18-447 Lecture 2:
Computer Arithmetic: Adders

James C. Hoe
Dept of ECE, CMU

January 14, 2009

Announcements:	No class on Monday
	Verilog Refresher next Wednesday
	Review P&H Ch 3
Handouts:	Lab 1 and HW1 will be posted on Blackboard this weekend

Binary Number Representation

- Let $b_{n-1}b_{n-2}...b_2b_1b_0$ represent an n-bit unsigned integer
  - its value is $$\sum_{i=0}^{n-1} \underbrace{\overset{\text{weight of the i'th digit}}{2^i} b_i}_{\text{value of the i'th digit}}$$
  - a finite representation between 0 and 2^n-1
  - e.g., $1011_{two} = 8_{ten} + 2_{ten} + 1_{ten} = 11_{ten}$
    (more commonly rewritten as b'1011=11)
- Often written in Hex for easier human consumption
  - to convert, starting from the LSB, map 4 binary digits at a time into a corresponding hex digit; and vice versa
  - e.g., 1010_1011_{two}=AB_{hex}

For converting between binary and decimal, memorize decimal values of $2^0 \sim 2^{10}$, and remember 2^{10} is about 1000.

2's-Complement Number Representation

- Let $b_{n-1}b_{n-2}\ldots b_2b_1b_0$ represent an n-bit signed integer
  - its value is

$$-2^{n-1}b_{n-1}+\sum_{i=0}^{n-2}2^i b_i$$

  - a finite representation between -2^{n-1} and $2^{n-1}-1$
  - e.g., assume 4-bit 2's-complement

    b'1011 = -8 + 2 + 1 = -5

    b'1111 = -8 + 4 + 2 + 1 = -1

- To negate a 2's-complement number
  - add 1 to the bit-wise complement
  - assume 4-bit 2's-complement

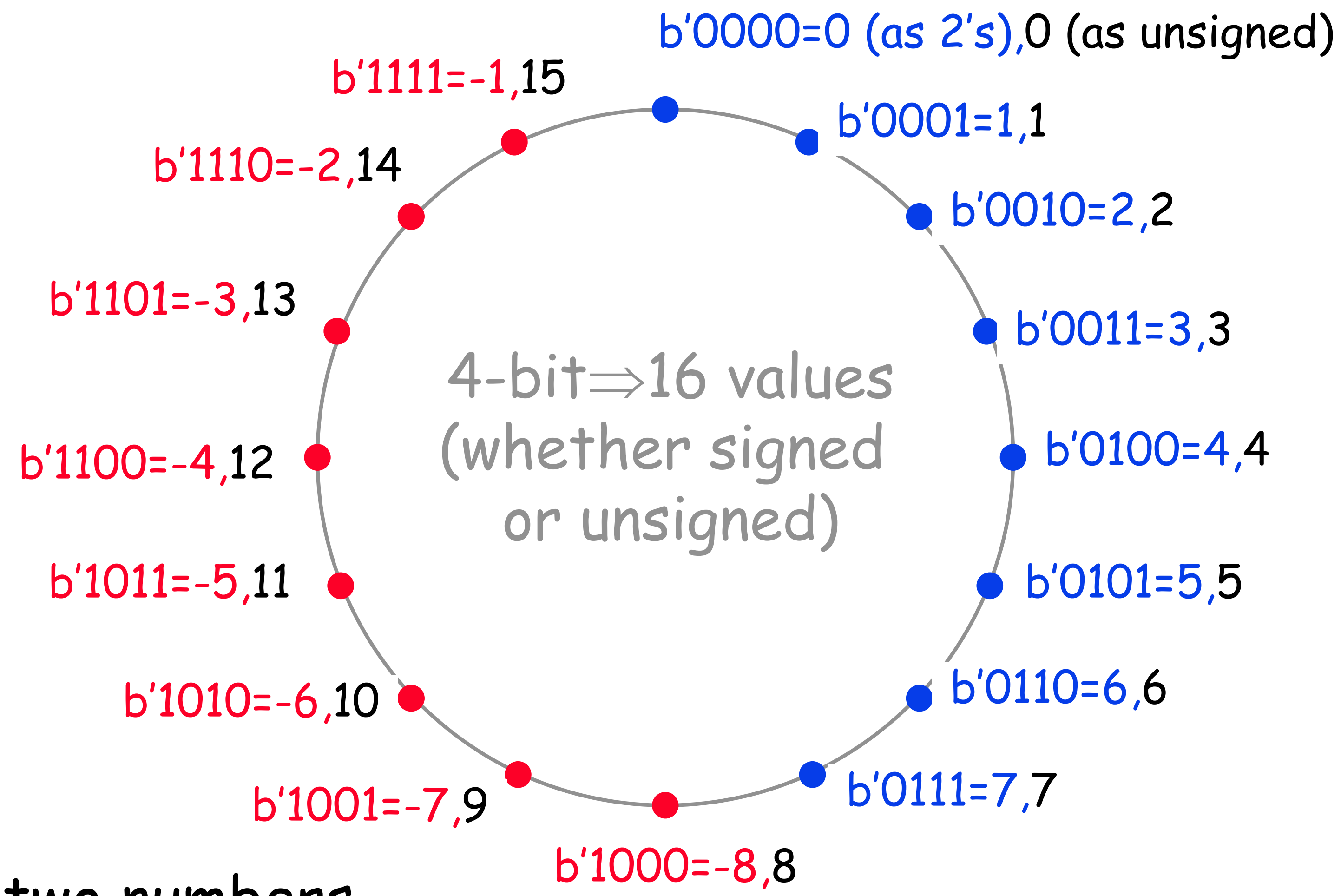
    (- b'1011) = b'0100 + 1 = b'0101 = 5

    (- b'0101) = b'1010 + 1 = b'1011 = -5

    (- b'1111) = b'0000 + 1 = b'0001 = 1

    (- b'0000) = b'1111 + 1 = b'0000 = 0

Intuition: a 4-bit example



- how to add two numbers
- what it means to "overflow" the number representation
- how to negate a number

