



# Digital Design

Chapter 4:  
Datapath Components

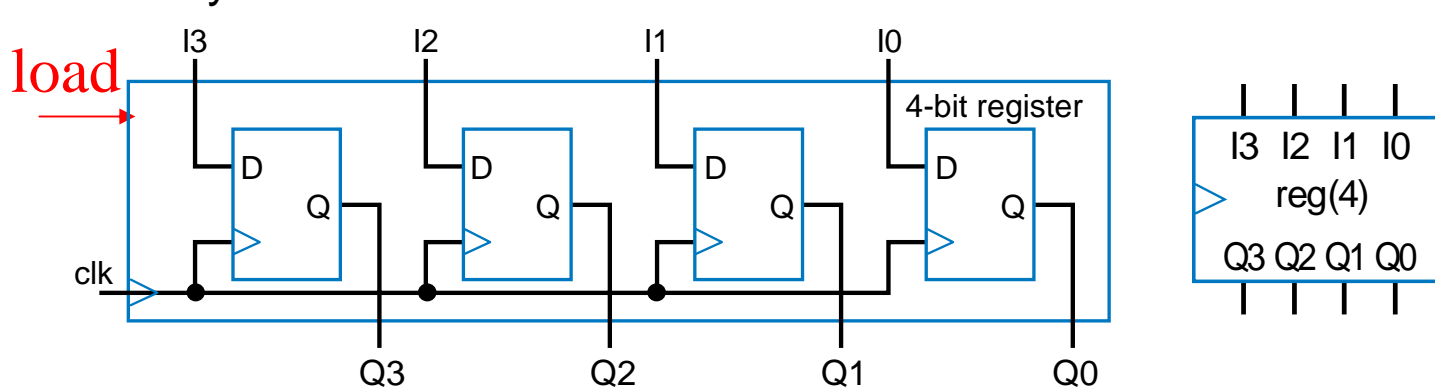
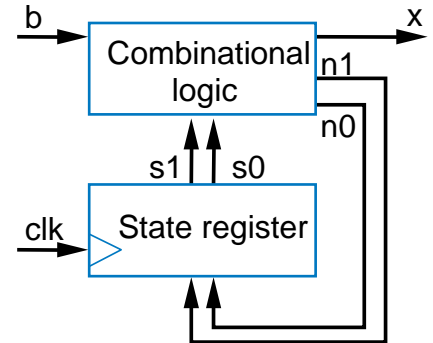
# Introduction

- Chapters 2 & 3: Introduced increasingly complex digital building blocks
  - Gates, multiplexors, decoders, basic registers, and controllers
- Controllers good for systems with control inputs/outputs
  - **Control** input: Single bit (or just a few), representing environment event or state
    - e.g., 1 bit representing button pressed
  - **Data** input: Multiple bits collectively representing single entity
    - e.g., 7 bits representing temperature in binary
- Need building blocks for data
  - **Datapath components**, aka register-transfer-level (RTL) components, store/transform data
    - Put datapath components together to form a **datapath**
- This chapter introduces numerous datapath components, and simple datapaths
  - Next chapter will combine controllers and datapaths into “processors”



# Registers

- Can store data, very common in datapaths
- Basic register of Ch 3: Loaded every cycle
  - Useful for implementing FSM -- stores encoded state
  - For other uses, may want to load only on certain cycles



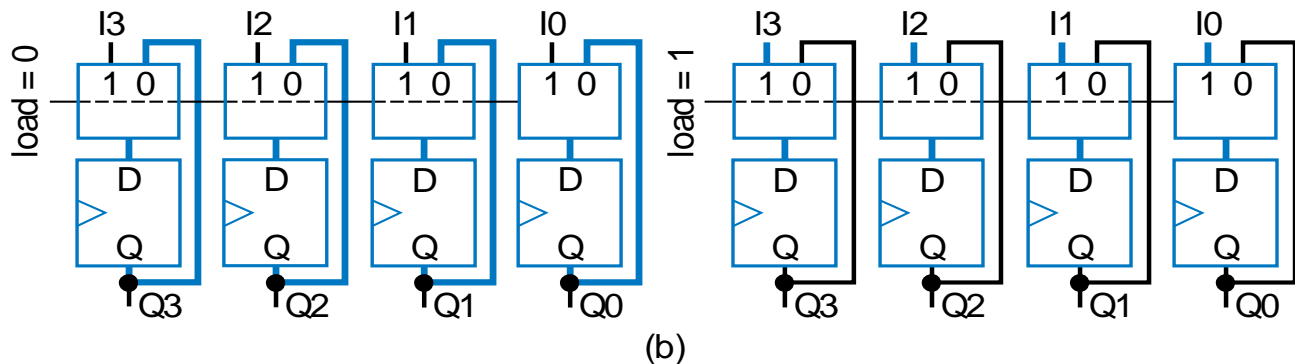
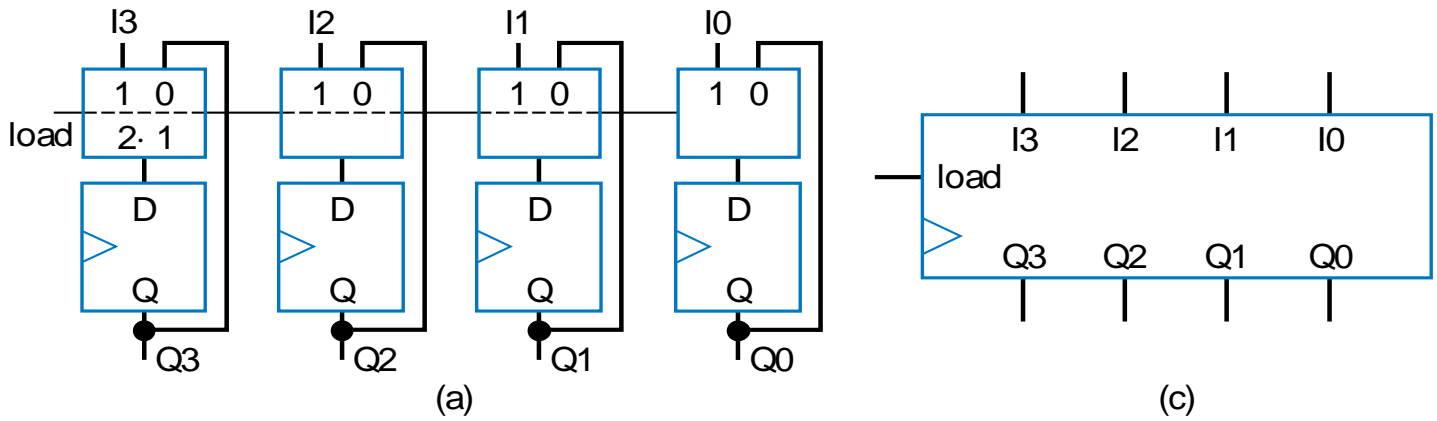
*Basic register loads on every clock cycle*

*How extend to only load on certain cycles?*



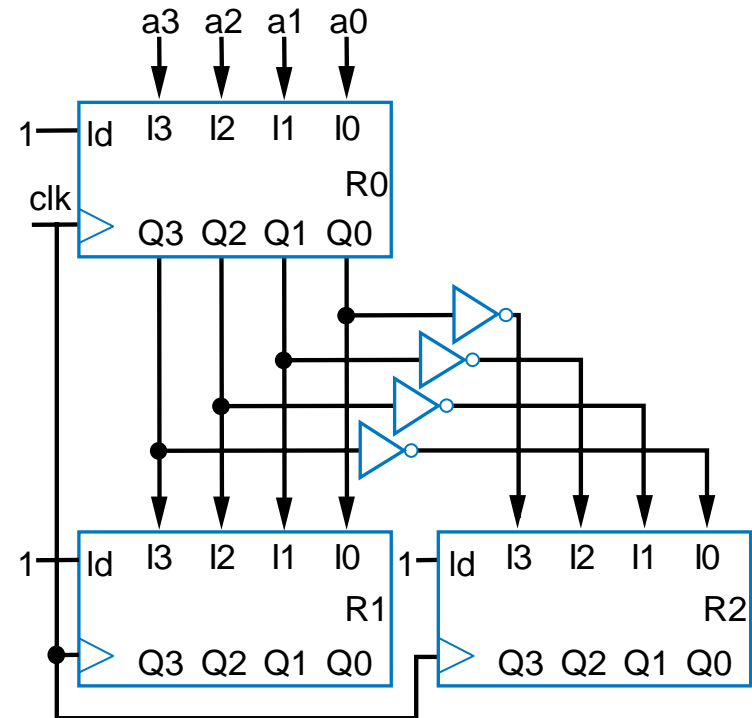
# Register with Parallel Load

- Add 2x1 mux to front of each flip-flop
- Register's load input selects mux input to pass
  - Either existing flip-flop value, or new value to load

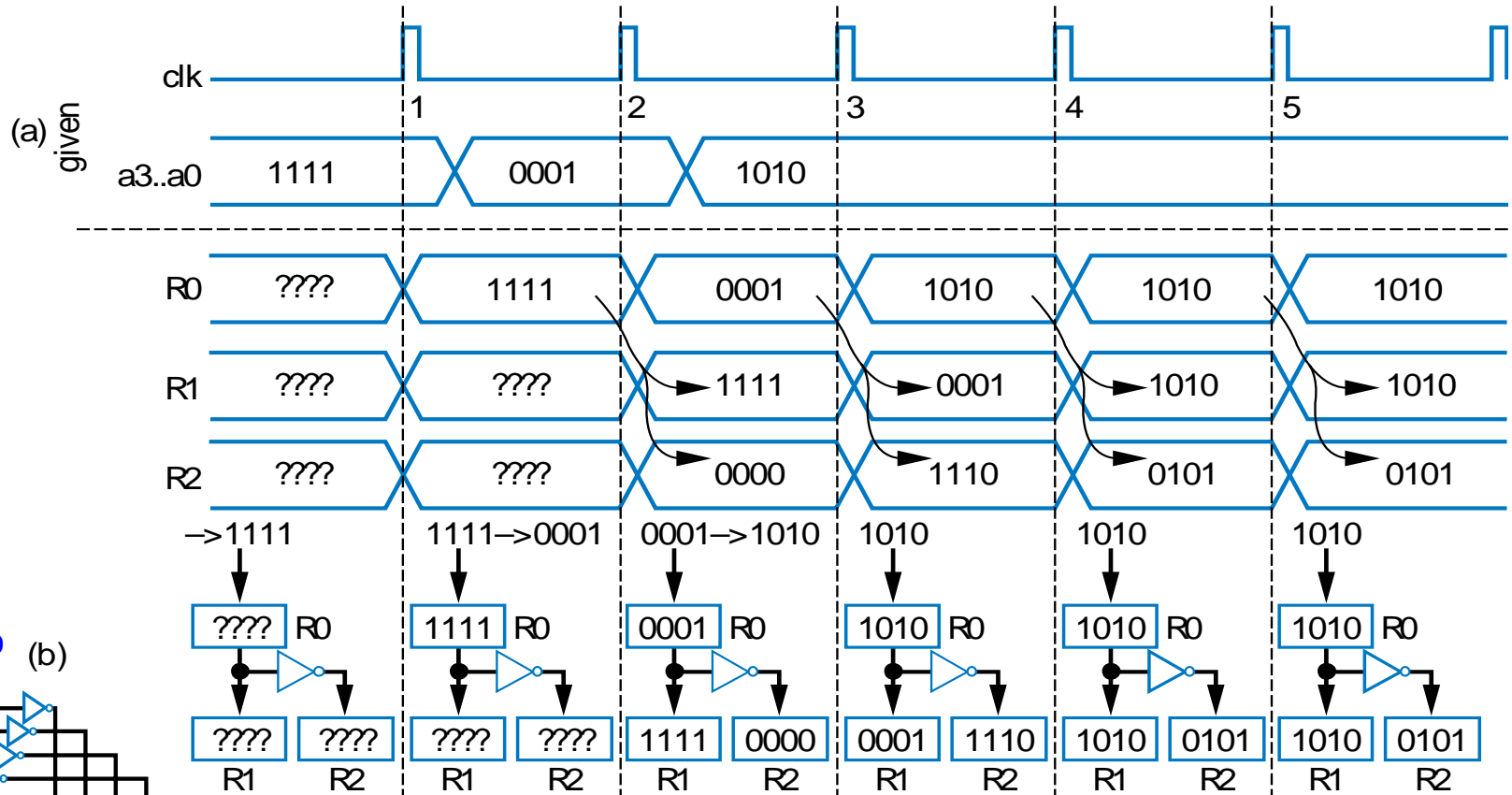


# Basic Example Using Registers

- This example will show how registers load simultaneously on clock cycles
  - Notice that all load inputs set to 1 in this example -- just for demonstration purposes

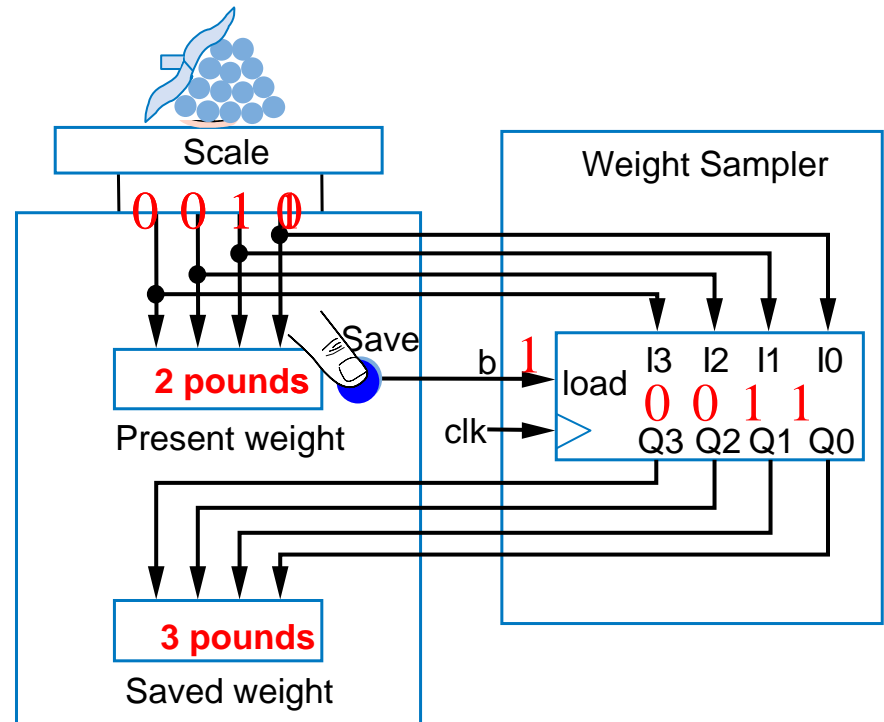


# Basic Example Using Registers



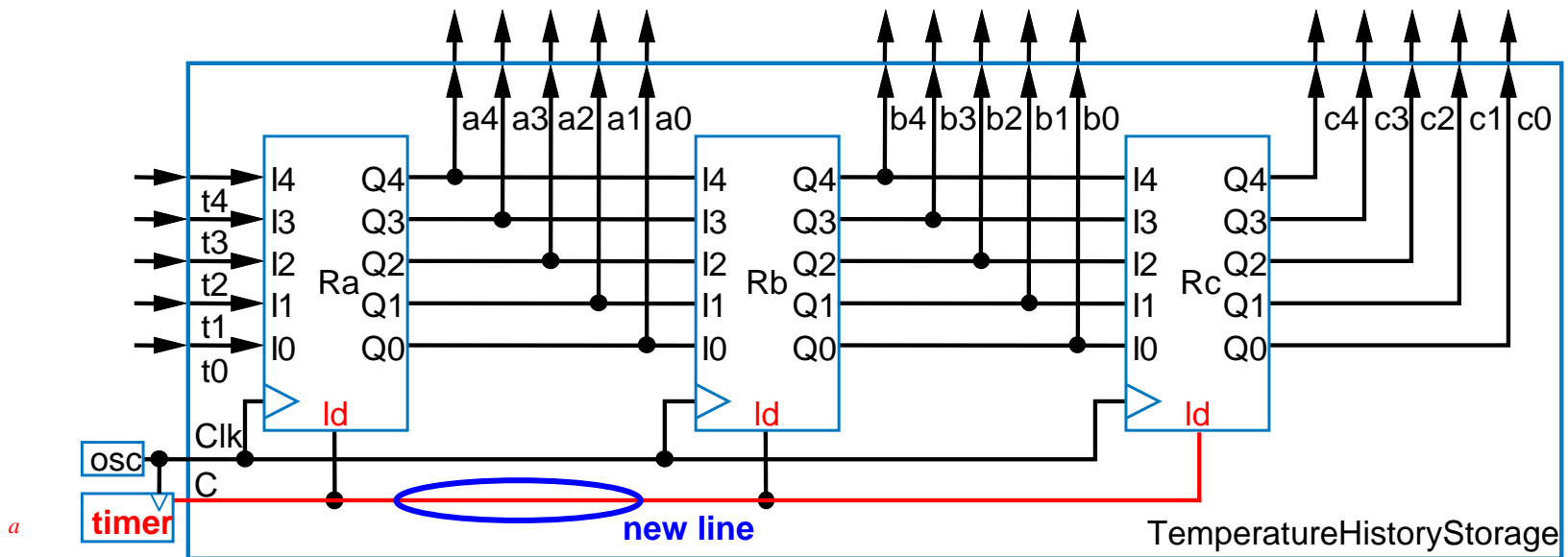
# Register Example using the Load Input: Weight Sampler

- Scale has two displays
  - Present weight
  - Saved weight
  - Useful to compare present item with previous item
- Use register to store weight
  - Pressing button causes present weight to be stored in register
    - Register contents always displayed as “Saved weight,” even when new present weight appears



# Register Example: Temperature History Display

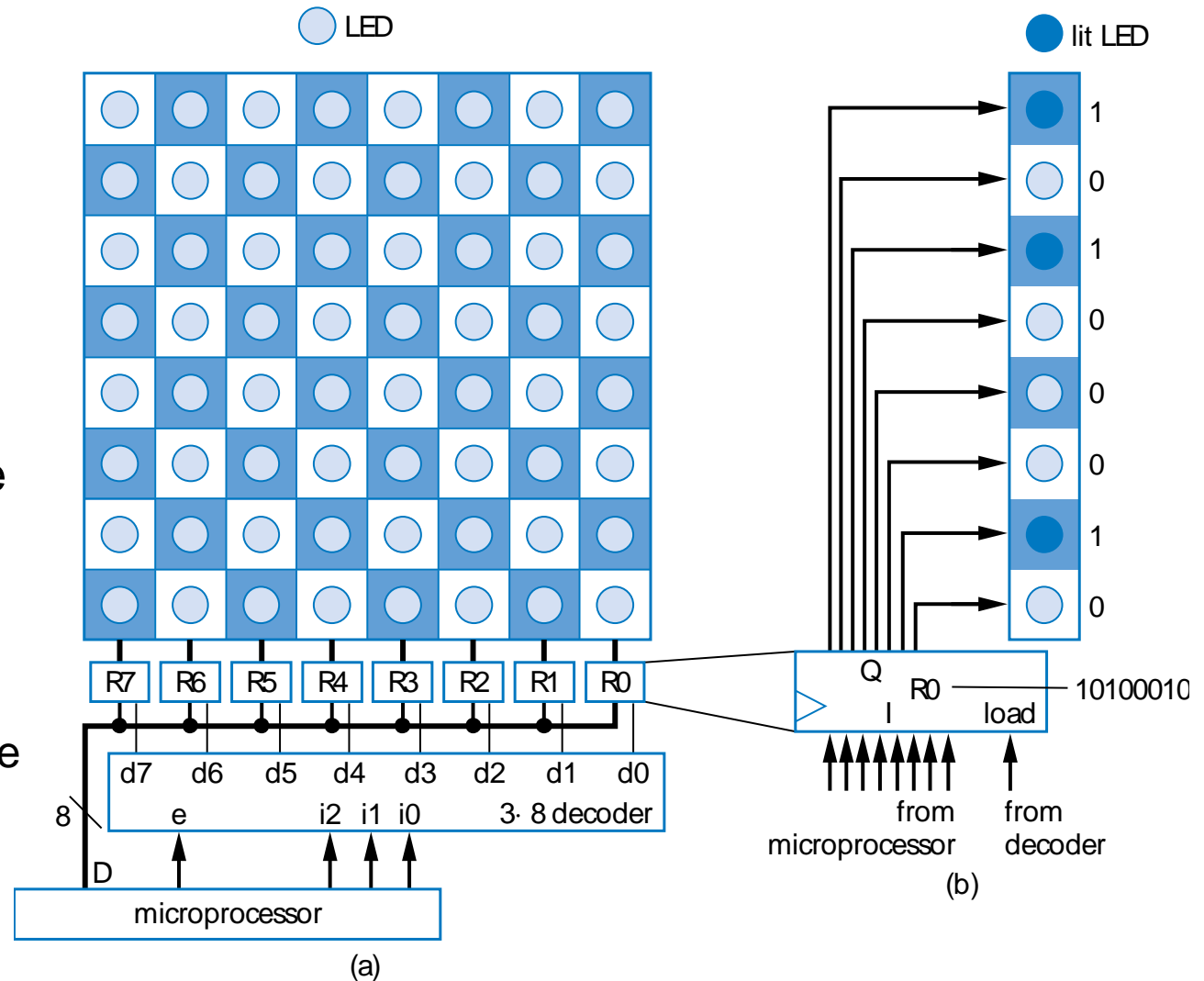
- Recall Chpt 3 example
  - Timer pulse every hour
  - Previously used as clock. Better design only connects oscillator to clock inputs -- use registers with load input, connect to timer pulse.



