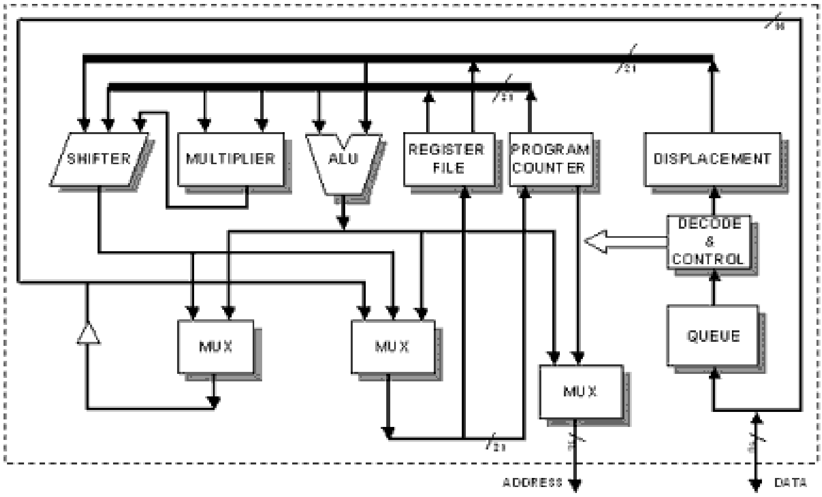
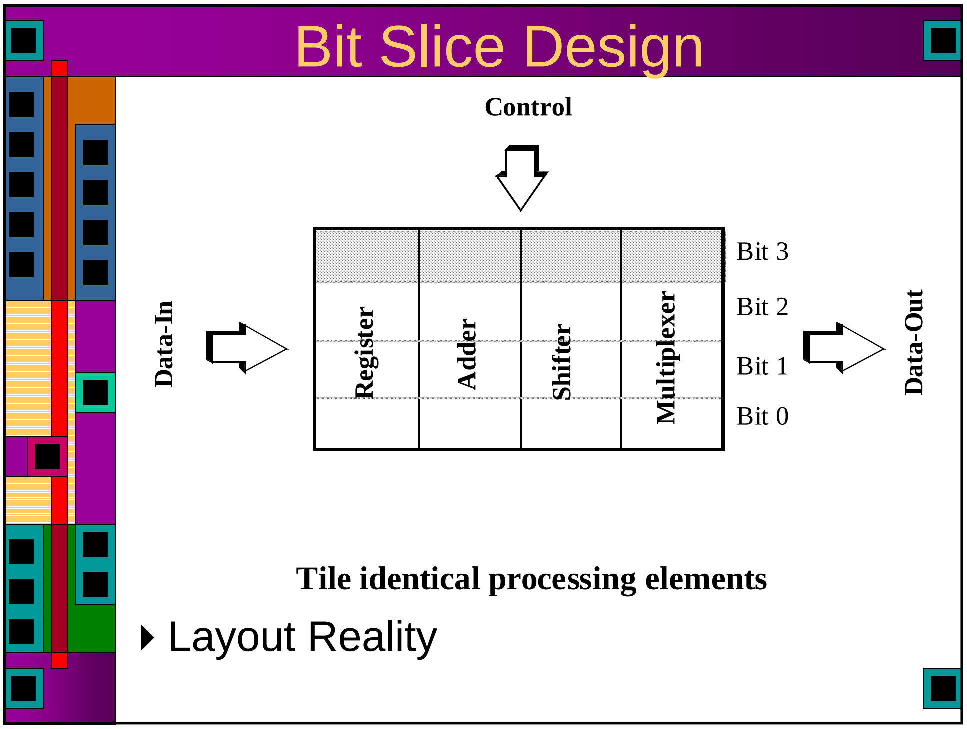
Where are we?

- Subsystem Design
  - Registers and Register Files
  - Adders and ALUs
  - Simple ripple carry addition
  - Transistor schematics
  - Faster addition
  - Logic generation
  - How it fits into the datapath

Data Path Design



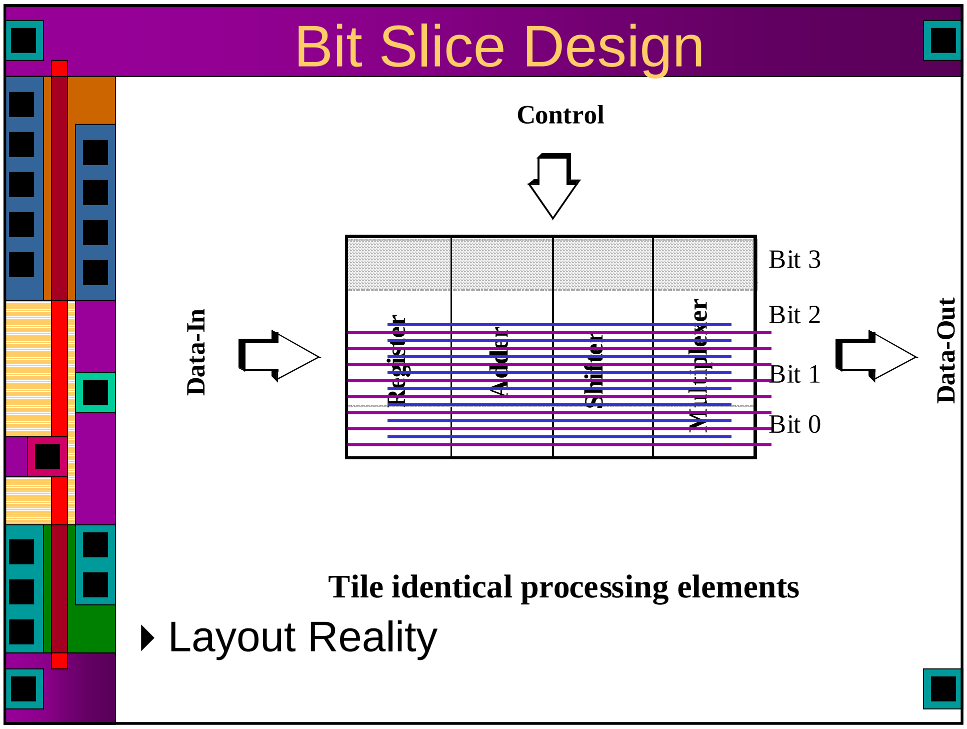
- Block-diagram style data path description



Bit Slice Design

Tile identical processing elements

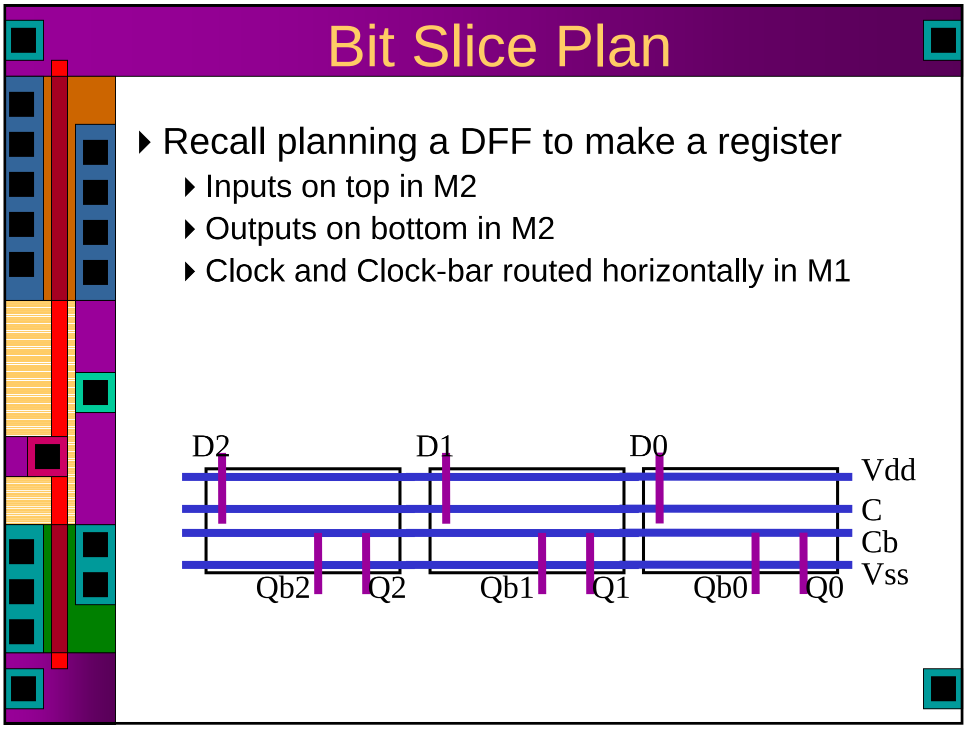
- Layout Reality



Bit Slice Design

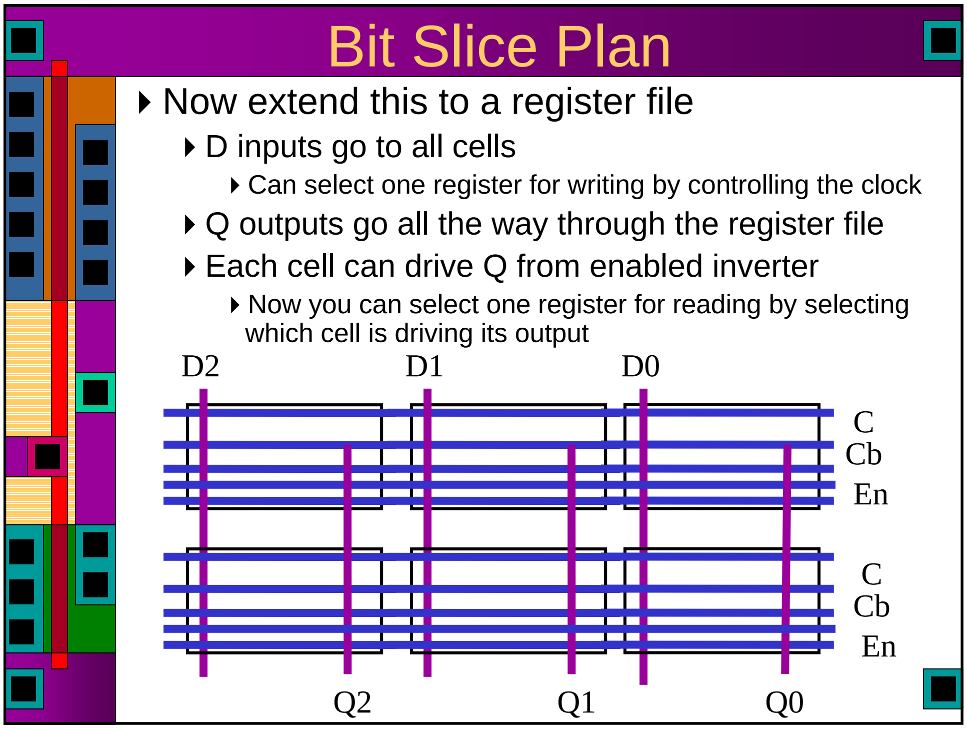
Tile identical processing elements

- Layout Reality



Bit Slice Plan

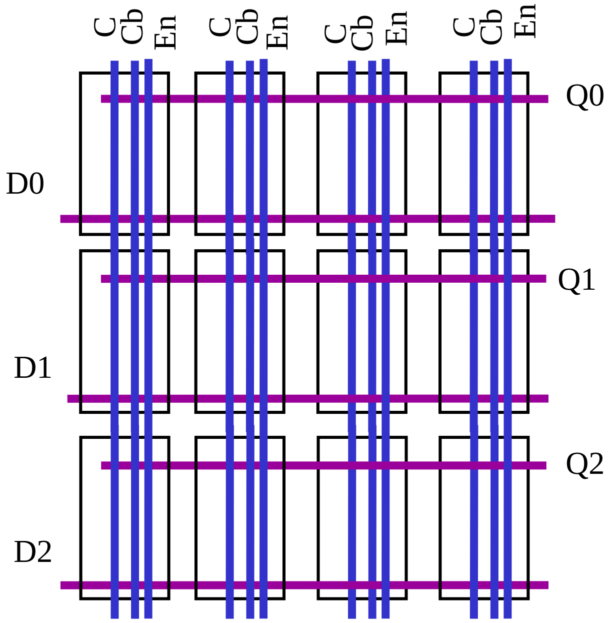
- Recall planning a DFF to make a register
  - Inputs on top in M2
  - Outputs on bottom in M2
  - Clock and Clock-bar routed horizontally in M1



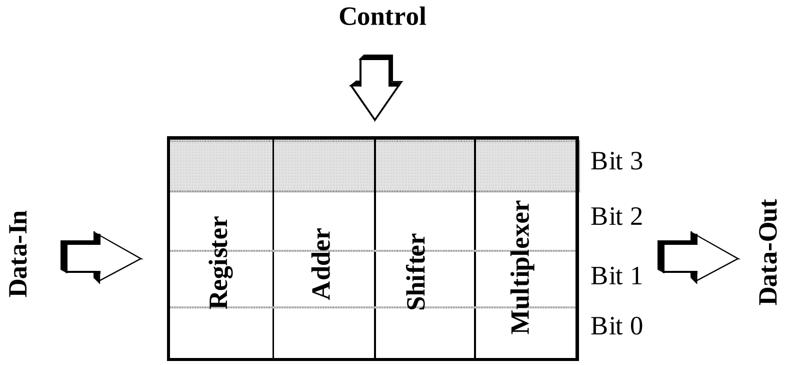
Bit Slice Plan

- Now extend this to a register file
  - D inputs go to all cells
    - Can select one register for writing by controlling the clock
  - Q outputs go all the way through the register file
  - Each cell can drive Q from enabled inverter
    - Now you can select one register for reading by selecting which cell is driving its output

Bit Slice Plan

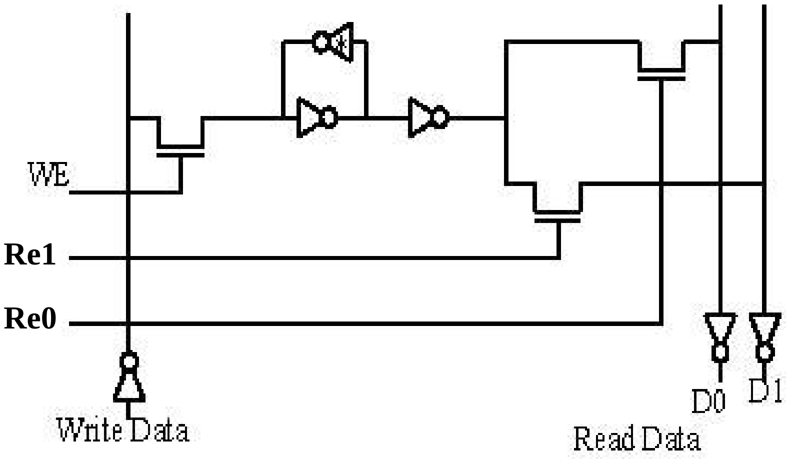


Bit Slice Design

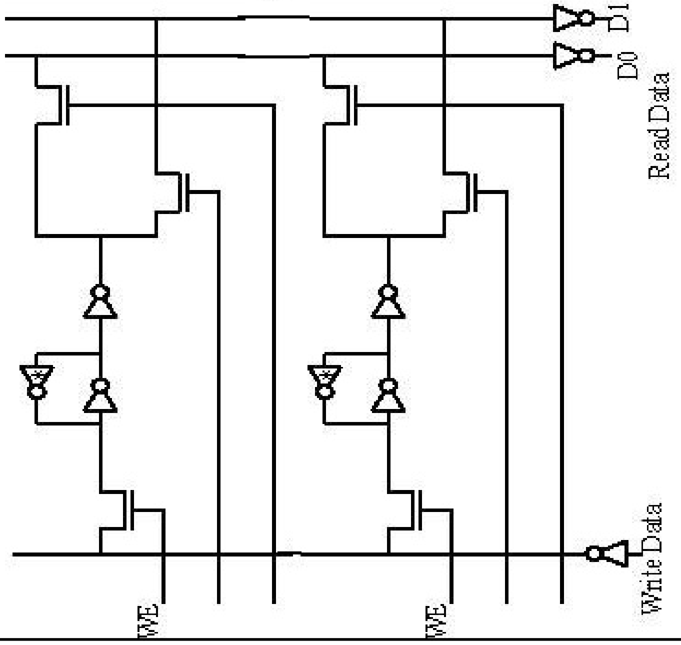


Tile identical processing elements

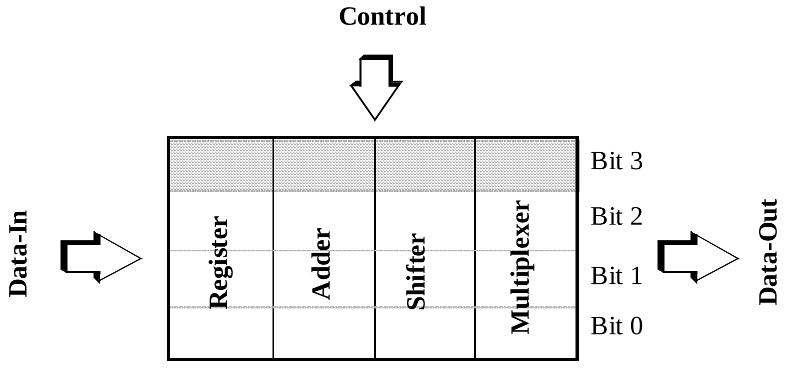
Multi-Port Register



Multi-Port Register



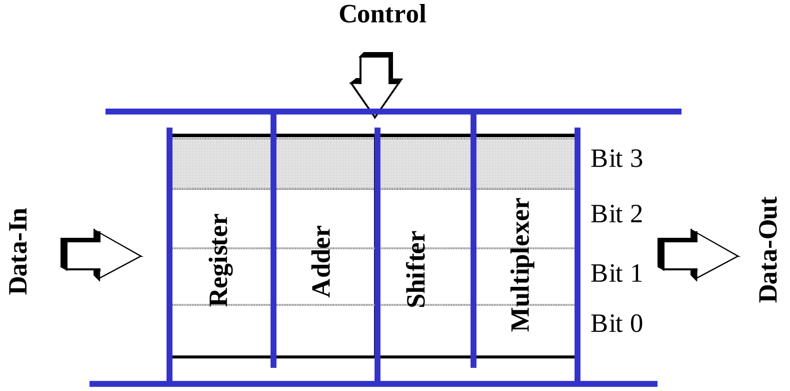
Bit Slice Design



Tile identical processing elements

- Where are power lines?

