Topic 13

Arithmetic Components

Components to be discussed

- Arithmetic and logic unit (ALU)
- Carry lookahead adder
- Incrementer
- Magnitude comparator
- Multiplier

Arithmetic-Logic Unit: ALU

- ***ALU***: Component that can perform any of various arithmetic (add, subtract, increment, etc.) and logic (AND, OR, etc.) operations, based on control inputs
- Key component in computer

TABLE 4.2 Desired calculator operations

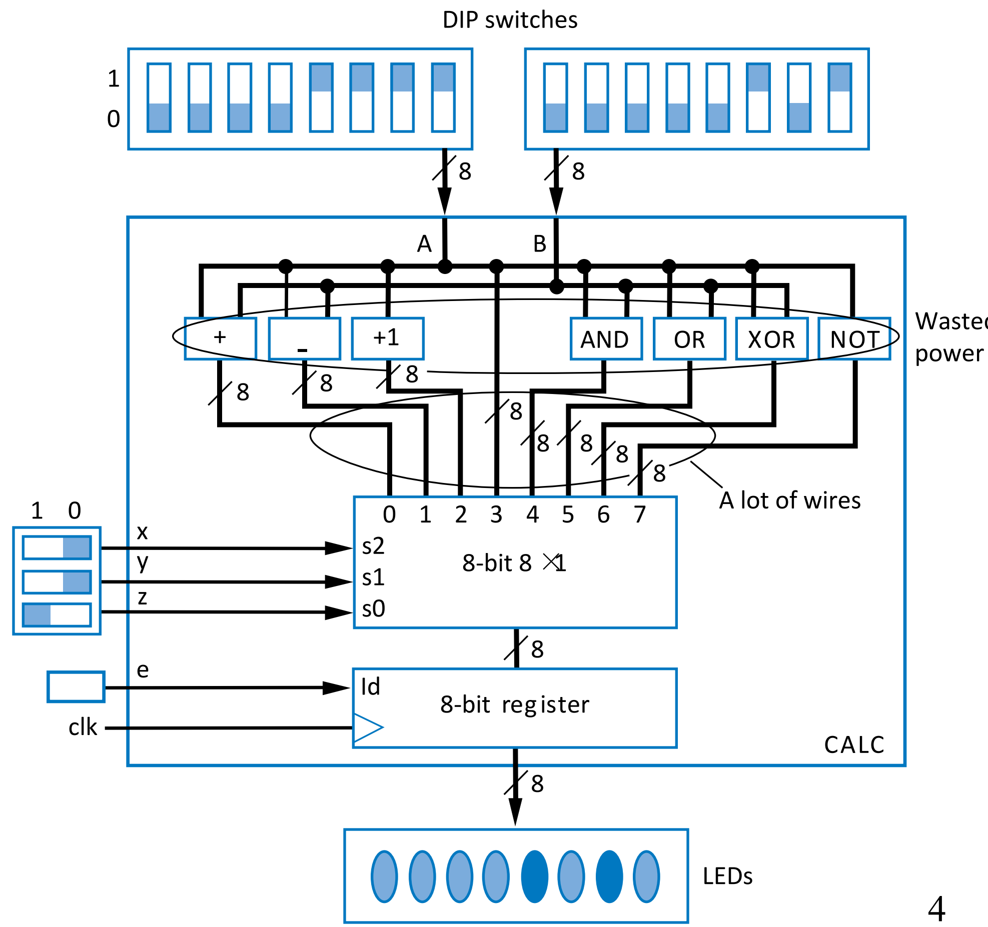
Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

Multifunction Calculator with an ALU

- ALU functions selected by a mux
  - But too many wires
  - Wasted power computing all those operations when at any time you only use one of the results

TABLE 4.2 Desired calculator operations

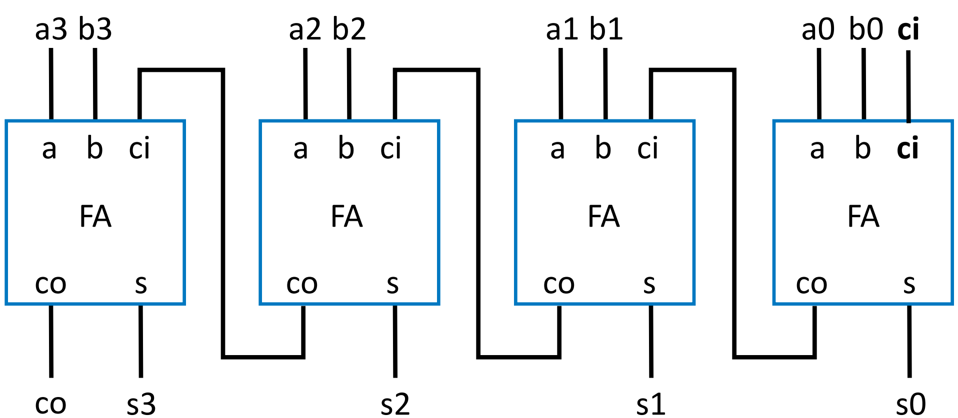
Inputs x	y	z	Operation	Sample output if A=00001111, B=00000101
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AN[redacted]	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000



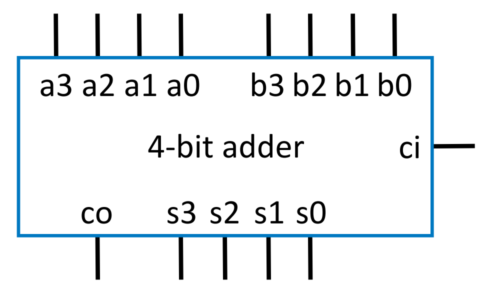
Carry-Ripple Adder

- Carry-ripple adder
  - 4-bit adder: Adds two 4-bit numbers, generates 5-bit output
    - 5-bit output can be considered 4-bit “sum” plus 1-bit “carry out”
  - Can easily build any size adder

carries:		c3	c2	c1	cin
B:		b3	b2	b1	b0
A:	+	a3	a2	a1	a0
	cout	s3	s2	s1	s0

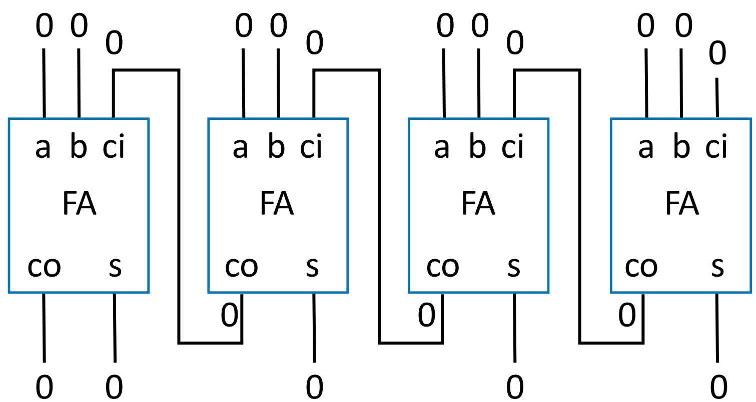


(a)

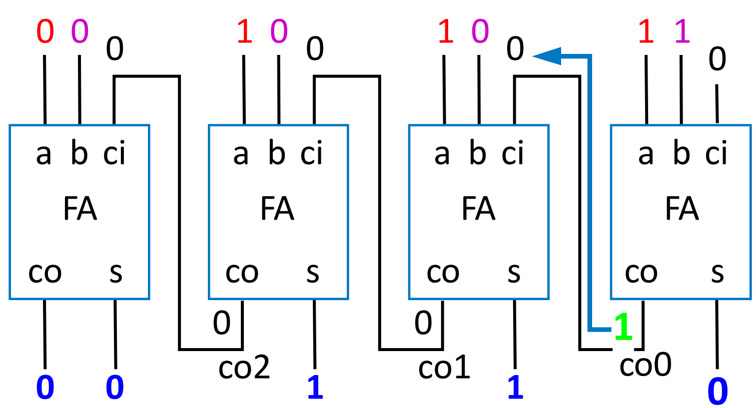


(b)

Carry-Ripple Adder's Behavior



Assume all inputs initially 0



0111+0001

(answer should be 01000)

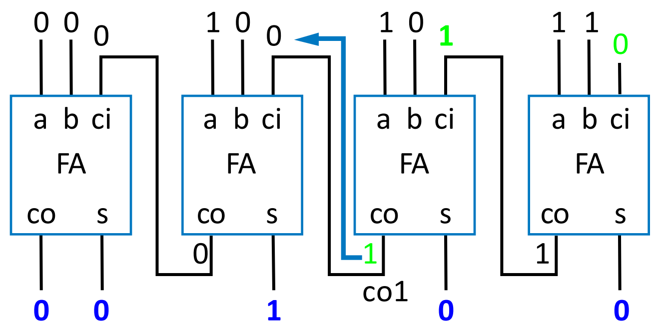
Output after 2 ns (1 FA delay)

Wrong answer -- something wrong? No -- just need more time for carry to ripple through the chain of full adders.

Carry-Ripple Adder's Behavior

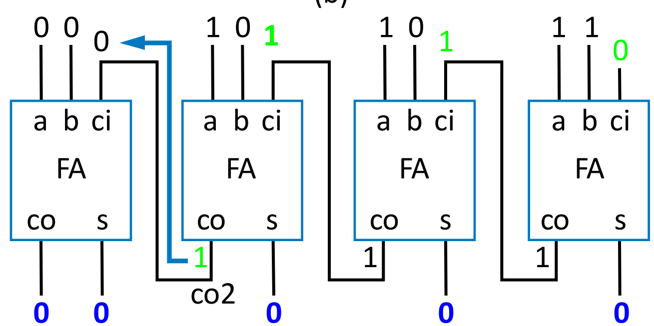
0111+0001

(answer should be 01000)



(b)

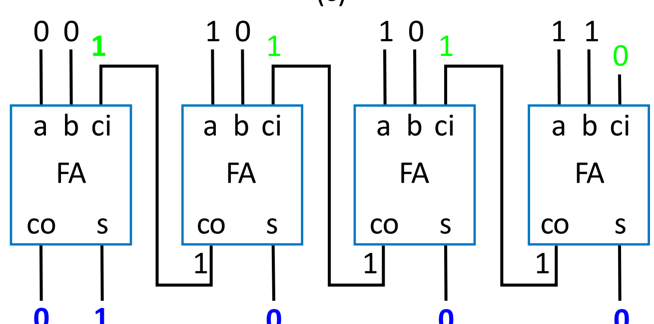
Outputs after 4ns (2 FA delays)



(c)

a

Outputs after 6ns (3 FA delays)



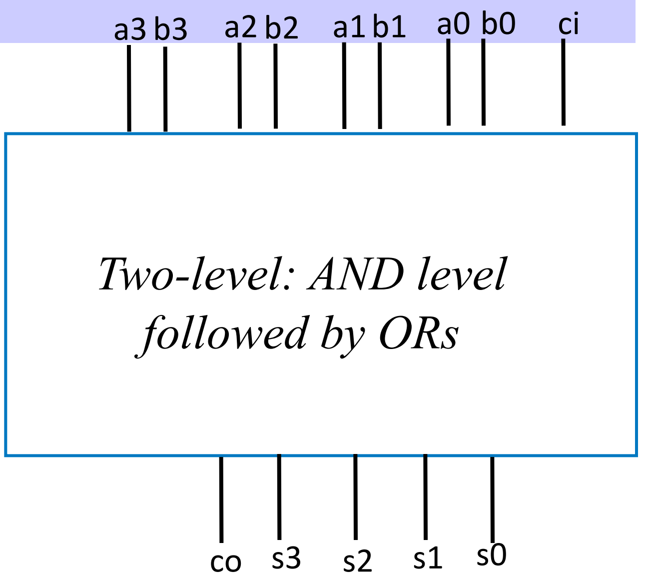
(d)

