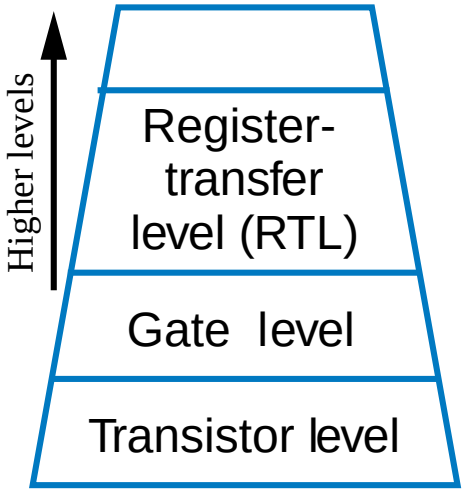
Custom Single-Purpose Processor Design

Introduction

- Combinational Logic Design
  - Capture Comb. behavior: Equations, truth tables
  - Convert to circuit: AND + OR + NOT → Comb. Logic
- Sequential Logic Design
  - Capture sequential behavior: FSMs
  - Convert to circuit: Register + Comb. logic → Controller
- Costum Single Purpose Processor Design
  - Capture behavior: High-level state machine
  - Convert to circuit: Controller + Datapath → Processor
  - Known as “RTL” (register-transfer level) design



Levels of digital design abstraction

Processors:
- Programmable (microprocessor)
- Custom

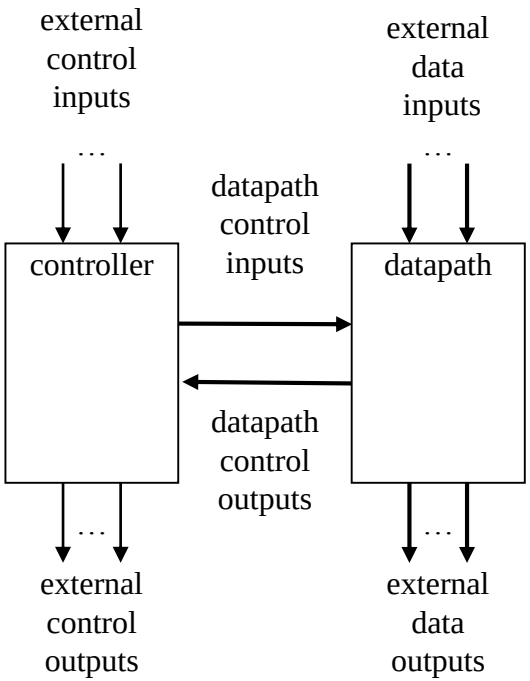
Note: Slides with animation are denoted with a small red "a" near the animated items

Objectives

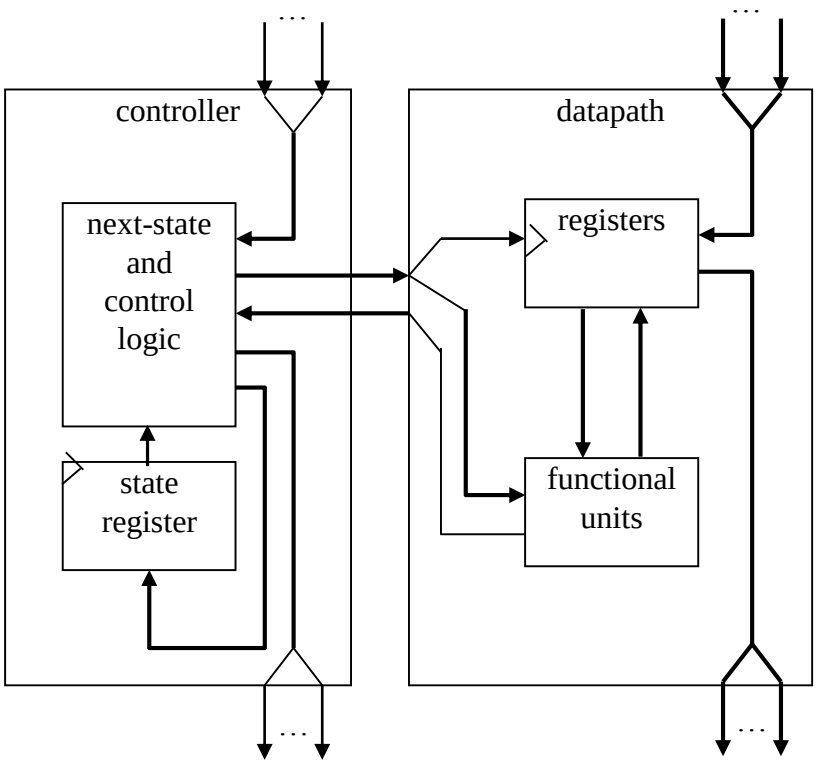
- Custom single-purpose processor design
- RT-level custom single-purpose processor design
  - Finite state machine with data (FSMD)
  - RTL design process
    - **register-transfer level** (**RTL**) is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals
- Optimizing custom single-purpose processor

Custom Single-Purpose Processor Design

A basic processor consists of a controller connected to a datapath.



controller and datapath



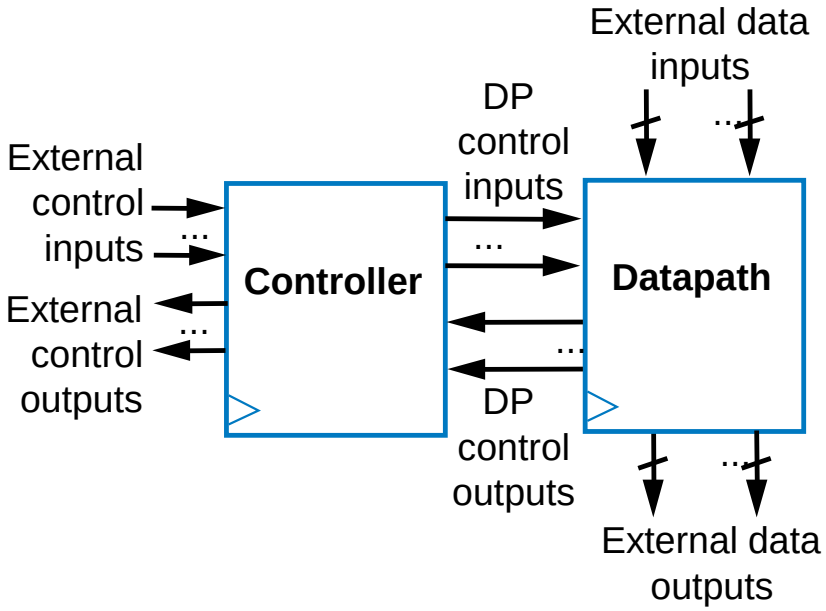
a view inside the controller and datapath

Custom Single-Purpose Processor Design

- A custom single-purpose purpose design is specialized to implement one specific computation.
- A complex state diagram called *finite state machine with data,* FSMD (or high-level state machine) is used for the design of custom single-purpose processors.
- In FSMD, states and arcs may include arithmetic expressions, and those expressions may use external inputs and outputs as well as variables. Our earlier state diagrams included only Boolean expressions, and those expressions could use only external inputs and outputs but not variables.

RTL Design Process

- Capture behavior
- Convert to circuit
  - Need target architecture
  - Datapath capable of HLSM's data operations
  - Controller to control datapath

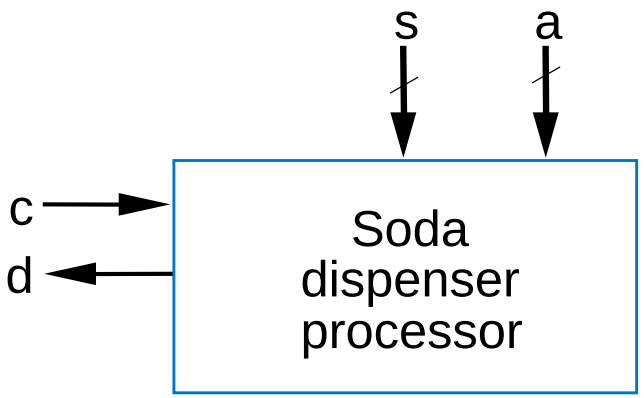


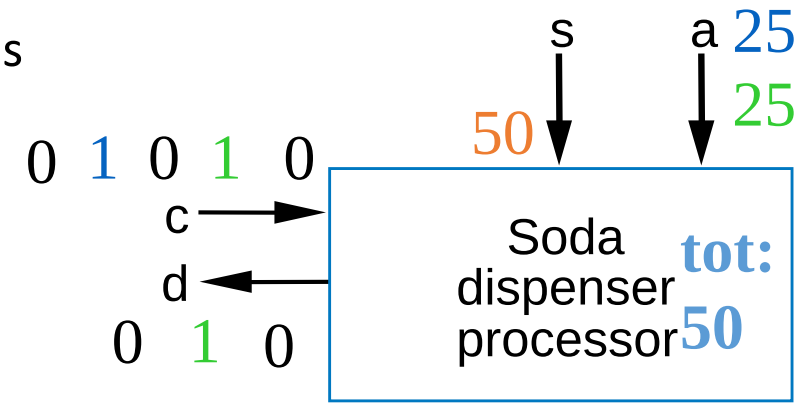
Single-Purpose Processor Design Method

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

Finite State Machine with Data (FSMD)