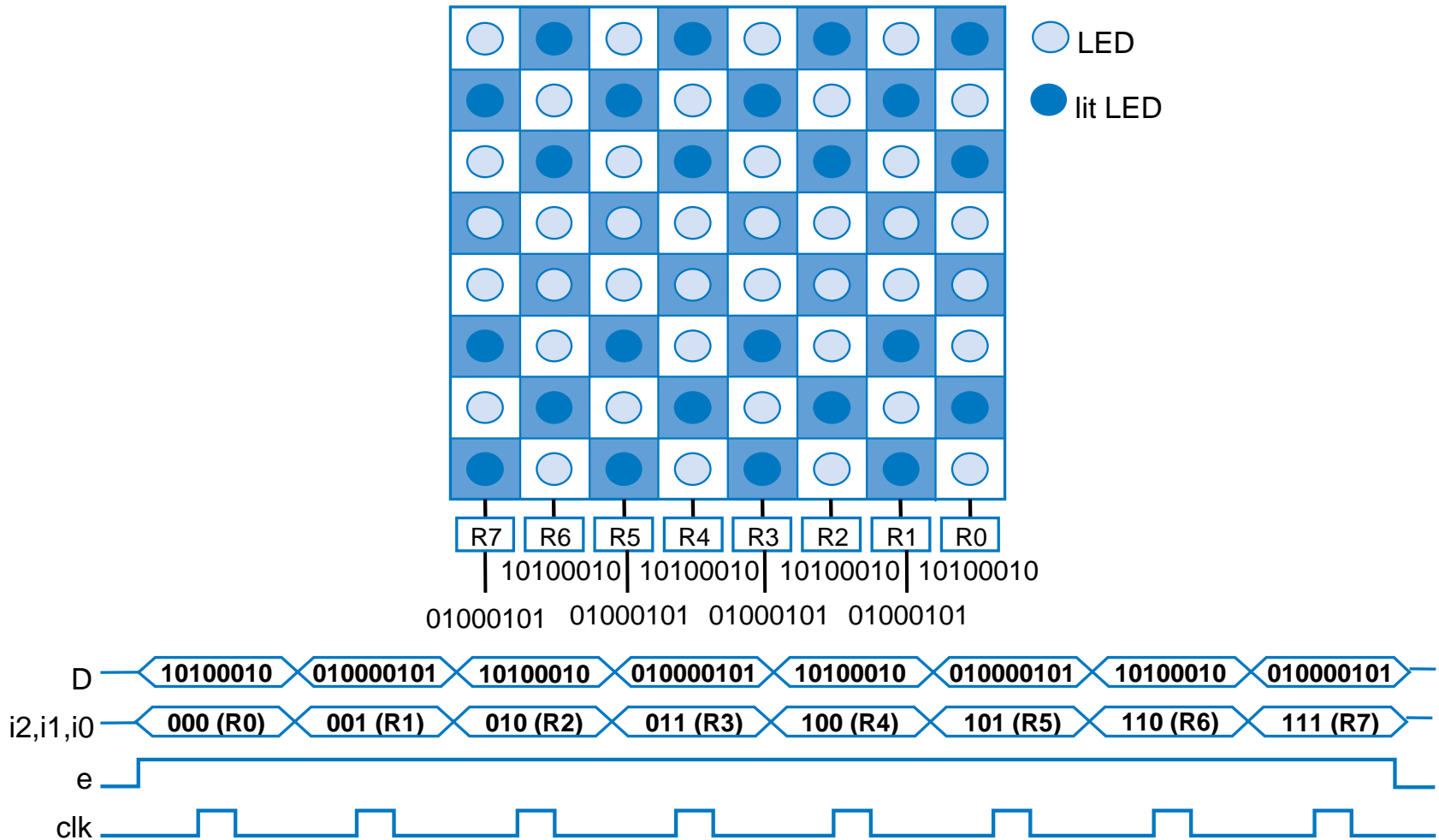


# Register Example: Computerized Checkerboard

- Each register holds values for one column of lights
  - 1 lights light
- Microprocessor loads one register at a time
  - Occurs fast enough that user sees entire board change at once

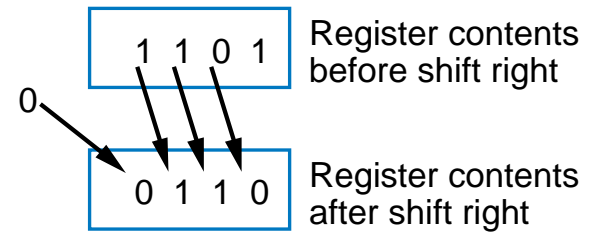


# Register Example: Computerized Checkerboard



# Shift Register

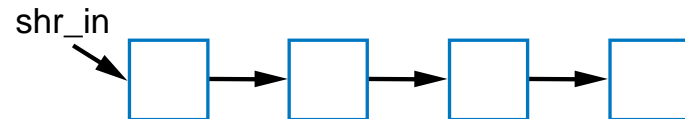
- Shift right
  - Move each bit one position right
  - Shift in 0 to leftmost bit



**Q: Do four right shifts on 1001, showing value after each shift**

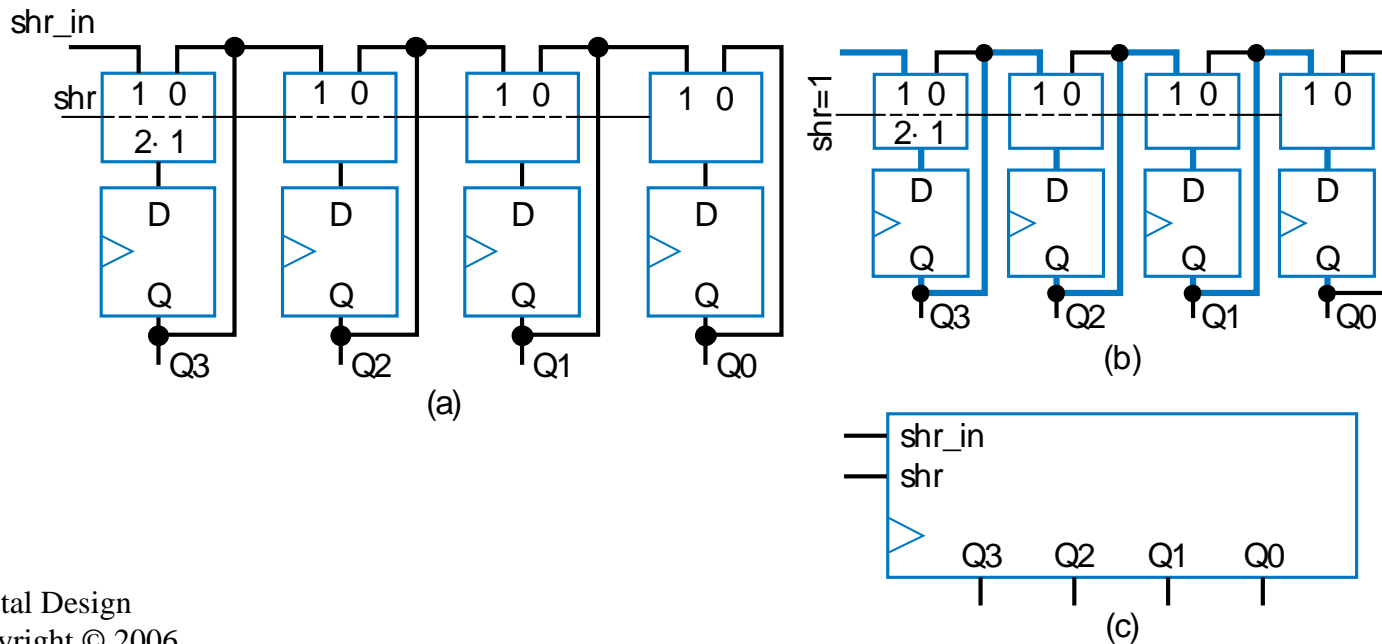
A: 1001 (original)  
0100  
0010  
0001  
0000

- Implementation: Connect flip-flop output to next flip-flop's input



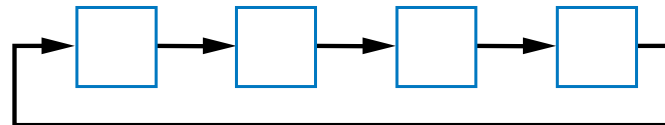
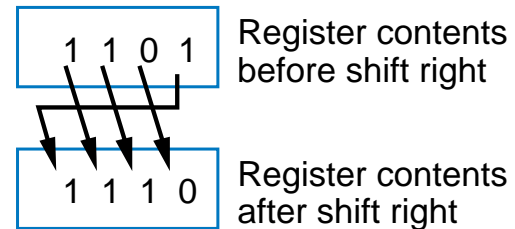
# Shift Register

- To allow register to either shift or retain, use 2x1 muxes
  - shr: 0 means retain, 1 shift
  - shr\_in: value to shift in
    - May be 0, or 1
- Note: Can easily design shift register that shifts left instead



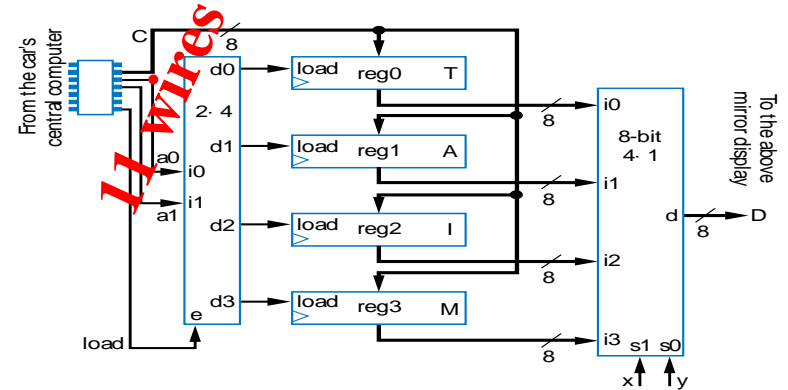
# Rotate Register

- Rotate right: Like shift right, but leftmost bit comes from rightmost bit

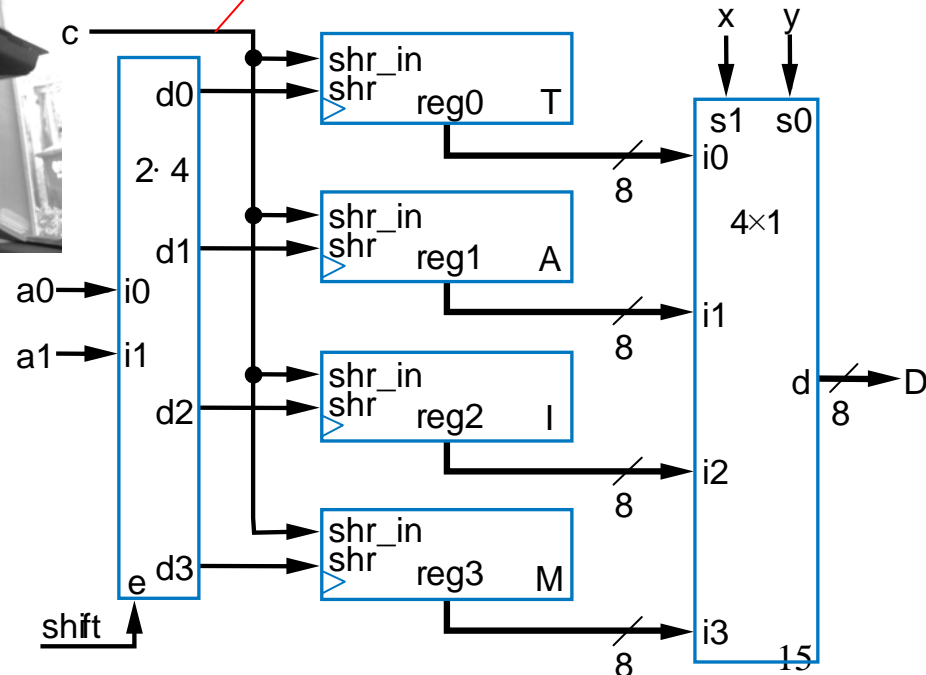


# Shift Register Example: Above-Mirror Display

- Earlier example:  $8 + 2 + 1 = 11$  wires from car's computer to above-mirror display's four registers
  - Better than 32 wires, but 11 still a lot -- want fewer for smaller wire bundles
- Use shift registers
  - Wires:  $1 + 2 + 1 = 4$
  - Computer sends one value at a time, one bit per clock cycle



Note: this line is 1 bit, rather than 8 bits like before

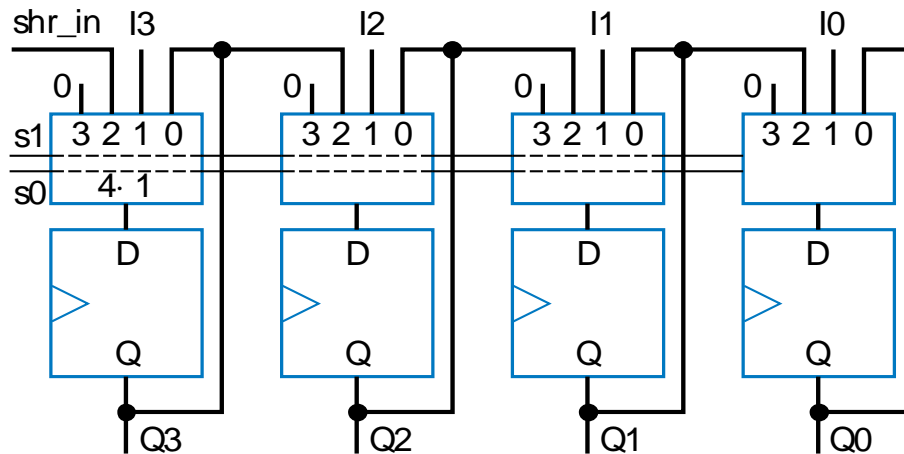


# Multifunction Registers

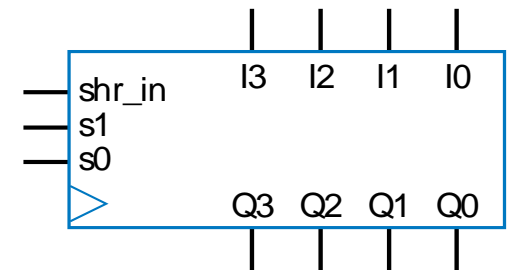
- Many registers have multiple functions
  - Load, shift, clear (load all 0s)
  - And retain present value, of course
- Easily designed using muxes
  - Just connect each mux input to achieve desired function

## Functions:

s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	(unused - let's load 0s)



(a)

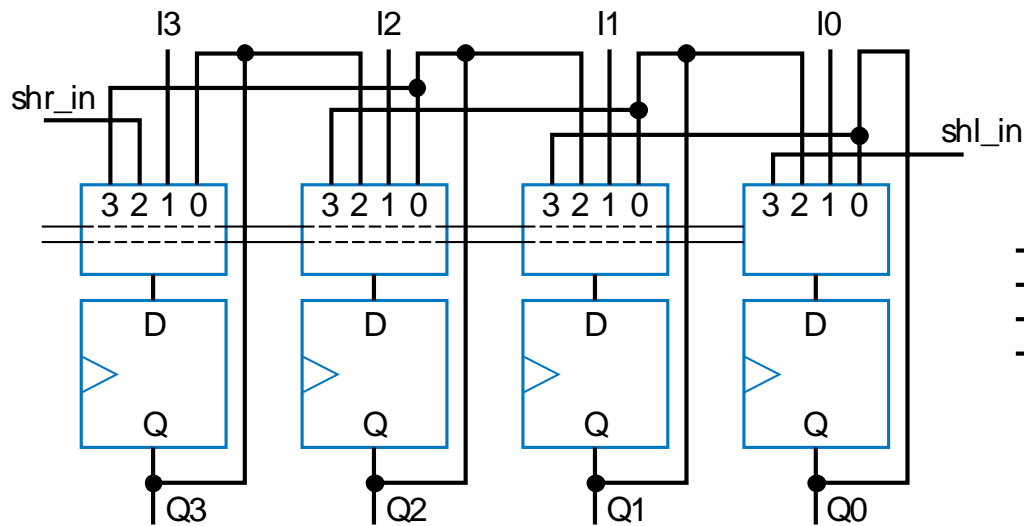


(b)

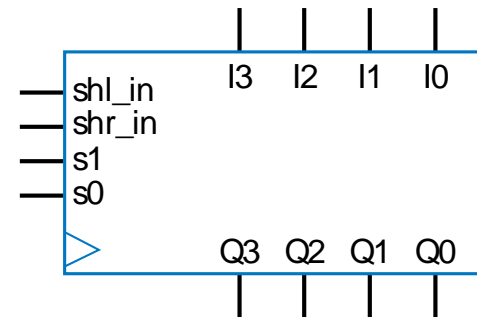


# Multifunction Registers

s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	Shift left



(a)



(b)

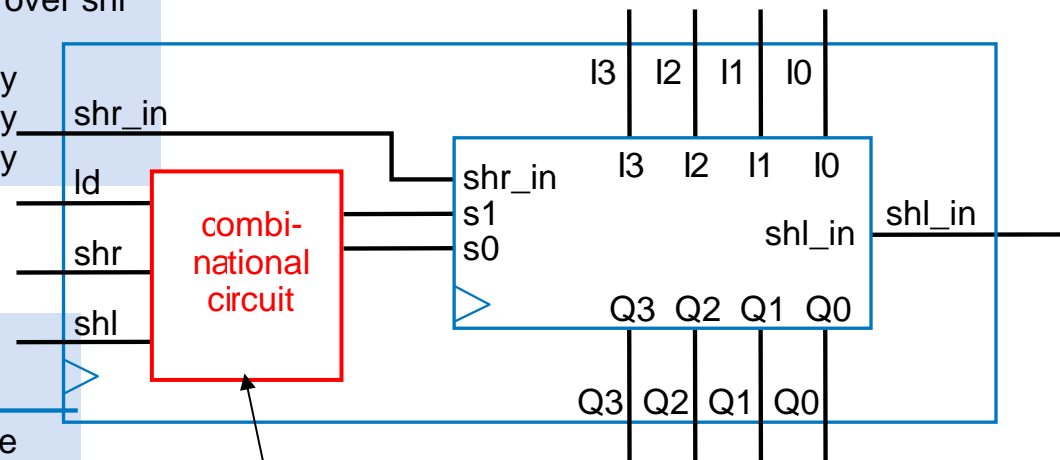


# Multifunction Registers with Separate Control Inputs

ld	shr	shl	Operation
0	0	0	Maintain present value
0	0	1	Shift left
0	1	0	Shift right
0	1	1	Shift right – shr has priority over shl
1	0	0	Parallel load
1	0	1	Parallel load – ld has priority
1	1	0	Parallel load – ld has priority
1	1	1	Parallel load – ld has priority

Truth table for combinational circuit

Inputs			Outputs		Note Operation
ld	shr	shl	s1	s0	
0	0	0	0	0	Maintain value
0	0	1	1	1	Shift left
0	1	0	1	0	Shift right
0	1	1	1	0	Shift right
1	0	0	0	1	Parallel load
1	0	1	0	1	Parallel load
1	1	0	0	1	Parallel load
1	1	1	0	1	Parallel load



$$s1 = ld' * shr' * shl + ld' * shr * shl' + ld' * shr * shl$$

$$s0 = ld' * shr' * shl + ld$$



# Register Operation Table

- Register operations typically shown using compact version of table
  - X means same operation whether value is 0 or 1
    - One X expands to two rows
    - Two Xs expand to four rows
  - Put highest priority control input on left to make reduced table simple

Inputs			Outputs		Note Operation
ld	shr	shl	s1	s0	
0	0	0	0	0	Maintain value
0	0	1	1	1	Shift left
0	1	0	1	0	Shift right
0	1	1	1	0	Shift right
1	0	0	0	1	Parallel load
1	0	1	0	1	Parallel load
1	1	0	0	1	Parallel load
1	1	1	0	1	Parallel load

ld	shr	shl	Operation
0	0	0	Maintain value
0	0	1	Shft left
0	1	X	Shift right
1	X	X	Parallel load



# Register Design Process

- Can design register with desired operations using simple four-step process

**TABLE 4.1** Four-step process for designing a multifunction register.

	<b>Step</b>	<b>Description</b>
1.	<i>Determine mux size</i>	Count the number of operations (don't forget the maintain present value operation!) and add in front of each flip-flop a mux with at least that number of inputs.
2.	<i>Create mux operation table</i>	Create an operation table defining the desired operation for each possible value of the mux select lines.
3.	<i>Connect mux inputs</i>	For each operation, connect the corresponding mux data input to the appropriate external input or flip-flop output (possibly passing through some logic) to achieve the desired operation.
4.	<i>Map control lines</i>	Create a truth table that maps external control lines to the internal mux select lines, with appropriate priorities, and then design the logic to achieve that mapping



# Register Design Example

- Desired register operations
  - Load, shift left, synchronous clear, synchronous set

## Step 1: Determine mux size

5 operations: above, plus maintain present value (don't forget this one!)

--> Use 8x1 mux

## Step 2: Create mux operation table

## Step 3: Connect mux inputs

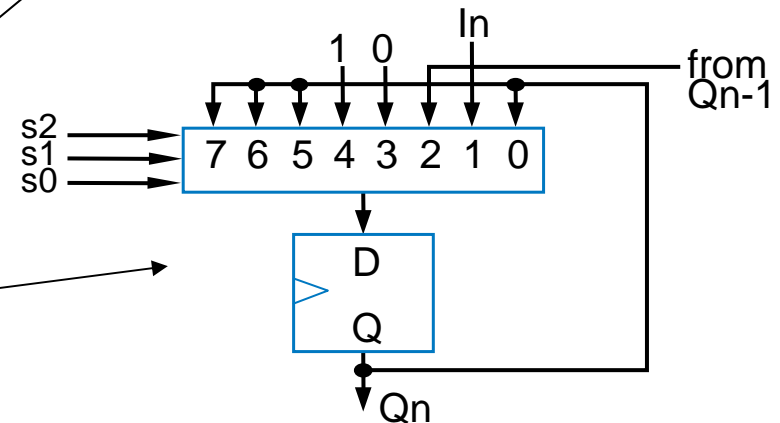
## Step 4: Map control lines

$$s2 = \text{clr}' * \text{set}$$

$$s1 = \text{clr}' * \text{set}' * \text{ld}' * \text{shl} + \text{clr}$$

$$s0 = \text{clr}' * \text{set}' * \text{ld} + \text{clr}$$

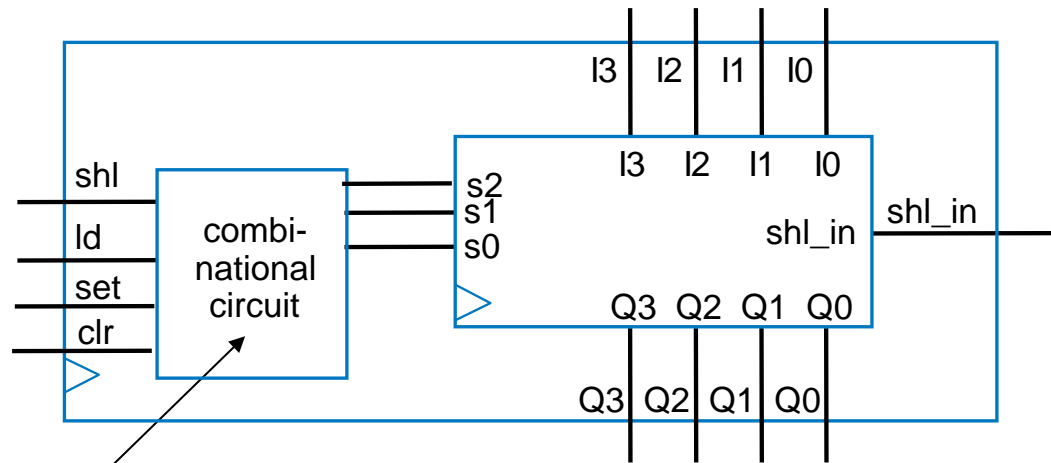
s2	s1	s0	Operation
0	0	0	Maintain present value
0	0	1	Parallel load
0	1	0	Shift left
0	1	1	Synchronous clear
1	0	0	Synchronous set
1	0	1	Maintain present value
1	1	0	Maintain present value
1	1	1	Maintain present value



Inputs				Outputs			Operation
clr	set	ld	shl	s2	s1	s0	
0	0	0	0	0	0	0	Maintain present value
0	0	0	1	0	1	0	Shift left
0	0	1	X	0	0	1	Parallel load
0	1	X	X	1	0	0	Set to all 1s
1	X	X	X	0	1	1	Clear to all 0s



# Register Design Example



## Step 4: Map control lines

$$s2 = \text{clr}' * \text{set}$$

$$s1 = \text{clr}' * \text{set}' * \text{ld}' * \text{shl} + \text{clr}$$

$$s0 = \text{clr}' * \text{set}' * \text{ld} + \text{clr}$$

Inputs				Outputs			Operation
clr	set	ld	shl	s2	s1	s0	
0	0	0	0	0	0	0	Maintain present value
0	0	0	1	0	1	0	Shift left
0	0	1	X	0	0	1	Parallel load
0	1	X	X	1	0	0	Set to all 1s
1	X	X	X	0	1	1	Clear to all 0s