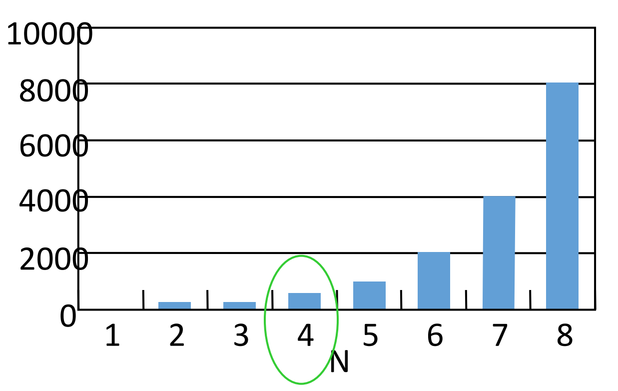
Output after 8ns (4 FA delays)

Correct answer appears after 4 FA delays

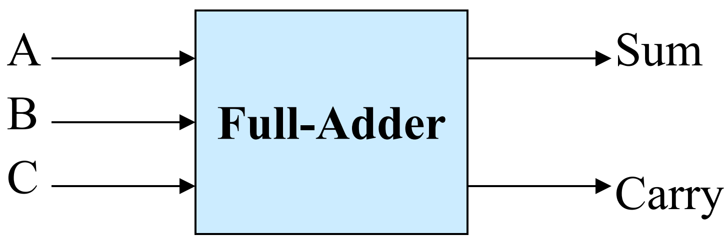
Faster Adder

- Faster adder – Use two-level combinational logic design process
  - 4-bit two-level adder is big
    - 9 input and 5 output combination circuit
  - Pro: Fast
    - 2 gate-level delays
  - Con: Large
    - Truth table would have $2^{(4+4+1)}$ =512 rows
    - Plot shows 4-bit adder would use about 500 gates





Recall Full Adder



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

$$\begin{aligned}
\text{Sum} &= A'B'C + A'BC' + AB'C' + ABC \\
&= \Sigma\, m(1, 2, 4, 7) \\
\text{Carry} &= A'BC + AB'C + ABC' + ABC \\
&= \Sigma\, m(3, 5, 6, 7)
\end{aligned}$$

$$\begin{aligned}
\text{Sum} &= A'B'C + A'BC' + AB'C' + ABC \\
&= A'(B'C + BC') + A(B'C' + BC) \\
&= A'(B \oplus C) + A(B \oplus C)' \\
&\quad (\text{let } B \oplus C = D) \\
&= A'D + AD' \\
&= A \oplus D \\
&= A \oplus B \oplus C
\end{aligned}$$

$$\begin{aligned}
\text{Carry} &= A'BC + AB'C + ABC' + ABC \\
&= A'BC + AB'C + AB\,(C' + C) \\
&= A'BC + AB'C + AB \\
&= (A'C + A)B + A(B'C + B) \\
&= (C + A)B + A(C + B) \\
&= CB + AB + AC + AB \\
&= AB + AC + BC \\
&= (A'B+AB')\,C + AB\,(C'+C) \\
&= (A \oplus B)C + AB
\end{aligned}$$

Faster Adder – Intuitive Attempt at "Lookahead"

- Produce carries directly

  c1 = a0b0 + a0c0 + b0c0

  c2 = a1b1 + a1c1 + b1c1

  = a1b1 + a1(a0b0+a0c0+b0c0) + b1(a0b0+a0c0+b0c0)

  c3 = a2b2 + a2c2 + b2c2

  = …… (replace c2)

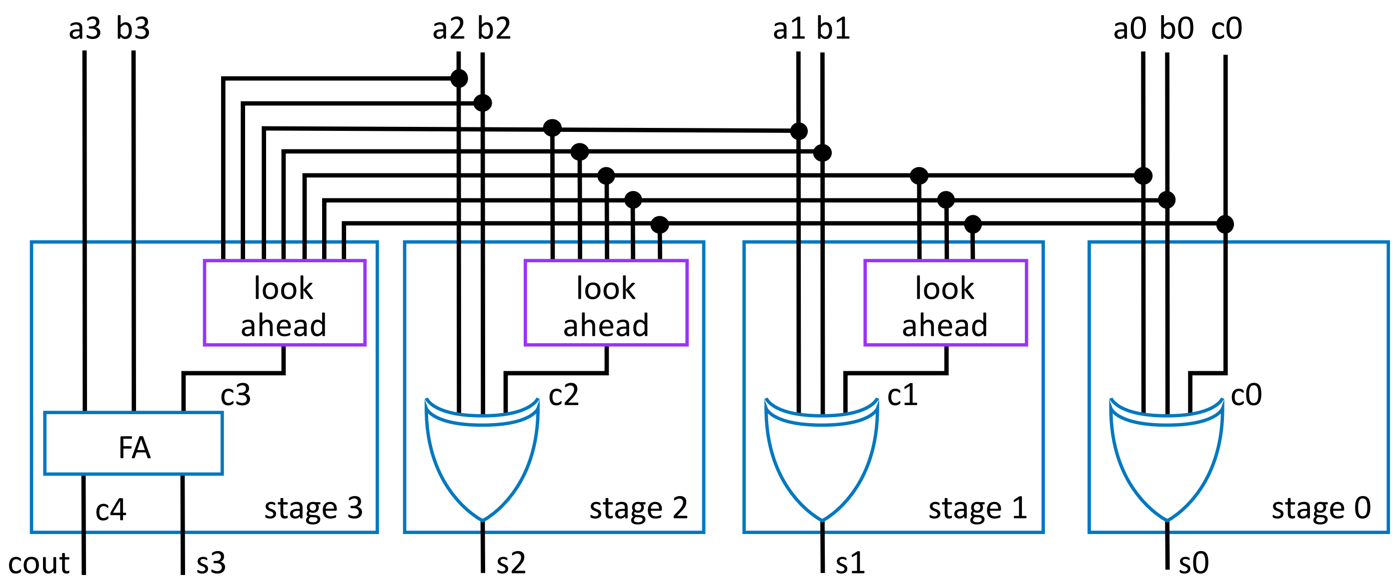
  c4 = a3b3 + a3c3 + b3c3

  = …… (replace c3)

- Carry outputs of all FAs are represented with a0, a1, a2, a3, b0, b1, b2, b3, and c0

Faster Adder – Intuitive Attempt at "Lookahead"

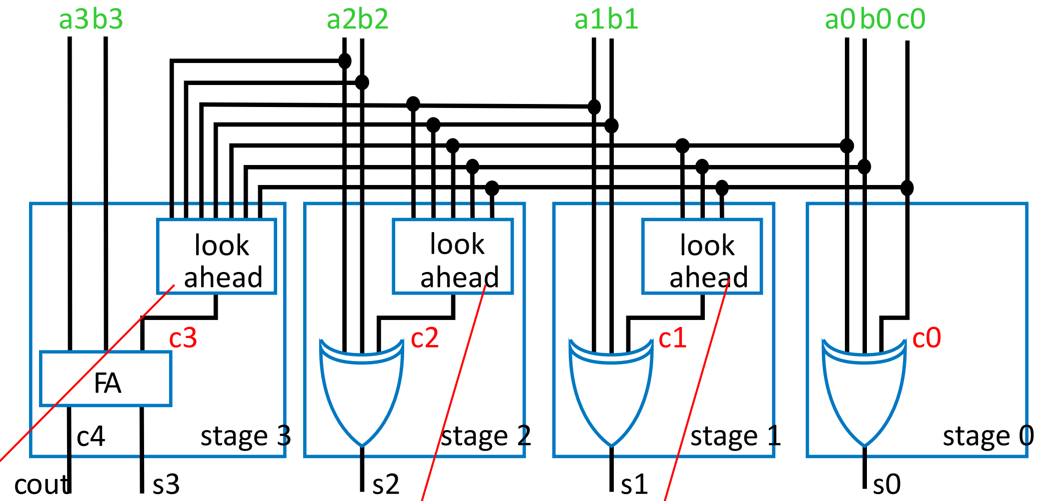
- Idea: Modify carry-ripple adder
  - don't wait for carry to ripple, but rather *directly compute* from inputs of earlier stages
  - Called "lookahead" because current stage "looks ahead" at previous stages



Notice – no rippling of carry

Faster Adder – Intuitive Attempt at “Lookahead”

- Carry lookahead logic
  - No waiting for ripple
  - 2-layer SOP logic
- Problem
  - Equations get too big
  - Not efficient
  - Need a better form of lookahead



c1 = b0c0 + a0c0 + a0b0

c2 = b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 + a1a0b0 + a1b1

c3 = b2b1b0c0 + b2b1a0c0 + b2b1a0b0 + b2a1b0c0 + b2a1a0c0 + b2a1a0b0 + b2a1b1 + a2b1b0c0 + a2b1a0c0 + a2b1a0b0 + a2a1b0c0 + a2a1a0c0 + a2a1a0b0 + a2a1b1 + a2b2

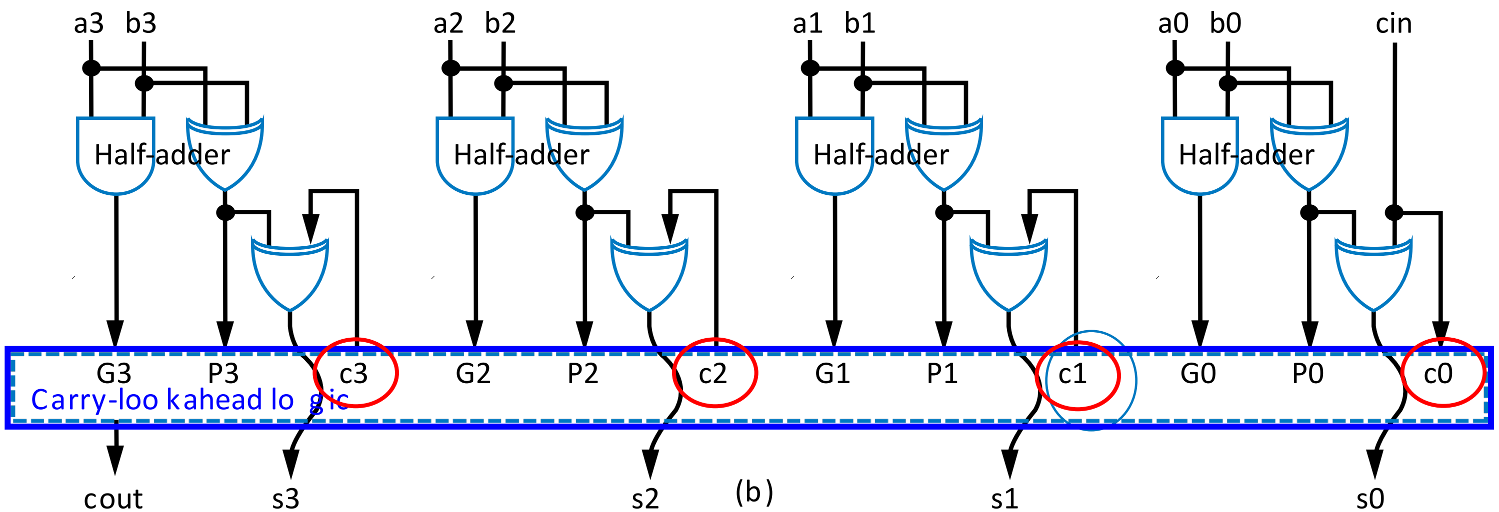
Huge number of gates

Better Form of Lookahead

- Recall Full Adder, another equation for carry

  Carry = ab + (a $\oplus$ b)c
- Define two terms
  - ***Propagate***: P = a $\oplus$ b
  - ***Generate***: G = ab
- Compute lookahead carries from *P* and *G* terms, *not from external inputs*
  - Cout = G + Pc
    - c1 = a0b0 + (a0$\oplus$b0)c0 = G0 + P0c0
    - c2 = a1b1 + (a1$\oplus$b1)c1 = G1 + P1c1
    - c3 = a2b2 + (a2$\oplus$b2)c2 = G2 + P2c2

