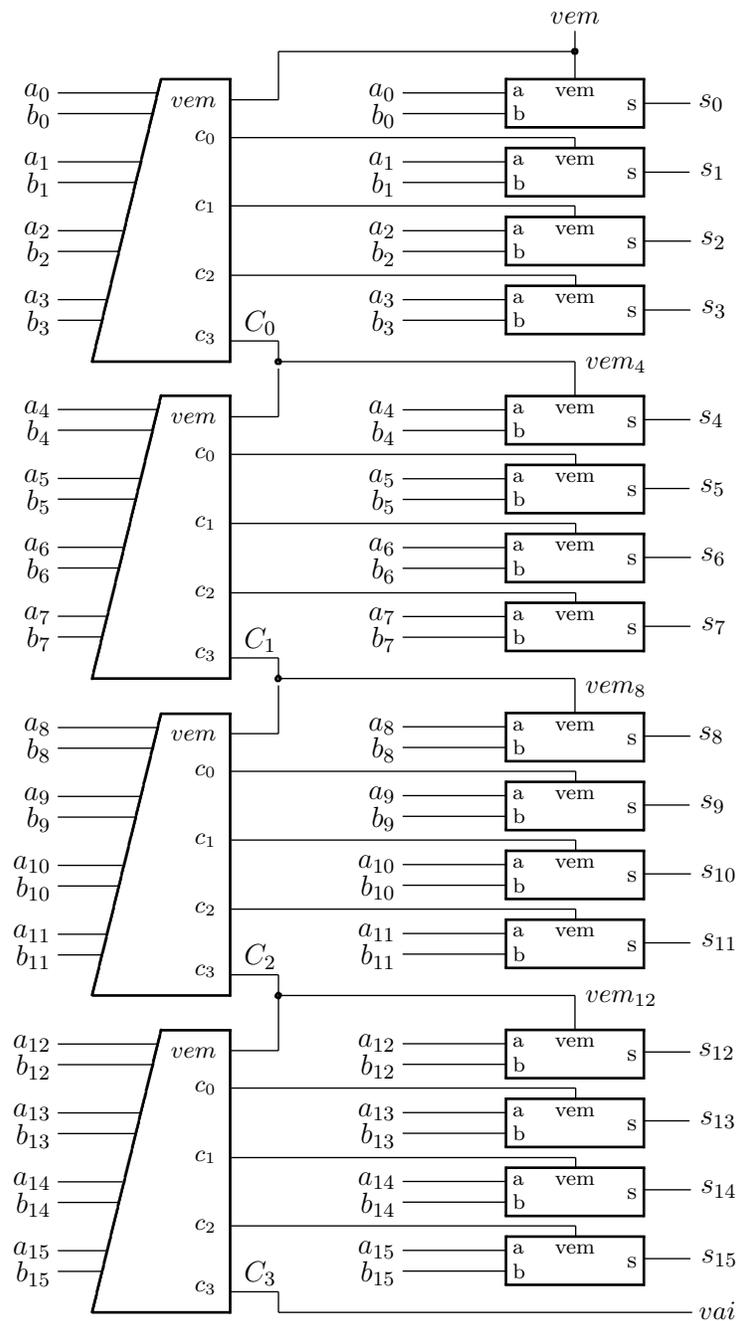


Figura 6.16: Somador de 16 bits com adiantamento de 4 bits.

A mesma ideia do adiantamento de 4 bits pode ser empregada para adiantar o vai-um em somadores de 16 bits, considerando-se agora grupos de 4 bits. As Equações 6.15 e 6.16 definem a propagação do vem-um através de um quarteto, e a geração do vai-um em qualquer posição do quarteto, respectivamente.

