6 CARRY-LOOKAHEAD ADDERS AND VARIATIONS IN FAST ADDERS

Chapter Goals

1.- Understand the carry-lookahead method and its many variations used in the design of fast adders.
2.- Study alternatives to the carry-lookahead method for designing fast adders

Chapter Highlights

Single- and multilevel carry lookahead.
Various designs for log-time adders.
Relating the carry determination problem to parallel prefix computation.
Implementing fast adders in VLSI

Neste capitulo vão ser explicados os somadores com "carry Lookahead", bem como variações de somadores que possuem o intuito de atingir maiores velocidades de computação.

CARRY-LOOKAHEAD ADDERS AND VARIATIONS IN FAST ADDERS: TOPICS

Topics in This Chapter
6.1 Unrolling the Carry Recurrence
6.2 Carry-Lookahead Adder Design
6.4 Carry Determination as Prefix Computation
6.5 Alternative Parallel Prefix Networks
7.1 Carry-Skyp Adders
7.3 Carry-Select Adders
7.4 Analysis of Carry Propagation
7.6 Modular Two-Operand Adders

Vejamos os tópicos que serão apresentados.

6.1 UNROLLING THE CARRY RECURRENCE

Recall the *generate*, *propagate*, *annihilate*, and *transfer* signals:

Signal	Binary
g_i	$x_i\, y_i$
p_i	$x_i \oplus y_i$
a_i	$x_i' y_i' = (x_i \vee y_i)'$
t_i	$x_i \vee y_i$
s_i	$x_i \oplus y_i \oplus c_i = p_i \oplus c_i$

The carry recurrence can be unrolled to obtain each carry signal directly from inputs, rather than through propagation

$$
\begin{aligned}
c_i &= g_{i-1} \vee c_{i-1} p_{i-1} \\
&= g_{i-1} \vee (g_{i-2} \vee c_{i-2} p_{i-2}) p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee c_{i-2} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee c_{i-3} p_{i-3} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee g_{i-4} p_{i-3} p_{i-2} p_{i-1} \vee c_{i-4} p_{i-4} p_{i-3} p_{i-2} p_{i-1} \\
&= \ldots
\end{aligned}
$$

SLIDE 3

Computer Arithmetic, Addition/Subtraction

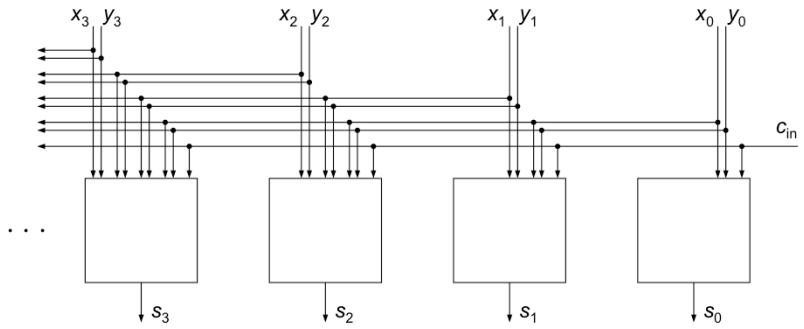
O somador ripple carry, embora simples em seu conceito, tem um longo atraso de computação devido às muitas portas no caminho crítico, que sai do bit menos significativo indo até o bit mais significativo.

O somador carry lookahead pode ser uma solução para uma computação mais rápida, quando comparado ao ripple carry.

Observemos os dois sinais pi e gi. A função pi = xi ⊕ yi é chamada de função de propagação. Sempre que pi for igual a 1, um carry de entrada é propagado através da posição do bit de Ci para Ci+1. Para pi igual a 0, a propagação do transporte através da posição do bit é bloqueada. A função gi = xi · yi e é chamada de função de geração. Sempre que gi é igual a 1, a saída de transporte da posição é 1, independentemente do valor de pi, portanto, um transporte foi gerado na posição. Quando gi é 0, um carry não é gerado, de modo que Ci+1 é 0 se o carry propagado pela posição de Ci também for 0. As funções de geração e propagação correspondem exatamente a um meio somador (half adder).

O carry lookahead é obtido desenvolvendo a relação de recorrência para ci, como demonstrado no slide.

6.1 UNROLLING THE CARRY RECURRENCE: FULL CARRY LOOKAHEAD



Theoretically, it is possible to derive each sum digit directly from the inputs that affect it

Carry-lookahead adder design is simply a way of reducing the complexity of this ideal, but impractical, arrangement by hardware sharing among the various lookahead circuits

Computer Arithmetic, Addition/Subtraction

SLIDE 4

Teoricamente, o que vamos buscando é fazer um somador sem propagação de (carry).

Como vemos a complexidade aumenta para entradas mais a esquerda (veja o número de setas que entram nos blocos).

Um Fan-in muito maior do que 4 é impraticável para portas CMOS únicas, devido à baixa imunidade a ruídos, problemas nos tempos de subida e descida e, portanto, maior atraso. Para a maioria das implementações, realizamos blocos carry lookahead de, no máximo, tamanho 4.

6.1 UNROLLING THE CARRY RECURRENCE: FOUR-BIT CARRY-LOOKAHEAD ADDER

Complexity reduced by deriving the carry-out indirectly

Full carry lookahead is quite practical for a 4-bit adder

$$c_1 = g_0 \vee c_0 p_0$$
$$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$$
$$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$$
$$c_4 = g_3 \vee g_2 p_3 \vee g_1 p_2 p_3 \vee g_0 p_1 p_2 p_3 \vee c_0 p_0 p_1 p_2 p_3$$

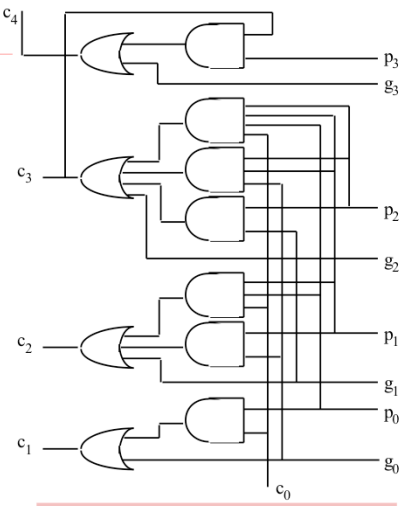
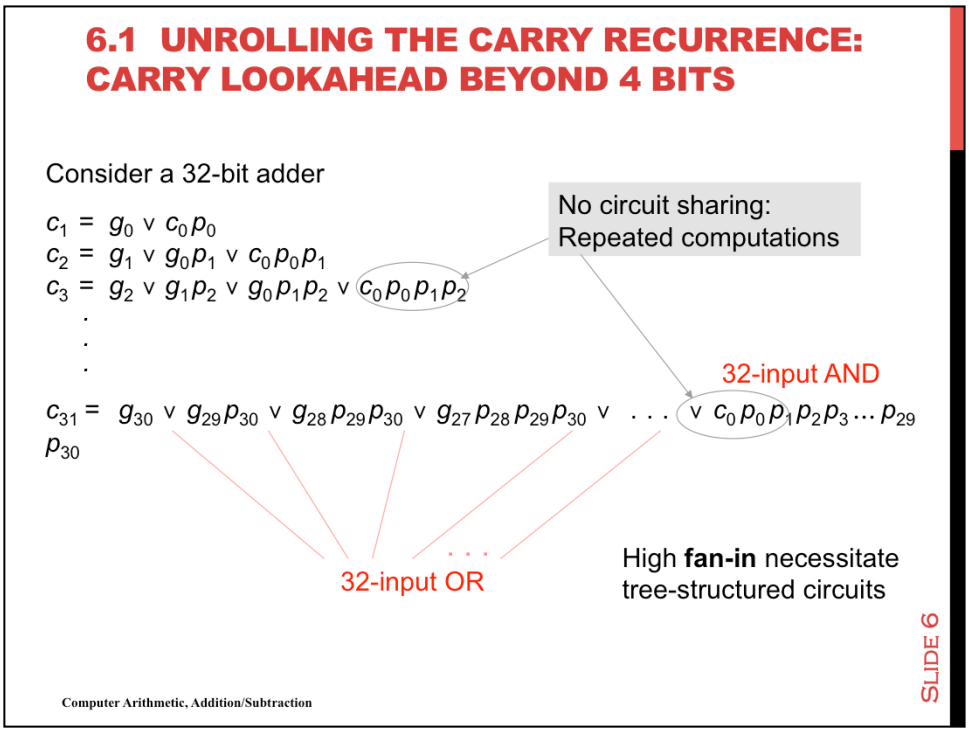


