



# Arithmetic Logic Unit Design

Adder  
Subtractor  
Logic Unit



# 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?

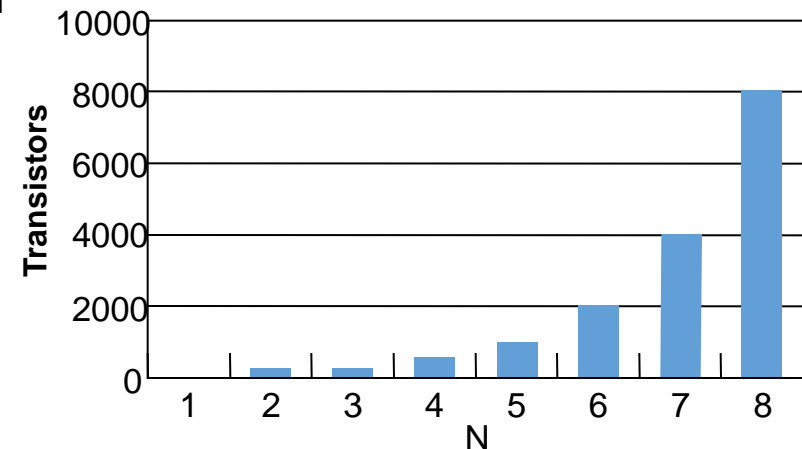
Inputs				Outputs		
a1	a0	b1	b0	c	s1	s0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0



# 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:  $2^{(16+16)} = \sim 4$  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	0
1	1	0	1	0	0	1
1	1	1	0	1	0	0
1	1	1	1	1	0	1



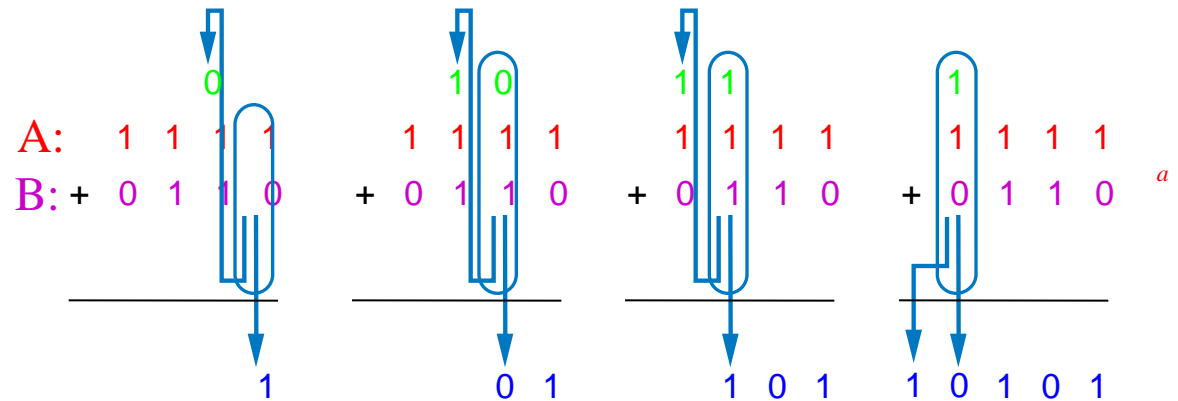
**Q: Predict number of transistors for 16-bit adder**

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



# Half-Adder

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

Step 1: Capture the function

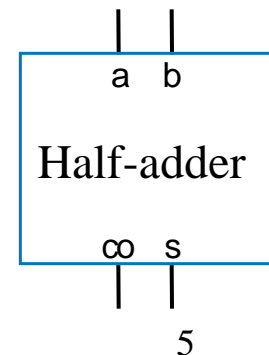
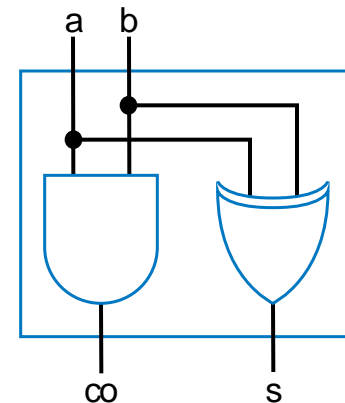
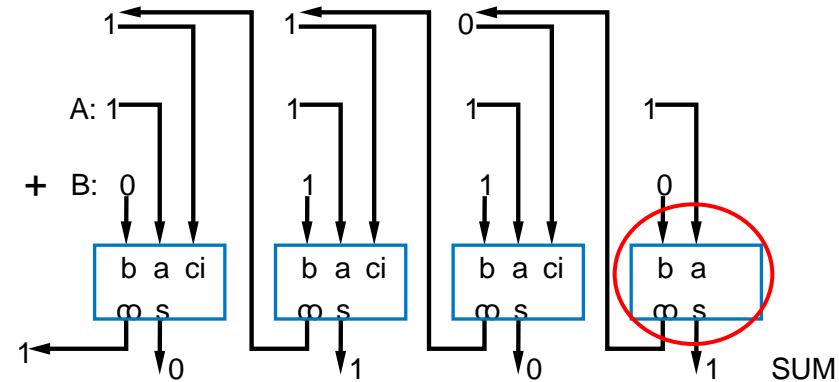
Inputs		Outputs	
a	b	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Step 2: Convert to equations