$$\begin{aligned}
P_0 &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \\
P_1 &= p_7 \cdot p_6 \cdot p_5 \cdot p_4 \\
P_2 &= p_{11} \cdot p_{10} \cdot p_9 \cdot p_8 \\
P_3 &= p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}
\end{aligned} \tag{6.15}$$

$$\begin{aligned}
G_0 &= g_3 \vee p_3 \cdot g_2 \vee p_3 \cdot p_2 \cdot g_1 \vee p_3 \cdot p_2 \cdot p_1 \cdot g_0 \\
G_1 &= g_7 \vee p_7 \cdot g_6 \vee p_7 \cdot p_6 \cdot g_5 \vee p_7 \cdot p_6 \cdot p_5 \cdot g_4 \\
G_2 &= g_{11} \vee p_{11} \cdot g_{10} \vee p_{11} \cdot p_{10} \cdot g_9 \vee p_{11} \cdot p_{10} \cdot p_9 \cdot g_8 \\
G_3 &= g_{15} \vee p_{15} \cdot g_{14} \vee p_{15} \cdot p_{14} \cdot g_{13} \vee p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}
\end{aligned} \tag{6.16}$$

A Equação 6.17 define a geração dos sinais de vai-um do quarteto. O sinal P_0 indica que ocorre a propagação de vai-um através dos quatro somadores dos bits menos significativos, enquanto o sinal G_0 indica que aqueles somadores produzem vai-um. O sinal C_3 é o vai-um adiantado do somador na posição mais significativa: $C_3 = c_{15} = vai$.

$$\begin{aligned}
C_0 &= G_0 \vee P_0 \cdot vem \\
C_1 &= G_1 \vee P_1 \cdot G_0 \vee P_1 \cdot P_0 \cdot vem \\
C_2 &= G_2 \vee P_2 \cdot G_1 \vee P_2 \cdot P_1 \cdot G_0 \vee P_2 \cdot P_1 \cdot P_0 \cdot vem \\
C_3 &= G_3 \vee P_3 \cdot G_2 \vee P_3 \cdot P_2 \cdot G_1 \vee P_3 \cdot P_2 \cdot P_1 \cdot G_0 \\
&\quad \vee P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot vem
\end{aligned} \tag{6.17}$$

Para tirar proveito desta otimização, no circuito da Figura 6.16, os sinais C_0, C_1, C_2 devem ser ligados, respectivamente a vem_4, vem_8, vem_{12} , e o sinal C_3 passa a ser o vai-um do somador de 16 bits, sendo portanto ligado a vai . Um diagrama com a otimização dos quartetos incluiria os circuitos que computam os P_i, G_i e C_i nos trapézios.

6.7 Somador com Seleção de Vai-um

Uma alternativa para a implementação de somadores rápidos é indicada na Figura 6.17, que mostra 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:

$$\langle V_3 S_{3..0} \rangle = A_{3..0} + B_{3..0} + v_0.$$

A parte mais significativa da soma é obtida em paralelo com a parte menos significativa, computando-se simultaneamente duas alternativas para o vai-um da parte menos significativa, uma com $v_4 = 0$, e a outra com $v_4 = 1$. Quando os sinais se propagam através do somador da parte menos significativa, e o valor de $V_3 = v_3$ fica estável, este é usado para escolher um dos dois resultados parciais, através do multiplexador de 4 bits de largura.

O vai-um da soma em 8 bits depende do valor de V_3 , e é dado por

$$V_8 = (V_{7z} \vee V_3) \wedge V_{7u}.$$

Este circuito é chamado de “somador com seleção de vai-um” (*carry select adder*) porque a metade mais significativa do resultado é *selecionada* pelo vai-um da metade menos significativa.

