Implementation of Tic-Tac-Toe Game via VGA Interface and VHDL

Digital Electronics (ENGS 31/ COSC 56)
Dartmouth College, Summer 2022
Students: Di Luo, Itish Goel
Instructors: Benjamin Livingston Dobbins, Tad Truex

Introduction

The goal of our final project was to implement the classic tic-tac-toe game in VHDL and to display the game on VGA monitor. Tic-tac-toe is originally a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with X or O. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner.[1] In our implementation, the game logic is stored in a Basys 3 FPGA board. The user interface occurs on the board and the state of the game is displayed via VGA output of the board onto a monitor.

Key specifications:

- Nine blocks on the screen represent nice spaces on the chessboard.
- Player 1 starts in the first round and has red chess pieces. Player 2 starts in the second round and has blue chess pieces. The players alternate turns.
- A selecting frame around the margin of the space indicates the next space that the current player wants to drop.
- Four side push buttons control the movement of the selecting frame. Central button controls the dropping of the chess piece. (**Appendix A**)
- Two switches respectively control the starting and resetting of the game. (**Appendix A**)
- When one of the players wins, the game stops and the final chessboard freezes.

At the end of the project, we successfully fulfilled all the specifications mentioned above and added more features that either improved the user experience or strengthened the game logic. Our final version of the project functionalized perfectly for Player vs. Player Tic-tac-toe.

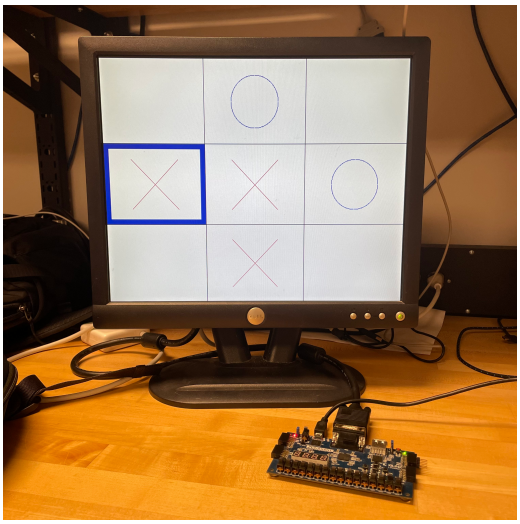

Figure 1: Demonstration of Game with VGA interface and FPGA board

[1] "Tic-Tac-Toe - Wikipedia". 2022. En.Wikipedia.Org. https://en.wikipedia.org/wiki/Tic-tac-toe.

