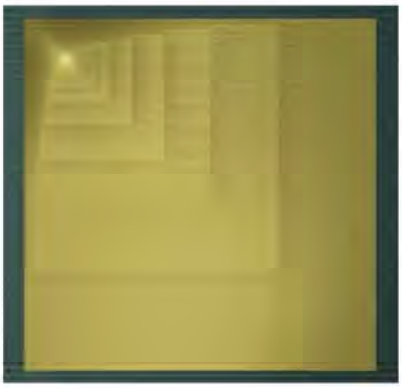


# Digital Design

## Chapter 4b:
## Datapath Components - Sequential

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

# Introduction

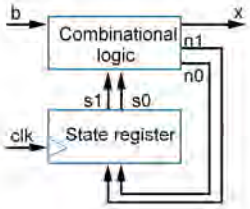
- Chapters 3: Introduced increasingly complex digital building blocks
  - Basic registers, and controllers
- Controllers good for systems with control inputs/outputs
  - ***Control*** input: Single bit (or just a few), representing environment event or state
    - e.g., 1 bit representing button pressed
  - ***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"

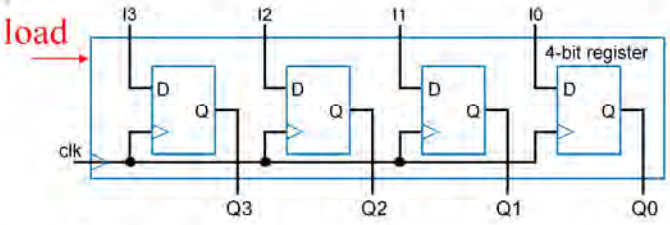


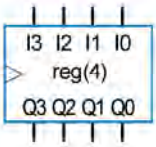
Note: Slides with animation are denoted with a small red "a" near the animated items