Better Form of Lookahead



- With *P* & *G*, the carry lookahead equations are much simpler
  - Equations before plugging in
    - c1 = G0 + P0c0
    - c2 = G1 + P1c1
    - c3 = G2 + P2c2
    - cout = G3 + P3c3

After plugging in:

c1 = G0 + P0c0

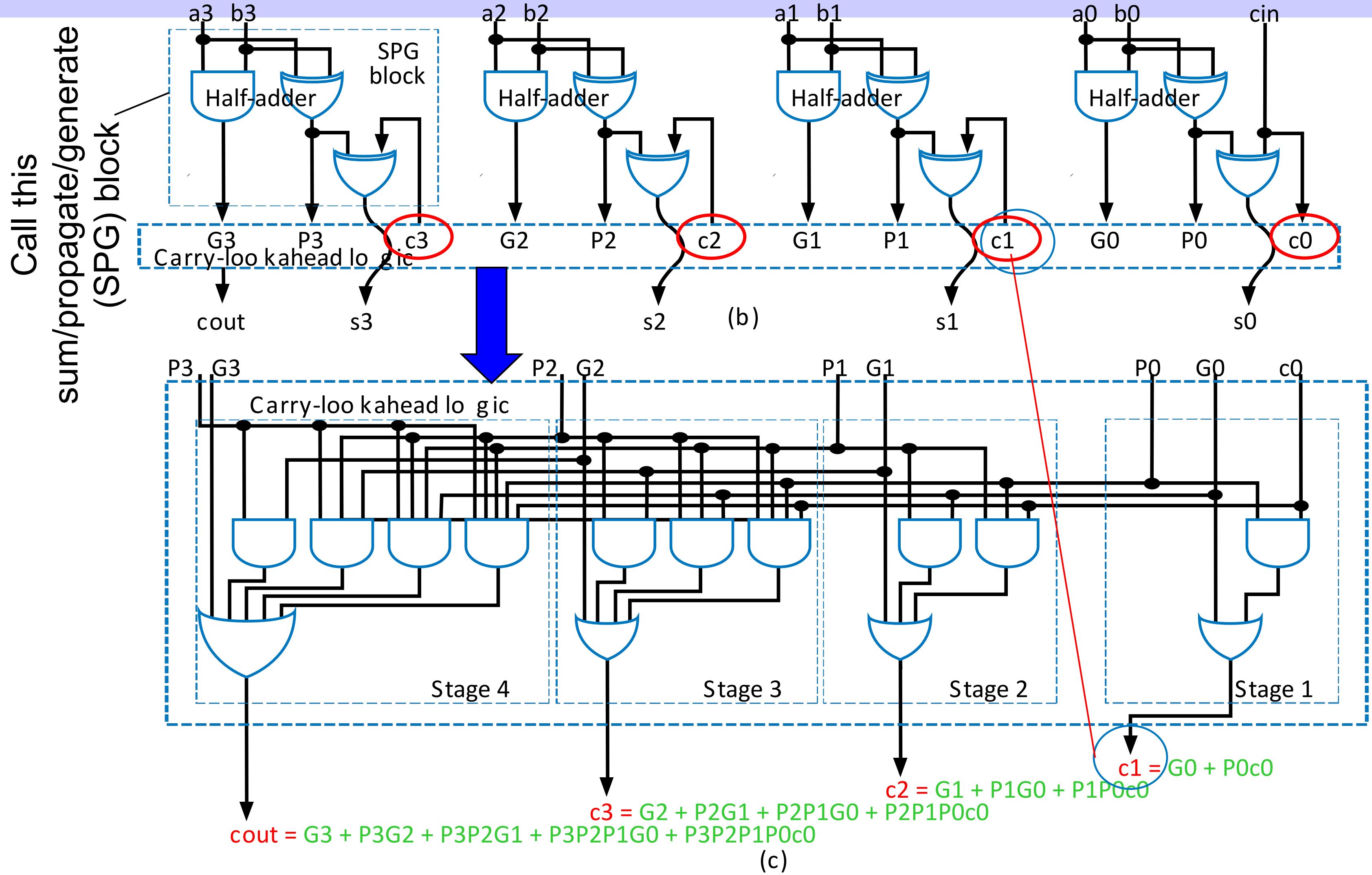
c2 = G1 + P1c1 = G1 + P1(G0 + P0c0)
c2 = G1 + P1G0 + P1P0c0

c3 = G2 + P2c2 = G2 + P2(G1 + P1G0 + P1P0c0)
c3 = G2 + P2G1 + P2P1G0 + P2P1P0c0

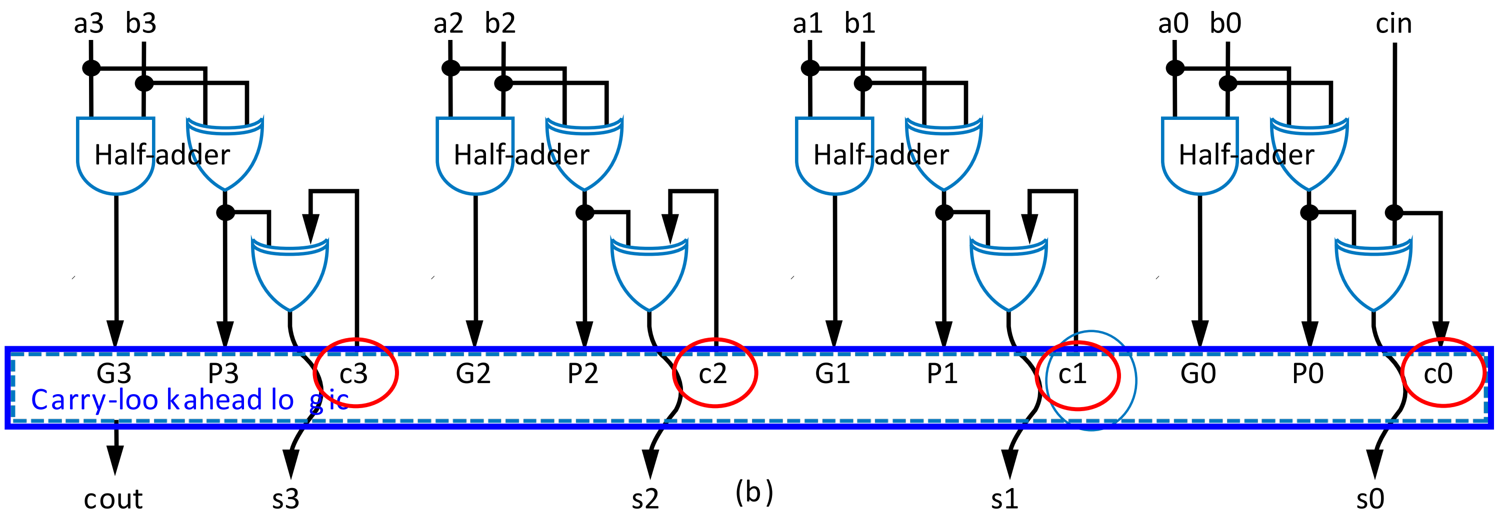
cout = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0c0

Much simpler than the intuitive lookahead

Better Form of Lookahead (cont.)

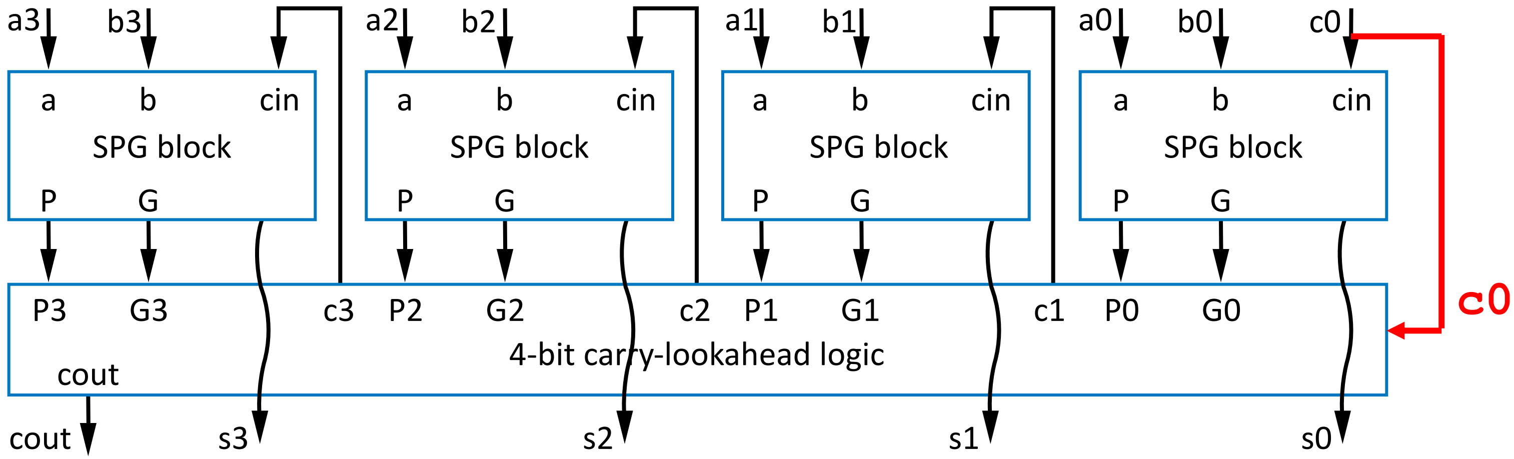


Better Form of Lookahead (cont.)



