50.002 - 1D Computation Structures

Cohort 3 Group 5

Andre Hadianto Lesmana

Antonio Miguel Canlas Quizone

Sarthak Ganoorkar

Yee Celine

Introduction

Bing Bong Pong

This game was inspired by one of the earliest graphical game called Ping Pong. We adapted the 2D aspect of the game and implement it with a twist as 1D instead. In 1D, the ball only travels in a straight line. In our implementation of the game, we used only binary system for all the functions and features available on our game.

Under the given constraints, our team considered several game ideas, all of which involved the clock signal of the FPGA, carefully-designed patterns, and how the game could appeal to players aesthetically and in an engaging manner. Hence, we decided on 1D-Bitpong, our other ideas can be seen in the appendix.

Game Description

Bing Bong Pong is an interactive 2-player game that tests each player's ability to react to visual stimuli through a high intensity, fast-paced game.

The game consists of a start button, a return button for each player, and a 1 x 16 LED Matrix to represent the ball's trajectory track. The LED Matrix has four green LEDs, two on each side, that represents the safezone for each player. The buttons act as a 'virtual paddle' for each player, which can deflect the ball when pressed on the safezone.

Player 1 serves first and player 2 will have a short period of time to return the serve by pressing Player 2's button when the ball reach the safezone. As the rally progress, the ball's speed will change at random and when either players miss the safezone, the player will lose a life. Each players start with 3 lives.

Test Scenarios

Functional Test Scenarios:

1. Check that the ball can be shifted left or right, using the ALU’s SHL/SHR functions.
2. The ball can be returned after pressing a player’s button.
3. Not pressing the button as the ball reaches safezone.
4. Pressing the button before the ball reaches safezone
5. Pressing the button on the other player’s turn
6. Both players return consecutively >= 3 times to increase the ball’s speed randomly.

Gameplay Test Scenarios:

1. A player loses a life, if button was not pressed.
2. A player loses a life, if button was pressed too early, or not in his/her turn.
3. A player loses a game after losing all 3 lives
4. Game goes back to IDLE/MENU after a player loses.
5. Pressing the Start button while game is still ongoing.
6. Pressing the button when a player loses a life i.e. ball should start from loser’s side.

User Manual

Rules:

1. Each player starts with three lives, the first player to lose all lives loses the game.
2. Each player is assigned to a dedicated safe zone.
3. The player has to press the button when the ball enters their safe zone
4. The player loses a life if he/she fails to press the button when the ball is inside the safezone.
5. As the rally progresses, the ball will change to move at random speeds to increase the difficulty.

Design Overview

Game Design

The game table will have a 16 bit LED matrix to represent the movement of the ball. The LEDs harbored by the safezone are **red** and the rest are **green**. When the game starts, both buttons for player 1 and player 2 will be enabled and the 3 LEDs next to each button will light up which represents the current lives of each player. Player 1 gets to start the game by pressing the start button. When the button is pressed, the LED next to the safe zone will turn green and start shifting towards player 2. If player 2 presses his button when the ball is still green, player 2 loses a life, otherwise the led next to the safezone will turn green and start shifting towards player 1. Player loses when he or she loses all three lives. As the game progresses, the shifting speed will vary to increase the difficulty.

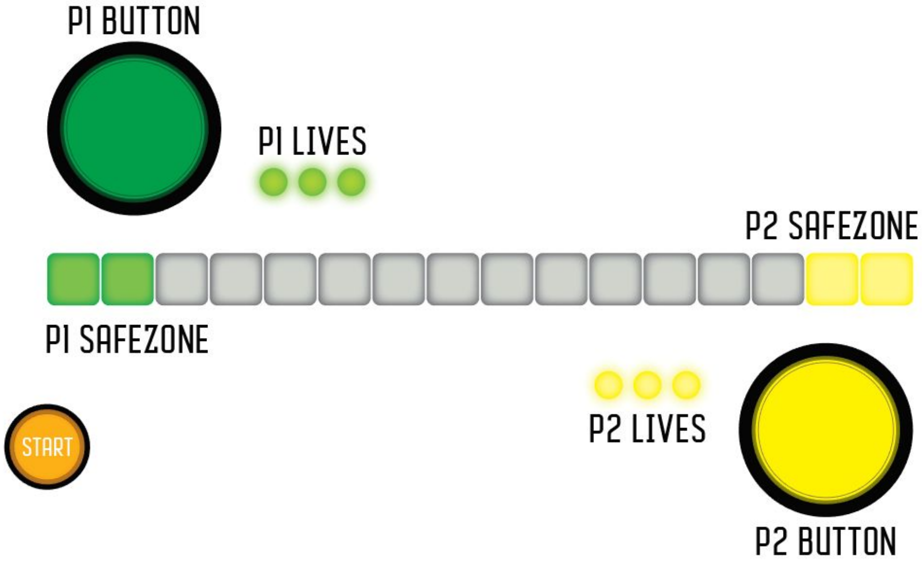


Figure 1

Design Overview

Components Budget

Materials	Size / Measure	Quantity	Costs	Total
Plywood	900 x 275 x 14 mm	2	S$8/ea	S$ 16
Plywood	874 x 15 x 5 mm	1	-	-
Arcade Buttons	100mm diameter	2	S$15/ea	S$ 30
Arcade Buttons	40mm diameter	1	-	-
Acrylic Sheet	900 x 275 x 3 mm	2	S$8/ea	S$ 16
Resistor	330 ohm	22	-	-
5mm LEDs	Green	4	-	-
5mm LEDs	Red	2	-	-
5mm LEDs	Yellow	12	-	-
10mm LEDs	Red	6	-	-
Breadboard	Normal size	1	-	-
Cables	Male to Female	66	-	-
Epoxy	-	1	-	-
Screws	10mm	36	-	S$2
Metal Sheet	150mm	2	-	-

Table 1

Design Overview

ALU Design

We used a 16-bit Arithmetic Logic Unit (ALU) to implement our desired gameplay.

1. SHL/SHR - The ball's position can be indicated for by a 1 in a string of 0s i.e. 0000001000000000. This string can be shifted to show the ball bouncing back and forth.
2. SUBC - Player 1 and player 2's lives are indicated with the last 2 bits (following binary value 0 up to 3) of a 16-bits long string respectively.
3. CMPEQ - We can verify if a player successfully returns the ball by comparing the player's input (if the button is pressed), the position of the ball and the player's safe zone.
4. CMPEQ - Each player's lives are also checked with CMPEQ to determine whether they have lose all their lives or not.

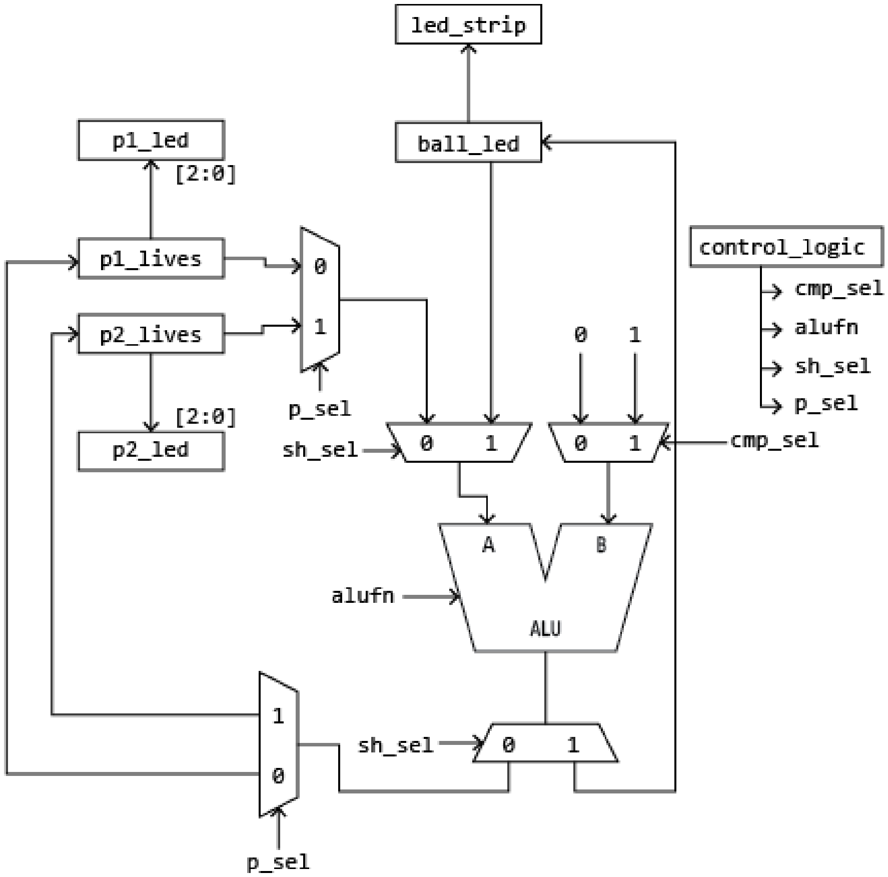
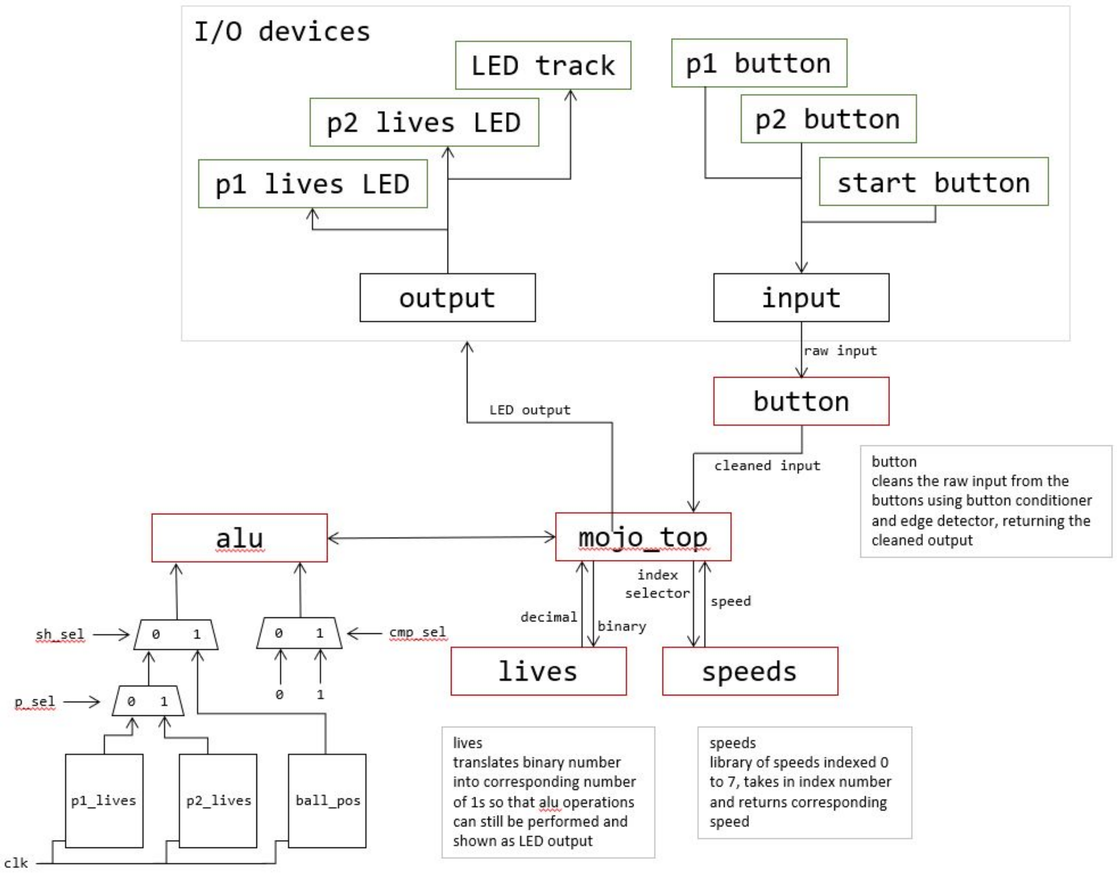


Figure 2



button
cleans the raw input from the buttons using button conditioner and edge detector, returning the cleaned output

lives
translates binary number into corresponding number of 1s so that alu operations can still be performed and shown as LED output

speeds
library of speeds indexed 0 to 7, takes in index number and returns corresponding speed

Figure 3

Steps in Building the Prototype

Software

We implemented each game function step by step to allow for easier debugging and testing.

1. Bounce LED

Instantiating components and connecting signals

We created variables by making dff so that we can read and write into them. `ball_pos` stores the position of the ball and will be used to encode which led on the track (16-bits wide to accommodate our 16-bit ALU functions) to light up.

Speed is incremented at every clk cycle (50MHz). We will use it to determine when to perform tasks like shifting of the ball to ensure the ball moves at a controlled paced.

Initialising values

In this implementation, we will only use shifter components of alu to shift the ball, so input a will be the `ball_pos` (to be shifted) and input b will be 1 (how much to shift by)

The ball position will be visualised on `io_led[1:0]`.

Setting condition to change states in FSM

Before the game starts, we initialise all values. We detect for button presses at every `clk` cycle, if player 1 presses the button, it goes to player 2's turn and vice versa.

Shifting ball at intervals

When speed overflows (occurs every 2^24 clk cycles = 0.336s), we will shift the ball using the alu and ball_pos is updated to be shown on `io_led`.

When it is player 1's turn, the ball moves left towards player 1 and vice versa.

Analysis

In this implementation, the ball changes direction no matter where the player presses the button, there is no safe zone or conditions for changing of states.

The game is stuck when the ball goes out of range. (the `ball_pos` value becomes 0 and there will be no more ball to shift)

2. Lives and Losing Condition

Safe Zone

We tried to implement the safe zone as a read only variable and checking it against the `ball_pos` (i.e. `safezone_p1` = 16b1100000000000000) However, because we were checking for button presses at every clk cycle, we decided it was easier to fix the values of the safe zone and check them as individual bits. (i.e. if `ball_pos[15]` == 1 | `ball_pos[14]` == 1)

Instantiating components and connecting signals

We renamed the states to be more representative of the state and implemented the button conditioner and edge detector.

Checking ball_pos with safe zone

In each of the players' turns, we check:

If the `ball_pos` = 0, which would mean that the ball went out of range while it was the player's turn meaning the player failed to press the button and the ball went out of range

If the button is pressed and

If `ball_pos` is in the player's safe zone by checking if the bit at the safe zone is = 1 then it will be the other player's turn or the player loses a life

Lives

We implemented the lives as dff that are 16 bits wide (to perform subtraction operation using the ALU). The output of the dff is also used as the input to the led to visualise the number of lives each player has.

However, due to the duality of the functions, we were unable to implement subtract since lives will be initialised as 16b111 to light up 3 LEDs for the lives and subtracting is performed in binary and will give 16b110 and subsequently 16b101.
Thus we decided to implement the loss of a life using shift operation of ALU.

Initialising values

We connected the led to visualise the lives of both players and initialised the lives of both players (when player 1 loses a life, `lives_p1` will be shifted left).

Implementing loss of lives at respective states

When a player loses a life, the state machine enters the respective lose states and checks if the player has any lives left, if the player has no more lives left, the state machine enters the game over state. Else, the player that lost a life gets to press to resume the game/get the ball to start moving again.

Game over state

To check which player lost the game, we check if player 1 has any lives left. If player 1 has no lives, he was the one that triggered the game over. Else, it must have been player 2 that triggered the game over. The players can replay the game by pressing the start.

3. Various Speeds

Attempt 1

We tried to implement an FSM embedded in another FSM, where the outer FSM controls the speed of the ball position and the inner FSM runs through the sequence to be performed in at the slowed speed

However, in our implementation of the game in a single speed, some of the states require sequences to be performed at every clock cycle (50MHz) i.e. checking for button presses and ALU calculations, and some sequences to be performed at various speeds i.e. shifting of the ball for varying ball speeds

There will be several duplicate modules of the embedded FSM

Attempt 2

We tried to use a counter to create a sclk signal (such as in Oka's example code for factorial)

#SIZE=1 because the components take in a 1 bit wide clock signal

#DIV implies that the counter increments once after every 2^24 clock cycles (50MHz) = 0.336s so the slowed clock is 2^24 x slower than the 50MHz clock.

We created a new counter for every ball speed we wanted to implement. However this implementation would have the same problems mentioned in attempt 1, there will be several duplicate FSM with the same functions, since a single FSM cannot be connected to 2 different clock signal.

Attempt 3

Instead of shifting the ball every time `ball_speed` overflowed, we used

- a dff to change the condition on `ball_speed` for the speed change of the ball
- another dff to determine when to change the condition on the `ball_speed`

Instantiating components and connecting signals

dff speeds stores the condition on `ball_speed`. It is 24 bits wide so that it can be compared with ball speed

dff `speed_counter` stores the condition to change speeds. It is 2 bits wide because we decided to change the speed after the ball is successfully returned 3 times thus only 2 bits is needed to count up to 3

Initialising values

Speed is initialised to 24h800000

Implementing condition to change speeds

`speed_counter` is incremented every time either player successfully returns the ball to the other player

Implementing the condition on ball_speed

`ball_speed` is incremented at every clock cycle (50MHz)

Instead of performing shifting of ball when ball_speed overflows, we perform the shift when `ball_speed` reaches a certain value, determined by dff speeds

We then reset the value of ball_speed to ensure that it does not continue to increment up to the max value and overflow

Setting the speeds

The value of speeds is changed when `speed_counter` counts up to 2b11 i.e. when the ball is successfully returned 3 times

In order to ensure the speed increases sequentially, we check the current value of speeds and reduce it to a smaller value

4. Modularising Circuit

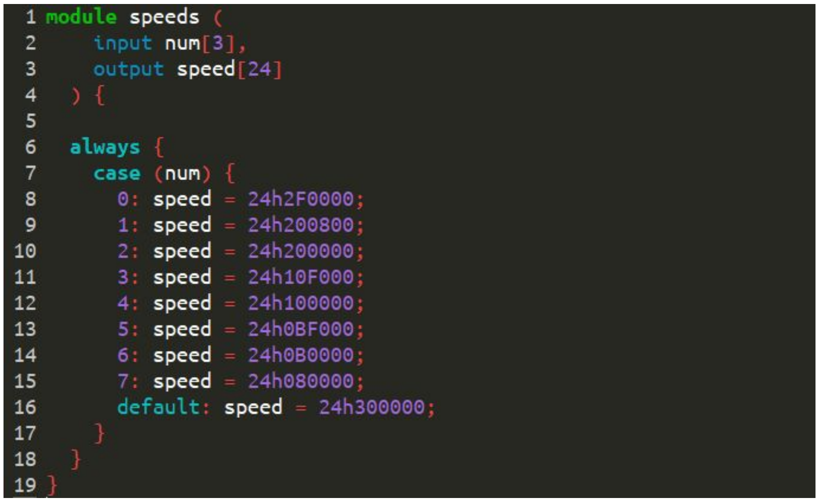
Speeds

We decided to create a module to control the speed/or randomise the speed

Speeds library takes in a 3 bit (to code up to 8 speeds) input to choose which speed outputs a 24 bit wide speed

Implementing speeds library

We attempted to make a pseudo random number using ball_speed which is rapidly incrementing and highly unpredictable to randomise speed of ball



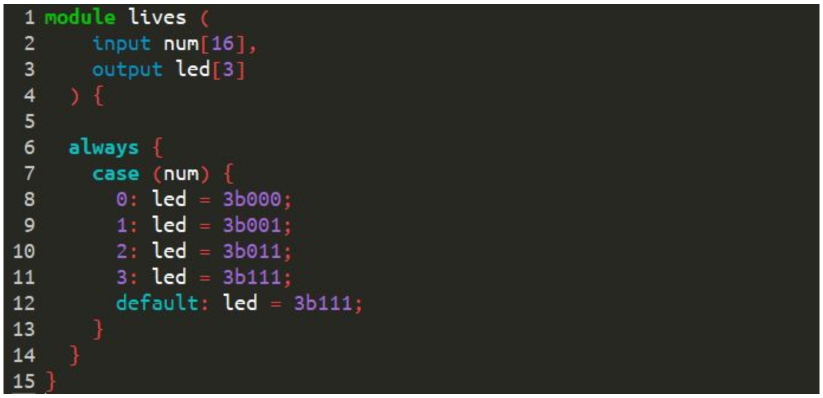
```
module speeds (
    input num[3],
    output speed[24]
  ) {

  always {
    case (num) {
      0: speed = 24h2F0000;
      1: speed = 24h200800;
      2: speed = 24h200000;
      3: speed = 24h10F000;
      4: speed = 24h100000;
      5: speed = 24h0BF000;
      6: speed = 24h0B0000;
      7: speed = 24h080000;
      default: speed = 24h300000;
    }
  }
}
```

Figure 4

Lives

We decided to create a module to help us translate the number of lives in binary to the correct output for the LED

Lives translator takes in the 16 bit input, player lives outputs the corresponding LED input



```
module lives (
    input num[16],
    output led[3]
  ) {

  always {
    case (num) {
      0: led = 3b000;
      1: led = 3b001;
      2: led = 3b011;
      3: led = 3b111;
      default: led = 3b111;
    }
  }
}
```

Figure 5

Hardware

The hardware was mainly built using wooden material with a touch of acrylic and spray paints. The creation of the hardware was divided into several parts before it was joined together using screws and epoxy.

1. Base

   Our hardware is relatively large in size and it requires a strong base in order to support itself. We bought pre-cut plywoods of dimensions 900 x 275 x 14 mm and also metal plates to connect the two plywood together with 50 mm spacing in between for the LED Matrix. Using spray paint and masking tape, we painted a table tennis replica on the base of our board.

2. Platform

   We created four platforms to lift our prototype, allowing the electrical components to be pasted behind the base.

3. 1x16 LED Matrix

   The LED matrix required higher accuracy level to produce the best effect possible. We 'cad' the prototype using Adobe Illustrator before sending it to be laser cut. We secured the LED matrix to the base using 4 screws on each side.

4. Electrical Components

   We used a breadboard to manage our cables properly. Each LEDs utilise two wires, one for the live wire and the other one is for grounding. The same goes for the buttons. Since we do not use the LEDs on the buttons, we do not need to provide extra wires for them. Everything is integrated in the breadboard located next to the Mojo for easy addressing.

Design Issues

ALU Computation

A single ALU does not allow for multiple computations to occur at the same time. For example, it is impossible for the ALU to shift the ball to the right and subtracting the player's lives at the same time. Therefore, the implementation for our code required use to ensure that the entire operation occur at different states.

Mojo I/O shield

Many of the data pins available on the I/O shield were already used by the on-board switches and LEDs. This makes it difficult to test the game as we often don't realize that the problem lies in the used data pin.

Hardware

Human accuracy can never beat machine's when it comes to hardware, especially in the aspect of precise cutting of resources and materials. The pre-cut-to-size wood that we ordered was not accurate to the mm which caused us some trouble with our LED Matrix slot as it came out to be 1 mm wider than the existing slot, requiring us to remeasure our design.

State Transition Diagram

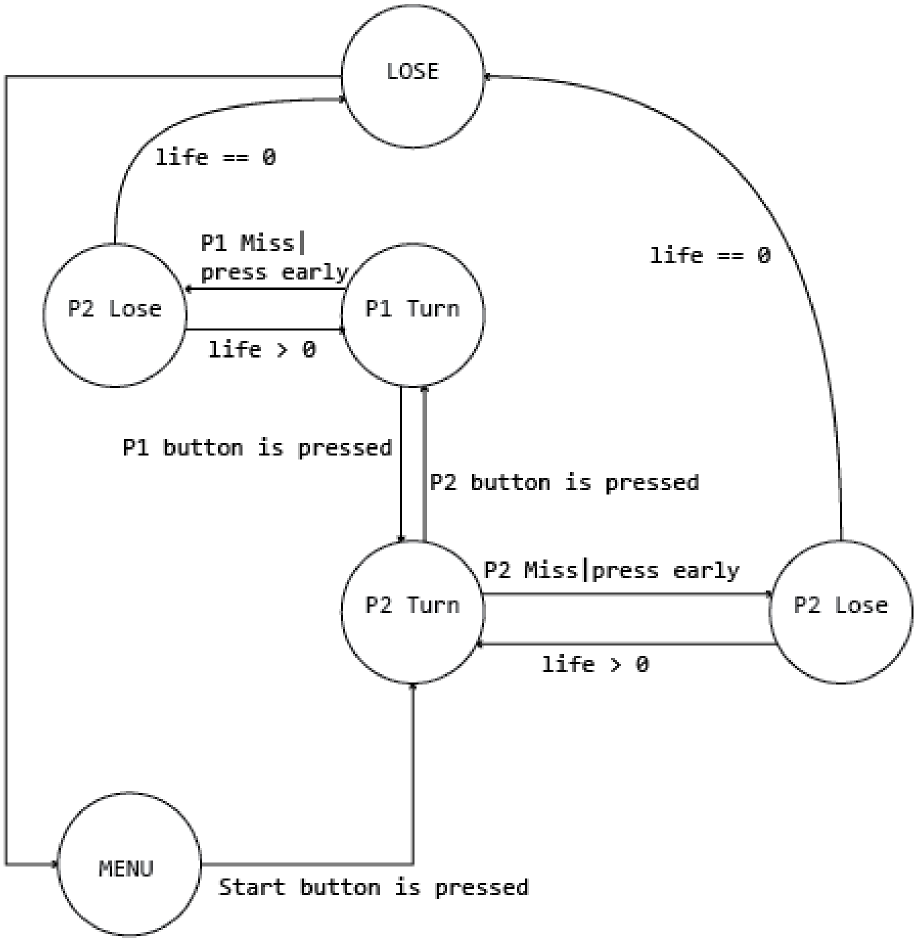


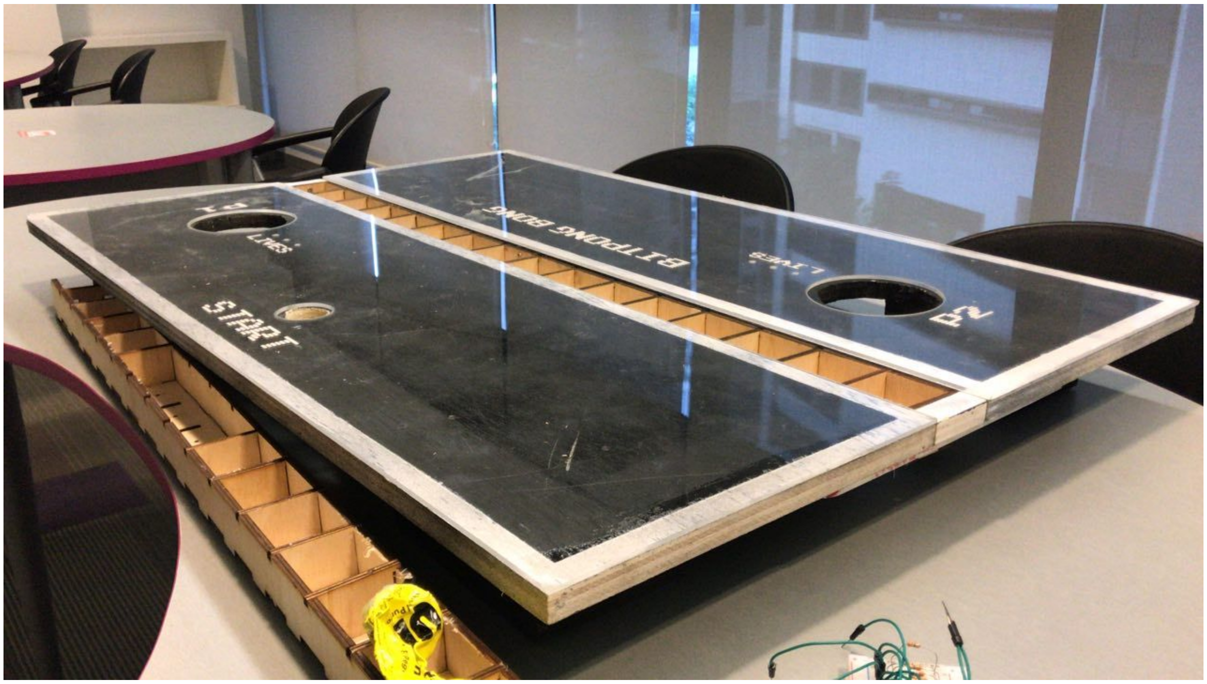
Figure 6

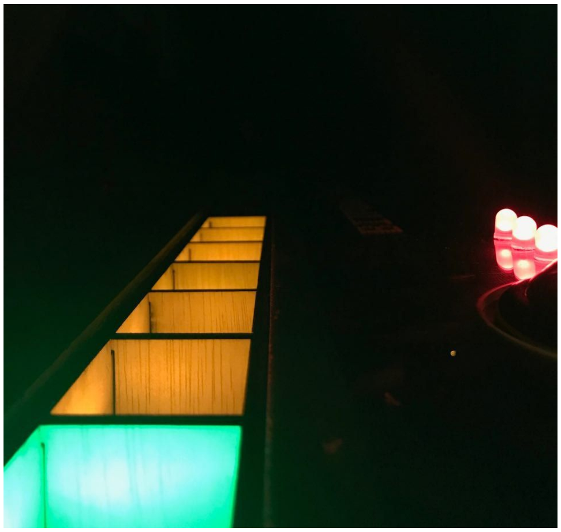
Summary

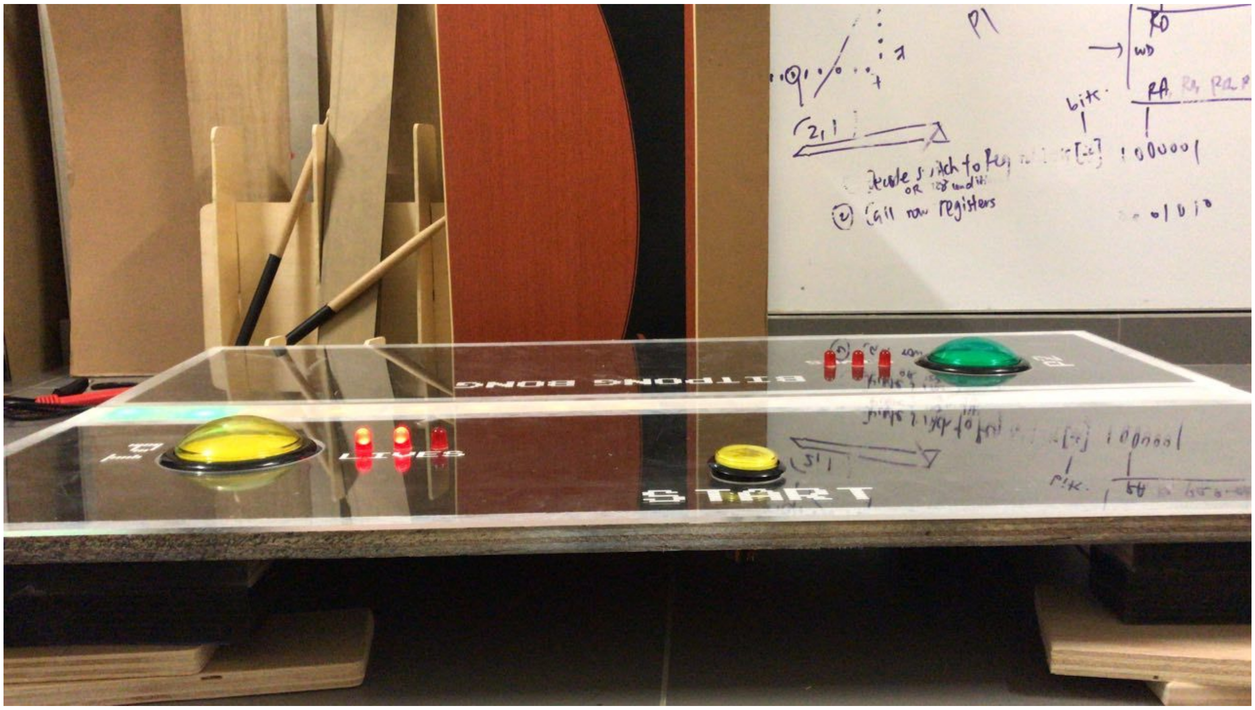
In conclusion, we have successfully created a game using a Mojo FPGA as its base, only utilising a 16-bit ALU. With other components integrated, our game has been proven to be engaging and eye-catching, fulfilling the ‘arcade-ish’ atmosphere which befits the topic of our project.

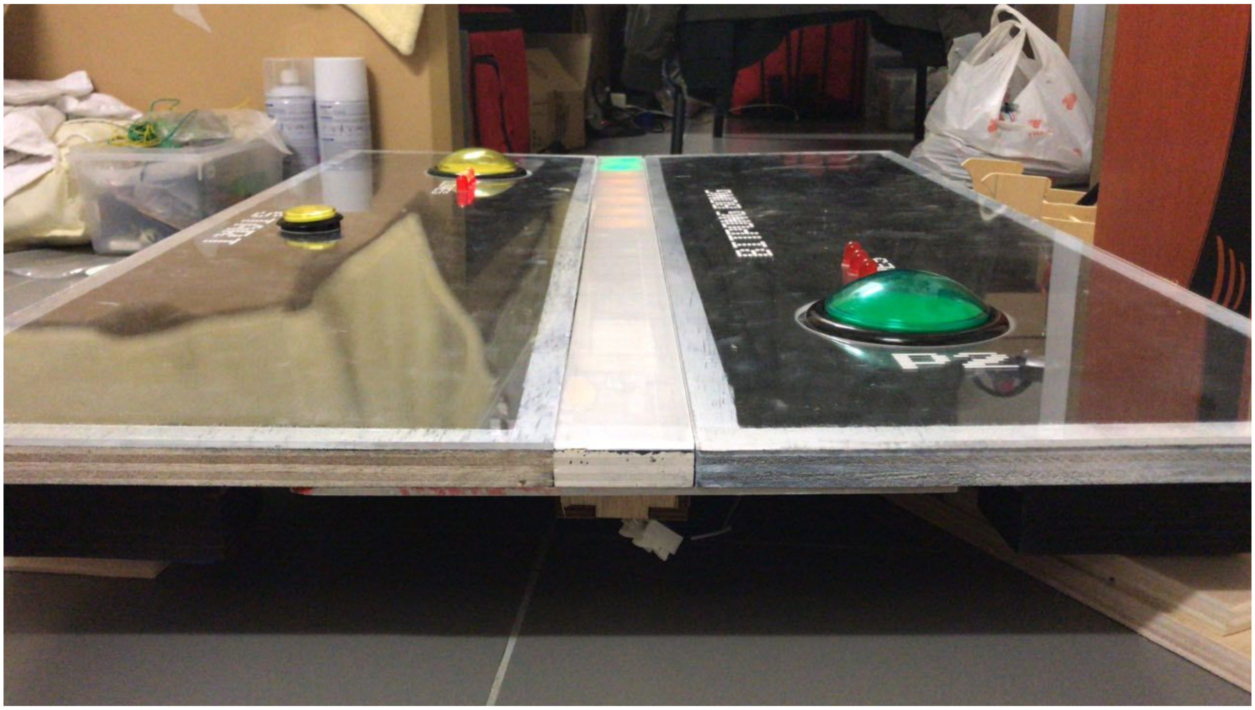
Appendix

Prototype Images



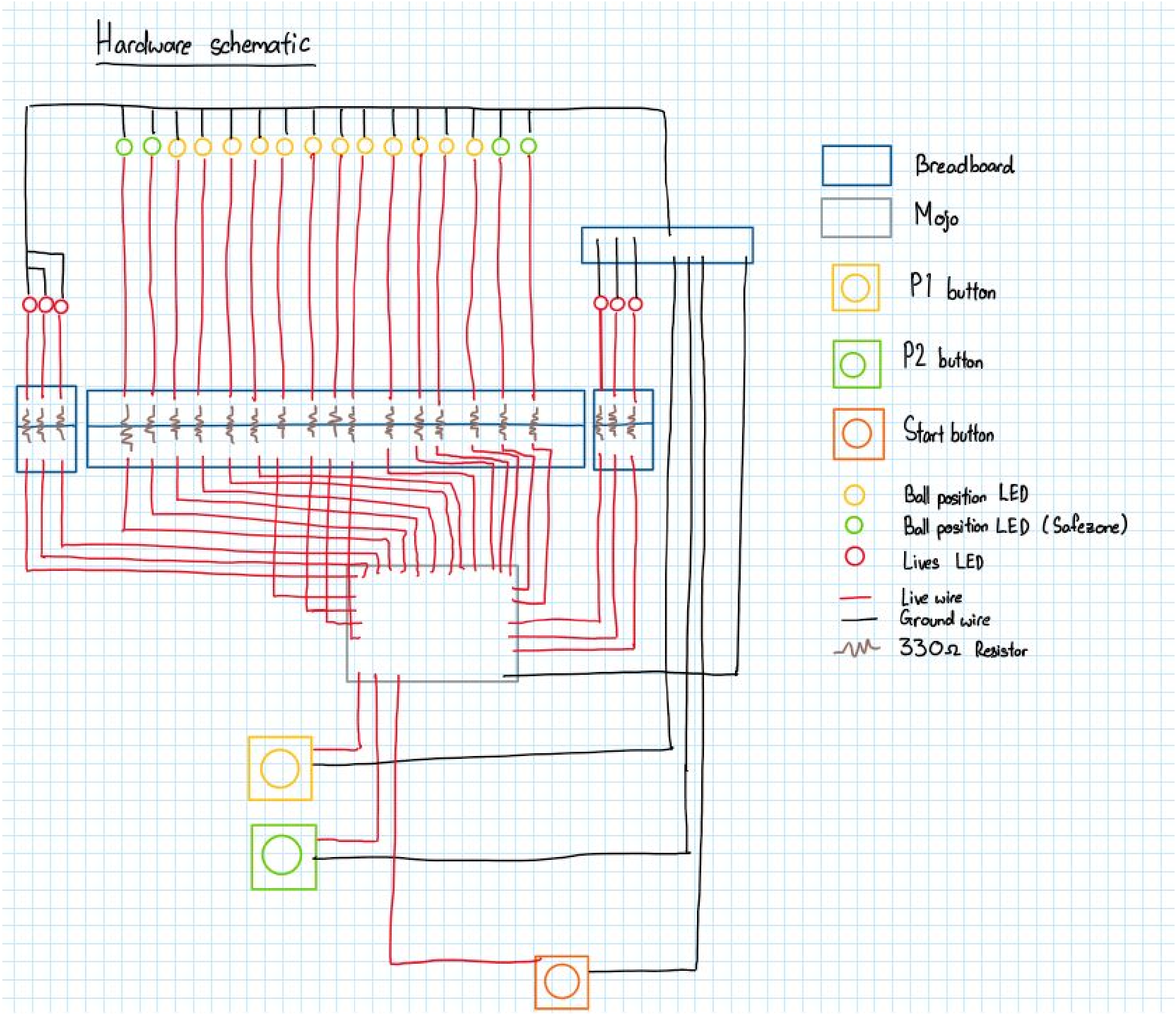



BITPONG BONG
LIVES
START
P1
call new registers


LIVES
BITPONG BONG
P2

BITPONG BONG
START
P2

Prototype Schematic



Game Ideas:

1. Lumberbit

   https://games.tlgrm.eu/gamebot/lumberjack

   Rules

   1. Toggle left and right to chop off chunks of the tree trunk as fast as possible within time limit.
   2. Don’t get hit by tree branches of trunk above.

3. Simplified version of the game is implemented as a column of black and white squares, player will press left button when the bottom most square is white and press right button when it is black.

Implementation Logic

1. Pseudo random string of 0s and 1s pre-generated and stored in memory.
2. String is retrieved and fed to display.
3. Player press the corresponding button. At every button press, the input will be fed into the ALU and compared with the LSB of the string.
4. If input is correct the score is incremented by 1, otherwise the game ends.
5. The string is fed into the ALU again to be shifted by 1 bit to the right.

2. Mario Mania

Rules

1. Listen to short melody (Mario sequence)
2. Replicate melody by means of pressing a button in the same rhythm and tempo.

Implementation Logic

1. The first press of button starts (acts as a reset button) resets the counter to 0.
2. With every subsequent clock tick, a 0 is added to a string if the button is not pressed and 1 is added if the button is pressed.
3. After the string reaches a specified length, the input string is fed into the ALU and compared against the correct string sequence.
4. The score is calculated by subtracting the number of 0s between the corresponding 1s in both strings.

3. Patterny Thingy

Rules

1. Watch and remember the sequence in which the LEDs flash.

   2. Press the corresponding buttons in the same order as fast as possible.
   3. Stages get progressively harder.

   Implementation Logic

   1. Assign each LED/button with a value (i.e. 11, 10, 01)
   2. Random number generator generates sequence of LED/button values (i.e. 11, 10, 01).
   3. Sequence is fed to LEDs.
   4. Use counter that increments with every button pressed to store sequence of buttons pressed.
   5. Compare user input sequence with the correct sequence.
   6. If they are the same, increment score by 1, otherwise the game ends.

4. BitPong (and initial references)

   Make your own 1D Pong Game