CS M152A

Project Report

TA: Tyler Albarran

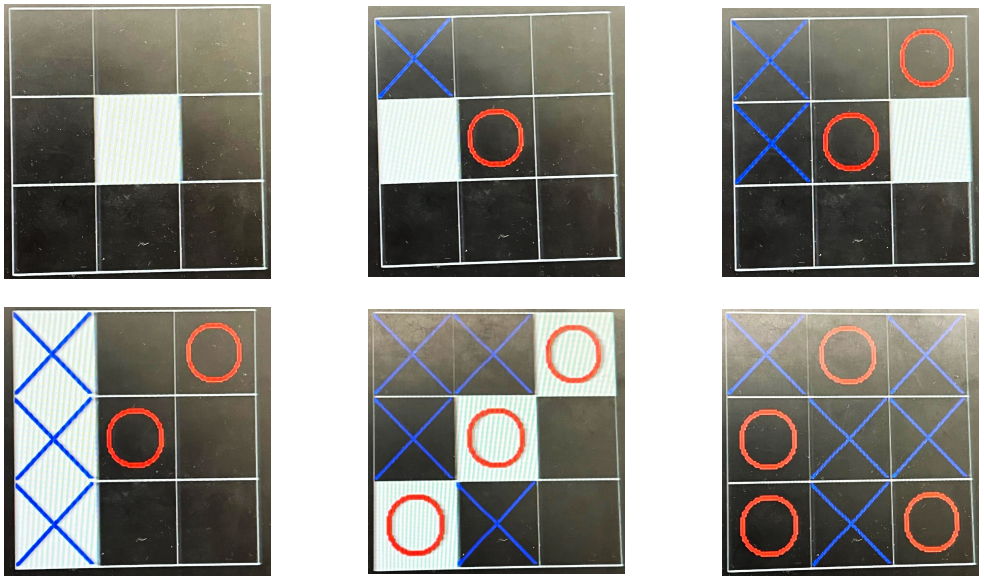
Matthew Fiorella
Jack Zhi

Introduction

The project implements the classic tic-tac-toe game on the Spartan-6 FPGA. The game board size is 3x3, and two human players are involved. The board of the game will be displayed on a monitor via VGA. The player who has the turn can press 4 buttons, UP, DOWN, LEFT, or RIGHT to move a blinking cursor on the board. When the cursor is moved to the target grid, the player can press the ENTER button to record the move, and if the game is not ended yet, the other player gets the turn. During the game, the 7-segment display will show the current location of the cursor, and which player's move is now pending. We also need to constantly check whether or not the game should be over. There can be three outcomes: tie, Player 0 wins, or Player 1 wins. When a game-over condition is met, the game ends automatically, and no new moves are recorded. The final outcome will be shown on the 7-segment display. The user can press the RESET button to start a new game with an empty board.