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

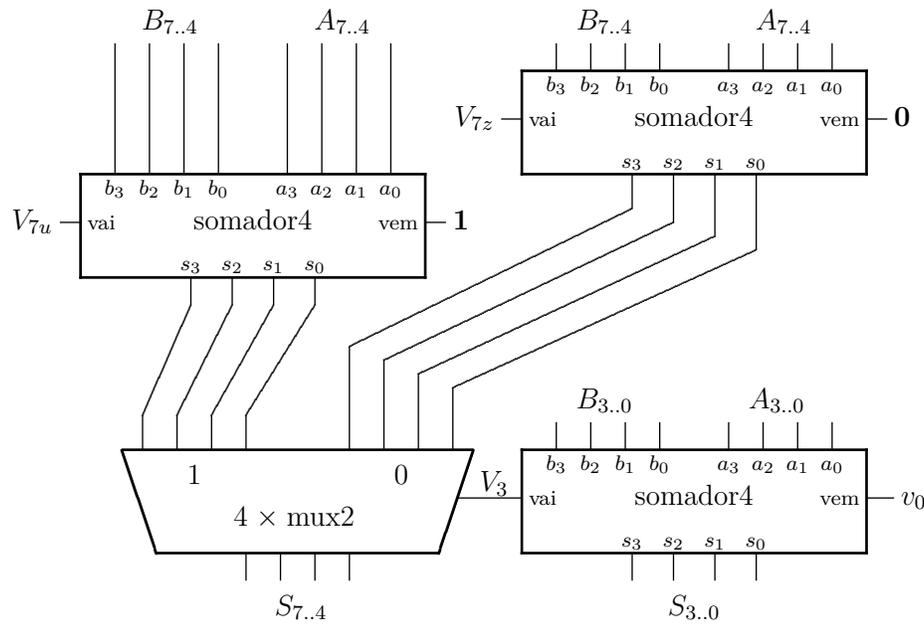


Figura 6.17: Somador com seleção de vai-um.

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 deste 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 deste novo projeto seria mais curto que aquele do circuito da Figura 6.17.

Exercícios

Ex. 6.20 Estime o tempo de propagação do somador de 16 bits com o adiantamento de vai-um definido na Equação 6.17, 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 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. 6.22 Repita o Ex. 6.21, 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.

$$\begin{array}{r} 1\ 0\ 1\ 1 \\ \times 1\ 0\ 0\ 1 \\ \hline \end{array}$$

A multiplicação, sem simplificações, é mostrada na Figura 6.18. 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.

$$\begin{array}{r} 11 \\ \times 9 \\ \hline 99 \end{array} \qquad \begin{array}{r} 1\ 0\ 1\ 1 \\ \times 1\ 0\ 0\ 1 \\ \hline 1\ 0\ 1\ 1 \quad \times 1 \\ \textit{1}^1\ 0\ 1\ 1 \quad \times 0 \\ \textit{1}^1\ 0\ 1\ 1 \quad \times 0 \\ +\ 1\ 0\ 1\ 1 \quad \times 1 \\ \hline 1\ 1\ 0\ 0\ 0\ 1\ 1 \end{array}$$

Figura 6.18: Exemplo de multiplicação: 11×9 .

O resultado da multiplicação da Figura 6.18 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 6.18.

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 6.18, 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 6.18. 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).

$$\begin{aligned}
 s &: \mathbb{B} \\
 A, B &: \mathbb{B}^4 \\
 R &: \mathbb{B}^5 \\
 mp1 &: \mathbb{B} \times (\mathbb{B}^4 \times \mathbb{B}^4) \mapsto \mathbb{B}^5 \\
 mp1(s, A, B, R) &\equiv num(R) = [num(A) + num(B)] \triangleleft s \triangleright [0 \& A]
 \end{aligned}
 \tag{6.18}$$

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 6.18. Da quarta parcela resultam ainda os quatro bits mais significativos do produto, (z_7, z_6, z_5, z_4) . A Figura 6.19 mostra o multiplicador de 4×4 bits, com os multiplicando X e multiplicador Y no topo, e o produto Z na base.