Bit Slice Design

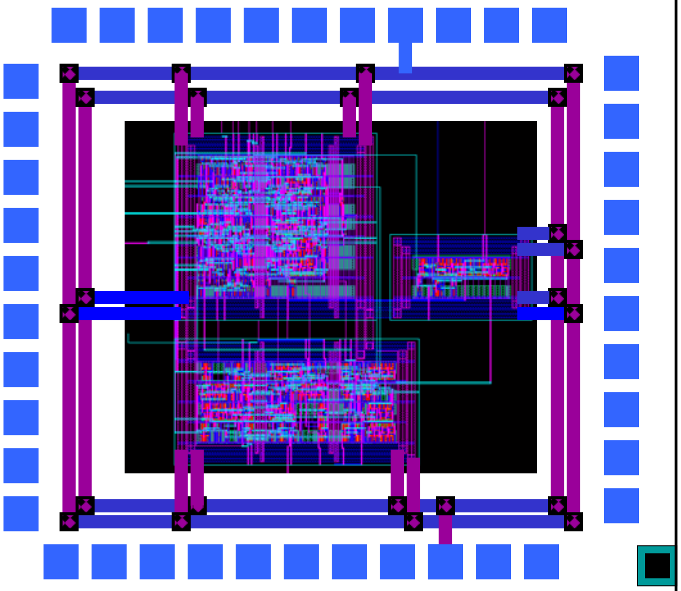


Tile identical processing elements

- Where are power lines? Basic Comb scheme

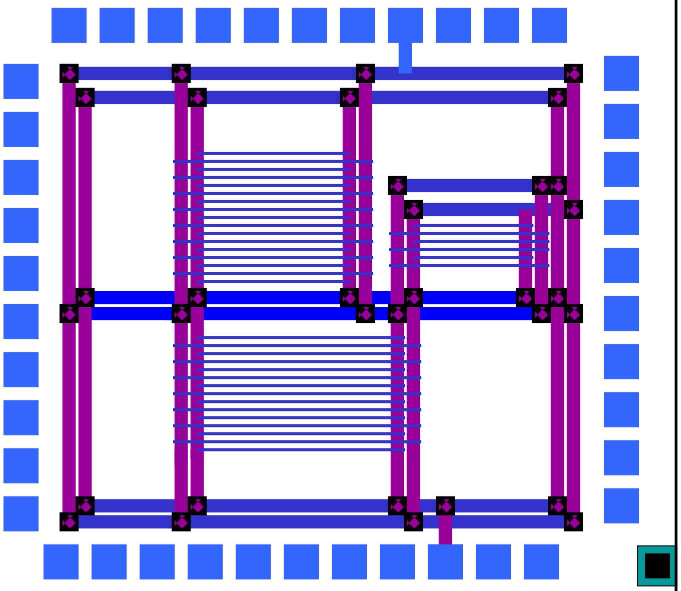
Chip-Wide View of Power

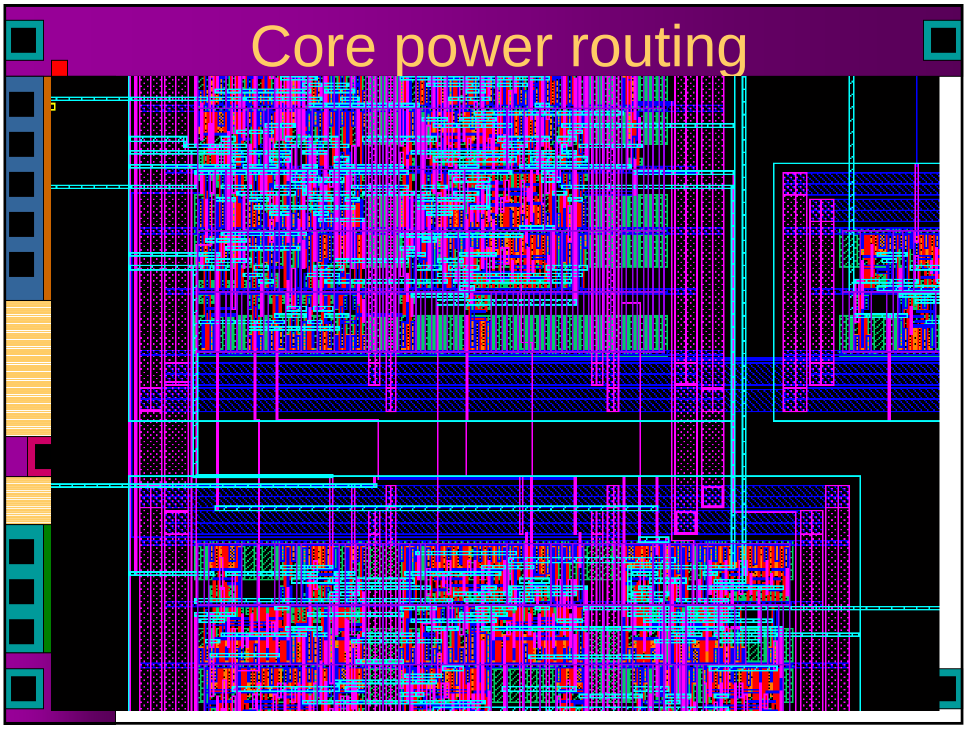
- Power Routing is a global chip-wide issue
- Here's another approach
- Note the Vdd and Gnd pads
- Global rings with combs for regions of the chip



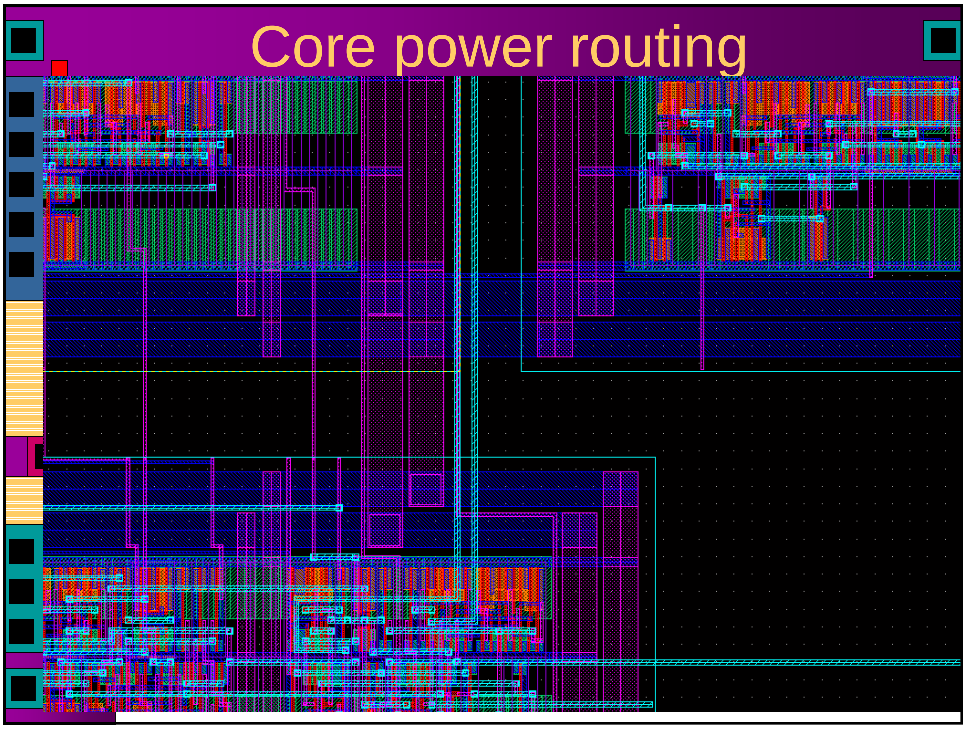
Chip-Wide View of Power

- Power Routing is a global chip-wide issue
- Here's another approach
- Note the Vdd and Gnd pads
- Global rings with combs for regions of the chip





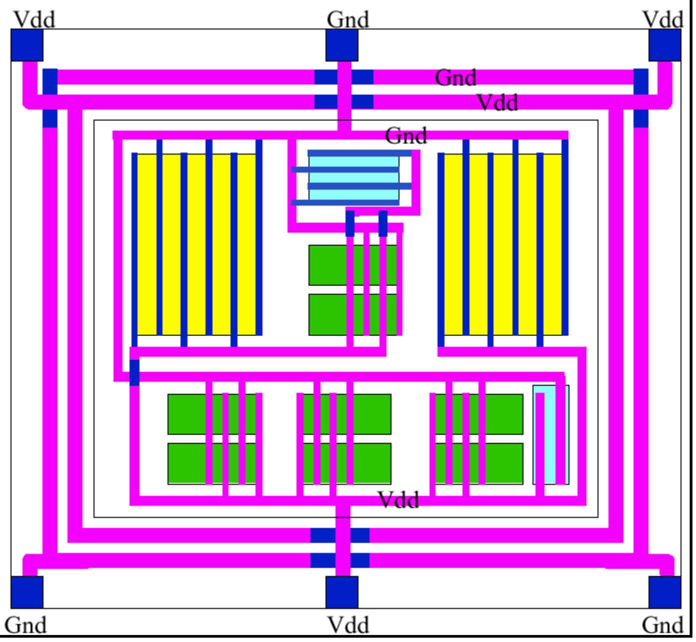
Core power routing



Core power routing

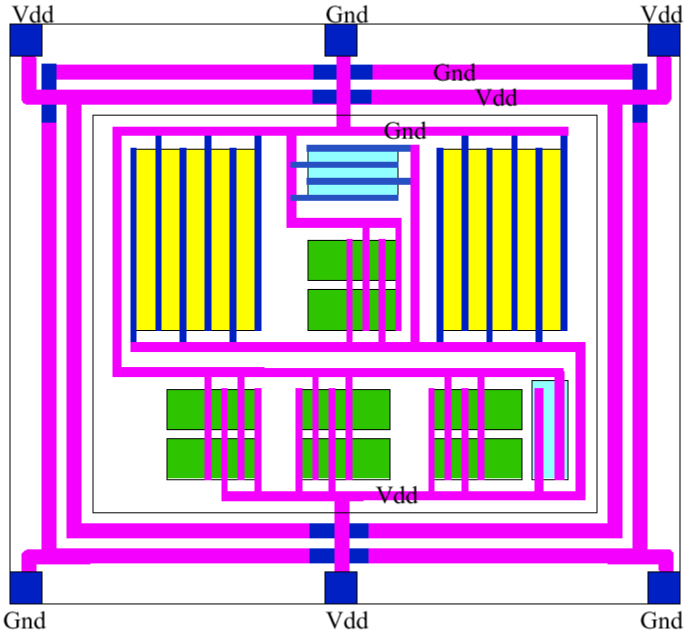
Chip-Wide View of Power

- Another view of the same issue
- Watch out for routing blockages!



A Tweak on the Scheme

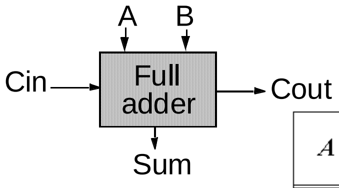
- Same basic scheme
- But with no internal jumpers
- Jumpers are restricted to outer loops



Adders Etc.

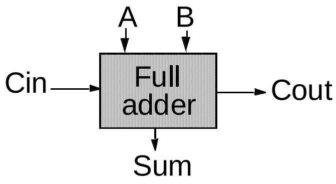
- Check out Chapter 10 in your text

Basic Addition: 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

Boolean Equations



- $$\begin{aligned}\mathrm{SUM} &= A \oplus B \oplus C \\ &= C(AB + \bar{A}\bar{B}) + \bar{C}(A\bar{B} + \bar{A}B)\end{aligned}$$
- $$\begin{aligned}\mathrm{CARRY} &= AB + AC + BC \\ &= AB + C(A + B)\end{aligned}$$
- **Above equations may be implemented as complex gates**
- **These equations may be manipulated to yield:**

$$\mathrm{SUM} = ABC + (A + B + C)\overline{\mathrm{CARRY}}$$

A Direct Implementation

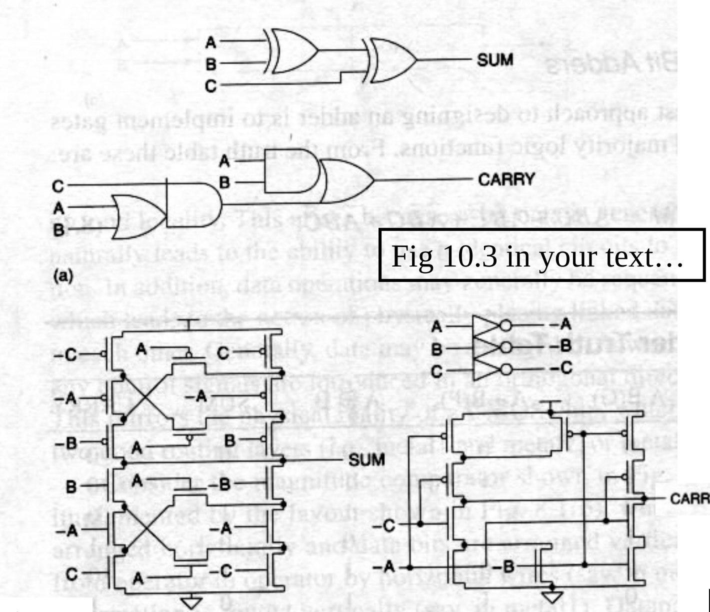
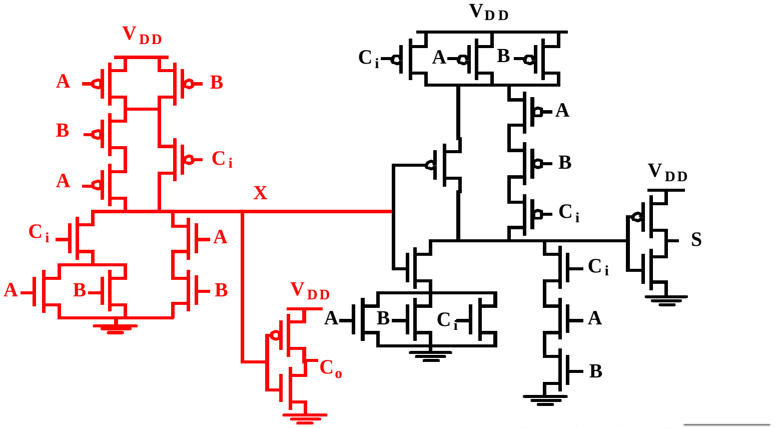


