EECS 151/251A Discussion 9

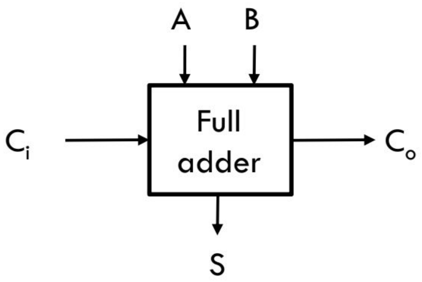
Zhaokai Liu

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



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

$$S = A\,\overline{B}\,\overline{C_i} + \overline{A}\,B\,\overline{C_i} + \overline{A}\,\overline{B}\,C_i + A\,B\,C_i$$

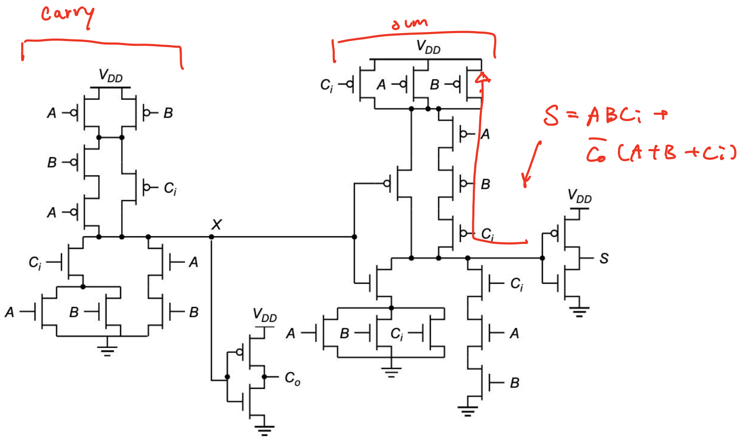
$$C_o = A\,B + B\,C_i + A\,C_i$$

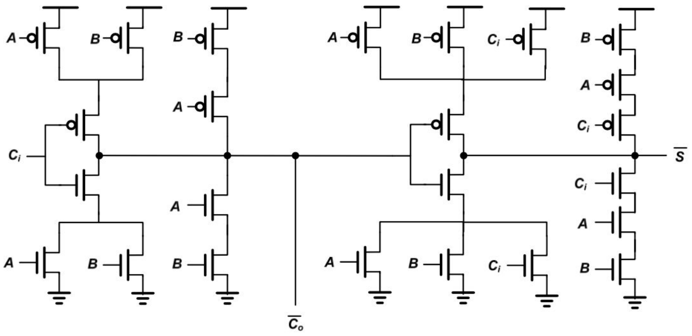
$a\ b\ c_i$	c_{i+1}	s
000	0	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	1	1

- 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

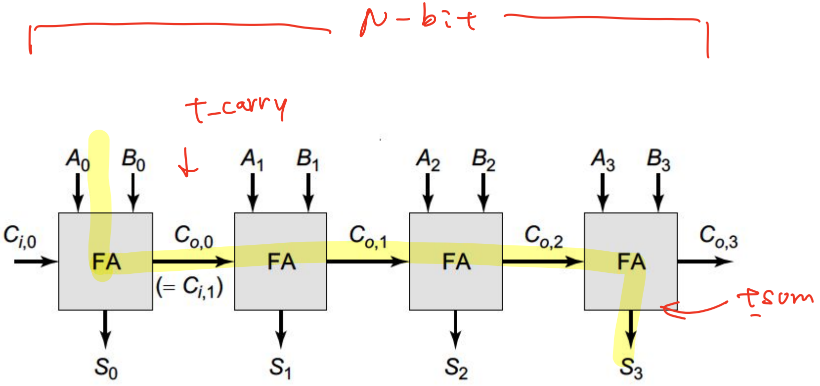
Direct mapping of logic function → A better structure: The mirror adder





Ripple-carry Adder

For a 1-bit added, assume: t_sum > t_carrier



What's the critical path here?

