

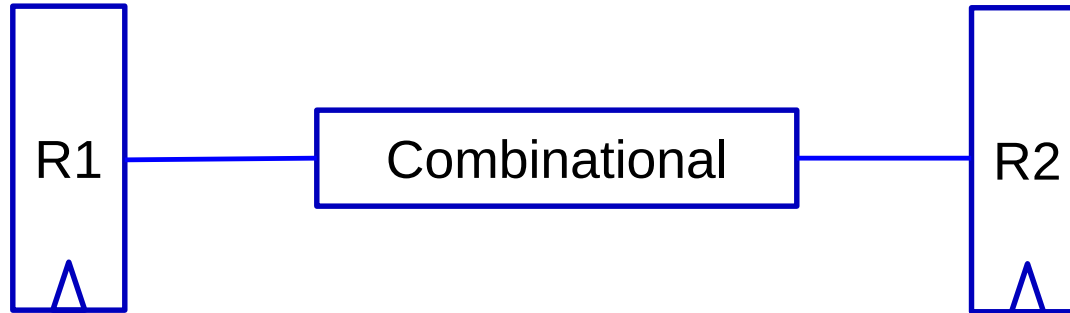


CSE140: Components and Design Techniques for Digital Systems

Timing constraints

Pietro Mercati

Summary on timing constraints



Combinational:

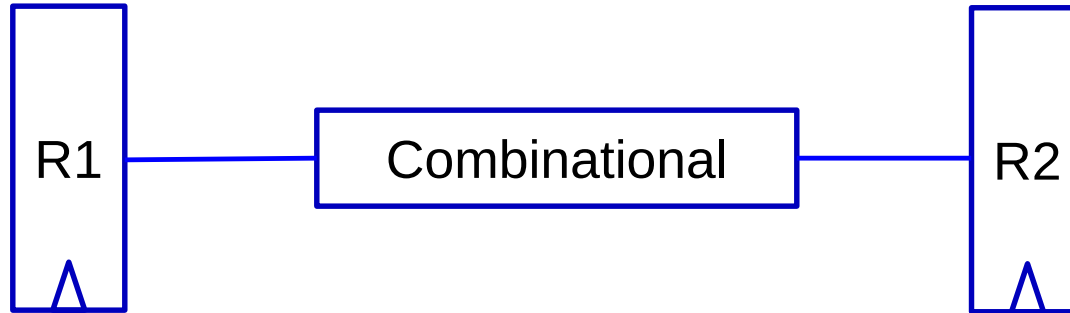
- Maximum delay = Propagation delay t_{pd}
- Minimum delay = Contamination delay t_{cd}

Flip Flops:

- Input:
 - Setup time t_{setup}
 - Hold time t_{hold}
- Output:
 - Propagation clock-to-Q time t_{pcq}
 - Contamination clock-to-Q time t_{ccq}

Once the logic/FFs are built, these timing characteristics are fixed properties

Summary on timing constraints



Constraint inequalities:

- Without clock skew:

$$T_c \geq t_{setup} + t_{pcq} + t_{pd}$$

Setup time
constraint

$$t_{hold} < t_{ccq} + t_{cd}$$

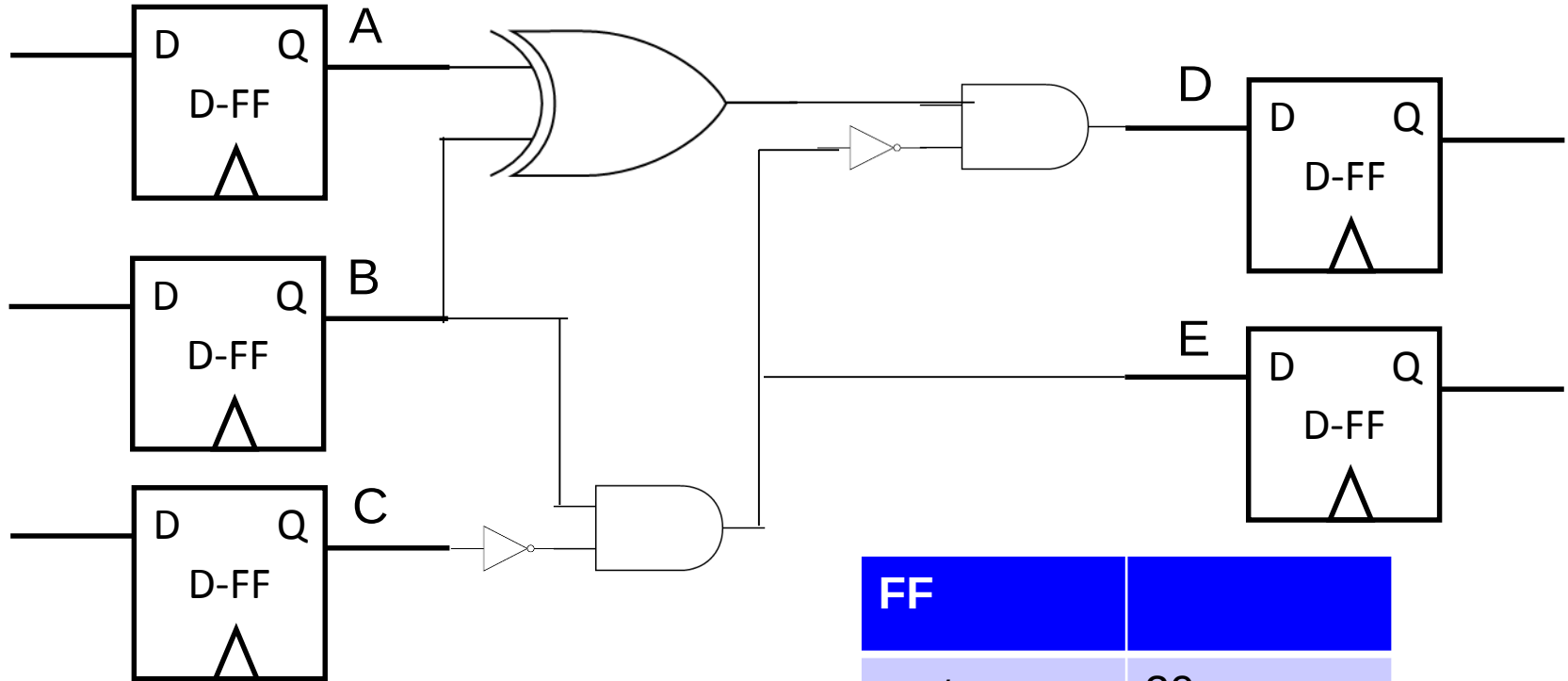
Hold time
constraint

- With clock skew:

$$T_c \geq t_{setup} + t_{pcq} + t_{pd} + t_{skew}$$

$$t_{skew} + t_{hold} < t_{ccq} + t_{cd}$$

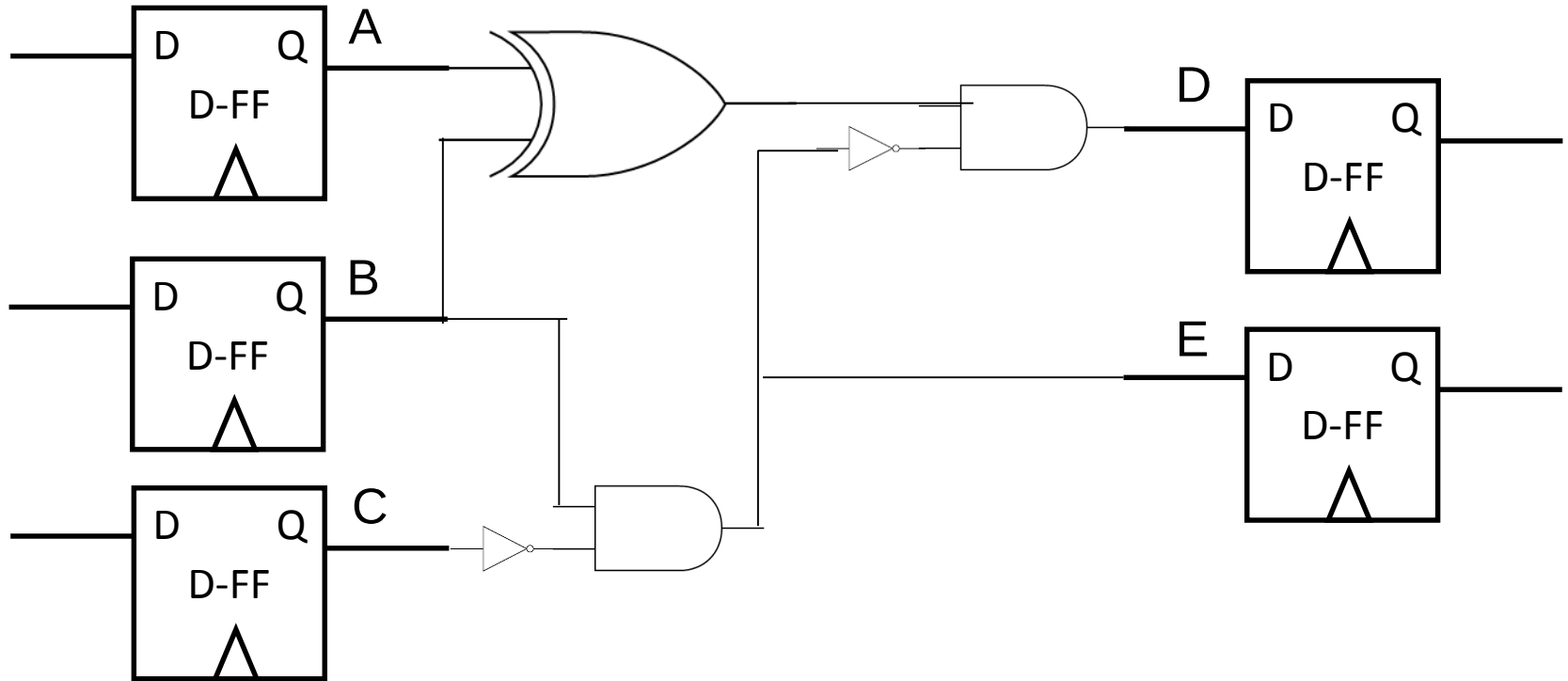
Example: timing constraints



	t_{pd}	t_{cd}
AND	20ns	10ns
NOT	10ns	10ns
XOR	110ns	50ns

FF	
t_{setup}	20ns
t_{hold}	30ns
t_{ccq}	10ns
t_{pcq}	70ns

Example: timing constraints

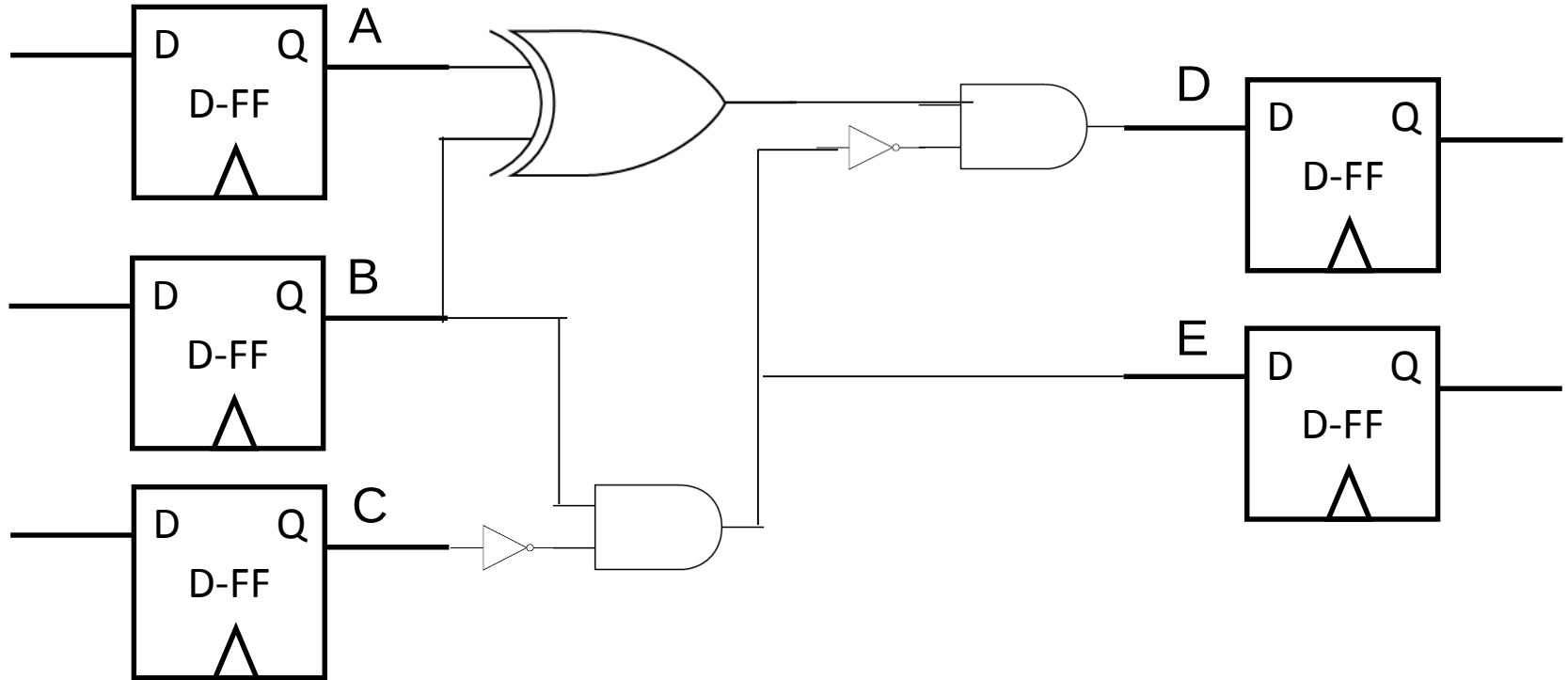


1) Assume $t_{skew} = 0$, find the maximum frequency

$$T_c \geq t_{setup} + t_{pcq} + t_{pd} = 20 + 70 + (110 + 20)$$

$$f_{MAX} = \frac{1}{T_{c_{MIN}}} = \frac{1}{220} 10^9 \cong 4.5\text{MHz}$$

Example: timing constraints



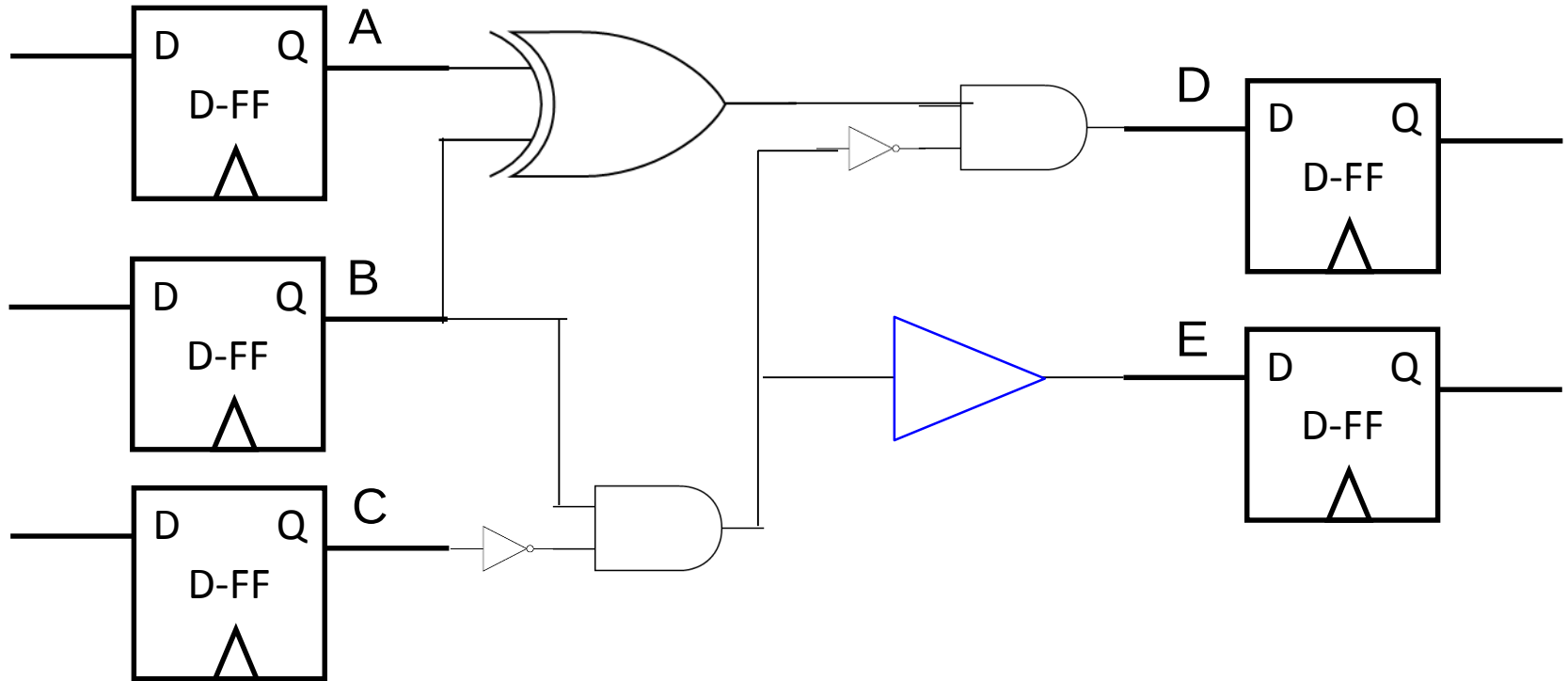
2) Does the circuit have a hold violation?

$$t_{hold} < t_{ccq} + t_{cd}$$

$$30 < 10$$

Hold time violation !

Example: timing constraints



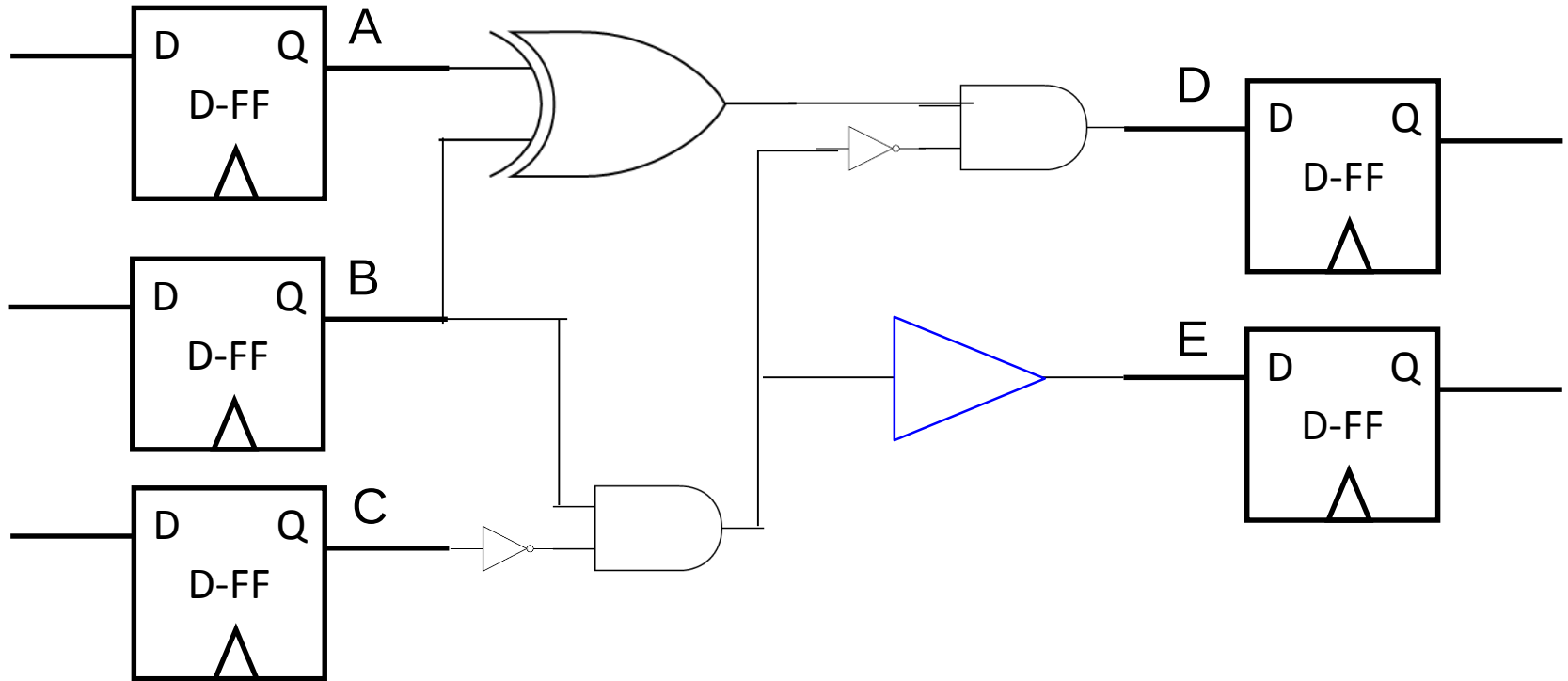
3) Where to place a buffer with $t_{cd} = t_{pd} = 25$ to solve this hold time violation?

$$t_{hold} < t_{ccq} + t_{cd}$$

$$30 < 10 + 10 + 25$$

Hold time
violation
solved!

Example: timing constraints

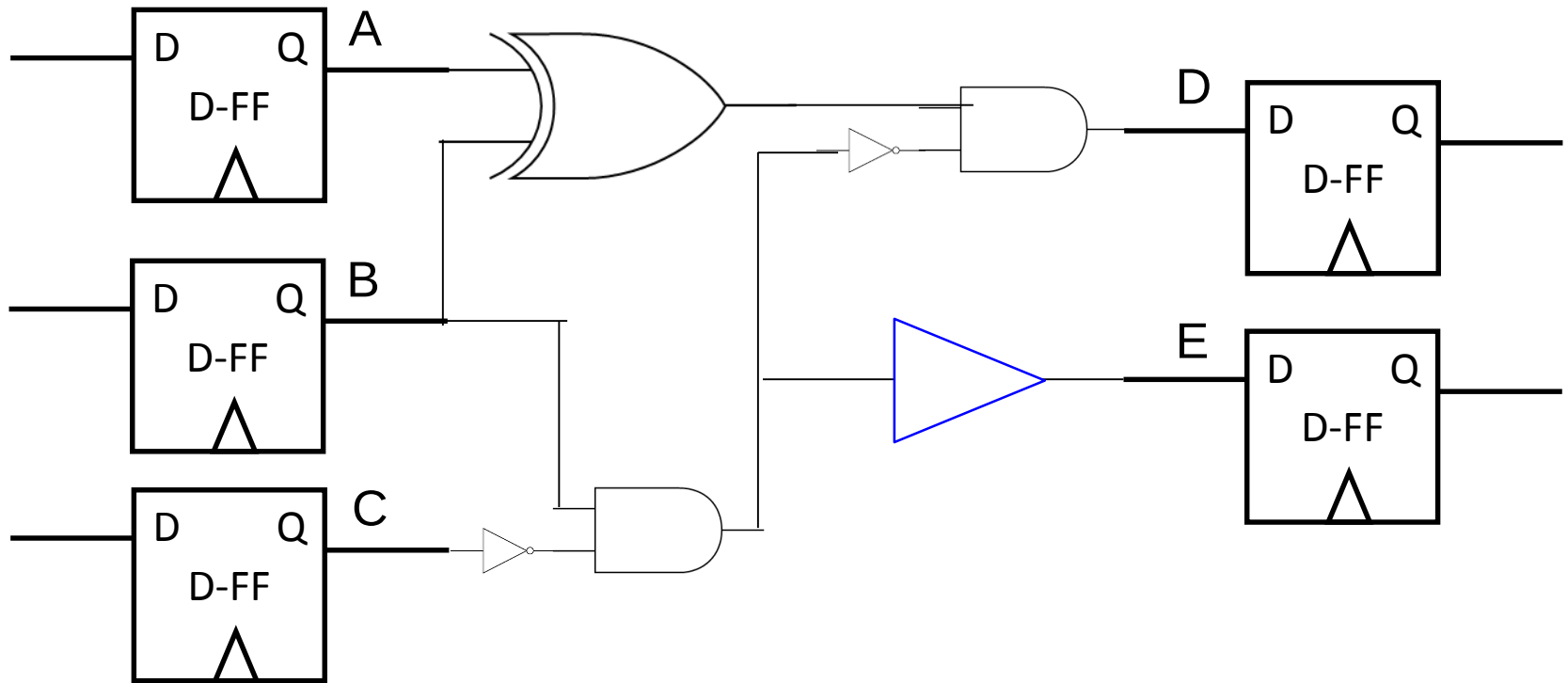


4) Assume $t_{skew} = 20ns$, find the maximum frequency

$$T_c \geq t_{setup} + t_{pcq} + t_{pd} + t_{skew} = 20 + 70 + (110 + 20) + 20$$

$$f_{MAX} = \frac{1}{T_{cMIN}} = \frac{1}{240} 10^9 \cong 4.16\text{MHz}$$

Example: timing constraints



5) Assume $t_{skew} = 20ns$ and the additional buffer, do we have a hold time violation?

$$t_{hold} + t_{skew} < t_{ccq} + t_{cd}$$

$$30 + 20 < 10 + 10 + 25$$

Hold time violation !



