Cole Schreiner
Due 12/08/2023
CSE100/L - F23 Logic Design Laboratory
Lab Section: 03
Professor Martine Schlag
University of California, Santa Cruz

Lab 6 Report - Osmosis

Description

The objective of this lab was to engineer several state machines for the implementation of a game called “Osmosis”. This was accomplished by implementing a series of circuits in Vivado through Verilog code. This involved creating several state machines to govern the physics and rules of the game, with the ultimate goal of implementing and testing the circuit on the BASYS3 board.

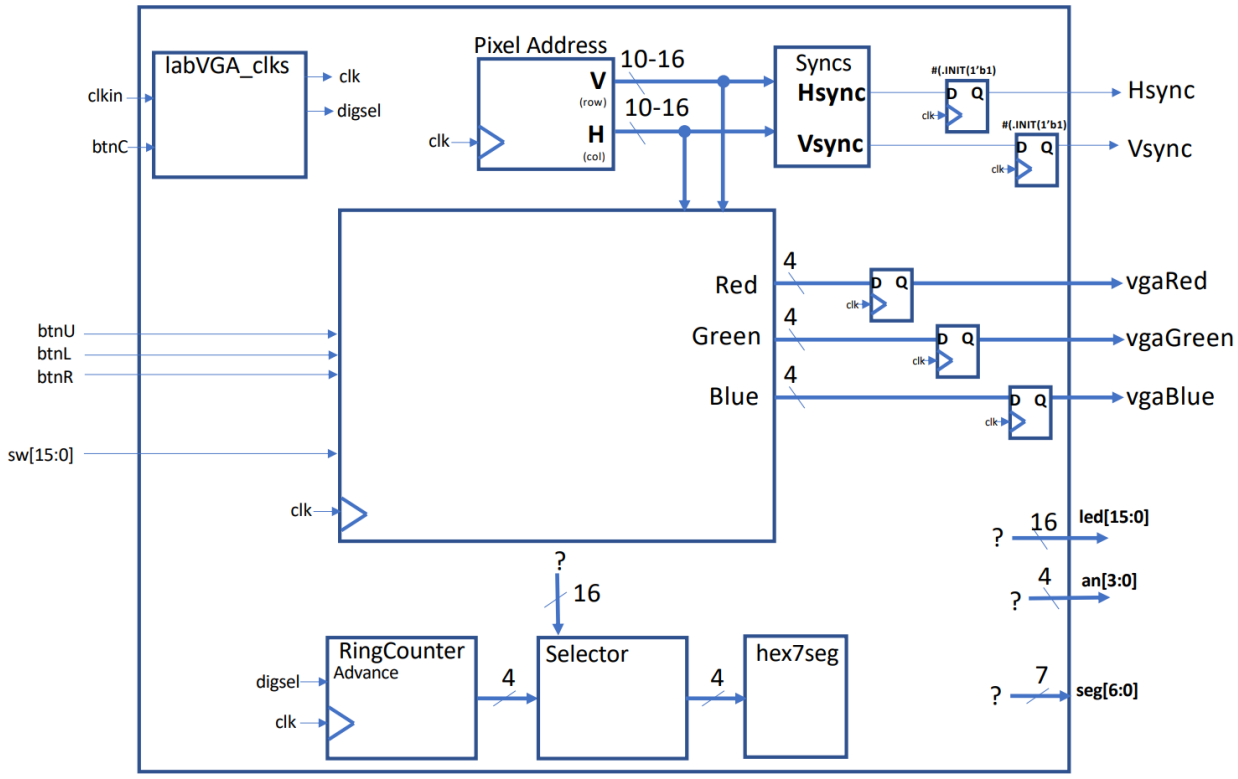
The game utilizes much more of the BASYS3 board’s capabilities. In Previous labs, we used buttons, switches, LEDs, and seven-segment displays. In this lab, we took it a step further and employed the VGA output on the board to play the game from a monitor. The game begins when the player presses the up button (btnU) to trigger the “Go” state of the state machine. From here, 4 blue balls and 4 red balls begin to move on the screen, bouncing when they hit a wall. There is a line going down the middle which controls the “membrane” and allows certain colored balls to pass through or bounce off of it. The core aspect of this game is trying to separate the red and blue balls on the screen before the timer runs out.