# 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?

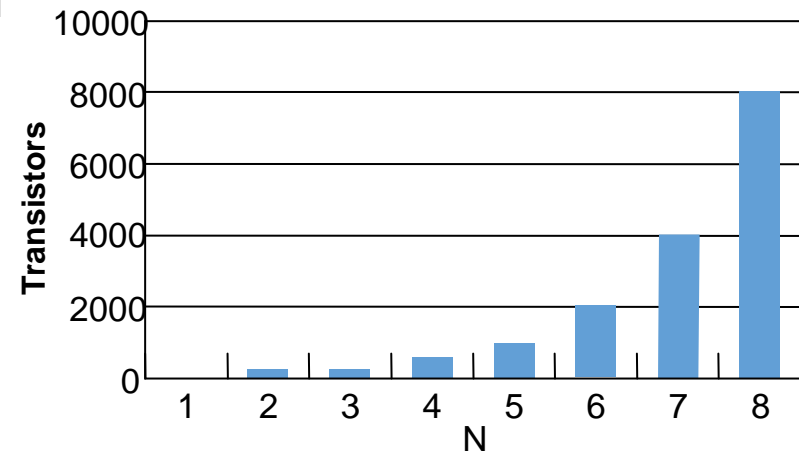
Inputs				Outputs		
a1	a0	b1	b0	c	s1	s0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0



# 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	0	1
1	0	1	0	1	1	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



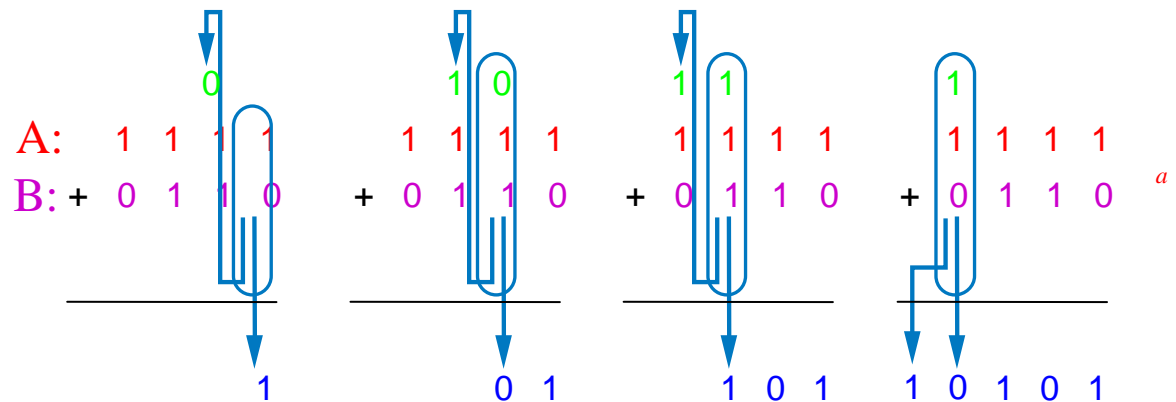
**Q: Predict number of transistors for 16-bit adder**

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



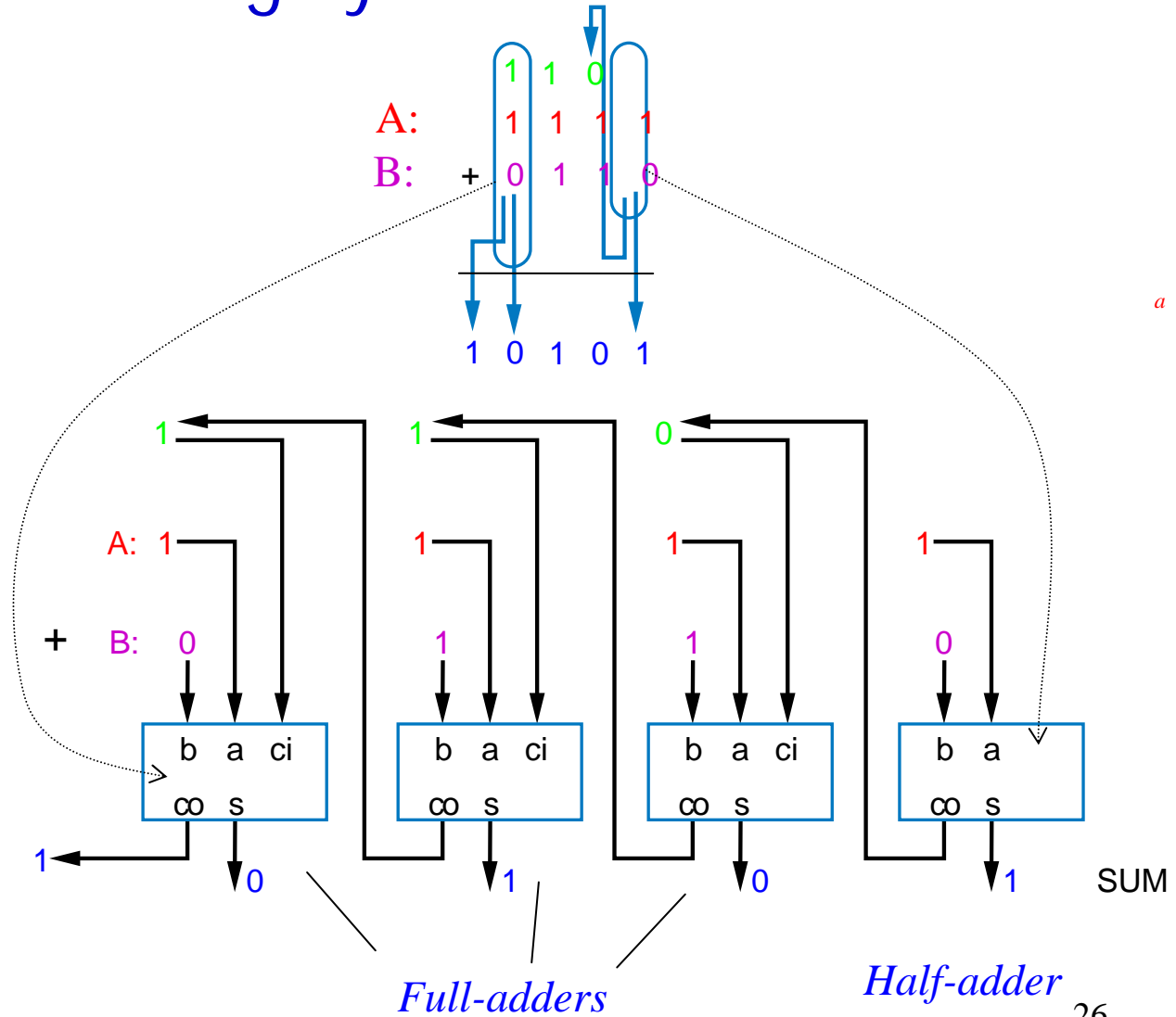
# Alternative Method to Design an Adder: Imitate Adding by Hand

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



# 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

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

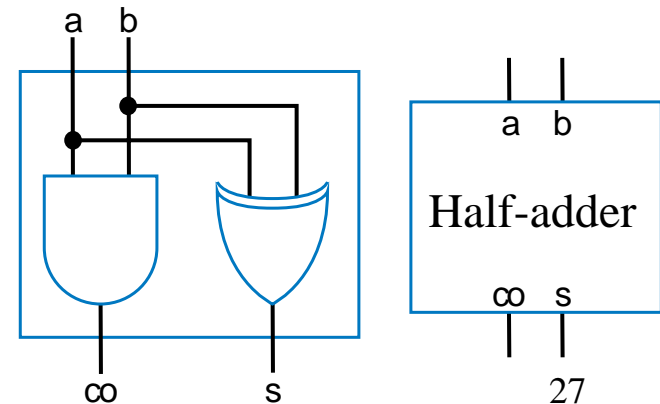
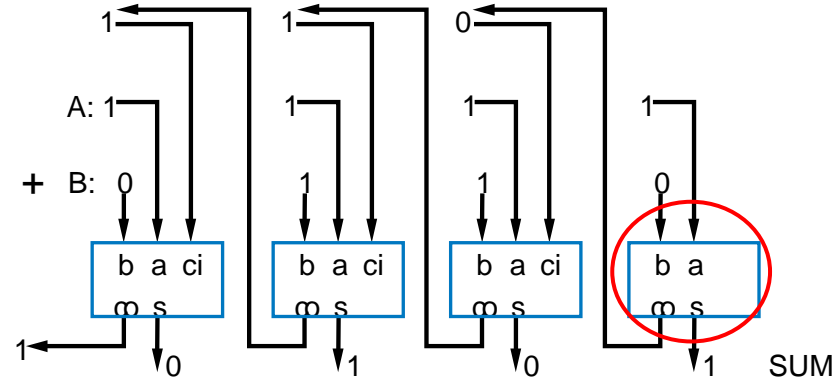
Step 1: Capture the function

Step 2: Convert to equations

$$co = ab$$

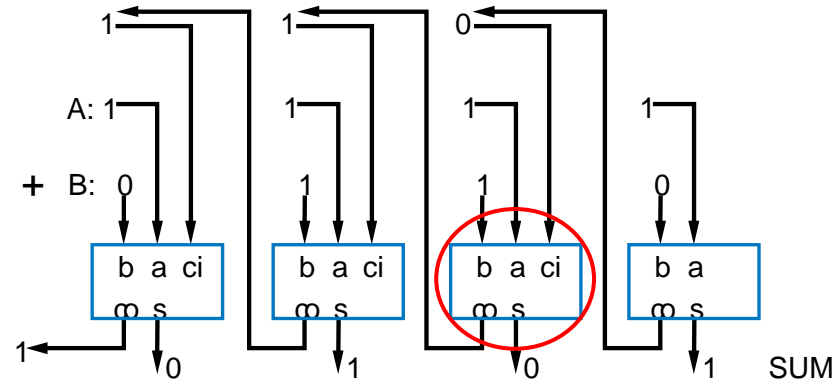
$$s = a'b + ab' \text{ (same as } s = a \text{ xor } b)$$

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

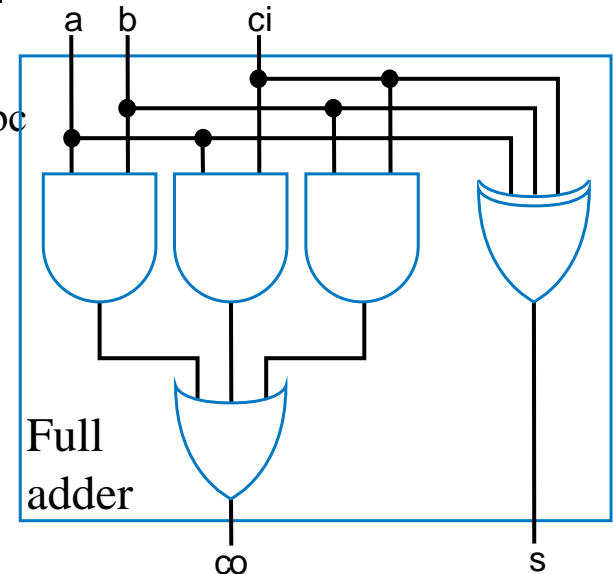
Inputs			Outputs	
a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

## Step 2: Convert to equations

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

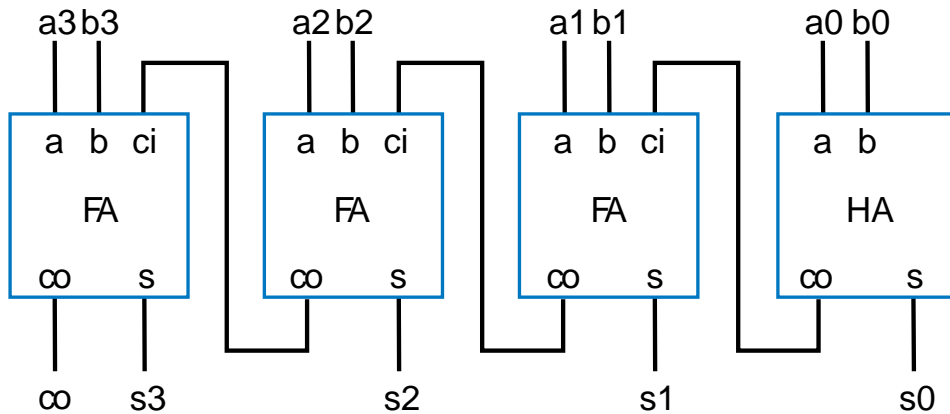
$$\begin{aligned} s &= a'b'c + a'bc' + ab'c' + abc \\ s &= a'(b'c + bc') + a(b'c' + bc) \\ s &= a'(b \text{ xor } c)' + a(b \text{ xor } c) \\ s &= a \text{ xor } b \text{ xor } c \end{aligned}$$

## Step 3: Create the circuit

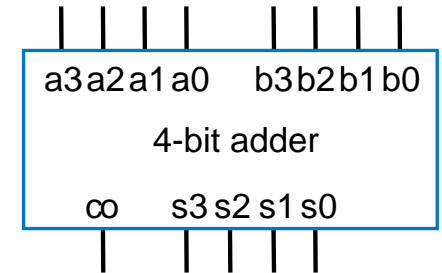


# Carry-Ripple Adder

- 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



(a)

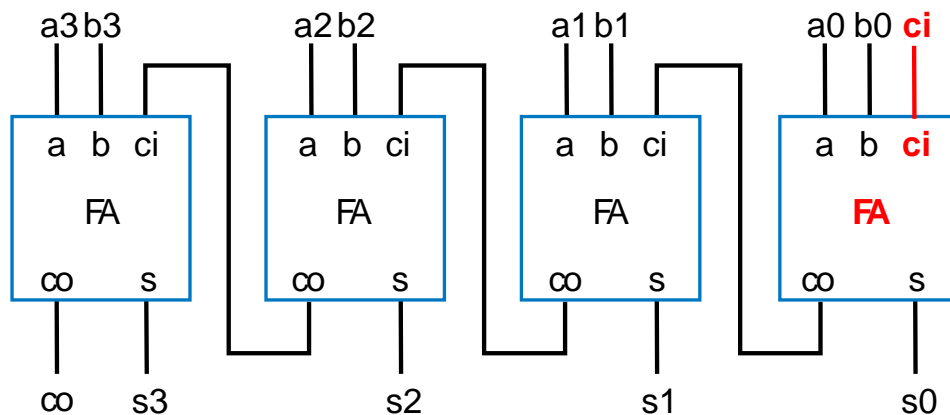


(b)

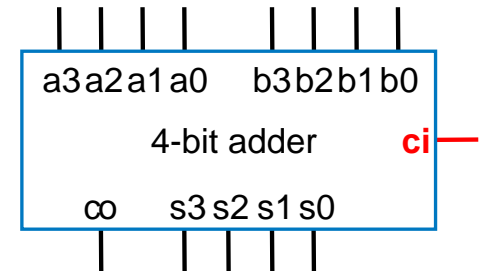


# Carry-Ripple Adder

- 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



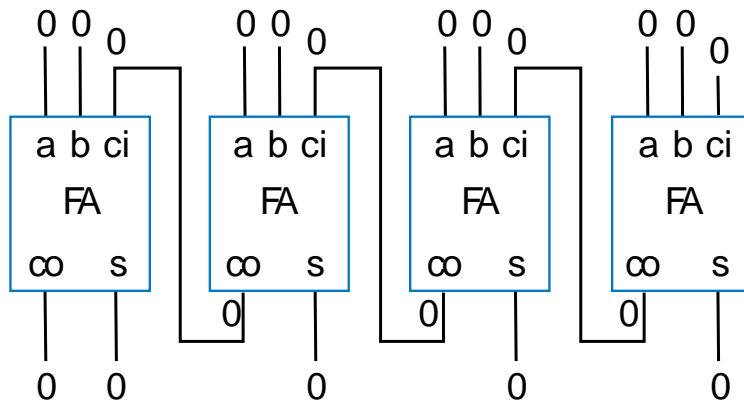
(a)



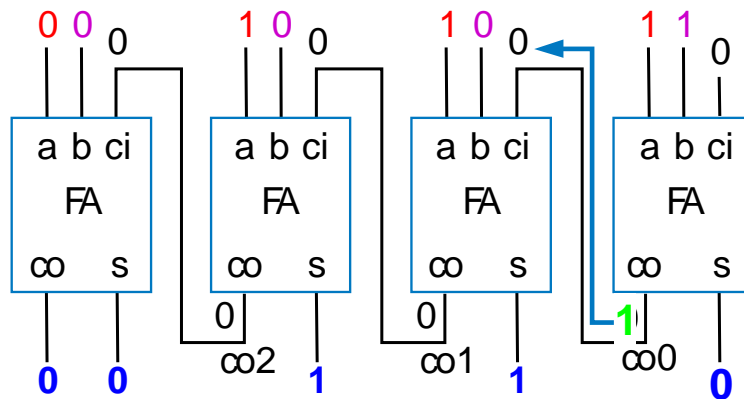
(b)



# Carry-Ripple Adder's Behavior



Assume all inputs initially 0



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

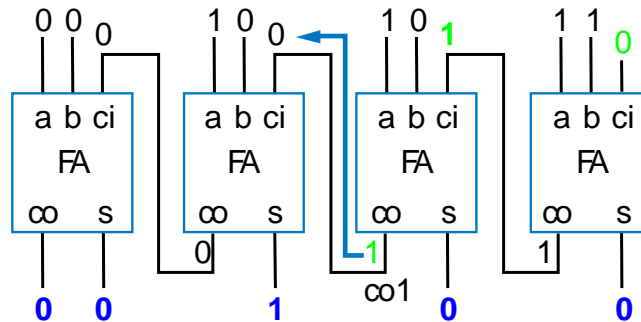
Output after 2 ns (1FA delay)

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

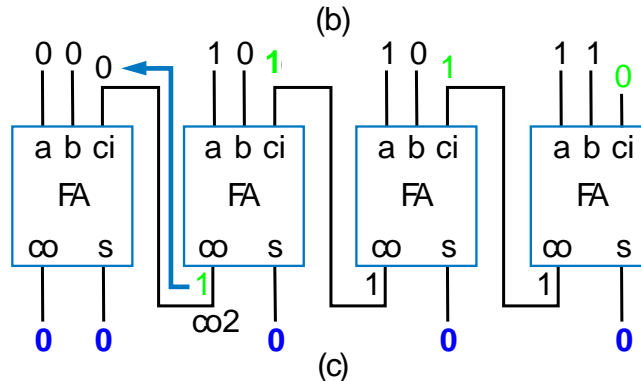


# Carry-Ripple Adder's Behavior

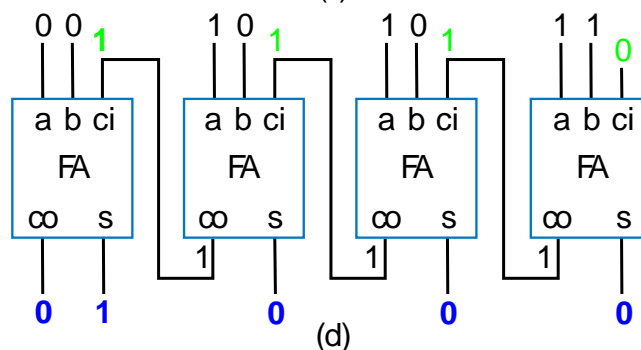
**0111 + 0001**  
 (answer should be 01000)



Outputs after 4ns (2 FA delays)



Outputs after 6ns (3 FA delays)

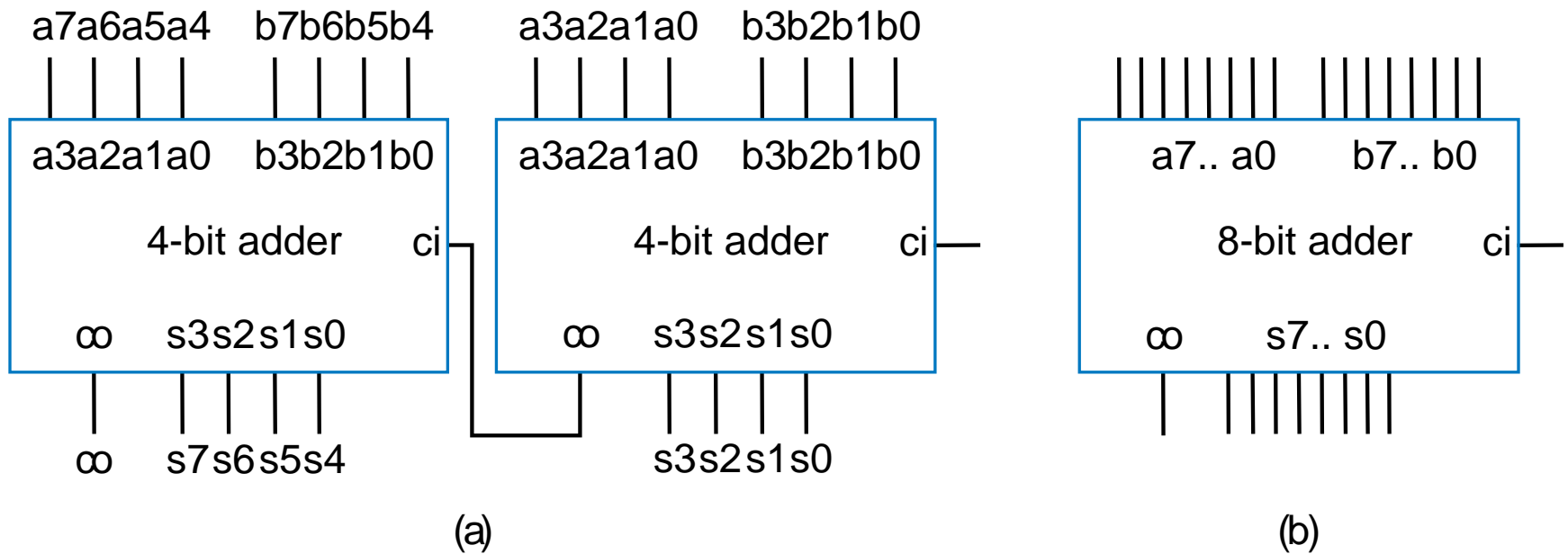


Output after 8ns (4 FA delays)

