Lab 4 Report

- Zihan Liu (105144205)
- Utsav Munendra (805127226)

Introduction and Requirements

We created a maze game with a random walker using the FPGA.

- A joystick was used to get user's input to navigate the user's block across the maze.
  - The user can move their block up, down, left or right.
  - The movement speed is one block per second.
  - The user cannot move into walls
- The goal is to navigate the user's block to the exit point.
  - The user wins when their blocks reaches the exit block across the maze.
  - When user successfully navigates to the exit point, the game is considered completed and the background color changes to signify the completion of the game.
- A randomly-walking wall also traversed the maze at 2 moves per second
  - The user can be blocked by this moving wall, which starts closer to the exit. Similar to the user, the random walker wall cannot move into walls.
- A reset button is available to restart the maze game.

Following modules were implemented:

- VGA module to display maze blocks
- Debouncer for the reset button
- Joystick control module
- LFSR module for pseudo-random number generation, needed for random walker

In addition, we also implemented the seven-segment display module to output debug information.

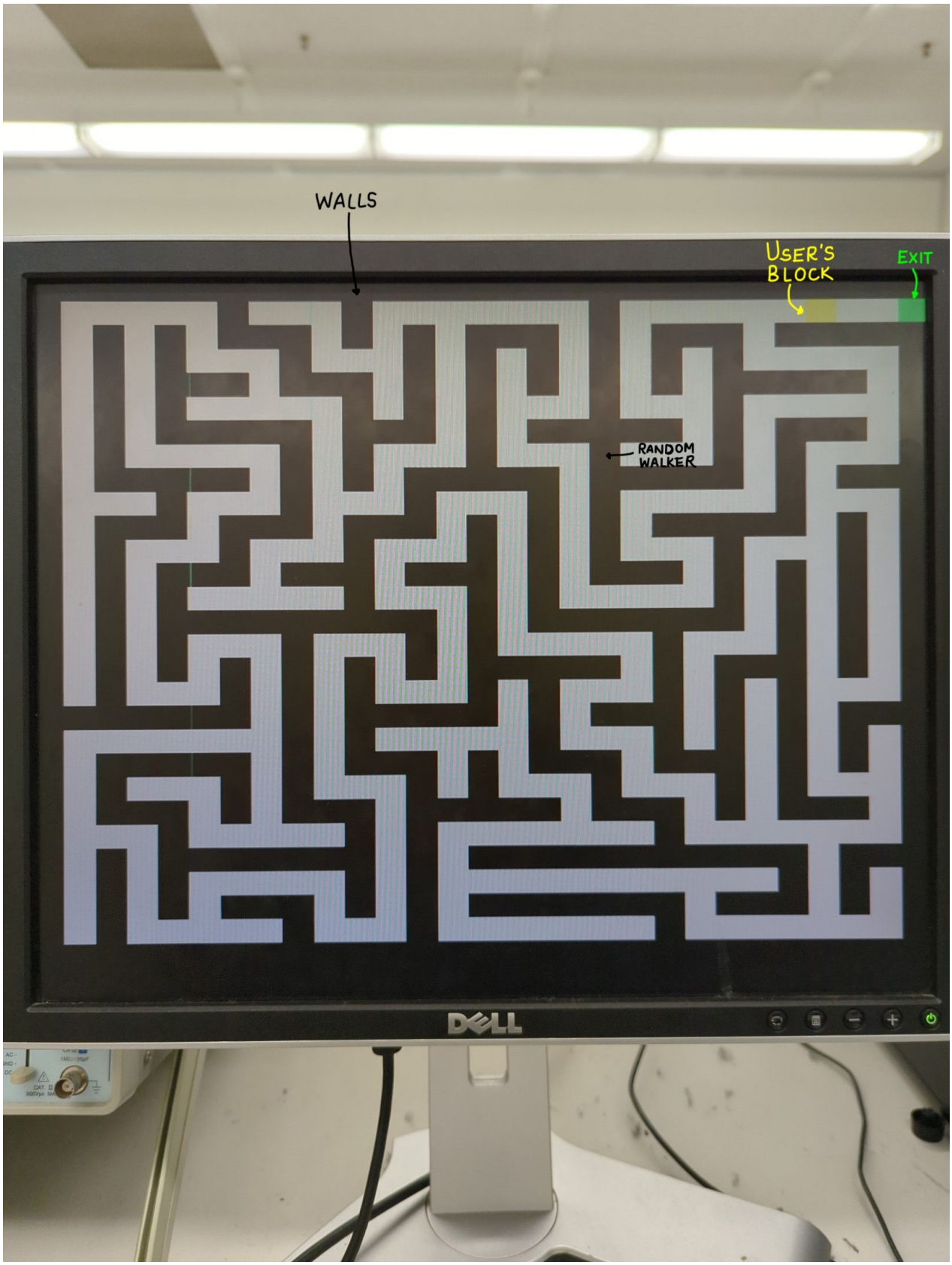


Image from the game just before the user is about to win

Design Descriptions

Our project contains several modules:

Joystick module

The purpose of this module is to process the inputs from the joystick and transform the input into a direction (up/down/left/right). We adapted our implementation from an open source project that followed the Serial Peripheral Interface (SPI) protocol. The original output of this module are two numbers, each ranging from 0 to 1000, that represent the horizontal and vertical movement of the joystick. We converted these two numbers into the direction of the movement using the following logic:

1. If x > 700, move left
2. If x < 300, move right
3. If y > 700, move up
4. If x < 300, move down

See `Joystick.v` for more detail regarding our implementation.

```
module Joystick(
    input CLK,          // 100Mhz onboard clock
    input RST,          // Button D
    input MISO,         // Master In Slave Out, Pin 3, Port JA
    input [2:0] SW,     // Switches 2, 1, and 0
    output SS,          // Slave Select, Pin 1, Port JA
    output MOSI,        // Master Out Slave In, Pin 2, Port JA
    output SCLK,        // Serial Clock, Pin 4, Port JA
    output [2:0] LED,   // LEDs 2, 1, and 0
    output [3:0] AN,    // Anodes for Seven Segment Display
    output [6:0] SEG,   // Cathodes for Seven Segment Display

    output isLeft,      // High when user wants to move left