Design

Top_Level

The “Top_Level” module serves as the central hub for interconnecting all of the components of the game. It encompasses the integration of all modules: the game’s state machines, creating size and rules for the balls, creating the screen region to display, and a multitude of counters and flip-flops.

As you can see from the diagram below, the entirety of the game is controlled by several input signals: the clock (clkin), 4 buttons (btnU, btnR, btnL, btnD), and the board's switches (sw[15:0]). The outputs of the Top_Level module are all indicators to the player about what is happening in the game. That is, the LEDs tell the player which switches are on, the anodes which light up the 7-segment displays are responsible for displaying the timer, and the display monitor which encompasses the game's visuals.



The majority of the Top_Level module is wiring together all of the various components inside of it through the use of wires as well as plugging in each component's inputs and outputs into each other. This wiring creates the entirety of the project so that all modules can work together; we need the rule for the game, rules for the ball, a way to make 8 balls riven their rules, borders for the display, etc. 3 essential parts of Top_Level are the 16-bit counters I implemented for the various timers of the game. Below is an example of code showing these counters:

```
            // -> // 1 second counter
        wire Second1_out, Second1_dtc;
        wire [15:0] Second1;
        Counter_16Bit AsingleSecond (.clk(clk), .UP(1'b0), .DW(frame),
.LD(Second1_dtc), .sw(16'd60), .Q(Second1), .UTC(), .DTC(Second1_dtc));
```

```
            // -> // 8 second timer at the start
        wire Second8, Second8_out, Second8_dtc;
        Counter_16Bit Counter16bit (.clk(clk), .UP(1'b0), .DW(frame),
.LD(CountingDown), .sw(16'd480), .Q(Second8_out), .UTC(),
.DTC(Second8_dtc));

            // -> // Game Timer
            wire [15:0] GameCount_out, TimeAnode;
        Counter_16Bit GameCountdownTimer (.clk(clk), .UP(1'b0),
.DW((Second1 == 16'd0) & Play), .LD(Go), .sw({8'd0, sw[15:8]}),
.Q(TimeAnode), .UTC(), .DTC(GameCountdown));
```

The 1-second counter, denoted by “Second1”, operates based on the clock signal (clk) and the frame signal. It counts down from a preset value (60) and facilitates timekeeping within the game.

The 8-second timer (Second8) ensures a smooth start to the game. It counts down from a preset value (480) upon initialization, providing a brief window before the game actually begins.

The game countdown timer, represented by “GameCountdownTimer”, is intricately connected to various game conditions. It counts down from the value specified by the switches (sw[15:8]) and is loaded based on game initiation (Go). Its functionality is tied to the overall game state, ensuring accurate timekeeping during gameplay.

In the following sections, I will discuss more of the logic behind these components and how they interact with each other to form the visual game as we see it.

