# 5 – RLT and Verilog Review

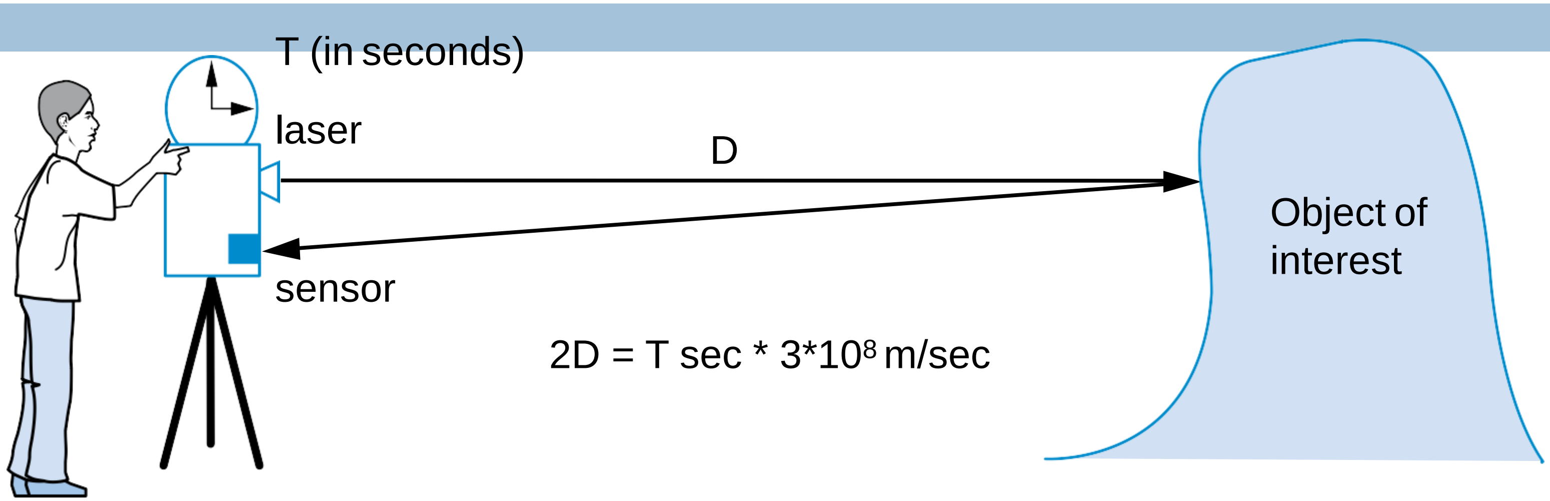
ECE 474A/574A
COMPUTER-AIDED LOGIC DESIGN

# RTL Design Method

| | Step | Description |
|---|---|---|
| **Step 1:** | Capture the high-level FSM | 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 bit inputs and outputs |
| **Step 2:** | Create a datapath | Create a datapath to carry out the data operations on the high-level state machine |
| **Step 3:** | Connect the datapath to the controller | Connect the datapath to the controller block. Connect external Boolean inputs and output to the controller block |
| **Step 4:** | 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 |

# Laser-Based Distance Measurer

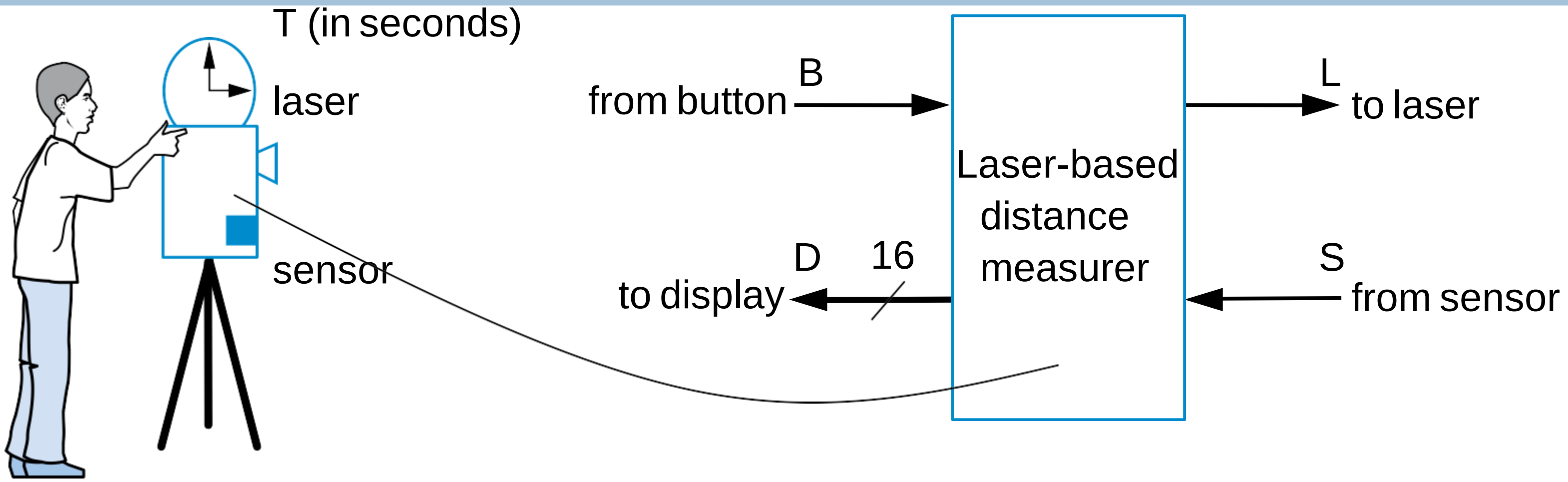
## Step 1: Capture a high-level state machine



- Example of how to create a high-level state machine to describe desired processor behavior
- Laser-based distance measurement – pulse laser, measure time T to sense reflection
  - Laser light travels at speed of light, $3*10^8$ m/sec
  - Distance is thus D = T sec * $3*10^8$ m/sec / 2

# Laser-Based Distance Measurer

## Step 1 : Capture a high-level state machine



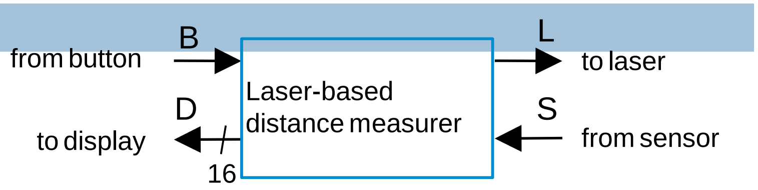
- Inputs/outputs
  - *B*: bit input, from button to begin measurement
  - *L*: bit output, activates laser
  - *S*: bit input, senses laser reflection
  - *D*: 16-bit output, displays computed distance

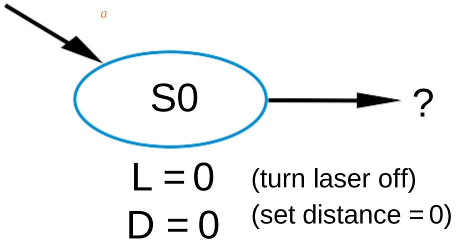
# Laser-Based Distance Measurer

## Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)

Outputs: L (bit), D (16 bits)





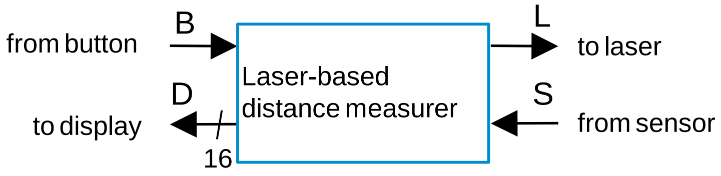
- Step 1: Create high-level state machine
- Begin by declaring inputs and outputs
- Create initial state, name it **S0**
  - Initialize laser to off (L=0)
  - Initialize displayed distance to 0 (D=0)

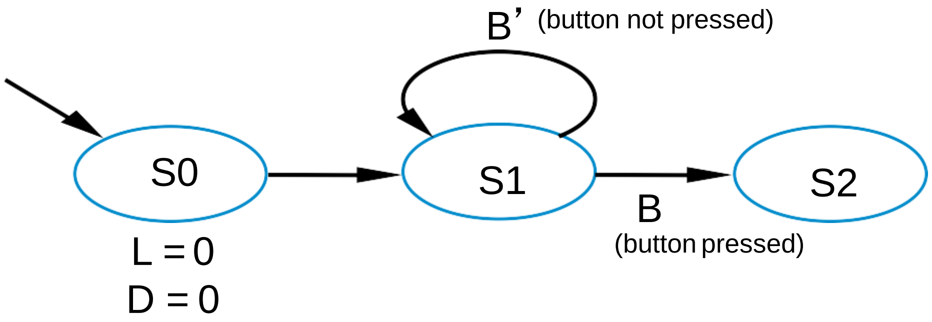
# Laser-Based Distance Measurer

## Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)

Outputs: L (bit), D (16 bits)





- Add another state, call **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?