    output isRight,     // High when user wants to move right
```

```
    output isUp,         // High when user wants to move up
    output isDown        // High when user wants to move down
);
```

SevenSegment module

This module was used to display debugging information on the board. We used it to display the row and col of the random walker when we were building and testing the *RandomNumGen* module. The final product does not depend on it. We adapted this module from our implementation of the Stopwatch project.

See `SevenSegment.v` for detail information about our implementation.

```
module SevenSegment(
    input hundredHz,        // 500 Hz clock with 50% duty cycle
    input[3:0] digit0,      // Rightmost digit
    input[3:0] digit1,      // Middle right digit
    input[3:0] digit2,      // Middle left digit
    input[3:0] digit3,      // Leftmost digit

    output reg [7:0] seg,   // Segments
    output reg [3:0] an     // Digit-selector
);
```

Debouncer module

We used this module to handle the debouncing of button R, which we used to reset the maze. We adapted a similar implementation as the one we used in Lab 3. We first transform the 100-Hz clock into a pulse and only sample the button R at the pulse. We confirmed that the button R has been pressed when two consecutive samplings of button R yield opposite value and we output the signal.

See `Debouncer.v` for more details.

```
module Debouncer(
```

```
    input clk,              // Master clock
    input hundredHz,        // 500 Hz clock with 50% duty cycle
    input resetButton,      // Reset Button from the FPGA

    output reg btnR // Signal that btnR was pressed
);
```

RandomNumGen module

We used a 4 bit random number generator. The module is parameterized by the number of bits needed. We supply this module with a one hertz clock. At every clock tick, the 4 bits are left shifted by one, and a new random bit is calculated. For a 4 bit output, the new bit is computed as the XNOR of the earlier 4th and 3rd bit:

```
r_XNOR = r_LFSR[4] ^~ r_LFSR[3];
```

See `RandomNumGen.v`

```
module RandomNumGen(
   input i_Clk,                          // Clock
   input i_Enable,                       // Start generation
   input i_Seed_DV,                      // Set high to start
   input [NUM_BITS-1:0] i_Seed_Data,     // Initial seed value

   output [NUM_BITS-1:0] o_LFSR_Data,  // Pseudo-random number
   output o_LFSR_Done                    // High when generation d
one
);
```

VGA and Top-level Module

The rows and columns of the maze are parameterized. HSync goes high for a bit to signal that a row of pixels is done, and VSync does high for a bit to signal that all of the rows are done. See below for the explanation on the rest of the inputs and outputs.