Fig. 6.1 Four-bit carry network with full lookahead.

Vejamos a implementação acima. Aqui estão descritos os sinais c1, c2, c3 e c4 para um CLA de 4 bits.

Para reduzir a complexidade de c4 usamos c3 directamente na implementação. Isso porque o sinal c3, que pode ser observado na equação, se repete em parte do sinal da equação c4.



Então, como vimos anteriormente para uma implementação real e para reduzir o fan-in e custos associados a area, podemos re-utilizar computações já implementadas como é mostrado aqui.

6.1 UNROLLING THE CARRY RECURRENCE: ONE SOLUTION TO THE FAN-IN PROBLEM

Multilevel lookahead

Example: 16-bit addition

Radix-16 (four digits)

Two-level carry lookahead (four 4-bit blocks)

Either way, the carries c_4, c_8, and c_{12} are determined first

c_{16}	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
c_{out}				?				?				?				c_{in}

SLIDE 7

Computer Arithmetic, Addition/Subtraction

Nos exemplos anteriores vimos que podemos reutilizar porções de hardware que precedem um determinado sinal. No exemplo de circuito de 4 bits, vimos que c3 é utilizado para formular c4.

Considere então a adição de 16 bits de dois números binários que são caracterizadas por seus sinais gi e pi. Para cada dígito, estendende-se da posição de bit i para a posição de bit i + 3 dos números binários originais (onde i é um múltiplo de 4), os sinais de “generate" e “propagate" podem ser derivados da seguinte forma:

g[i,i+3] = (gi+3) ∨ (gi+2)(pi+3) ∨ (gi+1)(pi+2)(pi+3) ∨ (gi)(pi+1)(pi+2)(pi+3)

p[i,i+3] = (pi)(pi+1)(pi+2)(pi+3)

As equações que precedem i+3 podem ser interpretadas da mesma maneira que as equações de carry desenvolvidas no slide 3: as posições de quatro bits propagam coletivamente um carry ci de entrada se e somente se cada uma das quatro posições se propagar; eles geram coletivamente um transporte se um transporte é produzido na posição i + 3, ou é produzido na posição i + 2 e propagado pela posição i + 3. Veja então que podemos realizar o calculo dos carrys em blocos de 4 bits.

6.2 CARRY-LOOKAHEAD ADDER DESIGN

Block *generate* and *propagate* signals

$$g_{[i,i+3]} = g_{i+3} \vee g_{i+2}p_{i+3} \vee g_{i+1}p_{i+2}p_{i+3} \vee g_i p_{i+1}p_{i+2}p_{i+3}$$
$$p_{[i,i+3]} = p_i p_{i+1}p_{i+2}p_{i+3}$$

$$C_{i+4} = g_{[i,i+3]} \vee c_i p_{[i,i+3]}$$

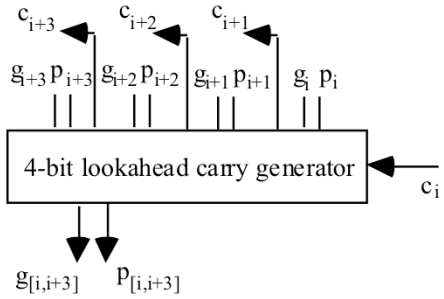
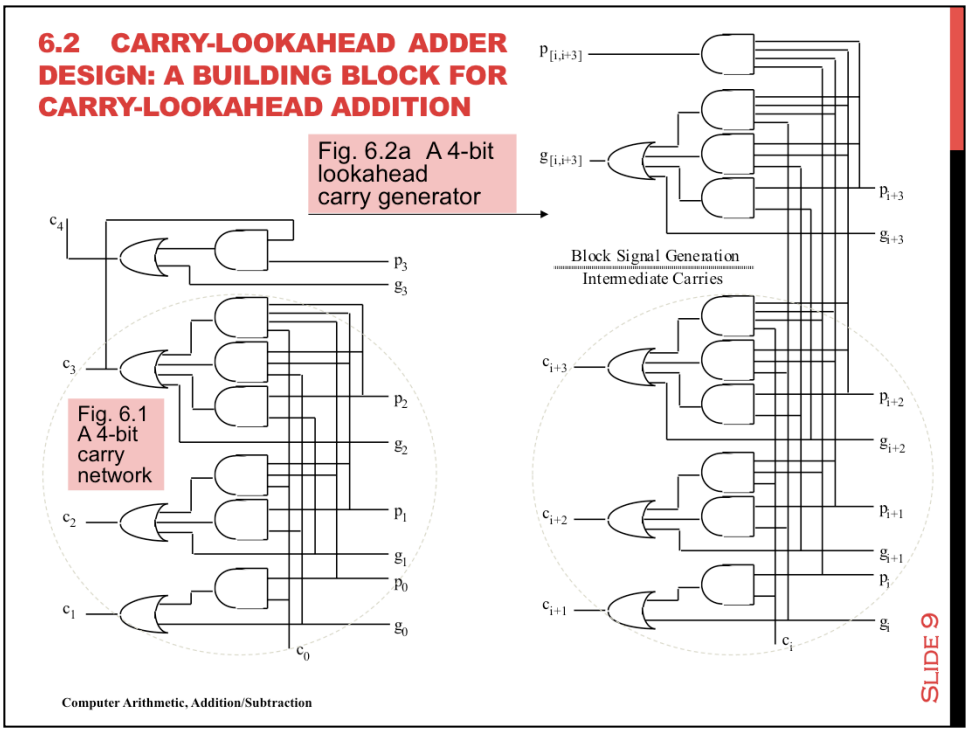
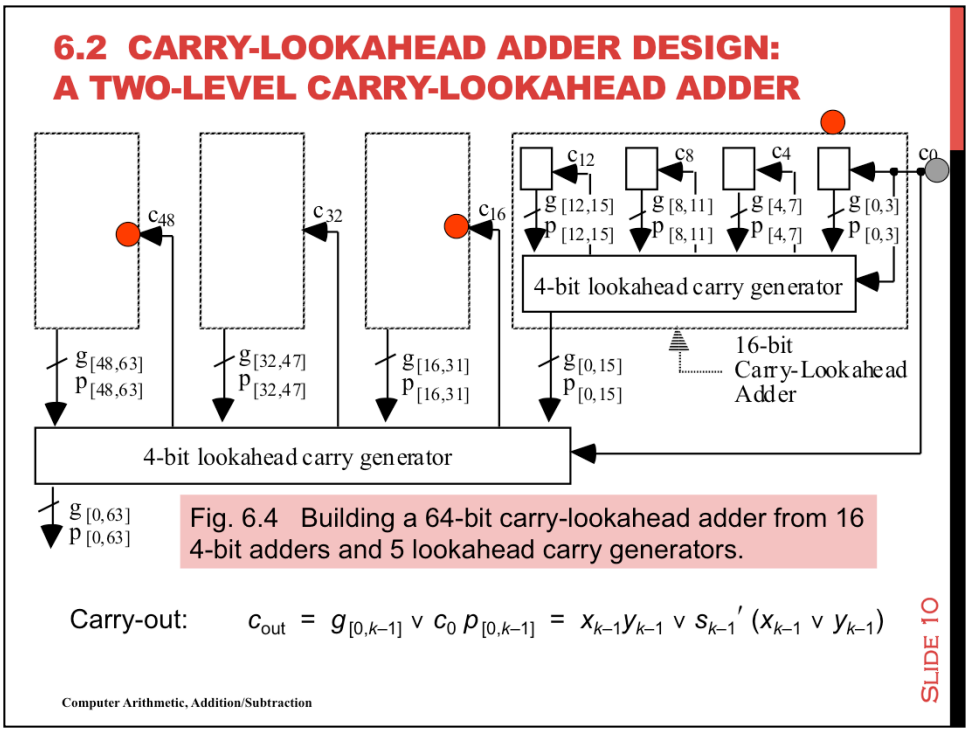


