Drew Clark
EE 271 – Summer 2018

# Lab 6: Two-Player Pong

**Abstract:**
The purpose of this lab was to implement everything we learned this quarter into a final project. A "Two-Player Pong" game was created allowing two players to compete on the DEC_SoC board with an 8x8 LED_Matrix as the play field. KEY[3:0] were used to move the paddles, SW[0] was used as a reset, and SW[9:6] were used to adjust the speed of the ball. The game is played by volleying the pong ball between player one and player two's paddles, a round is won when the ball makes it past the opponents paddle. HEX5 and HEX0 display the score for players one and players two respectively, counting up to 7.

**Introduction:**
A two player game of Pong is to be built on the DEC-SoC board. In order to move your paddle KEY[3] and KEY[2] are used for players one to move the paddle left and right, respectively, and KEY[1] and KEY[0] is used for player two. The ball and 2x1 paddles are displayed on an 8x8 LED matrix with the player one paddle on the top row, and player two paddle on the bottom row. Only one LEDR is to be illuminated at a time for the ball. The game is won if the ball light moves past the paddle into the top or bottom row of the LED matrix (i.e. bottom row, player one wins, top row, player two wins). The overall score is kept on HEX5 and HEX0 for player 1 and player 2 respectively, after a game is won the score is updated with the objective of being the first to win seven. The game is reset using SW[0]. The paddle requirements are shown in Table 1 below.

*Table 1 – User Paddle Requirements*

| SW[0] | KEY[3] or KEY[1] | KEY[2] or KEY[0] | Meaning |
|---|---|---|---|
| 0 | 0 | 0 | Hold |
| 0 | 0 | 1 | Paddle moves right (KEY[3] -> top paddle, KEY[1] -> bottom paddle) |
| 0 | 1 | 0 | Paddle moves left (KEY[2] -> top paddle, KEY[0] -> bottom paddle) |
| 0 | 1 | 1 | Paddle moves left (KEY[2] -> top paddle, KEY[0] -> bottom paddle) |
| 1 | X | X | Reset |

The ball is simply run off of two counters, one for horizontal movement, and another for vertical movement. These counters have the ability to change direction if given an input to do so (i.e. if ball hits wall or paddle, it needs to change direction accordingly).

The interaction between the ball and the paddle is outlined by the requirements below:

- If ball "hits" paddle straight on (same column), the ball should reverse vertical direction and hold horizontal direction
- If ball "hits" paddle on the edge it is travelling towards (one column off), the ball should reverse vertical and horizontal direction.
- If ball doesn't "hit" paddle as specified above, it travels into the top or bottom row of the matrix and the round is over with a score given to the opposite team.

This design will be built with the built in 50MHz clock on the DEC_SoC1 board divided to a speed of about 768 Hz. User input should be filtered such that the output is TRUE for only one cycle for every button press (i.e. can't hold down a button). Metastability must also be dealt with for user inputs.

All components are run on the same clock, but in order to slow down the ball speed, and internal counter is used. Once the counter reaches a set maximum, a flag is activated to move the ball one space. The maximum for this counter is controlled by SW[9:6] such that the ball speed is adjustable by toggling the switches.

**Materials:**

- Altera Terasic DE1-SoC development board with power and USB cables
- PC with Altera Quartus Prime 17.0 and ModelSim
- 8x8 LED Matrix

**Procedure:**

The first step for this sequential logic design was to create a block diagram based off of the requirements from Table 1 and the ball/paddle interface requirements. Modules are needed to deal with metastability, filter user input, managing the paddles, determining the ball coordinates, managing the ball/paddle interface, and determining the winner/keeping score. The block diagram can be seen in Figure 1 on the next page.

Each block on the block diagram represents a single module of code, some modules call other modules. The next step was to design each module which included building a state diagram and a state table if it was an FSM. SystemVerilog code was then written based on the requirements, and a testbench module was created to simulate the module and compared to the state table or expected outputs.

<u>meta module:</u>

The meta module is simply two DFF's in series. A given input will go through two clock cycles and be given as the output. The systemVerilog code for this was written and can be found in the appendix. A testbench module was made for simulation and tested to ensure the output was delayed two clock cycles to equal the input.

All user input was run through the meta module, KEY[3:0] and SW[0], except SW[9:6] to control ball speed.

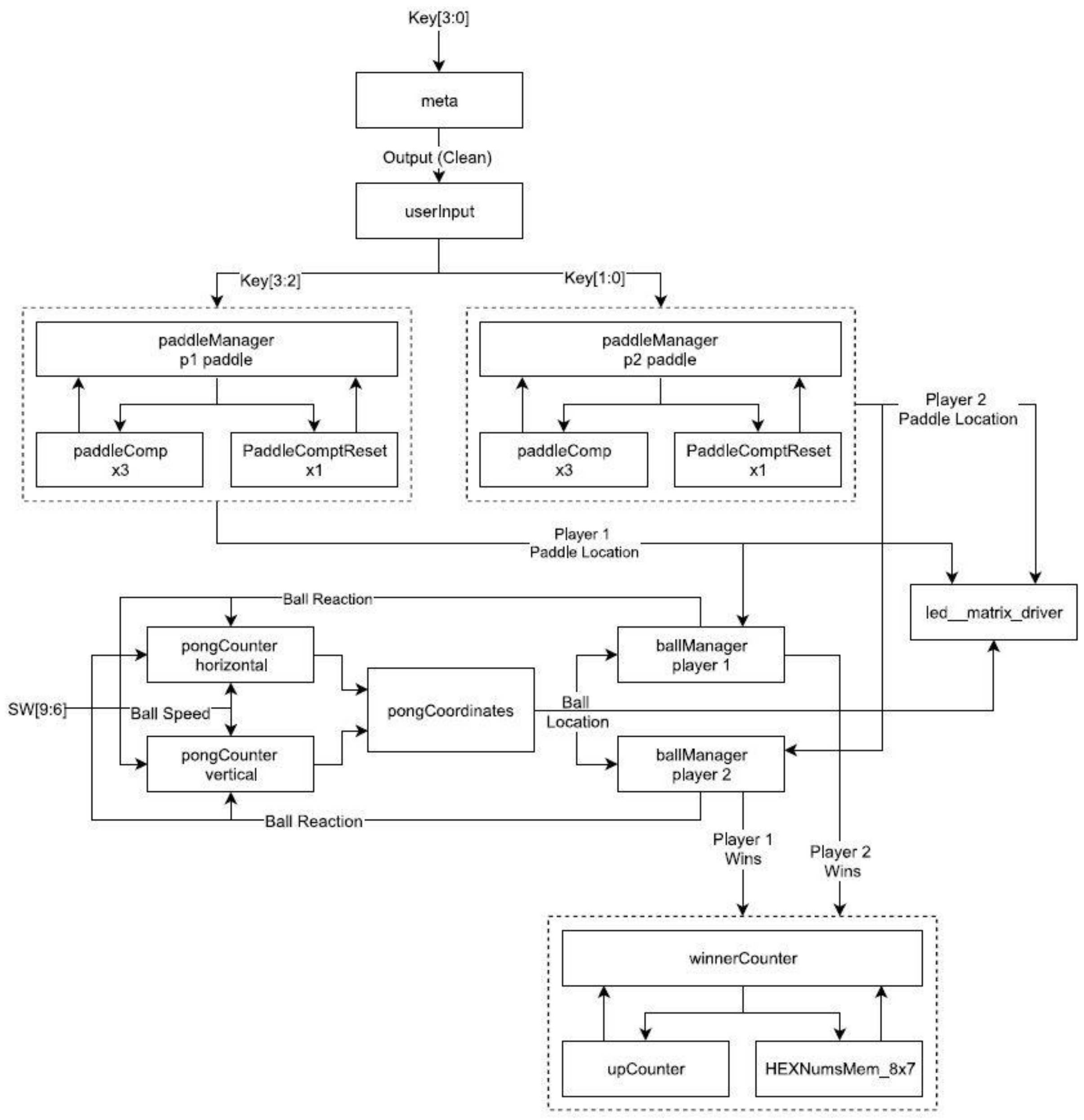


*Figure 1 - Block Diagram*

<u>userInput module:</u>

The userInput FSM required two states, an input, and an output. The input is controlled by the user, after running through the meta module. The output is only true once for each key press, so initially when the input changes to TRUE, the output will become TRUE, but if the input remains true for greater than one clock cycle, it will become FALSE. The state diagram and state table are shown in Figure 2 and Table 2 respectively.

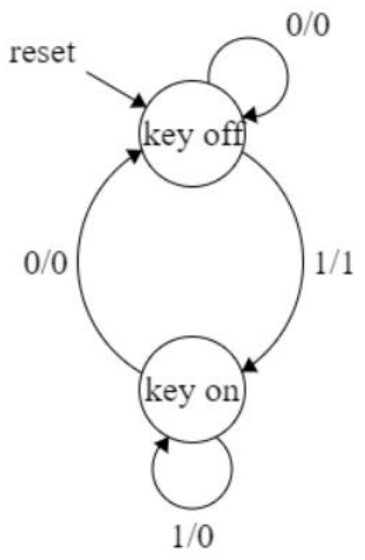


*Figure 2 - userInput FSM diagram*

*Table 2 – userInput state table*

| *PS* | *keyIn* | *keyOut* | *NS* |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |

The systemVerilog code for this was written based on the state diagram and table above and can be found in the appendix. A testbench module was made for simulation and tested against the state table.

<u>paddleManager, paddleComp, and paddleCompReset modules:</u>
The paddleManager is simply an interface that calls paddleComp 3 times and paddleCompReset once. It outputs a [7:0] array to for the paddle to be fed into the LED matrix. It was chosen to implement the paddleManager to make the game more easily scalable. For example, for a 16x16 LED matrix, you would then output a [15:0] array and call paddleComp 7 times and paddleCompReset once.

paddleComp represents 2 LED lights and requires four states, one for each combination of lights (both on, both off, left on only, right on only). There are three inputs: two inputs from the userInput module that represent which key is pressed and which direction the LEDR should move, and a third enabling input. The enable input is used when the module is in the “off” state, the only way it can move out of the “off” state is if the enable is TRUE. The enable input is sent by a bordering module when the LED need to move past it’s border towards the current module (i.e. for the paddleComp representing row 2 and 3 (p23), if the paddleComp representing row 4 and 5 (p45) is in the “On” state and a key is pressed to move the paddle right, 5 needs to turn off, 4 needs to remain on, and 3 needs to turn on, so an enabling flag will be sent from be output from p45 to enable p23 to accomplish this.)

