
Lecture 5

More Adders & Multipliers

Computer Systems Laboratory
Stanford University
horowitz@stanford.edu

Copyright © 2007 Mark Horowitz

Overview

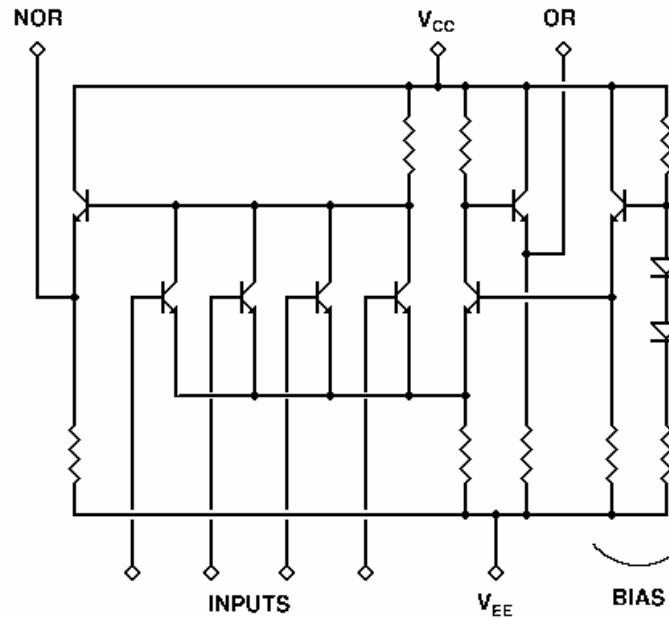
- Readings
 - Stojanovic (for next lecture on latches/flops)
Comparison of Latches and Flops
Also Chapt 11 in Chandrakasan
 - Harris Skew Tolerant Domino
(Won't discuss until later)
- Today's topics
 - Ling Adders
 - Multiplication
 - Booth recoding
 - CSA
 - Tree combiners

Ling Adder

- Huey Ling (IBM, 1981) reformulated Pg and Gg for speed
- The problem: Want to minimize logic delays for a 64b add
 - Start with radix-4 for only three levels of PG logic
 - Generate $P_{3:0}$, $G_{3:0}$ from inputs to save a stage
 - Uh-oh: that's a pretty complicated gate
- The normal equations for $P_{3:0}$ and $G_{3:0}$ are:
 - $G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0))$
 - $P_{3:0} = P_3P_2P_1P_0$
- Left as an exercise to the reader ☺
 - Generating $G_{3:0}$ from $A[3:0]$, $B[3:0]$, C_{in} takes 15 terms, stack=5

+Ling And ECL Logic

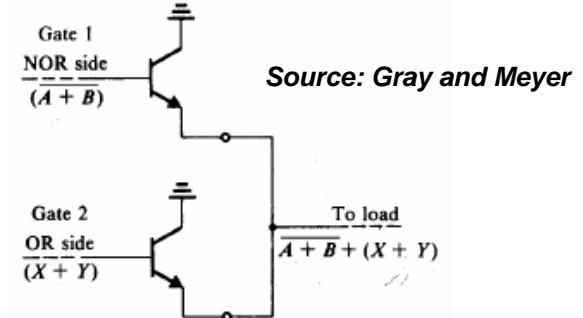
- Ling exploited the then-prevalent design style of ECL
 - Emitter-coupled logic – a very fast current steering bipolar style
 - $V_{CC} = 0V$, $V_{EE} < -1.7V$; here, inputs range from -0.9 to $-1.7V$
 - CMOS equivalent is called SCL (source coupled logic)
 - Gate operates with current steering



Source: Motorola MECL data sheet

+Benefits of ECL Logic

- ECL logic supports a “Wired-OR” configuration (or “Dot-OR”)
 - Two logic gates have outputs X and Y
 - Short their outputs together
 - If either output goes high
 - The result is pulled high – an OR function
- ECL gives a way to OR together complex logic “for free”
 - Ling used this to create moderately complex OR functions
- Is there an analogous circuit style in CMOS?
 - Domino precharge/discharge logic
 - Different pulldown stacks on the same node get “OR-ed”
 - Not exactly the same but close...

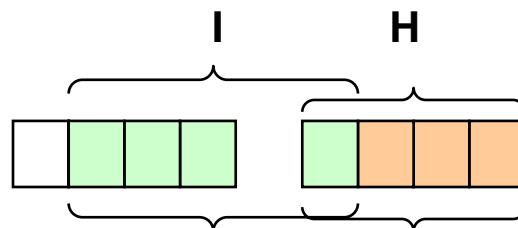


Simplifying G4 and P4

- Expand G4 term partially (but not all the way to A, B, C_{in})
 - $G_{3:0} = G_3 + P_3 * G_2 + P_3 * P_2 * G_1 + P_3 * P_2 * P_1 * G_0$
- Key observations: if P=A+B, then Gg=1 implies Pg=1
 - $G4 = P_3 * G_3 + P_3 * G_2 + P_3 * P_2 * G_1 + P_3 * P_2 * P_1 * G_0$
 - $G4 = P_3 * (G_3 + G_2 + P_2 * G_1 + P_2 * P_1 * G_0) = P_3 * H4$
 - Call this H4 a “pseudo-carry” term
- H4 is easier to compute than G4 is
 - Recall G4 takes 15 terms, stack of 5
$$H4 = A_3 B_3 + A_2 B_2 + A_2 A_1 B_1 + B_2 A_1 B_1 + A_2 A_1 A_0 B_0 + A_2 B_1 A_0 B_0 + B_2 A_1 A_0 B_0 + B_2 B_1 A_0 B_0$$
 - H4 takes only 8 terms, fanin of 4
 - A significant speed win

What Good Is H4?

- Rewrite: $H4 = G_3 + G_2 + P_2^*G_1 + P_2^*P_1^*G_0$
- Can I make a tree structure with H terms?
 - Good: my current group of four doesn't use P_3 , so why bother?
 - Bad: the next group of four does need P_3 ...
- So define a “pseudo-propagate” term $I4$
 - $I_{3:0} = P_2P_1P_0P_{-1}$ or $I_{7:4} = P_6P_5P_4P_3$ and so on (what's P_{-1} ?)
 - In general $I_{i:j} = P_{i-1:j-1}$



Using H and I

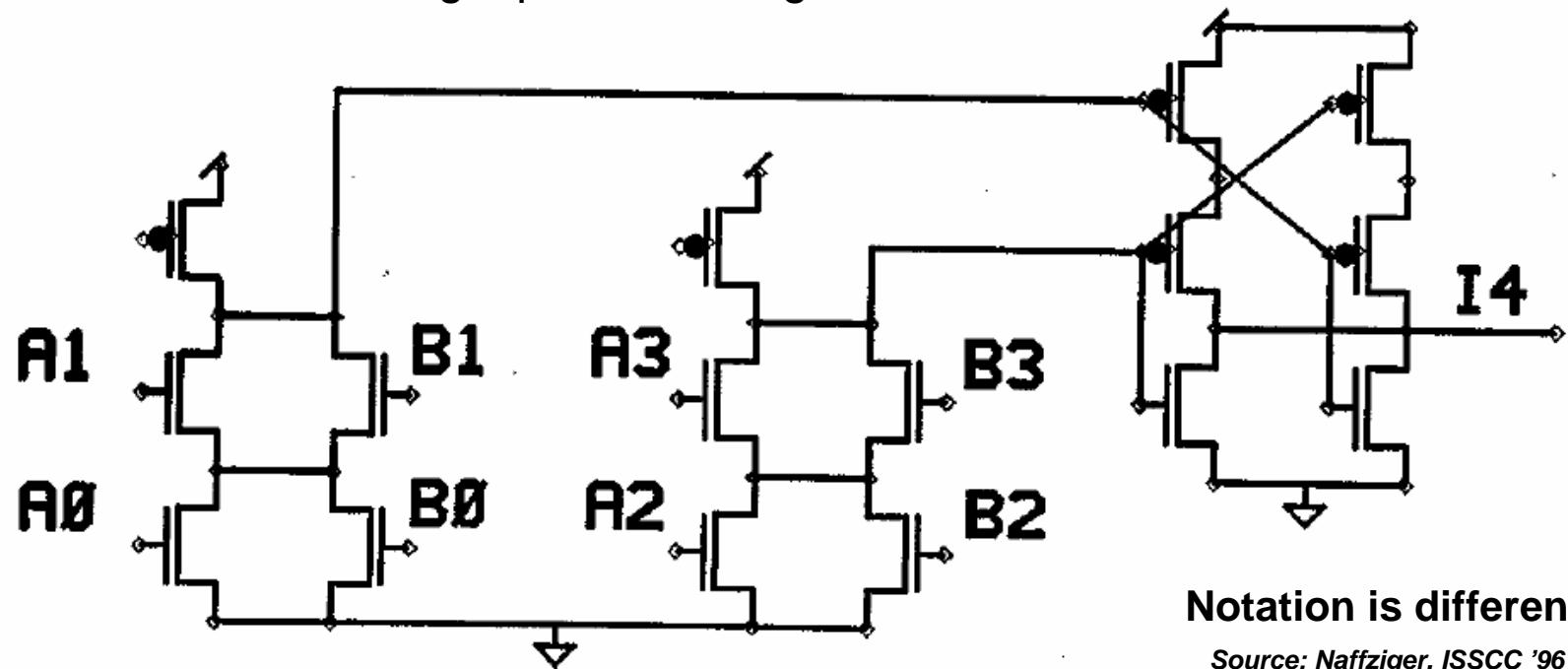
- They let us use the same tree structure as before (“off by one”)
 - With Ps and Gs: $G_{i:j} = G_{i:k} + P_{i:k}G_{k-1:j}$ and $P_{i:j} = P_{i:k}P_{k-1:j}$
 - With Hs and Is: $H_{i:j} = H_{i:k} + I_{i:k}H_{k-1:j}$ and $I_{i:j} = I_{i:k}I_{k-1:j}$
- Normally this type of optimization would not matter much
 - Trick only works with P and G, and not Pg and Gg
 - This means you get savings only at the first level of tree
 - But adders are carefully optimized, and every bit helps
- Ultimately need to add the missing P back to generate Carry
 - Put C_{in} into Ig_0 (in the open slot for P_{-1})
 - When you generate C from H, I
 - $Cin_{i+1} = P_i (H_{i:0} + I_{i:0})$, not much slower than normal Carry
 - In carry select adders, P_i can be added to the local chains

