EECS 427
Lecture 8: Adders
Readings: 11.1-11.3.3

Reminders

- HW3 – project initial proposal: due Wednesday 10/7
  - You can schedule a half-hour appointment with me to discuss your initial proposal before submission
  - Email your initial proposal (one per group) in doc format to zhengya@eecs.umich by 7 pm on Wednesday night
  - Limit your write-up to 4 pages maximum. Keep it brief and clear.
- Quiz 1 on Wednesday 10/14
  - Samples (with solutions) from past terms are posted
  - Half-lecture review next Monday
- CAD4 is due Wednesday 10/14
  - You can submit it by Thursday 10/15 at noon

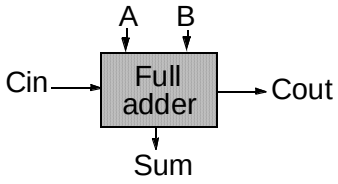
Last Time

- Dynamic circuits – high performance
  - Dynamic logic – non-ratioed, dynamic power only, no static current, higher activity, low noise margin
  - Domino logic – can be safely cascaded, only non-inverting logic
  - Footless domino – ripple precharge, delayed clock, extra power
  - Dual-rail domino – both inverting and non-inverting outputs
  - NP CMOS – cascade n- and p- evaluation networks

Overview

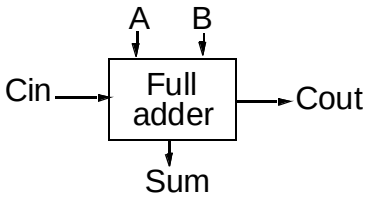
- Full adder
  - Mirror adder
  - Transmission-gate adder
  - Manchester carry chain
- Adder topologies
  - Ripple carry
  - Carry bypass
  - Carry select
  - Carry lookahead
- Carry lookahead adder
  - Kogge-Stone
  - Radix 2 or Radix 4
  - Sparse tree, Brent-Kung

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



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

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

$$C_o = AB + BC_i + AC_i$$

Express Sum and Carry as a function of P, G, D

Define 3 new variables which ONLY depend on A, B WHY?

Generate (G) = AB

Propagate (P) = A ⊕ B

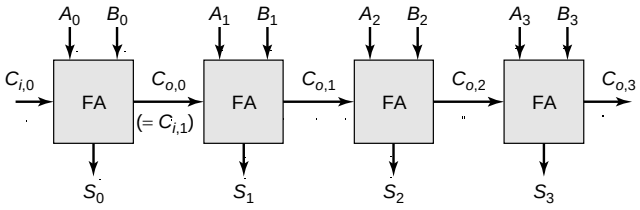
Delete = $\overline{A}\,\overline{B}$

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

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

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

The Ripple-Carry Adder



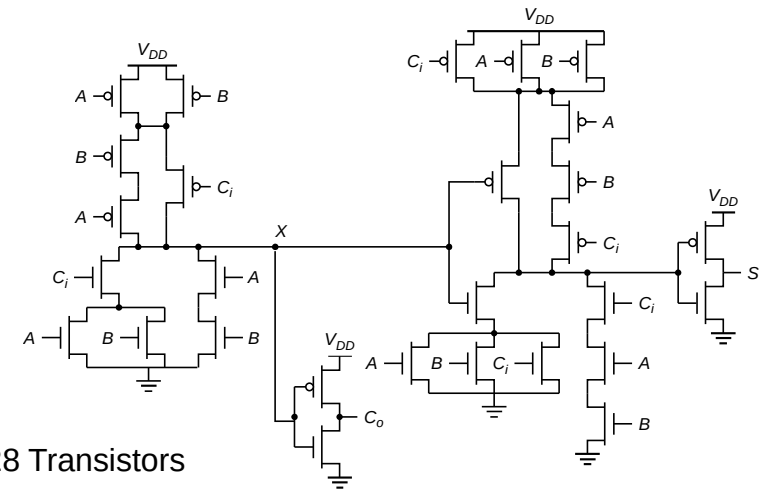
Worst case delay linear with the number of bits

