ECE 350: Digital Systems
Professor John A. Board
Final Project Technical Report

Tic-Tac-Toe

Max Tobin and Andy Ko

April 21, 2026

Duke University

maxwell.tobin@duke.edu & andy.ko@duke.edu

Abstract

In this project, we designed and implemented a complete hardware–software system that realizes an interactive Tic-Tac-Toe game on an FPGA using a custom 5-stage pipelined processor. The system integrates physical user input, memory-mapped I/O, custom processor instructions, and VGA display output to enable real-time gameplay between a human player and an AI opponent.

Fig. 1 Full System

The overall system architecture consists of four primary components: (1) a modified pipelined processor capable of executing both standard and custom instructions, (2) a memory subsystem for storing board state and game variables, (3) a physical input interface consisting of a 3×3 button array with pull-down resistor networks, and (4) a VGA output system that renders the game visually. The Tic-Tac-Toe board is stored in data memory addresses 0–8 (0 = empty, 1 = human, 2 = AI), with additional memory-mapped addresses used for input (button presses) and output (game status signaling).

The processor executes an assembly program that implements the full game loop, including input polling, move validation, board updates, win and draw detection, and AI decision-making. The AI is implemented using a no-depth-limit minimax algorithm, which evaluates possible move sequences and assigns scores of +10 (AI win), −10 (human win), and 0 (draw), enabling optimal or near-optimal decisions within the processor's constraints.

To improve efficiency and reduce assembly complexity, we extended the processor with two custom R-type instructions: MOVEMASK and CHECKWIN. MOVEMASK converts the board state into a 9-bit player mask, while CHECKWIN evaluates all winning configurations using bitwise operations in constant time. These instructions eliminate repeated loops and branching in assembly and are integrated into the execute stage using existing pipeline mechanisms such as forwarding and hazard control.

The input system uses nine push buttons with a pull-down resistor network for stable signals, while the VGA controller renders the board in real time. Display updates occur only after finalized moves to prevent visual glitches during computation. Testing was conducted at multiple levels, including processor simulation, custom instruction validation, input testing, and full-system integration, with focus on minimax behavior, pipeline hazards, and display synchronization. Overall, this project demonstrates a complete hardware–software co-design, combining processor architecture, assembly programming, hardware interfacing, and algorithmic optimization to achieve efficient and responsive gameplay on an FPGA platform.

Inputs and Outputs

Inputs: The primary user input system consists of a 3×3 array of physical push buttons, where each button corresponds directly to one cell of the Tic-Tac-Toe board (indices 0–8). This mapping allows the user to select a move by pressing the button associated with the desired board position.

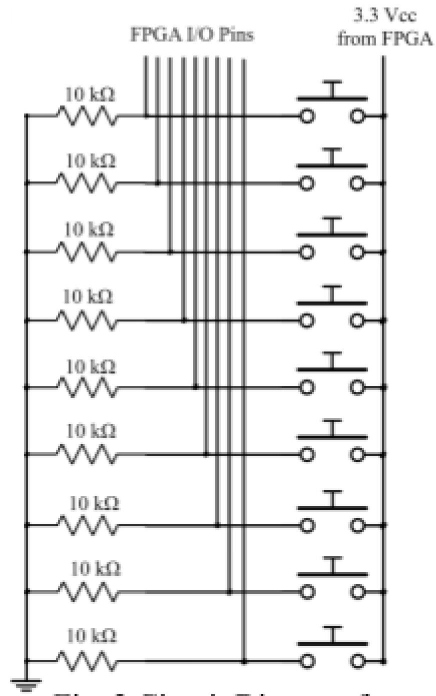

Fig. 2 Circuit Diagram for Buttons

Each button is connected to the FPGA through a pull-down resistor network, ensuring a stable default logic level of 0 when the button is not pressed. When a button is pressed, the corresponding signal transitions to logic high (1). This design prevents floating inputs, which could otherwise lead to unpredictable behavior due to noise or undefined voltage levels.

From a system perspective, the button inputs are exposed to the processor via memory-mapped I/O. Specifically:

- Address 4080 is reserved as the input register
- Reading from this address returns:
    - A value from 0–8 indicating which button is currently pressed
    - A value of 9 indicating that no button is pressed

The assembly program continuously polls this address during the input phase of the game loop. This polling mechanism allows the processor to detect user interaction without requiring interrupts, simplifying control flow within the assembly program.