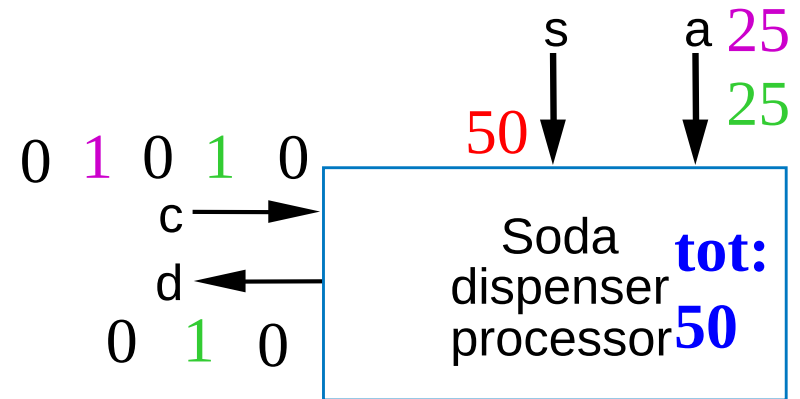
CSE140: Components and Design Techniques for Digital Systems

Register Transfer Level (RTL) Design

Slides from Tajana Simunic Rosing

High-Level State Machine

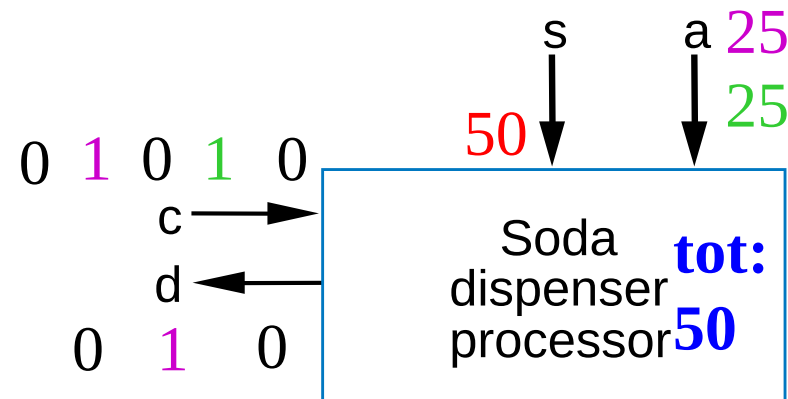
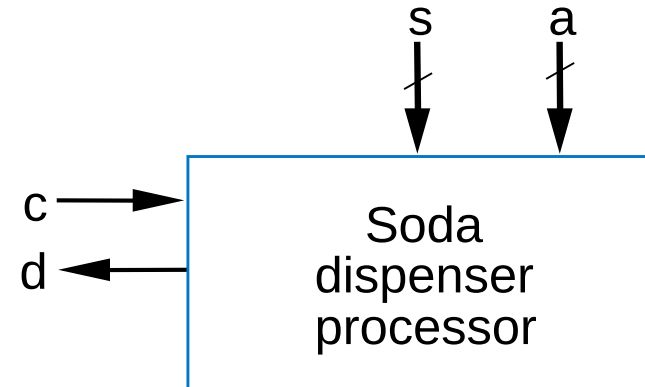
- Some behaviors may be too complex to describe by using classical FSMs
- Soda dispenser
 - *c*: bit input, 1 when coin deposited
 - *a*: 8-bit input: value of the deposited coin
 - *s*: 8-bit input: cost of a soda
 - *d*: bit output, processor sets it to 1 when total value of deposited coins equals or exceeds cost of a soda



Challenges in High-Level State Machines

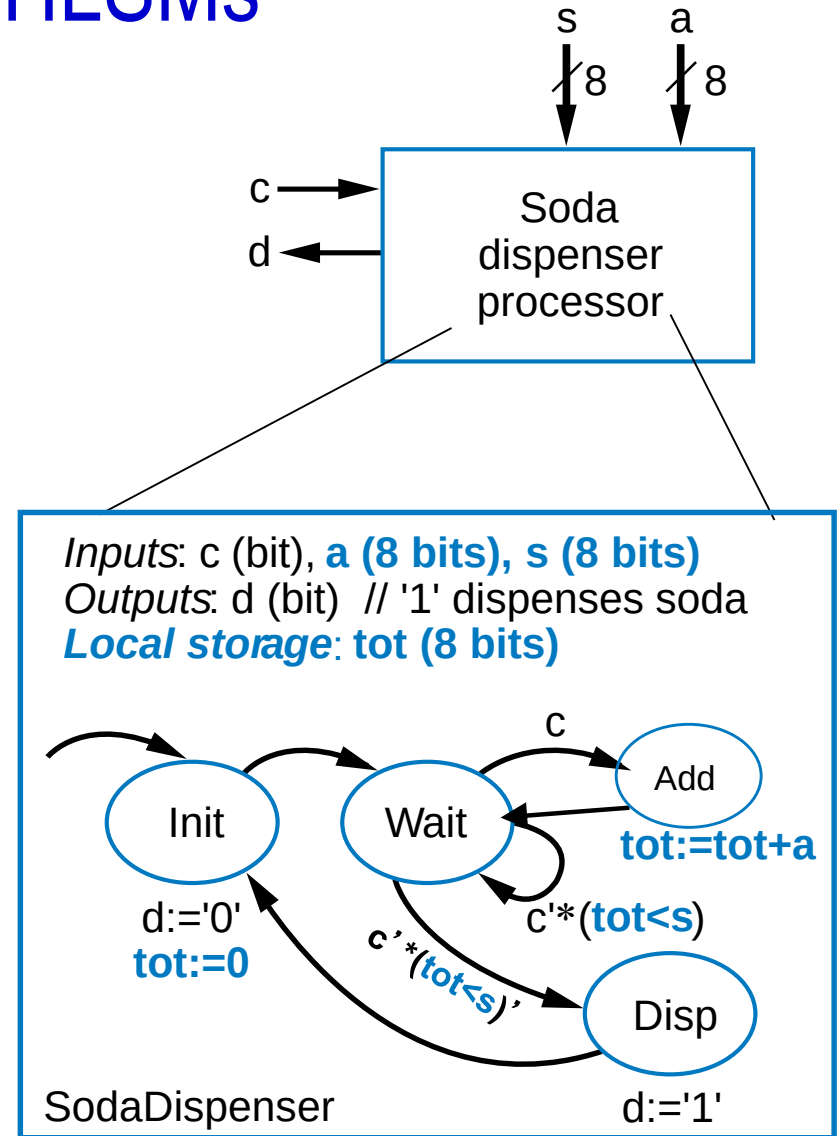
Which of the following makes the FSM design of this problem difficult?

- A. 8-bit input/output
- B. Tracking the current total
- C. Multibit comparison
- D. All of the above
- E. None of the above



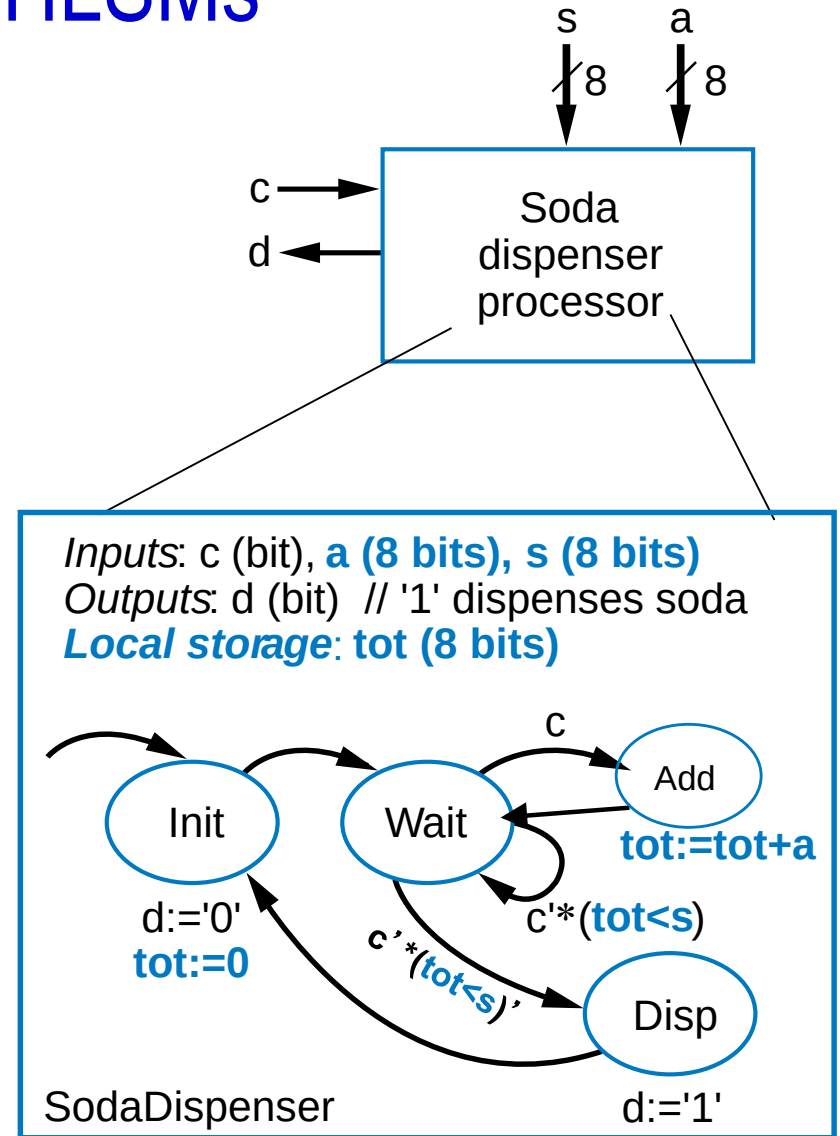
Benefits of HLSMs

- High-level state machine (HLSM) extends FSM with:
 - Multi-bit input/output
 - Local storage
 - Arithmetic operations
- Conventions
 - Numbers:
 - Single-bit: '0' (single quotes)
 - Integer: 0 (no quotes)
 - Multi-bit: "0000" (double quotes)
 - == for comparison equal
 - Multi-bit outputs *must* be registered via local storage
 - // precedes a comment



Benefits of HLSMs

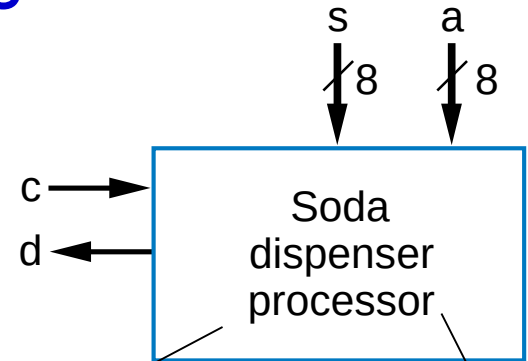
- High-level state machine (HLSM) extends FSM with:
 - Multi-bit input/output
 - Local storage
 - Arithmetic operations
- Conventions
 - Each transition is implicitly ANDed with a rising edge of the clock
 - Any bit output not explicitly assigned a value in a state is implicitly assigned to 0. This convention does not apply for multibit outputs
 - Every HLSM multibit output is registered



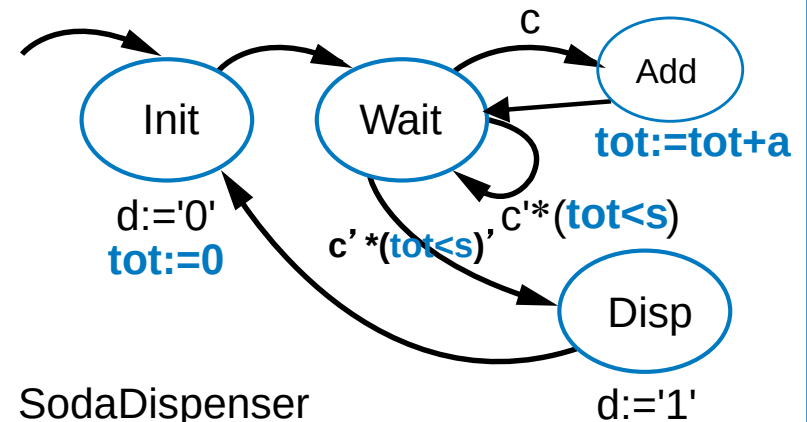
FSMs vs. HLSMs

How does the HLSM differ from the FSM for this problem?

