

# Schedule



- Week of 10/23: RTL design/HDL implementation
- Week of 10/30: lab 8 – laser distance measurer RTL design
- Week of 11/6-11/20: final project – reaction timer
- Tuesday 11/28: lab hardware return

# CSEE 4280 – Advanced Logic Design



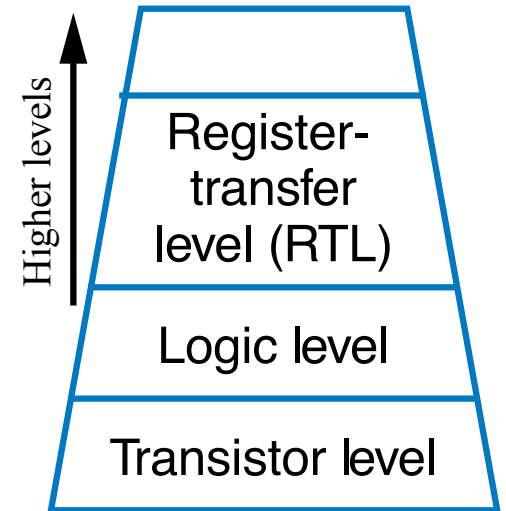
- Next semester
- Same room
- 40 seats
- Team of 2
- Focus on designing a **working CPU**, with **ALU**, **controller**, **datapath**, **memory**, etc., and a simple **instruction set**, all on a **FPGA** board, using **a combination of Verilog and graphic design** technique in Xilinx.

# **Design of Digital Systems**

Register Transfer Level (RTL) Design

# Introduction

- Combinational Logic Design
  - Capture Comb. behavior: Equations, truth tables
  - Convert to circuit: AND + OR + NOT → Comb. logic
- Sequential Logic Design
  - Capture basic sequential behavior: FSMs
  - Convert to circuit: Register + Comb. logic → Controller
- Datapath
  - Datapath components, simple datapaths
- *What if I/O, State Action, Transition involve high level data type (binary, integer, etc)?*
- Now
  - Capture behavior: **High-level state machine**
  - Convert to circuit: **Controller + Datapath → Processor**
  - Known as “RTL” (register-transfer level) design
    - Transferring data from registers, through datapath components, and back to registers.



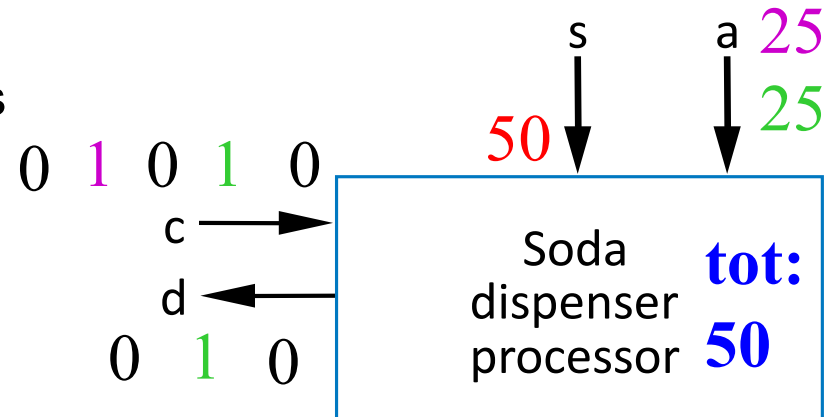
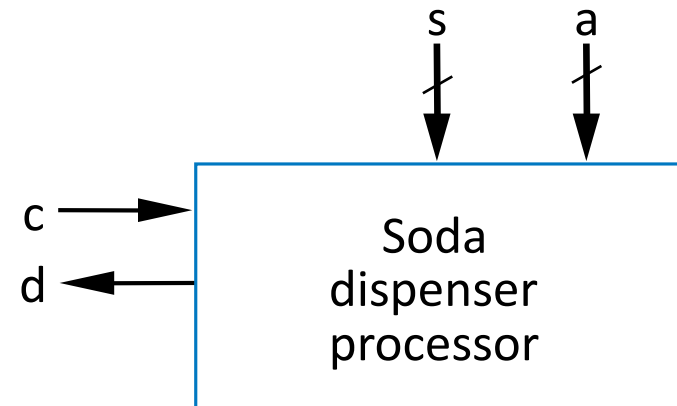
Levels of digital design abstraction

Processors:

- Programmable (microprocessor)
- Custom

# High-Level State Machines (HLSMs)

- Some behaviors too complex for equations, truth tables, or FSMs.  
**We need to use high-level state machine to capture them.**
- Ex: Soda dispenser
  - $c$ : bit input, 1 when coin deposited
  - $a$ : 8-bit input having value of deposited coin
  - $s$ : 8-bit input having cost of a soda
  - $d$ : bit output, processor sets to 1 when total value of deposited coins equals or exceeds cost of a soda
- FSM can't represent...
  - 8-bit input/output**
  - Storage of current total**
  - Addition (e.g.,  $25 + 10$ )**