- Some behaviors too complex for equations, truth tables, or FSMs
- 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)

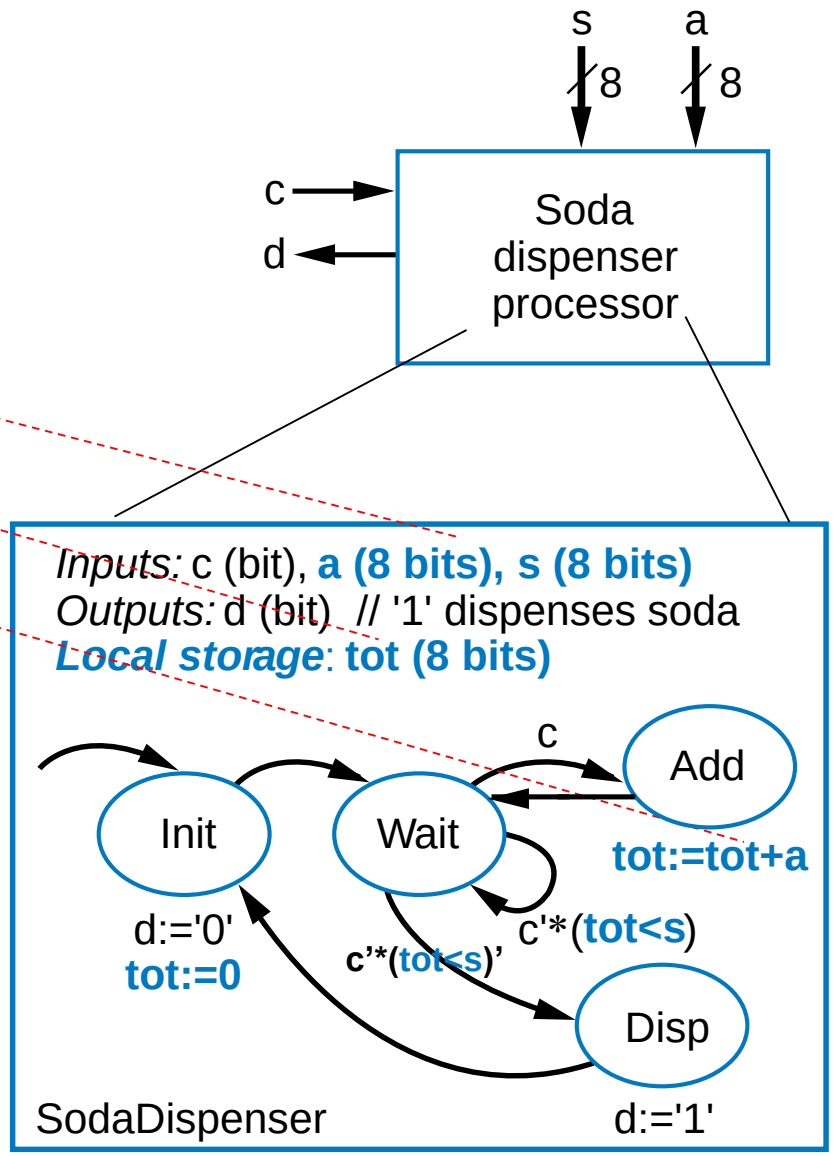




FSMD

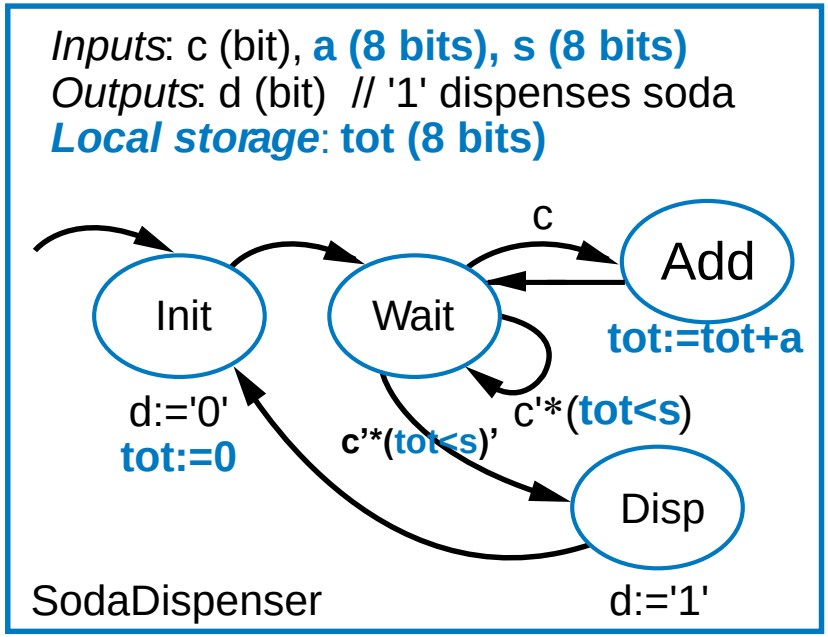
- High-level state machine (HLSM) extends FSM with:
  - Multi-bit input/output
  - Local storage
  - Arithmetic operations

- Conventions
  - Numbers:
    - Single-bit: '0' (single quotes)
    - Integer: 0 (no quotes)
    - Multi-bit: “0000” (double quotes)
  - == for equal, := for assignment
  - Multi-bit outputs *must* be registered via local storage
  - // precedes a comment

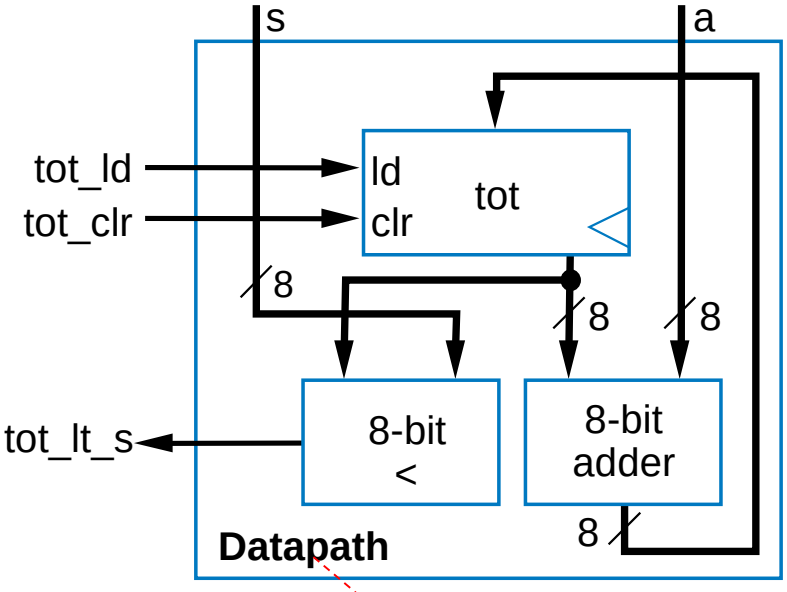


Example: Soda Dispenser

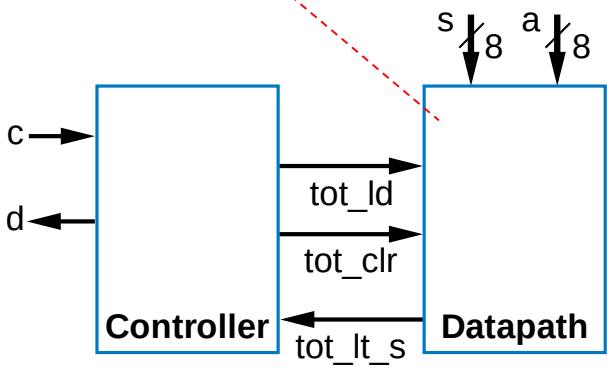
- Quick overview example.
  More details of each step to come.



Step 1



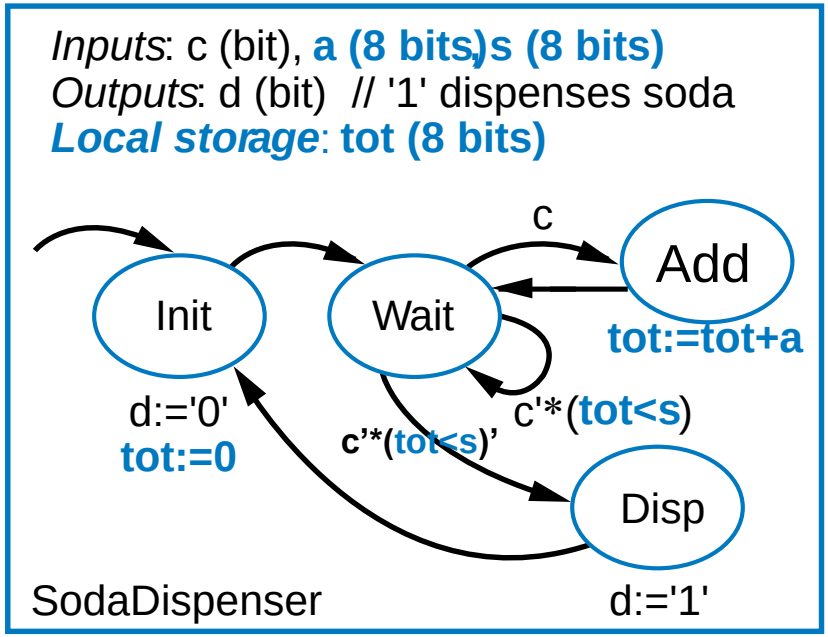
Step 2A



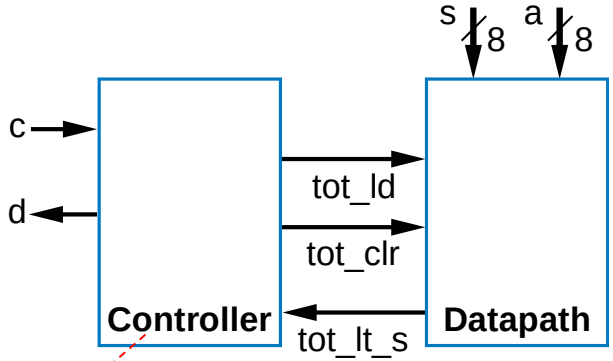
Step 2B

Example: Soda Dispenser

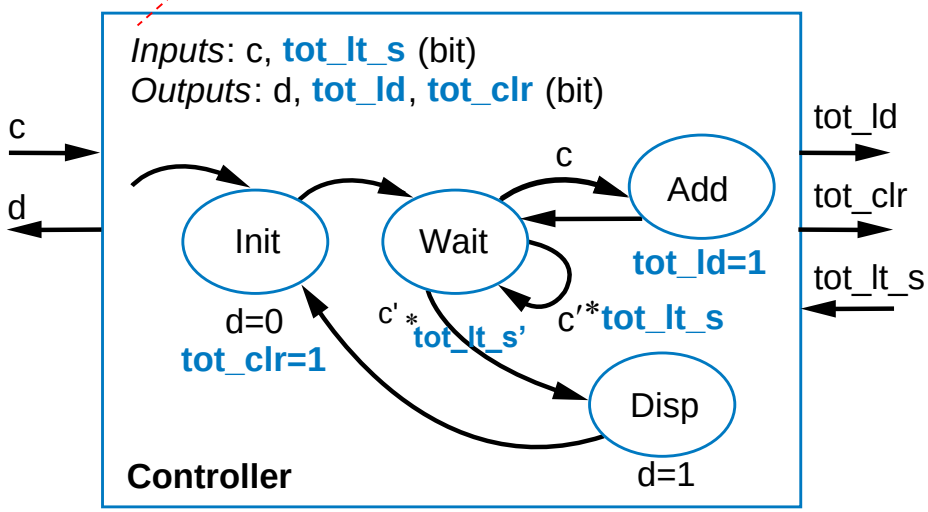
- Quick overview example.
  More details of each step to come.



Step 1



Step 2B

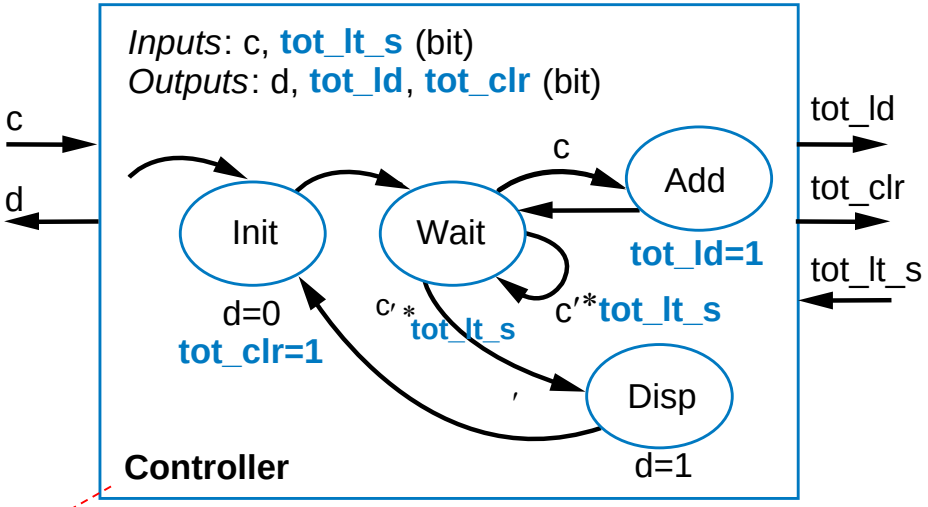


Step 2C

Example: Soda Dispenser

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

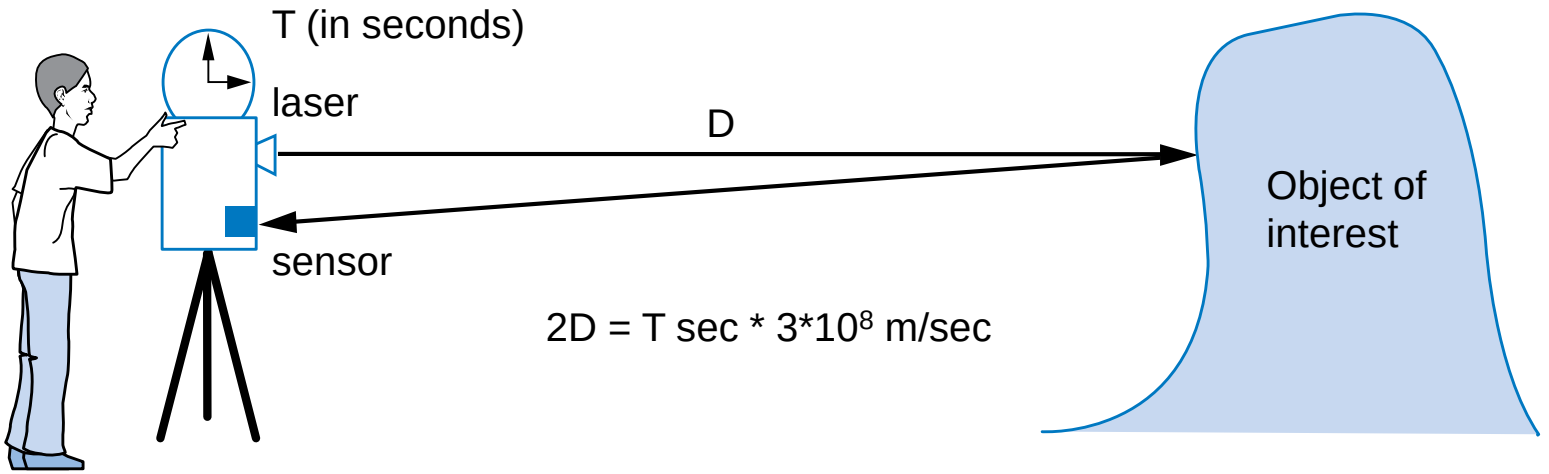
	s1	s0	c	tot_lt_s	n1	n0	d	tot_ld	tot_clr
Init	0	0	0	0	0	1	0	0	1
	0	0	0	1	0	1	0	0	1
	0	0	1	0	0	1	0	0	1
	0	0	1	1	0	1	0	0	1
Wait	0	1	0	0	1	1	0	0	0
	0	1	0	1	0	1	0	0	0
	0	1	1	0	1	0	0	0	0
	0	1	1	1	1	0	0	0	0
Add	1	0	0	0	0	1	0	1	0
			• • •				• • •		
Disp	1	1	0	0	0	0	1	0	0
			• • •				• • •		



Step 2C

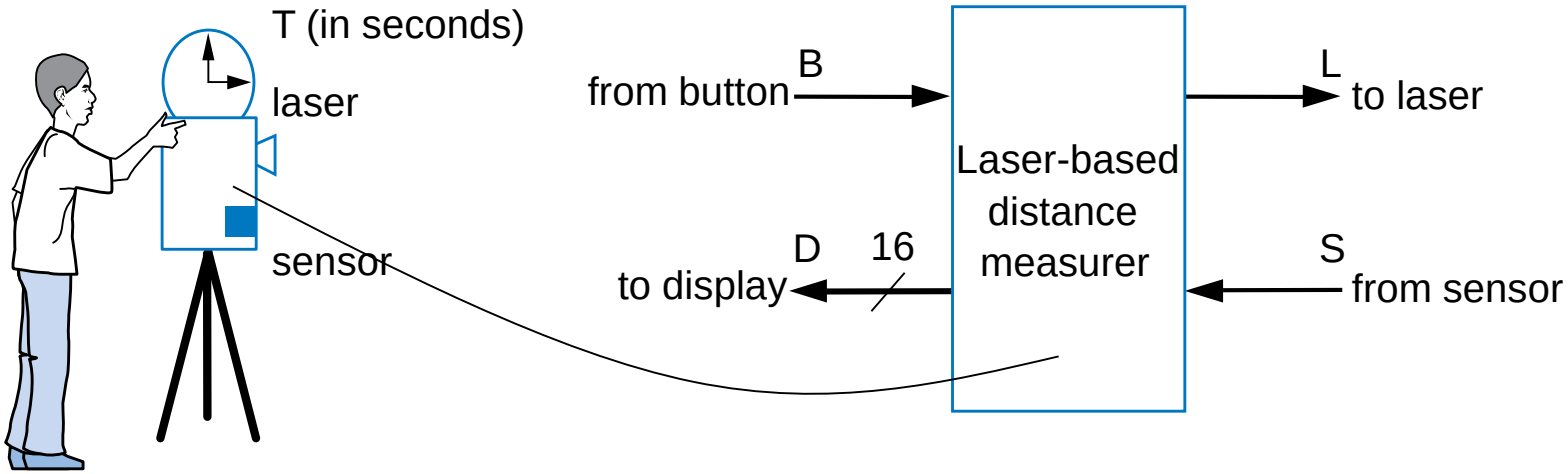
Use controller design process to complete the design