Ling Adder Implementation

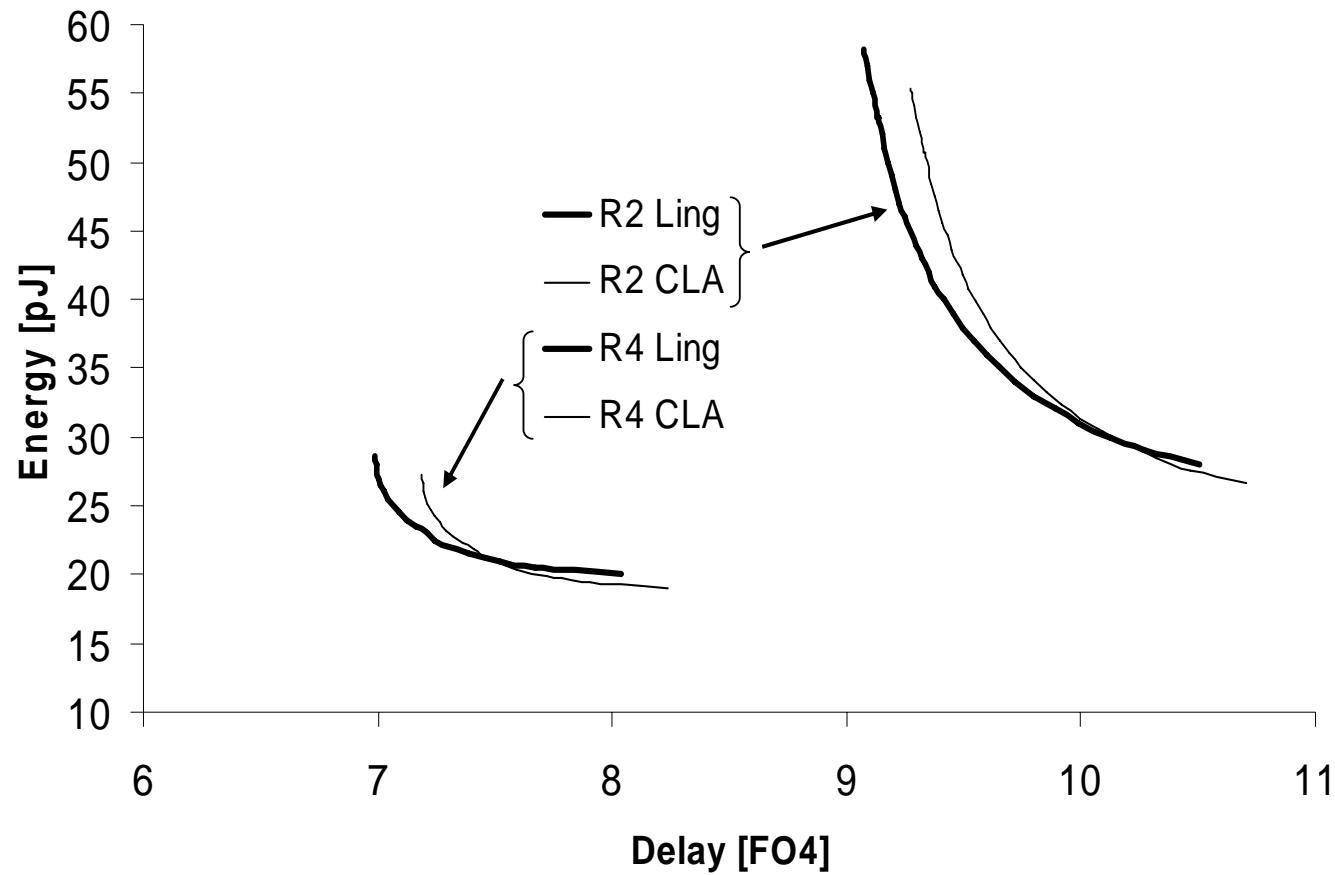
- Sam Naffziger (HP, 1996) presented a 64b adder
 - 7 FO4 delay (< 1nS): pretty darn fast
 - $0.5\mu\text{m}$ CMOS
 - This was a fairly optimized process ($\text{FO4} = 150\text{pS}$ at TTTT)
 - We'd usually expect 250pS at TTSS or 180pS at TTTT (360^*L_{gate})
 - Fairly small as well
 - 7000 transistors
 - $\frac{1}{4} \text{ mm}^2$
- In the homework you'll get to implement part of this adder
 - In Verilog, not spice
 - We'll give you skeleton Verilog and ask you to fill in the rest
 - Some errors in his slides (we'll detail them in the homework)

Aside – Domino Gate Factoring

- Domino gates have two stages
 - 2nd stage does not need to be an inverter
- Can build a 4 input AND gate by building two high stacks
 - And then using a pMOS NOR gate to combine



Ling vs. CLA



Source: Zlatanovici, ESSCIRC '03, and Bora Nikolic

Multiplication, Grade-School Level

- Product = Multiplicand * Multiplier
 - Multiplicand scaled by each digit in the multiplier → partial products
 - These partial products are shifted and added up

- Base-10 example: $119 * 182$
 - Partial products are: $119*2 = 238$, $119*8 = 952$; and $119*1 = 119$
 - Shift them and add them up

$$\begin{array}{r} \dots 238 \quad (2 * 119) \\ .952. \quad (8 * 119) \\ \underline{119\dots} \quad (1 * 119) \\ 21658 \end{array}$$

- This is perhaps easier to read in binary...

Multiplication, Grad-School Level

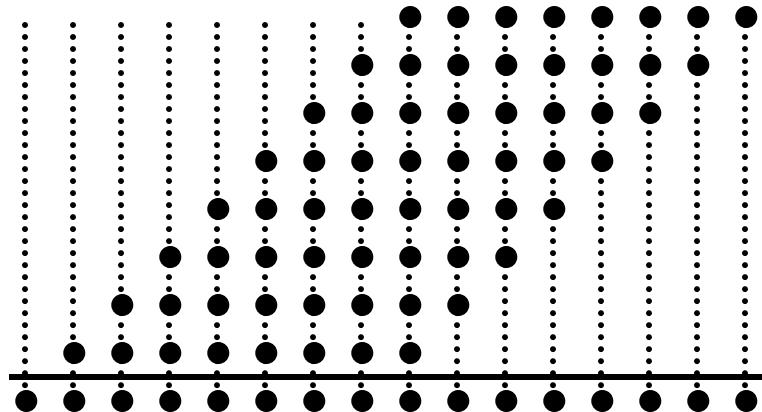
- Same basic idea, only now all digits are 0 or 1
 - But still have multiplicand, multiplier, and partial products
 - Ex: $119 = 01110111$; $182 = 10110110$

$$\begin{array}{r} \dots \dots \dots \dots \dots 1 0 1 1 0 1 1 0 \\ \dots \dots \dots \dots 1 0 1 1 0 1 1 0 . \\ \dots \dots \dots 1 0 1 1 0 1 1 0 . . \\ \dots \dots \dots 0 0 0 0 0 0 0 0 . . . \\ \dots \dots \dots 1 0 1 1 0 1 1 0 \\ \dots 1 0 1 1 0 1 1 0 \\ . 1 0 1 1 0 1 1 0 \\ \hline 0 0 0 0 0 0 0 0 \\ 0 1 0 1 0 1 0 0 1 0 0 1 1 0 1 0 = 21658_{10} \end{array}$$

- Hm. Is there an easier notation for this operation?

Dot Notation

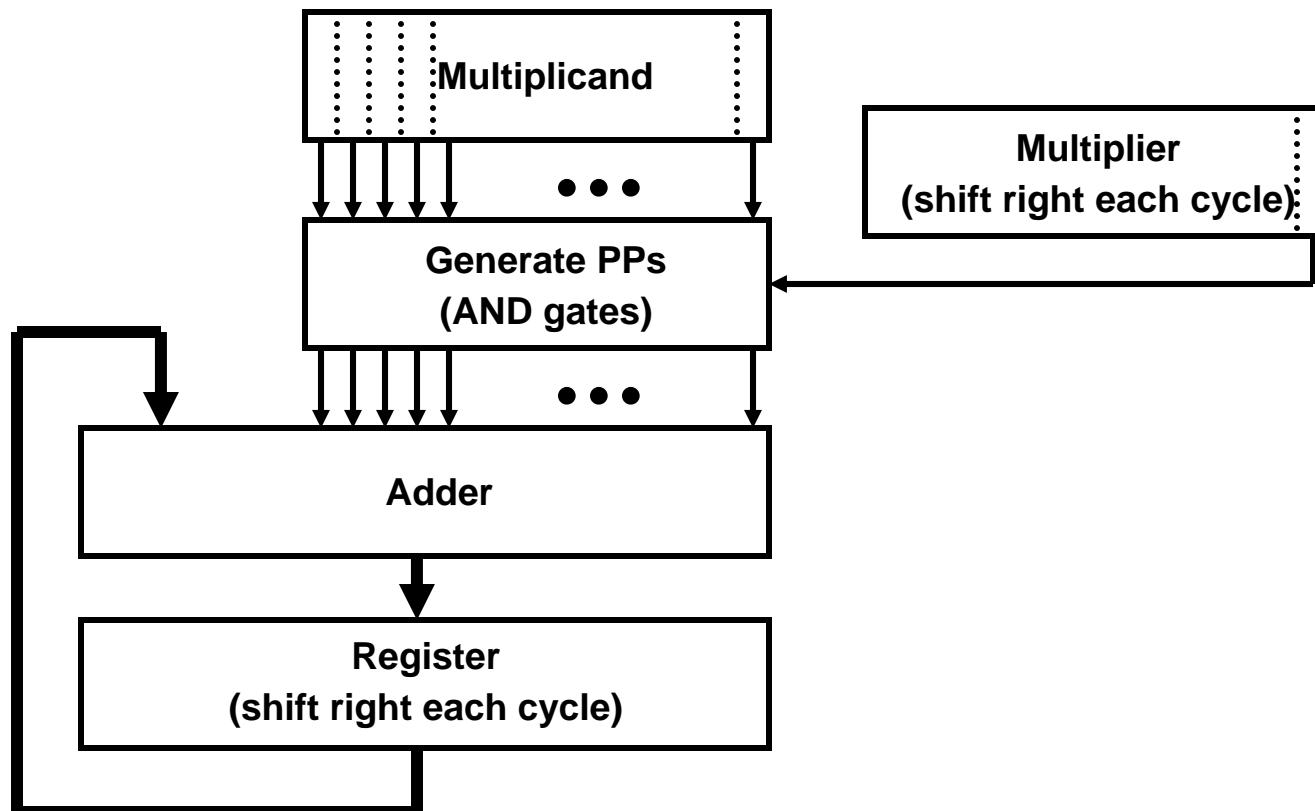
- Rows of dots are partial products, either a “1” or a “0”
 - Number of dots corresponds roughly to total hardware needed
 - Height of dot structure corresponds roughly to total latency