# 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



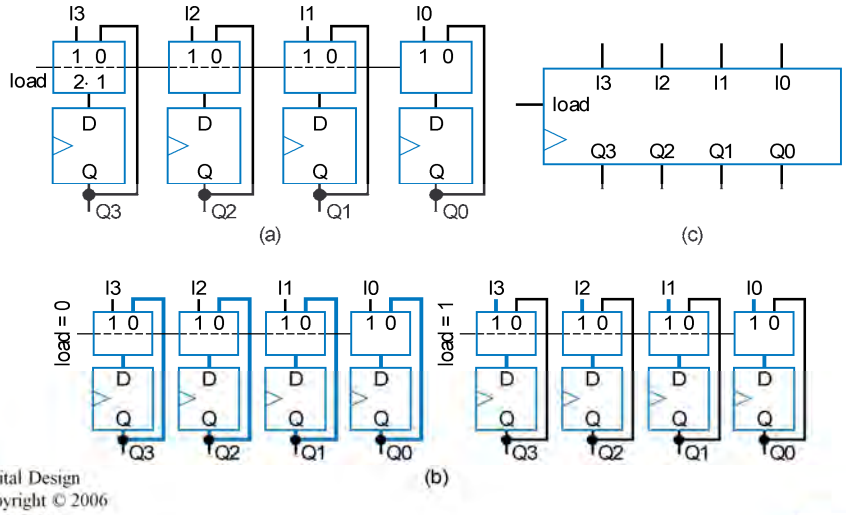




*Basic register loads on every clock cycle*

*How extend to only load on certain cycles?*

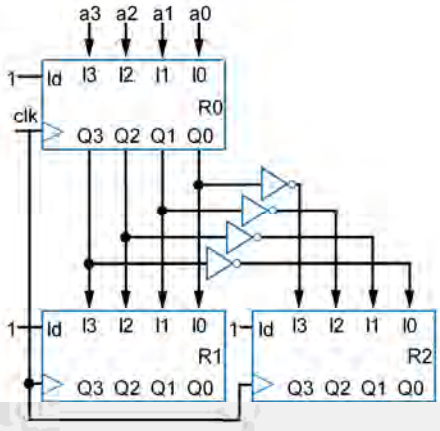
# Register with Parallel Load

- 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

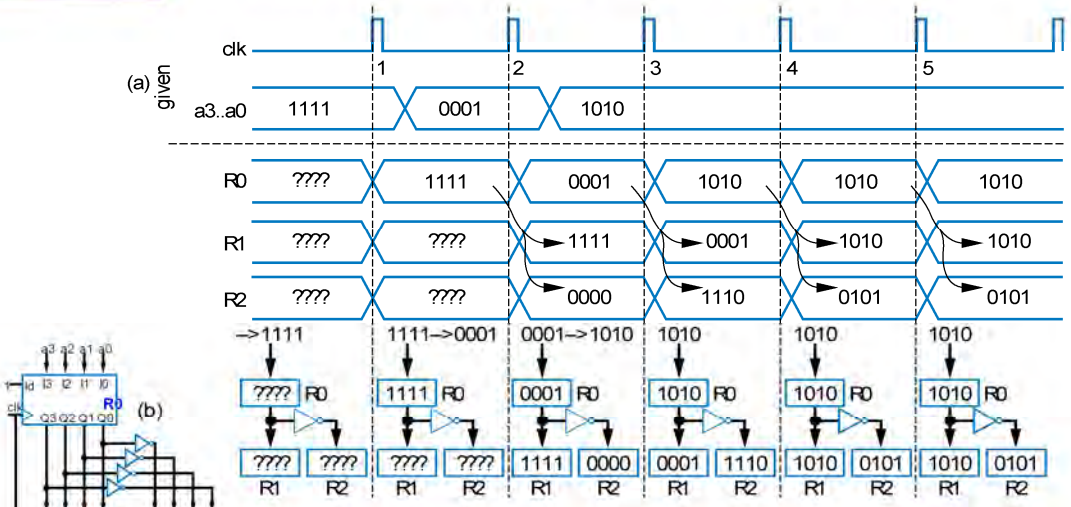


# Basic Example using Registers

- 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

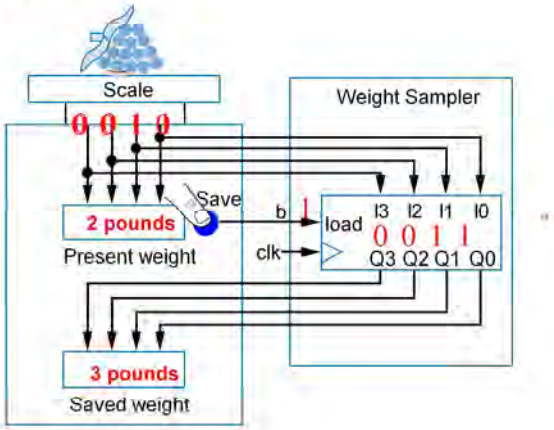


# Basic Example Using Registers (cont.)



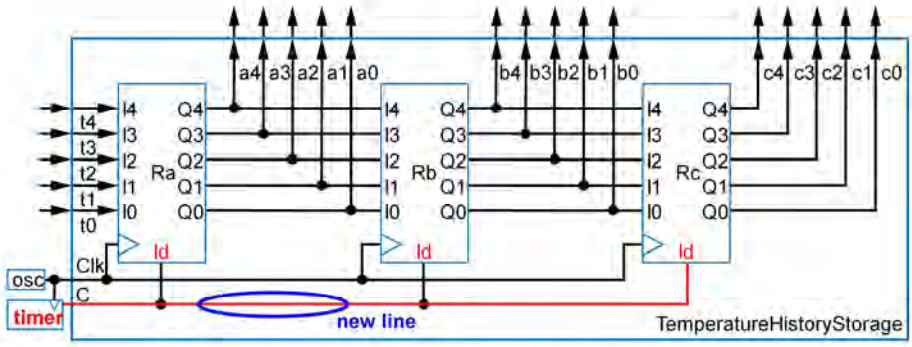
# Register Example using the Load Input: Weight Sampler

- 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

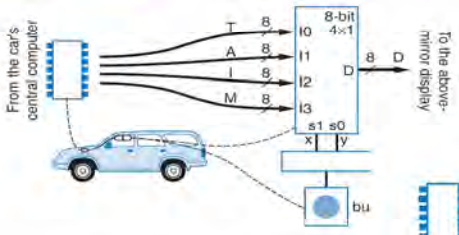


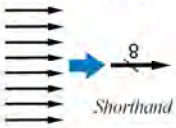
# Register Example: Temperature History Display

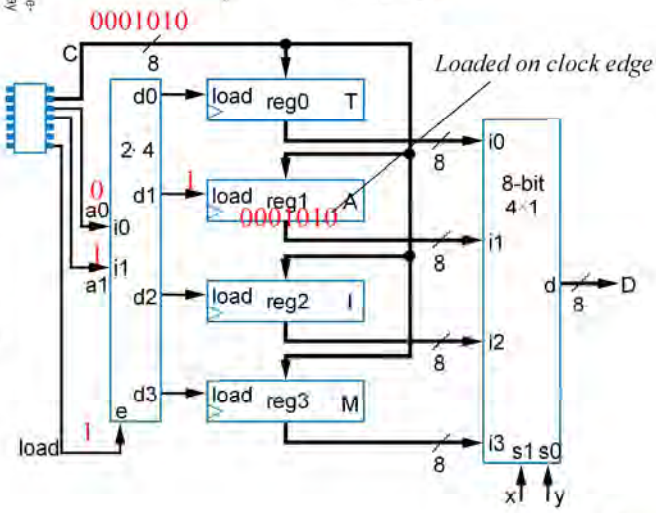
- Recall Chpt 3 example
  - Timer pulse every hour
  - Previously used as clock. Better design only connects oscillator to clock inputs -- use registers with load input, connect to timer pulse.