$t_d = O(N)$

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

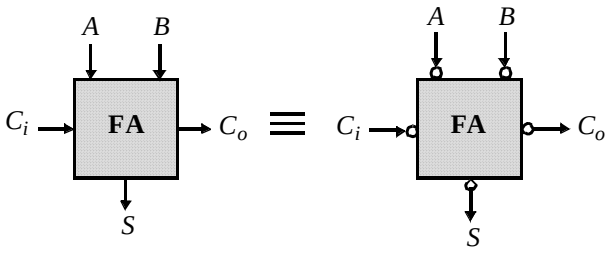
Goal: Make the fastest possible carry path circuit

Complementary Static CMOS Full Adder



28 Transistors

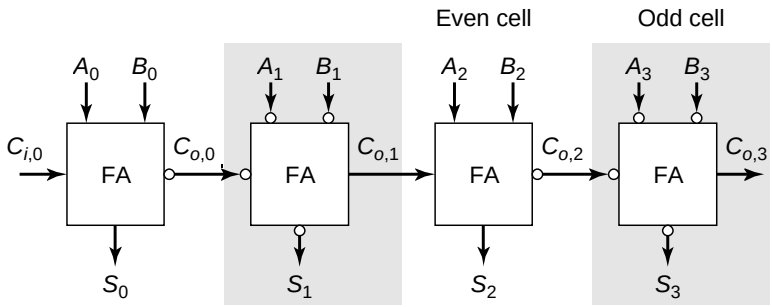
Inversion Property



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

$$\overline{C_o}(A,B,C_i) = C_o(\bar{A},\bar{B},\bar{C_i})$$

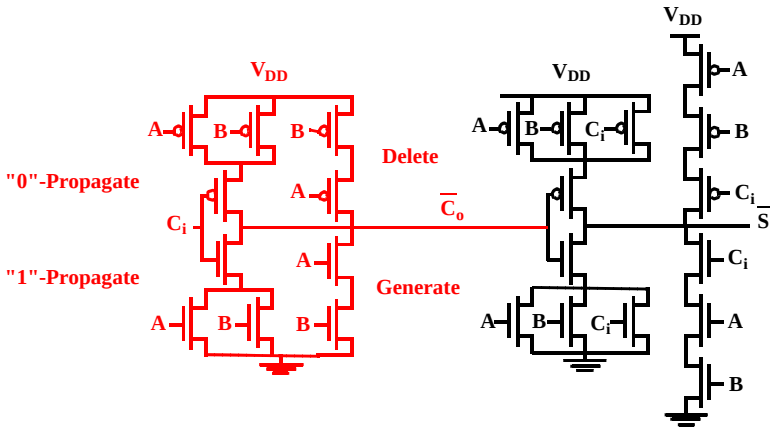
Minimize Critical Path by Reducing Inverting Stages Along Carry Path



Exploit Inversion Property

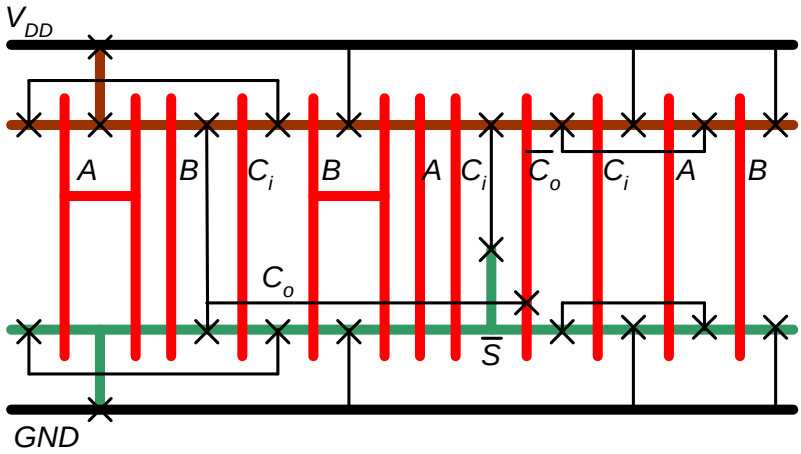
Allows us to remove inverter in carry chain → at what cost?