GameState

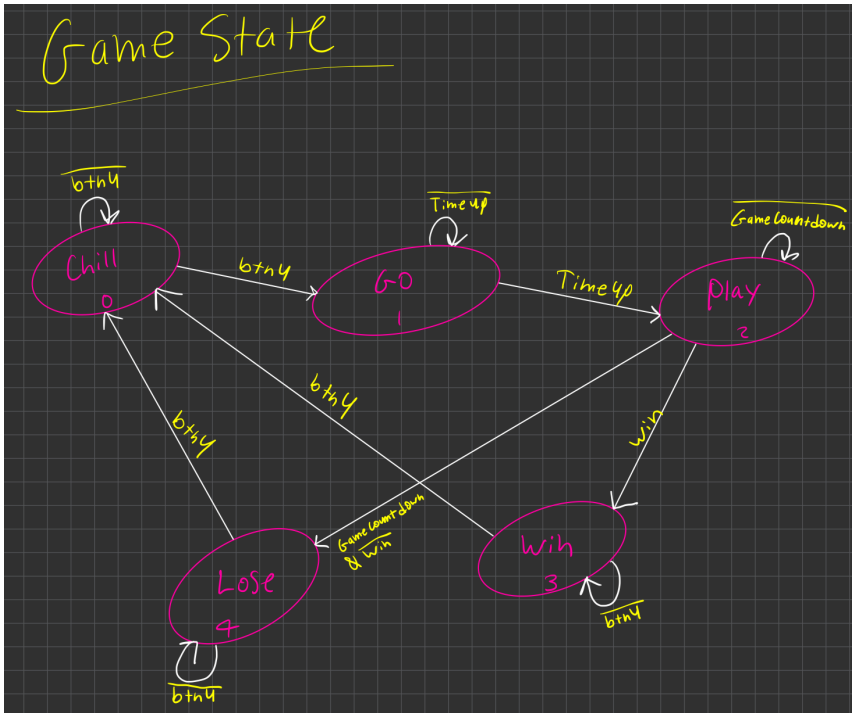
GameState is a state machine module responsible for the rules of the game. It is the core logic behind what has happened, what is happening, and what will be happening in terms of the game’s overall mechanics. It orchestrates the transitions between the game's various states and functions. The system exists in a specific state at any given time, transitioning to other states in response to external inputs while generating outputs based on its current state.

This module was fairly simple to create from memory as it operates based on five key states: Chill, Go, Play, Win, and Lose as you can see in my code segment below:

```
            // Chill
        assign NS[0] = (PS[0] & ~btnU) | (PS[3] & btnU) | (PS[4] &
btnU);
```

```
            // Go (lasts 8 seconds long) -  Flash Borders, Load/show
timer, Move balls
         assign NS[1] = (PS[0] & btnU) | ( PS[1] & ~TimeUp);
            // Play (after 8 seconds) -
         assign NS[2] = (PS[1] & TimeUp) | (PS[2] & ~GameCountdown &
~Win);
            // Win when blue and red balls are separated
         assign NS[3] = (PS[2] & Win) | (PS[3] & ~btnU);
            // Lose
         assign NS[4] = (PS[2] & GameCountdown & ~Win) | (PS[4] & ~btnU);
```

Additionally, a visual representation of this code can be seen from my state machine diagram below:



“Chill” is the starting state and is simply awaiting input from the player to press btnU to start the game. From there it moves on to "Go" which starts an 8-second countdown setting up the game. In this state, it flashes the borders of the screen, removes the middle membrane, displays the timer on the anodes, and begins the balls’ movement. After the 8 seconds are up, the flashing stops, the anodes start counting down on the seven-segment displays, and the middle membrane appears again. Here we are in the “Play” state. It is up to the player to press the left and right buttons on the board in order to separate the red and blue balls on either half of the screen before the timer runs out (as decided by the switches which add more time per round). The