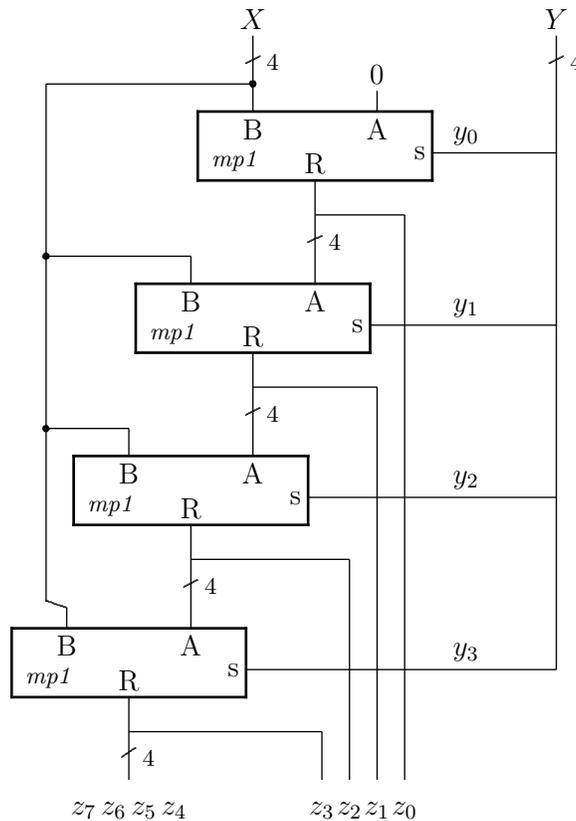


Figura 6.19: Primeira implementação da multiplicação em 4×4 bits.

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 contêm 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 6.18 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 6.18 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 .

$$\begin{array}{c|cc} \times & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array} \tag{6.19}$$

A Figura 6.20 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 6.20: Multiplicação de operandos de 4 bits.

Uma implementação direta da multiplicação é mostrada na Figura 6.21. Os blocos ‘s’ 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 .

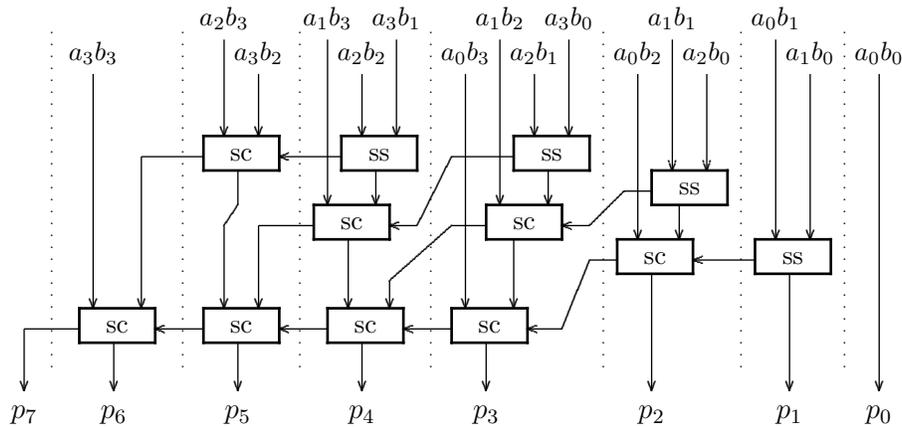


Figura 6.21: Implementação irregular da multiplicação em 4×4 bits.

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 $3 \times 2 = 6$.

$$\begin{array}{r}
 3 \\
 \times 2 \\
 \hline
 6
 \end{array}
 \qquad
 \begin{array}{r}
 1\ 1 \\
 \times 1\ 0 \\
 \hline
 1\ 1 \\
 + 1\ 1 \\
 \hline
 1\ 1\ 0
 \end{array}
 \begin{array}{l}
 \times 0 \\
 \times 1
 \end{array}$$

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:

$$\begin{array}{r}
 A_L\ 1 \\
 B_L\ \times 0 \\
 \hline
 00
 \end{array}
 \qquad
 \begin{array}{r}
 A_H\ 1 \\
 B_L\ \times 0 \\
 \hline
 00
 \end{array}
 \qquad
 \begin{array}{r}
 A_L\ 1 \\
 B_H\ \times 1 \\
 \hline
 01
 \end{array}
 \qquad
 \begin{array}{r}
 A_H\ 1 \\
 B_H\ \times 1 \\
 \hline
 01
 \end{array}$$