Correct answer appears after 4 FA delays

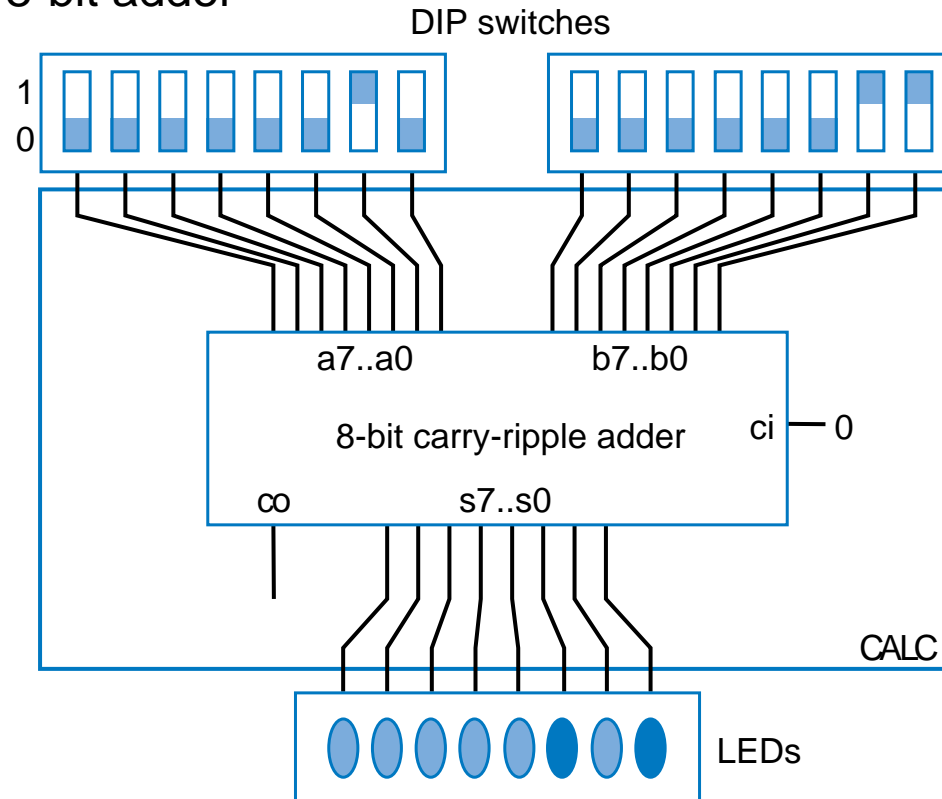


# Cascading Adders



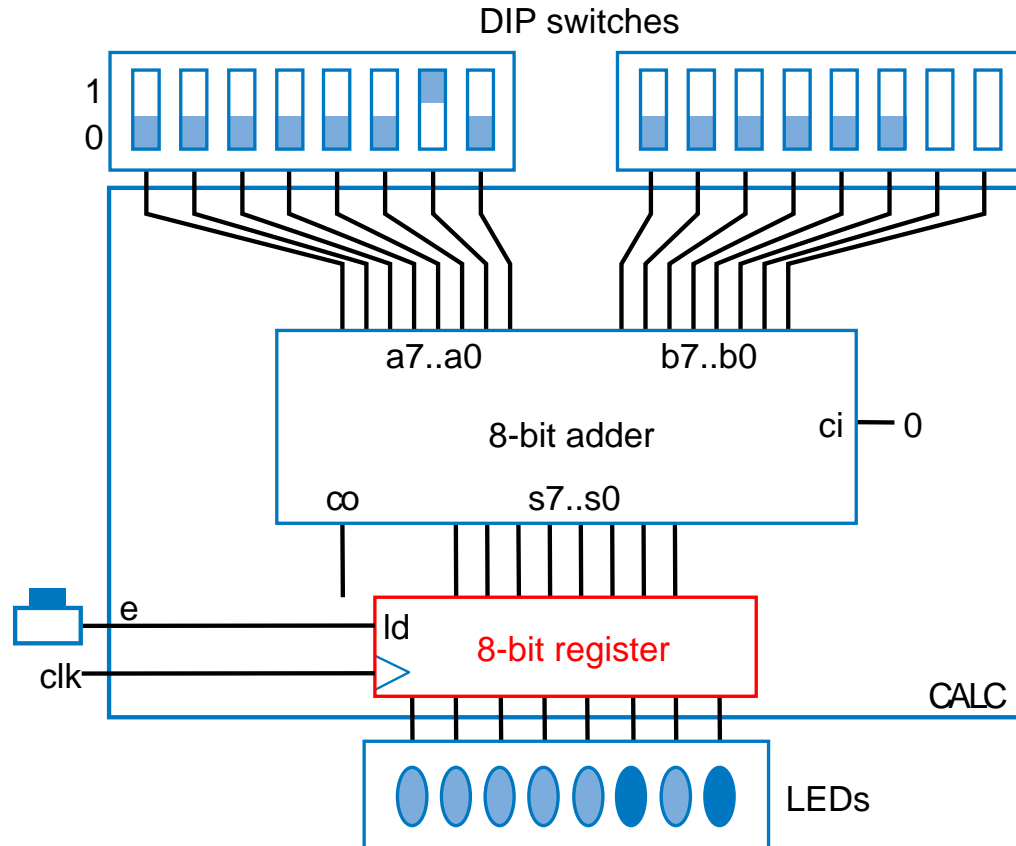
# Adder Example: DIP-Switch-Based Adding Calculator

- 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



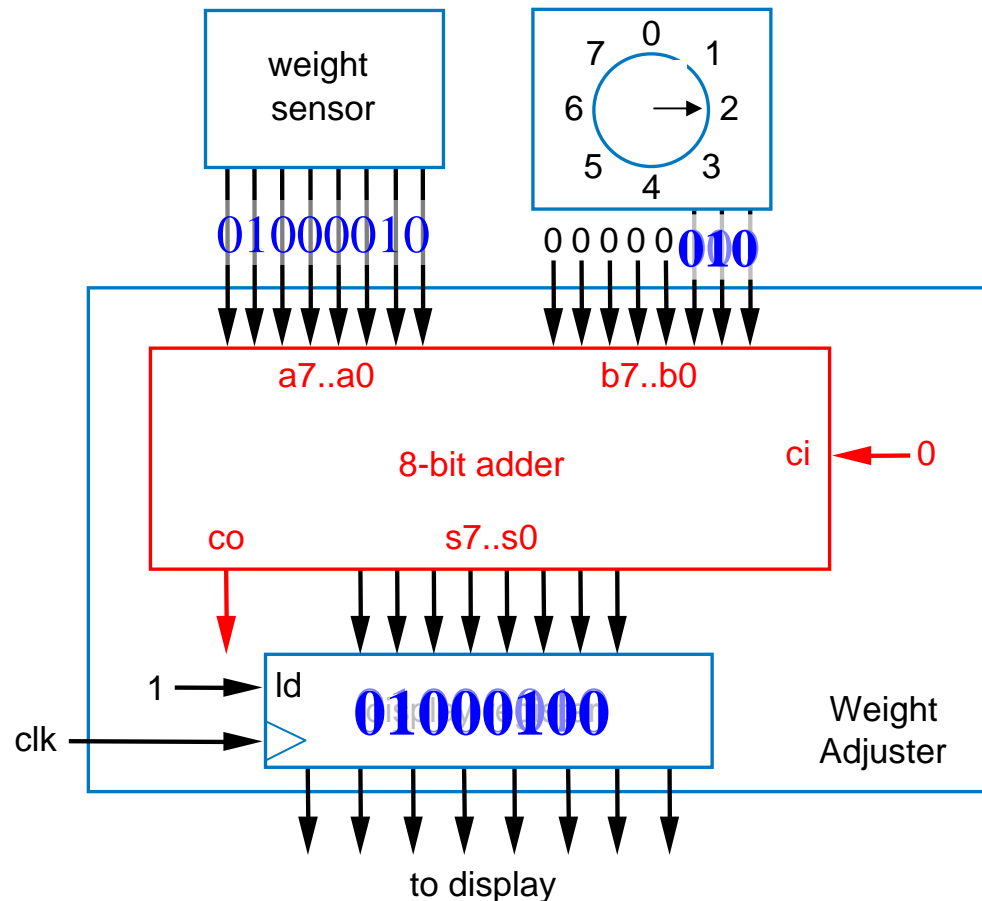
# Adder Example: DIP-Switch-Based Adding Calculator

- 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.



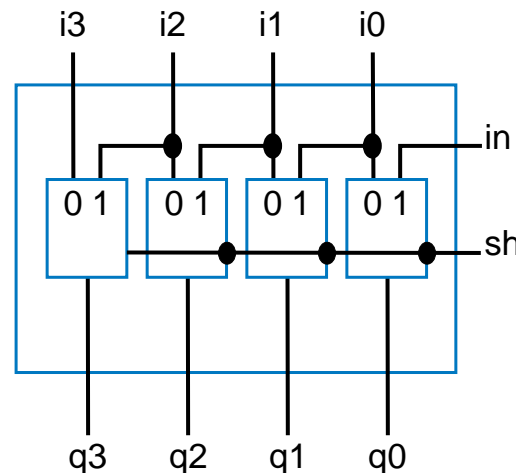
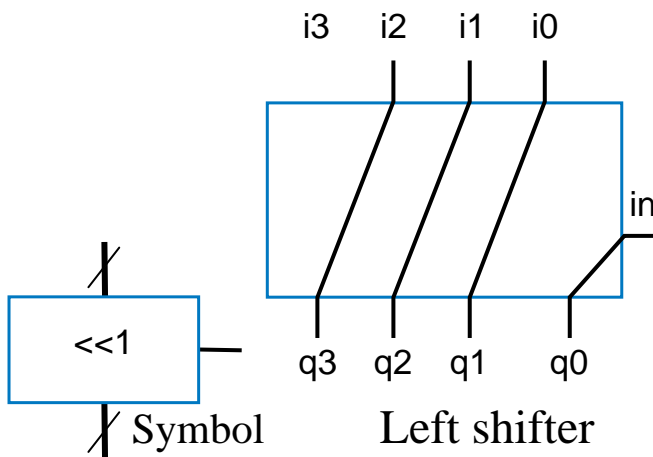
# Adder Example: Compensating Weight Scale

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

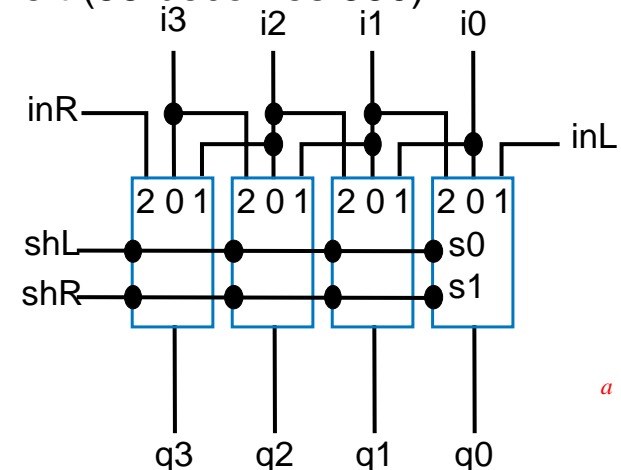


# Shifters

- Shifting (e.g., left shifting 0011 yields 0110) useful for:
  - Manipulating bits
  - Converting serial data to parallel (remember earlier above-mirror display example with shift registers)
  - Shift left once is same as multiplying by 2 (0011 (3) becomes 0110 (6))
    - Why? Essentially appending a 0 -- Note that multiplying decimal number by 10 accomplished just by appending 0, i.e., by shifting left (55 becomes 550)
  - Shift right once same as dividing by 2



Shifter with left shift or no shift

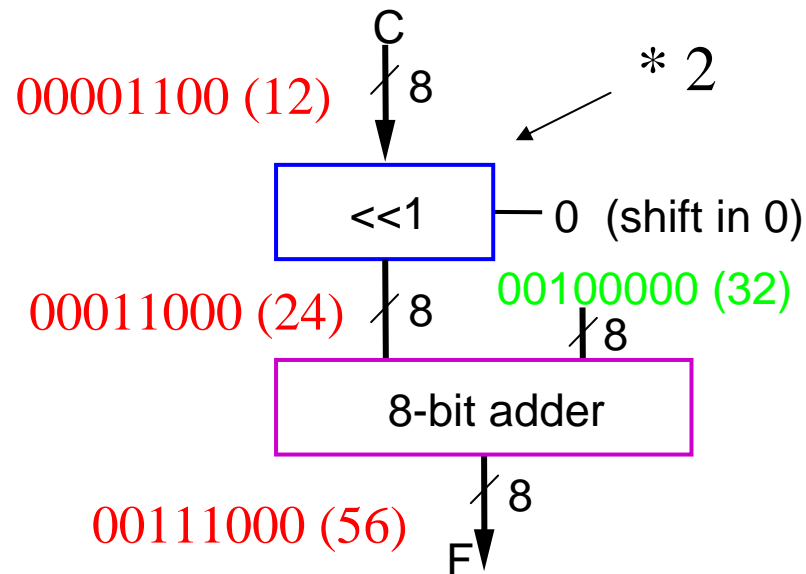


Shifter with left shift, right shift, and no shift



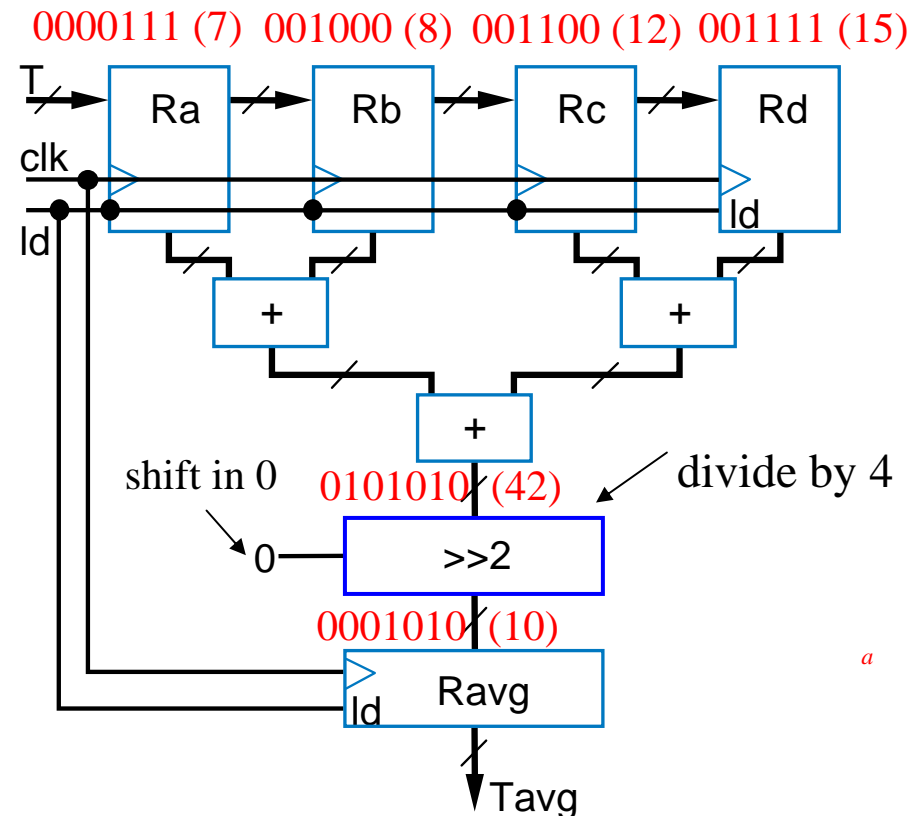
# Shifter Example: Approximate Celsius to Fahrenheit Converter

- Convert 8-bit Celsius input to 8-bit Fahrenheit output
  - $F = C * 9/5 + 32$
  - Approximate:  $F = C * 2 + 32$
  - Use left shift:  $F = \text{left\_shift}(C) + 32$



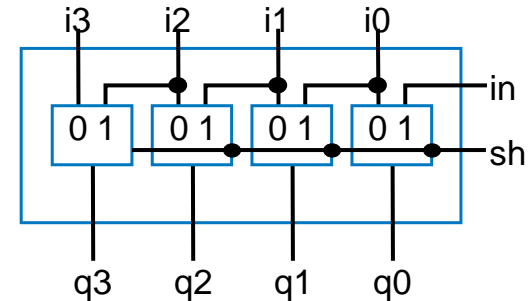
# Shifter Example: Temperature Averager

- Four registers storing a history of temperatures
- Want to output the average of those temperatures
- Add, then divide by four
  - Same as shift right by 2
  - Use three adders, and right shift by two



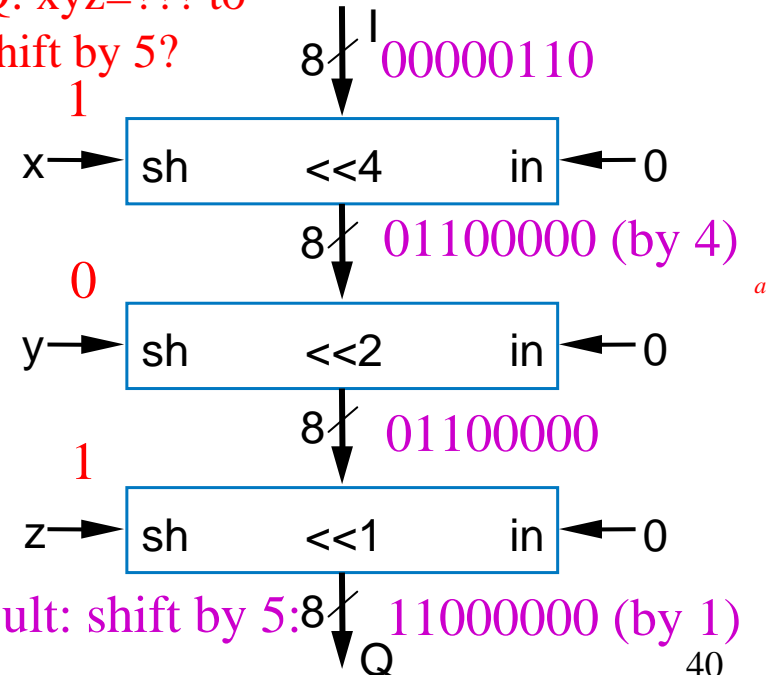
# Barrel Shifter

- A shifter that can shift by any amount
  - 4-bit barrel left shifter can shift left by 0, 1, 2, or 3 positions
  - 8-bit barrel left shifter can shift left by 0, 1, 2, 3, 4, 5, 6, or 7 positions
    - (Shifting an 8-bit number by 8 positions is pointless -- you just lose all the bits)
- Could design using 8x1 muxes and lots of wires
  - Too many wires
- More elegant design
  - Chain three shifters: 4, 2, and 1
  - Can achieve any shift of 0..7 by enabling the correct combination of those three shifters, i.e., shifts should sum to desired amount



Shift by 1 shifter uses 2x1 muxes. 8x1 mux solution for 8-bit barrel shifter: too many wires.

Q: xyz=??? to shift by 5?



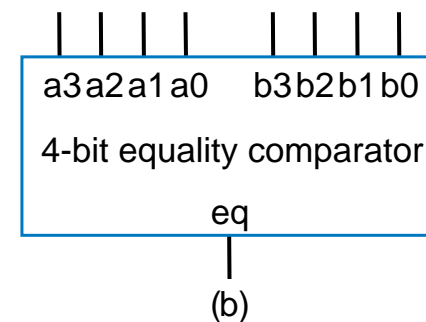
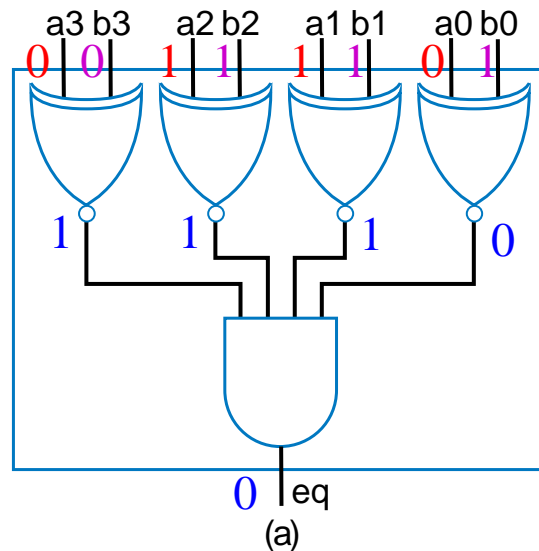
Net result: shift by 5: 11000000 (by 1)



# Comparators

- ***N-bit equality comparator***: Outputs 1 if two N-bit numbers are equal
  - 4-bit equality comparator with inputs A and B
    - $a_3$  must equal  $b_3$ ,  $a_2 = b_2$ ,  $a_1 = b_1$ ,  $a_0 = b_0$ 
      - Two bits are equal if both 1, or both 0
      - $eq = (a_3b_3 + a_3'b_3') * (a_2b_2 + a_2'b_2') * (a_1b_1 + a_1'b_1') * (a_0b_0 + a_0'b_0')$
    - Recall that XNOR outputs 1 if its two input bits are the same
      - $eq = (a_3 \text{ xnor } b_3) * (a_2 \text{ xnor } b_2) * (a_1 \text{ xnor } b_1) * (a_0 \text{ xnor } b_0)$

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  $a_3$  and  $b_3$ . If equal, compare  $a_2$  and  $b_2$ . And so on. Stop if comparison not equal -- whichever's bit is 1 is greater. If never see unequal bit pair,  $A = B$ .