A Better Structure: Mirror Adder



24 transistors

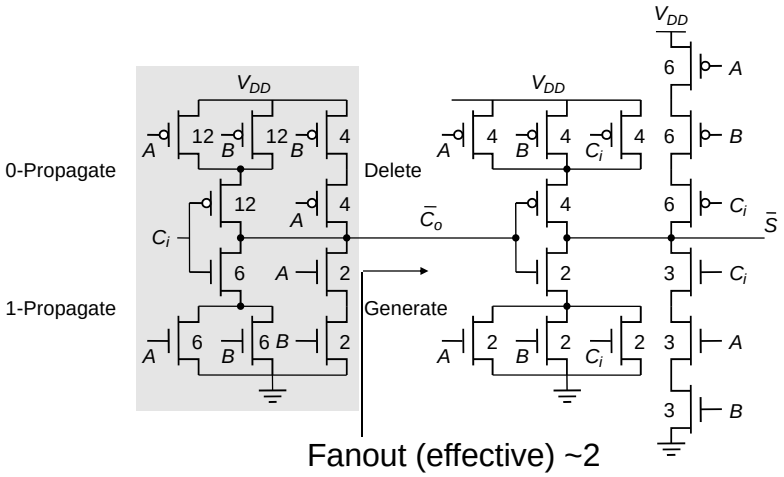
Mirror Adder Layout



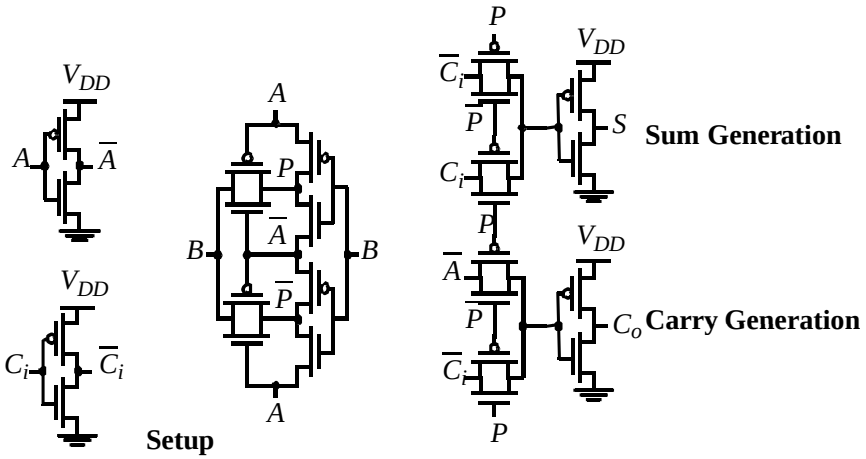
Mirror Adder Details

- NMOS and PMOS chains are completely symmetric. Maximum of 2 series transistors in carry generation
- In layout → critical to minimize capacitance at node C_o. Reduction of junction capacitances is particularly important
- Capacitance at node C_o is composed of 4 junction capacitances, 2 internal gate capacitances, and 6 gate capacitances in connecting adder cell
- Transistors connected to C_i are closest to output
- Only optimize transistors in carry stage for speed Transistors in sum stage can be small