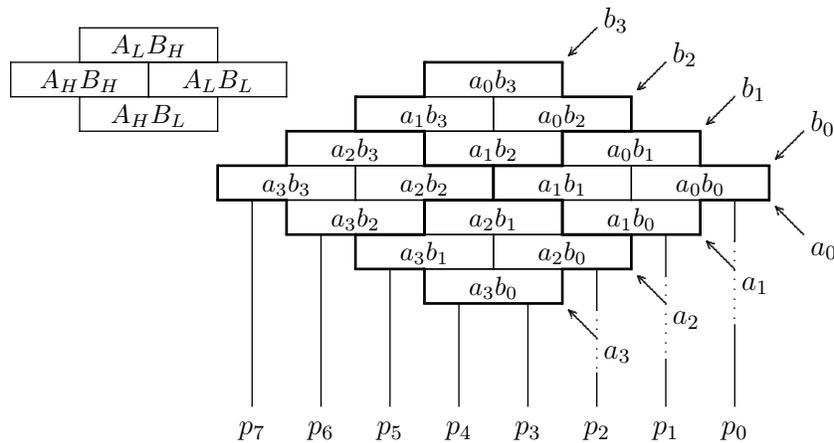


Figura 6.23: Implementação eficiente da multiplicação em 4×4 bits.

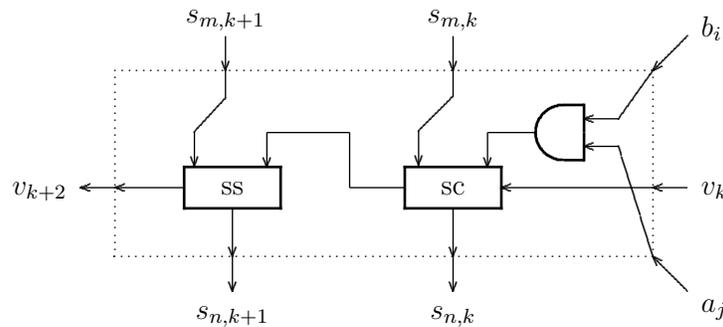


Figura 6.24: Bloco que multiplica e acumula os produtos parciais.

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.

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 6.25 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?

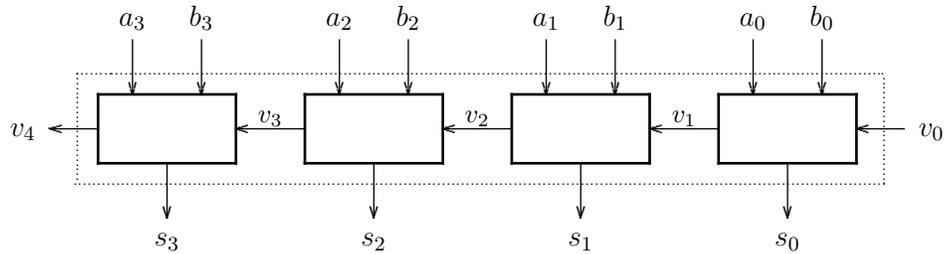


Figura 6.25: Circuito de testes do somador.

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 $x"1111"$?

$$\begin{array}{r}
 1111 \\
 +1111 \\
 \hline
 2222
 \end{array}
 \qquad
 \begin{array}{r}
 0001\ 0001\ 0001\ 0001 \\
 +0001\ 0001\ 0001\ 0001 \\
 \hline
 0010\ 0010\ 0010\ 0010
 \end{array}$$

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 $x"5555"$?

$$\begin{array}{r}
 5555 \\
 +5555 \\
 \hline
 AAAA
 \end{array}
 \qquad
 \begin{array}{r}
 0101\ 0101\ 0101\ 0101 \\
 +0101\ 0101\ 0101\ 0101 \\
 \hline
 1010\ 1010\ 1010\ 1010
 \end{array}$$

O que se descobre somando $x"AAAA"$ com $x"AAAA"$?

$$\begin{array}{r}
 AAAA \\
 +AAAA \\
 \hline
 1\ 5554
 \end{array}
 \qquad
 \begin{array}{r}
 1010\ 1010\ 1010\ 1010 \\
 +1010\ 1010\ 1010\ 1010 \\
 \hline
 1\ 0101\ 0101\ 0101\ 0100
 \end{array}$$

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 $x"0001"+x"FFFF"$?