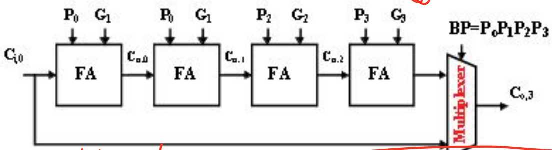
$(N-1) \cdot t\_carry + t\_sum$

Carry-bypass Adder

- N inputs/M bits in each block

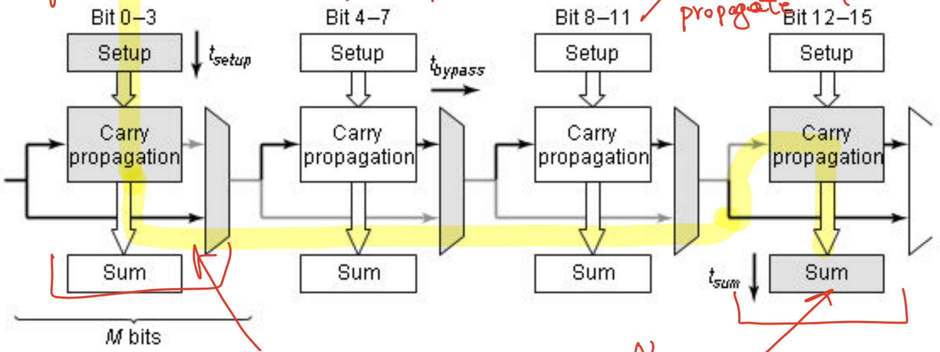
a b c_i	c_{i+1}	s
000	0	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	1	1

generate or kill in this stage calc. by Pi Gi ...