Yes, 0 is a positive number in CS

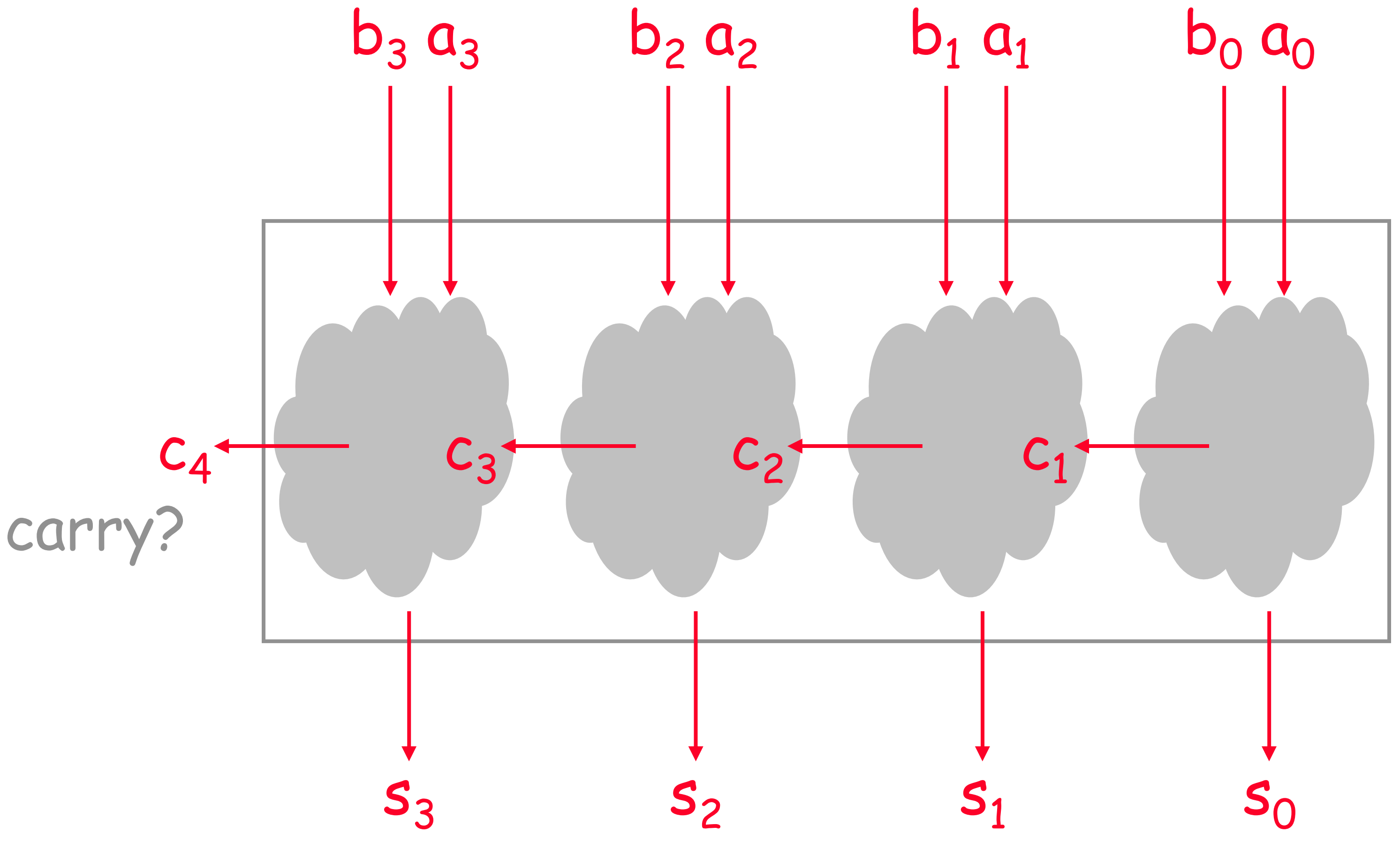
Smaller to Larger Binary Representation

- Unsigned numbers
  - pad the left with as many 0s as you need (aka 0-extension)
    e.g. 4'b1111 → 8'b0000_1111
- 2's-complement numbers
  - positive: pad the left with as many 0s as you need
  - negative: pad the left with as many 1s as you need
    e.g. 4b'1111 → 8'b1111_1111
    4b'1110 → 8'b1111_1110
  - or generically, pad the left with the same value as the original sign-bit as many times as necessary (aka signed-extension)

What about converting from larger to smaller representation?

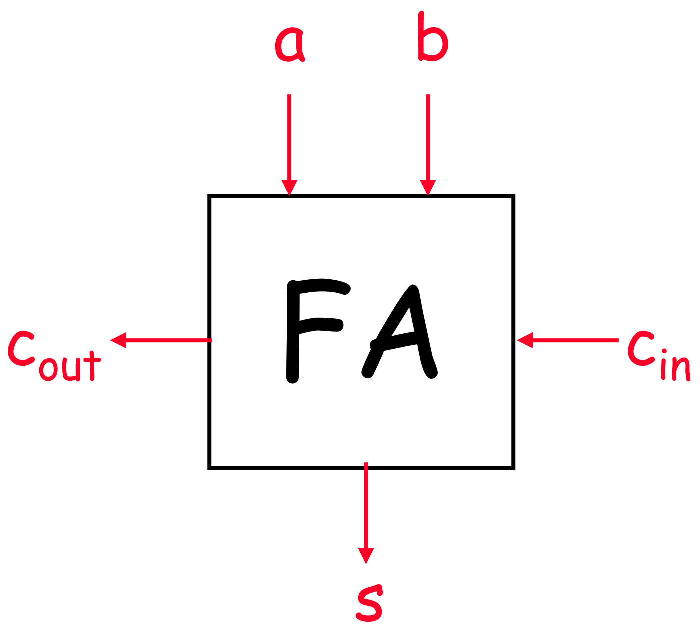
(Unsigned) Binary Addition

- Long Hand



What about subtraction?