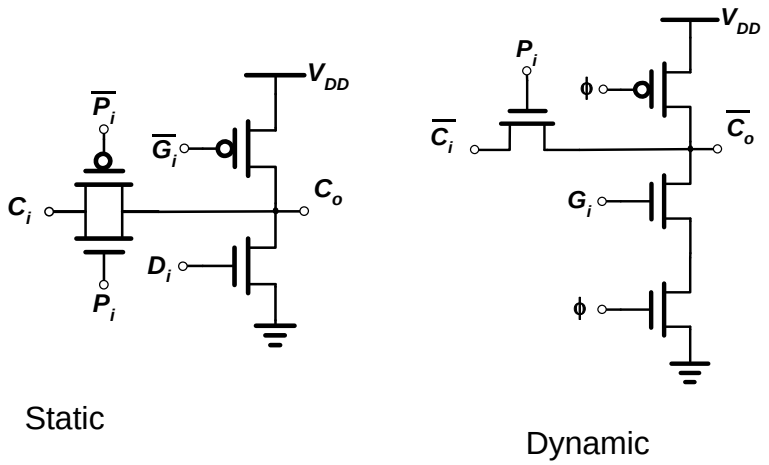
Sizing Mirror Adder



Transmission Gate Full Adder



Manchester Carry Chain

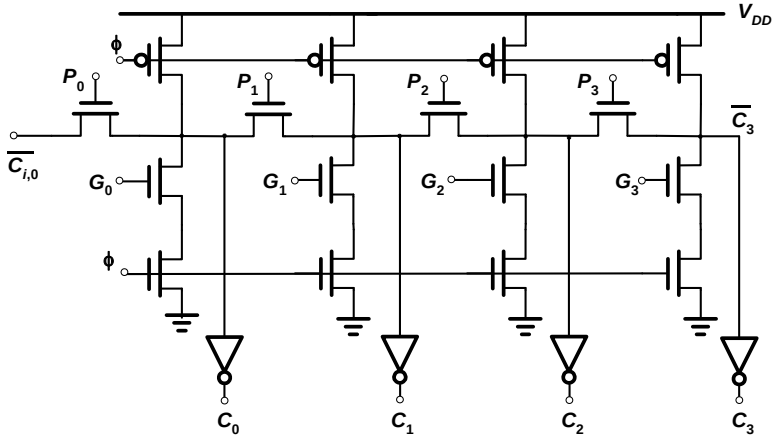


Static

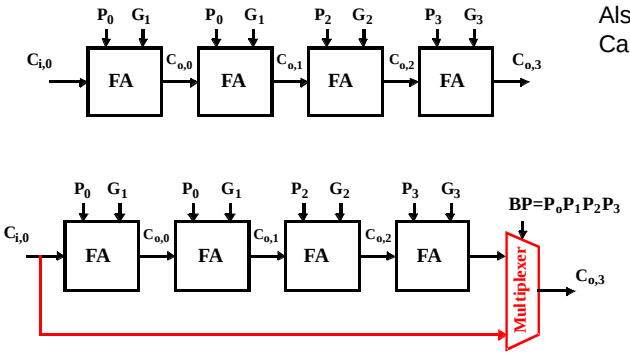
Dynamic

Manchester Carry Chain

- Implement P with pass-transistors
- Implement G with pull-up OR kill (delete) with pull-down (note inversion)
- Use dynamic logic to reduce complexity and speed up



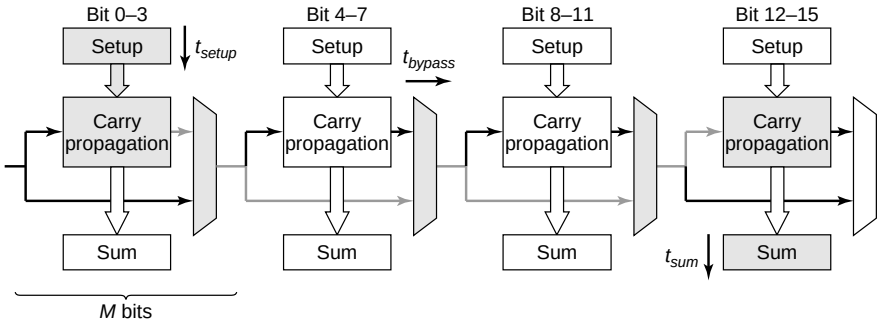
Carry-Bypass Adder



Also called Carry-Skip

Idea: If (P0 and P1 and P2 and P3 = 1) then $C_{o3} = C_{i,0}$, else "delete" or "generate"

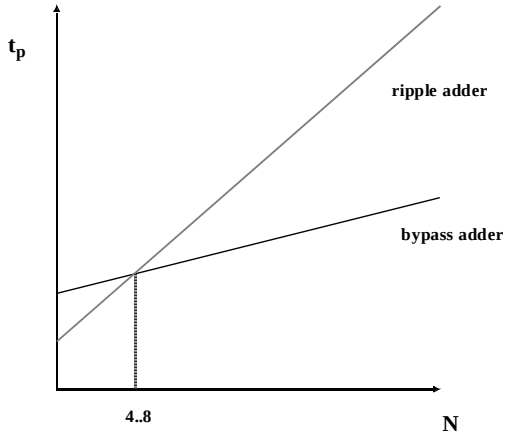
Carry-Bypass Adder (cont.)



