Lecture 11

Basic Building Blocks, Revisited

Adapted from: Mark Horowitz
Stanford University

Overview

- Reading

  + W H 10.2 Addition

- Introduction

  We now have all the tools we need to build a chip. We can write Verilog, create test benches, and then synthesize it to make it work. At the start of this class we talked about the basic building blocks for all VLSI: MOS transistors and wires. Now we want to look at the basic building blocks again, but at a higher level. When thinking about higher-level hardware design, what are the common motifs that we use? What does VLSI do well; how should we think about our problem?

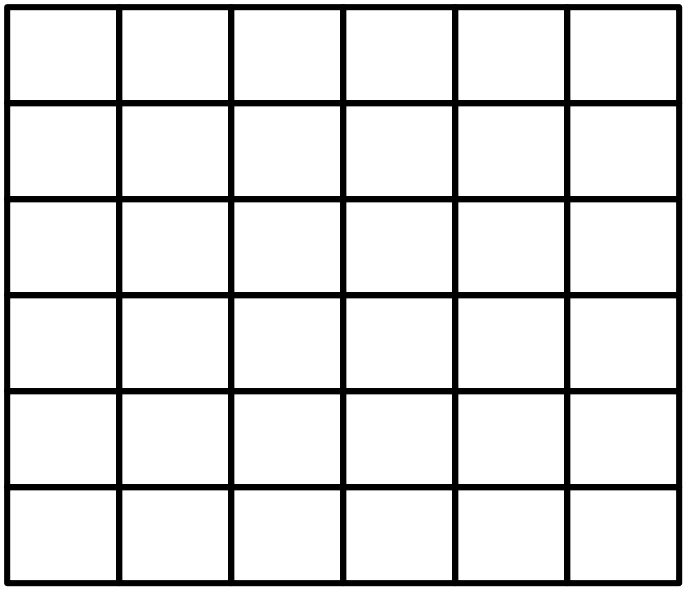
What Makes A Good VLSI Block?

Does something useful and is:

- Low power
  - Short wires
- Low area
  - Few wires
- Good performance
  - Short critical paths
- Has nice abstraction

Best Case Density, Design Time – 2D Array

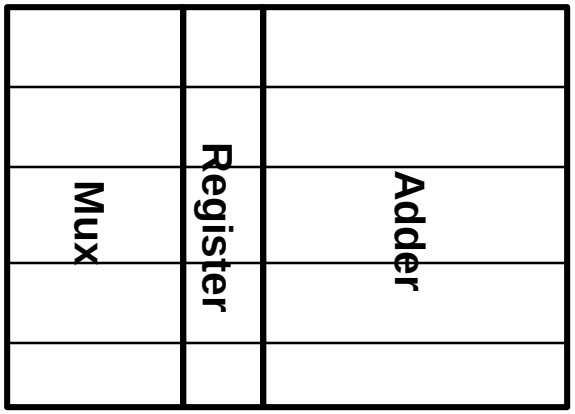
- Few wires (or short wires), lots of devices



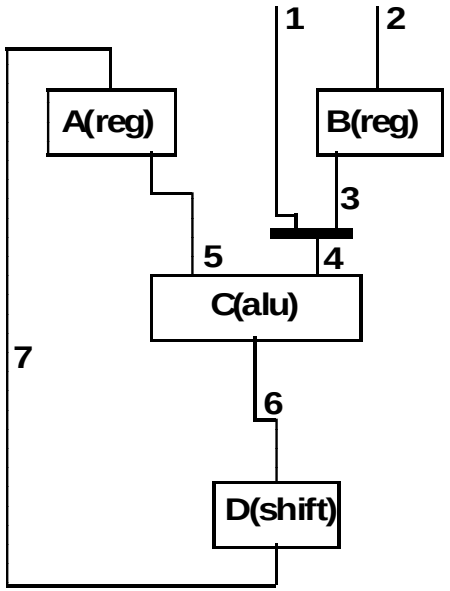
Good Case – 1D Array

- Replicated structure in one direction
  - Usually the bits in a word

- In the other direction you generally have different functions
  - Communication between function units is well defined
  - Mostly communication is between bits in same “slice”

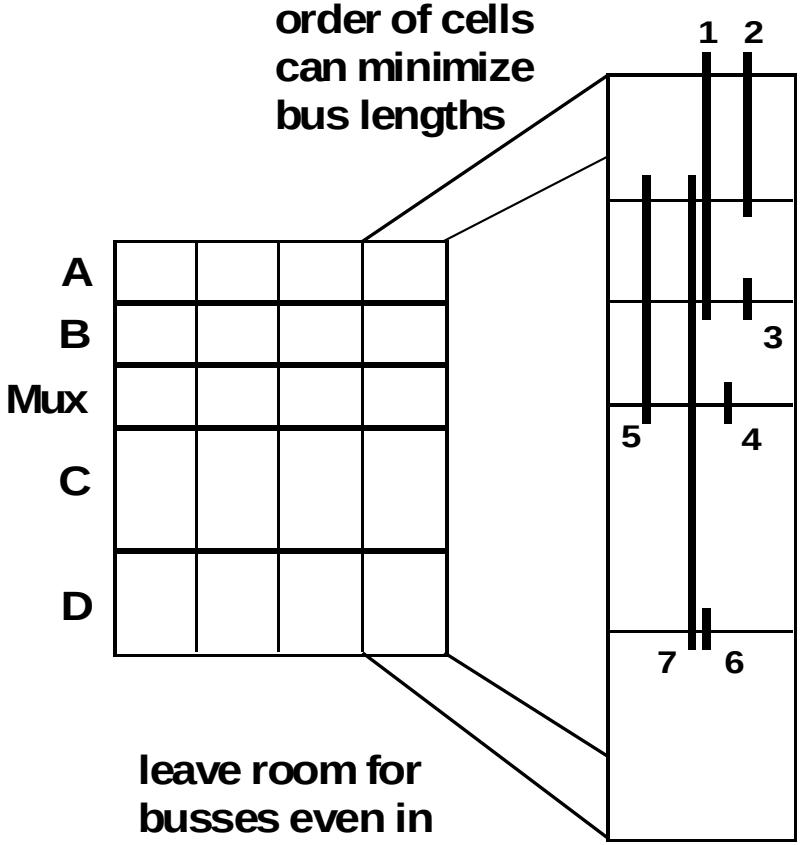


Datapaths



if all busses are 32 bits this is a nightmare to interconnect (note: ALU inputs need to be interleaved)

order of cells can minimize bus lengths

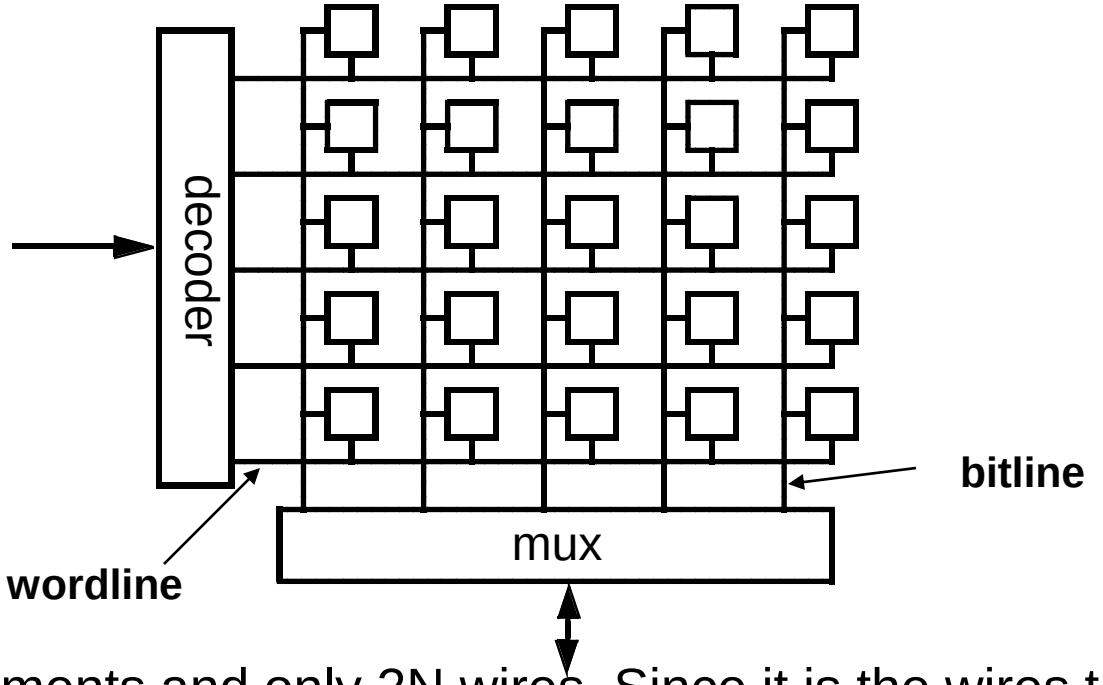


leave room for busses even in cells that don't need them

Implementing Datapaths / Arrays

- Datapaths? Arrays?
  - What does this have to do with my Verilog
  - I am just going to synthesize / place and route design
- For 2-D arrays
  - Certain structures (memories) come as larger blocks
    - Built using a different methodology
  - If you are creating an array of larger blocks
    - Create a separate cell in Verilog hierarchy
    - Then think of arraying that cell in your implementation
- For datapaths
  - Again create a separate file for the datapath
  - Possibly use a different placer for this section

Memories Have Very Regular Wiring



- It has N^2 elements and only 2N wires. Since it is the wires that occupy the most space, these structures are very dense. It is an easy way to use millions of transistors. The layout is quite dense.

What Do Chips Do?

- Mostly move data around
  - Sometimes data must be duplicated
  - Memories are good communication boxes
    - Anyone can store a valid word
    - Someone else can read it later to use it
- Pack and unpack data
  - Extracting headers from packets
  - Extracting byte chars, from words
- Operate on it
  - Arithmetic (+, *, /) in various types of number systems

What Do Chips Do, cont'd
Sequence and Control Stuff

- Need to control the part of machine moving data
  - Need to make sure it gets to the right place
  - At the right time
- Can need to coordinate with communication channels
- Or even the user

What is Data?

- Data is always encoded as a binary string
  - Chips only work on bits

- What does that binary string represent?
  - Elements of a set
    - Enumerated types, or presence of members

  - Characters (really an element of a set)

  - Numbers

Numbers

- Numbers are represented by N-bit binary strings.
  - N is chosen to minimize hardware/storage
    - And still get the right answer

- Programs generally use N=32 or 64
  - Hardware uses all different sizes
  - Modern computers have “vector” instructions
    - That allow subword operations(8bit, 16bit)

Numbers come in two forms:

- Fixed Point (Integer), and Floating Point (Real)

Fixed Point (a.k.a. Integer)

- There are two forms of fixed point, signed and unsigned
  - Difference is whether there is a sign bit
- Unsigned
  - Numbers range from 0 to 2^n -1
- Signed
  - Numbers are in two's complement form
  - Range from -2^{n-1} to 2^{n-1} -1
  - -1 is 11…11 (it is 0 -1)
- The binary point need not be at the end of the number
  - XXX.XXXXX, and this can be signed or unsigned

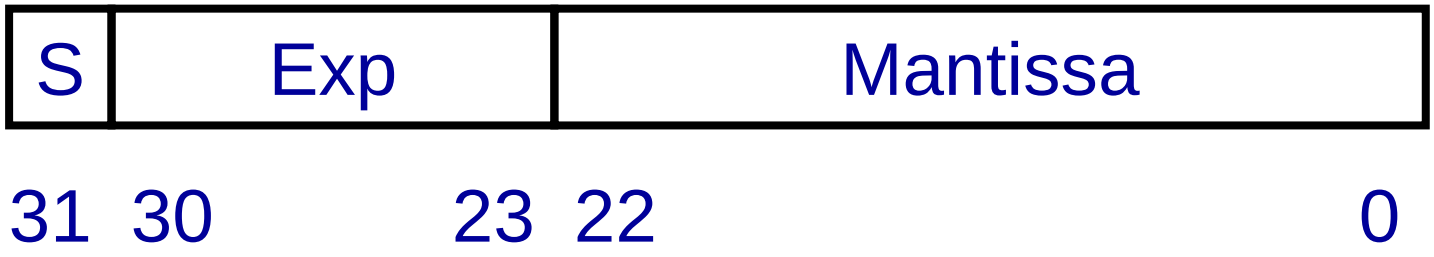
Fixed Point Numbers

- For fixed point numbers
  - The position of the binary point is not stored
  - The designer of the hardware needs to know this information
- For addition
  - Need to shift numbers to align the binary points
- For multiplication
  - Need to calculate where the binary point is in the product
  - And create the correct product
- Often used in signal processing applications
  - Where the input data has low resolution (images, sensors)

Floating-Point Numbers

- Floating Point
  - Even more complicated, contains an exponent field and a mantissa, which give it even more dynamic range.

