Datapath

Adders

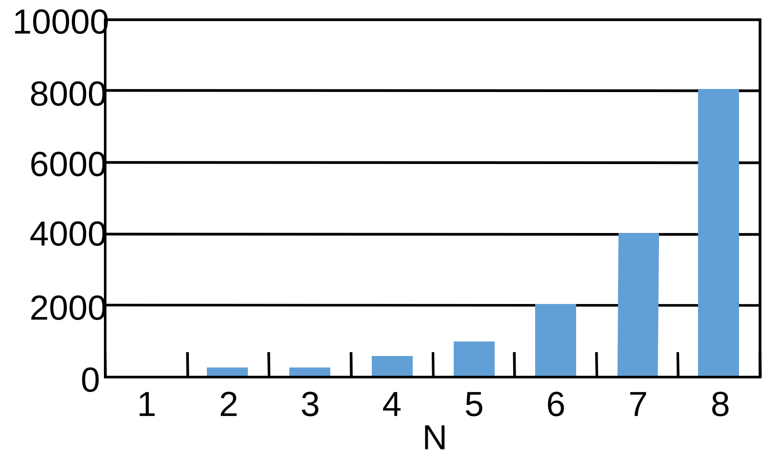
- Adds two N-bit binary numbers
  - 2-bit adder: adds two 2-bit numbers, outputs 3-bit result
  - e.g., 01 + 11 = 100   (1 + 3 = 4)
- Can design using combinational design process , but doesn't work well for reasonable-size N
  - Why not?

Inputs				Outputs		
a1	a0	b1	b0	c	s1	s0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Why Adders Aren't Built Using Standard Combinational Design Process

- Truth table too big
  - 2-bit adder's truth table shown
    - Has $2^{(2+2)}$ = 16 rows
  - 8-bit adder: $2^{(8+8)}$ = 65,536 rows
  - 16-bit adder: $2^{(16+16)}$ = ~4 billion rows
  - 32-bit adder: ...
- Big truth table with numerous 1s/0s yields big logic
  - Plot shows number of transistors for N-bit adders, using state-of-the-art automated combinational design tool

Inputs				Outputs		
a1	a0	b1	b0	c	s1	s0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

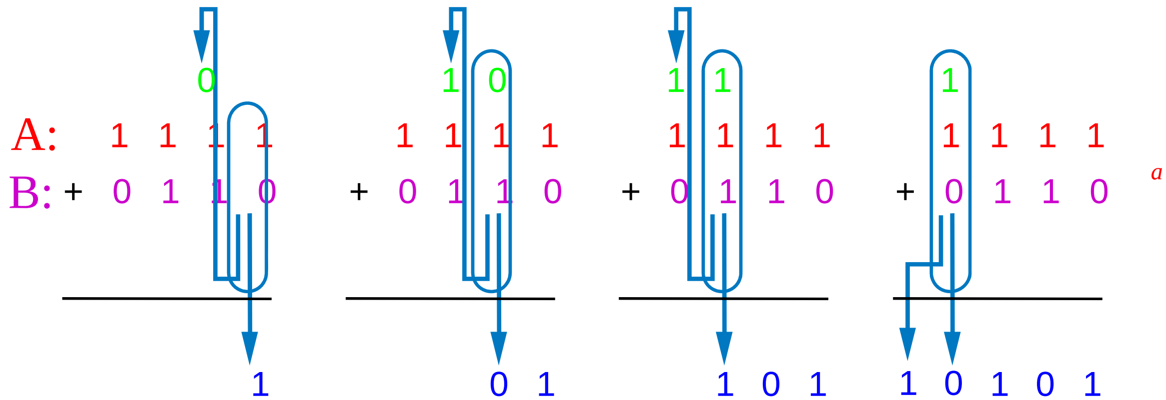


Q: Predict number of transistors for 16-bit adder

A: 1000 transistors for N=5, doubles for each increase of N. So transistors = $1000*2^{(N-5)}$. Thus, for N=16, transistors = $1000*2^{(16-5)}$ = 1000*2048 = 2,048,000. Way too many!

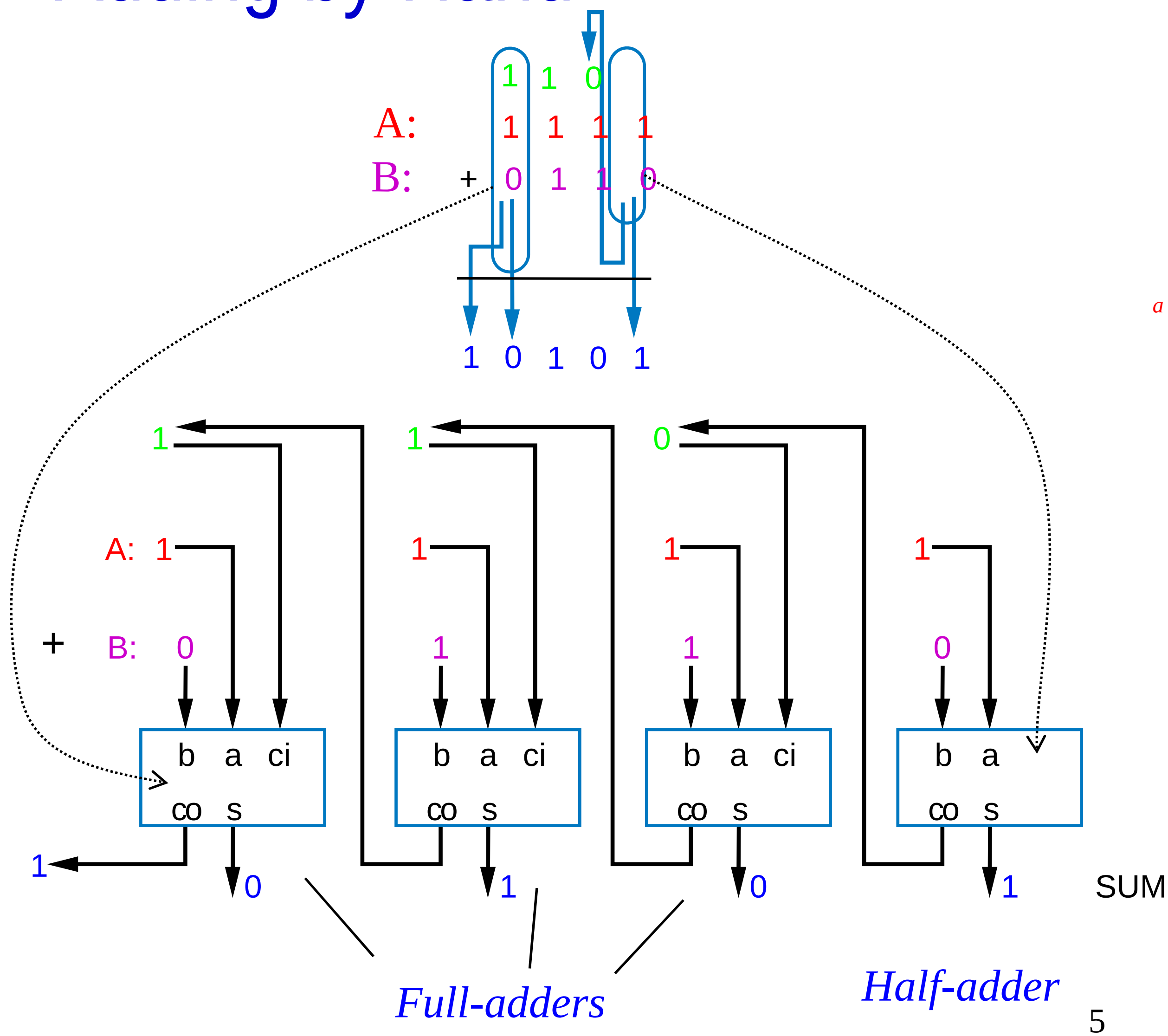
Alternative Method to Design an Adder: Imitate Adding by Hand