We used `for` loops in Verilog to assist us in loop unrolling a lot of if-statements. Those if-statements check if the current pixel is part of a block on the grid, then that pixel takes the color of that block.

See `MazeTop.v`

```
module MazeTop(
    input clk,                  // Master clock
    input btnRight,             // Right button
    input miso,                 // MISO for SPI-Joystick
    input [2:0] sw,             // Switches for debugging

    output ss,                  // Slave-select for SPI-Joystick
    output mosi,                // MOSI for SPI-Joystick
    output sclk,                // Slave clock for SPI-Joystick
    output [7:0] Led,           // Debug LEDs
    output [3:0] an,            // Debug Seven-Segment Anode
    output [6:0] seg,           // Debug Seven-Segment Segments
    output Hsync,               // VGA horizontal sync
    output Vsync,               // VGA vertical sync
    output reg [2:0] vgaRed,    // VGA red pixel value
    output reg [2:0] vgaGreen, // VGA green pixel value
    output reg [2:0] vgaBlue    // VGA blue pixel value
);
```

Verilog Maze-Block Code Generation using Python

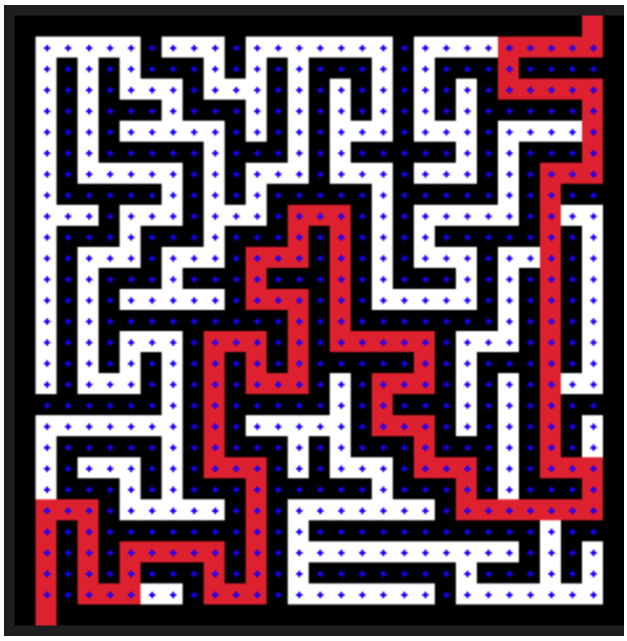
The maze contains about 360 walls. To demarcate which cells in the grid should be walls, we wrote a Python notebook to extract the walled cells indices from an image of the maze based on the color of the pixel at the sampling points in the middle of the grid cells.

We generated a maze image from https://keesiemeijer.github.io/maze-generator/.

See `maze.ipynb`

```
blocks[1][6] <= 1;
blocks[1][10] <= 1;
blocks[1][18] <= 1;
blocks[2][2] <= 1;
blocks[2][4] <= 1;
blocks[2][6] <= 1;
blocks[2][7] <= 1;
blocks[2][8] <= 1;
blocks[2][10] <= 1;
blocks[2][12] <= 1;
blocks[2][14] <= 1;
blocks[2][15] <= 1;
blocks[2][16] <= 1;
blocks[2][18] <= 1;
blocks[2][20] <= 1;
blocks[2][21] <= 1;
blocks[2][22] <= 1;
blocks[2][24] <= 1;
blocks[2][25] <= 1;
blocks[2][26] <= 1;
blocks[2][27] <= 1;
blocks[3][2] <= 1;
blocks[3][4] <= 1;
```

