ECE 350 Final Report

Matthew Bloom (mlb126), Connor Wells (cjw84)

Git Repository: https://gitlab.oit.duke.edu/cjw84/airhockey.git

Tabletop Air Hockey

Design and Specifications

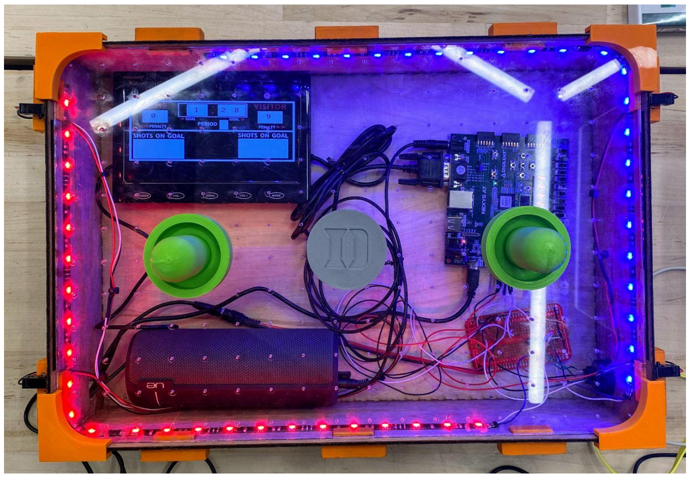


Figure 1: Top-down view of the completed table.

For our project, we created a tabletop-sized air hockey table. Our physical hardware consists of:

- A laser-cut plywood box designed using CAD tools with slots for goals and supports
- 3D-printed paddles, supports, corner pieces, and a Duke-themed puck
- Laser-cut acrylic sheet for the playing field with grid patterned holes for airflow

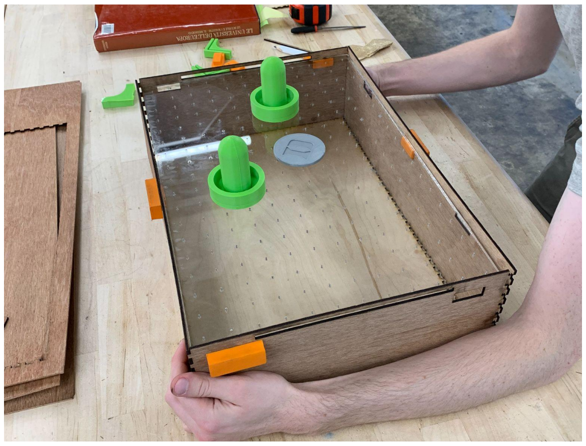


Figure 2: Construction of the physical frame for our design.

From a digital system perspective, we combined our processor's ability to run MIPS-like assembly code and interface with sensor inputs and outputs to implement the following features:

- Automatic detection of goal scoring
- Automatic display for player scores and countdown timer
- LED lights and a buzzer activated by a goal being scored
- User input to reset the score and time in order to start a new game (0-0, 2 minutes)

We utilized the Fusion 360 CAD tool to design all of the 3D-printed and laser-cut parts. Our processor as well as additional modules and modifications used to implement input and output were created using the Verilog hardware description language.

System Input/Output

The system's inputs were used to sense when a goal had been scored and to reset the game. They consisted of the following components:

- Two sets of 3mm IR LED break-beam sensors (transmitter-receiver pairs) positioned on the outside of the box at each goal as seen on the left and right sides in Figure 1
- Arcade button acting as a reset for all system modules

The outputs accomplished the goal of providing a display of the current game state and effects when someone scored, and included:

- VGA monitor acting as the scoreboard with automatic score updates and timing
- Two 12V non-addressable analog RGB LED strips for each player's side
- A buzzer in the form of a speaker connected via aux cable

All of the inputs and outputs (except for the VGA and audio) were connected to the 3.3V general purpose input/output (GPIO) headers located on the FPGA board. As seen in Figure 3, inputs are on the left GPIO header of the FPGA and outputs are on the right. The VGA and audio outputs are on the top left.