- IEEE Format Single-Precision Floating-Point numbers:
  - Contain Sign bit, exponent, and mantissa
  - Number represents (-1)S*(1.mantissa)*(2)Exp-127
  - Numbers are in sign magnitude form



Floating Point Numbers

- Current IEEE approved formats:

Name	Common name	Base	Digits	E min	E max
binary16	Half precision	2	10+1	-14	+15
binary32	Single precision	2	23+1	-126	+127
binary64	Double precision	2	52+1	-1022	+1023
binary128	Quadruple precision	2	112+1	-16382	+16383

- Addition
  - Need to align numbers, then add
    - And if the add was really a subtract, need to renormalize
- Multiplication
  - Need to compute the new exponent

FP Operation

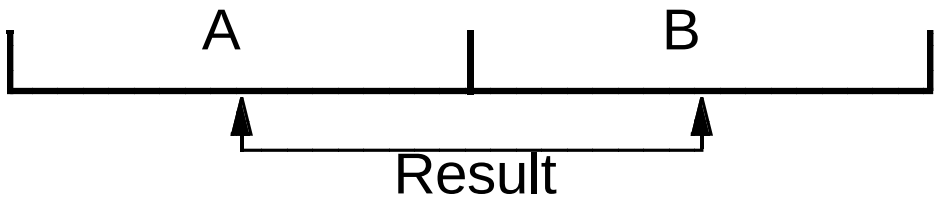
- Adder
- Multiplier

Other Operations - Shift

There are many different kinds of shifters.

- Simple Shifter
  - Shift the number to the right or left and fill-in with zeros
- Arithmetic Shifter
  - Left shifts are the same as simple shifter, but on right shifts use the sign bit to fill-in the new blank spaces (-2 shifted right by 1 gives -1)
    - Need to be careful about arithmetic right shifts. ‘1’ right shifted by 1 is 0. ‘-1’ right shifted by 1 is -1
- Barrel Shifter
  - Wrap the number onto a circle. The shift amount indicates where the new MSB will be. (Useful for rotating the bytes in a 32-bit word

Funnel Shifter



Is the most general kind of shifter

- Concatenates two n-bit words together
  - Then selects any contiguous n-bit subfield

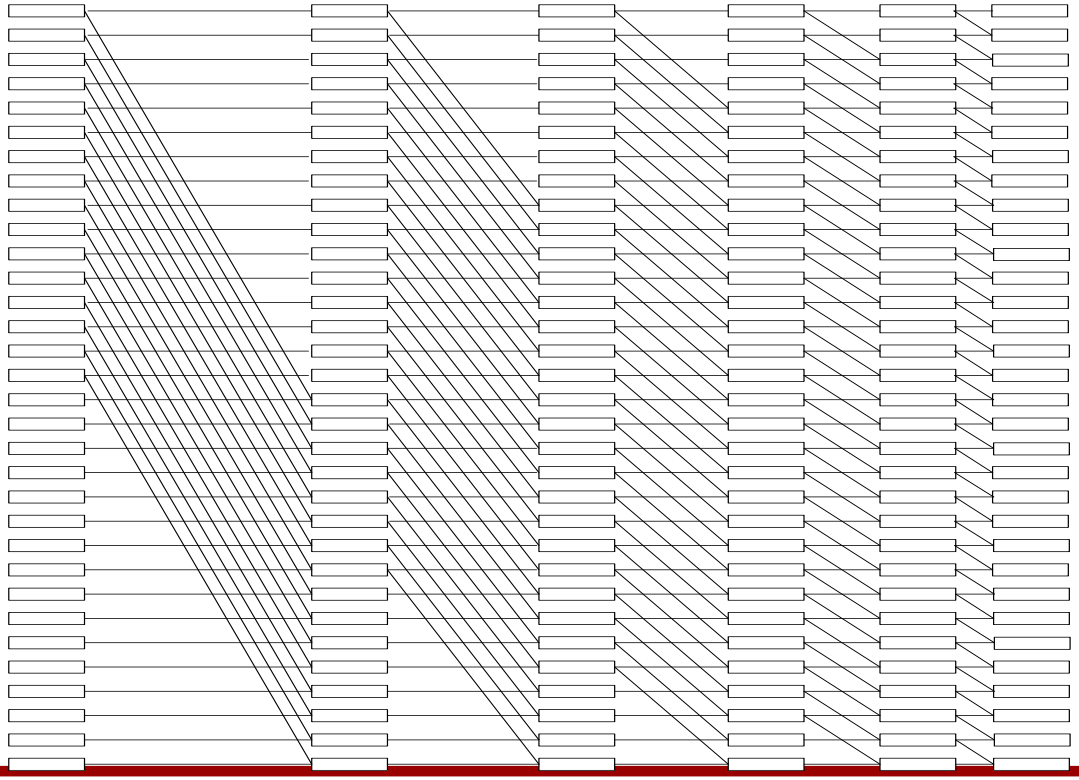
- Can implement all other shifters
  - If A=B get a barrel shifter
  - If A = sign bit, get arithmetic shifts
  - And it does byte inserts too.

Building a Shifter Using Switch Logic

- Each output is really just a large mux
  - Has a nice layout too.

A Mux Based Shifter

- Simplest shifter is 5 stages of 2:1 muxes (32 bit shift)
- Binary shift amount is used as the control signals

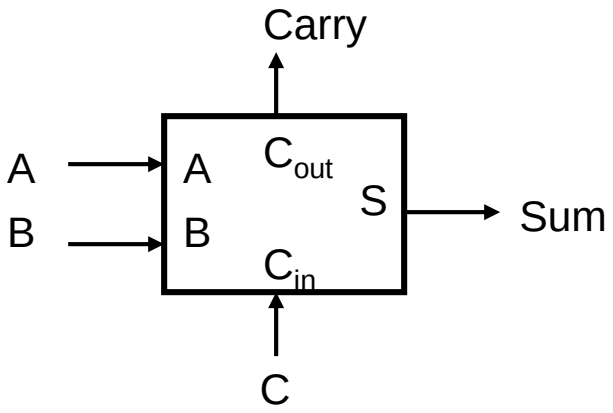


Look Quickly At Addition & Multiplication

- Both can be synthesized for you
  - And the results are going to be good
- But both implementation use tricks that are good to know
  - And can be used in other functions you might need
  - And these might not be as easy to automatically synthesize
- Adder's problem
  - Long sequential dependency (carry in to carry out)
  - We will convert linear delay to log delay using trees
- Multiplier's problem
  - Similar, but also needs to compute many adds

Adder Basic Building Block: Full Adder

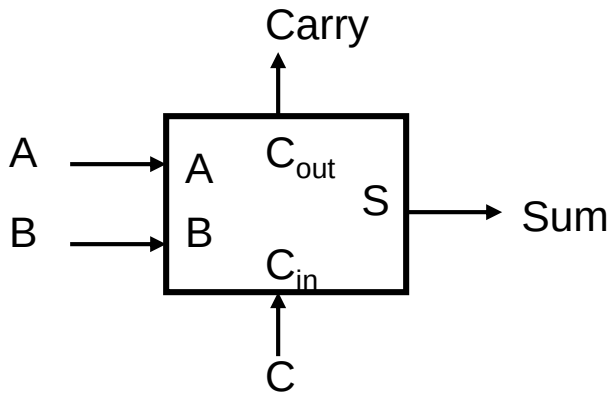
- Full adder adds three 1-bit numbers
- Inputs are A, B, and C
- Four possible output values: 0,1,2,3 -> coded on two bits
  - Carry is the MSB
  - Sum is the LSB



a	b	c	Carry	Sum
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

Logical Expressions for Full Adder

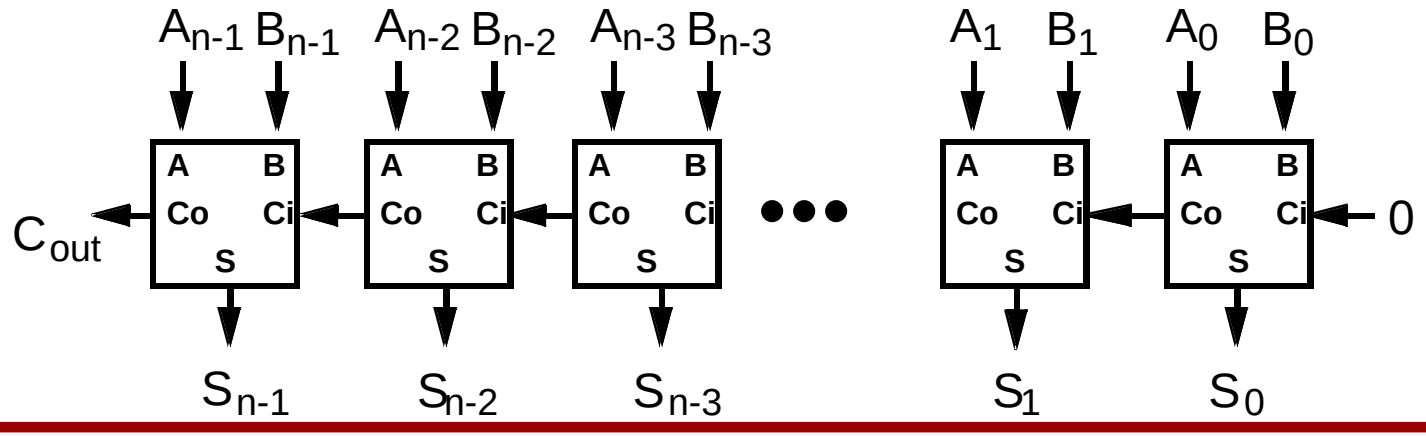
- Carry = AB + AC + BC
- Sum = A ⊕ B ⊕ C = A ~B ~C +~ A B ~C + ~A ~B C + A B C



a	b	c	Carry	Sum
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 Makes Adding Interesting: Adding N-bit Numbers

- Cascade N 1-bit full adders together
  - Two N-bit numbers connected to A and B inputs
  - C (Ci) and Carry (Co) inputs connected in a daisy chain
  - Set first Ci to be 0
  - Last Co indicates overflow
- Known as a Ripple Carry Adder (delay proportional to N)
  - It is used (good for small N)



Fast Adders

- Need to remove the linear dependency chain
  - Or make it much shorter
- How?
  - Need to do calculation in parallel
- But how, we have sequential dependency chain
  - Leverage that the equations for Cin are associative
  - Thus they can be done in any order
- Easiest seen by defining
  - PGK

PGK

- For a full adder, define what happens to Cout
  - Generate: C_{out} = 1 independent of C
    - G = A • B
  - Propagate: C_{out} = C
    - P = A $\oplus$ B
  - Kill: C_{out} = 0 independent of C
    - K = ~A • ~B

Using Generate / Propagate

- $C_i = G_i + P_i C_{i-1}$
  - Note you can use P = A+B here, why?
- $C_{i+1} = G_{i+1} + P_{i+1} C_i = G_{i+1} + P_{i+1} G_i + P_{i+1} P_i C_{i-1}$

  Group the terms in the above expression and name them as:

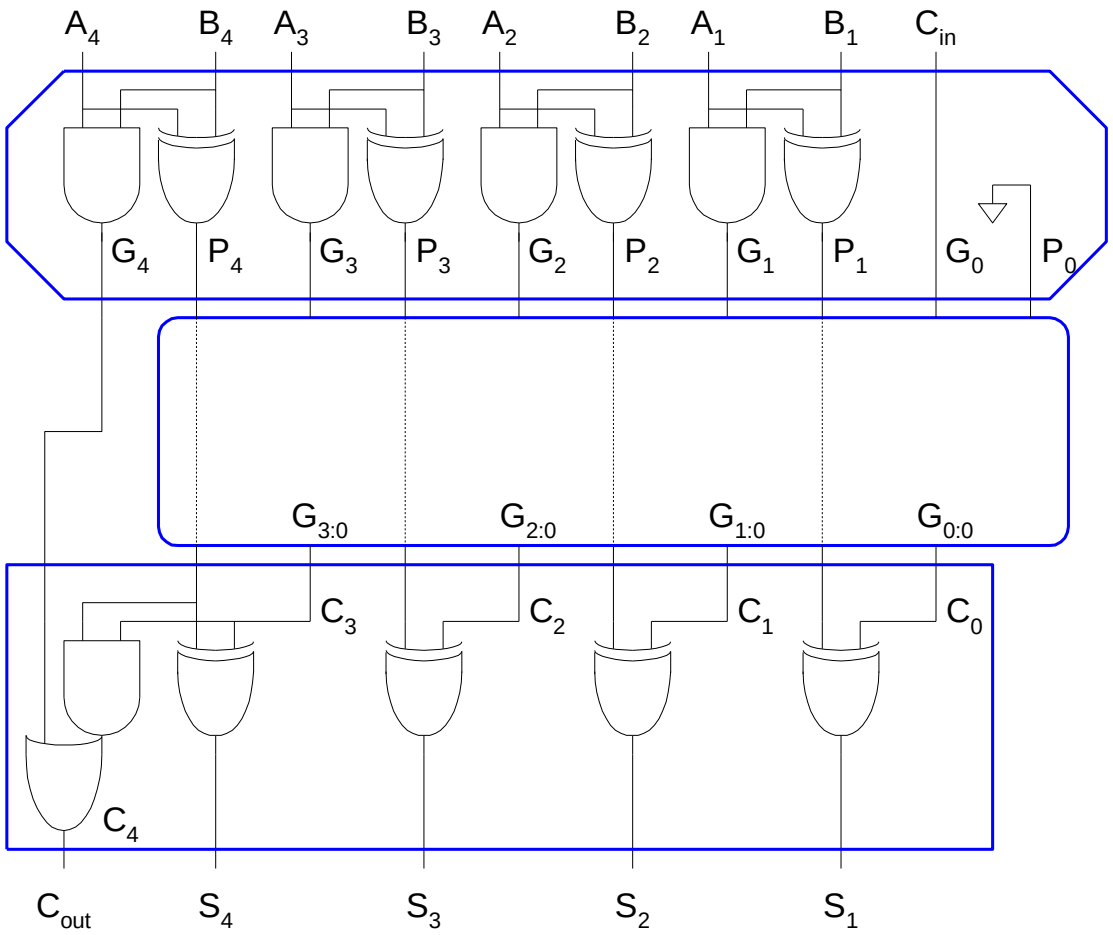
  $G_{i+1;i} = G_{i+1} + P_{i+1} G_i$

  $P_{i+1;i} = P_{i+1} P_i$
- Generate and propagate for groups spanning i:j

  i.e., $G_{i;j}$ (for $i > j$) = $G_{i;k} + P_{i;k} G_{k-1;j}$ (for $i >= k > j$)

  i.e., $P_{i;j}$ (for $i > j$) = $P_{i;k} P_{k-1;j}$ (for $i >= k > j$)
- Sum: $S_i = P_i$ XOR $G_{i-1;0}$