$$co = ab$$

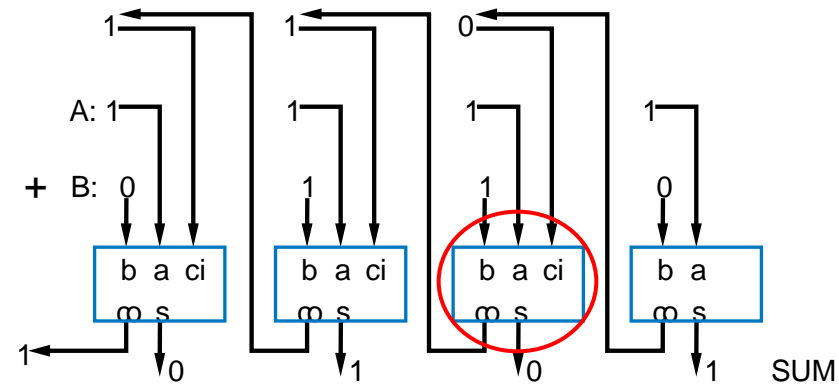
$$s = a'b + ab' \text{ (same as } s = a \text{ xor } b)$$

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

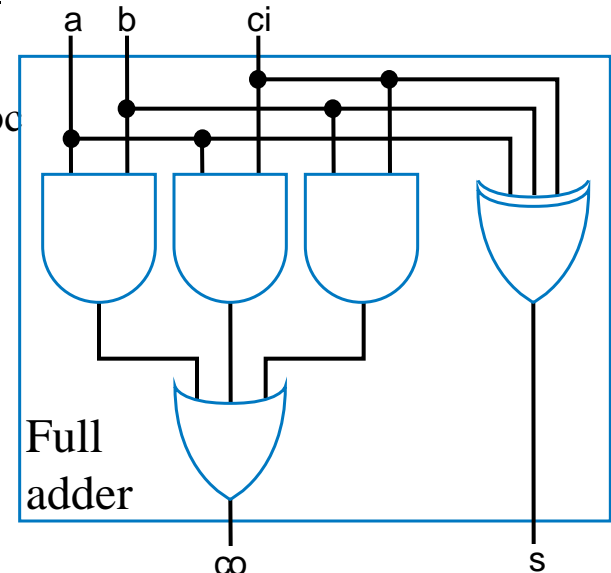
Inputs			Outputs	
a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

## Step 2: Convert to equations

$$\begin{aligned} co &= a'bc + ab'c + abc' + abc \\ co &= a'bc + abc + ab'c + abc + abc' + abc \\ co &= (a' + a)bc + (b' + b)ac + (c' + c)ab \\ co &= bc + ac + ab \end{aligned}$$

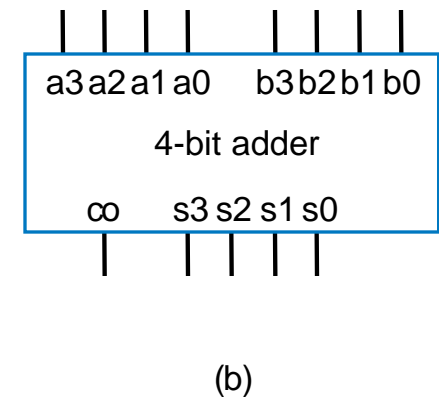
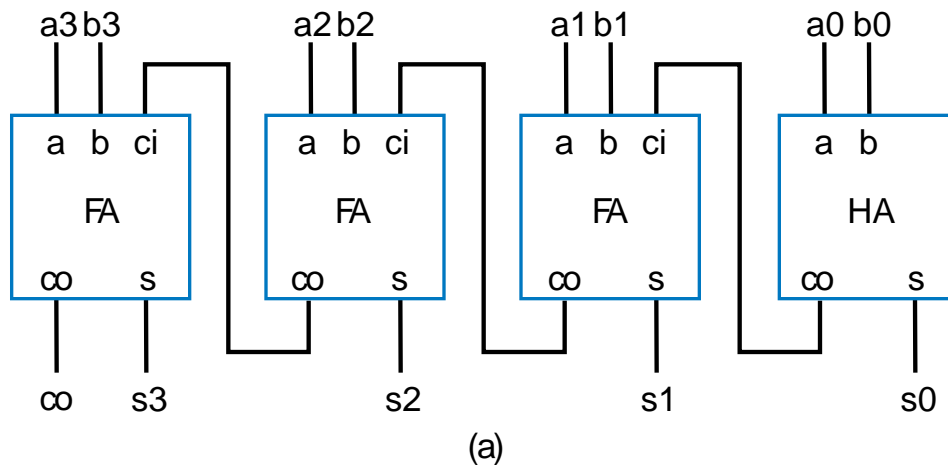
$$\begin{aligned} s &= a'b'c + a'bc' + ab'c' + abc \\ s &= a'(b'c + bc') + a(b'c' + bc) \\ s &= a'(b \text{ xor } c)' + a(b \text{ xor } c) \\ s &= a \text{ xor } b \text{ xor } c \end{aligned}$$