High Level Operation

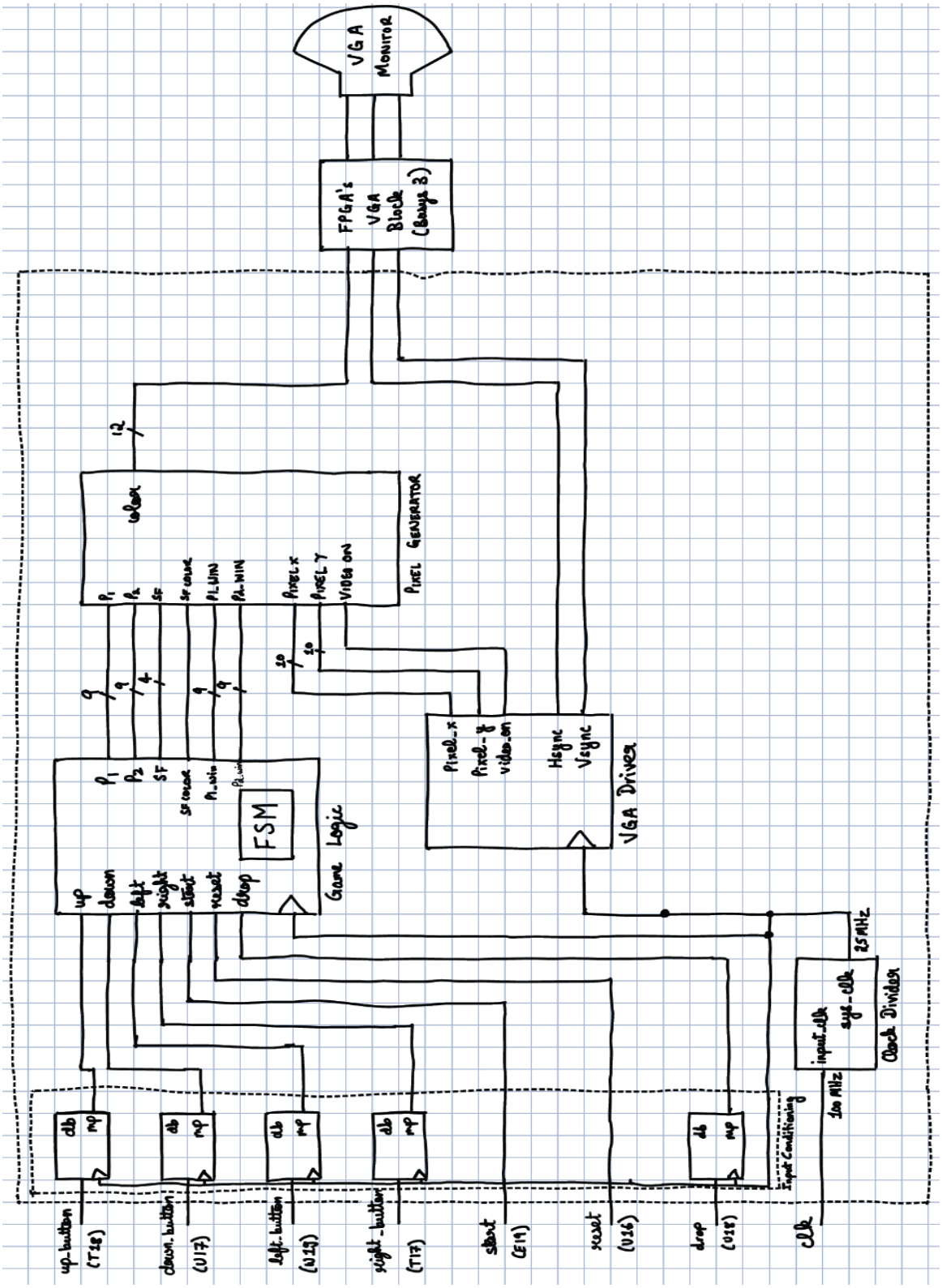

Figure 2: Functional Block Diagram

As can be seen from Figure 2, our project can be divided into five major blocks – Input Conditioning, Clock Divider, Game Logic, VGA Driver, and Pixel Generator. There are eight inputs from the FPGA board - four arrow buttons, a drop button, a start switch, a reset switch, and a 100MHz external clock.

The clock divider block is responsible for converting the 100MHz clock coming in from the FPGA to a 25MHz clock that synchronizes with the VGA monitor as the system clock, which is used by all of the synchronous blocks inside our project, for consistency. The source code of the clock divider block was from Lab 6.

Each of the five button inputs are debounced and monopulsed in an input conditioning block respectively to avoid metastability. The monopulsed button inputs and the external inputs from start and reset switches are then passed along to the game logic block. The source code of the input conditioning block was from Lab 5.

The VGA driver only takes the system clock as input. Vertical and horizontal sync signals *Vsync* and *Hsync* are generated to synchronize the interface with the VGA display. It also sends the coordinates of the current pixel *pixel_x* and *pixel_y* and the blanking indicator *video_on* to the pixel generator.

The game logic block has a Finite State Machine (FSM) that stores the game logic. The state updates according to button/switch inputs. In each state, FSM outputs proper control signals for register files storing the current status of different components of the chess board. The outputs of the registers are connected to the outputs of the block, where *p1* and *p2* indicates the spaces that each player have dropped chess pieces, *sf* and *sf_color* stores the location and the color of the selecting frame (which indicates who the next player is), and *p1_win* and *p2_win* have the locations of the spaces that make the player win when one of the players wins.

Next, the pixel generator uses a logic-defined lookup table to determine the color of the current pixel based on the pixel location and the blanking information from the VGA driver and the game information from the game logic block. The output is a 12-bit vector where each one of RGB takes four bits.

Finally, the two sync outputs from the VGA driver and the color output from the pixel generator are combined for display on the VGA monitor.

