

EECS 151/251A

Discussion 9

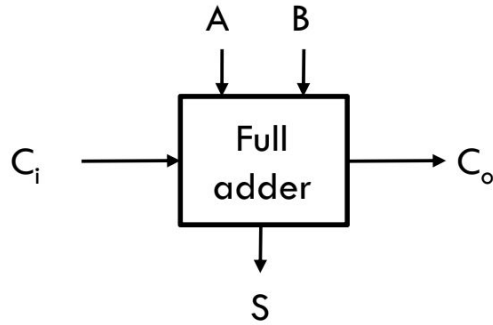
Zhaokai Liu

Agenda

- Adders
 - Single-bit Full Adder
 - Ripple-carry Adder
 - Carry-bypass Adder
 - Carry-lookahead Adder
 - CLA Trees
- Multipliers
 - Array Multiplier w/o and w/ CSA
 - Wallace Tree
 - Booth Recording
 - Baugh-Wooley Multiplication

Adders

Single-bit Full Adder



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

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

$$C_o = A B + B C_i + A C_i$$

a	b	c_i	c_{i+1}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

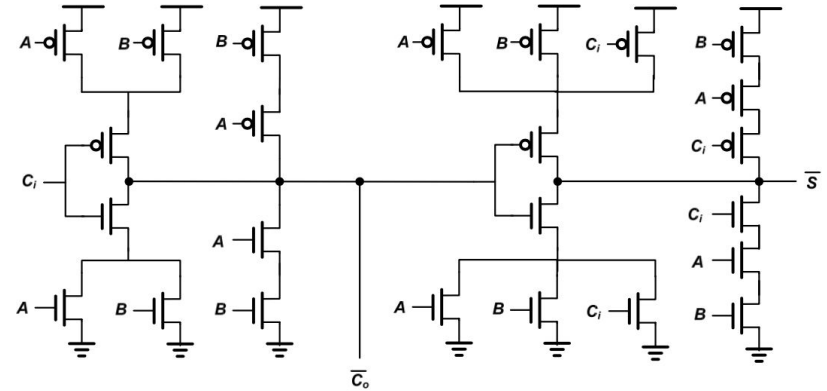
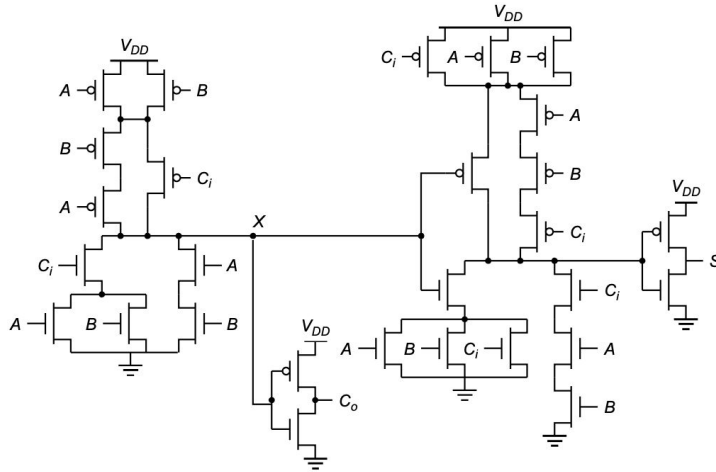
- A **full adder** implements a single-bit adder with carry in
- A **half adder** doesn't have a carry in, but still has a carry out
- The full adder is the primitive used in many adder topologies