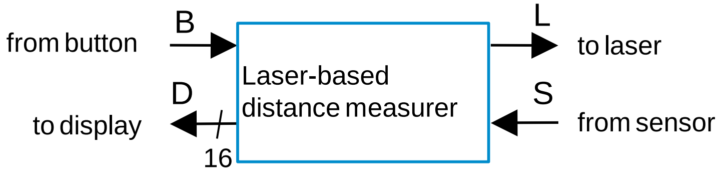
A: Turn on the laser

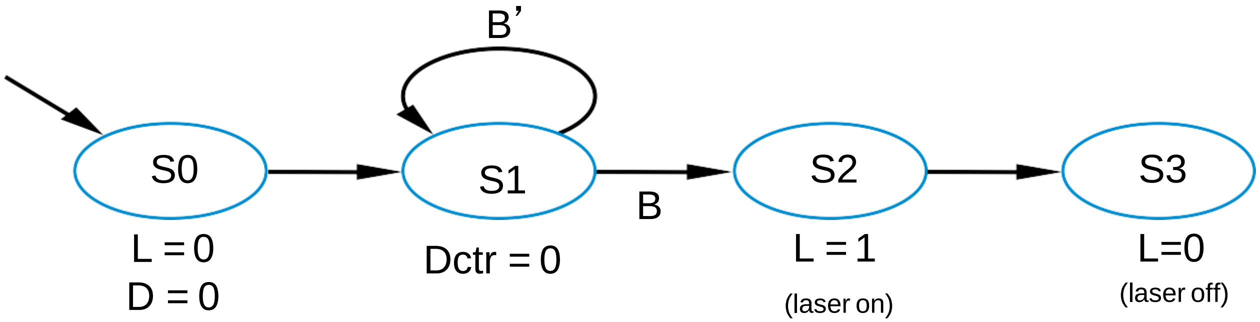
# Laser-Based Distance Measurer

## Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)

Outputs: L (bit), D (16 bits)





a

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

Q: What should the next state do?

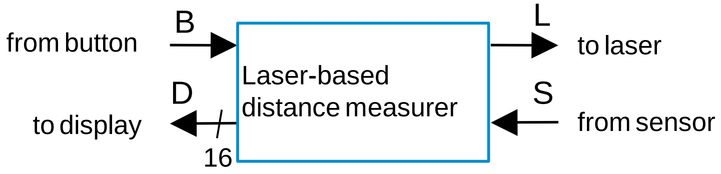
A: Start timer, wait to sense reflection

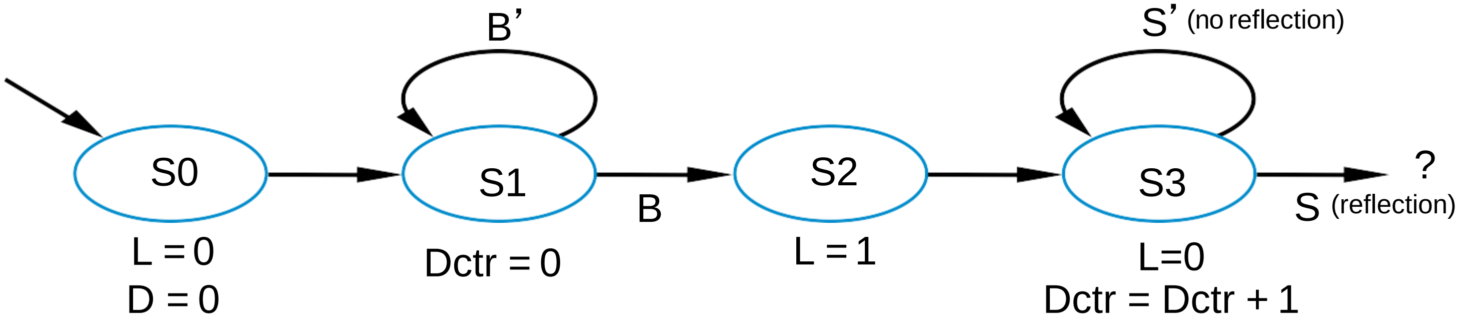
a

# Laser-Based Distance Measurer

## Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)



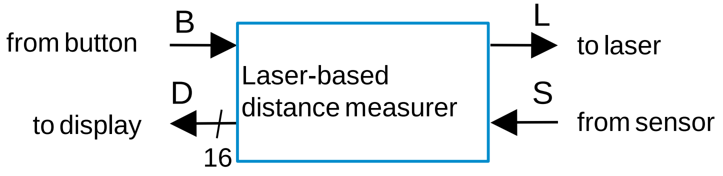


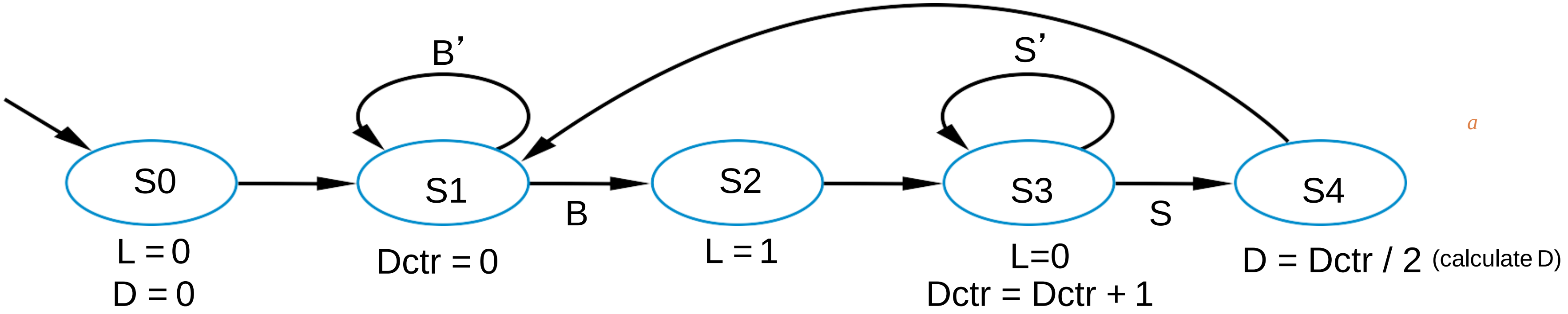
- Stay in **S3** until sense reflection (S)
- To measure time, count cycles for which we are in **S3**
  - To count, declare local register *Dctr*
  - Increment *Dctr* each cycle in **S3**
  - Initialize *Dctr* to 0 in **S1**. **S2** would have been O.K. too

# Laser-Based Distance Measurer

## Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)





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

# Laser-Based Distance Measurer

## Step 2: Create a Datapath

- Datapath must
  - Implement data storage
  - Implement data computations

- Look at high-level state machine, do three substeps
  a) Make data inputs/outputs be datapath inputs/outputs
  b) Instantiate declared registers into the datapath (also instantiate a register for each data output)
  c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations

***Instantiate***: to introduce a new component into a design.

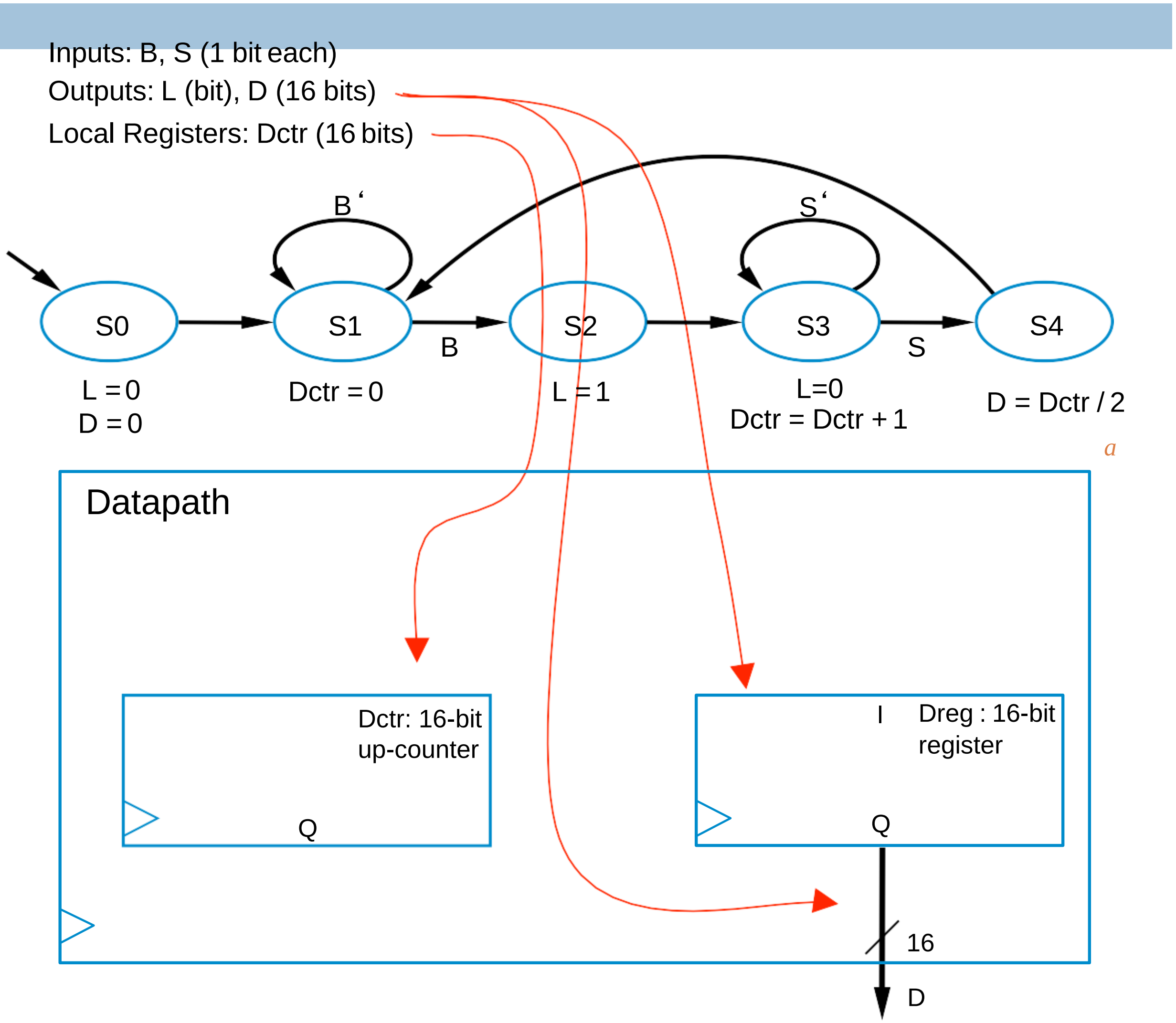
# Laser-Based Distance Measurer

## Step 2: Create a Datapath

a) Make data inputs/outputs be datapath inputs/outputs

b) Instantiate declared registers into the datapath (also instantiate a register for each data output)

c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations



Inputs: B, S (1 bit each)

Outputs: L (bit), D (16 bits)

Local Registers: Dctr (16 bits)

# Laser-Based Distance Measurer

## Step 2: Create a Datapath

a) Make data inputs/outputs be datapath inputs/outputs

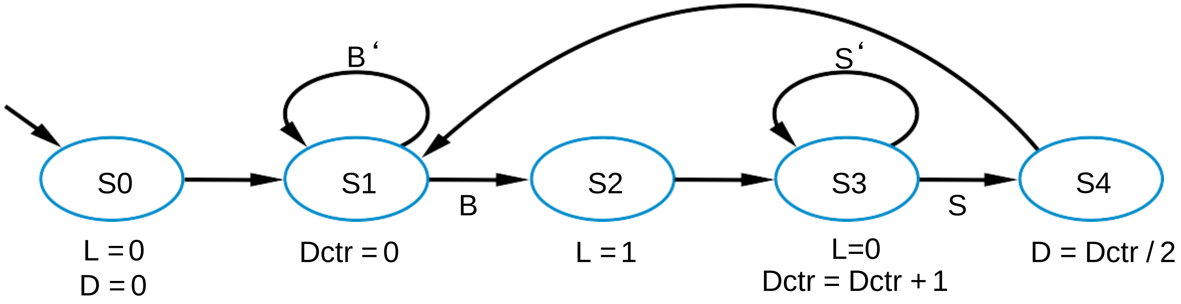
b) Instantiate declared registers into the datapath (also instantiate a register for each data output)

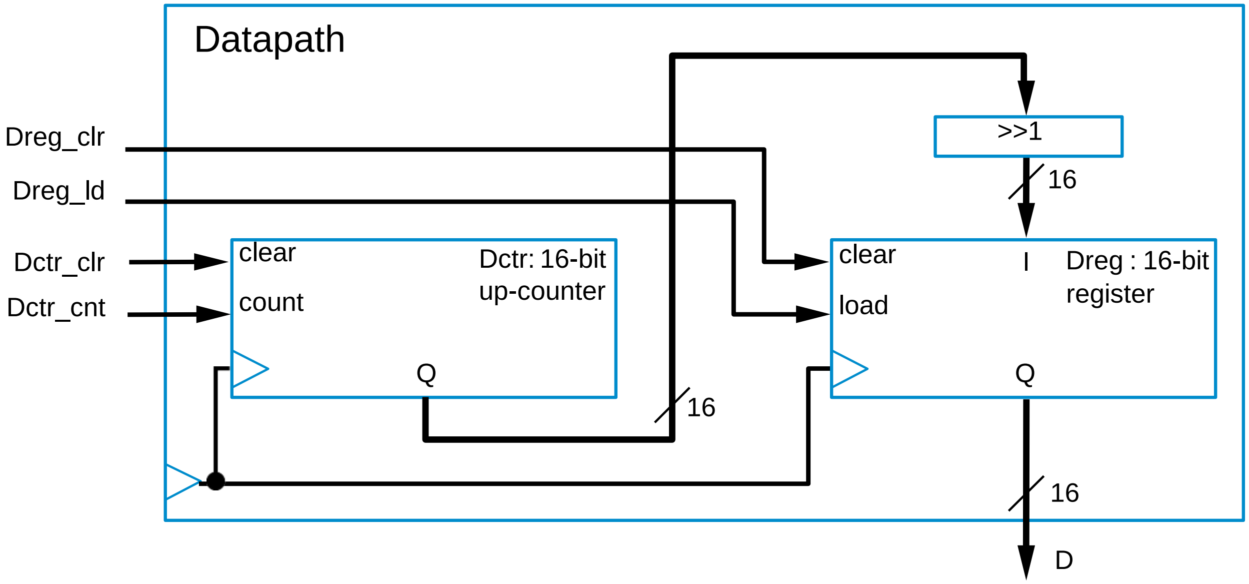
c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations

Inputs: B, S (1 bit each)

Outputs: L (bit), D (16 bits)