- A. The HLSM stores multibit data, but the FSM doesn't
- B. The FSM stores the state but the HLSM doesn't
- C. Implementing HLSM and FSM requires multibit data registers
- D. All of the above
- E. None of the above



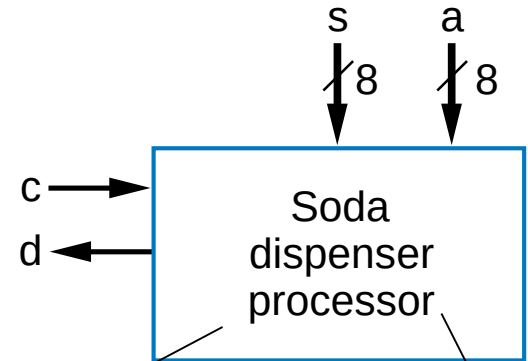
Inputs: c (bit), a (8 bits), s (8 bits)
 Outputs: d (bit) // '1' dispenses soda
 Local storage: tot (8 bits)



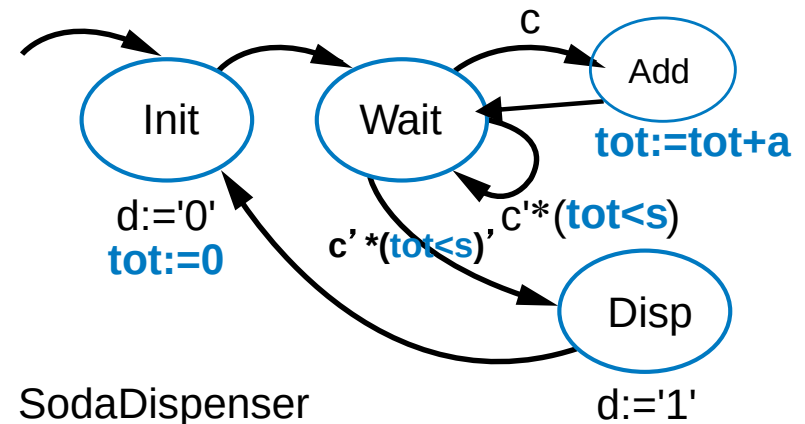
Similarities between FSMs & HLSMs

Which of the following are common between HLSMs and FSMs?

- A. Transitions happen at the edge of a clock
- B. They both have external complex data
- C. All of the above
- D. None of the above



Inputs: c (bit), **a (8 bits)**, **s (8 bits)**
Outputs: d (bit) // '1' dispenses soda
Local storage: tot (8 bits)



RTL Design Process

Step 1: Capture a high-level state machine

- Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is high level because the transition conditions and the state actions are more than just Boolean operations on single-bit input and outputs

Recommendations:

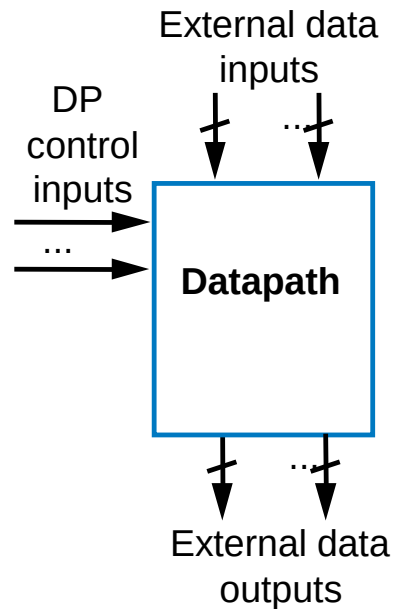
- Always list all inputs, outputs and local registers on top of your HLASM diagram
- Clearly specify the size in bits of each of them
- On states: update the value of registers, update of outputs
- On transitions: express conditions in terms of the HLASM inputs or state of the internal values and arithmetic operations between them.

RTL Design Process

Step 2: Convert it to a circuit

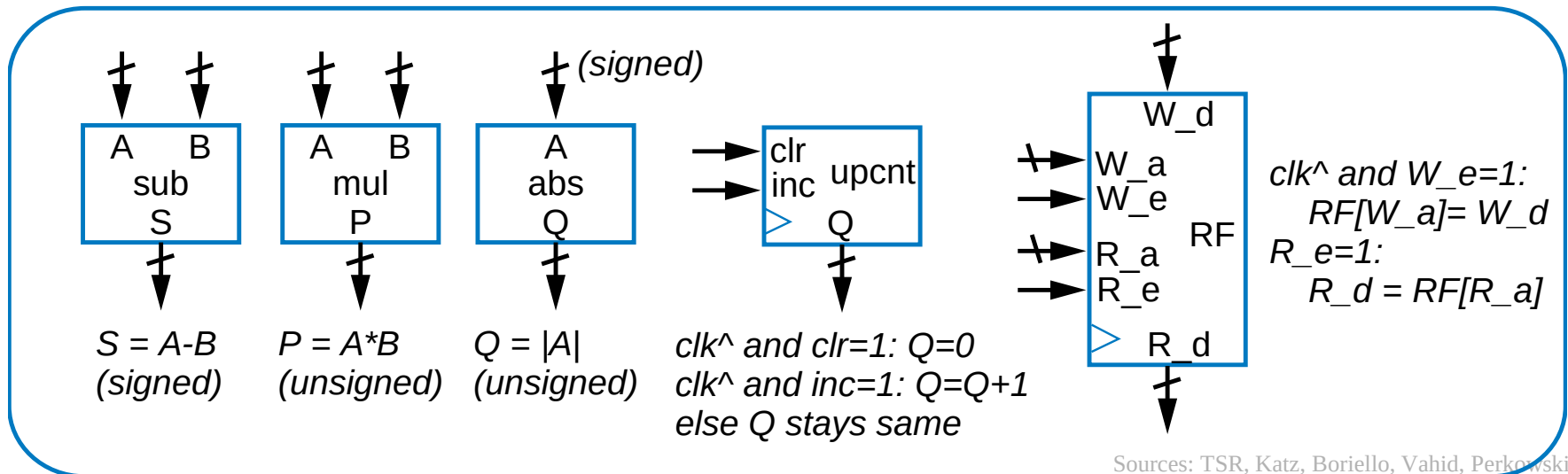
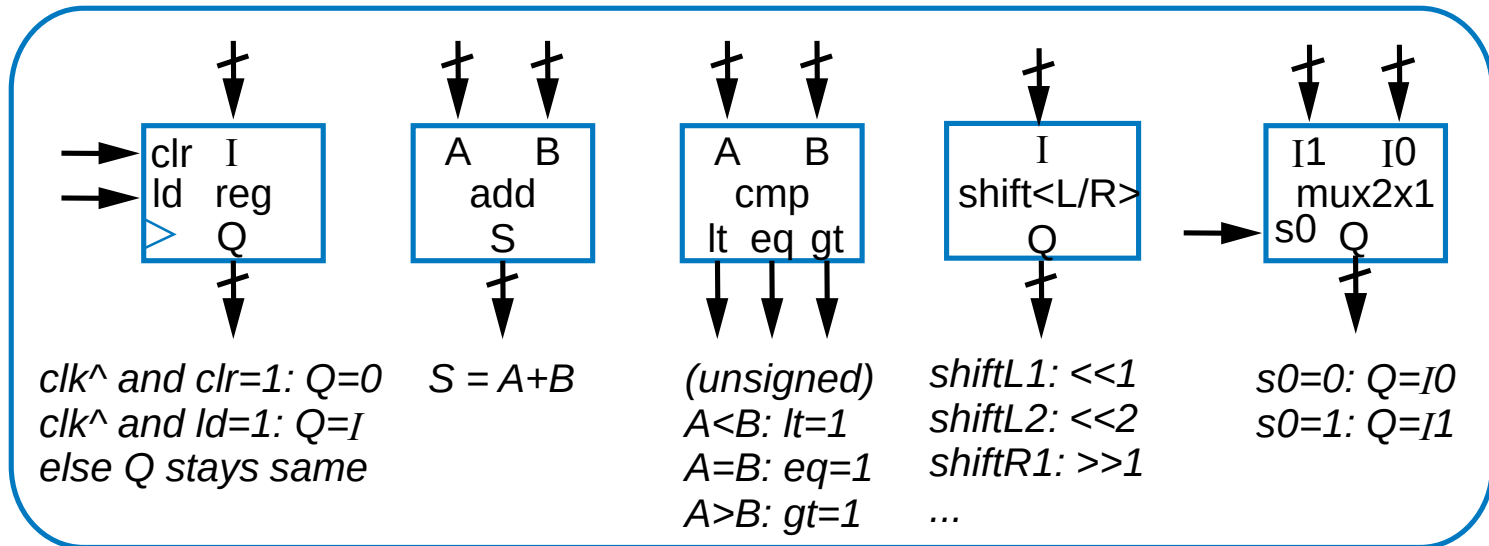
- 2.a: Create a datapath

- Create a datapath to carry out the data operations of the high level state machine
- Elements of your datapaths can be registers, adders, comparators, multipliers, dividers, etc.



RTL Design Process

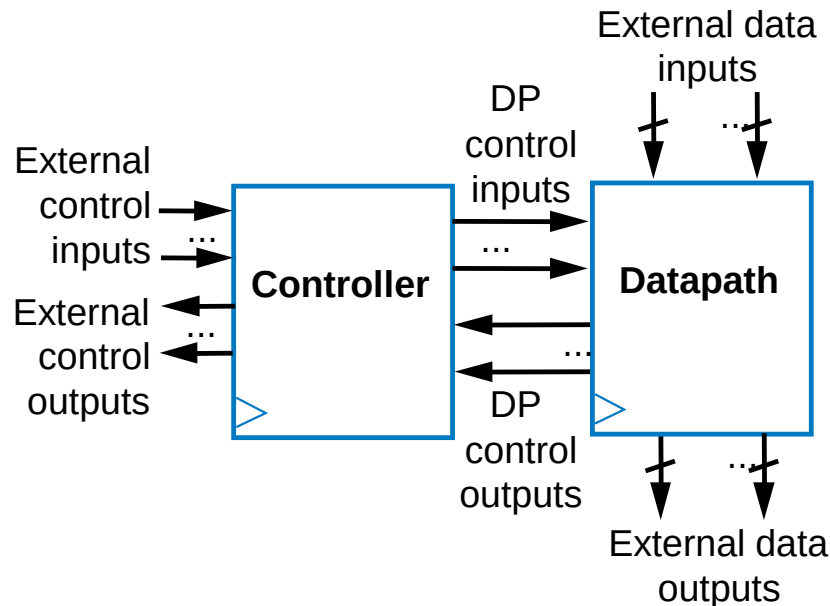
Datapath components:



RTL Design Process

Step 2: Convert it to a circuit

- **2.b: Connect the datapath to a controller**
 - Connect the datapath to a controller block. Connect the external control inputs and outputs to the controller block.
 - Clearly label all control signals that are exchanged between the datapath and the controller

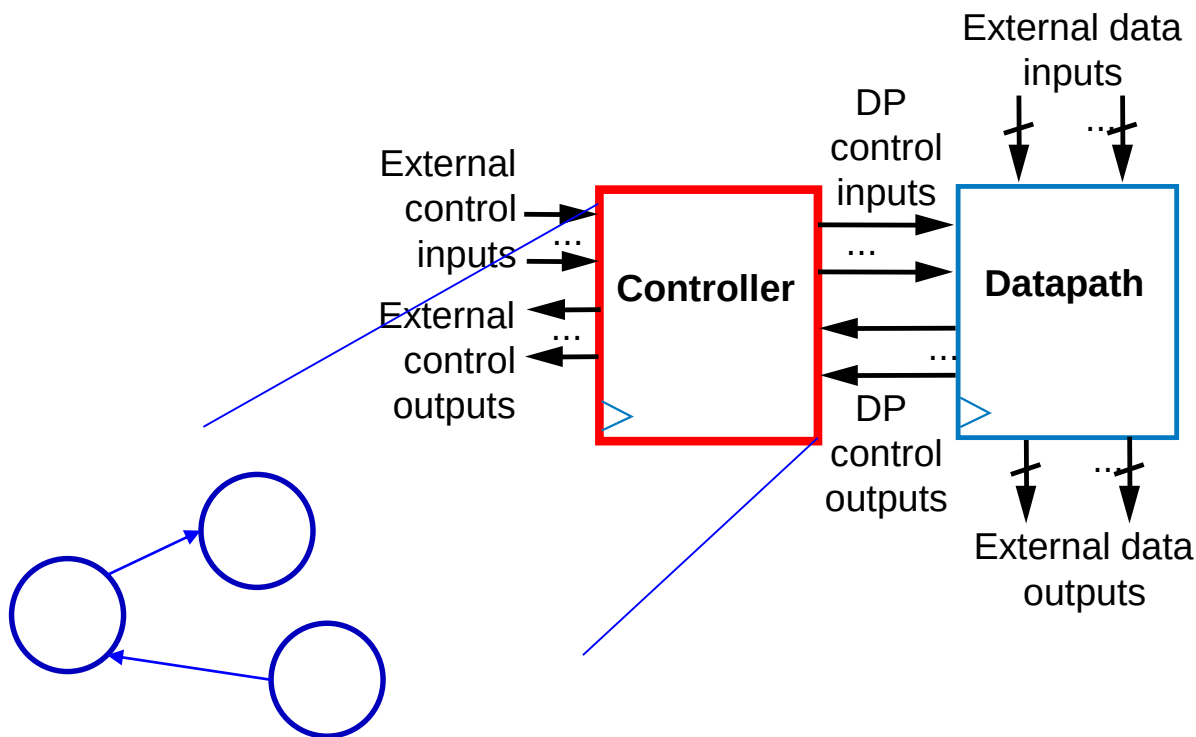


RTL Design Process

Step 2: Convert it to a circuit

- 2.c: derive the controller's FSM

- Convert the high-level state machine to a finite state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath

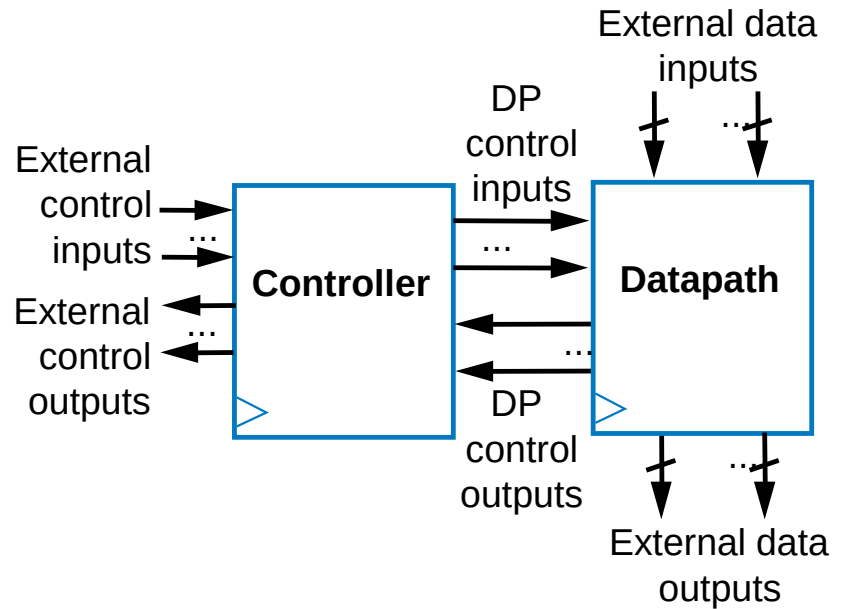


The controller FSM should have:

- Inputs:
 - Ext control inputs
 - DP control outputs
- Outputs:
 - Ext. control outputs
 - DP control inputs

RTL Design Process: summary

- Capture the behavior with HLSM
- Convert it to a circuit
 - High-level architecture (datapath and control path)
 - Datapath capable of HLSM's data operations
 - Design controller to control the datapath



Step 2.a: Create Datapath for Soda Dispenser

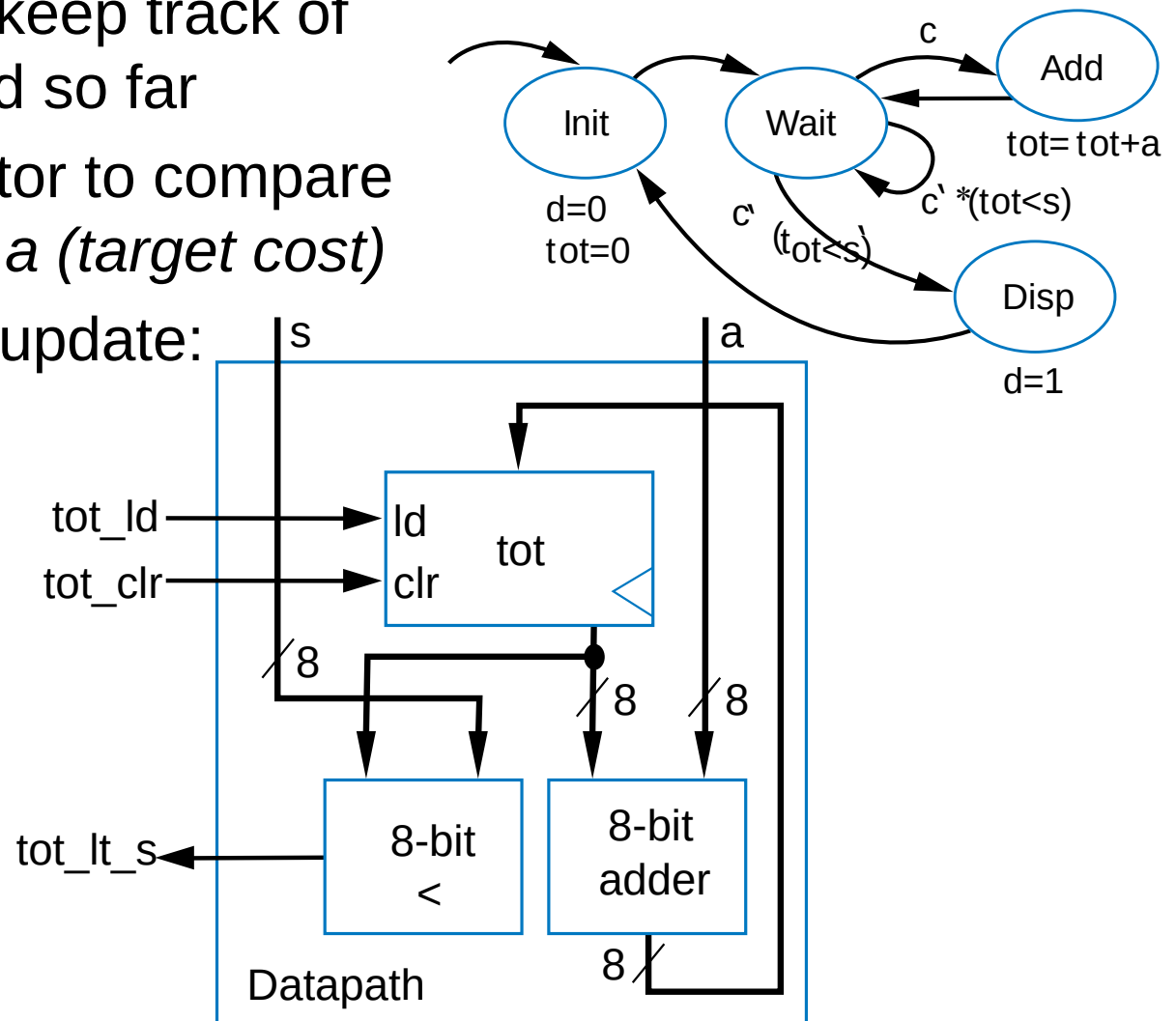


- Need *tot* register to keep track of the money deposited so far
- Need 8-bit comparator to compare *s* (current sum) and *a* (*target cost*)
- Need 8-bit adder to update: $tot = tot + a$
- Connect everything
- Create control IO

Inputs: *c* (bit), *a* (8 bits), *s* (8 bits)

Outputs: *d* (bit)

Local registers: *tot* (8 bits)

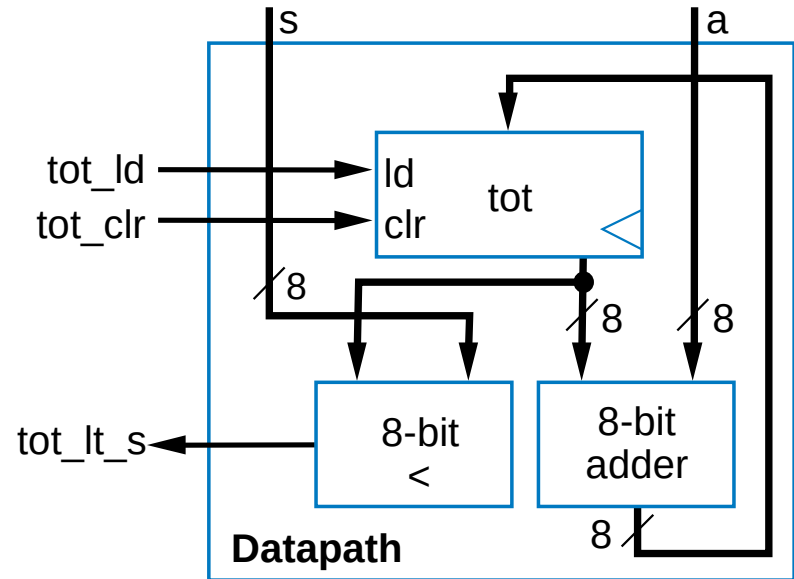
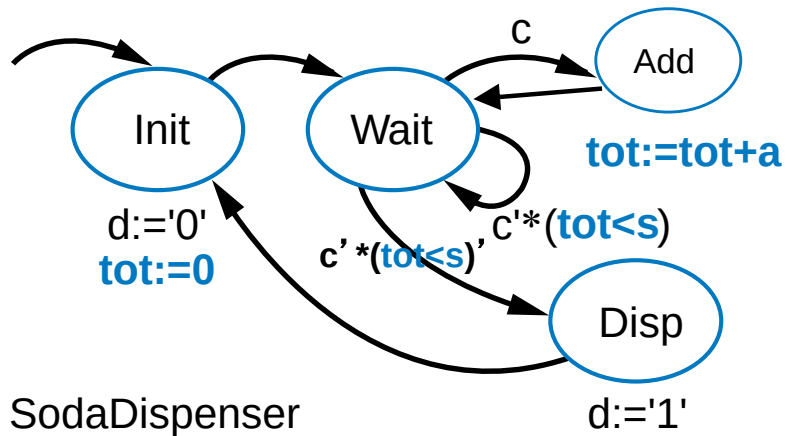


Signals in Soda Dispenser

Inputs: c (bit), a (8 bits), s (8 bits)

Outputs: d (bit) // '1' dispenses soda

Local storage: tot (8 bits)

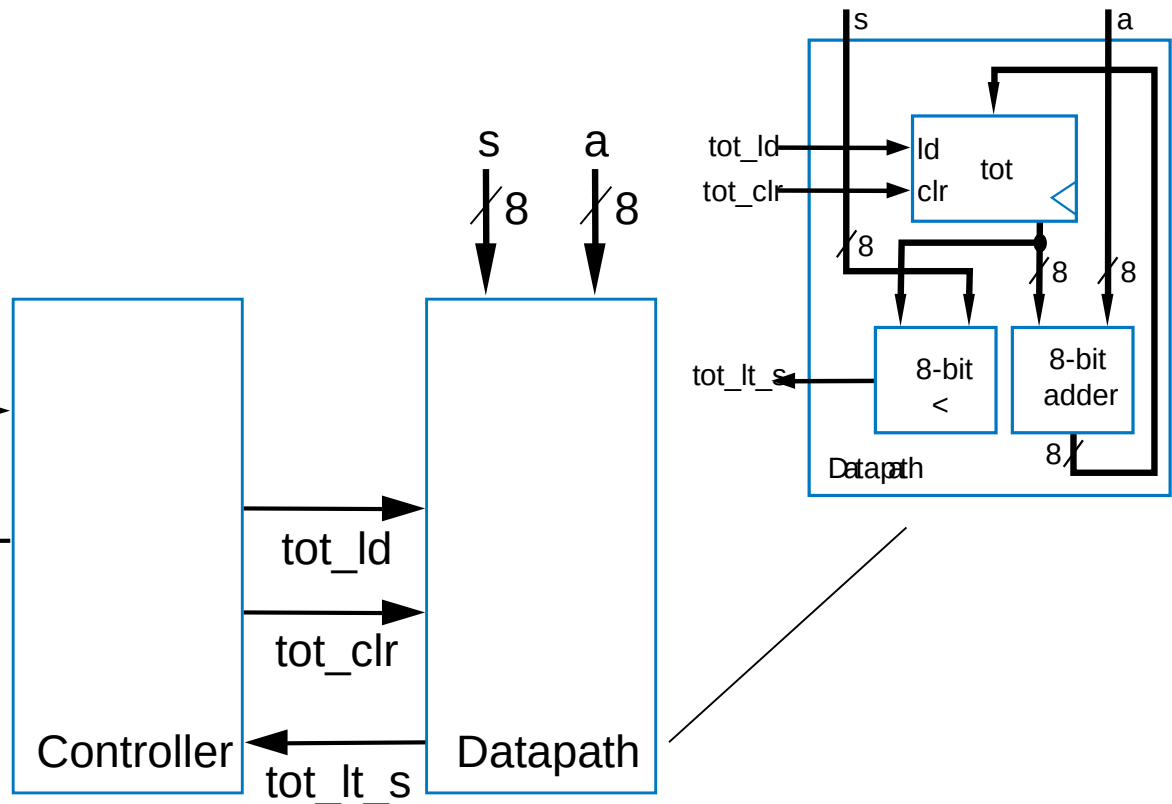


According to the current design, under which of the following conditions does the register output 'tot' change at the rising clock edge?

- A. Whenever the value of the coin inserted ('a') changes
- B. Whenever the cost of the soda ('s') changes
- C. When the signal tot_ld becomes high
- D. When the signal tot_clr becomes high
- E. Both C. & D.