Fig 10.3 in your text…

32 transistors

Use the Factored Equations

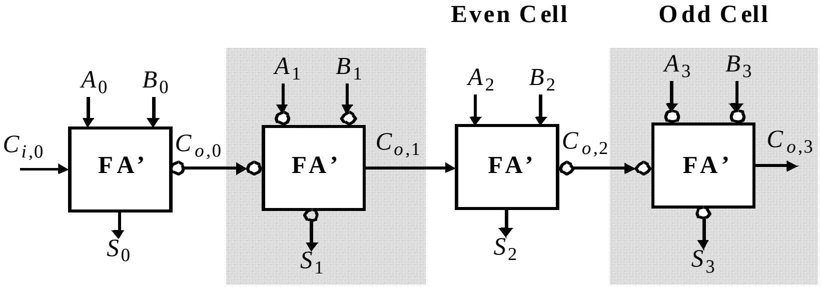


$$\mathrm{SUM} = \boldsymbol{ABC} + (\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C})\overline{\mathrm{CARRY}}$$

28 Transistors

- Fully static, complex gate implementation

Getting Rid of Inverters

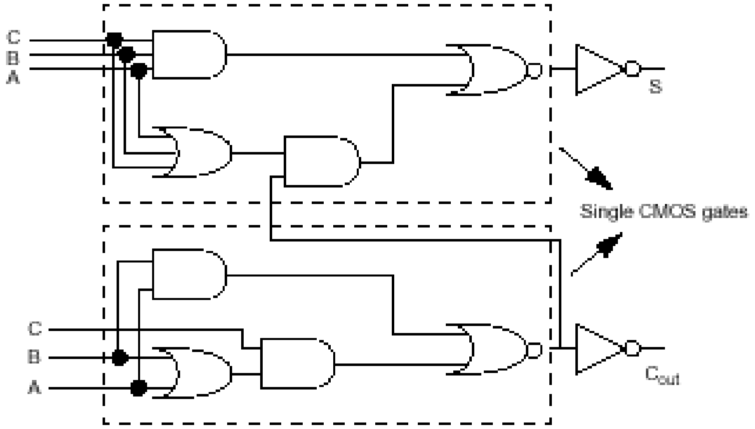


Exploit Inversion Property

Note: need 2 different types of cells

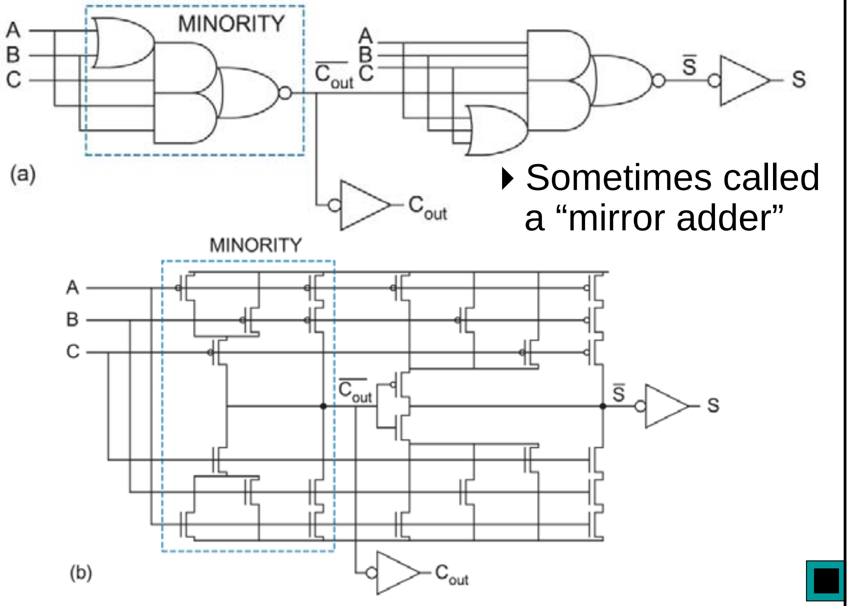
- Can improve performance by removing inverters from carry chain

A Better Static Gate



- Combine gates and reuse subterms

A Better Static Gate



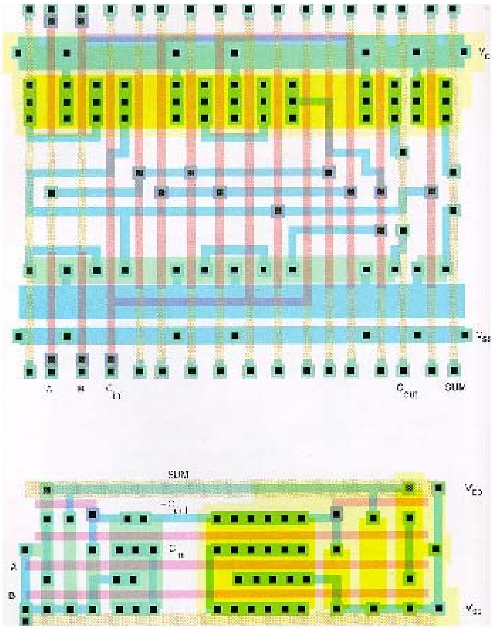
- Sometimes called a “mirror adder”

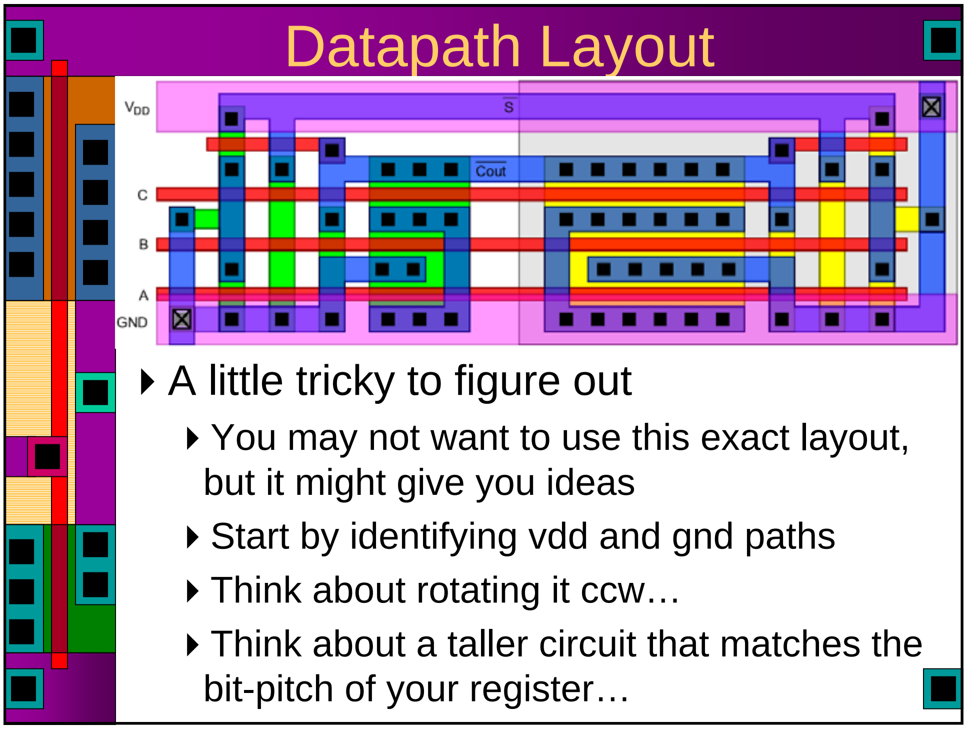
Mirror Adder Considerations

- •Feed the Carry-In to the inner inputs so the internal capacitance is already discharged
- •Make all transistors whose gates are connected to Cin and carry logic minimum size – minimizes branching effort on critical path (carry out)
- •Determine gate widths by Logical Effort – reduce effort from C to CoutB at the expense of Sum
- •Use relatively large transistors on critical path so that stray wiring cap is a small fraction of overall cap

Adder Layout

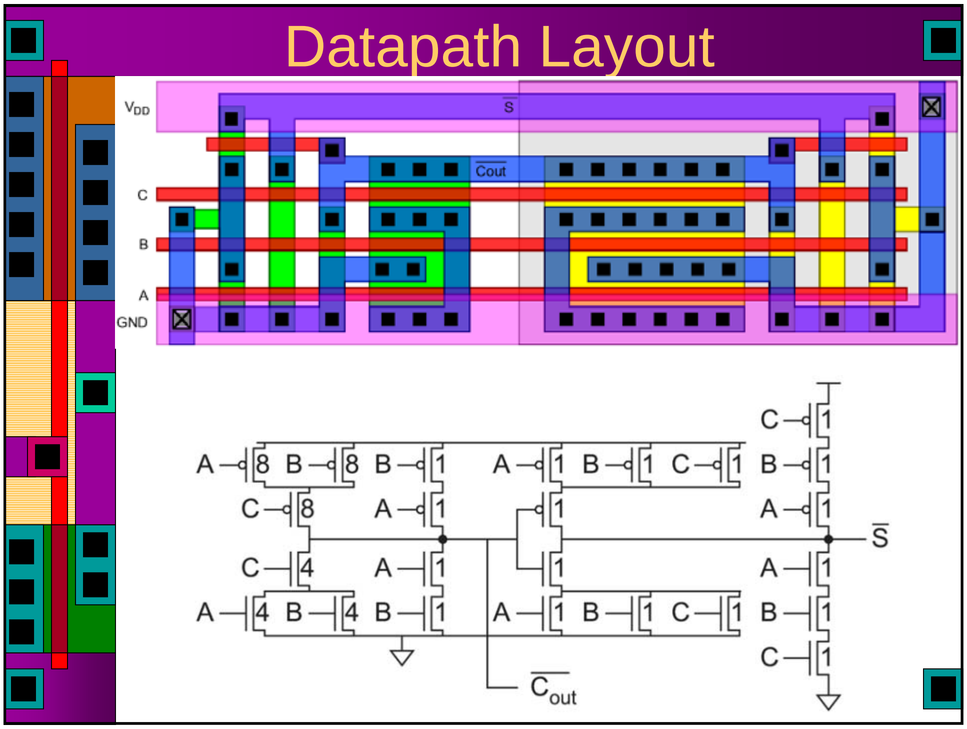
- Examples from Weste and Eshraghian
- “Standard Cell” vs. “Datapath”
- Definitely worth looking at carefully



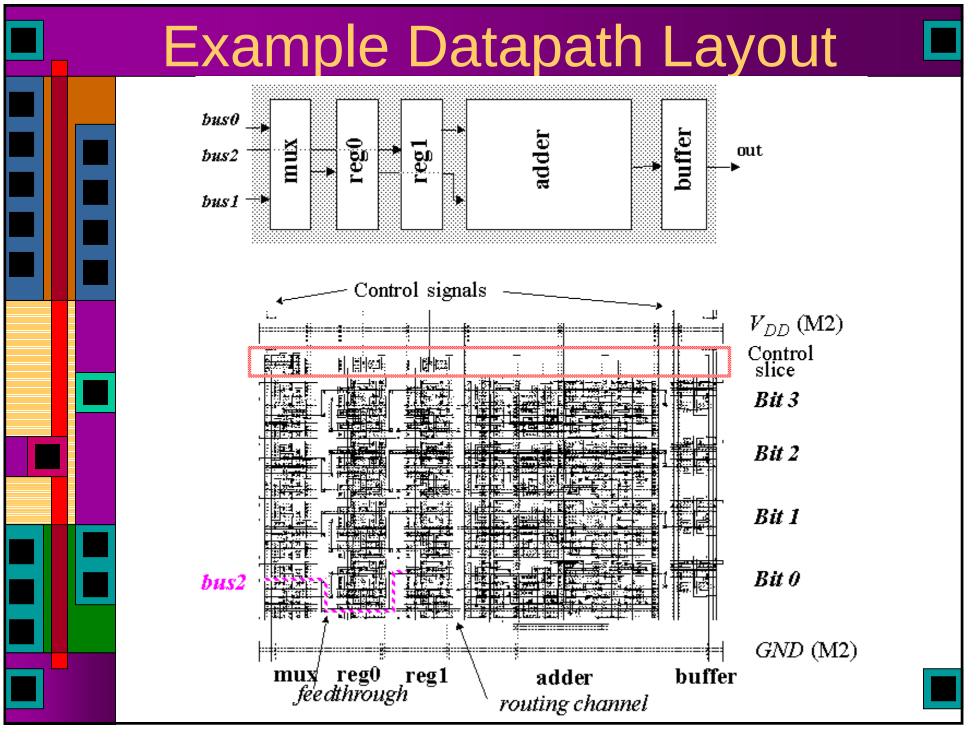


Datapath Layout

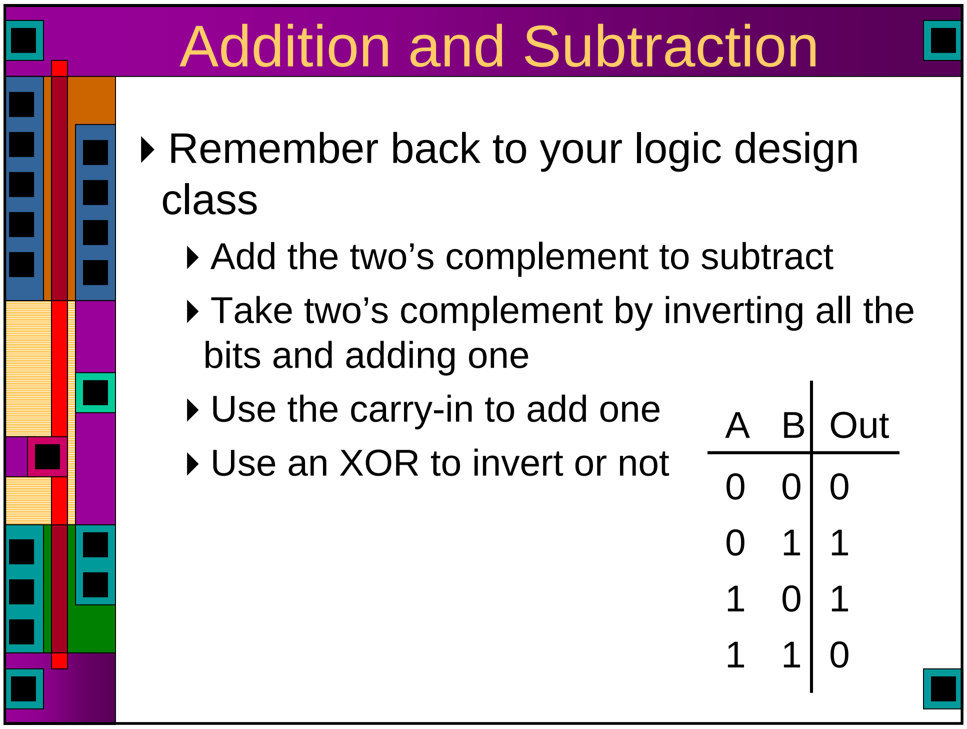
- A little tricky to figure out
  - You may not want to use this exact layout, but it might give you ideas
  - Start by identifying vdd and gnd paths
  - Think about rotating it ccw…
  - Think about a taller circuit that matches the bit-pitch of your register…



Datapath Layout



Example Datapath Layout

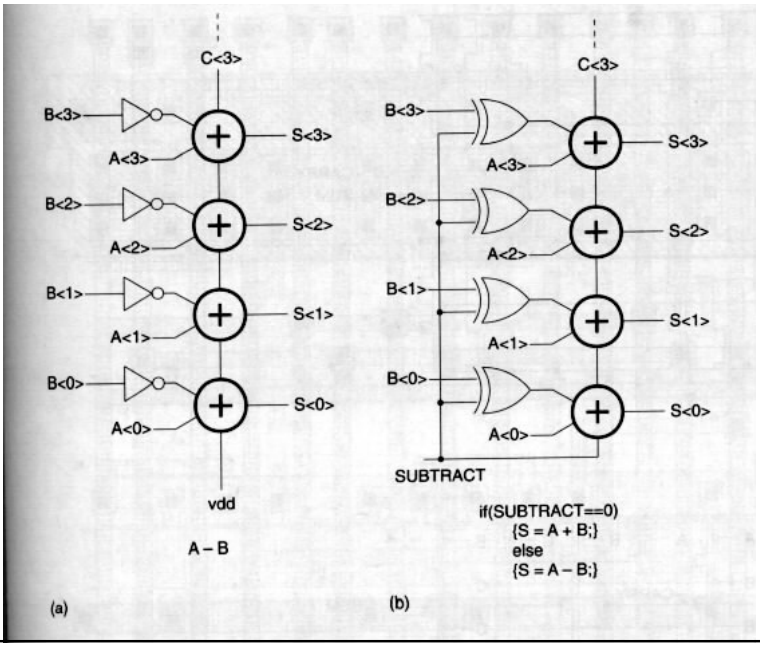


Addition and Subtraction

- Remember back to your logic design class
  - Add the two’s complement to subtract
  - Take two’s complement by inverting all the bits and adding one
  - Use the carry-in to add one
  - Use an XOR to invert or not