## Step 3: Create the circuit



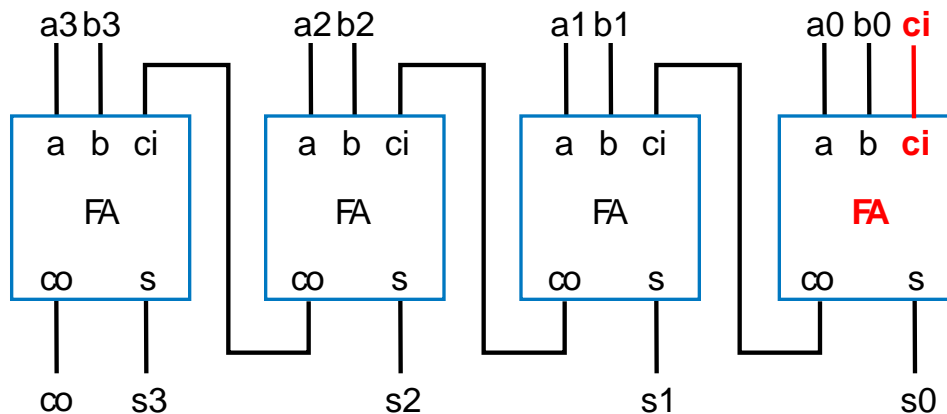
# Carry-Ripple Adder

- 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

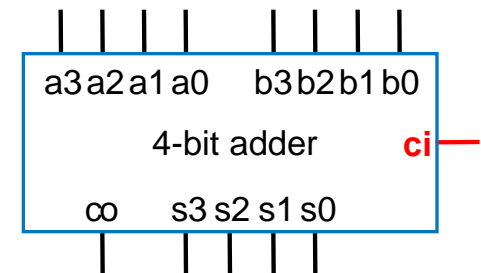


# Carry-Ripple Adder

- 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)

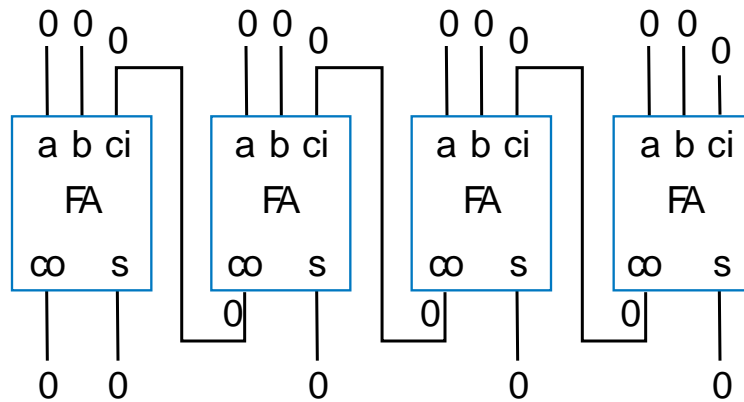


(b)

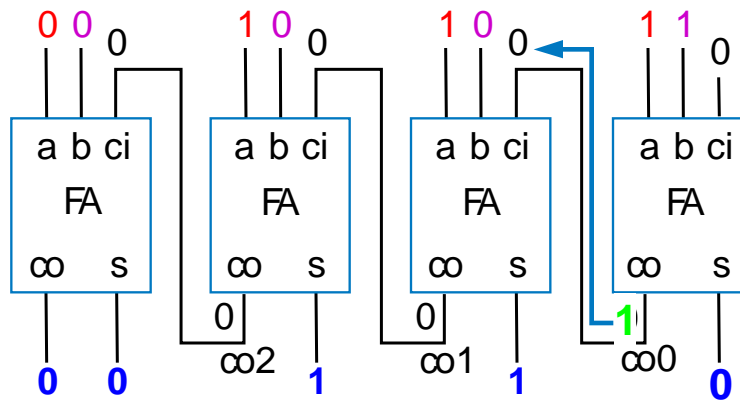




# Carry-Ripple Adder's Behavior



Assume all inputs initially 0



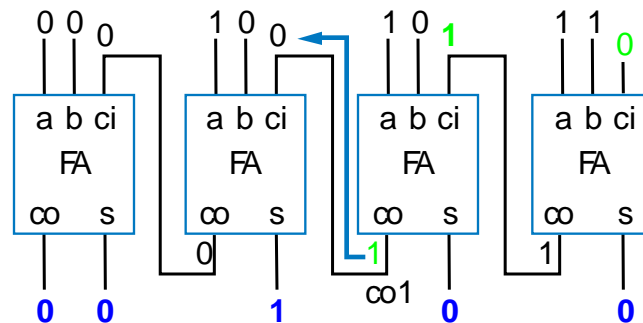
**0111+0001**  
(answer should be 01000)

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.

