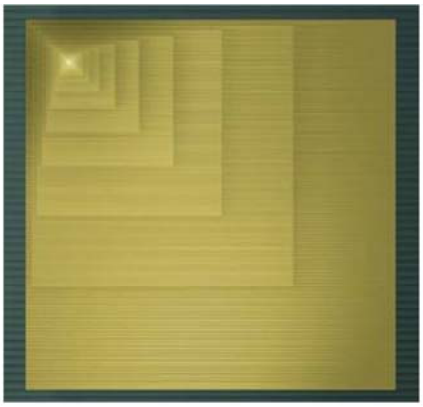


Digital Design

Chapter 4A: Datapath Components (Combinational)

Slides to accompany the textbook *Digital Design*, First Edition, by Frank Vahid, John Wiley and Sons Publishers, 2007.
http://www.ddvahid.com

Chapter 4 – Combinational Datapath Components

• More advanced common combinational building blocks, generally used to process data in a digital system.

• From the reading list on the website…

- 4.3 – Adders
- 4.5 – Comparators
- 4.7 – Multiplier – Array style
- 4.8 – Subtractor
- 4.9 – ALUs (Arithmetic-Logic Units)

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 of Ch 2, 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

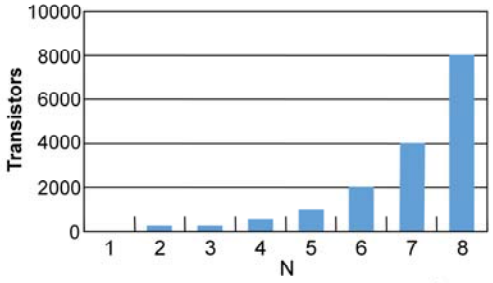
Why Adders Aren't Built Using Standard Combinational Design Process

- 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

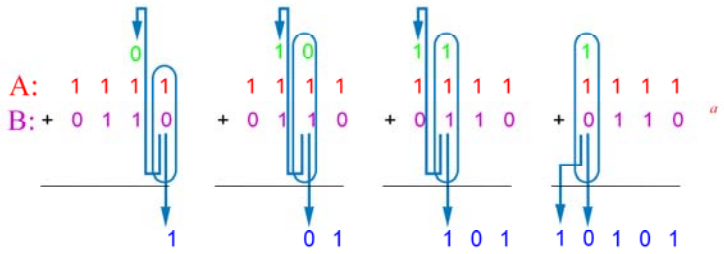
Q: Predict number of transistors for 16-bit adder

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!



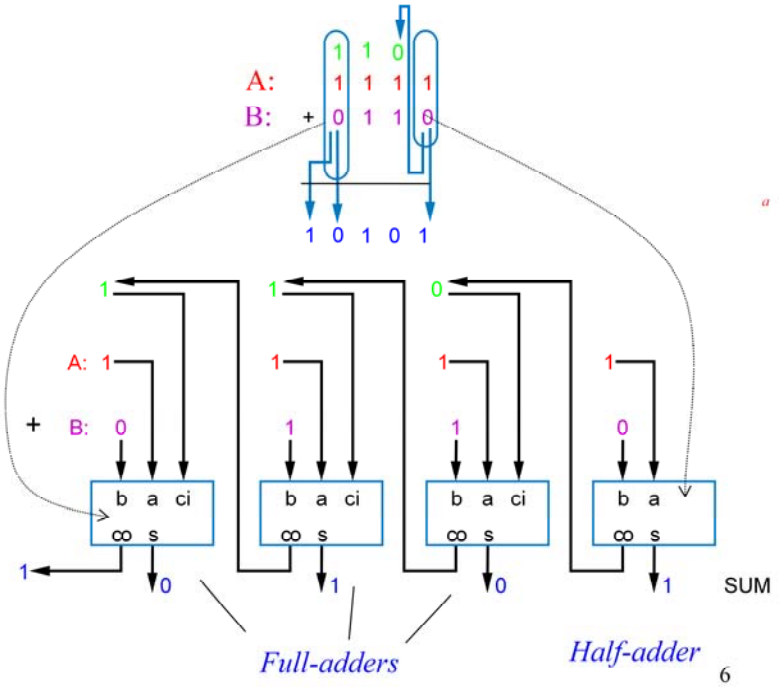
Alternative Method to Design an Adder: Imitate Adding by Hand

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



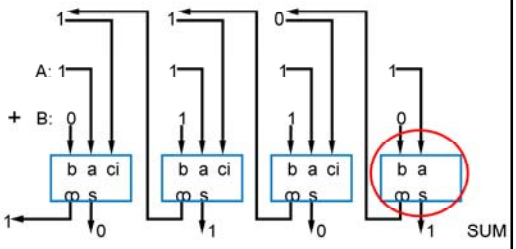
Alternative Method to Design an Adder: Imitate Adding by Hand

- 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

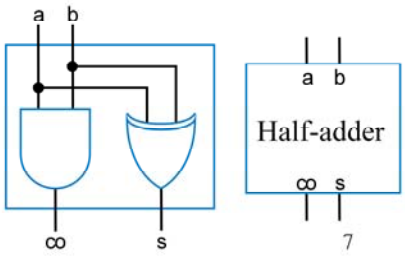
Inputs		Outputs	
a	b	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Step 2: Convert to equations

co = ab

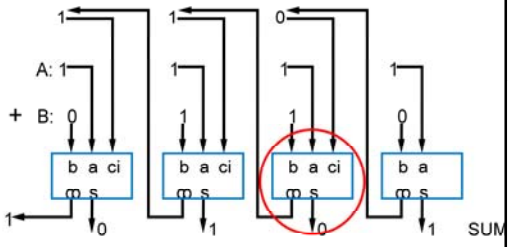
s = a'b + ab' (same as s = a xor b)

Step 3: Create the circuit



Full-Adder

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



Step 1: Capture the function

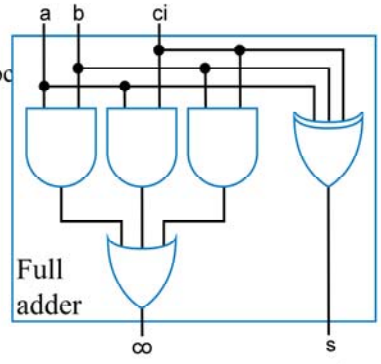
Inputs			Outputs	
a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Step 2: Convert to equations