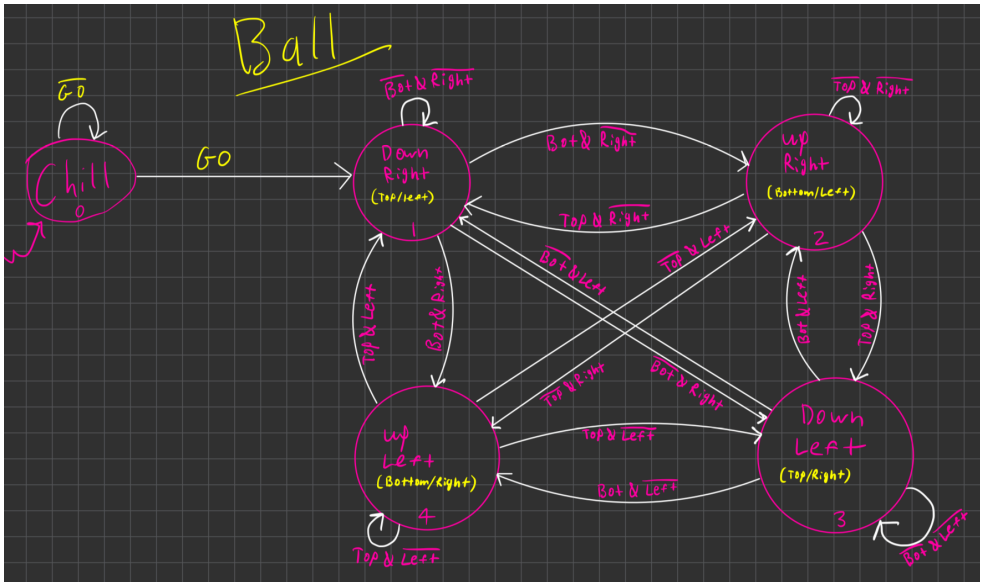
membrane is initially purple as both red and blue membranes are visible, allowing no balls to pass through. The player can press the left or right buttons, which will turn off the blue or red membranes. Red balls can pass through the blue membrane and blue balls can pass through the red membrane. If the player separates the red balls from the blue balls before the timer runs out, the state machine transitions to the “Win” state in which the timer stops and the border flashes. If however, the player is unable to separate the balls before time runs out, we transition to the “Lose” state and the timer anodes will flash indicating the player has lost that round. The Win and Lose states will remain until the player either starts a new round by pressing btnU, or the player can reset the game entirely using btnC (which resides in the Top_Level module).

Ball

The "Ball" module creates a state machine for what the balls should do when they reach a boundary. For example, if a ball hits the top border, it could be coming from the left or from the right. If the ball hits the top and is coming from the right, it transitions states so that it will continue moving right but must change course and move downwards.

```
//  State Machine
        // Chill / Start / Go
    assign NS[0] = PS[0] & ~Go;
        // Down-Right   Top/Left   Start
    assign NS[1] = (PS[0] & Go)              | (PS[1] & ~Bottom & ~Right)
| (PS[2] & Top & ~Right)    | (PS[3] & ~Bottom & Left) | (PS[4] & Top &
Left);
        // Up-Right     Bottom/Left
    assign NS[2] = (PS[1] & Bottom & ~Right) | (PS[2] & ~Top & ~Right)
| (PS[3] & Bottom & Left)   | (PS[4] & ~Top & Left);
        // Down-Left    Top/Right
    assign NS[3] = (PS[1] & ~Bottom & Right) | (PS[2] & Top & Right)
| (PS[3] & ~Bottom & ~Left) | (PS[4] & Top & ~Left);
        // Up-Left      Bottom/Right
    assign NS[4] = (PS[1] & Bottom & Right)  | (PS[2] & ~Top & Right)
| (PS[3] & Bottom & ~Left)  | (PS[4] & ~Top & ~Left);
```

As you can see from the above code, all walls are taken into account, including the direction the ball is heading it where it will move to next. This module is called by the “Balls_Moving” module to use these rules and apply it to a physical shape. This can be seen in my state machine below:



Balls_Moving

Balls_Moving is responsible for the sizing and shape of the 8 balls I create in Top_Level as well as creating the logic and shape of the membranes that the balls interact with. The size and shape of the membranes can be seen below:

```
    // Invisible membrane, No membrane, Purple membrane
    wire [3:0] MemInvis, MemNone;
    assign MemInvis = ((Hpixel >= 16'd316) & (Hpixel <= 16'd323) & (Vpixel
<= 16'd471) & (Vpixel >= 16'd8)) & ShowMem;
    assign MemPurp =  (MemInvis & ~btnR & ~btnL);
    assign MemBlue =  (MemInvis & ~btnL);
    assign MemRed =   (MemInvis & ~btnR);
```

The “MemInvis” is an invisible membrane responsible for the size and shape the the membrane’s edges. Red, blue, and purple are the membrane’s colors when manipulated by certain buttons on our board. The buttons essentially enable or disable those boarders.