Detailed Design Solution

1. VGA Driver

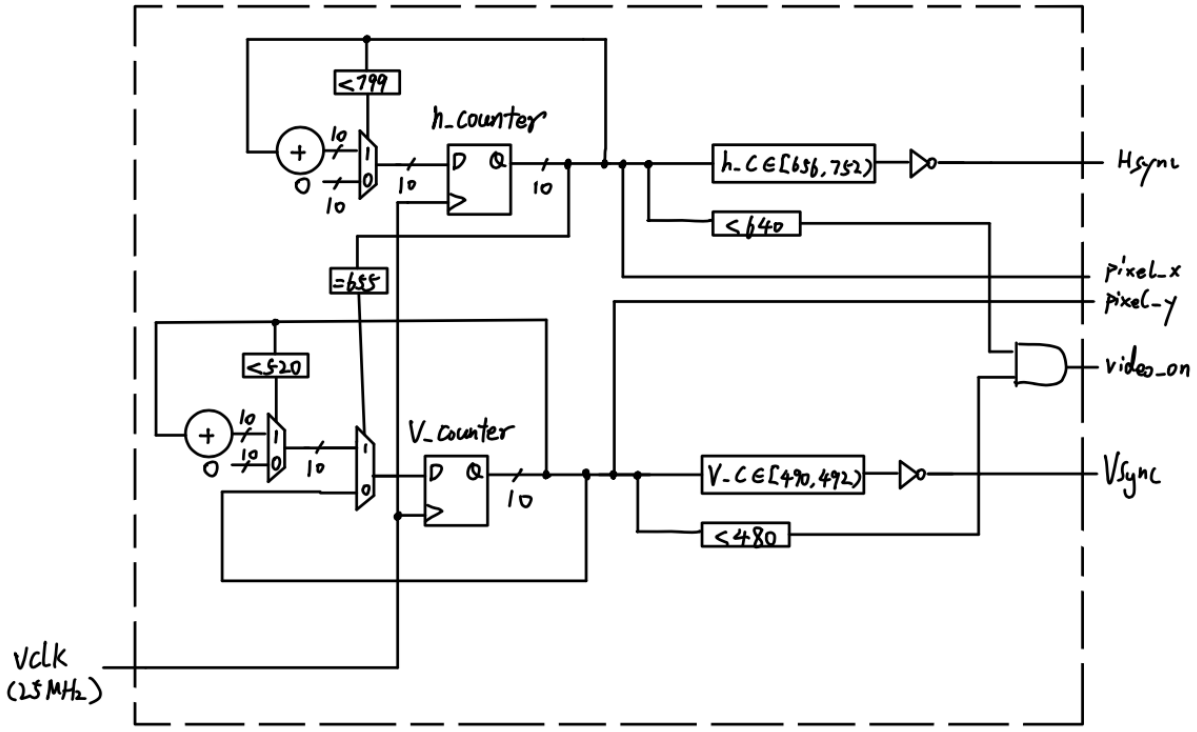

Figure 3: VGA Driver RTL Schematic

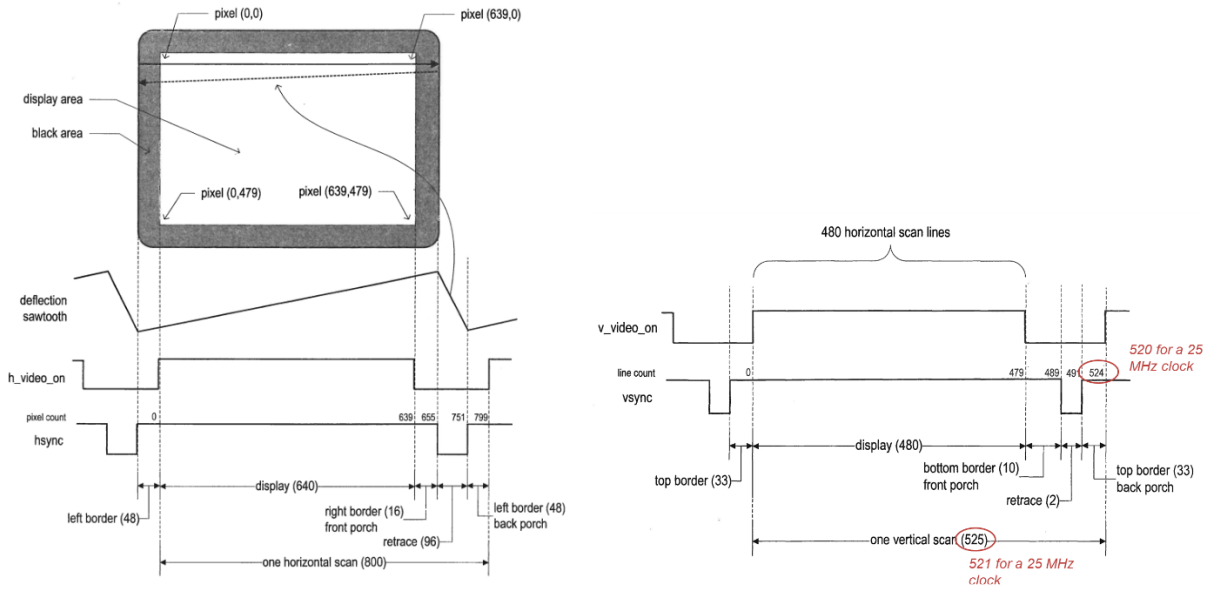

Figure 4: Horizontal and vertical VGA timing[2]

[2] Hansen, Eric. 2021. "Digital Electronics: Supplement To Engs 31 / Cosc 56 Chapter 24 Video Graphics (VGA)". Dartmouth.Edu. http://www.dartmouth.edu/~engs31book/chapter-VGA.html.

The VGA Driver takes in a single input – vclk, as seen in Figure 9, corresponds to the 25MHz system clock generated by the clock divider. Its outputs are comprised of Vsync_port and Hsync_port (single-bit std_logic datastreams passed onto the FPGA's VGA block), and 3 outputs to the pixel generator – video_on_port (accomplishes blanking depending on display area), as well as two 9-bit outputs that indicate the current pixel's x and y coordinates.

We have used a number of hardcoded VGA constants (left_border, h_display …) that enable the proper display of our pixel-wise color logic on the VGA monitor. These have been taken directly from the VGA class notes.

The actual processes inside the VGA Driver can be thought of as 3 broad groups – Sync generating processes, counter generating processes, and the video_on process. All three of these process types have both vertical and horizontal occurrences.

The sync generating processes – Hsync_proc and Vsync_proc – have a fairly simple logic. They are sensitive to h_counter and v_counter respectively, and are high ('1') except for the retracing period. The periodic single-bit outputs generated – Hsync_port and Vsync_port – are passed directly to the FPGA's VGA block.

The counter generating processes are defined as h_counter, and v_counter (as can be seen in Appendix B). They are sensitive to the system clock, and increment every clock cycle, within the constraints of the VGA constants (HSCAN and VSCAN) as specified earlier in the file. The h_counter and v_counter outputs are used internally by the sync generating and video on processes, and sequentially cycle through all the pixels on the screen, thereby enabling the final display on the VGA monitor.

Finally, the video on processes (V_video_on_proc and H_video_on_proc) simply determine whether the sequential process is currently within the active display area. They each set an internal signal (V_video_on and H_video_on) to high ('1') when in the active display area, and to low ('0'), when outside the active display area. The video_on_port output, for its part, is high only when both V_video_on and H_video_on are high, thereby ensuring that the video out is only enabled when the pixel is in the active display area, both from an x and y coordinate perspective.