# Register Example: Above-Mirror Display



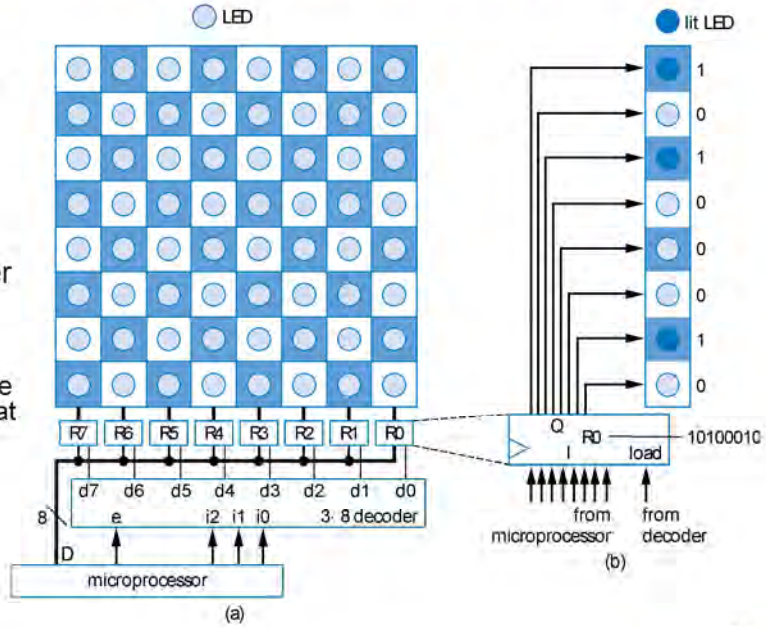

- Ch2 example: Four simultaneous values from car's computer
- To reduce wires: Computer writes only 1 value at a time, loads into one of four registers
  - Was: 8+8+8+8 = 32 wires
  - Now: 8 +2+1 = 11 wires



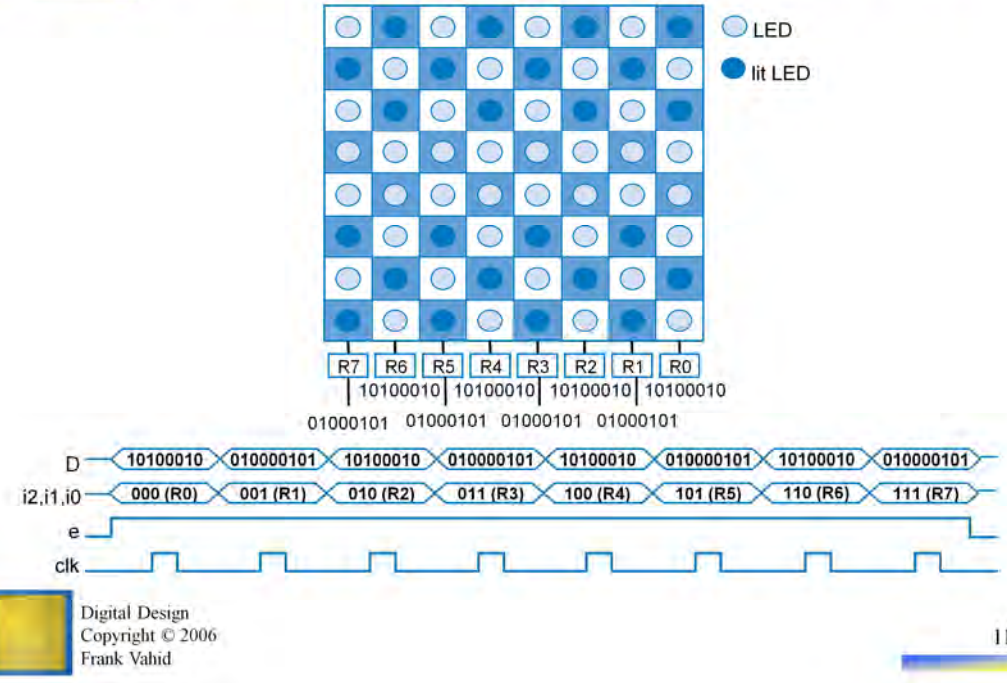
9

# 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

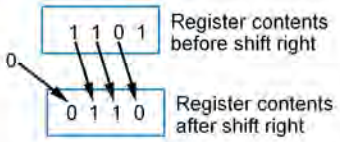


# Register Example: Computerized Checkerboard



# Shift Register

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



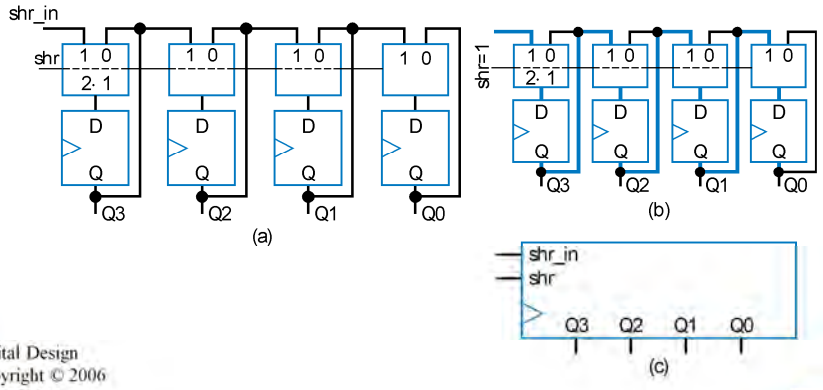

Q: Do four right shifts on 1001, showing value after each shift

A: 1001 (original)
0100
0010
0001
0000

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

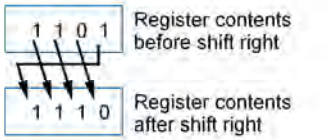
# Shift Register

- 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

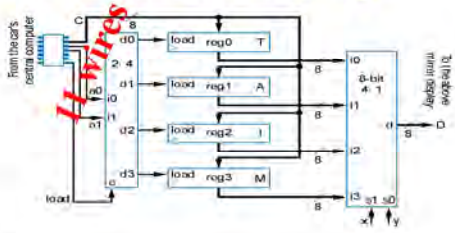
- Rotate right: Like shift right, but leftmost bit comes from rightmost bit
- a.k.a. Barrel Shifter