PG Logic

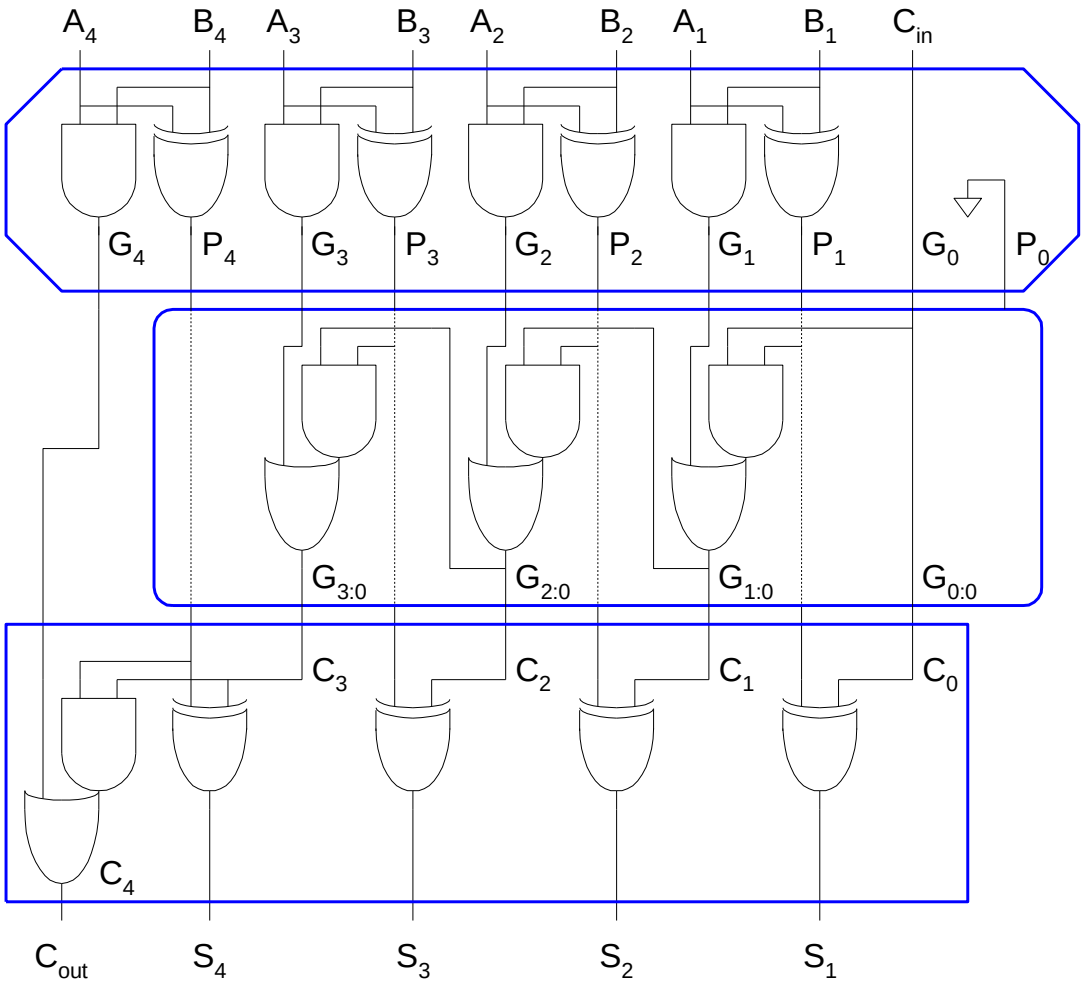


1: Bitwise PG logic

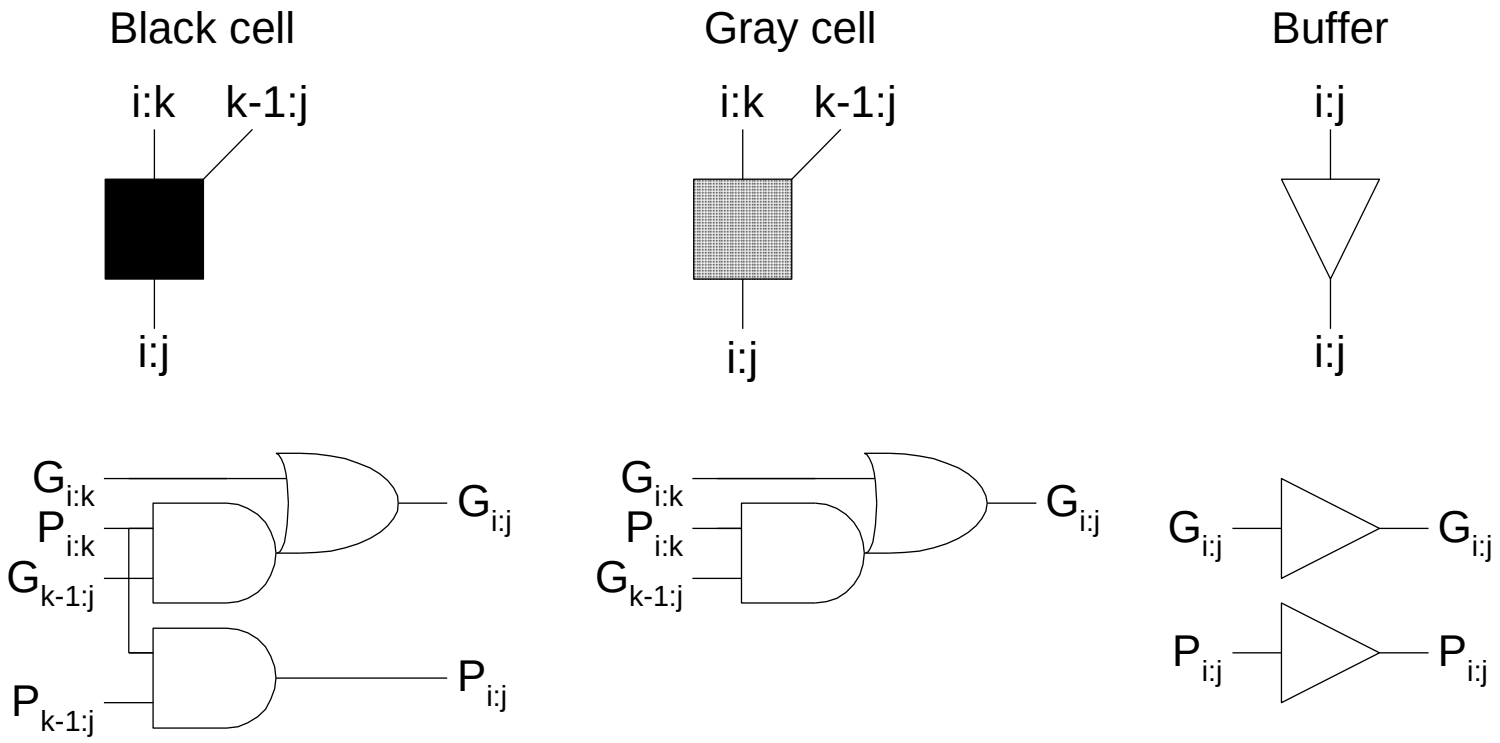
2: Group PG logic

3: Sum logic

Carry-Ripple Revisited



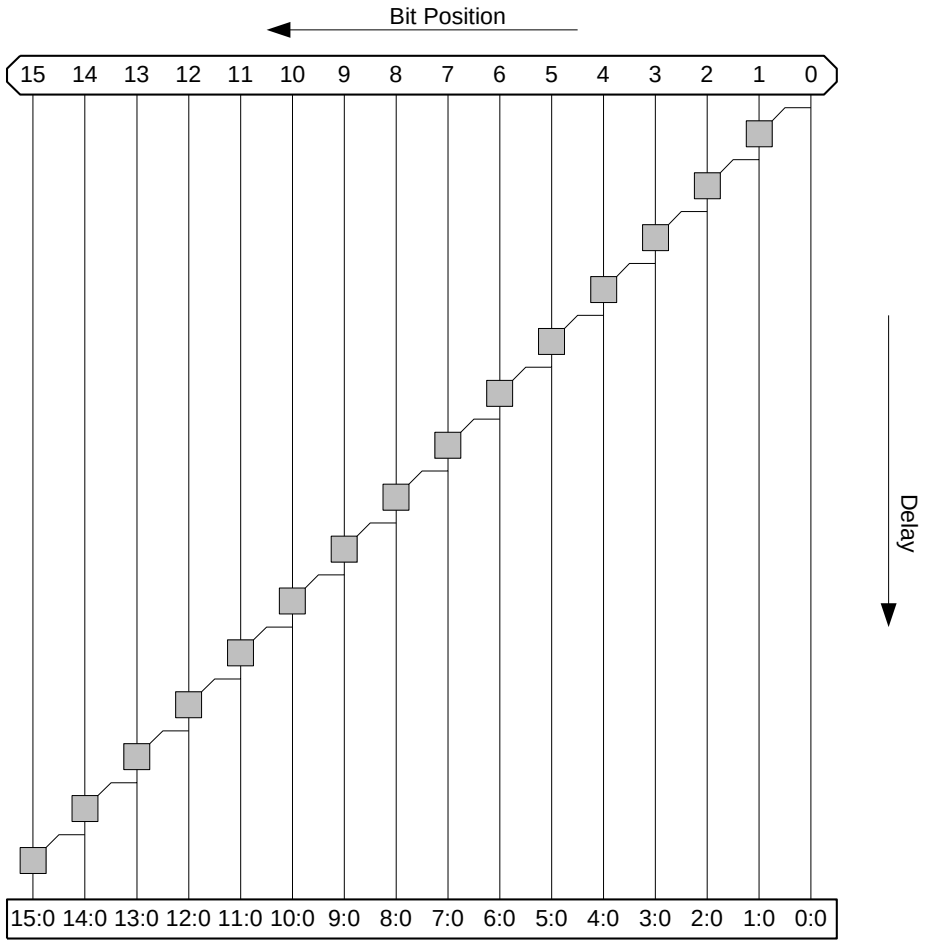
PG Diagram Notation



Carry-Ripple PG Diagram

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

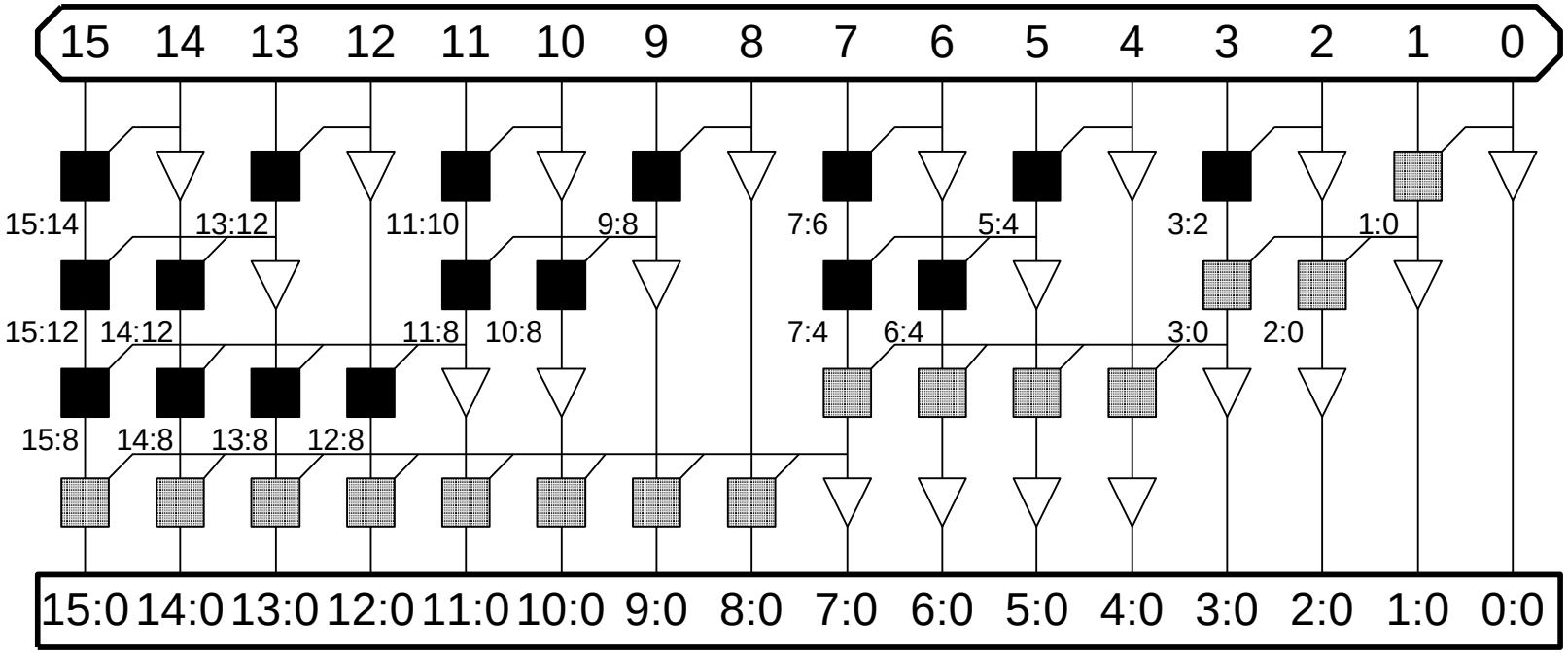
- Carry-ripple is slow through all N stages



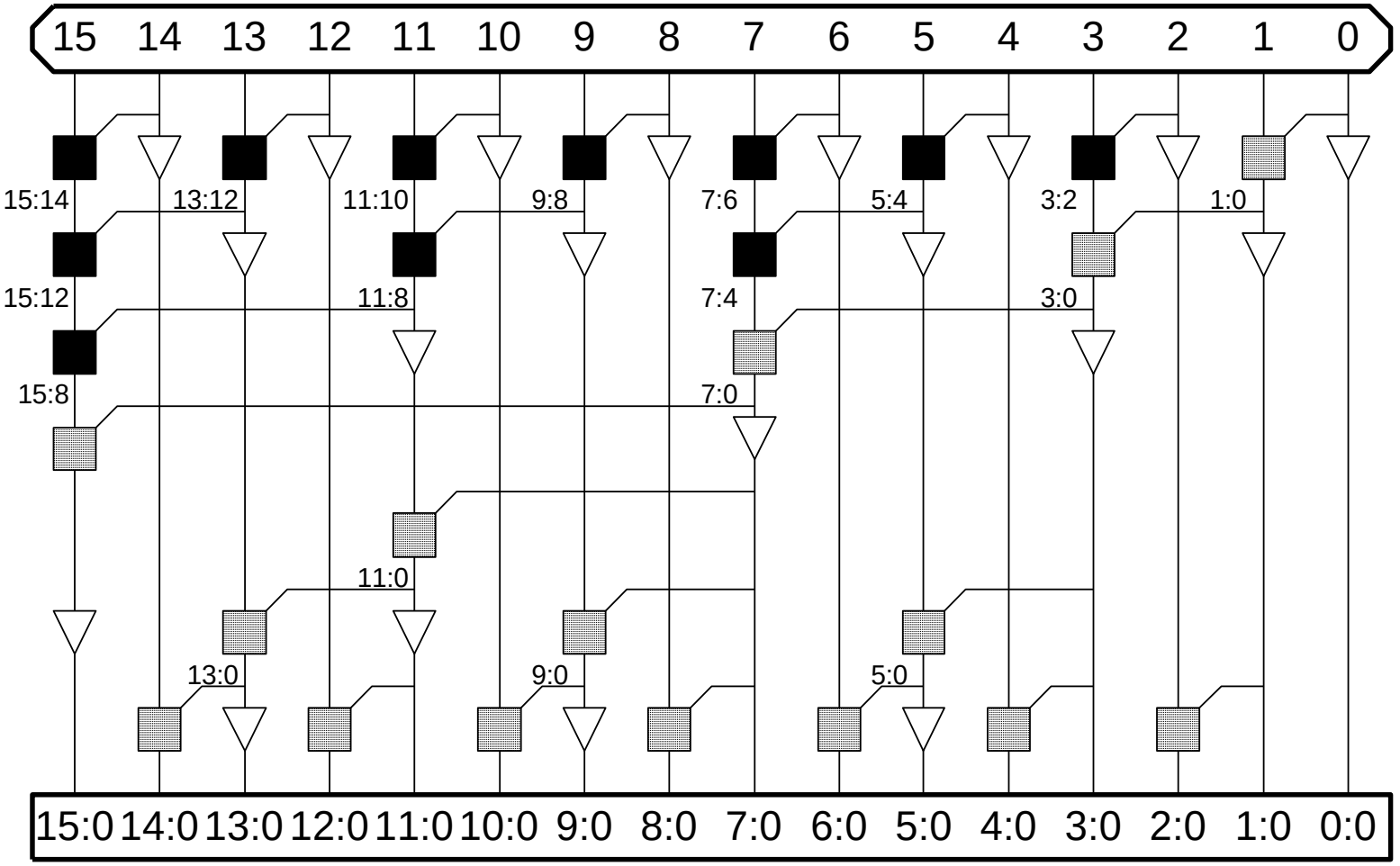
Tree Adder

- The P and G for the group computation is associative
  - Means you can compute the groups in any order
- Like a large fanin and gate
  - You can create a tree to compute the needed output
  - And the terms you create can be reused
  - The resulting delay is logarithmic with N
- Many variations on tree adders
  - Different ways reusing the outputs

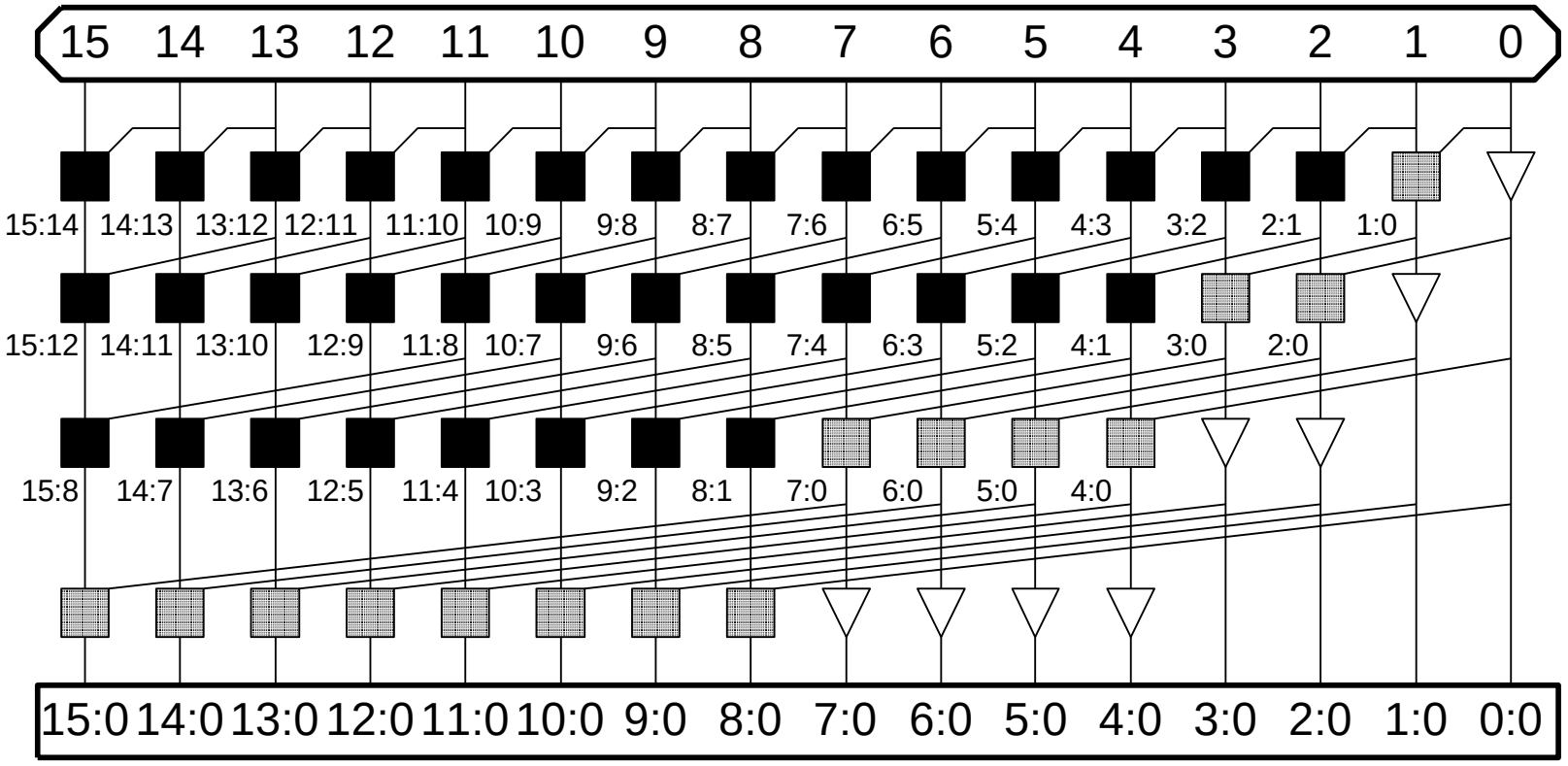
Sklansky


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15:14 13:12 11:10 9:8 7:6 5:4 3:2 1:0
15:12 14:12 11:8 10:8 7:4 6:4 3:0 2:0
15:8 14:8 13:8 12:8
15:0 14:0 13:0 12:0 11:0 10:0 9:0 8:0 7:0 6:0 5:0 4:0 3:0 2:0 1:0 0:0

Brent-Kung


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15:14
13:12
11:10
9:8
7:6
5:4
3:2
1:0
15:12
11:8
7:4
3:0
15:8
7:0
11:0
13:0
9:0
5:0
15:0 14:0 13:0 12:0 11:0 10:0 9:0 8:0 7:0 6:0 5:0 4:0 3:0 2:0 1:0 0:0