N-bit

propagate



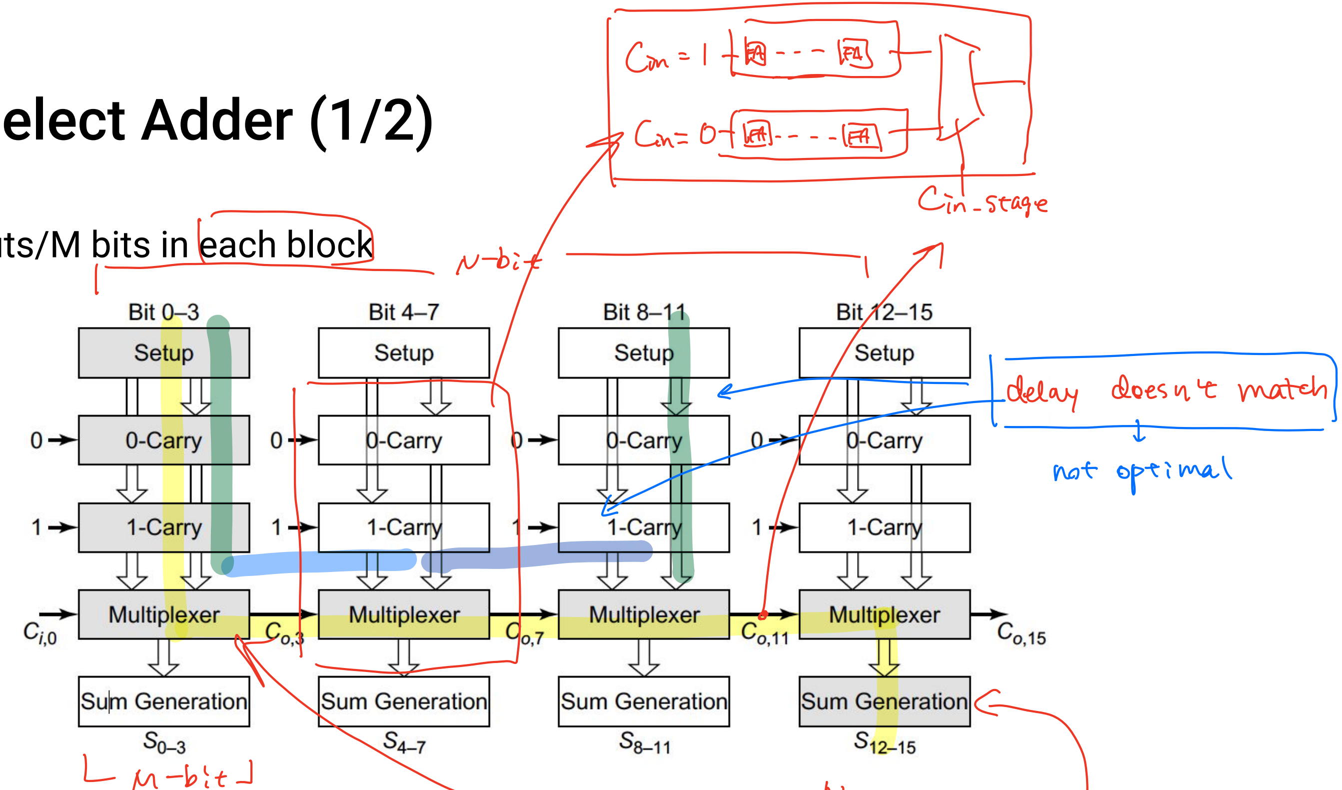
- **What's the critical path here?**

$t_{setup} + M \cdot t\_carry + (\frac{N}{M} - 1) \cdot t_{mux} +$

$(M-1)\, t\_carry + t_{sum}$

Carry-select Adder (1/2)

- N inputs/M bits in each block



- **What's the critical path here?** $t\_setup + M\,t_{carry} + \left(\frac{N}{M} - 1\right) t_{mux} + t\_sum$

Carry-select Adder (1/2)

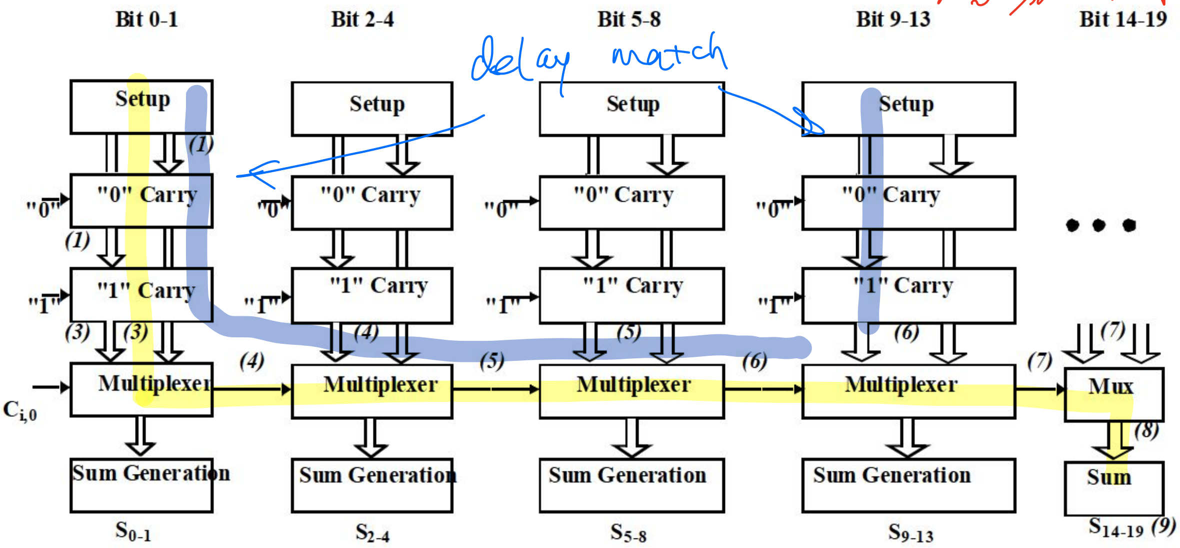
- N inputs/M bits in the first stage, P stages

idx	1	2	3	...	P
# bits	M	M+1			M+P-1

$N = M + (M+1) + \cdots + (M+P-1)$

when M is small N is large

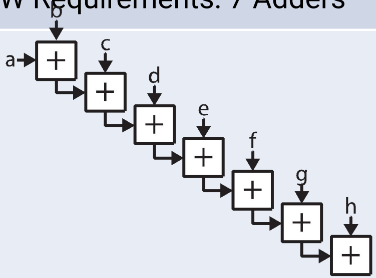
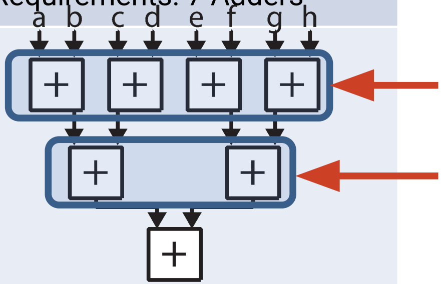
$N \approx P^2/2 \Rightarrow P = \sqrt{2N}$



- **What's the critical path here?** $t_{setup} + M\,t_{carry} + P\,t_{mux} + t_{sum}$
  (M=2 here)