To ensure reliable operation and clean user interaction, additional input-handling logic was implemented in software:

- A wait loop ensures that the system does not proceed until a valid button press is detected
- A release detection loop prevents repeated inputs when a button is held down
- Input values are validated before being used to update the board state

The input system was physically constructed using a protoboard, where all nine buttons were soldered into place. Each button branch includes a direct connection to the FPGA input pin, pull-down resistor (≈10 kΩ) tied to ground, and a shared 3.3V supply for the active-high signal. To improve durability and reliability all connections were soldered rather than breadboarded, hot glue was applied at connection points to provide strain relief, and zip ties were used to bundle and organize wires by button group.

This design minimizes the risk of wire detachment and reduces noise or accidental shorting between adjacent connections. The organized wiring layout also made debugging significantly easier, as each signal path could be traced visually. A reset button on the FPGA board is used to restart the processor and reinitialize the game.

Outputs: The system's primary output is a VGA display, which provides real-time visual feedback of the game state. The VGA output renders the 3×3 Tic-Tac-Toe grid, player moves (human and AI), and updated board state after each turn. The VGA controller is connected to the processor through the wrapper module and operates based on values stored in data memory and control registers.

In addition to the board state stored at memory addresses 0–8, the system uses a dedicated memory-mapped output register: address 4081 is assigned as the game status register. The assembly program writes to this address to indicate the outcome of the game:

- 1 → Human win
- 2 → AI win
- 3 → Draw

This register allows the VGA system to detect when the game has ended and update the display accordingly. By separating board state from game status, the design maintains a clean interface between computation and visualization. The VGA display is driven by a VGA controller module, which generates horizontal and vertical synchronization signals (hSync, vSync) along with RGB output signals. The controller continuously scans the display and determines pixel values based on the current game state.

The processor indirectly controls the display by updating board memory (which determines what is drawn in each cell), and game status (which determines whether the game is ongoing or complete). A key design decision was to decouple computation from display updates. Early in development, intermediate board states generated during the minimax algorithm were being written to memory and displayed, causing visual glitches where multiple X and O placements appeared simultaneously. This issue was resolved by ensuring that temporary minimax states are used only internally, and the display is updated only after a final move decision is made. This separation ensures a stable and intuitive user experience.

Processor Modifications

To support efficient Tic-Tac-Toe gameplay and AI evaluation, we extended the 5-stage pipelined processor with two custom instructions: MOVEMASK and CHECKWIN. These instructions were designed to offload repeated board-evaluation logic from assembly into hardware, significantly reducing instruction count, branching complexity, and execution time, especially within the minimax algorithm.

Both instructions were implemented as custom R-type ALU operations, integrated into the processor's decode and execute stages, and supported by the existing pipeline hazard handling mechanisms (forwarding, stalling, and flushing).

Instruction	Opcode (ALU op)	Type	Operation
movemask $rd, $rs, $rt	00000 (01000)	R	$rd = bitmask of board positions where board[i] == player
checkwin $rd, $rs	00000 (01001)	R	$rd = 1 if winning pattern detected, else 0

MOVEMASK Instruction: The MOVEMASK instruction was introduced to eliminate repeated loops over the board in assembly, which would otherwise require iterating through all nine memory locations with comparisons and branching—an especially inefficient process within the minimax algorithm where board evaluation occurs frequently. Instead, MOVEMASK converts the board state into a compact 9-bit mask, enabling constant-time bitwise reasoning about board ownership. The instruction takes as input a player identifier in register $rs (1 for human, 2 for AI) and operates on the Tic-Tac-Toe board stored in memory addresses 0–8, producing a 9-bit mask in register $rd where bit i is set to 1 if board[i] matches the player and 0 otherwise. Conceptually, this corresponds to initializing a mask to zero and iterating through all board positions, setting bits for matching entries; however, in this design, the entire operation is implemented directly in hardware rather than through loops and branching. As a result, the mask is computed in a single instruction execution, eliminating repeated memory accesses and significantly reducing control flow complexity in the assembly program.