$A = 1011$     $B = 1001$

**1**011   **1**001 **Equal**

**1**011   **1**001 **Equal**

10**1**1   100**1** **Unequal**

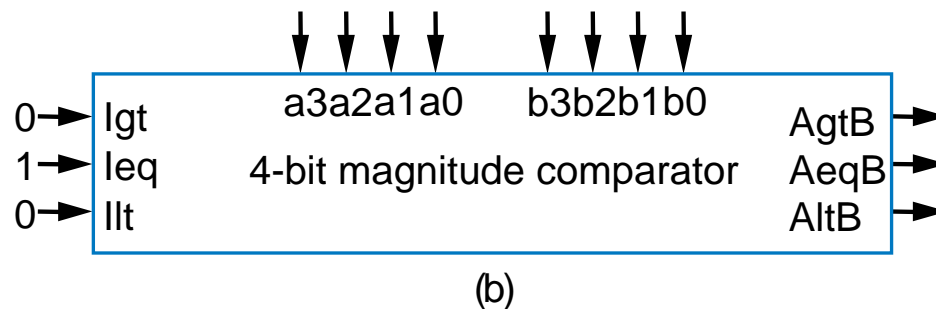
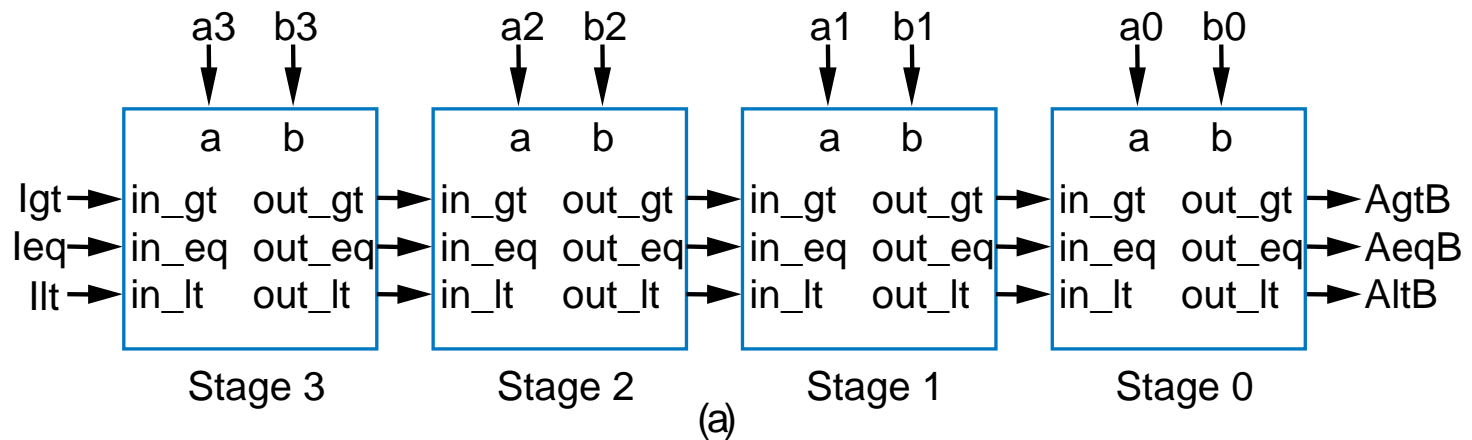
**So  $A > B$**

*a*

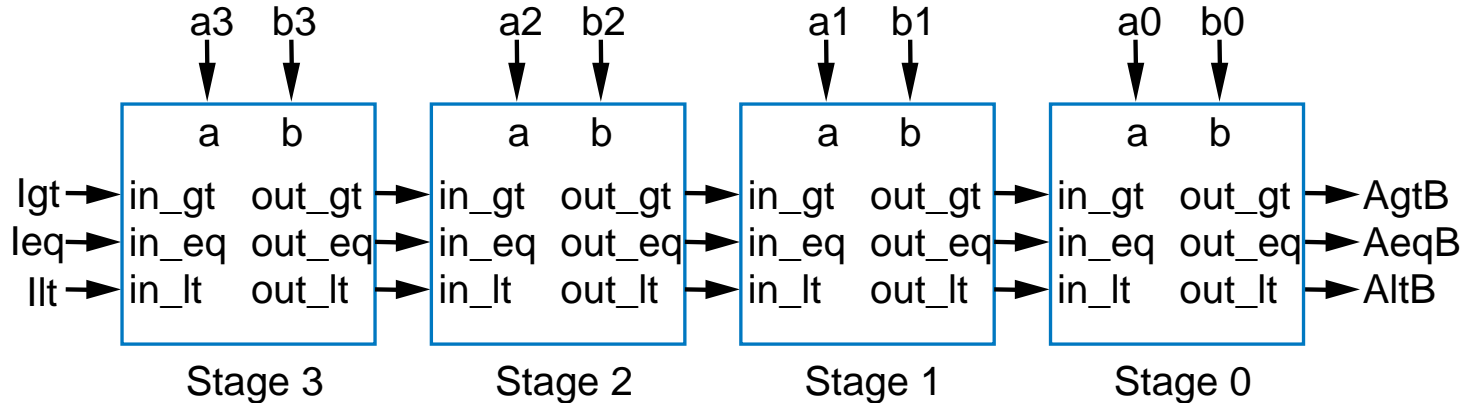


# Magnitude Comparator

- 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



# 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  $b=1$

- $out\_eq = in\_eq * (a \text{ XNOR } b)$

- $A = B$  (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)

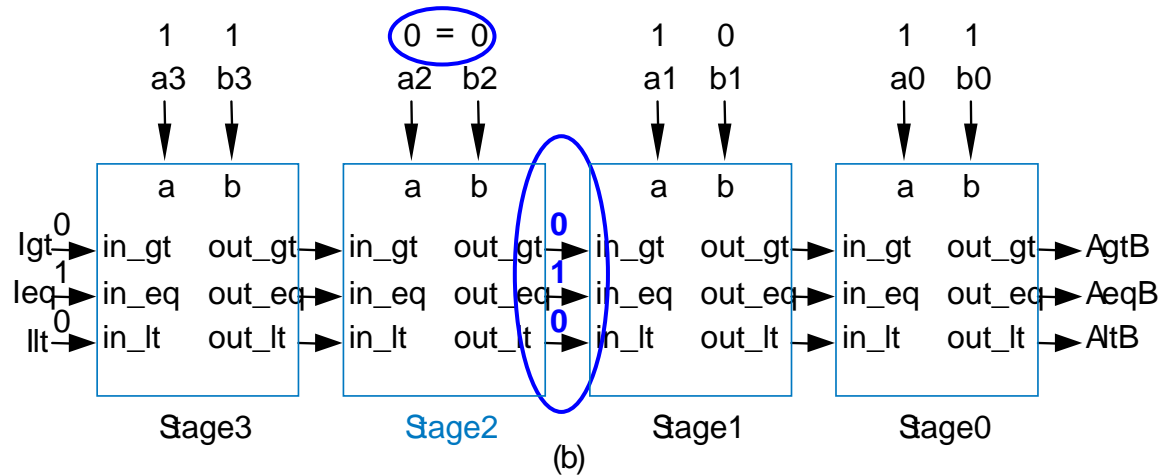
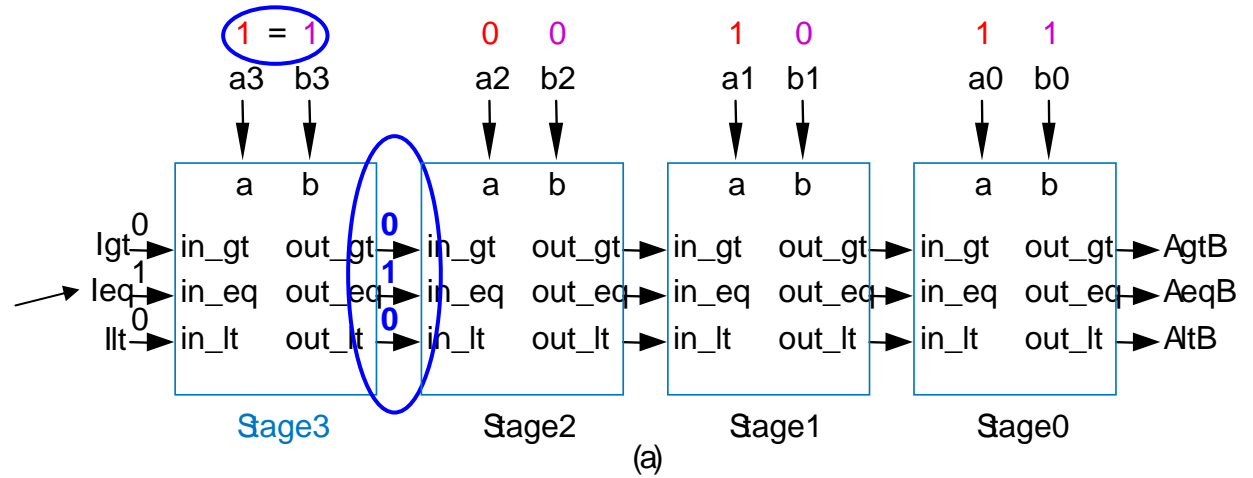


# Magnitude Comparator

- How does it work?

1011 = 1001 ?

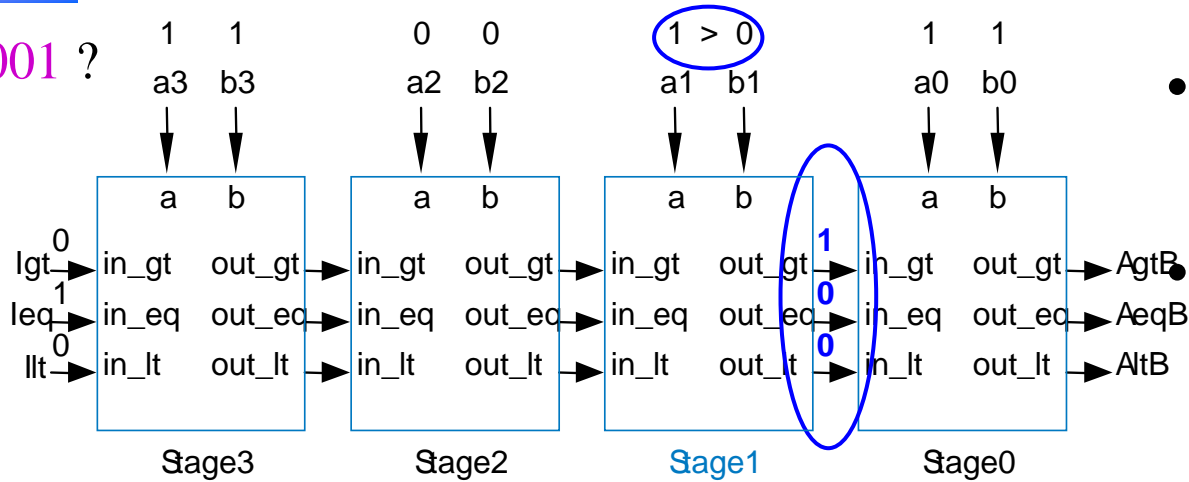
*Ieq=1 causes this stage to compare*



# Magnitude Comparator



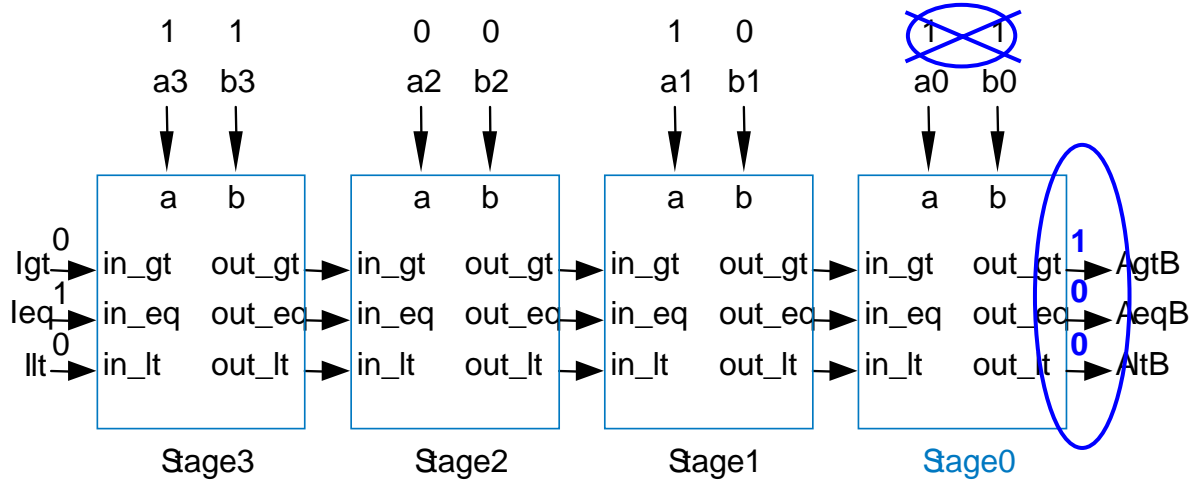
1011 = 1001 ?



(c)

- 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

a

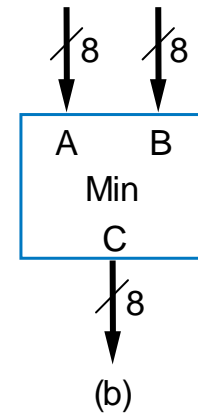
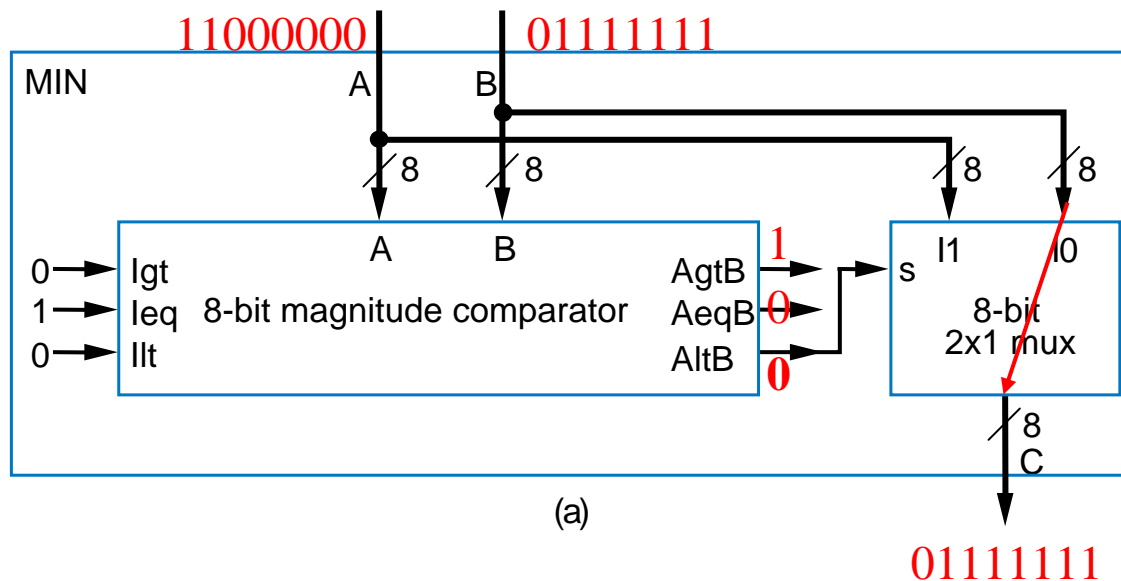


(d)



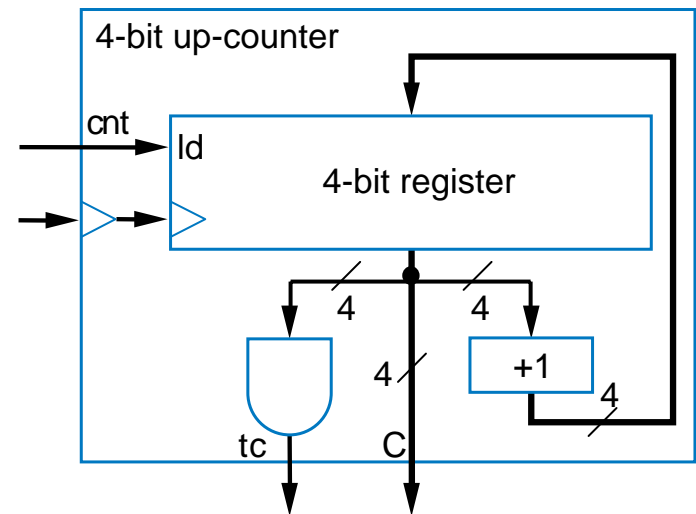
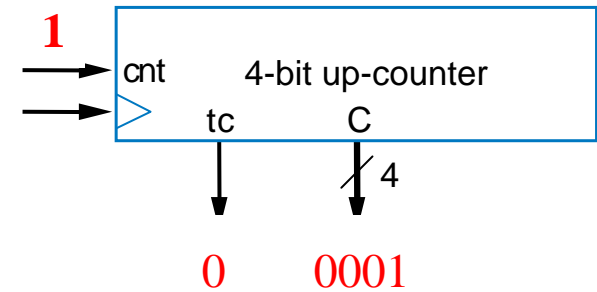
# Magnitude Comparator Example: Minimum of Two Numbers

- 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 B through mux. Else, pass A.



# Counters

- ***N-bit up-counter***: N-bit register that can increment (add 1) to its own value on each clock cycle
  - 0000, 0001, 0010, 0011, ..., 1110, 1111, 0000
  - Note how count “rolls over” from 1111 to 0000
    - Terminal (last) count, tc, equals 1 during value just before rollover
- **Internal design**
  - Register, incrementer, and N-input AND gate to detect terminal count



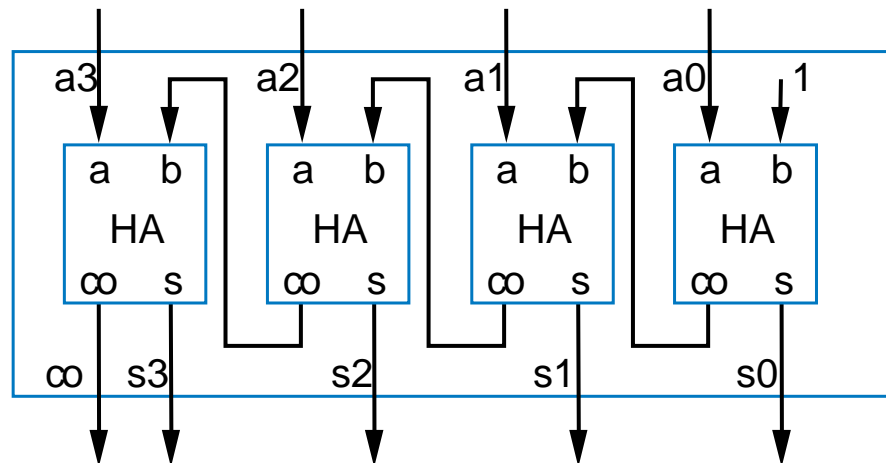
# Incrementer

- Counter design used incrementer
- Incrementer design
  - Could use carry-ripple adder with B input set to 00...001
    - But when adding 00...001 to another number, the leading 0's obviously don't need to be considered -- so just two bits being added per column
  - Use half-adders (adds two bits) rather than full-adders (adds three bits)

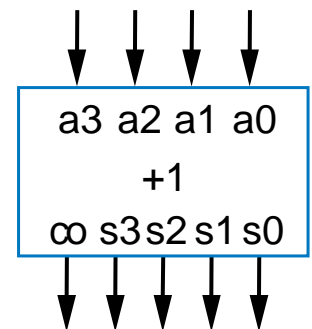
carries: 0 1 1

unused 0 0 1 1

$$\begin{array}{r}
 \text{unused} \\
 + \quad 1 \\
 \hline
 00100
 \end{array}$$



(a)



(b)



# Incrementer

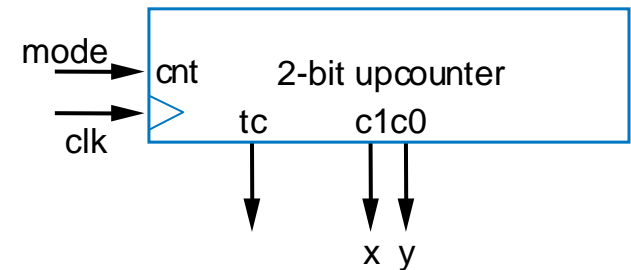
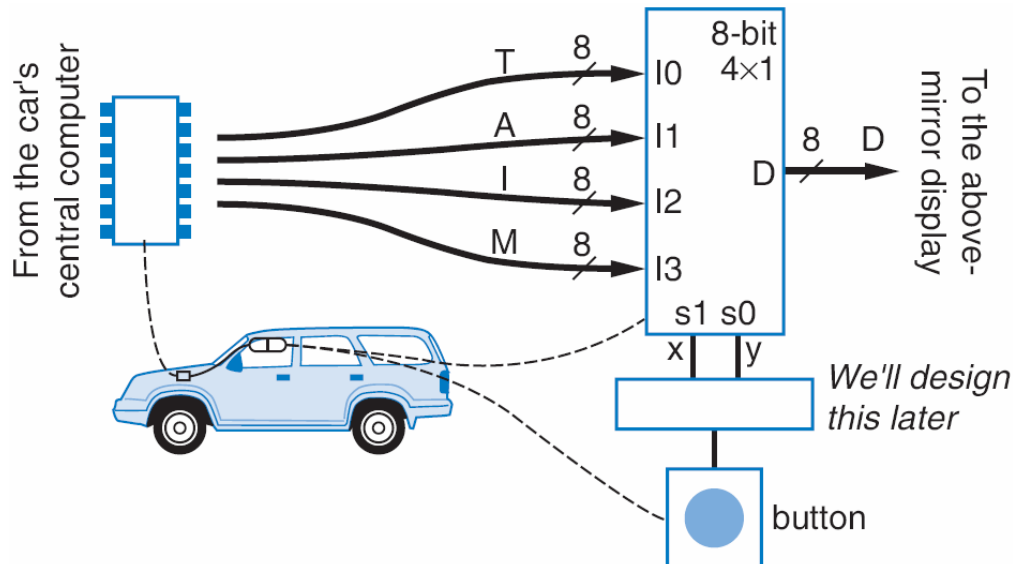
- Can build faster incrementer using combinational logic design process
  - Capture truth table
  - Derive equation for each output
    - $c_0 = a_3a_2a_1a_0$
    - ...
    - $s_0 = a_0'$
  - Results in small and fast circuit
  - Note: works for small N -- larger N leads to exponential growth, like for 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



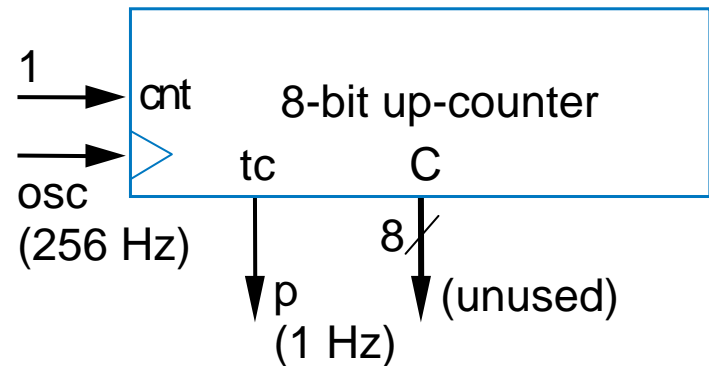
# Counter Example: Mode in Above-Mirror Display

- Recall above-mirror display example from Chapter 2
  - Assumed component that incremented  $xy$  input each time button pressed: 00, 01, 10, 11, 00, 01, 10, 11, 00, ...
  - Can use 2-bit up-counter
    - Assumes  $mode=1$  for just one clock cycle during each button press
      - Recall “Button press synchronizer” example from Chapter 3



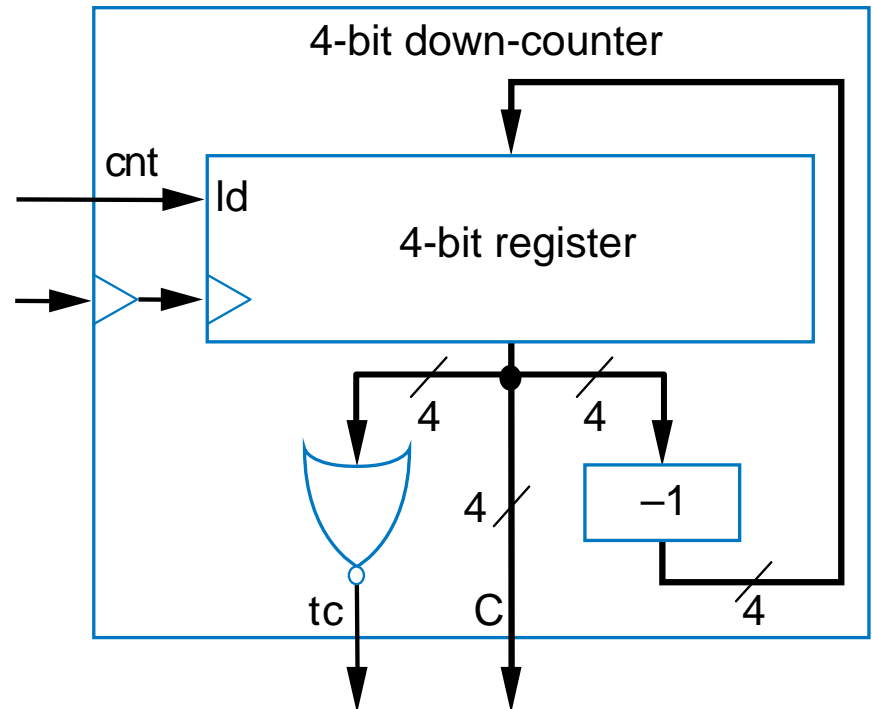
# Counter Example: 1 Hz Pulse Generator Using 256 Hz Oscillator

