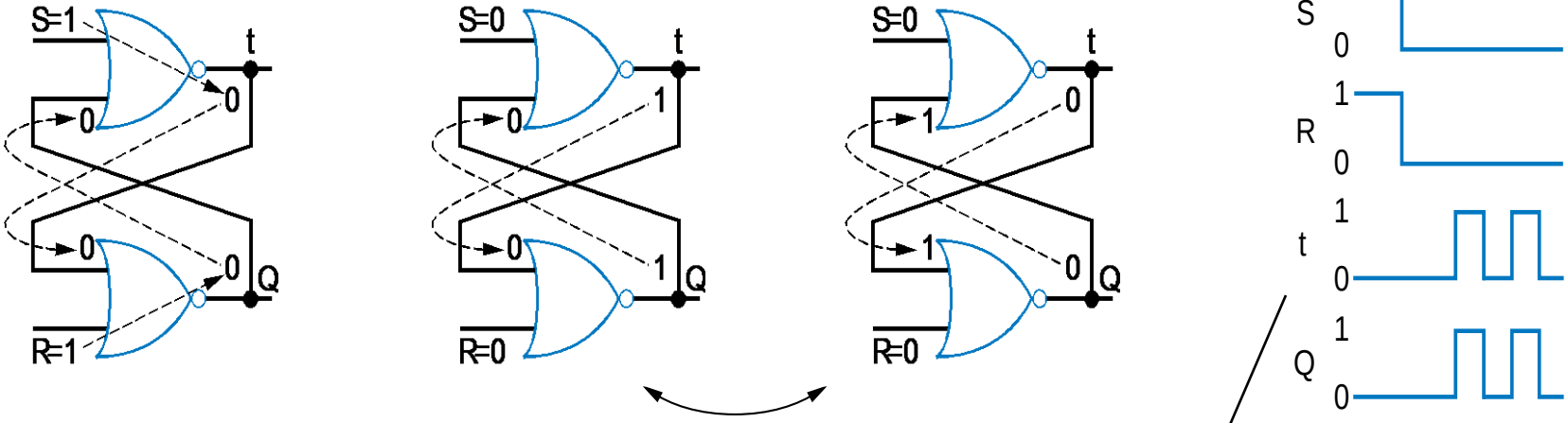
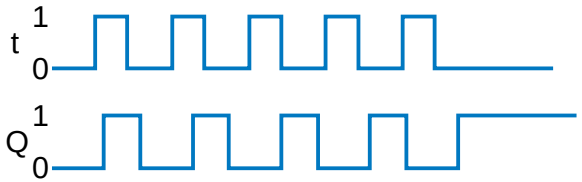
Sequential Logic Design

Problem with SR Latch

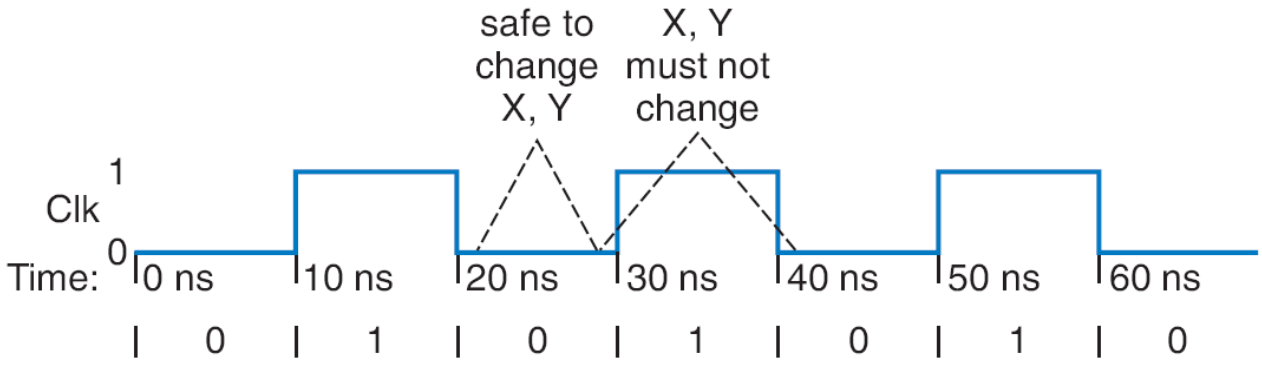
- Problem
  - If S=1 and R=1 simultaneously, we don't know what value Q will take



Q may oscillate. Then, because one path will be slightly longer than the other, Q will eventually settle to 1 or 0 – but we don't know which.



Clocks

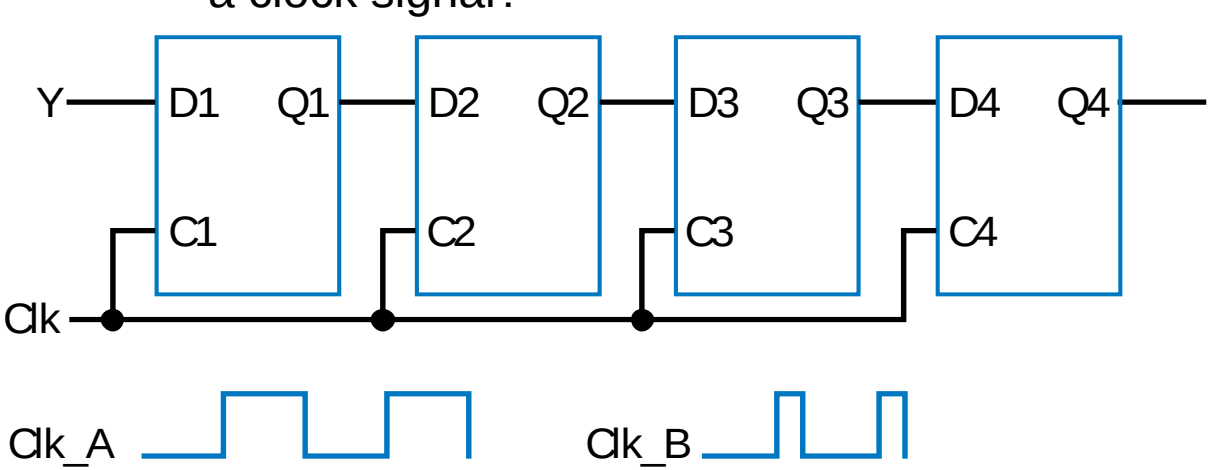


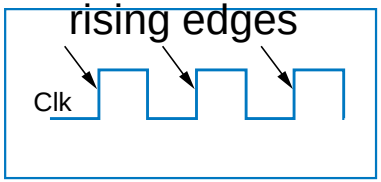
- ***Clock period***: time interval between pulses
  - Above signal: period = 20 ns
- ***Clock cycle***: one such time interval
  - Above signal shows 3.5 clock cycles
- ***Clock frequency***: 1/period
  - Above signal: frequency = 1 / 20 ns = 50 MHz
    - 1 Hz = 1/s

Freq	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	1 ns
100 MHz	10 ns
10 MHz	100 ns

Problem with Level-Sensitive D Latch

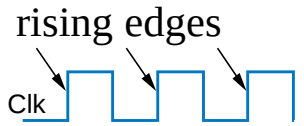
- D latch still has problem (as does SR latch)
  - When C=1, through how many latches will a signal travel?
  - Depends on for how long C=1
    - Clk_A -- signal may travel through multiple latches
    - Clk_B -- signal may travel through fewer latches
  - Hard to pick C that is just the right length
    - Can we design bit storage that only stores a value on the rising edge of a clock signal?



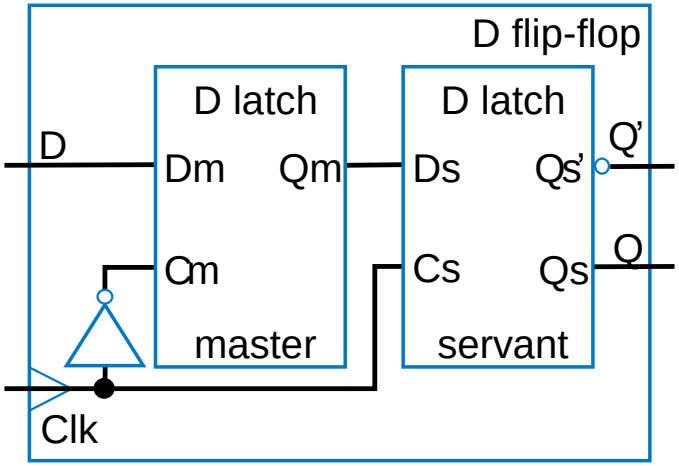


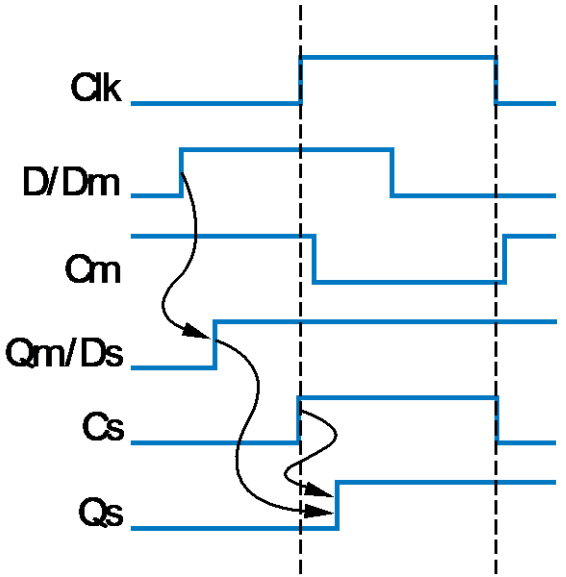
D Flip-Flop

- ***Flip-flop***: Bit storage that stores on clock edge, not level
- One design -- master-servant
  - Two latches, output of first goes to input of second, master latch has inverted clock signal
  - So master loaded when C=0, then servant when C=1
  - When C changes from 0 to 1, master disabled, servant loaded with value that was at D just before C changed -- i.e., value at D during rising edge of C



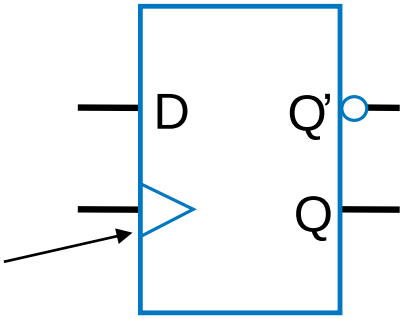
Note: Hundreds of different flip-flop designs exist



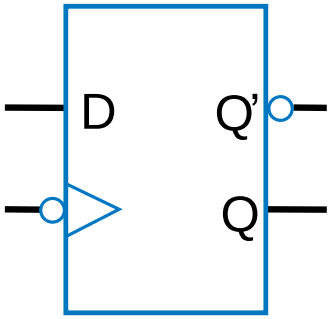


D Flip-Flop

The triangle means clock input, edge triggered

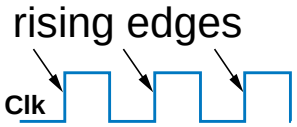


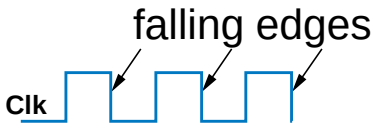
Symbol for rising-edge triggered D flip-flop



Symbol for falling-edge triggered D flip-flop

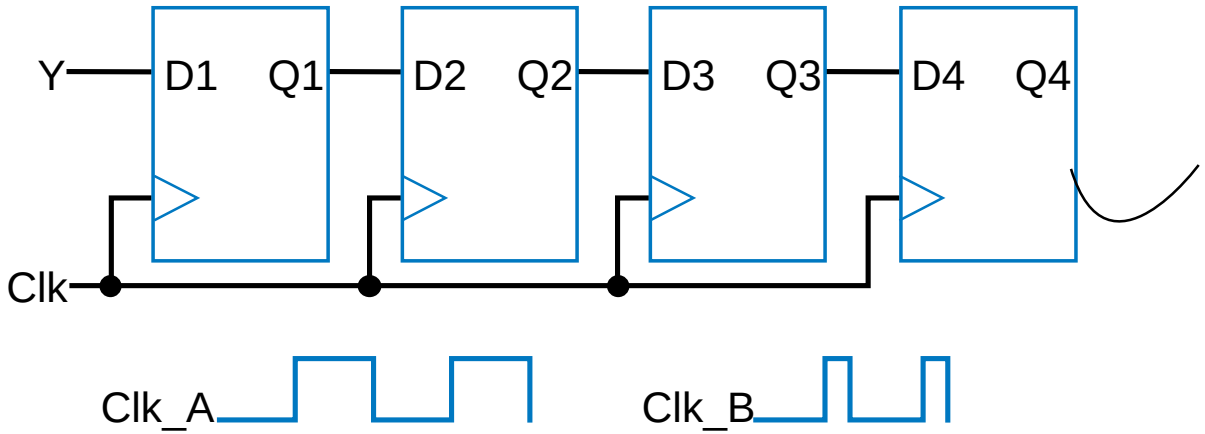
Internal design: Just invert servant clock rather than master





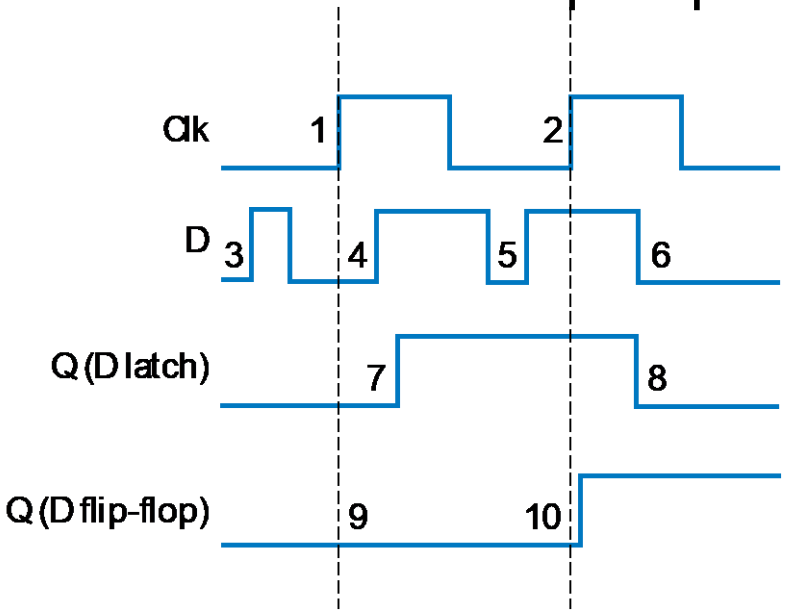
D Flip-Flop

- Solves problem of not knowing through how many latches a signal travels when C=1
  - In figure below, signal travels through exactly one flip-flop, for Clk_A or Clk_B
  - Why? Because on rising edge of Clk, all four flip-flops are loaded simultaneously -- then all four no longer pay attention to their input, until the next rising edge. Doesn't matter how long Clk is 1.