2. Game Logic

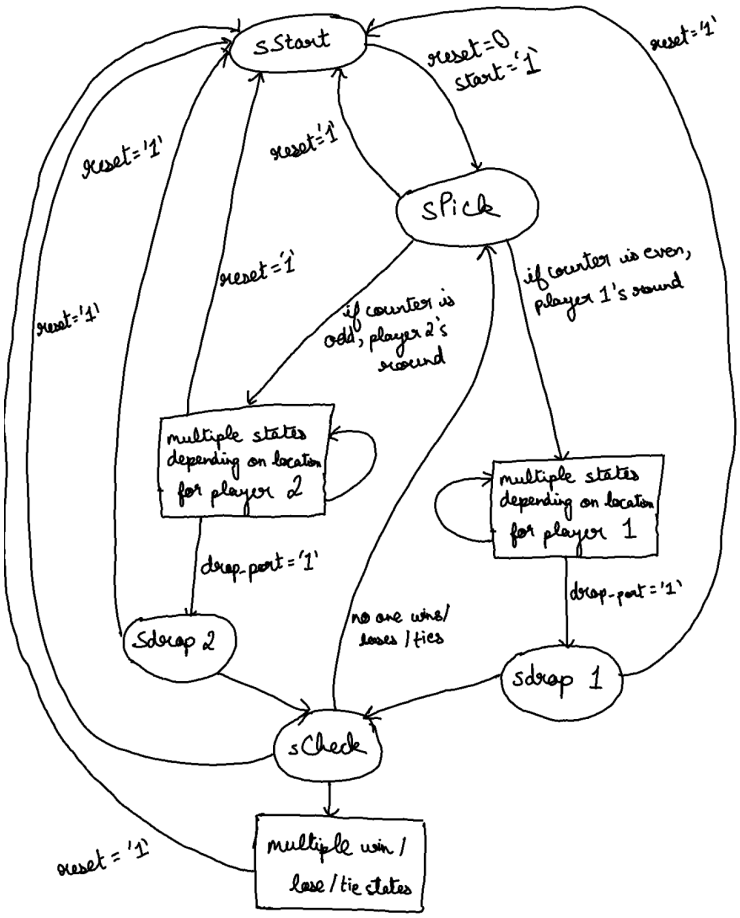

Figure 5: FSM Diagram

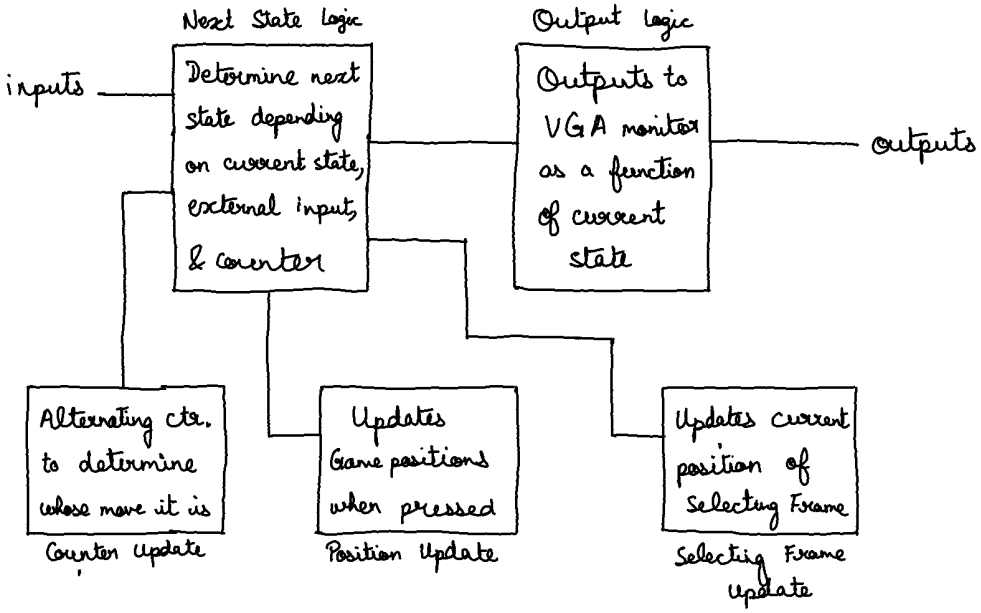

Figure 6: Game Logic Flow Chart

FSM Description:
The game starts in the sStart state. This is the default state for the game, where the positions and counters of the game are reset. At this point, if the user flips the start switch (E19) to 'on' position, the game moves into the sPick state.

The sPick state places the current player in the central square of the grid, with the corresponding grid square highlighted to convey it to the player. The current player is selected alternatively The sPick state acts as the default starting point after each player makes a move, and it is the other's turn (unless the game has already ended. Once in the sPick state, the player can move around using the arrow keys, which will lead to one of the 18 player-position states

There is one state each for both players in each of the 9 grid-locations, leading to a total of 18. As the player moves around, using the arrow-key setup, the VGA monitor shows the current location with a blue outline around the relevant-grid square. The 4 arrow keys can be used to move around among the different squares, which correspond to 9 unique states (sTopLeft1, sTopMid1…….). When the player is ready to commit to a given grid-square, they can press the drop button (U18), which will move the game into the sDrop1 or sDrop2 state depending on whether it was player 1 or 2's chance.