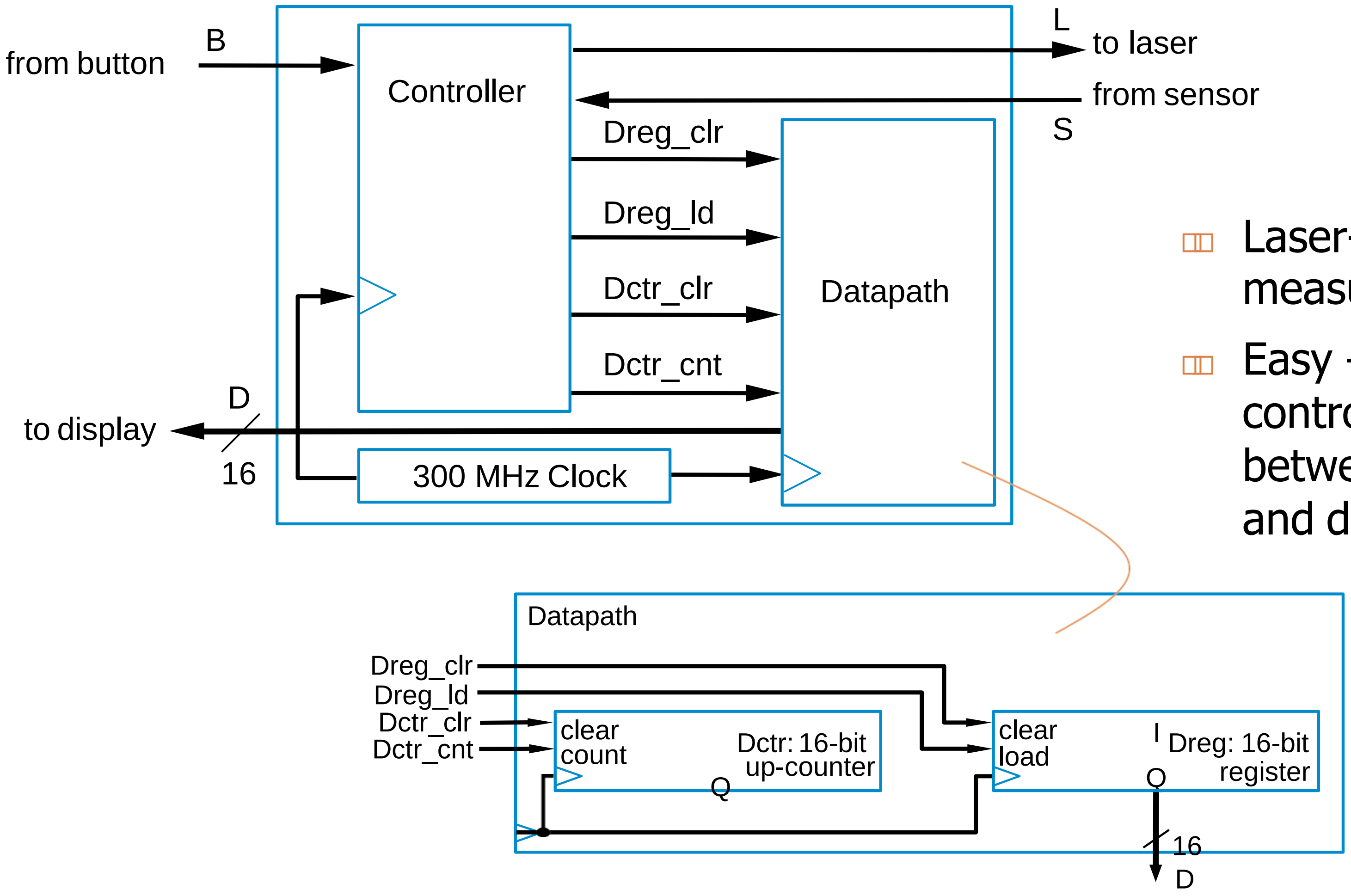
Local Registers: Dctr (16 bits)





# Laser-Based Distance Measurer

## Step 3: Connect datapath to controller



- Laser-based distance measurer example
- Easy – just connect all control signals between controller and datapath

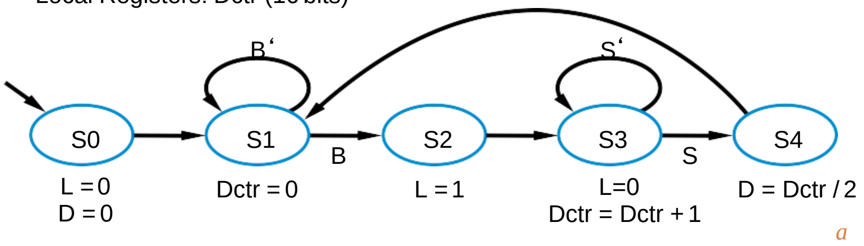
# Laser-Based Distance Measurer

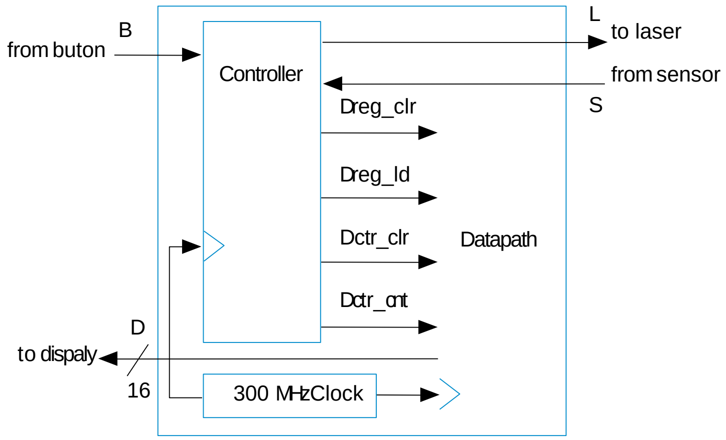
## Step 4: Derive controller’s FSM

### FSM has same structure as high-level state machine

- Inputs/outputs all bits now
- Replace data operations by bit operations using datapath

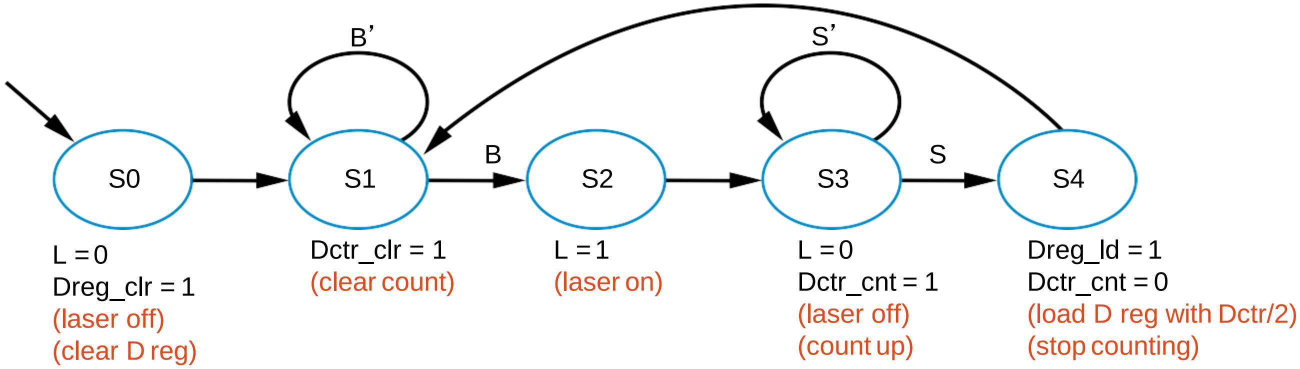
Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)





Inputs: B, S

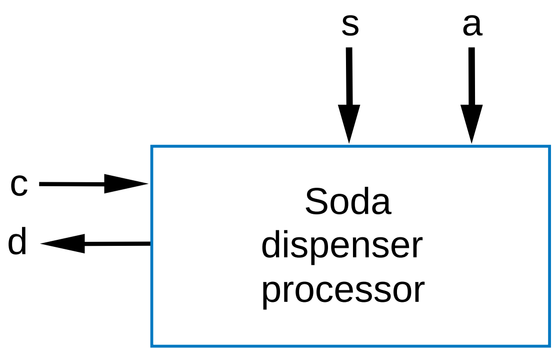
Outputs: L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt



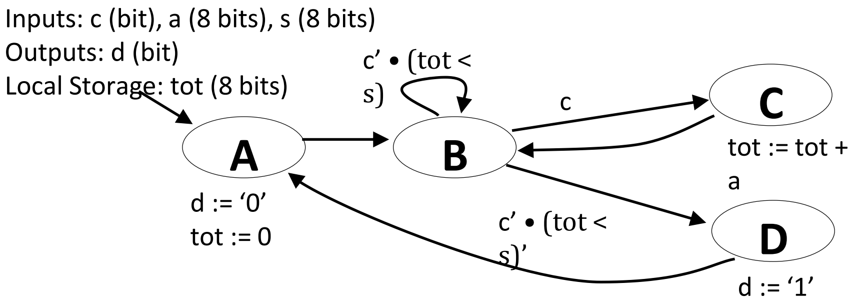
# Exercise

Using the RTL design process, design a soda dispenser with the following characteristics

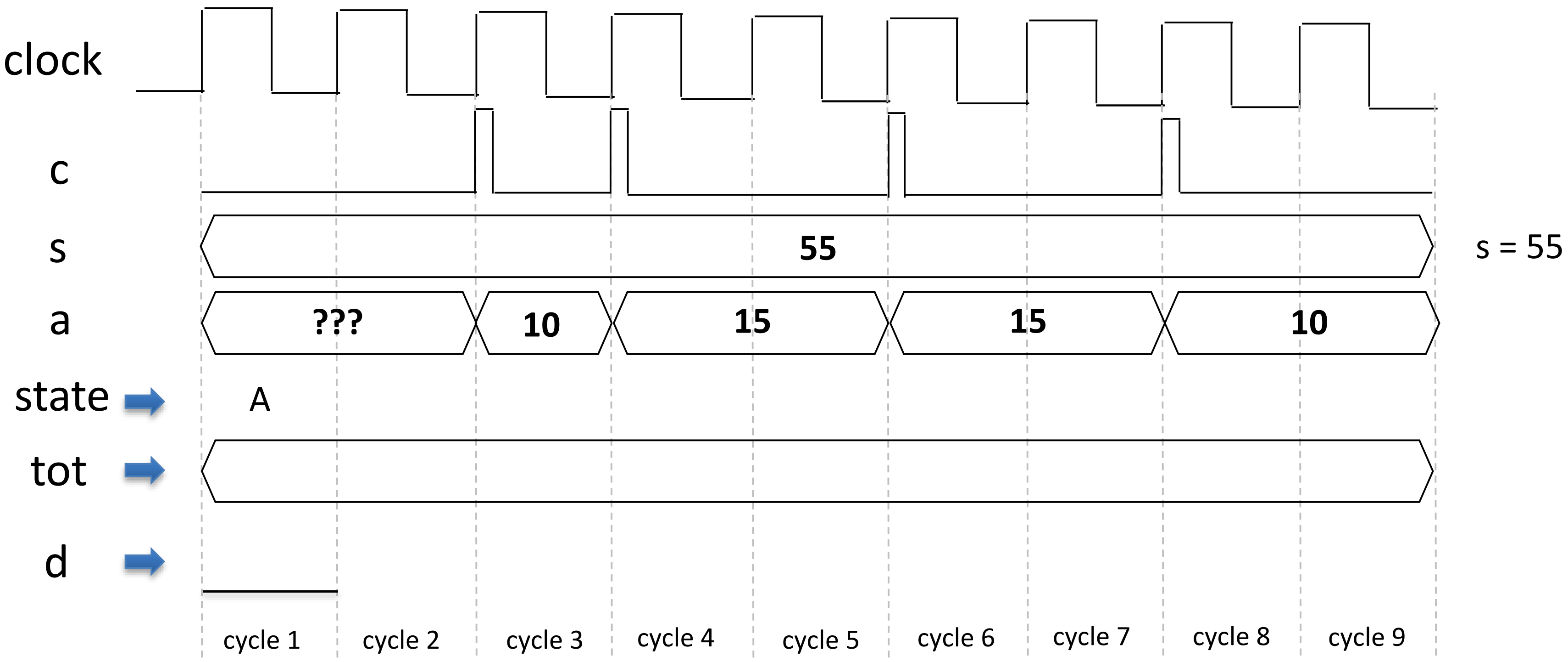
*c*: bit input, 1 when coin deposited
*a*: 8-bit input having value of deposited coin
*s*: 8-bit input having cost of a soda
*d*: bit output, processor sets to 1 when total value of deposited coins equals or exceeds cost of a soda



# HLSM Timing



Inputs: c (bit), a (8 bits), s (8 bits)
Outputs: d (bit)
Local Storage: tot (8 bits)



# Treadmill example

# Exercise

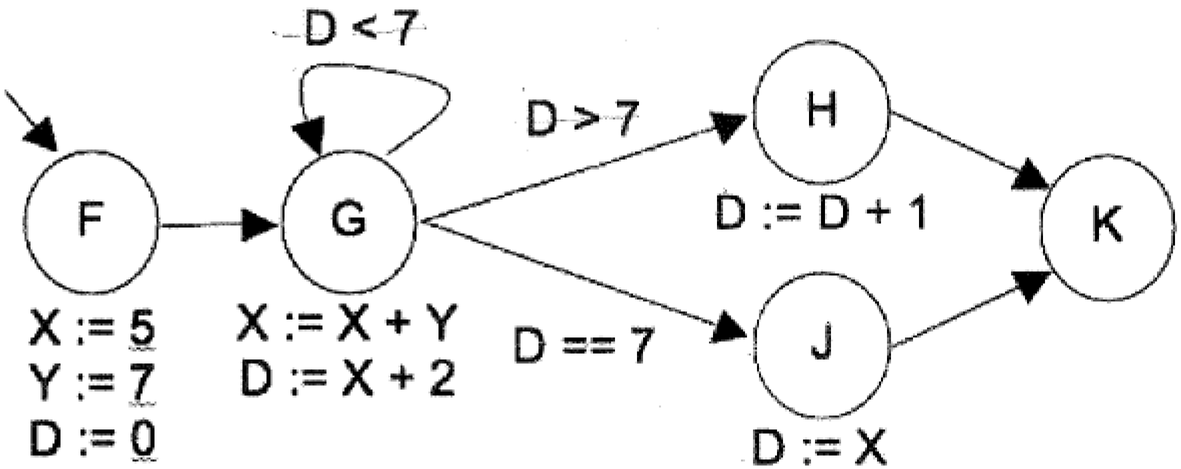
1. What is the value of D in state K?

   a. 7; b) 9; c) 12; d) 15; e) none of the above

2. Design the datapath and controller for the following HLSM

Outputs: X (8-bits), Y (8-bits)

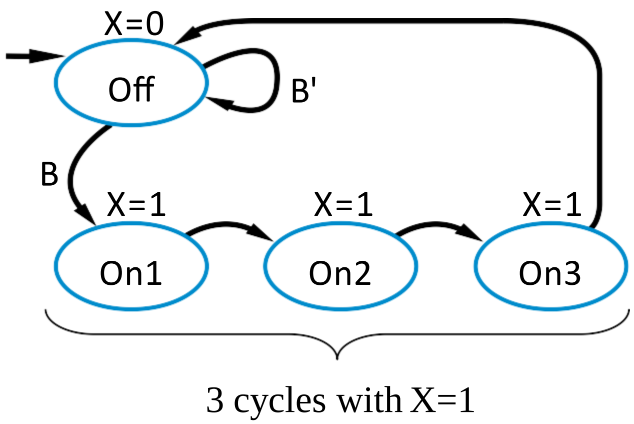
Local Storage: X (8-bits), Y (8-bits), D (8-bits)