- Alternative adder design: mimic how people do addition by hand
- One column at a time
  - Compute sum, add carry to next column



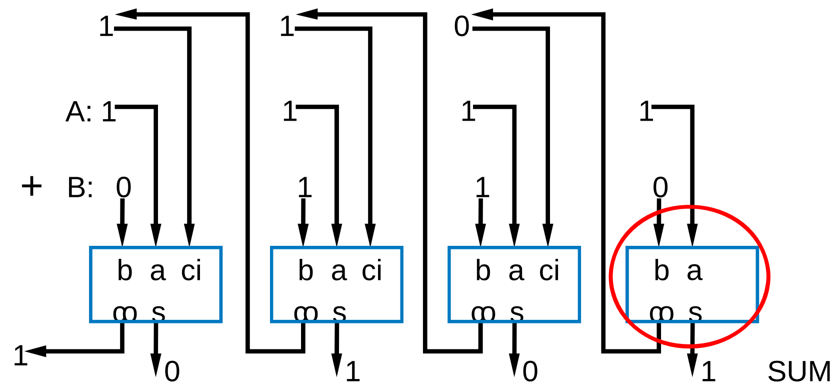
Alternative Method to Design an Adder: Imitate Adding by Hand

- Create component for each column
  - Adds that column's bits, generates sum and carry bits



Half-Adder

- ***Half-adder***: Adds 2 bits, generates sum and carry
- Design using combinational design process from Ch 2



Step 1: Capture the function

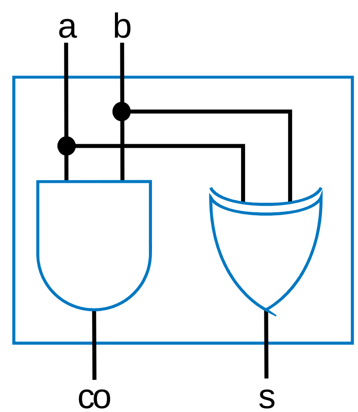
Inputs		Outputs	
a	b	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

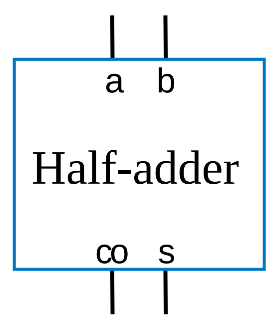
Step 2: Convert to equations

co = ab

s = a'b + ab' (same as s = a xor b)

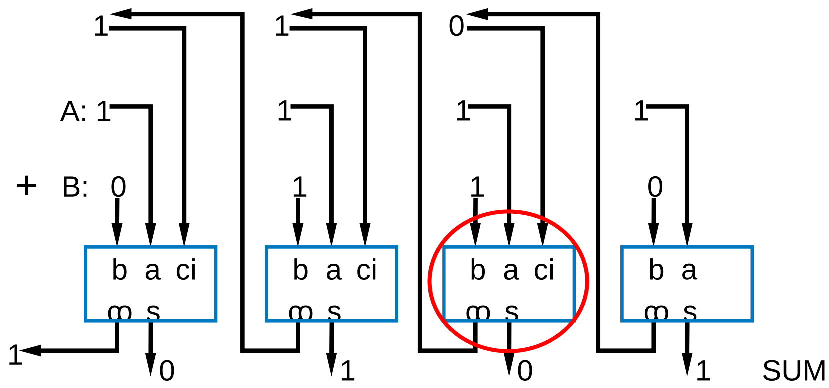
Step 3: Create the circuit





Full-Adder

- ***Full-adder:*** Adds 3 bits, generates sum and carry
- Design using combinational design process from Ch 2



<u>Step 1: Capture the function</u>

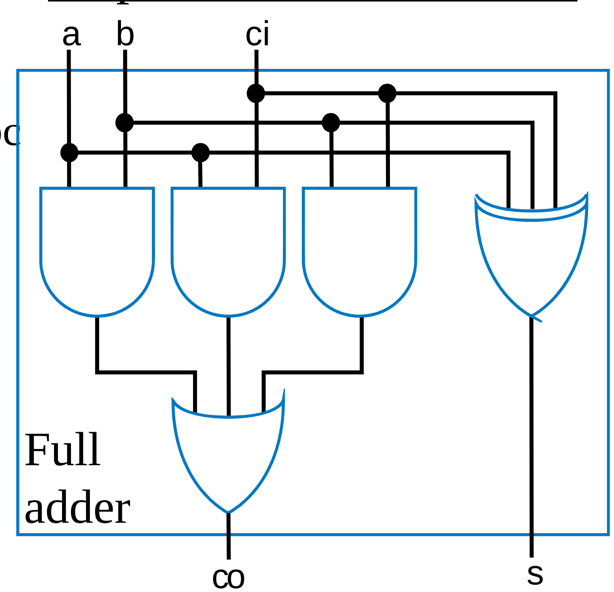
Inputs			Outputs	
a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

<u>Step 2: Convert to equations</u>

co = a’bc + ab’c + abc’ + abc
co = a’bc +abc +ab’c +abc +abc’ +abc
co = (a’+a)bc + (b’+b)ac + (c’+c)ab
co = bc + ac + ab

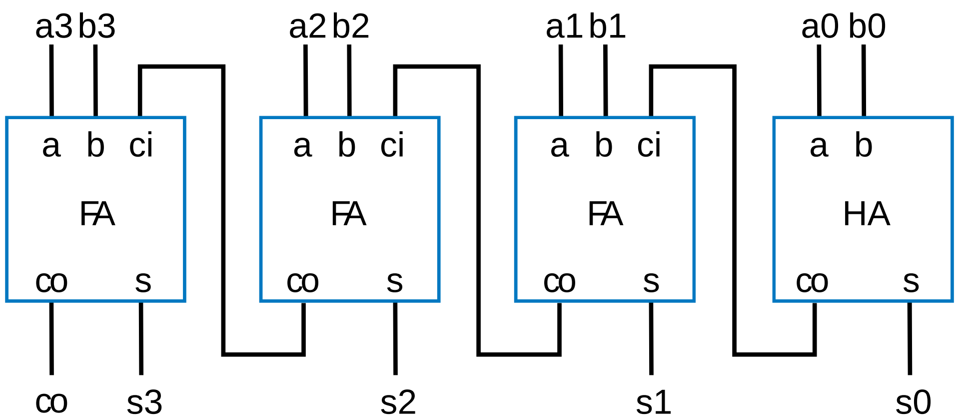
s = a’b’c + a’bc’ + ab’c’ + abc
s = a’(b’c + bc’) + a(b’c’ + bc)
s = a’(b xor c)’ + a(b xor c)
s = a xor b xor c

<u>Step 3: Create the circuit</u>

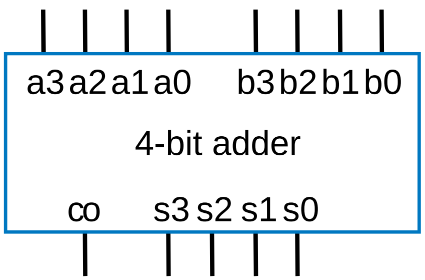


Carry-Ripple Adder

- Using half-adder and full-adders, we can build adder that adds like we would by hand
- Called a ***carry-ripple adder***
  - 4-bit adder shown: 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