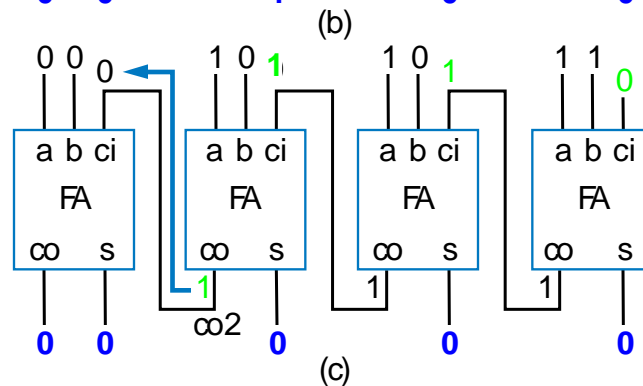


# Carry-Ripple Adder's Behavior

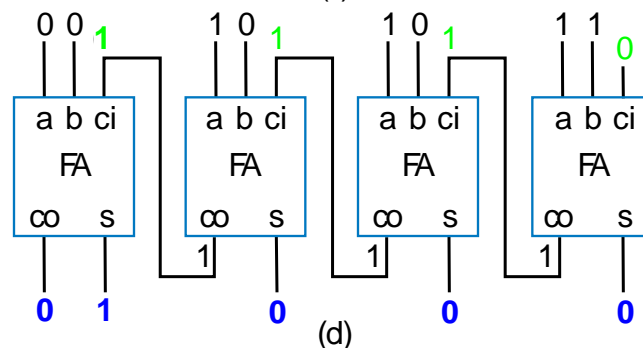


**0111 + 0001**  
(answer should be 01000)

Outputs after 4ns (2 FA delays)



Outputs after 6ns (3 FA delays)

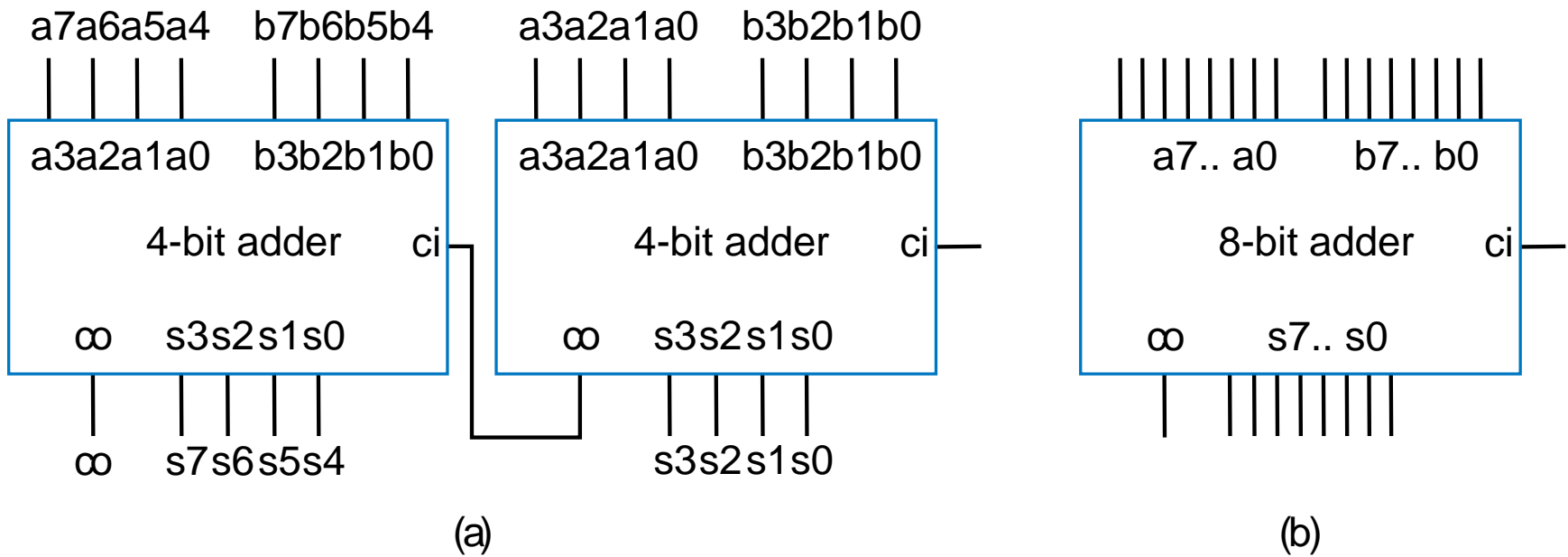


Output after 8ns (4 FA delays)