Kogge-Stone

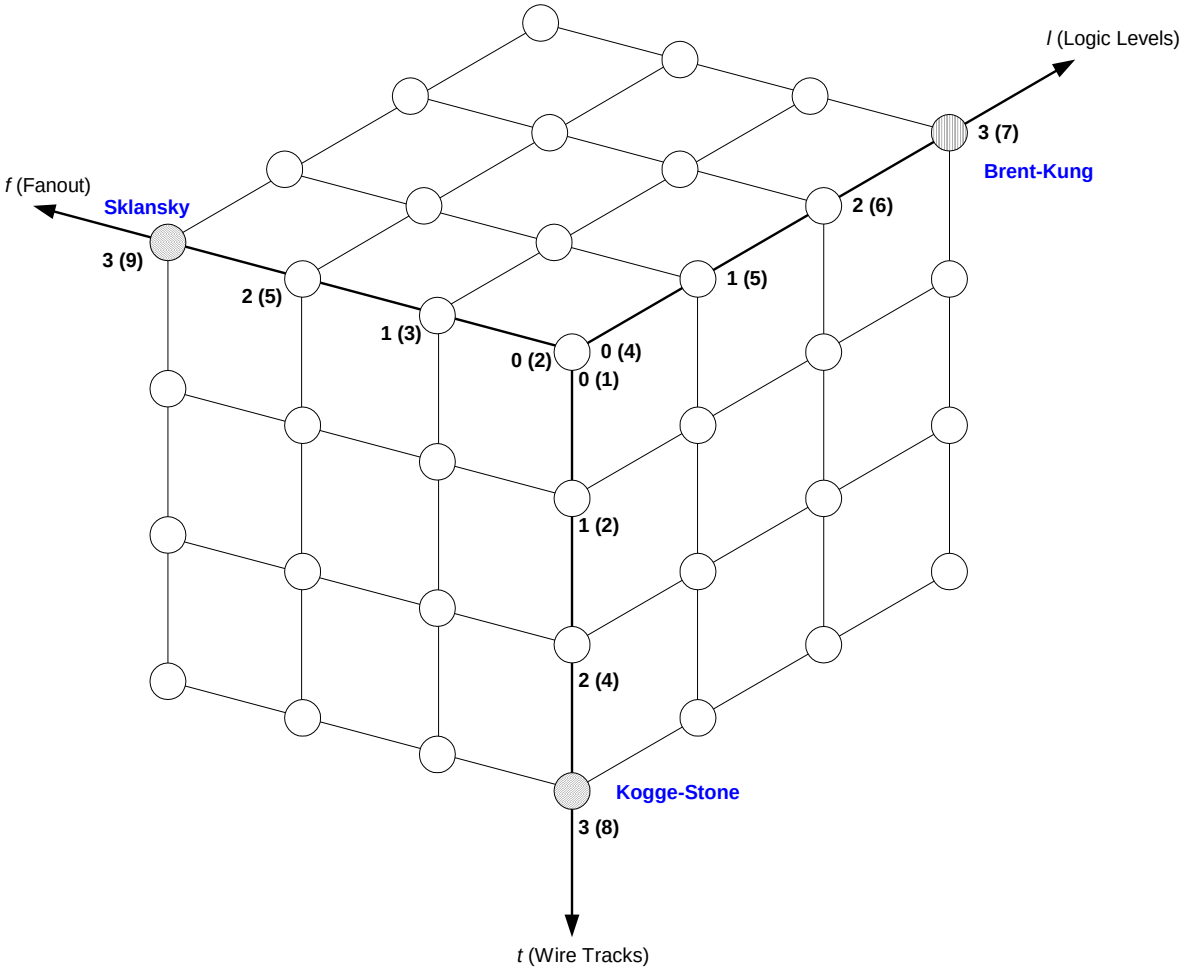

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15:14 14:13 13:12 12:11 11:10 10:9 9:8 8:7 7:6 6:5 5:4 4:3 3:2 2:1 1:0
15:12 14:11 13:10 12:9 11:8 10:7 9:6 8:5 7:4 6:3 5:2 4:1 3:0 2:0
15:8 14:7 13:6 12:5 11:4 10:3 9:2 8:1 7:0 6:0 5:0 4:0
15:0 14:0 13:0 12:0 11:0 10:0 9:0 8:0 7:0 6:0 5:0 4:0 3:0 2:0 1:0 0:0

Tree Adder Taxonomy (David Harris)

- Ideal N-bit tree adder would have
  - $L = \log N$ logic levels
  - Fanout never exceeding 2
  - No more than one wiring track between levels
- Describe adder with 3-D taxonomy (l, f, t)
  - Logic levels: $L + l$
  - Fanout: $2^f + 1$
  - Wiring tracks: 2^t
- Known tree adders sit on plane defined by

$$l + f + t = L\text{-}1$$

Tree Adder Taxonomy



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: 182*9 = 1638, 182*1 = 182; and 182*1 = 182
  - Shift them and add them up

```
01638   (9 * 182)
01820   (1 * 182)
18200   (1 * 182)
21658
```

- This is perhaps easier to read in binary…

Multiplication, Grad-School Level

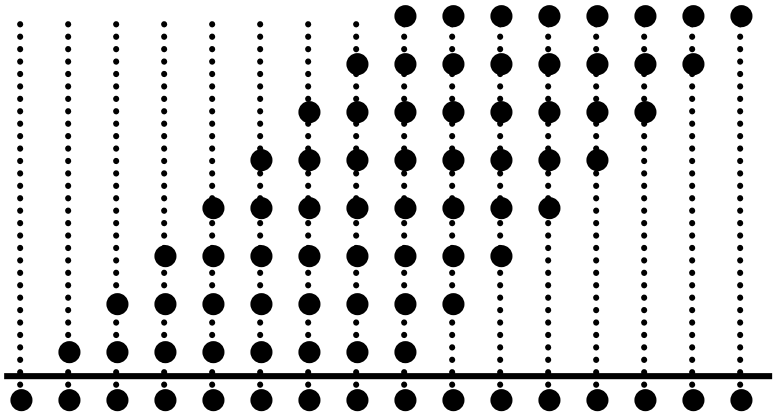
- Same basic idea, only now all digits are 0 or 1
  - But still have multiplicand, multiplier, and partial products
  - Ex: 119 = 0111 0111; 182 = 1011 0110

```
. . . . . . . . 1 0 1 1 0 1 1 0
. . . . . . . 1 0 1 1 0 1 1 0 .
. . . . . 1 0 1 1 0 1 1 0 . .
. . . . 0 0 0 0 0 0 0 0 . . .
. . . 1 0 1 1 0 1 1 0 . . . .
. . 1 0 1 1 0 1 1 0 . . . . .
. 1 0 1 1 0 1 1 0 . . . . . .
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)
```

- 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
    - Why?

Adding up the PPs

- Main work of multiplier is to add all the partial products together
  - This is going to take a large number of adders (or)
  - A large number of cycles for one adder

- What things can we do to make this faster
  - Reduce the number of partial products
    - Called modified booth recoding
    - Look at two bits of multiplier and create one partial product
  - Make the adders very simple (but still fast)
    - Use just full adders, but use a carry-save number representation
  - Use trees to make the critical path shorter

Technique #1: Coding to Reduce Work

- Observations:
  - Latency is determined by how many partial products I have
  - So reduce latency by reducing the number of partial products
  - If the multiplier bit is 0, there is no need to create a partial product

- Actually, we can do better
  - Suppose our multiplier is 0111 1111 (seven partial products)
  - Rewrite this as 0111 1111 = 1000 0000 – 0000 0001
  - Now I only need two partial products (but one is negative)

- This is called Booth encoding (1951)
  - Find strings of “1”s in the multiplier
  - Rewrite this as the difference between two numbers

Basic Booth Recoding

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

```
- . . . . . . . . 1 0 1 1 0 1 1 0
+ . . . . 1 0 1 1 0 1 1 0 . . .
- . . . 1 0 1 1 0 1 1 0 . . . .
+ 1 0 1 1 0 1 1 0 . . . . . . .
-------------------------------
0 1 0 1 0 1 0 0 1 0 0 1 1 0 1 0 = 21658₁₀
```

  - This is an improvement; six partial products to four

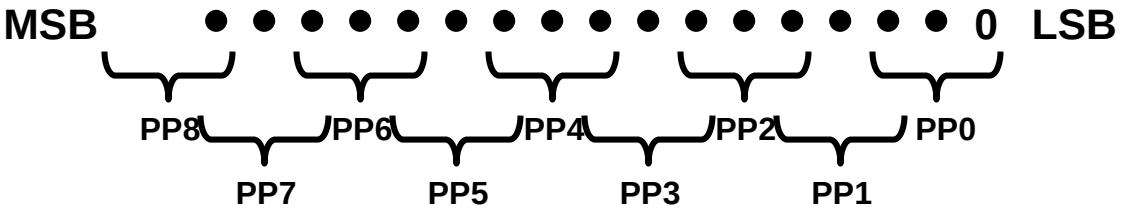
- Not always helpful; imagine input of 170 = 10101010
  - Recoding into differences of two numbers doesn't help at all
  - No string of 1's to exploit

- 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 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
  - +/- 2M are easy: just take +/- M and shift it over a bit

Carry-Save Adders

- If propagating the carry is slow, don't propagate the carry
  - "Doctor, it hurts when I do this." "Then don't do this."

- Carry-Save Adders represent the sum in a "redundant form"
  - Sum = sum_1 + sum_2
  - Compute sum and carry, but don't propagate the carry
  - In other words, Sum = sum_without_carries + carries
  - Need to do a final add with a carry propagate at the very end

0 1 0 1 0 1 1 0 1

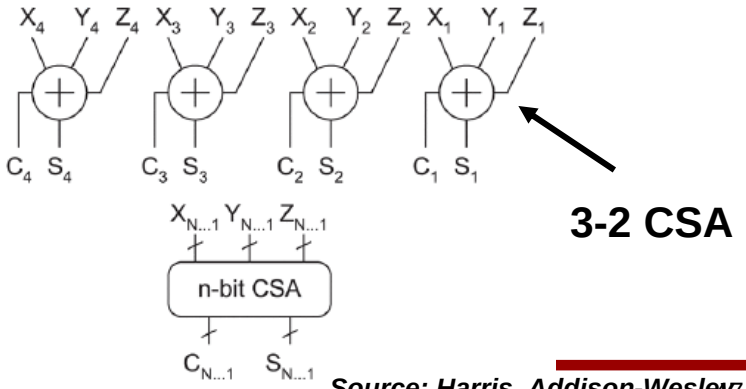
+ 1 1 1 0 1 0 1 1 1

==========

S 1 0 1 1 1 1 0 1 0

(Keep both S and C; add them later)