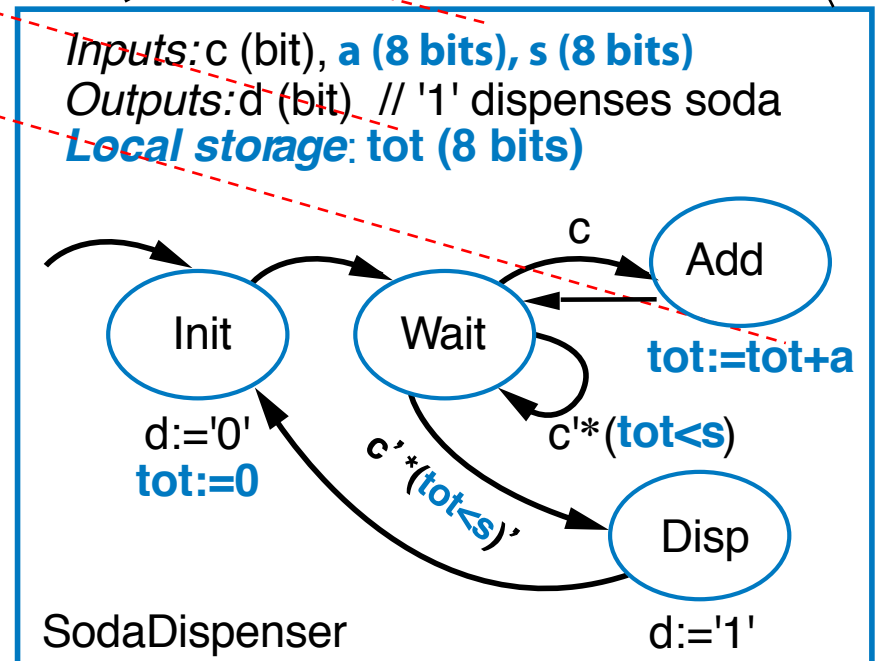
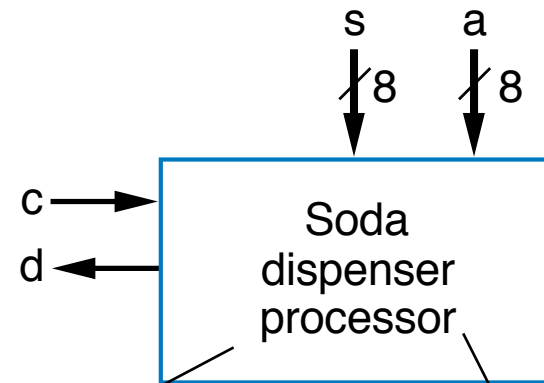
# HLSMs

- High-level state machine (HLSM) extends FSM with:

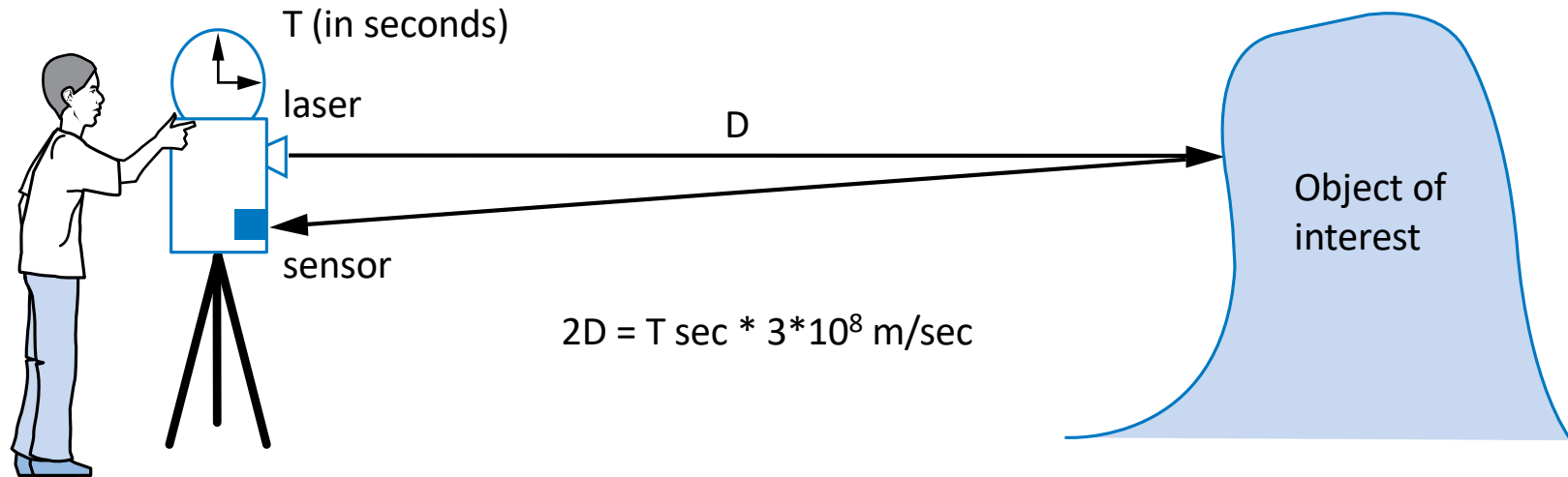
- Multi-bit input/output
- Local storage
- Arithmetic operations

- Conventions

- Numbers:
  - Single-bit: '0' (single quotes)
  - Integer: 0 (no quotes)
  - Multi-bit: "0000" (double quotes)
- == for equal (comparison), := for assignment
- Multi-bit outputs *must* be registered via local storage
- // precedes a comment
- Each transition is implicitly ANDed with a rising clock edge (synchronous)
- Any *bit* output not explicitly assigned a value in a state is implicitly assigned a '0'