This module also contains rules for when a ball is at the top, bottom, left, and right edges but the most important part of this module is the creation of the balls below:

```
        Counter_16Bit Ball1x (.clk(clk), .UP(Upx & frame), .DW(Downx &
frame), .LD(Count), .sw(BallStartx), .Q(Posx));
        Counter_16Bit Ball1y (.clk(clk), .UP(Upy & frame), .DW(Downy &
frame), .LD(Count), .sw(BallStarty), .Q(Posy));
```

```
    //------------------------ Making Ball #1 -----------------------\\
(Red)
    Ball MakinBall_1 (.clk(clk), .Go(btnU), .Top(Top), .Bottom(Bottom),
.Left(Left), .Right(Right),
                        .Count(Count), .Upx(Upx), .Upy(Upy),
.Downx(Downx), .Downy(Downy), .NS(NS));
    assign MakeBall = ((Posx <= Hpixel) & (Hpixel <= Posx + 16'd16) &
(Posy <= Vpixel) & (Vpixel <= Posy + 16'd16));
```

The “Balls_Moving” module must call the module “Ball” in order to take into account the rules for the balls and how they are supposed to interact with the display region. Next, I will be discussing the active region of the display.

Pixels

The "Pixels" module is my VGA controller which is responsible for measuring out the active region of the display. It uses two 16-bit counters in conjunction with multiple constraints for the size and active region the pixels are allotted to display on. This is crucial for the game to run properly so that the player can visually see what is happening in the game in order to interact with it. Additionally, after creating the pixel address module for this lab, I moved on to creating the borders as a natural next step.

Borders

The "Borders" module is called in Top_level as has the sole purpose of being the boundary for all 8 balls to stay within the game’s display region. This module is very simple as it simply creates a green wall running along the top, bottom, left, and right edges of the active region. The balls are not allowed to cross the border under any circumstances. The borders are simply a size region as defined below:

```
        assign HborderStart = 16'd632;      // Start of right side
border (green wall)
        assign HborderEnd   = 16'd639;      // Length
        assign VborderStart = 16'd472;      // Start of bottom border
        assign VborderEnd   = 16'd479;      // Height
```

There are some additional details in my code which assign these values to create the thickness of the border walls.

Conclusion

This lab, like many of the past labs, was extremely difficult. Specifically, wiring together the Top_Level and the logic behind the state machines had given me the most

trouble. Unfortunately, nearing the end of the project was extremely difficult to test as it required changing only a few lines of code but generating the bitstream every time took several minutes. By testing my code extensively, I was able to better understand the importance of creating a state machine. Additionally, I had to make small incremental changes to navigate through the project's substantial scope without becoming overwhelmed. In the end, my final product worked exactly as we were assigned and I had no trouble completing all aspects of this lab. If I did this lab again, I would have liked to have had the time to start earlier so that I could take a slower pace. In the end, I found that moving slower made me faster overall as I had fewer errors and mistakes to hunt down. I only wish that generating bitstream was a faster process.