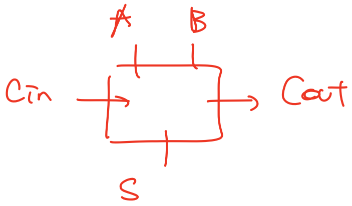
Quick Aside: Associativity

- 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



$G = A \cdot B$

$P = A \wedge B$

- **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 ^ 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 · 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

$$s_i = p_i \wedge c_i$$

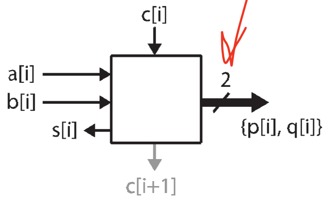
    - Carry-out is true if:
    - Carry generate is true
    - Propagate is true and carry-in is true

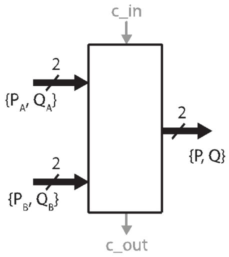
$$c_{i+1} = g_i + p_i \cdot c_i$$

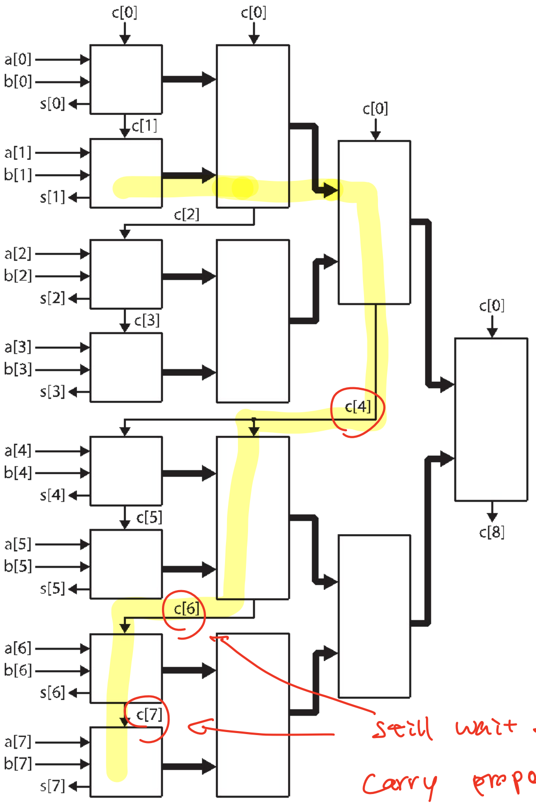
- However, sum and carry-out depend on carry-in

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





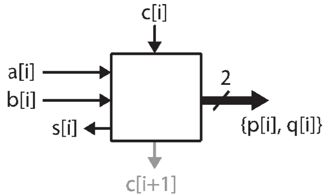


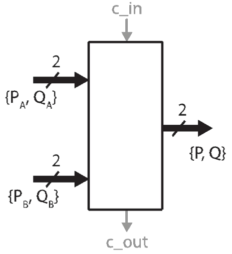
still wait for carry propagate

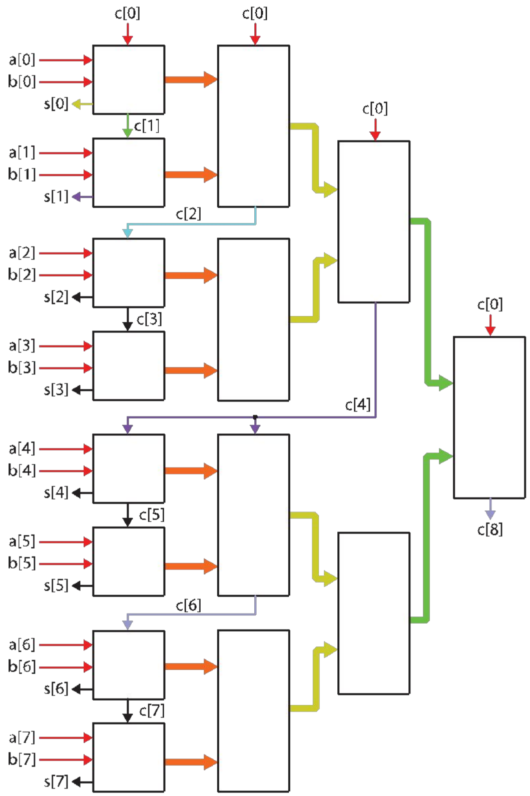
CLA: Grouping

- 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•P_B
  - G = G_B + G_A•P_B
  - C_{out} = G + C_{in}•P