There are four outputs: two outputs controlling the two LEDs and two output for enabling the other modules. The state diagram and state table for paddleComp are show in Figure 3 and Table 3 respectively to further demonstrate this. For the State diagram inputs and output are as follows (L)(R)(En) / (LED[1])(LED[0])(EnL)(EnR)

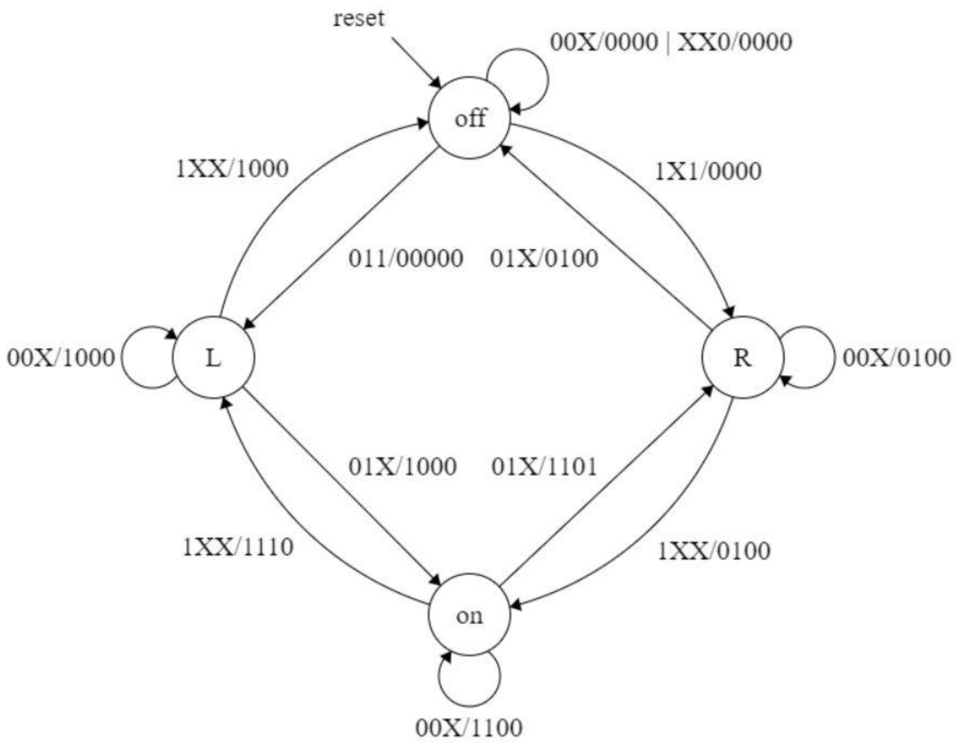


*Figure 3 - paddleComp FSM state diagram*

*Table 3 - State table for paddleComp*

| *En* | *PS$_1$* | *PS$_0$* | *L* | *R* | *LED[1]* | *LED[0]* | *EnL* | *EnR* | *NS$_1$* | *NS$_0$* |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| X | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| X | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| X | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| X | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| X | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| X | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| X | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| X | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| X | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| X | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| X | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |

paddleCompReset is the same as paddleComp except the reset is set to the "on" state as opposed to the "off" state.

The systemVerilog code for this was written for each module and a test bench module was created to compare it against the state tables. The code can be found in the appendix.

<u>pongCounter:</u>
The pong counter requires the counter to be able to increment or decrement depending on which direction the ball is moving. Furthermore, once the maximum or minimum is reached (000) or (111) respectively for the 3-bit counter used for our 8x8 led matrix, the counter needs to switch to incrementing, or decrementing respectively. The pongCounter module is called twice, once for the vertical movement of the ball, and once for the horizontal movement.

These requirements were implementing by using a simple FSM with two states, forward (F) for incrementing, and reverse (R) for a decrementing counter. The state diagram and state table are shown in Figure 4 and Table 4 respectively. For the state table, "CD" represents change direction (True when direction should be changed) and "D" represents direction "True when incrementing).

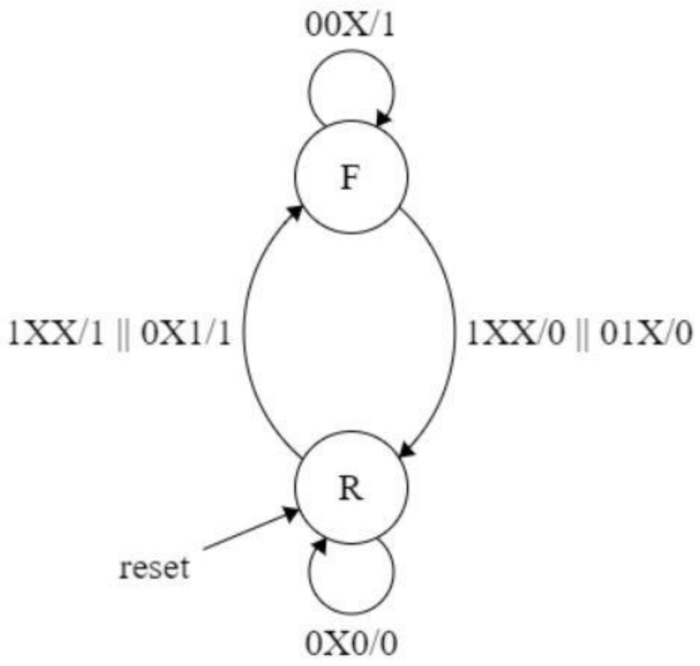


*Figure 4 - pongCounter state diagram*

*Table 4 - State table for pongCounter*

| *NS* | *CD* | *Max* | *Min* | *D* | *NS* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Additionally, a 9-bit up counter is used internally for the ball speed with a 9-bit input for the counter maximum.  If the counter reaches the counter maximum, a “moveBall” flag is set to true, otherwise the “moveBall” flag is false.  The counter maximum is adjustable using SW[9:6] which control the first 4 bits of the maximum, the last 5 bits are set to 1. So that the minimum the counter maximum can be is 000011111.

Finally, the output from the FSM above “Direction” and from the ballSpeed counter “moveBall” are fed into the ballCounter which runs off of the clock.  If “moveBall” is false, the counter holds, else it will increment if “Direction” is True or decrement if “Direction” direction is false.  This is shown in Table 5 below.

*Table 5 – pongCounter Requirements*

| moveBall | Direction | Meaning |
|---|---|---|
| 0 | 0 | Hold Ball |
| 0 | 1 | Hold Ball |
| 1 | 0 | Decrement Ball Location by 1 |
| 1 | 1 | Increment Ball Location by 1 |

The systemVerilog code for this was written for each module and a test bench module was created to compare it against the state tables.  The code can be found in the appendix.

<u>pongCoordinates:</u>

pongCoordinates simply takes the horizontal pongCounter and the vertical pongCounter and feeds them into a 3:8 decoder with an 8-bit output.  The decoder takes the binary value from the counter and makes that value of the 8-bit output 1, and all other values 0.  This 8-bit output serves as an index, the horizontal represents the column index and the vertical represents the row index.

pongCoordinates then takes these two 8-bit arrays, and traverses them to make an 8x8 matrix with all 0's except a 1 based on the column and row index. This 8x8 matrix represents the ball coordinates and is output to be fed into the LED matrix.

The systemVerilog code for this was written for each module and a test bench module was created to compare it against the state tables. The code can be found in the appendix.

ballManager:

The ballManager module take in the paddle location and ball location to perform the ball/paddle interaction. Since we only care about the ball/paddle interaction when the ball is traveling towards the paddle and is within one row of the paddle, a simply FSM design was created with 2 states, "off" and "evaluate", with inputs of "ball", "paddle flat" and "paddle edge", and outputs "change vertical direction" and "change horizontal direction".

"ball" is true when the direction of the ball travel is towards the paddle AND the ball is one row away from the paddle. "paddle flat" is true when the ball column is the same as one of the paddle columns. "paddle edge" is true when "paddle flat" is false, the ball is moving toward the paddle horizontally, and the ball is one column away from the paddle. The state diagram and table are shown in Figure 5 and Table 6 respectively.

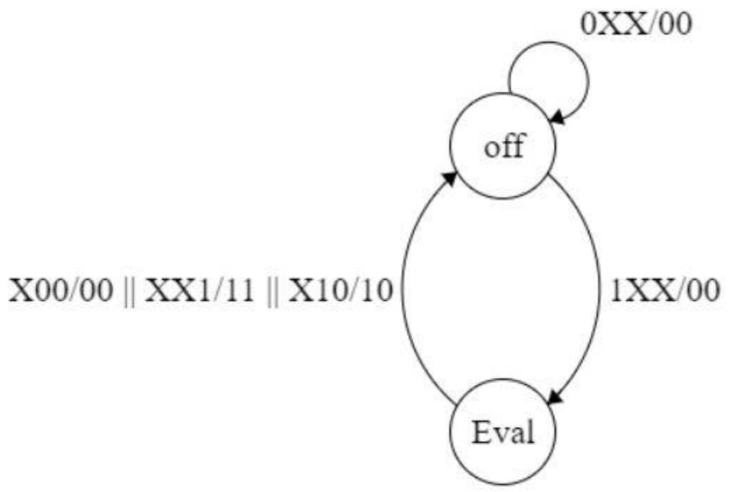


*Figure 5 - ballManager state diagram*

*Table 6 - State table for ballManager*

| *PS* | *Ball* | *$P_F$* | *$P_E$* | *$DC_V$* | *$DC_H$* | *NS* |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |

ballManager is called twice, once for paddle1 interaction, and once for paddle2 interaction.

Additionally, ballManager has an output “winner”. “winner” is set true if the ball location is ever in the top or bottom row, meaning the ball has made it past the paddle. This output is fed into the winnerCounter module.

The systemVerilog code for this was written for each module and a test bench module was created to compare it against the state tables. The code can be found in the appendix.

<u>winnerCounter:</u>
The purpose of this module is to update the scoreboard and playfield after each game is finished. Inputs “winnerOne” and “winnerTwo” are passed on from the ballManager if the LEDR ball enters the bottom or top row respectively. This inputs are passed into a counter to keep track of how many games are won. The counter is then passed through a ROM which takes in a binary number from the counter and outputs a corresponding 7-segment pattern to display the number in decimal on HEX5 for playerOne and HEX0 for playerTwo. winnerCounter also outputs a signal to reset the playfield once a game is won. There is also an input for a reset which resets the scores to 0, and an input for the clock

<u>upCounter:</u>
Simple up counter with size controlled by parameter WIDTH. For this lab a 3-bit counter is required to count to 7, so WIDTH is set equal to 3. winnerCounter includes an upCounter for each player passing on winnerOne or winnerTwo to increment the counter, and SW[0] reset to reset the counters to 0.