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

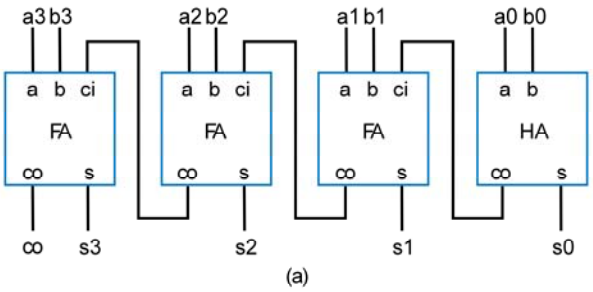
Step 3: Create the circuit



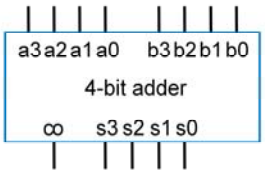
8

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



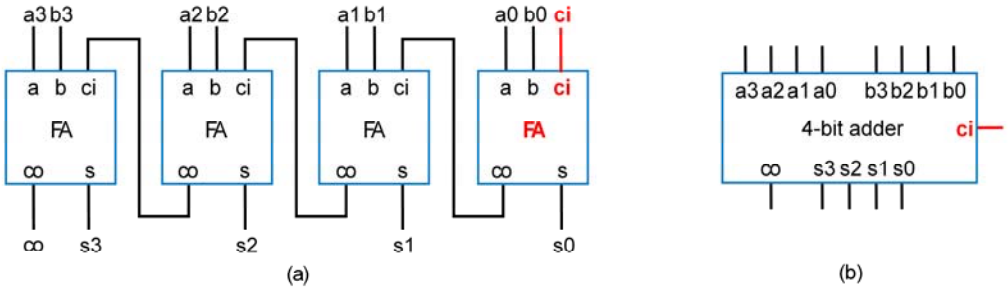
(a)



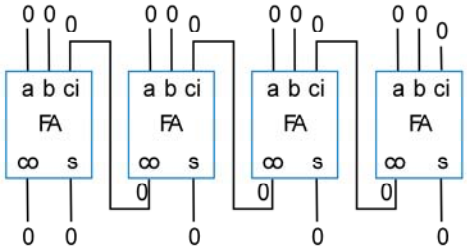
(b)

Carry-Ripple Adder

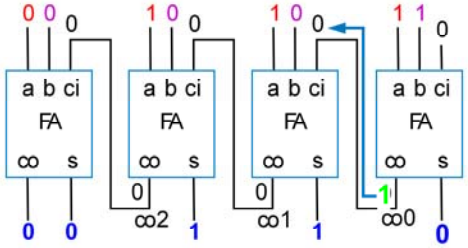
- Using full-adder instead of half-adder for first bit, we can include a “carry in” bit in the addition
  - Will be useful later when we connect smaller adders to form larger adders



Carry-Ripple Adder's Behavior



Assume all inputs initially 0

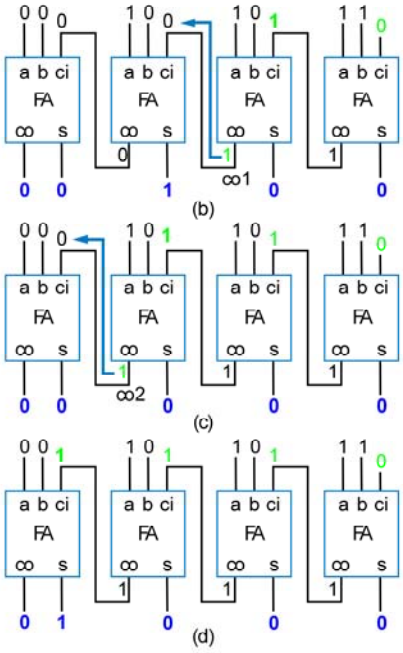


0111 + 0001
(answer should be 01000)

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)

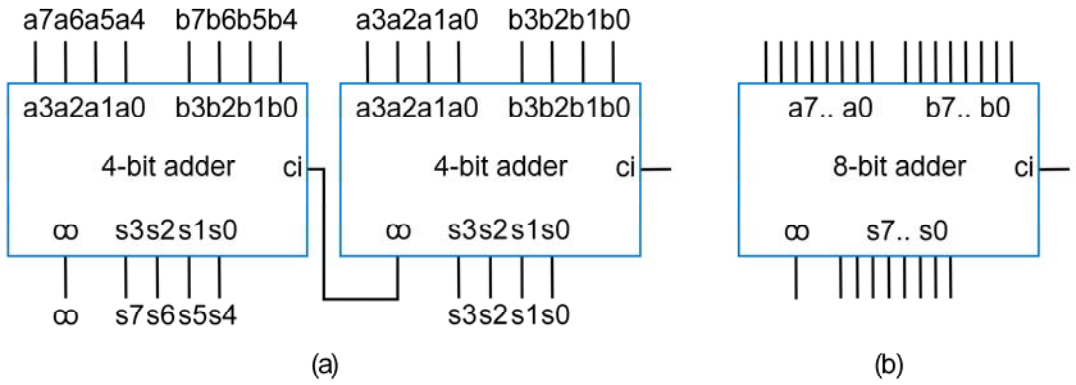
Outputs after 4ns (2 FA delays)

Outputs after 6ns (3 FA delays)

Output after 8ns (4 FA delays)

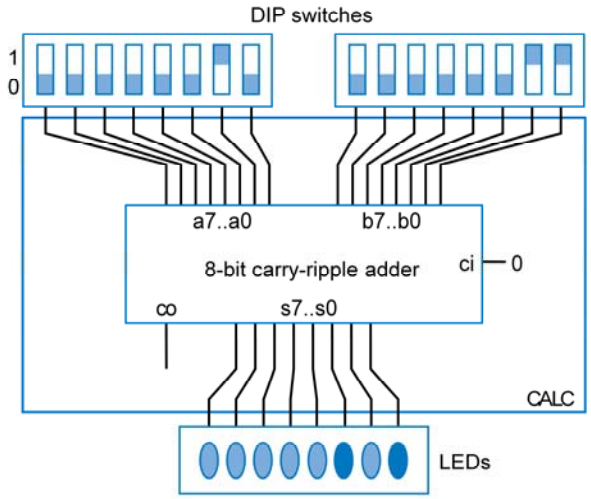
Correct answer appears after 4 FA delays

Cascading Adders



Adder Example: DIP-Switch-Based Adding Calculator

- Goal: Create calculator that adds two 8-bit binary numbers, specified using DIP switches
  - DIP switch: Dual-inline package switch, move each switch up or down
  - Solution: Use 8-bit adder

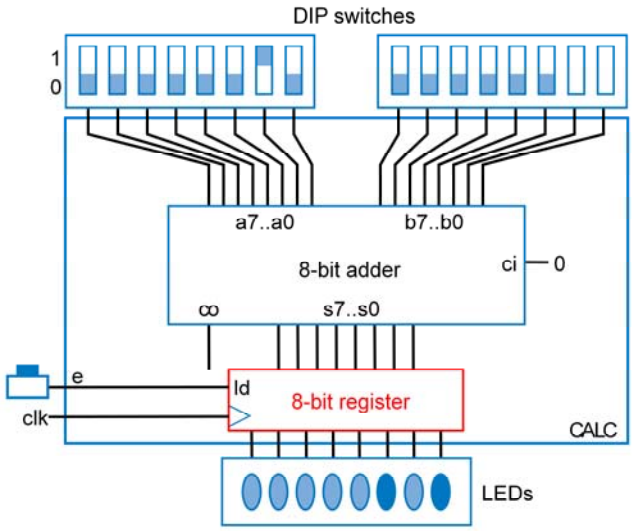


Adder Example: DIP-Switch-Based Adding Calculator

- To prevent spurious values from appearing at output, can place register at output
  - Actually, the light flickers from spurious values would be too fast for humans to detect -- but the principle of registering outputs to avoid spurious values being read by external devices (which normally aren't humans) applies here.