Correct answer appears after 4 FA delays

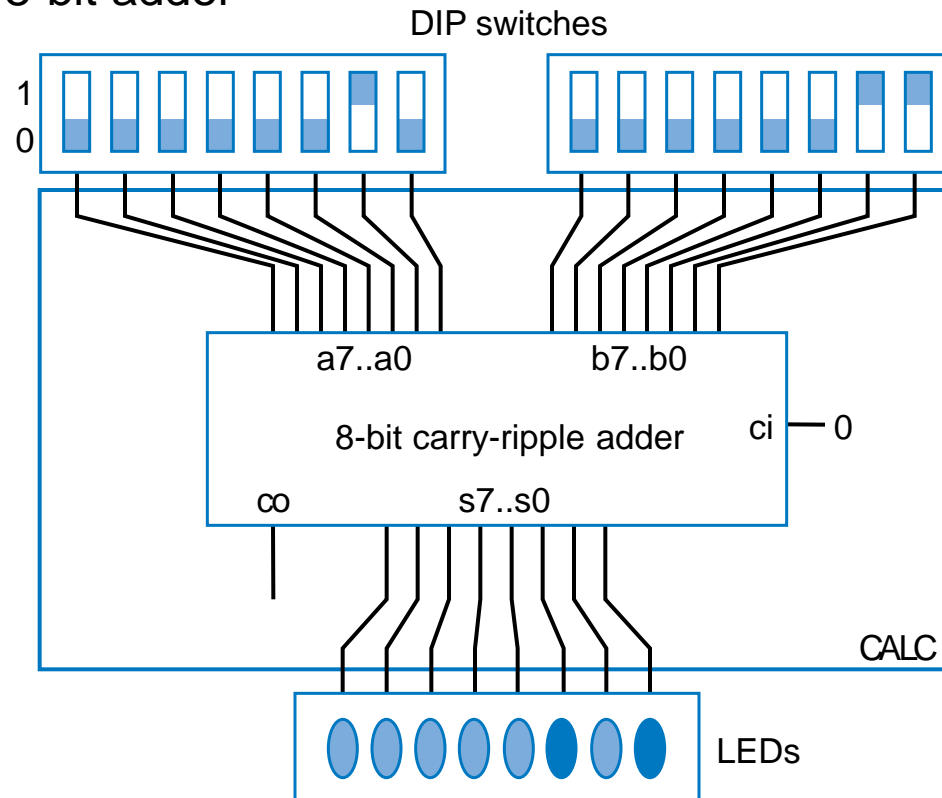


# Cascading Adders



# 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



# Representing Negative Numbers: Two's Complement

- Negative numbers common
  - How do we represent negative numbers in binary?
- Signed-magnitude
  - Use leftmost bit for sign bit
    - So -5 would be:
      - 1101 using four bits
      - 10000101 using eight bits
- Better way: Two's complement
  - Big advantage: Allows us to perform subtraction using addition
  - Thus, only need adder component, no need for separate subtractor component!



# Two's Complement is Easy to Compute: Just Invert Bits and Add 1

- In binary, it turns out that the two's complement can be computed **easily**
  - Two's complement of 011 is 101, because  $011 + 101$  is 1000
  - Could compute complement of 011 as  $1000 - 011 = 101$
  - **Easier method: Just invert all the bits, and add 1**
  - The complement of 011 is  $100+1 = 101$  -- it works!

Q: What is the two's complement of 0101?

A:  $1010+1=1011$

*a*

Q: What is the two's complement of 0011?

(check:  $0101+1011=10000$ )

A:  $1100+1=1101$

Note that the sign bit is 1

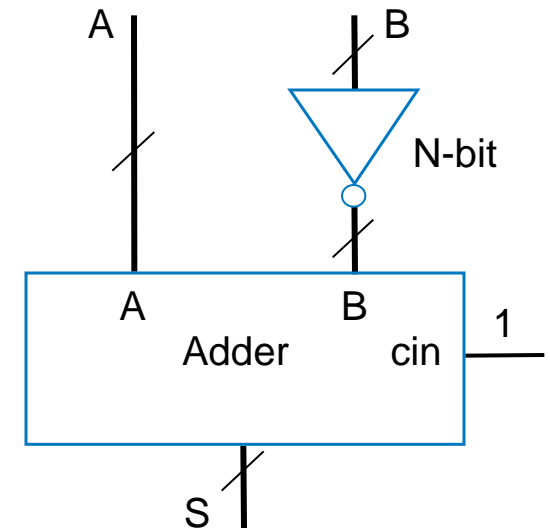


# 4-bit 2's Complement Notation

2's comp notation				decimal
Sign	a2	a1	a0	value
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	0
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