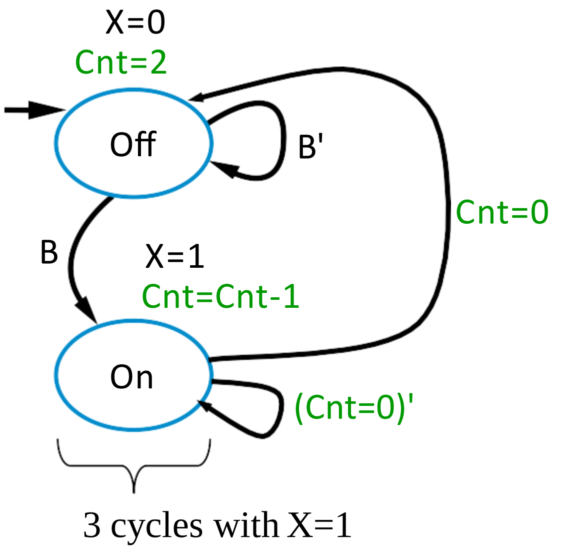
# High-Level State Machine Behavior

- 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

Inputs: B; Outputs: X



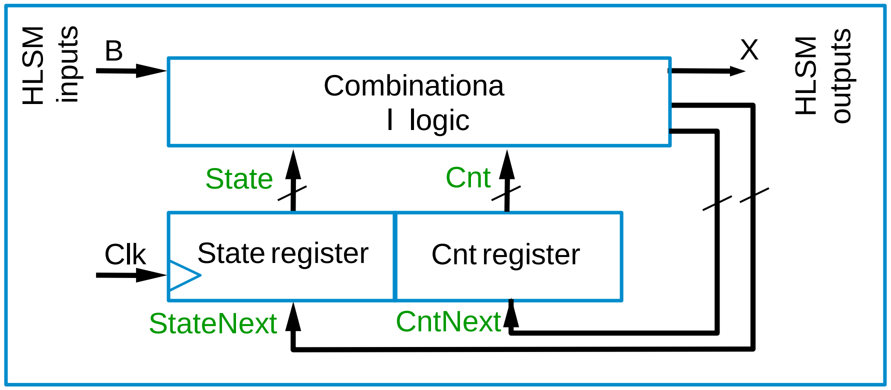
Inputs: B; Outputs: X; Register: Cnt(2)



# High-Level State Machine Behavior

- Module ports same as FSM
- Same two-procedure approach as FSM
  - One for combinational logic, one for registers
  - Registers now include explicit registers (Cnt)
    - Two reg variables per explicit register (current and next), just like for state register

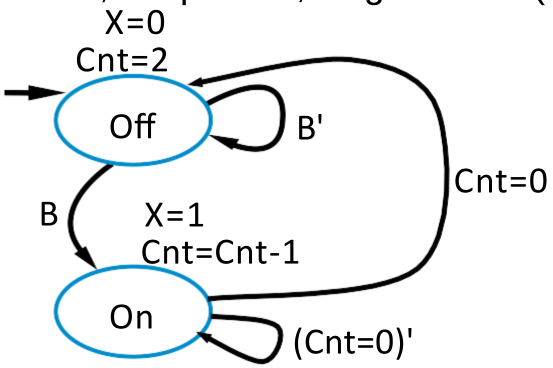
```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0,
             S_On  = 1;

   reg [0:0] State, StateNext;
   reg [1:0] Cnt, CntNext;

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

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

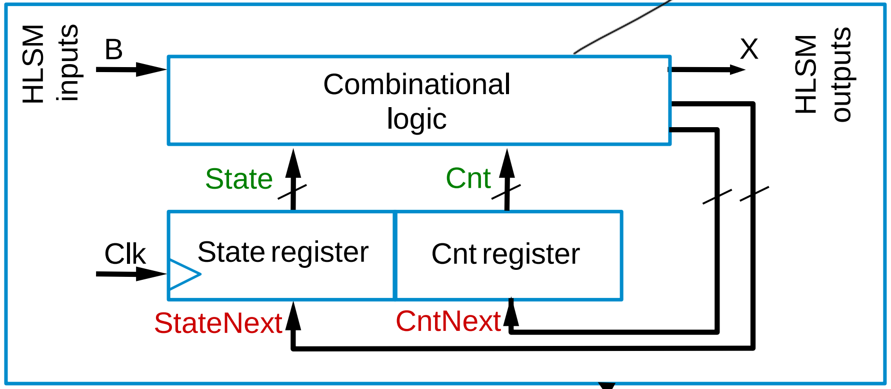


# High-Level State Machine Behavior

- CombLogic process
  - Describes actions and transitions

Inputs: B; Outputs: X; Register: Cnt(2)





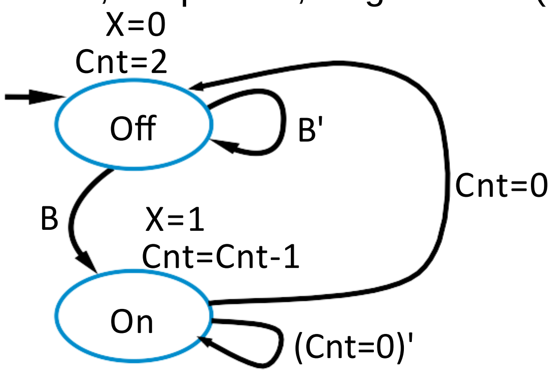
```
...
   reg [0:0] State, StateNext;
   reg [1:0] Cnt, CntNext;

   // CombLogic
   always @(State, Cnt, B) begin
      case (State)
         S_Off: begin
            X <= 0;
            CntNext <= 2;
            if (B == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On;
         end
         S_On: begin
            X <= 1;
            CntNext <= Cnt - 1;
            if (Cnt == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On;
         end
      endcase
   end
...
```

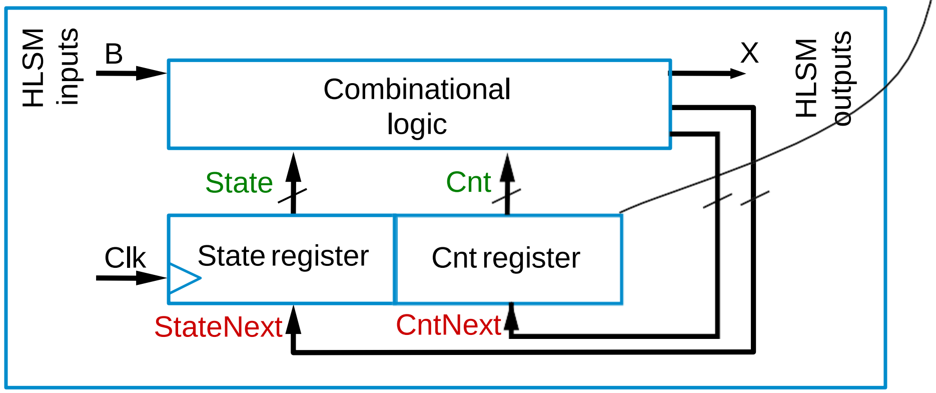
Note: Writes are to "next" variable, reads are from "current" variable. See target architecture to understand why.

# High-Level State Machine Behavior

- Regs process
  - Updates registers on rising clock

Inputs: B; Outputs: X; Register: Cnt(2)



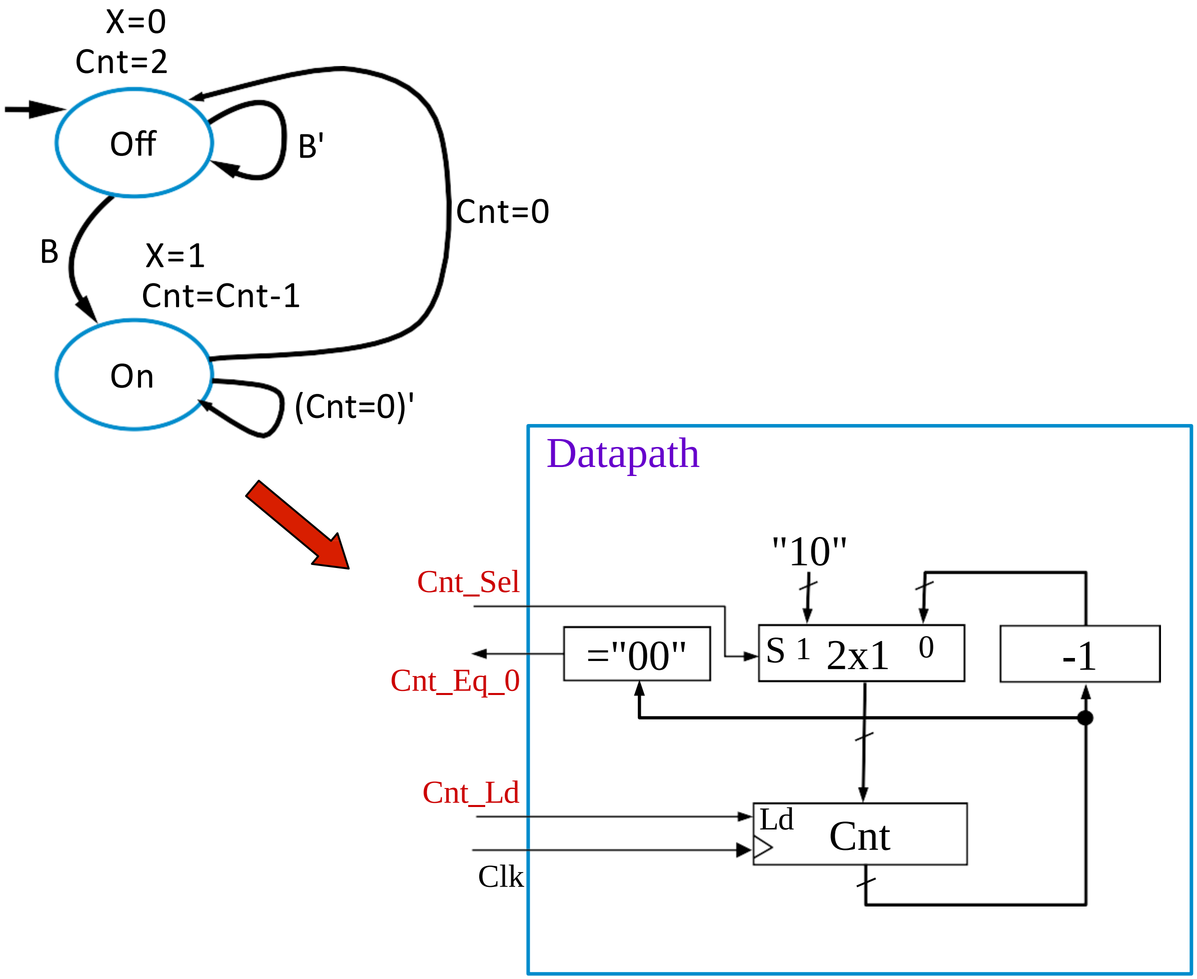