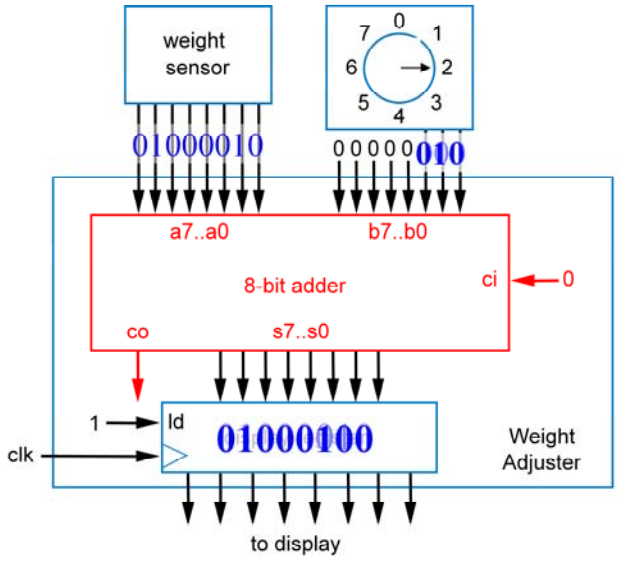
For now think of a register as a way to hold a value when ld is asserted.

We will learn more about registers later in this course.



Adder Example: Compensating Weight Scale

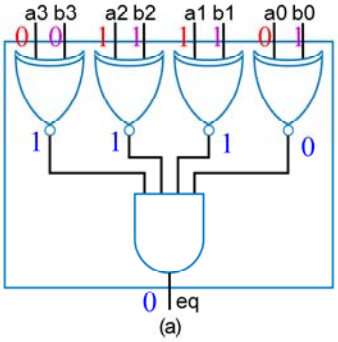
- Weight scale with compensation amount of 0-7
  - To compensate for inaccurate sensor due to physical wear
  - Use 8-bit adder

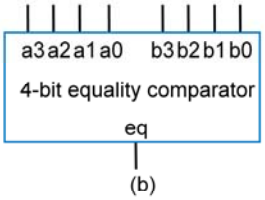


Comparators

- ***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)

$0110 = 0111$?





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

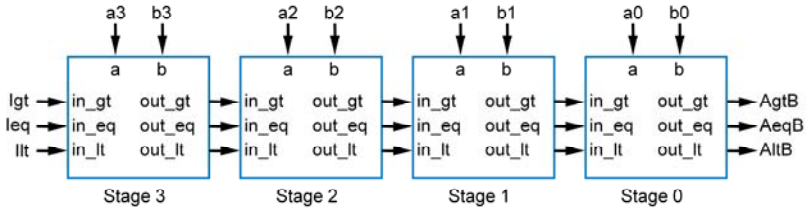
1**0**11 1**0**01 Equal

101**1** 100**1** **Unequal**

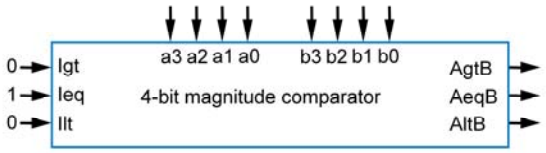
So A > B

Magnitude Comparator

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

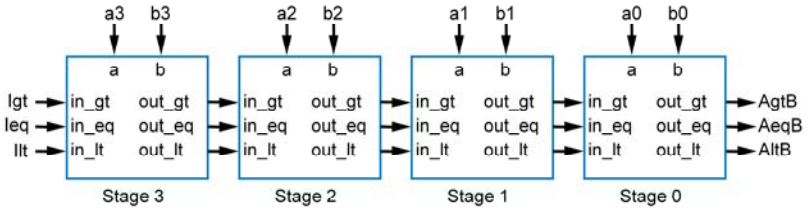


(a)



(b)

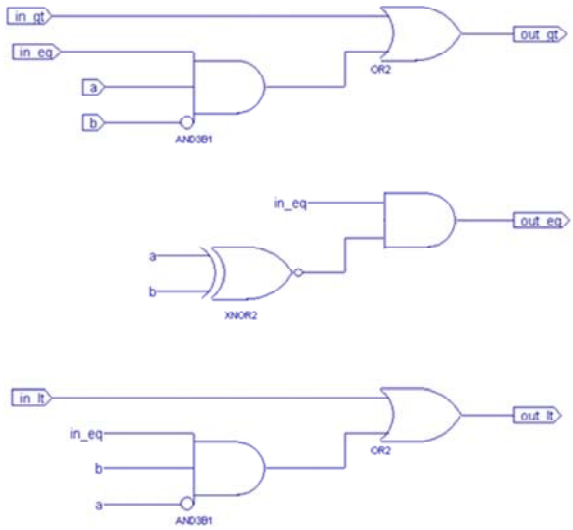
Magnitude Comparator



- 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)

Internal Stage design

- *Student activity:* take a couple minutes and draw the circuit for the internals of the each stage.

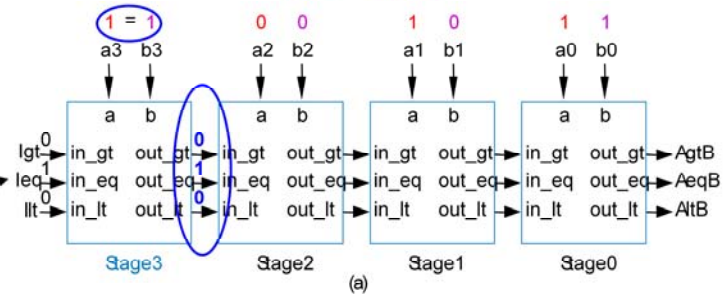


Magnitude Comparator

- How does it work?

