

Chapter 3

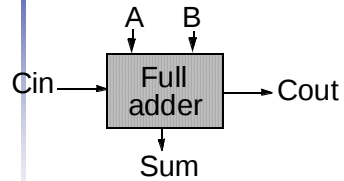
Arithmetic Circuits

Instructor: Prof. Peter Lian
Department of Electrical
Engineering & Computer Science
Lassonde School of Engineering
York University

Adders

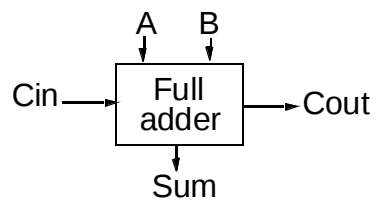
Ripple-Carry Adder
Carry-Bypass Adder
Carry-Select Adder
Carry-Lookahead Adder

Full-Adder



A	B	C_i	S	C_o	Carry status
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

The Binary Adder



$$S = A \oplus B \oplus C_i$$

$$= A\bar{B}\bar{C}_i + \bar{A}B\bar{C}_i + \bar{A}\bar{B}C_i + ABC_i$$

$$C_o = AB + BC_i + AC_i = AB + C_i(A \oplus B)$$

Generate, Propagate, and Delete

Define 3 new variable which ONLY depend on A, B

$$\text{Generate (G)} = AB$$

$$\text{Propagate (P)} = A \oplus B$$

$$\text{Delete} = \overline{A} \overline{B}$$

$$C_o(G, P) = G + PC_i$$

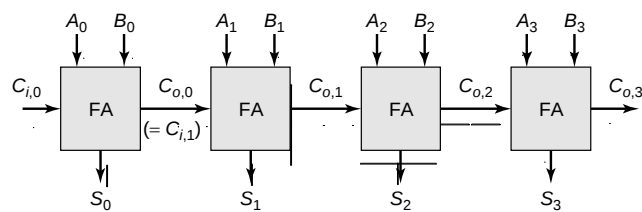
$$S(G, P) = P \oplus C_i$$

Can also derive expressions for S and C_o based on D and P

Note that we will be sometimes using an alternate definition for

$$\text{Propagate (P)} = A + B$$

The Ripple-Carry Adder



Worst case delay linear with the number of bits

$$t_d = O(N)$$

$$t_{adder} = (N-1)t_{carry} + t_{sum}$$

Goal: Make the fastest possible carry path circuit

Activity 1

Derive the values of A_k and B_k ($k=0, \dots, N-1$) so that the worst case delay is obtained for the ripple-carry adder.

Activity 1

Derive the values of A_k and B_k ($k=0, \dots, 7$) so that the worst case delay is obtained for the ripple-carry adder.

Solution:

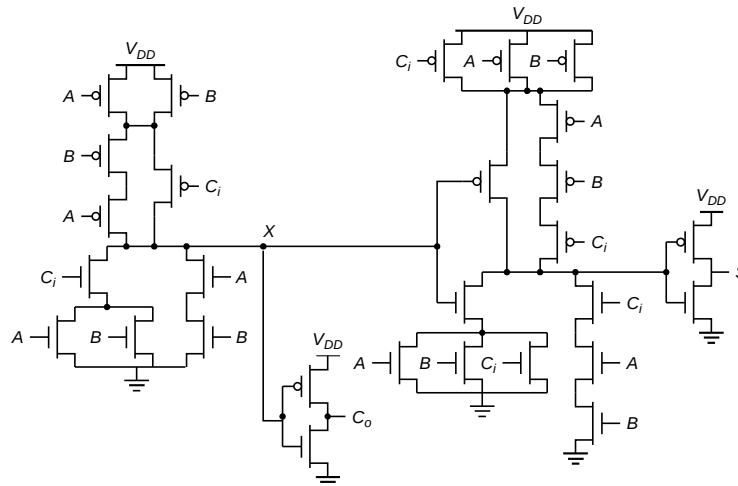
1. The worst case condition requires that a carry be generated at the *lsb* position $\rightarrow A_0$ and $B_0 = 1$
2. All other stages must be in propagate mode $\rightarrow$ either A_i or B_i must be high.
3. *msb* should change status $\rightarrow A_i$ and B_i both equal to 0 or 1.

Based on above conditions:

A=00000001

B=01111111

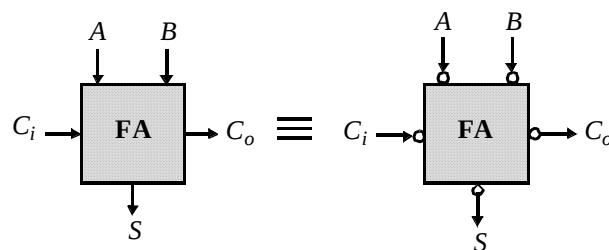
Complimentary Static CMOS Full Adder



$$C_o = AB + BC_i + AC_i, S = ABC_i + \bar{C}_o(A + B + C_i)$$

28 Transistors

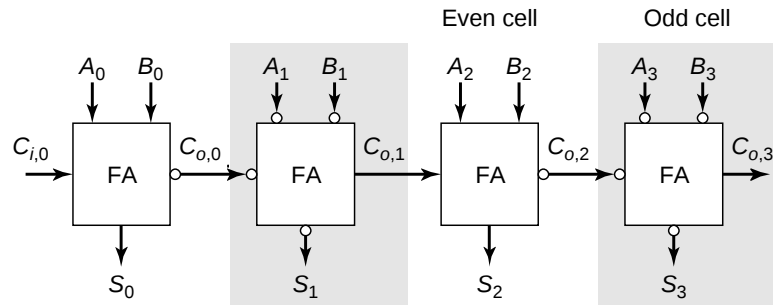
Inversion Property



$$\bar{S}(A, B, C_i) = S(\bar{A}, \bar{B}, \bar{C}_i)$$

$$\bar{C}_o(A, B, C_i) = C_o(\bar{A}, \bar{B}, \bar{C}_i)$$

Minimize Critical Path by Reducing Inverting Stages



Exploit Inversion Property

A Better Structure: The Mirror Adder

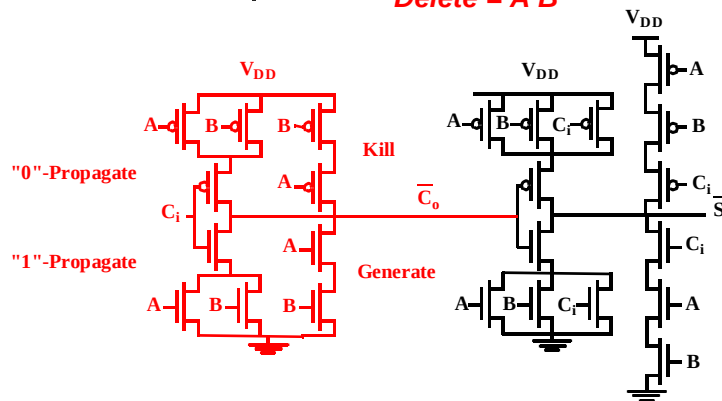
$$C_o(G, P) = G + PC_i$$

$$S(G, P) = P \oplus C_i$$

Generate (G) = AB

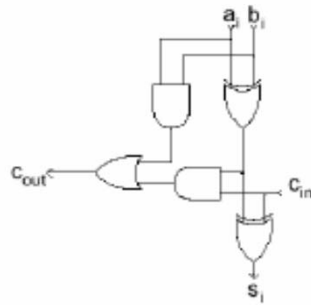
Propagate (P) = A ⊕ B

Delete = $\bar{A} \bar{B}$

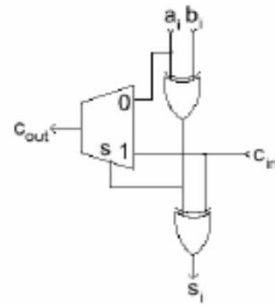


24 transistors

Full Adder Implementation



Standard CMOS

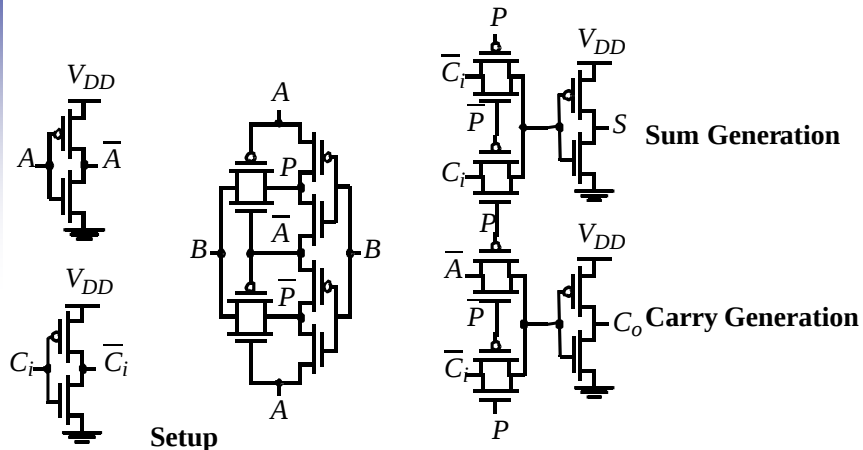


Multiplexer-based

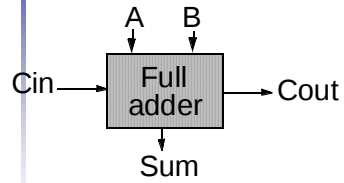
$$S = A \oplus B \oplus C_i$$

$$C_o = AB + C_i(A \oplus B)$$

Transmission Gate Full Adder



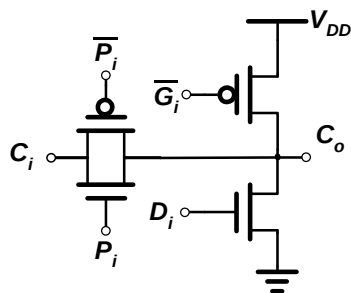
Full-Adder



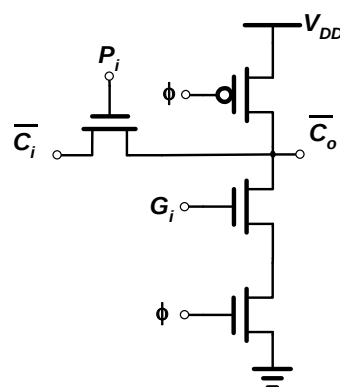
A	B	C_i	S	C_o	<i>Carry status</i>
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