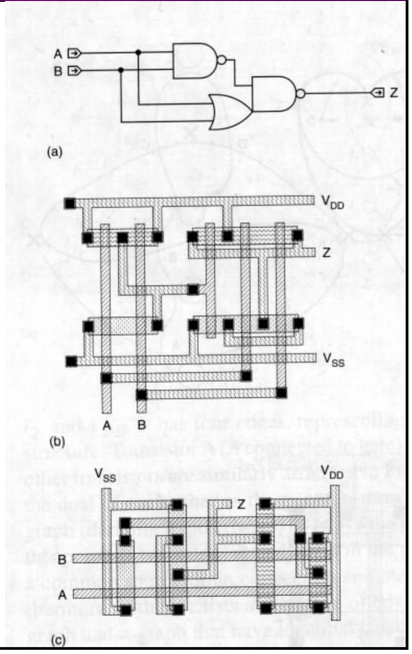
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

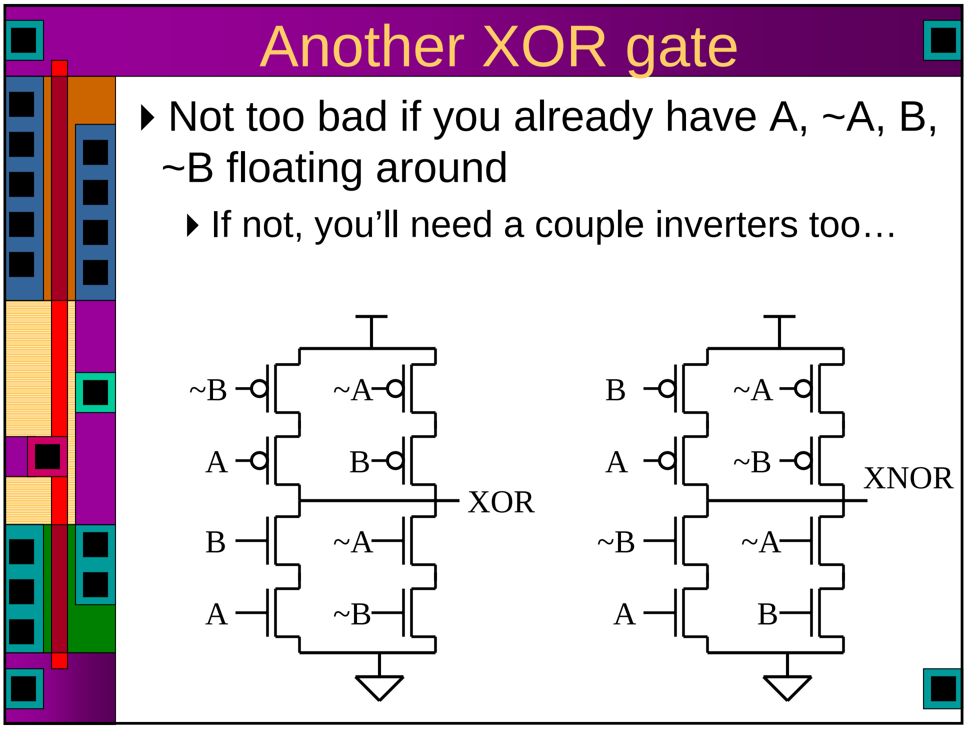
Two’s Complement Add/Sub



Aside: XOR Gates

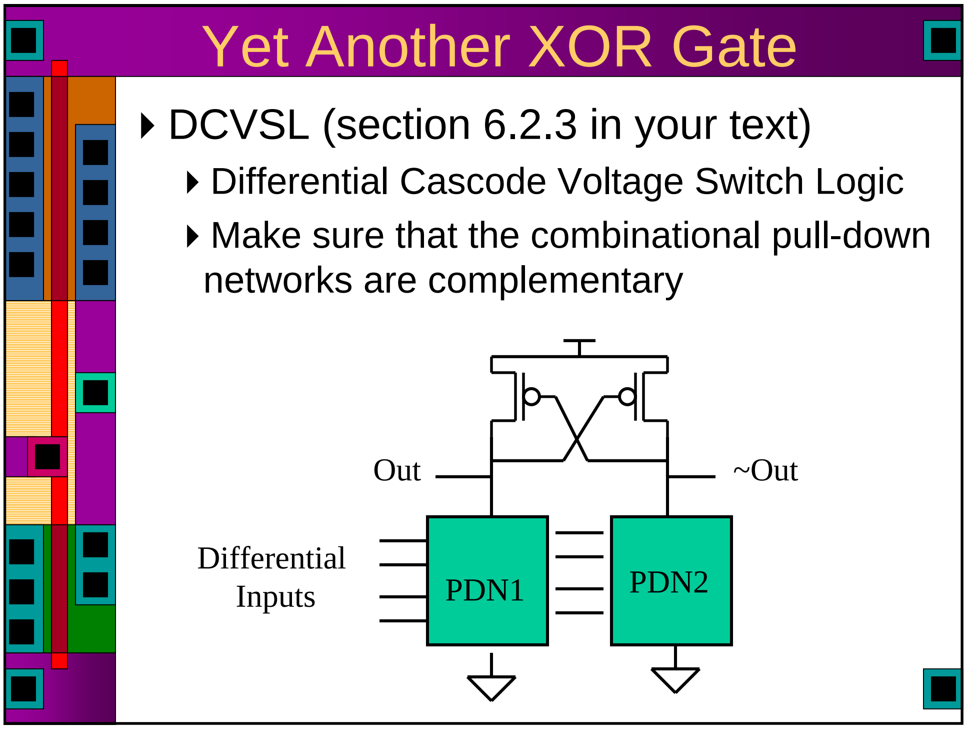
- Slightly tricky gate, ~AB + A~B
- Lots of different schematics…





Another XOR gate

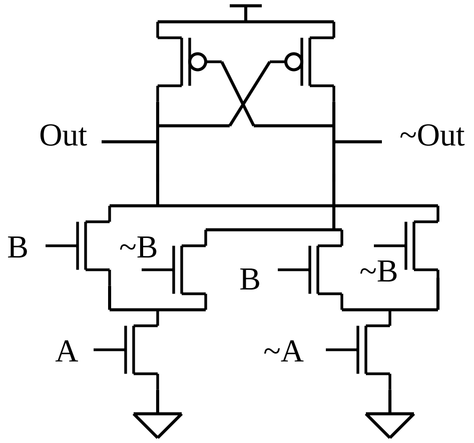
- Not too bad if you already have A, ~A, B, ~B floating around
  - If not, you'll need a couple inverters too…



Yet Another XOR Gate

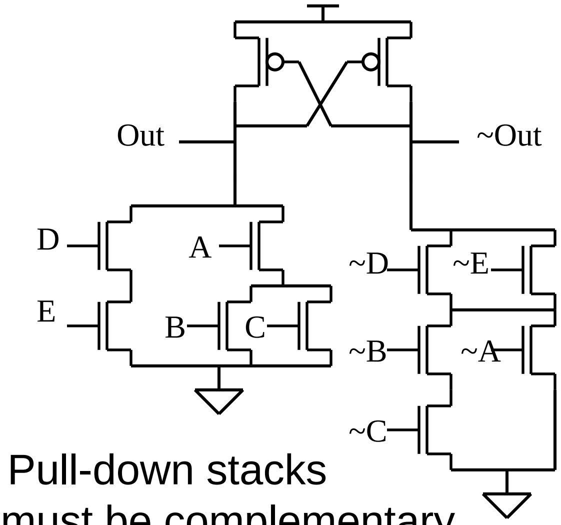
- DCVSL (section 6.2.3 in your text)
  - Differential Cascode Voltage Switch Logic
  - Make sure that the combinational pull-down networks are complementary

DCVSL XOR/XNOR



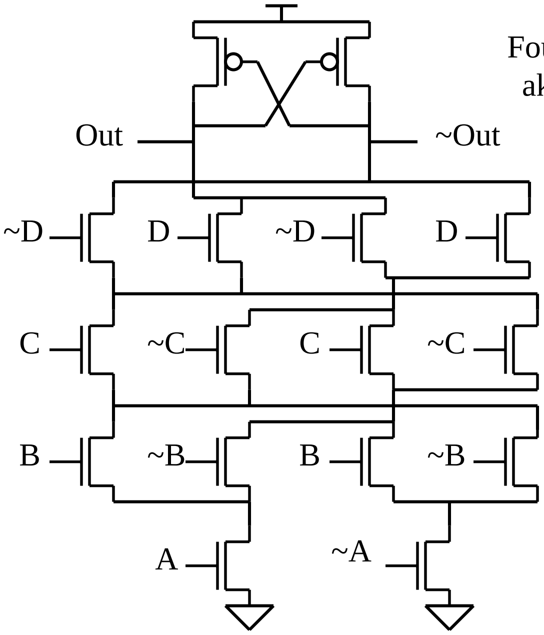
- Generates both XOR/XNOR
- Still static, but might be slower than others

Another DCVSL Example



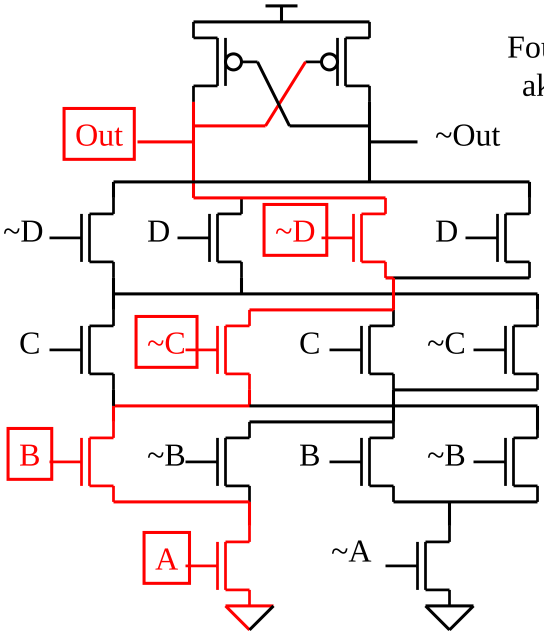
- Pull-down stacks must be complementary

DCVSL Large XOR



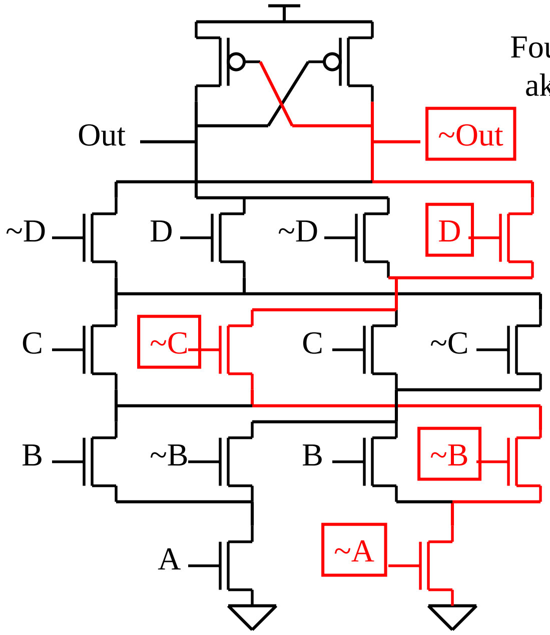
Four-input XOR
aka odd parity

DCVSL Large XOR



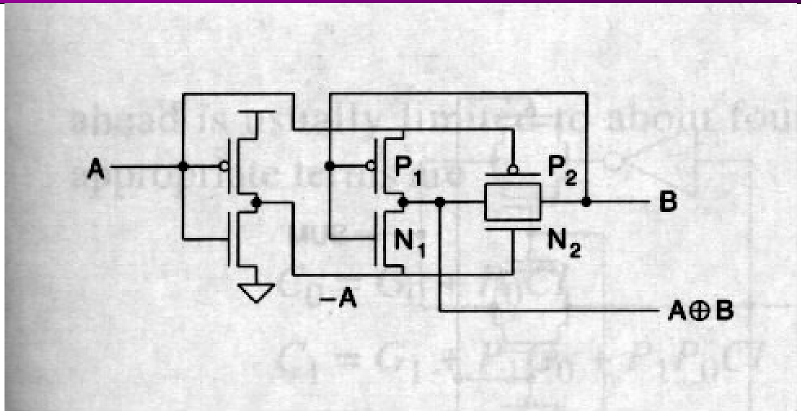
Four-input XOR
aka odd parity

DCVSL Large XOR



Four-input XOR
aka odd parity

Transmission Gate XOR

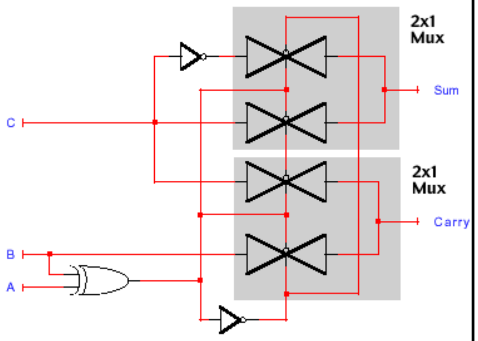


- Tiny, clever circuit
  - If A is high, N1, P1 act like inverter
  - If A is low, B is passed to the output through transmission gate

Transmission Gate Adder

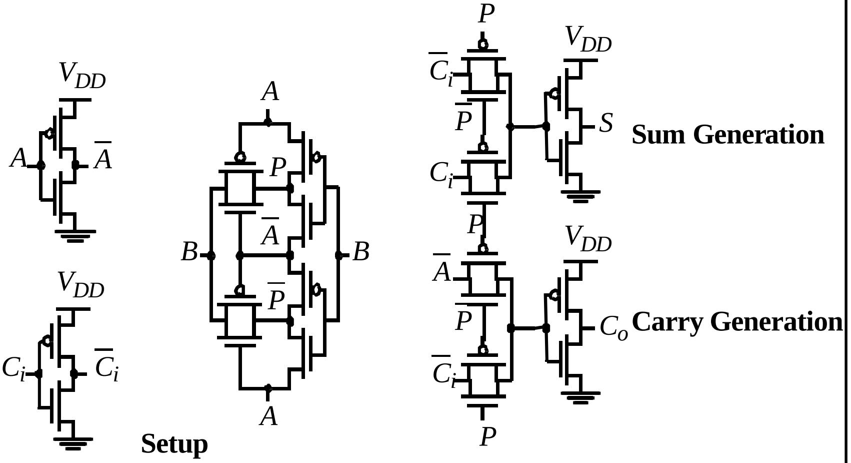
- Truth Table

A	B	C	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



- When $A \oplus B = 0$, SUM $= C$, and Carry $= B$.
- When $A \oplus B = 1$, SUM $= \bar{C}$, and Carry $= C$.
- Using the 6T XOR, this full adder uses 18T.