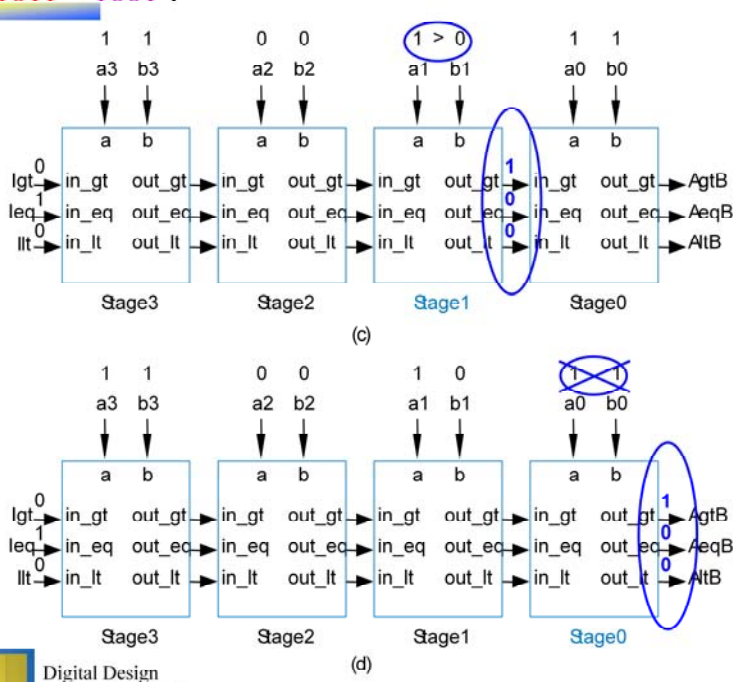
1011 = 1001 ?

Ieq=1 causes this stage to compare



Magnitude Comparator

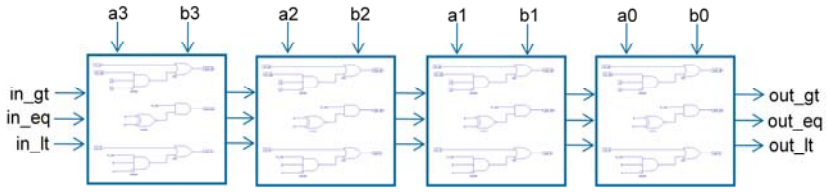
1011 = 1001 ?



- 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

Cascaded Comparator

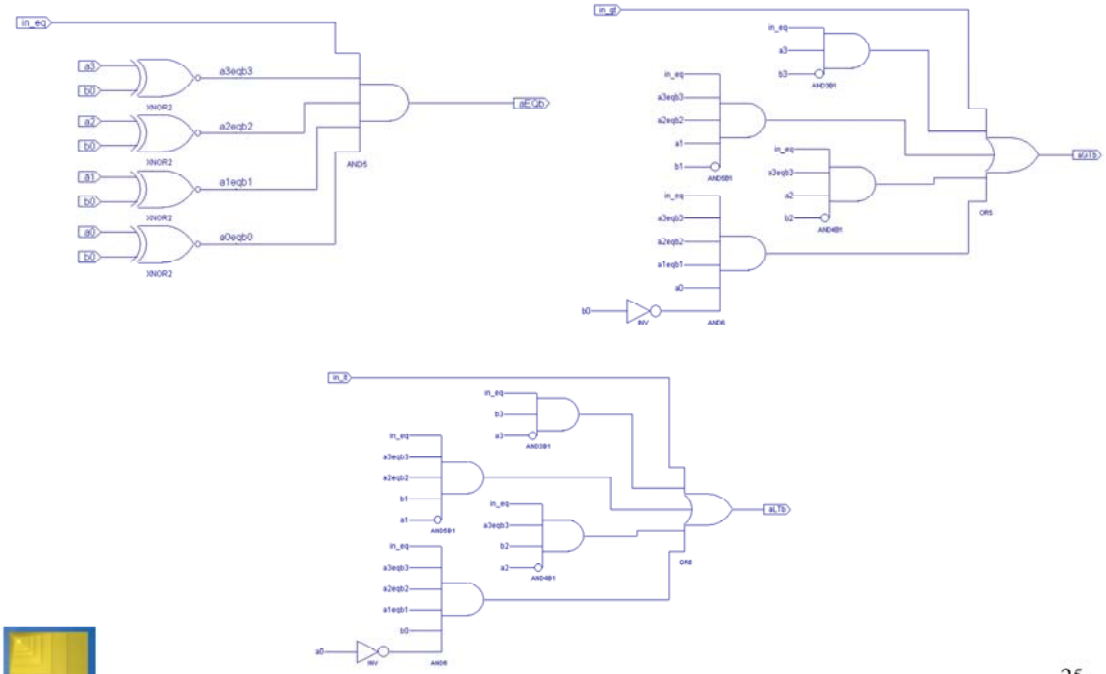
- Cascade these to make a 4-bit comparator



- What is the worst case propagation delay through this 4-bit Comparator?
  - Assume the propagation delay through an AND or an OR gate is tp, and the delay through an XOR gate is 2*tp … now calculate number of tp's.

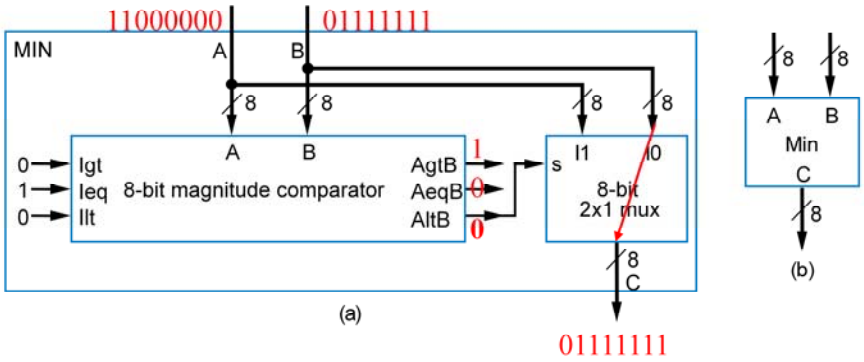
Answer: Worst case through a single cell is 3*tp (from a or b thru to out_eq). That change on that signal now takes 1*tp to get through the next stage and so, on. Unless one of the subsequent stages are less than or greater than (2*tp). Thus, worst case is 3*tp + 2*(1*tp) + 2*tp = 7*tp.

Alternative faster design



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.



