FPGA Based Tic-Tac-Toe Game

with VGA Output and AI Single Player

Matthew Button, Chris Lair, Kacper Wojtowicz
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
mcbutton@oakland.edu, kwojtowicz@oakland.edu, lair@oakland.edu

A tic-tac-toe game using a single 16-button keypad as input interface for both players and VGA output to display board and pieces. Also included is a single player switch that can be flipped at any time to transfer control from the second player to an AI player.

I. Introduction

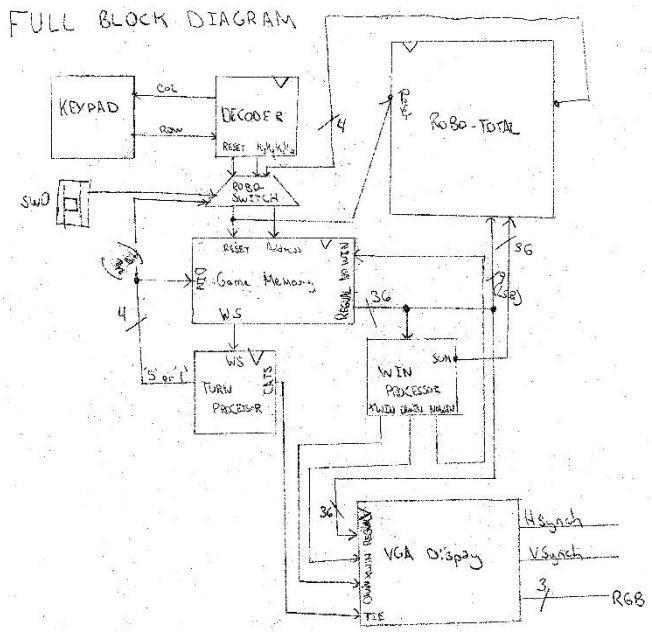
Tic-tac-toe is a simple yet universally known game, so the implementation of such a simple problem-set in hardware seemed like a fair challenge. The project includes so much of what was covered in ECE 2700, the components include, LUT's, multiplexors, registers, shift registers, adders, a finite state machine, and counters, truly the full gamut of class concepts. The VGA portion of the design did require some additional independent effort, and the driver is not borrowed from the course website but designed by itself. The base two player version was completed early which motivated us to add the additional single player components. This AI portion involved looking into the Min/Max Algorithm and its implications for a tic tac toe game and then transferring that concept into something ingestible by VHDL. While the purpose of the device is entertainment, we had to wade into the nuances of tic-tac-toe strategy and the challenges of modular design to build the system.

II. Methodology

The system was devised in layers that built upon each other to form the final product. First, we designed the Key Input then the Game Memory, the Turn Processor, the VGA Display, and finally the AI for the game. The following details each major component of the game highlighting design features and explaining processes. Component names are deliberately bolded and match names found in the attached project code.

A. Keypad Input

The "Digilent Pmod KYPD: 16-button Keypad" is utilized for user input. The keypad inputs can be read by activating a column of the keypad at each clock tick and reading the returned row value to know which key is pressed. Our keypad **decoder** for this keypad relies heavily on the one provided by its manufacturer Diligent [1]. However, this code was condensed because we only utilize the upper left 3 by 3 of the keypad, and the 'D' key for asynchronous reset. The decoder output is 5 bits, the first designates reset and the remaining 4 represent the cell number of a corresponding tic-tac-toe move. A full top diagram starting with the keypad decoder and including all other system components is shown below:



B. Game Memory

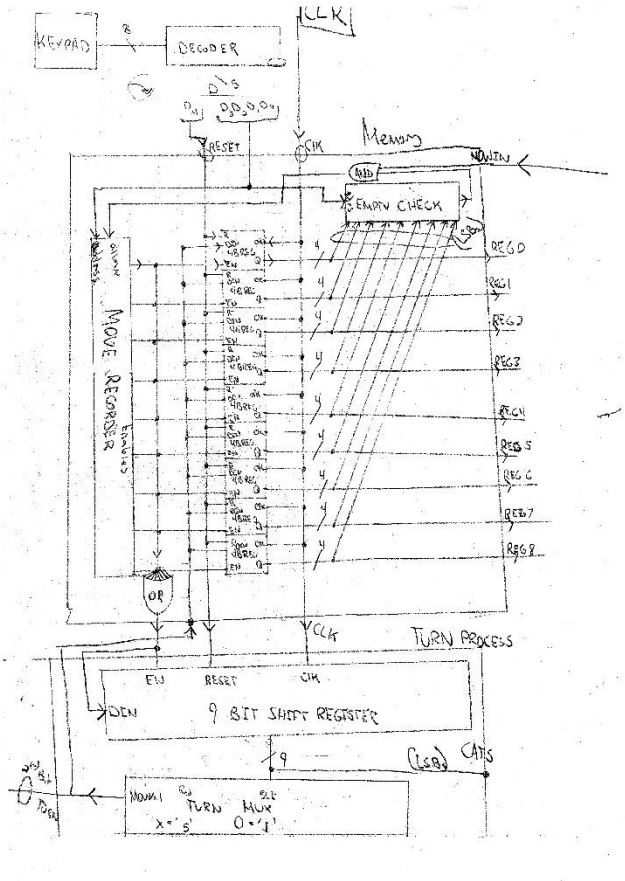
The **game memory** consists of nine 4-bit registers. These registers numbered 0 to 8 represent the empty cells of a tic-tac-toe grid. Reset sets all these registers to '0'. These registers are all fed the same data input value, either '5' or '1' for 'X' or 'O' or for the case of the VGA output "Red" or "Blue". Once an address comes down from the **keypad decoder**, it is sent to an **empty check** module which takes the LSB's of the registers and verifies that no player has won and that the desired cell is empty and playable.

0	**1**	**2**
3	**4**	**5**
6	**7**	**8**

Simultaneously, the address is sent to **move recorder** multiplexor which activates enable for the desired cell register. When a register is successfully overwritten a "write success" signal is sent to the **turn processor**, and this unit proceeds to the next turn by alternating the register data input between '1' and '5'. This memory setup prevents playing in the same cell twice, keeping to the rules of tic-tac-toe, and furthermore prevents double-playing when a long button hold could cause an input to be read twice. It also prevents input immediately after a player win.

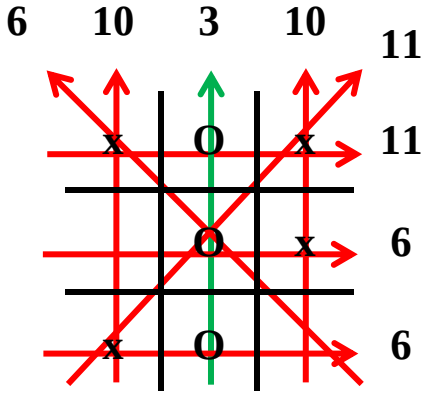
C. Turn Processor

The game tracks turns with a single 9-bit right shift register. Reset zeros out all nine bits. Each time the **turn processor** receives a '1' from the **write success** bit in the Game Memory, a '1' is added into this shift register. A multiplexor then determines if the register outputs are an even number of ones, indicating X's turn or an odd number of ones indicating O's turn. This **turn multiplexor** then alternates outputting '5' or '1' to the memory as turns progress. If all nine of the bits in the register are one this indicates a possible tie so a special **cats** bit is output to help in determining a winner later. The block diagram of the data path from keypad to memory and turn processing is included below:



