SANTA CLARA UNIVERSITY

Fatemeh Tehranipoor, James Lewis, and Tokunbo Ogunfunmi **ELEN/COEN 21L**

Laboratory #8: Data Registers, Shift Registers, and Test Pong Controller

For lab sections Monday-Friday Nov.16 – Nov. 20, 2020

I. OBJECTIVES

In this laboratory you will:

- Design and test a bidirectional shift register.
- Implement and test a four-state finite state machine (FSM) to control a shift register.

PROBLEM STATEMENT

This lab involves designing and implementing the first part of a one-dimensional Pong game, described on the following pages. The game will be completed in lab 9. The display for the game will require a bidirectional shift register to hold the position of the "ball." The shift register will be designed and tested in this lab using flip-flops.

A flip-flop stores one bit of data. The data can change only when an active clock edge appears at the clock input. The most common type of flip-flop is the D-type, which simply transfers the value of the D input to the Q output when the active clock edge occurs.

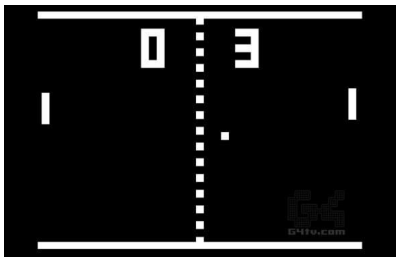
An *n*-bit data storage register stores *n* bits in *n* flip-flops. The flip-flops in the register have a common clock connection so that all *n* bits are changed simultaneously on the same active clock edge.

An *n*-bit shift register will shift the *n* bits which are stored in the *n* flip-flops by loading each flip-flop with the output of its adjacent flip-flop. The shift register can shift bits to the left or to the right. The flip-flop at one end of the shift register will have no adjacent flip-flop to capture its bit, and its output bit will be lost. The flip-flop at the other end of the shift register will have no adjacent flip-flop to provide its input bit, and its input will be connected to an external serial input. All of the flip-flops in the register have a common clock connection so that all *n* bits are changed simultaneously on the same active clock edge.

A shift register can be designed to shift "left" or "right" with the direction defined relative to the names of the flip-flop outputs. For example, *shift right could mean to shift from Q3 toward Q0*. A bi-directional shift register can be designed to shift in either direction depending on a control input. Universal shift registers, which can hold the current value unchanged, load input data, shift data left by one bit, or shift data right by one bit, are an important component of processor systems and of communication between processors and devices and sensors. The shift registers can be used to convert data between serial and parallel form for communication between processing units and communication links. They can also be used for arithmetic operations that multiply or divide by 2.

ONE-DIMENSIONAL PONG GAME DEFINITION AND BACKGROUND MATERIAL

A two-dimensional game of Pong allows a player to move a paddle up and down to hit a ball approaching the player's side of the game area. If the ball is hit by the paddle, it will reverse direction and travel toward the other's player's side of the game area. If a player fails to hit the ball with a paddle, the other player scores a point.



In our one-dimensional game of Pong, a 12-bit bidirectional shift register will replace the two dimensional playing area, and the position of the "ball" as it moves across the playing area will be represented by the state of all flip-flops in the shift register. Player 1 can "serve", which will put a one at the serial input to the shift register for one clock cycle and also set the shift direction to make the "ball" travel to the other end of the shift register. If the "ball" is not returned by a correctly timed "paddle" input, it will shift out of the serial output and into an end flip-flop. Player 1's score will be incremented and no further change will occur until one of the players "serves" again. (You can make your own rules about when each player serves.)

The **game will be implemented in two labs**. In this first lab, one of the players will be able to serve, but there will be no paddle input, so it will not be possible to "return" the ball. This will allow the serving and shifting to be checked. Also, a simple version of the pong controller will be implemented. A bidirectional shift register will track the position of the "ball" as it moves left and right. For this lab, we will just use a 4-bit shift register. In lab 9 we will extend this to the full 12 bits.

Inputs, Outputs, and States for *pong_controller1*

Pong has two players, RIGHT and LEFT. In controller 1, which is the version of the controller to be implemented in this lab, only the right player can serve and neither player has an active paddle. The following is a definition of the inputs, the outputs, and the four states for the controller.

Controller Inputs

iRSRV - is the input initiating a serve from player RIGHT.

iRESET –causes a synchronous reset to the idle state from any other state.