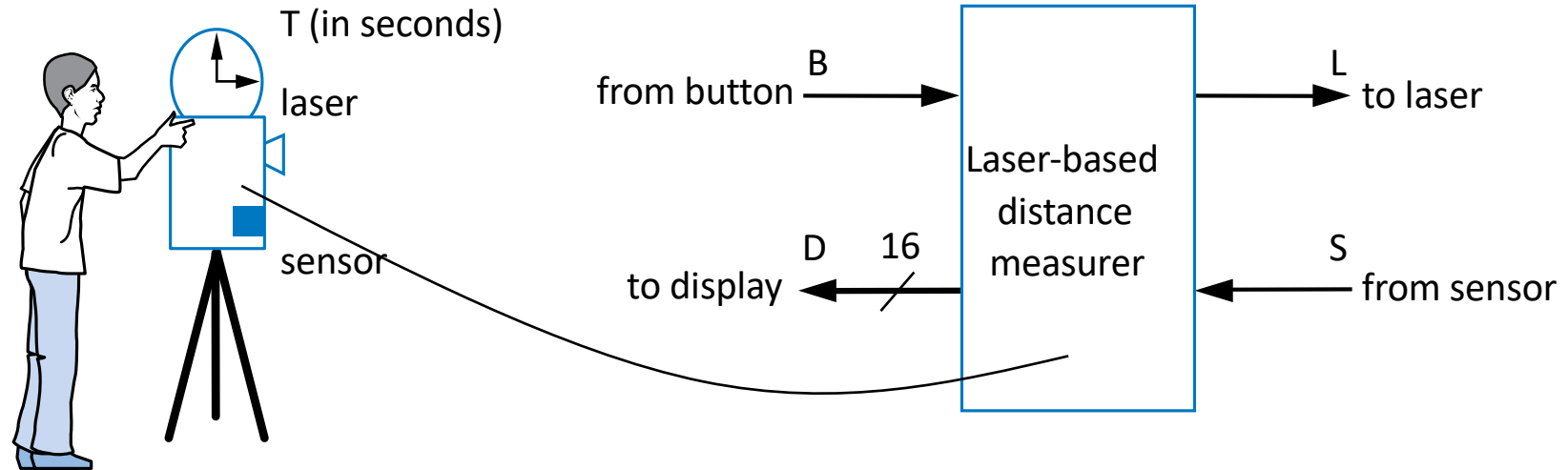


# Example 2: Laser-Based Distance Measurer



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

# Example: Laser-Based Distance Measurer



- 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

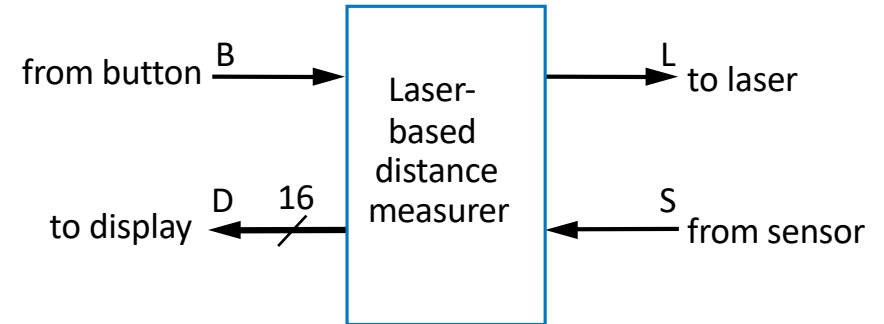
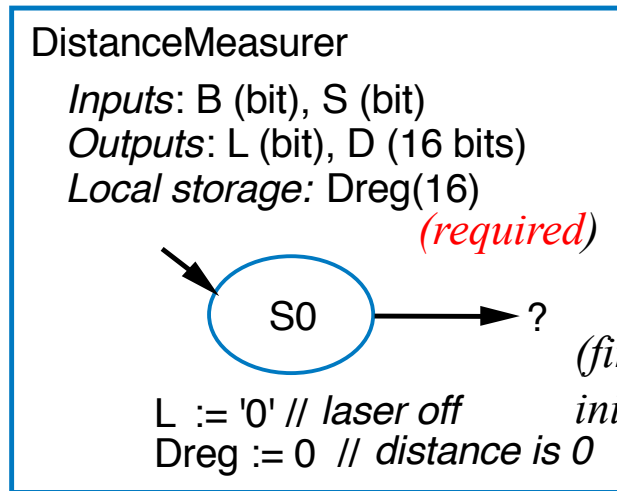


# Example: Laser-Based Distance Measurer



- **Sequence of events**
  - The system powers on, laser is initially off and system outputs a distance of 0 meter
  - The system waits for the user to press button B to start measurement
  - After button is pressed, the system turns on the laser (for one clock cycle)
  - After laser is on, the system waits for the sensor to detect the laser's reflection. Meanwhile, the system should count how many clock cycles occur from the time the laser was on until the reflection is detected.
  - After the reflection is detected, the system should use the number of cycles counted to calculate the distance of interest. The system should then return to waiting state for next measurement.

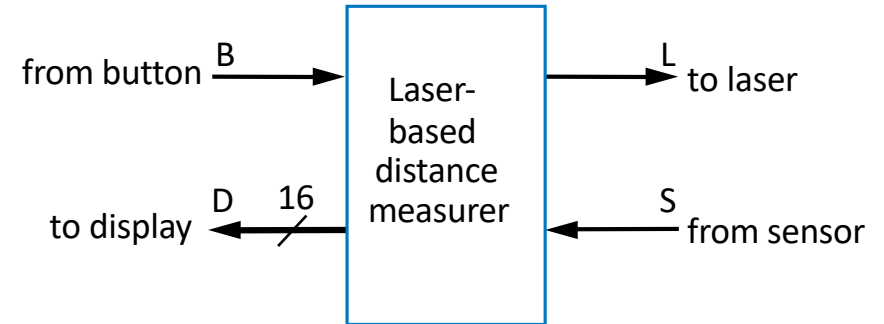
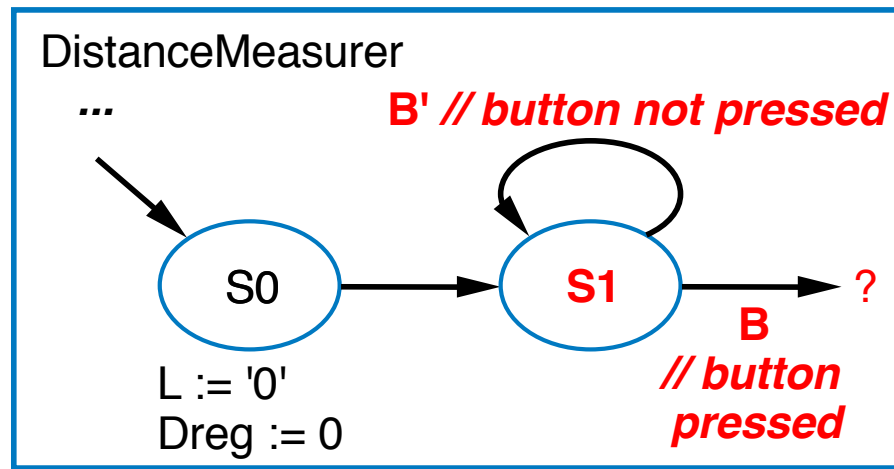
# Example: Laser-Based Distance Measurer