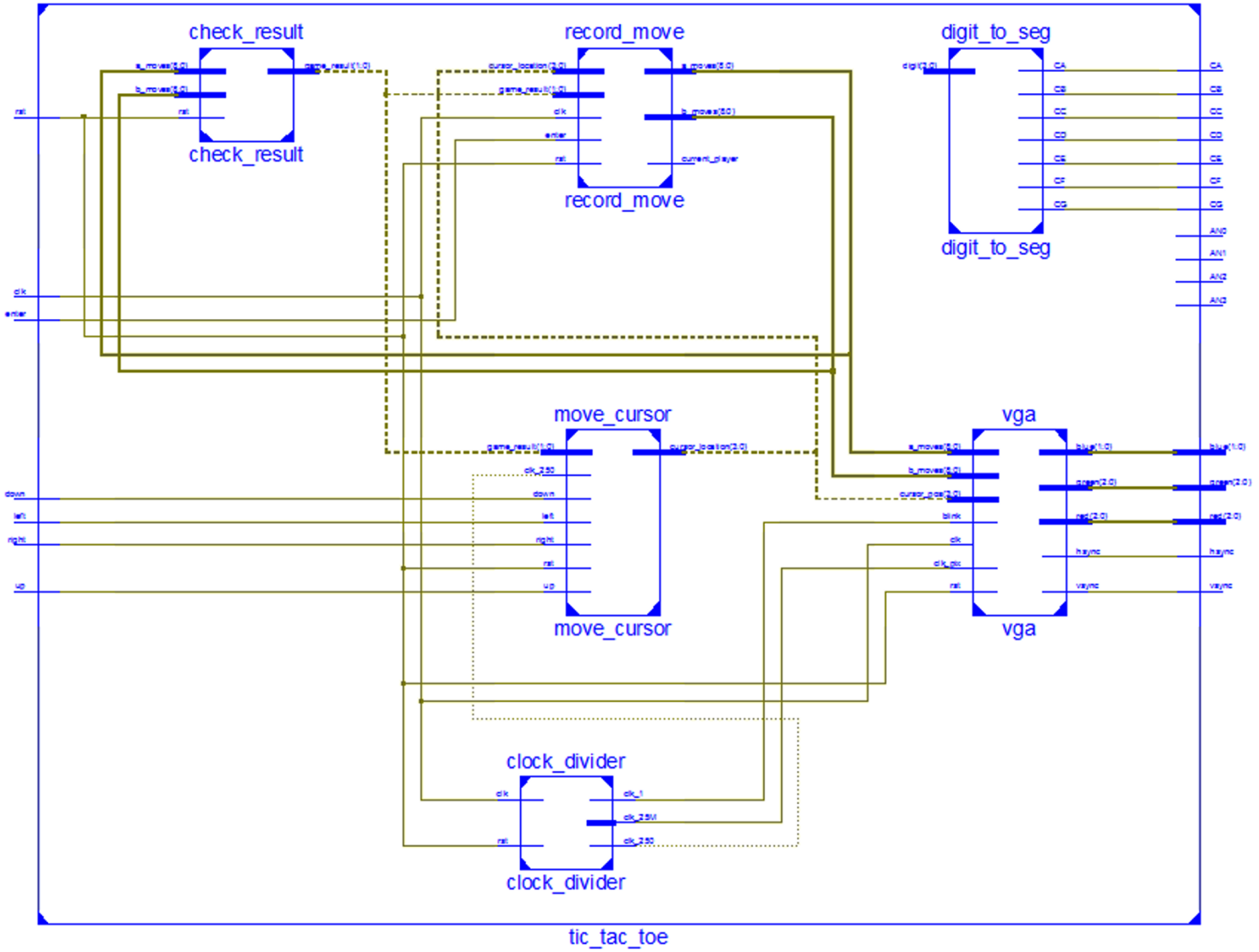
Grading Rubric:

- Empty game board displayed on VGA (10%)
- Cursor with blinking displayed on VGA (10%)
- Cursor location changes if the UP, DOWN, LEFT, RIGHT request is valid (15%)
- Cursor location does not change if the UP, DOWN, LEFT, RIGHT request is invalid (5%)
- Valid moves recorded and displayed after ENTER button pressed (15%)
- Invalid moves ignored after ENTER button pressed (5%)
- Turn handed over to other player after valid move (10%)
- Current cursor location 0-8 displayed on 7-segment display (5%)
- Current turn P0/P1 displayed on 7-segment LCD (5%)
- Final game result E/P1E/P2E displayed on 7-segment LCD after game over (10%)
- Cursor no longer moves and new moves not recorded after game over (5%)
- New game starts after RESET switch is on then off (5%)



Implemetation

The top-level design of the tic tac toe game accomplishes two primary goals. The first is to instantiate all of the sub modules that handle game functionality and VGA display. The second is to compute the proper digit and anode to show on the seven segment display so that the digit to segment converter outputs the correct signals. The block diagram for this module is as follows:



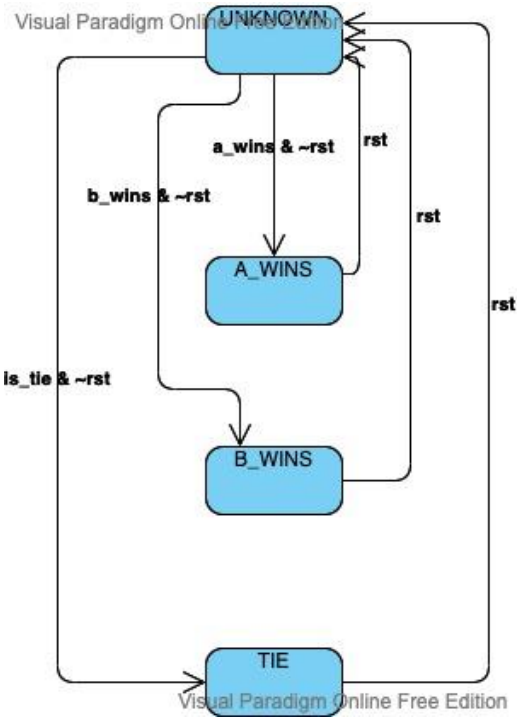
The functionality of each submodule is outlined below:

clock_divider (clock_divider.v): This module is a simple clock divider that outputs three separate clocks needed to manage three separate parts of the tic tac toe game. Utilizing counters, the clock_divider outputs a 1Hz clock to manage the cursor blinking, a 250 Hz clock to perform debouncing and manage the seven segment display, and a 25 MHz clock to function as the pixel clock for the VGA.

move_cursor (move_cursor.v): This module implements the ability to move the cursor. It accomplishes this goal by taking in six primary inputs: each of the four cursor movement directions, a 250 Hz clock for debouncing, and the current state of the game to determine if the cursor should even be moved. The first step this module takes is to debounce the directional inputs, as these inputs come directly from the FPGA board buttons. This is accomplished via a shift register running on the 250 Hz clock. The next step the module completes is to update the row and the column of the cursor based on the state and the directional input. If the state is not UNKNOWN, then the cursor is prevented from moving. It should also be noted that this module itself maintains the cursor row and column so that it can have some degree of memory. The final stage of the move_cursor module is to determine the cursor location index based on the new cursor row and column. This is necessary because the remaining modules utilize the cursor location index rather than cursor row and column.

current_move (current_move.v): This module stores the current move into the current players' moves array, and changes the current player to the next player (i.e If O just went, X is now the current player). A player's move is stored in a 9 bit array where the index of the bits that are high corresponds to the tile index where that player has made moves. This functionality is implemented with a series of conditional statements that utilizes a state input, a cursor location input, and the current player to set a certain index of each players' 9-bit move array to high. The state input ensures that no moves are being recorded in a finished game. The cursor location input allows the exact index of the current players' moves array to be set high. The current player is used to determine which players' moves array to adjust and is ultimately set to the other player after the turn.

check_result (check_result.v): This module checks each player's moves to determine if a win condition has been met or if the game has ended in a tie. The check_result block is effectively the state machine of the Tic Tac Toe game, updating the state to A_WINS or B_WINS if player A or player B won, TIE if the game ends in a tie, or UNKNOWN if the game is still ongoing. The state diagram of check_result is as follows:



digit_to_seg (digit_to_seg.v): This module is a simple digit to segment converter that takes a base 10 encoded digit and converts it to a representation that will be properly shown on the FPGAs seven-segment display. The unique feature about this module is that it converts the numbers 10 and 11 to P and E on the display respectively.

vga (vga.v): This module manages the game board display. This task is completed by rendering each pixel based on the current x and y pixel position. Essentially, this module executes a series of conditional statements that check the pixel position, game state, and cursor state to determine what should be rendered on the screen. For example, if the cursor position is in a game square, then that square will blink according to the blink clock. Another example of this is if a move has been completed in a certain board square, the vga module will draw the shape in that square according to which player made a move in that square. This module utilizes pixel art to draw the X's and O's, creating a 2D array where the X's and O's are marked with 1s and the background is marked with 0s. Additionally, upon a win, this module will highlight the winning combination of squares by calculating the square combination that induced the win.