Full Adder



c_{in}	a	b	c_{out}	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

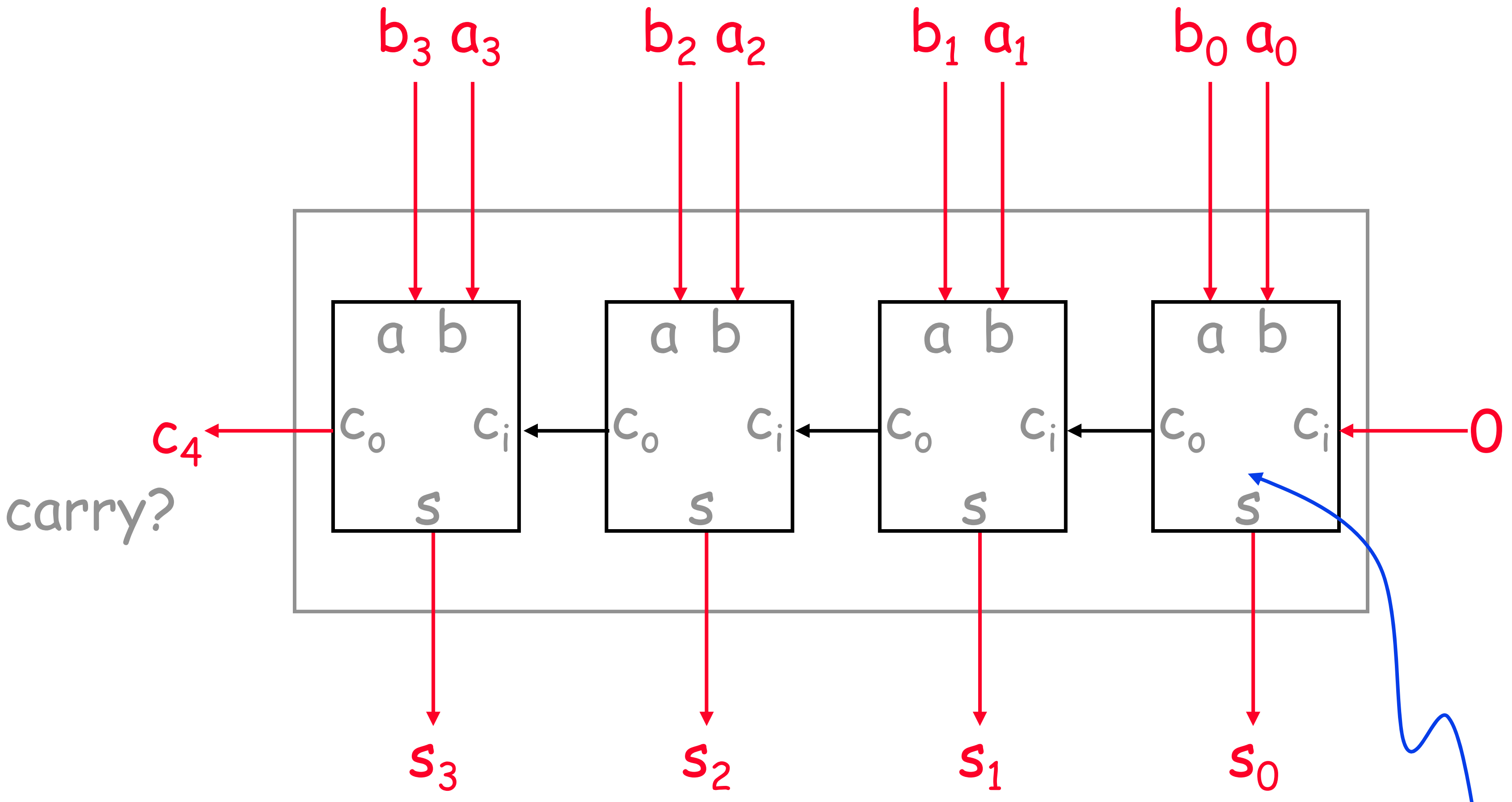
3-way majority   parity

$s = a \oplus b \oplus c_{in}$

$c_{out} = bc_{in} + ac_{in} + ab$

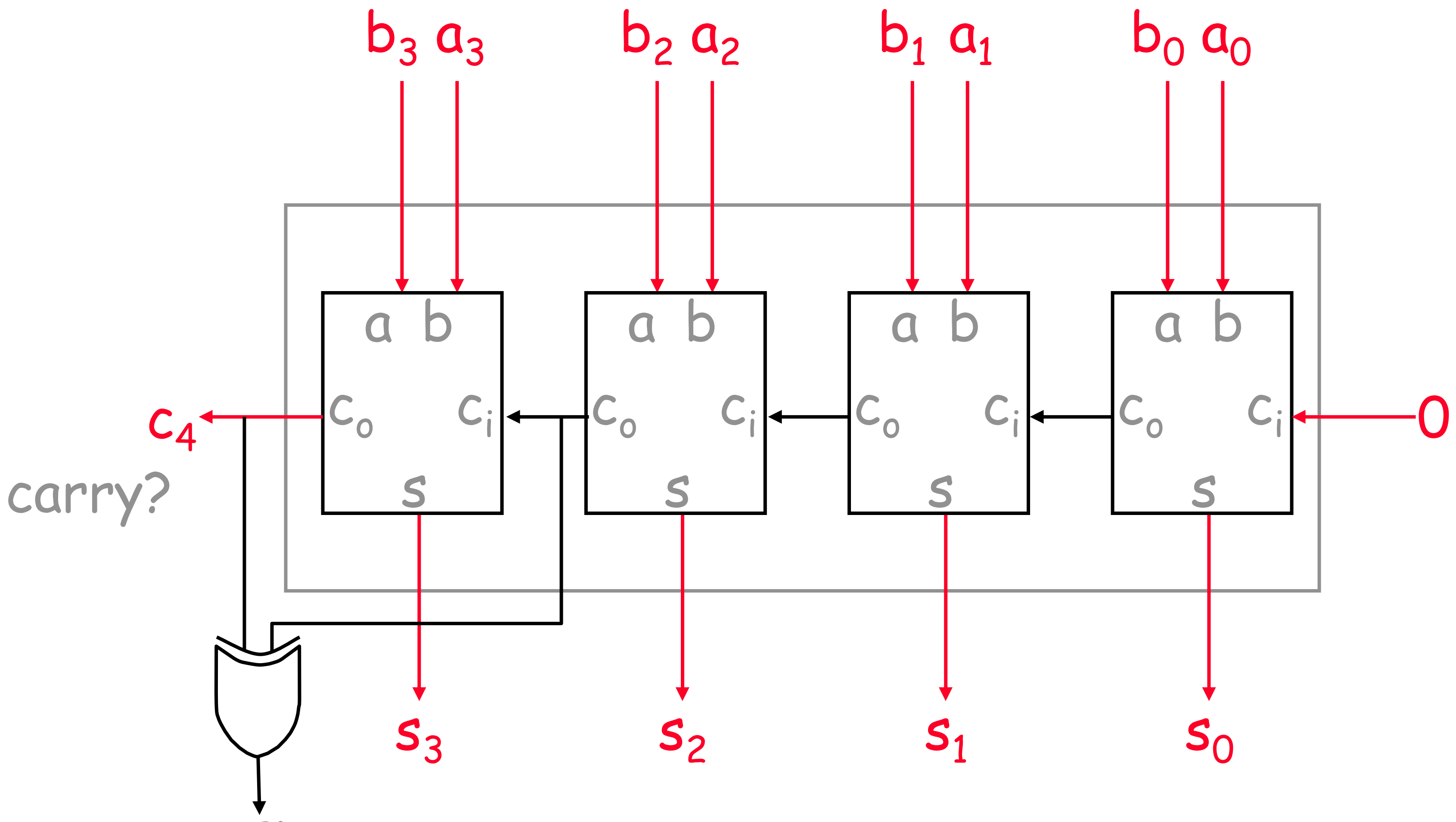
a, b, c_{in} are functionally indistinguishable as inputs

Unsigned Binary Addition



Could use a "half-adder", but let's wait

2's-Complement Addition

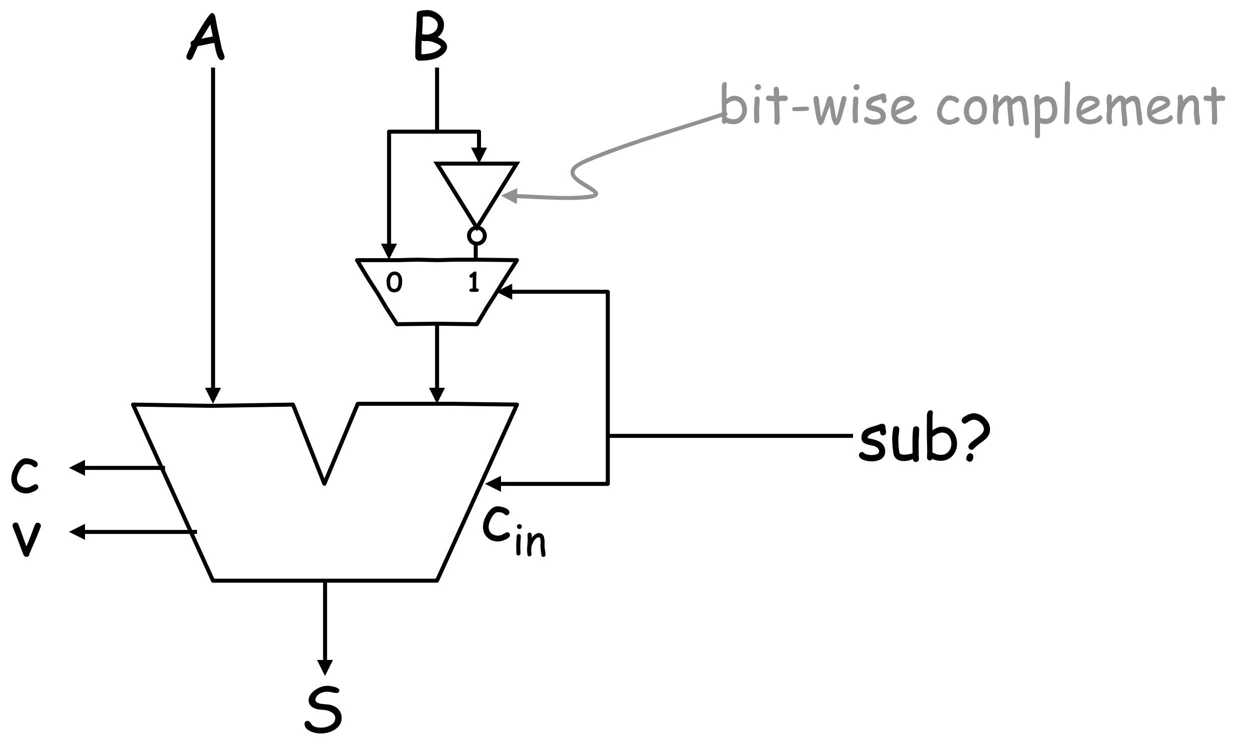


overflow?= $(a_3 \oplus b_3)$? 0 : $(a_3 \oplus s_3)$

- can't overflow when adding a pos. and a neg. number
- if 2 pos. numbers yield a neg. number $\Rightarrow$ V; vice