Another Version

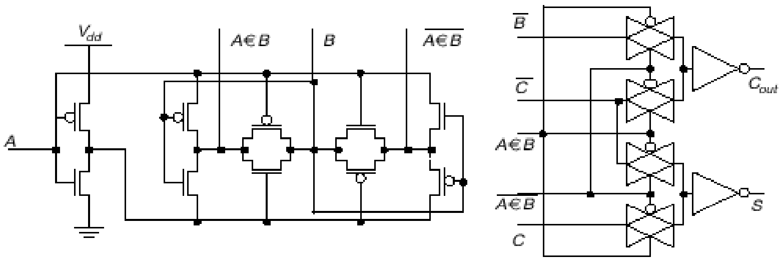


Sum Generation

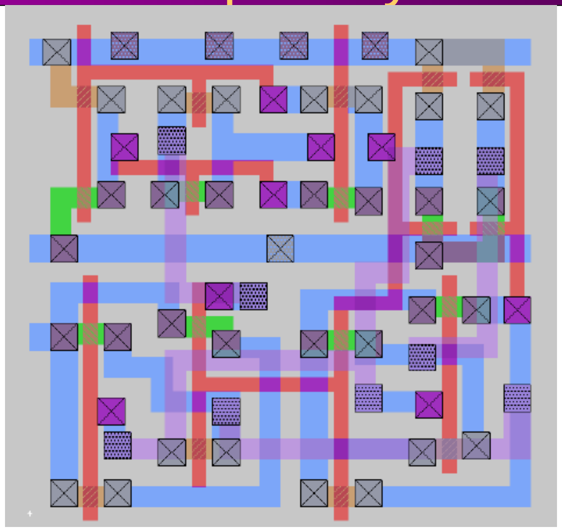
Carry Generation

Setup

Yet Another Version



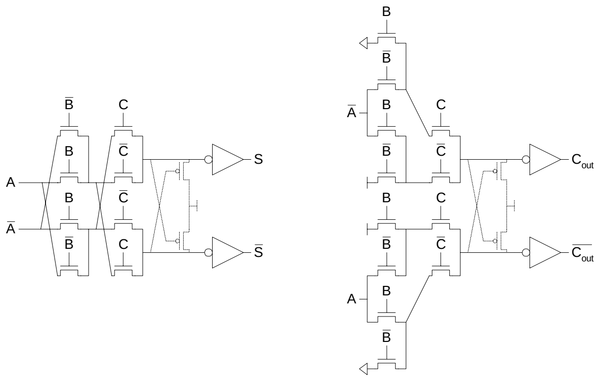
An Example Layout…



- Not the same style we’re used to seeing…

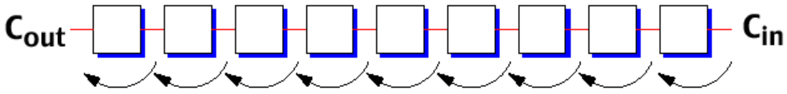
More Pass Transistors

- Complementary Pass Transistor Logic (CPL)
  - Slightly faster, but more area



Speeding Up Addition

- It all comes back to the carry circuit
  - Ripple carry delay goes from low-order to high-order bit
  - This determines the speed of the addition



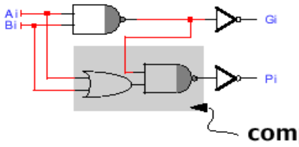
Delay is proportional to *n*

  - Many many ways to speed up the carry calculation

Section 10.2.2 in your text

Carry Lookahead

- A carry out C_i is generated from bit position i, when both A_i and B_i are '1' i.e. $G_i = A_iB_i$
- A carry in is propagated to the carry out at bit position i when either A_i or B_i is '1' (if both are '1' G_i will cover) e.g. $P_i = A_i \oplus B_i$



Sum = P ⊕ Ci

complex gate (6T)

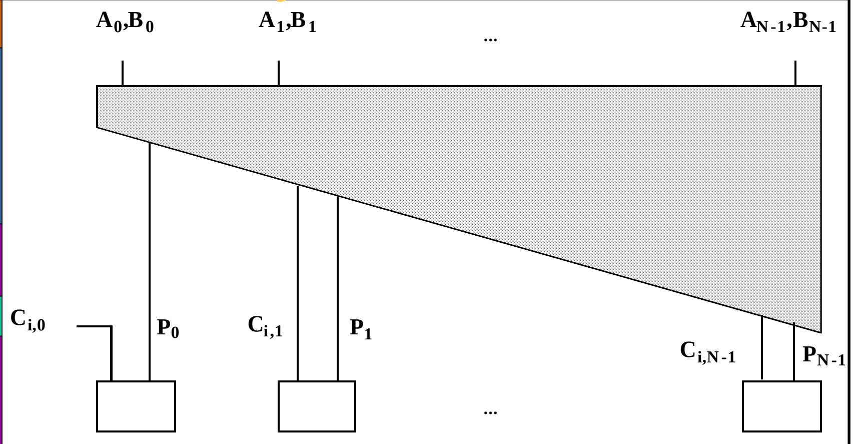
- Thus the carryout, $C_i = G_i + P_iC_{i-1}$

- Key is that the carry depends ONLY on A and B, not the carry-in
  - Catch is that the gates have large fan-in

Carry Lookahead

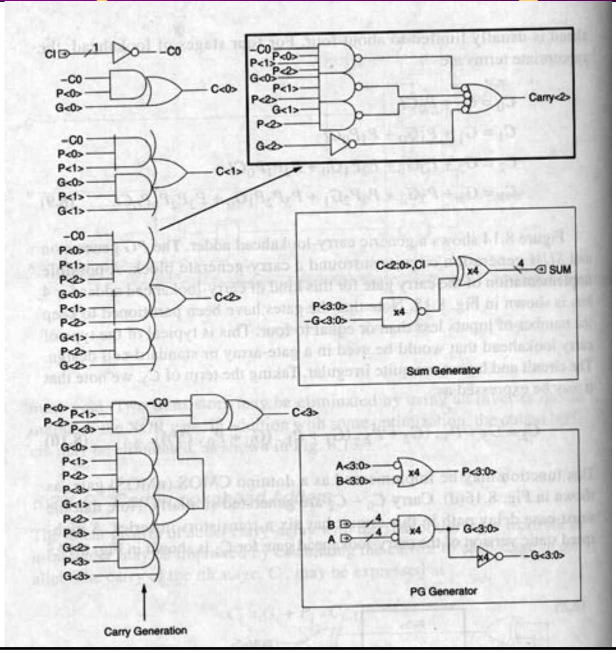
- Restated: Ci = Gi + Pi C(i-1)
- C0 = G0 + P0 Cin
- C1 = G1 + P1 C0
  = G1 + P1(G0 + P0 Cin)
  = G1 + P1 G0 + P1 P0 Cin
- C2 = G2 + P2G2 + P2P1G0 + P2P1P0Cin
- C3 = G1 + P3G2 + P3P2G1 + P3P2P1G0
  + P3P2P1P0Cin
- Or C3 = G3 + P3(G2 +P2(G1 + P1(G0 + P0 Cin)))

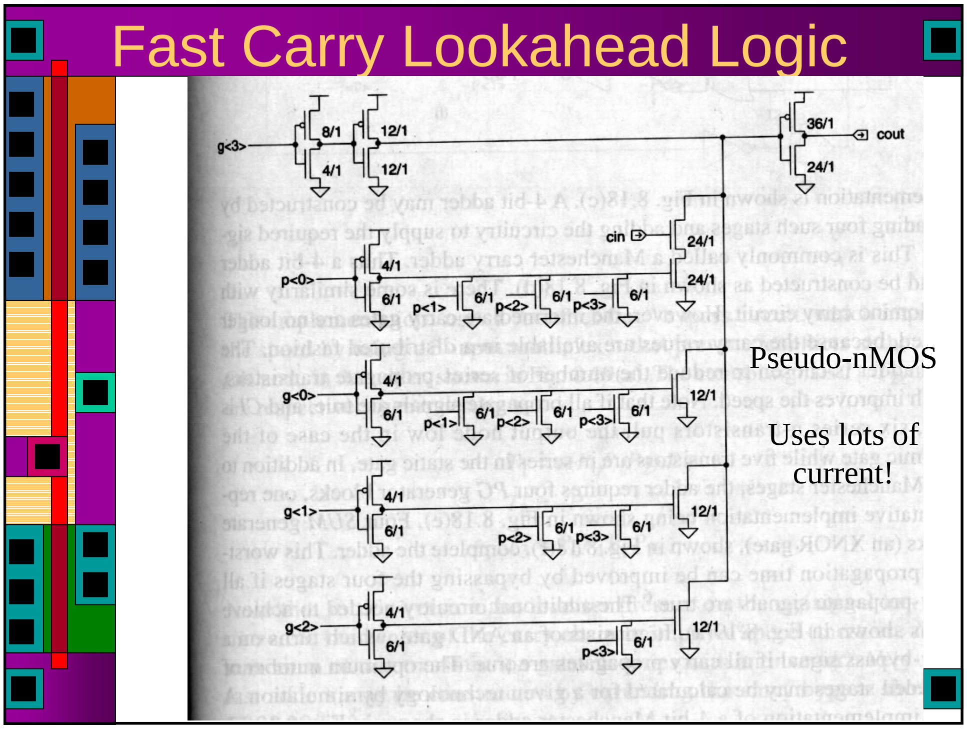
Carry Lookahead



- The C equations get larger with each stage
  - Usually do lookahead in small blocks (I.e. 4) and the combine in a tree

Carry Lookahead Logic

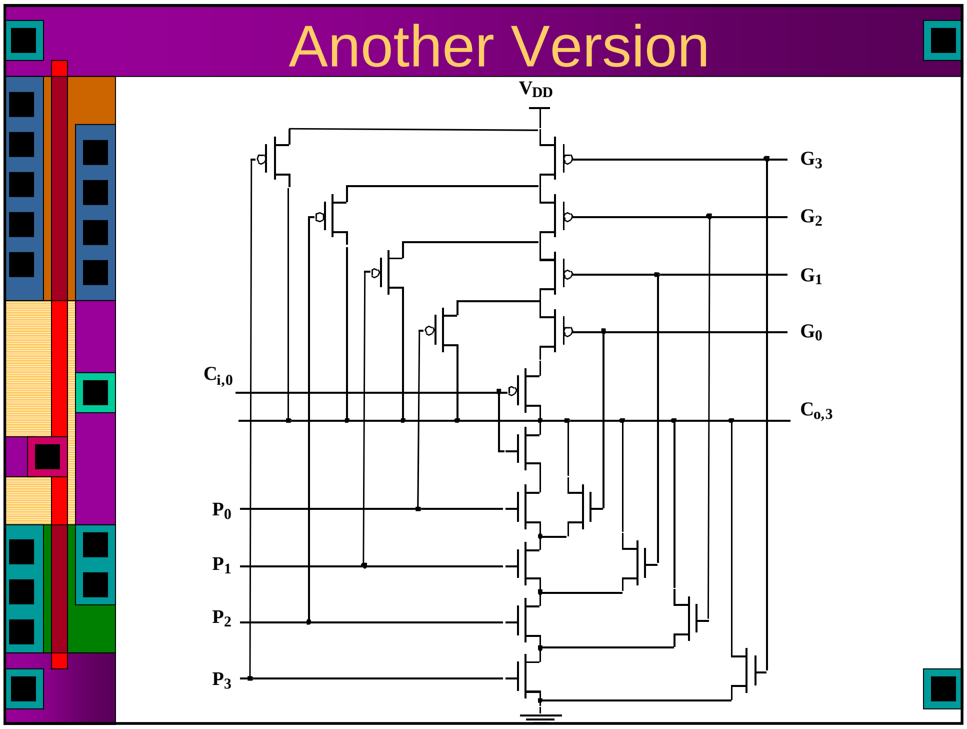




Fast Carry Lookahead Logic

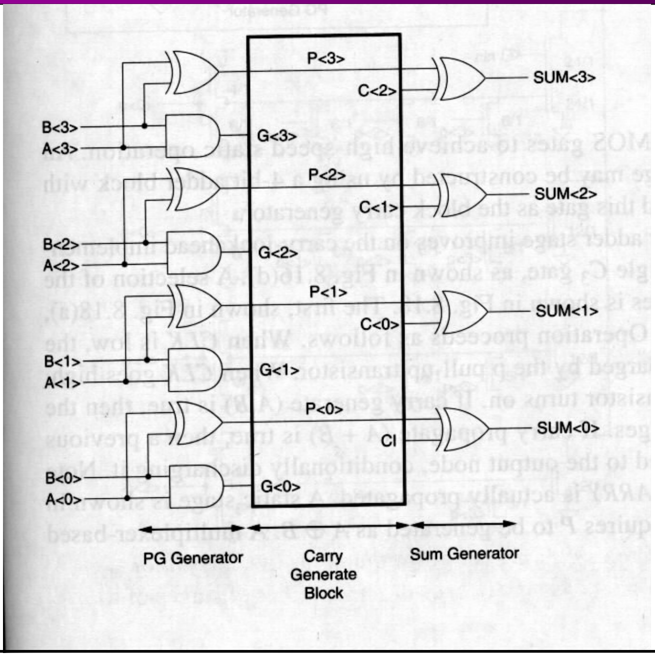
Pseudo-nMOS

Uses lots of current!

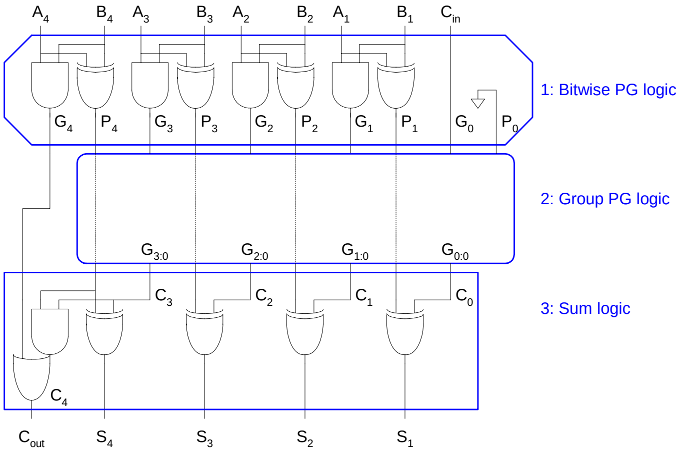


Another Version

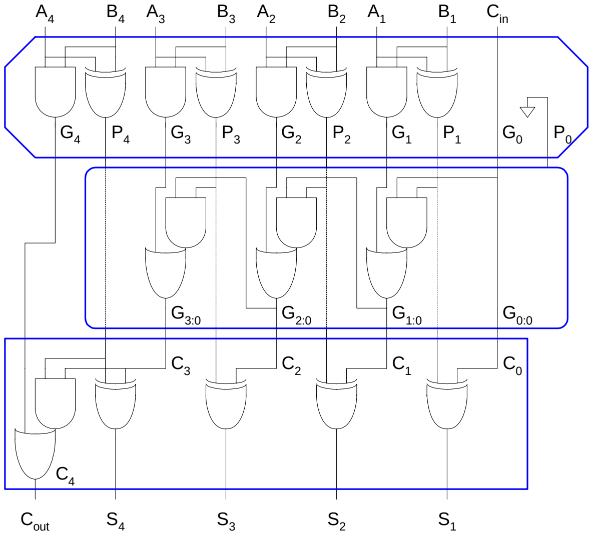
Another View



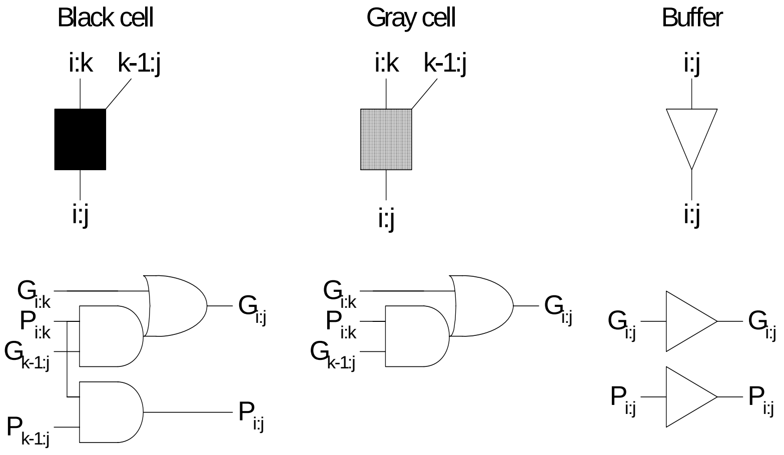
Another View



Ripple Carry



PG Diagram Notation


Black cell

i:k k-1:j

i:j

Gray cell

i:k k-1:j

i:j

Buffer

i:j

i:j

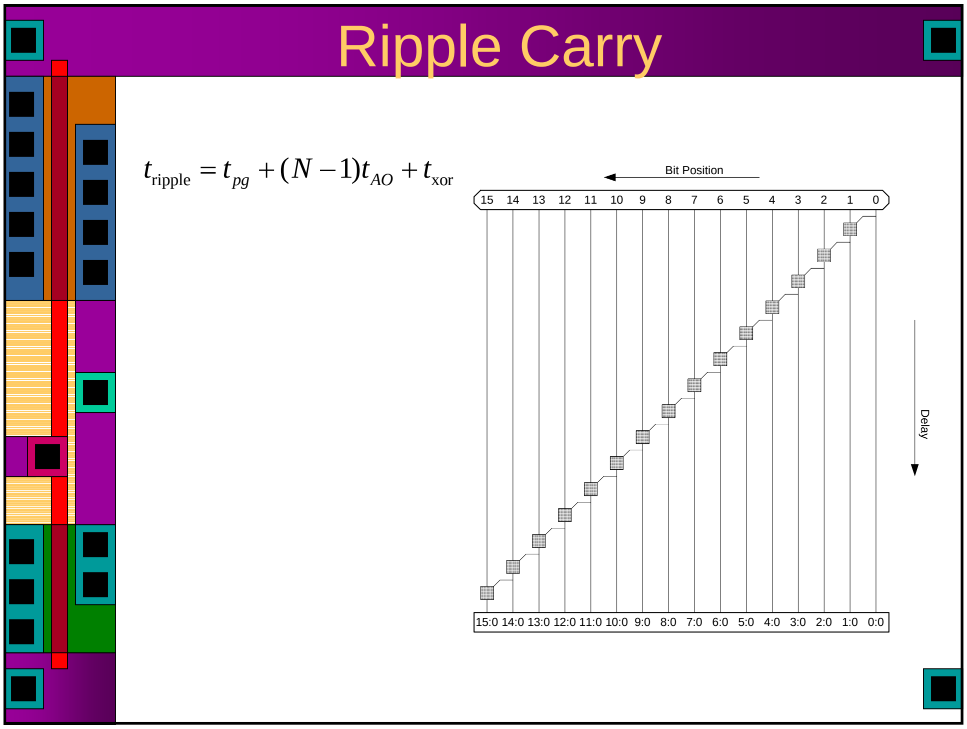
$G_{i:k}$ $P_{i:k}$ $G_{k-1:j}$ $G_{i:j}$