QLEFT – the input is 1 when the ball position in the shift register is at the leftmost position just before it shifts out of the shift register

Controller Outputs

LSI – This will be connected to the Left Serial In (LSI) bit of the shift register, and is meant to simulate a ball being put into play when player RIGHT serves.

S[1:0] – These are the two signals that will be used to control the shift register, which you will be designing in your pre-lab. (More details in the pre-lab section which follows.)

Symbolic state definitions used for pong_controller1:

sIdle

In the idle state the game is waiting for a left serve or a right serve input. It is entered by a reset from any state or as the next state after the ball gets to the end of either side without being returned. In the idle state, S = 11 which causes the shift register to be cleared on an active clock edge. It will stay in this state unless there is a right serve input, iRSRV, which causes the next state to be sRightPlayerServes. This is the default state.

sRightPlayerServes

In the right serve state, the RIGHT player is serving the ball. This state is entered only from sIdle when iRSRV is 1. The output will be S=10 so that on the next clock edge, the shift register will shift the ball to the left. The controller output LSI is 1 so that a 1 will be shifted into the shift register. The next state is always sMoveLeft unless there is a reset input.

sMoveLeft

In the move left state, the shift register shifts one position to the left on each clock pulse until the ball exits the shift register. The controller will stay in this state until the leftmost bit of the shift register is 1 or a reset input is 1. When QLEFT is 1, the next state will be sEndLeft. In the move left state S=10 so that the next clock edge always shifts the shift register contents to the left. All other outputs are 0.

sEndLeft

In the end left state, the ball has shifted out of the leftmost position without being returned by the LEFT player. In controller 1, this state is entered only from sMoveLeft when QLEFT is 1. The output will be S=00 so on the next clock edge, the shift register will hold the current values, which should all be zero. The next state is always sIdle.

State Table

Current State	Next State	Outputs		
		LSI	S[1]	S[0]
sIdle	sRightPlayerServes if iRSRV is 1, otherwise sIdle	0	1	1
sRightPlayerServes	sIdle if iRESET is 1, otherwise sMoveLeft	1	1	0
sMoveLeft	sIdle if iRESET is 1, sEndLeft if QLEFT is 1, otherwise sMoveLeft	0	1	0
sEndLeft	sIdle	0	0	0

II. PRE-LAB

In the pre-lab you will design the universal shift register component and you will draw the state transition diagram for the pong controller1.

Design the universal shift register component:

(i) Draw a schematic of a 4-bit data storage register using positive edge triggered D-type flip-flops.
The register is clocked by a signal CLK.
The register data input is a 4-bit input word X[3:0].
The register output is a 4-bit word Q[3:0].

(ii) Modify the register in (i) to be a 4-bit loadable shift right register that can operate as in (i) when input LOAD = 1 and can shift stored data by one bit to the right when input LOAD is 0. In this context**, shifting "to the right"** means shifting in the direction from the **most significant bit, Q[3] toward the least significant bit Q[0]**. When shifting right, an input RSI (right serial input) is shifted into the Q[3] flip-flop. The value in Q[0] before the active clock edge is lost after the active clock edge. (Note: This is an arithmetic point of view where shifting right should shift Q_n to Q_{n-1}.)