# Two's Complement Subtractor Built with an Adder

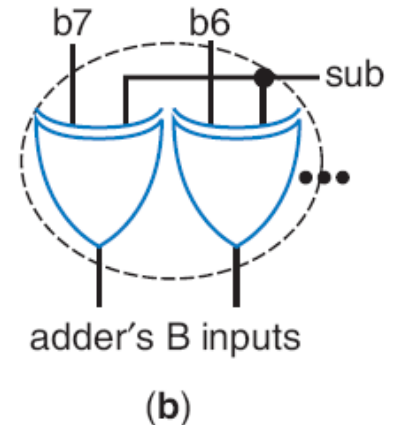
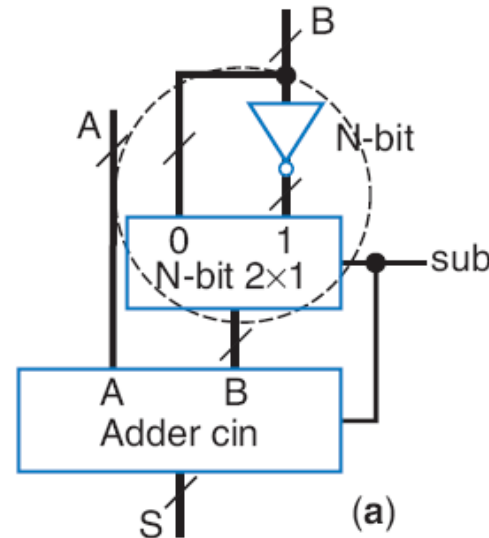
- Using two's complement
$$A - B = A + (-B)$$
$$= A + (\text{two's complement of } B)$$
$$= A + \text{invert\_bits}(B) + 1$$
- So build subtractor using adder by inverting B's bits, and setting carry in to 1





# 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



# 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
  - $\text{overflow} = a_3'b_3's_3 + a_3b_3s_3'$
  - Include "overflow" output bit on adder/subtractor

sign bits			
<div><div>0</div><div>1</div><div>1</div><div>1</div></div> <div><div>+</div><div>0</div><div>0</div><div>0</div><div>1</div></div> <div><div>1</div><div>0</div><div>0</div><div>0</div></div> <div>overflow</div> <div>(a)</div>	<div><div>1</div><div>1</div><div>1</div><div>1</div></div> <div><div>+</div><div>1</div><div>0</div><div>0</div><div>0</div></div> <div><div>0</div><div>1</div><div>1</div><div>1</div></div> <div>overflow</div> <div>(b)</div>	<div><div>1</div><div>0</div><div>0</div><div>0</div></div> <div><div>+</div><div>0</div><div>1</div><div>1</div><div>1</div></div> <div><div>1</div><div>1</div><div>1</div><div>1</div></div> <div>no overflow</div> <div>(c)</div>	

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:  $\text{overflow} = c_3 \text{ xor } c_4$

1 1 1 0 1 1 1	0 0 0 1 1 1 1	0 0 0 1 0 0 0
+ 0 0 0 1	+ 1 0 0 0	+ 0 1 1 1
<hr/>	<hr/>	<hr/>
0 1 0 0 0	1 0 1 1 1	0 1 1 1 1
overflow (a)	overflow (b)	no overflow (c)

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



# 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
- Motivation:
  - Suppose we want multi-function calculator that not only adds and subtracts, but also increments, ANDs, ORs, XORs, etc.

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	$S = A \text{ AND } B$ (bitwise AND)	S=00000101
1	0	1	$S = A \text{ OR } B$ (bitwise OR)	S=00001111
1	1	0	$S = A \text{ XOR } B$ (bitwise XOR)	S=00001010
1	1	1	$S = \text{NOT } A$ (bitwise complement)	S=11110000