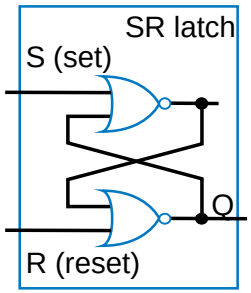


D Latch vs. D Flip-Flop

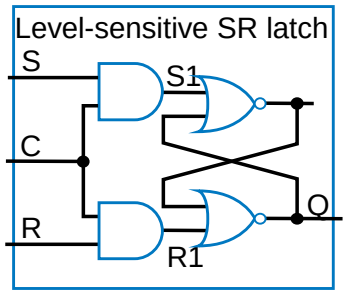
- Latch is level-sensitive: Stores D when C=1
- Flip-flop is edge triggered: Stores D when C changes from 0 to 1
  - Saying “level-sensitive latch,” or “edge-triggered flip-flop,” is redundant
  - Two types of flip-flops -- rising or falling edge triggered.
- Comparing behavior of latch and flip-flop:



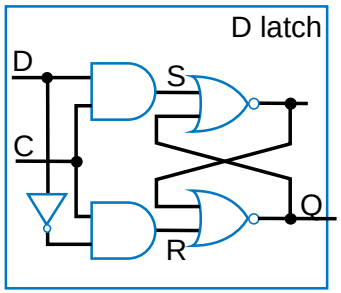
Bit Storage Summary



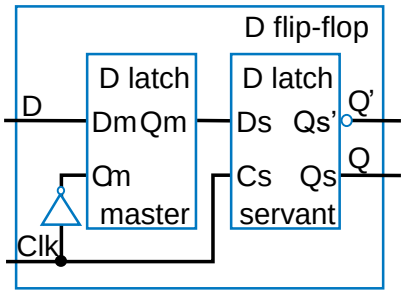
Feature: S=1 sets Q to 1, R=1 resets Q to 0. Problem: SR=11 yield undefined Q.



Feature: S and R only have effect when C=1. We can design outside circuit so SR=11 never happens when C=1. Problem: avoiding SR=11 can be a burden.



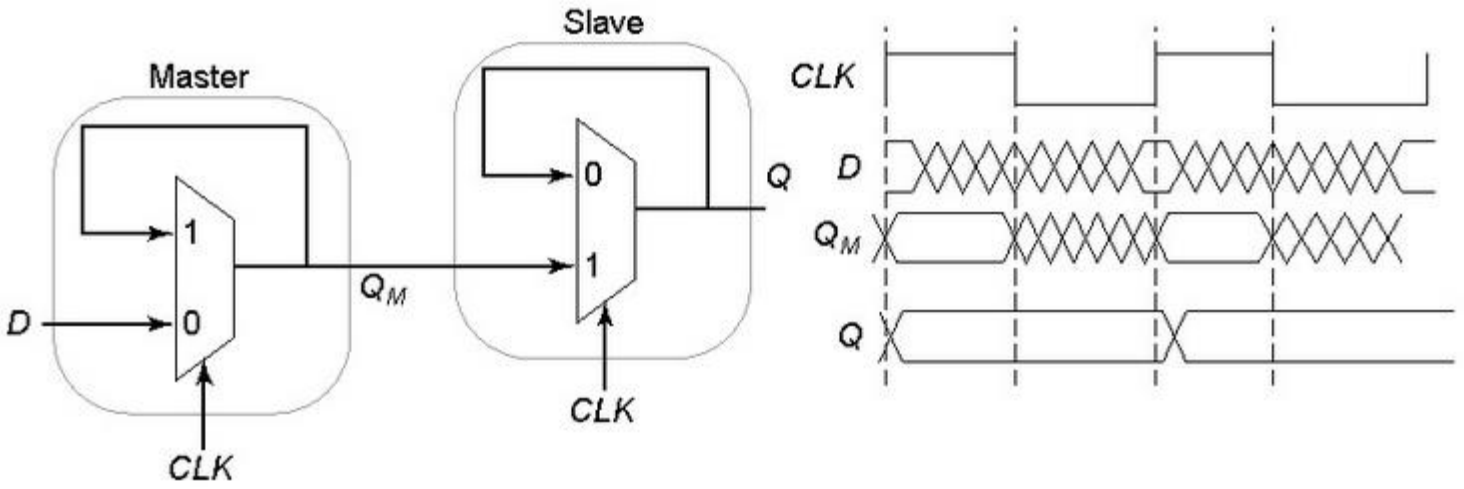
Feature: SR can't be 11 if D is stable before and while C=1, and will be 11 for only a brief glitch even if D changes while C=1. Problem: C=1 too long propagates new values through too many latches: too short may not enable a store.



Feature: Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle. Tradeoff: uses more gates internally than D latch, and requires more external gates than SR – but gate count is less of an issue today.

- We considered increasingly better bit storage until we arrived at the robust D flip-flop bit storage

Postive edge triggered registers based on multiplexers

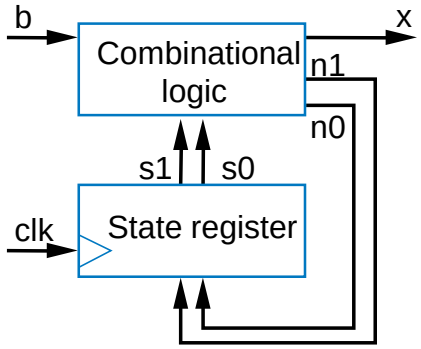


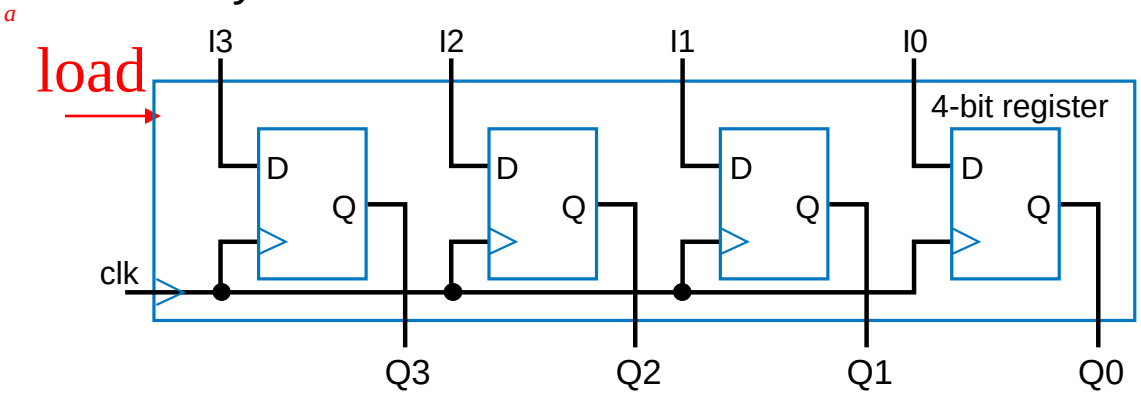
Introduction

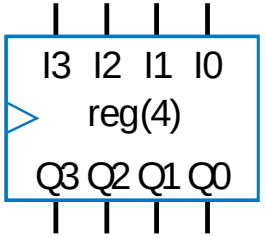
- Introduced increasingly complex digital building blocks
  - Gates, multiplexors, decoders, basic registers, and controllers
- Controllers good for systems with control inputs/outputs
  - ***Control*** input: Single bit (or just a few), representing environment event or state
    - e.g., 1 bit representing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

Registers

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





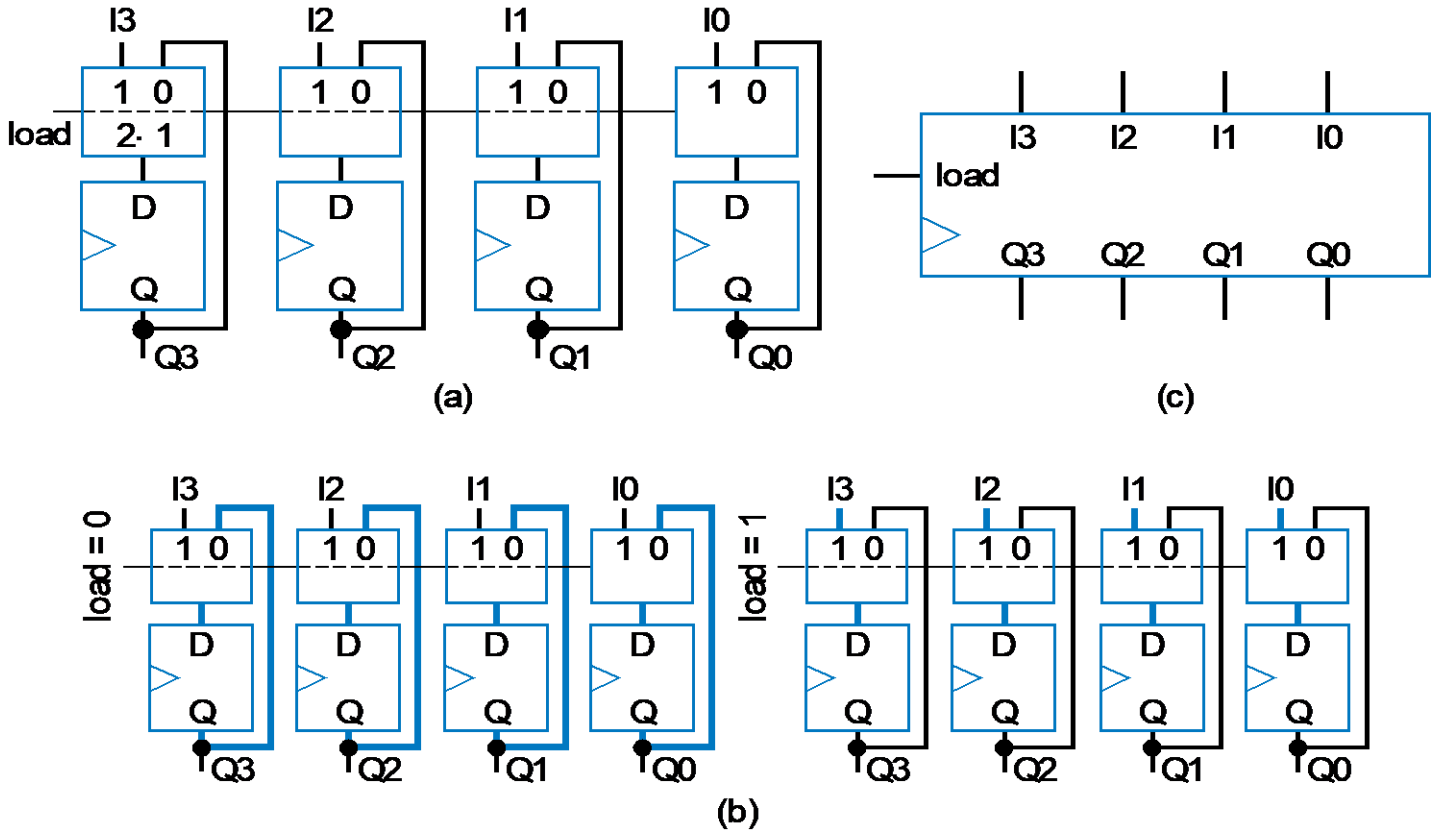


Basic register loads on every clock cycle

How extend to only load on certain cycles?