Static CMOS Full Adder

Direct mapping of logic function

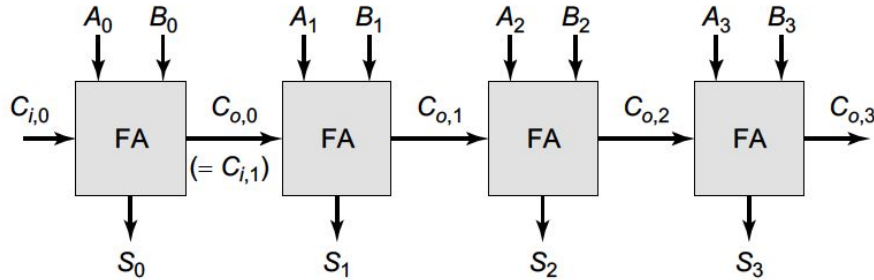


A better structure: The mirror adder



Ripple-carry Adder

For a 1-bit added, assume: $t_{\text{sum}} > t_{\text{carrier}}$

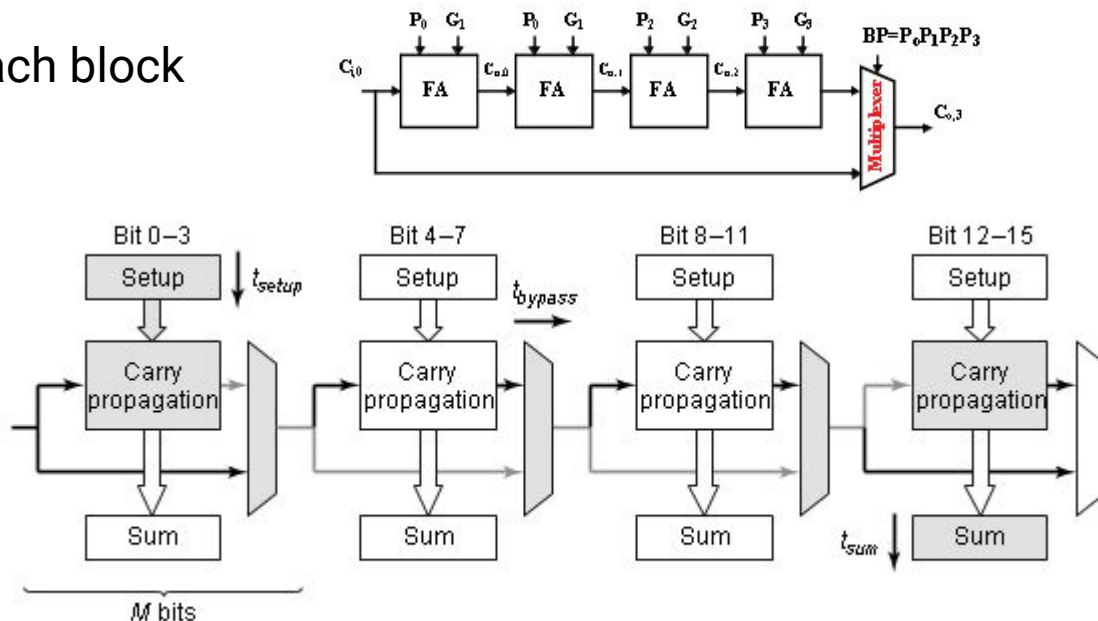


What's the critical path here?

Carry-bypass Adder

- N inputs/M bits in each block

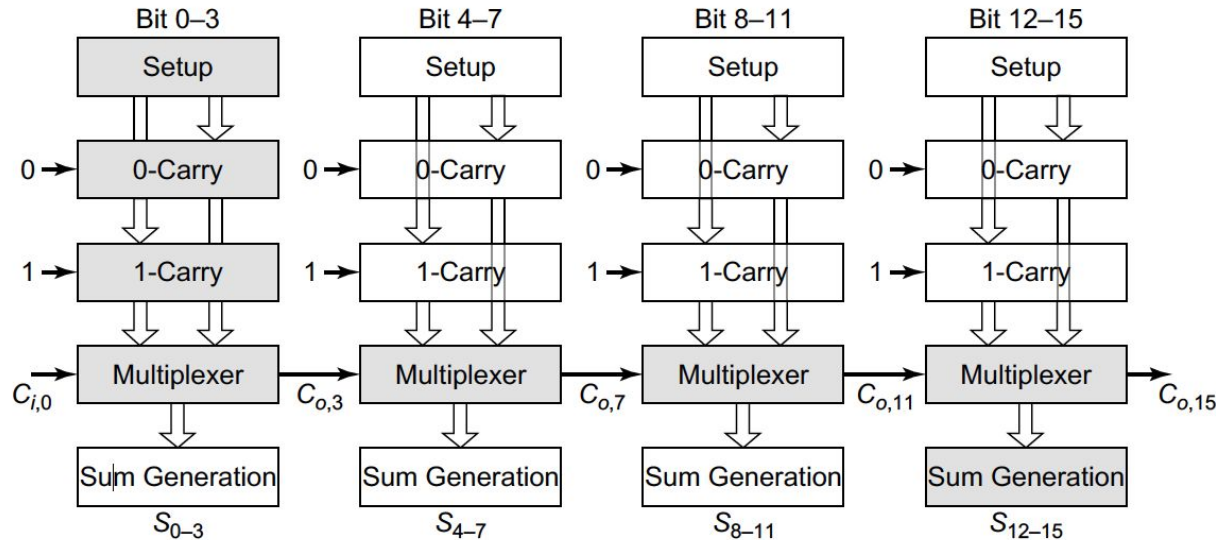
a	b	c_i	c_{i+1}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



- What's the critical path here?

Carry-select Adder (1/2)