The actual updation of the std_logic_vectors that contain information about the play history and each of the players' positions takes place in their respective drop states. Following this, the sDrop1 and sDrop2 states automatically lead into the sCheck state.

The sCheck state, for its part, is responsible for checking whether the set of current positions occupied by either of the players contains a contiguous section that would lead to a win for one of them (three contiguous horizontal, vertical, or diagonal grid squares occupied by the same player). The game also checks whether all of the squares have been occupied and the game is thereby a tie. If such a combination is established, the next state is one of 16 win-states (8 per player) or the sTie state. Otherwise, the game reverts to the sPick state.

Finally, the win and tie states lock-in the current state of the game such that moving the arrow button has no effect. They further update the p1_win_port and p2_win_port to reflect the blocks that have led to one of the players winning, so that this information can be conveyed on the VGA monitor.

Flipping the reset switch (U16) from any state except the win or tie states leads to a reversion to the sStart state, where the positions of the players (reflecting the game history) is cleared and the selection frame is reset back to the central grid.

The Game Logic block is responsible for the bulk of the input parsing, and updation of game states depending on these inputs. The block takes in 9 std_logic inputs – system clock, 4 arrow buttons (up, down, left, and right), drop button, start switch and reset switch. There are, in turn, 6 outputs from the game logic entity:

- P1_port → 9-bit std_logic vector that indicates the blocks occupied by player 1
- P1_win_port → 9-bit std_logic vector that indicates the blocks that make player 1 win
- P2_port → 9-bit std_logic vector that indicates the blocks occupied by player 2
- P2_win_port → 9-bit std_logic vector that indicates the blocks that make player 2 win
- Sf_color_port → 1-bit std_logic output that indicates which player's turn it is
- Sf_port → 4-bit std_logic vector that indicates the location of the selecting frame

The actual process logic of the game has been divided into 5 major process blocks, as can be seen in Appendix C. The first of these major blocks is NextStateLogic, which takes care of much of the interaction between the datapath and the controller. Inside NextStateLogic, we have outlined cases for all possible input combinations in each of the 40 different states in our FSM (sStart, sPick, 18 player-position states, 2 drop states, sCheck, and 17 win/tie states). The process itself is sensitive to the current state, the external button and switch inputs, as well as internal signals that we have defined to keep track of, and update the status of the game.

Next, the OutputLogic process contains logic for updating the output ports based on historical plays as well as the current location of the player (as stored in the current state of the game). The OutputLogic process is sensitive only to the current state of the game, which is controlled by the NextStateLogic block.

The CounterUpdate process controls the player turn-by-turn logic part of the game. The process itself is sensitive to the system clock and other internal signals surrounding the player-turn logic (counter_reset, counter_en, and counter_pause). While the counter update is essentially a D flip-flop (happens on the rising edge of the clock), there is additional conditionality in our logic; the counter is updated only when the player drops a cross or knot on a grid square. This is essentially based on standard Tic-Tac-Toe logic wherein a turn is considered to have happened only when a player actually moves, not on some recurring clock cycle.

The PositionUpdate block simply checks whether the player's attempt to drop his piece on the current grid-square is valid (if there is a pre-existing cross or knot in the square), and then proceeds to update the internal signals containing each player's positions (which are later mapped to output vectors at a later stage). The process itself is sensitive to the external clock, the position of the selection frame and internal signals (position_update_reset,

p1_position_update_en, p2_position_update_en) that determine whether a player has dropped his piece and if the update should happen.

Lastly, the SelectingFrameUpdate block is a fairly straightforward process that updates the external sf_position port based on an internal signal sf_position_idx. It acts as an effective D flip-flop, with the update happening at every rising edge of the clock. The process block is sensitive to both the system clock and internal signal sf_position_idx.