Example: Laser-Based Distance Measurer



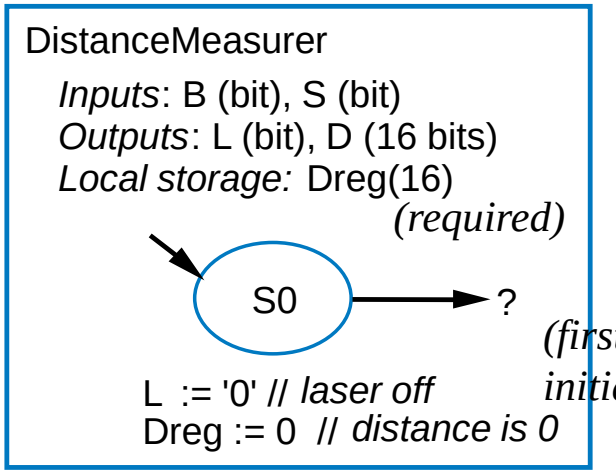
- Laser-based distance measurement – pulse laser, measure time T to sense reflection
  - Laser light travels at speed of light, $3*10^8$ m/sec
  - Distance is thus D = (T sec * $3*10^8$ m/sec) / 2

Example: Laser-Based Distance Measurer

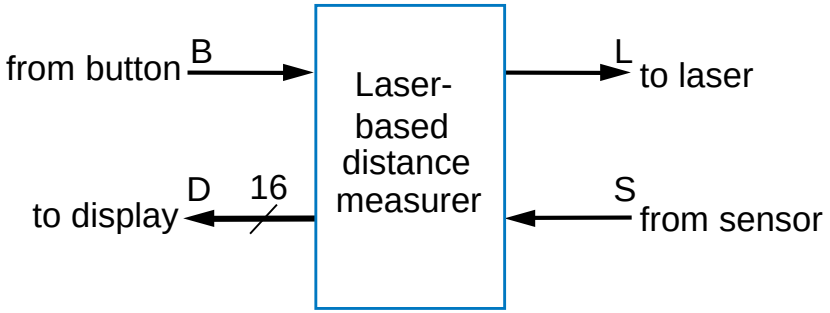


- Inputs/outputs
  - *B*: bit input, from button, to begin measurement
  - *L*: bit output, activates laser
  - *S*: bit input, senses laser reflection
  - *D*: 16-bit output, to display computed distance

Example: Laser-Based Distance Measurer



(first state usually initializes the system)

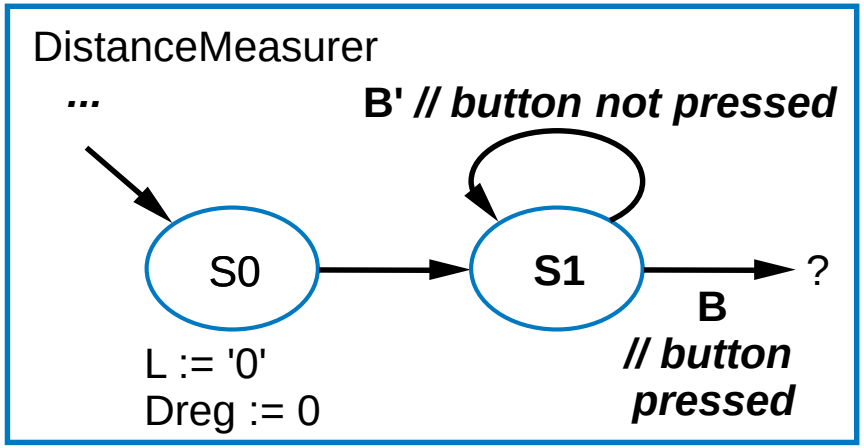


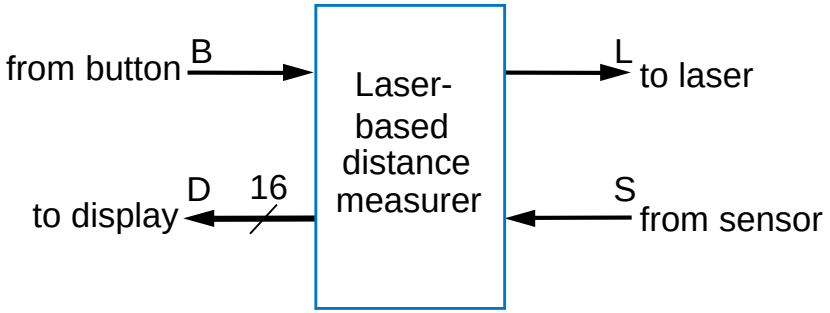
Step 1: Create FSMD

- 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

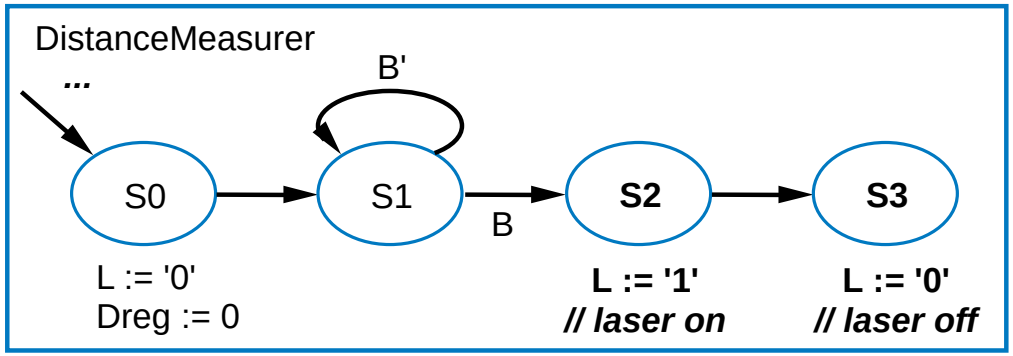


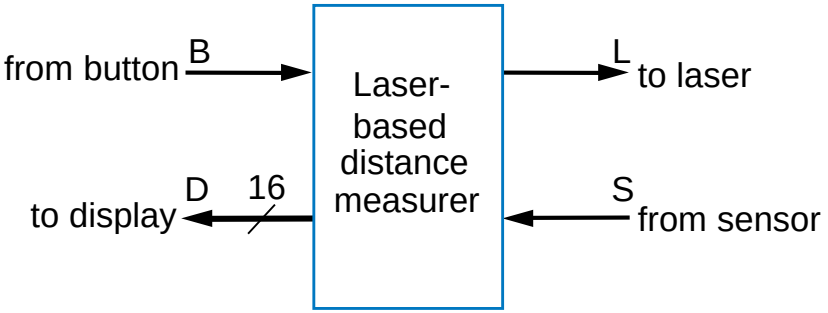


- Add another state, **S1**, that waits for a button press
  - B' – stay in **S1**, keep waiting
  - B – go to a new state **S2**

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

Example: Laser-Based Distance Measurer



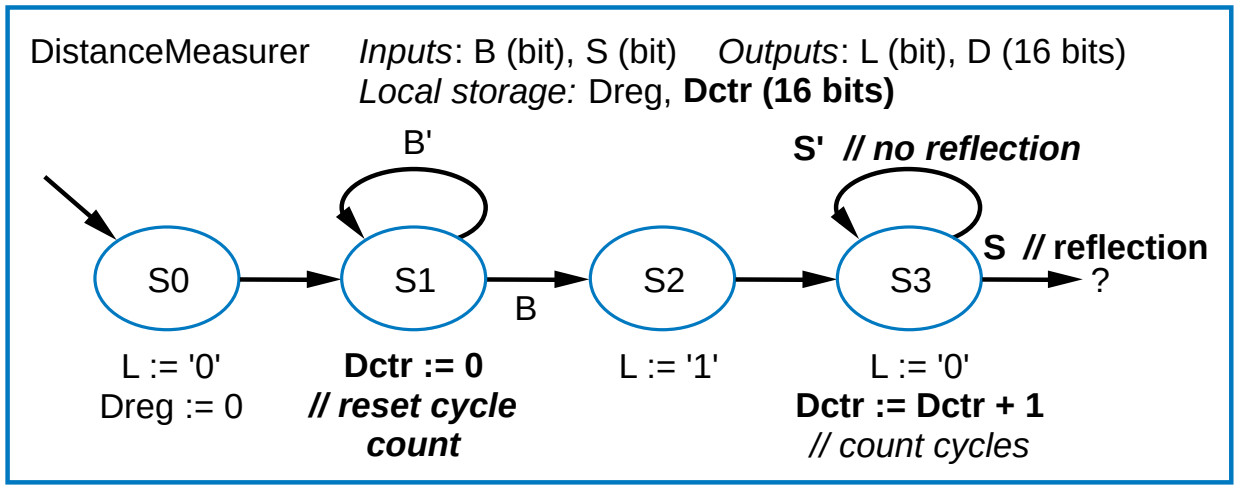


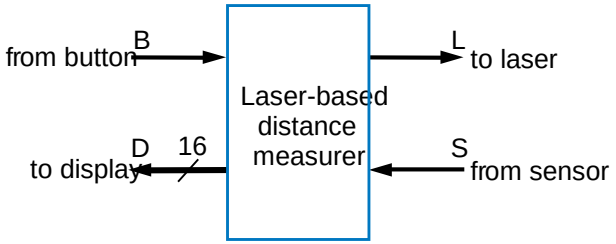
- Add a state **S2** that turns on the laser (L:='1')
- Then turn off laser (L:='0') in a state **S3**