simple_480p (simple_480p.v): This module controls the vga driving, effectively managing the current x and y pixel position as well as the hsync and vsync signals. While the VGA display is calibrated to the 640x480 pixel screen, there is additional logic that occurs once the current pixel is past those boundaries. The VGA driver also iterates through a blanking period that includes a Front Porch, Sync Period, and Back porch. This makes the dimensions being managed actually 800x525 rather than 640x480. One main takeaway from this is that nothing should be rendered while the pixel position is in the blanking period, but the hsync and vsync signals should be set appropriately during the horizontal and vertical syncing periods so that the monitor knows that the signals being sent to it are correct. Another notable aspect of this is that it informs the user on how to set the pixel clock. Because the display is 800x525 pixels and there is a 60Hz refresh rate on the monitor the pixel clock needs to operate at $800 \times 525 \times 60 \approx 25\,MHz$ to ensure every pixel is rendered before the refresh. On every positive edge of this clock, the VGA driving module will make sure to adjust the x and y pixel position accordingly, incrementing the y position by one if at the end of the back porch, resetting the x and y positions if the final pixel is reached, and simply incrementing the x position by one otherwise.

When all of these modules are compiled together, the following report is generated by Xilinx:

tic_tac_toe Project Status (03/15/2022 - 18:54:37)			
Project File:	project.xise	**Parser Errors:**	No Errors
Module Name:	tic_tac_toe	**Implementation State:**	Synthesized
Target Device:	xc6slx16-3csg324	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	10 Warnings (0 new)
Design Goal:	Balanced	• **Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	
Environment:	System Settings	• **Final Timing Score:**	

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	138	18224	0%
Number of Slice LUTs	1926	9112	21%
Number of fully used LUT-FF pairs	88	1976	4%
Number of bonded IOBs	28	232	12%
Number of BUFG/BUFGCTRLs	3	16	18%
Number of DSP48A1s	3	32	9%

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Tue Mar 15 18:54:35 2022	0	10 Warnings (0 new)	19 Infos (0 new)
Translation Report					
Map Report					
Place and Route Report					
Power Report					
Post-PAR Static Timing Report					
Bitgen Report					

Testbench

check_result.v

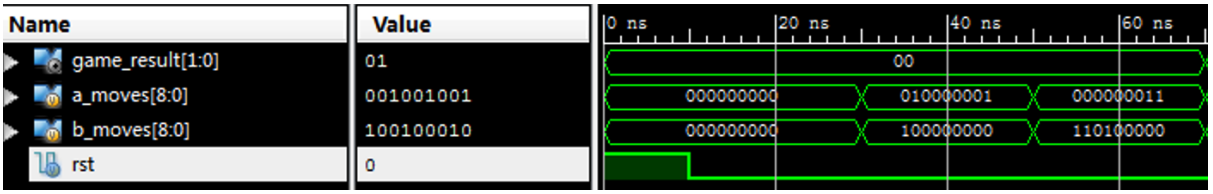
The testbench for check_result.v sends different combinations of a_moves (moves made by Play 0 so far) and b_moves (moves by made Player 1 so far), and we need to check whether game_result is set properly: UNKNOWN, A_WINS, B_WINS, or TIE. Note that check_result is combinational and thus does not need a clock as an input.