2's-Complement Subtraction

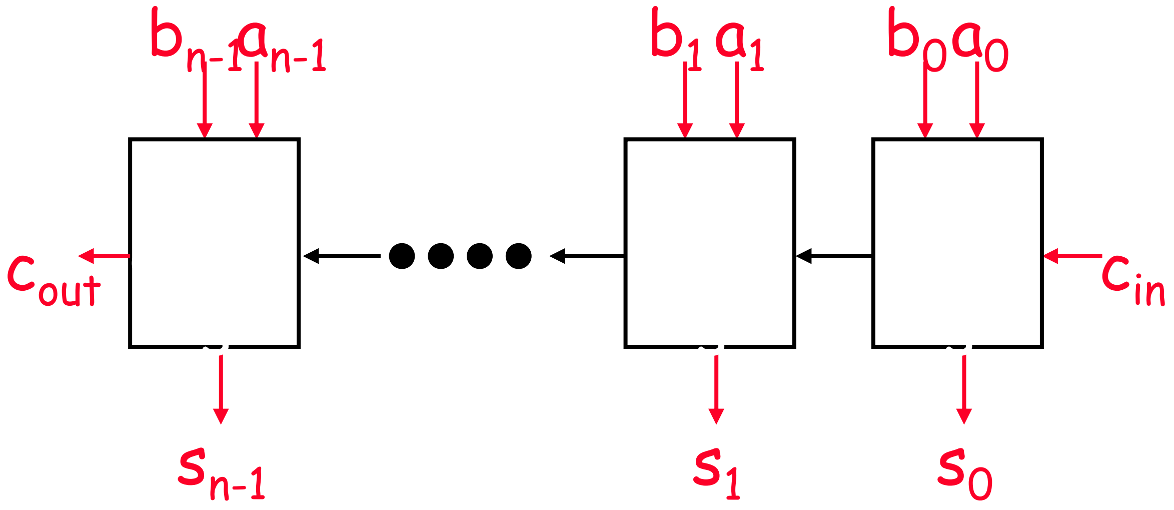
- Subtracting is like adding the negative
- Negation is easy in a 2's-complement representation



How do you build a comparator (i.e., >, <)?

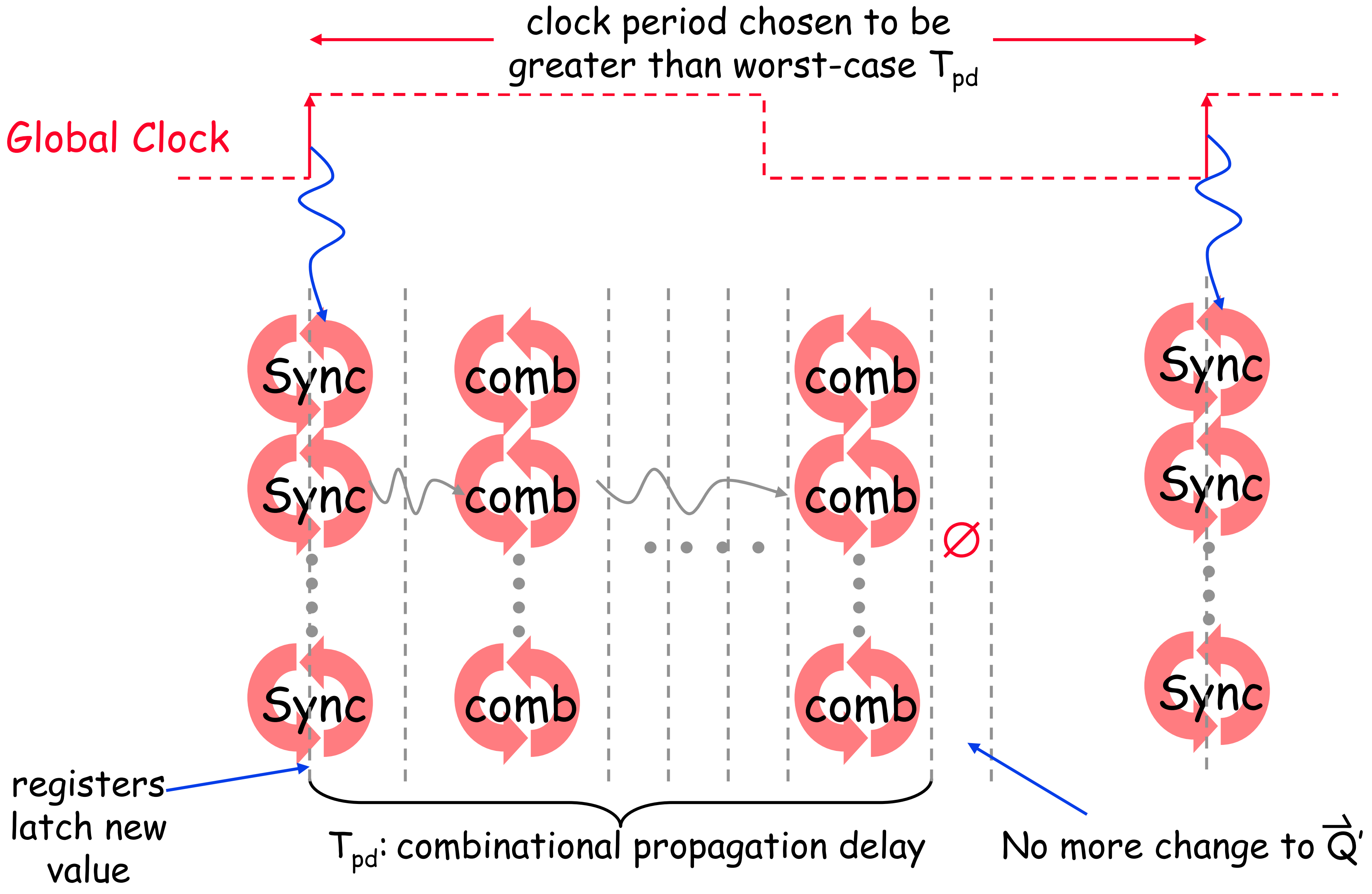
Analysis of an n-bit "Ripple-Carry" Adder

- Size/Complexity: O(n)
  - n x SizeOf(Full Adder)
- Critical Path Delay: O(n)
  - n x DelayOf(Full Adder)
  - n x 2 gate delays (assuming 2-level SOP is used)



Electrical & Computer ENGINEERING

18-447 Simplified Timing


clock period chosen to be greater than worst-case T_{pd}
Global Clock
Sync
comb
∅
registers latch new value
T_{pd}: combinational propagation delay
No more change to $\vec{Q}'$

What about setup-time, hold-time, skew and such?

High-Performance Adder?

- Intel P4 is designed around a clock period that is twice the 16-bit adder latency

- Using a rough estimation

  gate delay ≈ 0.5 ns-per-micron x feature-size

  a 90nm process has gate delay = 45ps

- If Intel used a ripple-carry adder then P4 should be running ~ 1/ (2x2x16x45ps) = 347MHz

- Alternatively speaking, 3GHz P4 would have to add 2 16-bit numbers in ~4 gate delays

Cutting Down the Carry Chain

- How to reduce the carry-propagation delay?

  Remember, long-hand is how most of us add, but not the only way

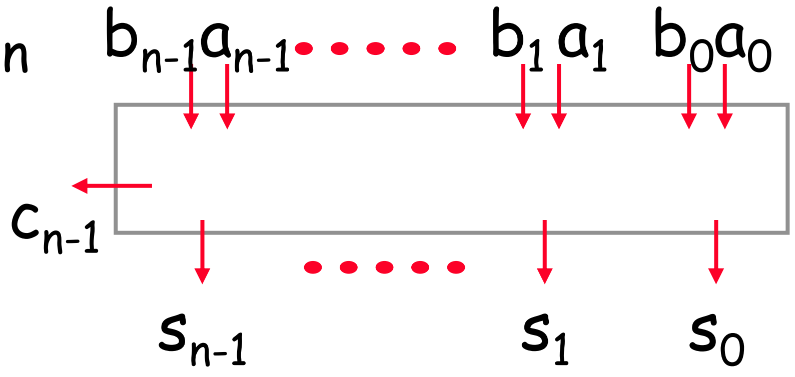
- Can we compute an intermediate carry signal without first computing the earlier ones
  - e.g., let c_m (or s_m) be a function of $a_m....a_0$ and $b_m....b_0$