3. Pixel Generation

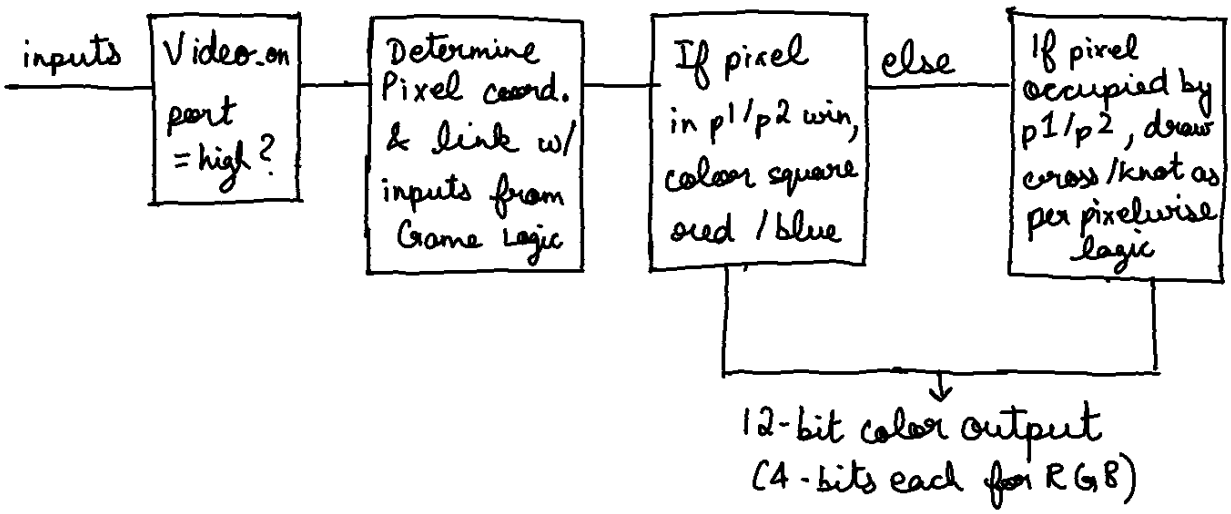

Figure 11: Pixel Generation Logic Flow Chart

The Pixel Generator block is the last high level block, before pixel-wise information is conveyed to the FPGA's VGA block for display on the VGA monitor. It takes in information from both the VGA Driver and the game logic controller blocks. The former sequentially passes it along two 9-bit inputs that indicate the current pixel's x and y coordinates on the VGA grid, as well as a single-bit video_on_port input that conveys whether or not the said pixel is in the active display area. The game logic controller, on the other hand, passes along all of its outputs directly to the pixel generator with the 6 total datalines (p1_port, p1_win_port, p2_port, p2_win_port, sf_color_port, and sf_port) conveying the complete current state of the as described in section 3 above. The pixel generator block thus correlates the current pixel information coming in from the VGA driver with the current state of the game coming in from the game logic controller to generate a 12-bit output that represents the color for the said pixel given the current state of the game.

As seen in Appendix D, there is a single process inside the pixel generator block that we have termed pixel_rendering. The process is sensitive to the inputted pixel coordinates (which we have converted into two unsigned variables), the video_on_port, as well as all the aforementioned inputs from the game logic controller. Therefore, any change in any of the inputs will drive the process to re-evaluate.

The first step in the process is an evaluation of whether the video_on_port is high. In case it is not, no further logic is needed as there is nothing to be displayed outside of the active display area. Next, we have outlined multiple cases to determine which grid square the current pixel lies in. For instance, a pixel with x coordinate between 160 and 320, and a y coordinate between 426 and 640 would lie in the center-right grid square, or the 5th grid square if we were to start counting from 0 in the top left grid square. Once we have determined the grid-square, we are only concerned with the data-points we have for that particular location. In this instance, we would be concerned only with what each of the inputs have to say for the 5th location in the vectors.

Having determined the location, we correlate the information we have from the data inputs, with the pixel-location data, giving precedence to the p1_win and p2_win information as compared to the p1_port and p2_port. Continuing with the example, we would check:

1. If p1_win_port(5) is high → color the pixel red
2. Else If p2_win_port(5) is high → color the pixel blue
3. Else
    a. If p1_port(5) is high → carry out additional logic for cross (color pixels that make up the cross red, and the rest white)
    b. Else If p1_port(5) is high → carry out additional logic for knot (color pixels that make up the knot blue, and the rest white)

The background color for our simulation is white, so we have included a default case in each of the grid-square cases, to take care of the residual white background. Further, we have used greater than and less than comparables in our code, which leaves the actual grid-line colors undefined. This is a conscious choice, as it leaves the grid_line color allocation to the very end, where we have set it to black, thereby giving a visual effect of grid-lines.