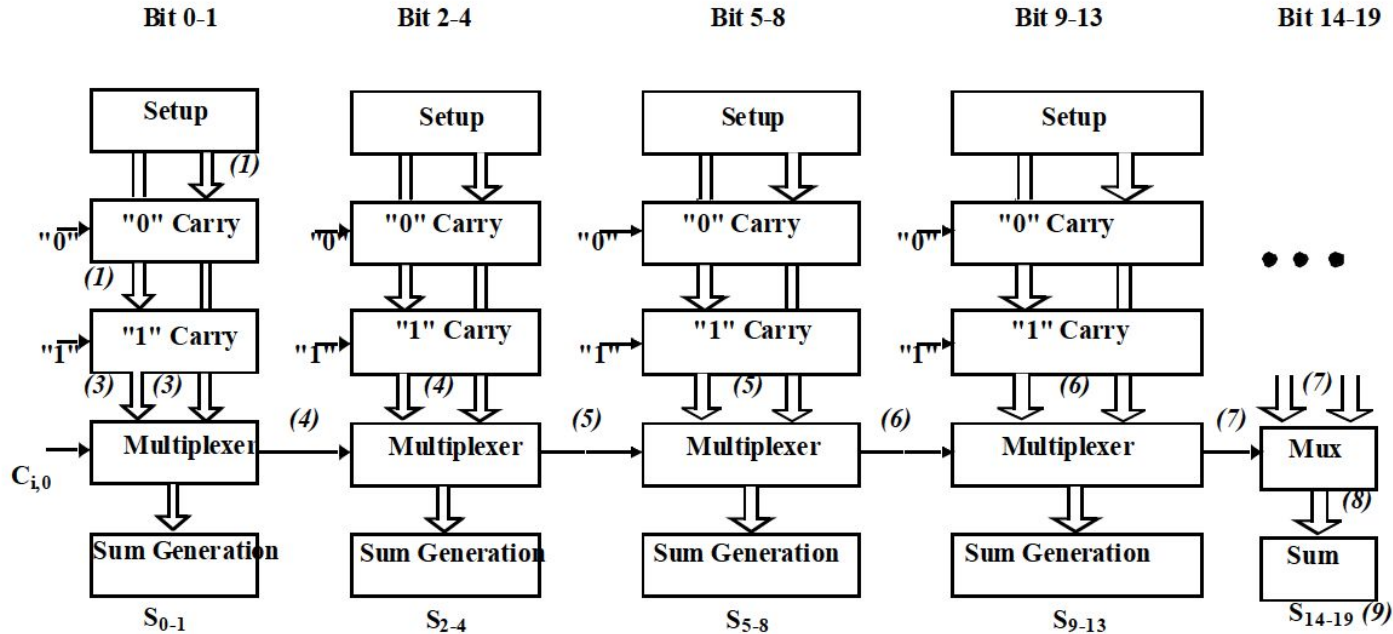
- N inputs/M bits in each block



- What's the critical path here?

Carry-select Adder (1/2)

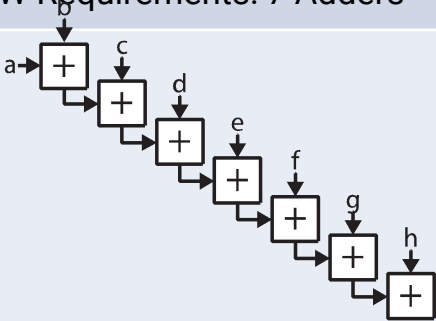
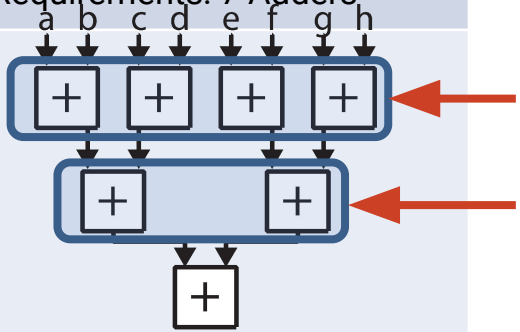
- N inputs/M bits in the first stage, P stages



- What's the critical path here?

Quick Aside: Associativity

- An operator, #, is associative iff: $(a \# b) \# c = a \# (b \# c)$
- Addition*, multiplication, AND, OR, XOR, are associative
- Allows for tree computation
- Ex. $a + b + c + d + e + f + g + h$

$(((((a+b) + c) + d) + e) + f) + g) + h$	$((a+b) + (c+d)) + ((e+f) + (g+h))$
Delay: 7 Additions HW Requirements: 7 Adders	Delay: 3 Additions HW Requirements: 7 Adders
	

Adders in
same layer
can be
computed
in parallel!

Carry-Lookahead Adder: Redefining FAs

- **Problem: carry logic not associative -> linear FA chain**
- Solution: re-define FAs to generate 2 new signals
 - **g** (Generate): True if adder is guaranteed to **generate a carry**
$$g_i = a_i \cdot b_i$$
 - **p** (Propagate): True if carry-out equals carry-in (**propagate carry-in**)
$$p_i = a_i \wedge b_i$$
- Both **g** & **p** have no dependence on carry-in (c_i)

Carry-Lookahead Adder: Redefining FAs

- Sum & carry-out of FA defined in terms of these new signals
 - Sum is true if:
 - A single input is true, carry-in is false
 - Inputs are both 0 or 1, carry-in is true

$$s_i = p_i \oplus c_i$$

- Carry-out is true if:
 - Carry generate is true
 - Propagate is true and carry-in is true

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

- However, sum and carry-out depend on carry-in

Carry-Lookahead Adder: Redefining FAs

- **Problem: carry logic not associative -> linear FA chain**
- Solution: re-define FAs to generate 2 new signals
 - **g** (Generate): True if adder is guaranteed to **generate a carry**
$$g_i = a_i \cdot b_i$$
 - **p** (Propagate): True if carry-out equals carry-in (**propagate carry-in**)
$$p_i = a_i \wedge b_i$$
- Both **g** & **p** have no dependence on carry-in (c_i)

Carry-Lookahead Adder: Redefining FAs

- Sum & carry-out of FA defined in terms of these new signals
 - Sum is true if:
 - A single input is true, carry-in is false
 - Inputs are both 0 or 1, carry-in is true