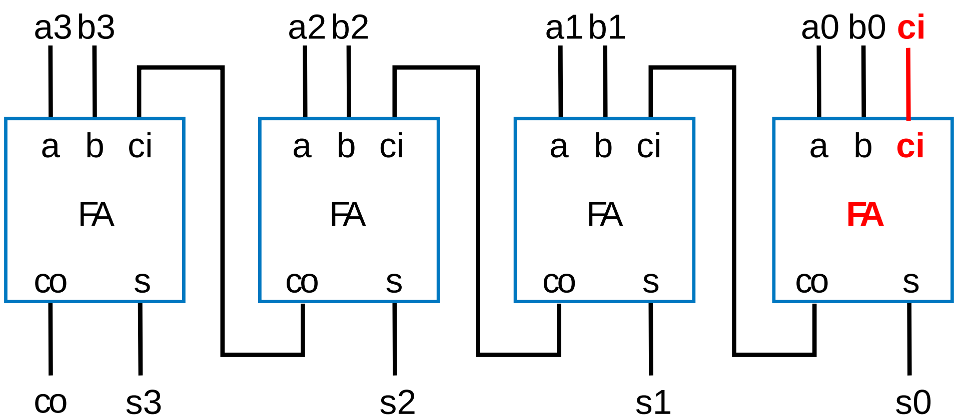
(a)



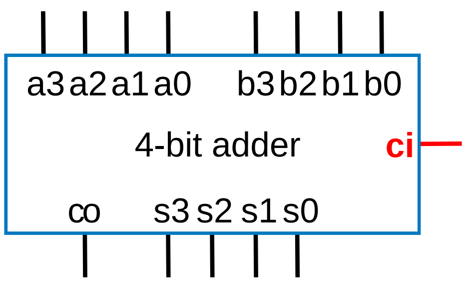
(b)

Carry-Ripple Adder

- Using full-adder instead of half-adder for first bit, we can include a "carry in" bit in the addition
  - Will be useful later when we connect smaller adders to form bigger adders

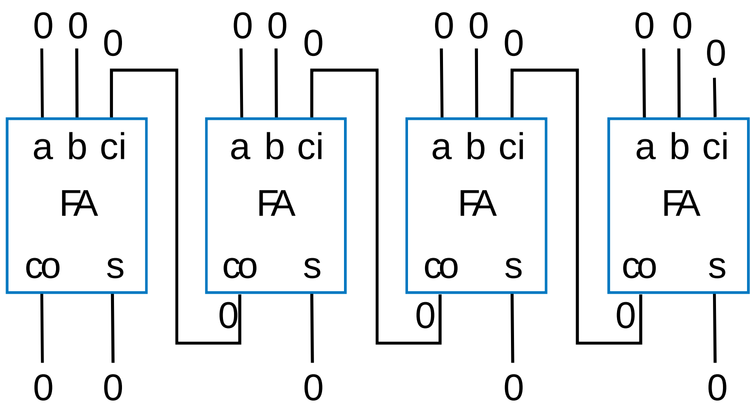


(a)

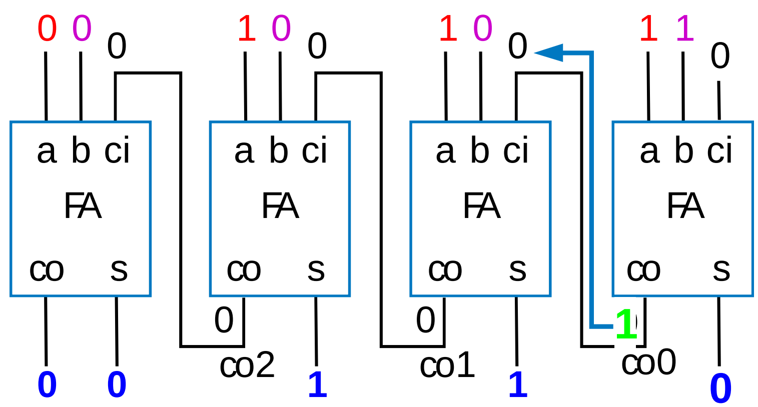


(b)

Carry-Ripple Adder's Behavior



Assume all inputs initially 0



a

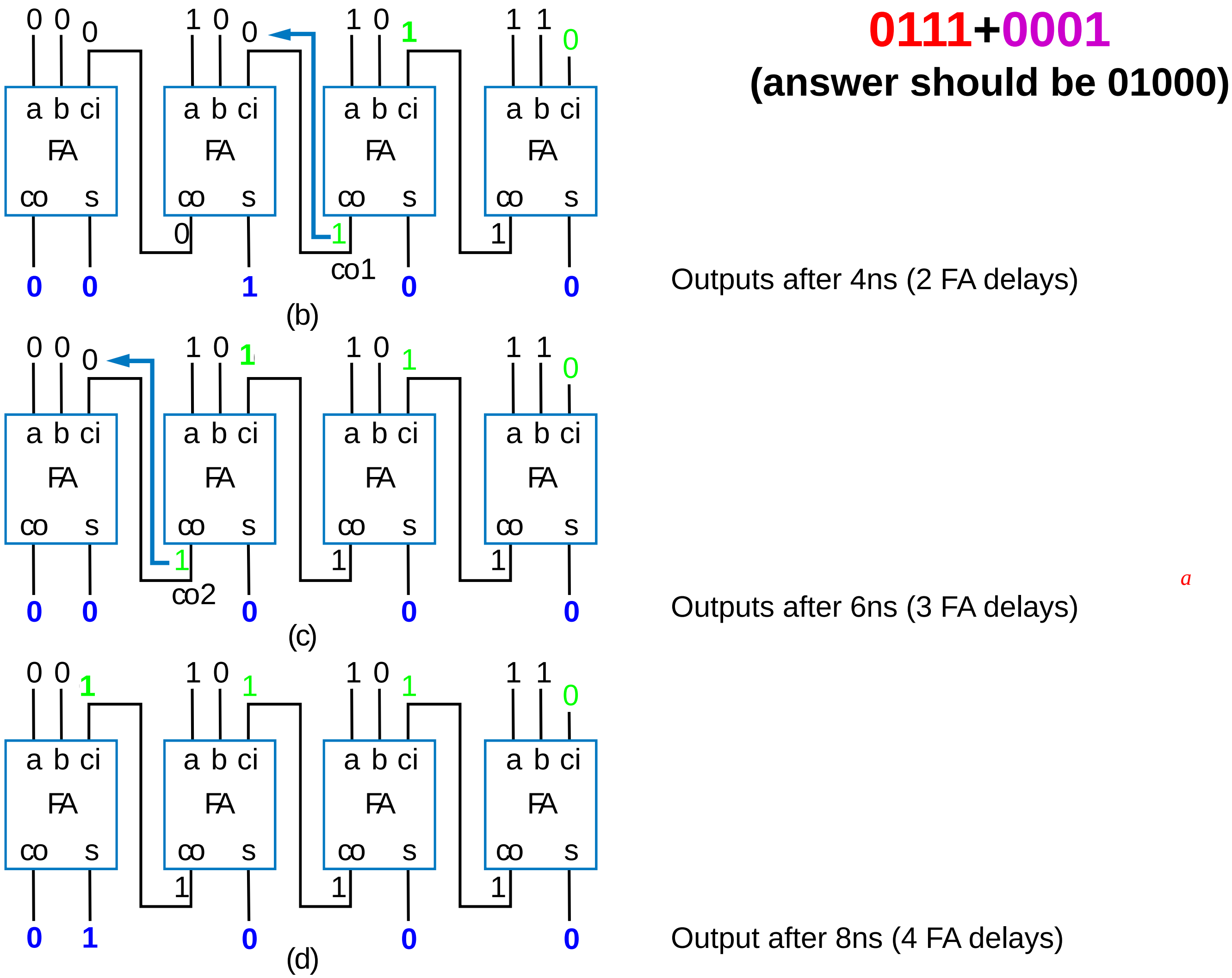
0111+0001

(answer should be 01000)