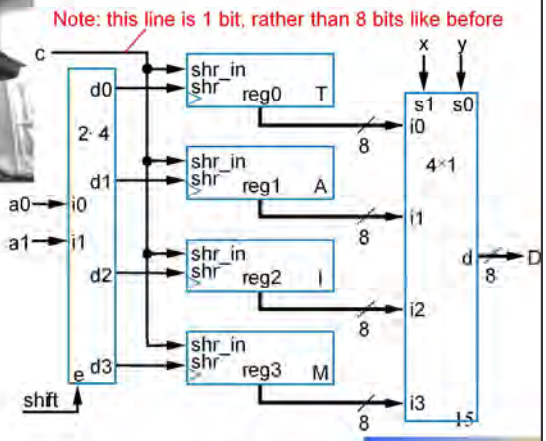



- Shift registers can be made to shift left as well …

# Shift Register Example: Above-Mirror Display

- Earlier example: 8 +2+1 = 11wires 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



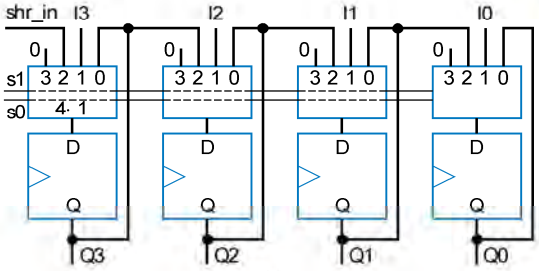



# Multifunction Registers

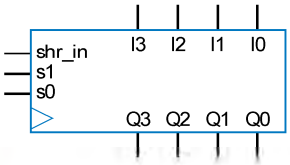
- Many registers have multiple functions
  - Load, shift, clear (load all 0s)
  - And retain present value, of course
- Easily designed using muxes
  - Just connect each mux input to achieve desired function

Functions:

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



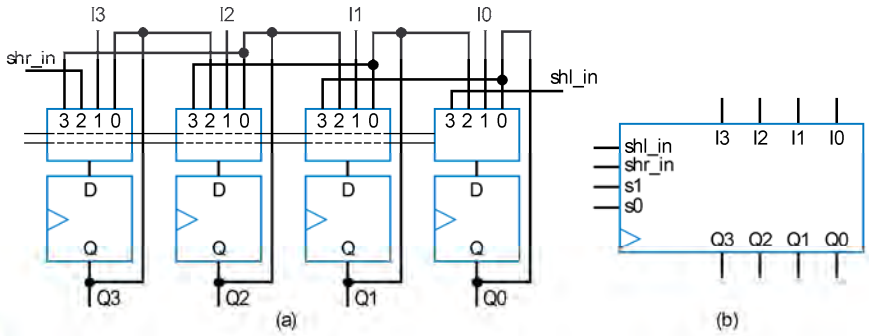

(a)




(b)

# Multifunction Registers

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

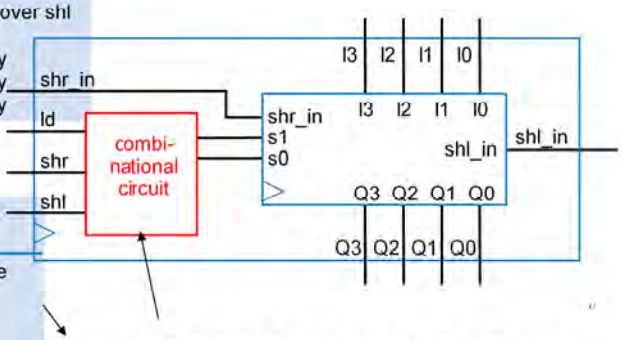


# Multifunction Registers with Separate Control Inputs

| 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 |
|---|---|---|---|---|---|
| ld | shr | shl | s1 | s0 | Operation |
| 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 |
|---|---|---|---|---|---|
| ld | shr | shl | s1 | s0 | Operation |
| 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 | Shift 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.**

| | Step | Description |
|---|---|---|
| 1. | *Determine mux size* | 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. | *Create mux operation table* | Create an operation table defining the desired operation for each possible value of the mux select lines. |
| 3. | *Connect mux inputs* | 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. | *Map control lines* | 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

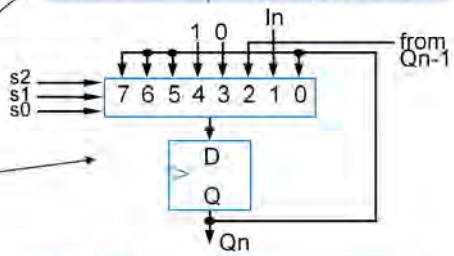
| s2 | s1 | s0 | Operation |
|---|---|---|---|
| 0 | 0 | 0 | Maintain present value |
| 0 | 0 | 1 | Parallel load |
| 0 | 1 | 0 | Shift left |
| 0 | 1 | 1 | Synchronous clear |
| 1 | 0 | 0 | Synchronous set |
| 1 | 0 | 1 | Maintain present value |
| 1 | 1 | 0 | Maintain present value |
| 1 | 1 | 1 |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