Multiplier – Array Style

- Design a multiplier by mimicing 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)
```

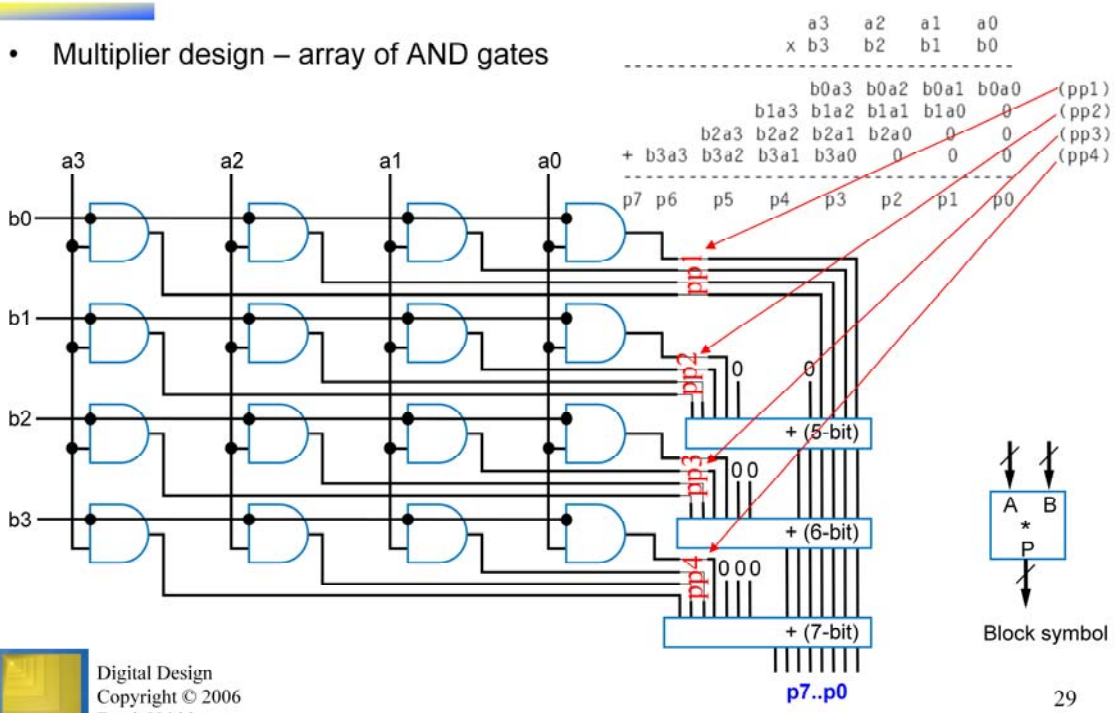
Multiplier – Array Style

- Generalized representation of multiplication by hand

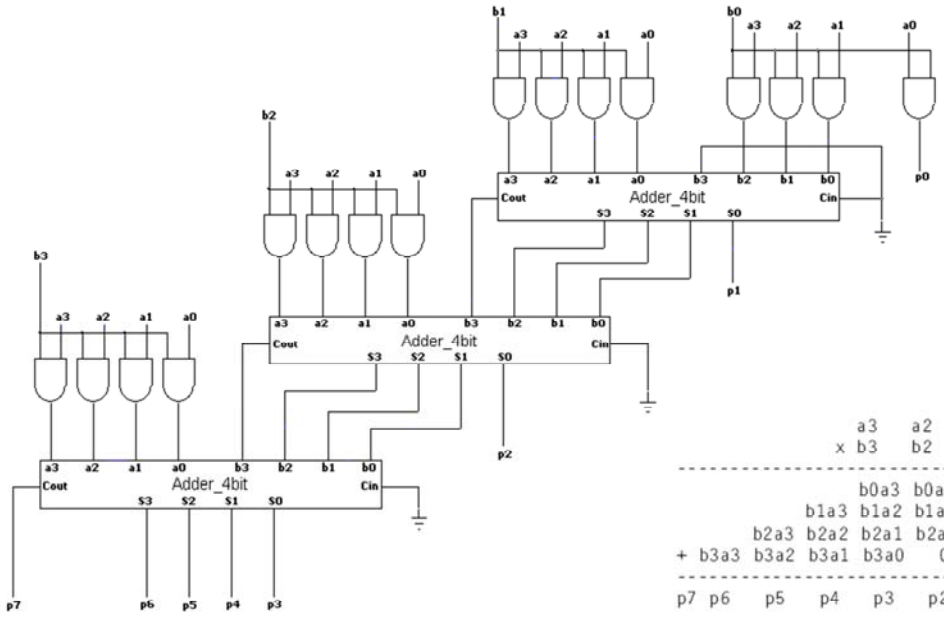
```
                  a3   a2   a1   a0
                x b3   b2   b1   b0
-------------------------------------
                  b0a3 b0a2 b0a1 b0a0   (pp1)
             b1a3 b1a2 b1a1 b1a0   0    (pp2)
        b2a3 b2a2 b2a1 b2a0   0    0    (pp3)
+ b3a3 b3a2 b3a1 b3a0   0    0    0    (pp4)
-------------------------------------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

Multiplier – Array Style

- Multiplier design – array of AND gates

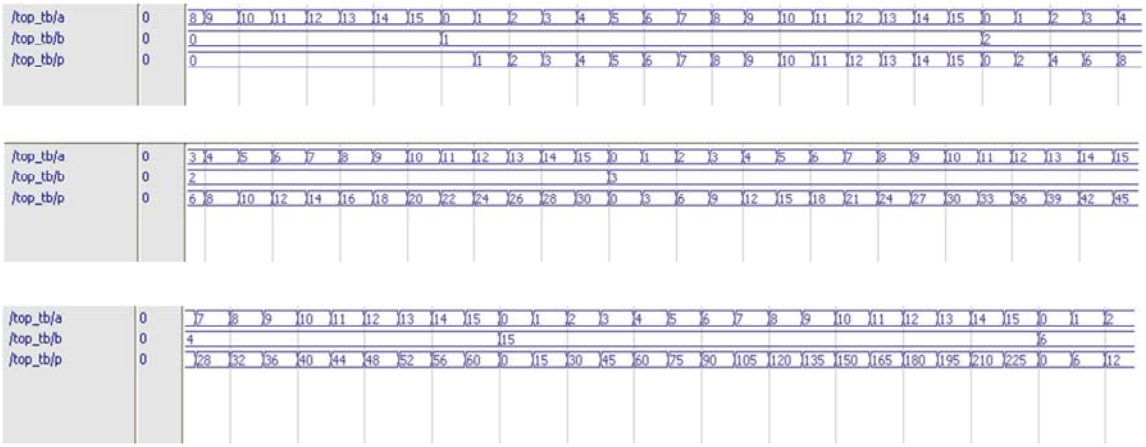


Multiplier – Array Style (alternative)




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

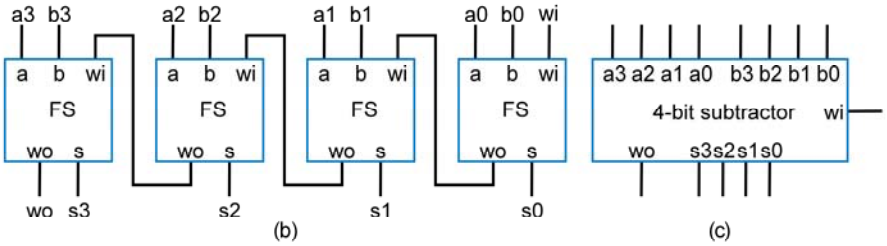
Multiplier – Simulation of alt. crkt.



Notice: all the bit-vectors (busses) are shown in unsigned integer number format. This is accomplished, in Modelsim, by right clicking on signal name and selecting (Radix -> Unsigned).