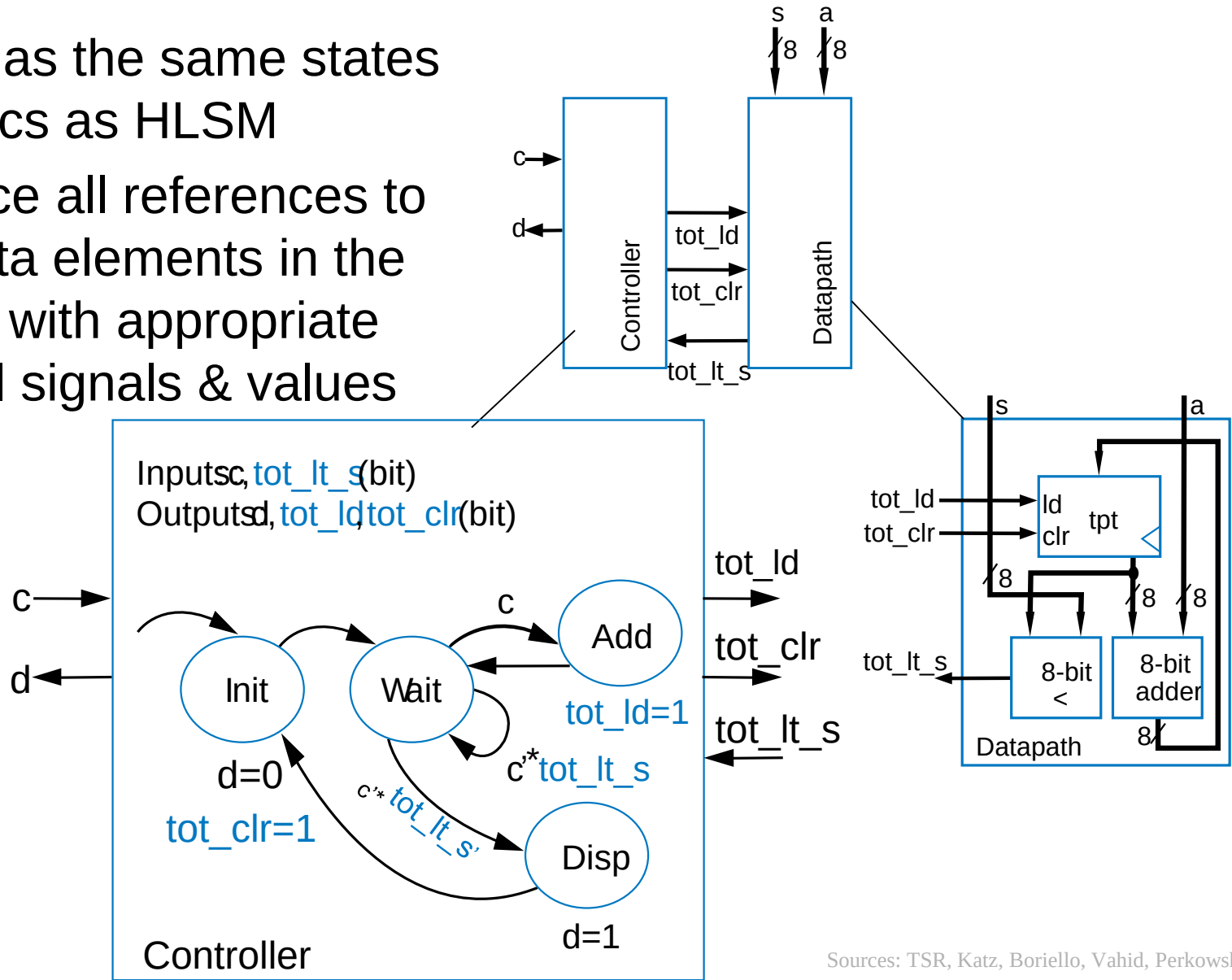
Step 2.b: Connect Datapath to a Controller

- Controller's inputs
 - External input c (coin detected)
 - Input from datapath comparator's output, which we named tot_lt_s
- Controller's outputs
 - External output d (dispense soda)
 - Outputs to datapath to load and clear the tot register



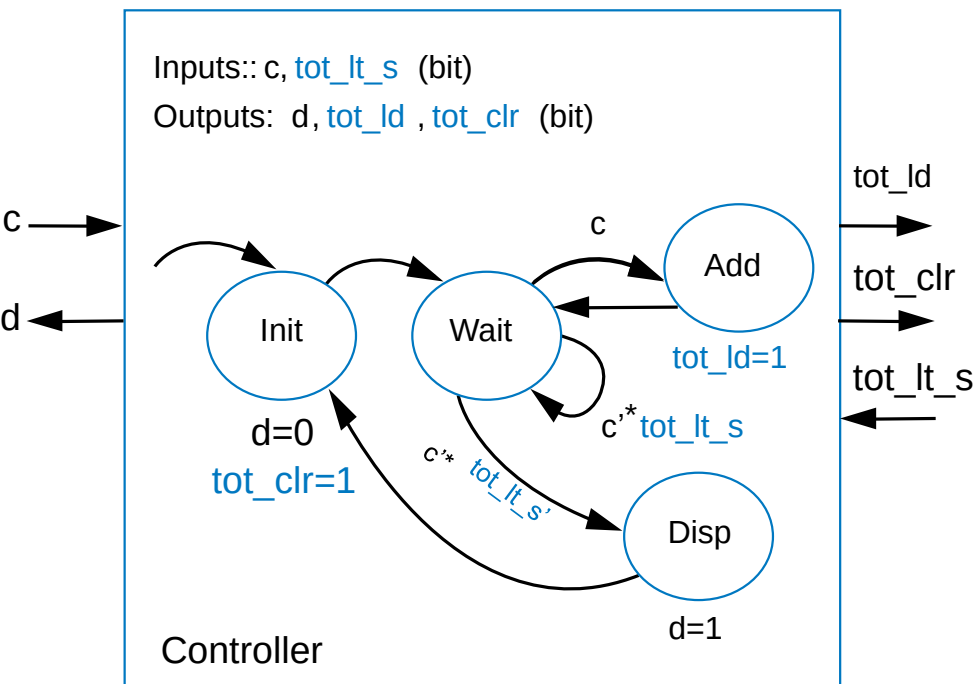
Step 2.c – Derive the Controller's FSM

- FSM has the same states and arcs as HLSM
- Replace all references to the data elements in the HLSM with appropriate control signals & values



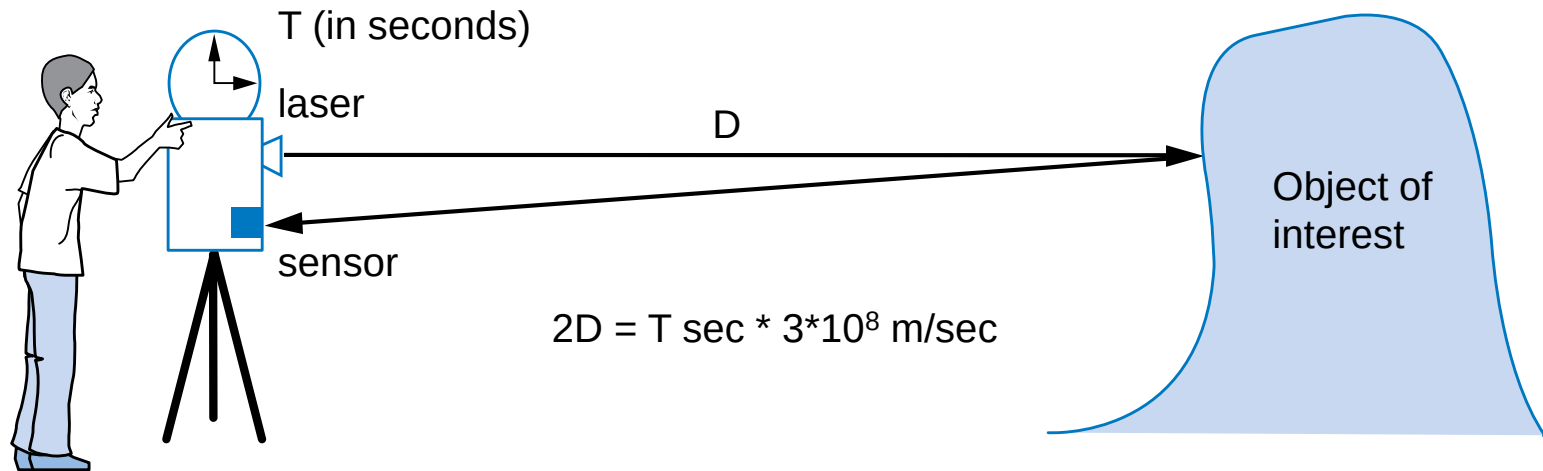
Final Step: Implement the controller FSM

Implement the FSM as a state register and logic



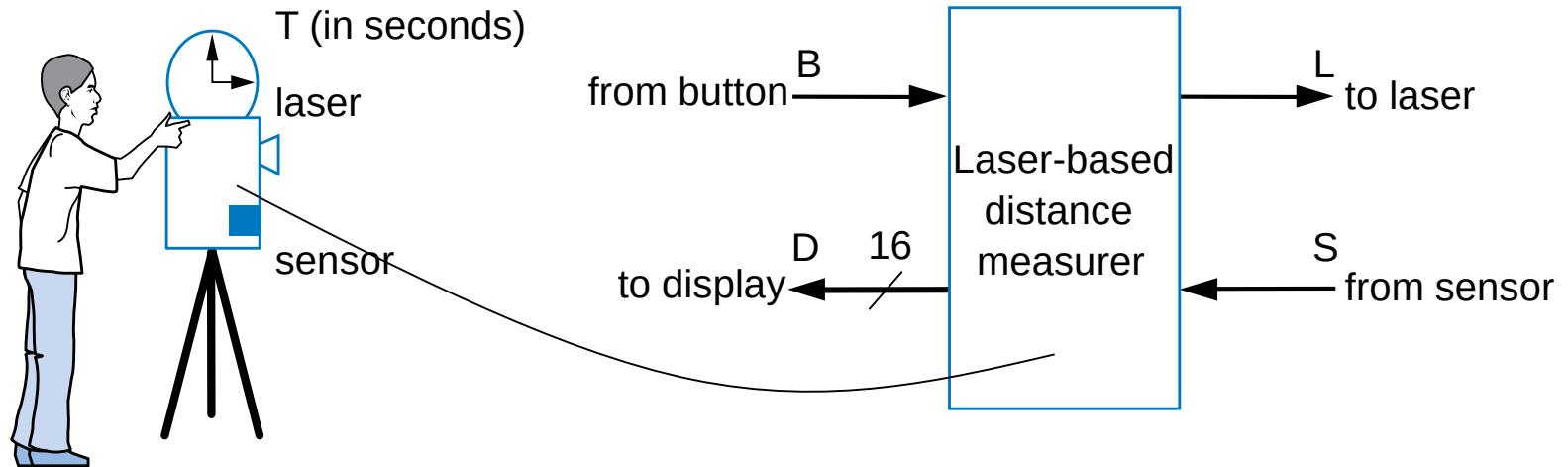
	s1	s0	c	tot_lt_s	n1	n0	d	tot_ld	tot_clr
Init	0	0	0	0	0	1	0	0	1
	0	0	0	1	0	1	0	0	1
	0	0	1	0	0	1	0	0	1
	0	0	1	1	0	1	0	0	1
Wait	0	1	0	0	1	1	0	0	0
	0	1	0	1	0	1	0	0	0
	0	1	1	0	1	0	0	0	0
	0	1	1	1	1	0	0	0	0
Add	1	0	0	0	0	1	0	1	0
	...	...	...	...	...	...	...	...	...
Disp	1	1	0	0	0	0	1	0	0
	...	...	...	...	...	...	...	...	...