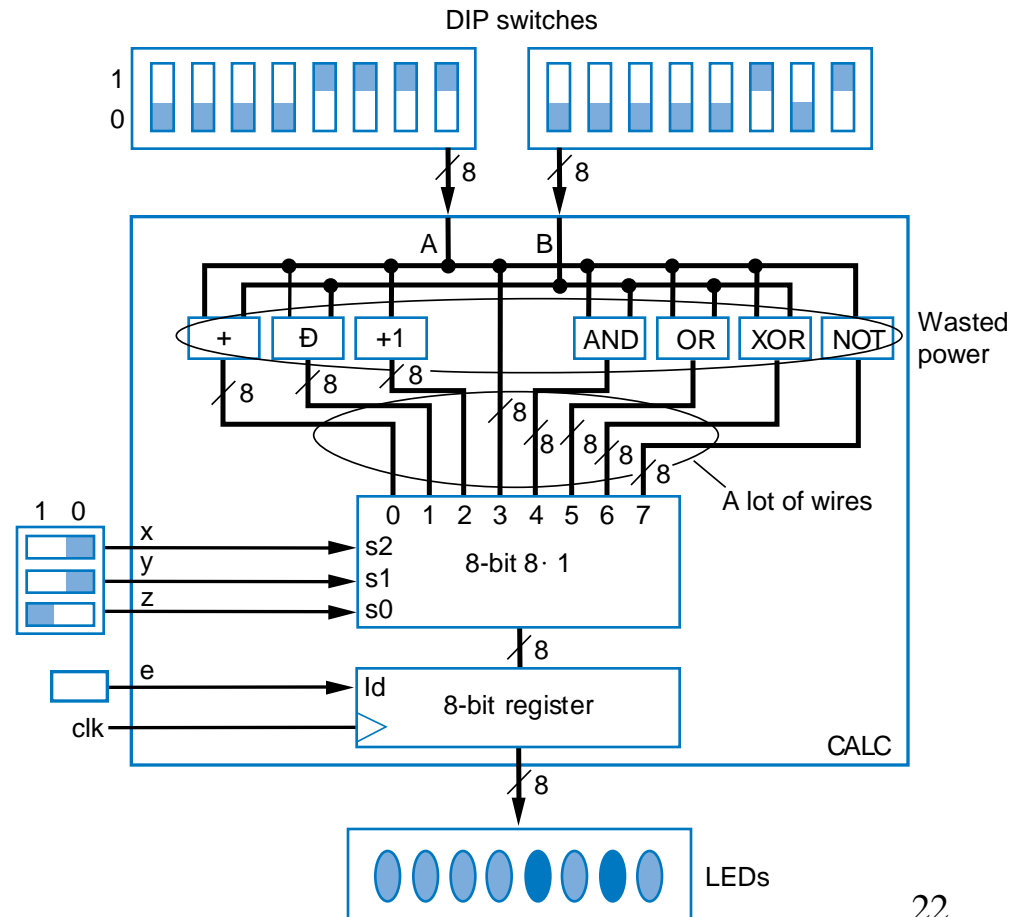


# Multifunction Calculator without an ALU

- 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

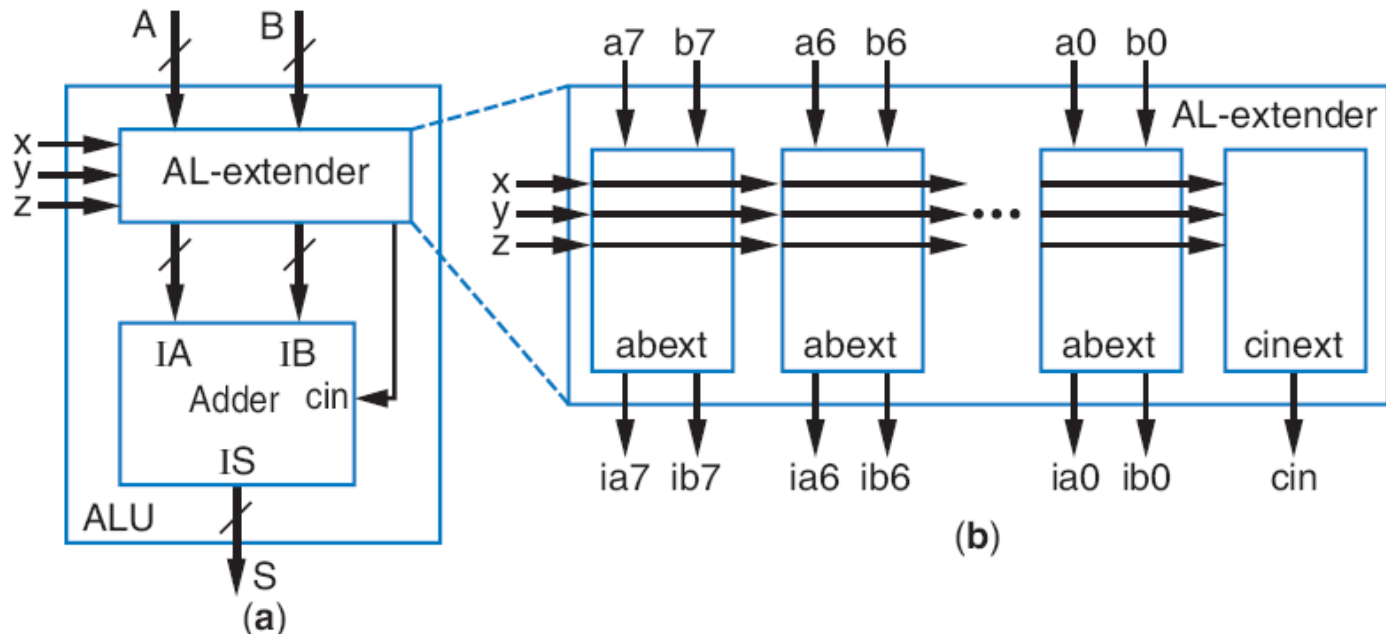
TABLE 4.2 Desired calculator operations

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	$S = A \text{ AND } B$ (bitwise AND)	S=00000101
1	0	1	$S = A \text{ OR } B$ (bitwise OR)	S=00001111
1	1	0	$S = A \text{ XOR } B$ (bitwise XOR)	S=00001010
1	1	1	$S = \text{NOT } A$ (bitwise complement)	S=11110000



# ALU

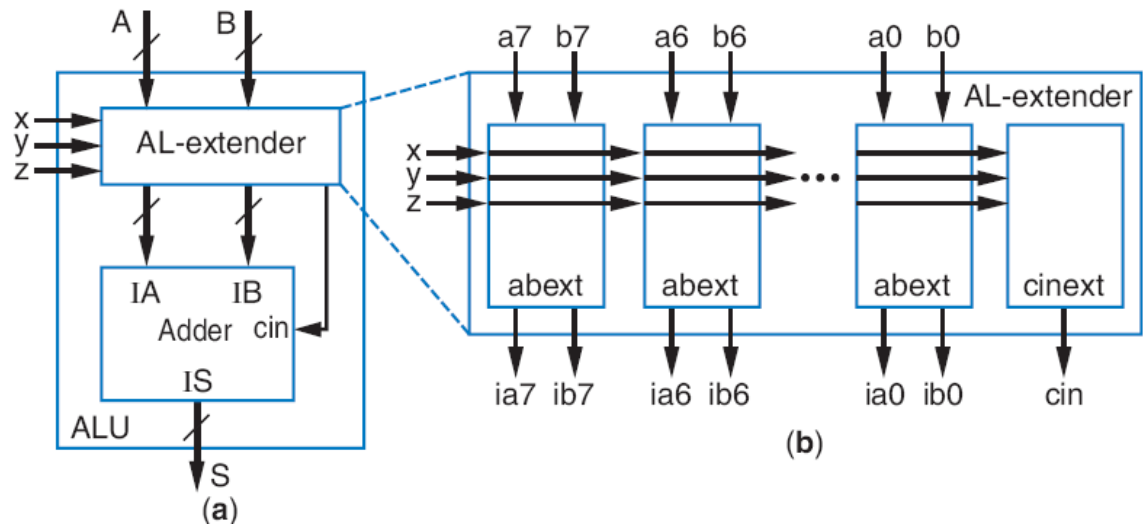
- More efficient design uses ALU
  - ALU design not just separate components multiplexed (same problem as previous slide!),
  - Instead, ALU design uses single adder, plus logic in front of adder's A and B inputs
    - Logic in front is called an arithmetic-logic extender