- With *P* & *G*, sum outputs are
  - s0 = P0 ⊕ cin
  - s1 = P1 ⊕ c1
  - s2 = P2 ⊕ c2
  - s3 = P3 ⊕ c3

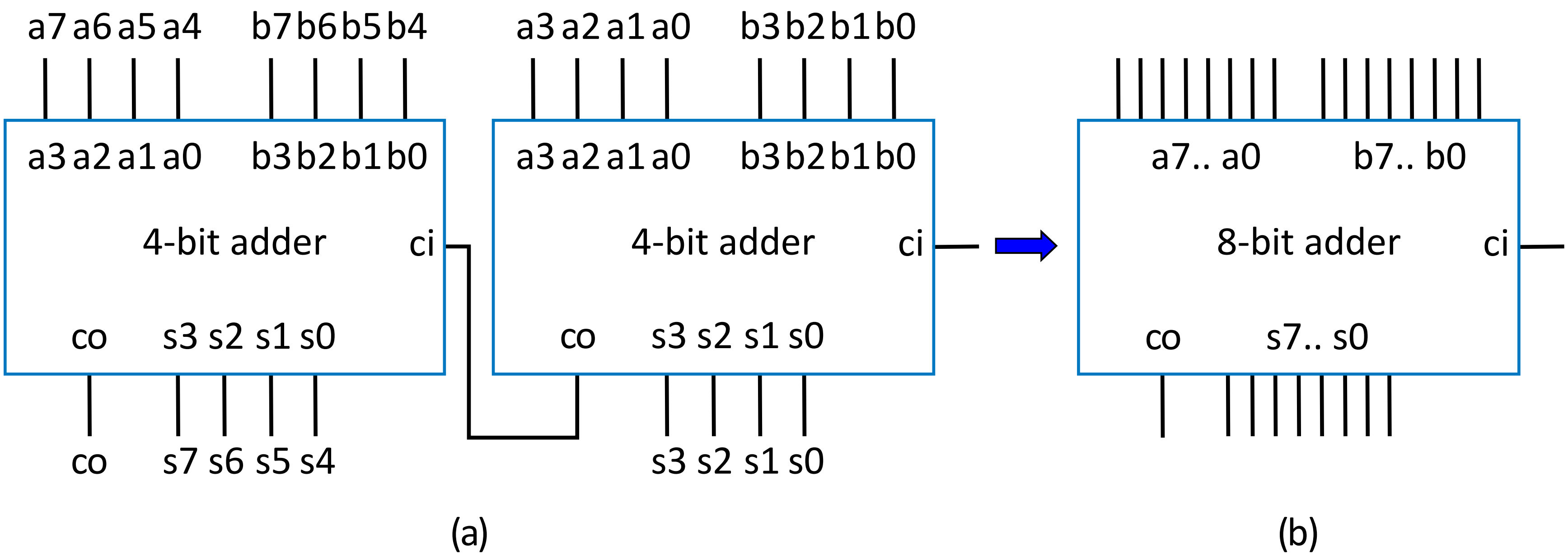
Carry-Lookahead Adder -- High-Level View



- Fast -- only 4 gate level delays
  - Each stage has SPG block with 2 gate levels
  - Carry-lookahead logic quickly computes the carry from the propagate and generate bits using 2 gate levels inside
- Reasonable number of gates
- Nice balance between intuitive lookahead and pure combinational logic