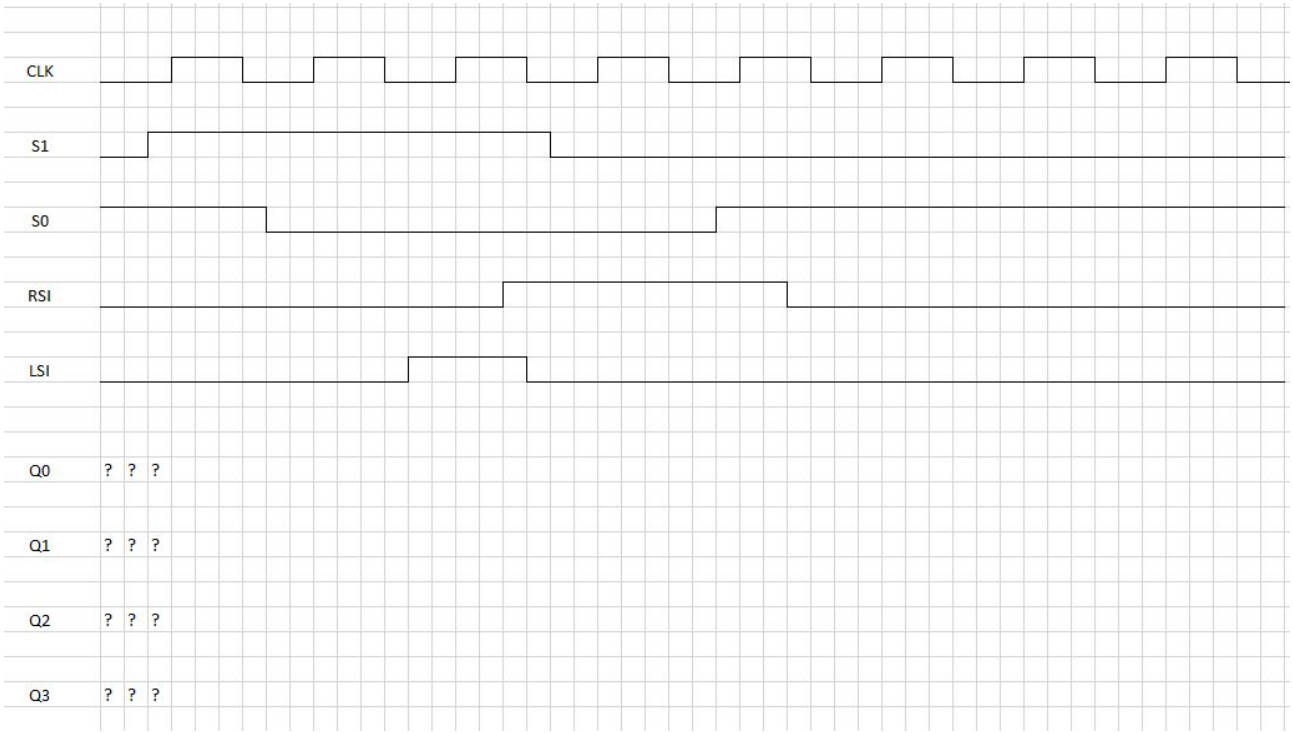
Draw this loadable shift register schematic using your circuit from (i) and 2:1 multiplexers. Clearly show what is connected to each of the data inputs of the flip-flops.

(iii) Modify the loadable shift register in (ii) to be a 4-bit "universal shift register" with four possible operations. On an active clock edge, it can shift in either direction, parallel load a 4-bit value (as in (i)) or simply not change the current values of the flip-flops. Two control signals, S1 and S0, will determine the operation as described in the table below, where $Q_k(t)$ represents the value of Q[k] before the active clock edge and $Q_k(t+1)$ represents the value of Q[k] after the active clock edge. There will also be two inputs, RSI and LSI, which are the external serial inputs to be used when shifting right or left respectively. This shift register will be the basis for the display in the one-dimensional pong game.

Draw the schematic for the universal shift register clearly indicating what is connected to each of the data inputs of the flip-flops. Use 4:1 multiplexers instead of 2:1 multiplexers.

S1	S0	operation	For internal flip-flops
0	0	Hold data unchanged	$Q_k(t+1) = Q_k(t)$
0	1	Shift right	$Q_k(t+1) = Q_{k+1}(t)$
1	0	Shift left	$Q_k(t+1) = Q_{k-1}(t)$
1	1	Load from an external input	$Q_k(t+1) = X_k(t)$

(iv) Complete the timing diagram below for your universal shift register. Assume that X[2]=0 and all the other X inputs are 1. The X inputs do not change. Think of a systematic approach for testing your shift register, and write the first four steps of your test plan.



Make a state transition diagram:

(v) Draw the state transition diagram for the four-state *pong_controller1* from the state table in the controller definition section above.

(vi) State assignment: Choose binary values for each of your four states.

Submit your schematics, answers to questions, and timing diagram for the PreLab.

III. PROCEDURE

Part 1: Implement and Test a Verilog 4-bit Universal Data Shift Register, *USR4*.

1. Write a Verilog module, *USR4*, for your 4-bit universal shift register based on your pre-lab part (iii) schematic. The inputs will be X[3:0], RSI, LSI, S[1:0], and CLK. The output will be Q[3:0]. An if-else structure or a case statement can be used for the multiplexed inputs.
2. Add one instance of *USR4* to a Verilog testbench. Using the example of the testbench you used in the previous lab, instantiate a clock signal and connect it to the CLK input of your shift register.
3. Add the capability to your testbench for controlling the other inputs to shift register, and to observe the 4-bit Q output.

4. Check that the shift register operation is correct using a test plan. Try various inputs and ensure that the register is loading your data correctly. Make sure that the least significant bit of your input is also the least significant bit of the 4-bit output.
5. Demonstrate your shift register to you lab assistant.

Part 2: Implement and Test a first Verilog *pong_controller1*

6. Copy Verilog module, *USR4*, to new module *pong4*, which will be a **modified 4-bit universal shift register**. It will not have the 4-bit parallel input X[3:0]. When S is 11, the shift register value will be set to 0000 instead of loading from an external input.
7. Write Verilog module *pong_controller1*. See the text Figure 6.29 (on the last page of this lab assignment) for one example of a state machine implementation. Compare your two state assignment values from your prelabs and select one of them for the parameter statement.
    a. Controller inputs: Clock, QLEFT, iRESET, and iRSRV.
    b. Controller outputs: S[1:0] and LSI.
    c. This controller should also provide your present state as an output, as it will be helpful to be able to observe the state of the controller during simulation.

You will now create a top-level module that includes both your controller module and the shift register. In this module, the controller module will now provide the shift register inputs rather than having them be generated by code in your testbench, as was done in Part 1.

8. Add one instance of *pong4* and one instance of *pong_controller1* to the top-level module.
    a. As in Part 1, connect your clock signal to the CLK input of *pong4*. And also connect it to the Clock input of *pong_controller1*.
    b. Connect the Q[3] output from *pong4* to the QLEFT input of *pong_controller1*.
    c. Connect the LSI and S outputs of *pong_controller1* to the corresponding inputs of *pong4*. We are not going to use RSI in this lab, but we should never leave inputs unconnected. So make some connection to this input. You can set it to a constant 1 or 0, or you can connect the LSI signal to it if you like.
9. Instantiate your top-level module in a testbench.
10. Add the capability in your testbench to control the iRSRV and iRESET input signals.
11. Make sure your testbench has the ability to observe the outputs of *pong4,* as well as the outputs of your controller that indicate its present state.

Testing:

12. Assert the reset input and verify that the controller goes to the idle state. Then set the reset input back to zero
13. Verify that the controller stays in the idle state, across several clock cycles, as long as the serve input signal is not asserted.
14. Provide a right serve input, and verify that the state changed to sRightPlayerServes.
15. In the following clock cycles, verify that the state changes sequence through the correct states and that the controller then goes back to the idle state.

Demonstrate your working Pong1 game to your lab assistant. If time permits, add additional states so that player LEFT can also serve.

IV. REPORT

1. Include schematics, test plan, Verilog, and observed results.
2. Describe the most challenging part of this lab.
3. For lab part 1, if you loaded the register with 1101, what value would that represent if you interpreted it as an unsigned integer? With RSI=0, if you shifted to the right what binary pattern would you see and how would it be interpreted as an unsigned number? With RSI=0, if you shifted to the right a second time what binary pattern would you see and how would it be interpreted as an unsigned number? Explain how right shifting is related to dividing by 2.
4. In the previous question, for unsigned interpretation of the values, RSI was set to zero. To divide by 2 using a right shift for signed integer interpretation using 2's complement representation, what should RSI be? Why?
5. How would two 4-bit universal shift registers be connected to form an 8-bit universal shift register? Show a schematic in which each 4-bit register is shown as a block component with inputs and outputs. Do not show the internal components and connections of the 4-bit universal shift register.
6. For lab part 2, if the controller is in the idle state and the right serve input is high during an active clock edge, what is the next state? For the next 6 active clock edges, specify the state that the controller is in after each clock edge.
7. If by mistake QLEFT were connected to Q[0] instead of Q[3], how would your answer to the previous question change?
8. Show how you would modify your state transition diagram **to allow both players to serve**. How many additional states would you need? What additional inputs and outputs would be needed?

REFERENCE:

Verilog examples of finite state machines can be found in the text in Figures 6.29 and 6.95. Figure 6.29 is copied below.

For clarity in your implementation of *pong_controller1*, use "NSTATE" for next state instead of "Y" in Figure 6.29. Use "PSTATE" for the present state instead of "y" in Figure 6.29. The default next state should be sIDLE.

```
module simple (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output z;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

   // Define the next state combinational circuit
   always @(w, y)
      case (y)
         A: if (w)    Y = B;
            else      Y = A;
         B: if (w)    Y = C;
            else      Y = A;
         C: if (w)    Y = C;
            else      Y = A;
         default:     Y = 2'bxx;
      endcase

   // Define the sequential block
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

   // Define output
   assign z = (y == C);

endmodule
```

Figure 6.29 Verilog code for the FSM in Figure 6.3.