```
. . .
   // Regs
  always @(posedge Clk) begin
     if (Rst == 1 ) begin
        State <= S_Off;
        Cnt <= 0;
     end
     else begin
        State <= StateNext;
        Cnt <= CntNext;
     end
  end
. . .
```

# Top-Down Design: HLSM to Controller and Datapath

- Deriving a datapath from the HLSM

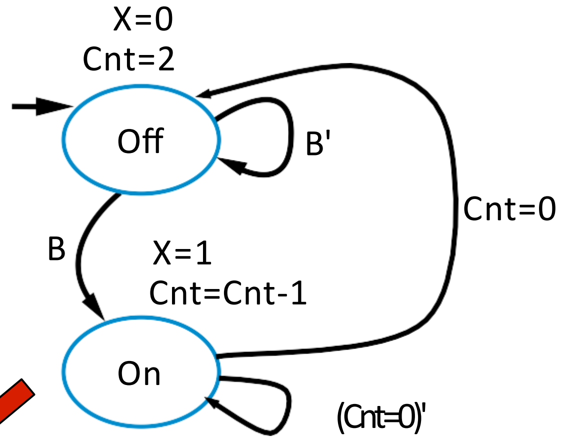
Inputs: B; Outputs: X; Register: Cnt(2)

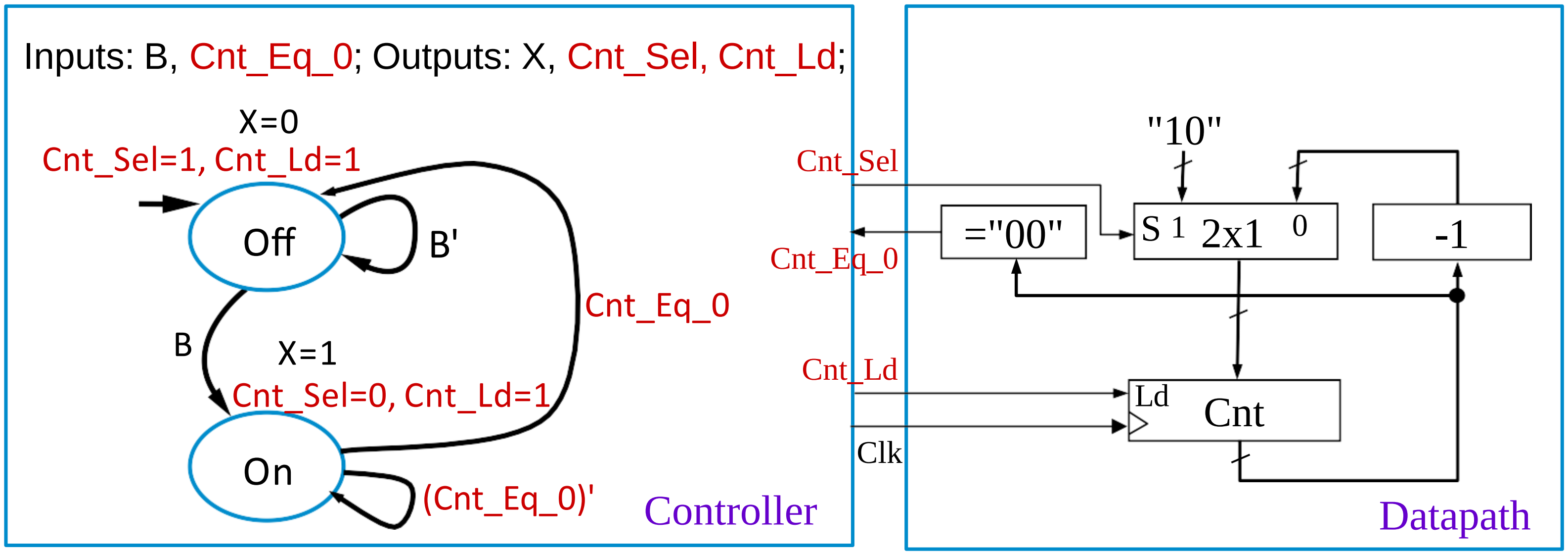


# Top-Down Design: HLSM to Controller and Datapath

- Deriving a controller
  - Replace HLSM by FSM that uses the datapath

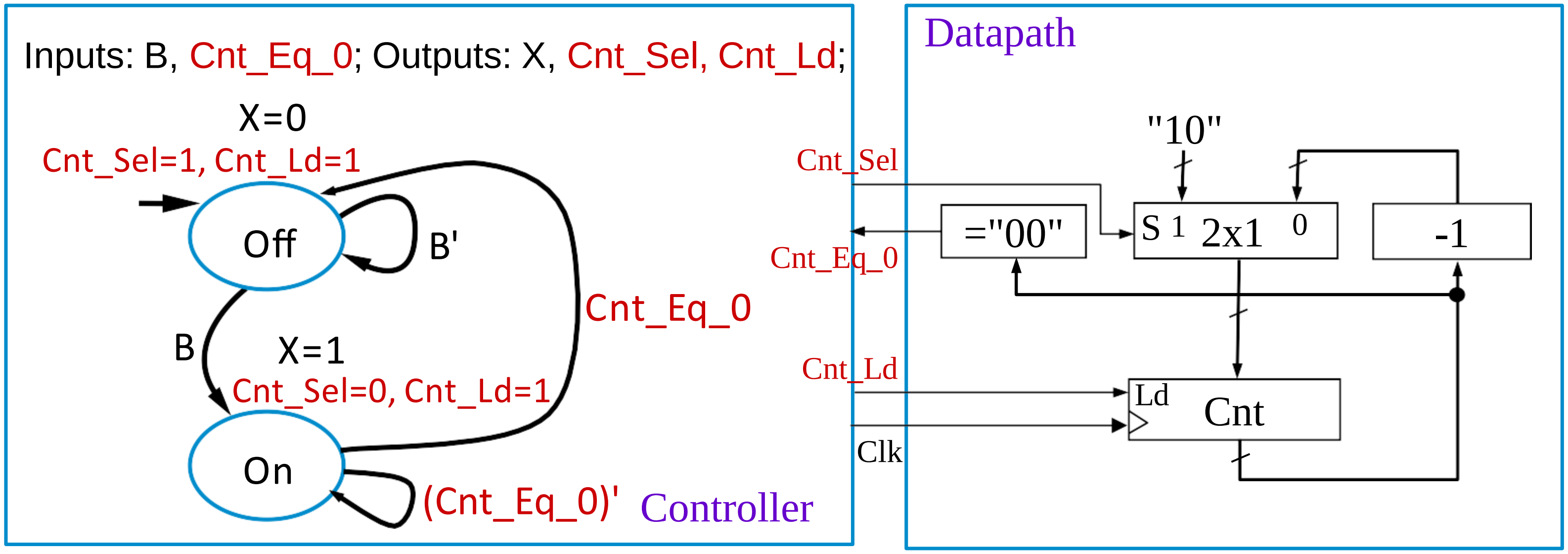
Inputs: B; Outputs: X; Register: Cnt(2)




Inputs: B, Cnt_Eq_0; Outputs: X, Cnt_Sel, Cnt_Ld;

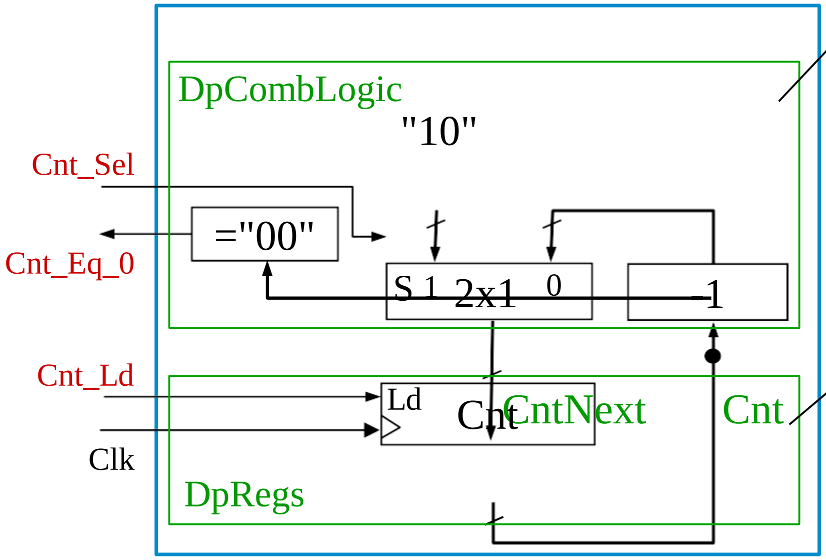
# Top-Down Design: HLSM to Controller and Datapath

- Describe controller and datapath in VHDL
  - One option: structural datapath, behavioral (FSM) controller
  - Let's instead describe both behaviorally



# Describing a Datapath Behaviorally

- Two procedures
  - Combinational part and register part
  - Current and next signals shared between the two parts
  - Just like for FSM behavior

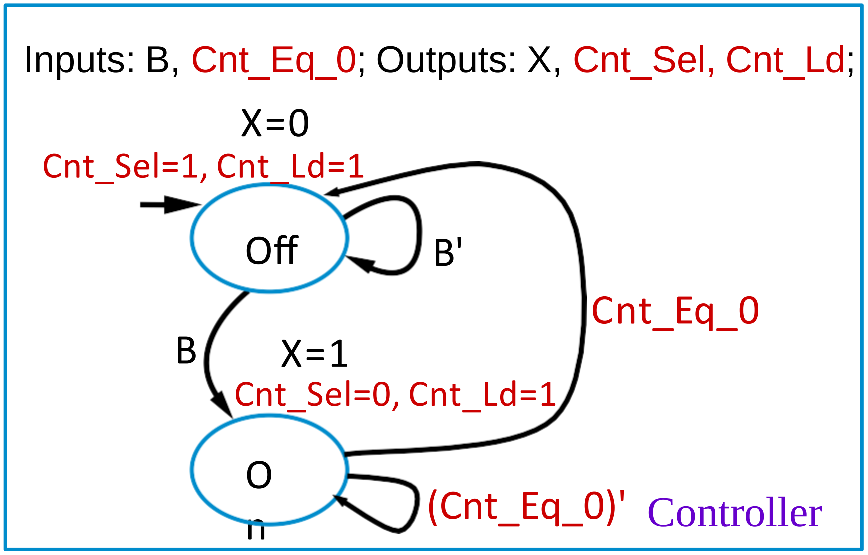