Implementation in processor.v: The MOVEMASK instruction is detected in the decode stage as a custom R-type instruction using its ALU operation field and routed to a custom execution path within the ALU. In the execute stage, the processor accesses the board values, compares each entry to the player identifier, and constructs a 9-bit mask using shift and bitwise OR operations, where each bit indicates whether a board position belongs to the player. The result is then selected through the ALU result multiplexer, passed through the pipeline, and written back in the writeback stage like a standard ALU operation. MOVEMASK integrates cleanly into the existing pipeline by leveraging forwarding and hazard detection; its result can be immediately used by subsequent instructions via X/M and M/W forwarding paths, while load-use hazards trigger stalls when necessary. Because it behaves like a standard ALU operation and produces its result in the execute stage, no additional bypass paths or structural changes were required.

Example: Given Board: [2, 0, 1,
2, 1, 0,
0, 0, 2]

AI player = 2

Hardware computes: 1 << 0 = 000000001
1 << 3 = 000001000
1 << 8 = 100000000

Final output: $rd = 100001001

This mask is then passed to the CHECKWIN instruction

CHECKWIN Instruction: The CHECKWIN instruction eliminates the need to manually evaluate all eight winning conditions in assembly, which would otherwise require multiple loads, comparisons, and branches for each row, column, and diagonal. Instead, it performs this entire operation as a single constant-time instruction. CHECKWIN takes as input a 9-bit player mask in register $rs, generated by the MOVEMASK instruction, and outputs a single-bit result in register $rd, which is set to 1 if a winning configuration exists and 0 otherwise. The instruction operates by comparing the mask against all possible winning patterns using bitwise operations, specifically checking whether (mask & win_pattern) == win_pattern for each case. If any comparison evaluates to true, a win is detected. This logic is executed within the execute stage, where all eight winning patterns are evaluated in parallel, allowing the processor to determine the result in a single instruction cycle without requiring loops or branching in assembly.

000000111 row 0
000111000 row 1
111000000 row 2
001001001 col 0
010010010 col 1
100100100 col 2
100010001 diagonal
001010100 diagonal

Implementation in processor.v: The CHECKWIN instruction is decoded in the decode stage as a custom ALU operation and routed to the execute stage, where the processor takes the 9-bit player mask as input and compares it against all eight predefined winning patterns. This is performed by computing bitwise AND operations between the mask and each pattern and checking whether the result matches the pattern; if any comparison is true, the output is set to 1, otherwise it is 0. This logic is implemented combinationally within the processor, allowing the result to be computed in a single cycle, after which it is passed through the ALU result multiplexer and written back in the writeback stage like a standard ALU instruction. CHECKWIN integrates efficiently into the pipeline by leveraging existing forwarding and control mechanisms; since it often consumes the output of MOVEMASK, forwarding paths allow the mask to be used immediately without waiting for writeback. Additionally, because its result is frequently used in branch instructions such as beq or bne, the processor's branch resolution in the execute stage and corresponding flush logic ensure correct control flow, and no structural hazards are introduced since CHECKWIN shares existing ALU resources and fits cleanly within the datapath.

The addition of MOVEMASK and CHECKWIN significantly improved efficiency by moving board-evaluation logic into hardware, eliminating repeated loops and reducing branching in the assembly program. This enabled constant-time win detection and simplified the minimax implementation, particularly in performance-critical sections. As a result, instruction count was reduced, leading to faster execution and cleaner control flow. Without these instructions, minimax would require nested loops and repeated board scanning, making it less efficient and harder to implement correctly.

Assembly Program and System Integration

The assembly program implements the full Tic-Tac-Toe game logic, including input handling, board updates, win detection, and AI decision-making. The board is stored in data memory addresses 0–8, where each location represents a cell (0 = empty, 1 = human, 2 = AI). Additional memory locations are used for the stack, intermediate values, and memory-mapped I/O. The program begins by initializing the stack pointer and clearing the board using a clear_board routine, which ensures a consistent starting state before entering the main game loop.

The main loop continuously polls the memory-mapped input address (4080) to detect button presses. When a valid input is detected, the selected index is passed to the place_move function. To ensure correct behavior, the program:

- Waits for a button press before proceeding
- Prevents repeated inputs when a button is held
- Rejects invalid moves (occupied cells)

This polling-based design simplifies control flow while maintaining responsive input handling.

The place_move function validates and updates the board for both human and AI moves. After each move, the program checks for a win or draw condition. Win detection uses the custom CHECKWIN instruction, which evaluates all winning patterns in constant time using bitwise operations. Draw detection is performed by checking whether all board positions are filled. If a terminal state is reached, the result is written to the memory-mapped status register (4081) for display.