Output after 2 ns (1FA delay)

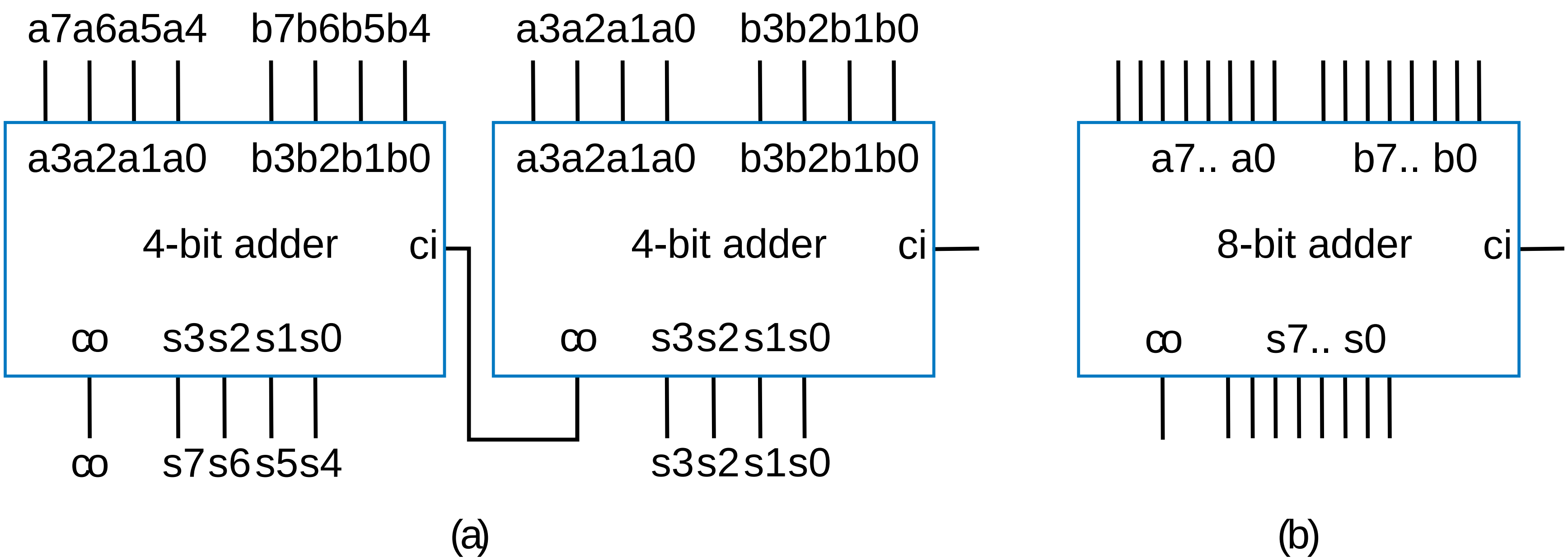
Wrong answer -- something wrong? No -- just need more time for carry to ripple through the chain of full adders.

Carry-Ripple Adder's Behavior



Correct answer appears after 4 FA delays

Cascading Adders



Multiplier – Array Style

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

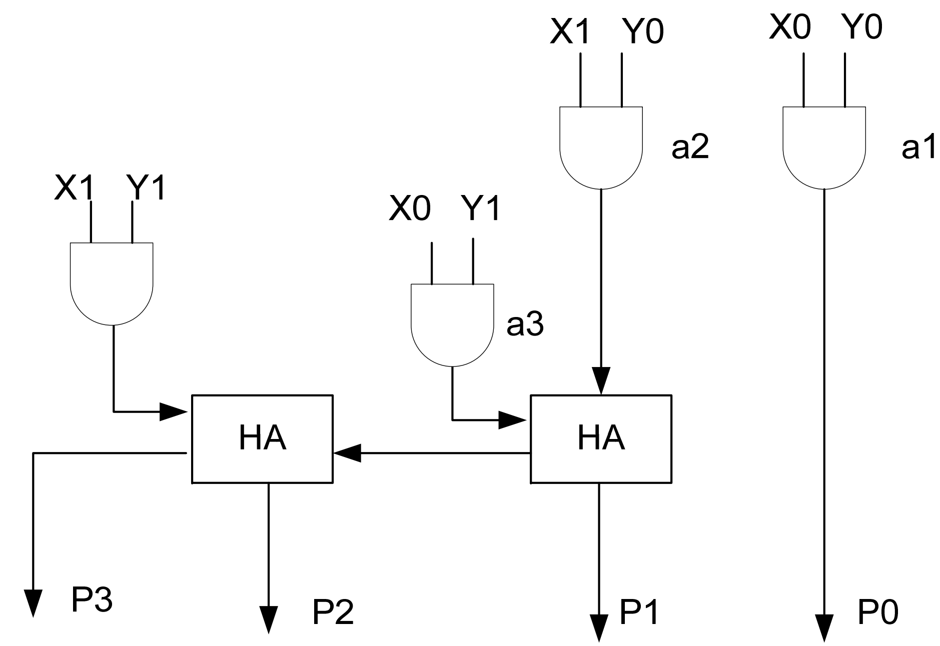
0110	(the top number is called the *multiplicand*)
0011	(the bottom number is called the *multiplier*)
----	(each row below is called a *partial product*)
0110	(because the rightmost bit of the multiplier is 1, and 0110*1=0110)
0110	(because the second bit of the multiplier is 1, and 0110*1=0110)
0000	(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)

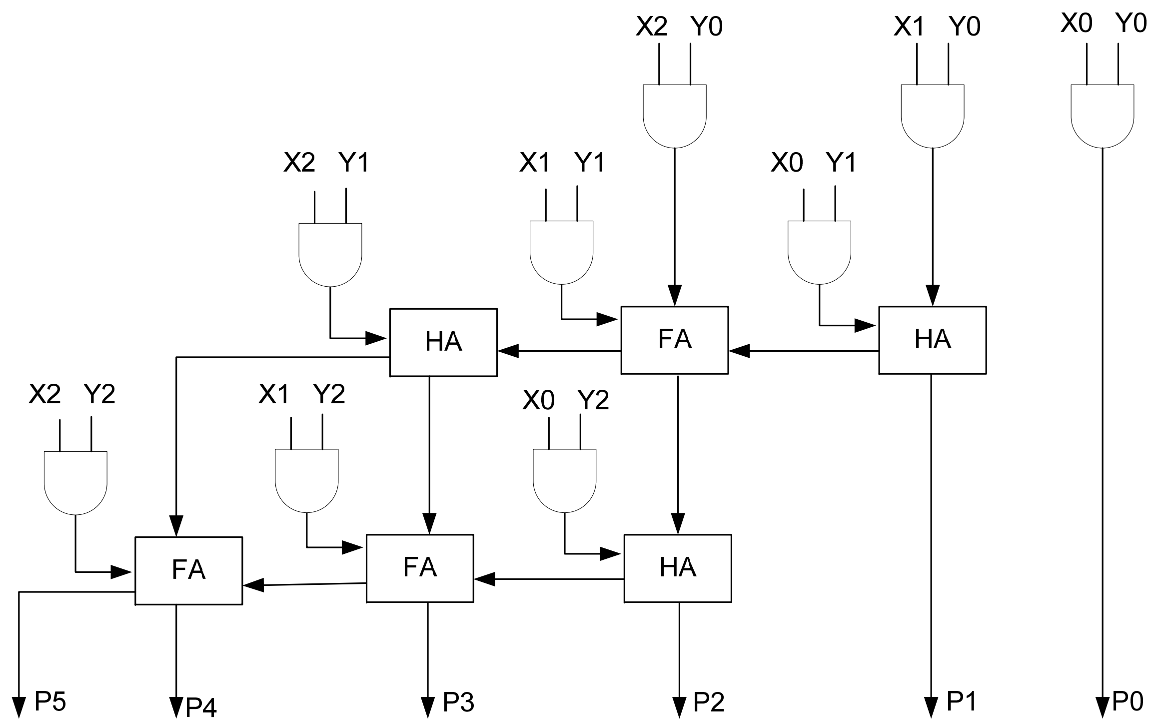
00010010	(the *product* is the sum of all the partial products: 18, which is 6*3)