$$s_i = p_i \oplus c_i$$

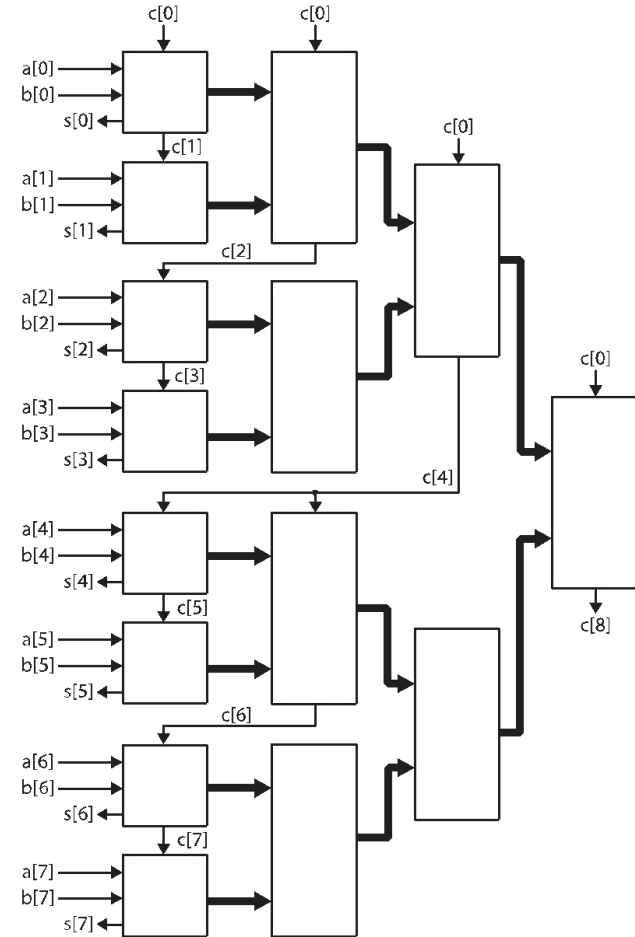
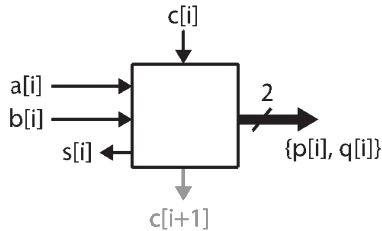
- Carry-out is true if:
 - Carry generate is true
 - Propagate is true and carry-in is true

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

- However, sum and carry-out depend on carry-in

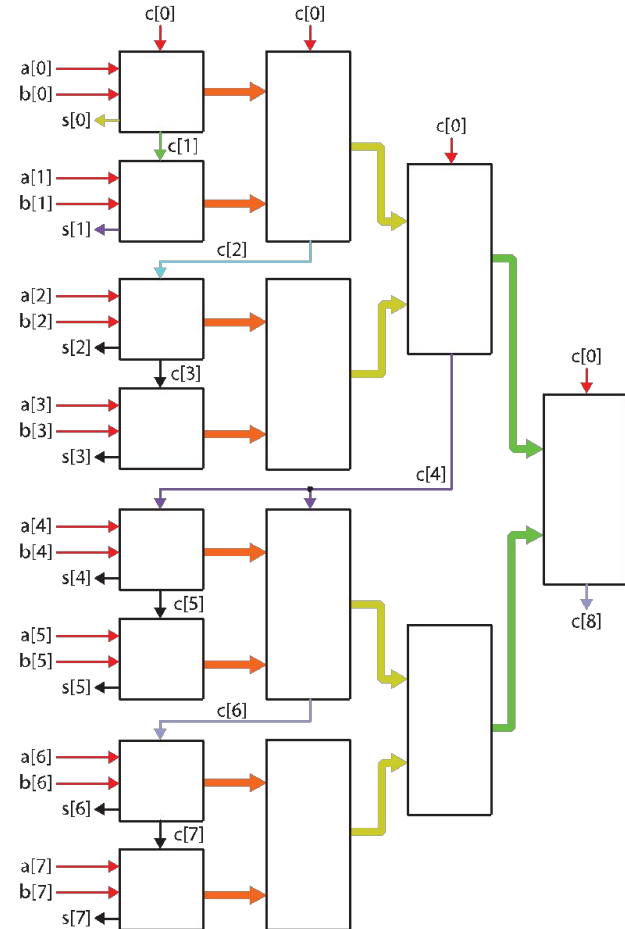
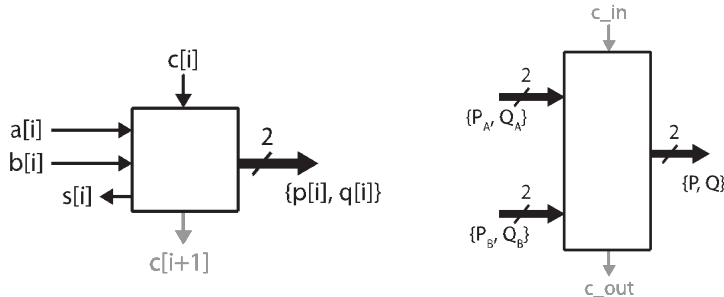
CLA: Tree Structure

- Smallest blocks are modified full adders
- Calculate g and p immediately
- Wait for carry-in to compute sum bit
- Some FAs are required to create carry-out