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

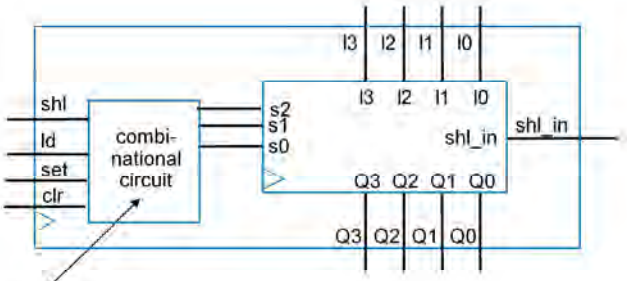
s2 = clr'*set
s1 = clr'*set'*ld'*shl + clr
s0 = clr'*set'*ld + clr

| Inputs | | | | Outputs | | | |
|---|---|---|---|---|---|---|---|
| clr | set | ld | shl | s2 | s1 | s0 | Operation |
| 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 |



21

# Register Design Example




Step 4: Map control lines

s2 = clr'*set
s1 = clr'*set'*ld'*shl + clr
s0 = clr'*set'*ld + clr

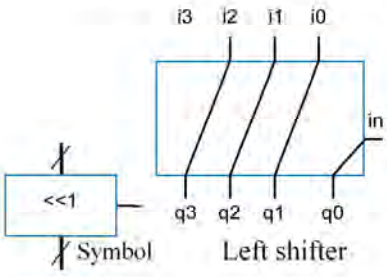
| Inputs | | | | Outputs | | | |
|---|---|---|---|---|---|---|---|
| clr | set | ld | shl | s2 | s1 | s0 | Operation |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | Maintain present value |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | Shift left |
| 0 | 0 | 1 | X | 0 | 0 | 1 | Parallel load |
| 0 | 1 | X | X | 1 | 0 | 0 | Set to all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

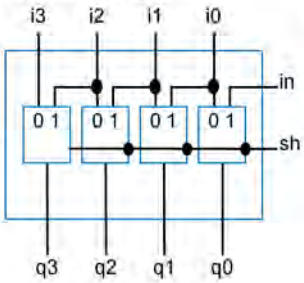


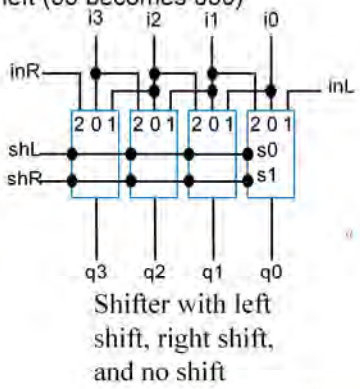
22

# 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 be appending 0, i.e., by shifting left (55 becomes 550)
  - Shift right once same as dividing by 2



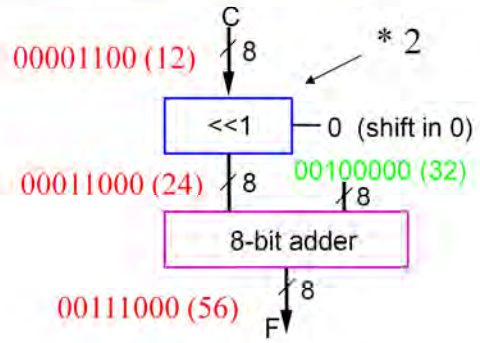






(a)

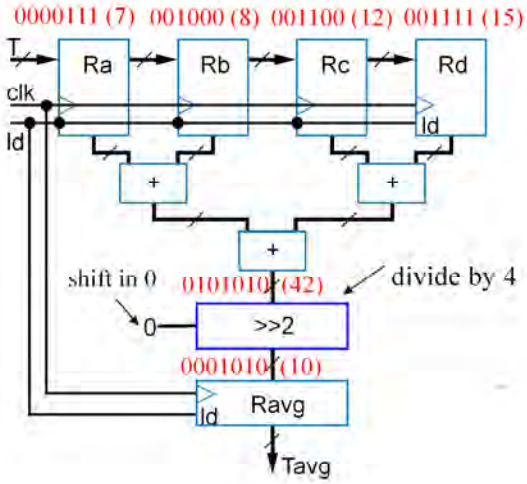
# Shifter Example: Approximate Celsius to Fahrenheit Converter

- Convert 8-bit Celsius input to 8-bit Fahrenheit output
  - F = C * 9/5 + 32
  - Approximate: F = C*2 + 32
  - Use left shift: F = left_shift(C) + 32



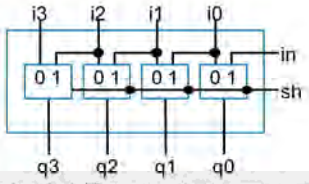
# Shifter Example: Temperature Averager

- 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

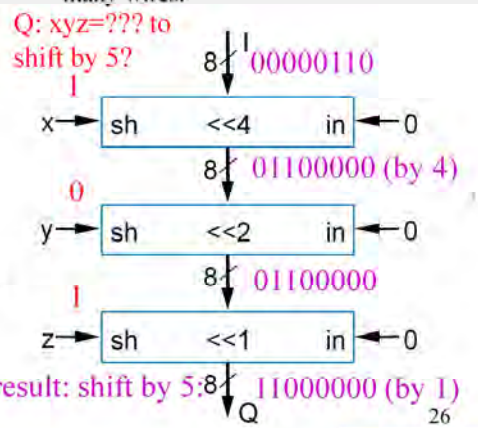


# Barrel Shifter

- A shifter that can shift by any amount
  - 4-bit barrel left shift 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

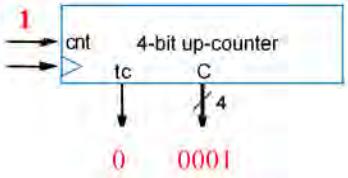


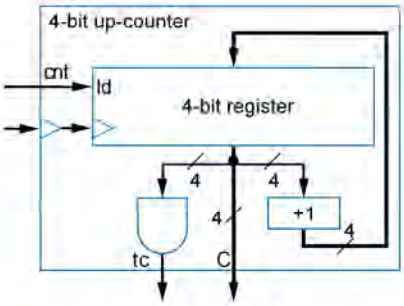

Shift by 1 shifter uses 2x1 muxes. 8x1 mux solution for 8-bit barrel shifter: too many wires.