Subtractor

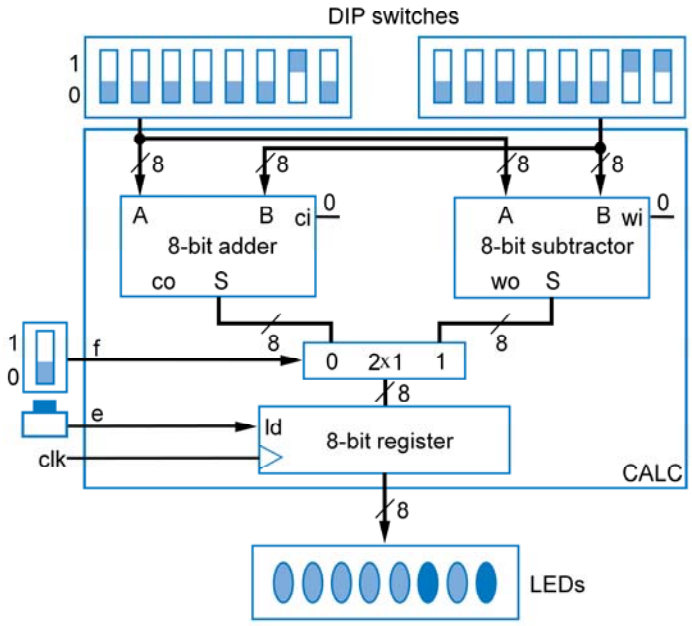
- Can build subtractor as we built carry-ripple adder
  - Mimic subtraction by hand
  - Compute borrows from columns on left
    - Use full-subtractor component:
      - wi is borrow by column on right, wo borrow from column on left



- In practice, it would be designed differently.
- Use signed numbers and { A – B → A + (-B) }
  - more on this later …

Subtractor Example: DIP-Switch Based Adding/Subtracting Calculator

- Extend earlier calculator example
  - Switch f indicates whether want to add (f=0) or subtract (f=1)
  - Use subtractor and an 8-bit, 2x1 mux



Subtractor Example:

Color Space Converter – RGB to CMYK

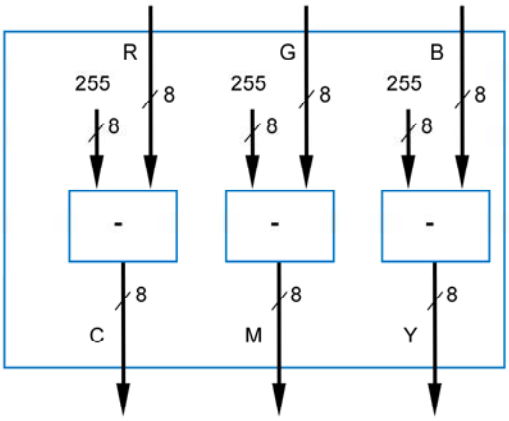
- Color
  - Often represented as weights of three colors: red, green, and blue (RGB)
    - Perhaps 8 bits each, so specific color is 24 bits
      - White: R=11111111, G=11111111, B=11111111
      - Black: R=00000000, G=00000000, B=00000000
      - Other colors: values in between, e.g., R=00111111, G=00000000, B=00001111 would be a reddish purple
  - Good for computer monitors, which mix red, green, and blue lights to form all colors

- Printers use opposite color scheme
  - Because inks *absorb* light
  - Use complementary colors of RGB: Cyan (absorbs red), reflects green and blue, Magenta (absorbs green), and Yellow (absorbs blue)

Subtractor Example:

Color Space Converter – RGB to CMYK

- Printers must quickly convert RGB to CMY
  - C=255-R, M=255-G, Y=255-B
  - Use subtractors as shown



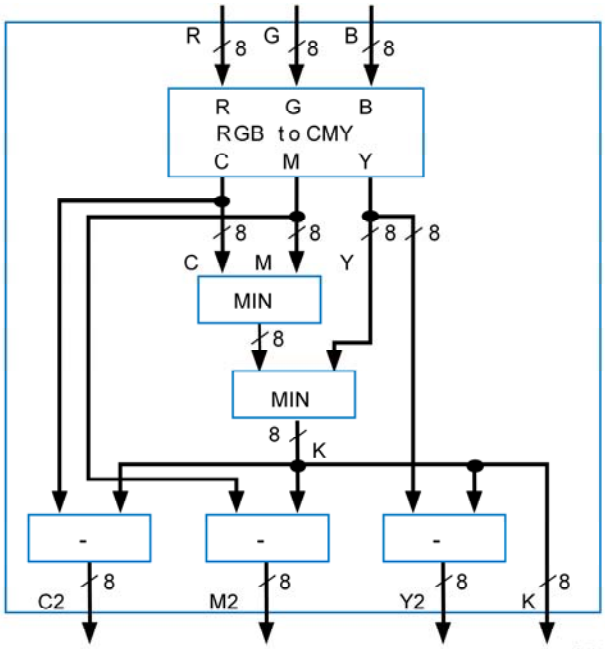
Subtractor Example:
Color Space Converter – RGB to CMYK

- Try to save colored inks
  - Expensive
  - Imperfect – mixing C, M, Y doesn't yield good-looking black
- Solution: Factor out the black or gray from the color, print that part using black ink
  - e.g., CMY of (250,200,200)= (200,200,200) + (50,0,0).
    - (200,200,200) is a dark gray – use black ink

Subtractor Example:

Color Space Converter – RGB to CMYK

- Call black part K
  - (200,200,200): K=200
  - (Letter "B" already used for blue)
- Compute minimum of C, M, Y values
  - Use MIN component designed earlier, using comparator and mux, to compute K
  - Output resulting K value, and subtract K value from C, M, and Y values
  - Ex: Input of (250,200,200) yields output of (50,0,0,200)

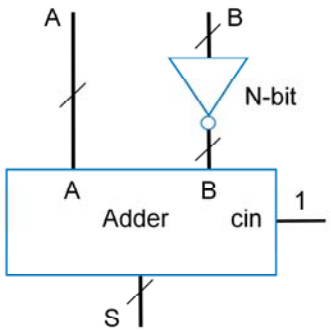


Representing Negative Numbers: 2's Complement

- Negative numbers common
  - How represent in binary?
  - As we already know, 2's complement is the most common method of representing negative numbers in binary.
- 2's complement
  - Big advantage: Allows us to perform subtraction and addition using the same circuit.
  - Thus, only need adder component, no need for separate subtractor component!