- Suppose have 256 Hz oscillator, but want 1 Hz pulse
  - 1 Hz is 1 pulse per second -- useful for keeping time
  - Design using 8-bit up-counter, use tc output as pulse
    - Counts from 0 to 255 (256 counts), so pulses tc every 256 cycles



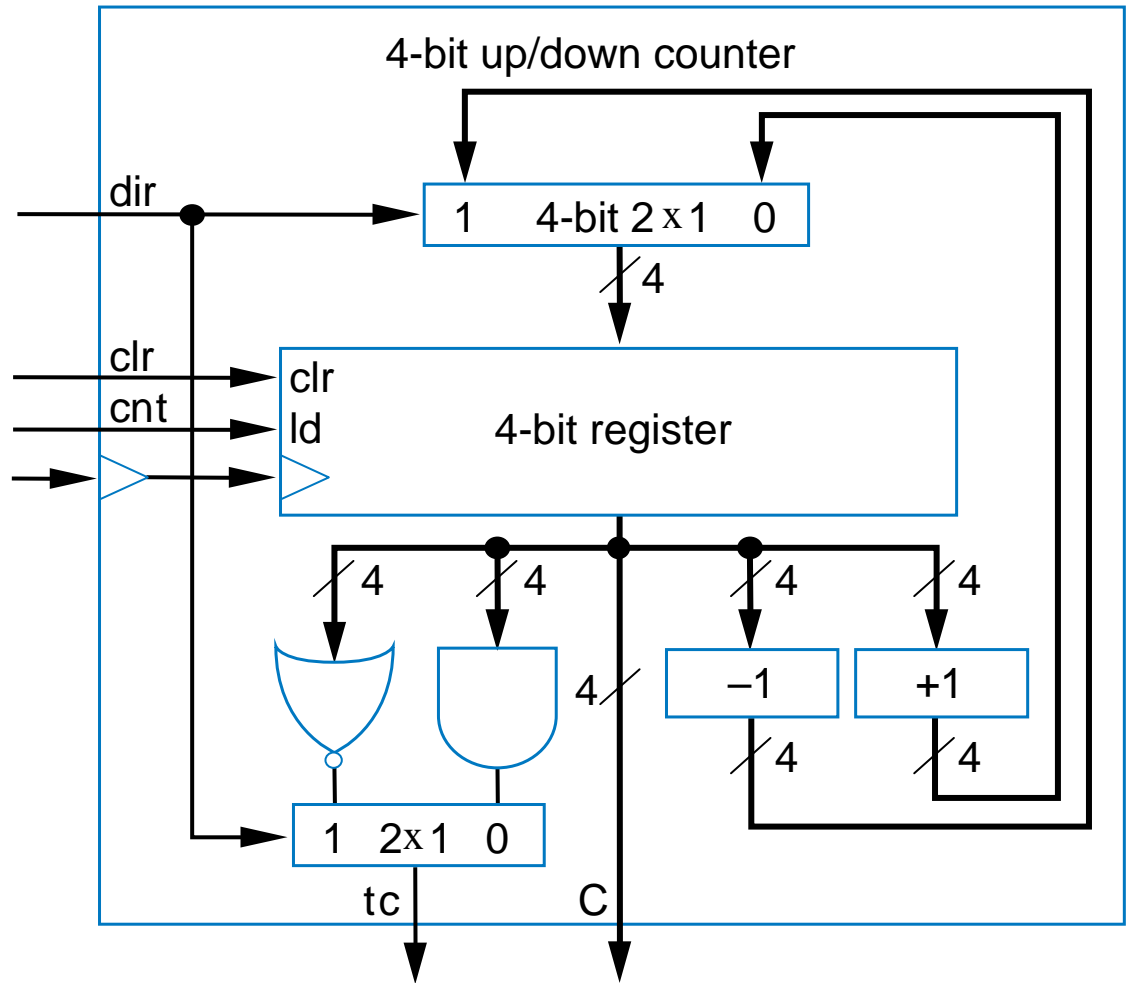
# Down-Counter

- 4-bit down-counter
  - 1111, 1110, 1101, 1100, ..., 0011, 0010, 0001, 0000, 1111, ...
  - Terminal count is 0000
    - Use NOR gate to detect
  - Need decrementer (-1) – design like designed incrementer



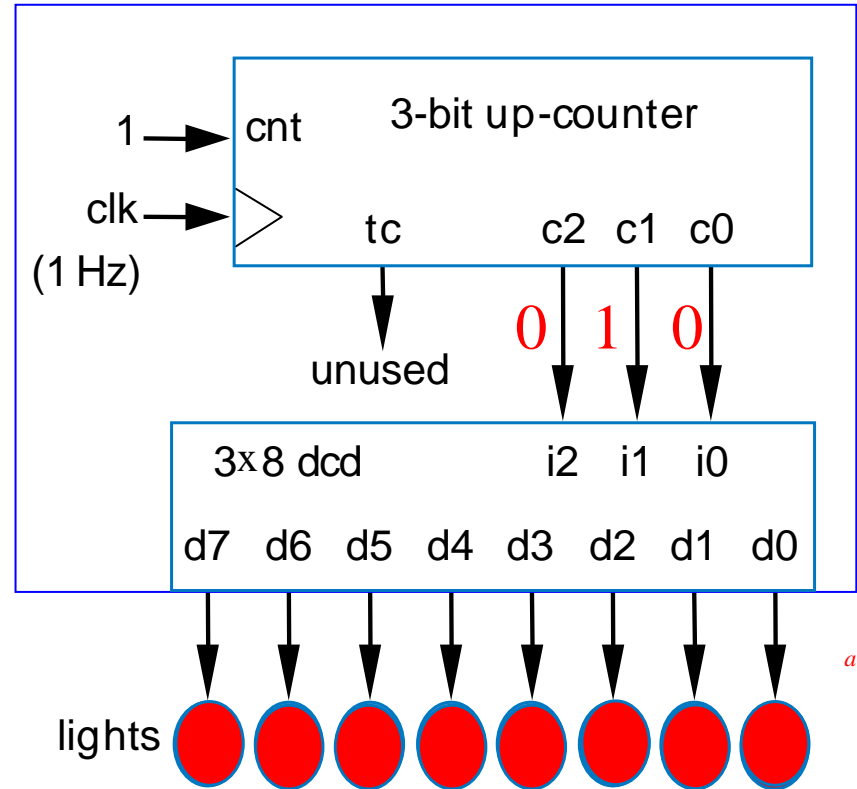
# Up/Down-Counter

- Can count either up or down
  - Includes both incrementer and decrementer
  - Use dir input to select, using 2x1: dir=0 means up
  - Likewise, dir selects appropriate terminal count value



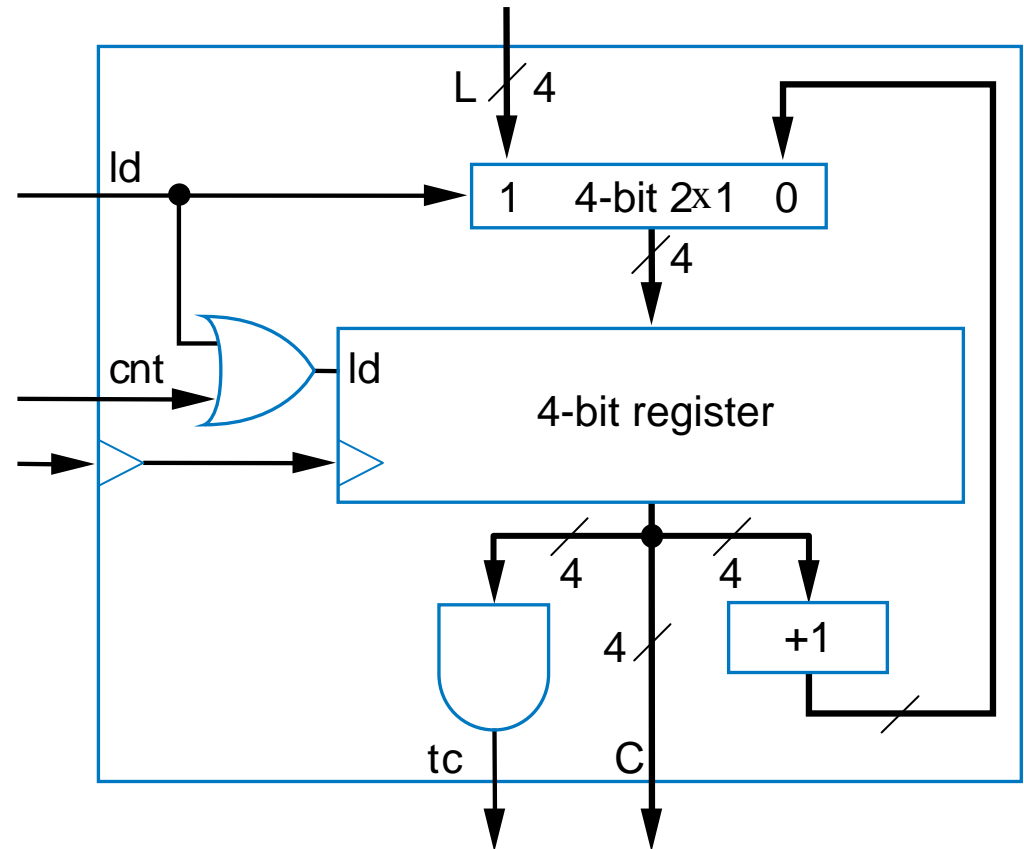
# Counter Example: Light Sequencer

- Illuminate 8 lights from right to left, one at a time, one per second
- Use 3-bit up-counter to counter from 0 to 7
- Use 3x8 decoder to illuminate appropriate light
- Note: Used **3-bit** counter with 3x8 decoder
  - *NOT* an 8-bit counter – why not?



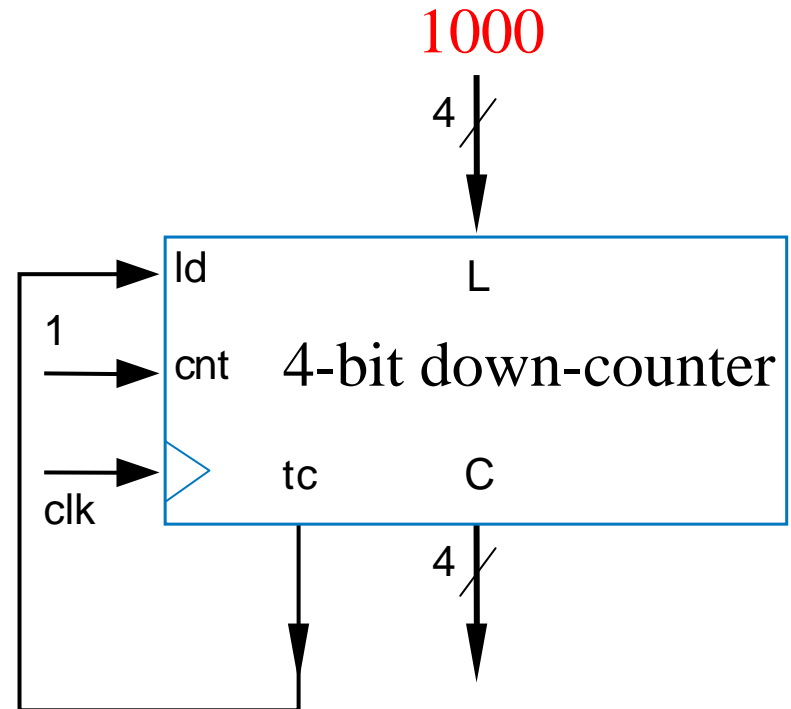
# Counter with Parallel Load

- Up-counter that can be loaded with external value
  - Designed using 2x1 mux
  - $ld$  input selects incremented value or external value
  - Load the internal register when loading external value or when counting



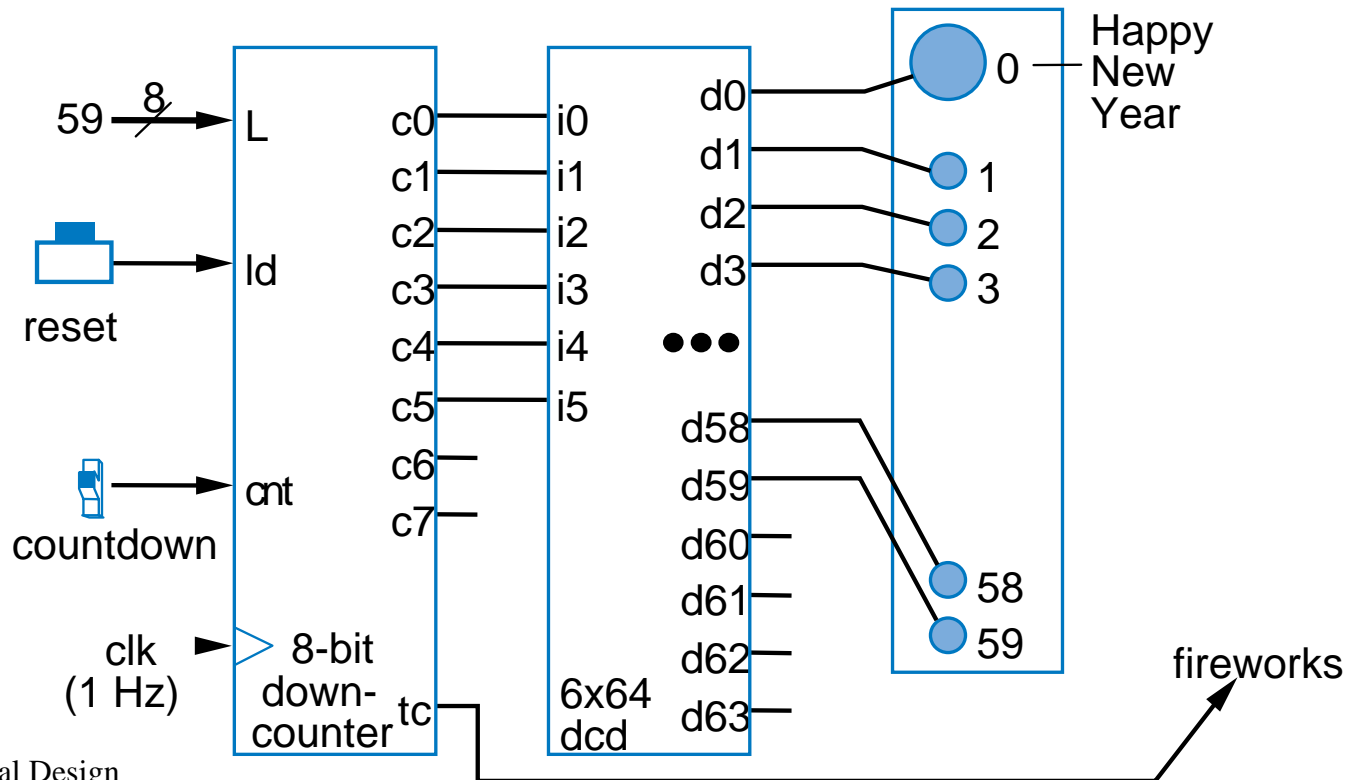
# Counter with Parallel Load

- Useful to create pulses at specific multiples of clock
  - Not just at N-bit counter's natural wrap-around of  $2^N$
- Example: Pulse every 9 clock cycles
  - Use 4-bit down-counter with parallel load
  - Set parallel load input to 8 (1000)
  - Use terminal count to reload
    - When count reaches 0, next cycle loads 8.
  - Why load 8 and not 9? Because 0 is included in count sequence:
    - 8, 7, 6, 5, 4, 3, 2, 1, 0 → 9 counts



# Counter Example: New Year's Eve Countdown Display

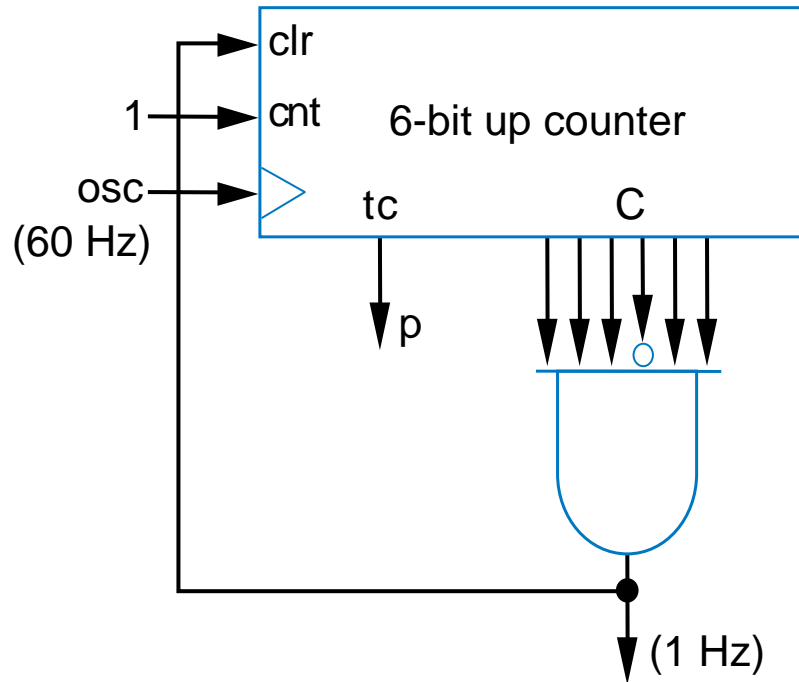
- Chapter 2 example previously used microprocessor to counter from 59 down to 0 in binary
- Can use 8-bit (or 7- or 6-bit) down-counter instead, initially loaded with 59



# Counter Example:

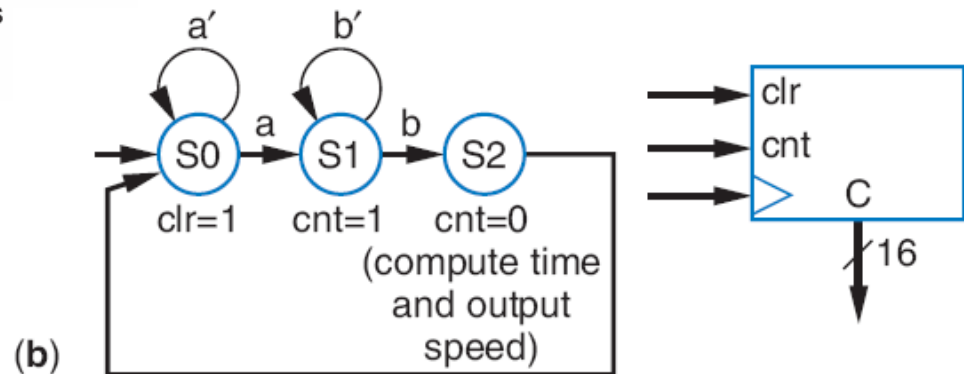
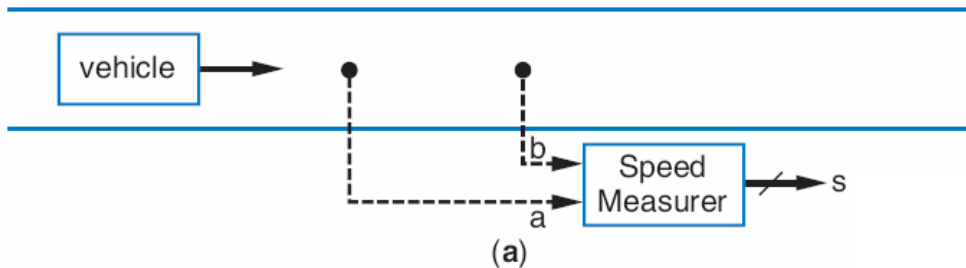
## 1 Hz Pulse Generator from 60 Hz Clock

- U.S. electricity standard uses 60 Hz signal
  - Device may convert that to 1 Hz signal to count seconds
- Use 6-bit up-counter
  - Can count from 0 to 63
  - Create simple logic to detect 59 (for 60 counts)
    - Use to clear the counter back to 0 (or to load 0)



# Timer

- A type of counter used to measure time
  - If we know the counter's clock frequency and the count, we know the time that's been counted
- Example: Compute car's speed using two sensors
  - First sensor (a) clears and starts timer
  - Second sensor (b) stops timer
  - Assuming clock of 1kHz, timer output represents time to travel between sensors. Knowing the distance, we can compute speed