<u>HEXNumsMem_8x7:</u>
Memory that converts 3-bit binary input to 7-segment pattern for decimal equivalent. winnerCounter passes along data from upCounter as the address into HEXNumsMem_8x7 which then returns a 7-bit binary number for the 7-segment display on the DE_SoC1 board.

The code for these modules were written in SystemVerilog and a test bench module was created, simulated, and compared it to the requirements. The code can be found in the appendix.

Code was then written to map to the FPGA on the DEC-SoC board which can be found in the Appendix named “DE1_SoC.sv”. SW[0] was used for the reset and KEY[3:0] were used for the paddle inputs. SW[9:6] were used to control the ball speed.

The design was downloaded to the FPGA and the switches, KEY, and LED Matrix were used to test the implementation against the requirements and state tables.

**Results:**
The meta code worked as expected when written into SystemVerilog. The output was delayed two clock cycles. The simulation of meta is shown in Figure 6.

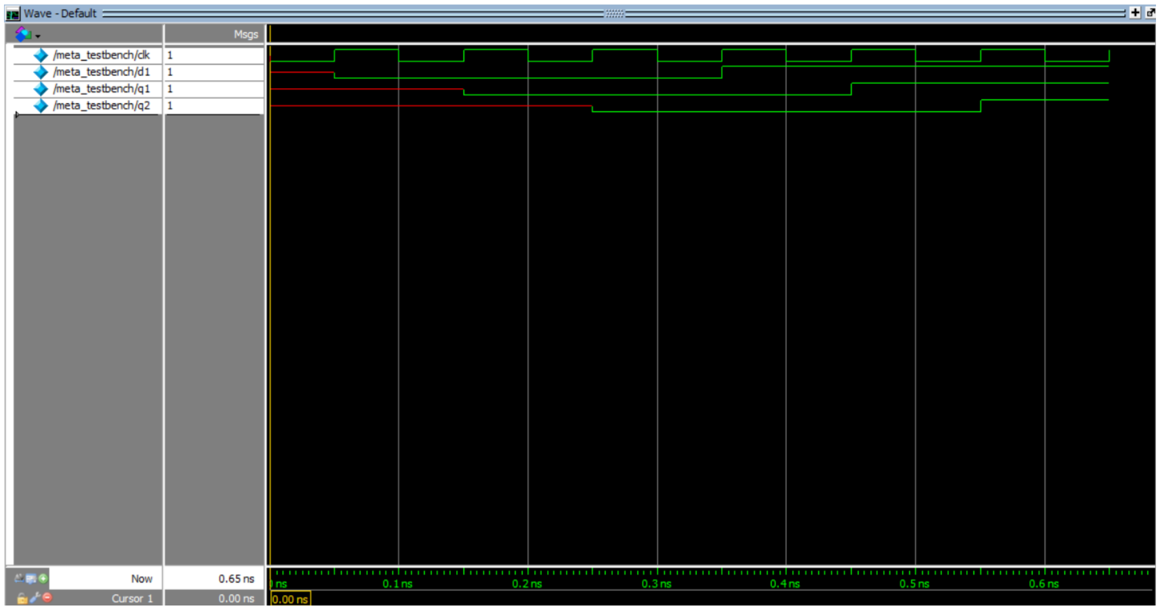


*Figure 6 - meta_testbench Simulation*

The userInput code worked as expected. When input turned TRUE, output became TRUE for one cycle and then return to FALSE if input stayed TRUE. The simulation of userInput is shown in Figure 7.

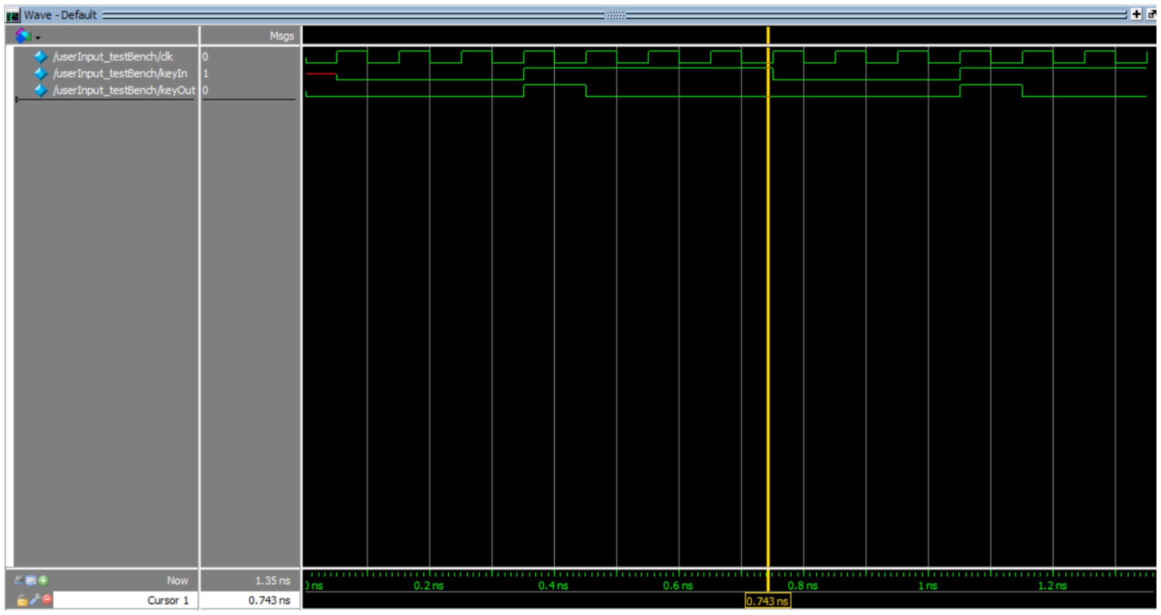


*Figure 7 - userInput_testbench Simulation*

The paddleManager code worked as expected by calling paddleComp and paddleCompReset appropriately and outputting an 8-bit array for the paddle location. The simulation for paddleManager is shown in Figure 8.

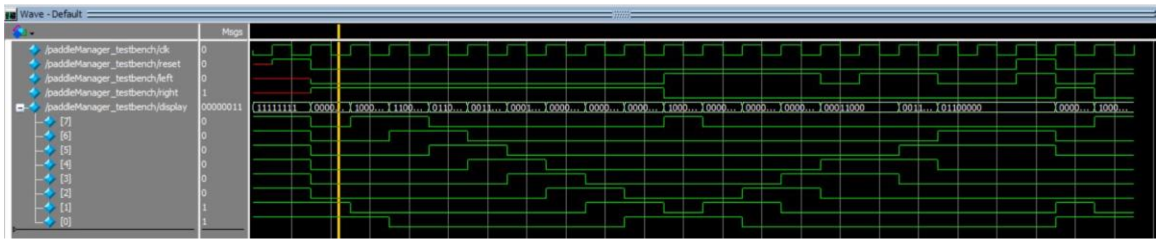

Wave - Default
Msgs
/paddleManager_testbench/clk
/paddleManager_testbench/reset
/paddleManager_testbench/left
/paddleManager_testbench/right
/paddleManager_testbench/display
00000011

*Figure 8 - paddleManager_testbench Simulation*

The paddleComp and paddleCompReset code worked as expected. The simulations were compared to Table 3. The simulations for paddleComp is shown in Figure 9.

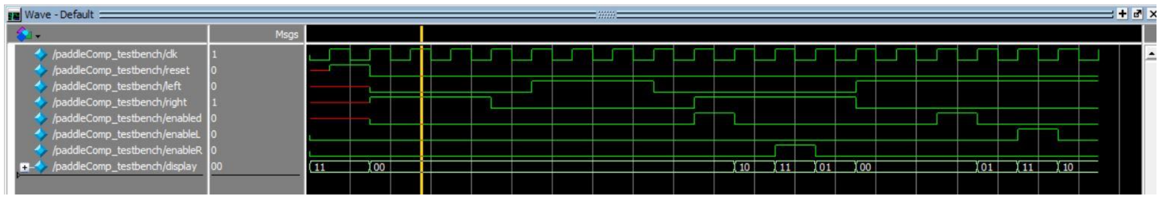

Wave - Default
Msgs
/paddleComp_testbench/clk
/paddleComp_testbench/reset
/paddleComp_testbench/left
/paddleComp_testbench/right
/paddleComp_testbench/enabled
/paddleComp_testbench/enableL
/paddleComp_testbench/enableR
/paddleComp_testbench/display

*Figure 9 - paddleComp_testbench Simulation*

The pongCounter code worked as expected. Test scenarios were used for when the ball hit walls or a changeDirection flag was input and the simulations were compared to Table 4. The simulation for pongCounter is shown in Figure10.

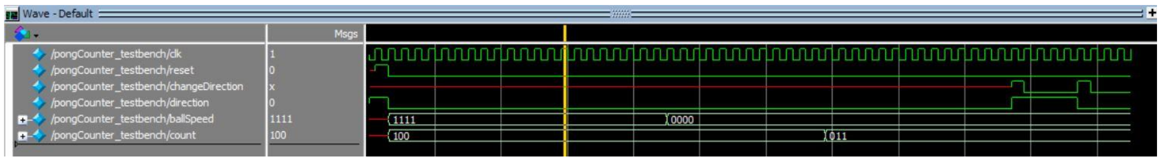

Wave - Default
Msgs
/pongCounter_testbench/clk 1
/pongCounter_testbench/reset 0
/pongCounter_testbench/changeDirection x
/pongCounter_testbench/direction 0
/pongCounter_testbench/ballSpeed 1111
/pongCounter_testbench/count 100

*Figure 10 - pongCounter_testbench Simulation*

The pongCoordinates code worked as expected. Testing showed that one, and only one, light was always true and was where it was expected based on the counter inputs. The simulation for pongCoordinates is shown in Figure 11.