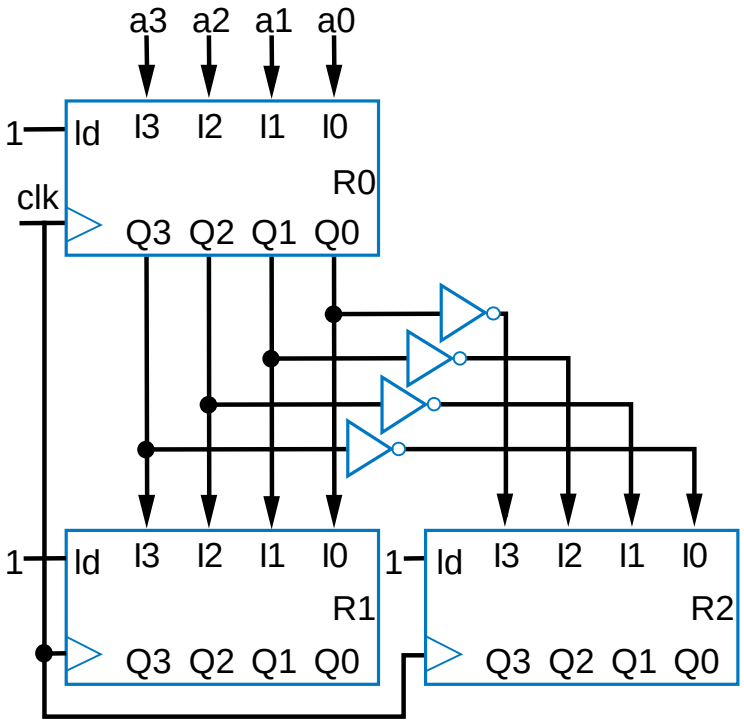
Register with Parallel Load

- 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

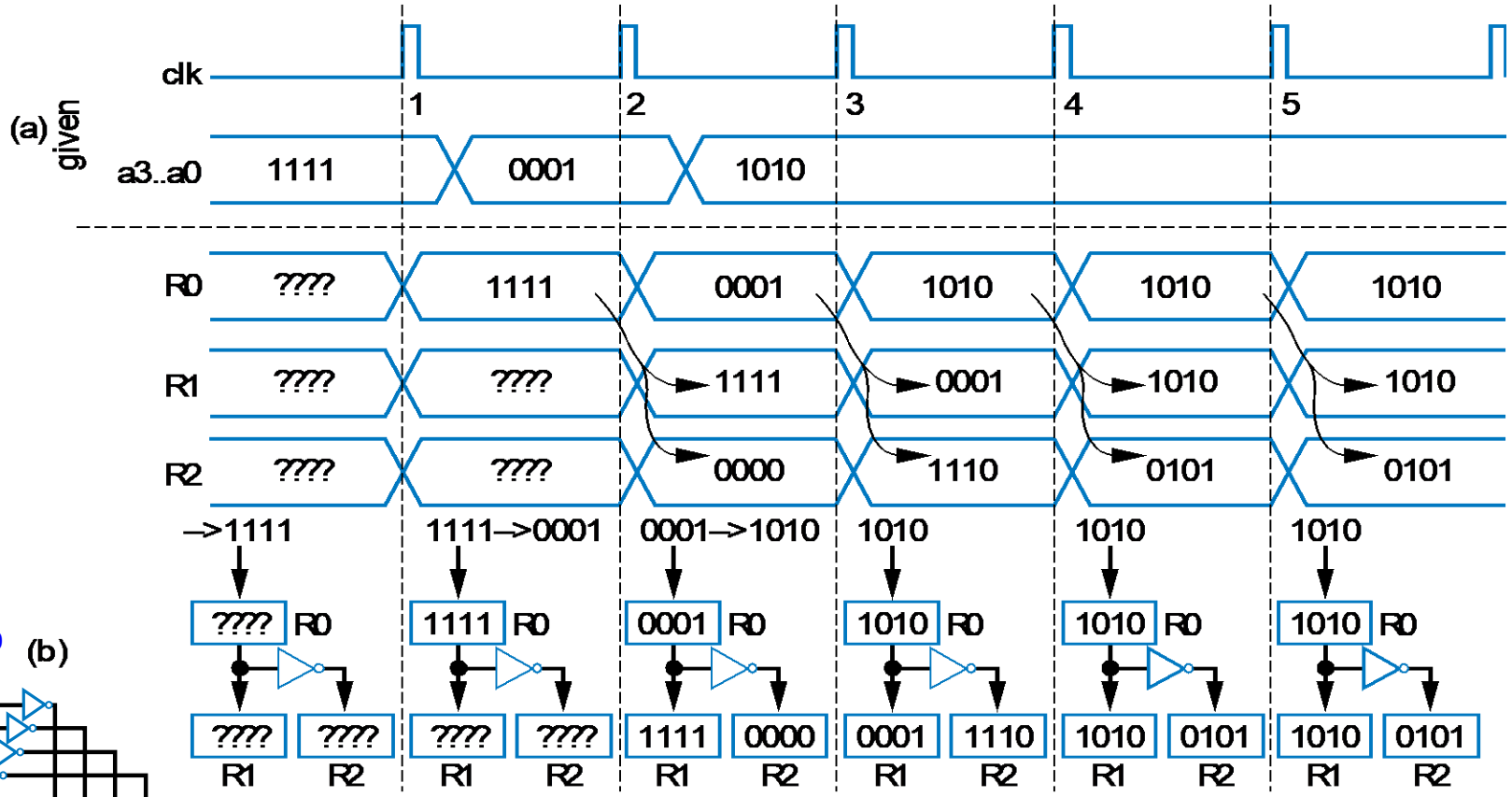


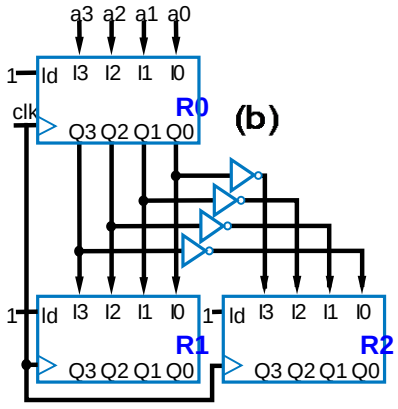
Basic Example Using Registers

- 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



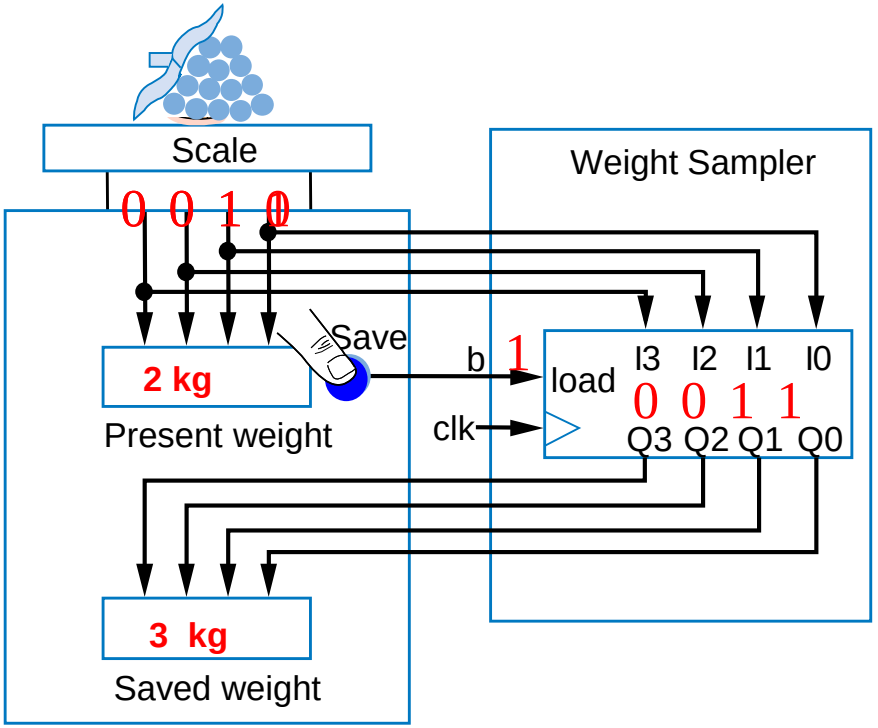
Basic Example Using Registers





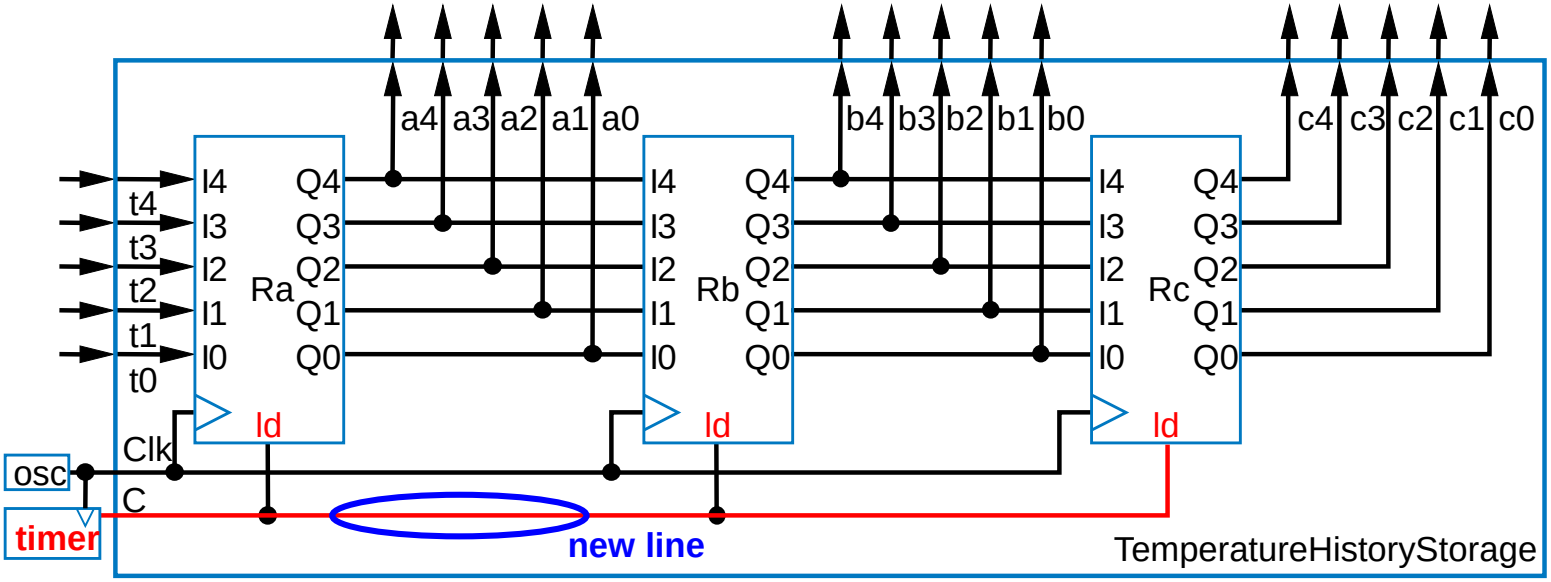
Register Example using the Load Input: Weight Sampler

- 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



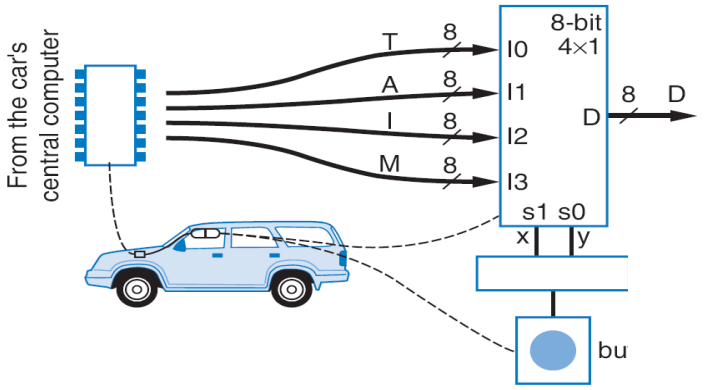
Register Example: Temperature History Display

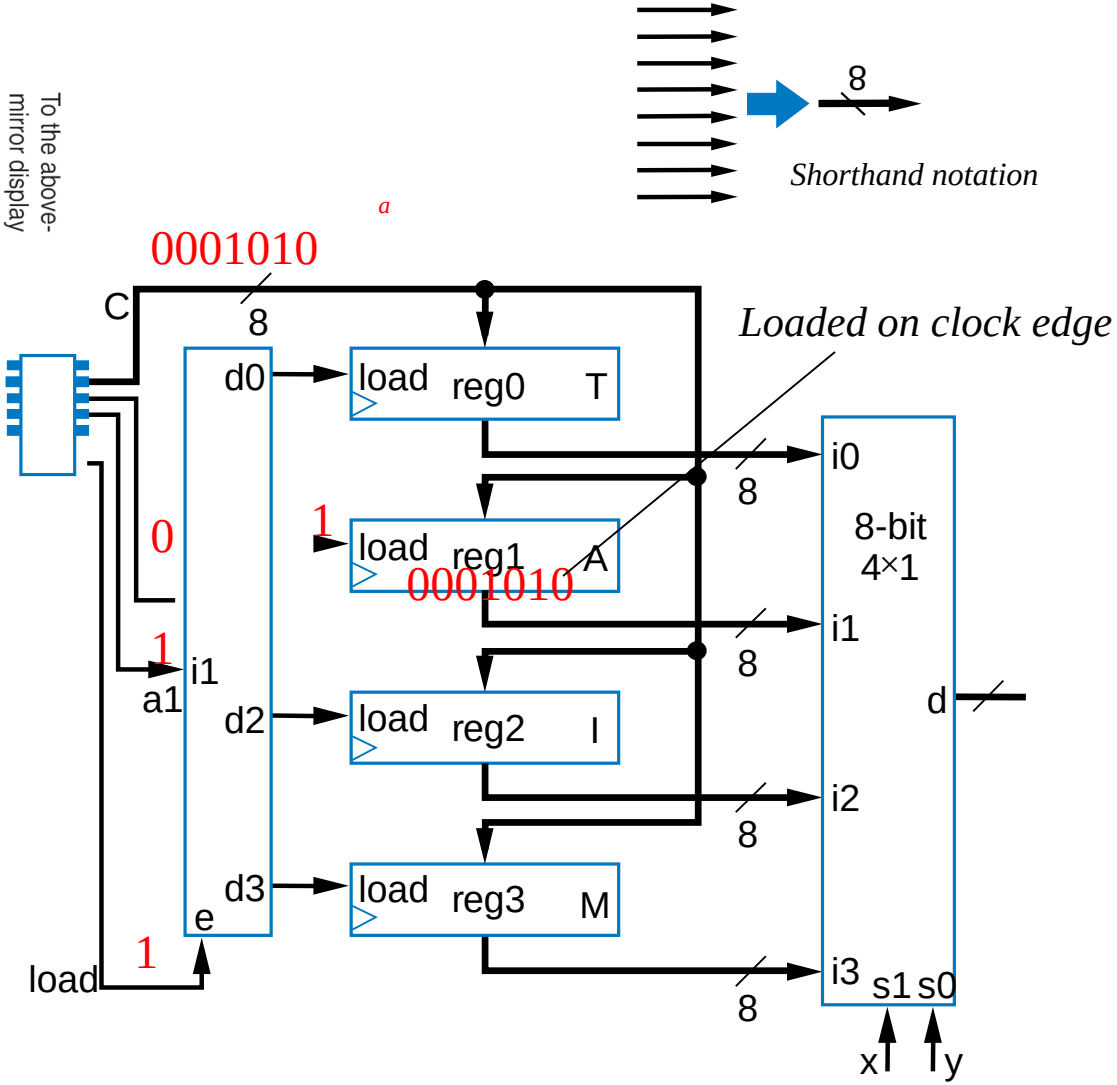
- Recall Chpt 3 example
  - Timer pulse every hour
  - Previously used as clock. Better design only connects oscillator to clock inputs -- use registers with load input, connect to timer pulse.



a

Register Example: Above-Mirror Display

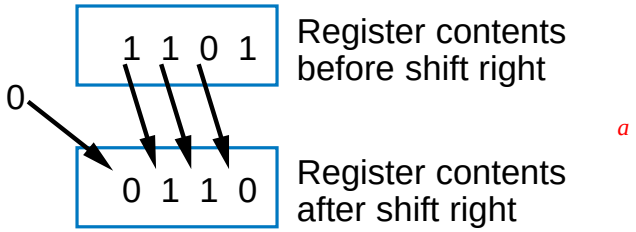




- example: Four simultaneous values from car's computer
- To reduce wires: Computer writes only 1 value at a time, loads into one of four registers
  - Was: 8+8+8+8 = 32 wires
  - Now: 8 +2+1 = 11 wires

Shift Register

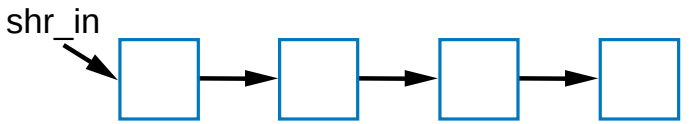
- Shift right
  - Move each bit one position right
  - Shift in 0 to leftmost bit