Estes 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.

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 introduzidos 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.

Exercícios

Ex. 6.23 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. 6.24 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. 6.25 Escreva um conjunto de tuplas $\langle A, B, F, R, T \rangle$ para 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. 6.26 Complete o circuito da Figura 6.23, acrescentando todas as ligações que faltam na periferia. *Pista:* veja o bloco da Figura 6.24.

Ex. 6.27 Efetua as otimizações possíveis no circuito da sua resposta ao Ex. 6.26.

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.

Índice Remissivo

Símbolos

T_A , 61
 T_D , 115
 T_I , 104, 111
 T_M , 112, 114
 T_P , 111
 T_p , 162
 $T_{C,x}$, 113–115
%, 18–20
 \bigvee , 45
 \bigwedge , 45
 \equiv , 36
 \gg , 149
 \wedge , 30, 36
 $\triangleleft \triangleright$, 36, 42, 63
 \Leftrightarrow , 36
 \Rightarrow , 36, 37, 40, 44
 \ll , 149
 \neg , 30, 36, 147
 \vee , 30, 36
 \oplus , 36, 50, 62
 \mapsto , 41
 \mathbb{N} , 140
 \bar{a} , veja \neg
 π , 25
 \setminus , 74
decod, 69
demux, 72
e, 24
mod, 152
mux, 64
 \mathbb{B} , 29–33, 38, 41
 \mathbb{Z} , 41, 144
 \mathbb{N} , 41, 42, 144
num, 43, 52, 69, 74
 num^{-1} , 52
 \mathbb{R} , 41
 $\langle \rangle$, 29, 41, 42, 170

A

abstração, 27
 bits como sinais, 27–33, 54
 tempo discretizado, 111, 113
acumulação de produtos parciais, 163–168
adição, 146
adiantamento de vai-um, 157–161
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
barrel shifter, 152
básculo, 132
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
buffer, 118
buffer three-state, 134
buraco, 83
byte, 11

C

C,
 deslocamento, 154
C, linguagem, 154
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
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, 157
 de magnitude, 157

Complementary Metal-Oxide Semiconductor, *veja CMOS*

complemento, *veja* \neg

complemento de dois, 139–145, 147–157

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

deslocamento, 149–152
 aritmético, 149, 152, 157
 exponencial, 153
 lógico, 149, 156

rotação, 152

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, 159

fan-out, 79, 81, 104–107, 111, 115, 162

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* búsculo
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
 multiplicação, 163–168
 acumulação de produtos parciais, 163–168
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
overflow, 142–144, 148, 157, 163, 169

P

paridade,
 ímpar, 47
 par, 47
 período mínimo, *veja* relógio
pipelining, *veja* segmentação
 piso, *veja* [v]
 ponto fixo, 145

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
 complemento de dois, 141
 concreta, 27
 decimal, 17
 hexadecimal, 19
 octal, 18
 ponto fixo, 145
 posicional, 17
 precisão, 23

resistência, 84, 95, 100
 ROM, 12, 74–77, 121–128
 rotação, 152, 157
 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, 139, 146–147
 adiantamento de vai-um, 157
 cadeia, 146
 completo, 99, 146
 parcial, 98
 seleção de vai-um, 161
 teste, 168
 somatório, 33
 spice, 28
 SRAM, 132–134
 SSD, 14
 status, 156
 subtração, 147–148
 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
 teste,
 cobertura, 170
 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
transmission gate, veja *porta de transmissão*
TTL,
 74148, 71
tupla, veja $\langle \rangle$
 elemento, 32
 largura, 43

U

ULA, 154–157, 170
 status, 156
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