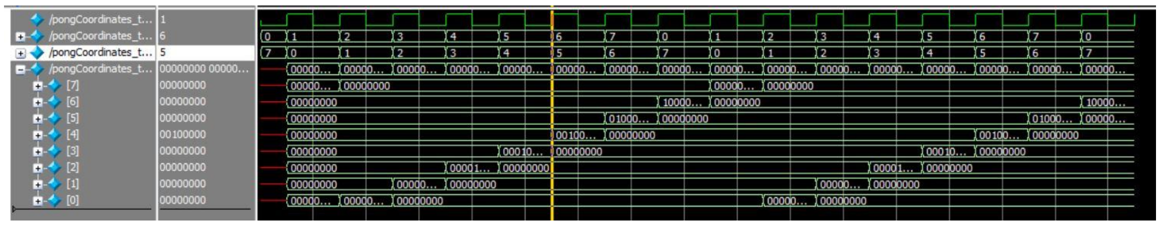


*Figure 11 - pongCoordinates_testbench Simulation*

The ballManager code worked as expected. Test scenarios were used for when the ball hit an edge of the paddle, flat part of the paddle, or missed the paddle and the simulation was compared to Table 6. The simulation for ballManager is shown in Figure 12.

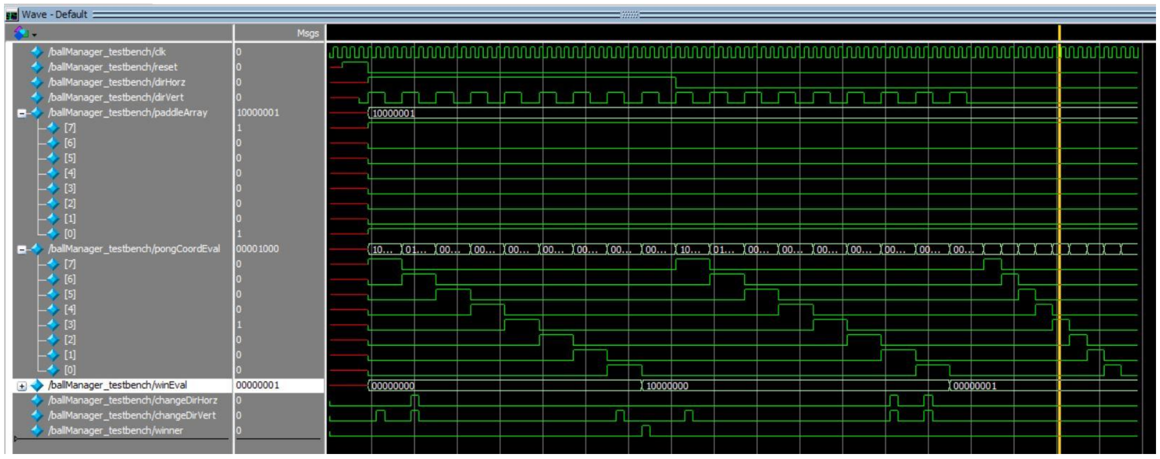

Wave - Default
Msgs
/ballManager_testbench/clk 0
/ballManager_testbench/reset 0
/ballManager_testbench/dirHorz 0
/ballManager_testbench/dirVert 0
/ballManager_testbench/paddleArray 10000001
/ballManager_testbench/pongCoordEval 00001000
/ballManager_testbench/winEval 00000001
/ballManager_testbench/changeDirHorz 0
/ballManager_testbench/changeDirVert 0
/ballManager_testbench/winner 0

*Figure 12 - ballManager_testbench Simulation*

The upCounter worked as expected when written into SystemVerilog. The value was increased by 1 with every increment from 0 to 7 and then restarted at 0. The simulation of meta is shown in Figure 13.

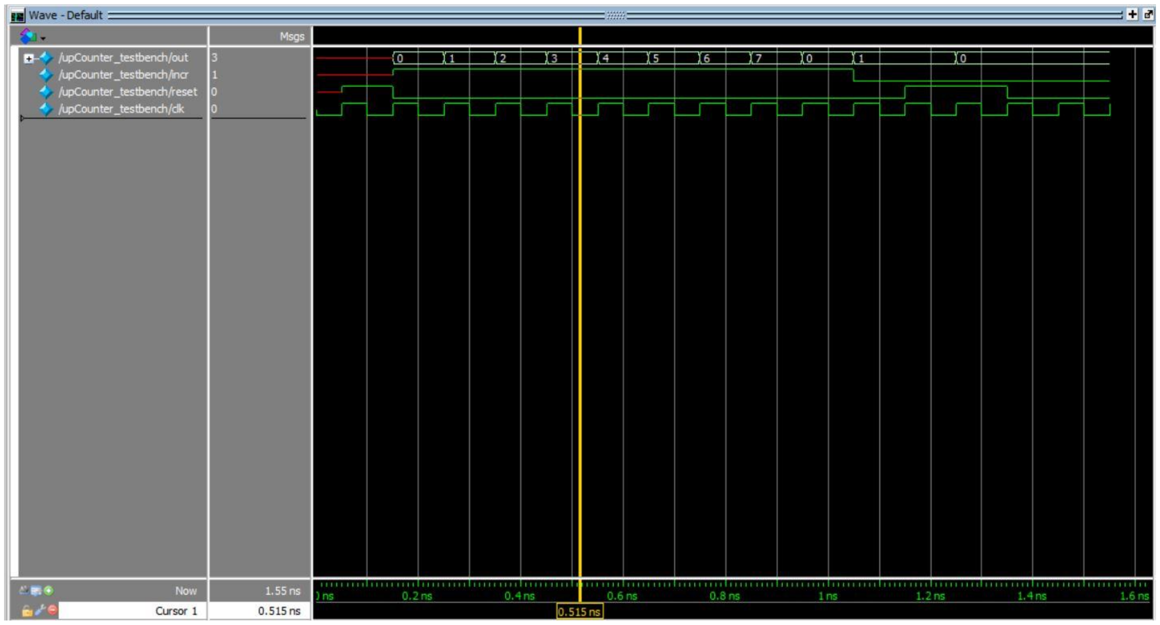

Wave - Default
Msgs
/upCounter_testbench/out 3
/upCounter_testbench/incr 1
/upCounter_testbench/reset 0
/upCounter_testbench/clk 0
0 1 2 3 4 5 6 7 0 1 0
Now 1.55 ns
Cursor 1 0.515 ns
0.515 ns

*Figure 13 - upCounter_testbench Simulation*

The HEXNumsMem_8x7 code worked as expected. Each binary address from 0-7 resulted in the correct 7-segment binary output for the decimal equivalent. The simulation is shown in Figure 14.

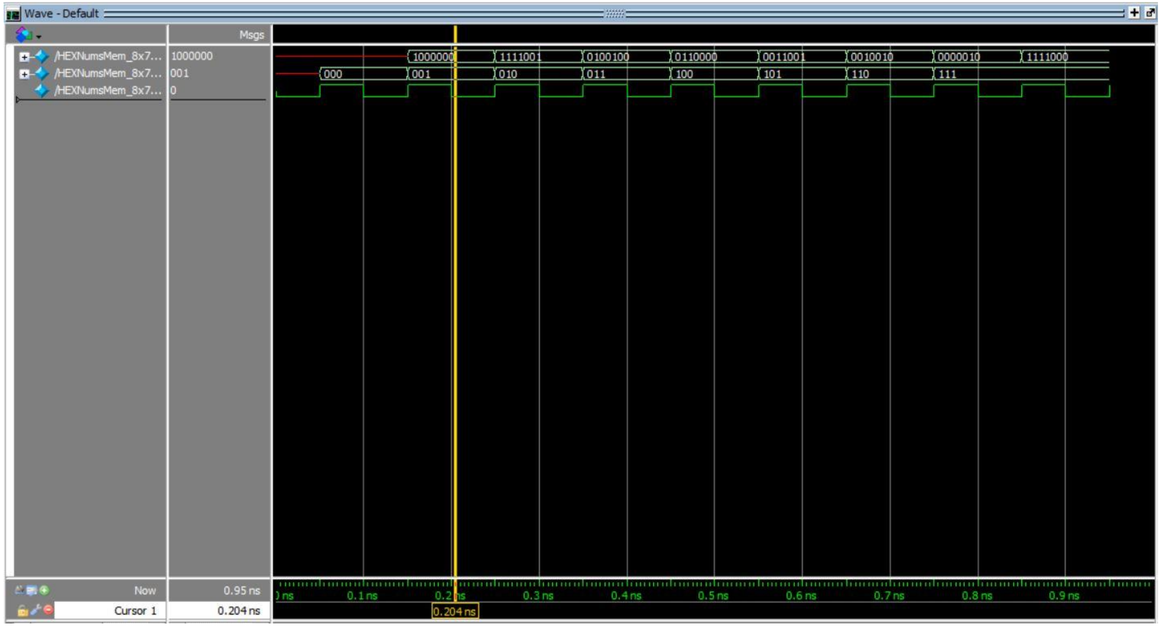


*Figure 14 – HEXNumsMem_8x7_testbench Simulation*

The winnerCounter code worked as expected. The simulation is shown in Figure 15.

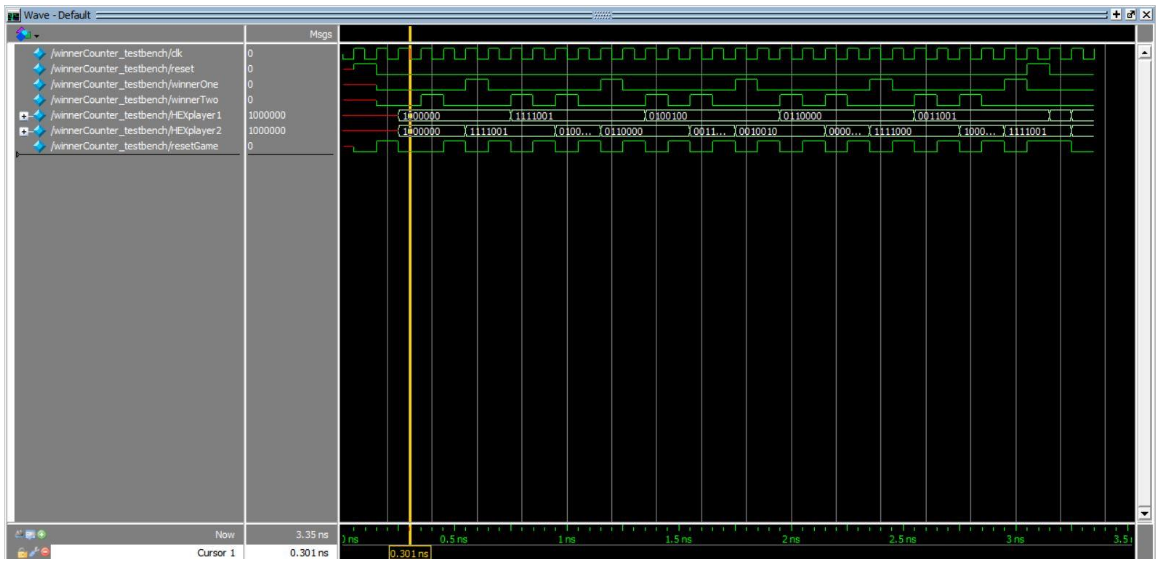

Wave - Default
Msgs
/winnerCounter_testbench/clk 0
/winnerCounter_testbench/reset 0
/winnerCounter_testbench/winnerOne 0
/winnerCounter_testbench/winnerTwo 0
/winnerCounter_testbench/HEXplayer1 1000000
/winnerCounter_testbench/HEXplayer2 1000000
/winnerCounter_testbench/resetGame 0
Now 3.35 ns
Cursor 1 0.301 ns
0.301 ns

Figure 15 - winnerCounter_testbench Simulation

After performing the simulations, the design was downloaded to the FPGA and tested on the development board by playing the Pong game. Bugs were checked by pressing down both keys, letting player one win, letting player two win, resetting in the middle of a game, resetting after a game, leaving reset to TRUE and trying to play, holding down a key, hitting the ball on the edge, hitting the ball flat, changing ball speed during the game, and finally holding a key while pressing the other key. All scenarios performed as expected. Figure 16 shows a freeze-frame picture of the game being played on the 8x8 LED matrix.