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

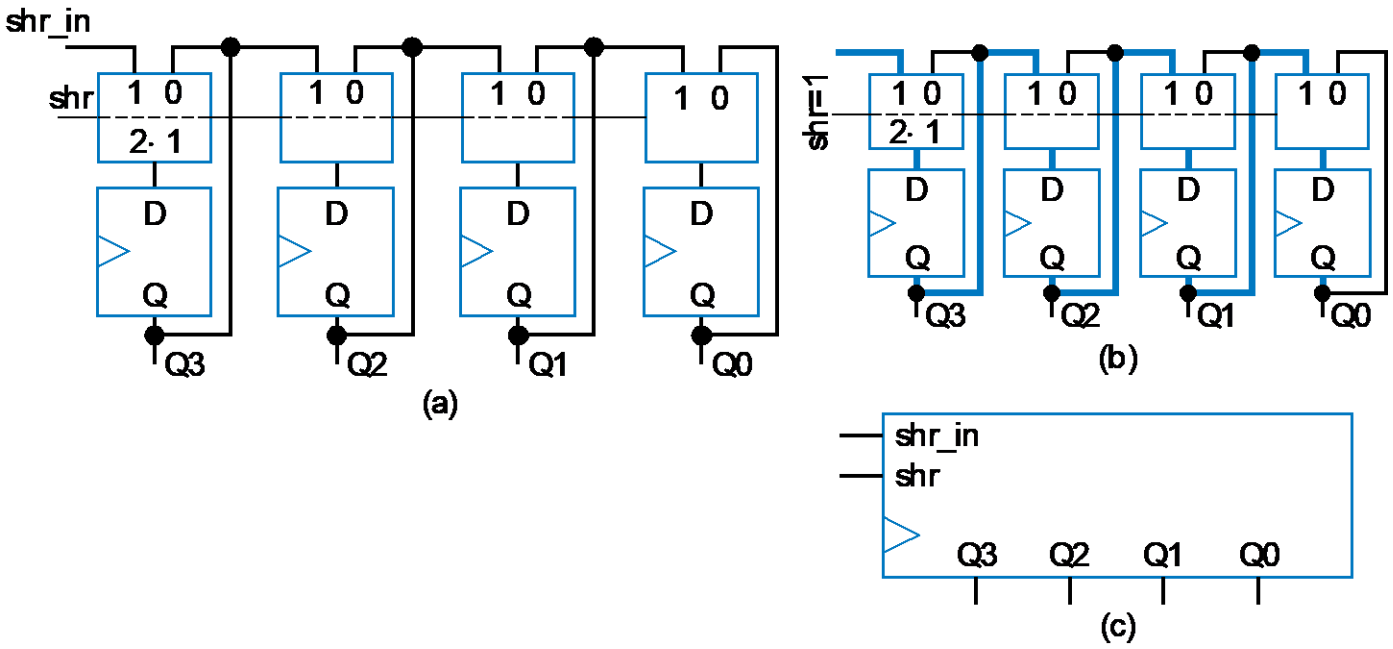
A: 1001 (original)
0100
0010
0001
0000

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



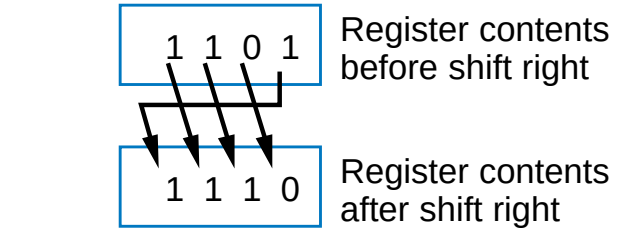
Shift Register

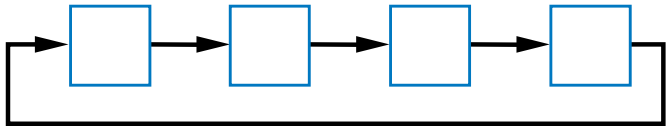
- To allow register to either shift or retain, use 2x1 muxes
  - shr: 0 means retain, 1 shift
  - shr_in: value to shift in
    - May be 0, or 1
- Note: Can easily design shift register that shifts left instead



Rotate Register

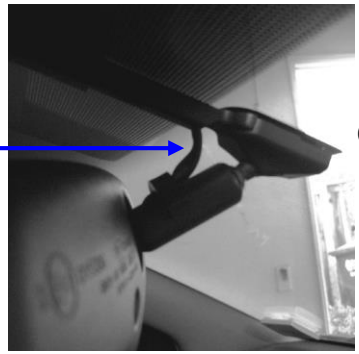
- Rotate right: Like shift right, but leftmost bit comes from rightmost bit

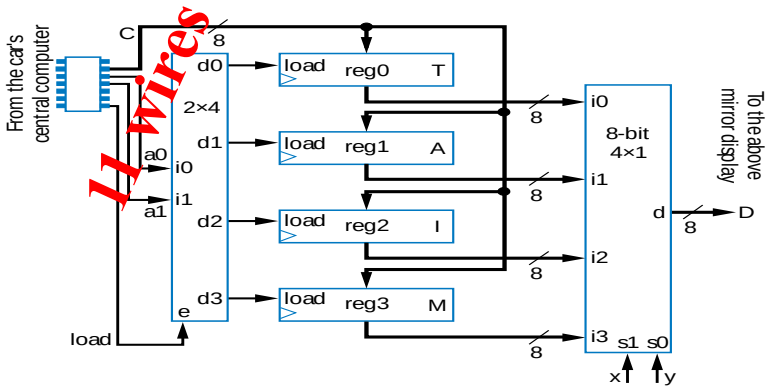


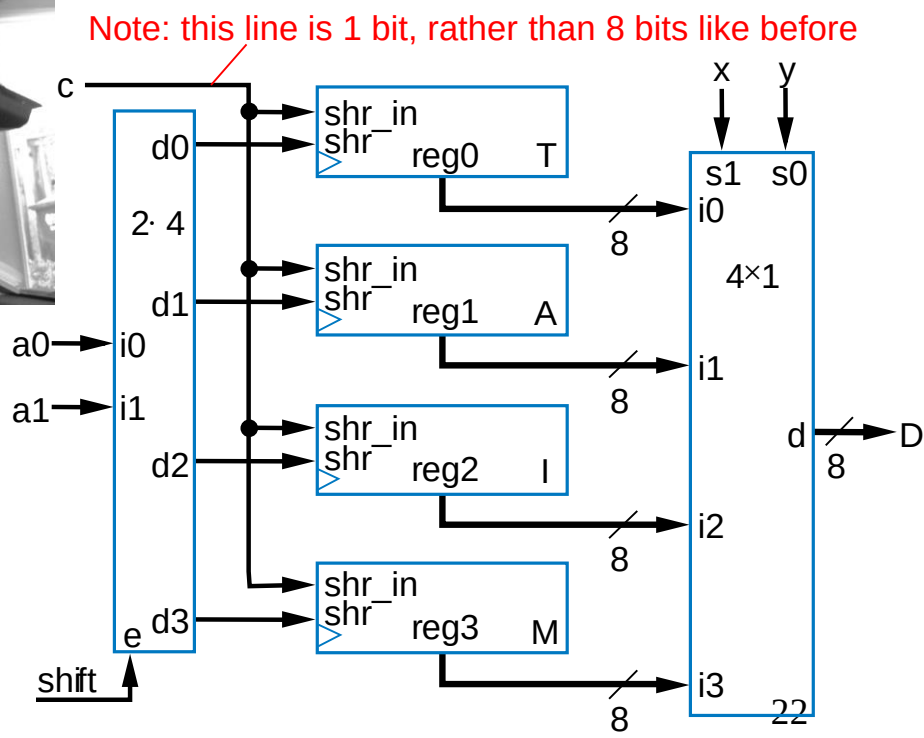


Shift Register Example: Above-Mirror Display

- Earlier example: 8 +2+1 = 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





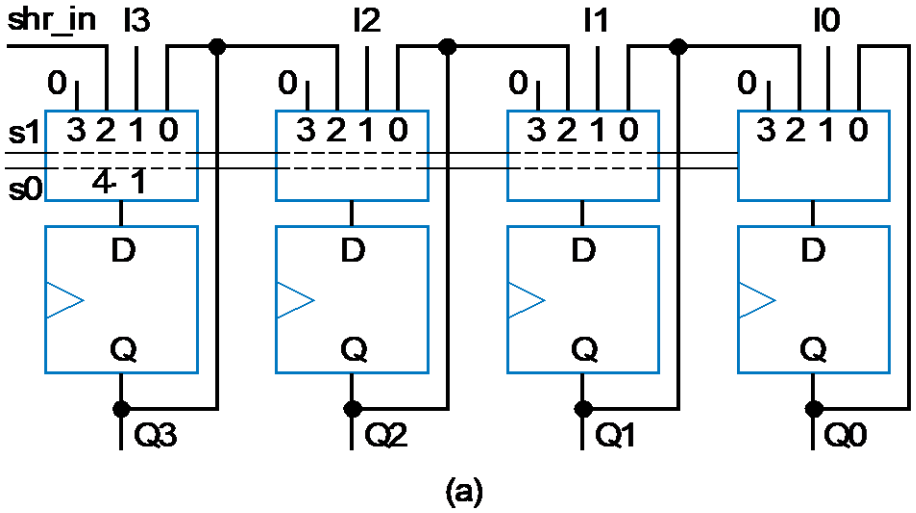


Multifunction Registers

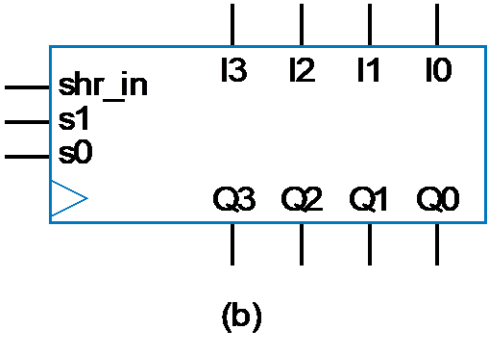
- Many registers have multiple functions
  - Load, shift, clear (load all 0s)
  - And retain present value, of course
- Easily designed using muxes
  - Just connect each mux input to achieve desired function

Functions:

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



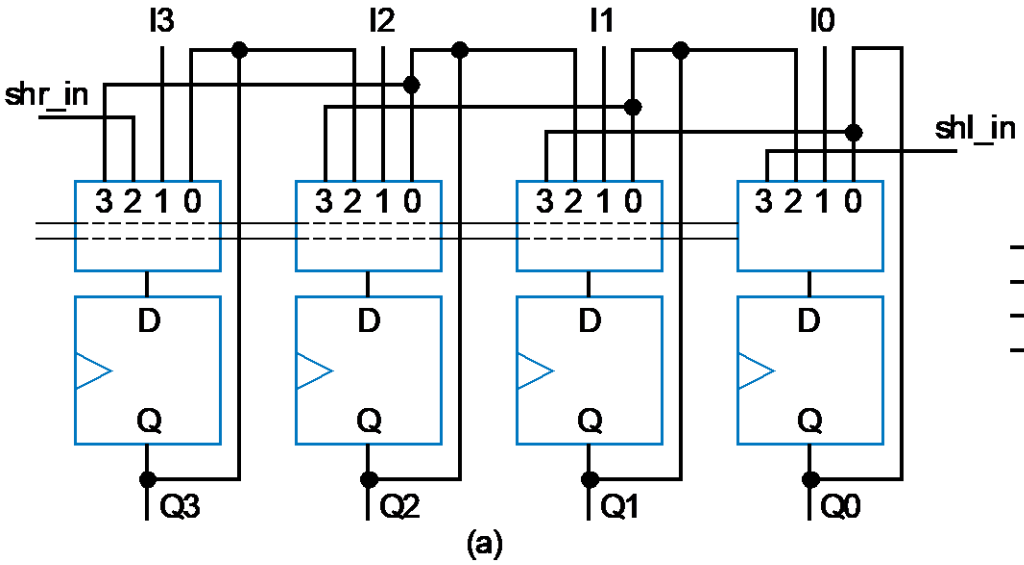
(a)

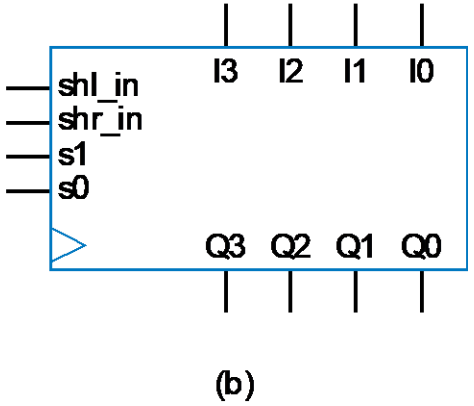


(b)

Multifunction Registers

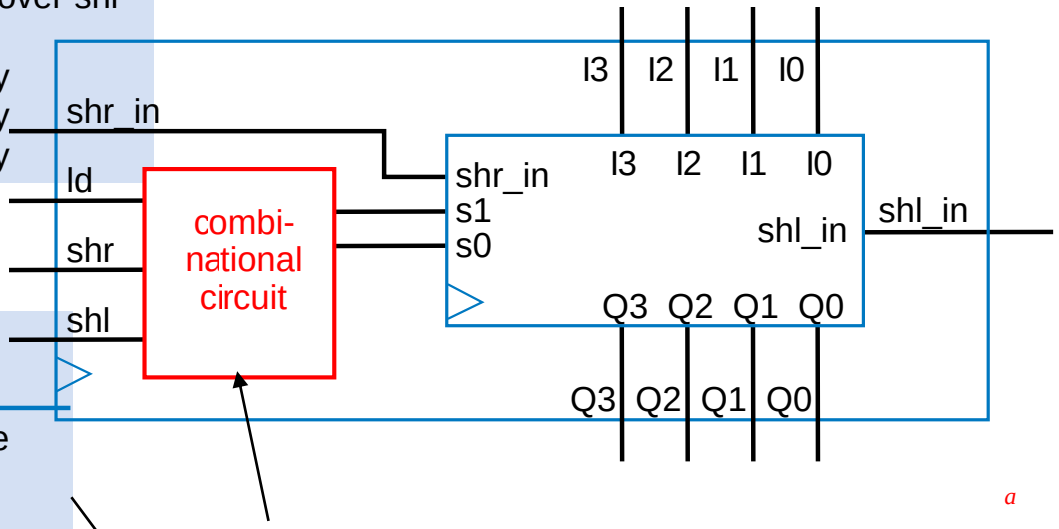
s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	Shift left





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 ld	shr	shl	Outputs s1	s0	Note 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

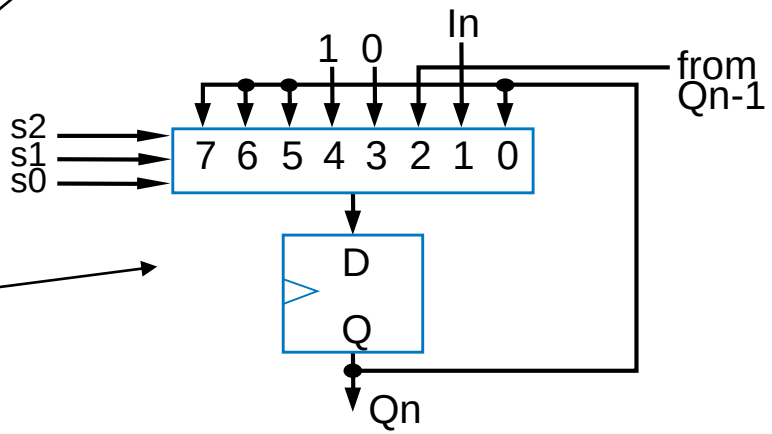
s2	s1	s0	Operation
0	0	0	Maintain present value
0	0	1	Parallel load
0	1	0	Shift left
0	1	1	Synchronous clear
1	0	0	Synchronous set
1	0	1	Maintain present value
1	1	0	Maintain present value
1	1	1	Maintain present value