Fig. 6.2b Schematic diagram of a 4-bit lookahead carry generator.

Então, podemos projetar nosso gerador de carry lookahead como um bloco de 4 bits, combinando os sinais g e p de blocos adjacentes ou sobrepostos.



Dado o gerador de carry lookahead de 4 bits se torna fácil sintetizar somadores mais amplos com base em um esquema de carry lookahead multinível



Por exemplo, para construir um somador de carry lookahead de dois níveis de 16 bits, precisamos de quatro somadores de 4 bits e um gerador de carry lookahead de 4 bits, conectado entre si.

Já o slide acima, dá um exemplo de somador de 64 bits, que pode ser realizado com 16 somadores de 4 bits e 5 geradores de carry lookahead também de 4 bits.

6.4 CARRY DETERMINATION AS PREFIX COMPUTATION

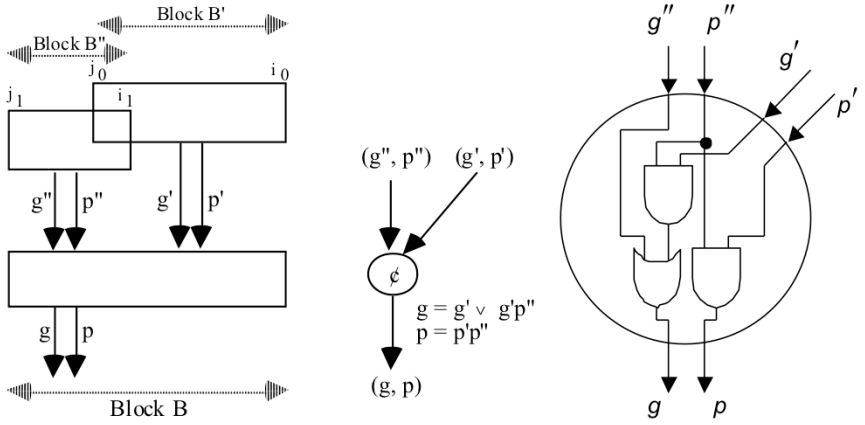


Fig. 6.5 Combining of *g* and *p* signals of two (contiguous or overlapping) blocks B' and B" of arbitrary widths into the *g* and *p* signals for block B.

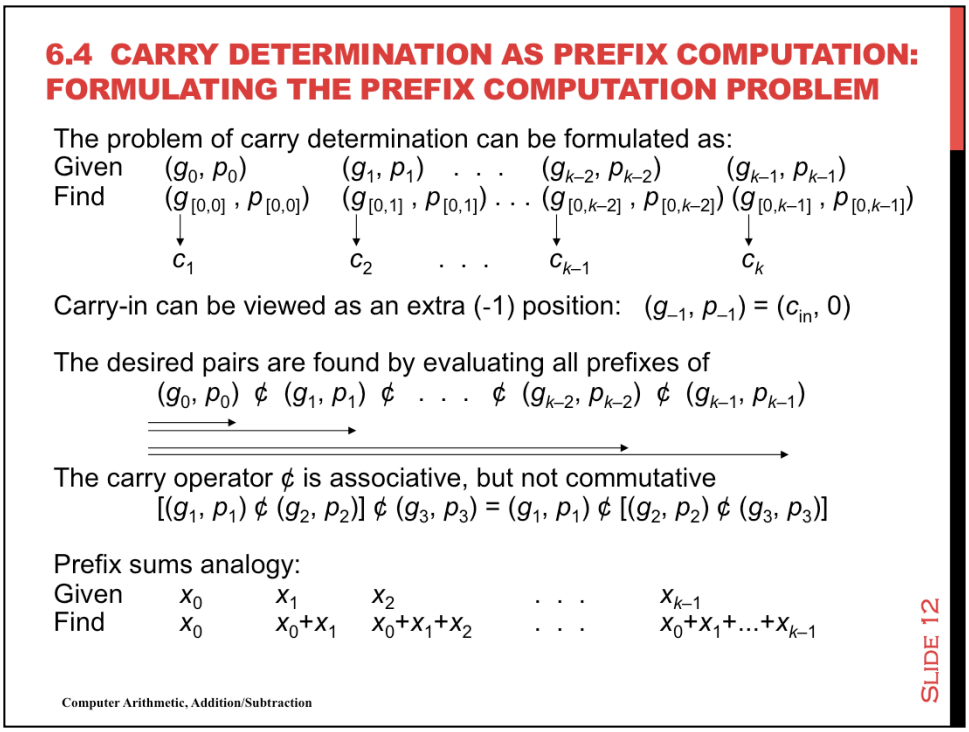
Considere dois blocos que se sobrepõe. Os chamaremos de B' e B" e seus pares de sinais de geração e propagação associados (g', p') e (g", p"), respectivamente. Conforme mostrado na figura, os sinais de geração e propagação para o bloco B, mesclado, podem ser obtidos a partir das equações:

g= g" ∨ g' p"

p = p'.p"

Ou seja, a geração de carry no bloco maior ocorre se o grupo da esquerda gera um carry (g") ou o grupo da direita gera um carry e o da esquerda o propaga (g' p"), enquanto a propagação ocorre se ambos os grupos propagam o carry (p'.p")

Notamos que na figura que ilustra o bloco B , os índices i0, j0, i1 e j1 que definem os dois blocos sobrepostos são de fato imateriais e as mesmas expressões podem ser escritas para quaisquer dois grupos adjacentes de qualquer largura.