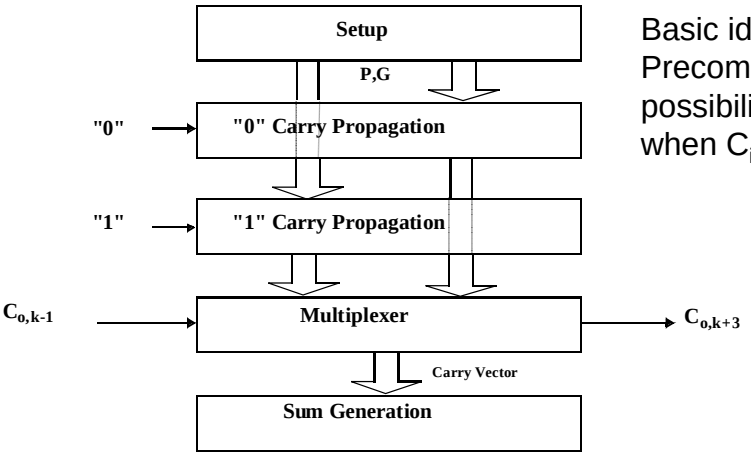
$$t_{adder} = t_{setup} + M_{tcarry} + (N/M\text{-}1)t_{bypass} + (M\text{-}1)t_{carry} + t_{sum}$$

Inner blocks do not contribute to worst-case delay since they have time to compute while bits 0-3 are propagating (assuming they have a generate or delete)

Carry Ripple versus Carry Bypass

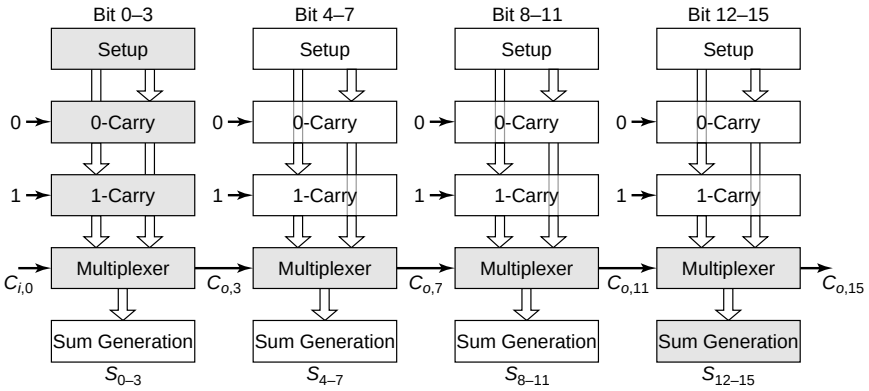


Carry-Select Adder



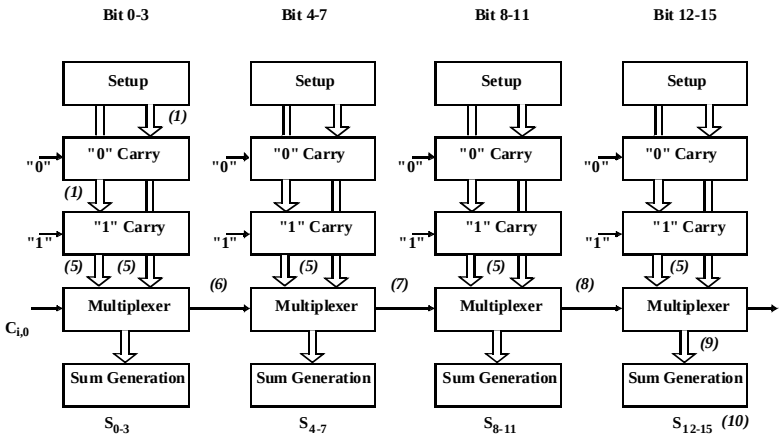
Basic idea: Precompute both possibilities, choose when C_{in} available

Carry Select Adder: Critical Path



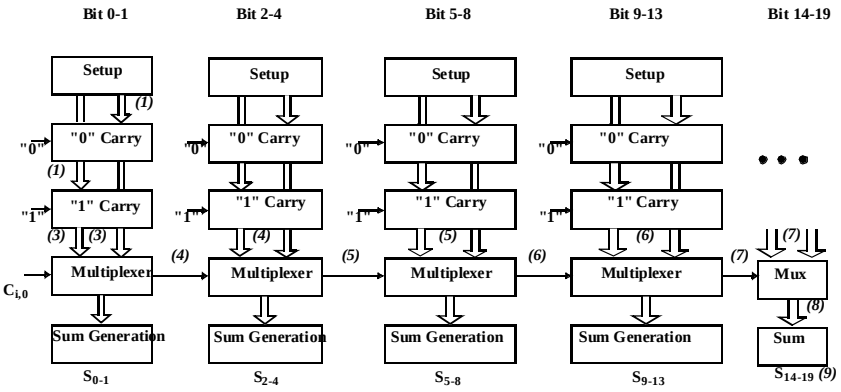
$$t_{add} = t_{setup} + Mt_{carry} + (N/M)t_{mux} + t_{sum}$$