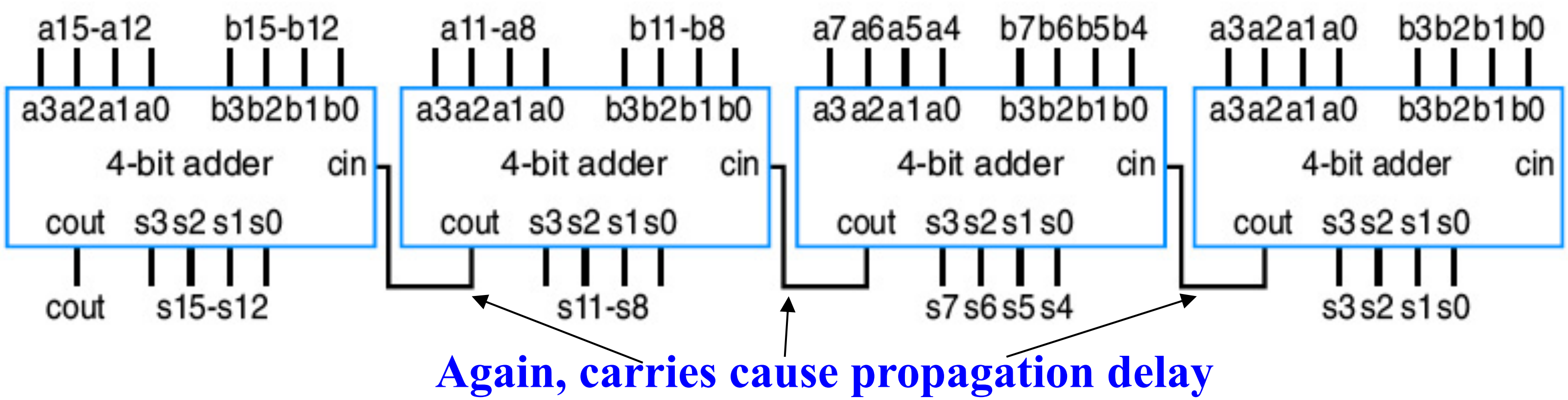
Cascading Adders

- Example: construct an 8-bit adder with two 4-bit adders



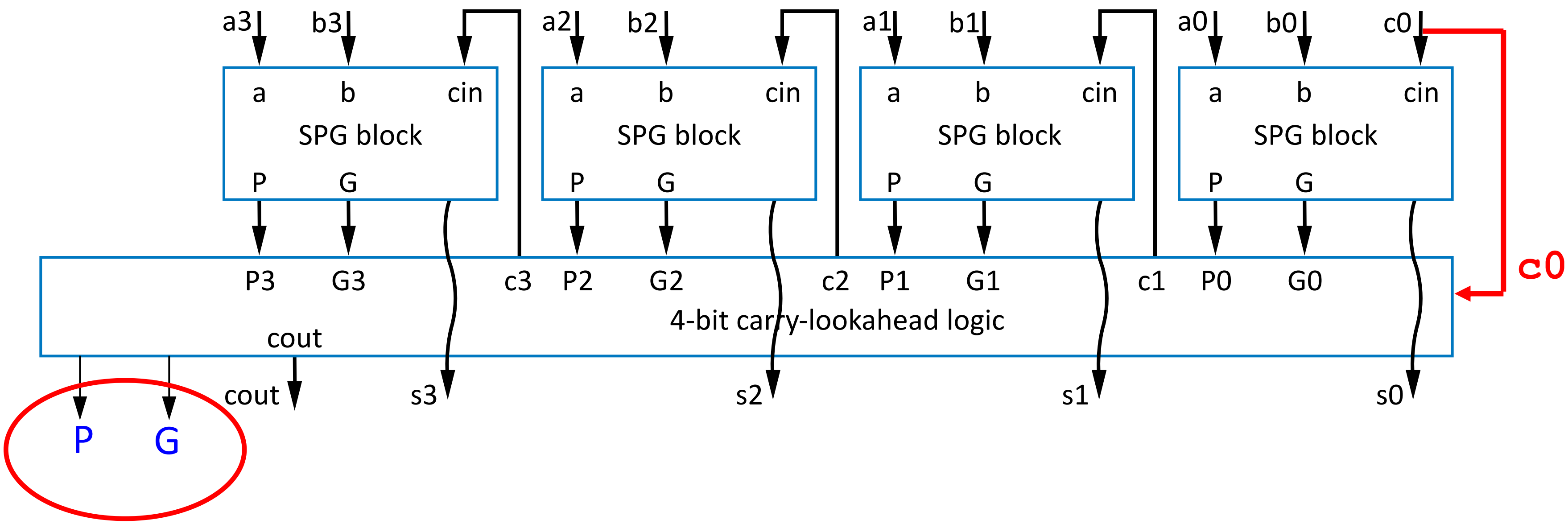
Cascading Adders to CLA Addres

- Example: construct an 16-bit adder with four 4-bit adders



Carry-Lookahead Adder -- High-Level View

- Adding two more outputs: P, G

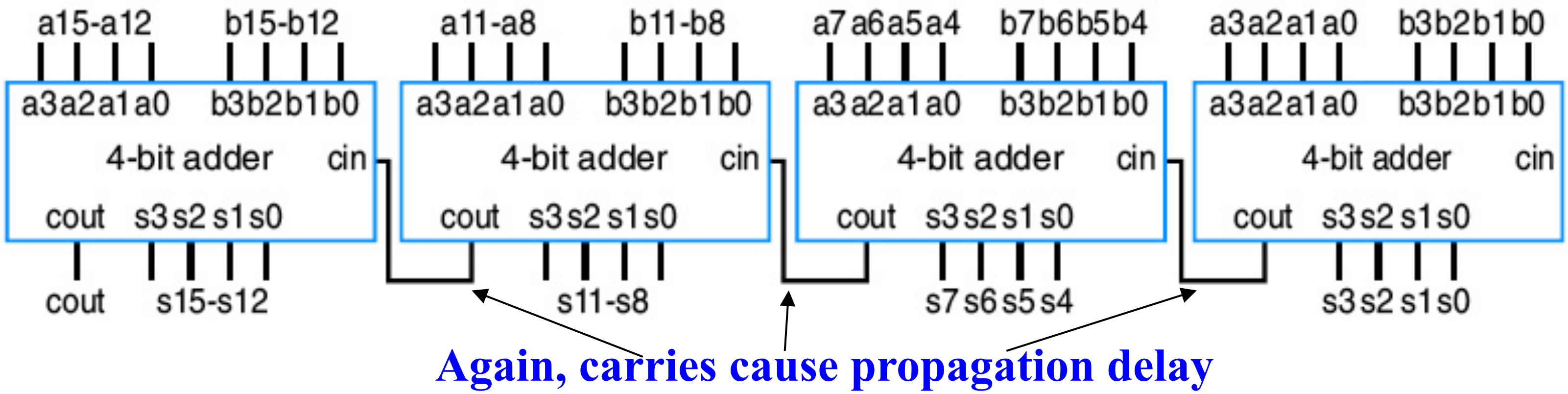


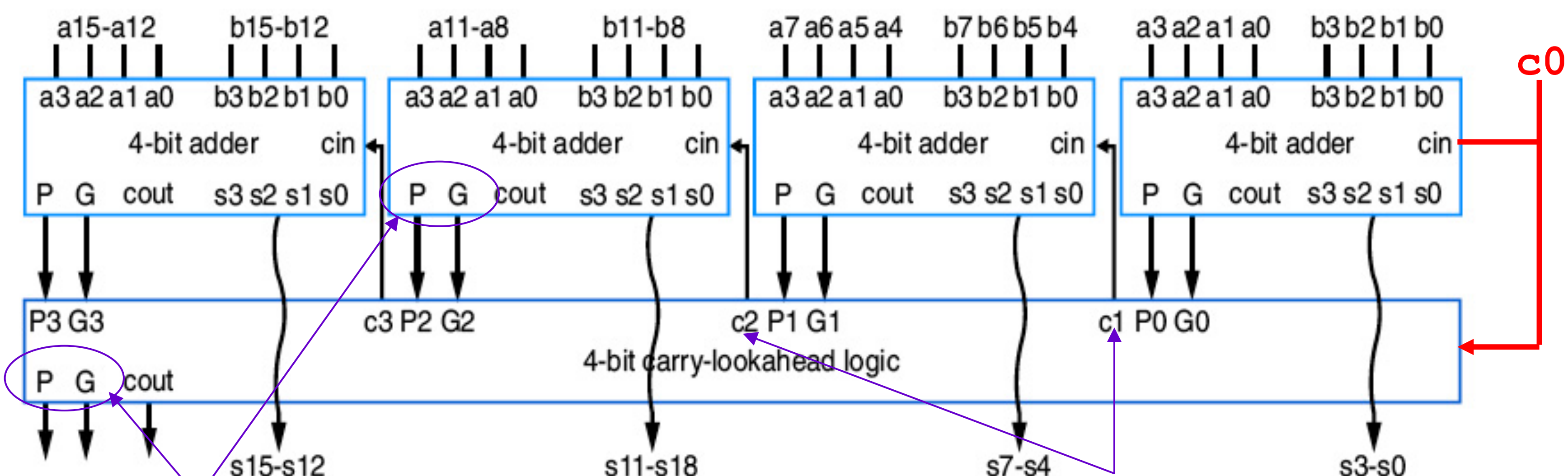
P = P3P2P1P0

G = G3+P3G2+P3P2G1+P3P2P1G0

Cascading Adders to CLA Addres

- Example: construct an 16-bit adder with four 4-bit adders



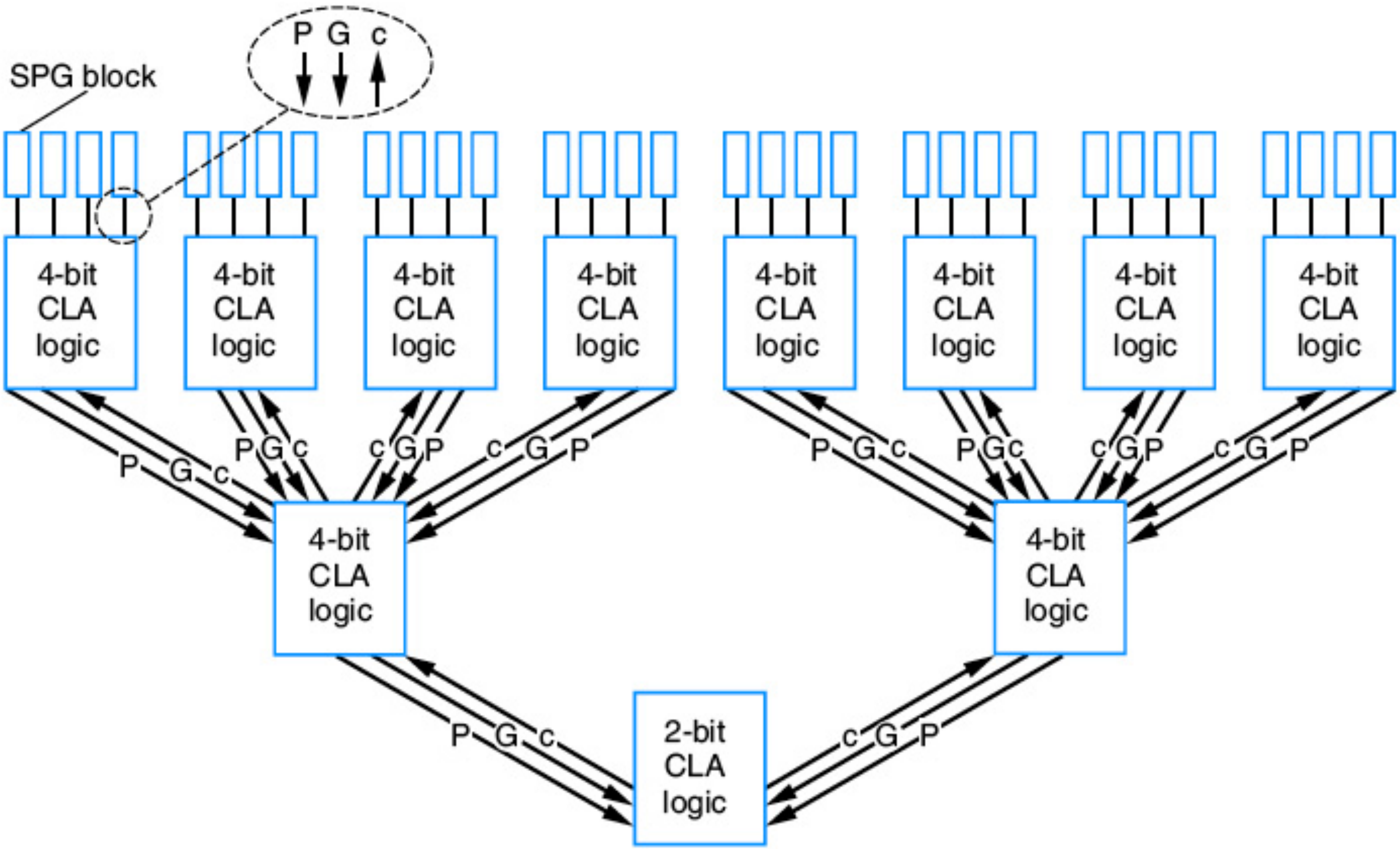


$P = P3P2P1P0$
$G = G3+P3G2+P3P2G1+P3P2P1G0$

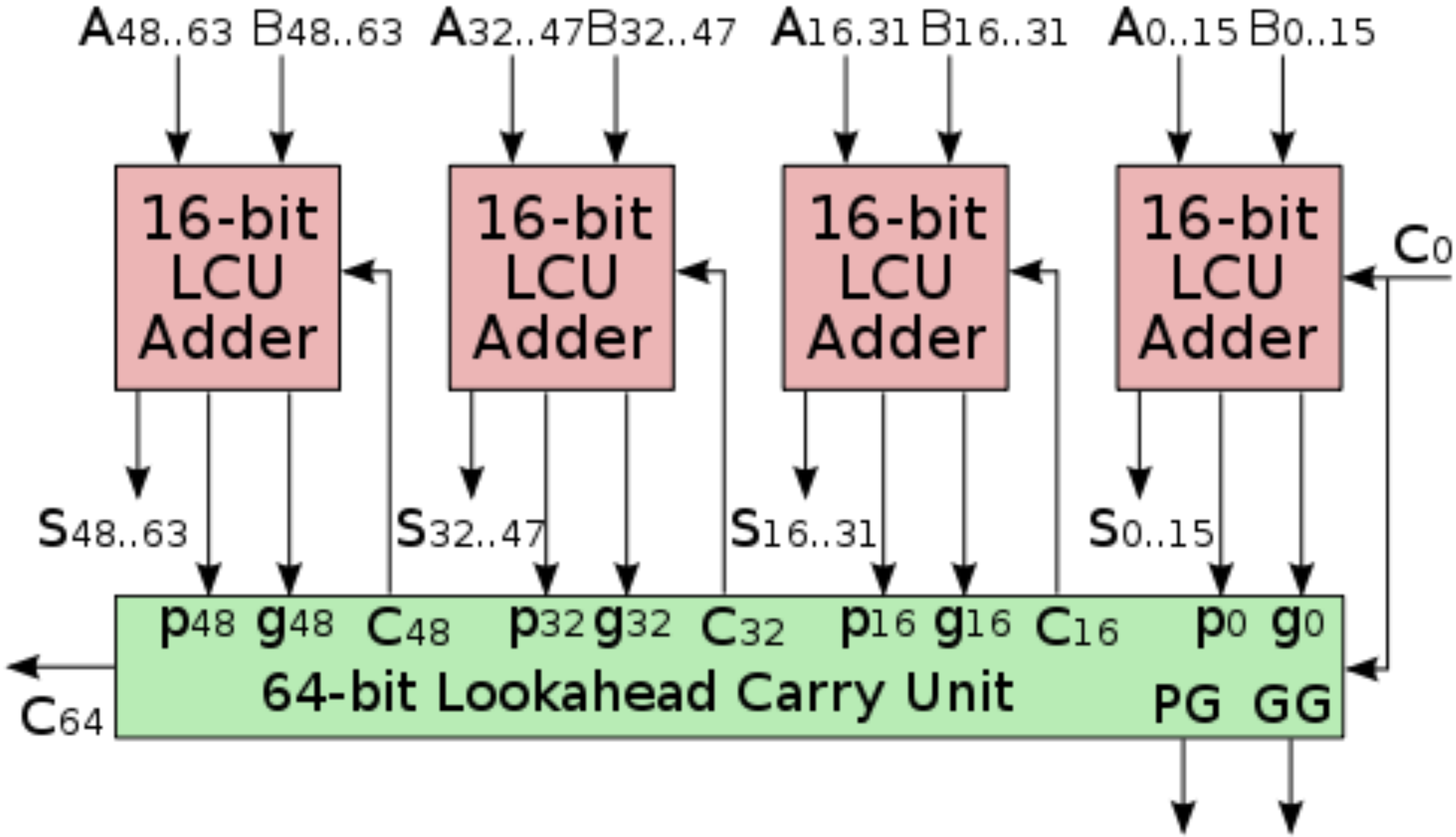
$C1 = G0 + P0C0$
$C2 = G1 + P1C1 = G1+P1G0+P1P0C0$
……

CLA Adders

- Multi-level CLA structure



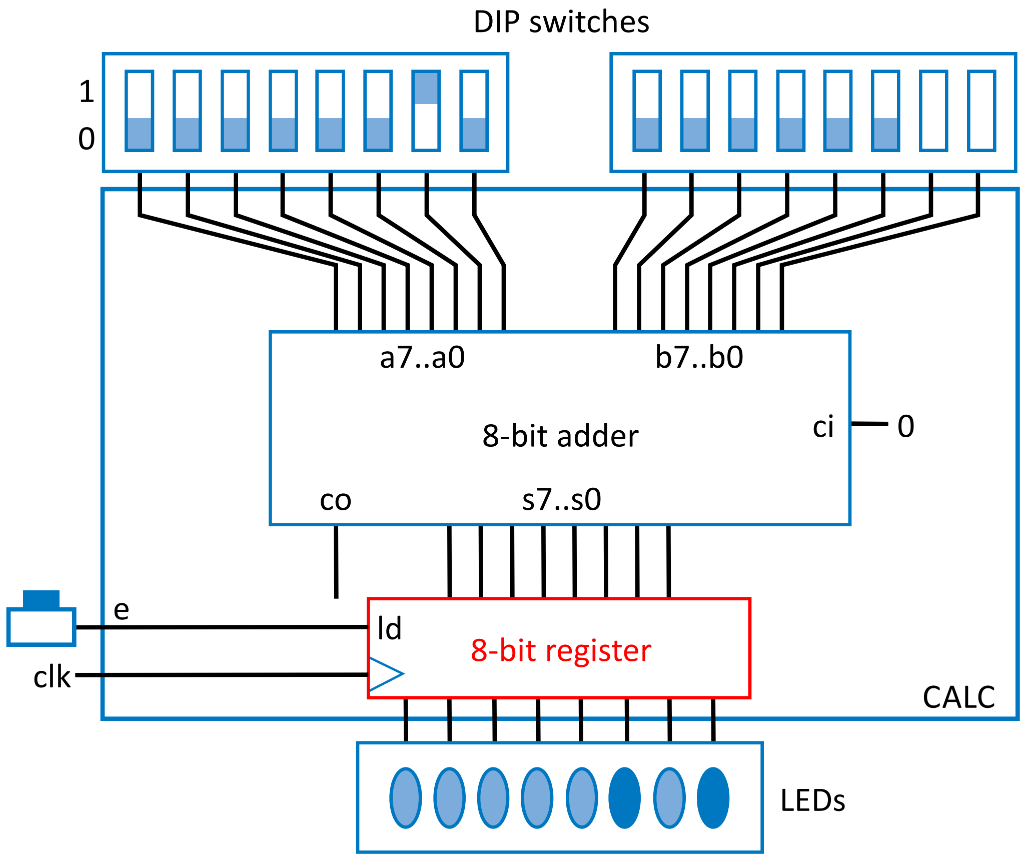
CLA Adders



Source: www.wikipedia.org

Adder Example: DIP-Switch-Based Adding Calculator

- To prevent spurious values from appearing at output, can place register at output