The AI is implemented using a no-depth-limit minimax algorithm (It'll continuously run minimax until all the way until the end of game is reached). The find_best_move function evaluates all valid moves by temporarily placing an AI piece and calling the minimax function. Minimax recursively explores future states and assigns scores:

- +10 for an AI win
- -10 for a human win
- 0 for a draw

Each move is simulated, evaluated, and then undone before testing the next option. The best move is selected and applied to the board. Because minimax uses nested function calls, correct stack management (saving return addresses and restoring values) is essential for proper execution.

The system integrates multiple hardware and software components:

- Processor (processor.v) executes the assembly program and handles custom instructions
- Instruction Memory (ROM.v) stores the compiled program (final.mem)
- Data Memory (RAM.v) stores board state, stack data, and I/O values
- Button Input (button_input.v) maps physical button presses to memory-mapped input
- VGA Controller (vga_controller.v) renders the game state
- Wrapper (Wrapper.v) connects all modules together

The assembly code is written in .s format and converted into a .mem file, which is loaded into instruction memory. During execution, button inputs update memory, the processor computes game logic, and the VGA controller displays the resulting board state. This integration enables real-time interaction between hardware and software components.

Testing Plan & Results

Testing for this project was conducted iteratively across multiple levels of the system, beginning with isolated processor and assembly verification and progressing to full hardware–software integration. This staged approach allowed us to identify and isolate issues early, ensuring that each subsystem functioned correctly before combining them into a complete interactive system.

Processor and Assembly Testing: Initial testing focused on verifying the correctness of the processor and assembly program within the Vivado simulation environment. This stage was critical for ensuring that the underlying execution model behaved as expected before introducing hardware inputs or VGA output. Specifically, we verified that:

- The processor correctly fetched and decoded instructions
- Data memory updates occurred at the correct addresses
- Board values were stored and retrieved properly from memory locations 0–8
- Branch and jump instructions reached the correct labels
- Function calls and returns behaved correctly through the use of register r31
- Stack operations functioned properly during nested calls

This level of testing was especially important because the minimax algorithm relies heavily on recursion. Any issues with saving return addresses, restoring arguments, or undoing temporary board states could cause incorrect evaluations or infinite loops. By validating these behaviors early, we ensured a stable foundation for more complex functionality.

Custom Instruction Testing: After confirming baseline processor functionality, we tested the custom instructions MOVEMASK and CHECKWIN independently. These tests were designed to validate both correctness and integration within the pipeline.

For MOVEMASK, we provided known board configurations and verified that the generated bitmask correctly reflected player positions. For CHECKWIN, we systematically tested all eight possible winning configurations (three rows, three columns, and two diagonals). Each pattern was validated to ensure that the instruction correctly returned a win condition only when the mask fully matched a winning configuration. These tests confirmed that:

- The custom ALU logic in the execute stage produced correct results
- The instruction decoder properly identified custom ALU operations
- The outputs integrated correctly with subsequent instructions

Button Input Testing: Once the processor and assembly logic were verified, we tested the physical input system. Each of the nine buttons was evaluated individually to ensure reliable and accurate signal transmission. The testing focused on confirming that each button:

- Produced a clean logic high signal when pressed
- Returned to a stable logic low when released
- Mapped to the correct board index in memory
- Did not interfere with neighboring button signals

The pull-down resistor network proved essential during this phase. Without it, floating inputs caused inconsistent readings and false triggers. After implementing the resistor network and improving physical wiring, input behavior became stable and repeatable.

VGA Testing: The VGA output system was tested using controlled board states to ensure that the display accurately reflected memory values. By manually setting board memory, we verified that the correct visual output was produced. We confirmed that:

- Empty board states displayed correctly
- Human moves appeared in the correct locations
- AI moves appeared after computation
- End-game states (win or draw) were displayed appropriately

This phase helped isolate display-related issues from logic errors. If incorrect visuals appeared, we could determine whether the issue originated from memory updates, assembly logic, or VGA rendering.