Linear Carry Select



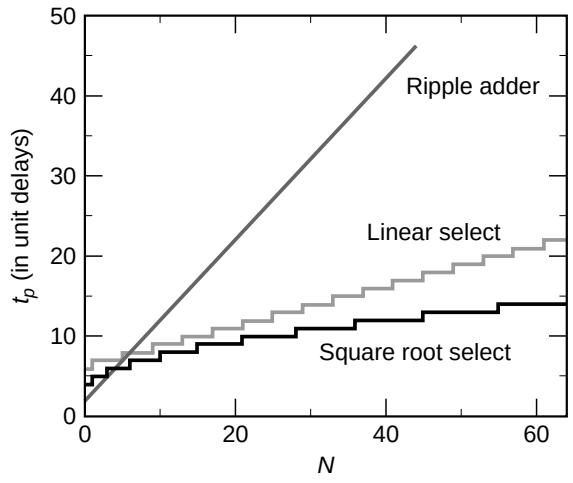
$$t_{add} = t_{setup} + M*t_{carry} + (N/M)t_{mux} + t_{sum}$$

Square Root Carry Select

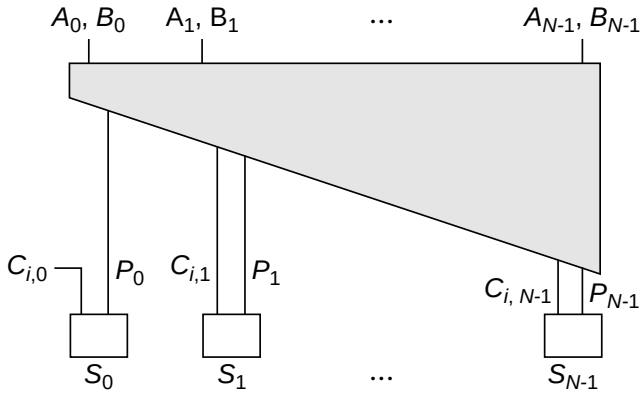


$$t_{add} = t_{setup} + Mt_{carry} + \left(\sqrt{2N}\right)t_{mux} + t_{sum}$$

Adder Delays - Comparison



Lookahead - Basic Idea



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

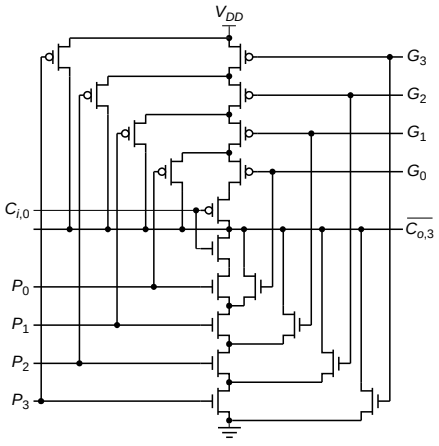
4-bit Lookahead Adder

Expanding Lookahead equations:

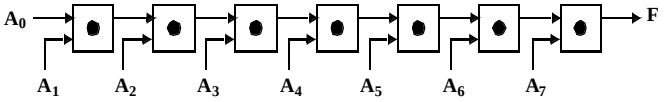
$$C_{o,k} = G_k + P_k(G_{k-1} + P_{k-1}C_{o,k-2})$$

All the way:

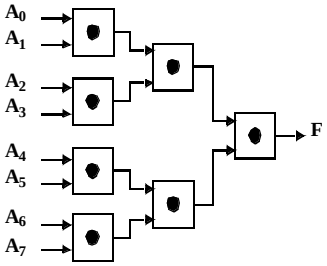
$$C_{o,k} = G_k + P_k(G_{k-1} + P_{k-1}(\ldots + P_1(G_0 + P_0 C_{i,0})))$$