Incrementer

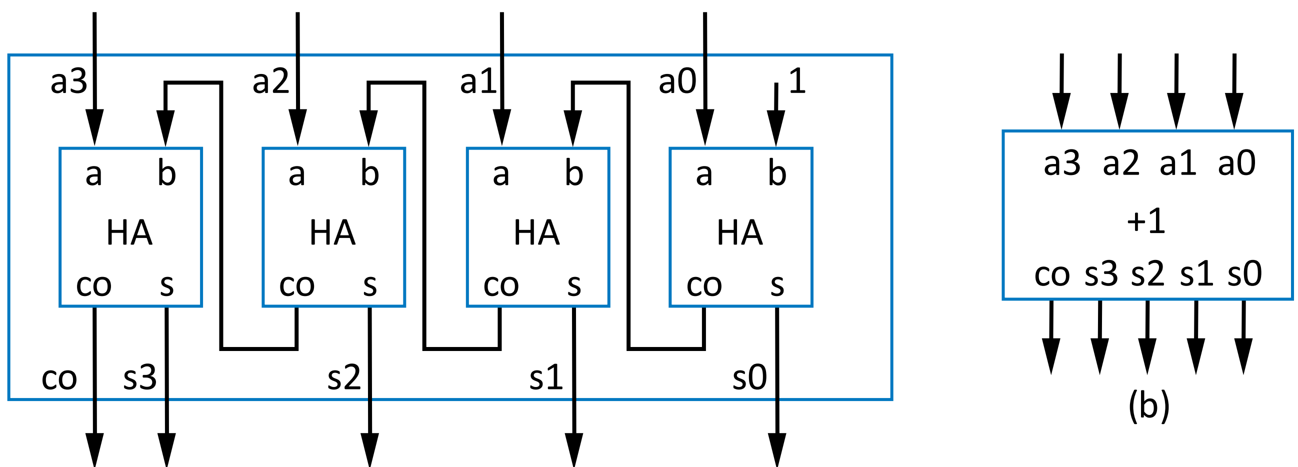
- Traditional design procedure
  - Capture truth table
  - Derive equation for each output
    - c0 = a3a2a1a0
    - ...
    - s0 = a0’
  - Results in small and fast circuit
  - Works for small operand
    - larger operand leads to exponential growth, like for N-bit adder

Inputs				Outputs				
a3	a2	a1	a0	c0	s3	s2	s1	s0
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0	0
0	1	0	0	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	1	0	0	0	1	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	1	0
1	0	1	0	0	1	0	1	1
1	0	1	1	0	1	1	0	0
1	1	0	0	0	1	1	0	1
1	1	0	1	0	1	1	1	0
1	1	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0

Incrementer

- Alternative incrementer design
  - Could use N-bit adder with one of the inputs set to 1
  - Use half-adders (adds two bits) rather than full-adders (adds three bits)
  - Slower but simpler

```
carries:    0 1 1
            0 0 1 1
unused
  +               1
  ---------------
          0 0 1 0 0
```

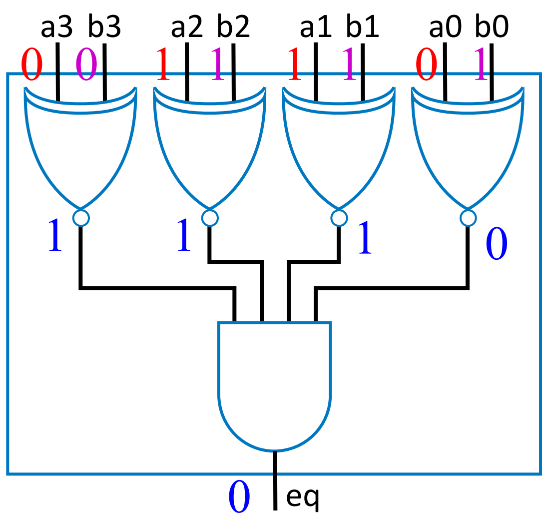


(a)　(b)

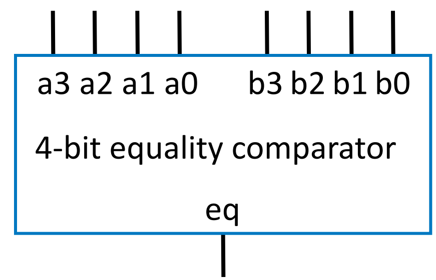
Comparators

- ***N-bit equality comparator***: Outputs 1 if two N-bit numbers are equal
- Example: 4-bit equality comparator with inputs A and B
  - Approach 1: combinational design procedure
  - Approach 2: recall functionality of XOR and XNOR

0110 = 0111 ?



(a)



(b)

Magnitude Comparator

- ***N-bit magnitude comparator***:
  - Indicates whether A>B, A=B, or A<B, for its two N-bit inputs A and B
  - Design approach 1: combinational design procedure
  - Design approach 2: Consider how compare by hand.

A=1011 B=1001

1011 **1**001 Equal

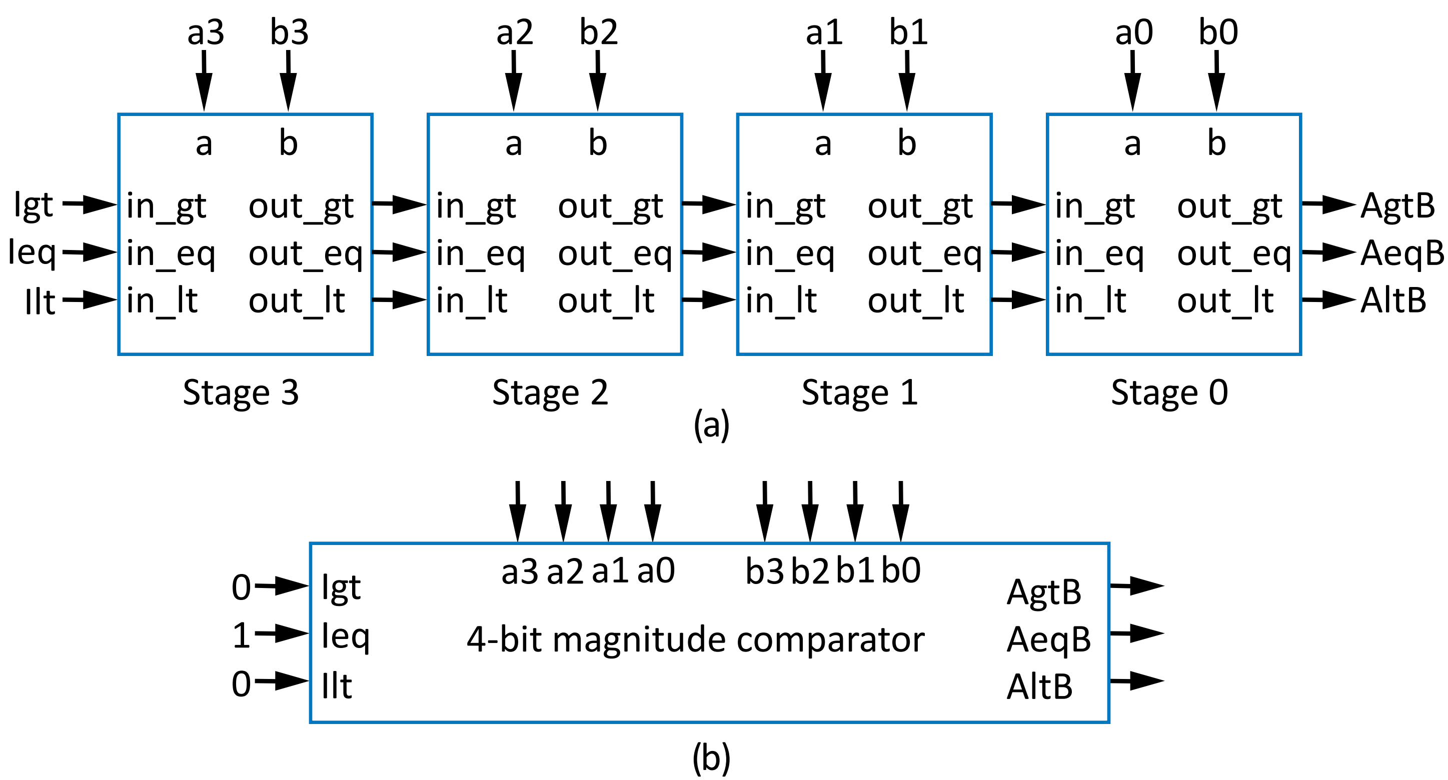
1**0**11 1**0**01 Equal

10**1**1 10**0**1 **Unequal**

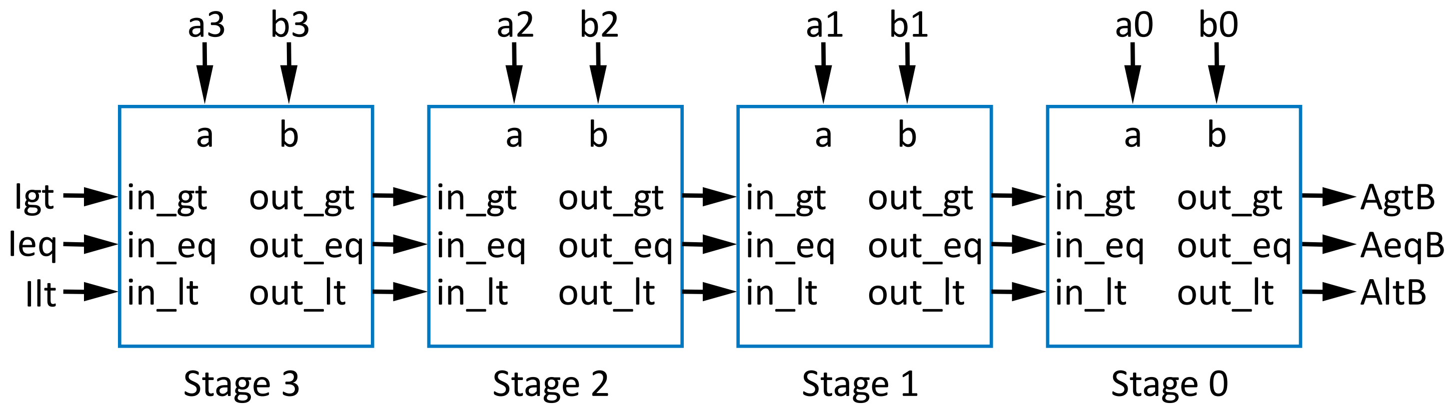
So A > B

Magnitude Comparator

- By-hand example leads to idea for design
  - Start at left, compare each bit pair, pass results to the right
  - Each stage has 3 inputs indicating results of higher stage, passes results to lower stage