Full-System Testing: After validating each subsystem independently, we conducted full-system testing to evaluate end-to-end functionality. This involved running the complete gameplay loop using the physical button interface and VGA display. Each test cycle included:

- Pressing a physical button to select a move

- Validating and placing the human move
- Checking for a human win condition
- Executing the minimax algorithm for AI decision-making
- Placing the AI move
- Checking for an AI win or draw condition
- Repeating the loop until game completion

Full-system testing exposed several critical issues, including display glitches caused by intermediate minimax states, pipeline timing issues introduced by custom instructions, and hardware reliability concerns. These findings directly informed the fixes described in the Challenges section, such as separating computation from display updates, verifying pipeline hazard handling, and improving physical wiring.

Challenges

Display Glitching During Minimax Execution: One of the first major issues encountered was a visual glitch where all board cells appeared to rapidly flash both X and O symbols during gameplay. Instead of displaying a single valid move at each turn, the board appeared to cycle through multiple possible states, creating an unstable and confusing user experience.

The root cause of this issue was the implementation of the minimax algorithm. During execution, minimax explores all possible future game states by temporarily writing moves to the board in memory. Because the VGA display was directly tied to the board memory, these intermediate states were being rendered in real time. As a result, every simulated move within the minimax search was briefly displayed, even though those states were not part of the actual gameplay.

To resolve this issue, the system was redesigned to separate computation from output. Temporary board states generated during minimax were restricted to internal computation and no longer directly affected the display. The display was updated only after the AI selected a final move. Additionally, the introduction of the MOVEMASK and CHECKWIN instructions helped isolate board evaluation logic from display updates, further stabilizing the visual output. This change ensured that only valid, finalized game states were rendered on the VGA display.

Pipeline Hazards with Custom Instructions: After integrating the MOVEMASK and CHECKWIN instructions into the processor, a second major issue emerged in the form of intermittent glitches and incorrect behavior during execution. These issues were not present before the addition of the custom instructions, indicating that the problem was related to how these instructions interacted with the pipeline.

The underlying cause was identified as pipeline hazards, specifically data hazards where instruction results were being used before they were fully available. Since both MOVEMASK and CHECKWIN produce results in the execute stage and are often used immediately by subsequent instructions (such as branches or further computations), incorrect timing could lead to stale or invalid values being propagated through the pipeline.

To address this, we relied on and carefully verified the processor's existing hazard control mechanisms, including forwarding and stalling. Forwarding paths from the X/M and M/W pipeline stages ensured that recently computed values could be used without waiting for full writeback. In cases where forwarding was insufficient, the processor's load-use hazard detection logic inserted stalls to delay execution until the correct data was available. Additionally, instruction sequencing in the assembly code was adjusted to ensure proper timing between dependent instructions. These fixes ensured that MOVEMASK and CHECKWIN operated correctly within the pipeline, eliminating the glitches and restoring stable execution.

Possible Improvements

AI and Algorithm Adjustability: One major area for improvement is the AI implementation. The current system uses a no-limit minimax algorithm, which provides reasonable gameplay but results in the human player basically never being able to achieve a win. Allowing the player to decrease the search depth through an adjustable nozzle would allow the AI to make weaker decisions, which would allow the user to win against the AI.

Additionally, a more advanced approach could involve replacing minimax with a machine learning–based model trained on Tic-Tac-Toe game data. This would shift decision-making from rule-based logic to learned behavior, potentially allowing for faster move selection and more dynamic gameplay.

Adjustable Difficulty: Improvements could also be made to the physical construction of the system. Refining wire sizing and routing would allow for a more compact and organized enclosure. While the current design uses soldering, hot glue, and zip ties for stability, a more polished version could incorporate a custom PCB or a dedicated casing to improve durability and aesthetics.

Multi-System and Networking Capabilities: A more advanced extension of the project would be the addition of a distributed multiplayer mode. This could be implemented by connecting two FPGA systems wirelessly, allowing two users to play against each other in real time. Such a feature would introduce challenges in communication protocols and synchronization but would significantly expand the functionality of the system.