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

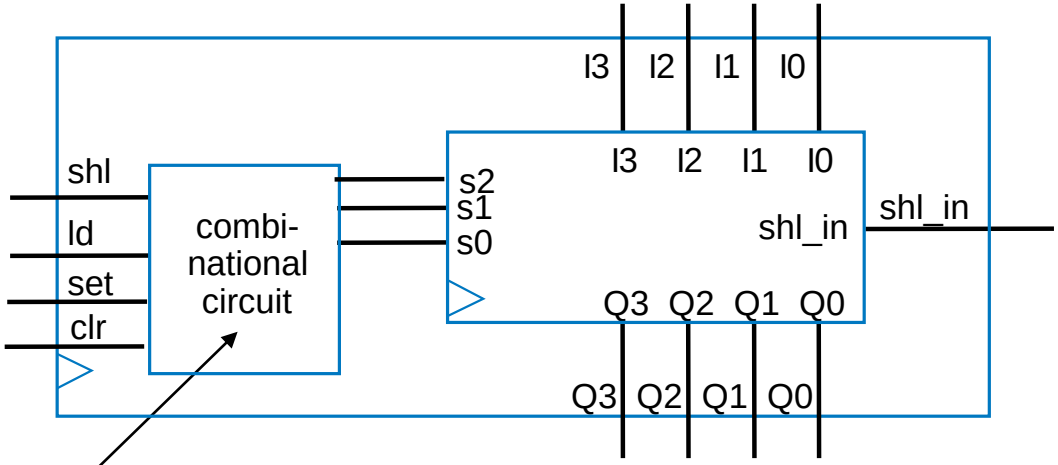


Step 4: Map control lines

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

Register Design Example



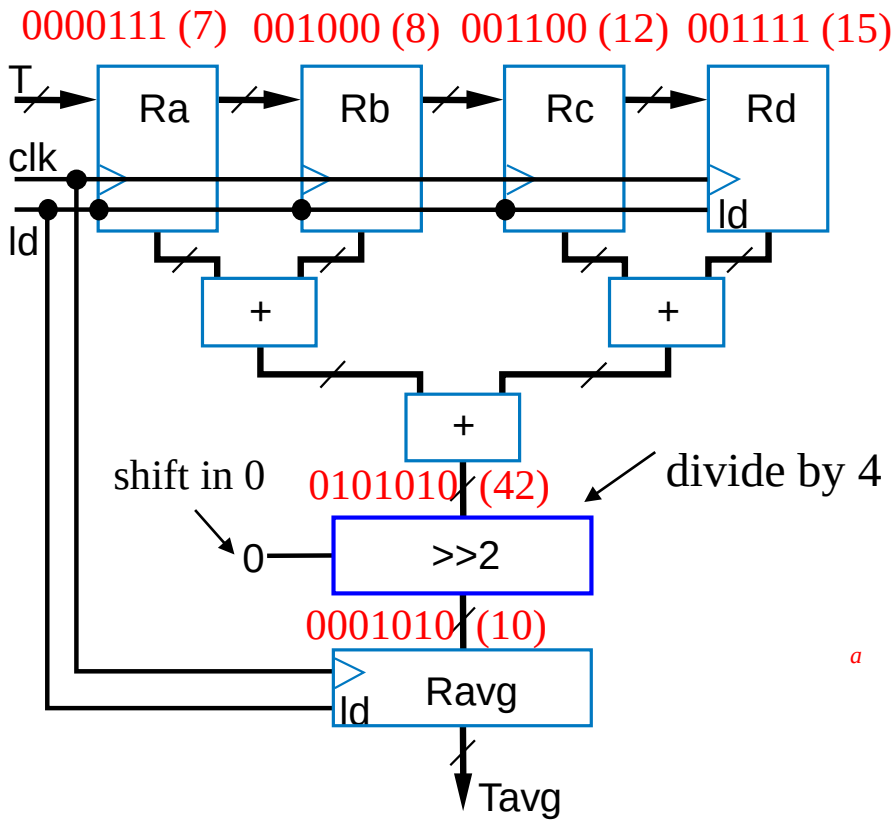
Step 4: Map control lines

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

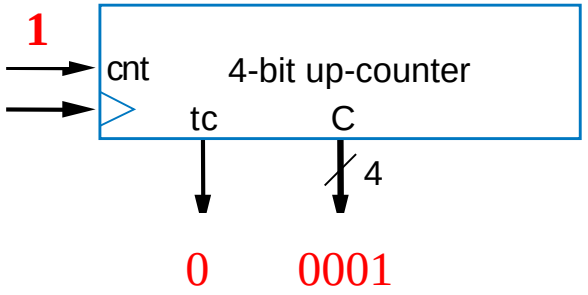
Shifter Example: Temperature Averager

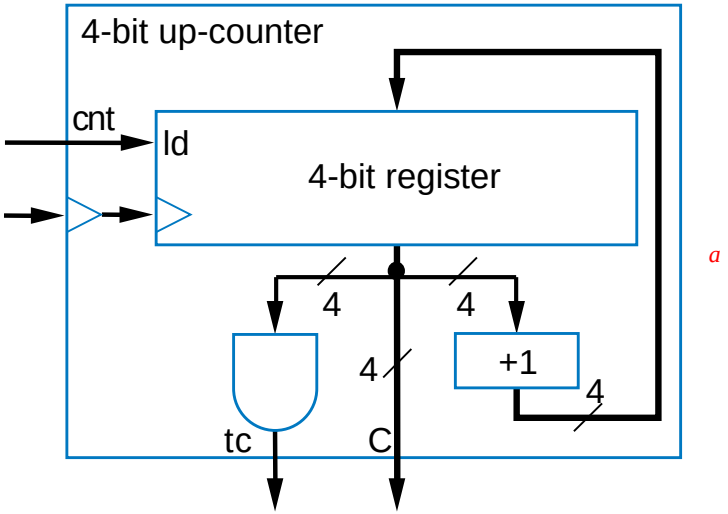
- Four registers storing a history of temperatures
- Want to output the average of those temperatures
- Add, then divide by four
  - Same as shift right by 2
  - Use three adders, and right shift by two



Counters

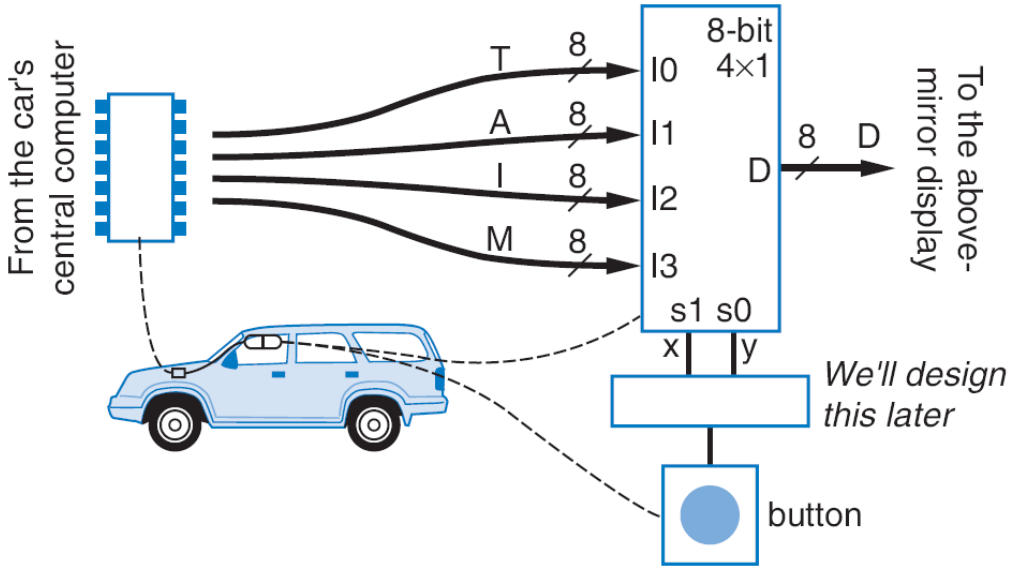
- ***N-bit up-counter***: N-bit register that can increment (add 1) to its own value on each clock cycle
  - 0000, 0001, 0010, 0011, ..., 1110, 1111, 0000
  - Note how count "rolls over" from 1111 to 0000
    - Terminal (last) count, tc, equals1 during value just before rollover
- Internal design
  -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

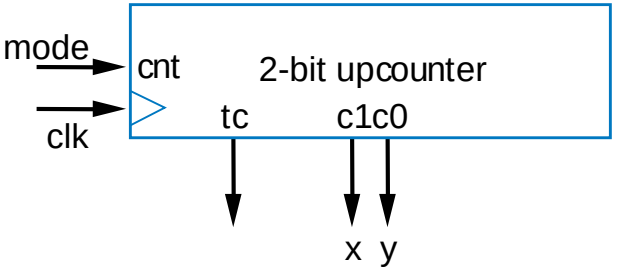




Counter Example: Mode in Above-Mirror Display

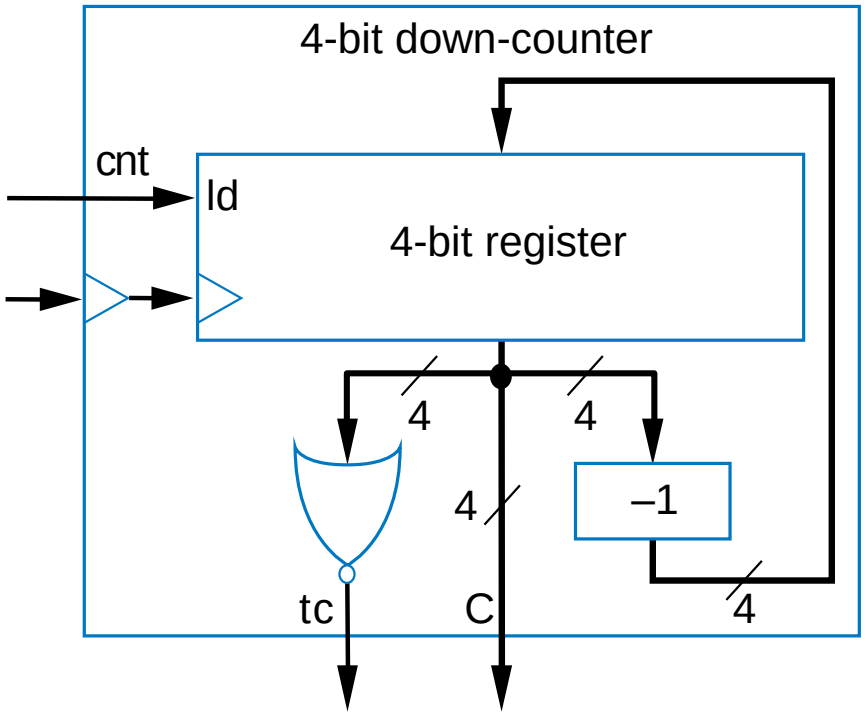
- Assumed component that incremented xy input each time button pressed: 00, 01, 10, 11, 00, 01, 10, 11, 00, ...
- Can use 2-bit up-counter
  - Assumes mode=1 for just one clock cycle during each button press
    - Recall “Button press synchronizer” example from Chapter 3