- Result of multiplying two n-bit numbers is a 2n-bit number
 - Integer operations keep the LSB n bits
 - Floating point operations keep the MSB n bits (toss out precision)

Simplest Multiplier

- A very simple multiplier iterates over n cycles
 - Smallest area (fewest dots), longest latency (maximum dot height)



Remove Unnecessary Partial Products

- Speed up the operation by avoiding adding partial products = 0
 - Unless multiplier = 111..1, there are always some 0 partial products
 - Just shift if multiplier bit is 0; don't bother adding the 0
 - In our example, from 8 to 6 partial products
- We can do better: consider a multiplier of 01111111
 - Requires seven partial products if we ignore the 0
 - Rewrite this as 10000000 – 00000001
 - Now I only need two partial products, although one is negative!
- Called “Booth encoding” (1951)
 - Skip strings of 1's in the multiplier
 - Encode as the difference of two numbers

Basic Booth Recoding

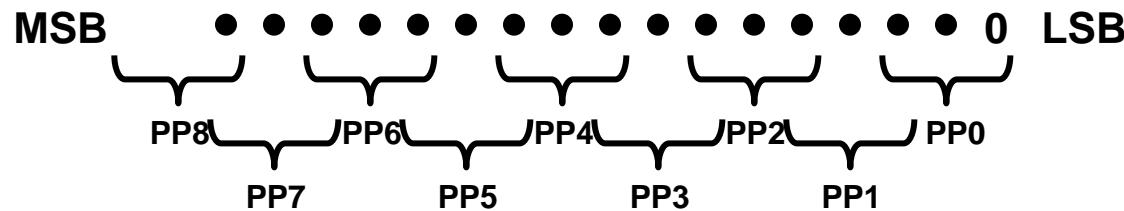
- Apply this to our example: $118 = 01110111$
 - Write 0111 as $1000 - 0001$; this string shows up twice

$$\begin{array}{r} - \dots \dots \dots \dots 1 0 1 1 0 1 1 0 \\ + \dots \dots 1 0 1 1 0 1 1 0 \dots \\ - \dots \dots 1 0 1 1 0 1 1 0 \dots \dots \\ + \underline{1 0 1 1 0 1 1 0 \dots \dots \dots} \\ 0 1 0 1 0 1 0 0 1 0 0 1 1 0 1 0 = 21658_{10} \end{array}$$

- This is an improvement; six partial products to four
- Not always helpful; imagine input of $170 = 10101010$
 - Recoding into differences of two numbers doesn't help at all
 - No string of 1's to exploit
- Problem: Variable #s of PPs are hard to support in hardware

Modified Booth Recoding

- Look at the multiplier three bits at a time
 - Try to figure out if we're starting, inside, or finishing a string of 1s
 - Overlap the three bits to help us figure this out
 - Really encoding just two bits at a time, but in context of three bits
- 16b multiplier always generates 9 partial products (PP0-PP8)
 - In general will create $\text{floor}(0.5*(n+2))$ partial products
 - Pad the LSB with a 0, and the MSBs with enough 0s



Modified Booth Recoding Rules

- Get different PPs depending on the rules (here, M=multiplicand)
 - If we're starting a string of 1's, put a $-M$ at string's LSB
 - If we're ending a string of 1's, put a $+M$ one left of string's MSB
 - If we're inside or outside a string, do nothing
 - Isolated 1's are treated as is

Bit1	Bit0	Prev	Output	Comment
0	0	0	0	Outside a string of 1's. Do nothing
0	0	1	$+M$	Ended a string of 1's. Put $+M$ at MSB+1
0	1	0	$+M$	Isolated 1; treat as is
0	1	1	$+2M$	Ended a string of 1's. Put $+M$ at MSB+1
1	0	0	$-2M$	Starting a string of 1's. Put $-M$ at LSB
1	0	1	$-M$	Start & end. Put $+M$ at MSB+1 and $-M$ at LSB
1	1	0	$-M$	Starting a string of 1's. Put $-M$ at LSB
1	1	1	0	Inside a string of 1's. Do nothing

- This needs $+M$, $-M$, $+2M$, and $-2M$
 - $\pm 2M$ are easy: just take $\pm M$ and shift it over a bit

Example of Modified Booth Recoding

- Recall our multiplier was $118 = 01110111$

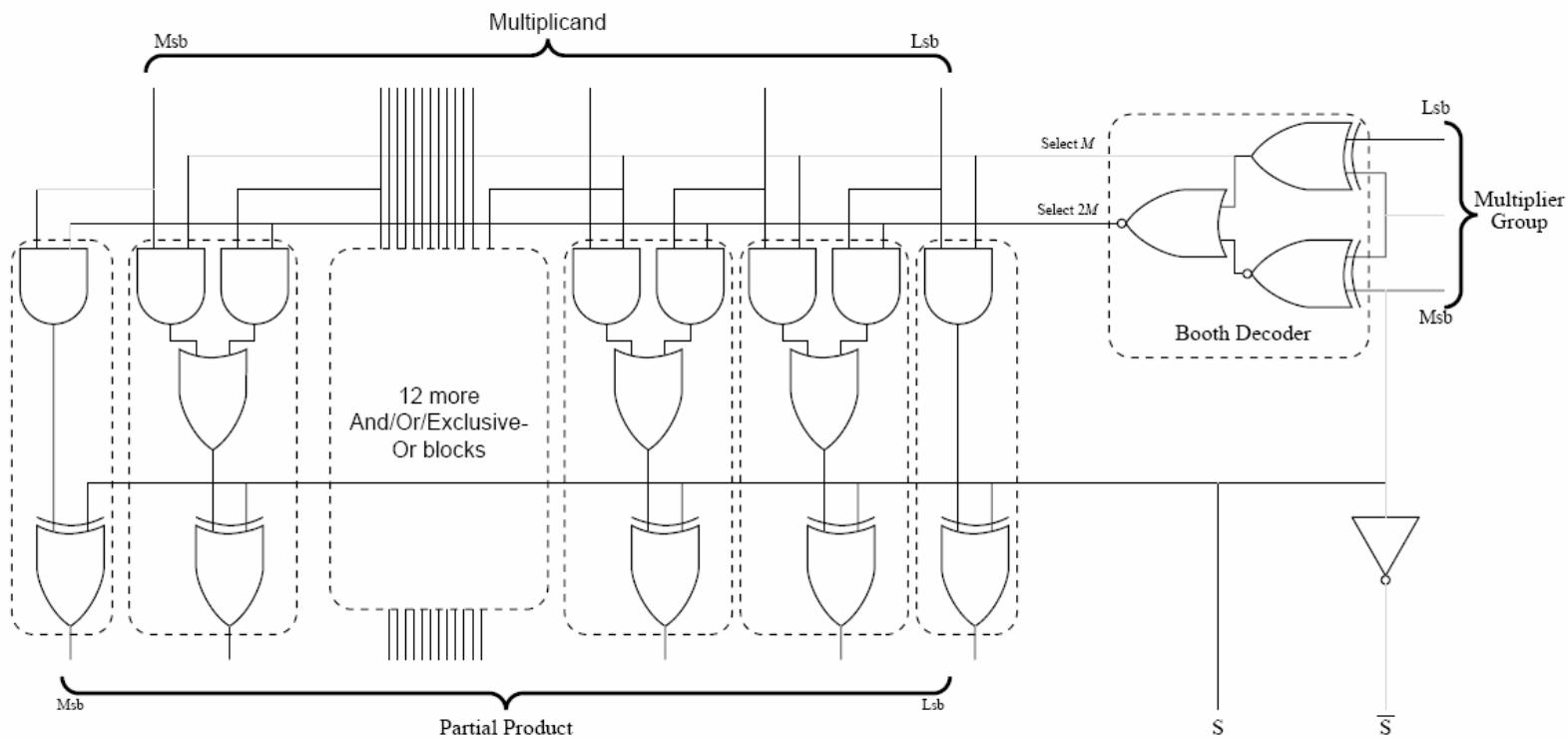
$$\begin{array}{cccccccccc} 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ & \brace{1} & \brace{1} & \brace{0} & \brace{1} & \brace{1} & \brace{1} & \brace{0} \\ & -M & & -M & & & & \\ +2M & & +2M & & & & & \end{array}$$

- Same as before; modified Booth = original Booth for this case
- Writing it out this time
 - Use two's complement notation for the negative numbers

$$\begin{array}{r} 1 1 1 1 1 1 1 0 1 0 0 1 0 1 0 \quad (-M) \\ \dots \dots 1 0 1 1 0 1 1 0 \dots \quad (2M) \\ 1 1 1 0 1 0 0 1 0 1 0 \dots \quad (-M) \\ 1 0 1 1 0 1 1 0 \dots \dots \quad (2M) \\ \hline 0 1 0 1 0 1 0 0 1 0 0 1 1 0 1 0 = 21658_{10} \end{array}$$