We first set rst to high to clear the output. Then we test two different cases where the game outcome is still undetermined.

```
initial begin
    // Initialize Inputs
    a_moves = 0;
    b_moves = 0;
    rst = 1;

    // Wait 100 ns for global reset to finish
    #10;
    rst = 0;
    #20;
    a_moves = 9'b010000001; // result unknown
    b_moves = 9'b100000000;
    #20;
    a_moves = 9'b000000011; // result unknown
    b_moves = 9'b110100000;
    #20;
```

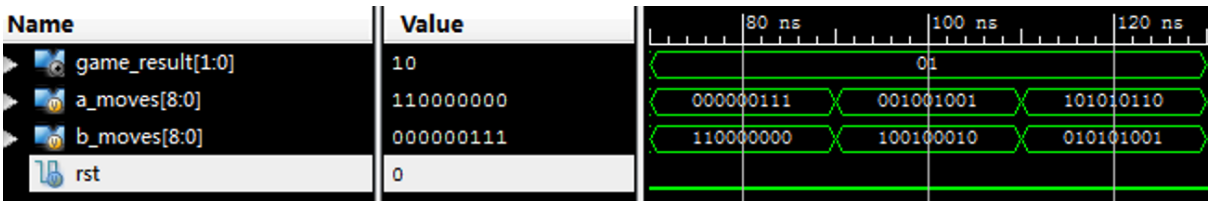
At reset, game_result is set back to UNKNOWN (00). The two cases also output game_result as UNKNOWN:



The following cases test different conditions when A can win. This includes A occupying an entire row, an entire column, or an entire diagonal.

```
a_moves = 9'b000000111; // A wins, row
b_moves = 9'b110000000;
#20;
a_moves = 9'b001001001; // A wins, column
b_moves = 9'b100100010;
#20;
a_moves = 9'b101010110; // A wins, diagnal
b_moves = 9'b010101001;
#20;
```

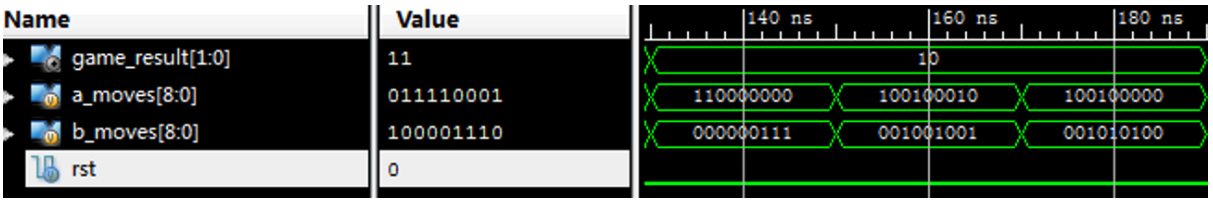
The game_result is properly set to A_WINS (01).



We then apply the same cases, while switching the values of a_moves and b_moves. These correspond to the 3 cases where B can win: occupying an entire row, an entire column, or an entire diagonal.

```
b_moves = 9'b000000111; // B wins, row
a_moves = 9'b110000000;
#20;
b_moves = 9'b001001001; // B wins, column
a_moves = 9'b100100010;
#20;
b_moves = 9'b001010100; // B wins, diagnal
a_moves = 9'b100100000;
#20;
```

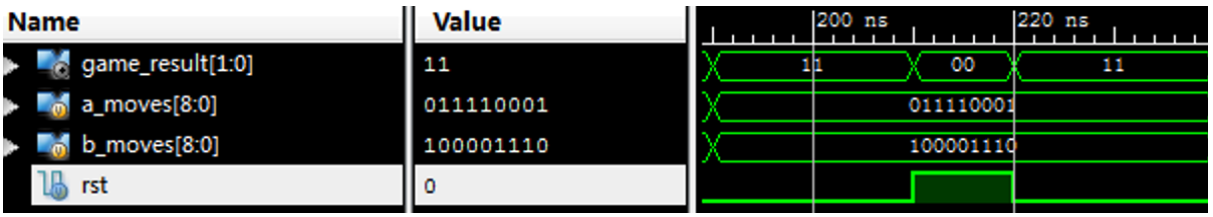
The game_result is properly set to B_WINS (10).