CLA: Grouping

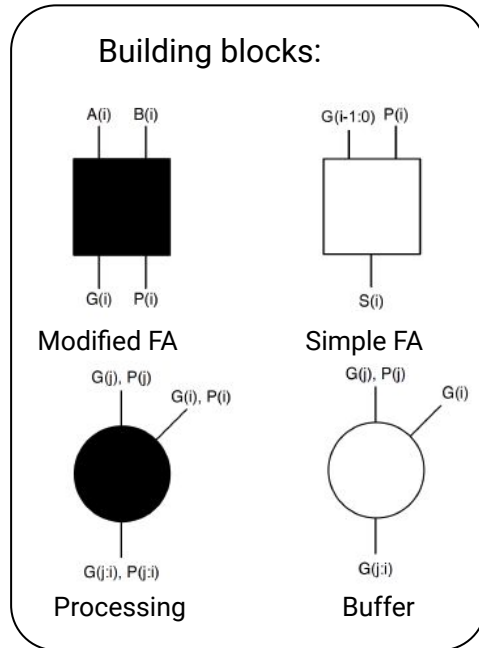
- Group together adders and create P & G for higher levels of the hierarchy.
 - P = entire group propagates a carry
 - G = entire group generates a carry
 - P & G can be computed without carry-in
- Carry-in required to generate carry-out
 - $P = P_A \cdot P_B$
 - $G = G_B + G_A \cdot P_B$
 - $C_{\{out\}} = G + C_{\{in\}} \cdot P$



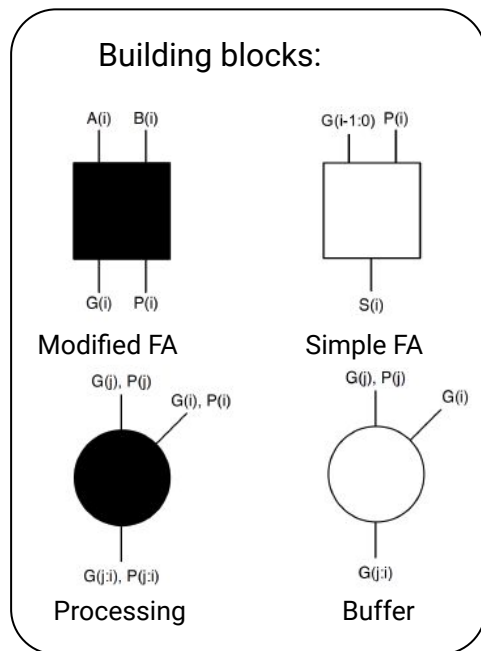
Parallel Prefix Adder

- Remaining problem: CLA as described still ripples carry through groups in first layer of tree
- Solution: unroll the expression for the carry bit
 - $c_0 = 0$ (unsigned)
 - $c_1 = g_0 + p_0 \cdot c_0 = g_0$
 - $c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 g_0$
 - $c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0$
 - $c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$
 - Recall p 's and g 's can be computed in parallel (not dependent on carry-in)
 - These operations are associative -> "prefix tree" (parallel) computation!
- Overall flow:
 - Break into group of bits -> Precalculate P_i, G_i in each group -> combine the groups in a tree structure -> calculate carries in parallel -> simple full adder to generate sum

Prefix Tree Adder Graphs: 3-bit Kogge-Stone adder



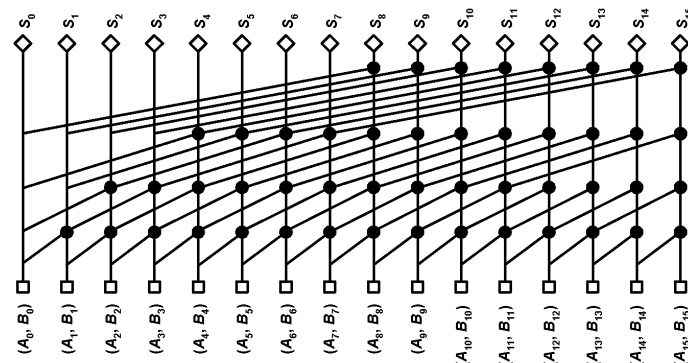
Prefix Tree Adder Graphs



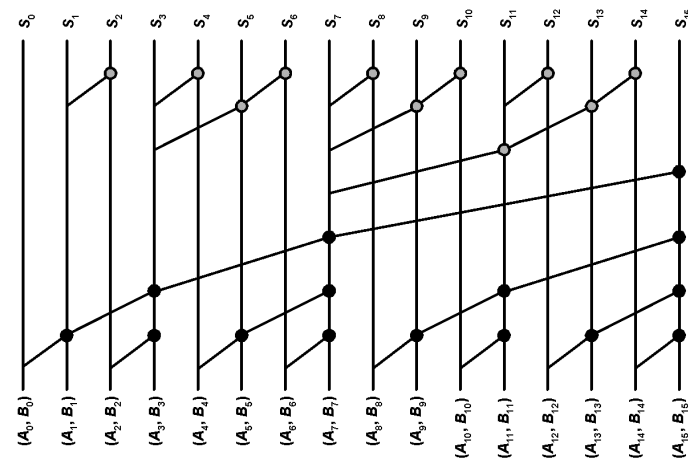
Min. critical path & fanout
Cost: more logic resources

Tradeoff logic reuse w/
critical path

Most reuse (min. area)
Cost: longest critical path



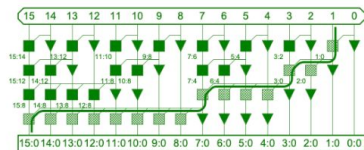
Kogge-Stone



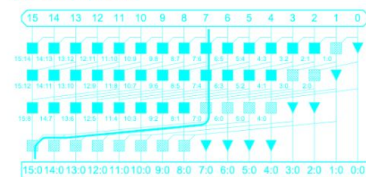
Brent-Kung

“A Taxonomy of Parallel Prefix Networks”, Harris [2003]