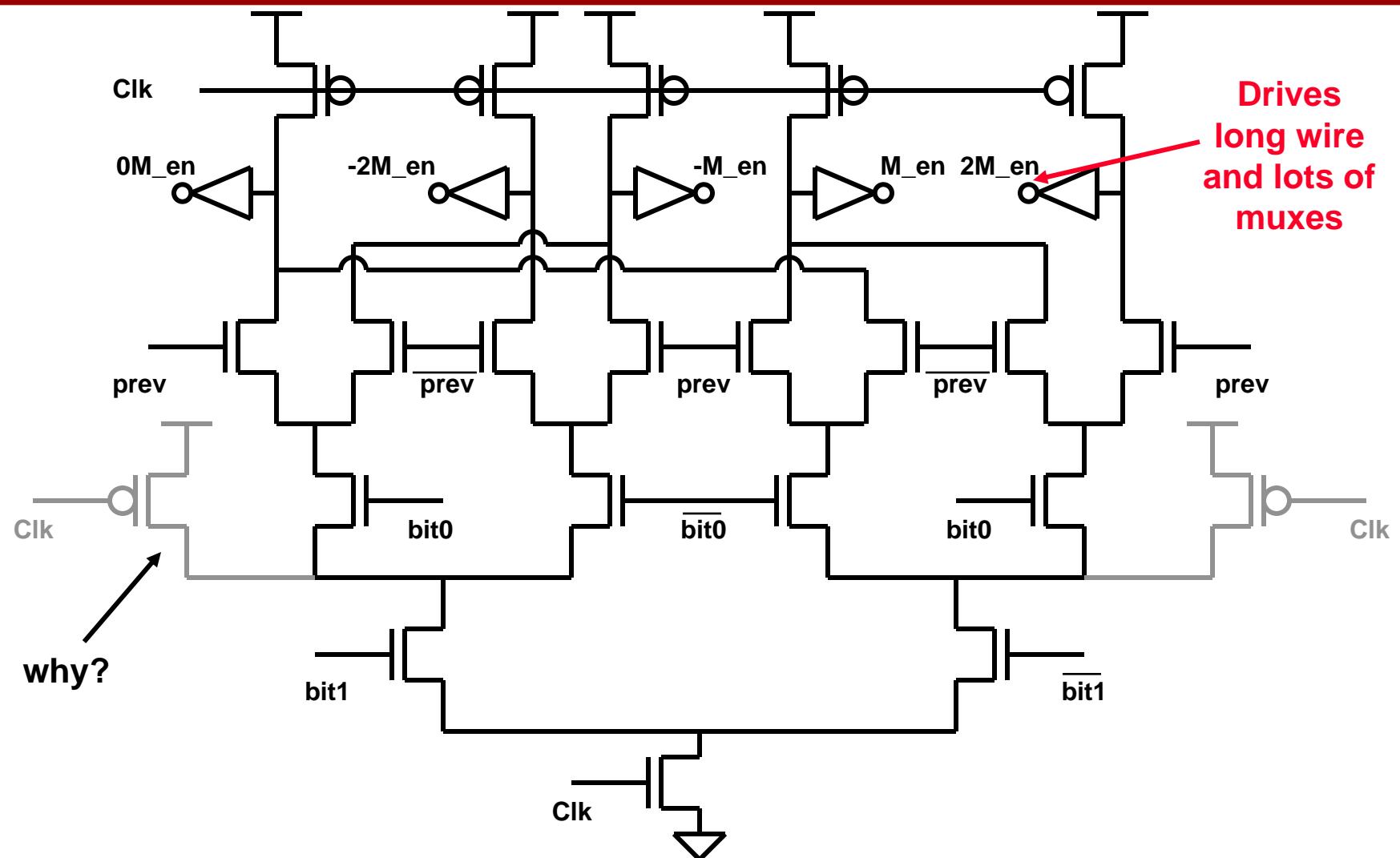
Modified Booth Recoding Circuits

- A plain-vanilla CMOS implementation
 - Booth decoder followed by 16 individual Booth muxes

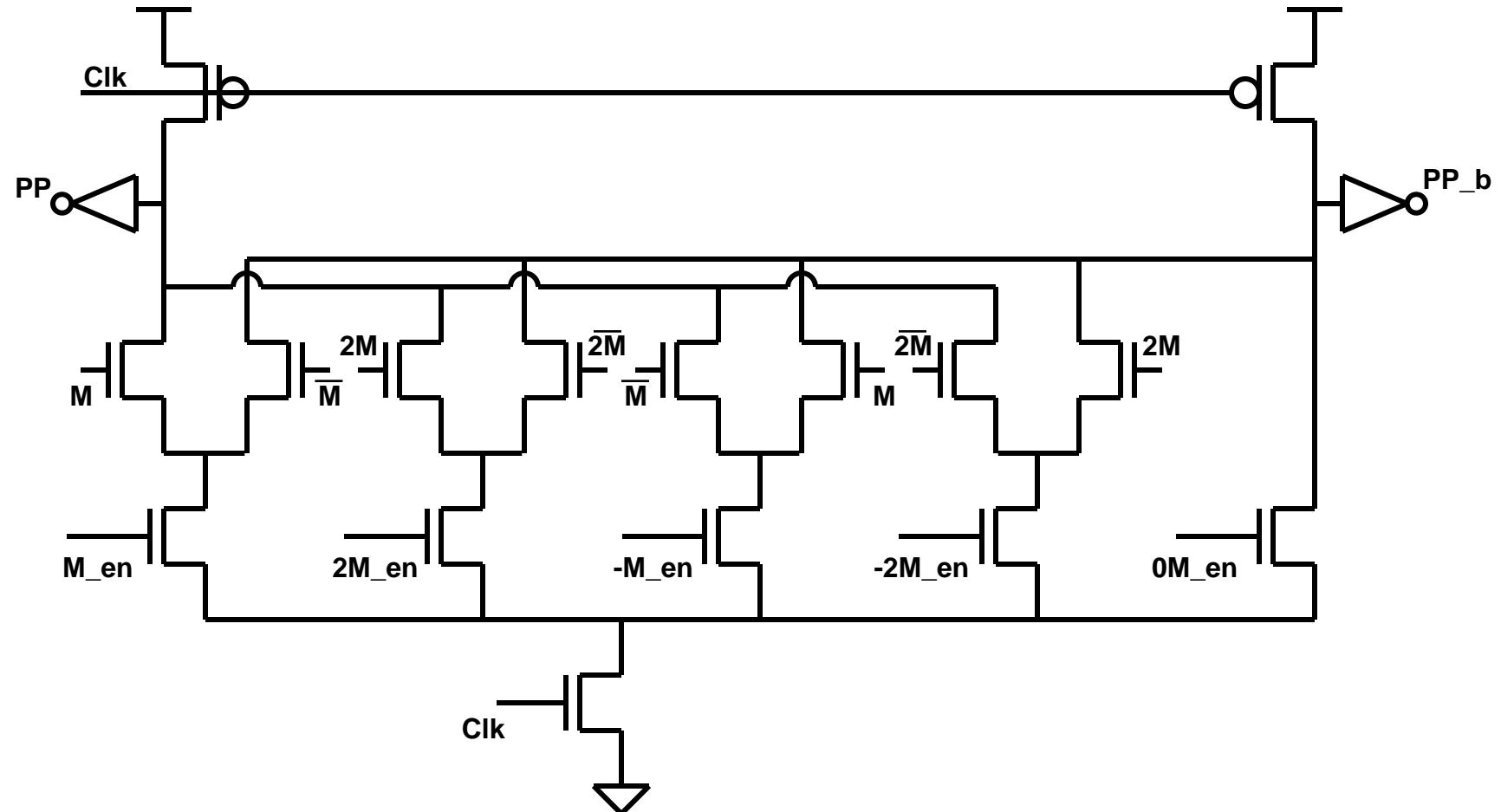


Source: Bewick, Stanford, 1994

Modified Booth Decoder in Domino



Modified Booth Mux in Domino



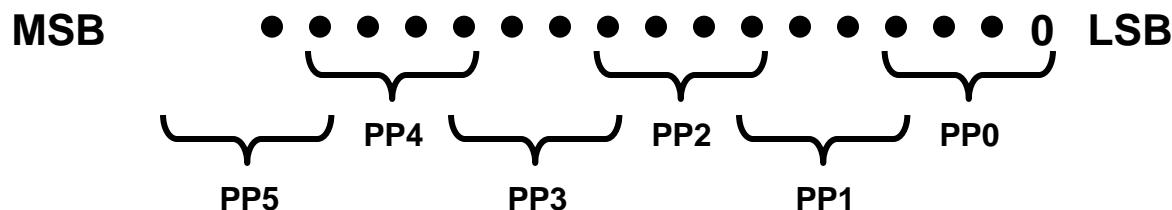
Can We Extend This Paradigm?

- Look at multiplier four bits at a time and hunt for strings of 1's
 - Recode three bits at a time, but using context of four bits

Bit2	Bit1	Bit0	Prev	Output	Comment	.
0	0	0	0	0	Outside a string of 1's. Do nothing	.
0	0	0	1	+M	Ended a string of 1's. Put +M at MSB+1	.
0	0	1	0	+M	Isolated 1; treat as is	.
0	0	1	1	+2M	Ended a string of 1's. Put +M at MSB+1	.
0	1	0	0	+2M	Isolated 1; like above but shifted	.
0	1	0	1	+3M	Isolated 1 plus an ending to string of 1's	.
0	1	1	0	+3M	Start&end: +M at MSB+1 and -M at LSB	.
0	1	1	1	+4M	Ended a string of 1's. Put +M at MSB+1	.
1	0	0	0	-4M	Starting a string of 1's. Put -M at LSB	.
1	0	0	1	-3M	End&start: +M at MSB+1 and -M at LSB	.
1	0	1	0	-3M	Isolated 1 plus a start to a string of 1's	.
1	0	1	1	-2M	End&start: +M at MSB+1 and -M at LSB	.
1	1	0	0	-2M	Starting a string of 1's. Put -M at LSB	.
1	1	0	1	-M	End&start: +M at MSB+1 and -M at LSB	.
1	1	1	0	-M	Starting a string of 1's. Put -M at LSB	.
1	1	1	1	0	Inside a string of 1's. Do nothing	.