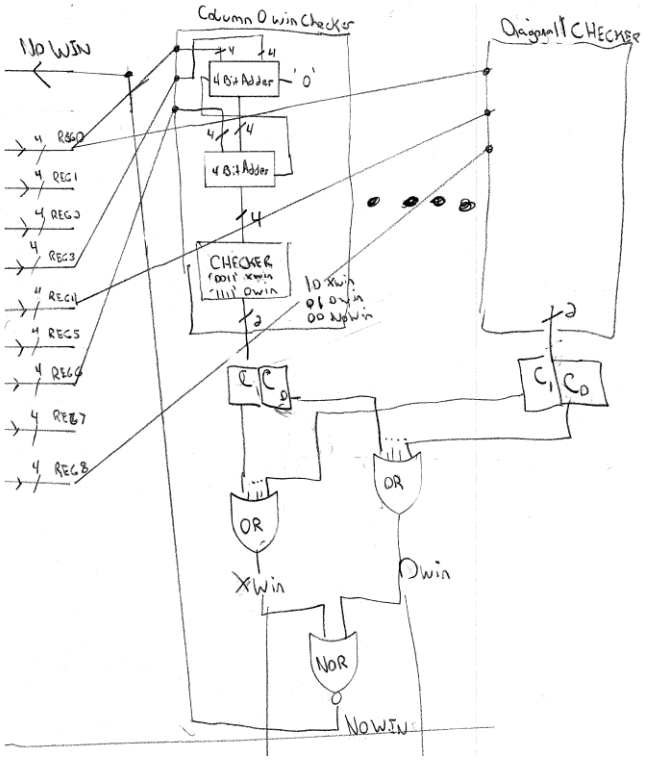
D. Win Processor

The system determines a win by adding up the register values for sets of three registers that form a win. Obviously, you can win tic-tac-toe with three in a row, column, or diagonal, and so there are 8 win combinations. Because 'X' is represented by '5' then three in a row will mean a sum of '15' and conversely because 'O' is represented by '1' a sum



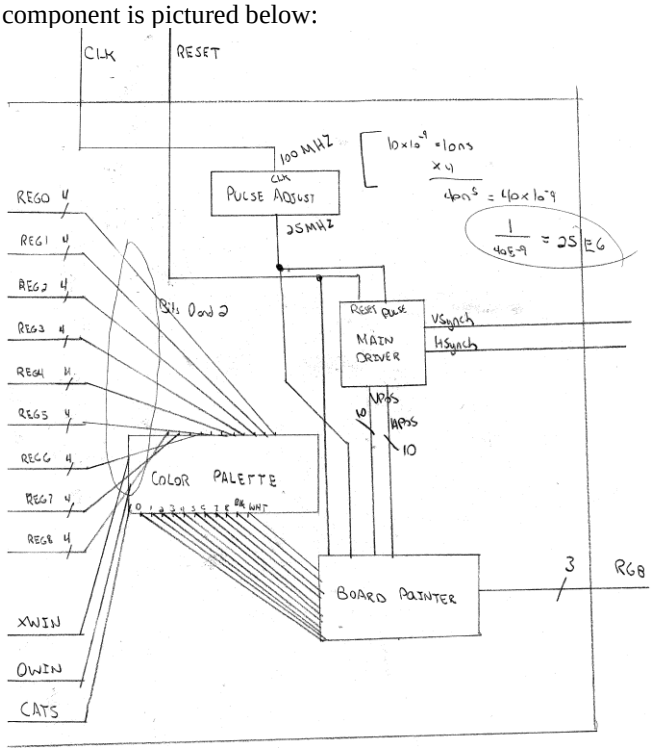
of '3' indicates an 'O' win. For each win combination the system combines two 4-bit adders to add three of the registers, and then uses two comparators to test if they sum to '15' or '3'. Each of the 8 combinations is tested and if any or multiple result in a win the game halts and a win signal is output. If no player has won a "no win" signal allows players to keep placing moves in memory.

Above is a sample board configuration with the sums of each win combination shown, notice the center column sums to '3' indicating a win. Also below is the block diagram for **win processing**:



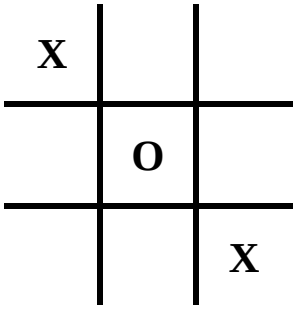
E. VGA Display

The **VGA display** was the most daunting task of the project but ended up being the easiest. To obtain a resolution of 640x480 pixels with an industry standard refresh rate of 60 Hz we utilized timing and display parameters from TinyVGA.com [2]. The VGA Display includes a **pulse generator** which acts as a 25 MHZ clock for the full display. A **main driver** uses four behavioral process to shift the horizontal and vertical position of pixels while also producing H-Synch and V-Synch signals and turning the display on when said positions fall into the monitor range. The current pixel position is then fed to the board painter which uses a series of elaborate if statements to determine shapes and colors on the board. These areas are drawn out in horizontal ribbons first and then vertical positions are divided out. We utilized this simple driver because tic-tac-toe has a simple aesthetic. The driver produces a 3 by 3 white grid, and then within the grid nine small piece can be placed. Additionally, on the left side a banner appears in the case of a win or tie to indicate that result. The colors of each part of the board are defined in a **color palette** component. This component draws the zeroth and second bits of each memory register to determine if and 'X' or 'O' was played and then returns a color based on those inputs. Either black for empty, red for 'X', and blue for 'O'. If the single player switch is on, the second 'O' player switches colors to green to indicate the AI is playing. There is also a **win color** component that uses LUT's to output the correct color for the left-hand win/tie banner. The **board painter** uses these color inputs to output a 3-bit RGB signal, so we have the potential for eight stunning colors. The block diagram for the VGA Display component is pictured below:



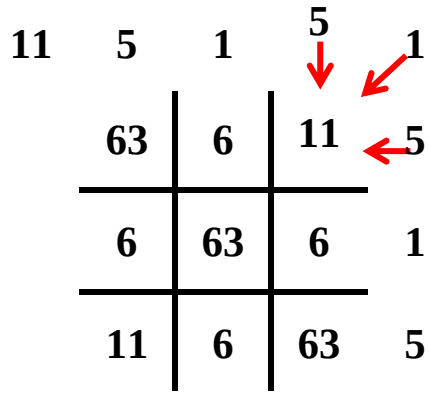
F. AI Single Player

The AI portion was complete after the base game was done. A multiplexor **robo-switch** controlled by the current turn and by the single player switch (sw0) swaps between human and robot players. This switch is inserted between the decoder and memory for the original game. A component **robo-vision** determines a value for a given cell based on the win combination sums that overlap it. It also uses the final bit from the memory registers to verify whether a spot is empty. Occupied cell positions are given a maximum value of 63. This component implements using two 6-bit adders and eight comparators. A sample the board configuration is shown below:

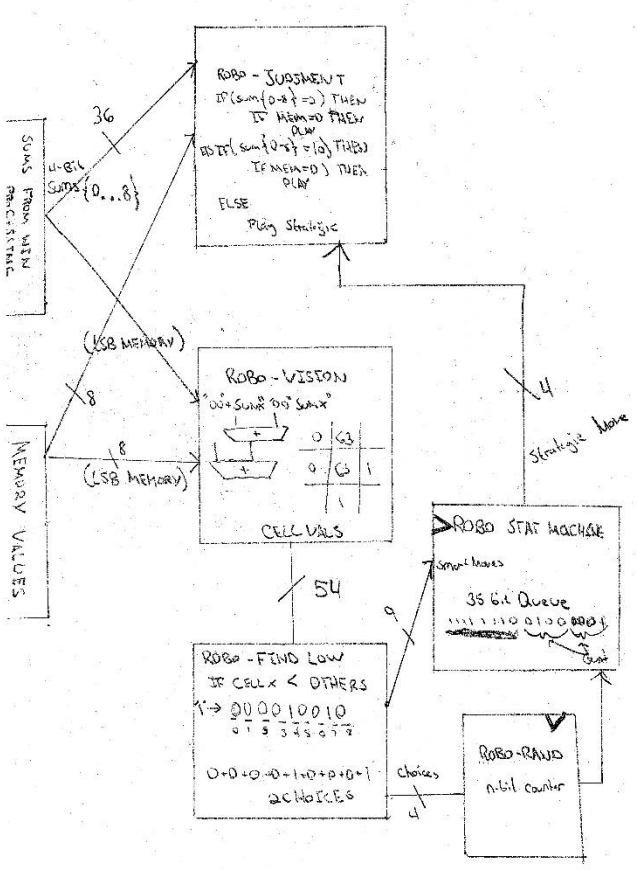


This particular diagonal board configuration (on the left) results in the bordering sums below. Cells 0, 4, and 8 are marked 63 because they are occupied. The remaining cells are marked based on the sum of bordering win sums that overlap that cell. For example Cell 2 is marked 11 or 5+1+5.