group P_i and G_i







Parallel Prefix Adder

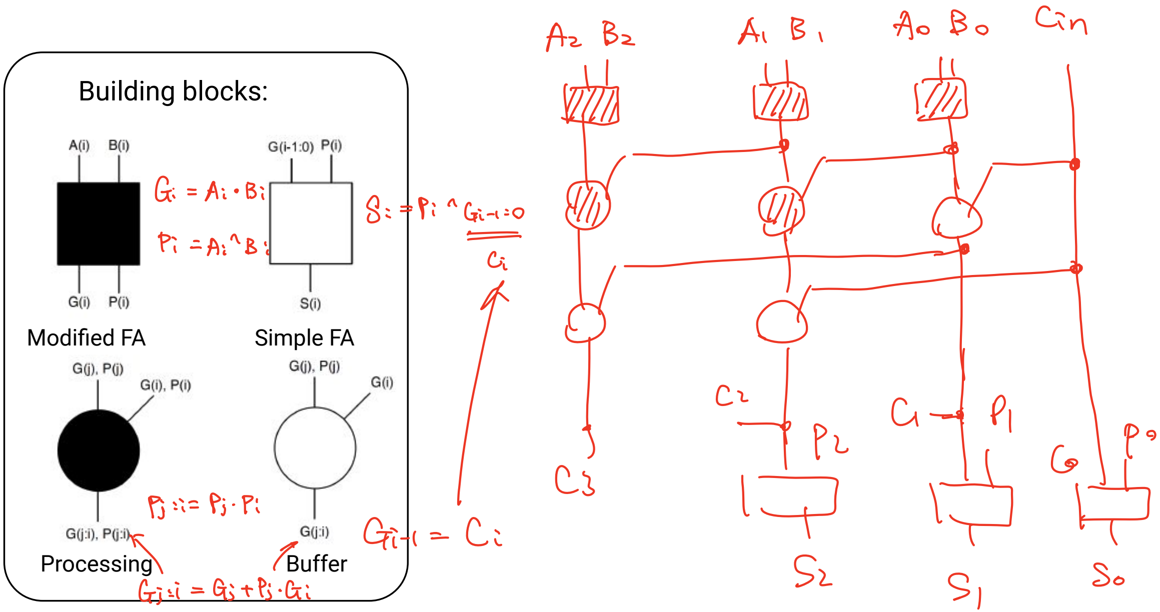
- Remaining problem: CLA as described still ripples carry through groups in first layer of tree
- Solution: unroll the expression for the carry bit
  - $c_0 = 0$ (unsigned) $C_0 = 1$
  - $c_1 = g_0 + p_0 \cdot c_0 = g_0$ $C_1 = g_0 + P_0$
  - $c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 g_0$ $C_2 = g_1 + P_1(g_0 + P_0) = g_1 + g_0 P_1 + P_0$
  - $c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0$ $\vdots$
  - $c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$
  - Recall p's and g's can be computed in parallel (not dependent on carry-in)
  - These operations are associative -> "prefix tree" (parallel) computation!

  $G_{3:1}$ $P_{3:1}$

- 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

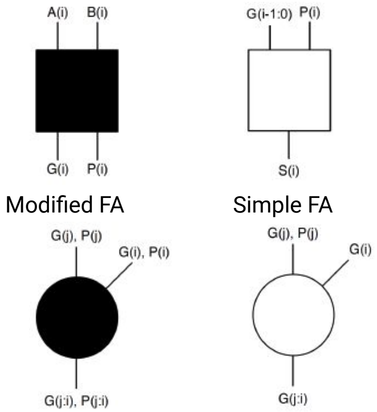
generate P_i and g_i $\longrightarrow$ group of $G_{j:i}$ $P_{j:i}$ $\longrightarrow$ calc. S_i and C_i

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



Prefix Tree Adder Graphs

Building blocks:



Modified FA

Simple FA

Processing

Buffer

~ $\log_2 N$

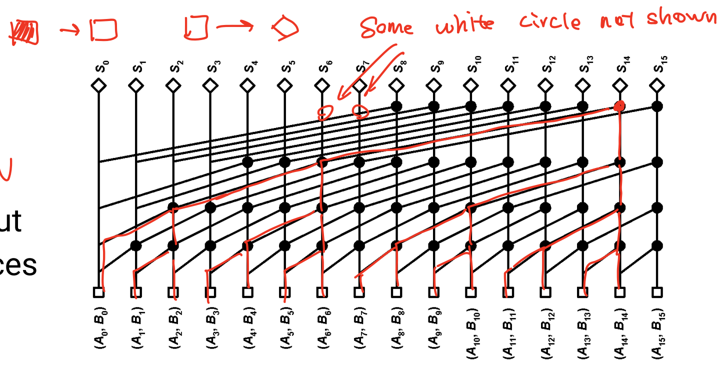
Min. critical path & fanout

Cost: more logic resources

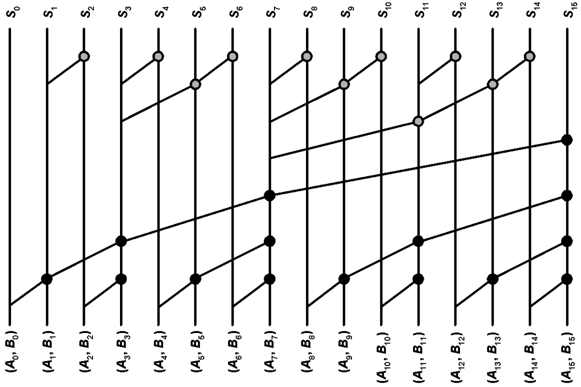
Tradeoff logic reuse w/ critical path

Most reuse (min. area)