    $c_2 = (a_1a_0b_0)+(a_1a_0c_0)+(a_1b_0c_0)+(b_1a_0b_0)+(b_1a_0c_0)$
    $+(b_1b_0c_0)+(a_1b_1)$
  - Complexity grows exponentially in n

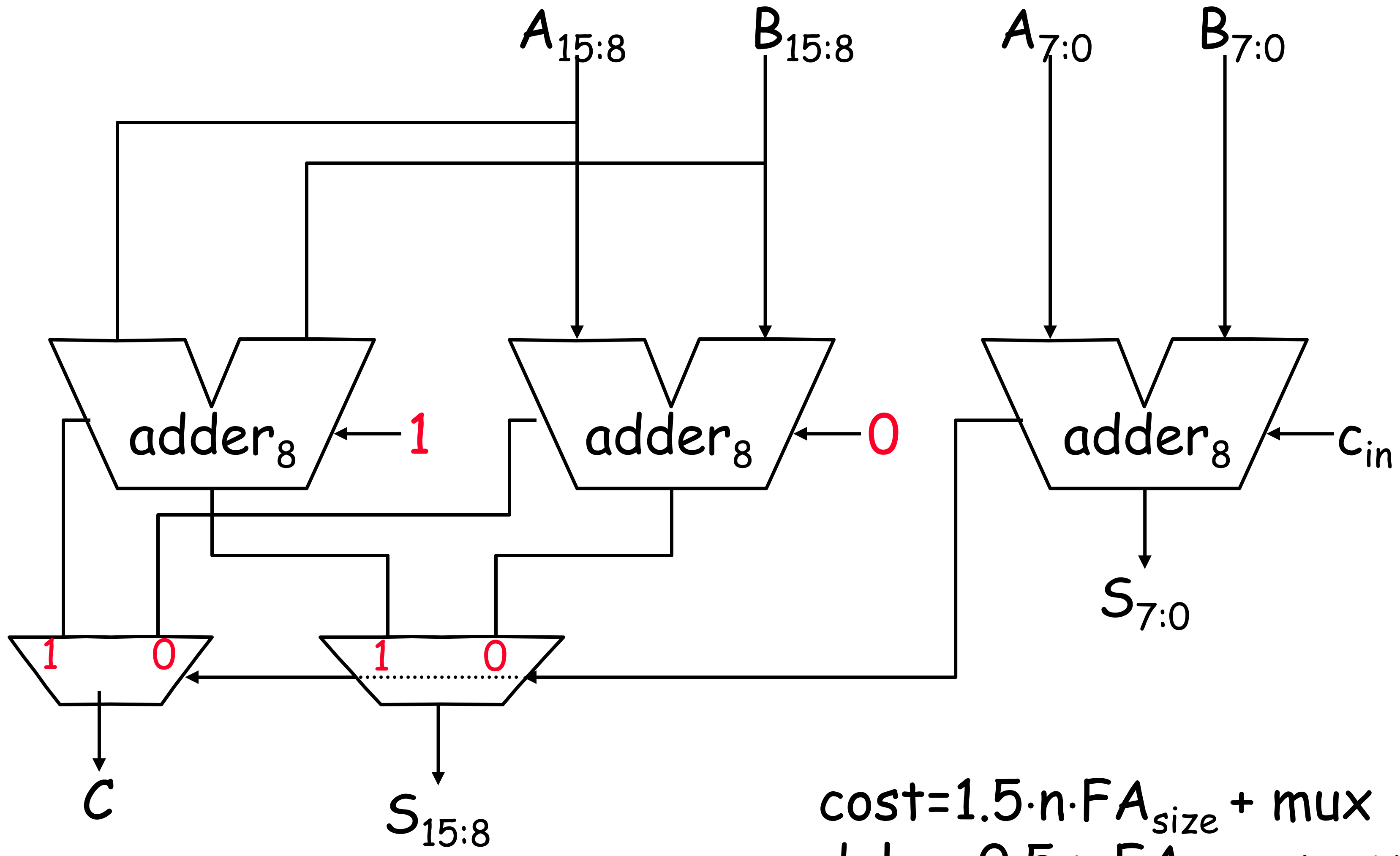
    exponential isn't too bad for small n's
  - gate delay is 2, independent of n

    true for small n's

What about large n's?



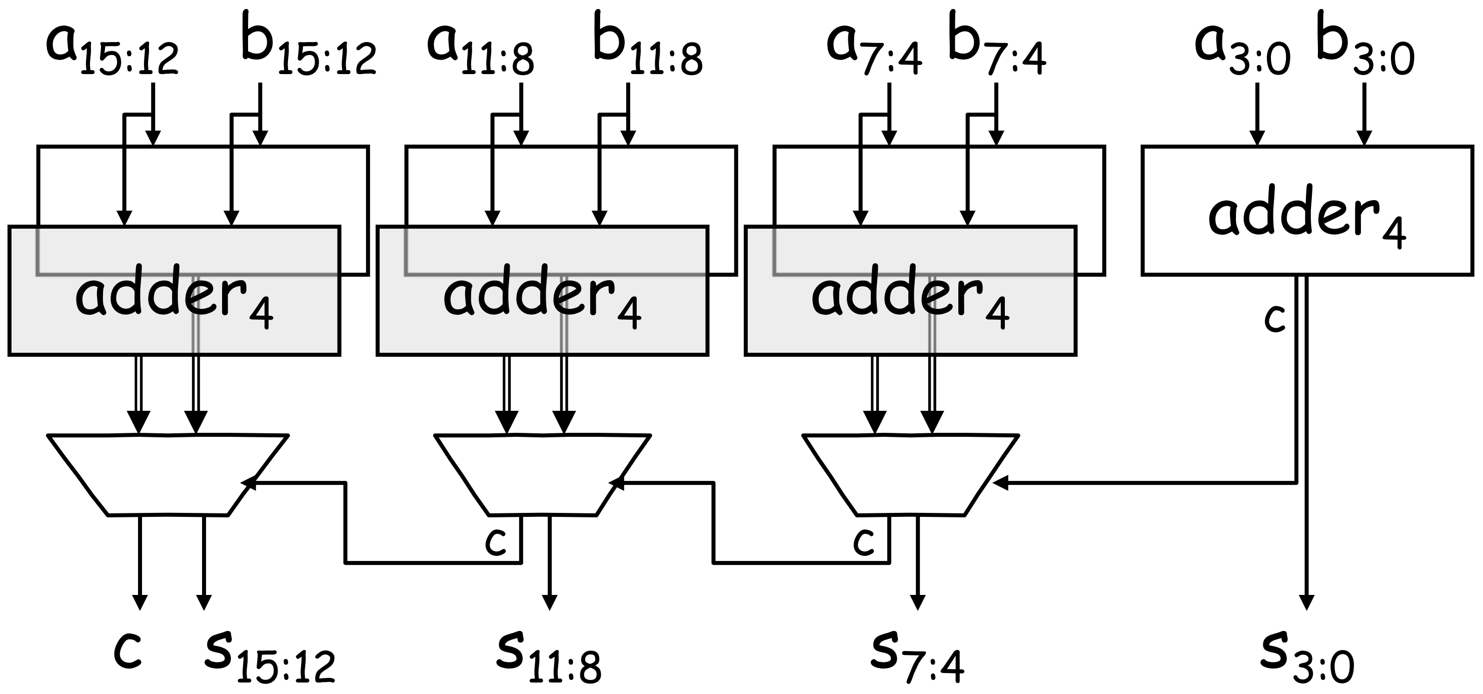
Carry-Select Adder



3 adders operate in parallel!!

cost=$1.5 \cdot n \cdot FA_{size}$ + mux
delay=$0.5 \cdot n \cdot FA_{delay}$+mux-delay
if n=16 $\Rightarrow$ ~16 gate-delays