Vamos definir então um novo operador com o intuito de determinar o sinal de "carry“ de forma mais rápida. O operador

¢ (g, p) facilitará as computações que seguirão.

(g, p) = (g’ , p’) ¢ (g”, p”) significa g = g” ∨ g’ p” , p = p’ p”

O operador **¢** é associativo, o que significa que a ordem de avaliação não afeta o valor da expressão. Etretanto não é comutativo, pois **g” ∨ g’ p”** em geral não é igual a **g’ ∨ g” p’.**

De forma mais simplificada, nosso objetivo será calcular o sinal “g”. O “g” será equivalente aos carries que serão aplicados na soma final.

6.5 ALTERNATIVE PARALLEL PREFIX NETWORKS: LADNER-FISHER (LF)

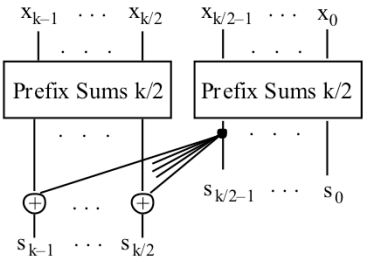
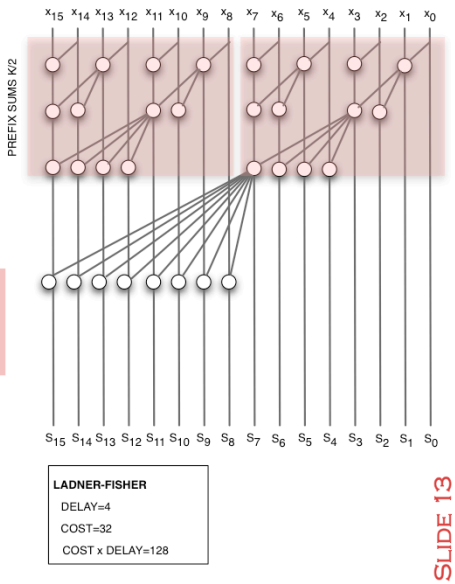


Fig. 6.7 Ladner-Fischer parallel prefix sums network built of two *k*/2-input networks and *k*/2 adders.



Delay recurrence $D(k) = D(k/2) + 1 = \log_2 k$
Cost recurrence $C(k) = 2C(k/2) + k/2 = (k/2) \log_2 k$

LADNER-FISHER
DELAY=4
COST=32
COST x DELAY=128

Computer Arithmetic, Addition/Subtraction

SLIDE 13

Chamaremos essas novas previsões que determinarão os sinais de carry de redes de prefixos. Cada circulo no gráfico de prefixos consiste na aplicação do operador ¢. Então o circulo que está ligado aos primeiros sinais X1 e X0 serão equivalentes à **g = g" ∨ g' p" , p = p' p"** gerando tanto um sinal g quanto um sinal p na saida.

Ao fim do cálculo e dada entradas iniciais do circuito iguais a ‘x” e “y”, podemos utilizar o sinal “g” para fazer a soma “x” xor “y” xor “g”.

Podemos usar várias estratégias para sintetizar uma rede de soma de prefixo paralela. A Figura 6.7 é baseada em uma abordagem de dividir e conquistar, conforme proposto por Ladner e Fischer

As entradas k / 2 de ordem inferior são processadas pela sub-rede à direita para calcular as somas de prefixo s0, s1, ..., sk / 2−1.

As somas de prefixo parciais são calculadas para os valores k / 2 de ordem mais alta (a sub-rede esquerda). Por sim, sk / 2−1 (a saída mais à esquerda da primeira sub-rede) é adicionado a eles para completar o cálculo.

6.5 ALTERNATIVE PARALLEL PREFIX NETWORKS: BRENT-KUNG (BK)

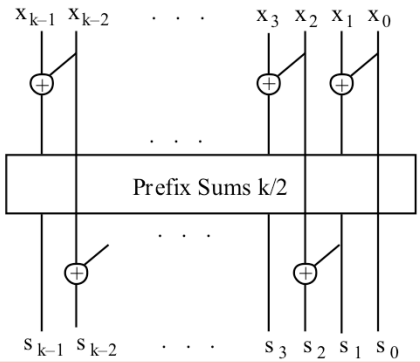
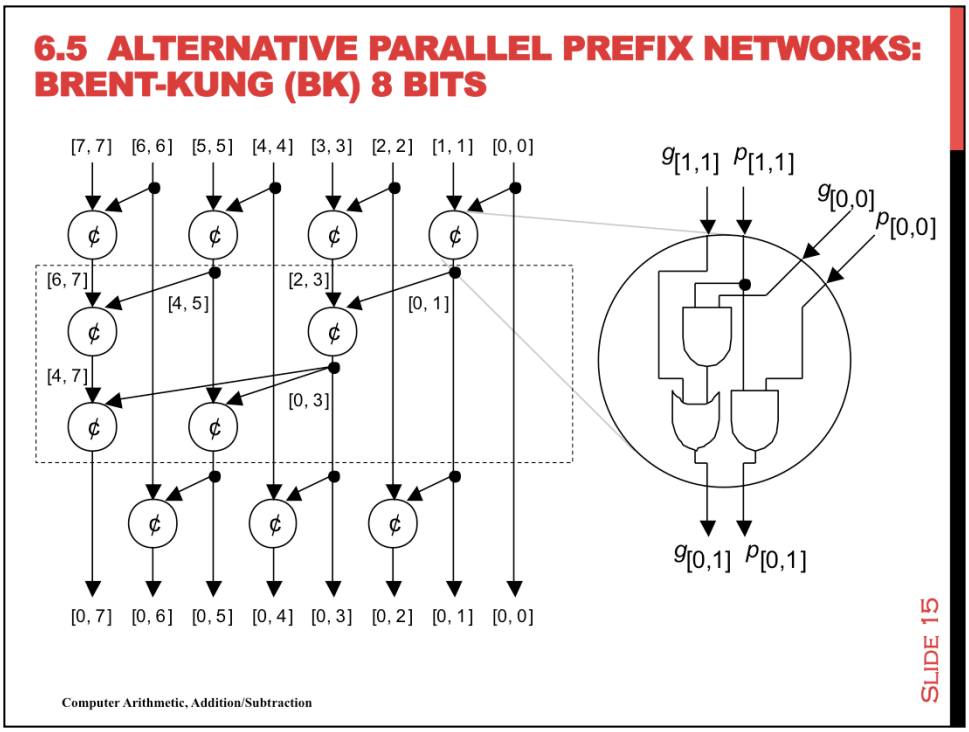


Fig. 6.8 Parallel prefix sums network built of one $k/2$-input network and $k - 1$ adders.