- 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*

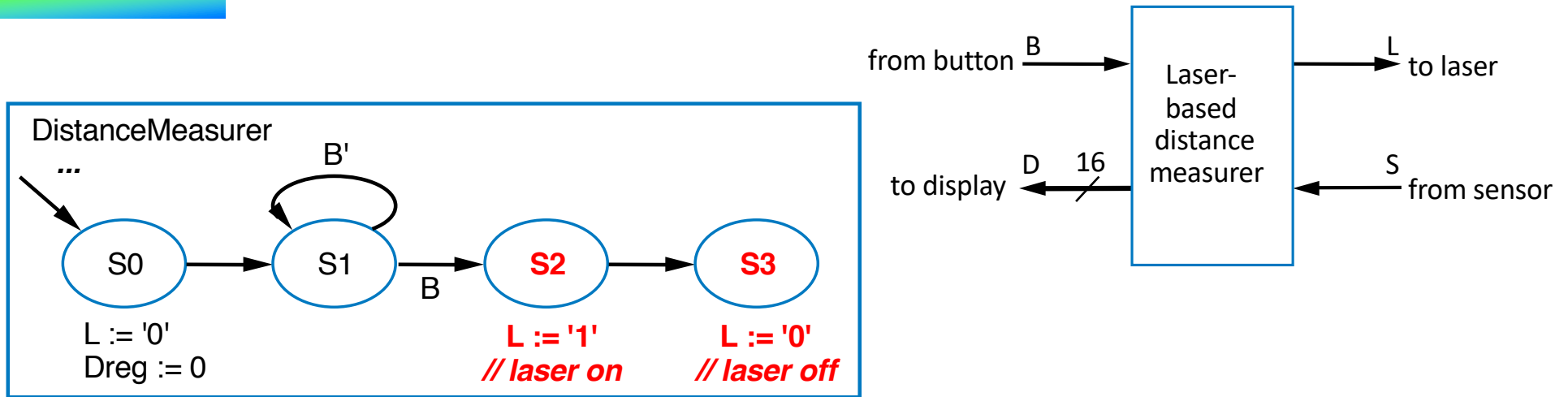
# Example: Laser-Based Distance Measurer



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

Q: What should S2 do?    A: Turn on the laser

# Example: Laser-Based Distance Measurer

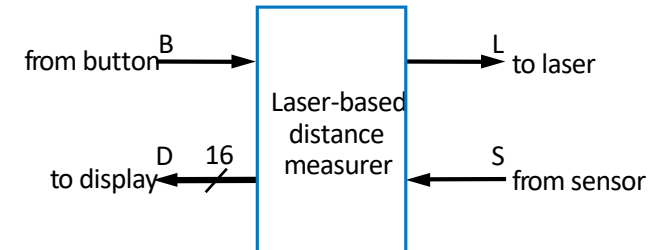
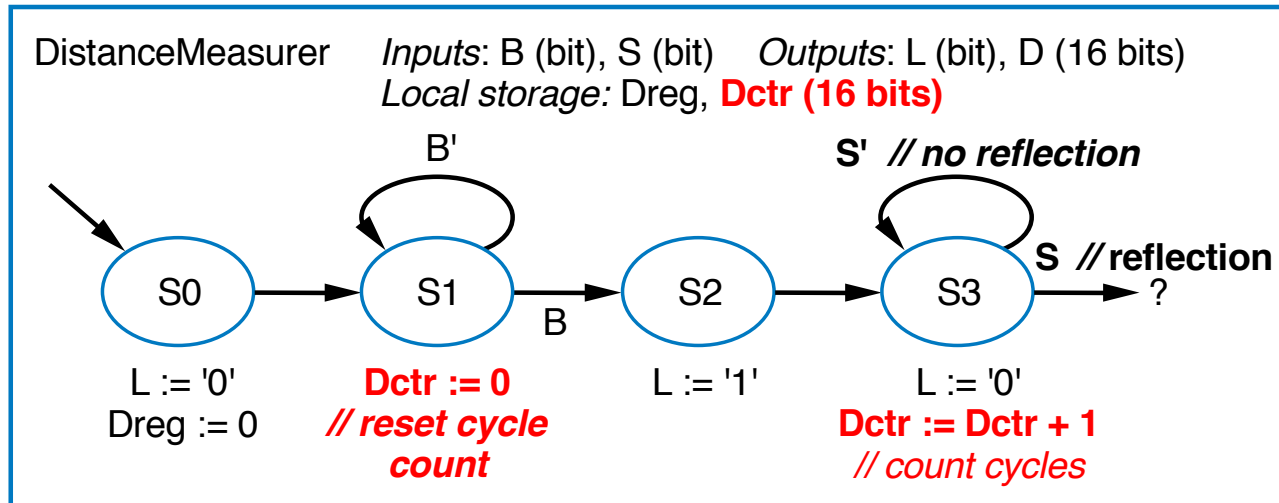


- Add a state **S2** that turns on the laser ( $L := '1'$ )
- Then turn off laser ( $L := '0'$ ) in a state **S3**