$P_{k-1:j}$ $P_{i:j}$

$G_{i:k}$ $P_{i:k}$ $G_{k-1:j}$ $G_{i:j}$

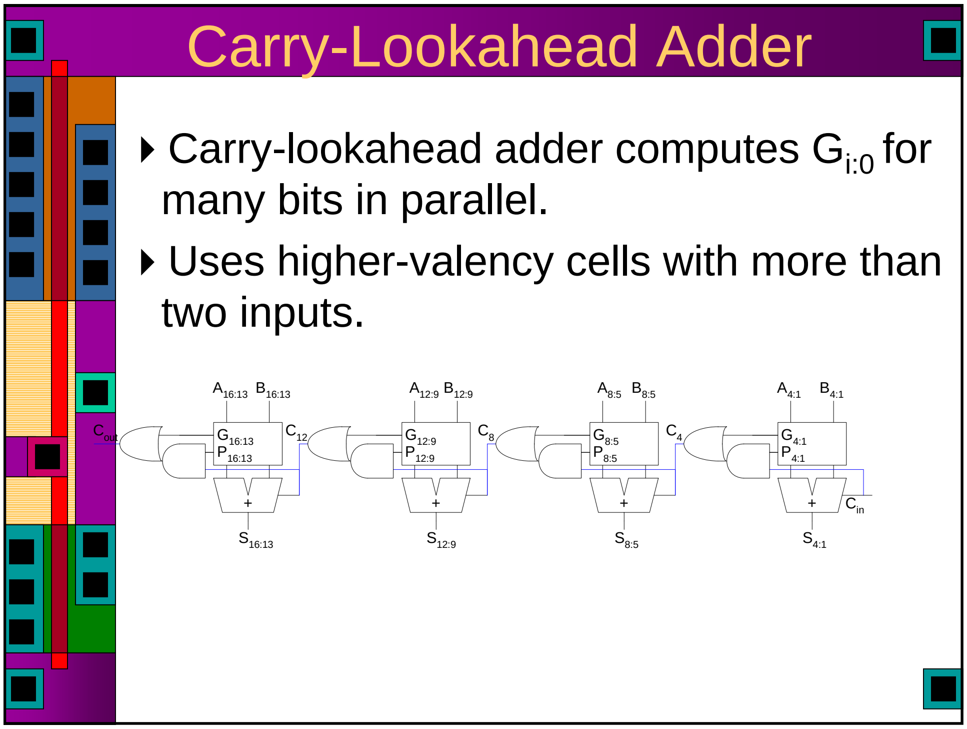
$G_{i:j}$ $G_{i:j}$

$P_{i:j}$ $P_{i:j}$



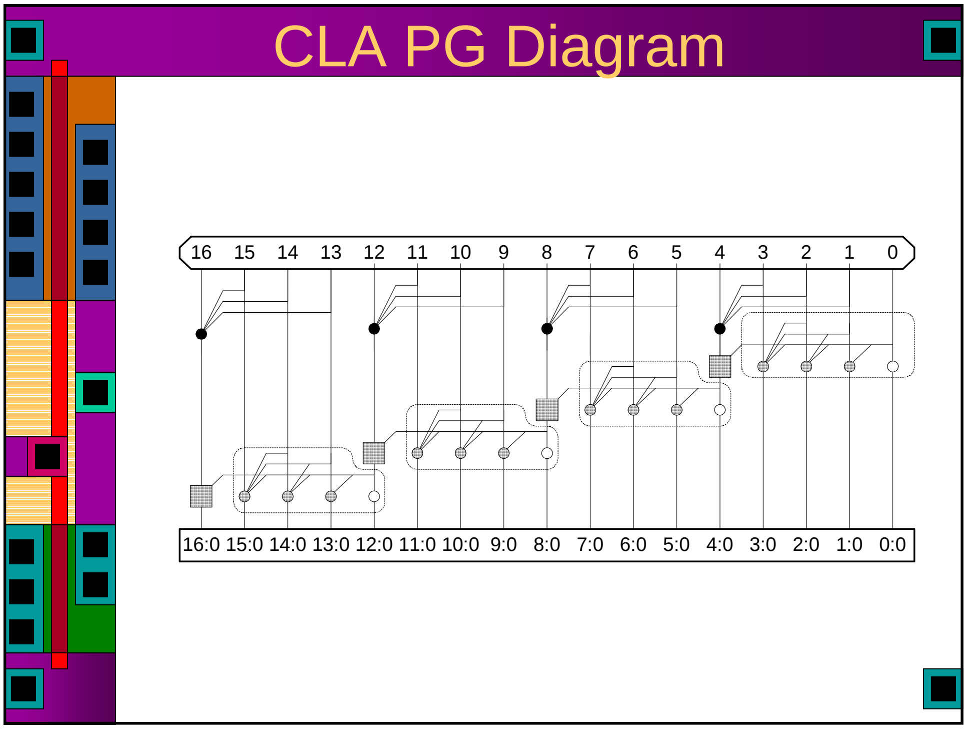
Ripple Carry

$$t_{\text{ripple}} = t_{pg} + (N-1)t_{AO} + t_{\text{xor}}$$

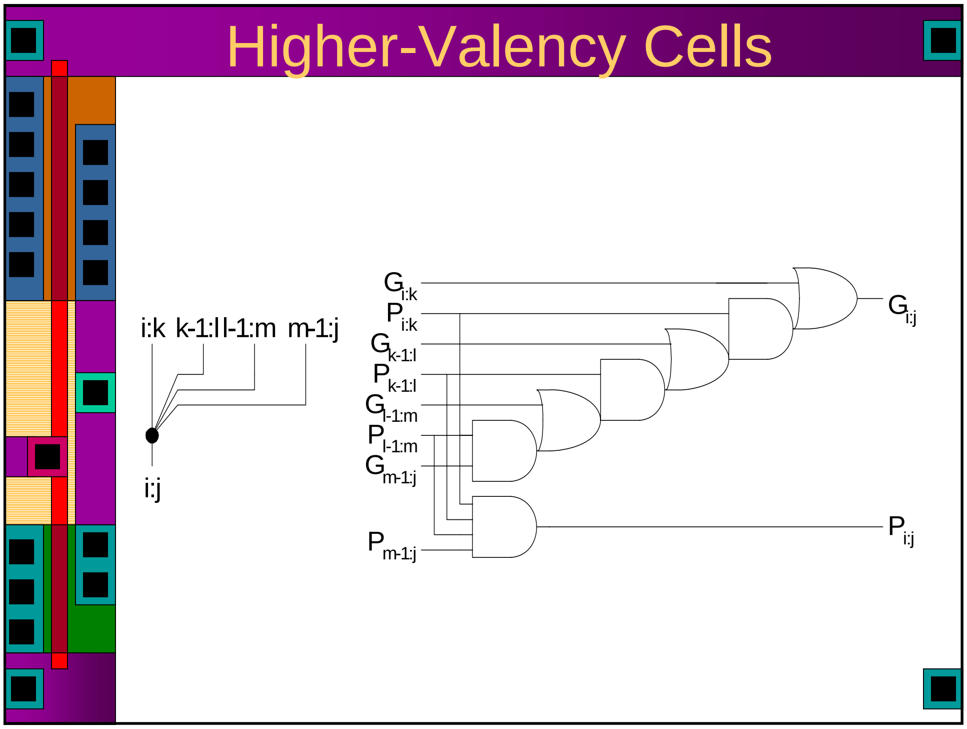


Carry-Lookahead Adder

- Carry-lookahead adder computes $G_{i:0}$ for many bits in parallel.
- Uses higher-valency cells with more than two inputs.



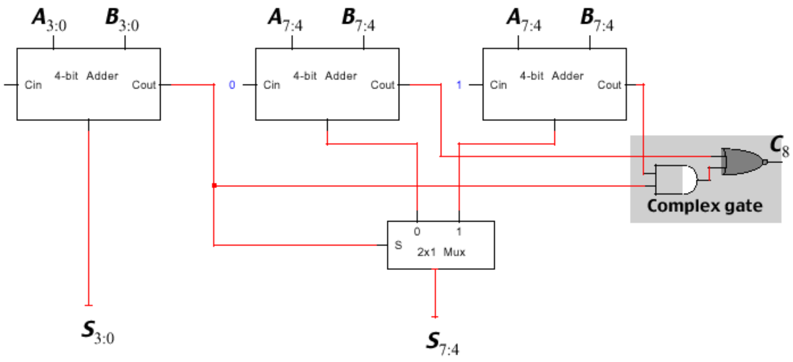
CLA PG Diagram



Higher-Valency Cells

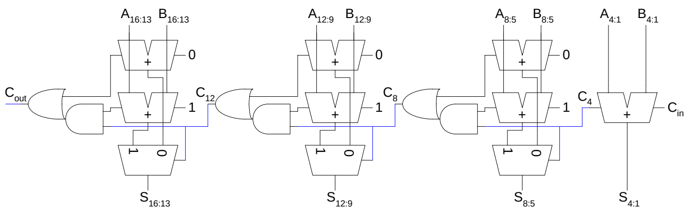
Carry-Select Adder

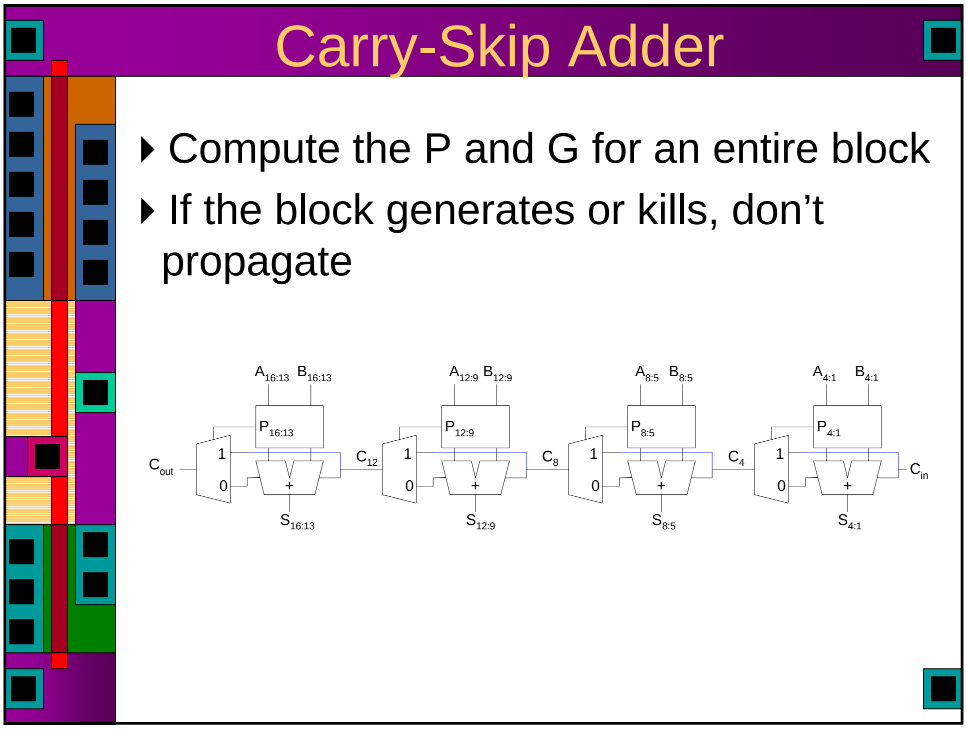
- Carry-Select
  - Compute result for a block based on carry-in of 1 and carry-in of 0, then select the right one



Carry-Select Adder

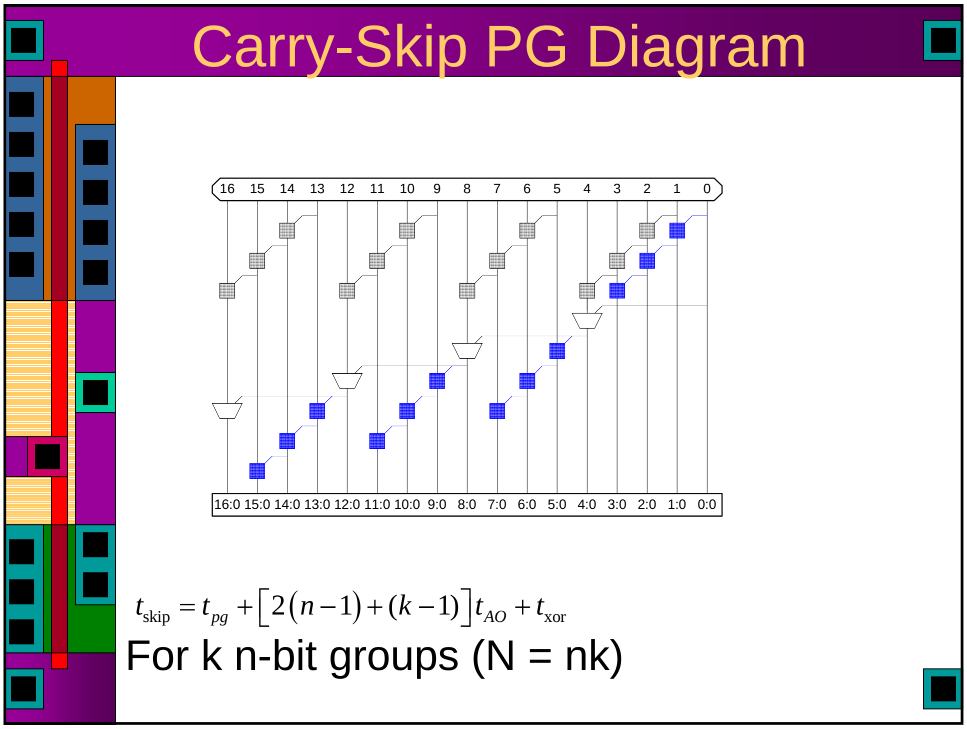
- Trick for critical paths dependent on late input X
  - Precompute two possible outputs for X = 0, 1
  - Select proper output when X arrives
- Carry-select adder precomputes n-bit sums
  - For both possible carries into n-bit group





Carry-Skip Adder

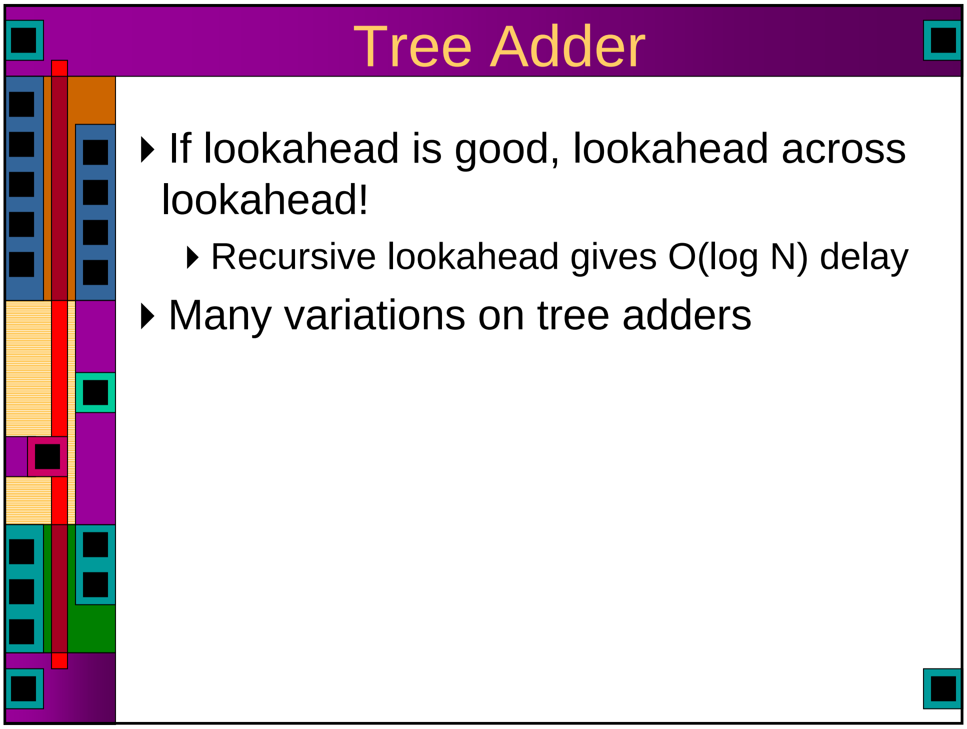
- Compute the P and G for an entire block
- If the block generates or kills, don't propagate