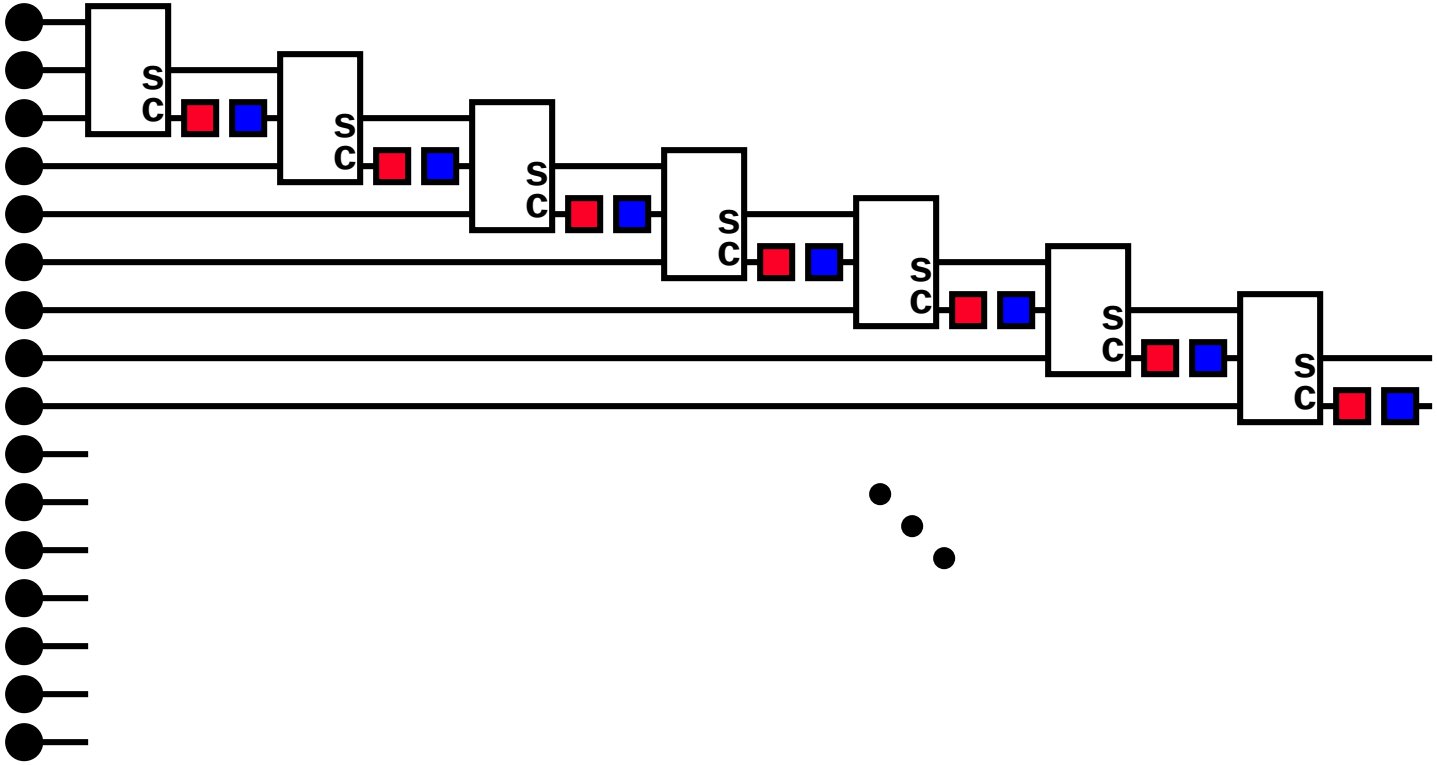
C 0 1 0 0 0 0 1 0 1



Source: Harris, Addison-Wesley, 2004

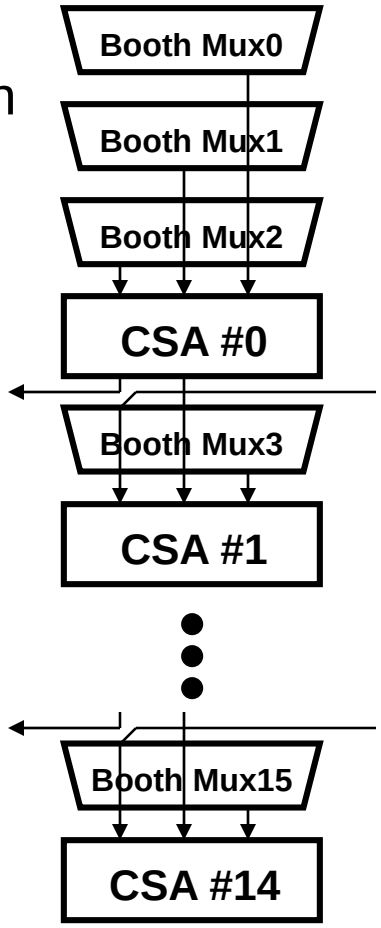
Using CSAs In Multipliers

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



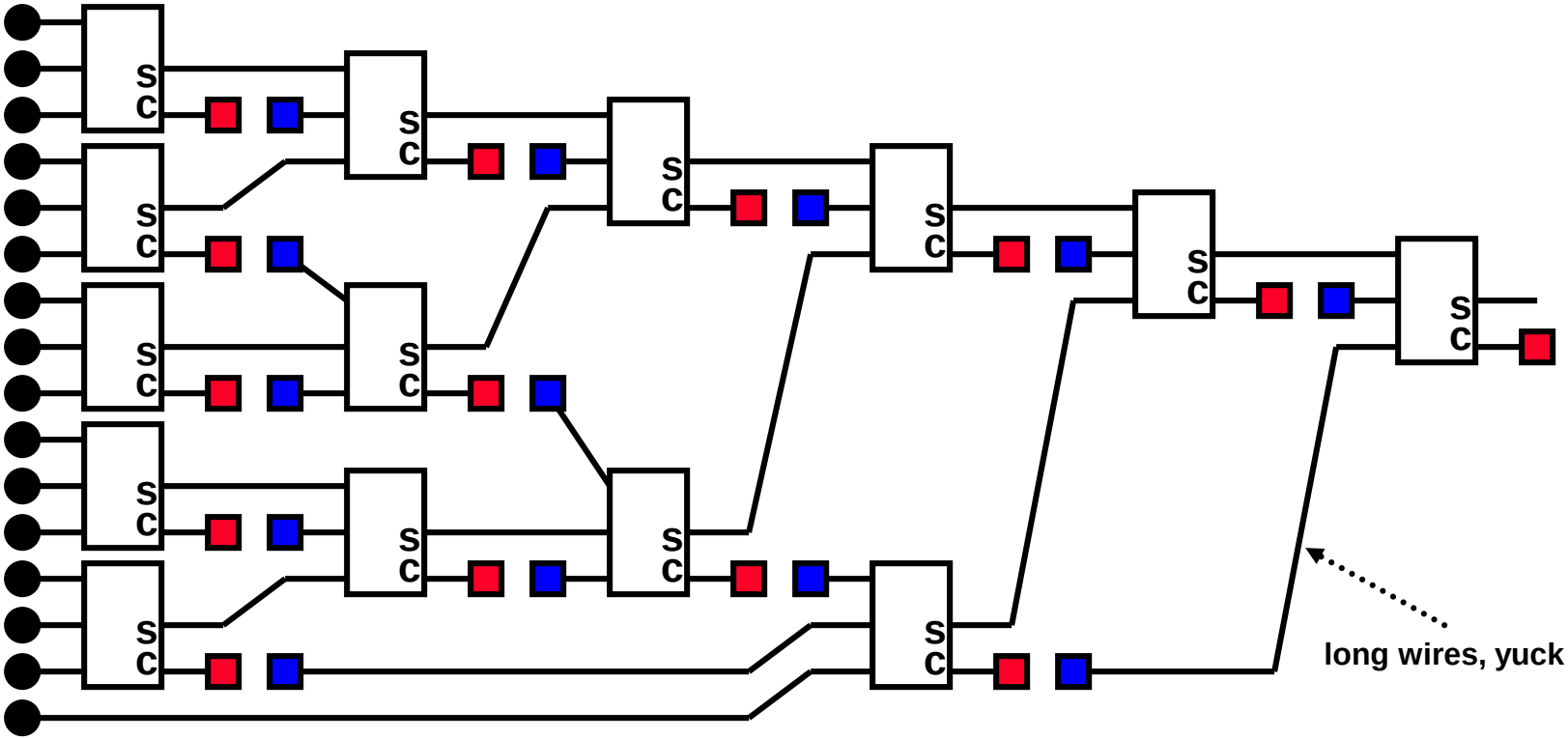
Block Diagram of This Array

- 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?



Can Use Tree Here Too

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

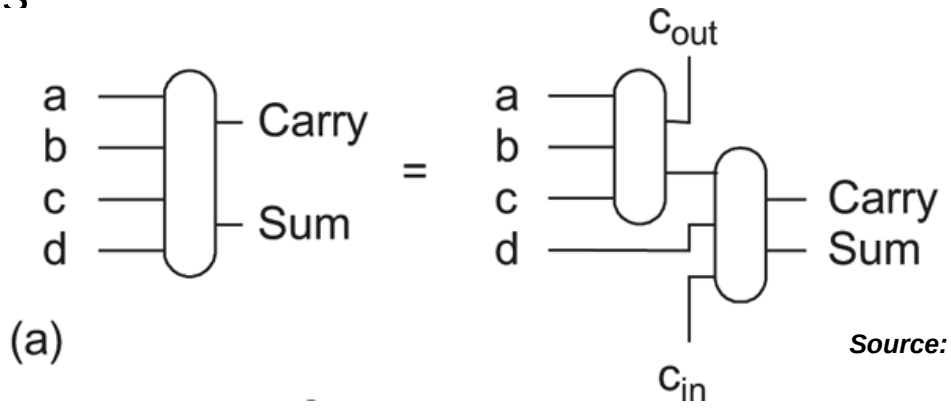


Problem With 3-2 Wallace Trees

- 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



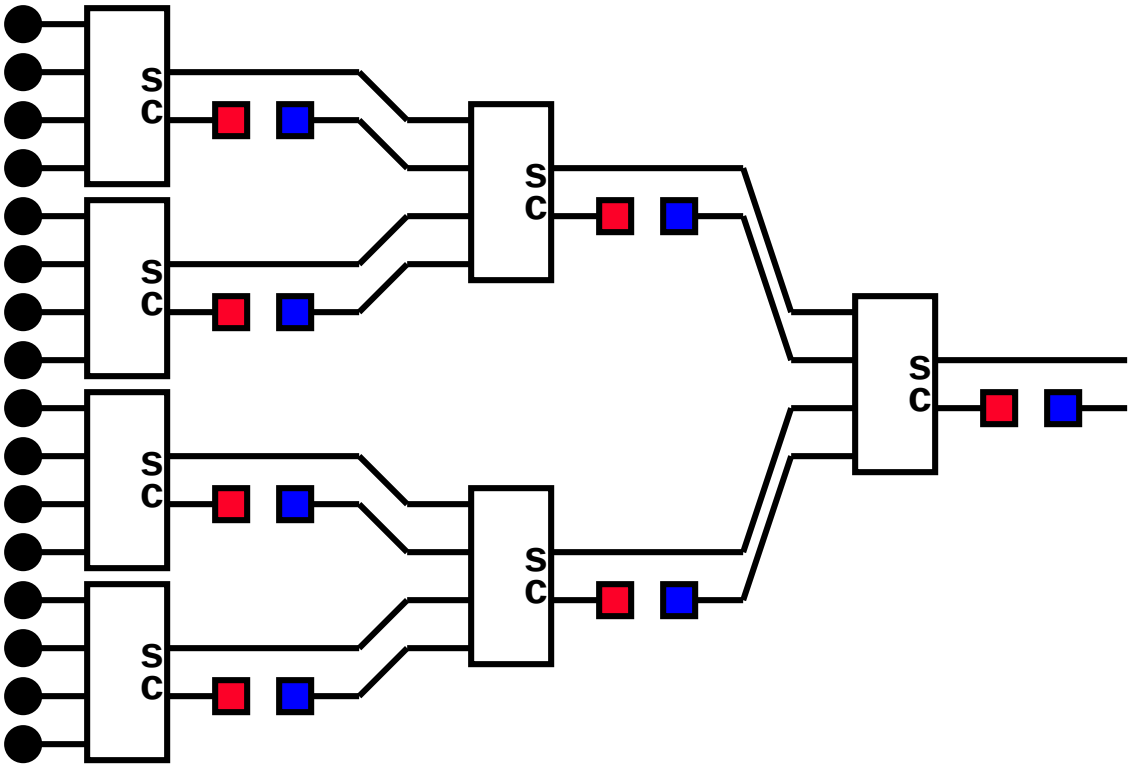
Source: Harris, Addison-Wesley, 2004

  - "Wait," you say. "This is really a 5-3 compressor."
  - Yes, that's right. Stop interrupting.