Another RTL Design: Laser-Based Distance Measurer



- Laser-based distance measurement – pulse laser, measure time T to sense reflection
 - Laser light travels at speed of light, $3 \cdot 10^8 \text{ m/sec}$
 - Distance is thus $D = T \text{ sec} * 3 \cdot 10^8 \text{ m/sec} / 2$

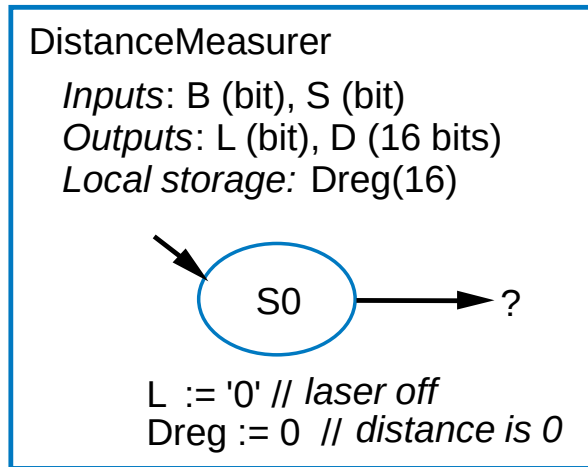
Laser-Based Distance Measurer IO



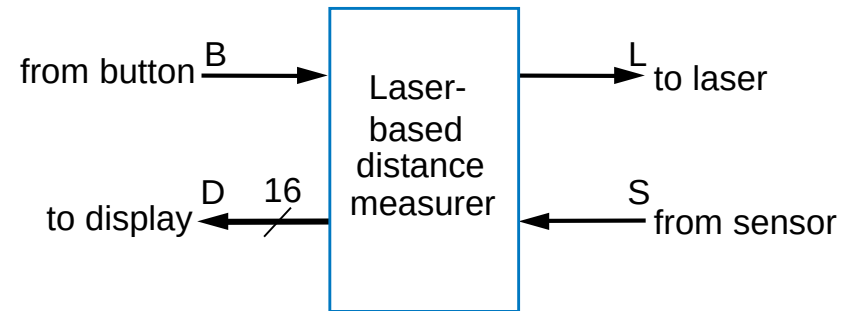
- Inputs/outputs

- B : bit input, from button, to begin measurement
- L : bit output, activates laser
- S : bit input, senses laser reflection
- D : 16-bit output, to display computed distance

Laser-Based Distance Measurer: HLSM



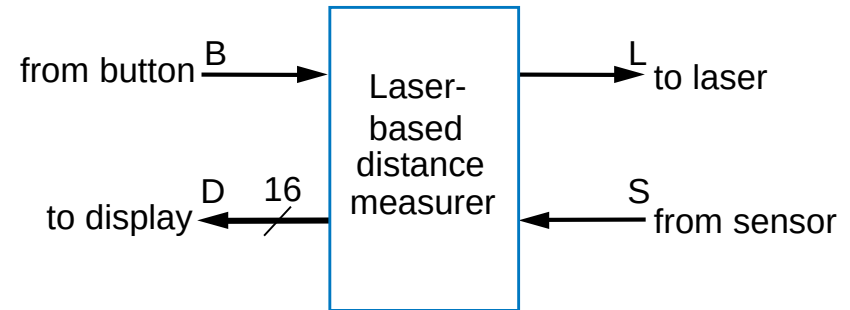
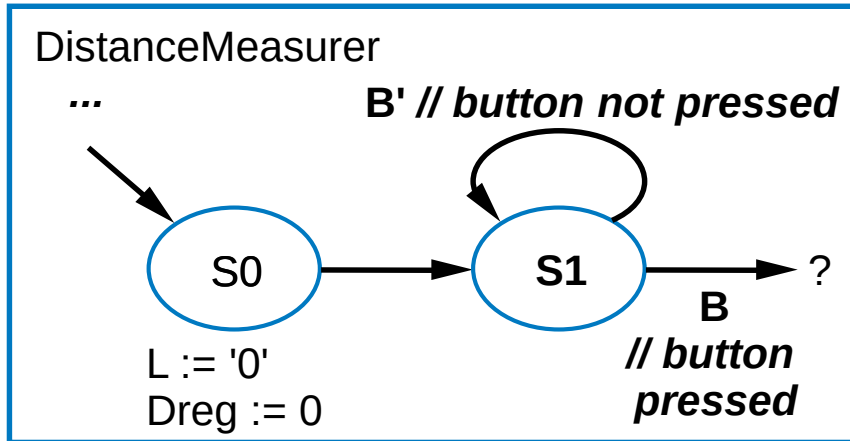
(first state usually initializes the system)



- Declare inputs, outputs, and local storage
 - Dreg required for multi-bit output
- Create initial state, name it **S0**
 - Initialize laser to off ($L := '0'$)
 - Initialize displayed distance to 0 ($Dreg := 0$)

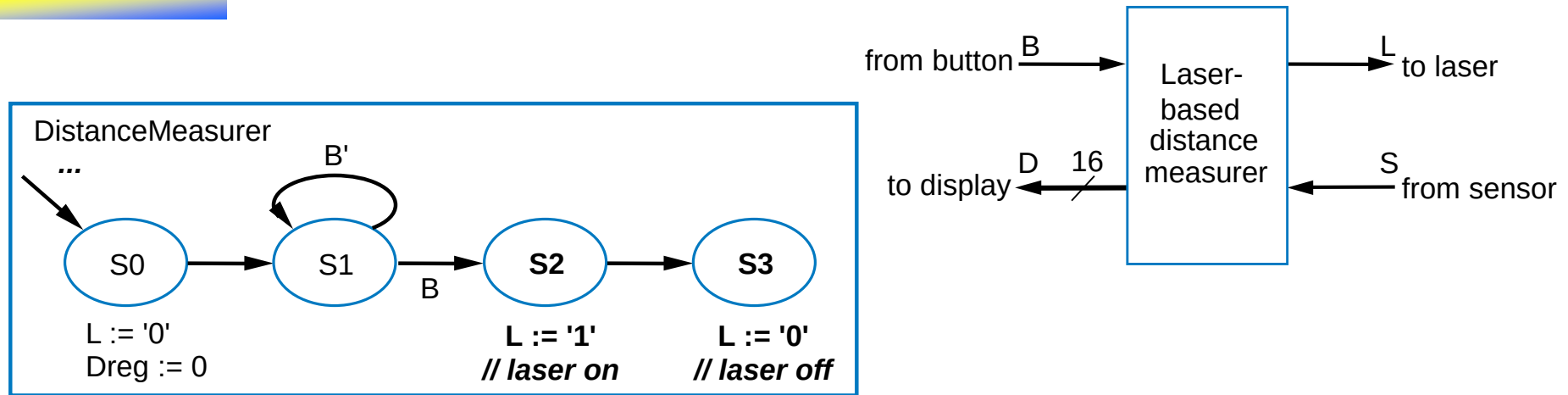
Recall: '0' means single bit, 0 means integer

Laser-Based Distance Measurer: HLSM



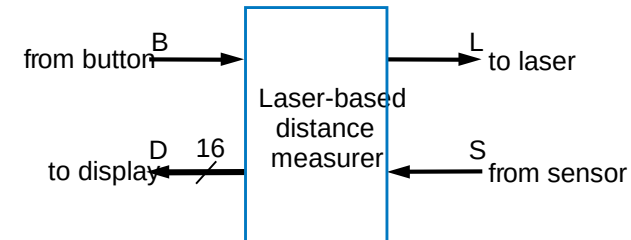
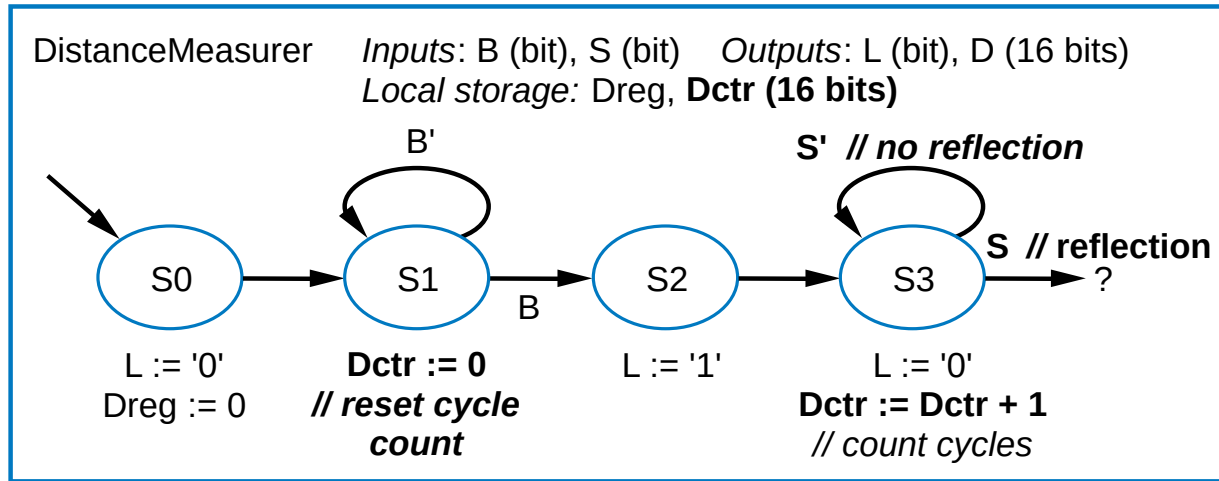
- Add another state, **S1**, that waits for a button press
 - B' – stay in **S1**, keep waiting
 - B – go to a new state **S2**

Laser-Based Distance Measurer: HLSM



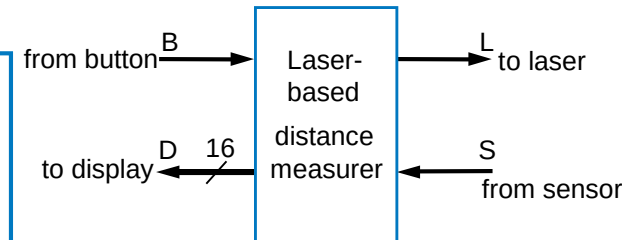
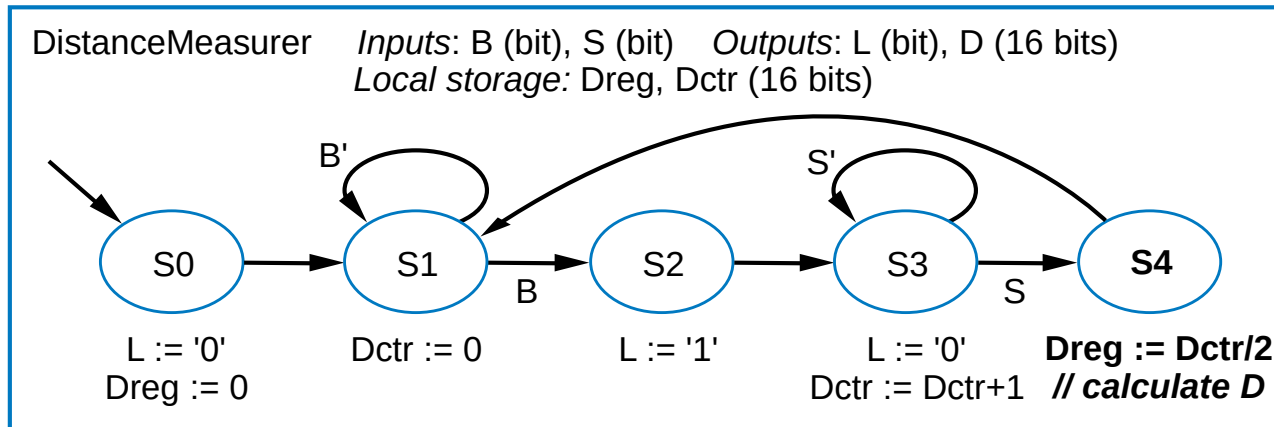
- Add a state **S2** that turns on the laser (L:='1')
- Then turn off laser (L:='0') in a state **S3**

Laser-Based Distance Measurer: HLSM



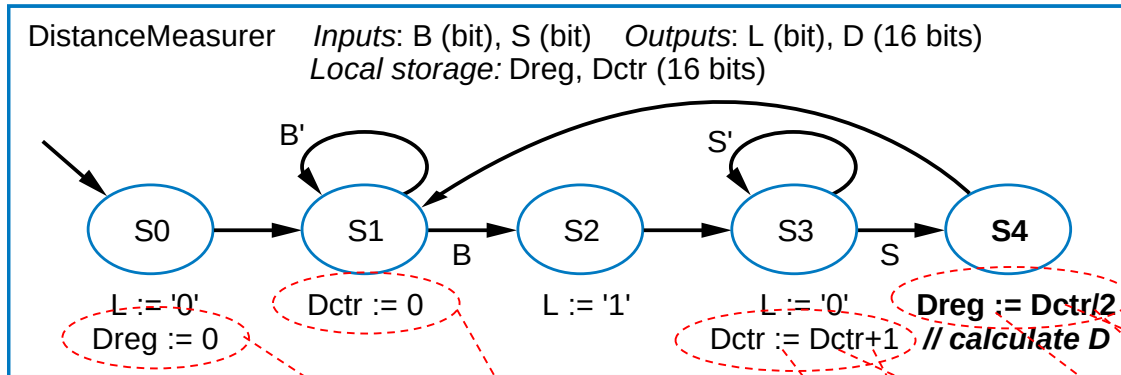
- Stay in **S3** until sense reflection (S)
- To measure time, count cycles while in **S3**
 - To count, declare local storage *Dctr*
 - Initialize *Dctr* to 0 in **S1**. In **S2** would have been O.K. too.
 - Don't forget to initialize local storage—common mistake
 - Increment *Dctr* each cycle in **S3**