These cell values are then sent to **robo-brain** which first uses a series of comparators to find the lowest value. And then outputs a 9-bit vector that indicates which cells share the share this lowest value. For the exampled configuration 6 is the lowest cell value, and it is shared by cells 1, 3, 5, and 7. So the output vector looks like "010101010". The bits of this vector are then summed to 4, this represents the number of equally optimal choices the AI must select amongst. Because the AI plays as player 2 or O and 'O' is a '1' in memory (rather than a '5') the lowest value cells indicate optimal moves. By being lowest, that cell is closer to an O win (which sums to 3) and closer to other O placements. This simple cell ranking is how the AI selects moves that are strategic. **Robo-state-machine** driven by the clock then examines each of the nine bits of "010101010", if it encounters a '1' then that bit's 4-bit position is appended to a long 36-bit shift register. After 9 cycles (examining all 9 bits) the state machine outputs a "done" signal and the 36-bit shift register holds all the optimal moves for that round at its



tail end. The **robo-state machine** then looks at an n-bit **robo-random counter** and selects one of the optimal moves based on the its value. This n-bit counter, actually takes the aforementioned 4, representing the number of optimal choices, as its limit. So, while the state machine builds a queue of optimal choices in the shift register, the counter counts from 1 to 4 until the "done" state uses this semi-random counter value to pick a strategic move. This strategic move value is then sent to a **robo-judgement** component. This component first checks the bordering sums for a 2, indicating a potential O win. If it finds one its plays in that row, column, or diagonal. Otherwise it checks for a border sum of 10 indicating a potential X win. It then blocks that win. This win checking utilizes the border sums with a series of comparators. It then checks the final bits of the memory registers to determine which cell in a particular row, column, or diagonal is unoccupied. Only when the AI cannot win, and cannot stop a win, does it then utilize the strategic move computed by the **robo-brain**. The output of the **robo-judgement** is fed to the **robo-switch** and allows the AI to play against the human. The block diagram of the AI component is shown below:

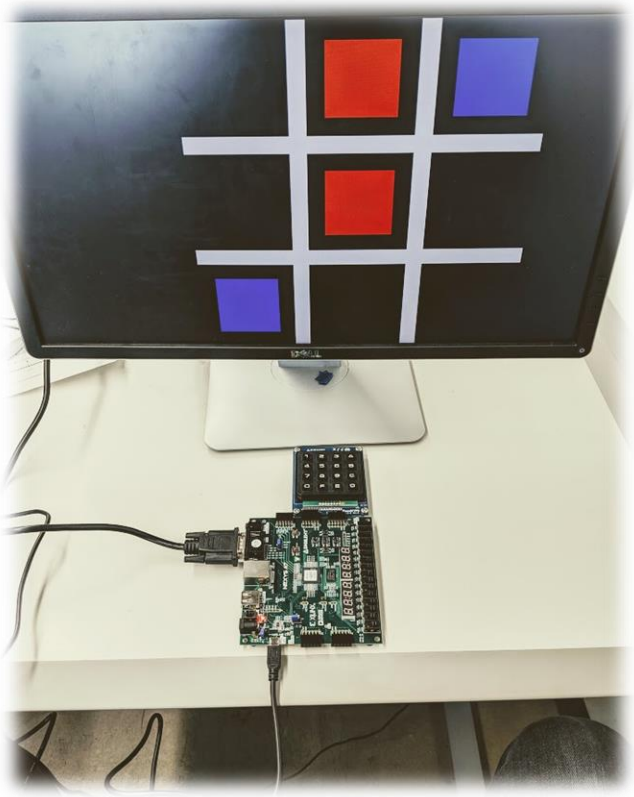


III. EXPERIMENTAL SETUP

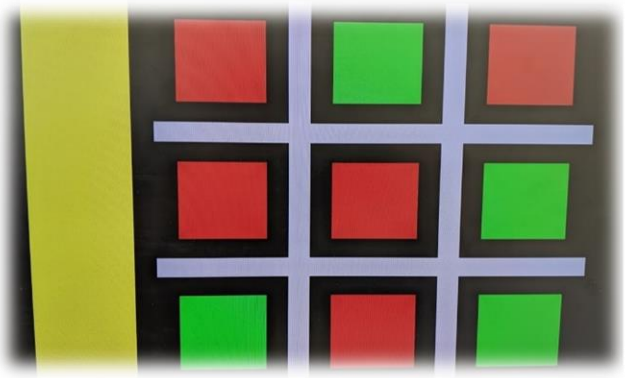
All testing was done using Vivado v2019.1. Because the VGA output doesn't reveal much about the inner workings of the system a second top file **tic-tac-toe testable** was included in simulation sources to allow use to verify internal processes. We frequently ran up to the maximum pin count of 110 when simulating and had to revise the test bench when validating different components. Modern monitors may switch away from the VGA input upon a reset but can switched back easily. As mentioned previously, the "Digilent Pmod KYPD: 16-button Keypad" is used for input. A standard VGA cable connects the board to any monitor.

IV. RESULTS

An image showing the working setup in action is shown below:



Below is another image with the AI switch on showing the screen more closely, after a tie:



V. Conclusions

A primary takeaway from this project is the complexity that can be involved even in simple problems. Tic-tac-toe is not overtly sophisticated but designing a system to implement it was exceedingly challenging.

The inflexibility of our VGA Driver is something that could be improved. The driver does not scale well and measuring out pixel lengths and widths for every board element is extremely tedious. The project itself allowed us to fully understand the VGA protocol. We also wish we could have implemented a running win/loss tracker so players could assess their tic-tac-toe skill. Another misstep was the AI algorithm. First, we tried to make it completely asynchronous, and ended up making the move queue a kind of latch that changed instantly when the optimal move vector did. However, this was unreliable and stopped working frequently. So, we changed the queue process into a state machine, but then our single "done" state wasn't long enough so we extended it to three "done" states. Once we got it functional, we realized it didn't perform well because it was biased against the center and corner cells. Center and corner cells involve 3 and 4 overlapped win combinations not just 2 like the side cells which inflates their cell values. To fix this we arbitrarily subtracted '5' from corner cell rankings and '10' from the center cell rankings. After adjusting the AI performs much better winning consistently. Although, since the AI uses a counter to decide between equivalent choices prejudice toward certain choices arise. Because the counter is frequently reset to '0' when the number of move choices changes, we believe there is a bias toward low address cells. Reforming the selection method with a different technique for randomization would be preferred. Also, this project highlights the advantages of parallel arithmetic. All the win sums, and cell values, and even the lowest value are computed in parallel and simultaneously. This is very fast and very powerful. Instead of programming an algorithm for the AI, because of this parallel nature of the FPGA we could have implemented a neural net and taught the AI how to win. This next step is something outside the scope of ECE 2700 but because of the rigor we employed when designing the project, we feel it is a step that is not far out of reach.

References

[1] Digilent, "Pmod KYPD," Pmod KYPD [Reference.Digilentinc]. [Online]. Available: https://reference.digilentinc.com/reference/pmod/pmodkypd/start. [Accessed: 25-Nov-2019]

[2] Tinyvga.com. (2019). VGA Signal 640 x 480 @ 60 Hz Industry standard timing. [online] Available at: http://tinyvga.com/vga-timing/640x480@60Hz [Accessed 25 Nov. 2019].

[3] Diligent, "Nexys A7 Reference Manual," Nexys A7 Reference Manual [Reference.Digilentinc]. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual. [Accessed: 26-Nov-2019]