Project Media

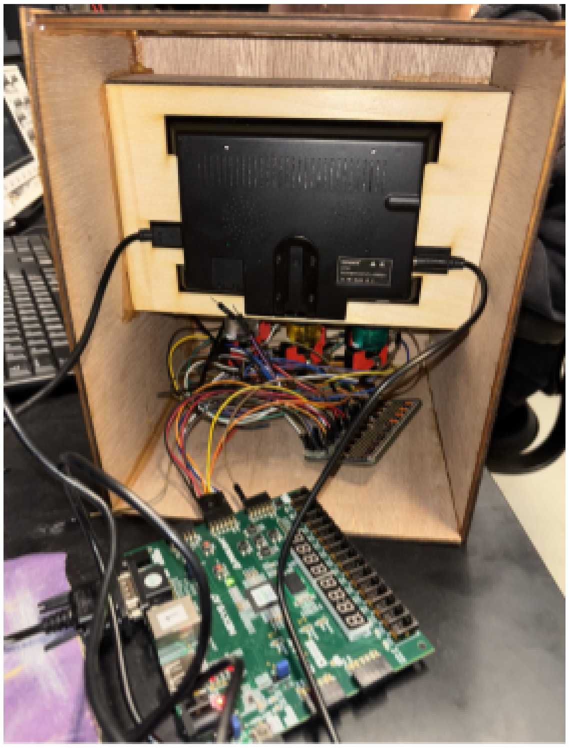

Fig. 3 Full Wiring Integration with FPGA with glue and zip ties for strain relief

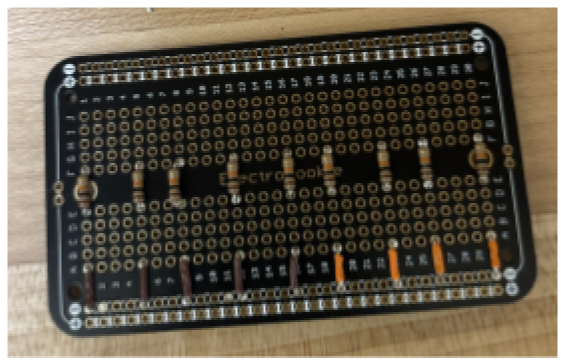

Fig. 4 Soldering of Pull-Down Resistors

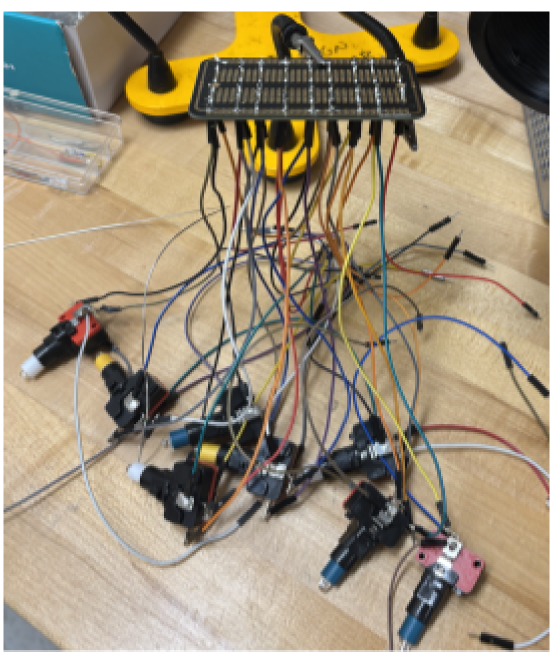

Fig. 5 Fully Soldered Protoboard

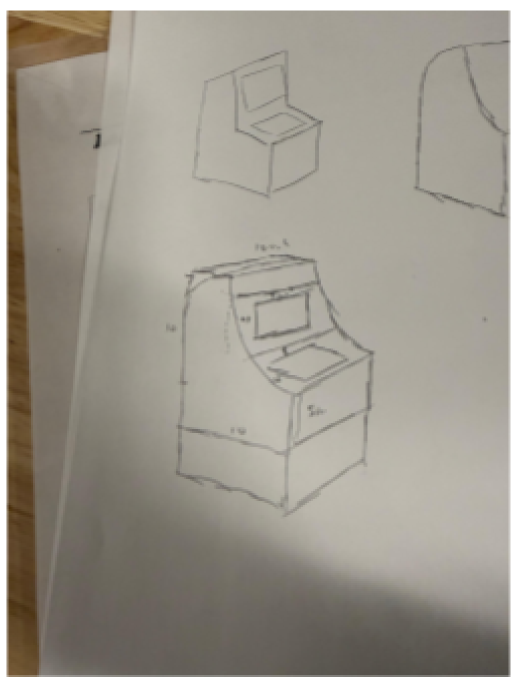

Fig. 6 Initial Sketch of Design