  - Extender modifies the A 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			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	$S = A \text{ AND } B$ (bitwise AND)	S=00000101
1	0	1	$S = A \text{ OR } B$ (bitwise OR)	S=00001111
1	1	0	$S = A \text{ XOR } B$ (bitwise XOR)	S=00001010
1	1	1	$S = \text{NOT } A$ (bitwise complement)	S=11110000

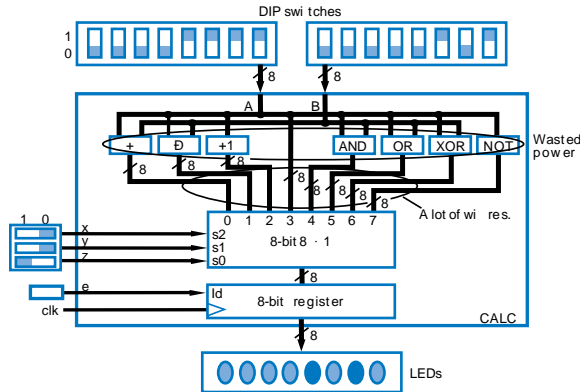


- xyz=000: Want  $S=A+B$  – just pass a to ia, b to ib, and set cin=0
- xyz=001: Want  $S=A-B$  – pass a to ia, b' to ib, and set cin=1
- xyz=010: Want  $S=A+1$  – pass a to ia, set ib=0, and set cin=1
- xyz=011: Want  $S=A$  – pass a to ia, set ib=0, and set cin=0
- xyz=1000: Want  $S=A \text{ AND } B$  – 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 Example: Multifunction Calculator



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