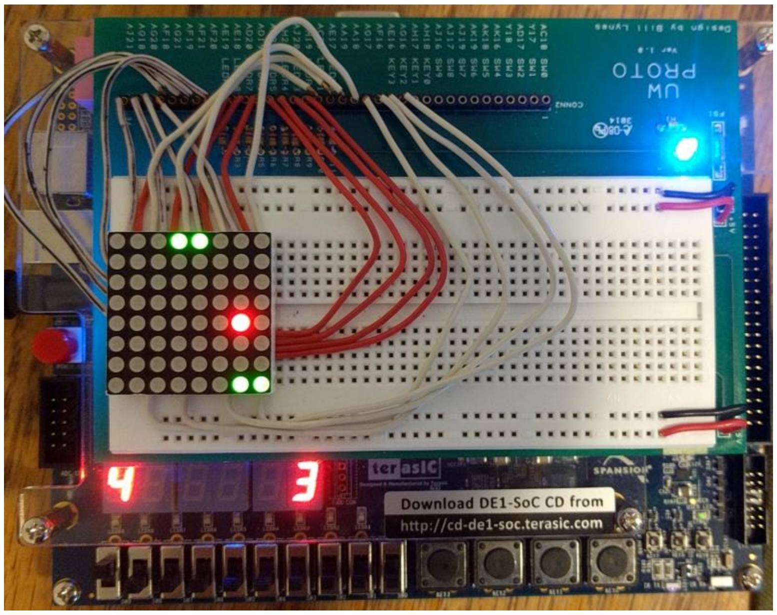
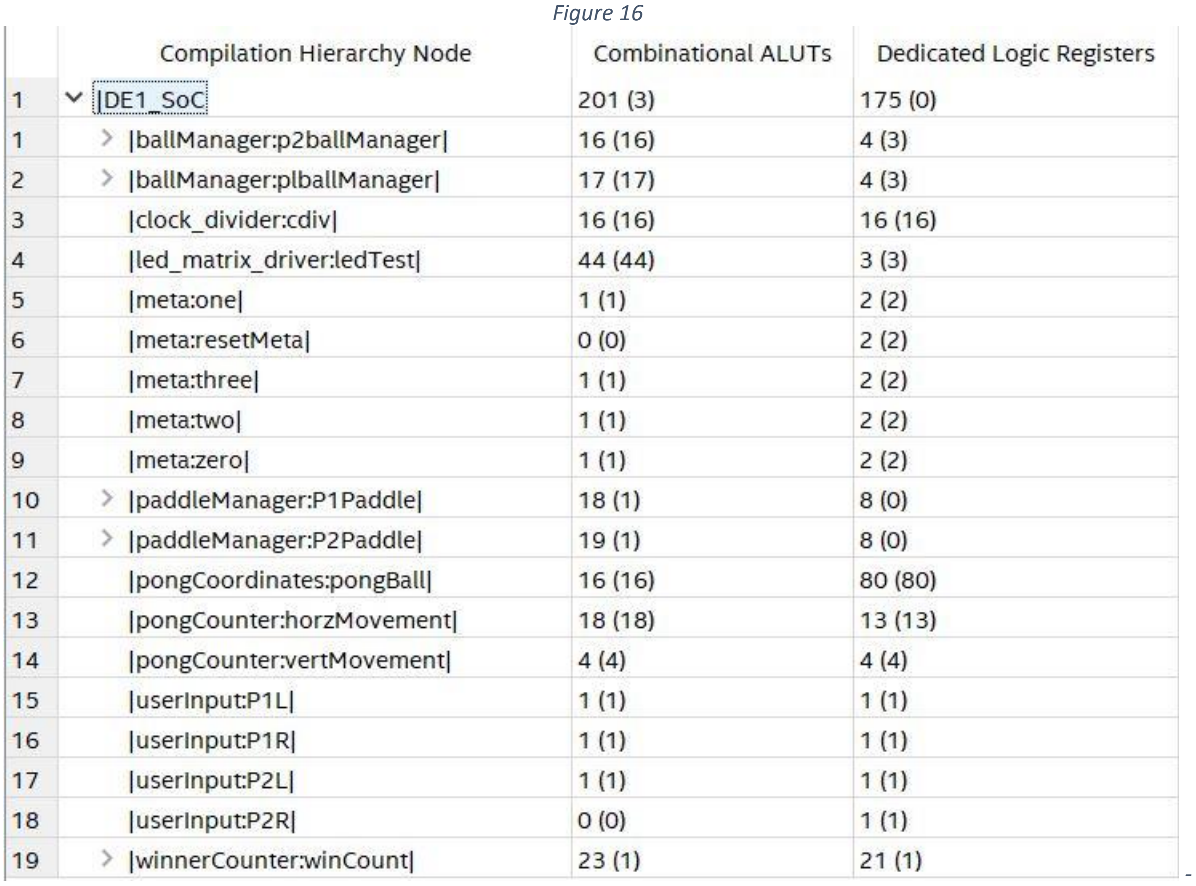


Figure 16 - Picture of DE1-SoC board with 8x8 Matrix playing Pong

One of the requirements for this design was to create as small as a circuit as possible in terms of number of FPGA logic and DFF resources. The size was computed by adding the “LC Combinationals” (number of

FPGA logic elements) and "LC Registers" (number of DFFs) and subtracting the cost of the clock_divider. The resource utilization is shown in Figure 10 below. The total size of the design is 344 based on the formula described above.

*Figure 16*



| | Compilation Hierarchy Node | Combinational ALUTs | Dedicated Logic Registers |
|---|---|---|---|
| 1 | \|DE1_SoC | 201 (3) | 175 (0) |
| 1 | \|ballManager:p2ballManager\| | 16 (16) | 4 (3) |
| 2 | \|ballManager:plballManager\| | 17 (17) | 4 (3) |
| 3 | \|clock_divider:cdiv\| | 16 (16) | 16 (16) |
| 4 | \|led_matrix_driver:ledTest\| | 44 (44) | 3 (3) |
| 5 | \|meta:one\| | 1 (1) | 2 (2) |
| 6 | \|meta:resetMeta\| | 0 (0) | 2 (2) |
| 7 | \|meta:three\| | 1 (1) | 2 (2) |
| 8 | \|meta:two\| | 1 (1) | 2 (2) |
| 9 | \|meta:zero\| | 1 (1) | 2 (2) |
| 10 | \|paddleManager:P1Paddle\| | 18 (1) | 8 (0) |
| 11 | \|paddleManager:P2Paddle\| | 19 (1) | 8 (0) |
| 12 | \|pongCoordinates:pongBall\| | 16 (16) | 80 (80) |
| 13 | \|pongCounter:horzMovement\| | 18 (18) | 13 (13) |
| 14 | \|pongCounter:vertMovement\| | 4 (4) | 4 (4) |
| 15 | \|userInput:P1L\| | 1 (1) | 1 (1) |
| 16 | \|userInput:P1R\| | 1 (1) | 1 (1) |
| 17 | \|userInput:P2L\| | 1 (1) | 1 (1) |
| 18 | \|userInput:P2R\| | 0 (0) | 1 (1) |
| 19 | \|winnerCounter:winCount\| | 23 (1) | 21 (1) |

*Resource Utilization*

**Conclusion:**
This lab helped improve development and simulation skills of complex sequential logic designs requiring multiple FSMs to communicate. The requirements and process weren't provided as was the case for previous labs requiring more planning than usual. This was done by designing a Pong game onto the DE1_SoC development board. A block diagram was created to represent the necessary module, and then state diagrams and tables were created for each block representing an FSM. Simulation and implementation of the software solution was performed to help improve SystemVerilog skills for sequential logic and to test the final design. The final design was then tested for bugs on the DEC_SoC board and LED matrix.

This lab was much more in depth than previous lab, but it utilized all material that was covered over the course of the quarter in order to implement a usable product with multiple features and external hardware.