Manchester Carry Chain

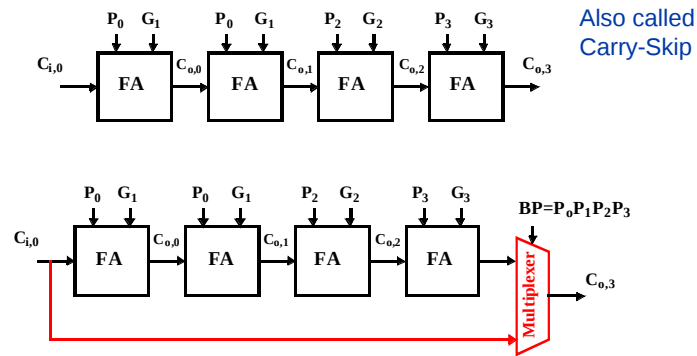
Static



Dynamic

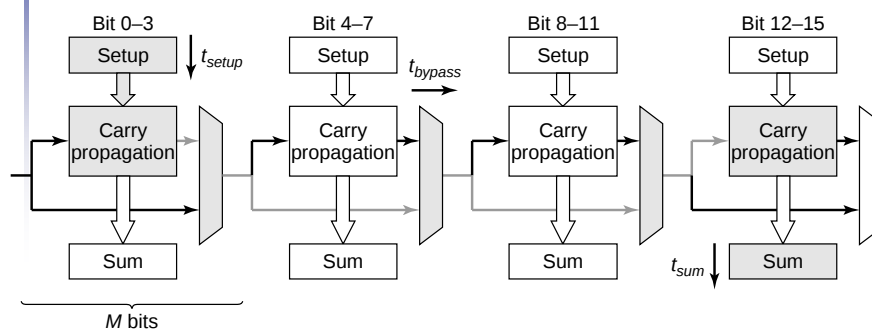


Carry-Bypass Adder



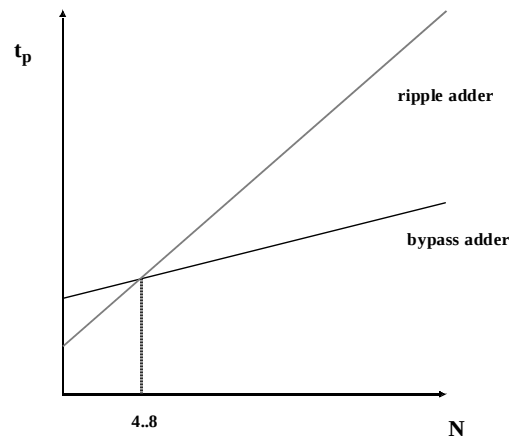
Idea: If (P_0 and P_1 and P_2 and $P_3 = 1$) then $C_{o3} = C_0$, else “kill” or “generate”.

Carry-Bypass Adder

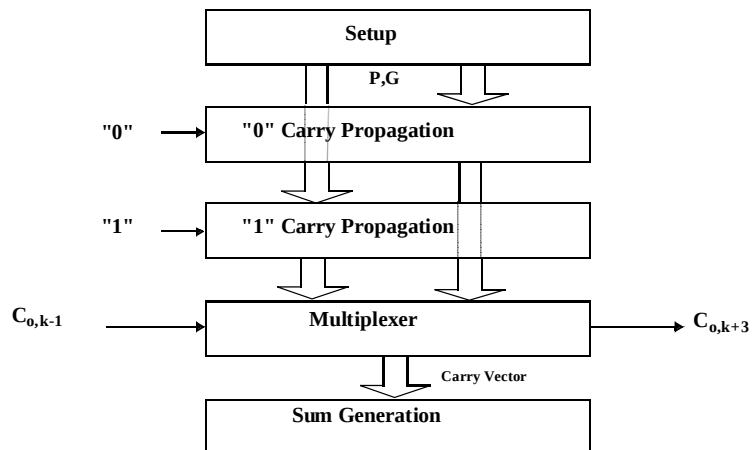


$$t_{adder} = t_{setup} + M_{t_{carry}} + (N/M-1)t_{bypass} + (M-1)t_{carry} + t_{sum}$$