## 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  $6 * 3$ )

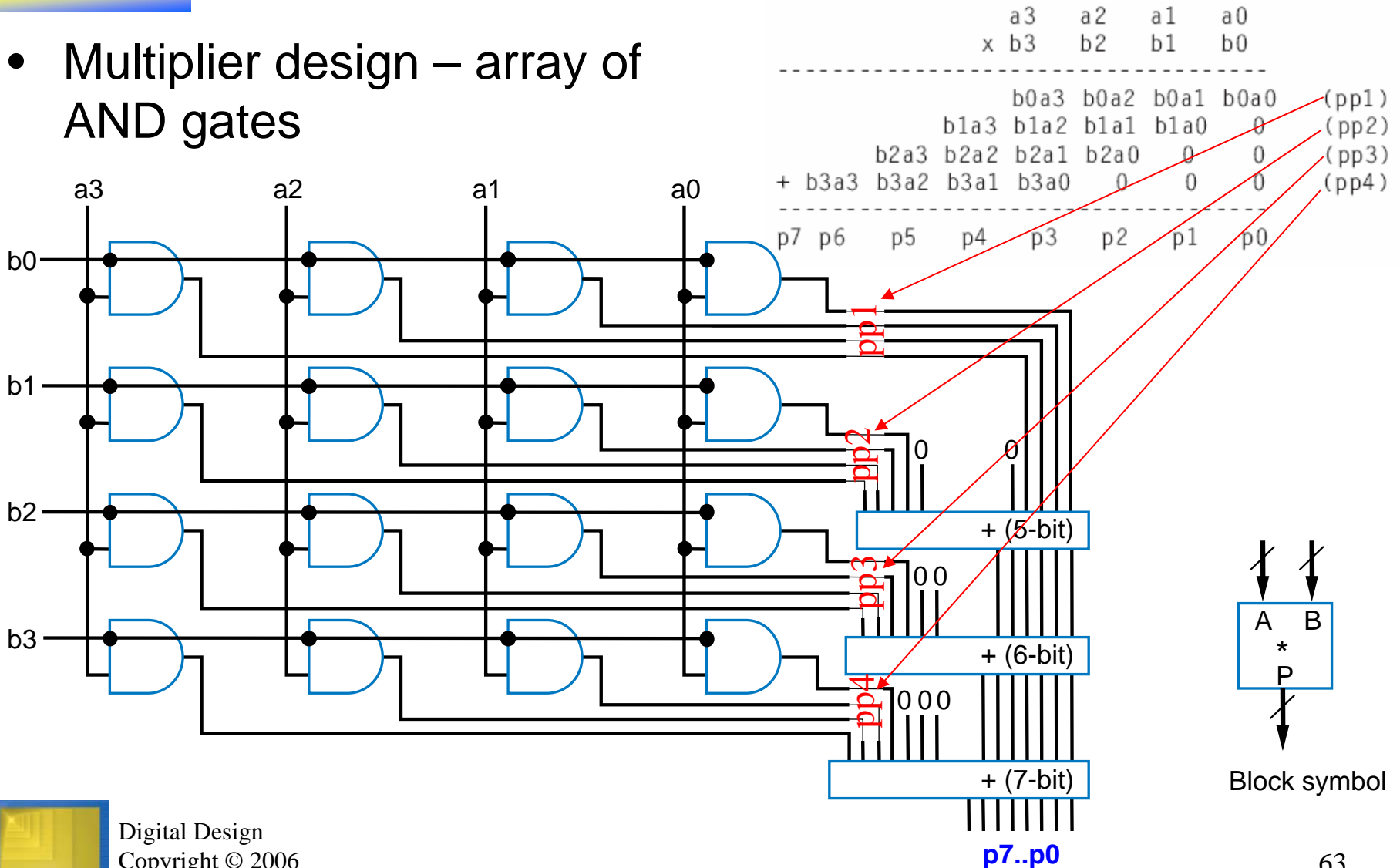
```





# Multiplier – Array Style

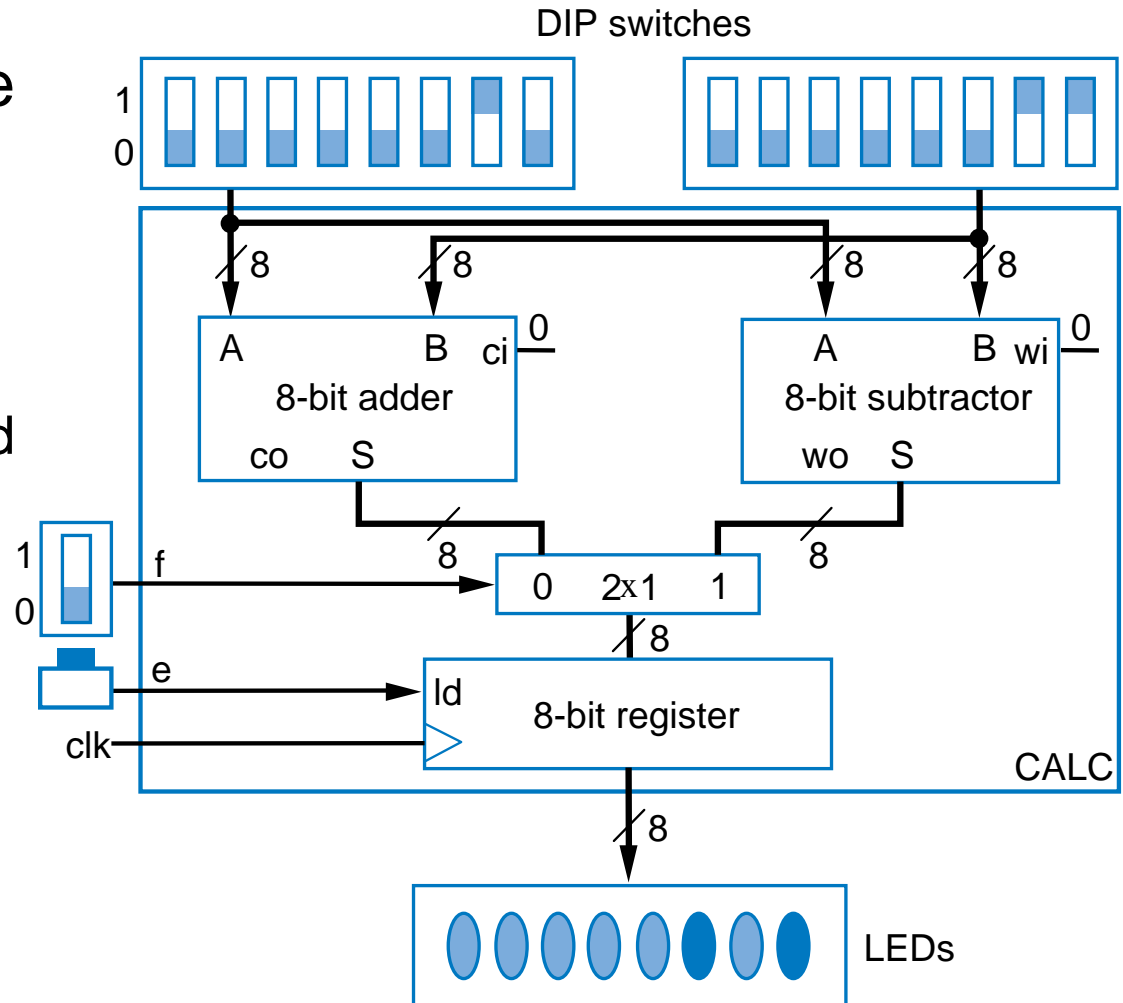
- Multiplier design – array of AND gates





# 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 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



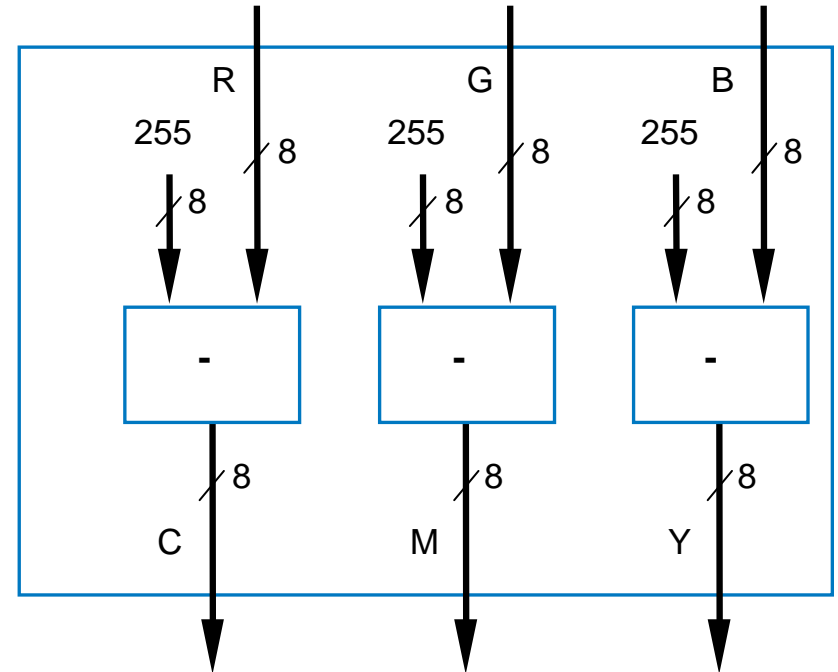
- Printers use opposite color scheme
  - Because inks *absorb* light
  - Use complementary colors of RGB: Cyan (absorbs red), reflects green and blue, Magenta (absorbs green), and Yellow (absorbs blue)



# 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

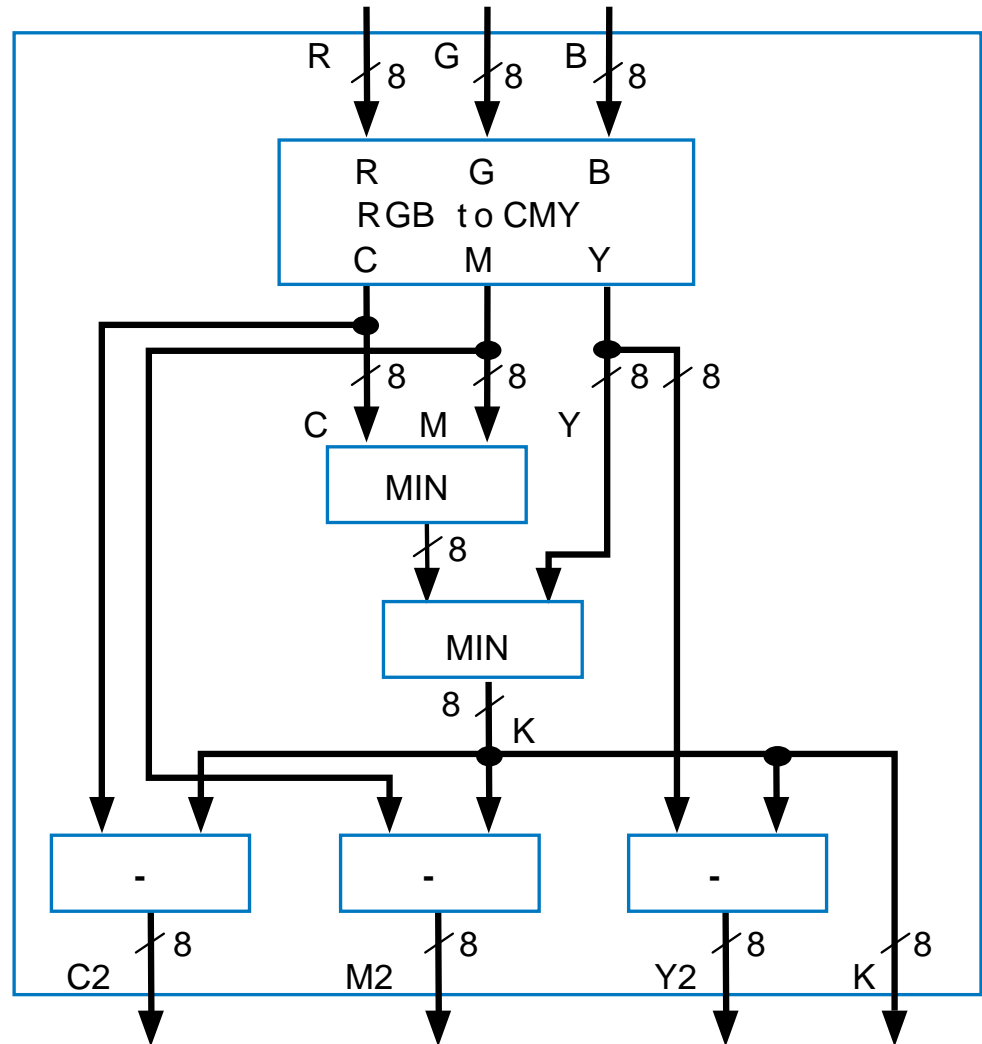
- Try to save colored inks
  - Expensive
  - Imperfect – mixing C, M, Y doesn't yield good-looking black
- Solution: Factor out the black or gray from the color, print that part using black ink
  - e.g., CMY of (250,200,200)=  
(200,200,200) + (50,0,0).
    - (200,200,200) is a dark gray – use black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

- Negative numbers common
  - How represent 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!



# Ten's Complement

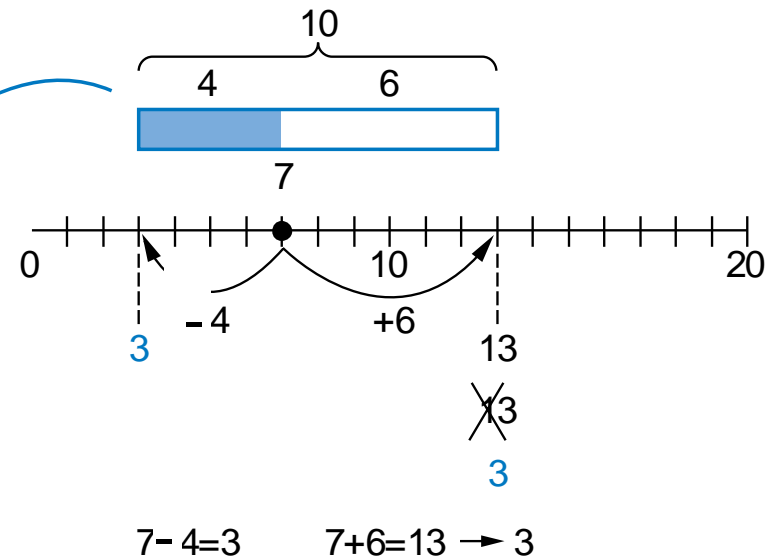
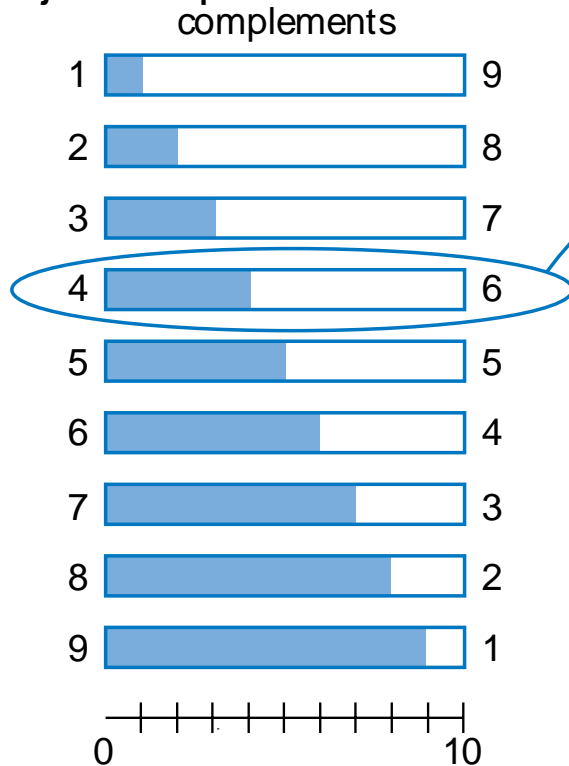
- Before introducing two's complement, let's consider ten's complement
  - But, be aware that computers DO NOT USE TEN'S COMPLEMENT. Introduced for intuition only.
  - Complements for each base ten number shown to right – Complement is the number that when added results in 10

1	→	9
2	→	8
3	→	7
4	→	6
5	→	5
6	→	4
7	→	3
8	→	2
9	→	1



# Ten's Complement

- Nice feature of ten's complement
  - Instead of subtracting a number, adding its complement results in answer exactly 10 too much
  - So just drop the 1 – results in subtracting using addition only



Adding the complement results in an answer exactly 10 too much – dropping the tens column gives the right answer.



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

- Hold on!
  - Sure, adding the ten's complement achieves subtraction using addition only
  - But don't we have to perform *subtraction* to have determined the complement in the first place? e.g., we only know that the complement of 4 is 6 by subtracting  $10-4=6$  in the first place.
- True – but 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!

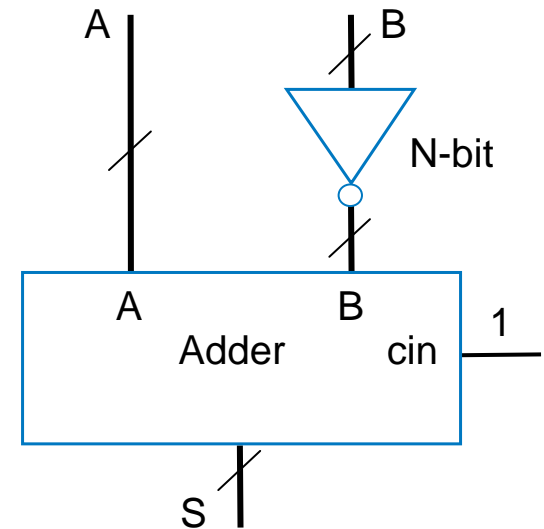
**Q: What is the two's complement of 0101?**    A:  $1010+1=1011$  a  
(check:  $0101+1011=10000$ )

**Q: What is the two's complement of 0011?**    A:  $1100+1=1101$



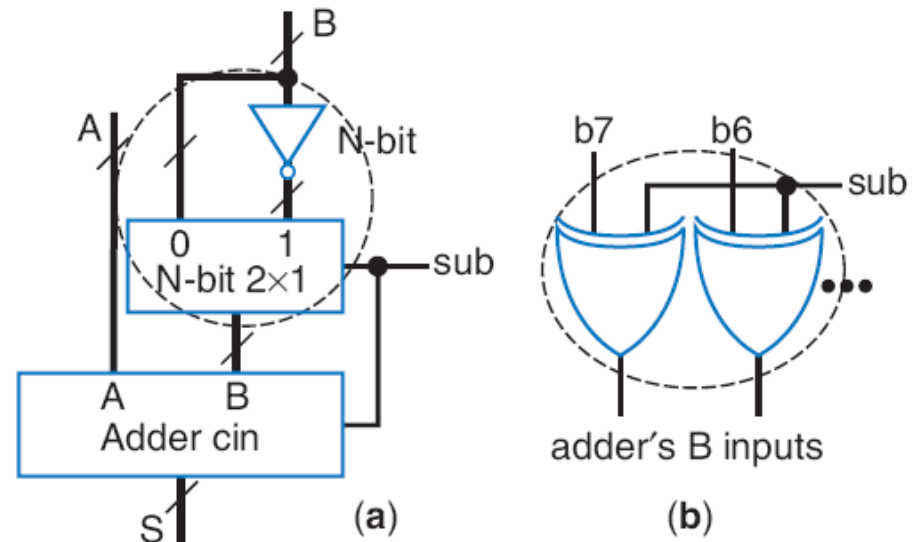
# Two's Complement Subtractor Built with an Adder

- 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



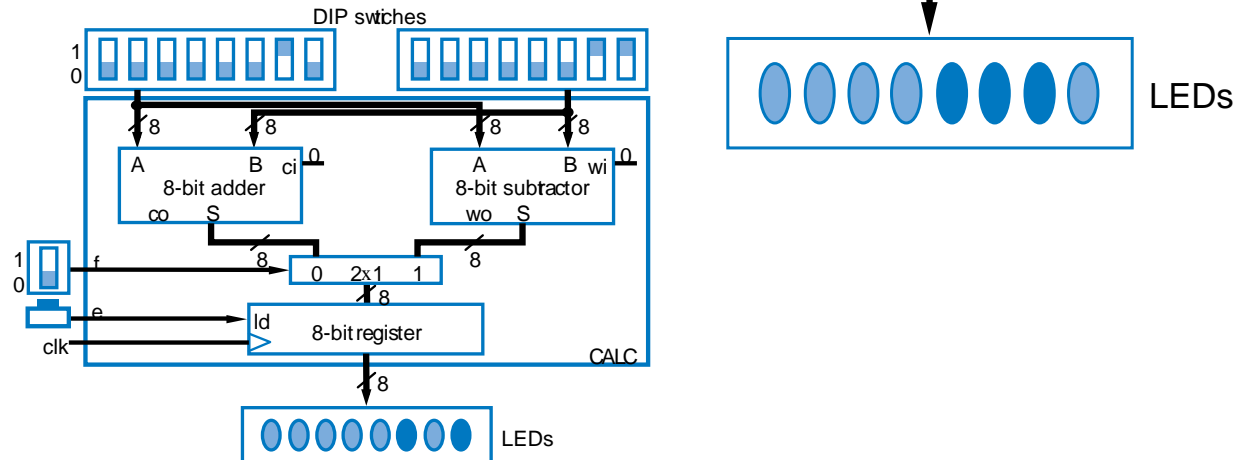
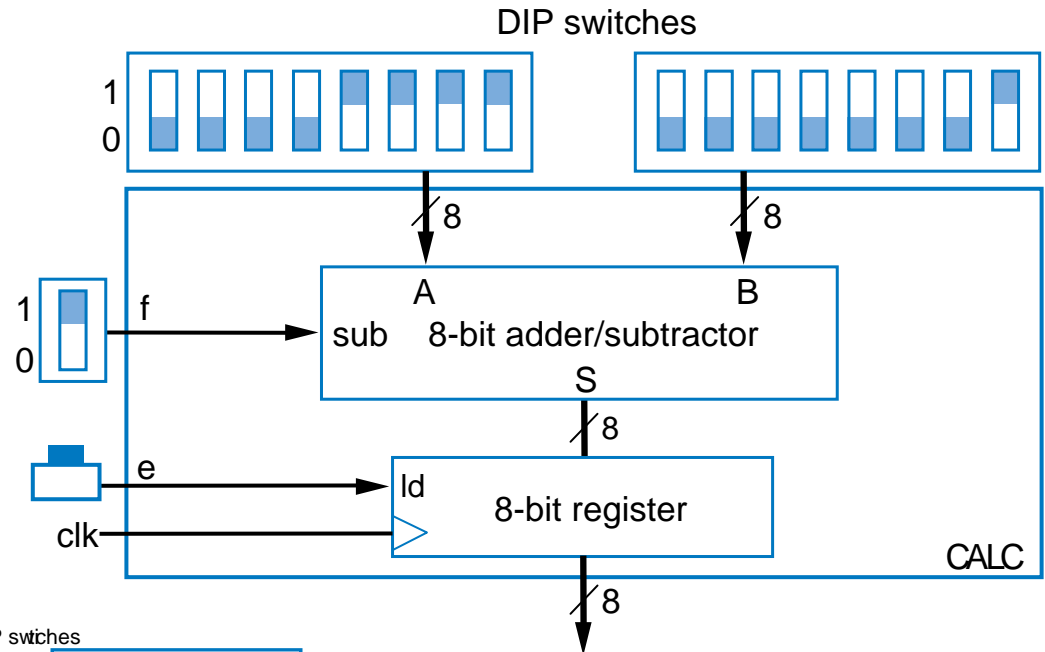
# Adder/Subtractor

- 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



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

sign bits			