Auto-generated verilog code

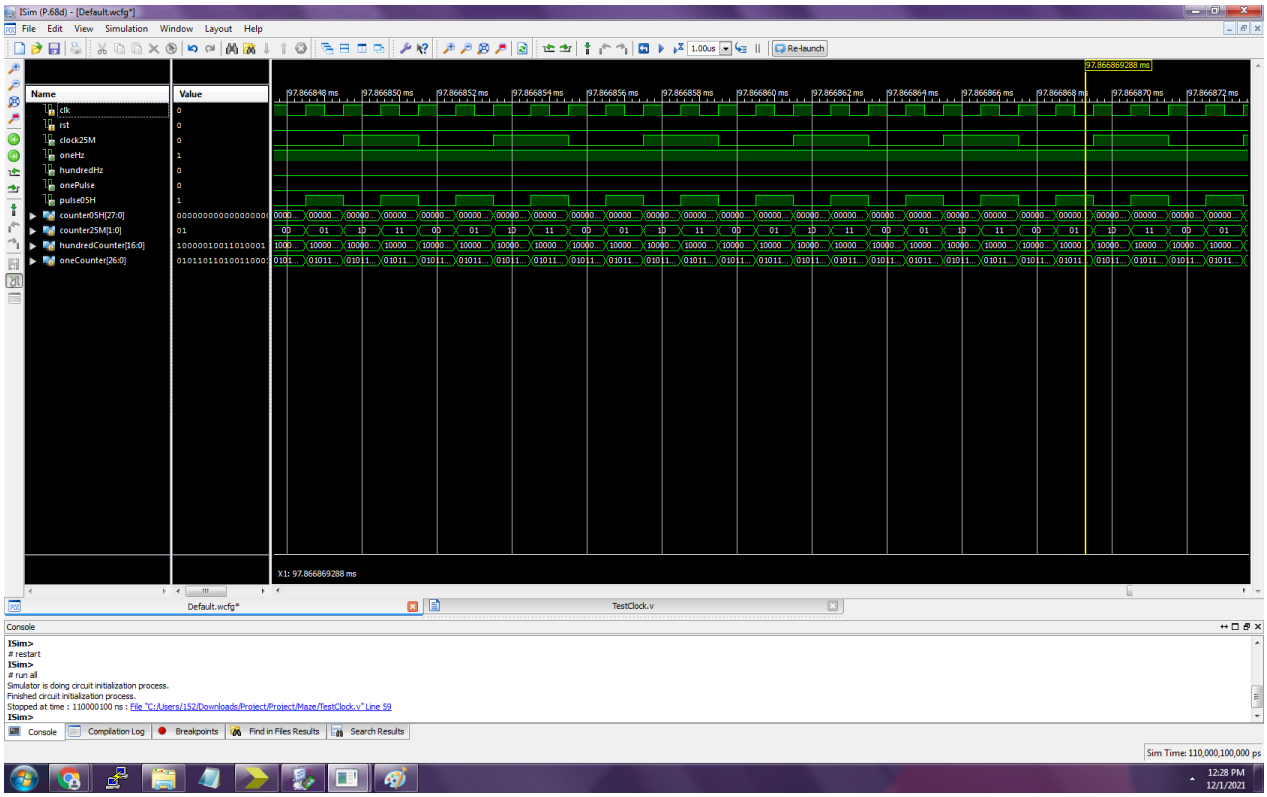


Sampling points from a maze image