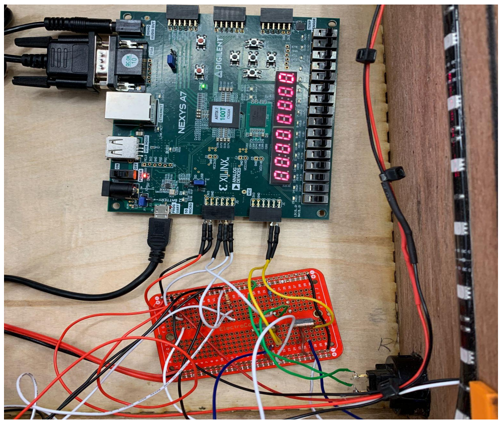


Figure 3: Circuit containing the I/O interface.

Circuit Diagram

As seen in the circuit diagram displayed in Figure 4, our project contained multiple circuits providing power and logical functionality to the:

- IR break-beam system (FPGA input)
- Reset system (FPGA input)
- LED strip system (FPGA output)
- Audio system (FPGA output)

It was physically constructed using a solderable breadboard connected to the various inputs and outputs of the FPGA.

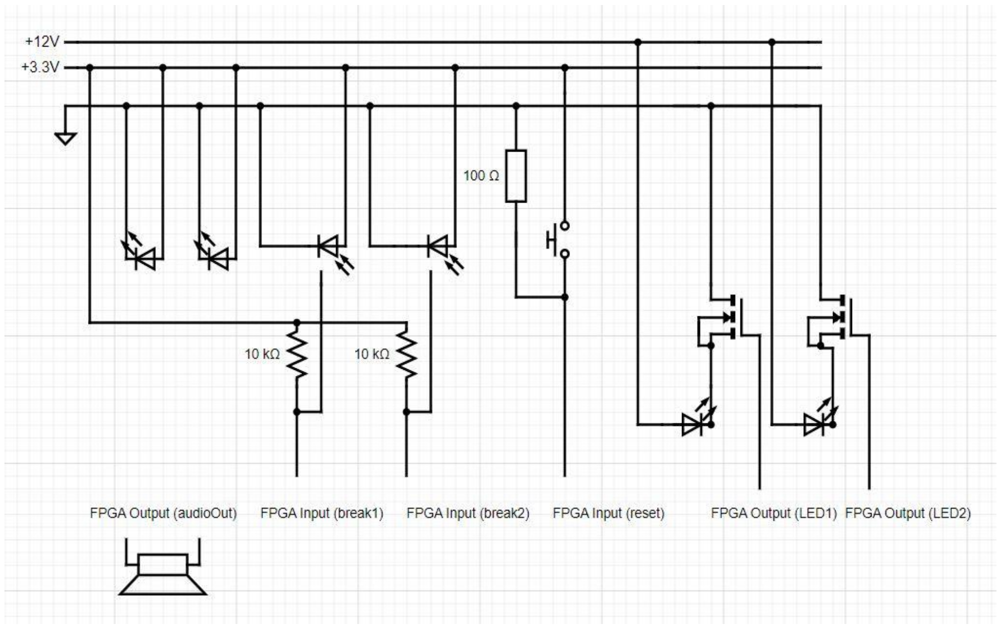


Figure 4: Circuit diagram with FPGA I/O indicated.

First, the IR break-beam system is composed of an IR LED transmitter and IR receiver. Both components are powered by 3.3V from the FPGA power supply. The IR receiver outputs a logical 1 to the FPGA input (break1 and break2) if the beam is complete and 0 if the beam is interrupted by an object. The logical output utilizes a pull-up resistor for the FPGA input to determine the digital output value of the sensor based on the voltage. Two IR LED transmitter and receiver pairs were necessary to detect goalscoring at each goal. Goal scoring is detected by the change in the logical output value of the sensor from 1 to 0 to 1 as the puck crosses the sensor.

Second, the reset system is composed of a button with a pull-down resistor. The pull-down resistor is required so that the logical input to the FPGA (reset) is 0 when the button is not pressed and 1 when the button is pressed. When the button is pressed, it acts as a short circuit pulling the logical input to FPGA to 1; when the button is not pressed, it acts as an open circuit pulling the logical output to FPGA to 0. The reset button will reset the timer and the players' scores for a new game.