$\begin{array}{r} \textcircled{0} \ 1 \ 1 \ 1 \\ + \textcircled{0} \ 0 \ 0 \ 1 \\ \hline \textcircled{1} \ 0 \ 0 \ 0 \end{array}$	$\begin{array}{r} \textcircled{1} \ 1 \ 1 \ 1 \\ + \textcircled{1} \ 0 \ 0 \ 0 \\ \hline \textcircled{0} \ 1 \ 1 \ 1 \end{array}$	$\begin{array}{r} \textcircled{1} \ 0 \ 0 \ 0 \\ + \textcircled{0} \ 1 \ 1 \ 1 \\ \hline \textcircled{1} \ 1 \ 1 \ 1 \end{array}$	
overflow	overflow	no overflow	
(a)	(b)	(c)	

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

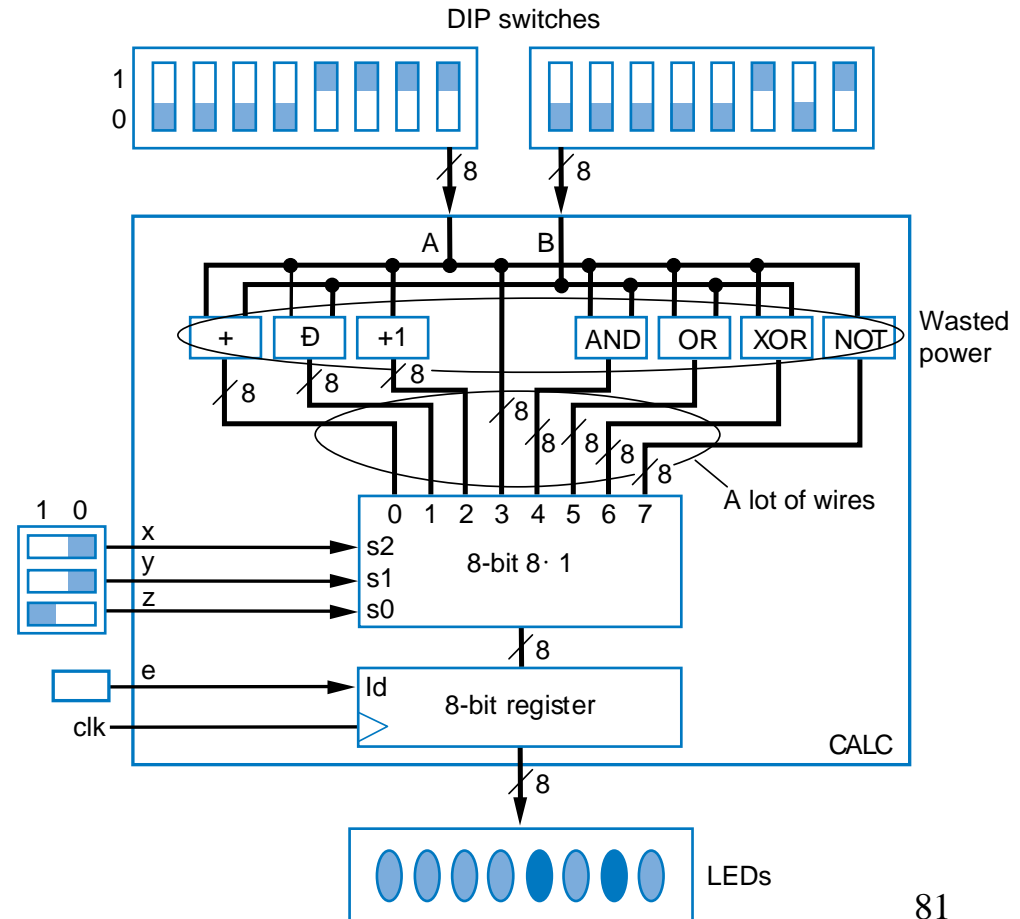


# Multifunction Calculator without an ALU

- 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

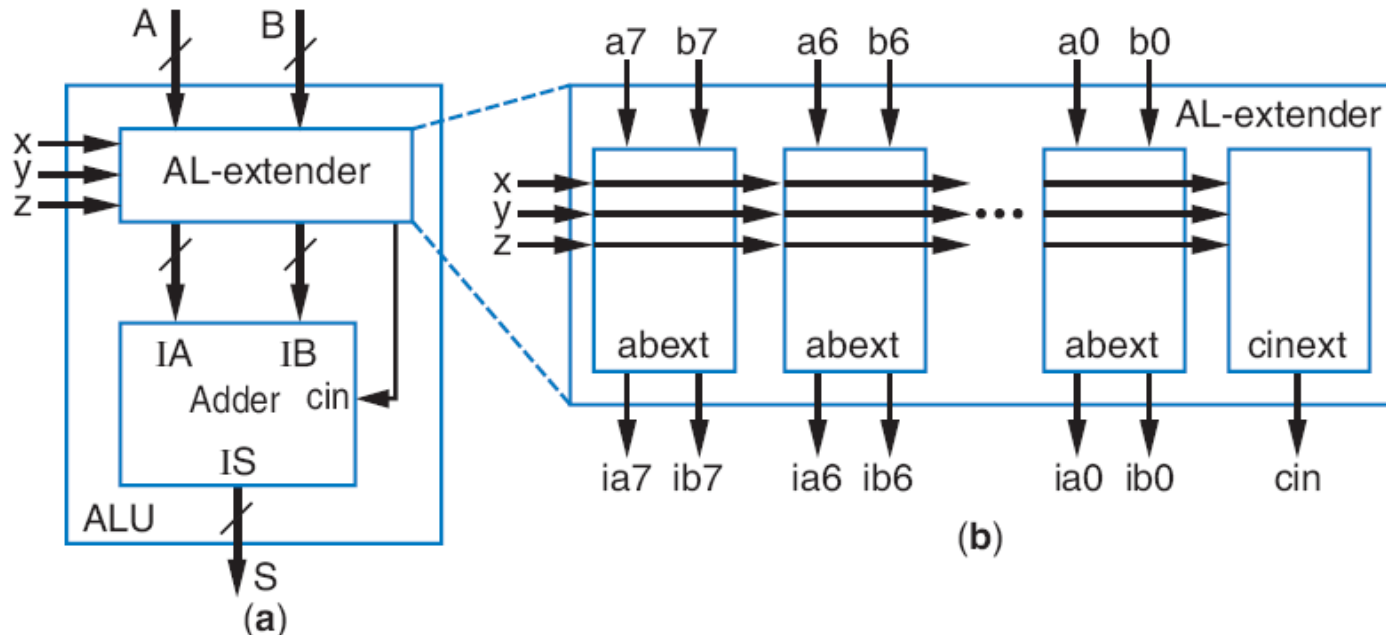
TABLE 4.2 Desired calculator operations

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

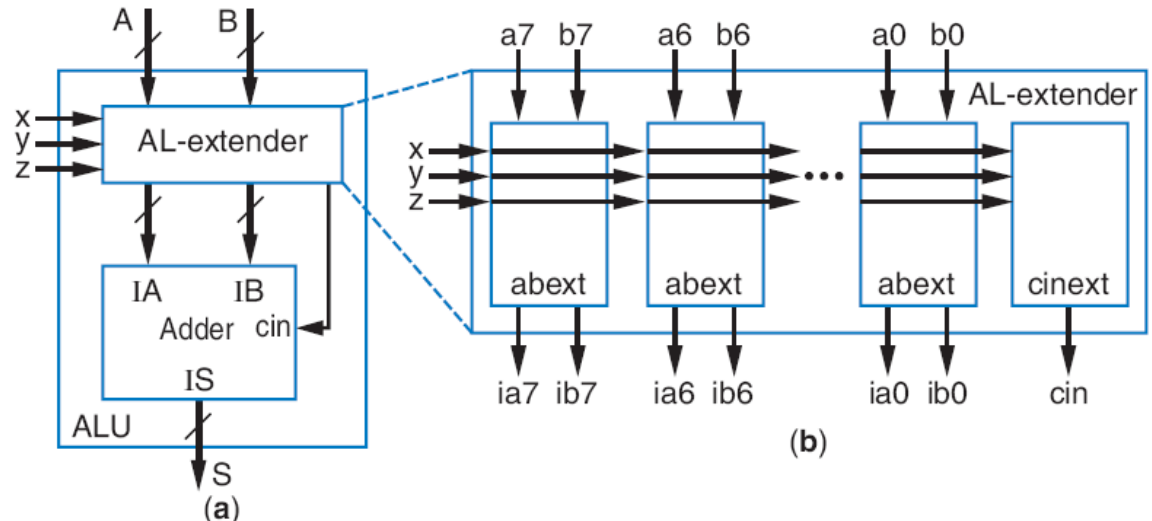
- More efficient design uses ALU
  - ALU design not just separate components multiplexed (same problem as previous slide!),
  - Instead, ALU design uses single adder, plus logic in front of adder's A and B inputs
    - Logic in front is called an arithmetic-logic extender
  - Extender modifies the A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

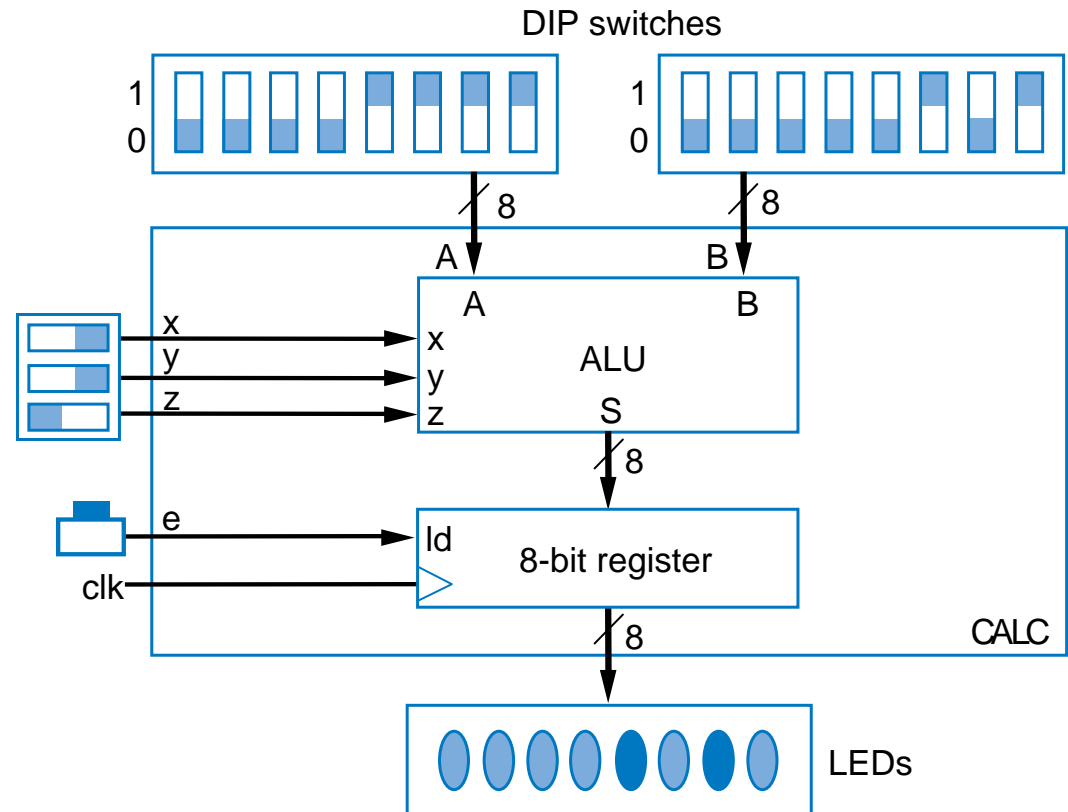
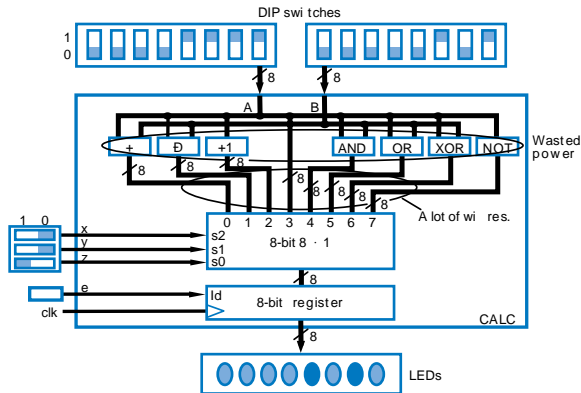
Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	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  $S=A \text{ AND } B$  – 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

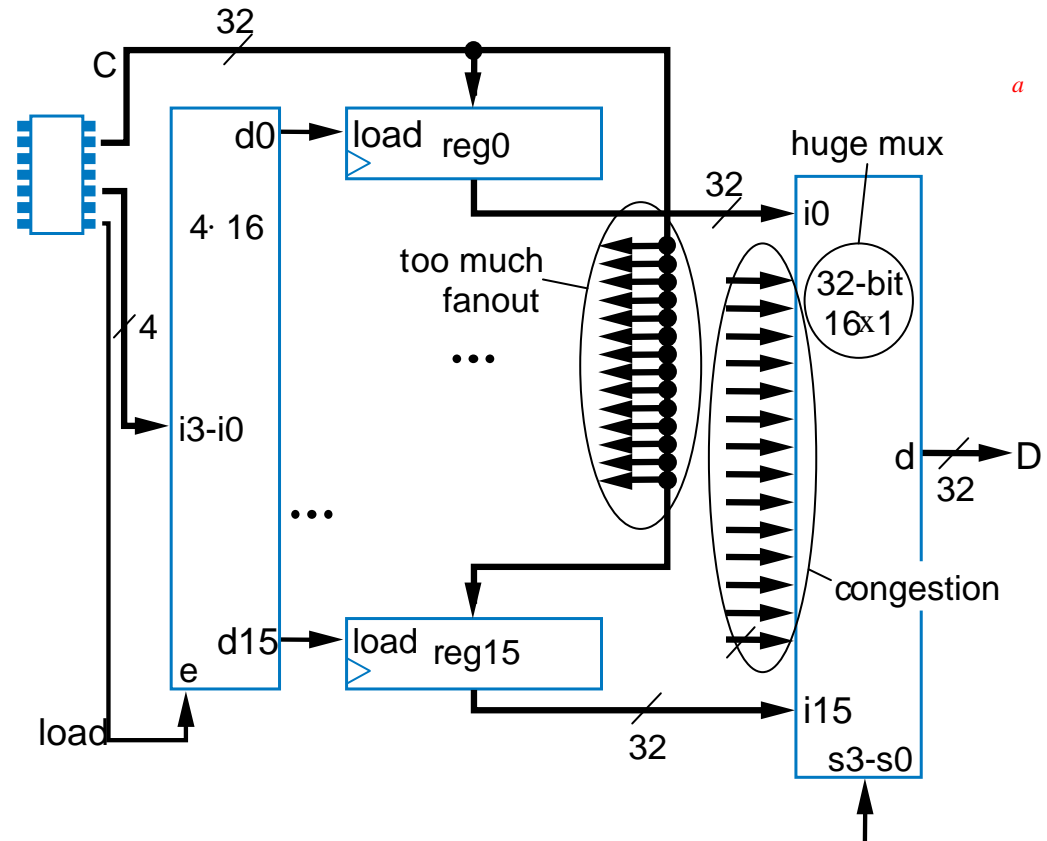


# Register Files

- ***MxN register file***

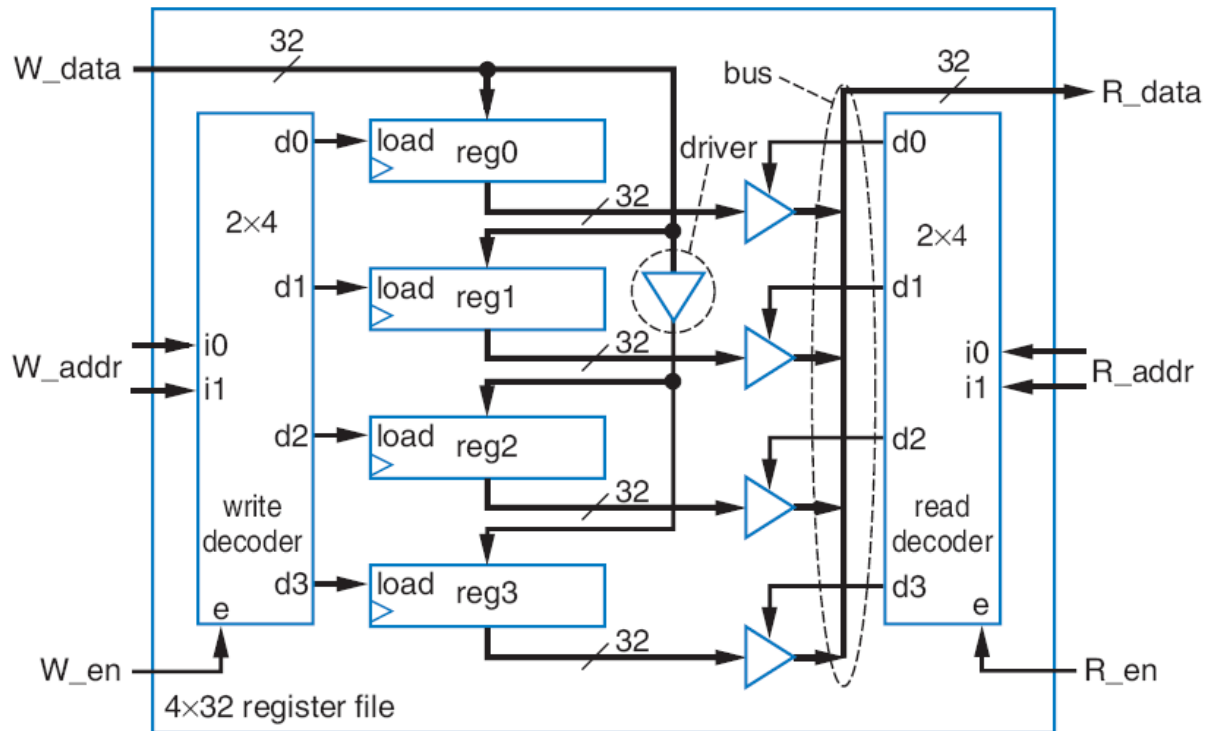
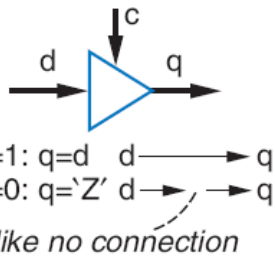
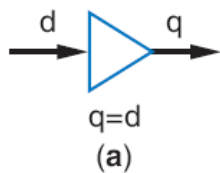
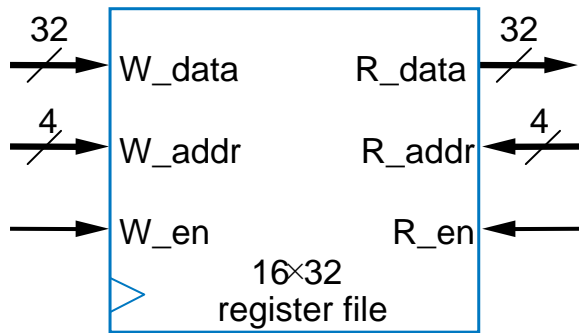
component provides efficient access to  $M$   $N$ -bit-wide registers

- If we have many registers but only need access one or two at a time, a register file is more efficient
- Ex: Above-mirror display (earlier example), but this time having 16 32-bit registers
  - Too many wires, and big mux is too slow



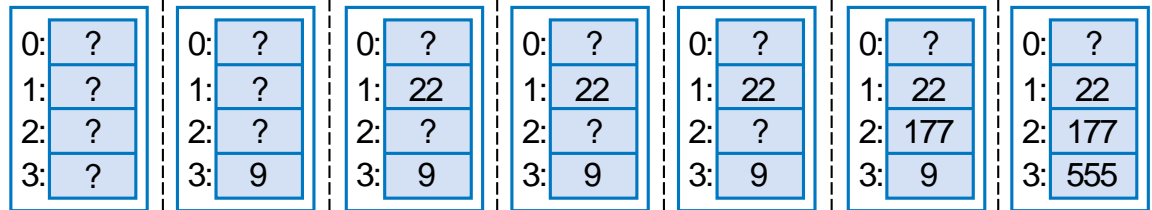
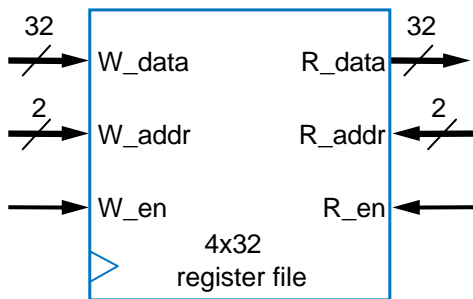
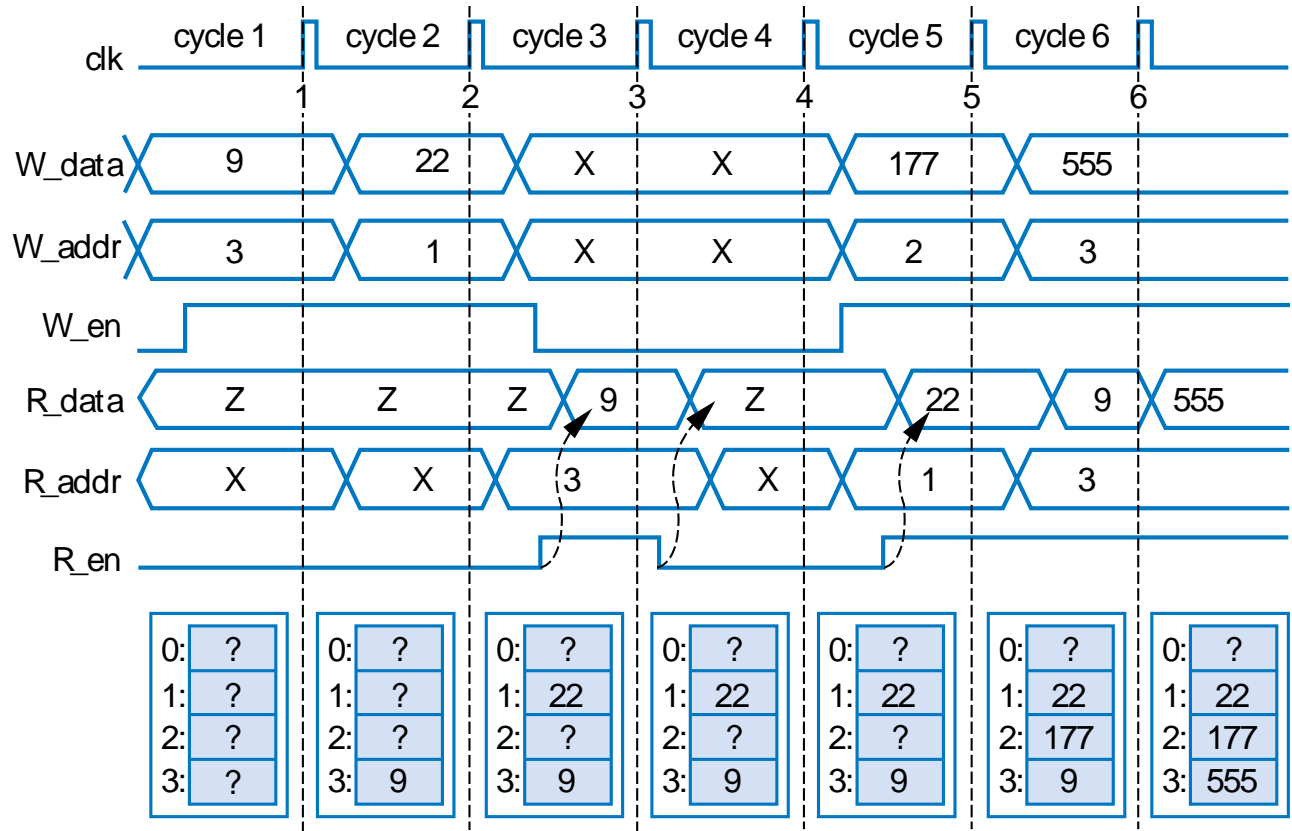
# Register File

- Instead, want component that has one data input and one data output, and allows us to specify which internal register to write and which to read



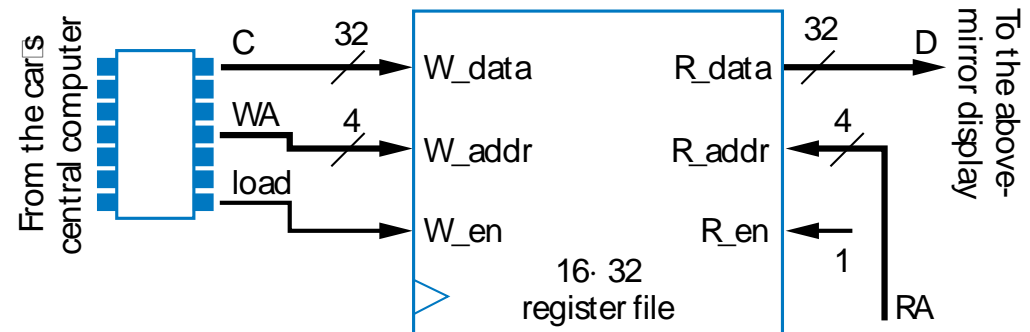
# Register File Timing Diagram

- Can write one register and read one register each clock cycle
  - May be same register



# Register-File Example: Above-Mirror Display

- 16 32-bit registers that can be written by car's computer, and displayed
  - Use 16x32 register file
  - Simple, elegant design
- Register file hides complexity internally
  - And because only one register needs to be written and/or read at a time, internal design is simple



# Chapter Summary

- Need datapath components to store and operate on multibit data
  - Also known as register-transfer-level (RTL) components
- Components introduced
  - Registers
  - Shifters
  - Adders
  - Comparators
  - Counters
  - Multipliers
  - Subtractors
  - Arithmetic-Logic Units
  - Register Files
- Next, we'll combine knowledge of combinational logic design, sequential logic design, and datapath components, to build digital circuits that can perform general and powerful computations