Carry-Skip PG Diagram

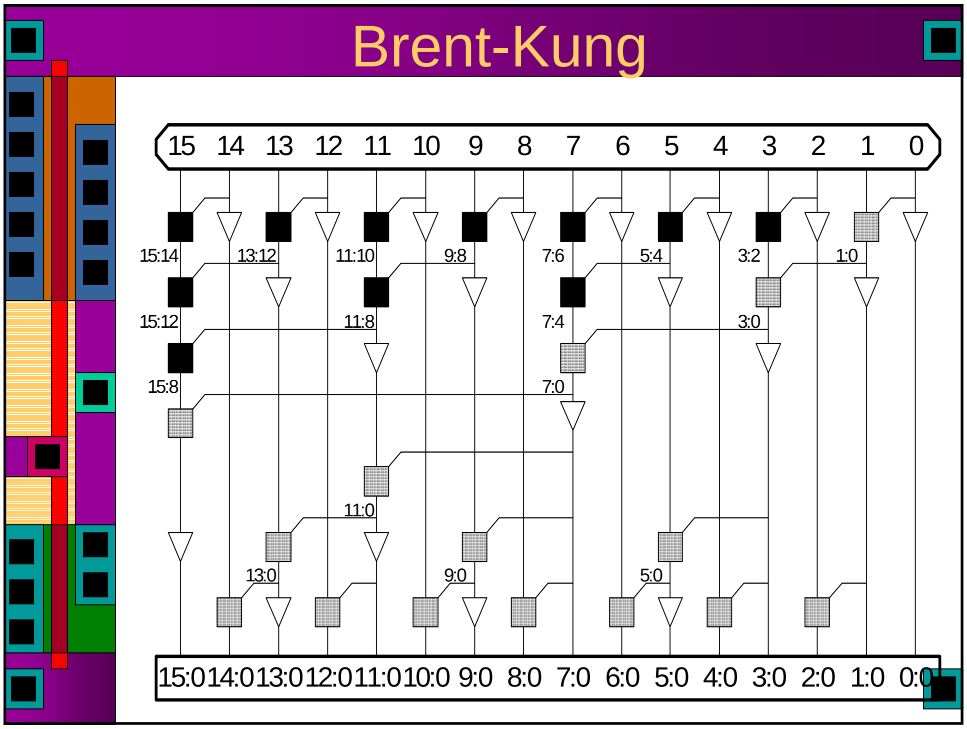
$$t_{\text{skip}} = t_{pg} + \left[2\left(n-1\right)+(k-1)\right]t_{AO} + t_{\text{xor}}$$

For k n-bit groups (N = nk)

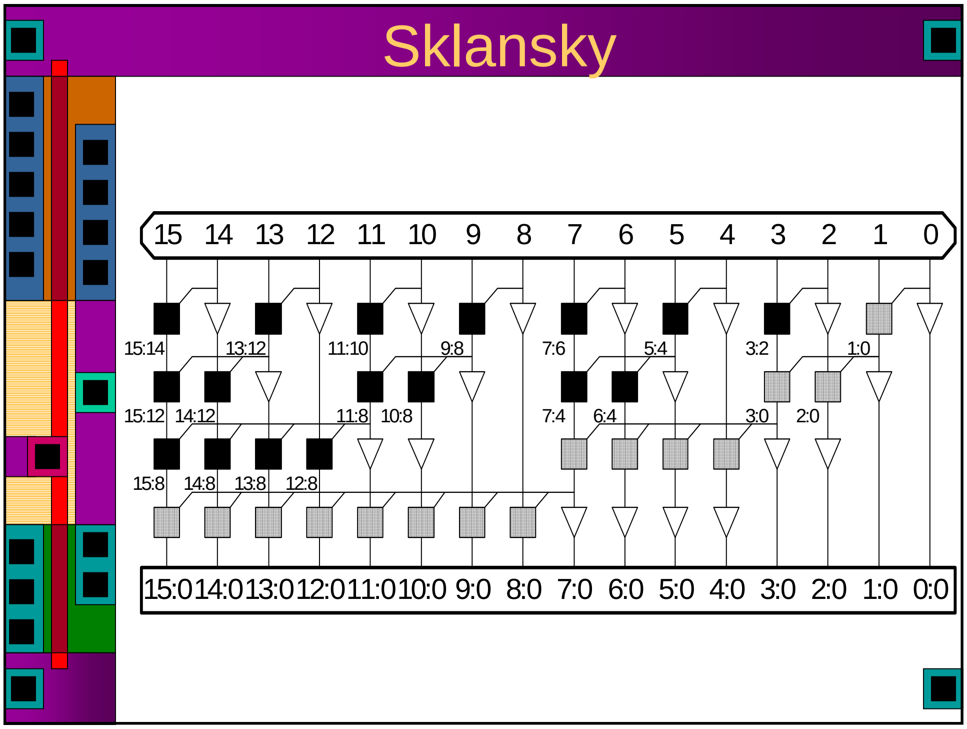


Tree Adder

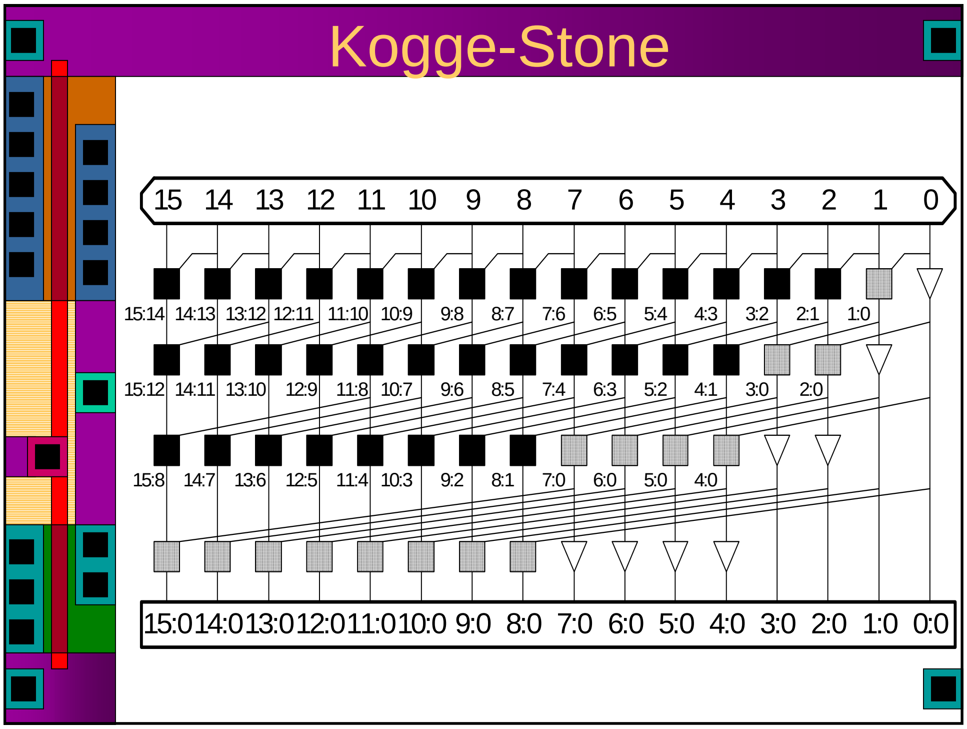
- If lookahead is good, lookahead across lookahead!
  - Recursive lookahead gives O(log N) delay
- Many variations on tree adders



Brent-Kung



Sklansky

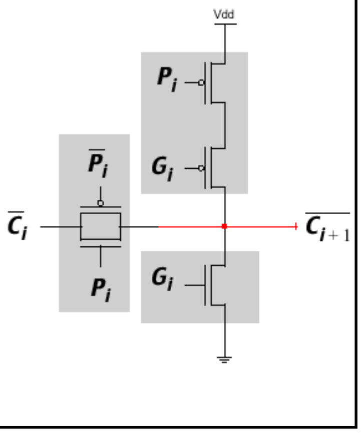


Kogge-Stone

Manchester Carry Chain

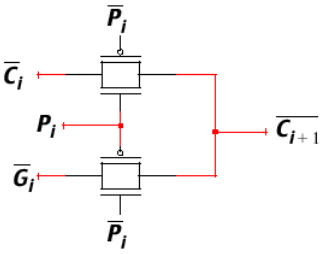
- Instead of changing the architecture of the adder, use a clever circuit to ripple the carry more effectively

- Propagate and generate signals computed in about two gate delays
- Active low carry is propagated through a chain of transmission gates
- The three shaded areas of the circuit are mutually exclusive, and represent P_i, G_i, and $\overline{P_i}\overline{G_i}$.



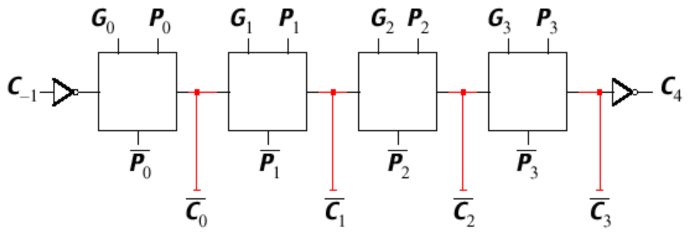
Alternate Implementation

- The Manchester carry chain computation may also be implemented with a 2x1 mux.



Four Bit Block

- Signal propagation through a chain of transmission gates must be restored after about 4 gates



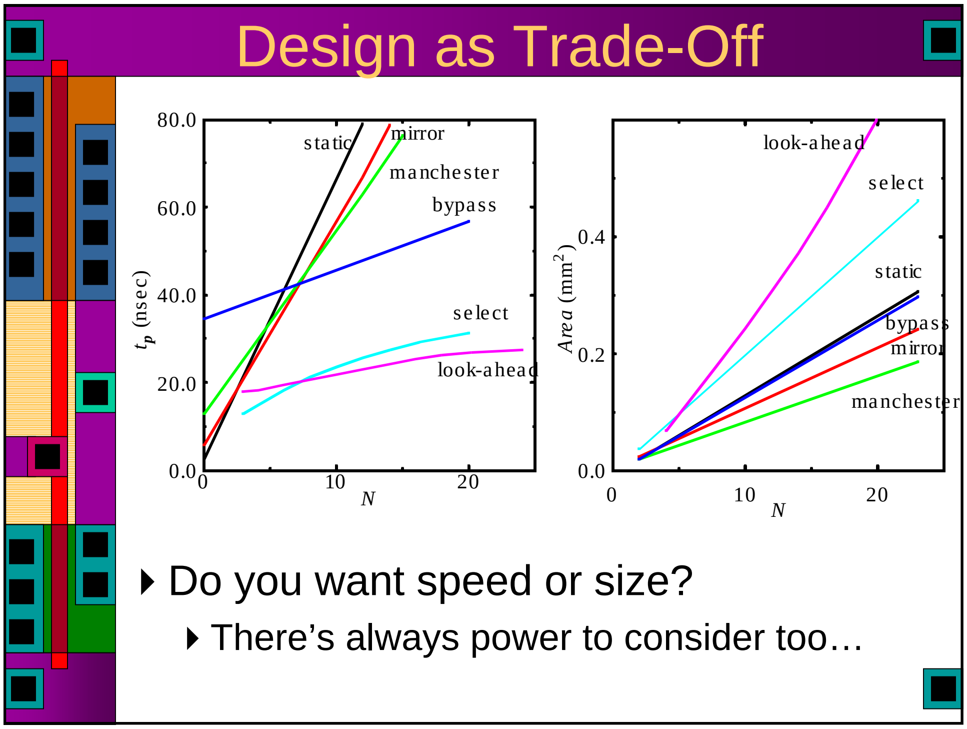
- A block propagate (bypass) circuit may be added to further improve performance on wide adders

Summary

Adder architectures offer area / power / delay tradeoffs.

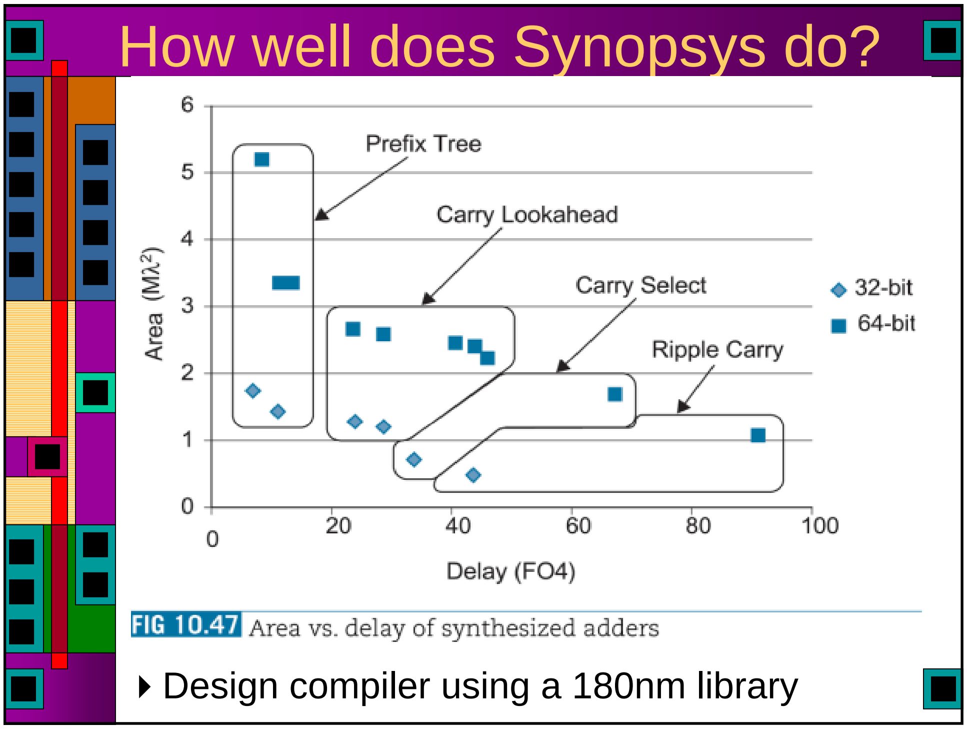
Choose the best one for your application.

Architecture	**Classification**	**Logic Levels**	**Max Fanout**	**Tracks**	**Cells**
Carry-Ripple		N-1	1	1	N
Carry-Skip n=4		N/4 + 5	2	1	1.25N
Carry-Inc. n=4		N/4 + 2	4	1	2N
Brent-Kung	(L-1, 0, 0)	$2\log_2 N – 1$	2	1	2N
Sklansky	(0, L-1, 0)	$\log_2 N$	N/2 + 1	1	0.5 $N\log_2 N$
Kogge-Stone	(0, 0, L-1)	$\log_2 N$	2	N/2	$N\log_2 N$



Design as Trade-Off

- Do you want speed or size?
  - There’s always power to consider too…



How well does Synopsys do?

FIG 10.47 Area vs. delay of synthesized adders

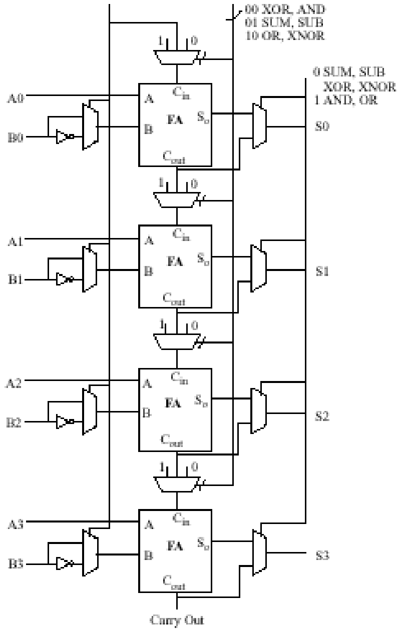
- Design compiler using a 180nm library

What should you use?