Magnitude Comparator



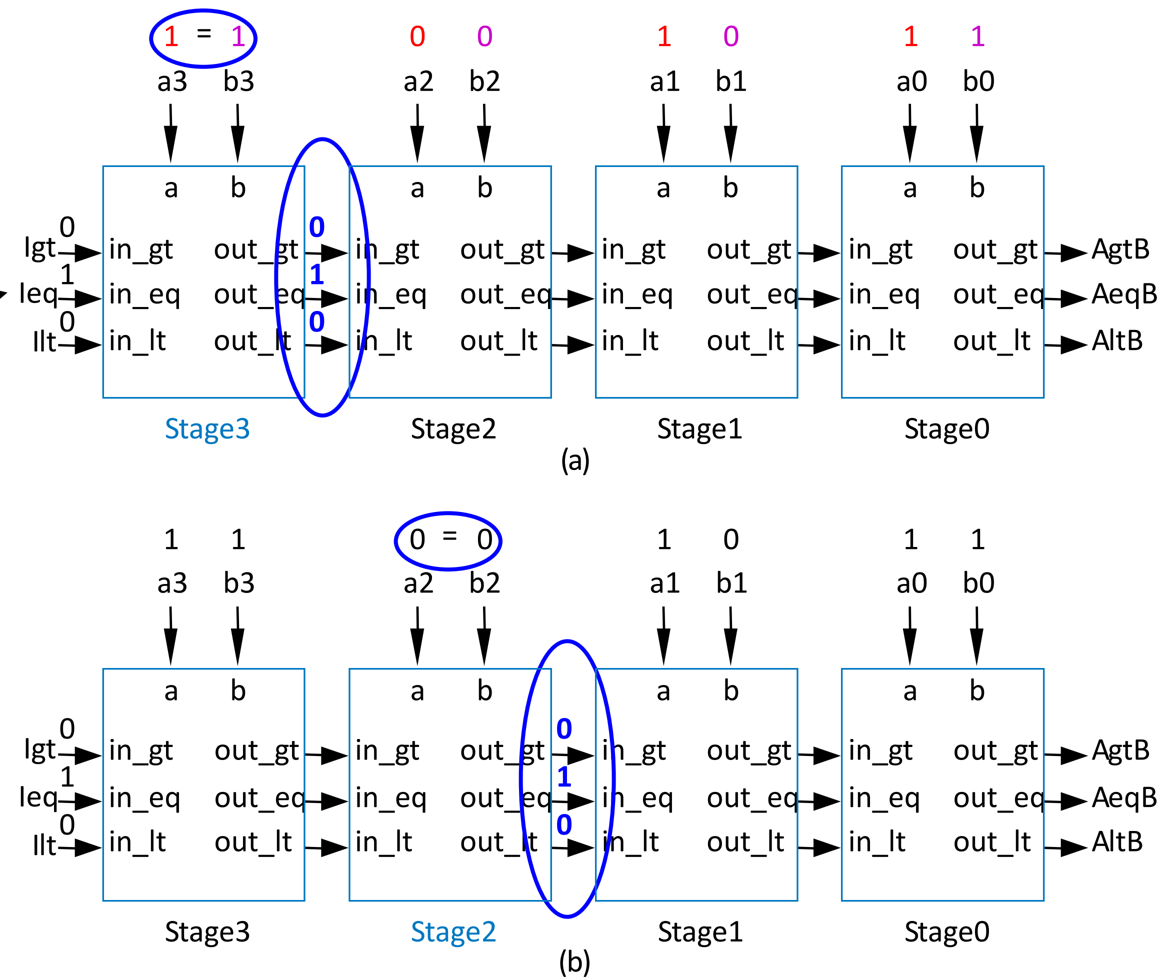
- Each stage:
  - out_gt = in_gt + (in_eq * a * b’)
    - A>B (so far) if already determined in higher stage, or if higher stages equal but in this stage a=1 and b=0
  - out_lt = in_lt + (in_eq * a’ * b)
    - A<B (so far) if already determined in higher stage, or if higher stages equal but in this stage a=0 and b=1
  - out_eq = in_eq * (a XNOR b)
    - A=B (so far) if already determined in higher stage and in this stage a=b too
  - Simple circuit inside each stage, just a few gates (not shown)

Magnitude Comparator

- How does it work?

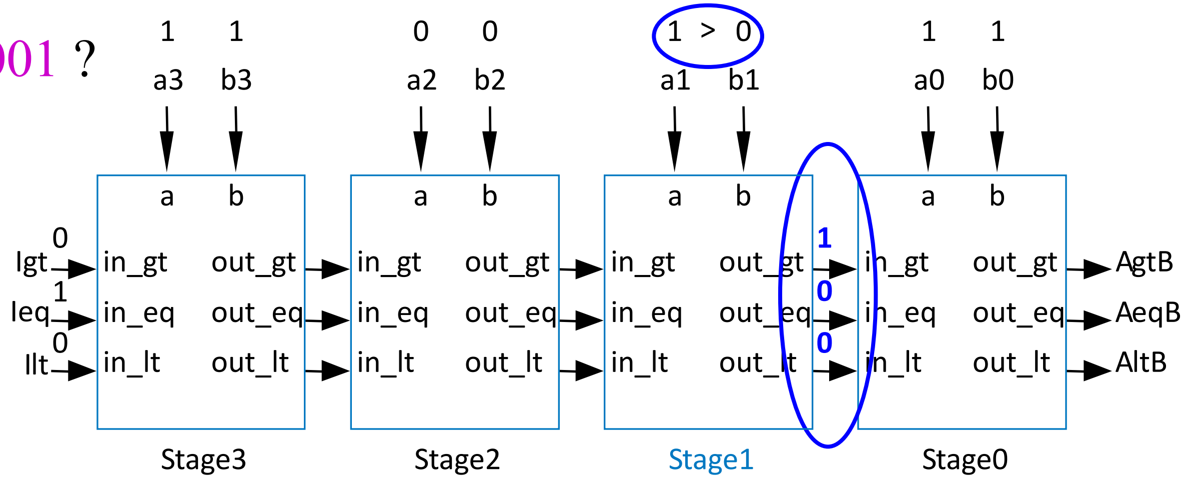
1011 = 1001 ?

Ieq=1 causes this stage to compare



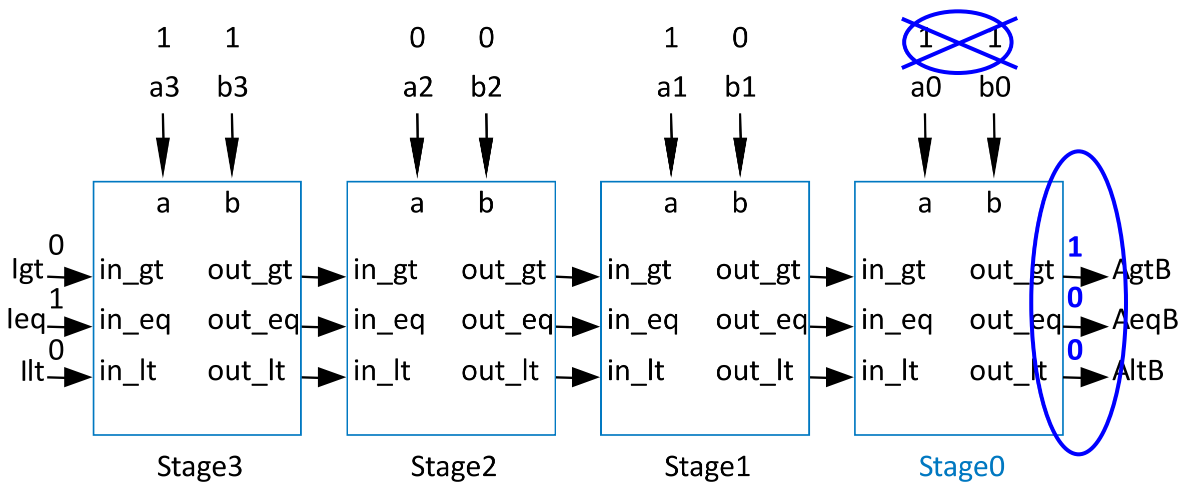
Magnitude Comparator

$1011 = 1001$?



(c)

a



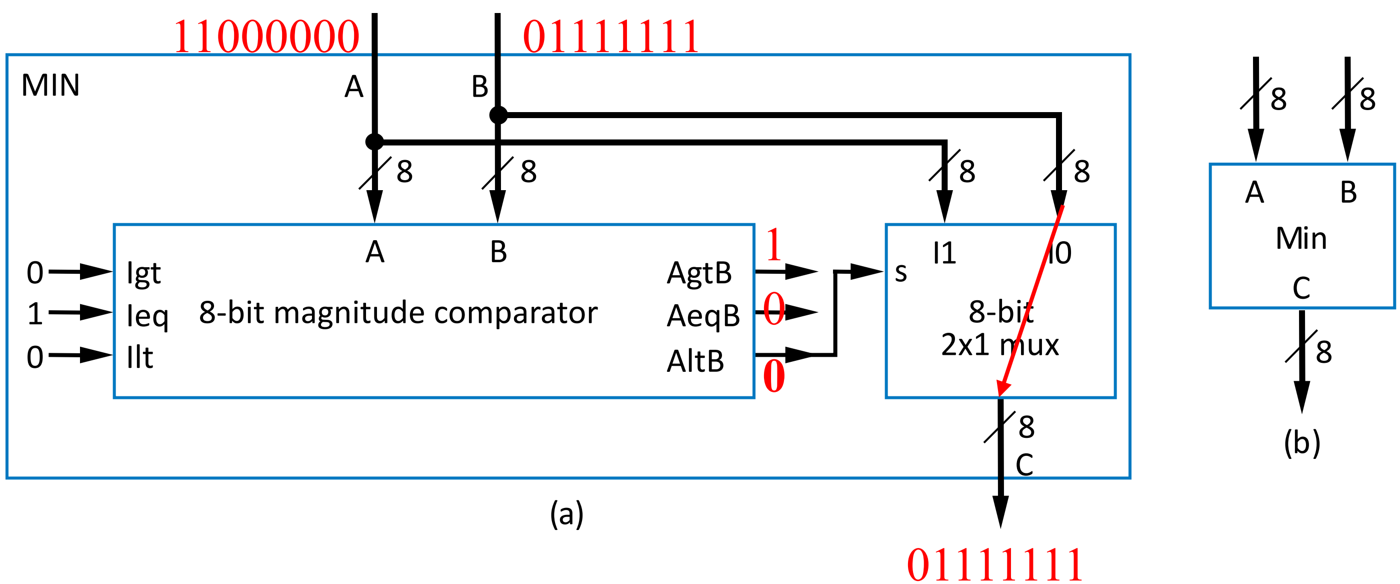
(d)

- Final answer appears on the right
- Takes time for answer to “ripple” from left to right
- Thus called “carry-ripple style” even though there’s no “carry” involved

Magnitude Comparator Example: Minimum of Two Numbers

- Design a combinational component that finds the minimum of two 8-bit numbers
  - Solution: Use 8-bit magnitude comparator and 8-bit 2x1 mux
    - If A<B, pass A through mux. Else, pass B.

a



What if inputs are 2's complement???