Logarithmic Lookahead Adder



$t_p \sim N$



$t_p \sim \log_2(N)$

Carry Lookahead Trees

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

$$C_{o,1} = G_1 + P_1 G_0 + P_1 P_0 C_{i,0}$$

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

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

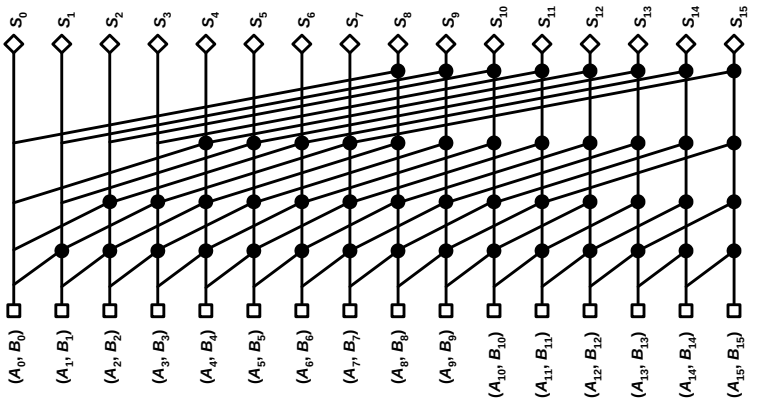
Can continue building the tree hierarchically.