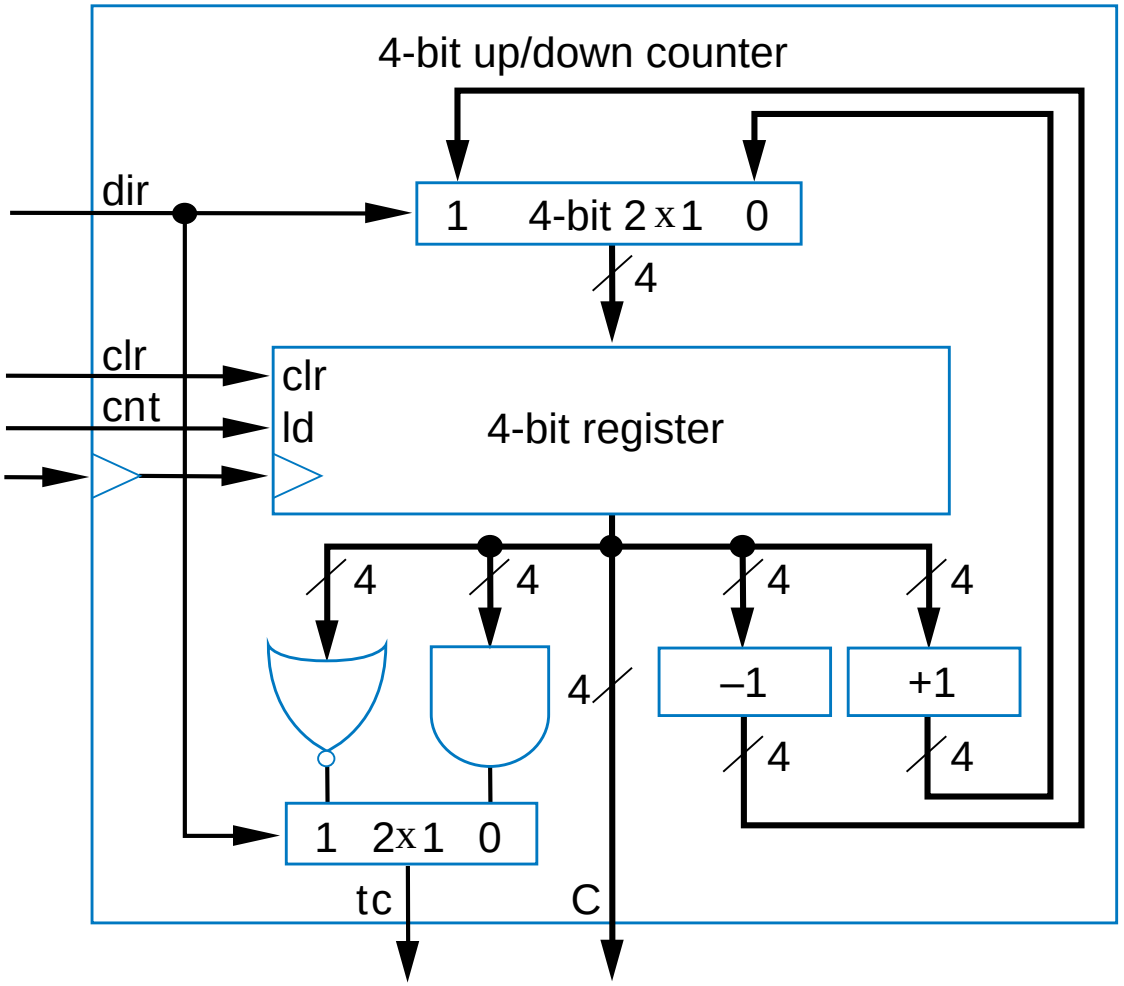
Down-Counter

- 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



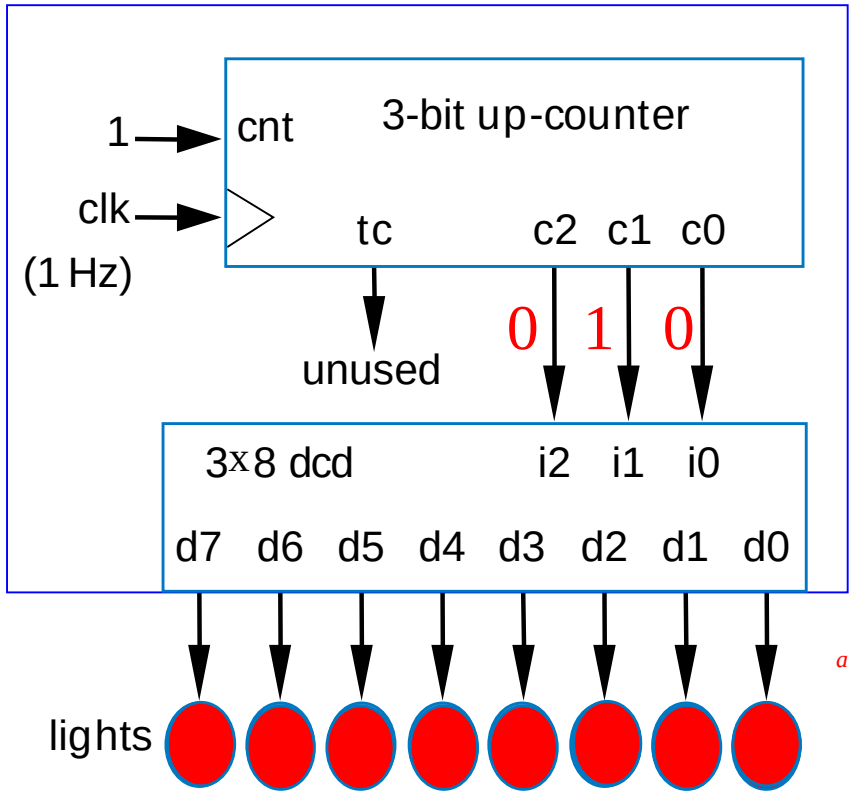
Up/Down-Counter

- 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



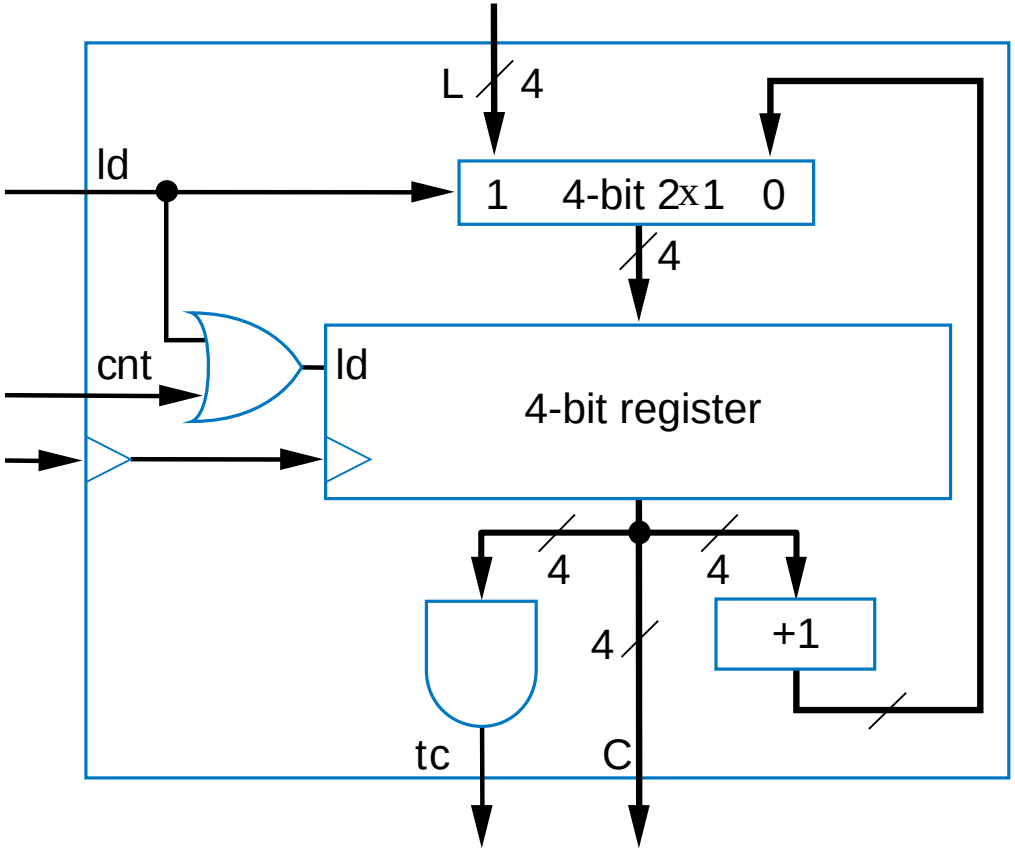
Counter Example: Light Sequencer

- 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?



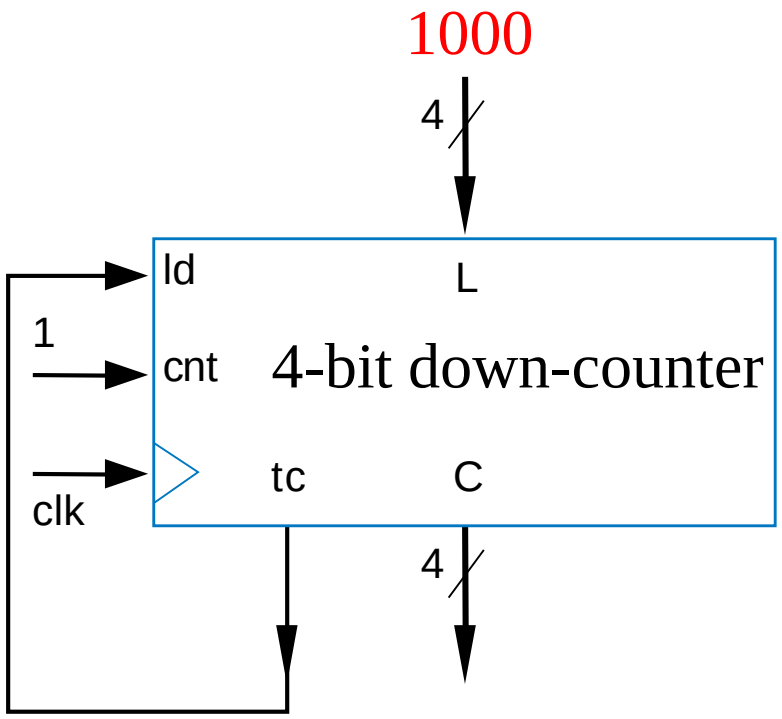
Counter with Parallel Load

- 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



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

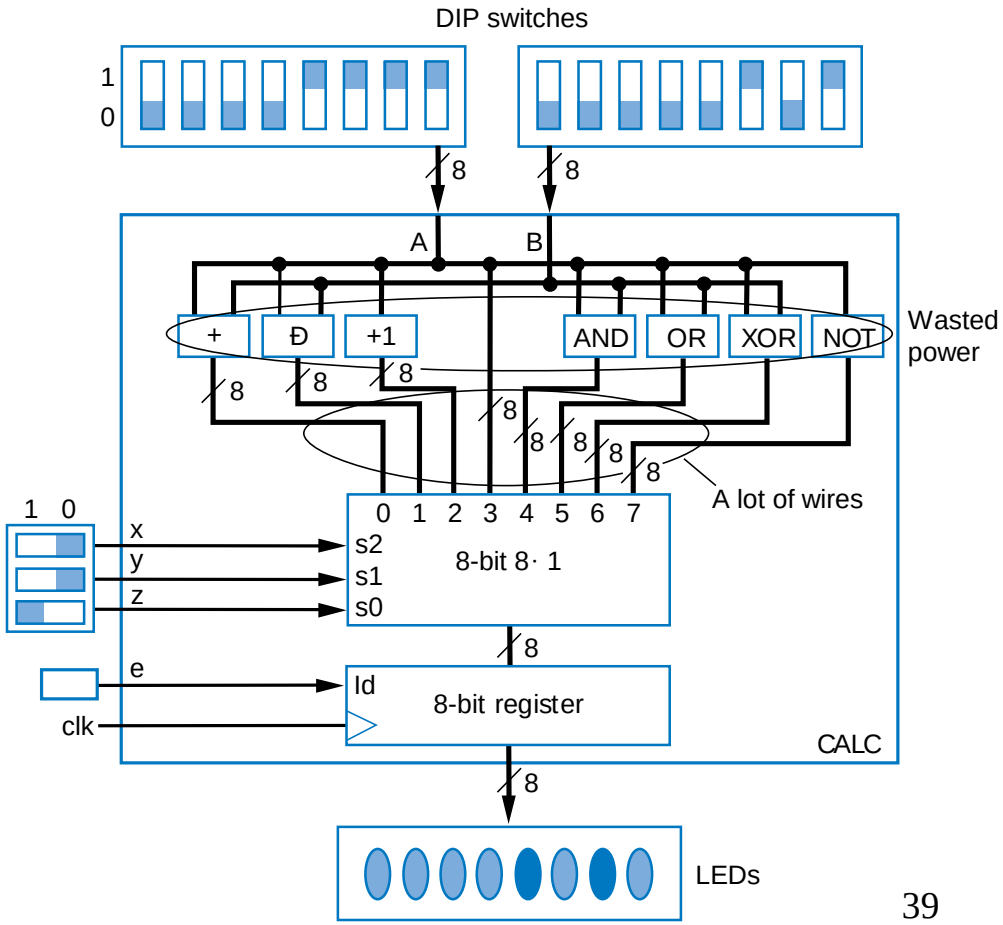
Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1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

Multifunction Calculator without an ALU

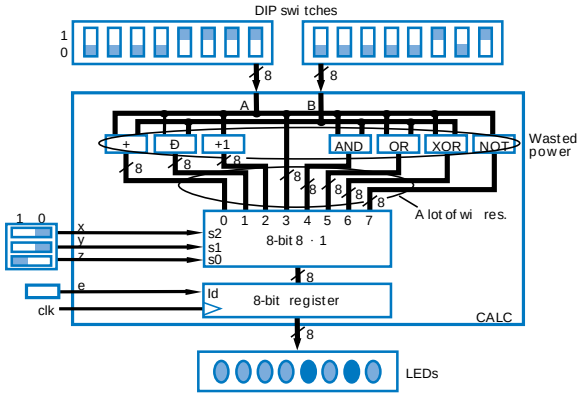
- Can build multifunction calculator using separate components for each operation, and muxes
  - But too many wires, and wasted power computing all those operations when at any time you only use

TABLE 4.2 Desired calculator operations

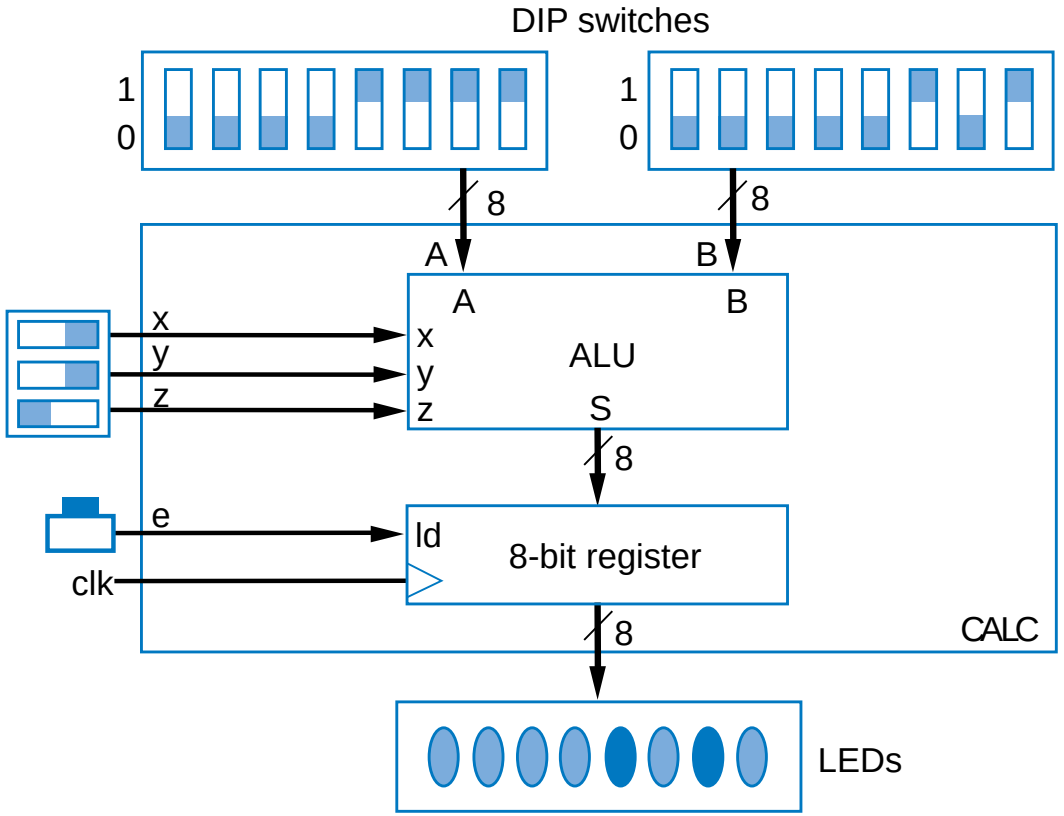
Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AN[redacted]	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010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