Multi-Stage CSA



$\text{cost}=(2k-1)/k \cdot n \cdot FA_{size}$ + mux's --- for k-stage

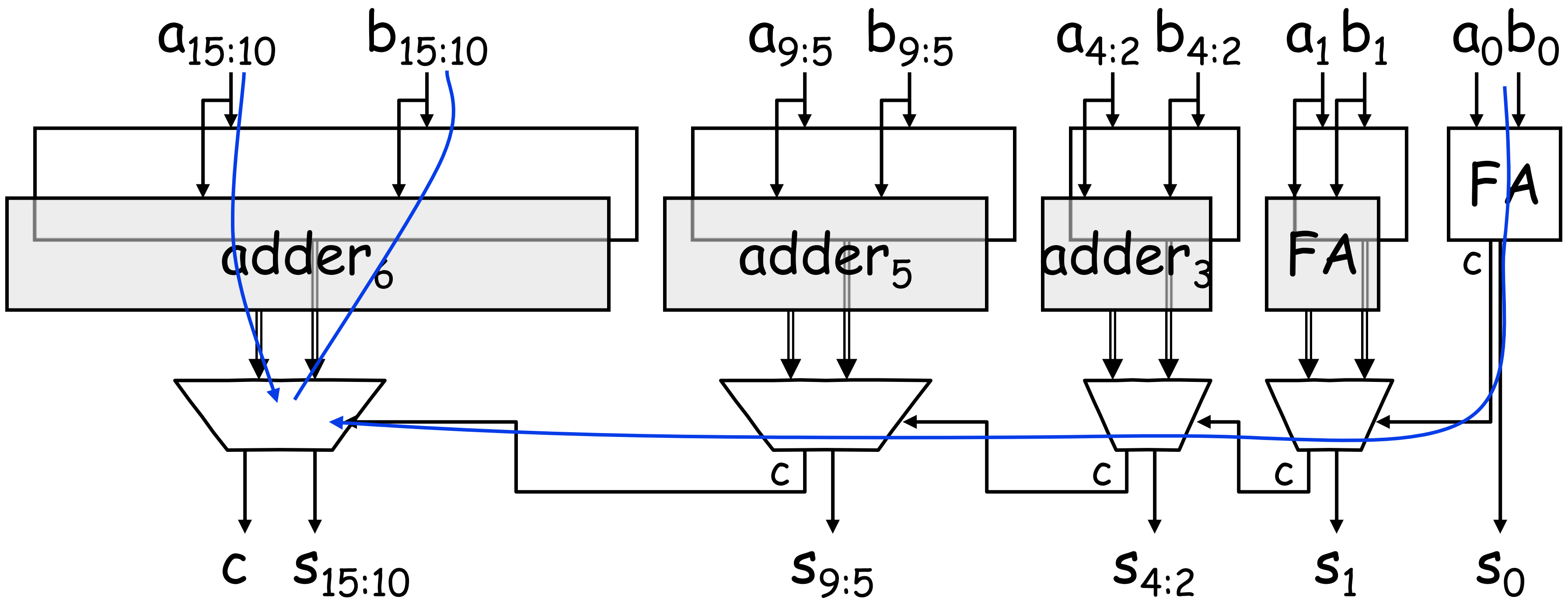
$\text{delay}=n \cdot FA_{delay}/k + (k-1) \cdot$ mux-delay

$k=4, n=16 \Rightarrow$ ~8 gate-delay + 3 mux-delay

$k=8, n=16 \Rightarrow$ ~4 gate-delay + 7 mux-delay

$k=16, n=16 \Rightarrow$ ~2 gate-delay + 15 mux-delay

Variable-Length CSA



-doubles the cost
-delay set by the longest adder stage, grows by $O(n^{1/2})$ with careful critical path tuning

Can we have cut-down the carries without 2x cost?

Carry Generate and Propagate

c_{in}	a	b	c_{out}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

c_{in}	a	b	c_{out}
X	0	0	0
c_{in}	0	1	c_{in}
c_{in}	1	0	c_{in}
X	1	1	1

- If $a \cdot b$ then c_{out} is 1 regardless of c_{in} (carry generate)
- if $a \oplus b$ then c_{out} is the same as c_{in} (carry propagate)

$g_i = a_i \cdot b_i$

$p_i = a_i \oplus b_i$ local decisions based on a_i and b_i only

Small Carry-Look-Ahead Adder

Given $g_i = a_i \cdot b_i$

$p_i = a_i \oplus b_i$ $\qquad c_{i+1} = g_i + (p_i \cdot c_i)$

Thus

$c_1 = g_0 + (p_0 \cdot c_0)$

$c_2 = g_1 + (p_1 \cdot c_1) = g_1 + (p_1 \cdot (g_0 + (p_0 \cdot c_0))) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 c_0$

$c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 c_0$

$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 c_0$

and so on

- We can compute c_n in $O(\log n)$ gate delay and $O(n^2)$ size, only manageable for small n
- Given c_n we can compute s_n for a constant additional delay

Prefix Carry-Look-Ahead

Given

$c_1=g_0+(p_0\cdot c_0)$

$c_2=g_1+(p_1\cdot c_1)=g_1+(p_1\cdot(g_0+(p_0\cdot c_0)))=g_1+p_1\cdot g_0+p_1\cdot p_0c_0$

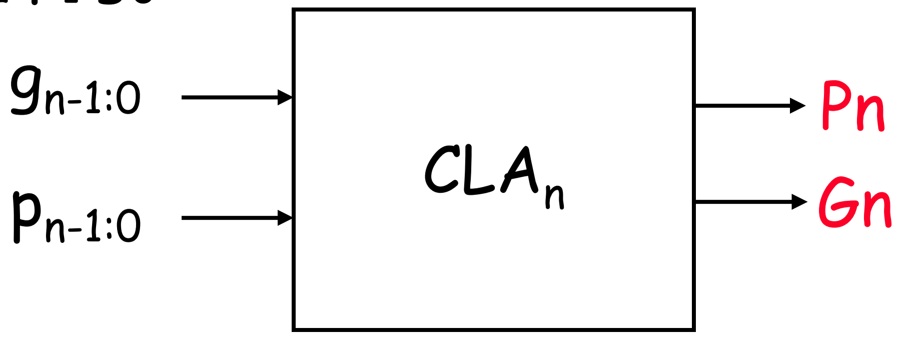
$c_3=g_2+p_2\cdot g_1+p_2\cdot p_1\cdot g_0+p_2\cdot p_1\cdot p_0c_0$

$$c_4=\underbrace{g_3+p_3\cdot g_2+p_3\cdot p_2\cdot g_1+p_3\cdot p_2\cdot p_1\cdot g_0}_{G}+\underbrace{p_3\cdot p_2\cdot p_1\cdot p_0}_{P}c_0$$