In general

$G_{j:k} = G_j + P_j G_{j-1:k}$

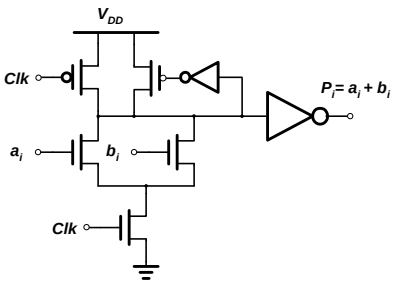
$P_{j:k} = P_j P_{j-1:k}$

Tree Adder

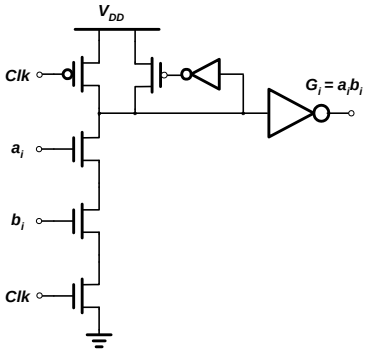


16-bit radix-2 Kogge-Stone tree

Domino Adder

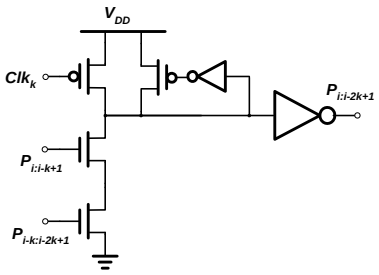


Propagate

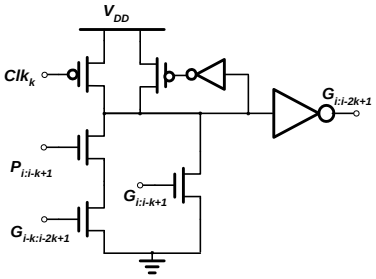


Generate

Domino Adder

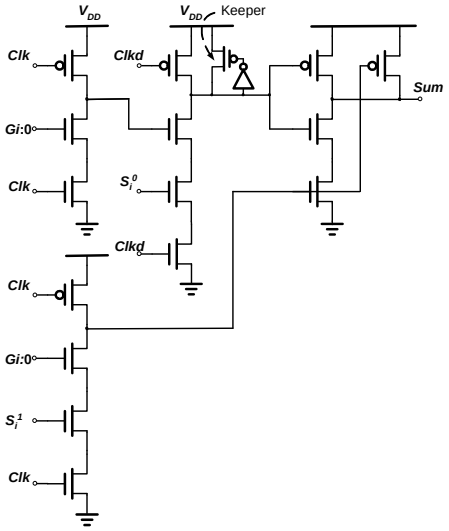


Propagate

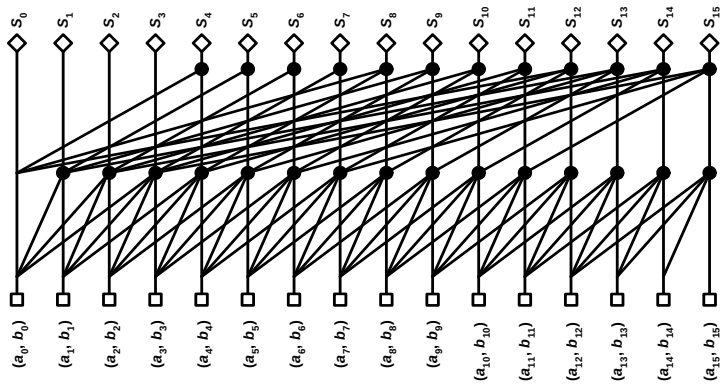


Generate

Domino Sum

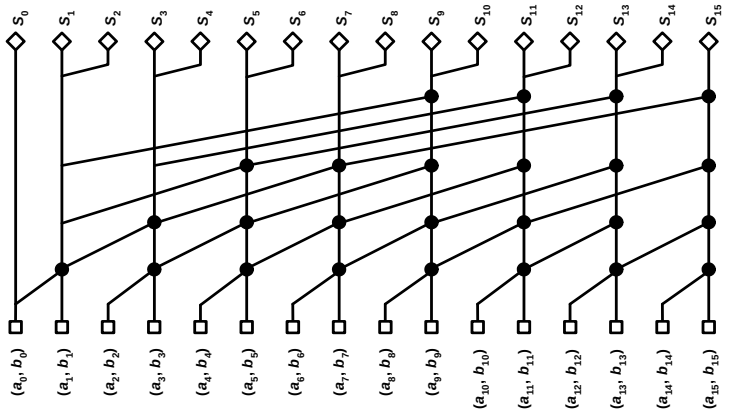


Tree Adder



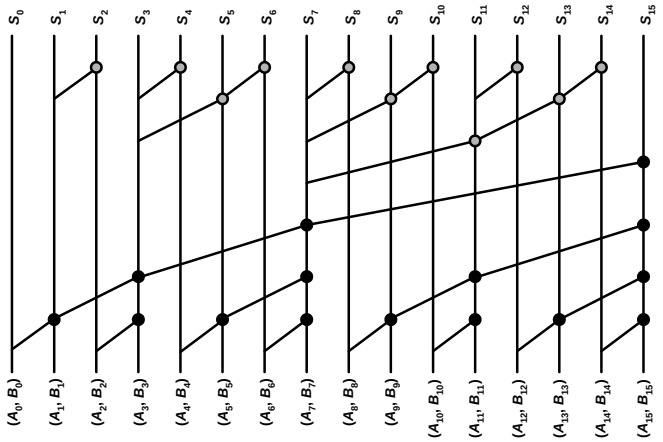
16-bit radix-4 Kogge-Stone Tree

Sparse Tree



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

Tree Adder



Brent-Kung Tree

Radix 2 vs. Radix 4

- Fanout consideration
  - Easier to drive large fanouts with Radix-2
  - Radix-4 has fewer stages and could have speed advantage when driving low fanouts
- Wire length consideration
  - Radix-4 has longer wires (64 bits: crosses 48 bit slices vs. 32 in radix-2). Fewer logic stages precedes large wireload.
- Logic consideration
  - Radix-2 has lower stack heights

Full vs. Sparse Trees

- Not all the carries are calculated in sparse trees
  - Only every 2nd, 4th, etc.
  - Reduced (uneven) input capacitance
  - Fewer transistors and wires
  - Lower power
- Sparse tree adders need to recover missing carries
  - Ripple (extra gate delay)
  - Precompute (extra fanout)
  - Complex precomputation can get into the critical path

Overview

- Full adder
  - Mirror adder – size it carefully
  - Transmission-gate adder – efficient XOR implementation
  - Manchester carry chain – long RC chain
- Adder topologies
  - Ripple carry – simple, small adders
  - Carry bypass – skip in groups
  - Carry select – precomputation to shorten the critical path
  - Carry lookahead – compute carry-in for higher order bits to shorten the critical path
- Carry look-ahead adder
  - Kogge-Stone – full tree, regular interconnect, consistent fanout, large area and high power
  - Radix 2 or Radix 4 – tradeoff between gate complexity and tree depth
  - Sparse tree, Brent-Kung – fewer gates, fewer wires, some high-fanout gates slows down the performance