- Ripple if timing allows
  - Compact, easy
- CLA or carry-skip work well for 8-16 bits
- For 32, and especially 64 bits tree adders are faster
- Adders designed and tiled by hand will be much smaller (and probably faster) than synthesized adders

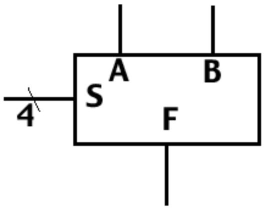
Logic Functions

- Use the features of the full adder cell to generate logic functions
- Lots of other ideas in your text…



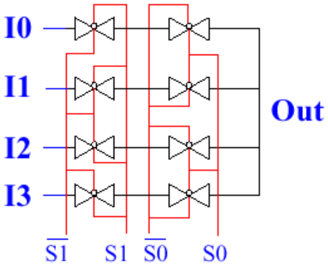
General Logic Generator

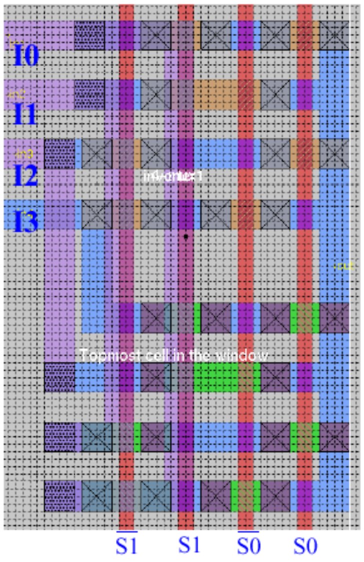
- 4x1 multiplexor can implement any function of two variables



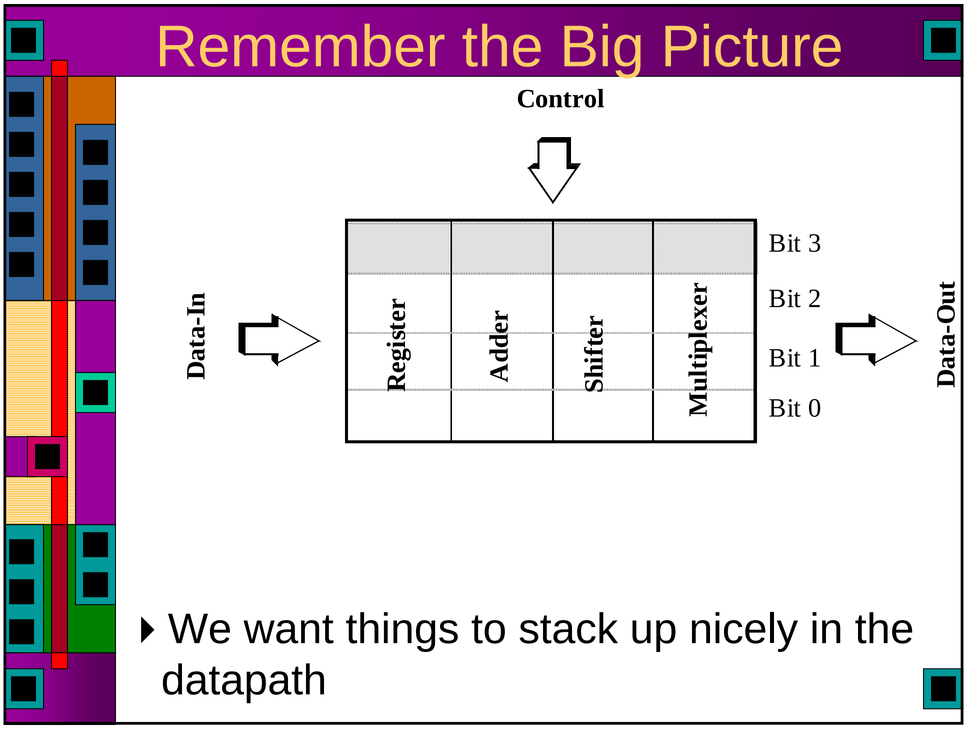
- Simply place the truth table for F on the inputs of the mux.
- The operands A and B will select the correct value of the function

One Possible MUX Version



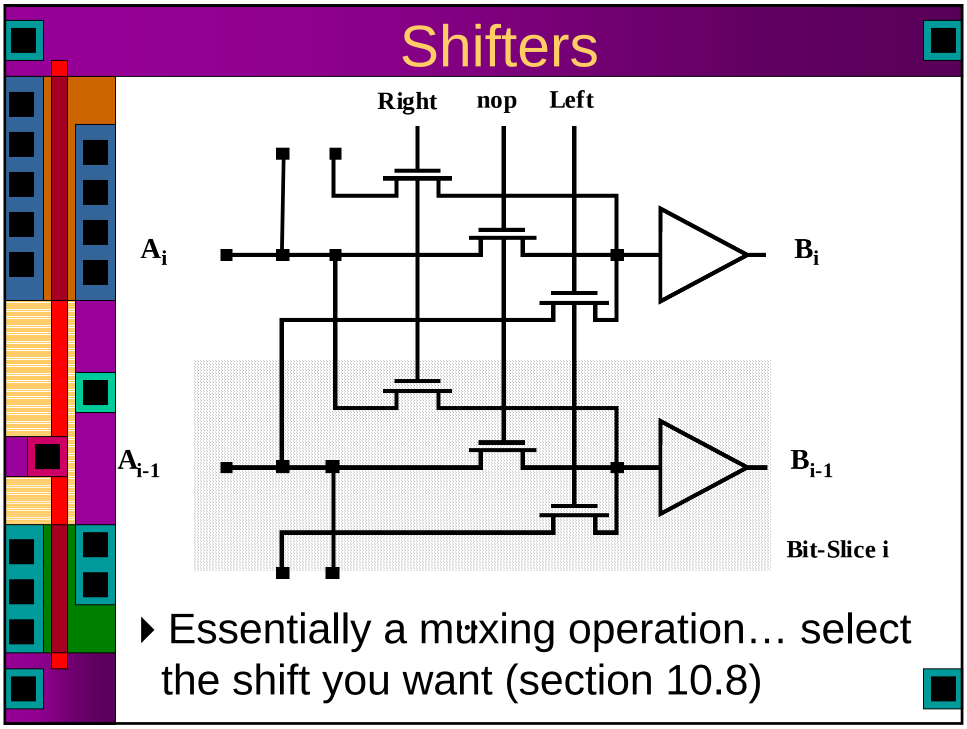


- Note: Two t-gates in series do not need the internal connection between p-fet and n-fet



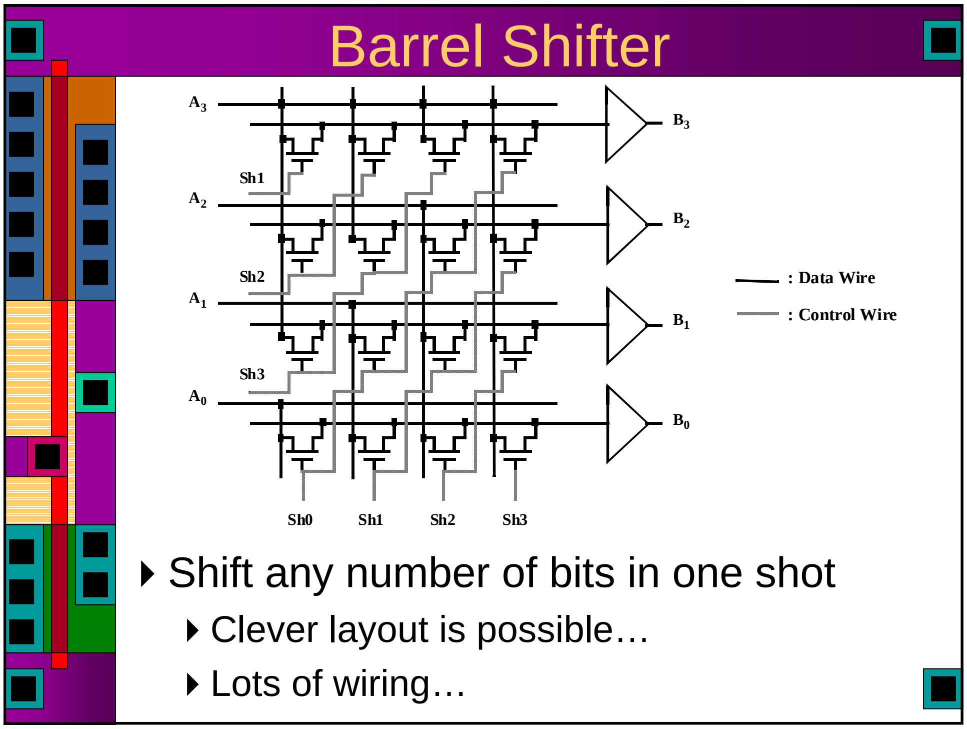
Remember the Big Picture

- We want things to stack up nicely in the datapath



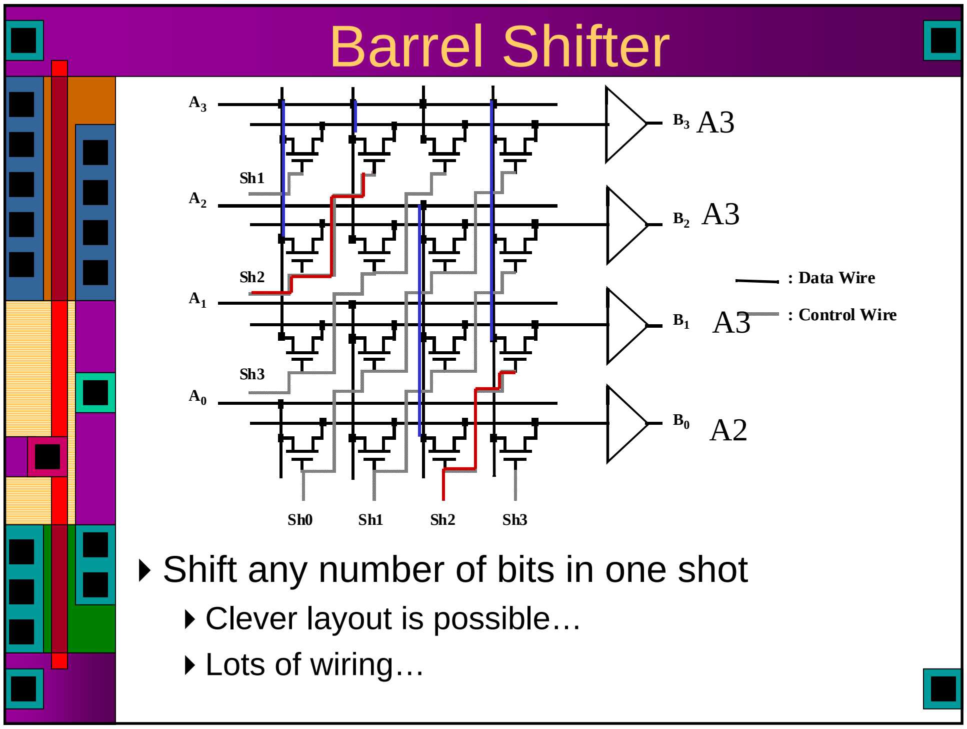
Shifters

- Essentially a muxing operation… select the shift you want (section 10.8)



Barrel Shifter

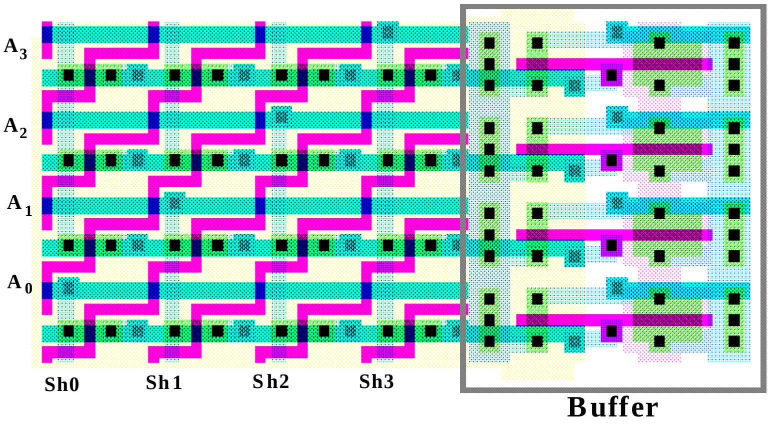
- Shift any number of bits in one shot
  - Clever layout is possible…
  - Lots of wiring…



Barrel Shifter

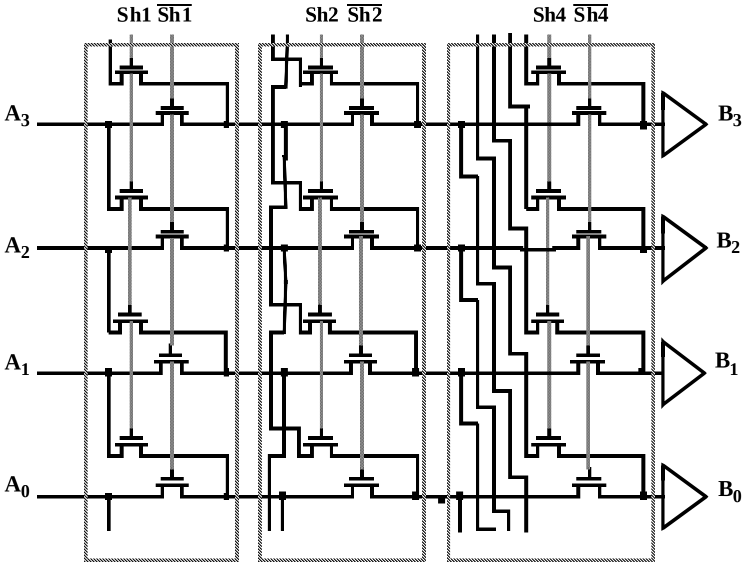
- Shift any number of bits in one shot
  - Clever layout is possible…
  - Lots of wiring…

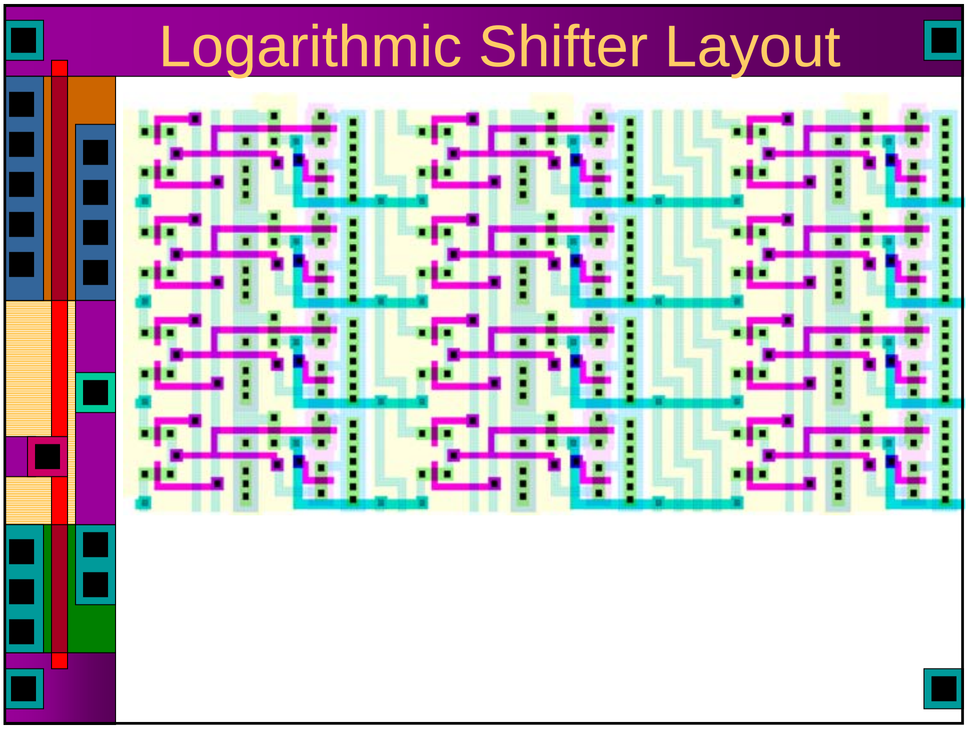
Four by Four Barrel Shifter



- Note the zig-zag control wire in poly

Logarithmic Shifter





Logarithmic Shifter Layout