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
```

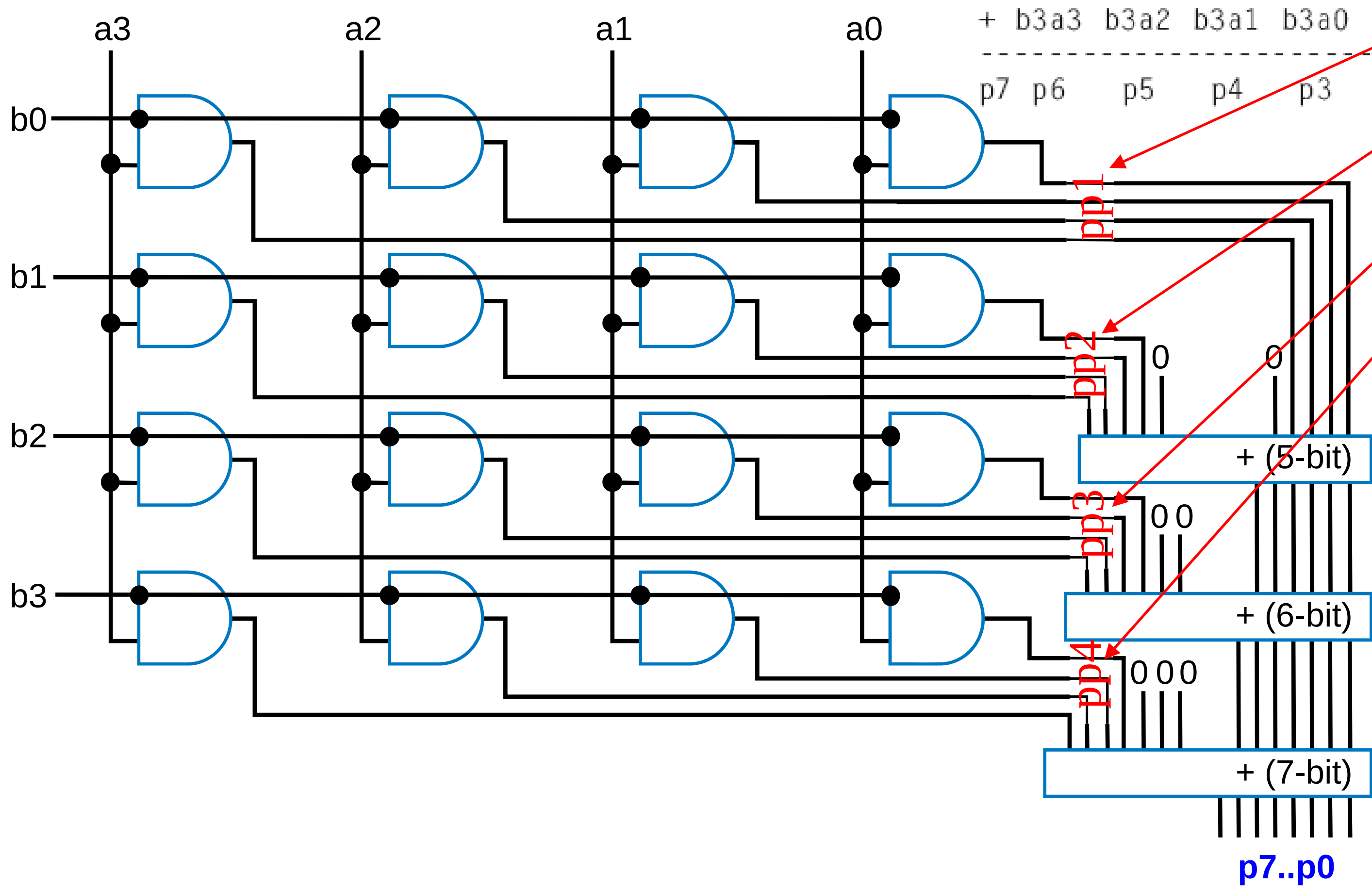

X1 Y0
X0 Y0
a2
a1
X1 Y1
X0 Y1
a3
HA
HA
P3
P2
P1
P0

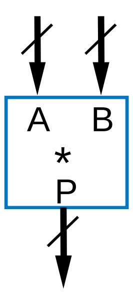

X2 Y0
X1 Y0
X0 Y0
X2 Y1
X1 Y1
X0 Y1
HA
FA
HA
X2 Y2
X1 Y2
X0 Y2
FA
FA
HA
P5
P4
P3
P2
P1
P0

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
```

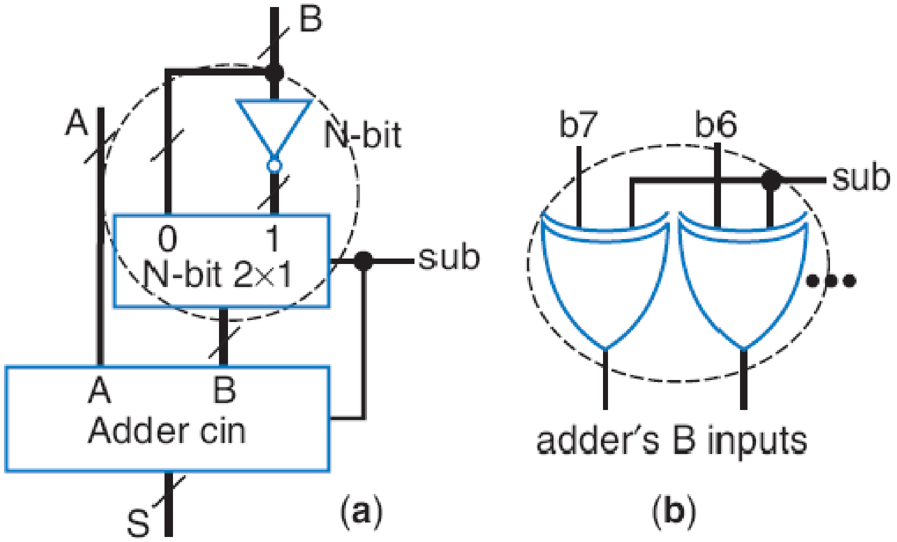




Block symbol

Adder/Subtractor

- Adder/subtractor: control input determines whether add or subtract
  - Can use 2x1 mux – sub input passes either B or inverted B
  - Alternatively, can use XOR gates – if sub input is 0, B's bits pass through; if sub input is 1, XORs invert B's bits



Overflow

- Sometimes result can't be represented with given number of bits
  - Either too large magnitude of positive or negative
  - e.g., 4-bit two's complement addition of 0111+0001 (7+1=8). But 4-bit two's complement can't represent number >7
    - 0111+0001 = 1000 WRONG answer, 1000 in two's complement is -8, not +8
  - Adder/subtractor should indicate when overflow has occurred, so result can be discarded

Detecting Overflow: Method 1

- Assuming 4-bit two's complement numbers, can detect overflow by detecting when the two numbers' sign bits are the same but are different from the result's sign bit
  - If the two numbers' sign bits are different, overflow is impossible
    - Adding a positive and negative can't exceed largest magnitude positive or negative
- Simple circuit
  - overflow = a3'b3's3 + a3b3s3'
  - Include "overflow" output bit on adder/subtractor

sign bits

(a)	(b)	(c)
0 1 1 1	1 1 1 1	1 0 0 0
+ 0 0 0 1	+ 1 0 0 0	+ 0 1 1 1
1 0 0 0	0 1 1 1	1 1 1 1
overflow	overflow	no overflow

If the numbers' sign bits have the same value, which differs from the result's sign bit, overflow has occurred.