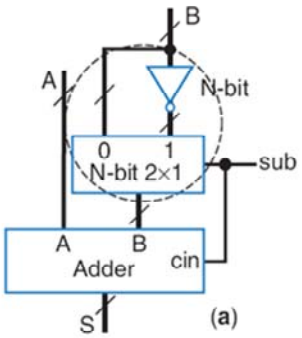
Two's Complement Subtractor Built with an Adder

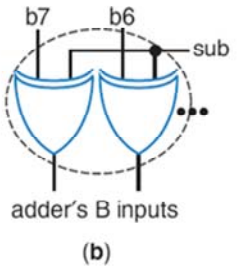
- Using two's complement
  A – B = A + (-B)
  = A + (two's complement of B)
  = A + invert_bits(B) + 1
- So build subtractor using adder by inverting B's bits, and setting carry in to 1



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



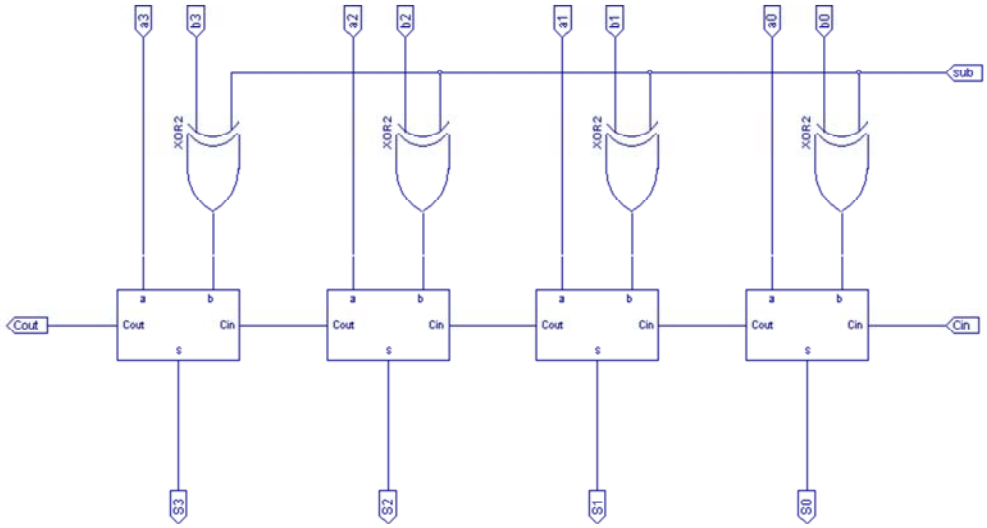


- Look at it another way … what we need is a selectable buffer/inverter …
- A 2-input XOR gate acts like a selectable buffer/inverter … connect as shown above (b).
  - when `cntl = 0`, then `out = in` (addition)
  - when `cntl = 1`, then `out = in'` (subtraction)

XOR

cntl	in	Out
0	0	0
0	1	1
1	0	1
1	1	0

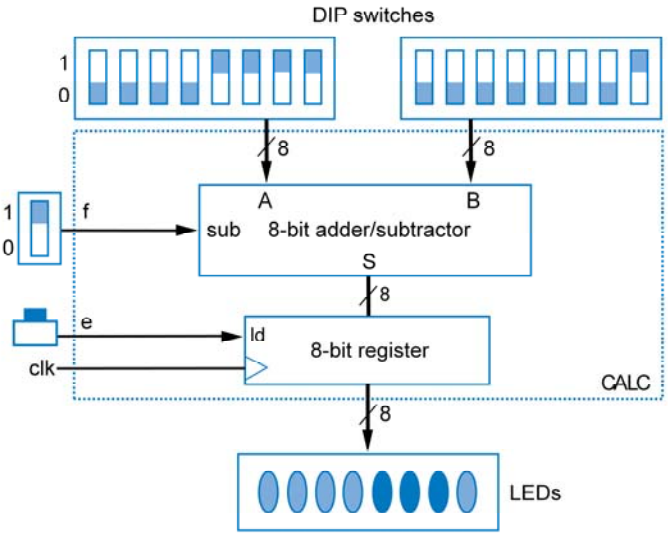
Adder/Subtractor Circuit

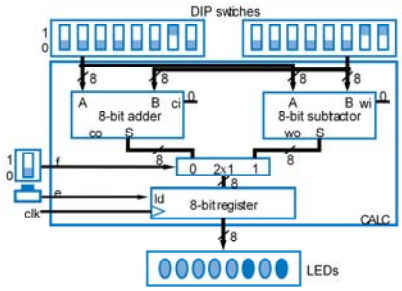


41

Adder/Subtractor Example: Calculator

- Previous calculator used separate adder and subtractor
- Improve by using adder/subtractor, and two's complement numbers



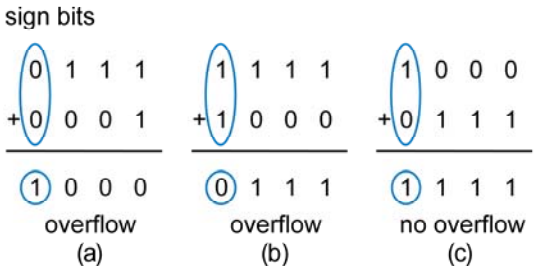


Overflow

- 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

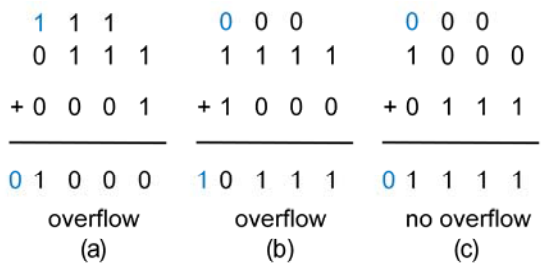
- Assuming 4-bit two's complement numbers, can detect overflow by detecting when the two numbers' sign bits are the same but are different from the result's sign bit
  - If the two numbers' sign bits are different, overflow is impossible
    - Adding a positive and negative can't exceed largest magnitude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



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

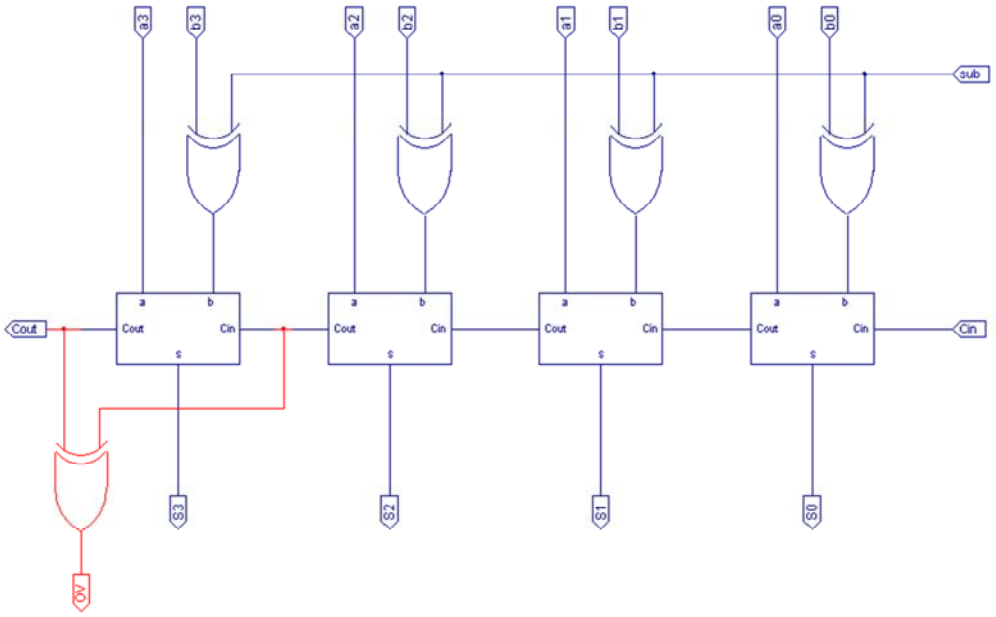
Detecting Overflow: Method 2

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



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

Adder/Subtractor w/ Overflow: Method 2 (circuit)