Multiplier

- Can build multiplier that mimics multiplication by hand
  - Notice that multiplying multiplicand by 1 is same as ANDing with 1

```
    0110   (the top number is called the multiplicand)
    0011   (the bottom number is called the multiplier)
    ----   (each row below is called a partial product)
    0110   (because the rightmost bit of the multiplier is 1, and 0110*1=0110)
   0110    (because the second bit of the multiplier is 1, and 0110*1=0110)
  0000     (because the third bit of the multiplier is 0, and 0110*0=0000)
+0000      (because the leftmost bit of the multiplier is 0, and 0110*0=0000)
--------
00010010   (the product is the sum of all the partial products: 18, which is 6*3)
```

Multiplier – Array Style

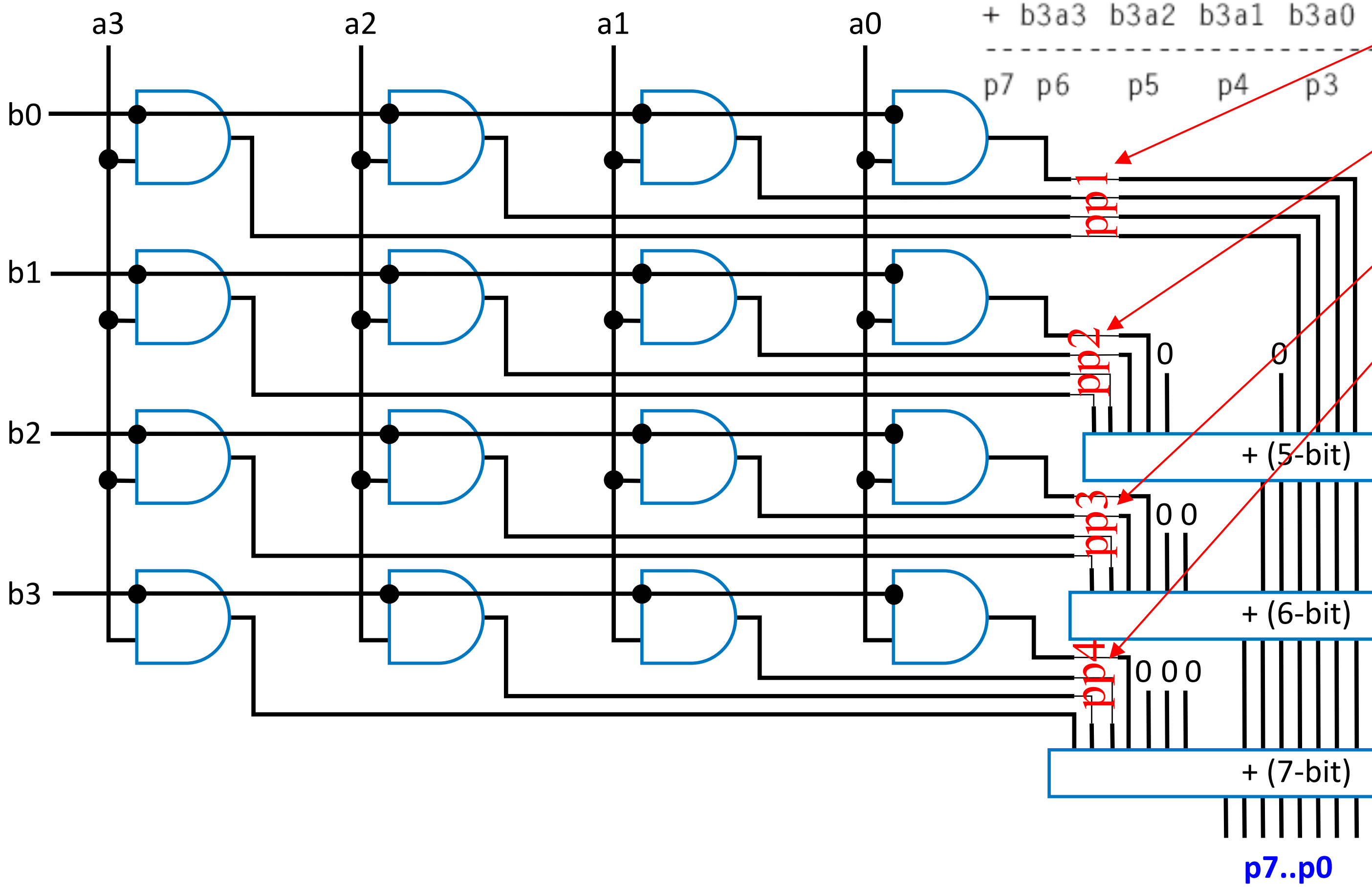
- Generalized representation of multiplication by hand

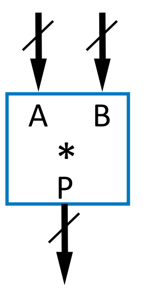
```
                       a3    a2    a1    a0
                     x b3    b2    b1    b0
----------------------------------------------
                       b0a3  b0a2  b0a1  b0a0    (pp1)
                 b1a3  b1a2  b1a1  b1a0    0     (pp2)
           b2a3  b2a2  b2a1  b2a0    0     0     (pp3)
+    b3a3  b3a2  b3a1  b3a0    0     0     0     (pp4)
----------------------------------------------
p7   p6    p5    p4    p3    p2    p1    p0
```

Multiplier – Array Style

- Multiplier design – array of AND gates

```
                     a3    a2    a1    a0
                   x b3    b2    b1    b0
-----------------------------------------
                 b0a3  b0a2  b0a1  b0a0   (pp1)
           b1a3  b1a2  b1a1  b1a0     0   (pp2)
     b2a3  b2a2  b2a1  b2a0     0     0   (pp3)
+ b3a3  b3a2  b3a1  b3a0     0     0     0   (pp4)
-----------------------------------------
p7 p6    p5    p4    p3    p2    p1    p0
```





Block symbol

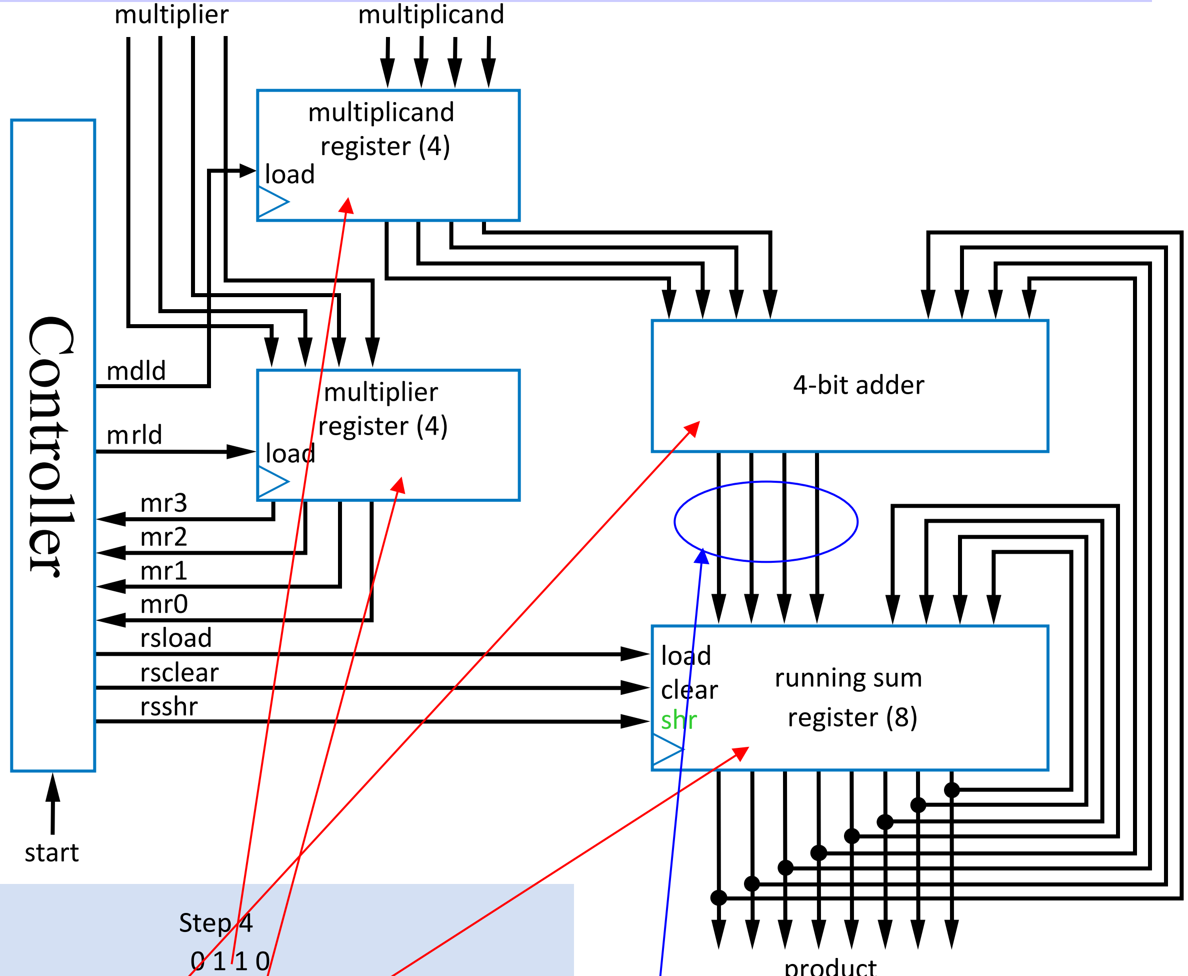
Smaller Multiplier -- Sequential (Add-and-Shift) Style

- Add-and-Shift
  - Don't compute all partial products simultaneously
  - Rather, compute one at a time (similar to by hand), maintain a ***running sum***

	Step 1	Step 2	Step 3	Step 4
	0 1 1 0	0 1 1 0	0 1 1 0	0 1 1 0
	* 0 0 1 1	* 0 0 1 1	* 0 0 1 1	* 0 0 1 1
(running sum)	0 0 0 0	0 0 1 1 0	0 1 0 0 1 0	0 0 1 0 0 1 0
(partial product)	+ 0 1 1 0	+ 0 1 1 0	+ 0 0 0 0	+ 0 0 0 0
(new running sum)	0 0 1 1 0	0 1 0 0 1 0	0 0 1 0 0 1 0	0 0 0 1 0 0 1 0

Smaller Multiplier -- Sequential (Add-and-Shift) Style

- Design circuit that computes one partial product at a time, adds to running sum
  – Note that shifting running sum right (relative to partial product) after each step ensures partial product added to correct running sum bits



Step 1	Step 2	Step 3	Step 4	
0110	0110	0110	0110	
+ 0011	+ 0011	+ 0011	+ 0011	
0000	00110	010010	0010010	(running sum)
+ 0110	+ 0110	+ 0000	+ 0000	(partial product)
00110	010010	0010010	00010010	(new running sum)

What about carry?

What if 2's complement?

Smaller Multiplier – Controller Design