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

a

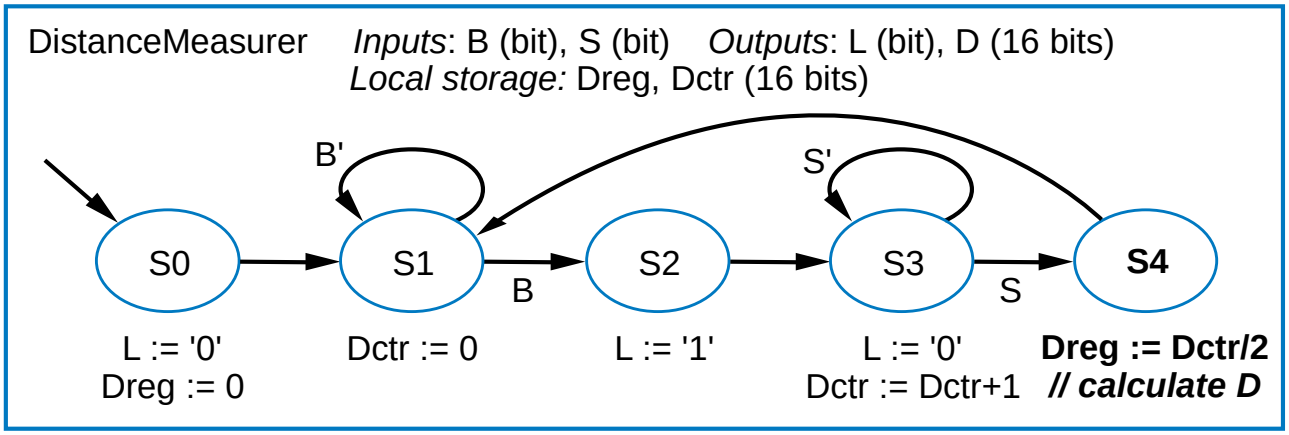
Example: Laser-Based Distance Measurer

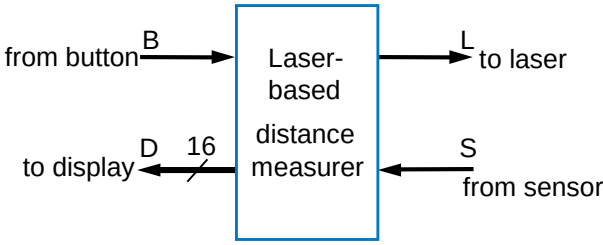




- Stay in **S3** until sense reflection (S)
- To measure time, count cycles while in **S3**
  - To count, declare local storage *Dctr*
  - Initialize *Dctr* to 0 in **S1**. In **S2** would have been O.K. too.
    - Don't forget to initialize local storage—common mistake
  - Increment *Dctr* each cycle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

Example: Laser-Based Distance Measurer

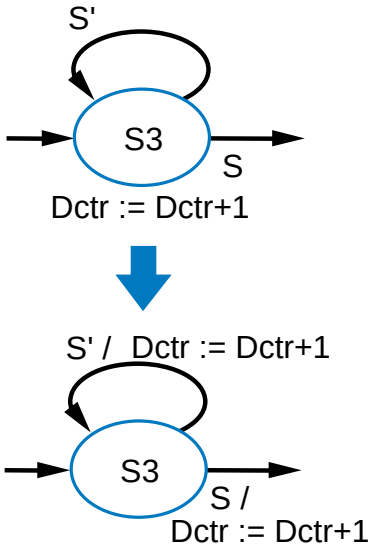




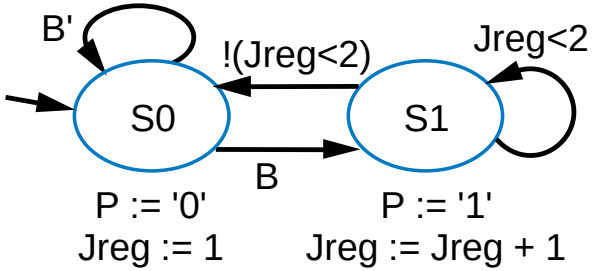
- Once reflection detected (S), go to new state **S4**
  - Calculate distance
  - Assuming clock frequency is 3×10^8, *Dctr* holds number of meters, so Dreg:=Dctr/2
- After **S4**, go back to **S1** to wait for button again

FSMD Actions: Updates Occur Next Clock Cycle

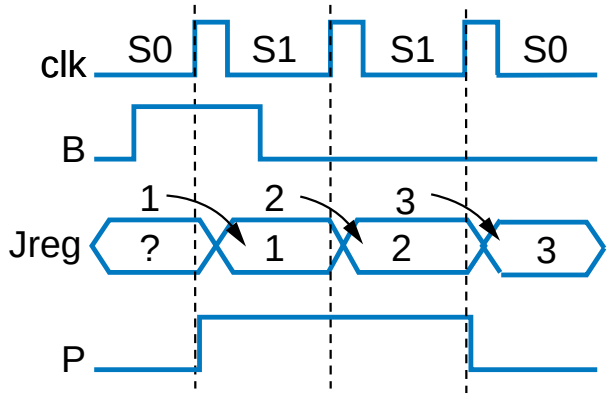
- Local storage updated on clock edges only
  - Enter state on clock edge
  - Storage writes in that state occur on *next* clock edge
  - Can think of as occurring on outgoing transitions
- ***Thus***, transition conditions use the OLD value, not the newly-written value
  - Example:



Inputs: B (bit)
Outputs: P (bit) // if B, 2 cycles high
Local storage: Jreg (8 bits)



(**a**)



(**b**)

Step 2A: Create a Datapath

- Datapath must
  - Implement data storage
  - Implement data computations
- Look at FSMD, do three sub-steps
  - (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

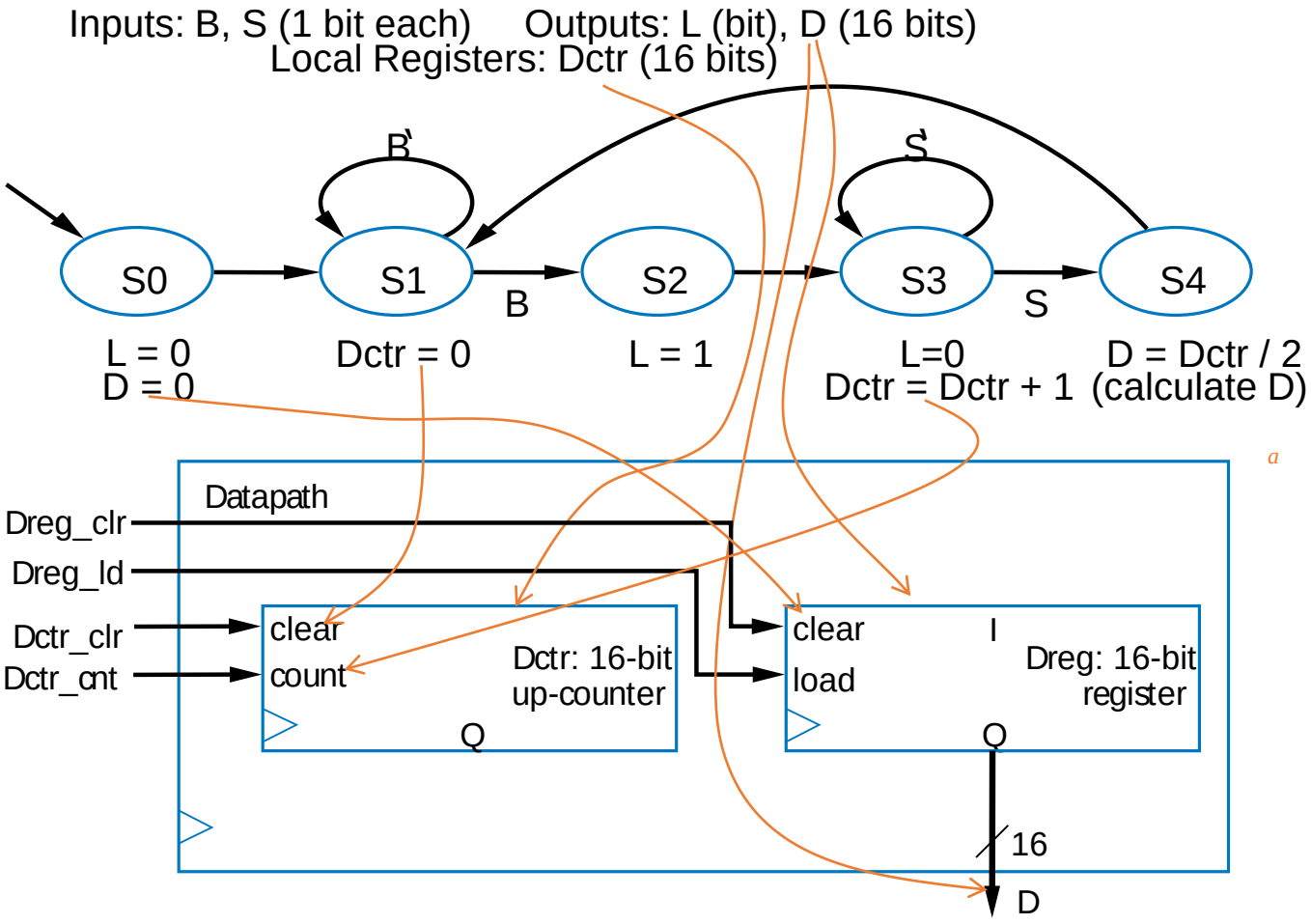
Instantiate: to introduce a new component into a design.