- 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

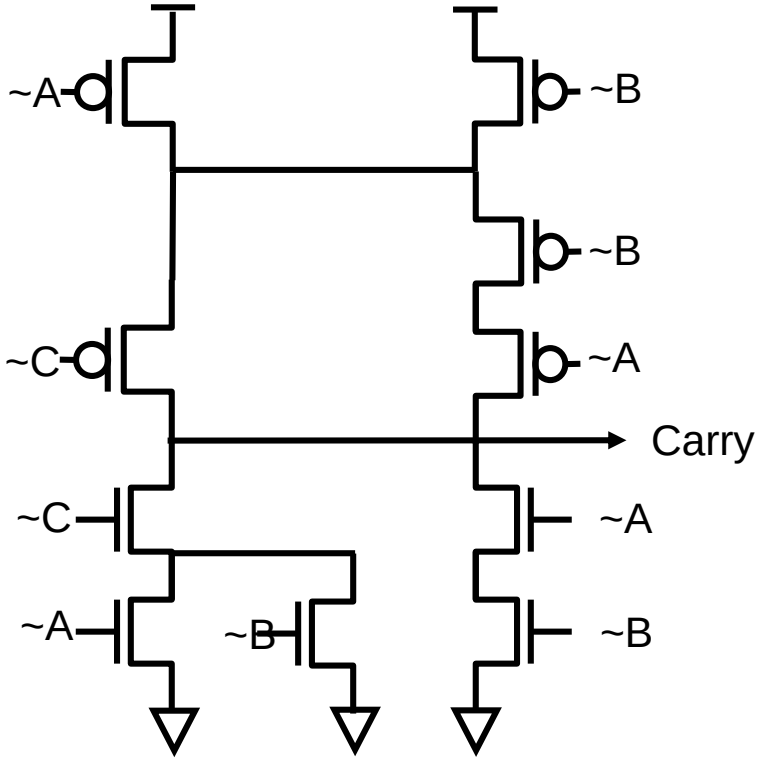


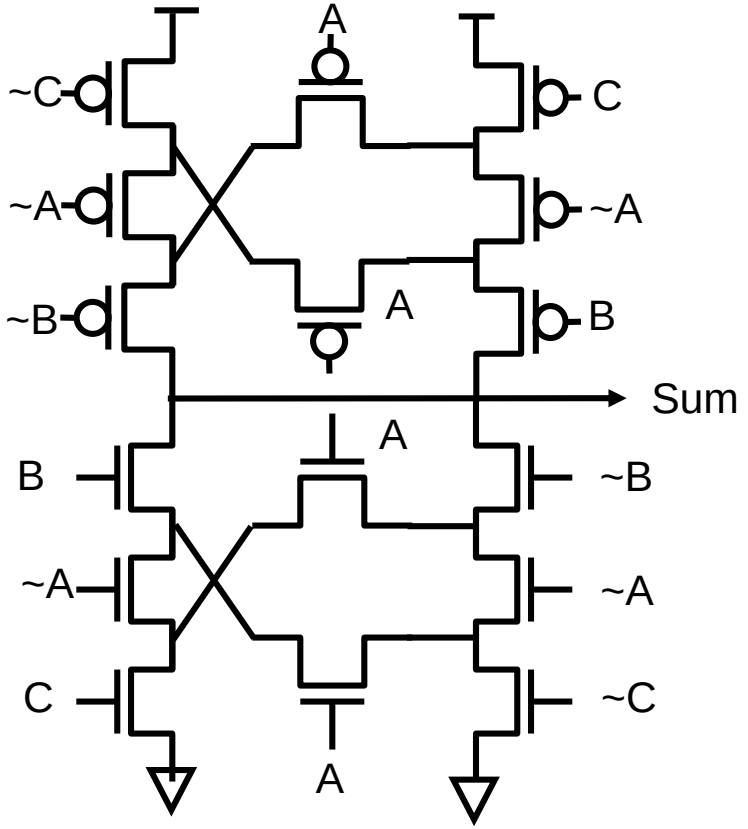
Logic Implementation of Full Adder

- Carry = AB + AC + BC
  - "At least two inputs are high"
  - Can be rewritten as AB + C(A+B)
- Sum = A $\oplus$ B $\oplus$ C = A ~B ~C +~ A B ~C + ~A ~B C + A B C
  - "One or three inputs are high"

+ Transistor Level Implementation

- 32 Transistors (including inverters for A,B, and C)
- Various forms of 3-input XOR are possible



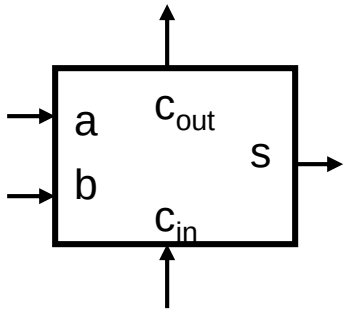


+ Shared Terms between Carry and Sum

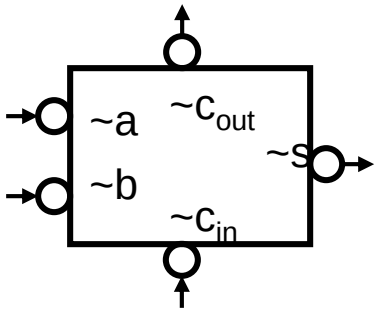
- XOR's are somewhat expensive to implement
  - Actually very similar to multiplexers
    - Transmission gate logic only works for adders
- We can avoid them by taking another look at the problem
  - SUM=1 when 1 or 3 inputs are high (but not 0 or 2)
  - Carry =1 when 2 or 3 inputs are high
- Possible to reuse the carry term in sum to save some transistors
  - Sum = ABC + ~Carry(A+B+C)
- Need to produce ~Carry instead of Carry for this to work

+ Aside: Interesting Property of Full Adders

- Inverting all the inputs creates an inverted output
  - Not true of normal gates (NANDs become NORS, etc)
- We can take advantage of this to produce ~Carry



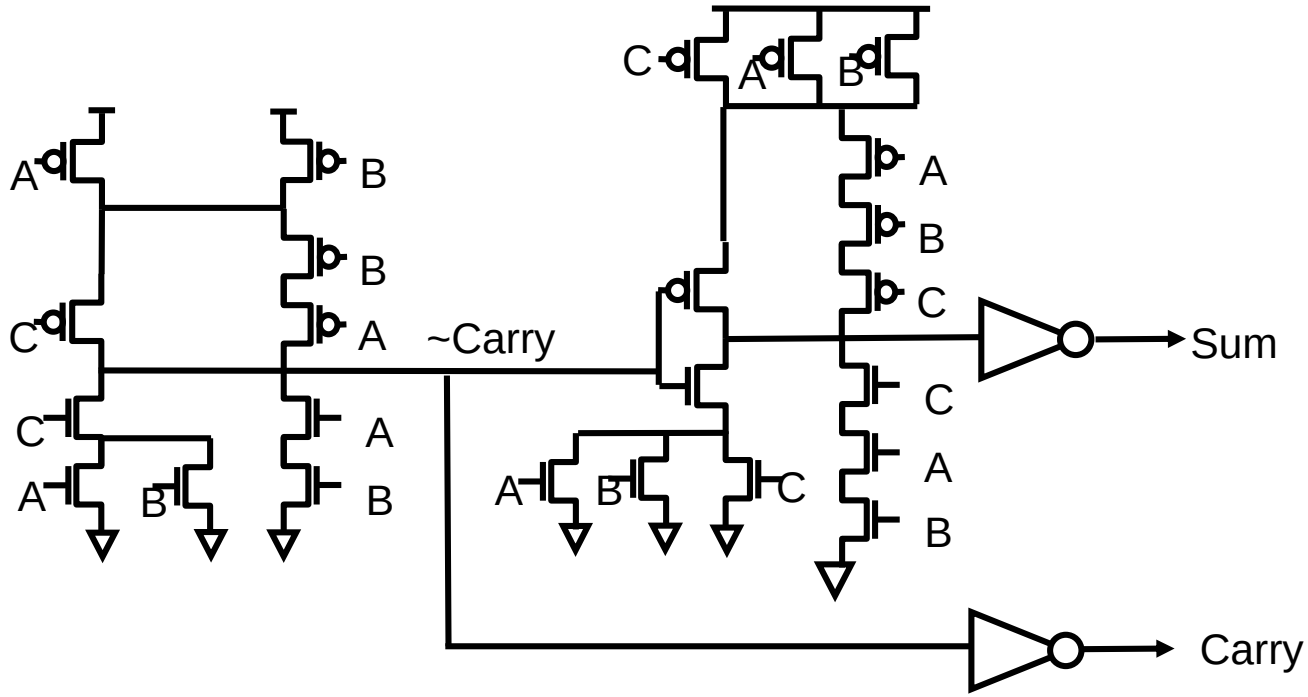
a	b	c	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



a	b	c	~Cout	~S
0	0	0	1	1
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	0	0

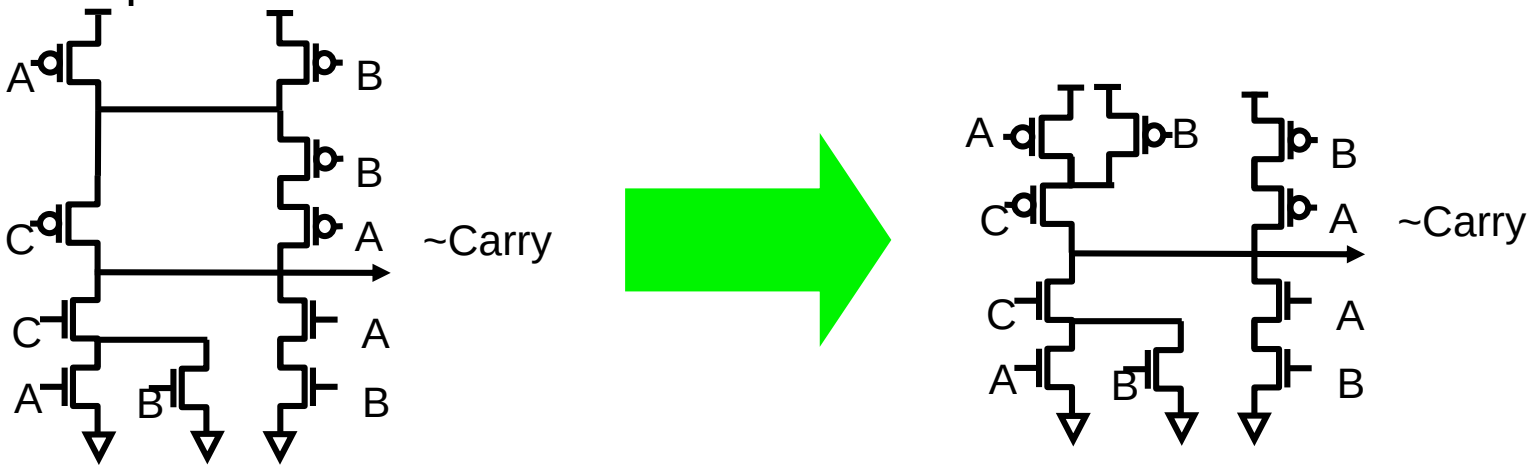
+ Full Adder With Reused Carry Term

- 28 Transistors (including inverters for Carry and Sum)
- Carry portion is identical to previous version
  - We inverted the inputs to get ~Carry



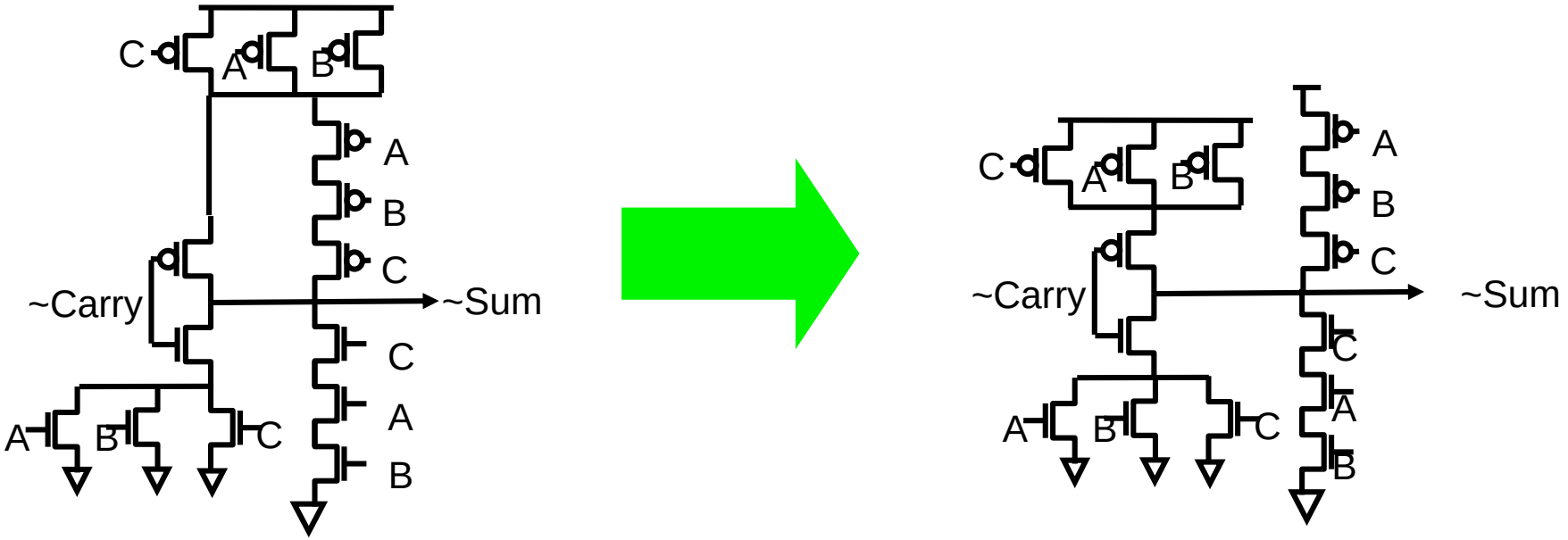
+ Subtle Improvement of Previous Carry Circuit