**Q: What do next?**    A: Start timer, wait to sense reflection

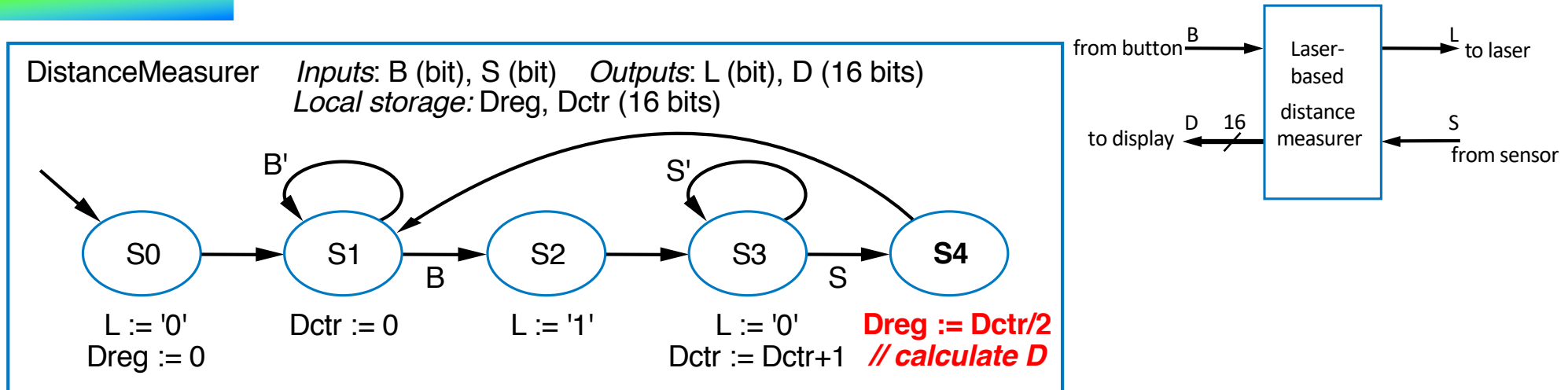
*a*

# Example: Laser-Based Distance Measurer



- 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**

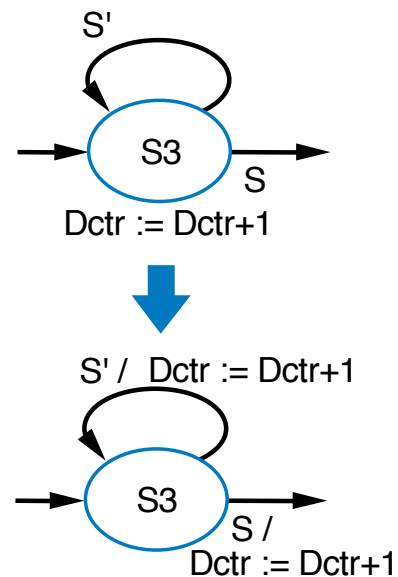
# Example: Laser-Based Distance Measurer



- Once reflection detected (S), go to new state **S4**
  - Calculate distance
  - Assuming clock frequency is  $3 \times 10^8$  (300 MHz), *Dctr* holds number of meters, so  $Dreg := Dctr/2$
- After **S4**, go back to **S1** to wait for button again

# HLSM Actions: Updates Occur Next Clock Cycle

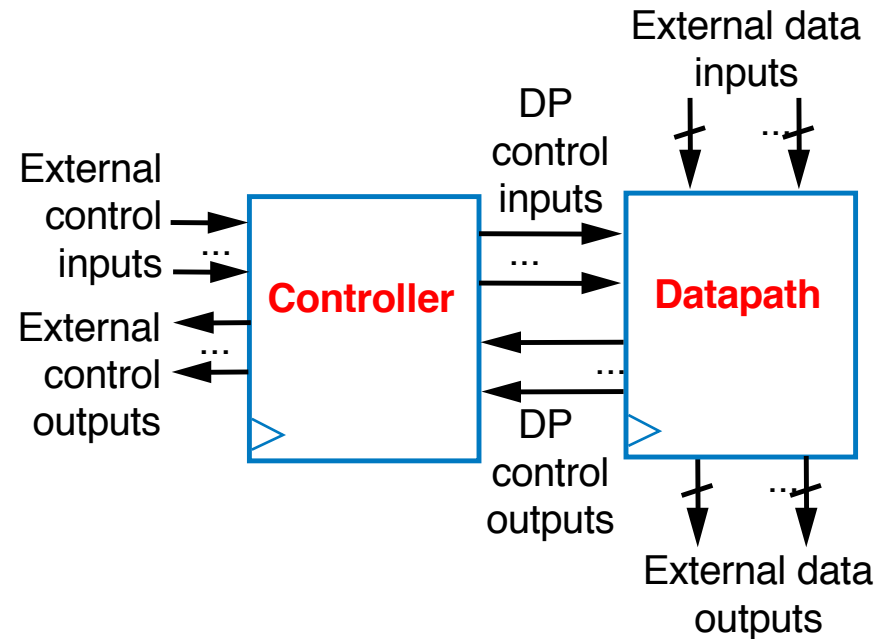
- Local storage updated on rising clock edges only
  - Enter state on clock edge
  - Storage writes in that state occur on *next* clock edge
  - Can think of as occurring on outgoing transitions
- Thus**, transition conditions use the OLD value, not the newly-written value.



When a rising clock edge causes a transition from **S2** to **S3**, **Dctr** will initially be 0. While in **S3** during that clock cycle, **Dctr** will still be 0. When the next rising clock edge arrives, **Dctr** will be updated to 1.

# RTL Design Process

- Capture behavior
  - HLASM
- Convert to circuit
  - Need standard processor architecture
  - Datapath capable of HLASM's data operations by having necessary **datapath components**
  - Controller to control **datapath components'** input signals to carry out specific actions.





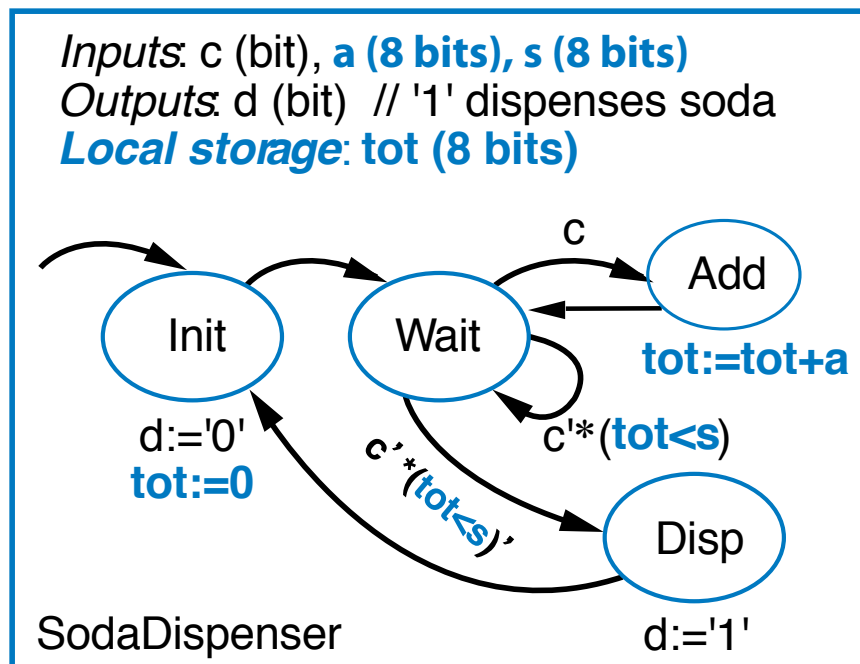
# RTL Design Process

	Step	Description
Step 1: Capture behavior	<i>Capture a high-level state machine</i>	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 inputs and outputs.
	2A <i>Create a datapath</i>	Create a datapath to carry out the data operations of the high-level state machine.
Step 2: Convert to circuit	2B <i>Connect the datapath to a controller</i>	Connect the datapath to a controller block. Connect external control inputs and outputs to the controller block.
	2C <i>Derive the controller's FSM</i>	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.