Delay recurrence $D(k) = D(k/2) + 2 = 2 \log_2 k - 1$ (–2 really)
Cost recurrence $C(k) = C(k/2) + k - 1 = 2k - 2 - \log_2 k$

Uma segunda maneira, ainda considerando o método de dividir e conquistar para calcular somas de prefixo, é o método proposto por Brent e Kung, que está representado na Fig. 6.8.

Aqui, as entradas são primeiro combinadas em pares para obter a seguinte sequência de comprimento k / 2:



Aqui apresentamos um somador BK de 8-bis.

O sinal entre barras "[ ]" que entram na malha de prefixos são equivalentes aos sinais porpagate e generate calculados a partir dos números os quais deseja-se somar.

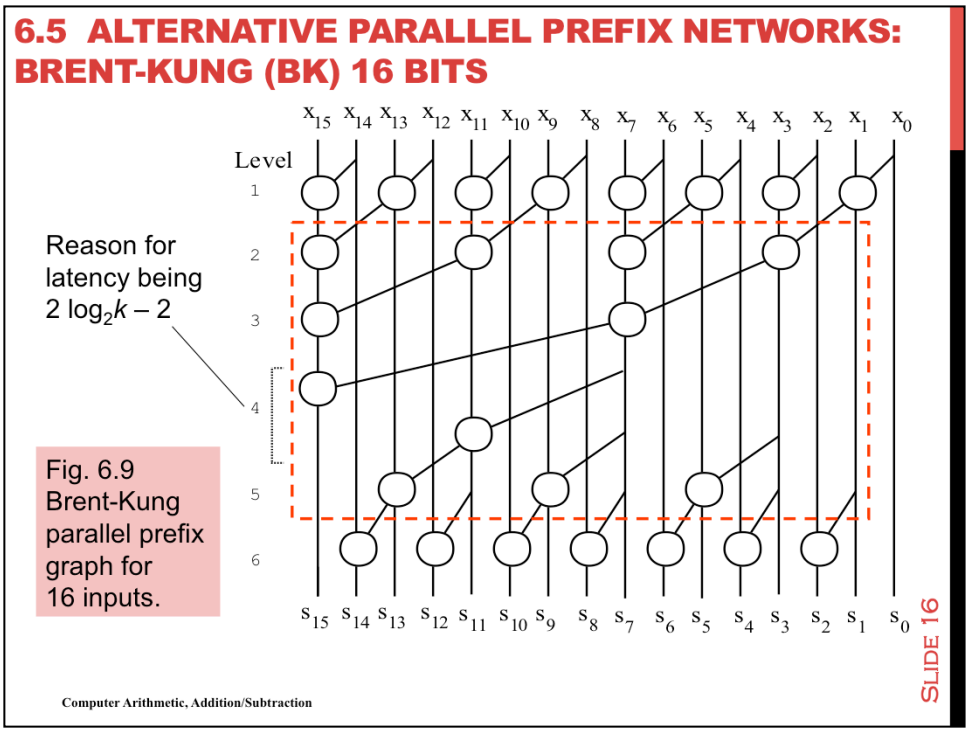
Por exemplo:

(Sinal g) (Sinal p) gi = xi or yi

[0 , 0] pi = xi and yi

Geraremos então um novo sinal em cada etapa para "p" e para "g".

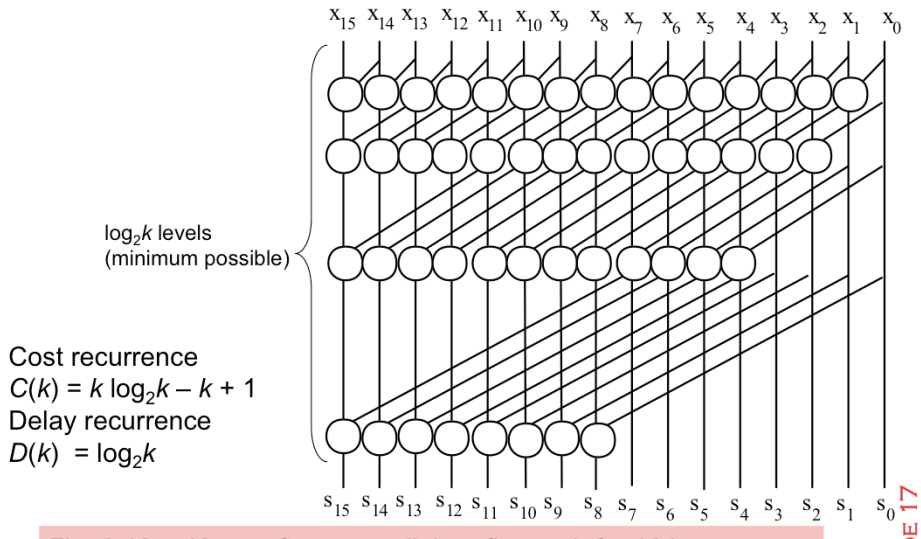
E como já havia sido citado, ao fim do cálculo e dada entradas iniciais do circuito iguais a 'x" e "y", podemos utilizar o sinal "g" para fazer a soma "$x↓i$" xor "$y↓i$" xor "$g↓ij$".



Aqui vemos um gráfico de Brent-kung de 16 bits.

Observe que, embora este gráfico da Fig. 6.9 pareça ter sete níveis, dois dos níveis próximos ao meio são independentes, o que implica um único nível de atraso. Em geral, um gráfico de prefixo paralelo de Brent-Kung com entrada k terá um atraso de 2 log2 k - 2 níveis e um custo de 2k - 2 - log2 k células.

6.5 ALTERNATIVE PARALLEL PREFIX NETWORKS: KOGGE-STONE (KS) 16 BITS



Cost recurrence
$C(k) = k \log_2 k - k + 1$
Delay recurrence
$D(k) = \log_2 k$

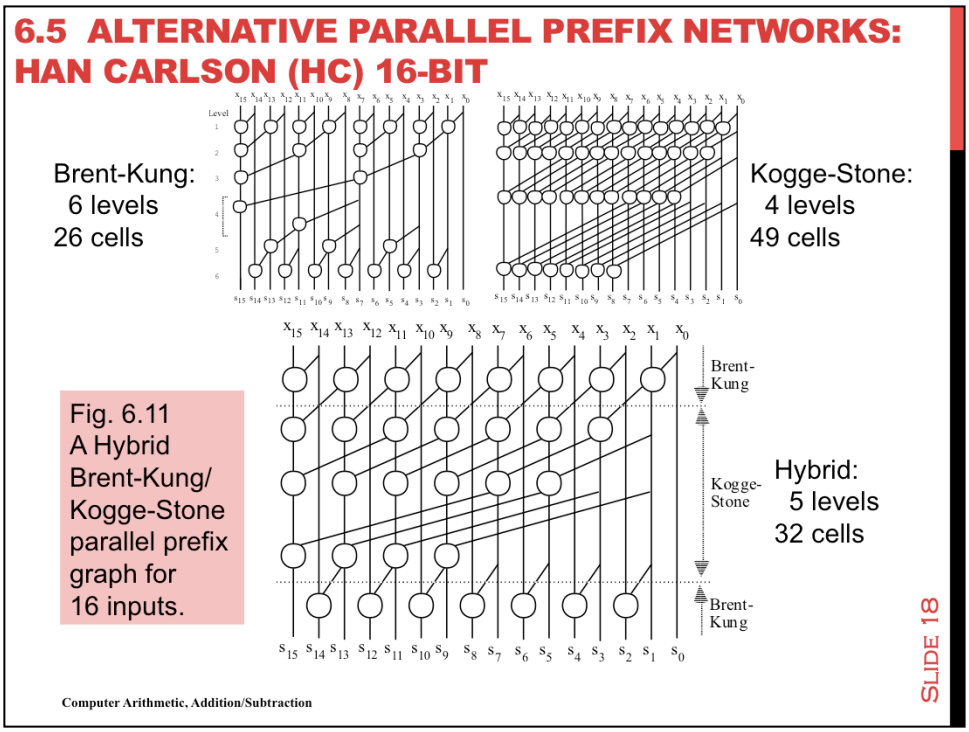
Fig. 6.10 Kogge-Stone parallel prefix graph for 16 inputs.