Finally, the effect of the selection frame is achieved because of a 10-pixel margin in the comparative logic of step 3 as compared to steps 1 or 2 (as seen above). If we reach step 3, our code checks whether the current grid_square is the location of the selection frame. If so, the entire square's would be colored red or blue respectively. However, nly the 10 pixel margin ends up being visible because of overwriting that happens in steps 3a or 3b respectively.

Conclusion & Evaluation

Our goal in this project was to implement a VGA version of tic-tac-toe game with VHDL. At the end of the project, we successfully completed all the specifications listed in the introduction that we planned to implement.

Furthermore, we accomplished extending our minimum viable product (MVP) by improving user experience and strengthening game logic. As for improving user experience, colors were carefully chosen to keep the contrast between different components with proper color saturation. Chess pieces were made into actual crosses and circles. Color of the selecting frame can indicate who the current player is. Regarding the enhancement of game logic, players were prevented from dropping to an occupied space. When one of the players wins, the blocks that have three pieces in a row/column/diagonal turn to fully colored rectangles instead of X or O to clearly show who and how wins.

In addition, we implemented a working prototype 1 that uses nine switches instead of 4+1 buttons to drop chess pieces, which could possibly be useful in some situations.

Reflection

In our design, what we like best is the modularity of the system that we achieved by applying RTL design. After having a top-level design, we implemented each main block and tested with them independently. Starting with the VGA driver, we created a VGAtest project in Vivado that has a newly-implemented VGA driver, two test patterns (**Appendix J**) in a pixel generation file based on a file from the project page on Canvas, and a top-level file incorporating these components. The testing proved the correctness of our VGA driver so that we could incorporate it into our prototypes without worry. We had similar tests for other components after implementing them respectively, which enabled us to find problems and to solve them as soon as possible and gave us more confidence through the process that our final product would work.

However, when we made a testbench for game logic, we had the greatest challenge. The simulation gave correct results but the actual testing on the VGA monitor failed. The reason was found to be the D-latches that we mistakenly created in the output logic of FSM. They didn’t cause any problems in the simulation. But because the FPGA board doesn't have good support for D-latches, they caused problems on our board. From the experience, we learned the limitations of testbench simulation and got a clearer sense on the relationship between controller and datapath, which enabled us to avoid creating latches when adding states to FSM in later prototypes.

In our design process, one thing we should have done differently was to have better paper designs for game logic and pixel generation before writing VHDL codes. We violated the RTL design process to go into the VHDL stage before having a valid paper design, which took us more time to think about higher level design questions when writing VHDL codes. So our suggestion to future groups considering similar projects is to strictly follow the steps of RTL design. Paper designs may seem to be time consuming and not highly related to the final product, but they actually save more time than figuring out problems in the VHDL stage.

Acknowledgments

Thanks to our course instructor Ben for pointing out the mistakes in the VGA driver, the reasoning behind the problems caused by D-latches, and so on; our lab instructor Tad for clearing up our ideas on the top-level design and suggesting new test pattern for VGAtest; TA Raif for helping Di reconstruct FSM when adding the functionality of checking who and how wins and figure out the math formulas for drawing circles and crosses (**Appendix D, line 86&94**).

In regard to contributions, both of us worked on figuring out the top-level design with Tad and the Introduction section of the Report. In addition, Di worked on Checkpoint 2 and 4-7, all VHDL codes, the Conclusion & Evaluation, Reflection, Acknowledgements, References, Appendix sections of the report, and revised High level operation and the first page of VGA driver in Detailed design solution. Itish worked on Checkpoint 3, Cover page, High level operation, and Detailed design solution sections of the report.

References

"Tic-Tac-Toe - Wikipedia". 2022. En.Wikipedia.Org. https://en.wikipedia.org/wiki/Tic-tac-toe.

Hansen, Eric. 2021. "Digital Electronics: Supplement To Engs 31 / Cosc 56 Chapter 24 Video Graphics (VGA)". Dartmouth.Edu. http://www.dartmouth.edu/~engs31book/chapter-VGA.html.