```
...
// Shared variables
reg Cnt_Eq_0, Cnt_Sel, Cnt_Ld;
// Controller variables
reg [0:0] State, StateNext;
// Datapath variables
reg [1:0] Cnt, CntNext;

// ------ Datapath Procedures ------ //
// DP CombLogic
always @(Cnt_Sel, Cnt) begin
   if (Cnt_Sel==1)
      CntNext <= 2;
   else
      CntNext <= Cnt - 1;

   Cnt_Eq_0 <= (Cnt==0)?1:0;
end

// DP Regs
always @(posedge Clk) begin
   if (Rst == 1 )
      Cnt <= 0;
   else if (Cnt_Ld==1)
      Cnt <= CntNext;
end
...
```

*Note use of previously-introduced conditional operator*

# Describing the Controller Behaviorally

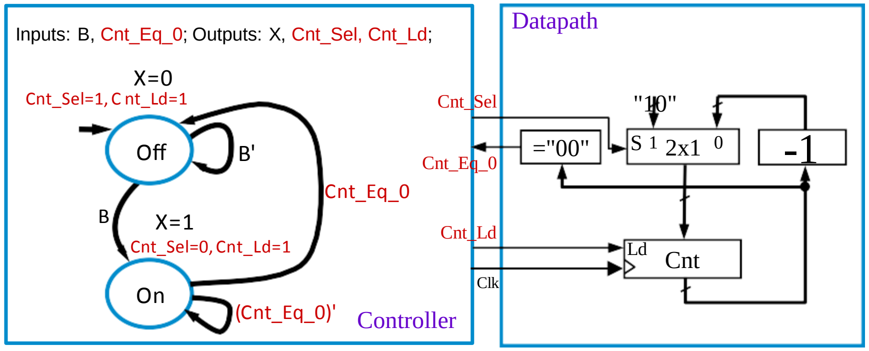
- Standard approach for describing FSM
  - Two procedures



```
...
// ------ Controller Procedures ------ //
// Ctrl CombLogic
always @(State, Cnt_Eq_0, B) begin
   case (State)
      S_Off: begin
         X <= 0; Cnt_Sel <= 1; Cnt_Ld <= 1;
         if (B == 0)
            StateNext <= S_Off;
         else
            StateNext <= S_On;
      end
      S_On: begin
         X <= 1; Cnt_Sel <= 0; Cnt_Ld <= 1;
         if (Cnt_Eq_0 == 1)
            StateNext <= S_Off;
         else
            StateNext <= S_On;
      end
   endcase
end

// Ctrl Regs
always @(posedge Clk) begin
   if (Rst == 1 ) begin
      State <= S_Off;
   end
   else begin
      State <= StateNext;
   end
end
...
```

# Controller and Datapath Behavior

- Result is one module with four procedures
  - Datapath procedures (2)
    - Combinational logic
    - Registers
  - Controller procedures (2)
    - Combinational logic
    - Registers



```
...
module LaserTimer(B, X, Clk, Rst);
   ...

   parameter S_Off = 0,
             S_On  = 1;

   // Shared variables
   reg Cnt_Eq_0, Cnt_Sel, Cnt_Ld;
   // Controller variables
   reg [0:0] State, StateNext;
   // Datapath variables
   reg [1:0] Cnt, CntNext;

   // ------ Datapath Procedures ------ //
   // DP CombLogic
   always @(Cnt_Sel, Cnt) begin
      ...
   end

   // DP Regs
   always @(posedge Clk) begin
      ...
   end

   // ------ Controller Procedures ------ //
   // Ctrl CombLogic
   always @(State, Cnt_Eq_0, B) begin
      ...
   end

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