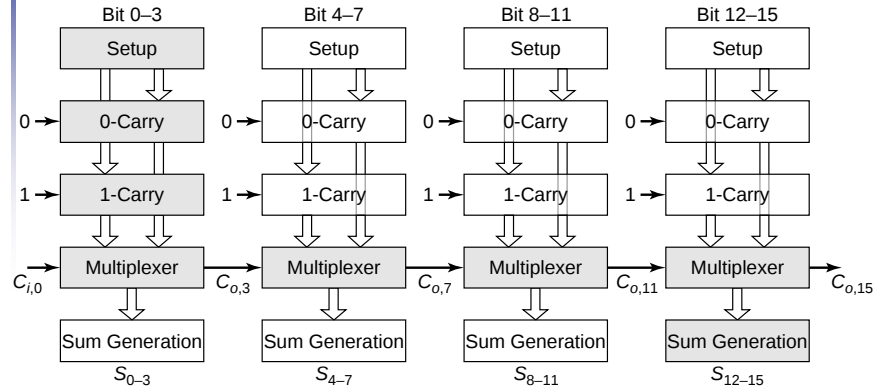
Carry Ripple vs. Carry-Bypass



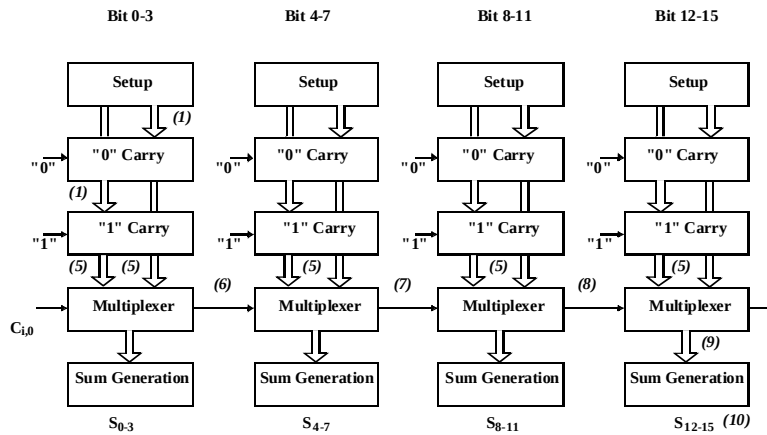
Carry-Select Adder



CSA: Critical Path

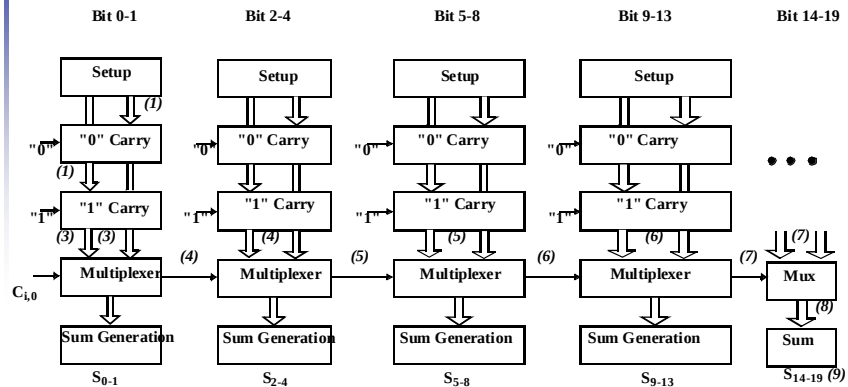


Linear Carry Select



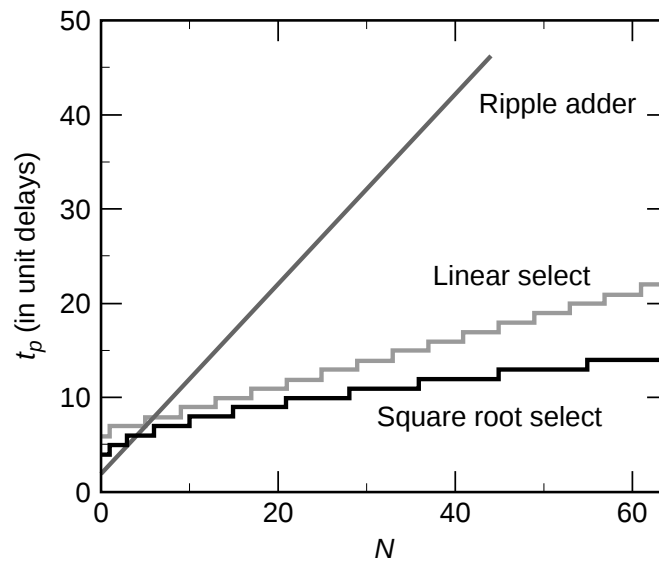
$$t_{add} = t_{setup} + \left(\frac{N}{M}\right)t_{carry} + Mt_{mux} + t_{sum}$$

Square Root Carry Select



$$t_{add} = t_{setup} + P \cdot t_{carry} + (\sqrt{2N})t_{mux} + t_{sum}$$

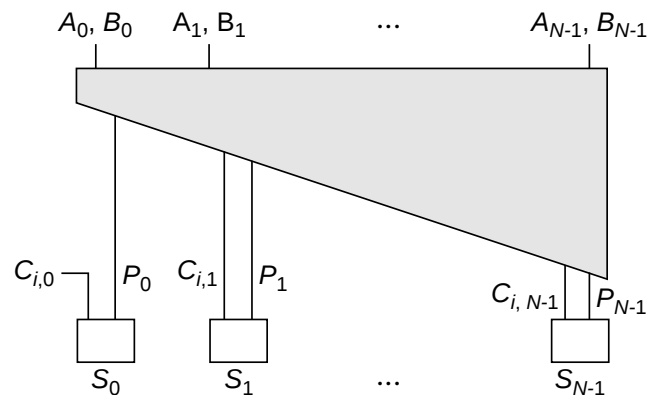
Adder Delays - Comparison



Carry-Lookahead Adders

- Carry-Lookahead Adder – CLA
- Adder trees
 - Radix of a tree
 - Minimum depth trees
 - Sparse trees
- Logic manipulation
 - Conventional vs. Ling
 - Stack height limiting