Booth-3 Recoding

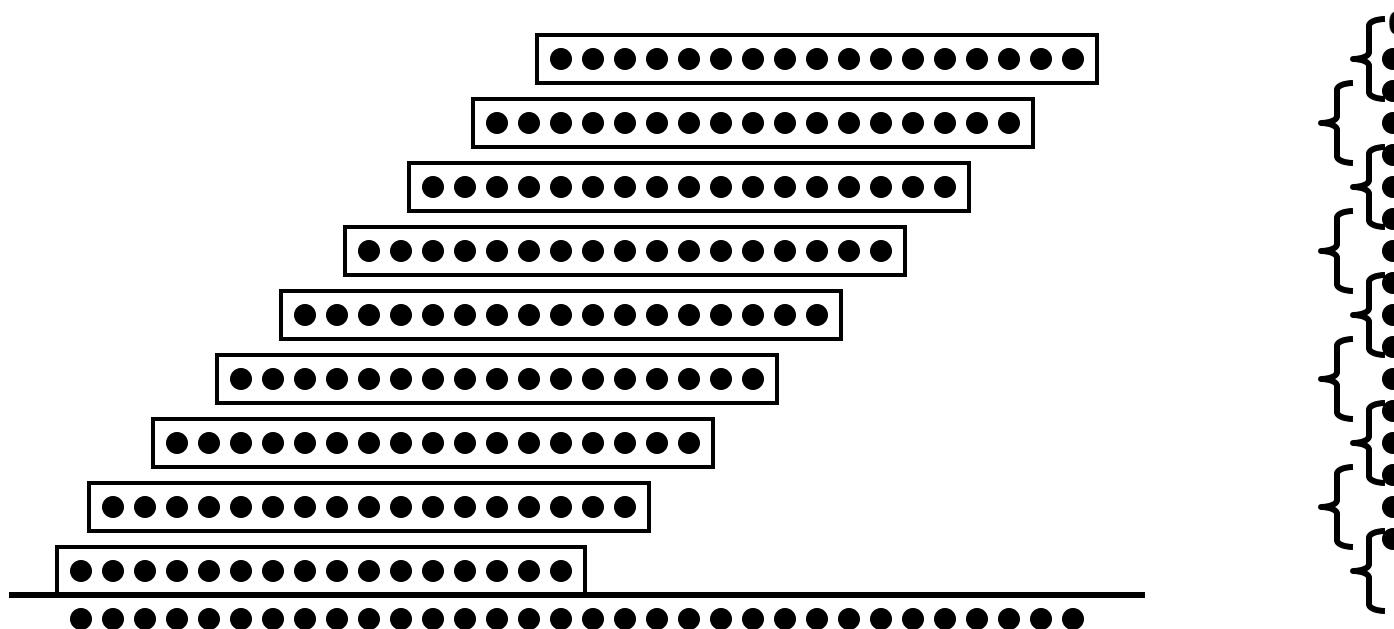
- Good part of this scheme: fewer partial products; faster



- Bad part of this scheme: Need to generate +/- 3M
 - Can take an additional add!
 - This is why Booth-3 is typically not used in designs
 - Higher-order Booth recoding gets worse
 - Booth-4 requires +/-3M, +/-5M, and +/-7M. Yikes.
- Clever tricks to get around this use “partially redundant forms”
 - Optional reading (Bewick) if you want to try this on your project

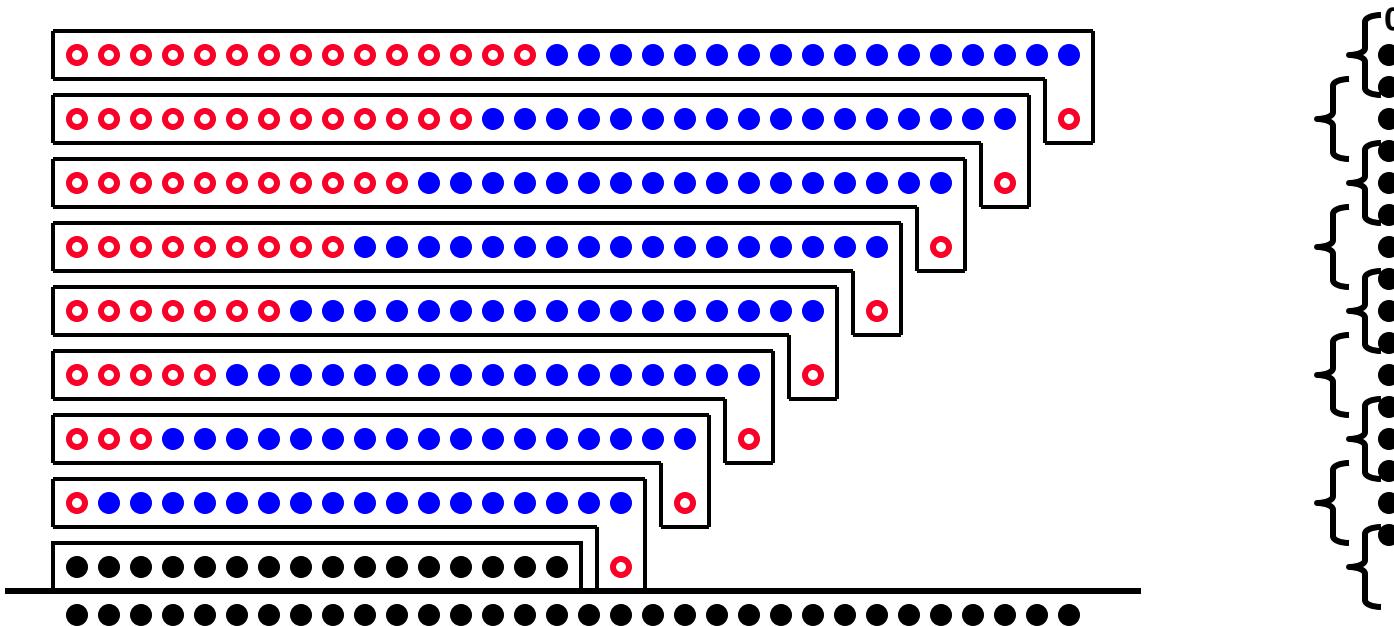
Negative Partial Products

- How do we deal with negative partial products?
- Consider a 16b multiplication using modified Booth recoding



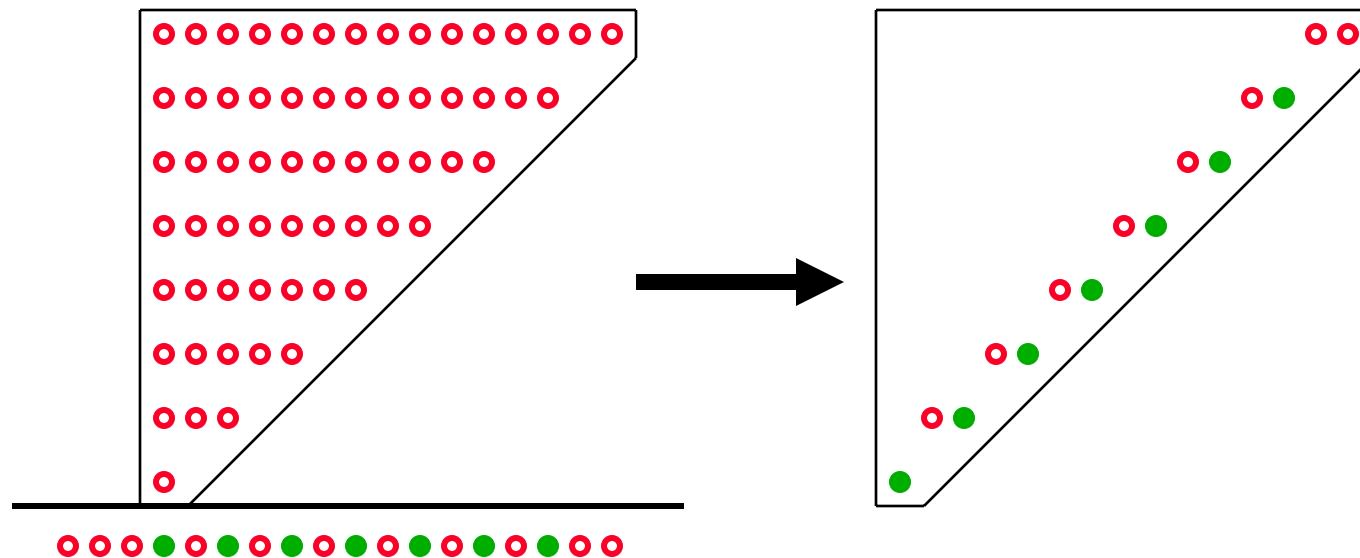
Add Sign Bits

- What if all the partial products were negative?
 - Invert all the bits (blue circles), add 1, and sign-extend
 - Notation: red circle = 1, green circle = 0
 - Note that last partial product is never negative