**Appendix**
<u>meta Code:</u>

```
        module meta(clk, d1, q1, q2);
                input logic clk, d1;
                output logic q1, q2;

                always_ff @ (posedge clk) begin
                        q1 <= d1;
                        q2 <= q1;
                end

        endmodule

        module meta_testbench();
                logic clk, d1;
                logic q1, q2;

                meta dut (clk, d1, q1, q2);

                // Set up the clock.
                parameter CLOCK_PERIOD = 100;
                initial begin
                        clk <= 0;
                        forever #(CLOCK_PERIOD/2) clk <= ~clk;
                end

                // Set up the inputs to the design.  Each line is a clock cycle
                initial begin

                @(posedge clk);
                                                                d1 <= 0;        @(posedge
        clk);

                @(posedge clk);

                @(posedge clk);
                                                                d1 <= 1;        @(posedge
        clk);

                @(posedge clk);

                @(posedge clk);
                        $stop; // End the simulation
                end
endmodule
```

<u>userInput Code:</u>

```
module userInput(clk, keyIn, keyOut);
	input logic clk, keyIn;
	output logic keyOut;

	// State variables.
	enum {keyOff, keyOn} ps, ns;

	// Next State logic
	always_comb begin
		case(ps)
			keyOff:  if(keyIn)	ns = keyOn;
					else			ns = ps;
			keyOn:  if(keyIn)	ns = ps;
					else			ns = keyOff;
		endcase
	end

	// Output logic
	assign keyOut = (ps == keyOff & ns == keyOn); //Only True once per key press

	// DFFs
	always_ff @ (posedge clk) begin
			ps <= ns;
	end

endmodule

module userInput_testBench();
	logic clk, keyIn;
	logic keyOut;

	userInput dut (clk, keyIn, keyOut);

	// Set up the clock.
	parameter CLOCK_PERIOD = 100;
	initial begin
		clk <= 0;
		forever #(CLOCK_PERIOD/2) clk <= ~clk;
	end

	// Set up the inputs to the design.  Each line is a clock cycle
	initial begin
								@(posedge clk);
		keyIn <= 0;			@(posedge clk);
								@(posedge clk);
								@(posedge clk);
		keyIn <= 1;			@(posedge clk);
								@(posedge clk);
```

```
                                                                @(posedge clk);
                                                                @(posedge clk);
                keyIn <= 0;                     @(posedge clk);
                                                                @(posedge clk);
                                                                @(posedge clk);
                keyIn <= 1;                     @(posedge clk);
                                                                @(posedge clk);
                                                                @(posedge clk);

                $stop; // End the simulation
        end
endmodule
```

paddleManager Code:

```
module paddleManager(clk, reset, left, right, display);
        input logic clk, reset, left, right;
        output logic [7:0] display;

        logic enableLL, enableLML, enableRML, enableRL, enableLR, enableLMR,
enableRMR, enableRR;

        //Call paddle component, seperate module used for reset
        paddleComp          LeftP                   (.clk(clk), .reset(reset),
.left(left), .right(right), .display(display[7:6]), .enableL(enableRR),
.enableR(enableLML), .enabled(enableLL | enableLR));
        paddleComp          LeftMidP                (.clk(clk), .reset(reset),
.left(left), .right(right), .display(display[5:4]), .enableL(enableLR),
.enableR(enableRML), .enabled(enableLML | enableLMR));
        paddleComp          RightMidP       (.clk(clk), .reset(reset), .left(left),
.right(right), .display(display[3:2]), .enableL(enableLMR), .enableR(enableRL),
.enabled(enableRML | enableRMR));
        paddleCompRes       RightP          (.clk(clk), .reset(reset), .left(left),
.right(right), .display(display[1:0]), .enableL(enableRMR), .enableR(enableLL),
.enabled(enableRL | enableRR));

endmodule


module paddleManager_testbench();
        logic clk, reset, left, right;
        logic [7:0] display;

        paddleManager dut (clk, reset, left, right, display);

        // Set up the clock.
        parameter CLOCK_PERIOD = 100;
        initial begin
```

```
			clk <= 0;
			forever #(CLOCK_PERIOD/2) clk <= ~clk;
		end

		// Set up the inputs to the design.  Each line is a clock cycle
			initial begin

						@(posedge clk);
			reset <= 1;
				@(posedge clk);
			reset <= 0;	left <= 0;	right <= 1;		@(posedge
	clk);

						@(posedge clk);

						@(posedge clk);

						@(posedge clk);

						@(posedge clk);

						@(posedge clk);

						@(posedge clk);

						@(posedge clk);

						@(posedge clk);
							left <= 1;	right <= 0;
		@(posedge clk);

						@(posedge clk);

						@(posedge clk);

						@(posedge clk);
							left <= 0;	right <= 0;
		@(posedge clk);
							left <= 1;	right <= 0;
		@(posedge clk);

						@(posedge clk);
							left <= 0;	right <= 0;
		@(posedge clk);

						@(posedge clk);
			reset <= 1;	left <= 1;	right <= 0;		@(posedge
	clk);
```

```
				reset <= 0;	left <= 0;	right <= 1;		@(posedge
		clk);
								left <= 1;	right <= 0;
			@(posedge clk);
				$stop; // End the simulation
			end
endmodule
```

<u>paddleComp Code:</u>

```
		module paddleComp(clk, reset, left, right, display, enableL, enableR, enabled);
			input logic			clk, reset, left, right, enabled;
			output logic	enableL, enableR;
			output logic	[1:0] display;

			// State variables.
			enum {on, off, L, R} ps, ns;

			// Next State logic
			always_comb begin
				case(ps)
						on:			if(left == 1)
							ns = L;
									else if(right == 1)
							ns = R;
									else
										ns = ps;

						off:			if(left == 1 & enabled == 1)
				ns = R;
									else if(right == 1 & enabled
		== 1)	ns = L;
									else
										ns = ps;

						R:				if(left == 1)
								ns = on;
									else if(right == 1)
							ns = off;
									else
										ns = ps;

						L:				if(left == 1)
								ns = off;
									else if(right == 1)
							ns = on;
									else
										ns = ps;
				endcase
		end

		// Output logic
		assign display[1] = (ps == on | ps == L);
		assign display[0] = (ps == on | ps == R);
		assign enableL = (ps == on & ns == L);
		assign enableR = (ps == on & ns == R);

		// DFFs
		always_ff @ (posedge clk) begin
			if (reset)
				ps <= off;
			else
				ps <= ns;
		end


endmodule

module paddleComp_testbench();
		logic clk, reset, left, right, enabled;
		logic enableL, enableR;
		logic [1:0] display;

		paddleComp dut (clk, reset, left, right, display, enableL, enableR, enabled);

		// Set up the clock.
		parameter CLOCK_PERIOD = 100;
		initial begin
			clk <= 0;
			forever #(CLOCK_PERIOD/2) clk <= ~clk;
		end

		// Set up the inputs to the design.  Each line is a clock cycle
			initial begin


		@(posedge clk);
			reset <= 1;
											@(posedge
clk);
			reset <= 0;			enabled <= 0;	left <= 0;		right <= 1;
			@(posedge clk);


		@(posedge clk);
```

```
			@(posedge clk);

				left <= 0;		right <= 0;			@(posedge clk);

				left <= 1;		right <= 0;			@(posedge clk);


			@(posedge clk);


			@(posedge clk);

				left <= 0;		right <= 0;			@(posedge clk);
											enabled <= 1;	left <= 0;
			right <= 1;			@(posedge clk);
											enabled <= 0;
												@(posedge clk);


			@(posedge clk);


			@(posedge clk);

				left <= 1;		right <= 0;			@(posedge clk);


			@(posedge clk);
											enabled <= 1;
												@(posedge clk);
											enabled <= 0;
									@(posedge clk);


			@(posedge clk);


			@(posedge clk);
				$stop; // End the simulation
			end
endmodule
```

paddleCompReset Code:

```
		module paddleCompRes(clk, reset, left, right, display, enableL, enableR, enabled);
			input logic			clk, reset, left, right, enabled;
```

```
        output logic    enableL, enableR;
        output logic    [1:0] display;

        // State variables.
        enum {on, off, L, R} ps, ns;

        // Next State logic
        always_comb begin
                case(ps)
                                on:             if(left == 1)
                                        ns = L;
                                                        else if(right == 1)
                                        ns = R;
                                                        else
                                                                ns = ps;

                                off:            if(left == 1 & enabled == 1)
                ns = R;
                                                        else if(right == 1 & enabled
== 1)   ns = L;
                                                        else
                                                                ns = ps;

                                R:                      if(left == 1)
                                                ns = on;
                                                        else if(right == 1)
                                        ns = off;
                                                        else
                                                                ns = ps;

                                L:                      if(left == 1)
                                                ns = off;
                                                        else if(right == 1)
                                        ns = on;
                                                        else
                                                                ns = ps;
                endcase
        end

        // Output logic
        assign display[1] = (ps == on | ps == L);
        assign display[0] = (ps == on | ps == R);
        assign enableL = (ps == on & ns == L);
        assign enableR = (ps == on & ns == R);

        // DFFs
        always_ff @ (posedge clk) begin
                if (reset)
```

```
                    ps <= on;
                else
                    ps <= ns;
        end


endmodule

module paddleCompRes_testbench();
        logic clk, reset, left, right, enabled;
        logic enableL, enableR;
        logic [1:0] display;

        paddleCompRes dut (clk, reset, left, right, display, enableL, enableR, enabled);

        // Set up the clock.
        parameter CLOCK_PERIOD = 100;
        initial begin
                clk <= 0;
                forever #(CLOCK_PERIOD/2) clk <= ~clk;
        end

        // Set up the inputs to the design.  Each line is a clock cycle
                initial begin


        @(posedge clk);
                reset <= 1;
                                                                        @(posedge
clk);
                reset <= 0;             enabled <= 0;   left <= 0;      right <= 1;
                @(posedge clk);


        @(posedge clk);


        @(posedge clk);

                left <= 0;      right <= 0;             @(posedge clk);

                left <= 1;      right <= 0;             @(posedge clk);


        @(posedge clk);


        @(posedge clk);
```

```
                left <= 0;          right <= 0;                 @(posedge clk);
                                                                enabled <= 1;    left <= 0;
        right <= 1;                 @(posedge clk);
                                                                enabled <= 0;
                                                                        @(posedge clk);


        @(posedge clk);


        @(posedge clk);

                left <= 1;          right <= 0;                 @(posedge clk);


        @(posedge clk);
                                                                enabled <= 1;
                                                                        @(posedge clk);
                                                                enabled <= 0;
                                                    @(posedge clk);


        @(posedge clk);


        @(posedge clk);
                $stop; // End the simulation
        end
endmodule
```