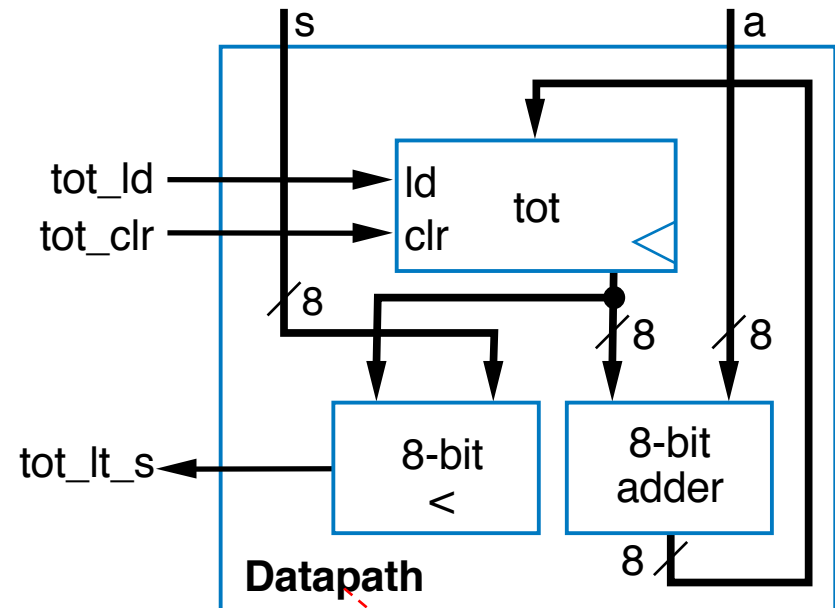
- Last step: select an appropriate clock frequency

# Example: Soda Dispenser from Earlier

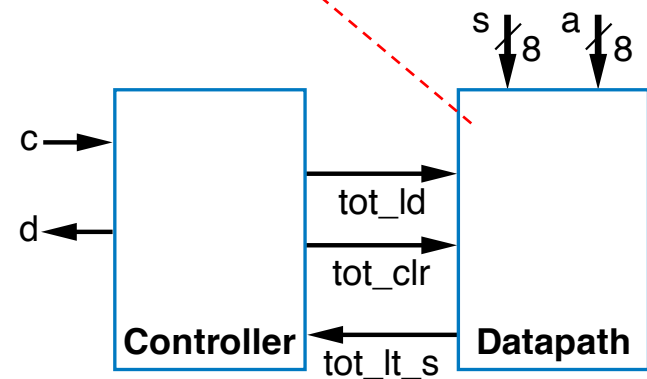
- Quick overview example.  
More details of each step to come.



Step 1



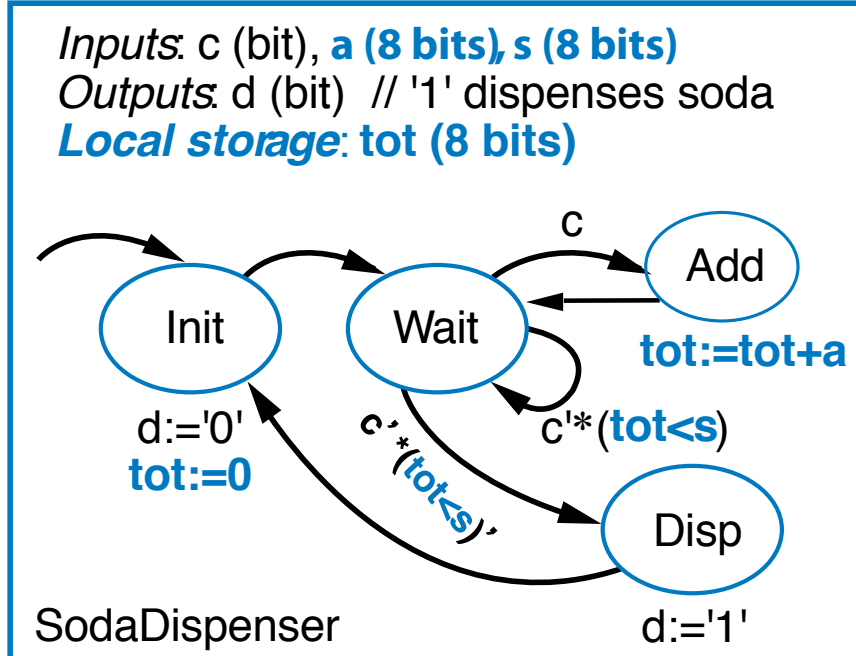
Step 2A



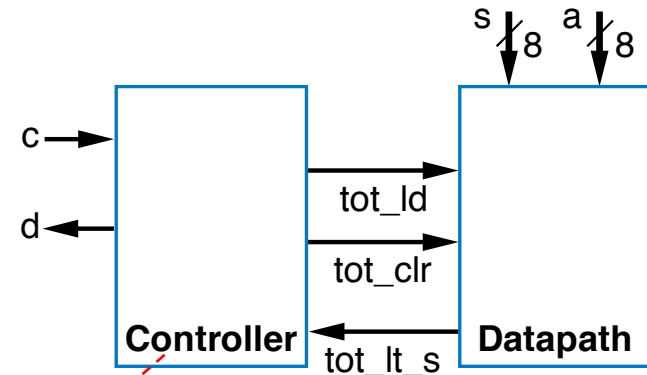
Step 2B

# Example: Soda Dispenser

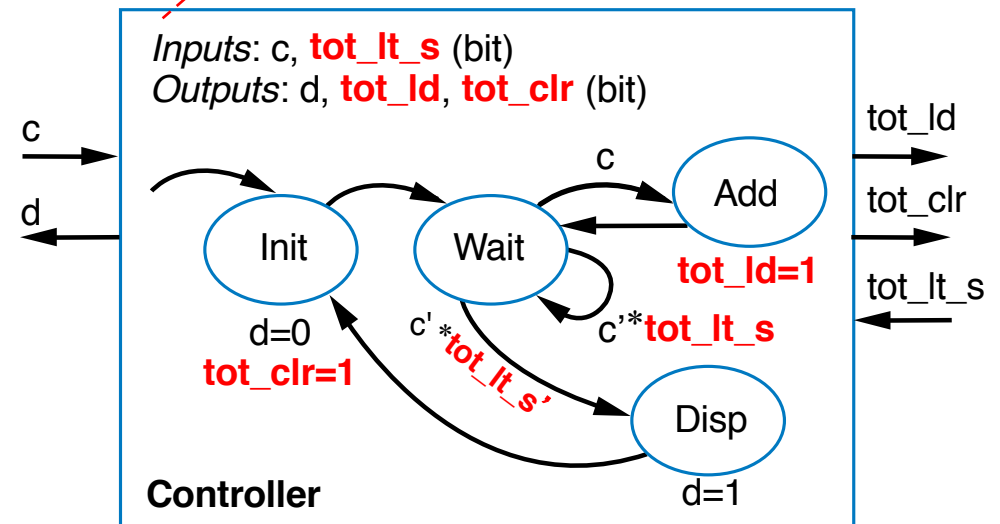
- Quick overview example.  
More details of each step to come.