Computer Arithmetic, Addition/Subtraction

SLIDE 17

A Figura 6.10 mostra um gráfico de prefixo paralelo de Kogge-Stone que tem o mesmo atraso do projeto mostrado na Fig. 6.7, mas evita seu problema de fan-out por meio de distribuições dos cálculos.

Um gráfico de prefixo paralelo de Kogge-Stone com entrada k tem um atraso de log2 k níveis e um custo de k log2 k - k + 1 células. Este representa a implementação mais rápida possível de um cálculo de prefixo paralelo se apenas blocos de duas entradas forem permitidos. No entanto, seu custo pode ser proibitivo para k grandes, tanto em termos do número de células quanto da fiação densa entre elas.



Muitos outros projetos de rede de prefixo paralelo são possíveis.

Por exemplo, realizando a combinação do circuito de Brent – Kung e Kogge-Stone para formar designs híbridos. Na Fig. 6.11, os quatro elementos intermediários dos seis níveis no projeto da Fig. 6.9 (representando um cálculo de prefixo paralelo de oito entradas) foram substituídos pela rede Kogge-Stone de oito entradas. O design resultante tem cinco níveis e 32 células, colocando-o entre os designs Brent – Kung original (seis níveis, 26 células) e Kogge – Stone original (quatro níveis, 49 células).

6.5 ALTERNATIVE PARALLEL PREFIX NETWORKS: SPEED-COST TRADEOFFS IN CARRY NETWORKS

Method	Delay	Cost
Ladner-Fischer	$\log_2 k$	$(k/2) \log_2 k$
Kogge-Stone	$\log_2 k$	$k \log_2 k - k + 1$
Brent-Kung	$2 \log_2 k - 2$	$2k - 2 - \log_2 k$
Han Carlson	$\log_2 k + 1$	$(k/2) \log_2 k$

k= 16-bits

Method	Delay	Cost
Ladner-Fischer	**4**	32
Kogge-Stone	**4**	49
Brent-Kung	6	**26**
Han Carlson	5	32

Aqui vemos uma relação entre custo e atraso dos gráficos que foram apresentados.

PROBLEMAS

Problema 6.1 Obtenha os gráficos para valores pares $n \in [4, 32]$ dos somadores Ladner-Fisher (LF), Brent-Kung (BK), Kogge-Stone (KS), e Han-Carlson (HC) para:

a)Área.

b)Atraso.

c)Produto área atraso (AT)

d)Produto área atraso quadrado (AT^2).

e)Fan-out.

Indique qual é a melhor opção para as diferentes figuras de mérito apresentadas acima.

SLIDE 20

Gabarito no Moodle

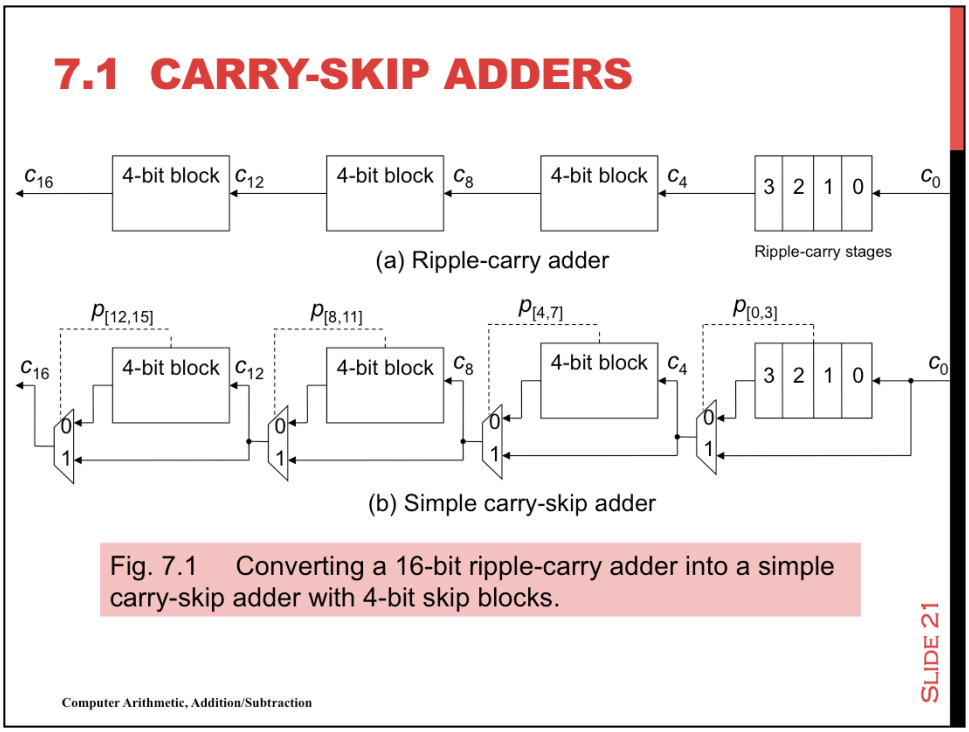
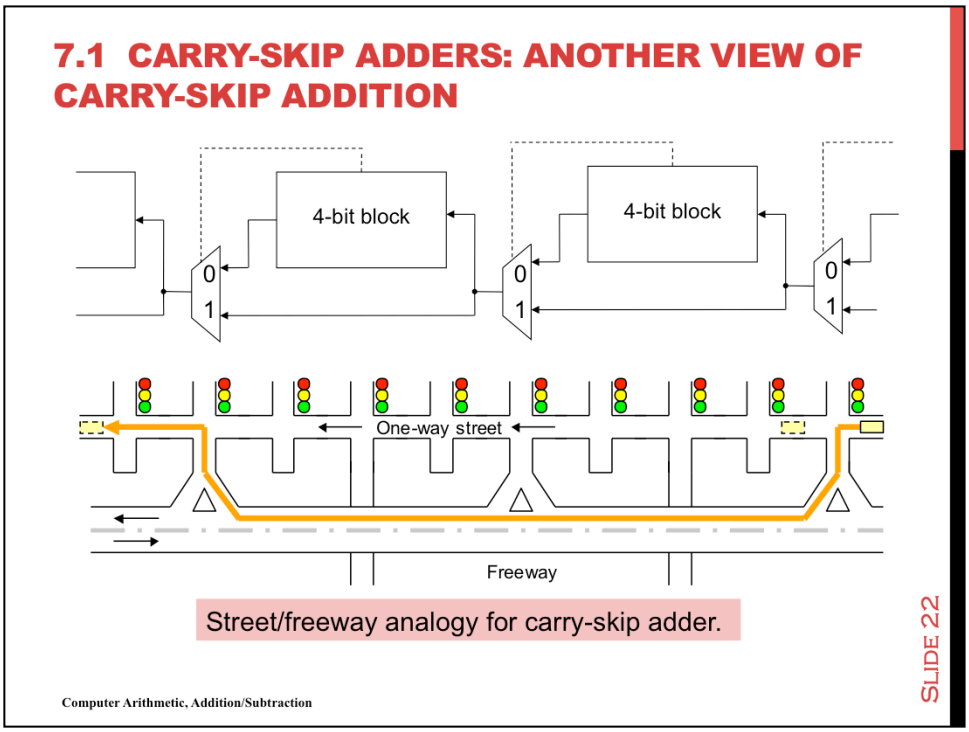