pongCounter Code:

```
    module pongCounter (clk, reset, ballSpeed, direction, changeDirection, count);

        input logic clk, reset, changeDirection;
        input logic [3:0] ballSpeed;
        output logic [2:0] count;
        output logic direction;

        logic [8:0] ballCounter;
        logic ballMove;

        logic [8:0] ballSpeed2;

        //Make first 4 bits adjustable by switches for adjusting ball spped
        assign ballSpeed2 = {ballSpeed[3],ballSpeed[2],ballSpeed[1],ballSpeed[0],
    1'b1, 1'b1, 1'b1, 1'b1, 1'b1};
```

```
	//Counter for ballMove
	always_ff @ (posedge clk) begin
		if (reset) begin
			ballCounter <= '0;
			ballMove <= 0;
		end
		//ballSpeed2 is adjustable, once ballCounter reaches ballSpeed2, set
ballMove to 1
		else if (ballCounter == ballSpeed2) begin
			ballCounter <= '0;
			ballMove <= 1;
		end
		//Else, increase ballCounter and keep ballMove at 0
		else begin
			ballCounter <= ballCounter + 1;
			ballMove <= 0;
		end
	end


	// State variables.
	enum {F, R} ps, ns;

	// Next State logic
	always_comb begin
		case(ps)
			//Change between forward and reverse states if change
direction is flagged, or it hits a wall
			F: if(changeDirection)					ns = R;
				else if (count == 3'b111)		ns = R;
				else
			ns = ps;
			R: if(changeDirection)					ns = F;
				else if (count == 3'b000)		ns = F;
				else
			ns = ps;
		endcase
	end

	// Output logic
	assign direction = (ns == F);

	// DFFs
	always_ff @ (posedge clk) begin
		if (reset)
			ps <= R;
		else
```

```
				ps <= ns;
		end

		//Counter
		always_ff @ (posedge clk) begin
			if(reset)
				count <= 3'b100;
			else if (direction)
				//Only increase count if ball move is true
				case(ballMove)
					1: count <= count + 1;
					0: count <= count;
				endcase
			else
				//If direction is false (i.e. reverse) decrease counter when ball
	move is true
				case(ballMove)
					1: count <= count - 1;
					0: count <= count;
				endcase
		end

	endmodule

	module pongCounter_testbench();

		logic clk, reset, changeDirection, direction;
		logic [3:0] ballSpeed;
		logic [2:0] count;

		pongCounter dut (.clk, .reset, .ballSpeed, .direction, .changeDirection,
	.count);

		// Set up the clock.
		parameter CLOCK_PERIOD = 100;
		initial begin
			clk <= 0;
			forever #(CLOCK_PERIOD/2) clk <= ~clk;
		end

		initial begin

						@(posedge clk);
			reset <= 1;
				@(posedge clk);
			reset <= 0;	ballSpeed = '1;				@(posedge
	clk);
```

```
                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);
                                                ballSpeed = '0;
                @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);
                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);

                                        @(posedge clk);
                                                changeDirection <= 1;
            @(posedge clk);
                                                changeDirection <= 0;
            @(posedge clk);
```

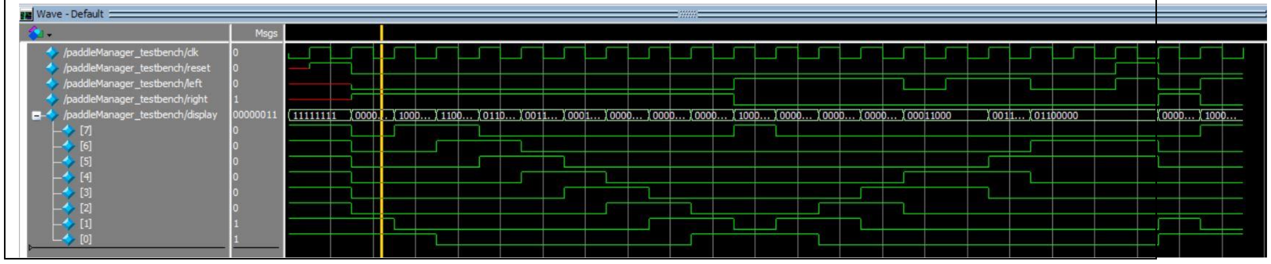
```
                                                @(posedge clk);

                                                @(posedge clk);

                                                @(posedge clk);
                                                        changeDirection <= 1;
                @(posedge clk);
                                                        changeDirection <= 0;
                @(posedge clk);

                                                @(posedge clk);

                                                @(posedge clk);
                    $stop; // End the simulation
                end

endmodule
```

<u>pongCoordinates Code:</u>

```
        module pongCoordinates (clk, horzCount, vertCount, pongCoord);

                input logic clk;
                input logic [2:0] horzCount, vertCount;
                output logic [7:0][7:0] pongCoord;

                logic [7:0] rows, cols;
                logic row, col;

                //Counters for vertical and horizontal are fed into decoder
                always @(posedge clk)
                        case (vertCount)
                                3'b000: rows = 8'b00000001;
                                3'b001: rows = 8'b00000010;
                                3'b010: rows = 8'b00000100;
                                3'b011: rows = 8'b00001000;
                                3'b100: rows = 8'b00010000;
                                3'b101: rows = 8'b00100000;
                                3'b110: rows = 8'b01000000;
                                3'b111: rows = 8'b10000000;
                        endcase

                always @(posedge clk)
                        case (horzCount)
                                3'b000: cols = 8'b00000001;
                                3'b001: cols = 8'b00000010;
                                3'b010: cols = 8'b00000100;
```

```
				3'b011: cols = 8'b00001000;
				3'b100: cols = 8'b00010000;
				3'b101: cols = 8'b00100000;
				3'b110: cols = 8'b01000000;
				3'b111: cols = 8'b10000000;
			endcase

		//Traverse through arrays of rows and columns to set entire matrix to 0,
except where ball is at
		always @(posedge clk)
			for(int r=7; r>=0; r--) begin
				if (rows[r]) begin
					for (int c = 7; c>=0; c--) begin
						if (cols[c])
							pongCoord[r][c] = 1;
						else
							pongCoord[r][c] = 0;
					end //column for loop
				end // row if statemnt
				else
					pongCoord[r] = '0;
			end //row for loop

endmodule

module pongCoordinates_testbench();

		logic clk;
		logic [2:0] horzCount, vertCount;
		logic [7:0][7:0] pongCoord;

		pongCoordinates dut (.clk, .horzCount, .vertCount, .pongCoord);

		// Set up the clock.
		parameter CLOCK_PERIOD = 100;
		initial begin
			clk <= 0;
			forever #(CLOCK_PERIOD/2) clk <= ~clk;
		end


		// Set up the inputs to the design.  Each line is a clock cycle
		initial begin
			horzCount <= '0;		vertCount <= '1;
						@(posedge clk);
			for (int i=0; i<=15; i++) begin
				horzCount <= horzCount+1;	vertCount <= vertCount+1;
			@(posedge clk);
```

```
                    end


                    $stop; // End the simulation
            end
        endmodule
```



<u>ballManager Code:</u>

```
        module ballManager (clk, reset, pongCoordEval, winEval, paddleArray, dirHorz,
        dirVert, changeDirHorz, changeDirVert, winner);

            input logic clk, reset, dirHorz, dirVert;
            input logic [7:0] paddleArray, pongCoordEval, winEval;
            output logic changeDirHorz, changeDirVert, winner;

            logic ball, paddleFlat, paddleEdge, winner2;

            //If the ball is traveling towards the paddle and is in the Evaluation Zone (row
        next to paddle row) set to 1, else set to 0
            assign ball = dirVert & (pongCoordEval[7] | pongCoordEval[6] |
        pongCoordEval[5] | pongCoordEval[4] | pongCoordEval[3] | pongCoordEval[2] |
        pongCoordEval[1] | pongCoordEval[0]);


            always @ (posedge clk) begin
                    //If paddle is in same column as ball, set paddleFlat to 1, else 0
                    for(int i = 7; i>=0; i--) begin
                            if (paddleArray[i] & pongCoordEval[i]) begin
                                    paddleFlat = 1;
                                    break;
                            end
                            paddleFlat = 0;
                    end

                    //If paddleFlat is false, check to see if ball is traveling towards edge of
        paddle.
                    //One column away from paddle, travelling horizontally towards
        paddle
                    //If so, set paddleEdge to 1, else to 0
```

```
			if (~paddleFlat) begin
				for (int i = 6; i>=1; i--) begin
					if ((dirHorz & paddleArray[i+1] & pongCoordEval[i]) |
(~dirHorz & paddleArray[i-1] & pongCoordEval[i])) begin
						paddleEdge = 1;
						break;
					end
					paddleEdge = 0;
				end
			end
		end

		// State variables.
		enum {off, evaluate} ps, ns;

		// Next State logic
		always_comb begin
			case(ps)
				// ball is true, set state to evaluate
				off:				if(ball)	ns = evaluate;
									else		ns = ps;
				// Only in evaluate for one clock cylce, then back to "off"
state
				evaluate:		ns = off;
			endcase
		end

		// Output logic
		//If ball is even in paddle zone, set winner2 to true, else false
		assign winner2 = (winEval[7] | winEval[6] | winEval[5] | winEval[4] |
winEval[3] | winEval[2] | winEval[1] | winEval[0]);
		userInput winFilter(.clk(clk), .keyIn(winner2), .keyOut(winner)); //put through
userInput module so that it only increments by one per clock cycle

		//If ball hits edge, change horizontal direction
		assign changeDirHorz = ((ps == evaluate) & paddleEdge);

		//If ball hits paddle, change vertical direction
		assign changeDirVert = ((ps == evaluate) & (paddleFlat | paddleEdge));

		// DFFs
		always_ff @ (posedge clk) begin
			if (reset)
				ps <= off;
			else
				ps <= ns;
		end
```

```
endmodule

module ballManager_testbench();

        logic clk, reset, dirHorz, dirVert;
        logic [7:0] paddleArray, pongCoordEval, winEval;
        logic changeDirHorz, changeDirVert, winner;

        ballManager dut (.clk, .reset, .pongCoordEval, .winEval, .paddleArray,
.dirHorz, .dirVert, .changeDirHorz, .changeDirVert, .winner);

        // Set up the clock.
        parameter CLOCK_PERIOD = 100;
        initial begin
                clk <= 0;
                forever #(CLOCK_PERIOD/2) clk <= ~clk;
        end

        initial begin




                        @(posedge clk);





                        @(posedge clk);
                reset <= 1;



        @(posedge clk);





                        @(posedge clk);

        dirVert <= 0;


        @(posedge clk);
                reset <= 0;      dirHorz <= 1;     dirVert <= 1;      paddleArray =
8'b10000001;    pongCoordEval = 8'b10000000;  winEval = '0;
        @(posedge clk);
```

```
                        @(posedge clk);

dirVert <= 0;


@(posedge clk);




                        @(posedge clk);

dirVert <= 1;
            pongCoordEval = 8'b01000000;
                                    @(posedge clk);




                        @(posedge clk);

dirVert <= 0;


@(posedge clk);




                        @(posedge clk);

dirVert <= 1;
            pongCoordEval = 8'b00100000;
                                    @(posedge clk);




                        @(posedge clk);

dirVert <= 0;
```

```
@(posedge clk);





        @(posedge clk);

dirVert <= 1;
    pongCoordEval = 8'b00010000;
            @(posedge clk);









        @(posedge clk);

dirVert <= 0;


@(posedge clk);





        @(posedge clk);

dirVert <= 1;
    pongCoordEval = 8'b00001000;
            @(posedge clk);




        @(posedge clk);

dirVert <= 0;


@(posedge clk);
```

```
                    @(posedge clk);

dirVert <= 1;
          pongCoordEval = 8'b00000100;
                              @(posedge clk);



                    @(posedge clk);

dirVert <= 0;


@(posedge clk);



                    @(posedge clk);

dirVert <= 1;
          pongCoordEval = 8'b00000010;
                              @(posedge clk);



                    @(posedge clk);

dirVert <= 0;


@(posedge clk);
```

```
                    @(posedge clk);

          dirVert <= 1;
               pongCoordEval = 8'b00000001;
                         @(posedge clk);




                    @(posedge clk);

          dirVert <= 0;


          @(posedge clk);





                    @(posedge clk);

          dirVert <= 1;
               pongCoordEval = '0;                    winEval =
     8'b10000000;    @(posedge clk);




                    @(posedge clk);

          dirVert <= 0;


          @(posedge clk);





                    @(posedge clk);
                                   dirHorz <= 0;    dirVert <= 1;

          pongCoordEval = 8'b10000000;
                    @(posedge clk);
```

```
                        @(posedge clk);

            dirVert <= 0;


            @(posedge clk);




                        @(posedge clk);

            dirVert <= 1;
                  pongCoordEval = 8'b01000000;
                              @(posedge clk);




                        @(posedge clk);

            dirVert <= 0;


            @(posedge clk);





                        @(posedge clk);

            dirVert <= 1;
                  pongCoordEval = 8'b00100000;
                              @(posedge clk);




                        @(posedge clk);
```

```
dirVert <= 0;


@(posedge clk);








                @(posedge clk);

dirVert <= 1;
        pongCoordEval = 8'b00010000;
                        @(posedge clk);




                @(posedge clk);

dirVert <= 0;


@(posedge clk);






                @(posedge clk);

dirVert <= 1;
        pongCoordEval = 8'b00001000;
                        @(posedge clk);




                @(posedge clk);

dirVert <= 0;
```

```
@(posedge clk);




                    @(posedge clk);

dirVert <= 1;
          pongCoordEval = 8'b00000100;
                              @(posedge clk);



                    @(posedge clk);

dirVert <= 0;


@(posedge clk);







                    @(posedge clk);

dirVert <= 1;
          pongCoordEval = 8'b00000010;
                              @(posedge clk);



                    @(posedge clk);

dirVert <= 0;


@(posedge clk);
```

```
                        @(posedge clk);

            dirVert <= 1;
                  pongCoordEval = 8'b00000001;
                              @(posedge clk);



                        @(posedge clk);

            dirVert <= 0;


            @(posedge clk);



                        @(posedge clk);

            dirVert <= 1;
                  pongCoordEval = '0;                         winEval =
      8'b00000001;    @(posedge clk);



                        @(posedge clk);

            dirVert <= 0;


            @(posedge clk);




                        @(posedge clk);
```

```
dirVert <= 0;
        pongCoordEval = 8'b10000000;
                        @(posedge clk);



                @(posedge clk);


                        pongCoordEval = 8'b01000000;
                                        @(posedge clk);



                @(posedge clk);


                        pongCoordEval = 8'b00100000;
                                        @(posedge clk);






                @(posedge clk);


                        pongCoordEval = 8'b00010000;
                                        @(posedge clk);



                @(posedge clk);


                        pongCoordEval = 8'b00001000;
                                        @(posedge clk);
                @(posedge clk);

                        pongCoordEval = 8'b00000100;
                                @(posedge clk);

                @(posedge clk);

                        pongCoordEval = 8'b00000010;
                                @(posedge clk);

                @(posedge clk);

                        pongCoordEval = 8'b00000001;
                                @(posedge clk);

                @(posedge clk);

                        pongCoordEval = '0;
        winEval = 8'b00000001; @(posedge clk);

                @(posedge clk);
                $stop; // End the simulation
        end

    endmodule
```