Third, the LED strip system is composed of non-addressable LED strips capable of providing red, green, or blue color output based on which of the R, G, or B pins is grounded. The LED strip is powered by a 12V power supply. In order to use the 3.3V logical output of the FPGA (led1 and led2) to control the 12V LED strip power, the use of a power transistor was required. A logical output of 1 from the FPGA will switch the NMOSFET to the ON state and ground the LED strip to complete the circuit and turn on the LEDs. A logical output of 0 from the FPGA will switch the transistor to the OFF state, disconnecting the LED strip from ground and turning off the LED strip. When a goal is automatically detected, the appropriate LED strip output of the FPGA will be a logical 1 such that the opposing player's side is illuminated. Additionally, at the conclusion of the game, when either player's score reaches 10 points or the timer expires, both LED strip outputs will be a logical 1 such that both players' sides are illuminated.

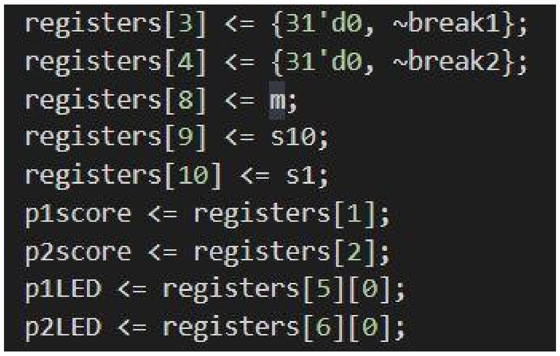
Fourth, the audio system is composed of the FPGA auxiliary audio out line to a speaker. When a goal is automatically detected, the audio out line will play a low frequency tone utilizing a pulse width modulation (PWM) mechanism. Additionally, at the conclusion of the game, when either player's score reaches 10 points or the timer expires, the audio out line will play the same tone.

Processor Modifications and Verilog Modules

To satisfy the design requirements, our project required the modification of the original processor by implementing or changing multiple Verilog modules:

- Register file module
- Countdown timing module
- VGA controller module
- Audio controller module

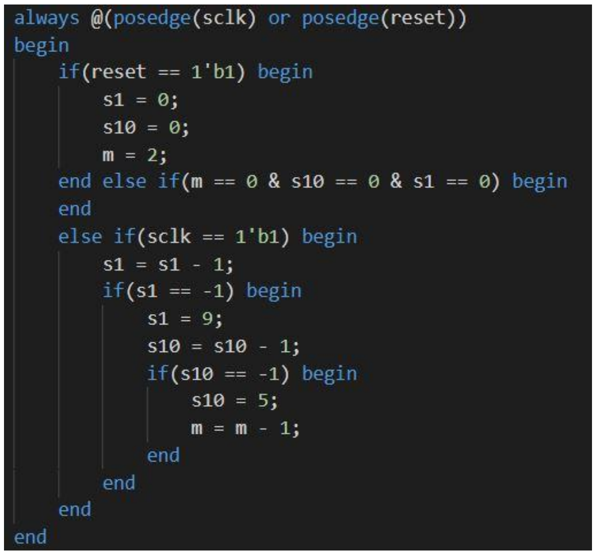
First, the processor is modified by directly writing registers in the register file with the logical values of the FPGA break-beam inputs and the timer digits every clock cycle such that the assembly code functions with the appropriate register values as seen in Figure 5. Additionally, the register values for player scores and LED states are output to the wrapper module via register wires every clock cycle such that the behavioral logic functions with the appropriate register values.



```
registers[3] <= {31'd0, ~break1};
registers[4] <= {31'd0, ~break2};
registers[8] <= m;
registers[9] <= s10;
registers[10] <= s1;
p1score <= registers[1];
p2score <= registers[2];
p1LED <= registers[5][0];
p2LED <= registers[6][0];
```

Figure 5: Verilog register file modifications.

Second, a countdown timing module is required to accurately track game time. The game time requires a minute value, tens second value, and ones second value. The countdown timing module utilizes the onboard FPGA 100MHz system clock and a clock divider to generate a 1Hz timer clock. The 1Hz clock is generated by decrementing a register with a value equal to 50 million every 100MHz clock cycle; when the register is 0, the 1Hz clock is negated. With a 1Hz timer clock, behavioral Verilog is used to appropriately decrement the minute value, tens second value, and ones second value every 1Hz clock cycle as seen in Figure 6. Additionally, the clock stops once all 3 values are 0. A module with similar logic was used to time the LEDs and sound to turn on for 4 seconds when a goal is scored.



```
always @(posedge(sclk) or posedge(reset))
begin
    if(reset == 1'b1) begin
        s1 = 0;
        s10 = 0;
        m = 2;
    end else if(m == 0 & s10 == 0 & s1 == 0) begin
    end
    else if(sclk == 1'b1) begin
        s1 = s1 - 1;
        if(s1 == -1) begin
            s1 = 9;
            s10 = s10 - 1;
            if(s10 == -1) begin
                s10 = 5;
                m = m - 1;
            end
        end
    end
end
```

Figure 6: Behavioral Verilog countdown timer logic.

Third, the VGA controller module is required to generate frames for the VGA monitor which displays the game scoreboard containing the countdown timer and player scores. The VGA controller utilizes the skeleton code provided in lab, whereby an image is converted to an image memory file with memory locations corresponding to memory locations in a separate color palette memory file. The VGA controller utilizes four RAM modules: two for background and sprite image memory and two for background and sprite image color palettes. The background image memory file is associated with the scoreboard and the sprite image memory file contains sprites for digits 0-9. The game scoreboard is a static background with boxes for dynamically updated sprites as seen in Figure 7.

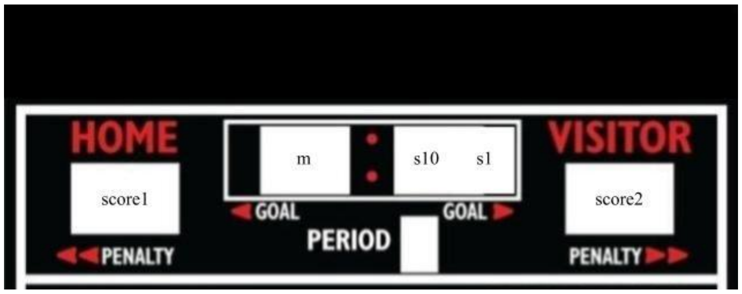
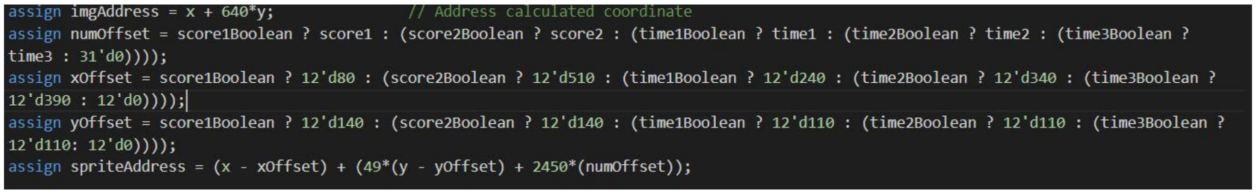


Figure 7: Game scoreboard and sprite regions.

The sprite regions are determined by checking if the current drawing pixel is within a particular x and y bound. Since each sprite region is associated with a dynamically changing sprite value, a corresponding offset (numOffset) is created to change what number is being read from the sprite memory file based on what region the current drawing pixel is within as seen in line two of Figure 8. Additionally, x and y offsets are associated with each region in order to align the starting address in the sprite memory file based on the current drawing pixel's x and y coordinates.



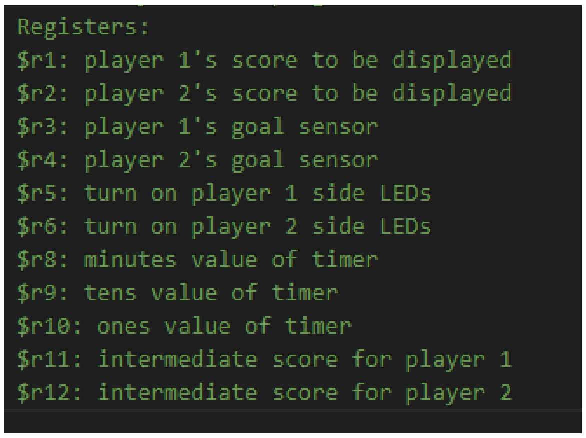
```
assign imgAddress = x + 640*y;                // Address calculated coordinate
assign numOffset = score1Boolean ? score1 : (score2Boolean ? score2 : (time1Boolean ? time1 : (time2Boolean ? time2 : (time3Boolean ?
time3 : 31'd0))));
assign xOffset = score1Boolean ? 12'd80 : (score2Boolean ? 12'd510 : (time1Boolean ? 12'd240 : (time2Boolean ? 12'd340 : (time3Boolean ?
12'd390 : 12'd0))));
assign yOffset = score1Boolean ? 12'd140 : (score2Boolean ? 12'd140 : (time1Boolean ? 12'd110 : (time2Boolean ? 12'd110 : (time3Boolean ?
12'd110: 12'd0))));
assign spriteAddress = (x - xOffset) + (49*(y - yOffset) + 2450*(numOffset));
```

Figure 8: Background image and sprite address memory location logic.

Fourth, the audio controller module is required to generate tones to be output to a speaker. We used the code provided and written during lab, with the modification that it now only outputs a PWM signal when an enable signal is high. We set this audio enable signal to be high when either of the LED outputs are on. Also, we changed the module to always index the memory file that determines the PWM frequency at the first and lowest frequency, producing a 261 Hz tone.

Assembly Code

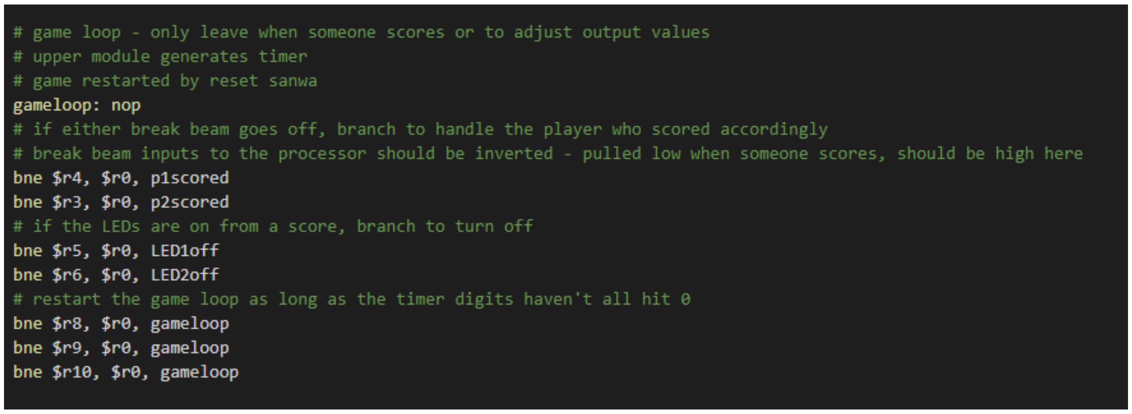
Our processor executed assembly language code with a MIPS-like ISA in order to update the score, turn on the LEDs and speaker, and end the game when a score limit of 10 is reached or the clock runs out. As discussed in the previous section, several of the register values depend on inputs to the system or other Verilog modules, while others control the outputs. Figure 9 summarizes the purposes of each register used in the code.



```
Registers:
$r1: player 1's score to be displayed
$r2: player 2's score to be displayed
$r3: player 1's goal sensor
$r4: player 2's goal sensor
$r5: turn on player 1 side LEDs
$r6: turn on player 2 side LEDs
$r8: minutes value of timer
$r9: tens value of timer
$r10: ones value of timer
$r11: intermediate score for player 1
$r12: intermediate score for player 2
```

Figure 9: A list of registers and their functions in the gamecontrol.s program.

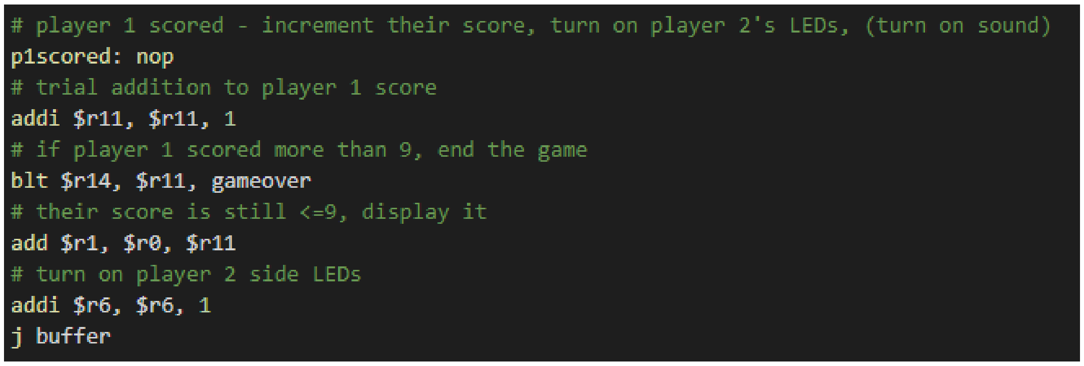
The code includes a basic game loop that polls the break beam sensors, turns off LEDs and sound if necessary, then restarts as long as the game time has not elapsed. When a beam is broken, the corresponding branch is taken, and if the LEDs are still on ($r5/6 = 1), the LED off branches return their corresponding register values to 0. Finally, as long as at least one of $r8-10 (which store the timer digits) is nonzero, the code branches back to the start of the loop.



```
# game loop - only leave when someone scores or to adjust output values
# upper module generates timer
# game restarted by reset sanwa
gameloop: nop
# if either break beam goes off, branch to handle the player who scored accordingly
# break beam inputs to the processor should be inverted - pulled low when someone scores, should be high here
bne $r4, $r0, p1scored
bne $r3, $r0, p2scored
# if the LEDs are on from a score, branch to turn off
bne $r5, $r0, LED1off
bne $r6, $r0, LED2off
# restart the game loop as long as the timer digits haven't all hit 0
bne $r8, $r0, gameloop
bne $r9, $r0, gameloop
bne $r10, $r0, gameloop
```

Figure 10: The game loop runs continuously until a score occurs or the game ends.

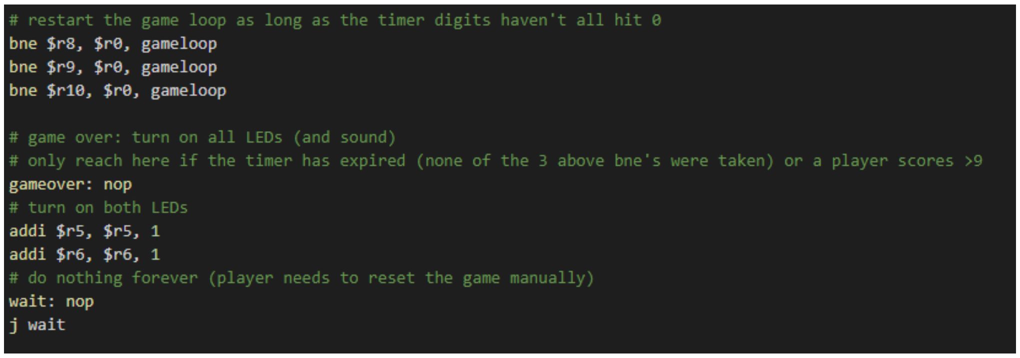
When either the p1scored or p2scored branches are taken, the program first performs a trial addition to $r11 or $r12 according to which beam was broken and therefore who scored - these act as intermediate registers to check if the player has just exceeded the score limit of 9. If they have, the blt is taken ($r14 is initialized to a value of 9) and the game ends. If they have not, their score is still 1 digit and can be displayed by moving the $r11/12 value to $r1/2, which are registers whose values are displayed on the VGA scoreboard. Finally, the register corresponding to the LED strip on the side of the player who was just scored *on* is set to 1, turning on the LEDs as well as the sound with some upper-level logic. Then, the code jumps to a buffer before returning to the game loop - more details on that in the following section.



```
# player 1 scored - increment their score, turn on player 2's LEDs, (turn on sound)
p1scored: nop
# trial addition to player 1 score
addi $r11, $r11, 1
# if player 1 scored more than 9, end the game
blt $r14, $r11, gameover
# their score is still <=9, display it
add $r1, $r0, $r11
# turn on player 2 side LEDs
addi $r6, $r6, 1
j buffer
```

Figure 11: Code section used to handle scoring - similar code exists for player 2.

The game reaches gameover and ends by either branching to gameover when a player scores more than 9 as discussed above, or when none of the branches at the end of the game loop are taken, which occurs when the timer hits 0:00. This state simply turns on the LED strips and sound by adding 1 to $r5 and $r6, then waits and executes nops, stalling the processor until the game restarts due to user input. $r5 and $r6 are assured to be 0 before this occurs due to the LED1/2off branches and the fact that the blt instructions are executed before turning on LEDs for a score.



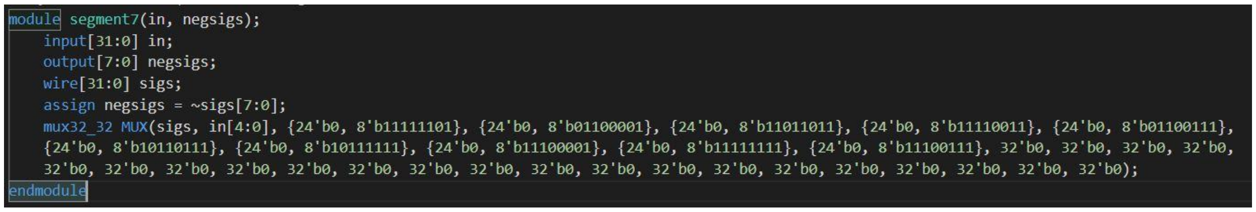
```
# restart the game loop as long as the timer digits haven't all hit 0
bne $r8, $r0, gameloop
bne $r9, $r0, gameloop
bne $r10, $r0, gameloop

# game over: turn on all LEDs (and sound)
# only reach here if the timer has expired (none of the 3 above bne's were taken) or a player scores >9
gameover: nop
# turn on both LEDs
addi $r5, $r5, 1
addi $r6, $r6, 1
# do nothing forever (player needs to reset the game manually)
wait: nop
j wait
```

Figure 12: The game over state code continues until the reset button is pressed.

Testing and Challenges

Throughout our project, we encountered many mechanical and digital challenges which we overcame by collaborative testing and solution generation. To debug and test our system, we needed to view the real-time values of sensors and registers during gameplay which we accomplished by implementing a 7-segment display module for the Nexys A7 hardware as seen in Figure 13. Additionally, for debugging requiring simultaneous viewing of multiple values, we hardwired our existing VGA controller module to display multiple debugging values.



```
module segment7(in, negsigs);
    input[31:0] in;
    output[7:0] negsigs;
    wire[31:0] sigs;
    assign negsigs = ~sigs[7:0];
    mux32_32 MUX(sigs, in[4:0], {24'b0, 8'b11111101}, {24'b0, 8'b01100001}, {24'b0, 8'b11011011}, {24'b0, 8'b11110011}, {24'b0, 8'b01100111},
    {24'b0, 8'b10110111}, {24'b0, 8'b10111111}, {24'b0, 8'b11100001}, {24'b0, 8'b11111111}, {24'b0, 8'b11100111}, 32'b0, 32'b0, 32'b0, 32'b0,
    32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0, 32'b0);
endmodule
```

Figure 13: Verilog implementation of the Nexys A7 7-segment display.

One issue that we addressed early on in the project was taking into account the timing characteristics of the beam break sensors when a goal was scored. We used an oscilloscope to read one of the receivers' outputs when scoring a puck, resulting in the waveform in Figure 14. We repeated this several times, and found that the signal was pulled low for about 40 ms, which at our processor speed would be equivalent to over a million cycles. Without proper debouncing, this would result in the score being incremented hundreds of thousands of times instead of just once each time a goal is scored.

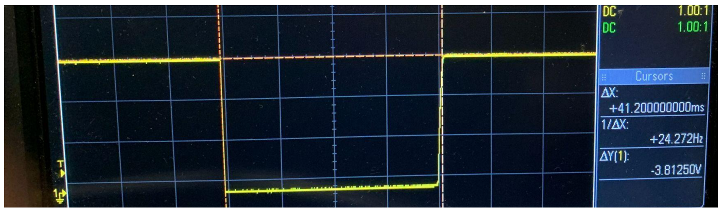
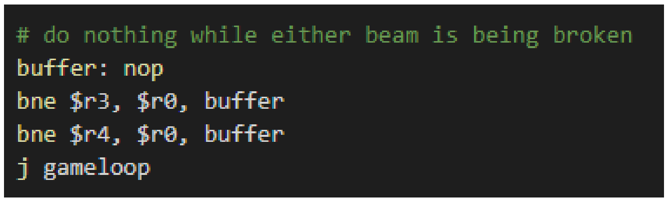


Figure 14: Oscilloscope display of the break beam sensor signal during goal scoring.

To solve this issue, we implemented the aforementioned buffer in the assembly code, pictured in Figure 15. It accomplishes debouncing of the beam break inputs by stalling, then using bne instructions so that as long as $r3 or $r4 (which correspond to the beam break inputs) is nonzero, the buffer restarts and stalling continues. Once the beam stops being broken, both registers are equal to 0 and the program returns to the game loop. This assures that the program only branches to increment the score once each time that the beam is broken.



```
# do nothing while either beam is being broken
buffer: nop
bne $r3, $r0, buffer
bne $r4, $r0, buffer
j gameloop
```

Figure 15: Assembly code “buffer” loop for debouncing.

Future Improvements and Features

If our project was not subject to time constraints, there are several features we would add including:

- Functional airflow
- Sensing additional game metrics like shots on goal or possession percentage

First, to add functional airflow to the tabletop air hockey table, we would need to add a compartment for housing a fan mounted on the underside of the existing base frame with a hole drilled for airflow. Holes would be drilled on the sides of the added compartment for air intake.

Second, to sense shots on goal or possession percentage, we would need to add a grid of Hall effect sensors to create a coordinate system below the playing field and install a magnet under the puck. Based on the location of the puck on either half of the playing field for a given number of clock cycles, one can maintain a register with a running total to extract the amount of time the puck is on either side of the playing field and therefore the possession percentage. This same system could also be used to track and display the live position of the puck. For tracking shots on goal, a similar sensing system could be used along with a register with the running total of the number of times the puck location is within a certain region or proximity to the goal.

Additional Photos/Appendix

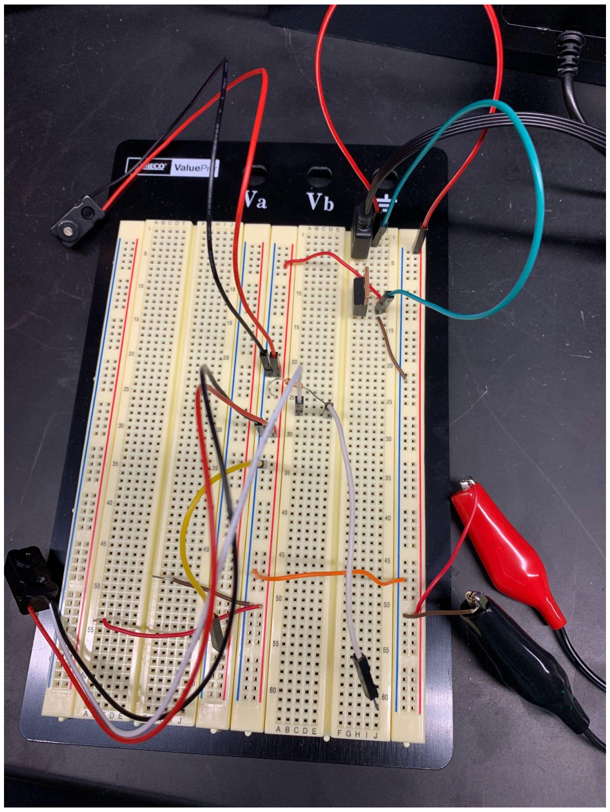


Figure 16: An early solderless breadboard prototype circuit used for testing the break beam sensor and LED functionalities. We used the oscilloscope and bench power supplies along with this circuit before soldering and using the wall power supplies.

Figure 17: Our smiling faces, and a side view of the table.