- We can make one more PMOS transformation
  - PMOS and NMOS are no longer duals, but it still works correctly
- Two advantages:
  - Three series PMOS transistors become two
  - Reduced capacitance on source of pMOS connected to C input



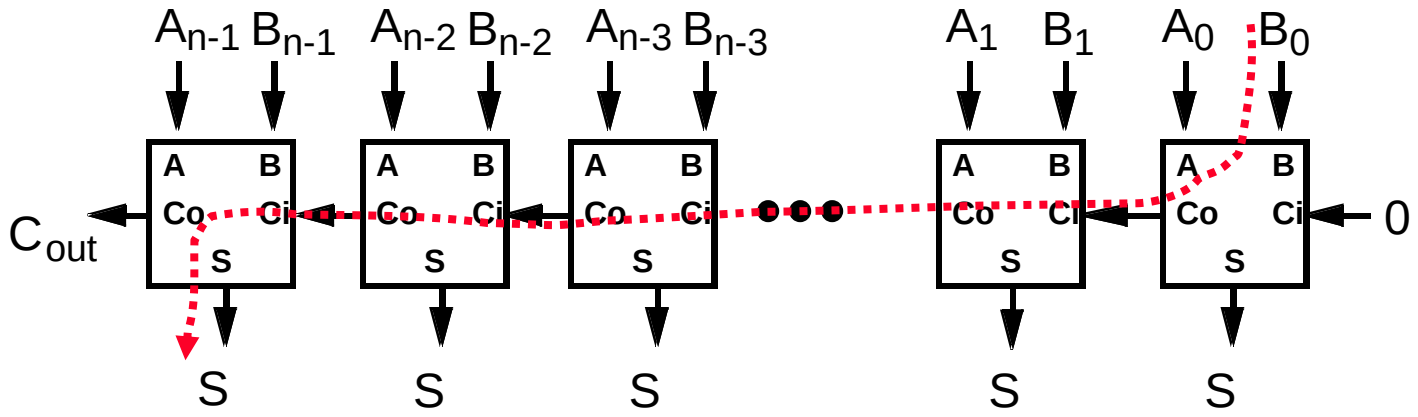
+ Improvement of Previous Sum Circuit

- Same type transformation as with Carry
- Similar advantages



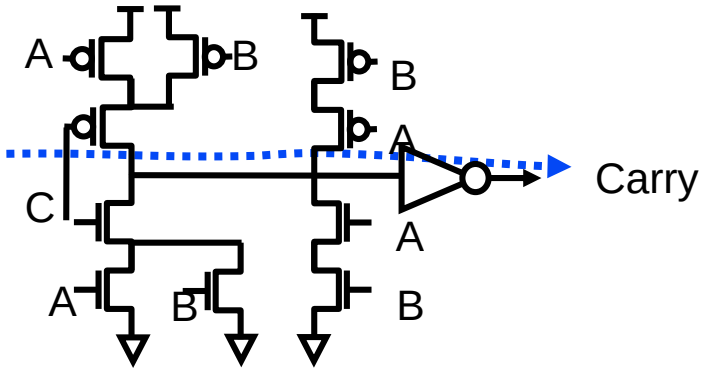
Critical Path of Ripple Carry Adders

- The final MSB can depend on the first carry
  - Consider the case of adding 0000001 to 01111111



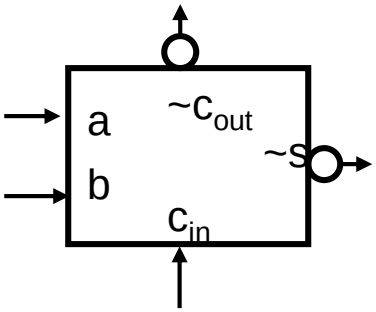
Critical Path of Ripple Carry Adders

- Our critical path has two gates per stage
  - One complex gate plus an inverter
- Can we reduce this to one?
  - If we can, do we want to?

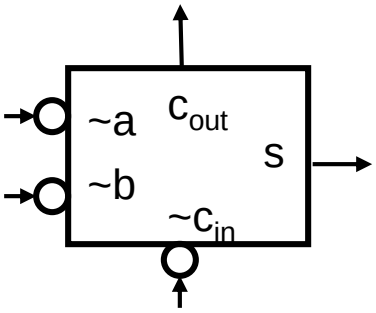


We Can: Inverting the Inputs

- What happens if we invert the inputs to the previous cell?
  - The outputs go from inverting to non-inverting
  - The same basic full adder cell can either take an inverted carry and produce a non-inverted one, or vice-versa



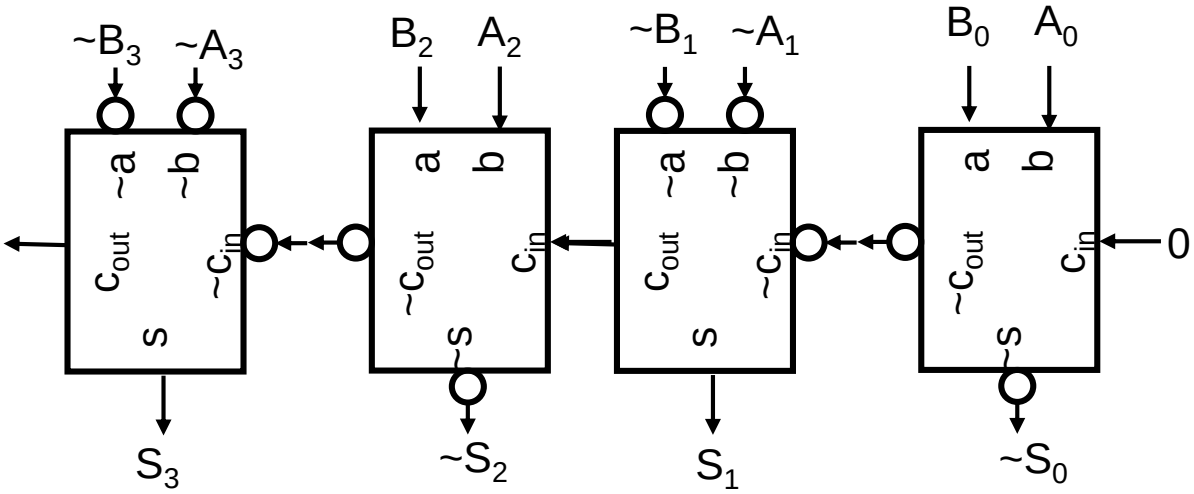
a	b	c	~Cout	~S
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	0



~a	~b	~c	Cout	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

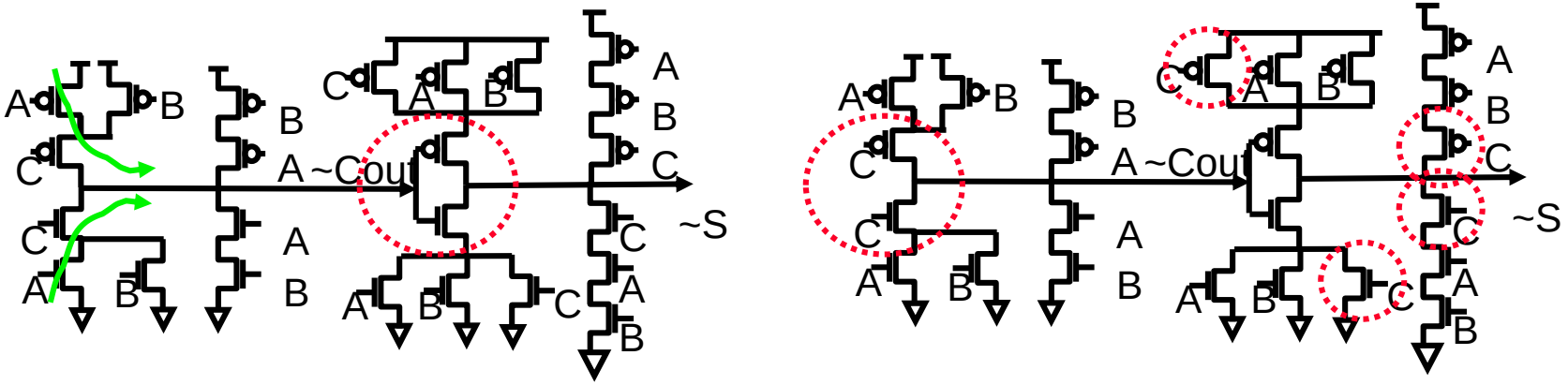
Alternating True and Complement Carries

- Concatenating Full Adders without inverters works fine
  - Inversions on carry chains cancel one another out
  - Need to invert sum output on even numbered stages
  - Need to invert A and B inputs on odd numbered stages
    - Not in critical path



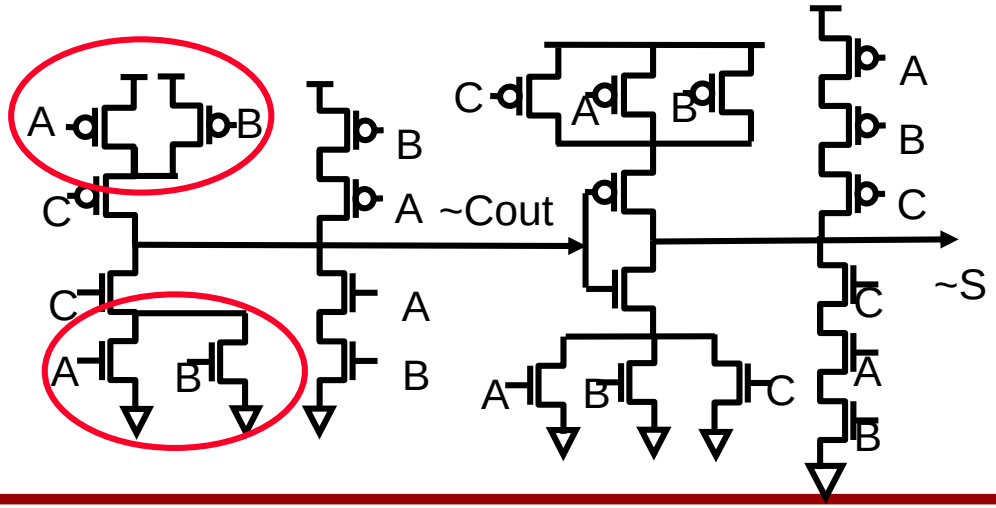
Evaluating Inverting Versus Non Inverting

- Fewer Stages is not necessarily better
  - Need to look at the total path effort
  - Find the “appropriate” number of stages
- Each Cout drives two gates
  - Next stage, and the sum gate (all the red circles)
- What is the stage effort?
  - Depends on the relative sizing of the devices



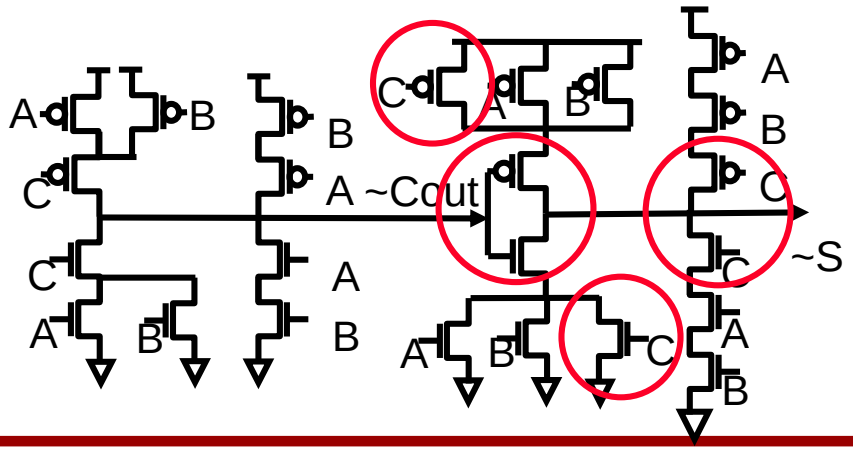
Speeding up the Inverting Full Adder

- Key is to reduce logical effort and branching effort
  - Slow down A and B paths to speed up Carry path
- Reducing logical effort
  - Make circled devices wide to reduce their resistance
    - Reduces logic effort for the C input
    - In the limit, LE will come close to 1 (inv)



Speeding up the Inverting Full Adder

- Reducing branching effort
  - Make circled devices small to reduce their loading
    - Since min width is usually specified, need to make the main drive large compared to the side loads



Layout

- Clever layout circumvents usual line of diffusion
  - Use wide transistors on critical path
  - Eliminate output inverters