# Counters

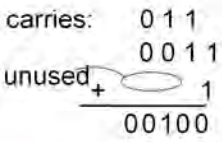
- ***N-bit up-counter***: N-bit register that can increment (add 1) to its own value on each clock cycle
  - 0000, 0001, 0010, 0011, ...., 1110, 1111, 0000
  - Note how count "rolls over" from 1111 to 0000
    - Terminal (last) count, tc, equals1 during value just before rollover
- Internal design
  - Register, incrementer, and N-input AND gate to detect terminal count

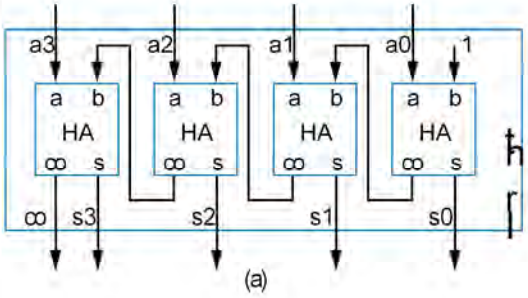


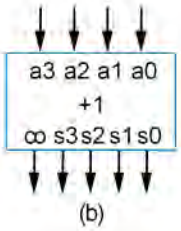



# Incrementer

- 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)



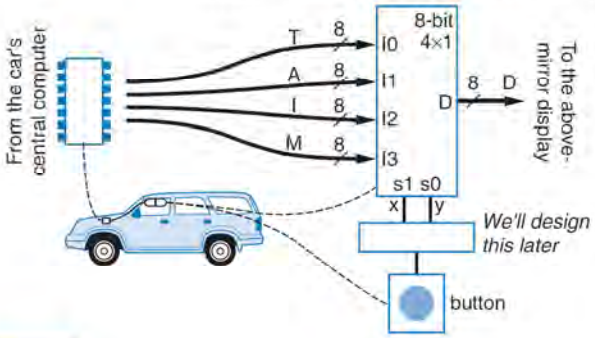
# Incrementer

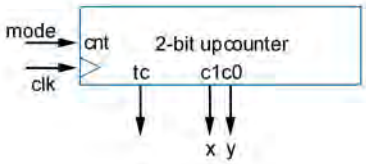
- Can build faster incrementer using combinational logic design process
  - Capture truth table
  - Derive equation for each output
    - c0 = a3a2a1a0
    - ...
    - s0 = a0'
  - Results in small and fast circuit
  - Note: works for small N -- larger N leads to exponential growth, like for N-bit adder

| 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

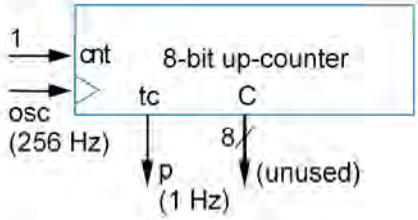
- Recall above-mirror display example from Chapter 2
  - Assumed component that incremented xy input each time button pressed: 00, 01, 10, 11, 00, 01, 10, 11, 00, ...
  - Can use 2-bit up-counter
    - Assumes mode=1 for just one clock cycle during each button press
      - Recall "Button press synchronizer"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



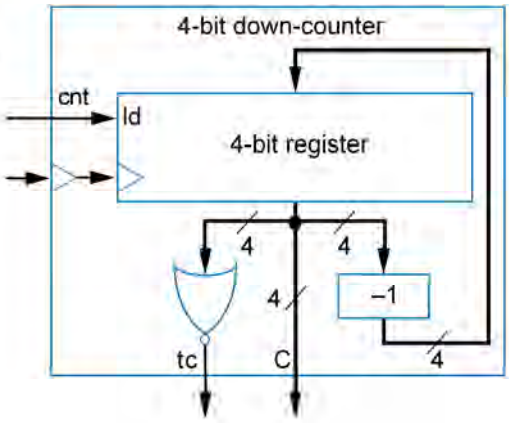
# Counter Example: 1 Hz Pulse Generator Using 256 Hz Oscillator