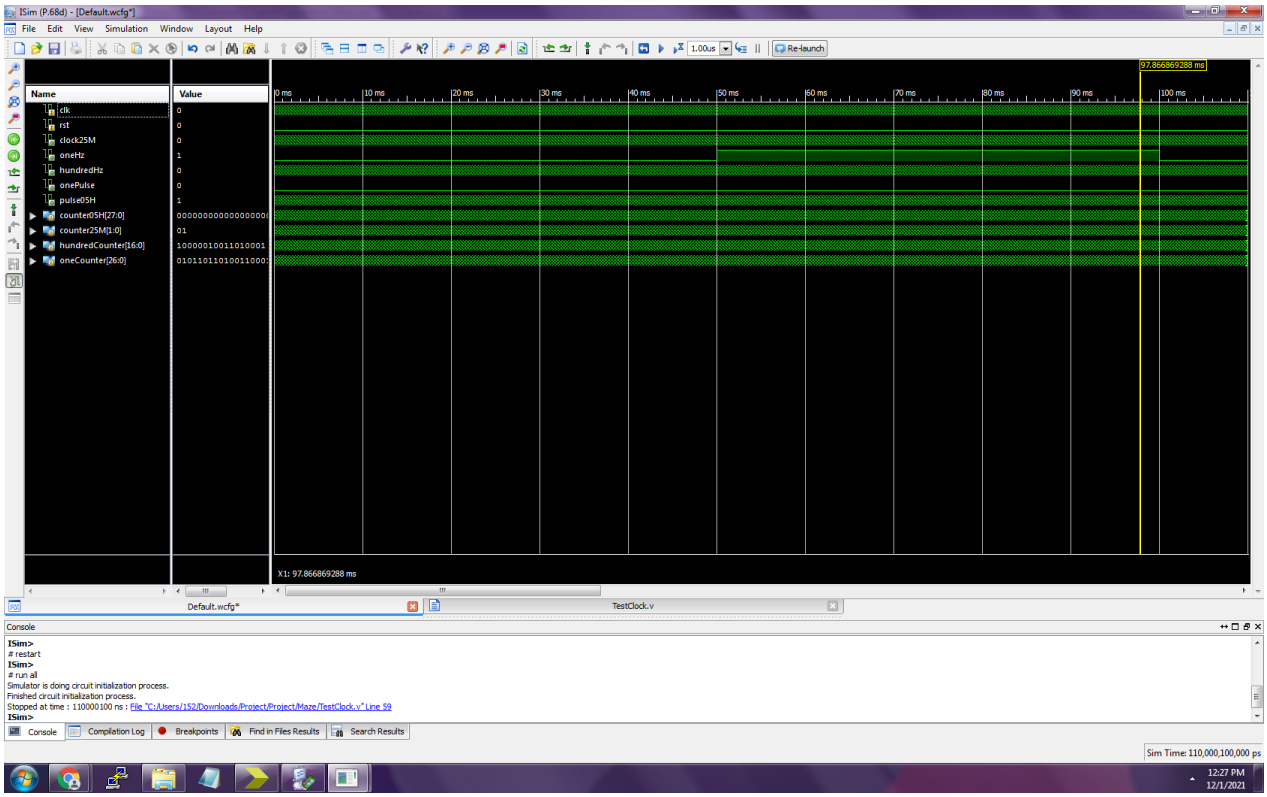
Simulations Documentation

The following section documents the test bench simulation of this project:

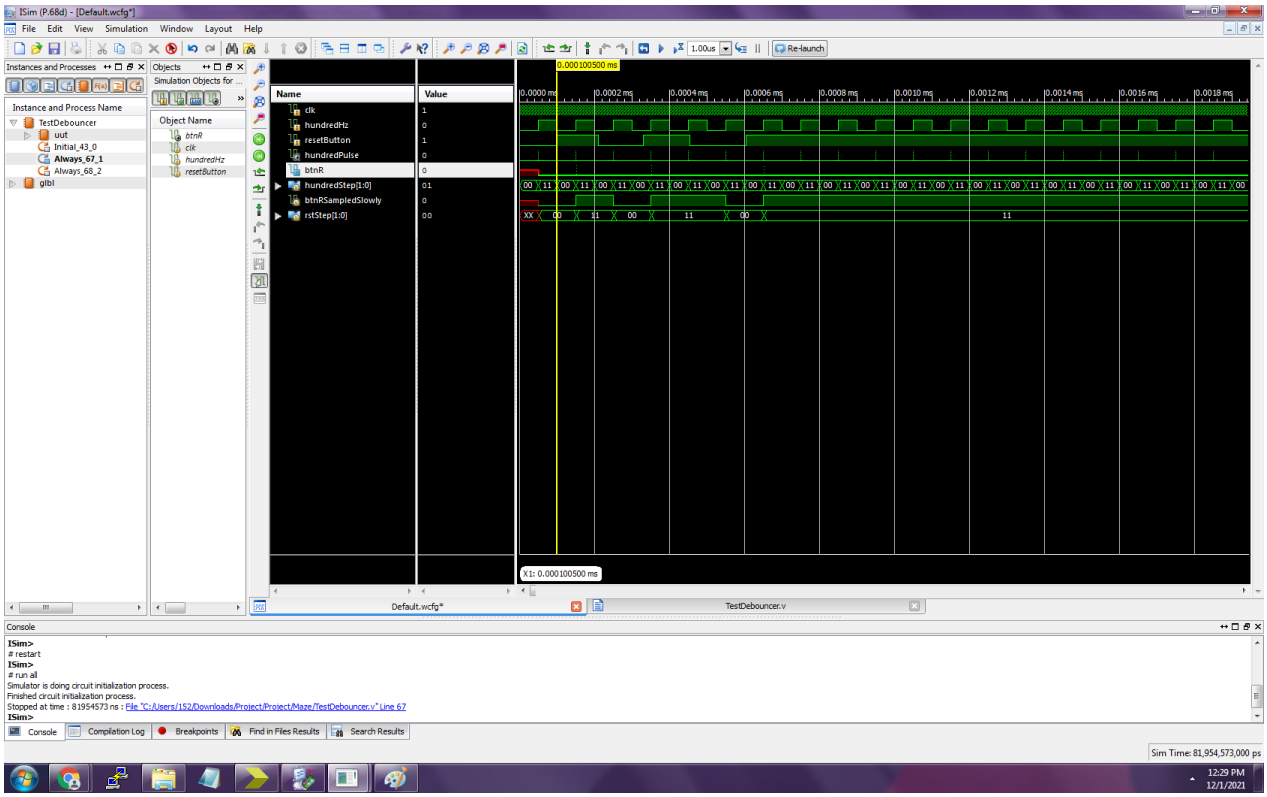
1. The screenshot below captures the test bench simulation for the 25-million Hz clock, which is used to implement the VGA. As shown in the image, the 25-million Hz clock has four-times the interval of the master clock *clk*, which has a frequency of 100-million Hz.



2. This screenshot demonstrates the simulation of the one Hz clock. This clock is used to sample the user input from the joystick module, which allows user to move one block per second. As shown in the image, the one Hz clock completes one cycle in 100ms (because the time scale is expedited to save time in simulation).



3. The last test bench simulation deals with the debouncing. We only implemented a debouncer for button R as it's the only button we used. We used the pulse of the 100 Hz clock to sample the input of button R.



Conclusions

We satisfied all of the criteria from our proposal:

- [✓] If the maze game is playable, give 100%
- If the maze game is not playable:
  - [✓] 20%: Game can accept input through either 4 buttons or the joystick. The input is in either one of four direction or no direction.
  - [✓] 30%: The monitor can display some color controllable from the FPGA.
  - [✓] 10%: The monitor can display a maze grid from the FPGA.
  - [✓] 20%: Random number generator outputs numbers that look (pseudo) random.
  - [✓] 20%: The player's block can move on the screen. The movement is correct, i.e, the player's movement is blocked by wall or other form of boundary.

Our design is modularized, has test benches for sub-modules wherever necessary, has an overarching test module, uses descriptive variable names and in properly indented. We use different frequency clocks, use a debouncer wherever needed and

are able to make a maze game.

One of the challenges in implementing this project was debugging whenever the behavior was not as expected. We had to implemented a SevenSegment module in order to output debugging information in real time. Creating a test bench for the sub-modules also helped considerably. Another challenge we faced was about the VGA display. Since we adapted our VGA module from a demo we found online, we didn't understand some of the implementation details. We were forced to resort to the method of trial and error, changing the value of one constant at a time and then running it, to be able to understand the code.

References

1. https://embeddedthoughts.com/2016/12/09/yoshis-nightmare-fpga-based-video-game/ (VGA)
2. https://github.com/AndreasKaratzas/graphics-driver/blob/main/vga_controller.v. (VGA)
3. https://simplefpga.blogspot.com/2013/02/random-number-generator-in-verilog-fpga.html. (Random number generation)
4. https://digilent.com/reference/_media/reference/pmod/pmodjstk/pmodjstk_demo_verilog.zip (Joystick)