Detecting Overflow: Method 2

- Even simpler method: Detect difference between carry-in to sign bit and carry-out from sign bit
- Yields simpler circuit: overflow = c3 xor c4

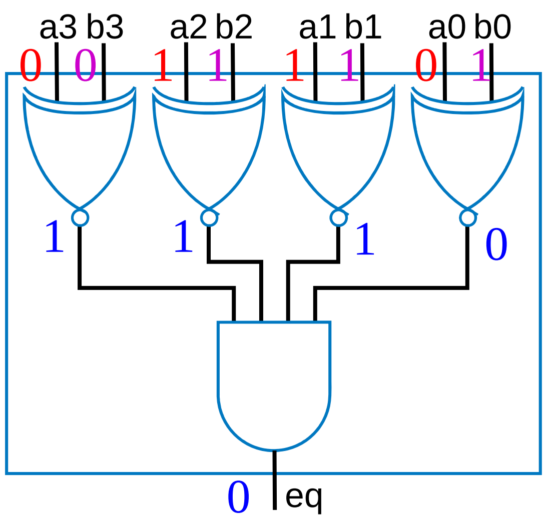
	(a)	(b)	(c)
carries	1 1 1	0 0 0	0 0 0
	0 1 1 1	1 1 1 1	1 0 0 0
	+ 0 0 0 1	+ 1 0 0 0	+ 0 1 1 1
sum	0 1 0 0 0	1 0 1 1 1	0 1 1 1 1
	overflow	overflow	no overflow

If the carry into the sign bit column differs from the carry out of that column, overflow has occurred.

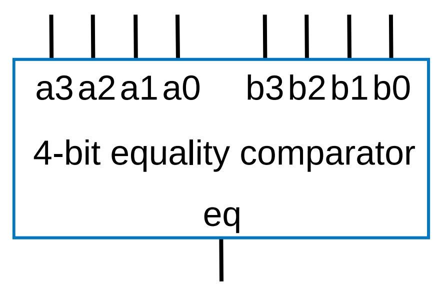
Comparators

- ***N-bit equality comparator***: Outputs 1 if two N-bit numbers are equal
  - 4-bit equality comparator with inputs A and B
    - a3 must equal b3, a2 = b2, a1 = b1, a0 = b0
      - Two bits are equal if both 1, or both 0
      - eq = (a3b3 + a3’b3’) * (a2b2 + a2’b2’) * (a1b1 + a1’b1’) * (a0b0 + a0’b0’)
    - Recall that XNOR outputs 1 if its two input bits are the same
      - eq = (a3 xnor b3) * (a2 xnor b2) * (a1 xnor b1) * (a0 xnor b0)

0110 = 0111 ?



(a)



(b)

a

Magnitude Comparator

- ***N-bit magnitude comparator***: Indicates whether A>B, A=B, or A<B, for its two N-bit inputs A and B
  - How design? Consider how compare by hand. First compare a3 and b3. If equal, compare a2 and b2. And so on. Stop if comparison not equal -- whichever's bit is 1 is greater. If never see unequal bit pair, A=B.

A=1011 B=1001

1011 **1**001 Equal

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

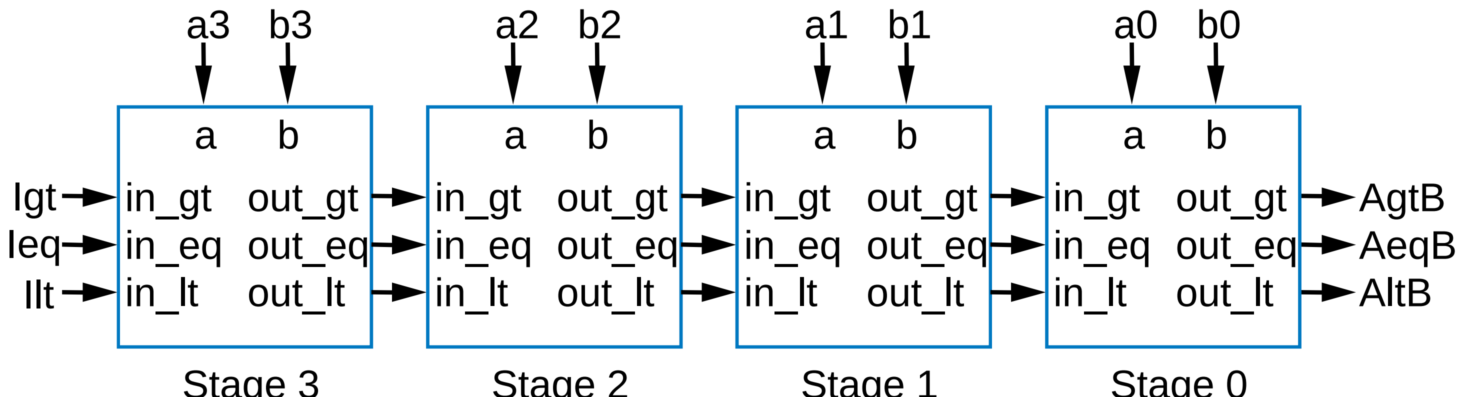
10**1**1 10**0**1 **Unequal**

So A > B

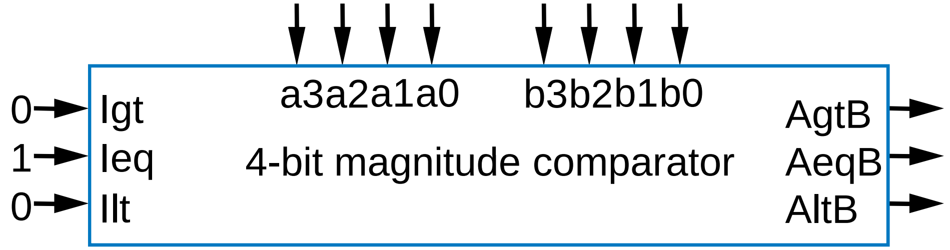
a

Magnitude Comparator

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

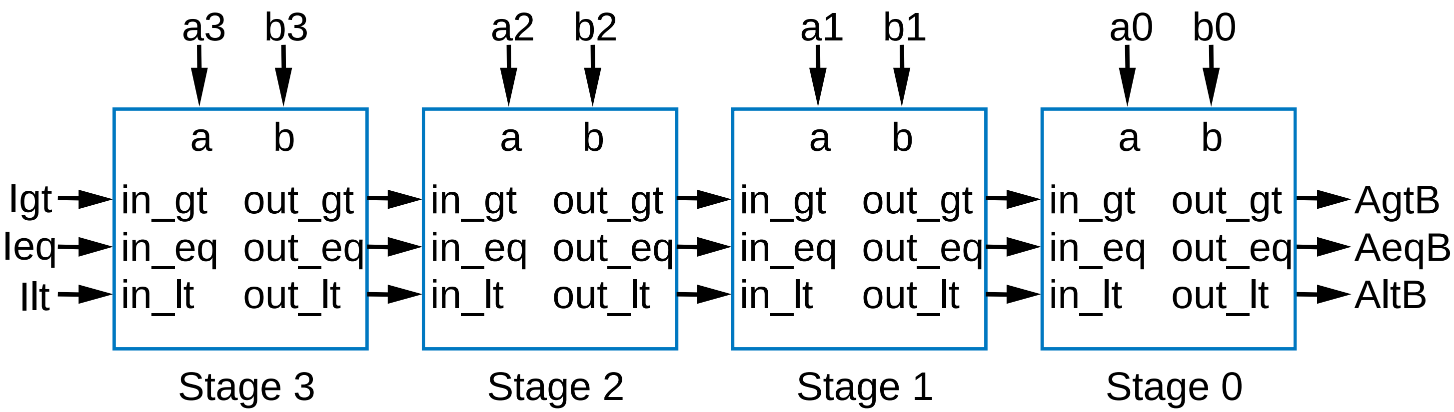


(a)