As a 4-arity group

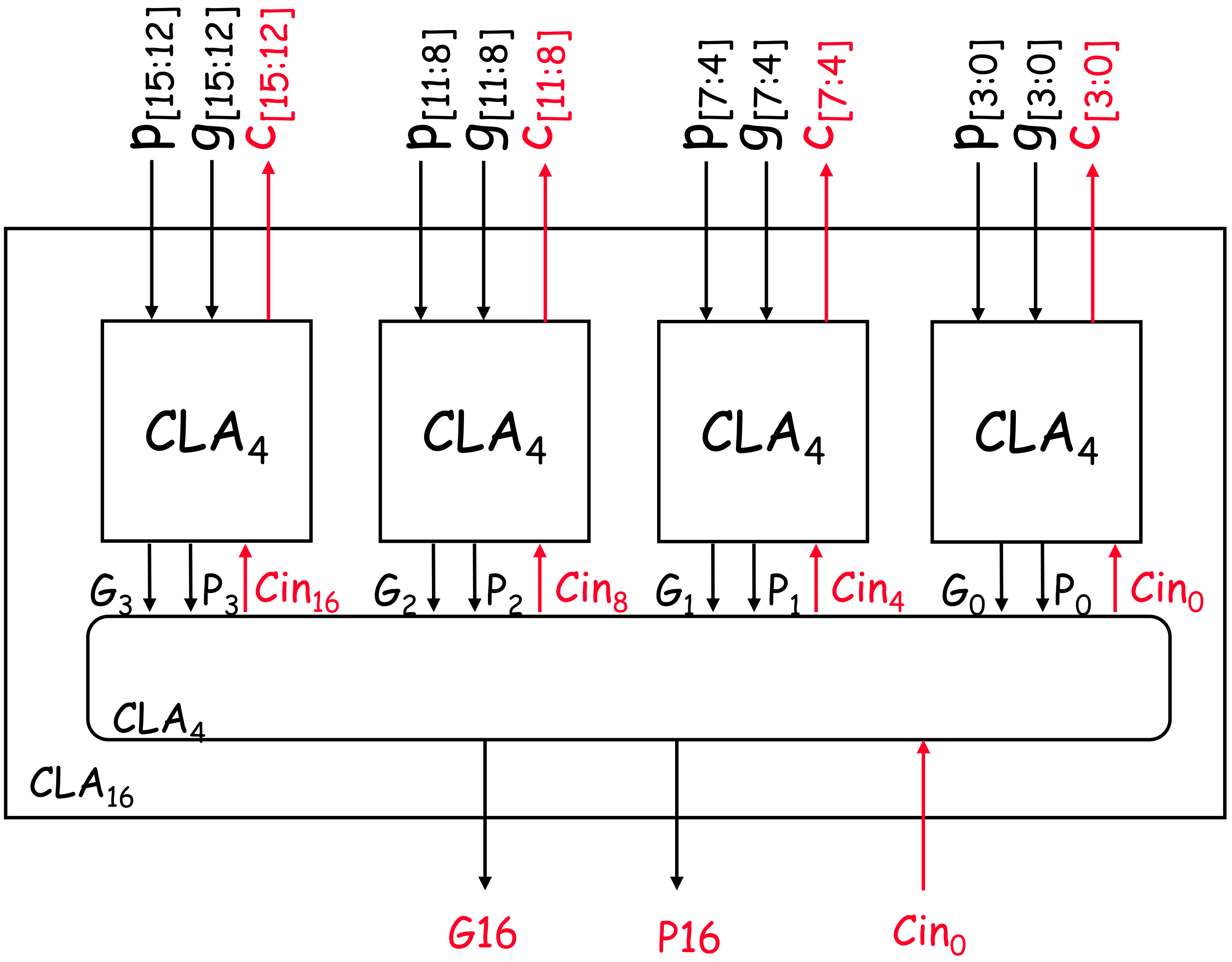
$G4=g_3+p_3\cdot g_2+p_3\cdot p_2\cdot g_1+p_3\cdot p_2\cdot p_1\cdot g_0$