Step 1



Step 2B

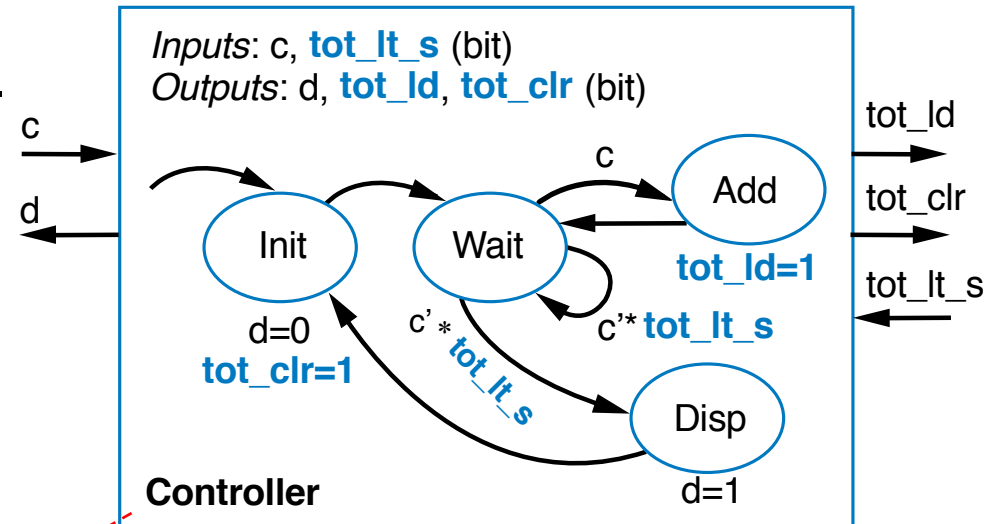


Step 2C

# Example: Soda Dispenser

- Quick overview example.  
More details of each step to come.

	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
		...				...			



Use controller design process  
(Sequential circuit) to complete the  
design

Init: 00; Wait: 01; Add: 10; Disp: 11

Detailed steps uploaded to ELC (in the lecture notes folder)

# Final Implementation of Soda Dispenser

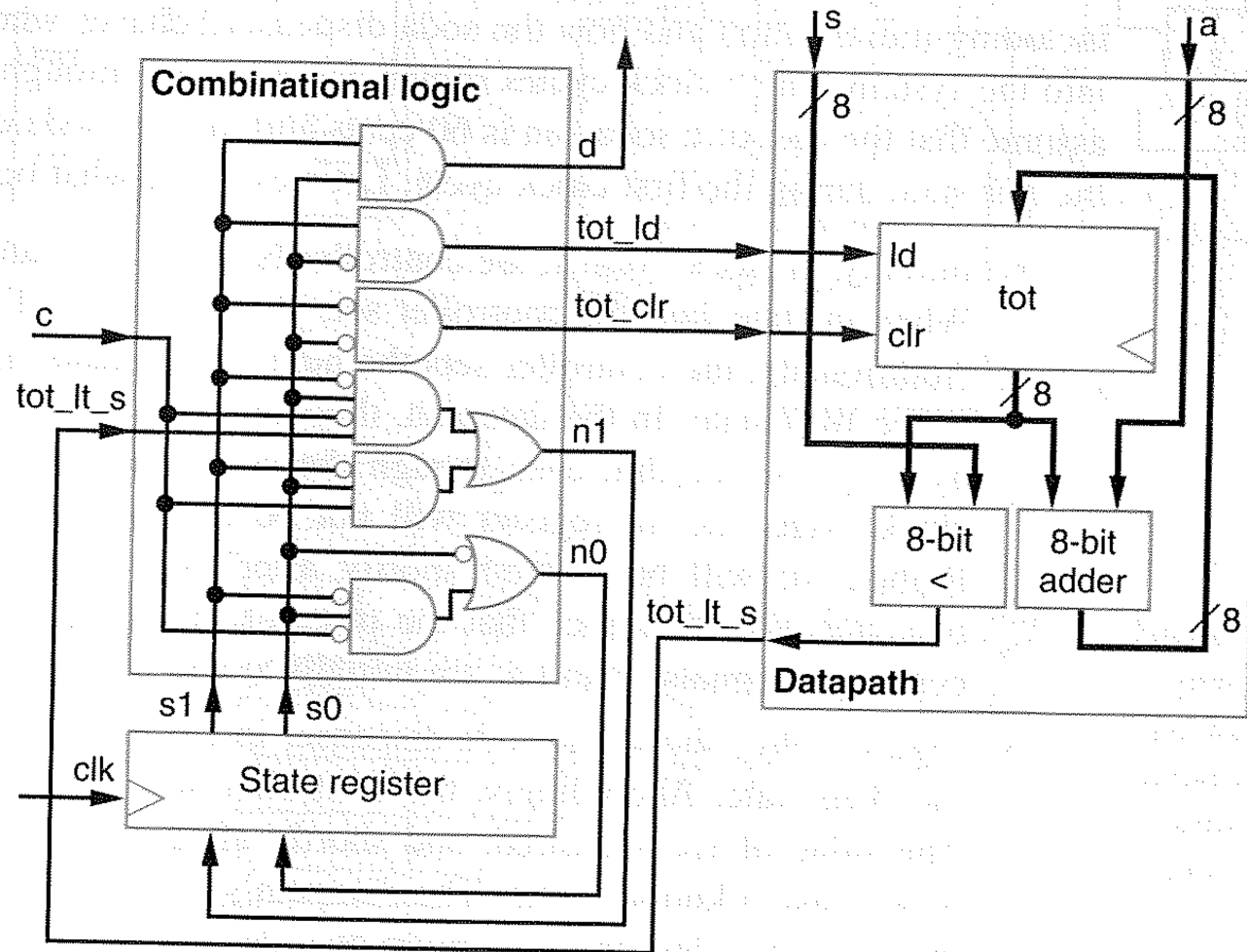


Figure C.8 Final implementation of the soda machine controller with datapath.