(b)

Magnitude Comparator



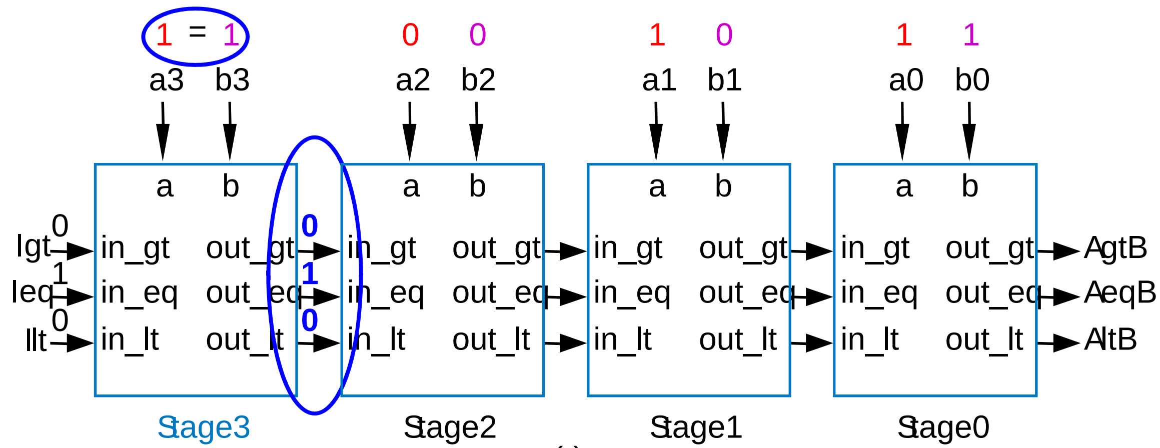
- Each stage:
  - out_gt = in_gt + (in_eq * a * b')
    - A>B (so far) if already determined in higher stage, or if higher stages equal but in this stage a=1 and b=0
  - out_lt = in_lt + (in_eq * a' * b)
    - A<B (so far) if already determined in higher stage, or if higher stages equal but in this stage 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

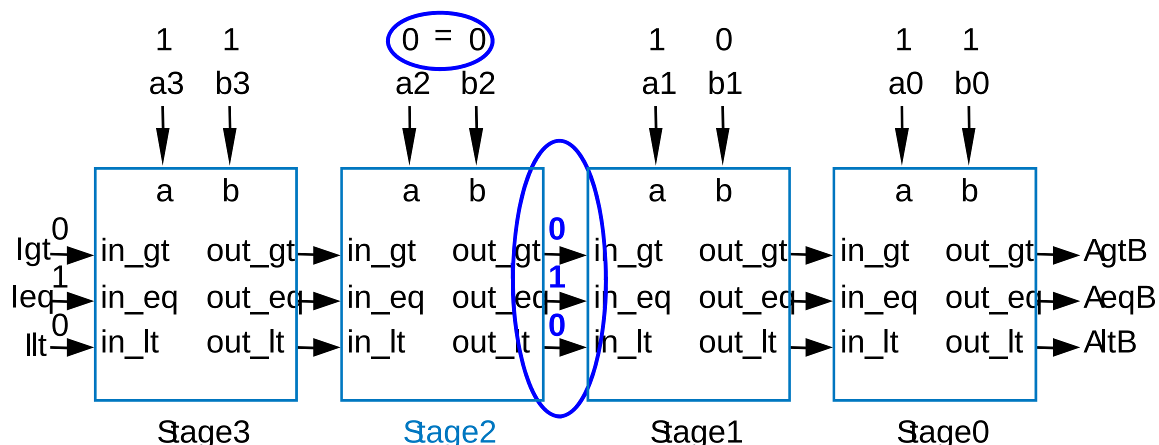
- How does it work?

Ieq=1 causes this stage to compare

1011 = 1001 ?



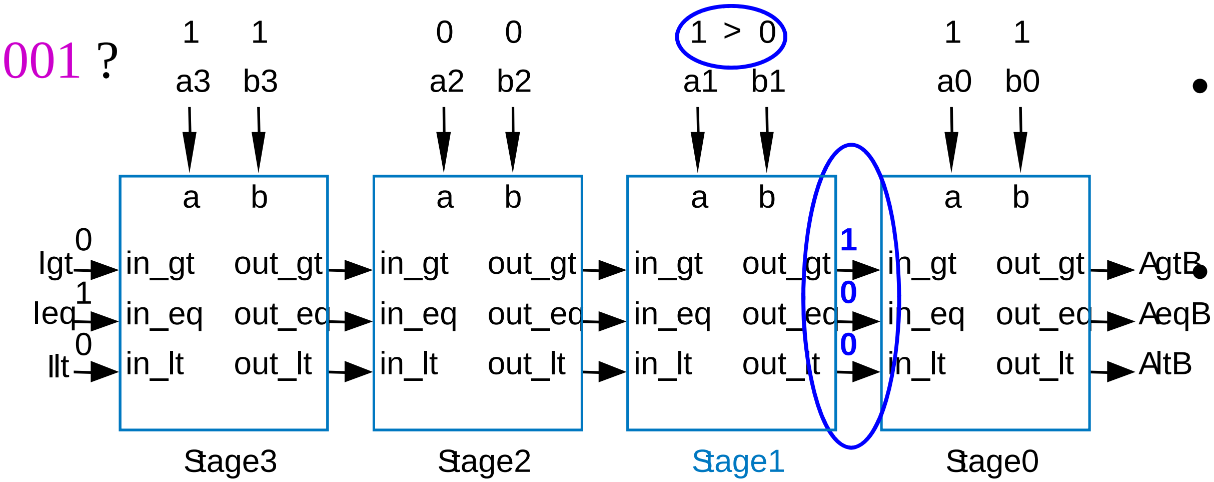
(a)



(b)

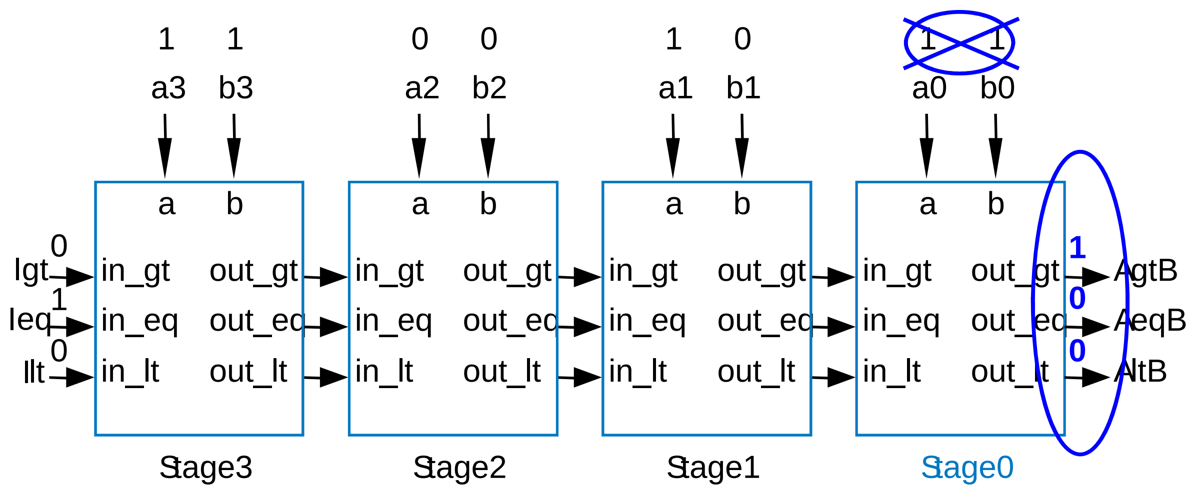
Magnitude Comparator

1011 = 1001 ?