$P4=p_3\cdot p_2\cdot p_1\cdot p_0$



Electrical & Computer ENGINEERING

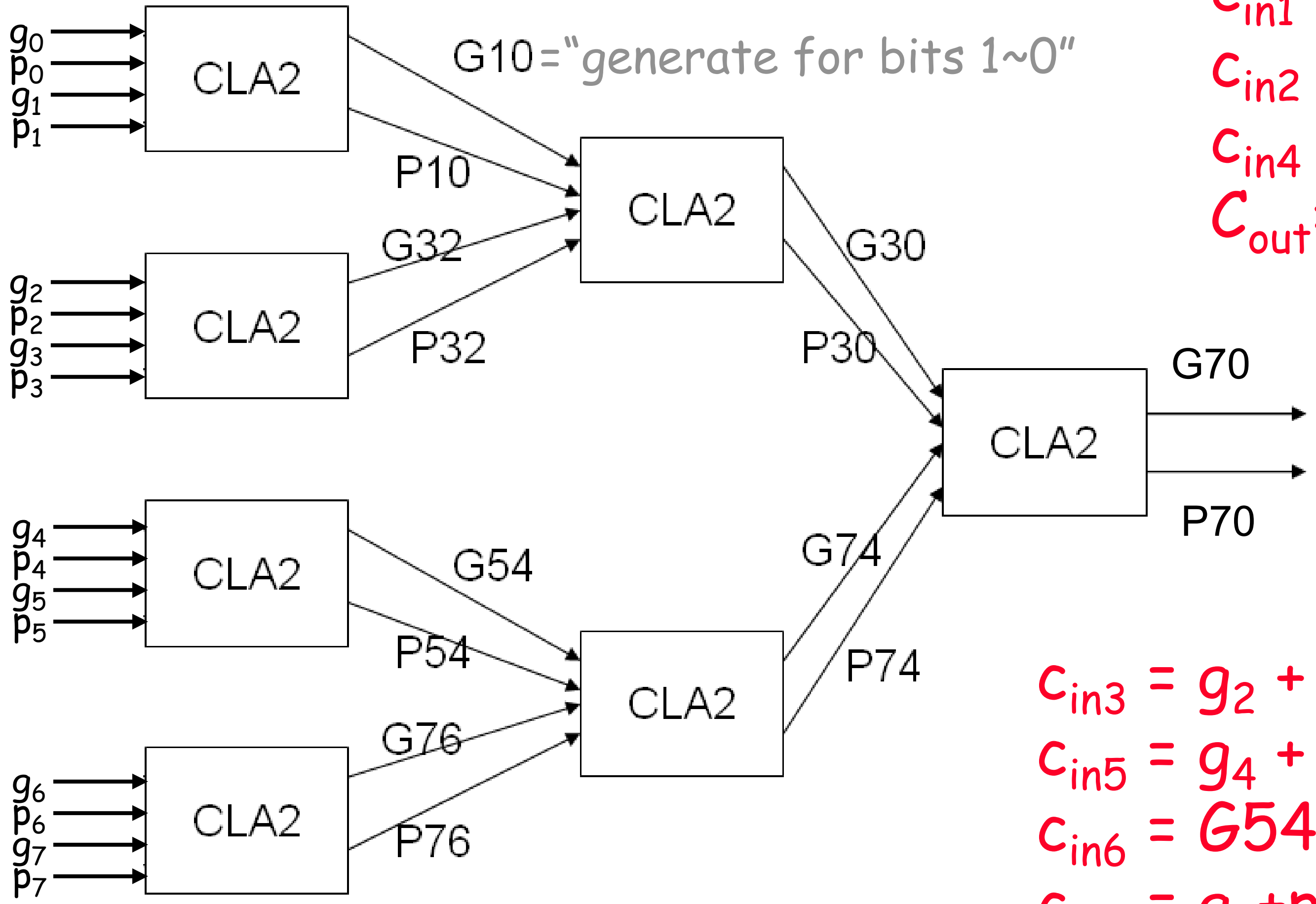
Prefix Carry-Look-Ahead



This structure can be recursed: O(log n) delay, O(n) size

Computing Individual Carries

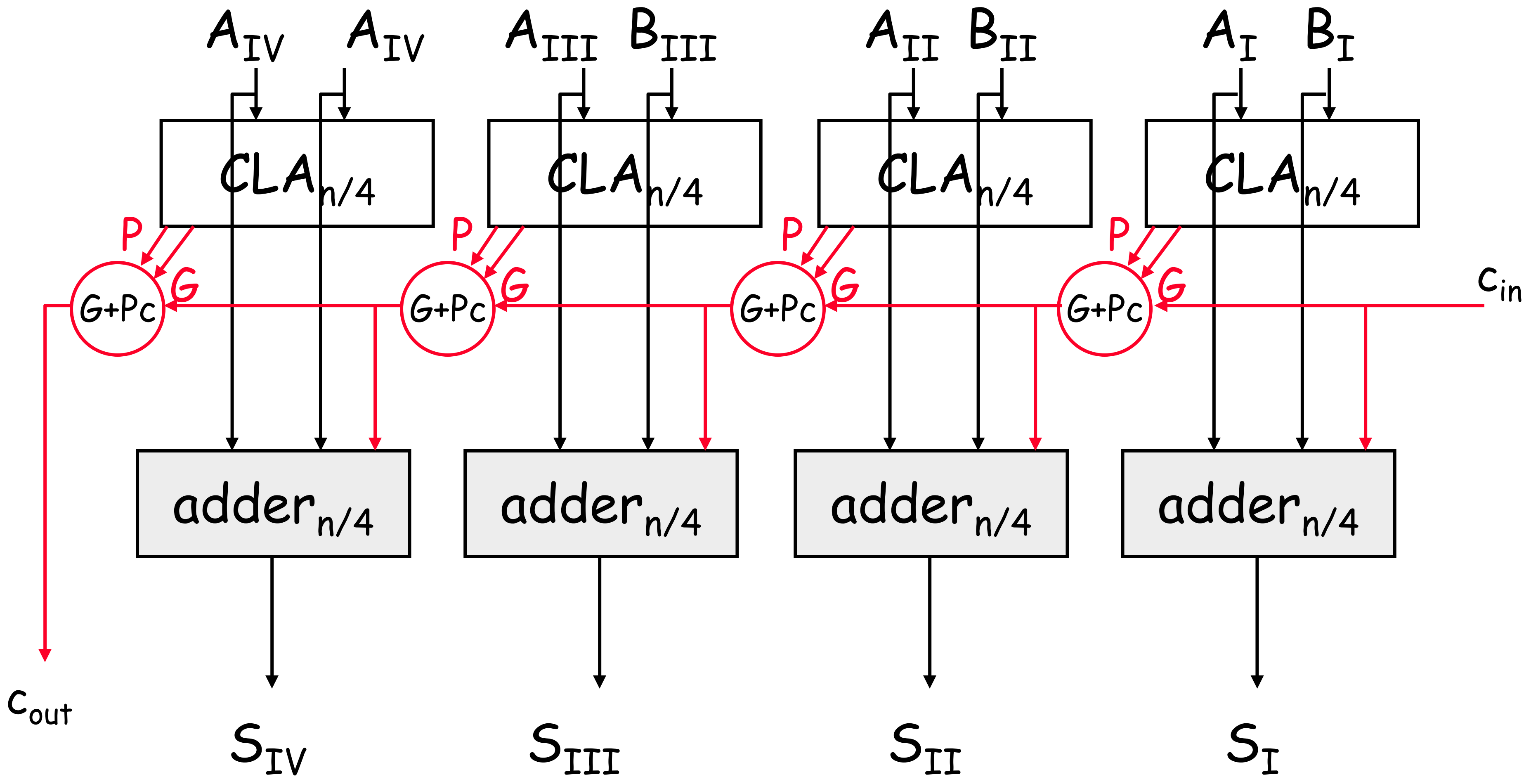
Example: 8-bit, 2-ary CLA



c_{in0} = Cin
$c_{in1} = g_0 + p_0 \cdot Cin$
$c_{in2} = G10 + P10 \cdot Cin$
$c_{in4} = G30 + P30 \cdot Cin$
$C_{out} = G70 + P70 \cdot Cin$

$c_{in3} = g_2 + p_2 \cdot c_{in2}$
$c_{in5} = g_4 + p_4 \cdot c_{in4}$
$c_{in6} = G54 + P54 \cdot c_{in4}$
$c_{in7} = g_6 + p_6(G54 + P54 \cdot c_{in4})$

Large Adder using Carry-Skip



Fast enough and cheaper than computing individual ci's by G.P.

Adder at a Glance

- Ripple Adder
  - O(n) size, O(n) delay
- Carry-Select Adder
  - O(n) size, O($n^{0.5}$) delay
- Carry-Look-Ahead Adder
  - O(n^2) size, O(log n) delay
- Prefix Adder
  - O(n) size, O(log n) delay
- But, remember all approaches have design sweet-spots and make different tradeoffs
- There also are circuit-level adder tricks

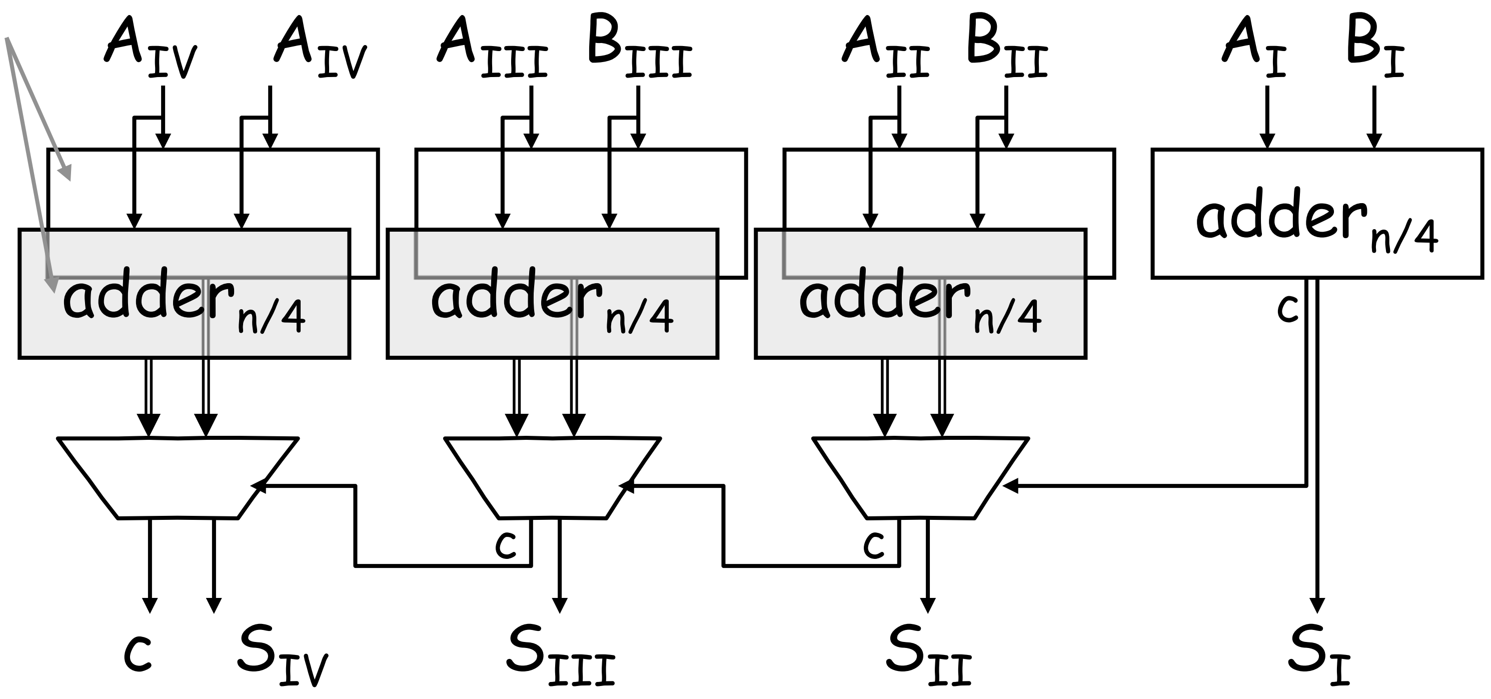
(e.g., Manchester carry chain)

Black Magic of Adder Design

- High-performance adder designs are extremely important to high-performance computing
- Studied extensively in theoretical frameworks
- Worked on extensively in practice
- Nevertheless remain very much a trial-and-error design exercise
- For a 64-bit adder, one might construct
  - adders of various (short) length using 2-level logic
  - a 16-bit adder from small adders with variable-length carry-select
  - a 32-bit adder from 2 16-bit CSA with CLA to determine carry for the upper 16 bits
  - a 64-bit 2-stage CSA adder from 3 32-bit adders

Building Wide Adders: the CSA approach

The fastest sub-adder you can muster



cost=(2k-1)·SizeOf(sub-adder) + mux's

delay=DelayOf(sub-adder)+(k-1)·mux-delay

-CSA pays ~2x the cost to avoid the carry delay

-Is there a cheap way to compute carry fast?