Cost: longest critical path

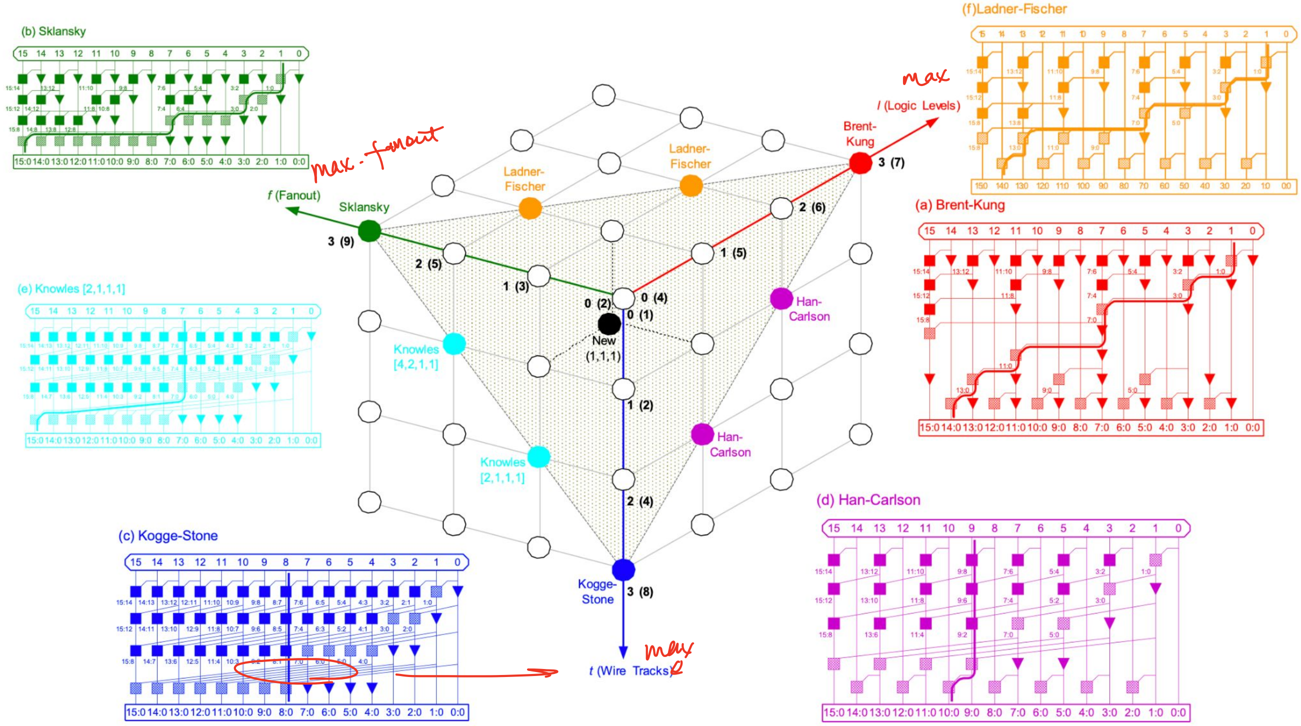


Kogge-Stone



Brent-Kung

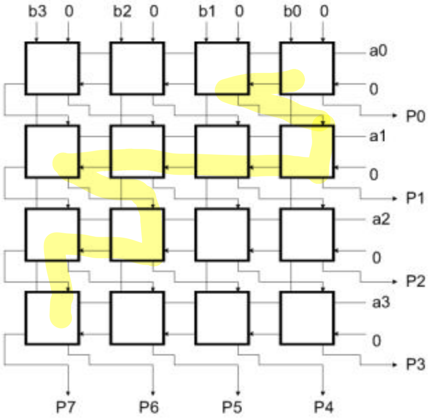
“A Taxonomy of Parallel Prefix Networks”, Harris [2003]



Multipliers

Unsigned Multiplication Example

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




```
  4'b0011 (3)
* 4'b0110 (6)
  -------
     0000
    0011
   0011
+ 0000
---------
 00010010 (18)
```

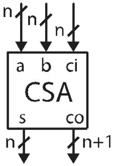
Partial Products

What's the critical path here?

Carry-Save Addition

C_i	0 0 0 0
a	0 1 1 0
b	0 1 0 1
C_o	1 0 0 0
S	0 0 1 1

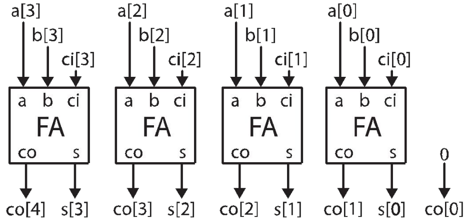
$a+b=C_o+S$



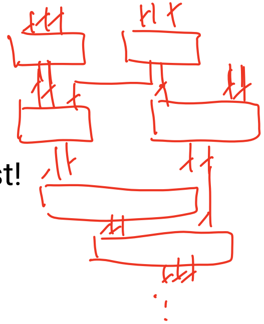
- When we generate a carry in a given column, add it to the 2 values in the next column.
    - S_i = A_i ^ B_i ^ C_i
    - C_{o, i+1} = A_i B_i + A_i C_i + B_i C_i
    - Delay adding carry bits until the end