- 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



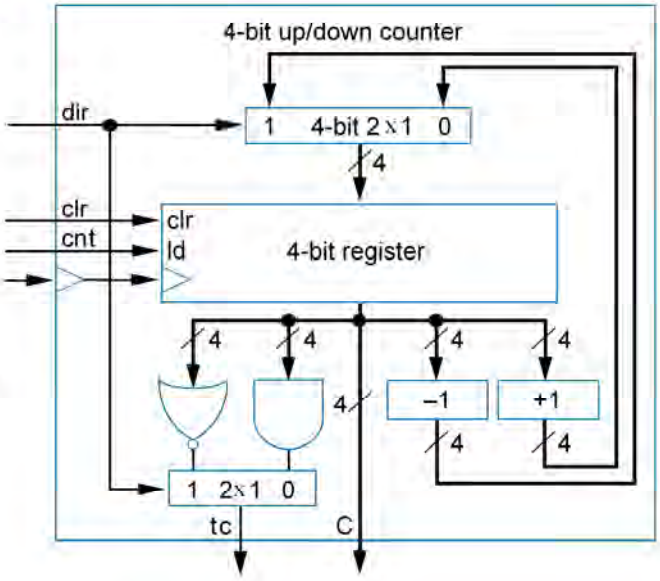
# Down-Counter

- 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



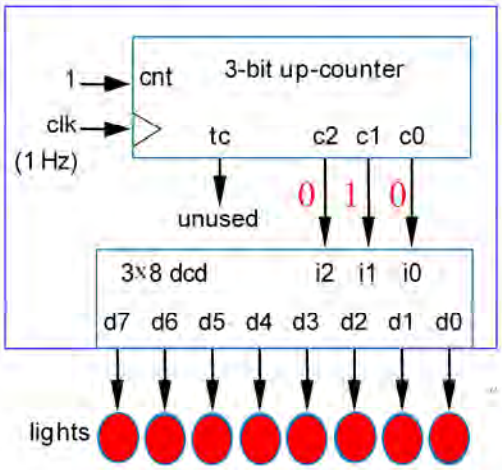
# Up/Down-Counter

- 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



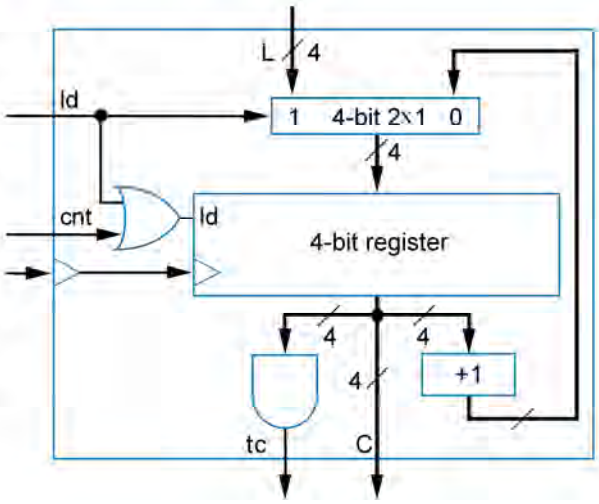
# Counter Example: Light Sequencer

- 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?



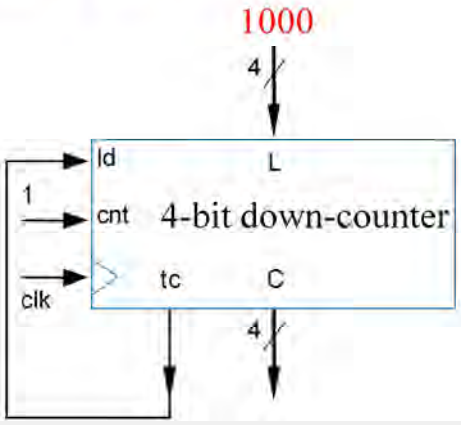
# Counter with Parallel Load

- 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



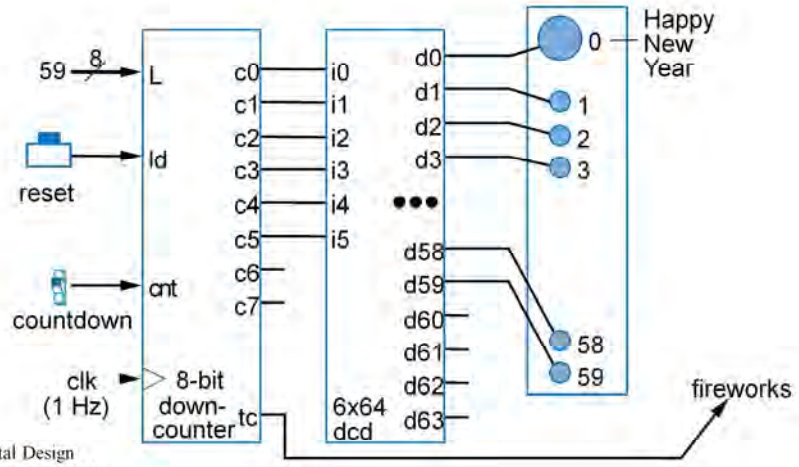
# Counter with Parallel Load

- 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



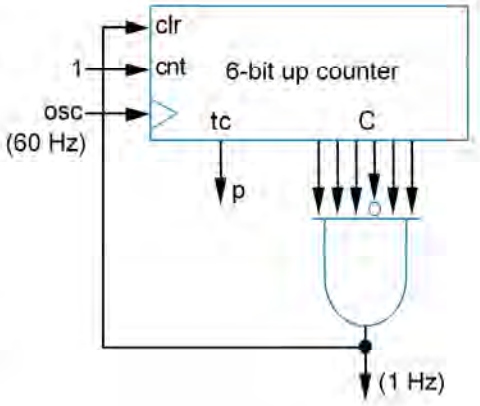
# Counter Example:
## New Year's Eve Countdown Display