The last case tests the output when the game should be over but tie. Finally, we test the reset again to clear the output.

```
a_moves = 9'b011110001; // tie
b_moves = 9'b100001110;
#20;
rst = 1; // reset game result to unknown, ignore moves
#10;
rst = 0;
#20;
```

The game_result is correctly set to TIE (11). At reset, it is set back to UNKNOWN (00).



record_move.v

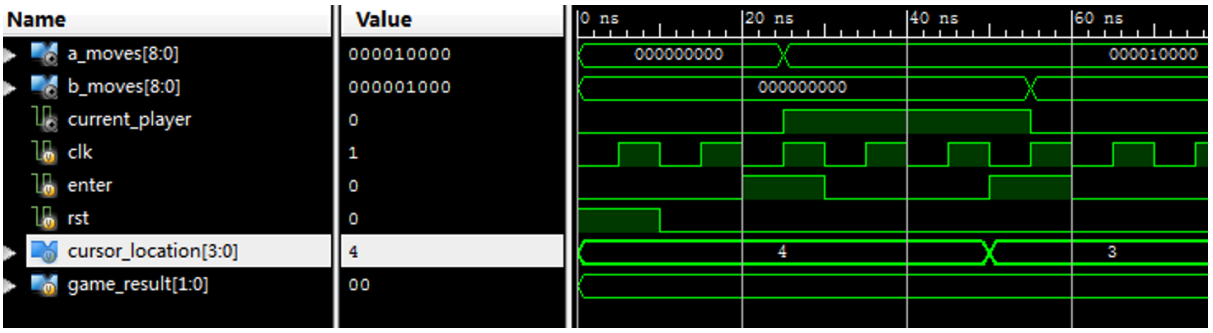
The testbench for record_move.v sends different locations of the cursor on the board and checks whether a move will be correctly validated and recorded on the board when the user hits the enter button.

We begin by sending a reset and place a move by Play 0 at grid 4 (default location).

```
// Initialize Inputs
clk = 0;
enter = 0;
rst = 1;
cursor_location = 4;
game_result = 0;

// Wait 100 ns for global reset to finish
#10;
rst = 0;
#10;
enter = 1;  // play A move at grid 4, turn to B
#10;
enter = 0;
#20;
```

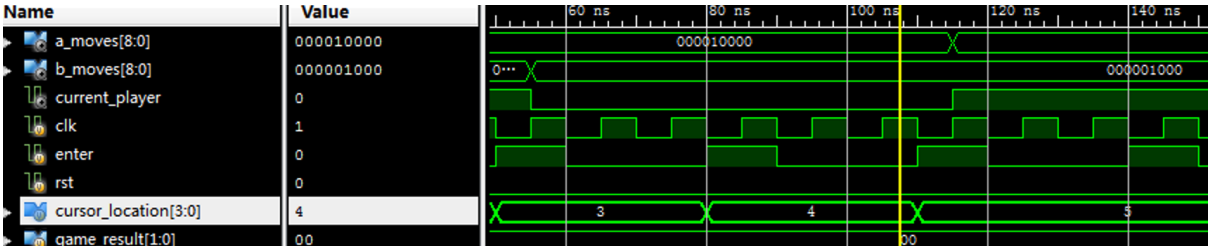
The a_moves is updated to 000010000, and the turn is handed over to Play 1.



Play 1 places a move at grid 3.

```
cursor_location = 3; // play B move at grid 3, turn to A
enter = 1;
#10;
enter = 0;
#20;
```

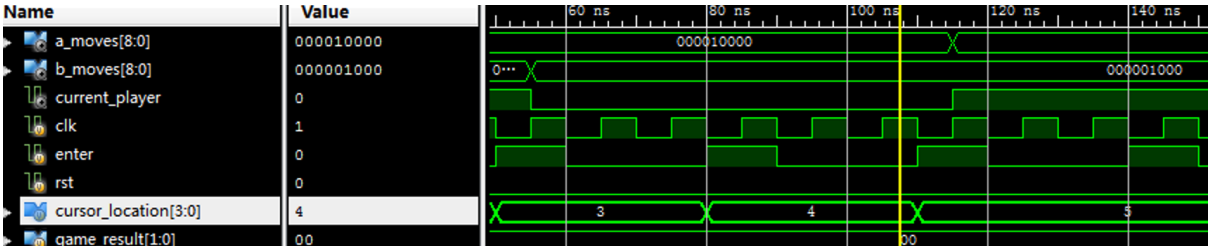
The b_moves is updated to 000001000, and the turn is handed over to Player 0.