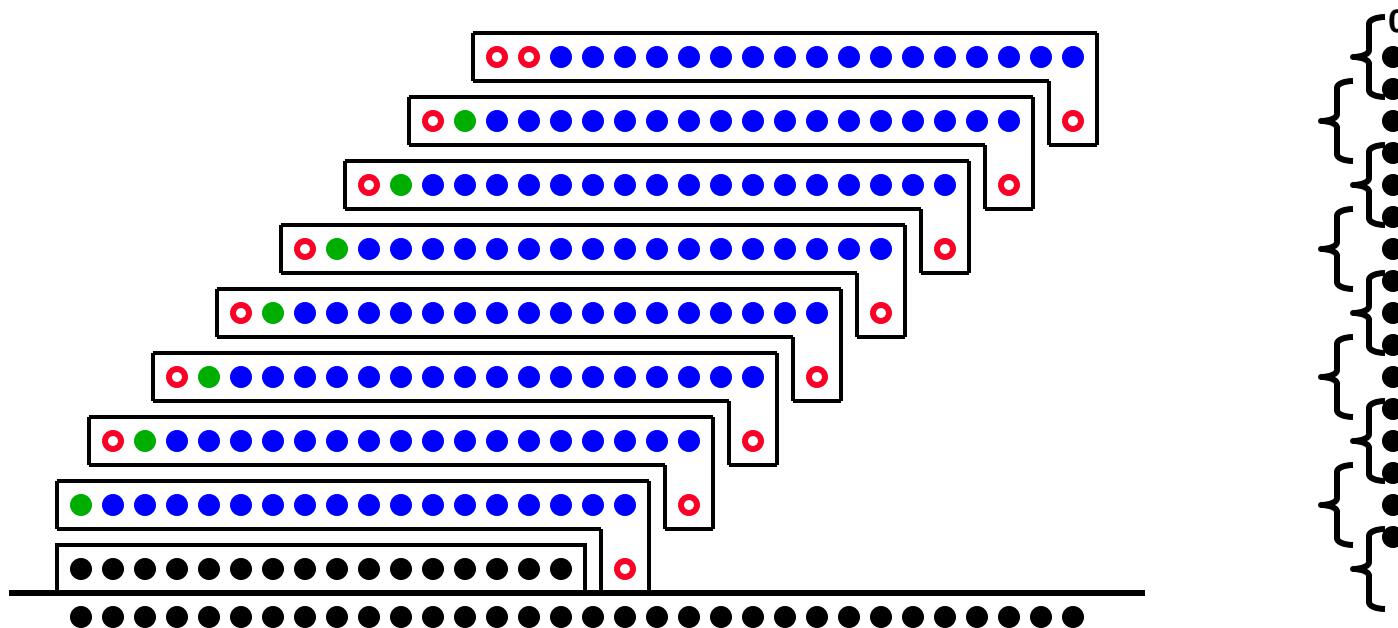
Dealing With Sign Extensions

- These red circles (all “1”s) are inconvenient
 - They make our multiplier unsquare – or at least, un-parallelepiped
 - Notation: red circle = 1, green circle = 0
- What do the 1’s add up to?



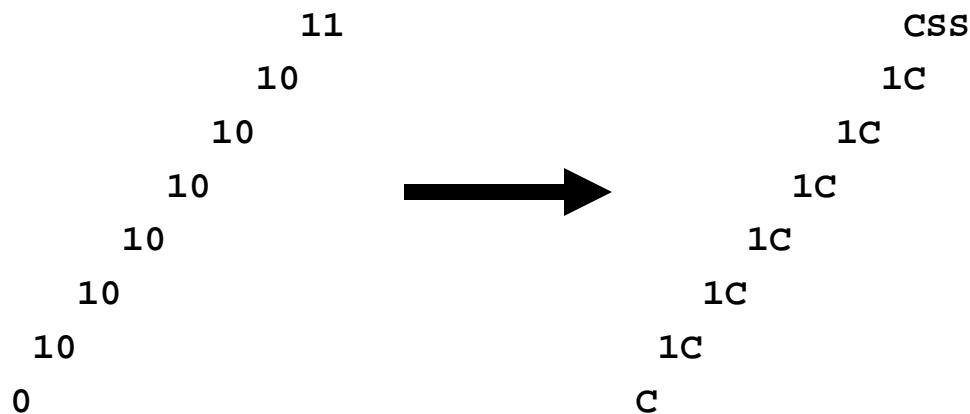
Reduce

- The red triangle (of 1s) can be reduced to a simpler form
 - Good thing, or else fanout would be huge
 - Notation: red circle = 1, green circle = 0



Sign Extension Constants

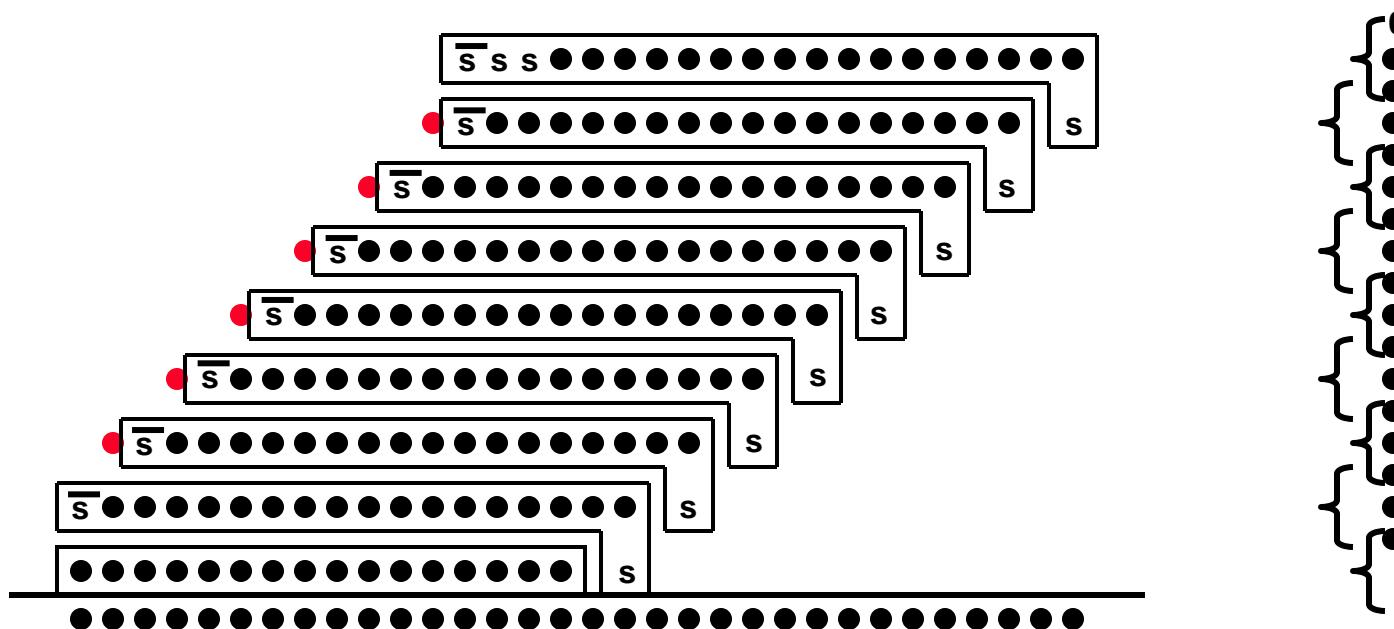
- Let's examine these extra sign extension bits more closely
 - $S = \text{sign bit} = 1$ if negative
 - Because fonts don't work well in Powerpoint, " C " = $S_{\bar{}}$



- Expression on the right is exactly the same as the left for $S=1$
 - And, it also works out for $S=0$ (all the terms drop out)

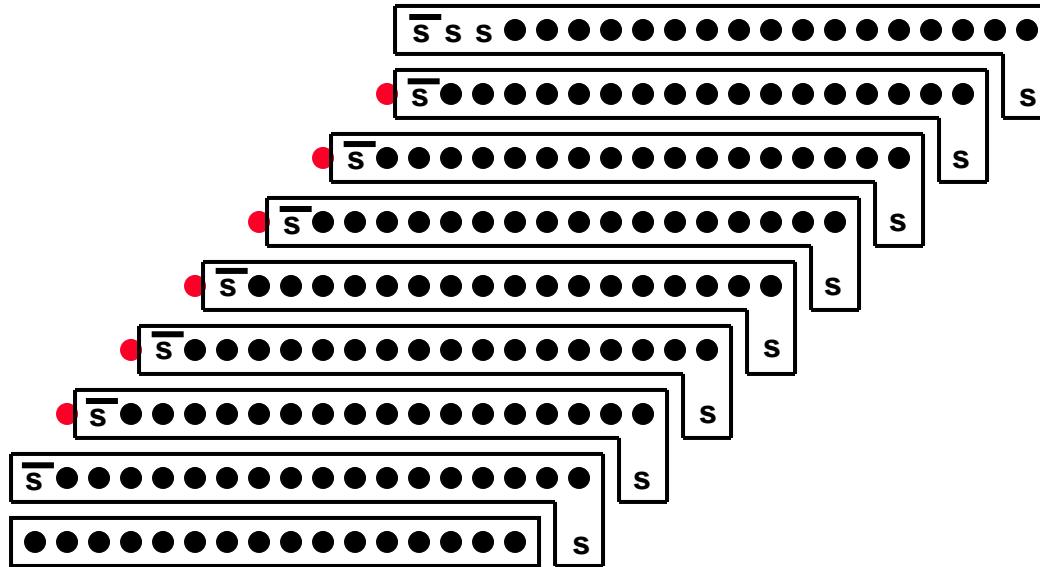
Allow Both Signs

- This is a fully general PP formation
 - Again, S=1 means a negative number



Add Up Partial Products

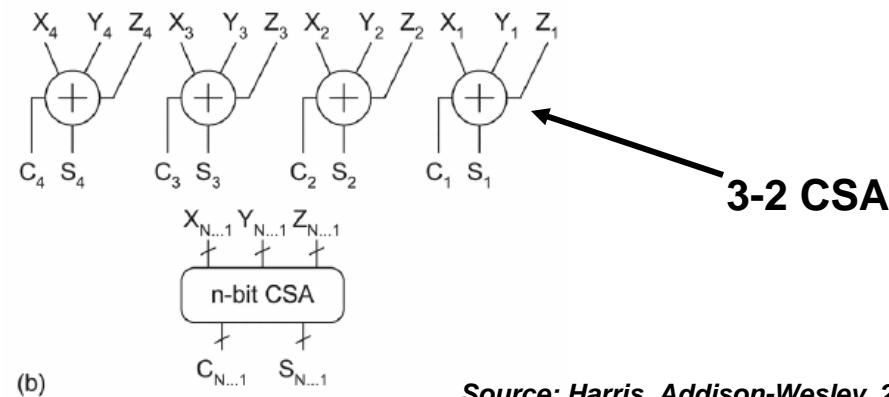
- So we can speed up the generation of the partial products
 - We still have to add them up, column by column



- Our simple iterative multiplier is slow with this add
 - Even if we optimize the number of partial products we generate
 - Adding more adders doesn't help; even fast adders are pretty slow

Carry-Save Adders

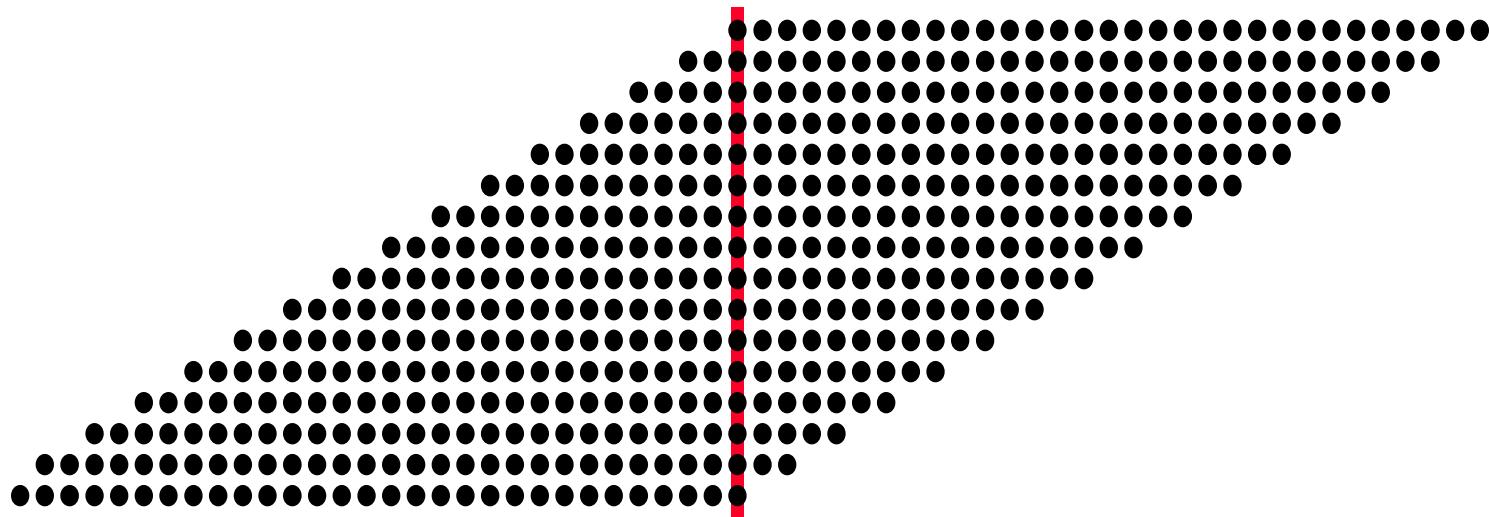
- For speed, delay carry propagation until later
 - There is no need for carry propagation after each sum
- Carry-Save Adders represent the sum in a “redundant form”
 - $\text{Sum} = \text{sum_1} + \text{sum_2}$
 - Compute sum and carry, but don’t propagate the carry
 - In other words, $\text{Sum} = \text{sum_without_carries} + \text{carries}$
 - Need to do a final add with a carry propagate at the very end



Source: Harris, Addison-Wesley, 2004

Using CSAs In Multipliers

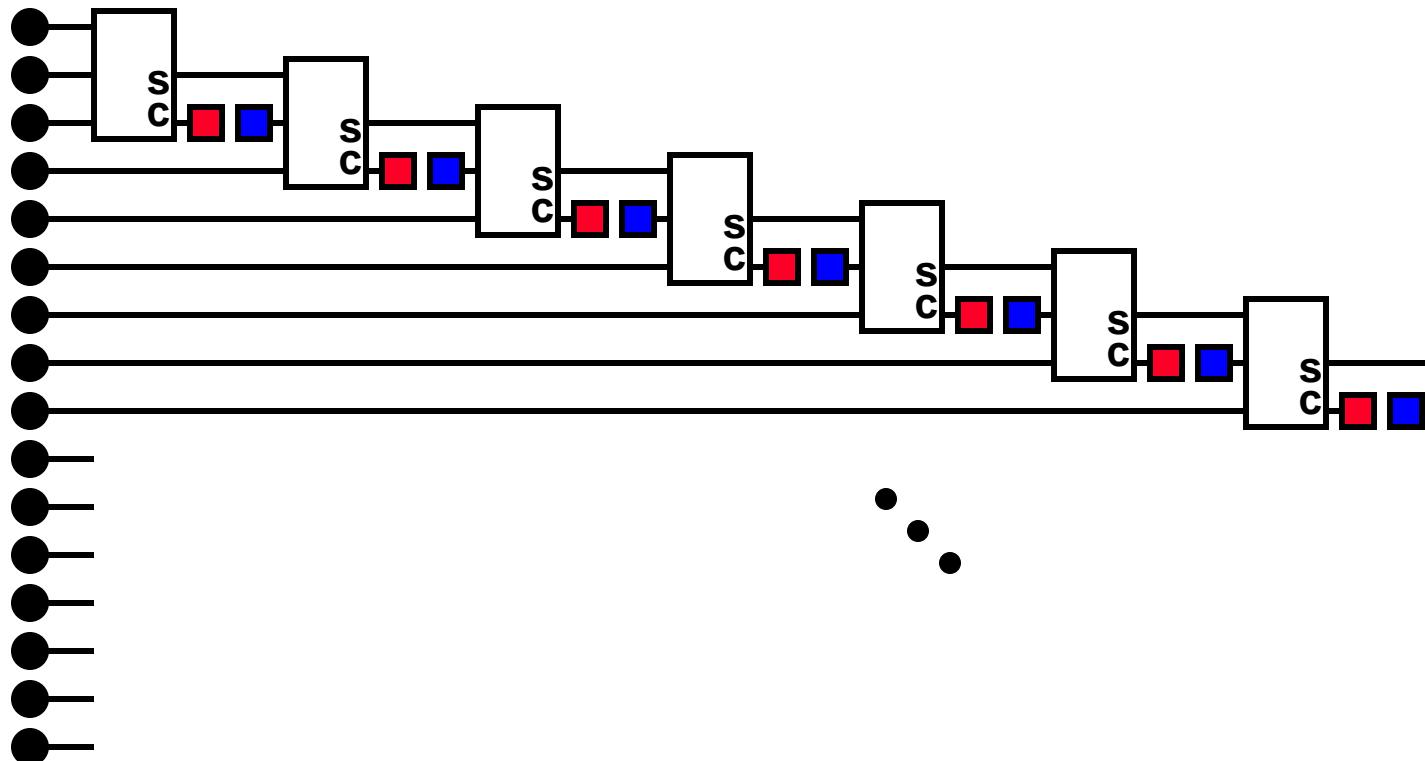
- Consider a 16-deep partial-product array
 - For example, a 30b multiplier using modified Booth recoding
 - Ignoring sign extensions in this dot diagram
 - Worst column is the center one; need to add 16 terms



- Add the columns up using 3-2 CSAs; avoid carry propagation

Using CSAs In Multipliers

- Group terms into a line of 3-2 CSAs
 - Sums stay in this column; carryouts go into left column (red)
 - Right column is giving me its carryouts (blue)

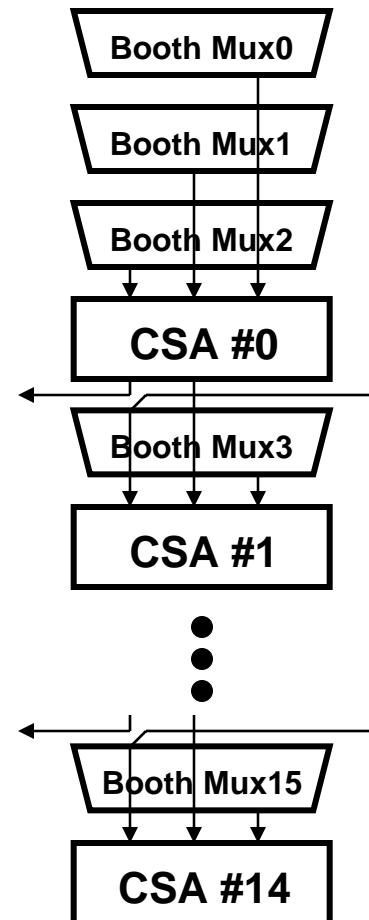


More About CSAs

- CSAs are small and fast
 - In Domino logic, a CSA is about 1.5 FO4
 - Very simple (just a full adder)
 - No carry ripple needed
- At each stage, redundant sum takes two inputs
 - Next partial product takes the third input
- One problem, of course, is at the very end
 - You need to sum up the redundant form
 - Shift the carry word over to a higher weight first
 - This takes a fast adder, but only one such adder