Laser-Based Distance Measurer: HLSM

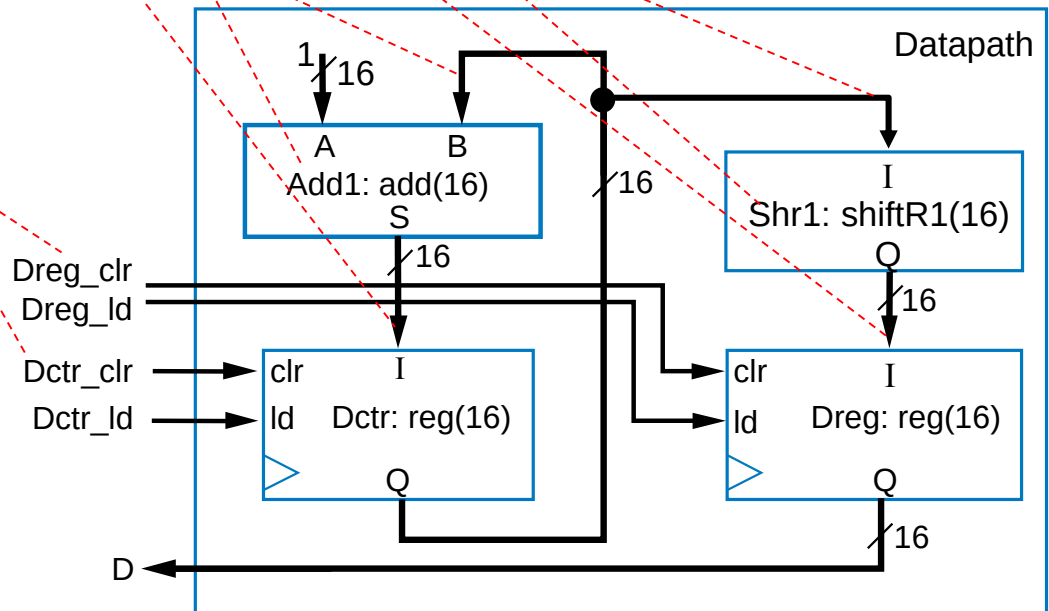


- Once reflection detected (S), go to new state **S4**
 - Calculate distance
 - Assuming clock frequency is 3×10^8 , $Dctr$ holds number of meters, so $Dreg := Dctr / 2$
- After **S4**, go back to **S1** to wait for button again

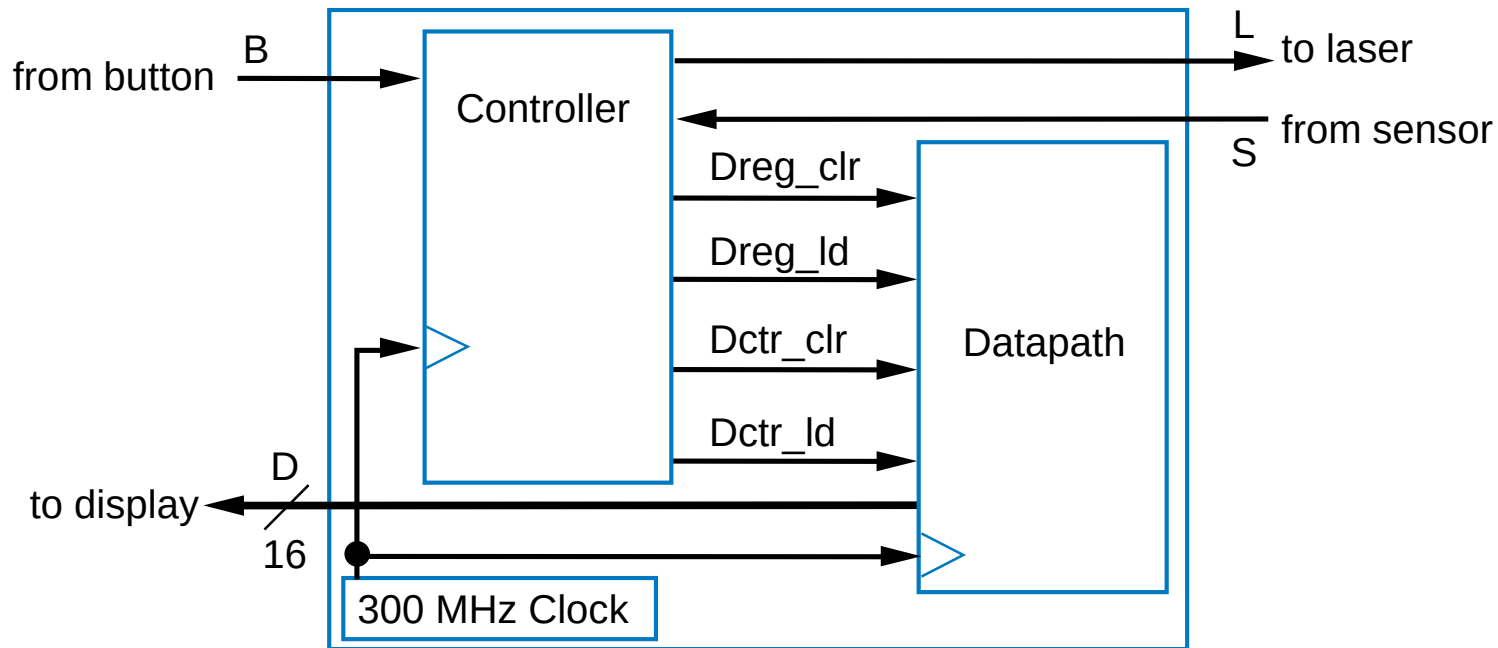
Laser-Based Distance Measurer: Create a Datapath



- HLSM data I/O $\rightarrow$ DP I/O
- HLSM local storage $\rightarrow$ reg
- HLSM state action and transition condition data computation $\rightarrow$ Datapath components and connections



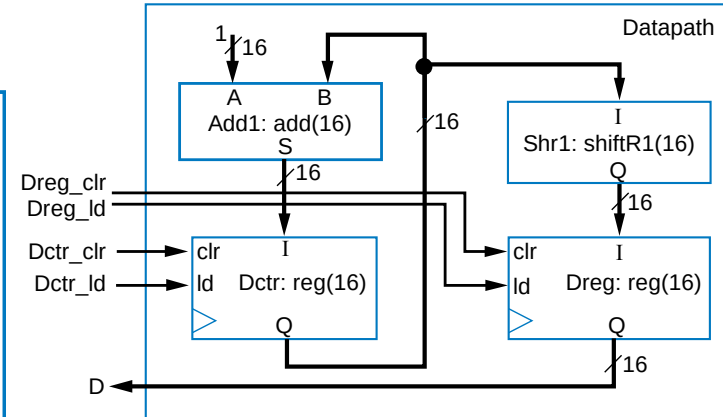
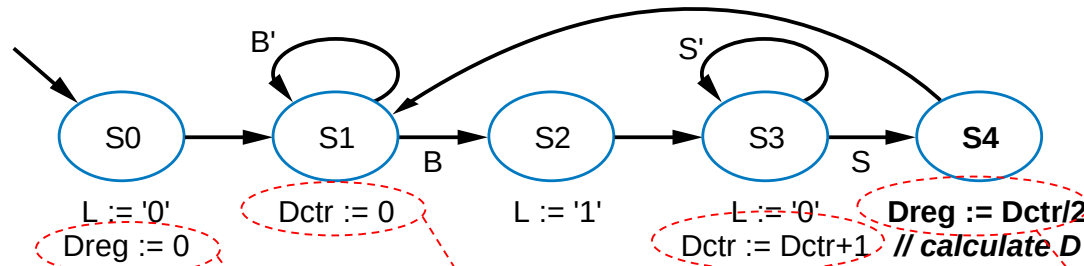
Laser-Based Distance Measure: Connecting the Datapath to a Controller



Laser-Based Distance Measurer: Derive the Controller FSM

HLSM

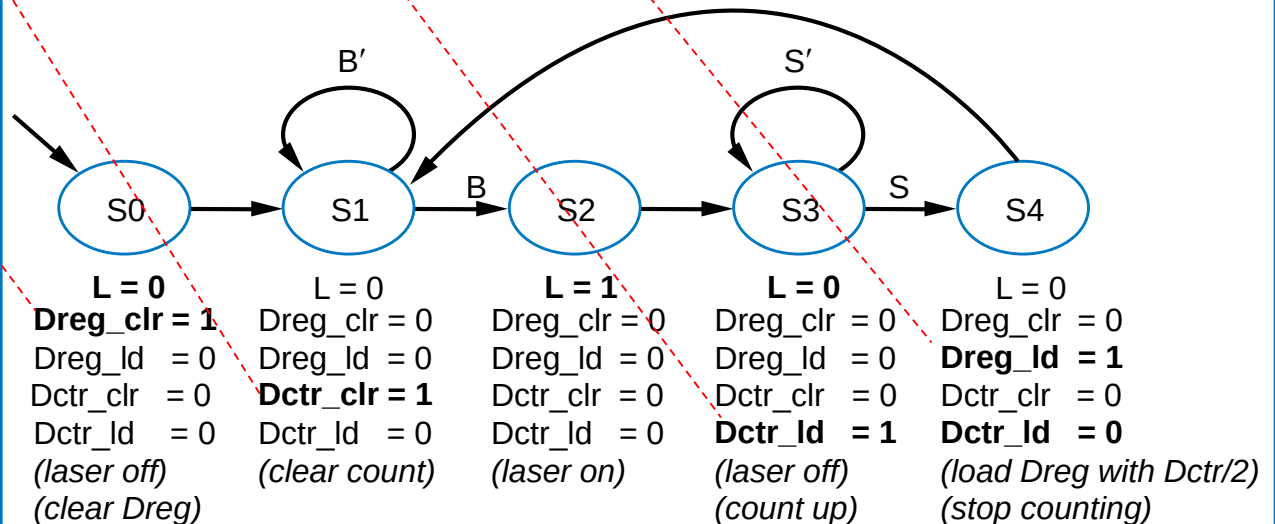
DistanceMeasurer Inputs: B (bit), S (bit) Outputs: L (bit), D (16 bits)
Local storage: Dreg, Dctr (16 bits)



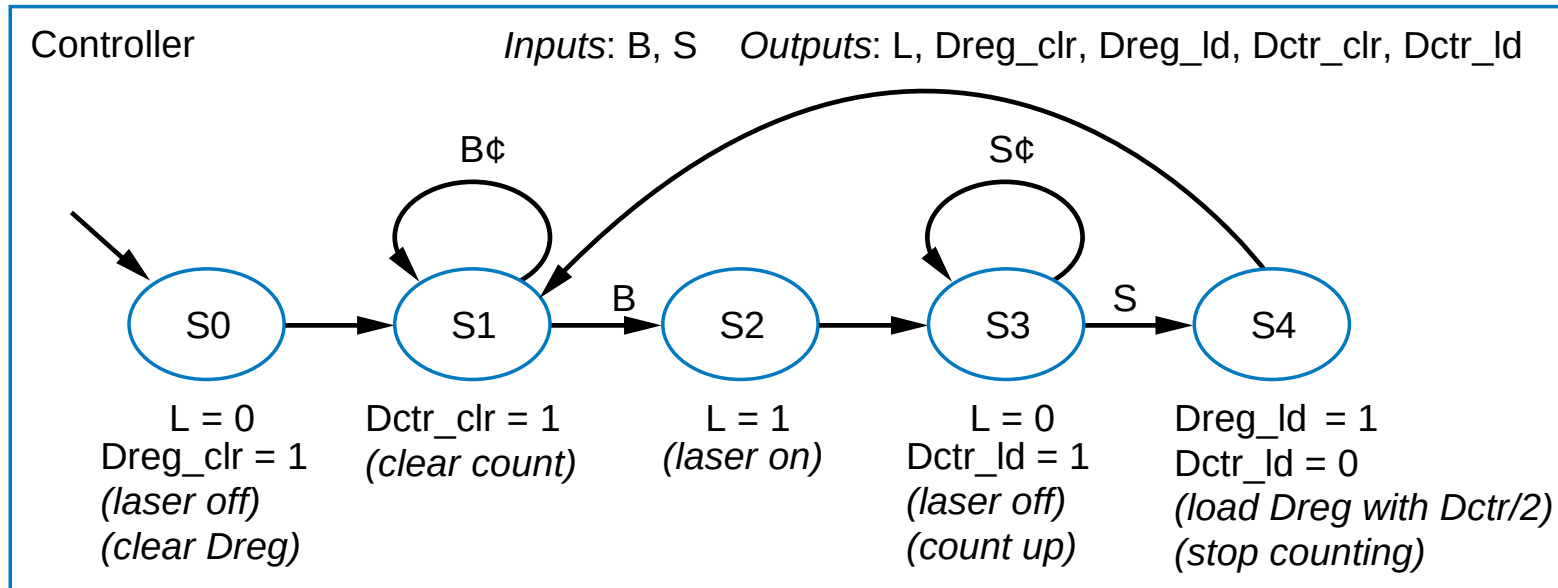
- FSM has same states, transitions, and control I/O
- Achieve each HLSM data operation using datapath control signals in FSM

Controller

Inputs: B, S Outputs: L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_ld



Laser-Based Distance Measurer: Simplify the Controller FSM



- Same FSM, using convention of unassigned outputs implicitly assigned 0

Some assignments to 0 still shown, due to their importance in understanding desired controller behavior