46

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
  - Addition/subtraction
  - Increment
  - Bitwise AND, OR, NOT
- Motivation:
  - Suppose want multi-function calculator that not only adds and subtracts, but also increments, ANDs, ORs, XORs, etc.
  - All microprocessors have ALUs

TABLE 4.2 Desired calculator operations

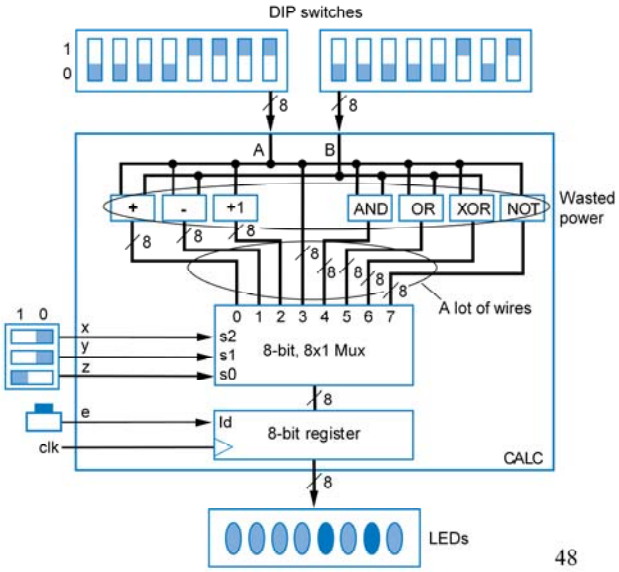
Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1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

Multifunction Calculator without an ALU

- Can build multifunction calculator using separate components for each operation, and muxes
  - But, too many wires, and wasted power computing all those operations when at any time you only use one of the results

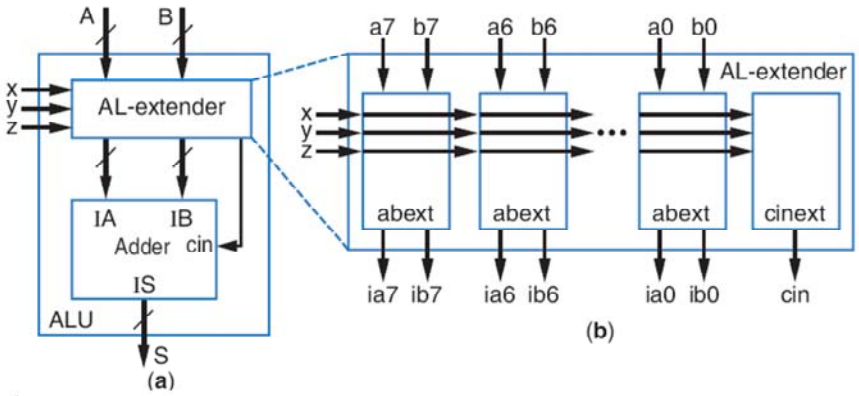
TABLE 4.2 Desired calculator operations

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



ALU

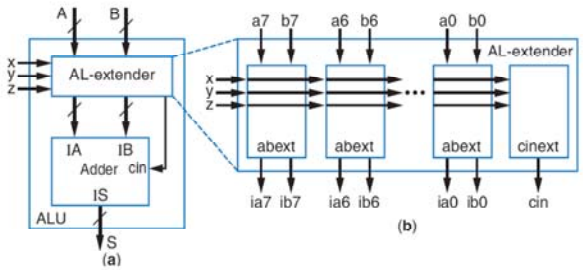
- More efficient design uses ALU
  - ALU design not just separate components multiplexed (same problem as previous slide!),
  - Instead, ALU design uses single adder, plus logic in front of adder's A and B inputs
    - Logic in front is called an arithmetic-logic extender
  - Extender modifies the A and B inputs such that desired operation will appear at output of the adder



Arithmetic-Logic Extender in Front of ALU

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 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



- xyz=000: Want S=A+B – just pass a to ia, b to ib, and set cin=0
- xyz=001: Want S=A-B – pass a to ia, b' to ib, and set cin=1
- xyz=010: Want S=A+1 – pass a to ia, set ib=0, and set cin=1
- xyz=011: Want S=A – pass a to ia, set ib=0, and set cin=0
- xyz=1000: Want S=A AND B – set ia=a*b, b=0, and cin=0
- others: likewise
- Based on above, create logic for ia(x,y,z,a,b) and ib(x,y,z,a,b) for each abext, and create logic for cin(x,y,z), to complete design of the AL-extender component

ALU – abExt Design

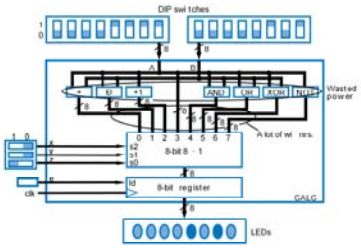
x	y	z	a	b	IA	IB	Cin
0	0	0	0	0			
0	0	0	0	1			
0	0	0	1	0			
0	0	0	1	1			
0	0	1	0	0			
0	0	1	0	1			
0	0	1	1	0			
0	0	1	1	1			
0	1	0	0	0			
0	1	0	0	1			
0	1	0	1	0			
0	1	0	1	1			
0	1	1	0	0			
0	1	1	0	1			
0	1	1	1	0			
0	1	1	1	1			
1	0	0	0	0			
1	0	0	0	1			
1	0	0	1	0			
1	0	0	1	1			
1	0	1	0	0			
1	0	1	0	1			
1	0	1	1	0			
1	0	1	1	1			
1	1	0	0	0			
1	1	0	0	1			
1	1	0	1	0			
1	1	0	1	1			
1	1	1	0	0			
1	1	1	0	1			
1	1	1	1	0			
1	1	1	1	1			

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

51

ALU Example: Multifunction Calculator



- Design using ALU is elegant and efficient
  - No mass of wires
  - No big waste of power