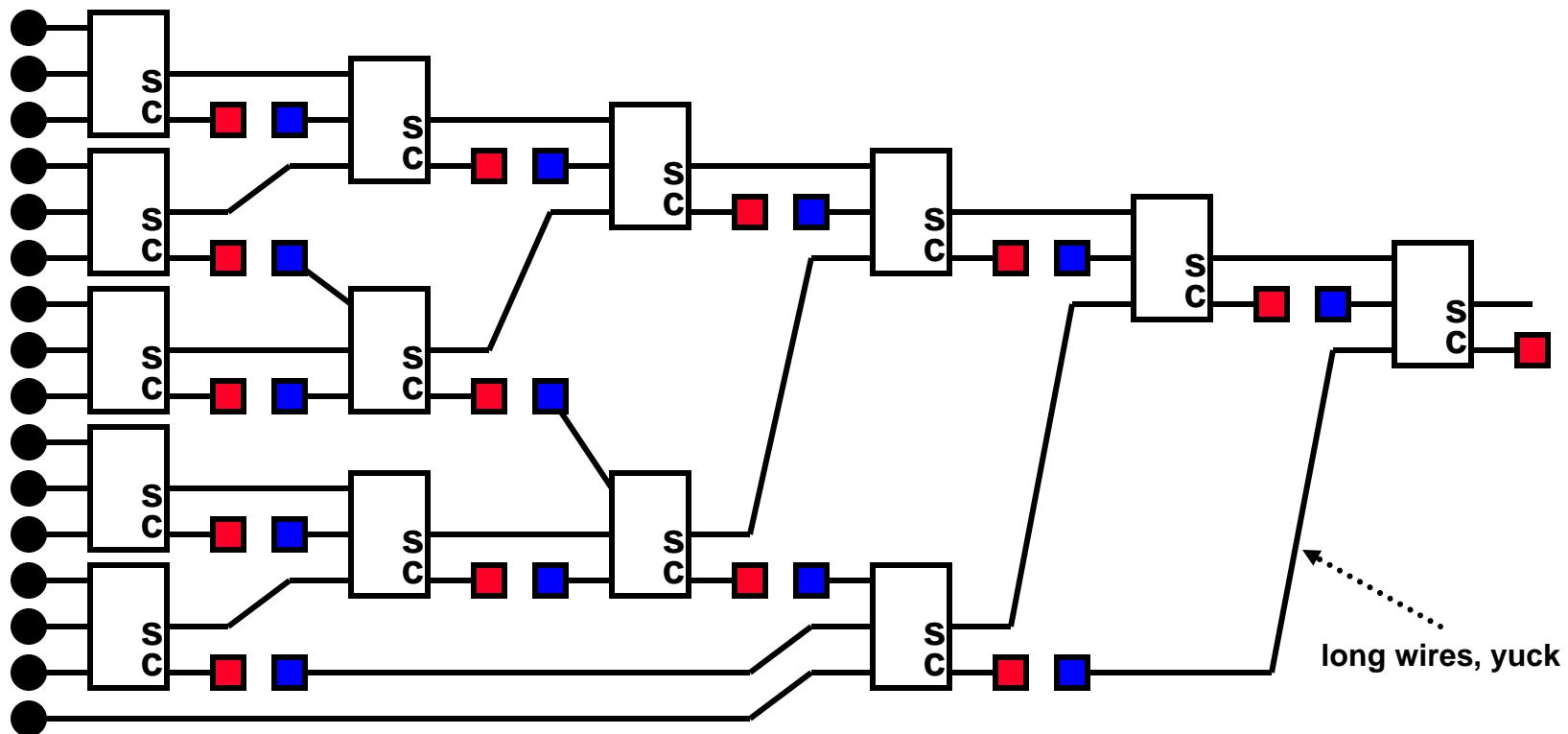
Block Diagram of This Array

- This sample adder has 16 partial products
 - Therefore 13 CSAs, all in the critical path
 - First CSA takes 3 partial products
- Very regular datapath, fairly short wires
- Long latency due to extended critical path
 - What if we move away from linear path?
 - What about logarithmic structures?



Using CSAs In Multipliers

- Group terms into a tree of 3-2 CSAs (a “Wallace Tree,” 1964)
 - Much shorter latency chain

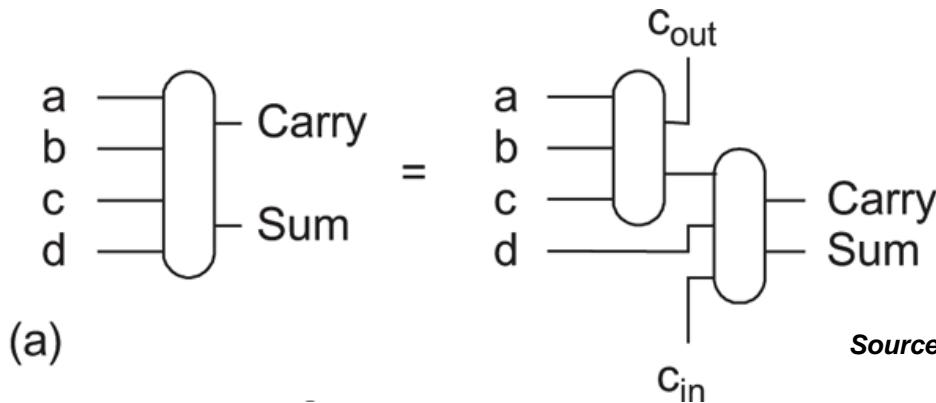


Problem With 3-2 Wallace Trees

- This seems good; critical path drops from 13 CSAs to 6
- But layout of this is messy
 - Irregular
 - Long wires that span multiple rows
 - 3-2 structures do not lend themselves nicely to trees
- Would much prefer to have a binary element for trees

4-2 Compressors

- Create a new element from two back-to-back 3-2 CSAs
 - Call this a 4-2 compressor: it “compresses” 4 inputs into 2 outputs

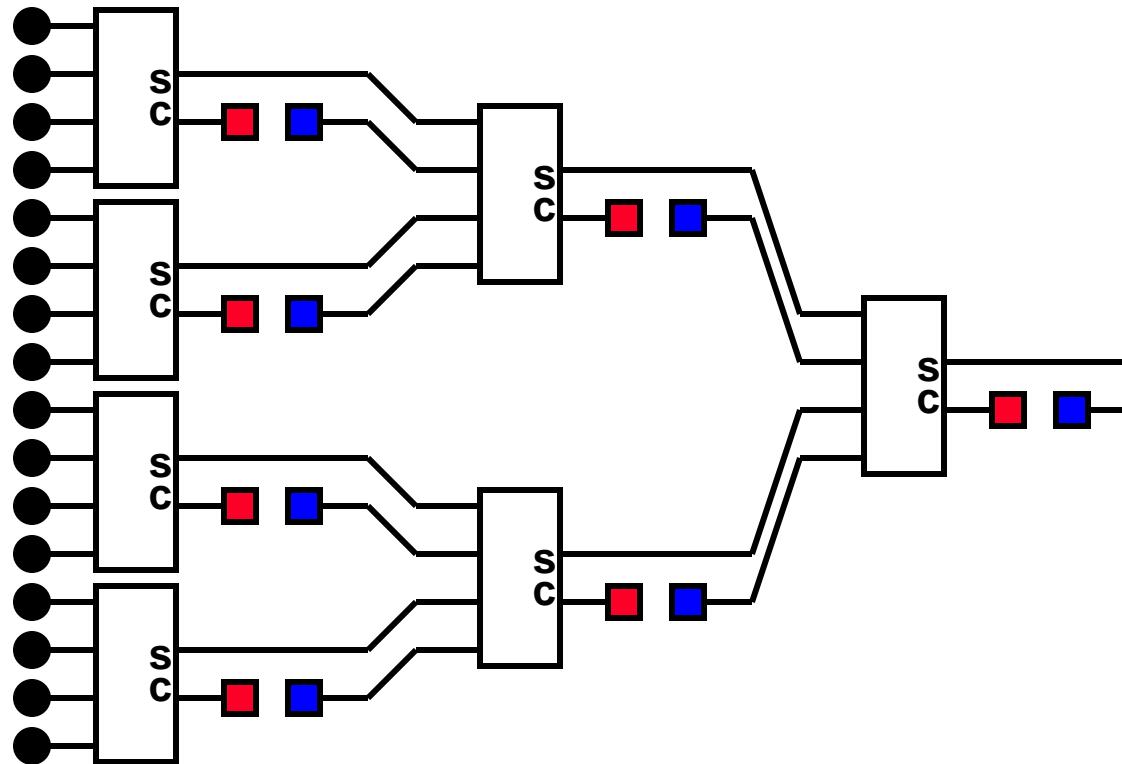


Source: Harris, Addison-Wesley, 2004

- “Wait,” you say. “This is really a 5-3 compressor.”
- Yes, that’s right. But 5-3 doesn’t sound remotely binary tree-like
- This element allows for much more regular layout and wiring

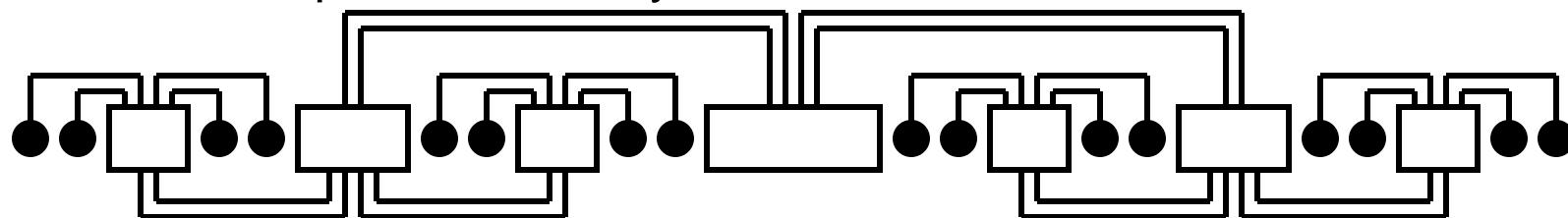
Using 4-2 Compressors In Multipliers

- Go back to the 16bit column example
 - In-between Cin and Cout terms (that make it 5-3) are not shown



Do 4-2 Compressors Fix Everything?

- 4-2 Compressors allow a regular layout (better than 3-2CSAs)
 - But still not as nice as the (slow) linear arrays
 - Still long wires, lots of routing tracks, lots of cross-overs
- Turn the picture sideways: bitslice



- Suppose this is our 30b multiplier w/ modified Booth recoding
 - What is the datapath height at each level?



Other Array Structures

- Some alternate methods of creating multiplier arrays
 - Even/odd arrays (Hennessy)
 - Array of arrays (Dhanesha)
 - Covers two partial arrays and four partial arrays
- I encourage you to look at these array structures
 - Perhaps you want to use them for your project
 - Trade off regularity and shortness of wires for latency
- Note that the readings are usually for floating point multipliers
 - Double-precision, so 53-bit mantissa
 - Booth encoding gives you 27 PPs, each 54b long (to support 2M)
 - With sign extension, you actually get 57b in first PP, 56b in rest