Play 0 tries to place a move at grid 4 which is already occupied.

```
cursor_location = 4; // play A move at grid 4 again, ignore
enter = 1;
#10;
enter = 0;
#20;
```

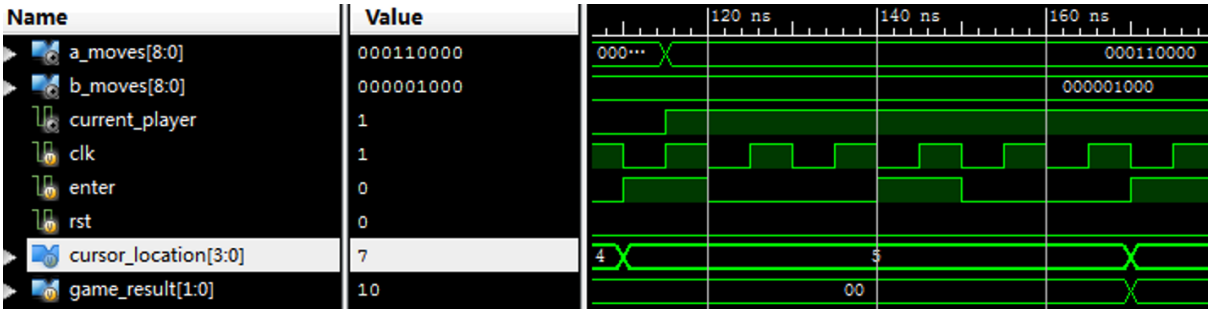
The request is ignored and a_moves remains at 000010000. Turn is not switched.



Play 0 places a move at grid 5.

```
cursor_location = 5; // play A move at grid 5, turn to B
enter = 1;
#10;
enter = 0;
#20;
```

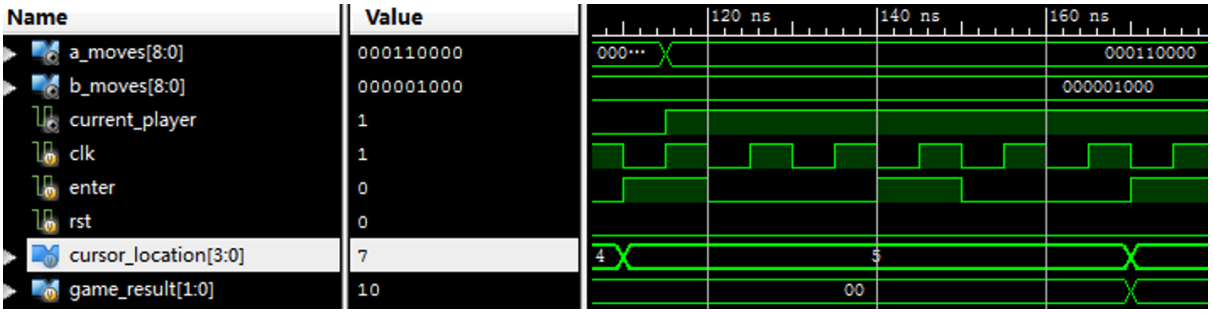
The a_moves is updated to 000110000, and the turn is handed over to Player 1.