LookAhead - Basic Idea



$$C_{0,k} = f(A_k, B_k, C_{0,k-1}) = G_k + P_k C_{0,k-1}$$

Generate, Propagate, and Delete

Define 3 new variable which ONLY depend on A, B

$$\text{Generate (G)} = AB$$

$$\text{Propagate (P)} = A \oplus B$$

$$\text{Delete} = \overline{A} \overline{B}$$

$$C_o(G, P) = G + PC_i$$

$$S(G, P) = P \oplus C_i$$

Can also derive expressions for S and C_o based on D and P

Note that we will be sometimes using an alternate definition for

$$\text{Propagate (P)} = A + B$$

Lookahead Adder

Lookahead Equations

Position i : $c_i = g_i + p_i c_{i-1}$

Position $i+1$:
$$\begin{aligned} c_{i+1} &= g_{i+1} + p_{i+1} c_i \\ &= g_{i+1} + p_{i+1} (g_i + p_i c_{i-1}) \\ &= g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_{i-1} \end{aligned}$$

Carry exists if:

- Generated in stage $i+1$
- Generated in stage i and propagated through $i+1$
- Propagated through both i and $i+1$

Block Lookahead

- Example for 4th bit carry:

$$c_{t+4} = g_{i+3} + p_{i+3}g_{i+2} + p_{i+3}p_{i+2}g_{i+1} + p_{i+3}p_{i+2}p_{i+1}g_i \\ + p_{i+3}p_{i+2}p_{i+1}p_i c_{i-1}$$

- Block generate and block propagate:

$$G_{i,i+3} = g_{i+3} + p_{i+3}g_{i+2} + p_{i+3}p_{i+2}g_{i+1} + p_{i+3}p_{i+2}p_{i+1}g_i$$

$$P_{i,i+3} = p_{i+3}p_{i+2}p_{i+1}p_i$$

$$c_{i+4} = G_{i,i+3} + P_{i,i+3}c_{i-1}$$

Look-Ahead: Topology

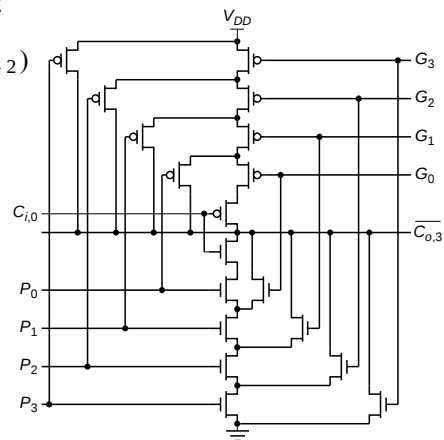
Expanding Lookahead equations:

$$C_{o,k} = G_k + P_k(G_{k-1} + P_{k-1}C_{o,k-2})$$

All the way:

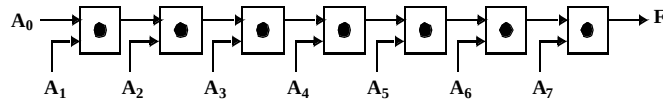
$$C_{o,k} = G_k + P_k(G_{k-1} + P_{k-1}(\dots + P_1(G_0 + P_0C_{i,0})))$$

Mirror implementation

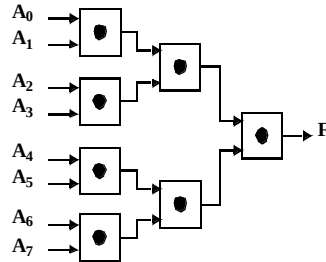


$$c_{o,3} = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_{i,0}$$

Logarithmic Look-Ahead Adder



$$t_p \sim N$$



$$t_p \sim \log_2(N)$$

Carry Lookahead Trees

$$C_{o,0} = G_0 + P_0 C_{i,0}$$

$$C_{o,1} = G_1 + P_1 G_0 + P_1 P_0 C_{i,0} = (G_1 + P_1 G_0) + (P_1 P_0) C_{i,0} = G_{1,0} + P_{1,0} C_{i,0}$$

$$C_{o,2} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{i,0} = G_{2,1} + P_{2,1} C_{o,0}$$

$$C_{o,3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{i,0} = G_{3,2} + P_{3,2} C_{o,1}$$

...

Can continue building the tree hierarchically.

Tree Adders

$$P_G = p_m \cdot p_l \quad m: \text{more significant bit}$$

$$G_G = g_m + p_m \cdot g_l \quad l: \text{less significant bit}$$

- Start from the input P, G and continue up the tree.
- dot operation:

$$\begin{aligned}(g, p) &= (g_m, p_m) \bullet (g_l, p_l) \\ &= (g_m + p_m \cdot g_l, p_m \cdot p_l)\end{aligned}$$

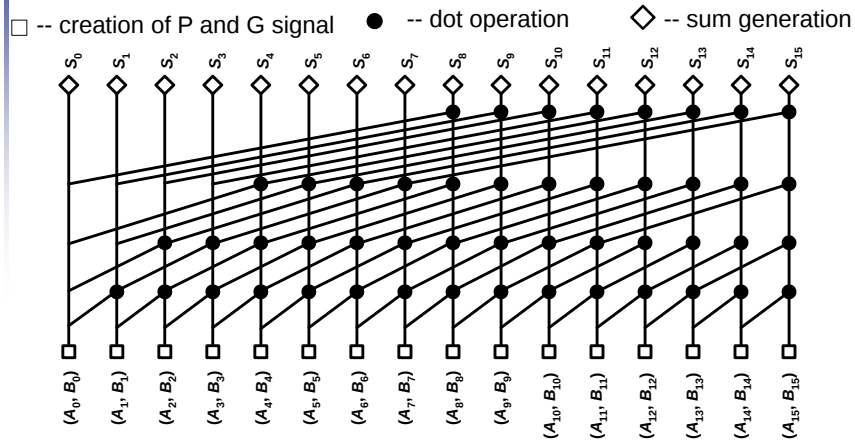
- Kogge, Stone, Trans on Comp. 1973

Activity 2

Show that $C_{0,3}$ of a 4-bit ripple-carry adder can be expressed as a function of 2 group carries, i.e.

$$C_{0,3} = (G_{3,2}, P_{3,2}) \cdot (G_{1,0}, P_{1,0}) \cdot (C_{i,0}, 0)$$

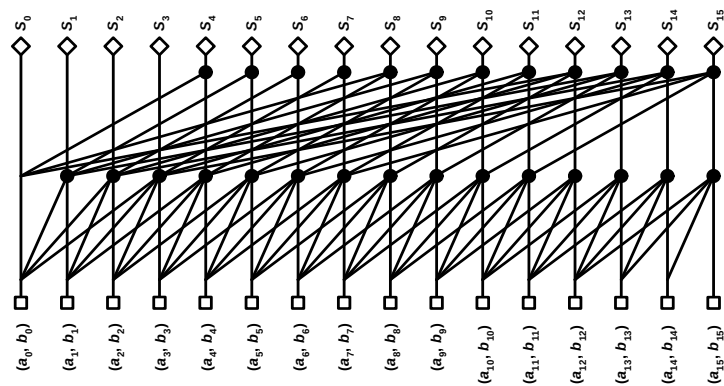
Tree Adders: Radix 2



16-bit radix-2 Kogge-Stone tree:

1. Building block of order 2
2. requires 49 dot operations

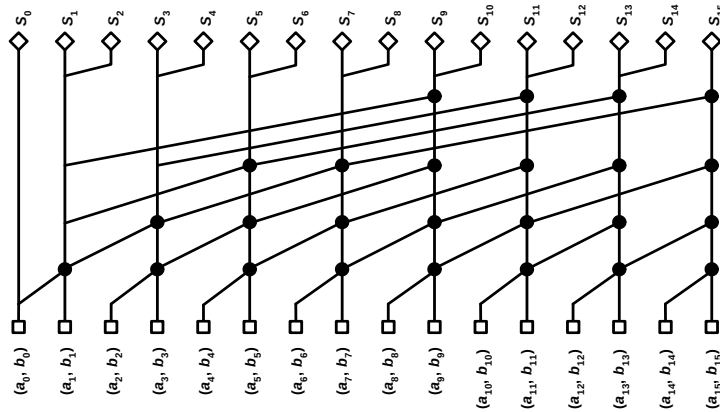
Tree Adders: Radix 4



16-bit radix-4 Kogge-Stone Tree:

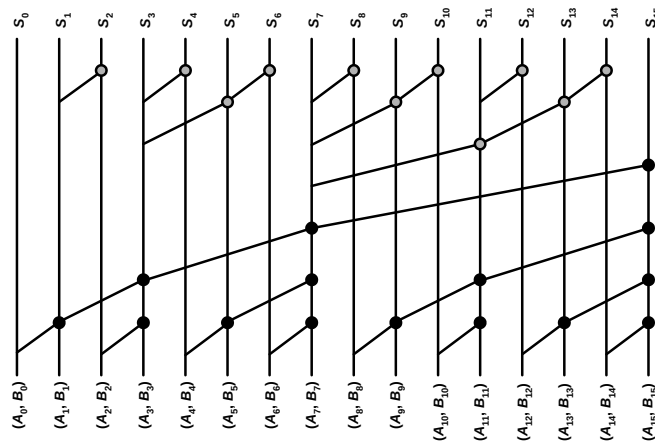
1. Build block of order 4
2. Two stages of carry logic
3. Complex circuit for group of 4 dot operation

Tree Adders: Sparse Trees



16-bit radix-2 sparse tree with sparseness of 2

Tree Adders: Brent-Kung Tree



Brent-Kung Tree:

1. Generate forward carries at 2^N-1 positions
2. Use only 27 dot operations

Ling Adder

- Variation of CLA

Conventional

Ling's equations

$$p_i = a_i \oplus b_i$$

$$t_i = a_i + b_i$$

$$g_i = a_i \cdot b_i$$

$$g_i = a_i \cdot b_i$$

$$G_i = g_i + p_i \cdot G_{i-1}$$

$$H_i = g_i + t_{i-1} \cdot H_{i-1}$$

$$S_i = p_i \oplus G_{i-1}$$

$$S_i = t_i \oplus H_i + g_i t_{i-1} H_{i-1}$$

Ling, IBM J. Res. Dev., 5, 1981

Ling Adder

- Conventional radix-4

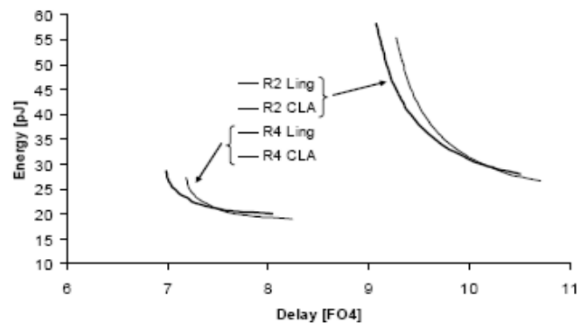
$$G_3 = g_3 + t_3 g_2 + t_3 t_2 g_1 + t_3 t_2 t_1 g_0$$

- Ling radix-4

$$\begin{aligned} H_3 &= g_3 + t_2 g_2 + t_2 t_1 g_1 + t_2 t_1 t_0 g_0 \\ &= g_3 + g_2 + t_2 g_1 + t_2 t_1 g_0 \end{aligned}$$

- Reduces the stack height (or width)
- Reduces input loading

Ling versus CLA



R. Zlatanovici and B. Nikolic, ESSCIRC 2003

Multipliers

The Binary Multiplication

$$\begin{aligned}
 Z = \vec{X} \times \vec{Y} &= \sum_{k=0}^{M+N-1} Z_k 2^k \\
 &= \left(\sum_{i=0}^{M-1} X_i 2^i \right) \left(\sum_{j=0}^{N-1} Y_j 2^j \right) \\
 &= \sum_{i=0}^{M-1} \left(\sum_{j=0}^{N-1} X_i Y_j 2^{i+j} \right)
 \end{aligned}$$

with

$$\begin{aligned}
 X &= \sum_{i=0}^{M-1} X_i 2^i \\
 Y &= \sum_{j=0}^{N-1} Y_j 2^j
 \end{aligned}$$

The Binary Multiplication

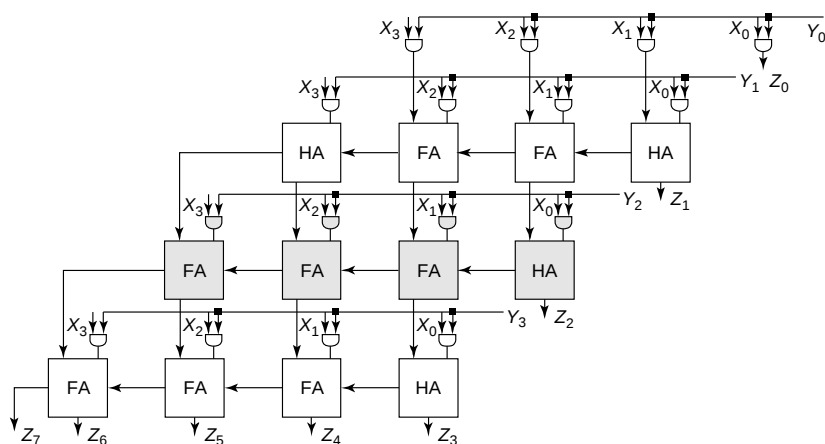
		1 0 1 0 1 0	Multiplicand: N bits
x		1 0 1 1	Multiplier: M bits
		1 0 1 0 1 0	} Partial products
		1 0 1 0 1 0	
		0 0 0 0 0 0	
		0 0 0 0 0 0	
+	1 0 1 0 1 0		
	1 1 1 0 0 1 1 1 0		Result: N+M bits

AND operation

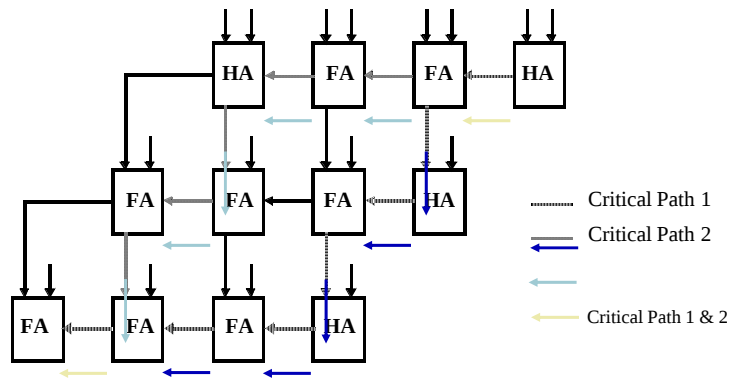
Shift-and-Add Multiplier

- Standard adder and shift-in the multiplicand
- Shift the result as well and add
- N cycles
- Parallel adders and more hardware (adders) instead.

The Array Multiplier

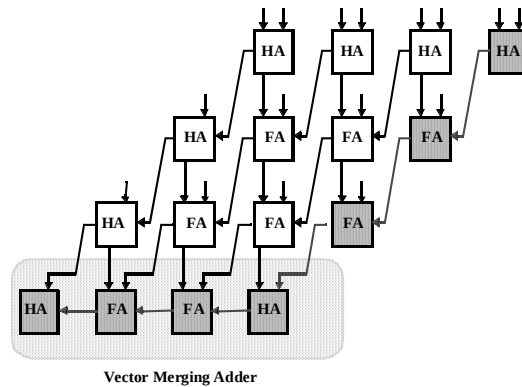


The MxN Array Multiplier— Critical Path



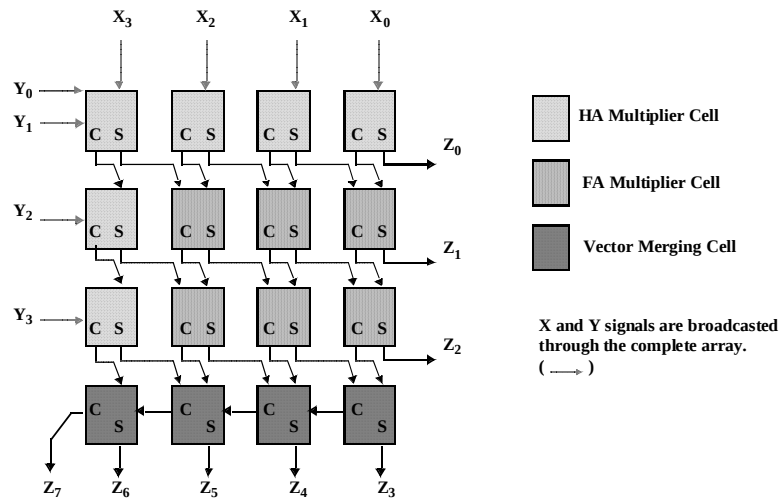
$$t_{mult} \approx [(M-1) + (N-2)]t_{carry} + (N-1)t_{sum} + (N-1)t_{and}$$

Carry-Save Multiplier



$$t_{mult} = (N-1)t_{carry} + (N-1)t_{and} + t_{merge}$$

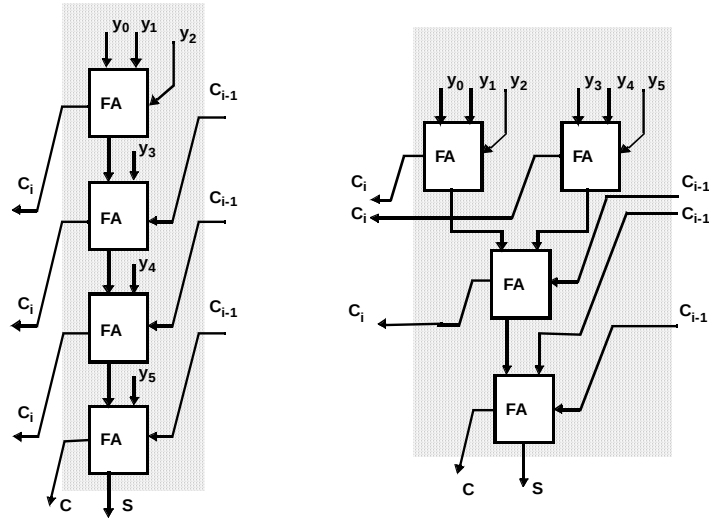
Multiplier Floorplan



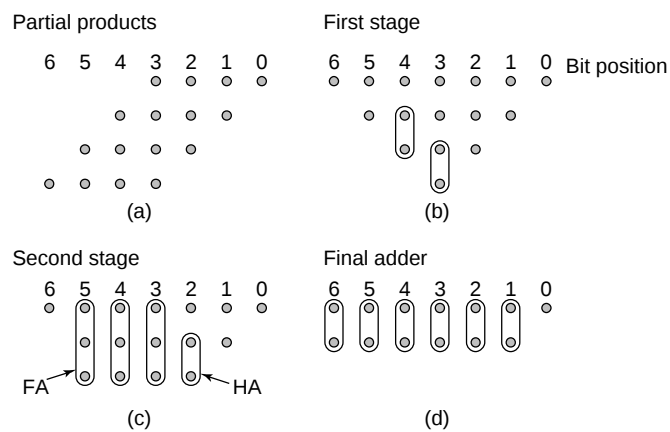
Multipliers

- Partial product generation
- Partial product accumulation
- Final summation

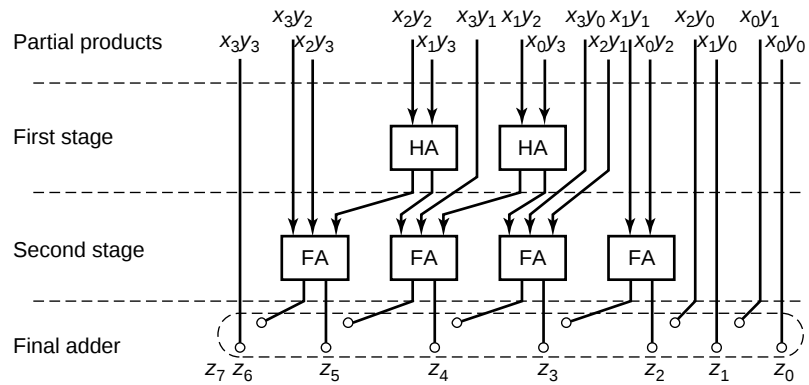
Wallace-Tree Multiplier



Wallace-Tree Multiplier



Wallace-Tree Multiplier



Multipliers —Summary

- Optimization Goals Different Vs Binary Adder
- Once Again: Identify Critical Path
- Other possible techniques
 - Logarithmic versus Linear (Wallace Tree Mult)
 - Data encoding (Booth)
 - Pipelining

FIRST GLIMPSE AT SYSTEM LEVEL OPTIMIZATION

Acknowledgement

- The materials are adopted from “Digital Integrated Circuits: A Design Perspective” by Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic, 2nd Edition, 2003, Pearson.