ALU Example: Multifunction Calculator

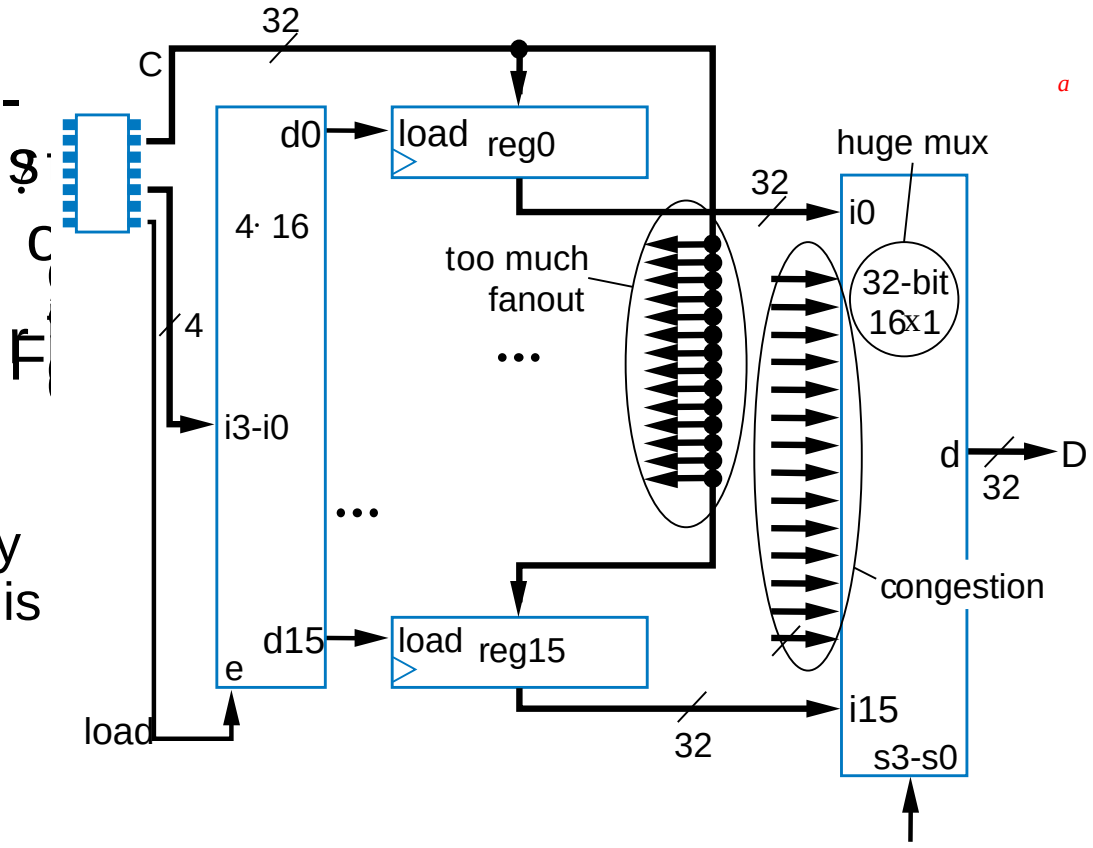


- Design using ALU is elegant and efficient
  - No mass of wires
  - No big waste of power



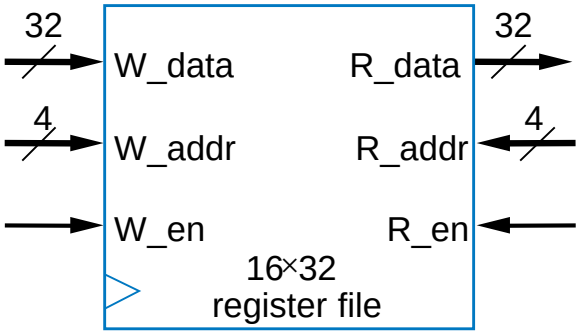
Register Files

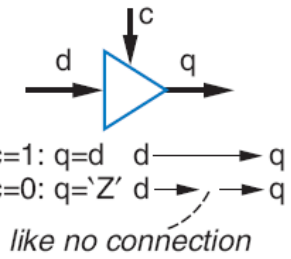
- ***MxN register file*** component provides efficient access to M N-bit-wide registers
  - If we have many registers but only need access one or two at a time, a register file is more efficient
  - Ex: Above-mirror display (earlier example), but this time having 16 32-bit registers
    - Too many wires,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

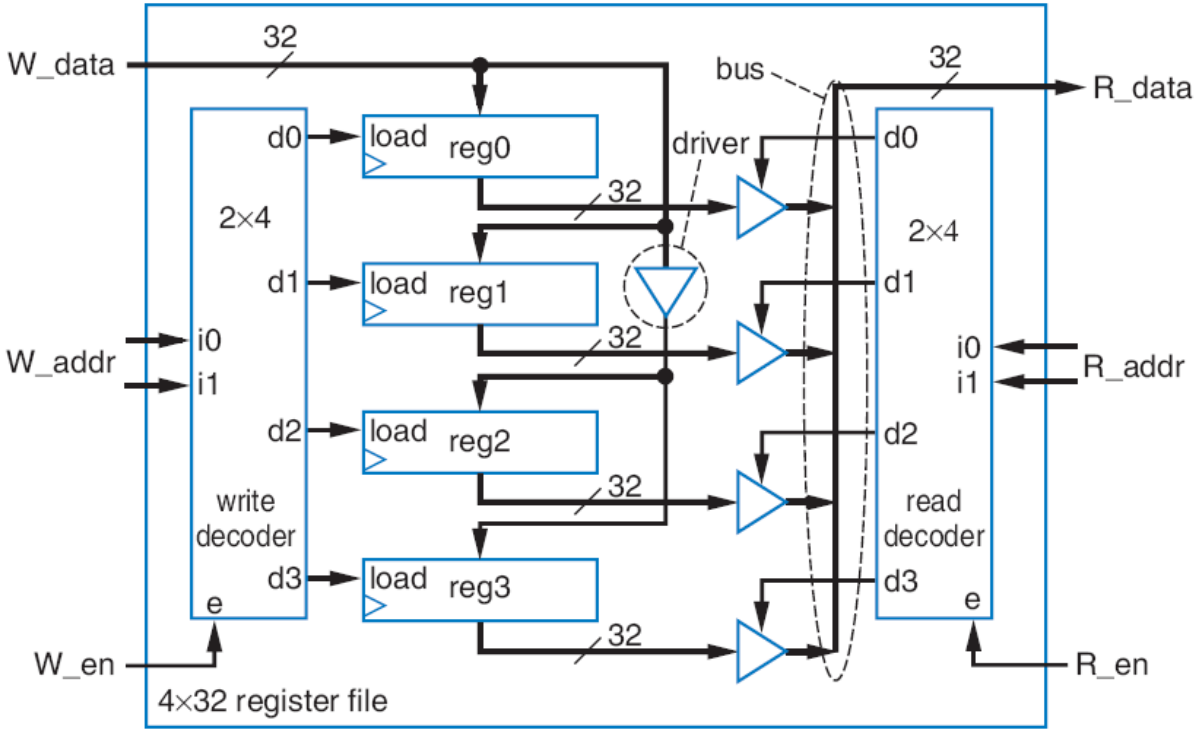


Register File

- 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

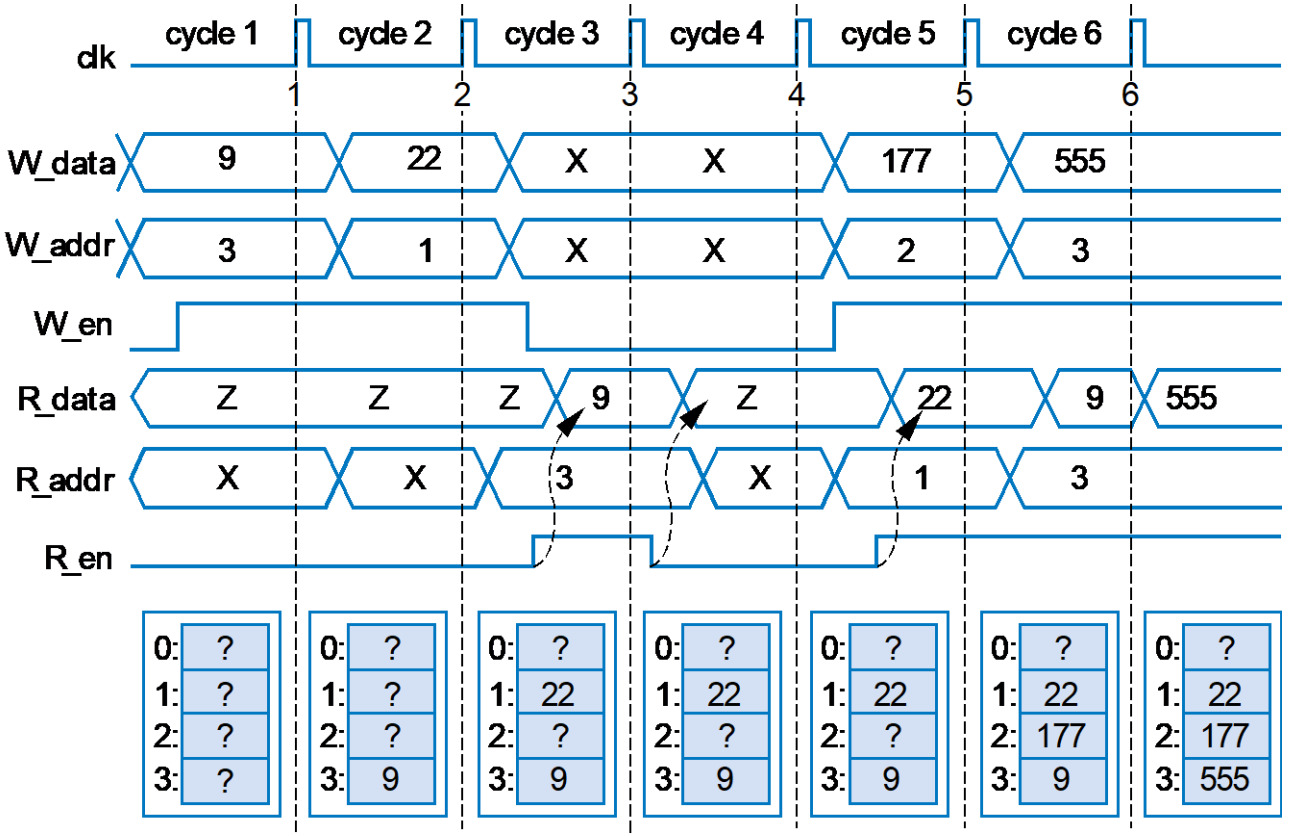


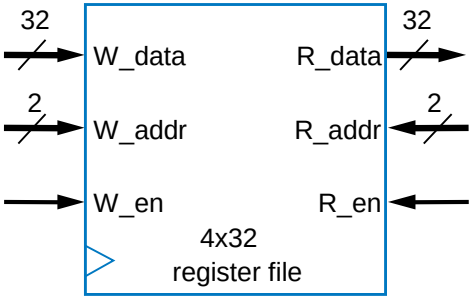




Register File Timing Diagram

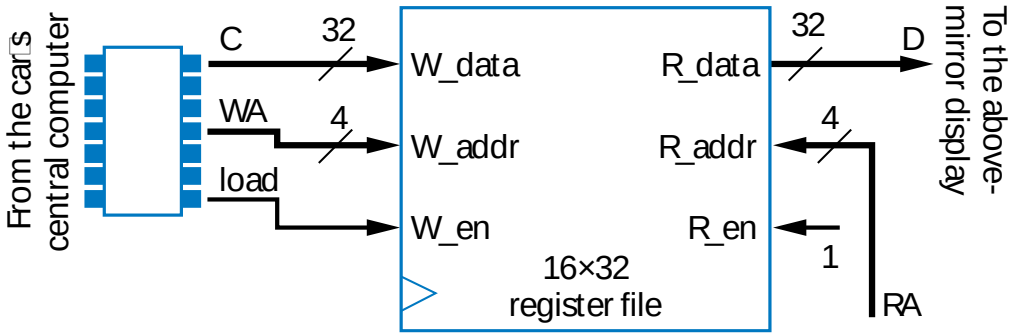
- Can write one register and read one register each clock cycle
  - May be same register





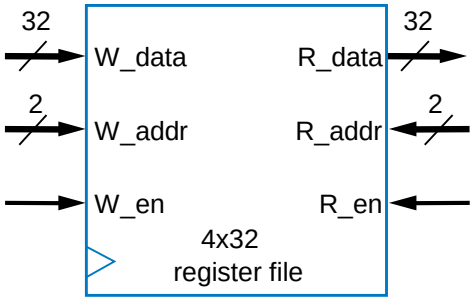
Register-File Example: Above-Mirror Display

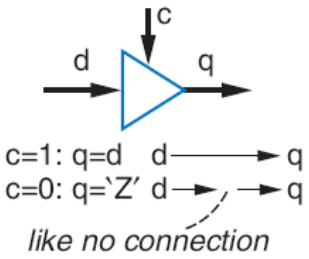
- 16 32-bit registers that can be written by car's computer, and displayed
  - Use 16x32 register file
  - Simple, elegant design
- Register file hides complexity internally
  - And because only one register needs to be written and/or read at a time, internal design is simple