Led_matrix_drive Code:

```
    module led_matrix_driver (clock, red_array, green_array, red_driver, green_driver,
    row_sink);
        input clock;
        input [7:0][7:0] red_array, green_array;
        output reg [7:0] red_driver, green_driver, row_sink;
        reg [2:0] count;

        always @(posedge clock)
            count <= count + 3'b001; always @(*)
            case (count)
                3'b000: row_sink = 8'b11111110;
                3'b001: row_sink = 8'b11111101;
                3'b010: row_sink = 8'b11111011;
                3'b011: row_sink = 8'b11110111;
                3'b100: row_sink = 8'b11101111;
                3'b101: row_sink = 8'b11011111;
                3'b110: row_sink = 8'b10111111;
                3'b111: row_sink = 8'b01111111;
            endcase

            assign red_driver = red_array[count];
            assign green_driver = green_array[count];

endmodule
```

upCounter Code:

```
    module upCounter #(parameter WIDTH = 3) (out, incr, reset, clk);
        output logic    [WIDTH-1:0] out; //Width parameter to control upperlimit of
    //counter (i.e WIDTH = 3, [2:0] out, 3-bit counter)
        input logic     incr, reset, clk;

        always_ff @ (posedge clk) begin
            if(reset)
                out <= '0;
            else if (incr)
                out <= out + 1;
        end

    endmodule
```

```
	module upCounter_testbench ();
		logic	[2:0] out;
		logic	incr, reset, clk;

		upCounter dut (.out, .incr, .reset, .clk);

		// Set up the clock.
		parameter CLOCK_PERIOD = 100;
		initial begin
			clk <= 0;
			forever #(CLOCK_PERIOD/2) clk <= ~clk;
		end

		initial begin
											@(posedge clk);
			reset <= 1;							@(posedge clk);
			reset <= 0;			incr <= 1;			@(posedge clk);
											@(posedge clk);
											@(posedge clk);
											@(posedge clk);
											@(posedge clk);
											@(posedge clk);
											@(posedge clk);
											@(posedge clk);
											@(posedge clk);
						incr <= 0;			@(posedge clk);
			reset <= 1;							@(posedge clk);
											@(posedge clk);
			reset <= 0;							@(posedge clk);
											@(posedge clk);
			$stop; // End the simulation
		end

endmodule
```

<u>HEXNumsMem_8x7 Code:</u>

```
	module HEXNumsMem_8x7 (data_out, addr, clk);

		output logic [6:0] data_out;
		input logic [2:0] addr;
		input logic clk;

		logic [6:0] mem [7:0];

		//Make 7-Segment patterns to represent numbers 0-7
		assign mem[0][6:0] = 7'b1000000; //0
		assign mem[1][6:0] = 7'b1111001; //1
		assign mem[2][6:0] = 7'b0100100; //2
		assign mem[3][6:0] = 7'b0110000; //3
		assign mem[4][6:0] = 7'b0011001; //4
		assign mem[5][6:0] = 7'b0010010; //5
		assign mem[6][6:0] = 7'b0000010; //6
		assign mem[7][6:0] = 7'b1111000; //7

		always_ff @ (posedge clk) begin
			data_out <= mem[addr];
		end

	endmodule

	module HEXNumsMem_8x7_testbench ();
		logic [6:0] data_out;
		logic [2:0] addr;
		logic clk;

		HEXNumsMem_8x7 dut (.data_out, .addr, .clk);

		// Set up the clock.
		parameter CLOCK_PERIOD = 100;
		initial begin
			clk <= 0;
			forever #(CLOCK_PERIOD/2) clk <= ~clk;
		end

		initial begin
							@(posedge clk);
			addr <= 3'b000;			@(posedge clk);
			addr <= 3'b001;			@(posedge clk);
			addr <= 3'b010;			@(posedge clk);
			addr <= 3'b011;			@(posedge clk);
			addr <= 3'b100;			@(posedge clk);
			addr <= 3'b101;			@(posedge clk);
			addr <= 3'b110;			@(posedge clk);
			addr <= 3'b111;			@(posedge clk);
							@(posedge clk);
			$stop; // End the simulation
		end

endmodule
```

winnerCounter Code:

```
module winnerCounter (clk, reset, winnerOne, winnerTwo, HEXplayer1, HEXplayer2,
resetGame);
	input logic clk, reset;
	input logic winnerOne, winnerTwo;
	output logic [6:0] HEXplayer1, HEXplayer2;
	output logic resetGame;

	logic [2:0] player1Score, player2Score;

	upCounter player1 (.out(player1Score), .incr(winnerOne), .reset(reset),
.clk(clk));
	upCounter player2 (.out(player2Score), .incr(winnerTwo), .reset(reset),
.clk(clk));
	HEXNumsMem_8x7 player1HEX (.data_out(HEXplayer1), .addr(player1Score),
.clk(clk));
	HEXNumsMem_8x7 player2HEX (.data_out(HEXplayer2), .addr(player2Score),
.clk(clk));

	always_ff @ (posedge clk) begin
		if (reset) begin //Main Reset
			resetGame <= 1;
		end else if (winnerOne | winnerTwo) begin //Winner, reset playfield
			resetGame <= 1;
		end else begin
			resetGame <= 0;
		end
	end

endmodule

module winnerCounter_testbench();
	logic clk, reset;
	logic winnerOne, winnerTwo;
	logic [6:0] HEXplayer1, HEXplayer2;
	logic resetGame;

	winnerCounter dut (clk, reset, winnerOne, winnerTwo, HEXplayer1,
HEXplayer2, resetGame);

	// Set up the clock.
	parameter CLOCK_PERIOD = 100;
	initial begin
		clk <= 0;
		forever #(CLOCK_PERIOD/2) clk <= ~clk;
	end

	// Set up the inputs to the design.  Each line is a clock cycle
		initial begin
```

```
                                                                                @(posedge clk);
                    reset <= 1;                                                 @(posedge clk);
                    reset <= 0; winnerOne <= 0; winnerTwo <= 0;                 @(posedge clk);
                                                                                @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                            winnerOne <= 1;                                     @(posedge clk);
                            winnerOne <= 0;                                     @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                             winnerOne <= 1;                                    @(posedge clk);
                             winnerOne <= 0;                                    @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                             winnerOne <= 1;                                    @(posedge clk);
                             winnerOne <= 0;                                    @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                             winnerOne <= 1;                                    @(posedge clk);
                             winnerOne <= 0;                                    @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                                                       winnerTwo <= 1;          @(posedge clk);
                                                       winnerTwo <= 0;          @(posedge clk);
                             winnerOne <= 1;                                    @(posedge clk);
                    reset <= 1;  winnerOne <= 0;                                @(posedge clk);
                    reset <= 0;                                                 @(posedge clk);
                                                                                @(posedge clk);
                    $stop; // End the simulation
            end
endmodule
```

DEC_SoC1 Code: