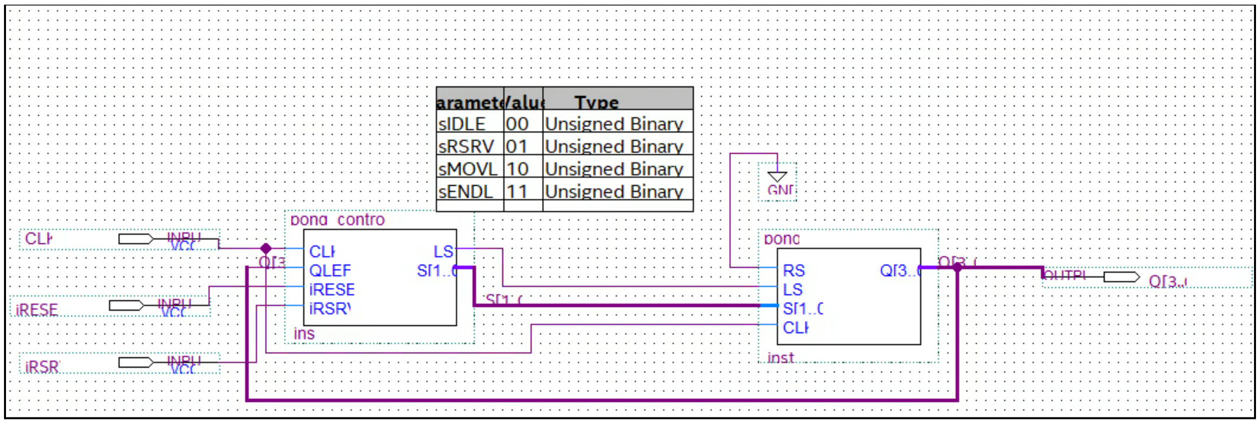
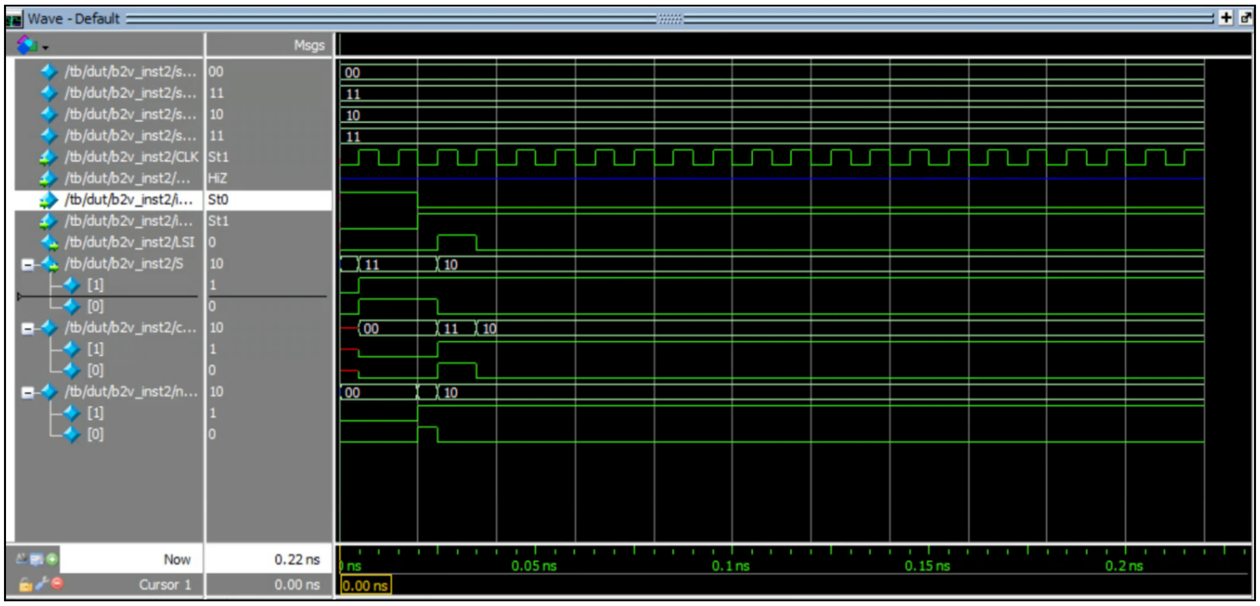
1. **Schematics, test plan, Verilog, and results:**
- All the modules/verilog used, as well as the schematic and waveforms included in this lab report. My test plan was to use the modules in the simulation. Everything worked according to plan. All of my Verilog files are attached in the hastebins below.

- USR4
- pong_controller1
- pong4
- tb
- top



aramete	/alue	Type
sIDLE	00	Unsigned Binary
sRSRV	01	Unsigned Binary
sMOVL	10	Unsigned Binary
sENDL	11	Unsigned Binary



Wave - Default
Msgs
/tb/dut/b2v_inst2/s... 00
/tb/dut/b2v_inst2/s... 11
/tb/dut/b2v_inst2/s... 10
/tb/dut/b2v_inst2/s... 11
/tb/dut/b2v_inst2/CLK St1
/tb/dut/b2v_inst2/... HiZ
/tb/dut/b2v_inst2/i... St0
/tb/dut/b2v_inst2/i... St1
/tb/dut/b2v_inst2/LSI 0
/tb/dut/b2v_inst2/S 10
[1] 1
[0] 0
/tb/dut/b2v_inst2/c... 10
[1] 1
[0] 0
/tb/dut/b2v_inst2/n... 10
[1] 1
[0] 0
Now 0.22 ns
Cursor 1 0.00 ns
0 ns
0.05 ns
0.1 ns
0.15 ns
0.2 ns
0.00 ns

Schematic Explanation:
- iRESET, which causes a synchronous reset to the idle state from any other state, takes it to the idle state.
- The state is idle 00 until iRSRV is 1.

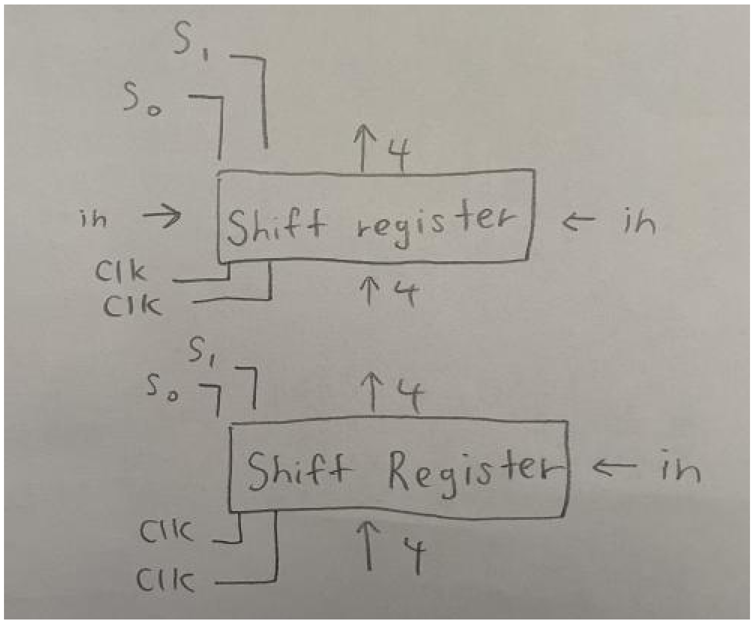
- When that happens, LSI becomes 1 and I send it to the next state sRSRV
- And I go to sMOVL state, which shifts the 1 through Q until state ENDL gets active
- Then it just goes back to idle. The state changes sequence through the correct states and the controller then goes back to the idle state.
- The state changes at positive edges of the clock and ignores negative ones as specified in the Verilog code.

2. The most challenging part of this lab was remembering how to use the Quartus software correctly as I needed to check the previous lab instructions to complete this lab. Once I figured out the software, everything was good.

3. 1101 would represent 13 in decimal. When shifted right with RSI=0, 0110 would represent 6 in decimal. If I shifted to the right a second time, 0011 would represent 3 in decimal. Shifting right with RSI=0 will change the initial value x into the value (x-a)/2, where a=0 if the LSB is 1 before shifting, or a=1 if the LSB is 0 before shifting. This is because the value of the rightmost bit is subtracted, and all of the other bits move right, dividing each of their values by two.