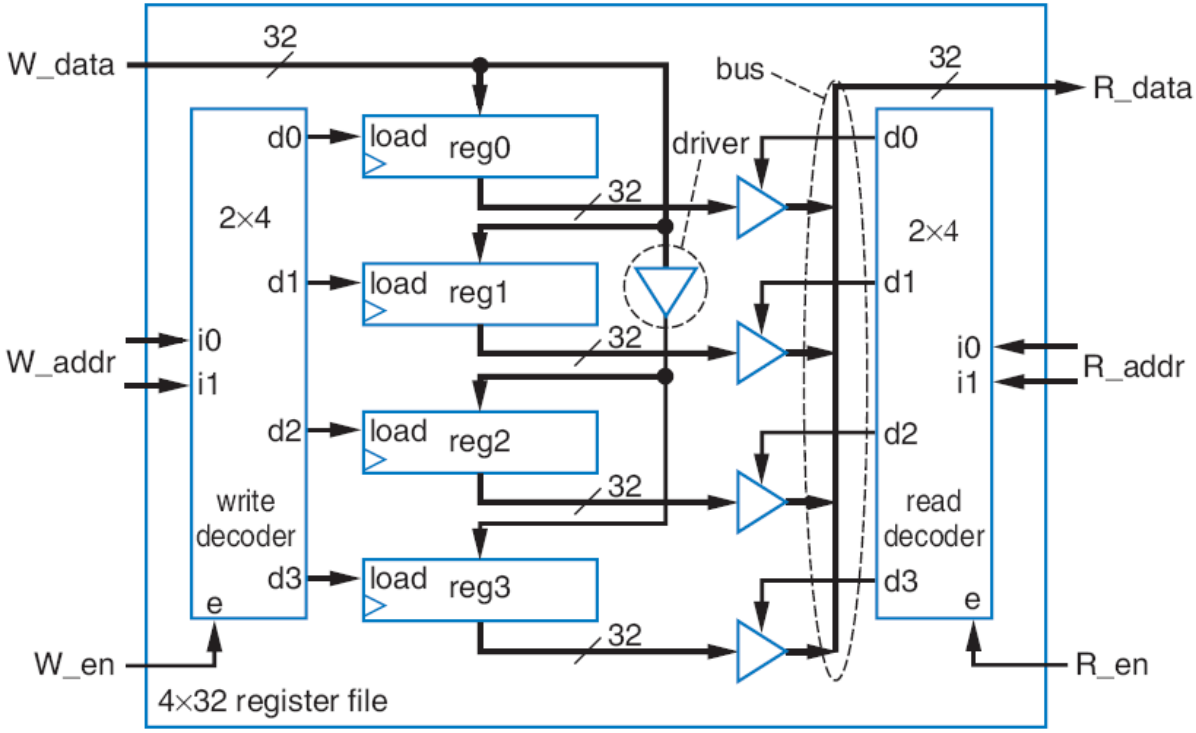


Register File

- 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

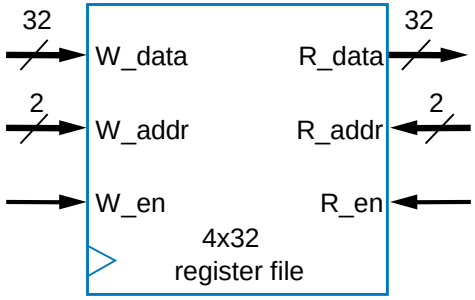


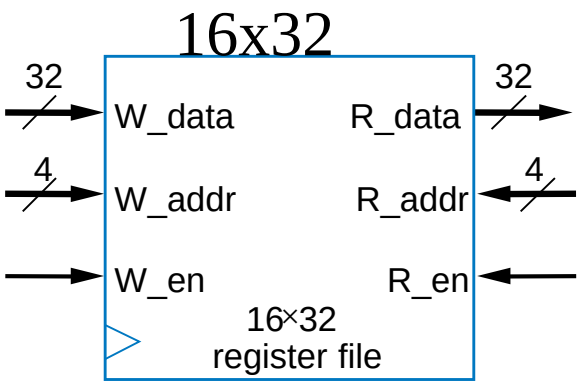


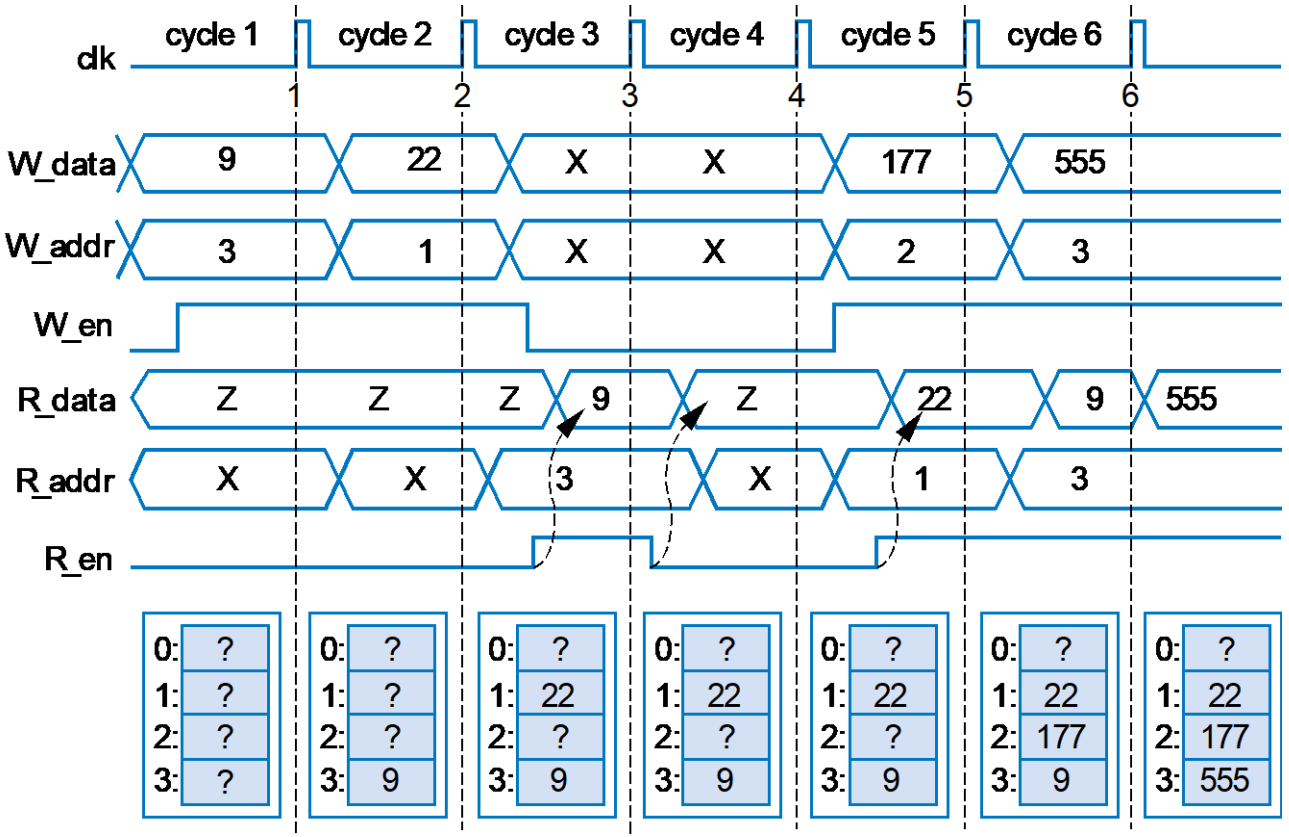


Register File Timing Diagram

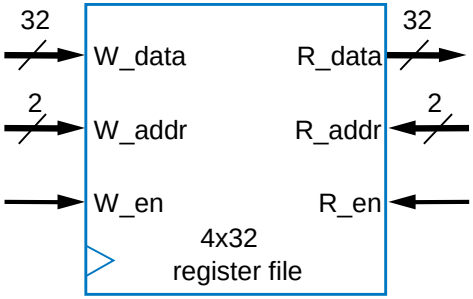
- Can write one register and read one register each clock cycle
  - May be same register



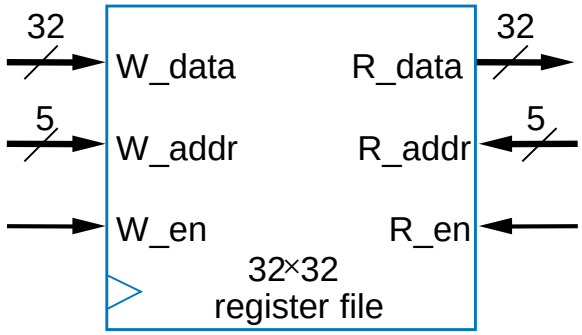




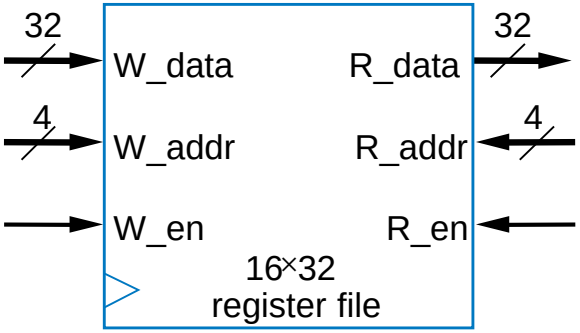
Register File -size



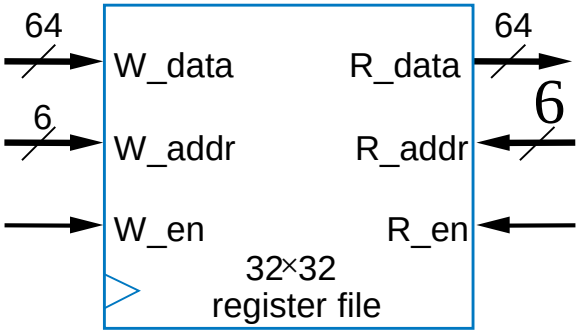
32x32



16x32

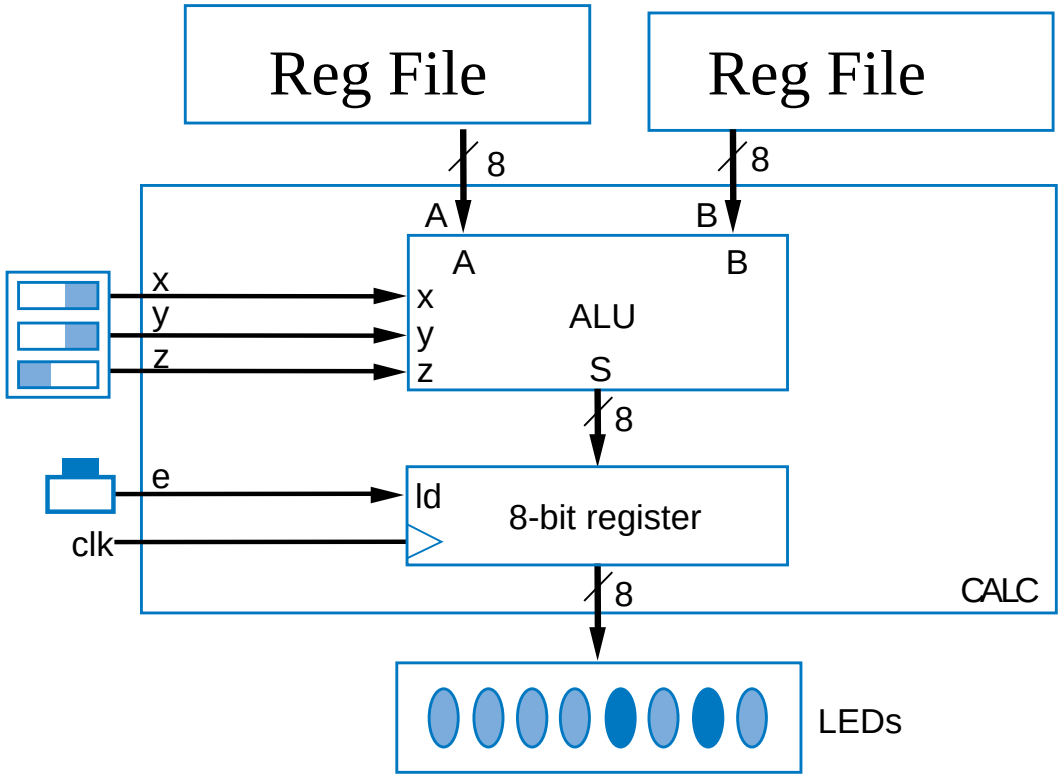


64x64

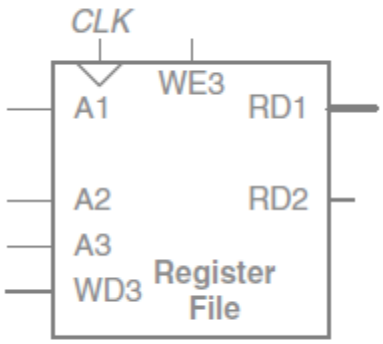


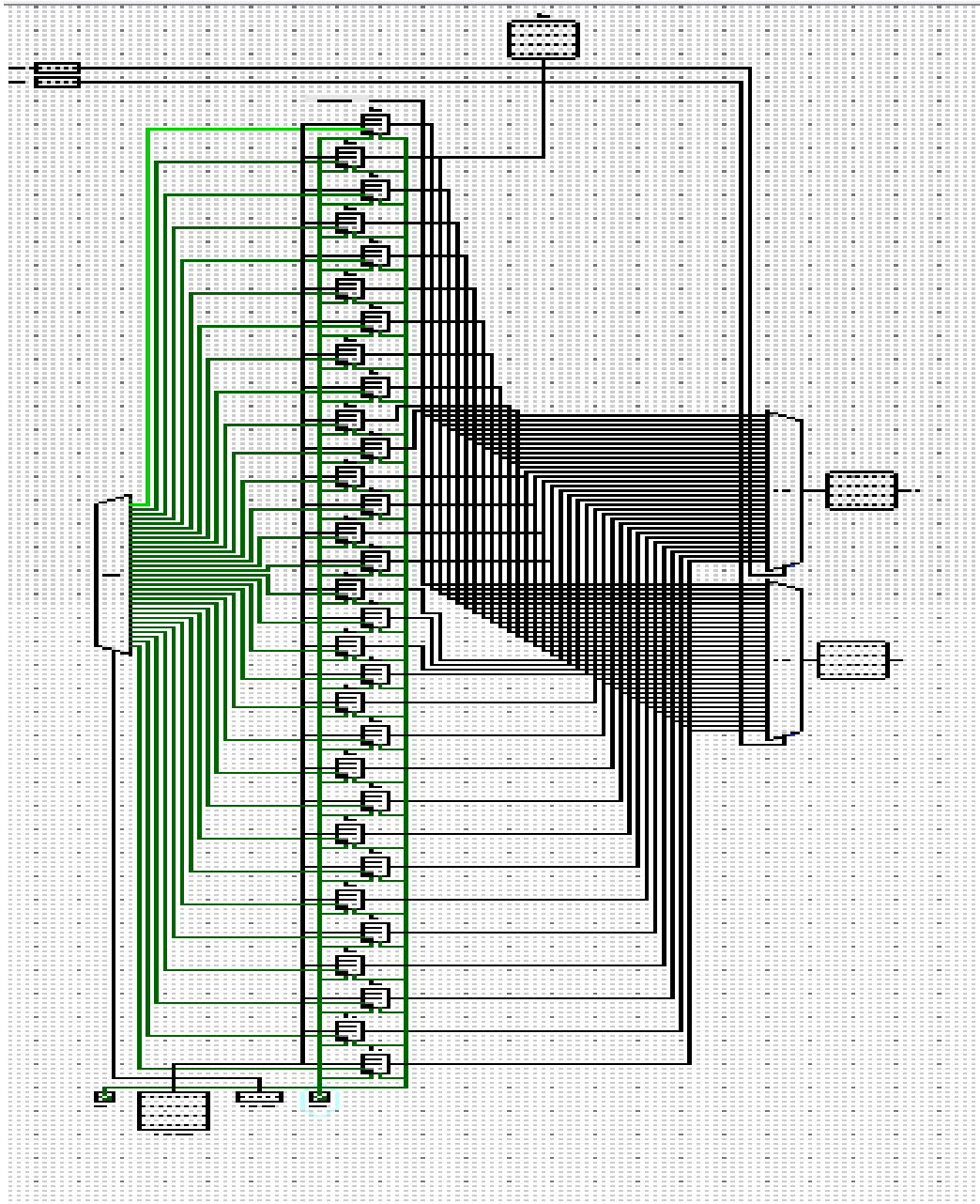
ALU Example:

- Design using ALU is elegant and efficient



Register File with Two Ports





ALU Example:

- Design using ALU is elegant and efficient