Step 2A: Laser-Based Distance Measurer

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

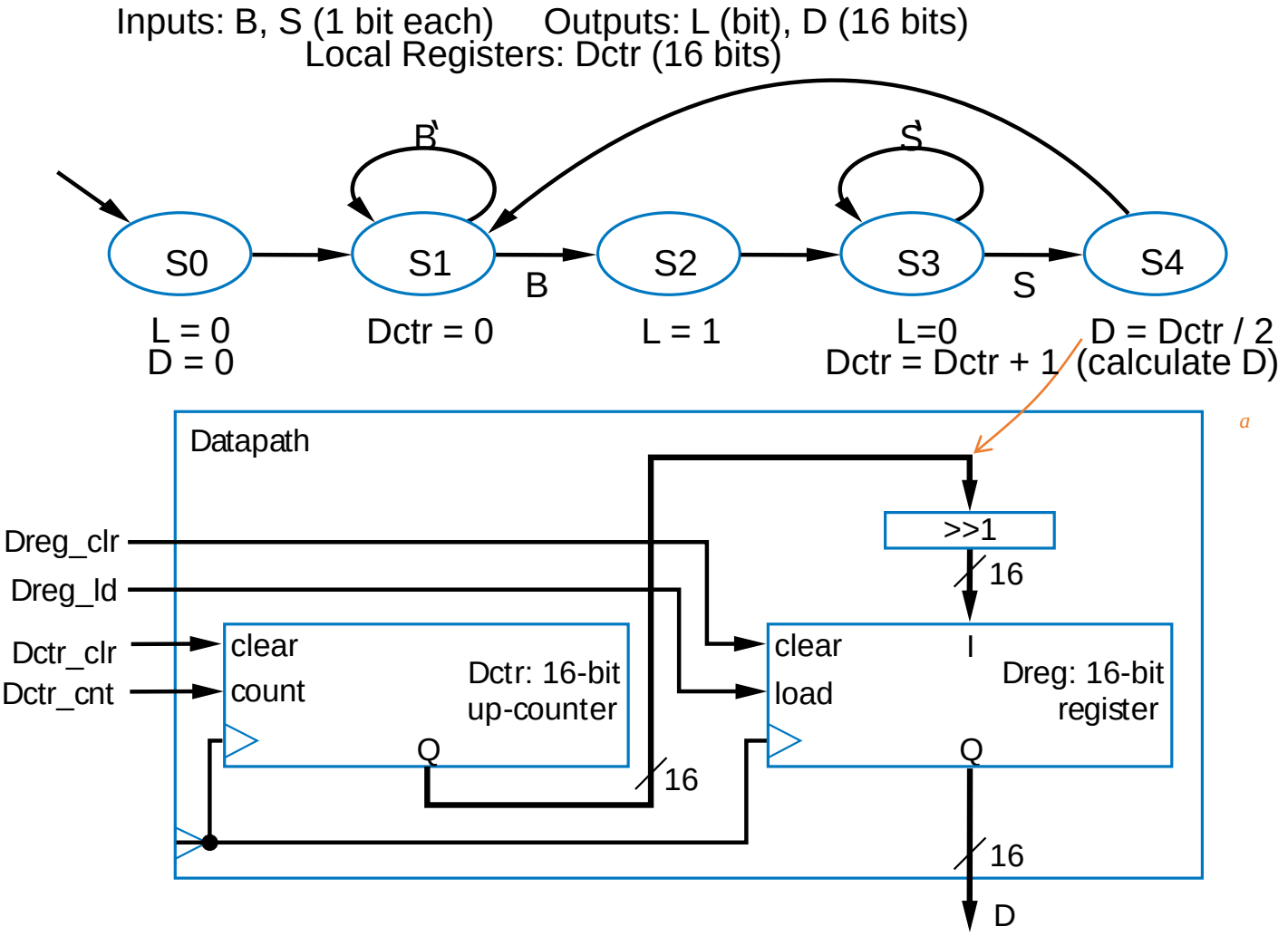
(b) Instantiate declared registers into the datapath (also instantiate a register for each data output)