Fig. 7.1 Converting a 16-bit ripple-carry adder into a simple carry-skip adder with 4-bit skip blocks.

Agora, considere um grupo ou bloco de 4 bits em um somador “ripple carry”, iniciando de um estágio i ao estágio i +3, onde i é um múltiplo de 4. Para o estágio i, um carry se propaga por meio desse grupo de 4 bits se e somente se ele se propaga por todos os quatro estágios. Assim, um sinal de propagação de grupo é definido como p [i, i + 3] = (pipi + 1) (pi + 2) (pi + 3), e que por sua vez são computáveis a partir de sinais de propagação individuais por uma única porta AND de quatro entradas.

Desta forma, para acelerar a propagação do carry, pode-se estabelecer caminhos de “desvio” ou “pular” (skip) em torno de blocos de 4 bits, como mostrado na Fig. 7.1b. Observe que o bloco de 4 bits possui um sinal de controle (enable) para o multiplexidor, que é proveniente da função p [i, i + 3] = (pipi + 1)(pi + 2) (pi + 3)].



Desta forma faremos uma analogia. Suponha que o tempo de condução seja o mesmo para um quarteirão da cidade ou um “quarteirão” da rodovia (entre duas saídas).

Caso se saiba se haverá um sinal vermelho em um dos conjuntos de 4 sinais de transito adjacentes em um determinaod bloco, podemos evitar o bloco completo pegando a “via-expressa”. [Sabemos que “o sinal estará vermelho” pela função p [i, i + 3] = (pipi + 1) (pi + 2) (pi + 3)]

7.3 CARRY-SELECT ADDERS

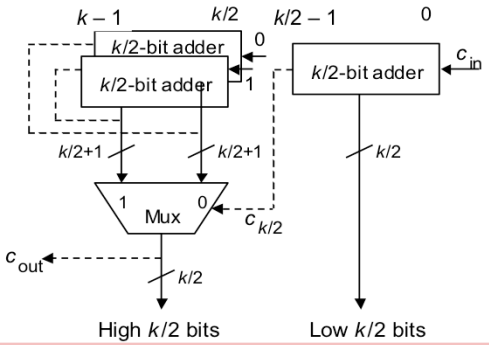


Fig. 7.9 Carry-select adder for *k*-bit numbers built from three *k*/2-bit adders.

$$C_{\text{select-add}}(k) = 3C_{\text{add}}(k/2) + k/2 + 1$$
$$T_{\text{select-add}}(k) = T_{\text{add}}(k/2) + 1$$

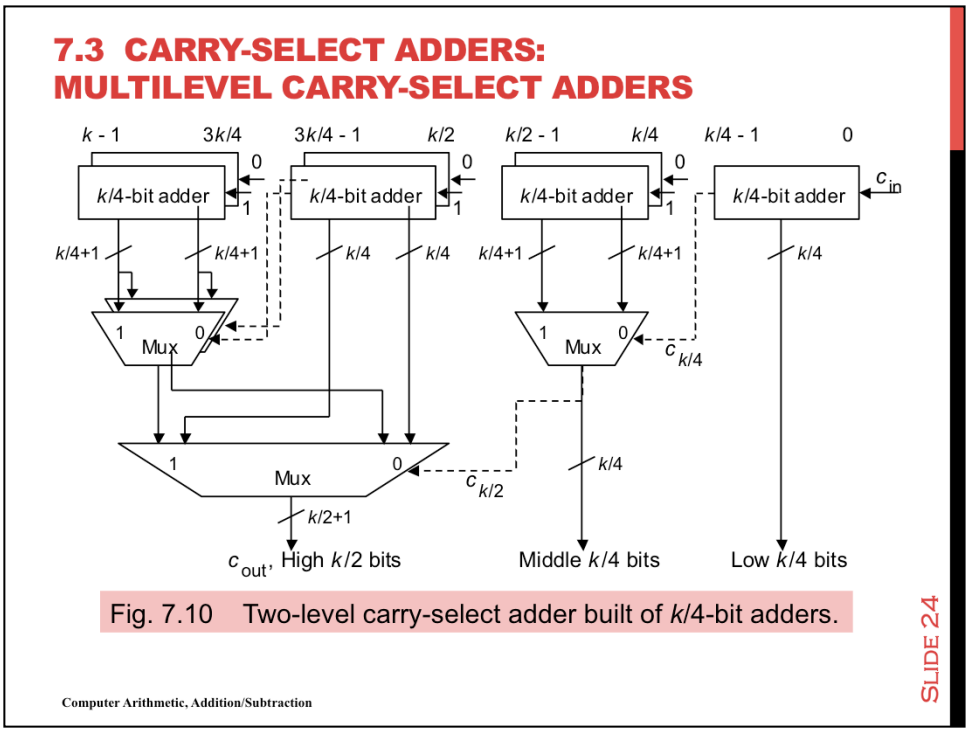
Computer Arithmetic, Addition/Subtraction

SLIDE 23

Um somador de carry-select (nível único) combina três somadores de k / 2 bits de qualquer projeto em um somador de k bits (Observar na Fig. 7.9).

Um somador de k / 2 bits é usado para calcular a metade inferior da soma de k bits diretamente.

Dois somadores k / 2 bits são usados para calcular os k / 2 bits superiores da soma e o carry-out em dois cenários diferentes: ck / 2 = 0 ou ck / 2 = 1. Os valores corretos para o carry-out do somador e os bits de soma nas posições k / 2 a k −1 são selecionados quando o valor de ck / 2 se torna conhecido.



A Figura 7.10 mostra como a ideia de carry-select pode ser levada um passo adiante para obter um somador de carry-select de dois níveis.

Os bits de soma e carry-ou são calculados para cada bloco de k / 4 bits (exceto para o mais à direita) em dois cenários.