(c)

a



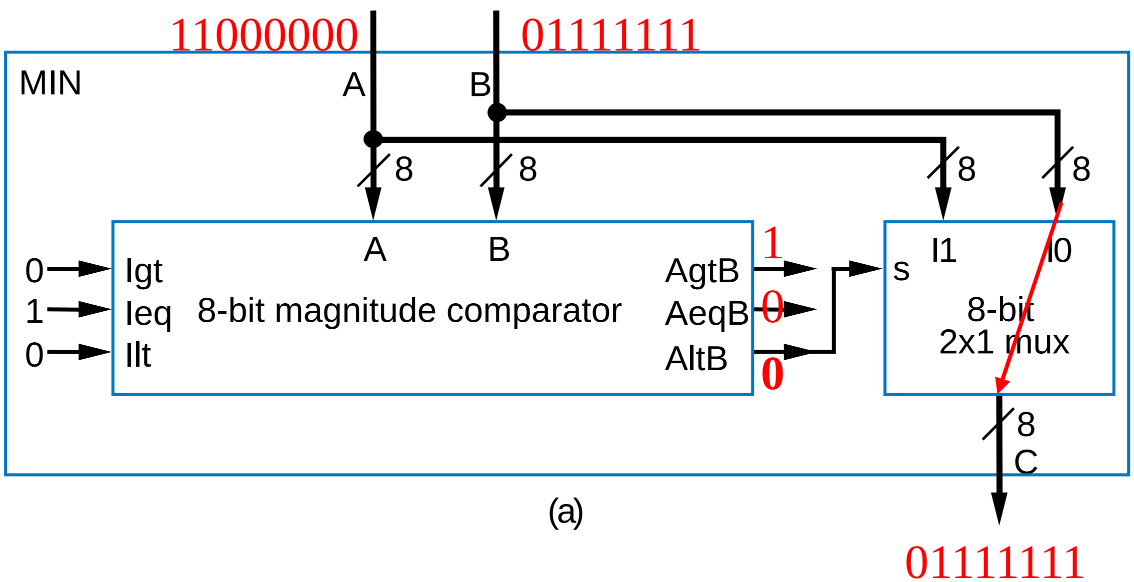
(d)

- Final answer appears on the right
- Takes time for answer to "ripple" from left to right
- Thus called "carry-ripple style" after the carry-ripple adder
  - Even though there's no "carry" involved

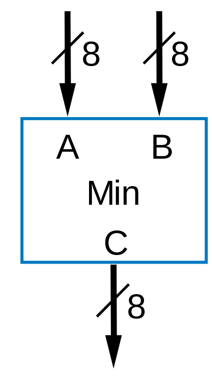
Magnitude Comparator Example: Minimum of Two Numbers

- Design a combinational component that computes 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

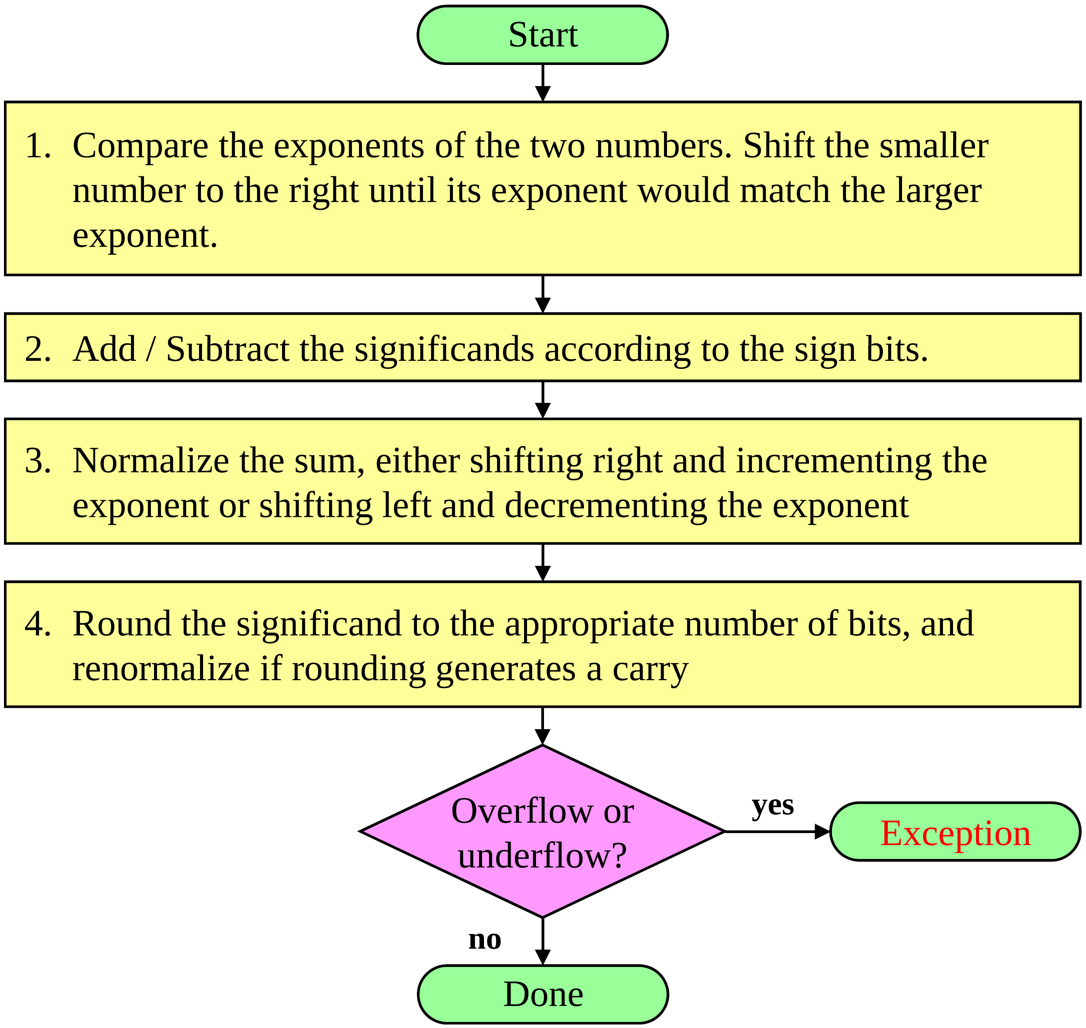


(a)



(b)

Floating Point Addition / Subtraction



Shift significand right by
$d = | E_X - E_Y |$

Add significands when signs of X and Y are identical,
Subtract when different
$X - Y$ becomes $X + (-Y)$

Normalization shifts right by 1 if there is a carry, or shifts left by the number of leading zeros in the case of subtraction

Rounding either truncates fraction, or adds a 1 to least significant fraction bit

Floating Point Adder Block Diagram