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

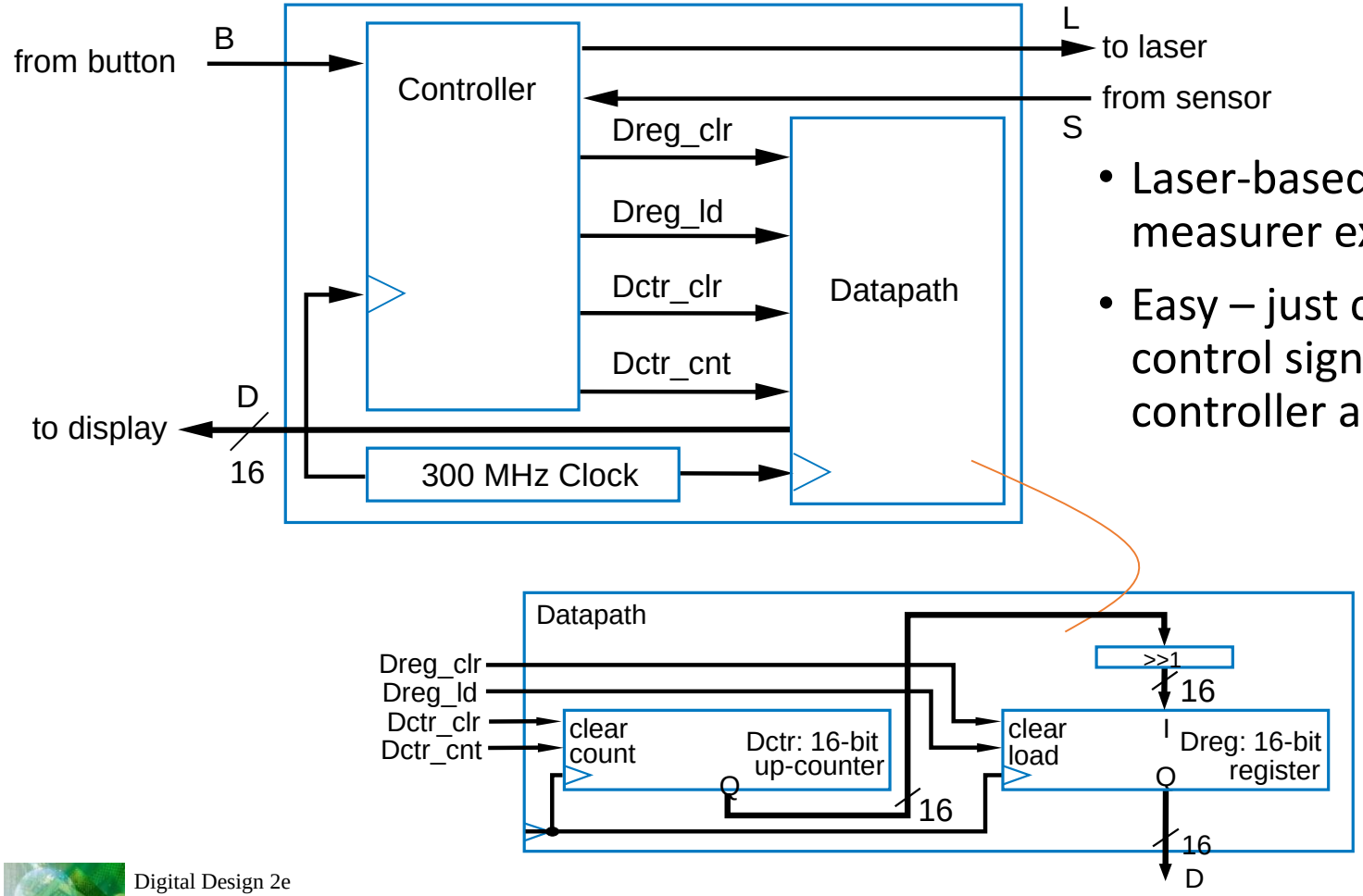


Step 2A: Laser-Based Distance Measurer

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

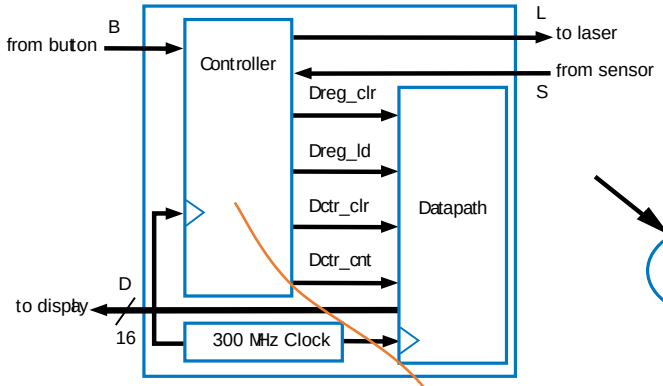


Step 2B: Connecting the Datapath to a Controller

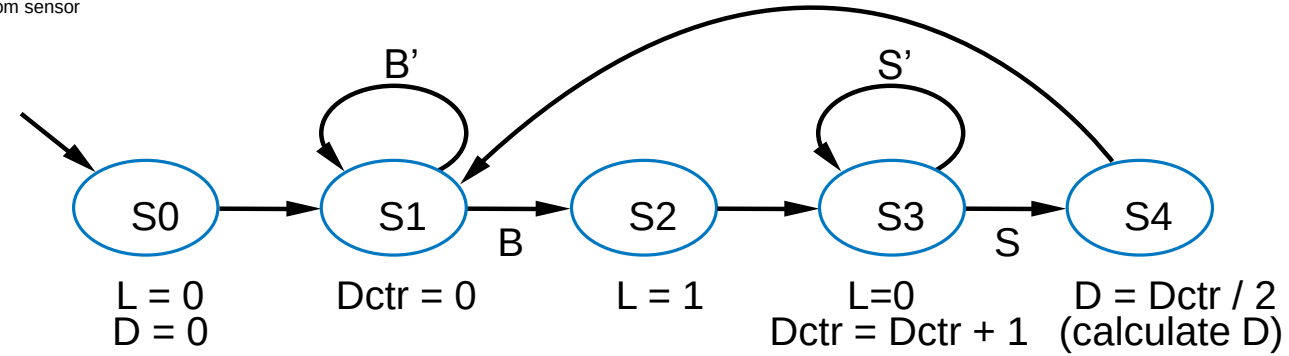


- Laser-based distance measurer example
- Easy – just connect all control signals between controller and datapath

Step 2C: Deriving the Controller's FSM

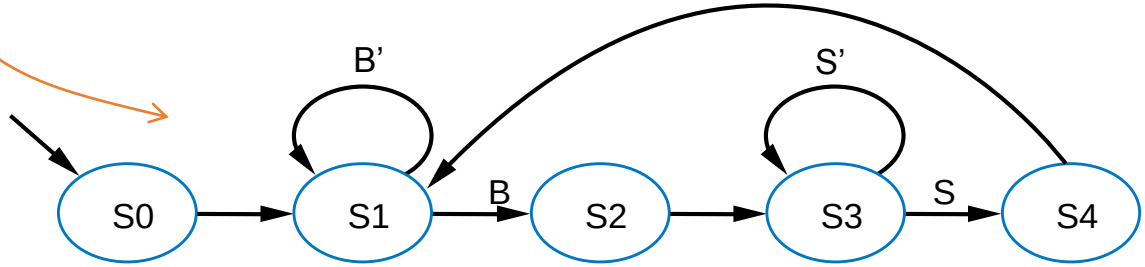


Inputs: B, S (1 bit each) Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)



- FSM has same structure as high-level state machine
  - Inputs/outputs all bits now
  - Replace data operations by bit operations using datapath

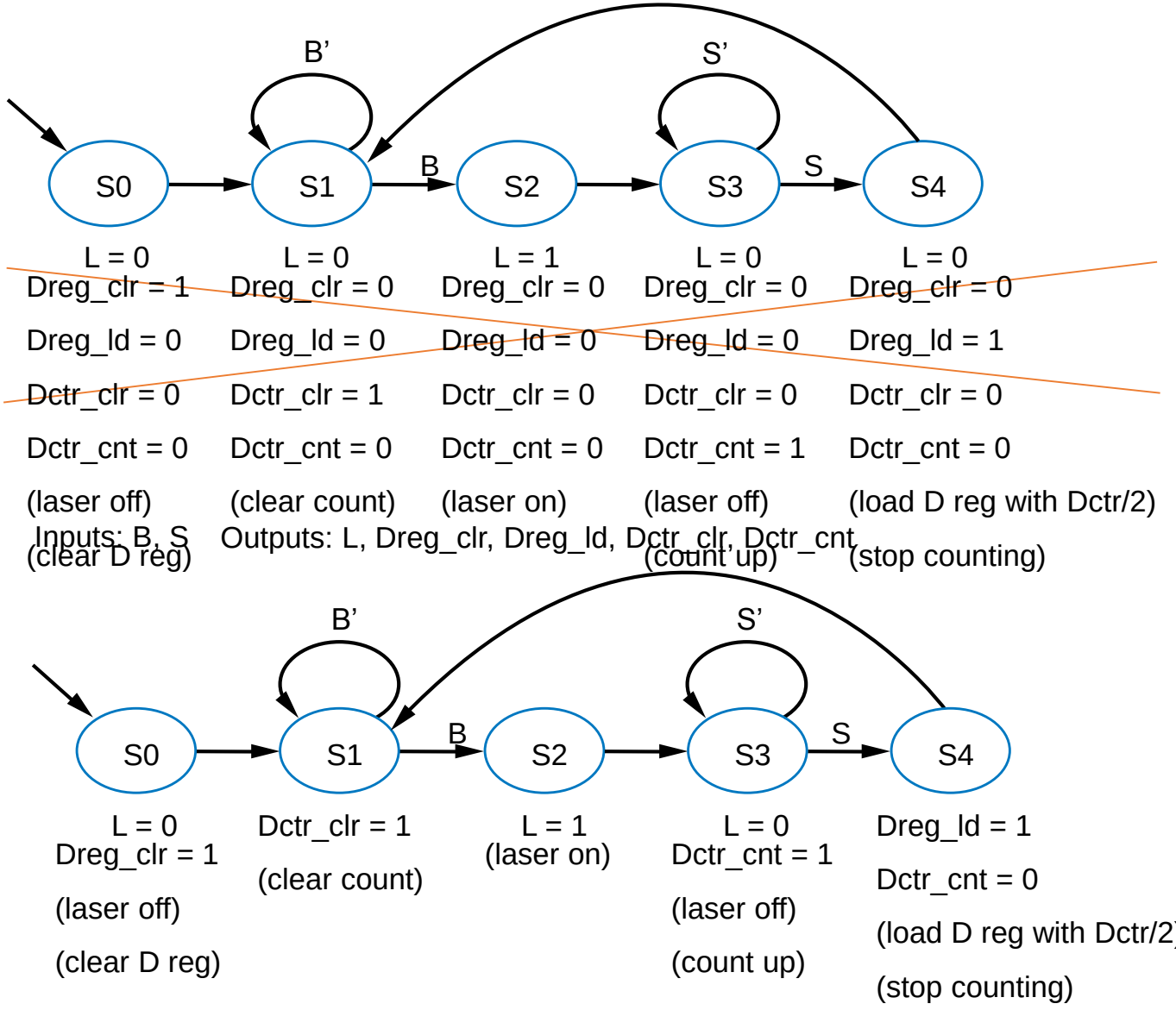
Inputs: B, S
Outputs: L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt



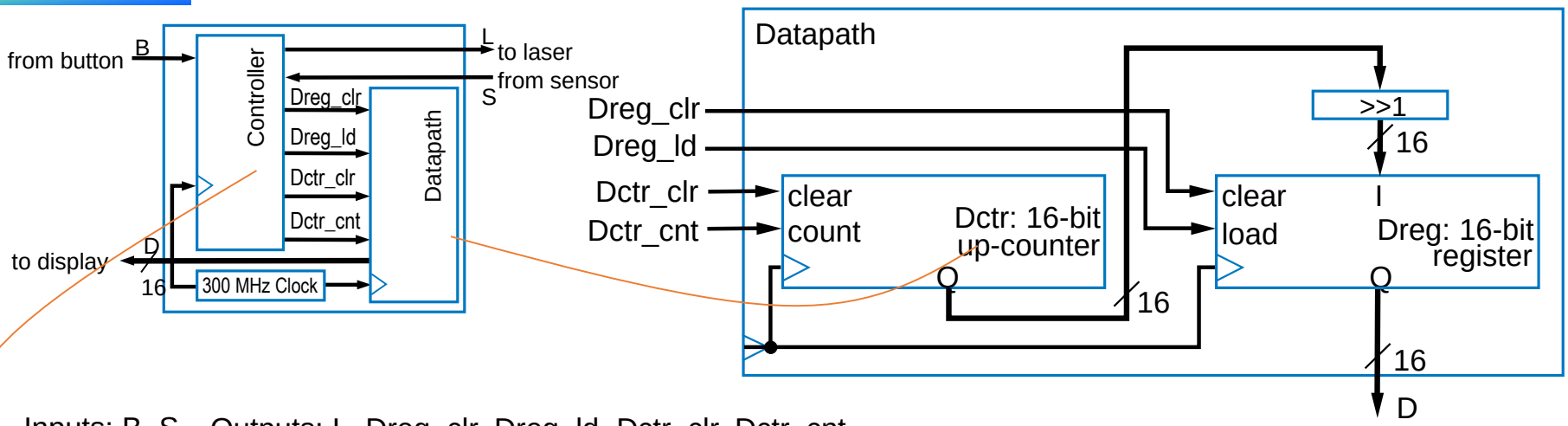
S0	S1	S2	S3	S4
L = 0	L = 0	L = 1	L = 0	L = 0
Dreg_clr = 1	Dreg_clr = 0	Dreg_clr = 0	Dreg_clr = 0	Dreg_clr = 0
Dreg_ld = 0	Dreg_ld = 0	Dreg_ld = 0	Dreg_ld = 0	Dreg_ld = 1
Dctr_clr = 0	Dctr_clr = 1	Dctr_clr = 0	Dctr_clr = 0	Dctr_clr = 0
Dctr_cnt = 0	Dctr_cnt = 0	Dctr_cnt = 0	Dctr_cnt = 1	Dctr_cnt = 0
(laser off)	(clear count)	(laser on)	(laser off)	(load D reg with Dctr/2)
(clear D reg)			(count up)	(stop counting)

Step 2C: Deriving the Controller's FSM

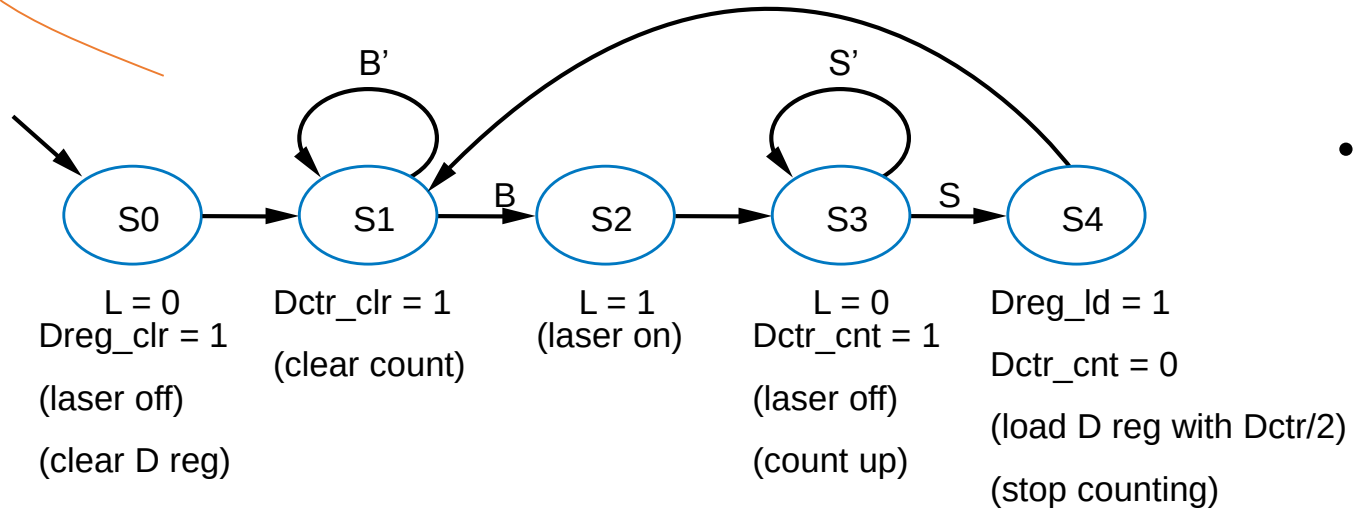
- Using shorthand of outputs not assigned implicitly assigned 0



Step 2C



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



- Implement FSM as state register and logic (Ch3) to complete the design

HLSM of Laser-Based Dist. Measurer: VHDL

- Architecture similar to FSM
  - Curr/next sigs for all regs
- Asynch reset forces to state S0
- Rising clock
  - Perform state's computation
  - Prepare to go to next state based on state and inputs
- Concurrent sig assignment sets D to Dreg always

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LaserDistMeasurer is
    port (
        clk, rst : in  std_logic;
        B, S     : in  std_logic;
        L        : out std_logic;
        D        : out std_logic_vector
                       (15 downto 0)
);
end LaserDistMeasurer;

architecture behavior of
        LaserDistMeasurer is

    type statetype is (S0, S1, S2, S3, S4);
    signal State, StateNext : statetype;

    signal Dctr, DctrNext:
         std_logic_vector(15 downto 0);
    signal Dreg, DregNext:
         std_logic_vector(15 downto 0);

    constant U_ZERO :
         std_logic_vector(15 downto 0)
                := "0000000000000000";
    constant U_ONE  :
         std_logic_vector(15 downto 0)
                := "0000000000000001";

begin

Regs: process(clk, rst)
begin
    if(rst = '1') then
        State <= S0;
        Dctr <= U_ZERO;
        Dreg <= U_ZERO;
    elsif(clk'event and clk='1') then
        State <= StateNext;
        Dctr  <= DctrNext;
        Dreg  <= DregNext;
    end if;
end process;
```

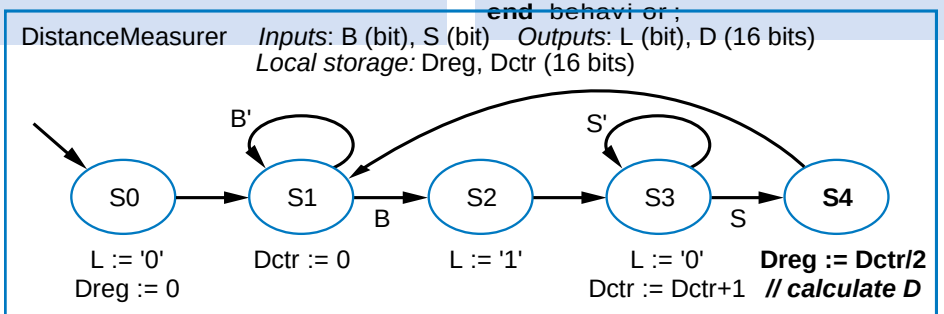
```
CombLogic: process(State, Dctr, B, S)
    begin
        case State is
            when S0 =>
                L <= '0'; -- laser off
                DregNext <= U_ZERO; --clr D
                DctrNext <= U_ZERO; --clr Dctr
                StateNext <= S1;
            when S1 =>
                DctrNext <= U_ZERO; --clr Dctr
                L <= '0'; -- laser off

                if(B = '1') then
                    StateNext <= S2;
                else
                    StateNext <= S1;
                end if;
            when S2 =>
                L <= '1';  --laser on
                DctrNext <= U_ZERO;
                StateNext <= S3;
            when S3 =>
                L <= '0';  -- laser off
                DctrNext <= Dctr + 1;

                if( S = '1') then
                    StateNext <= S4;
                else
                    StateNext <= S3;
                end if;
            when S4 =>
                DctrNext <= Dctr;
                DregNext <= SHR(Dctr, U_ONE);
                L <= '0';
                StateNext <= S1;
            when others =>
                DregNext <= U_ZERO;
                DctrNext <= U_ZERO;
                L <= '0';
                StateNext <= S0;
        end case;
    end process;

--assign Dreg output to D output
D <= Dreg;

end behavior;
```



Optimizing single-purpose processors

- Optimization is the task of making design metric values the best possible
- Optimization opportunities
  - original program
  - FSMD
  - datapath
  - FSM

Optimizing the original program

- Analyze program attributes and look for areas of possible improvement
  - number of computations
  - size of variable
  - time and space complexity
  - operations used
    - multiplication and division very expensive

Optimizing the FSMD

- Areas of possible improvements
  - merge states
    - states with constants on transitions can be eliminated, transition taken is already known
    - states with independent operations can be merged
  - separate states
    - states which require complex operations (a*b*c*d) can be broken into smaller states to reduce hardware size
  - scheduling

Optimizing the datapath

- Sharing of functional units
  - one-to-one mapping, as done previously, is not necessary
  - if same operation occurs in different states, they can share a single functional unit
- Multi-functional units
  - ALUs support a variety of operations, it can be shared among operations occurring in different states

Optimizing the FSM

- State encoding
  - task of assigning a unique bit pattern to each state in an FSM
  - size of state register and combinational logic vary
  - can be treated as an ordering problem
- State minimization
  - task of merging equivalent states into a single state
    - state equivalent if for all possible input combinations the two states generate the same outputs and transitions to the next same state

Chapter Summary

Custom single-purpose processors

- High-level state machines or FSMD
- RTL design process
  - 1. Capture behavior: Use FSMD
  - 2. Convert to circuit
    - A. Create datapath
    - B. Connect DP to controller
    - C. Derive controller FSM
- VHDL model can be directly created from FSMD
  - Another approach is to use a structural VHDL model that consists of controller and datapath components and their connections