- 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



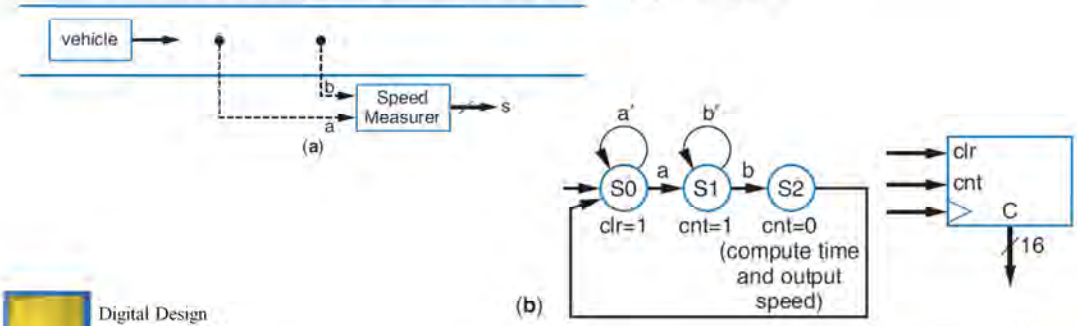
# Counter Example:
## 1 Hz Pulse Generator from 60 Hz Clock

- 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)



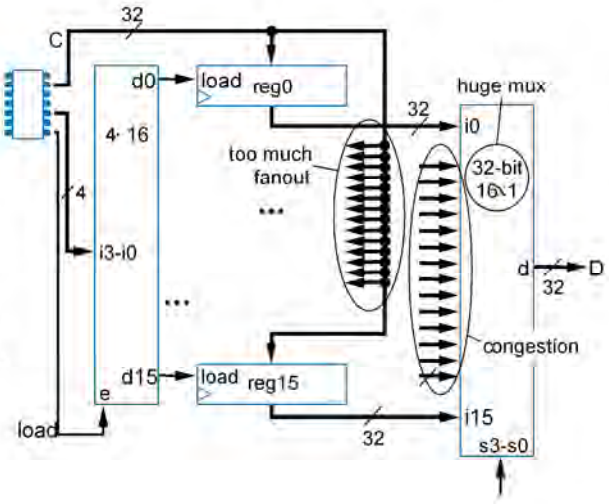

# Timer

- 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



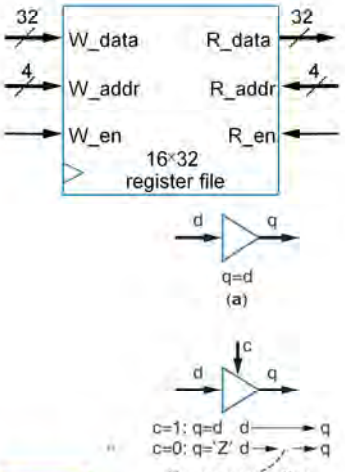
# Register Files

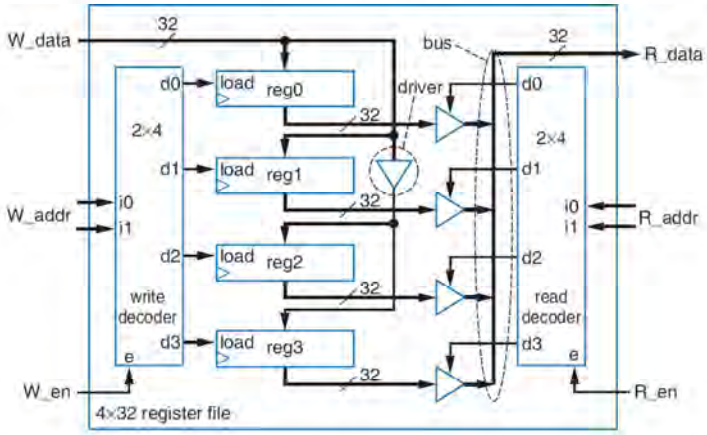
- ***MxN register file*** component provides efficient access to M N-bit-wide registers
  - If we have many registers but only need access one or two at a time, a register file is more efficient
  - Ex: Above-mirror display (earlier example), but this time having 16 32-bit registers
    - Too many wires, 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



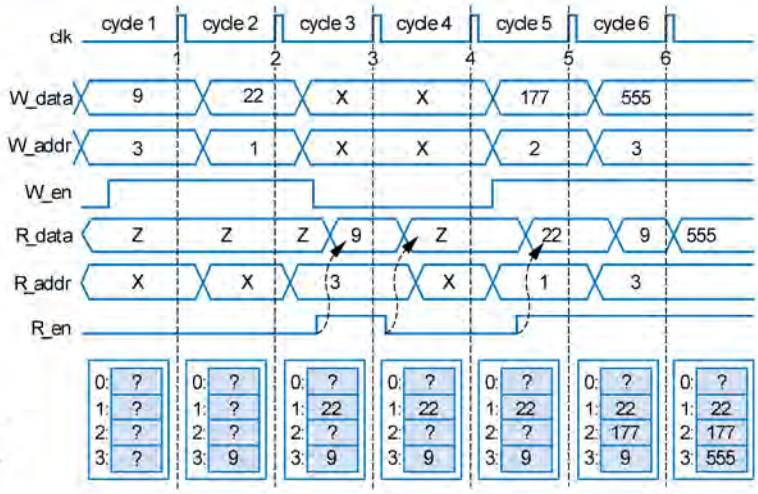



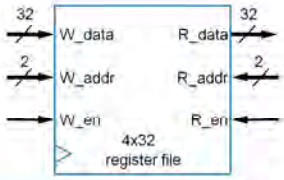



41

# 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