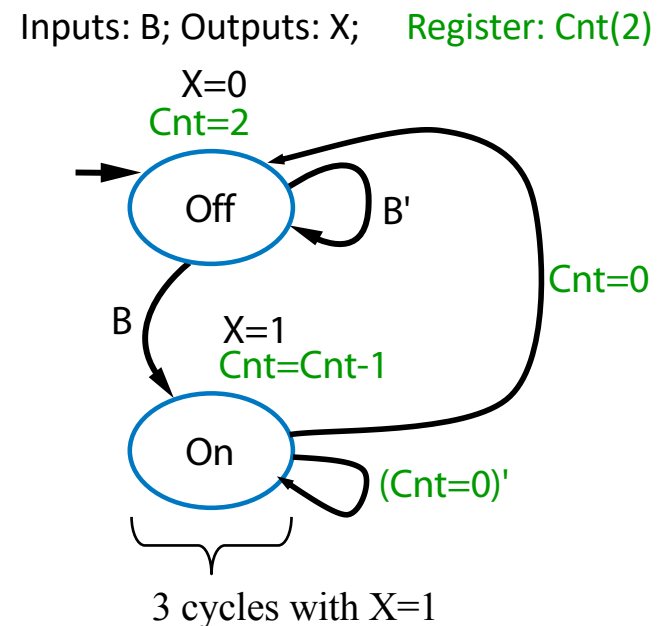
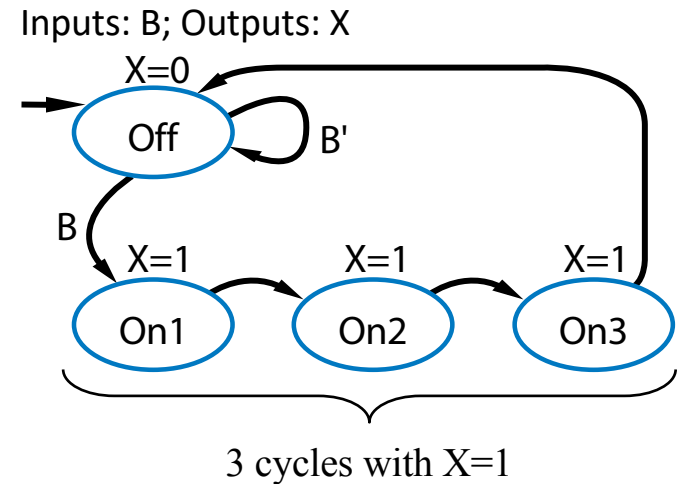



# **Hardware Description Languages (HDLs) for Digital Design**

HDL for RTL Design

# High-Level State Machine Behavior

- Register-transfer level (RTL) design captures desired system behavior using high-level state machine
  - Earlier example – 3 cycles high, used FSM
  - What if 512 cycles high? 512-state FSM?
  - **Better solution – High-level state machine that uses register to count cycles**
    - Declare **explicit register** Cnt (2 bits for 3-cycles high)
    - Initialize Cnt to 2 (2, 1, 0 → 3 counts)
    - "On" state
      - Sets  $X=1$
      - Configures Cnt for decrement on next cycle
      - Transitions to Off when Cnt is 0
      - **Note that transition conditions use current value of Cnt, not next (decremented) value**
    - For 512 cycles high, just initialize Cnt to 511





## Describing a State Machine using One Process (For lab 8 and your final project)



# One-Procedure State Machine Description

- Previously described FSM using two procedures
  - One for combinational logic, one for registers
  - Required "current" and "next" signals for each register
- A one-procedure description is actually better and easier
  - One procedure per HLSM
  - One variable per register
  - Simpler code with clear grouping of functionality
  - But may change timing

```
...
module LaserTimer(B, X, Clk, Rst);
...
    reg [0:0] State, StateNext;
    reg [1:0] Cnt, CntNext;

    // CombLogic
    always @(State, Cnt, B) begin
        ...
    end

    // Regs
    always @(posedge Clk) begin
        ...
    end
endmodule
```

```
...
module LaserTimer(B, X, Clk, Rst);
...
    reg [0:0] State;
    reg [1:0] Cnt;

    always @(posedge Clk) begin
        end
    endmodule
```

# One-Procedure State Machine Description

- Procedure sensitive to **clock only**
- If not reset, then executes current state's actions, and assigns next value of state
- Why didn't we describe with one procedure before?
  - Main reason – Procedure synchronous to clock only → *Inputs become synchronous*
    - Changes the state machine's timing
    - e.g., In state S\_off, and B changes to 1 in middle of clock cycle
    - Previous two procedure model
      - Change in b immediately noticed by combinational logic procedure, which configures next state to be S\_On. State changes to S\_On on next rising clock.
    - One procedure model
      - Change in b not noticed until next rising clock, so next state still S\_Off. After rising clock, procedure configures next state to be S\_On. State changes to S\_On on the next rising clock
        - or two rising clocks after b changed

```
...
module LaserTimer(B, X, Clk, Rst);
    ...
    parameter S_Off = 0,
              S_On  = 1;

    reg [0:0] State;
    reg [1:0] Cnt;

    always @(posedge Clk) begin
        if (Rst == 1 ) begin
            State <= S_Off;
            Cnt <= 0;
        end
        else begin
            case (State)
                S_Off: begin
                    X <= 0;
                    Cnt <= 2;
                    if (B == 0)
                        State <= S_Off;
                    else
                        State <= S_On;
                end
                S_On: begin
                    X <= 1;
                    Cnt <= Cnt - 1;
                    if (Cnt == 0)
                        State <= S_Off;
                    else
                        State <= S_On;
                end
            endcase
        end
    end
endmodule
```