4. To divide by 2 using signed integer interpretation, I needed to set RSI to match the value of the MSB before shifting. This makes sure that negative numbers stay negative and positive numbers stay positive.

5.



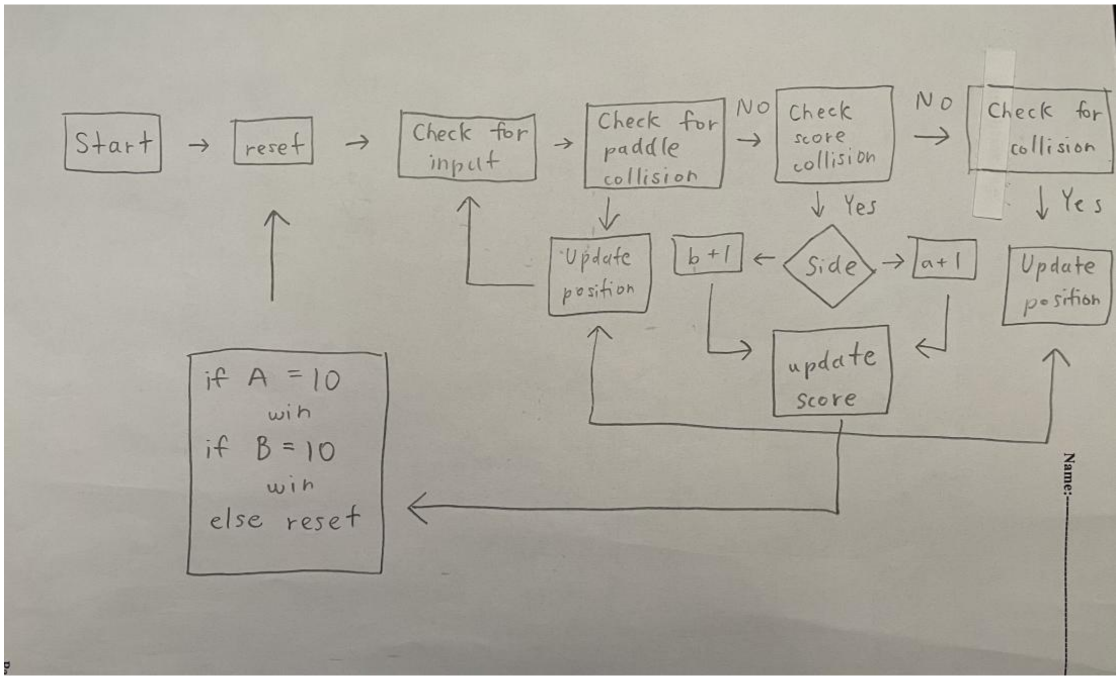
Above is a schematic in which each 4-bit register is shown as a block component. The two 4 bit universal registers can be connected to the universal 8 bit register by combining the two 4 bit outputs of the 4-bit registers into an 8-bit register since the input for the 8 bit register needs to be an 8-bit value.

6. Assuming I start in idle:
Clock Edge: [State] [Q]
1: RightPlayerServes 0001
2: MoveLeft 0010
3: MoveLeft 0100
4: MoveLeft 1000
5: EndLeft 1000
6: Idle 0000
7: RightPlayerServes 0001

7. If by mistake QLEFT were connected to Q[0] instead of Q[3], the code would cycle through each state with exactly one clock edge for each state as seen below.

Assuming I start in idle:
1: RightPlayerServes 0001
2: MoveLeft 0010
3: EndLeft 0010
4: Idle 0000
5: continues….

This is because when the code enters the MoveLeft state, it sees that Q[0] is already equal to 1, so it determines that the ball is on the left. This makes the next state is EndLeft. It switches after one clock cycle, causing the program to cycle through all states like this forever.



8. Above is a diagram which would allow both players to serve (assuming 10 points is a win).

If there is another player, there has to be another module that controls the paddle movement for that additional player, there would be 4 additional states for this player. There would need to be additional inputs and outputs to keep track of the collisions between the paddles and the balls, since the current inputs and outputs keep track of the collisions with the wall and only one, not both of the paddles.