Os três multiplexadores de primeiro nível, cada um dos quais com k / 4 + 1 bits de largura, mesclam os resultados de blocos de k / 4 bits com os de blocos de k / 2 bits. Observe como os sinais de transferência dos somadores que abrangem as posições de bit k / 2 a 3k / 4 - 1 são usados para selecionar os k / 4 bits mais significativos da soma nos dois cenários de ck / 2 = 0 ou ck / 2 = 1. Nesse estágio, k / 2 bits da soma final são conhecidos.

O multiplexador de segundo nível, que tem k / 2 + 1 bits de largura, é usado para selecionar os valores apropriados para os k / 2 bits superiores da soma (posições k / 2 a k - 1) e do carry-out.

PROBLEMAS

Problema 6.2 Faça a comparação da área, atraso e produto área atraso (AT) para um *Carry-Select-Adder* de 64-bits dividido em a) 4 somas e b) 2 somas. Assuma que a área e o atraso do somador está expressado como $n \times A_{adder}$, $n \times T_{adder}$, respectivamente, sendo n o número de bits, e a área e atraso do multiplexador 2:1 como uma unidade $(2/3) \times A_{adder}$, $(2/3) \times T_{adder}$.

SLIDE 25

Gabarito no Moodle

7.4 ANALYSIS OF CARRY PROPAGATION

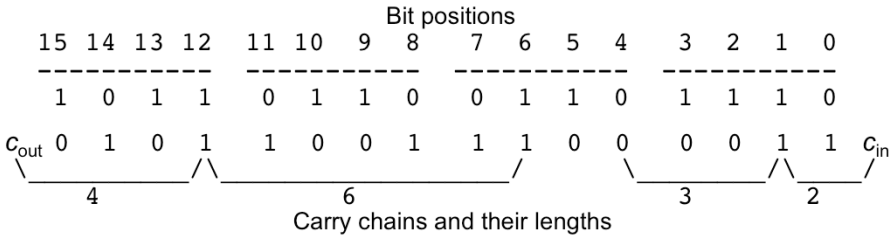


Fig. 7.11 Example addition and its carry propagation chains.

Given binary numbers with random bits, for each position *i* we have

Probability of carry generation = ¼ (both 1s)
Probability of carry annihilation = ¼ (both 0s)
Probability of carry propagation = ½ (different)

Aqui vemos um relação de probabilidade de propagação de carry.

O comprimento médio da cadeia de carry mais longa na adição de k bits é estritamente menor que log 2 k.

PROBLEMAS

Problema 6.3. Uma entrada fixa A = 31727_{10} deve ser somada com entradas variáveis B e C, todas de 16-bits. Ditas entradas variáveis só podem ter os seguintes valores:

- B = {2638_{10}, 31439_{10}, 14923_{10}}.
- C = {3041_{10}, 15343_{10}, 3192_{10}}.

Qual das somas pode ser implementada com um atraso menor?

Problema 6.4. Dois vetores de 12 bits A e B precisam de ser somados, onde A é sempre múltiplo $100_{(10)}$ e B múltiplo de $48_{(10)}$. Considerando os tempos de atraso do problema 6.2, obtenha a soma dos vectores com um atraso máximo de $5 \times T_{adder}$.

SLIDE 27

Gabarito no Moodle

7.6 MODULAR TWO-OPERAND ADDERS

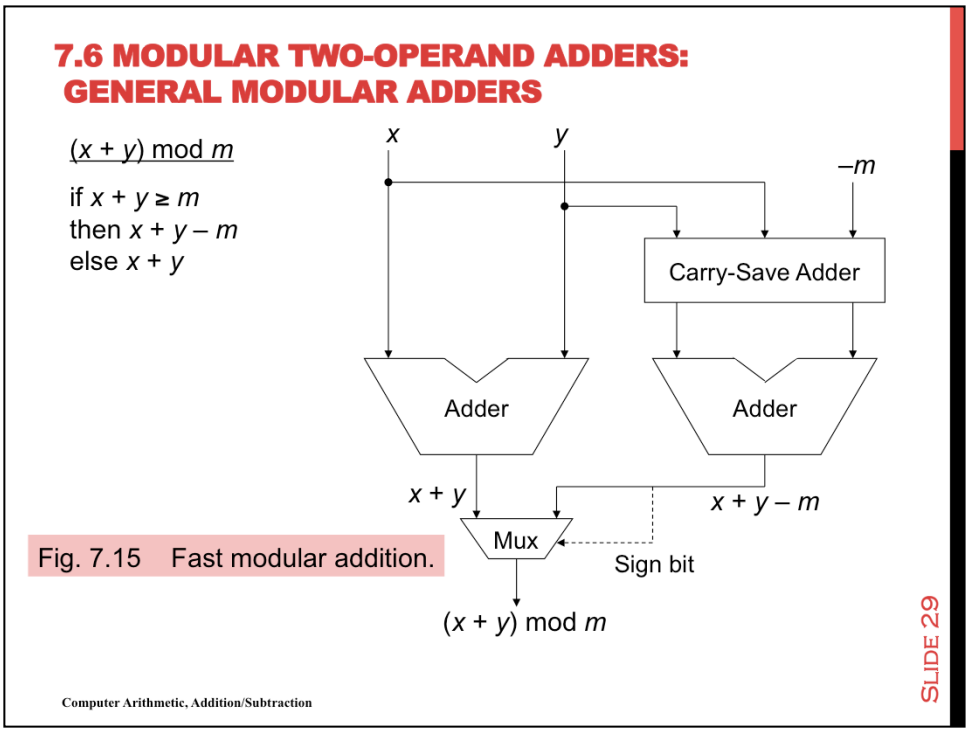
mod-2^k: Ignore carry out of position $k - 1$

mod-$(2^k - 1)$: Use End-Around Carry (EAC)

mod-$(2^k + 1)$: Use Inverted End-Around Carry (IEAC)

Number	Std. binary
0	0 0 . . . 0 0 0
1	0 0 . . . 0 0 1
2	0 0 . . . 0 1 0
.	.
.	.
.	.
2^k-1	0 1 . . . 1 1 1
2^k	1 0 . . . 0 0 0

Lembrando da aula 4 temos:



Considere a soma (x + y) mod m, sendo o módulo m igual a 15.

Caso x+y for igual a 4 por exemplo e porntanto menor do que 15 teremos que (x + y) mod m é igual ao próprio 4.

Entretanto caso a soma x+y for igual a 19, teríamos que o resultado correto de (x + y) mod m seria igual a 4.

Desta forma, para implementãção do circuito que realizará tal contagem teremos que realizar tanto a soma x+y normal quanto a soma de x+y-m. Escolheremos o resultado de x+y quando o resultado for menor do que o módulo. Do contrário escolheremos o resultado de x+y-m, por meio do mux.

PROBLEMAS

Problema 6.5. Implemente os seguintes somadores modulares:

a) $|A+B|_{29}$.

b) $||A+B|_{27}+C|_{29}$.

c) $||A+B|_{11}+C|_{13}$.

d) $|A+B|_{59}$

e) $|A+B|_{15}$

SLIDE 30

Gabarito no Moodle