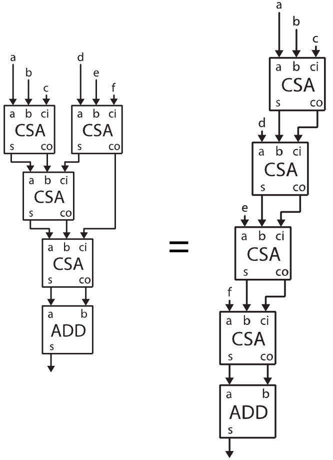
3 inputs

2 outputs

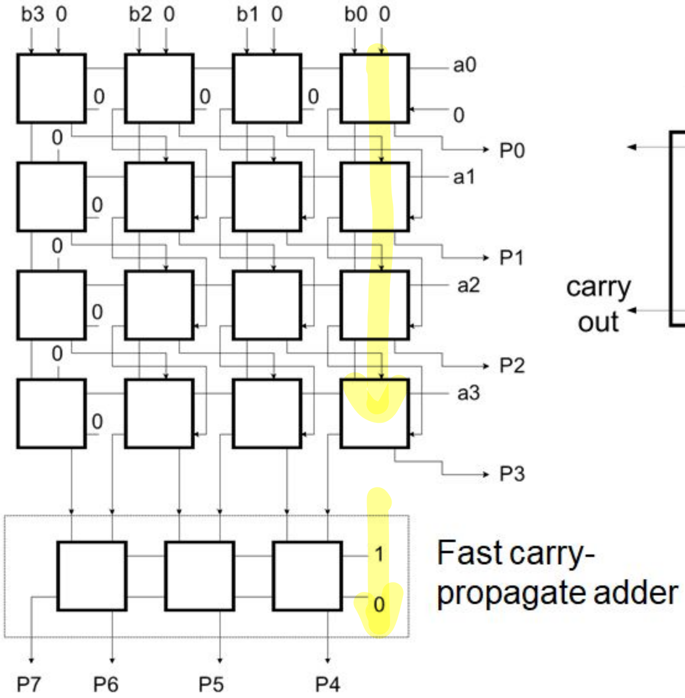


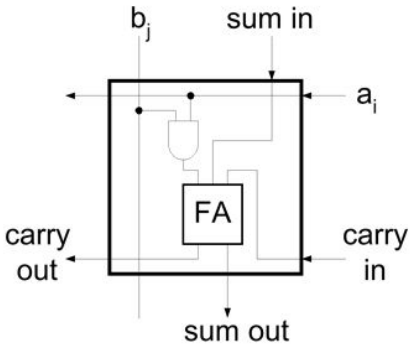
- Basis of CSA:
    - Takes in a, b, c_{in} (multi-bit)
    - Produces a sum and c_{out} (multi-bit)
- Benefits:
    - CSAs have no carry ripple => small & fast!
    - Only 1 standard CLA/PPA at end
    - Addition is associative => trees!





Array Multiplier w/ CSA





What's the critical path here?

$\sim \log_2 N$

Wallace Tree Multiplier

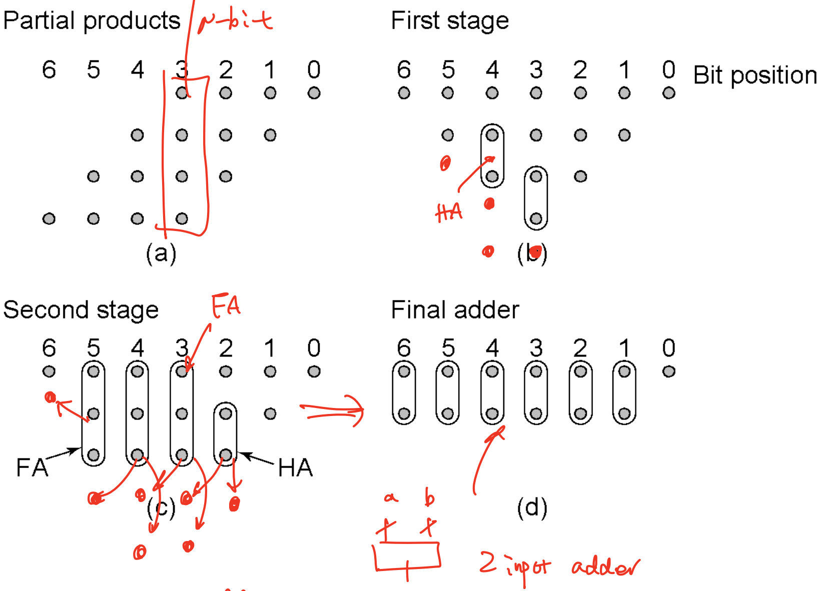
3:2 compression

Nbit → 2 bit

$N \cdot \left(\frac{2}{3}\right)^x = 2 \Rightarrow x \sim \log_{3/2}\left(\frac{N}{2}\right)$

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



tree adder

total delay $\sim \log_{(3/2)} N/2 + \log_2 N$

Radix and Multiplication

- Binary multiplication -> 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) -> halve number of partial products

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

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

Booth Recoding

- 4*A = A << 2
- 2*A = A << 1
- Recall: radix 4 multiplication => 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

carry in from last 2bit

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) $2+1=3=4-1 \Rightarrow$ sub A
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) $2+1+1=4 \Rightarrow$ left to next 2-bit

weight 2 1 1

Booth Recoding Example (Unsigned)

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

```
     4’b0110   (6)
   * 4’b0111   (7)
     --------
-      0110 ( Sub A)
+   01100   (Add 2A)
+  0000     ( Add 0)
   ---------
+  11111010 ( Sub A)
+   01100   (Add 2A)
+  0000     ( Add 0)
   ---------
```

~A+1

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 = ~A + 1
  - Add ~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]
+                      ~p1[3]   p1[2]  p1[1]  p1[0]    0
+               ~p2[3]  p2[2]   p2[1]  p2[0]    0      0
+   1   ~p3[3]  ~p3[2] ~p3[1]  ~p3[0]    0      0      0
----------------------------------------------------------
P[7]    P[6]    P[5]    P[4]    P[3]   P[2]   P[1]   P[0]
```