Play 1 tries to place a move at grid 5 which is already occupied.

```
enter = 1;  // play B move at grid 5, ignore
#10;
enter = 0;
#20;
```

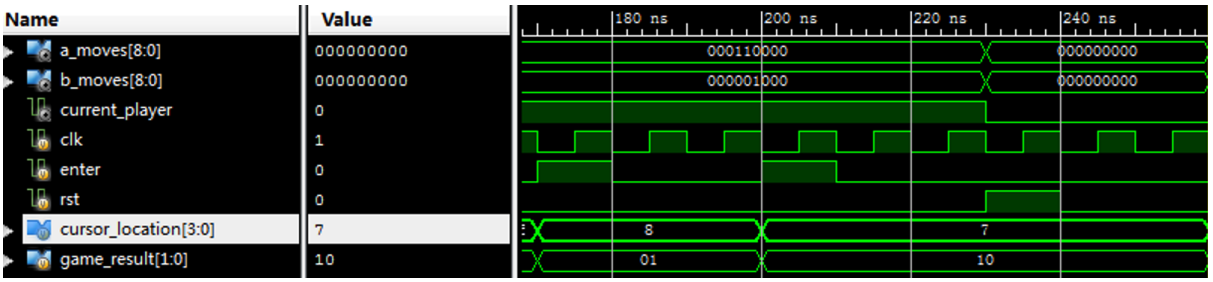
The request is ignored and b_moves remains at 000001000. Turn is not switched.



We manipulate the game_result manually to game-over states, such as A_WINS (01) and B_WINS (10). No more moves should be recorded after the game is over. Finally, a reset is set to clear the board.

```
game_result = 2'b01; // assume A wins, no moves allowed
cursor_location = 8;
enter = 1;
#10;
enter = 0;
#20;
game_result = 2'b10; // assume B wins, no moves allowed
cursor_location = 7;
enter = 1;
#10;
enter = 0;
#20;
rst = 1; // reset, all moves empty and Play A gets turn
#10;
rst = 0;
#20;
```

The values of a_moves and b_moves are not changed after enter is hit. Their values are reset to 0 and the current player is reset to Player 0.



Note: move_cursor.v is not tested in simulation because it is driven by a debouncer. It is tested directly on the FPGA where the 7-segment display shows the current location of the cursor.

Conclusion & Challenges

1. We experienced some issues with the enter input signal. During the final routing phase, ISE complained that it could be assigned to the physical button we used on the FPGA because it would cause timing issues. Attempts by overriding the error failed, as the enter button became completely unresponsive. After discussing the codes with TA, we found that it was because we used statements like “posedge enter” in an always statement to trigger some behaviors when enter was pressed. Theoretically, only clocks were supposed to be coded in this way. We modified the always statement so that it only depended on the master clock and the reset input, and constantly checked whether enter was high at each positive edge of the master clock, to resolve the issue.
2. We experienced some issues when trying to debounce the UP, DOWN, LEFT, RIGHT buttons. Even with a debouncer and a slower sampling clock, a single press would still be interpreted as multiple presses. We later modified the debouncing logic, so that the cursor move could only be registered when the signal went from 0 to 1, and another 1. Thus, even if the user pressed the button and held it for a long time, the cursor would only move once, because the signal would stay at 1 which did not satisfy the moving condition.
3. We experienced some issues in rendering the shapes on the VGA once a player moved. While driving the VGA was relatively simple, drawing complicated shapes was difficult because we were unsure of whether we should calculate the exact x and y parameters that would render the shapes correctly or if there we should take a simpler approach. We also considered storing the X and O shapes into ROM memory before abandoning this idea due to its complexity. Ultimately, we found a simpler approach that consisted of employing pixel art for the X and O shapes. We wanted our shapes to fit in 50x50 pixel board squares, so to render the X and O shapes we created 2500 bit arrays, structured like the two dimensional board squares. This meant we could have the array bits set to 1 in the positions where the X and O should be rendered, and 0 where the background should be rendered. This solution provided an easy fix to the problem of rendering complex shapes as we were able to find the pixel art we needed from an internet resource cited in the verilog code. The only remaining logic to calculate was how to index into these arrays based on the current pixel position, and this wasn’t too difficult. Once this problem was solved, building out the remaining conditional functionality of the vga module went smoothly.