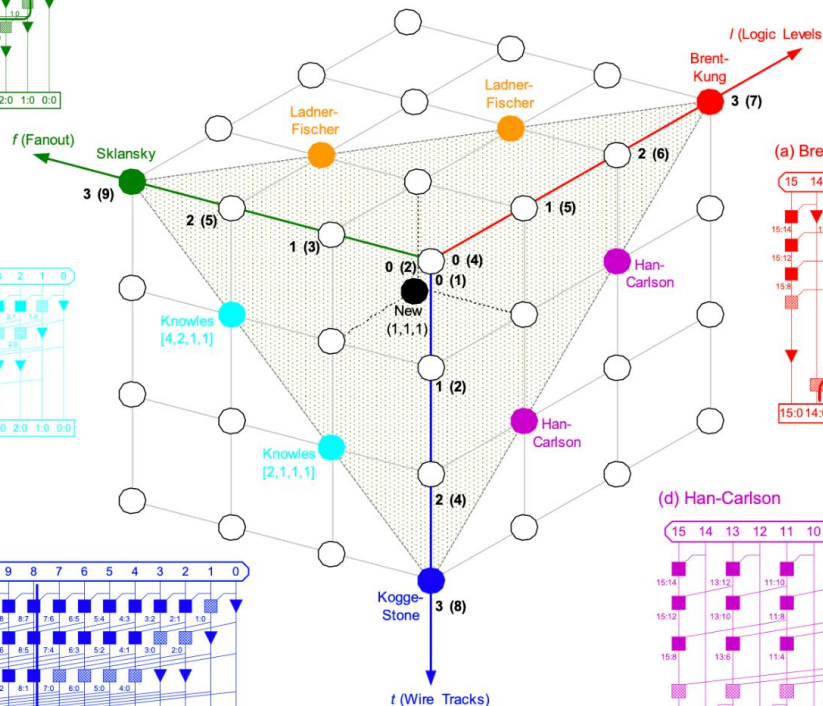
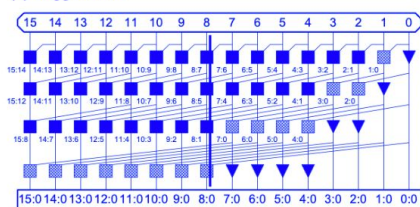
(b) Sklansky



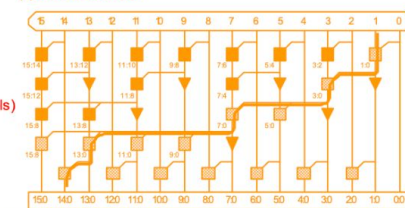
(e) Knowles [2, 1, 1]



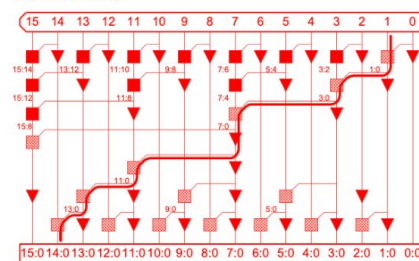
(c) Kogge-Stone



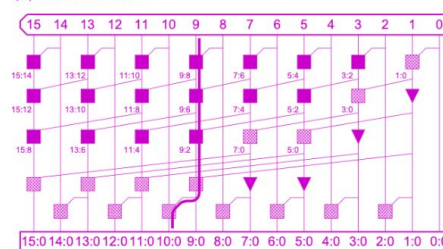
(f) Ladner-Fischer



(a) Brent-Kung



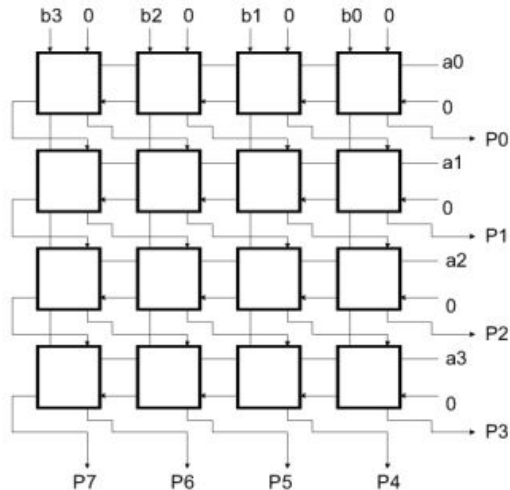
(d) Han-Carlson



Multipliers

Unsigned Multiplication Example

- Partial Products can be generated in parallel
- Challenge: improve the addition of partial products



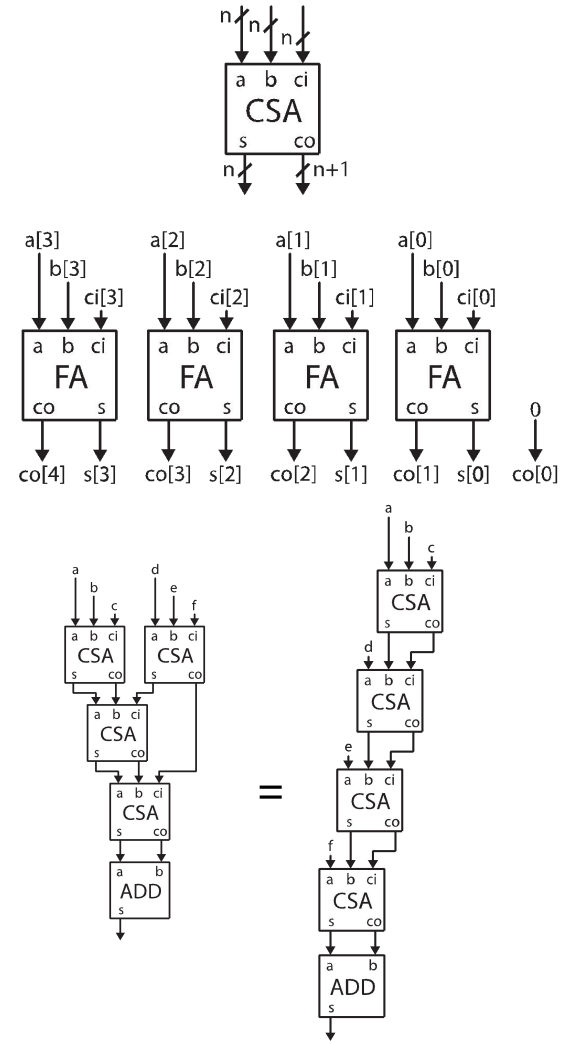
$$\begin{array}{r} 4'b0011 \quad (3) \\ * 4'b0110 \quad (6) \\ \hline \quad 0000 \\ \quad 0011 \\ \quad 0011 \\ + 0000 \\ \hline 00010010 \quad (18) \end{array}$$

Partial Products

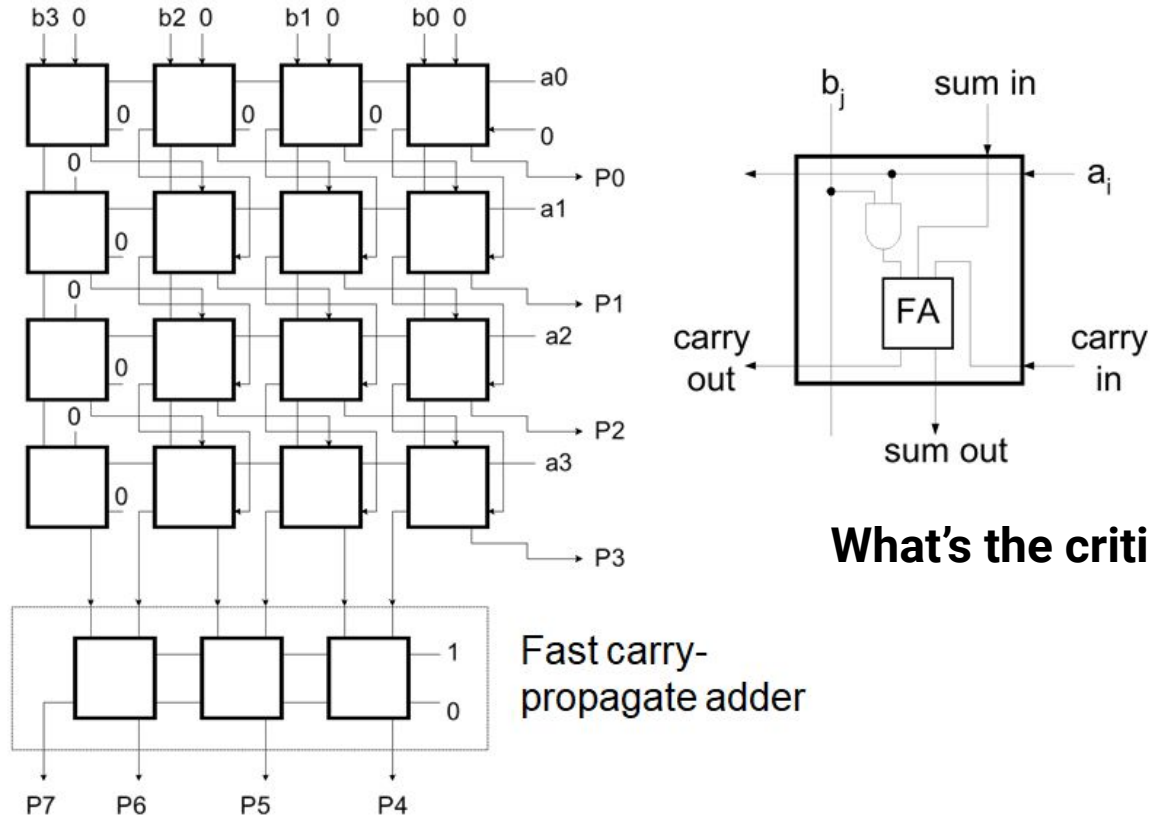
What's the critical path here?

Carry-Save Addition

- When we generate a carry in a given column, add it to the 2 values in the next column.
 - $S_i = A_i \oplus B_i \oplus C_i$
 - $C_{\{0, i+1\}} = A_i B_i + A_i C_i + B_i C_i$
 - Delay adding carry bits until the end
- Basis of CSA:
 - Takes in a, b, c_{in} (multi-bit)
 - Produces a sum and c_{out} (multi-bit)
- Benefits:
 - CSAs have no carry ripple => small & fast!
 - Only 1 standard CLA/PPA at end
 - Addition is associative => trees!



Array Multiplier w/ CSA



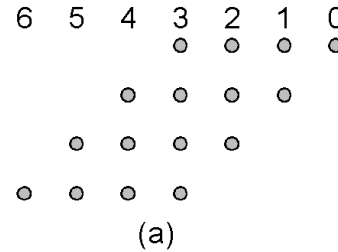
What's the critical path here?

Wallace Tree Multiplier

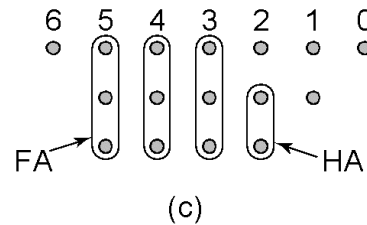
Method to construct Wallace Tree:

1. Draw a dot diagram where each column has as many dots as number of partial products
2. Group dots in the same column by 2 (half adder) or 3 (full adder)
3. Propagate carries and sum by adding one dot in the grouped column and one dot in the next column

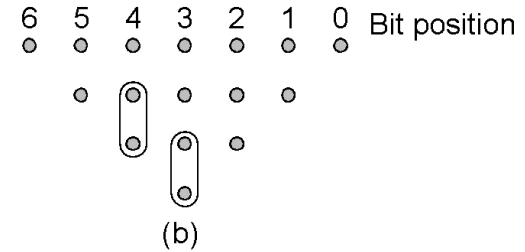
Partial products



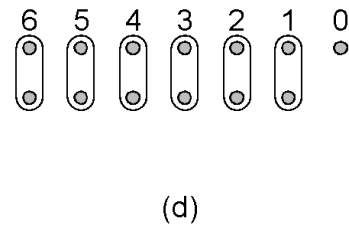
Second stage



First stage



Final adder



Radix and Multiplication

- Binary multiplication \rightarrow N partial products! Can we reduce this?
 - Yes! Let's use a larger radix (think: base)
- E.g. 2 bits at a time (radix 4) \rightarrow halve number of partial products

B Digit	Partial Product	Partial Product (Rewritten)
0	$0 \cdot A$	0
1	$1 \cdot A$	A
2	$2 \cdot A$	$4 \cdot A - 2 \cdot A$
3	$3 \cdot A$	$4 \cdot A - A$

- Recall: Multiplications by powers of 2 are left shifts
 - Let's use this property!

Booth Recoding

- $4*A = A \ll 2$
- $2*A = A \ll 1$
- Recall: radix 4 multiplication \Rightarrow shift left by 2 positions for next partial product
- Therefore, any $4*A$ term can be handled in the next partial product!
 - Multiplier looks a 3 (rather than just 2) bits
 - Extra bit is MSB of the previous

B Digit	Partial Product	Partial Product (Rewritten)
0	$0*A$	0
1	$1*A$	A
2	$2*A$	$4*A - 2*A$
3	$3*A$	$4*A - A$

Booth Recoding

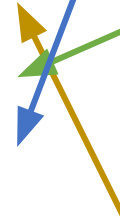
B_{i+1}	B_i	B_{i-1}	Action	Comment
0	0	0	Add 0	
0	0	1	Add A	Includes $+4*A$ from previous radix 4 digit = $+A$ in this position due to left shift by 2
0	1	0	Add A	
0	1	1	Add $2*A$	Includes $+4*A$ from previous round ($+A$ in this position). $*2$ is implemented as a left shift by 1
1	0	0	Sub $2*A$	$4*A$ will be added in when handling next radix 4 digit. $*2$ is implemented as a left shift by 1
1	0	1	Sub A	$4*A$ will be added in when handling next radix 4 digit. Includes $+4*A$ from previous radix 4 digit ($+A$ in this position)
1	1	0	Sub A	$4*A$ will be added in when handling next radix 4 digit.
1	1	1	Add 0	$4*A$ will be added in when handling next radix 4 digit. Includes $+4*A$ from previous radix 4 digit ($+A$ in this position)

Booth Recoding Example (Unsigned)

- $6 * 7$
- $B_{-1} = 0$

$$\begin{array}{r}
 4'b0110 \quad (6) \\
 * 4'b0111 \quad (7) \\
 \hline
 - \quad \quad 0110 \quad (\text{Sub } A) \\
 + \quad 01100 \quad (\text{Add } 2A) \\
 + \quad 0000 \quad (\text{Add } 0) \\
 \hline
 + \quad 1111010 \quad (\text{Sub } A) \\
 + \quad 01100 \quad (\text{Add } 2A) \\
 + \quad 0000 \quad (\text{Add } 0) \\
 \hline
 \end{array}$$

B_{i+1}	B_i	B_{i-1}	Action
0	0	0	Add 0
0	0	1	Add A
0	1	0	Add A
0	1	1	Add 2*A
1	0	0	Sub 2*A
1	0	1	Sub A
1	1	0	Sub A
1	1	1	Add 0



Signed Multiplication: Baugh-Wooley

- Recall: 2's complement MSB has negative weight
- Nominally:
 1. Subtract last partial product
 2. Sign-extend the rest of the partial products
- Recall 2's complement negation: $-A = \sim A + 1$
 - Add $\sim A + 1$ instead! Result: basically same hardware as unsigned mult.
 - Implementation: invert some bits, insert a 1 left of the first & last partial products

				1	p0[3]	p0[2]	p0[1]	p0[0]
+				$\sim p1[3]$	p1[2]	p1[1]	p1[0]	0
+			$\sim p2[3]$	p2[2]	p2[1]	p2[0]	0	0
+	1	$\sim p3[3]$	$\sim p3[2]$	$\sim p3[1]$	$\sim p3[0]$	0	0	0

P[7]	P[6]	P[5]	P[4]	P[3]	P[2]	P[1]	P[0]	