Appendix

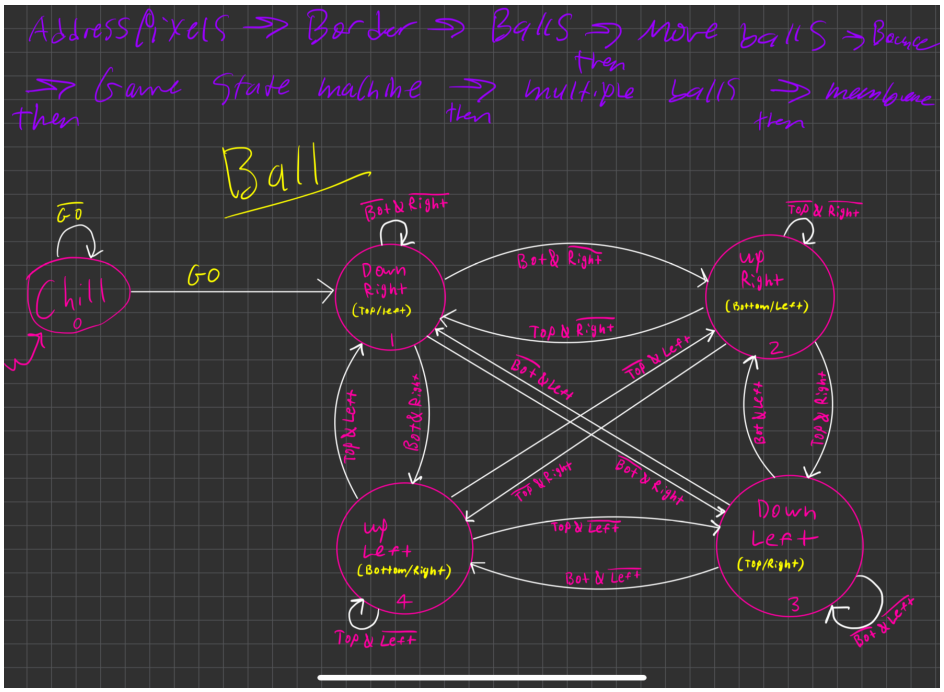


Fig. 1 Ball State Machine

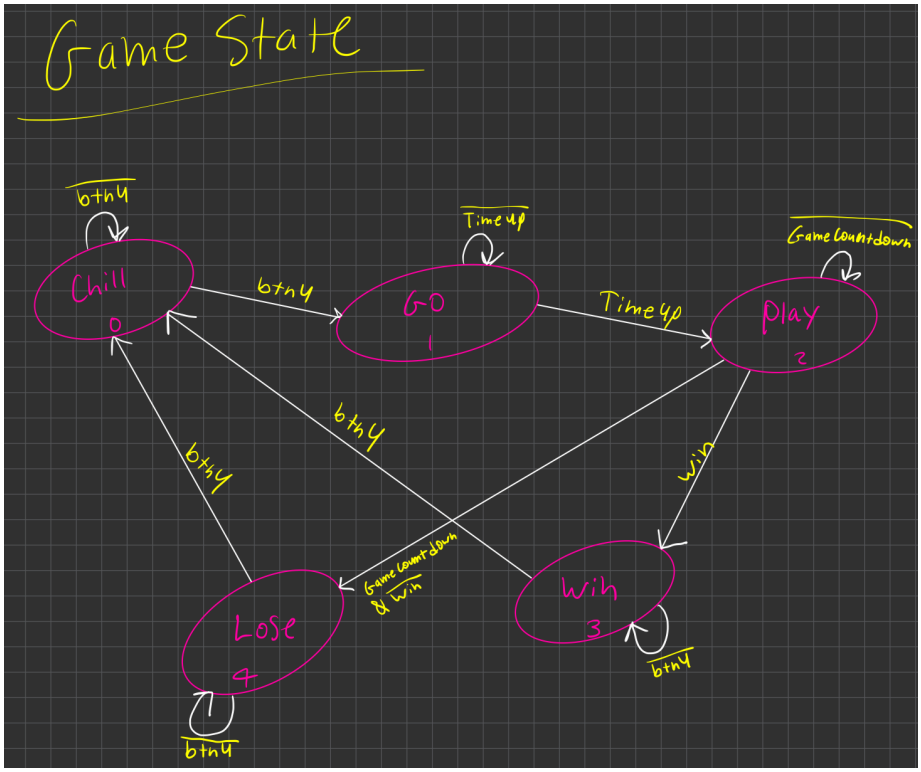


Fig. 2 Game State Machine

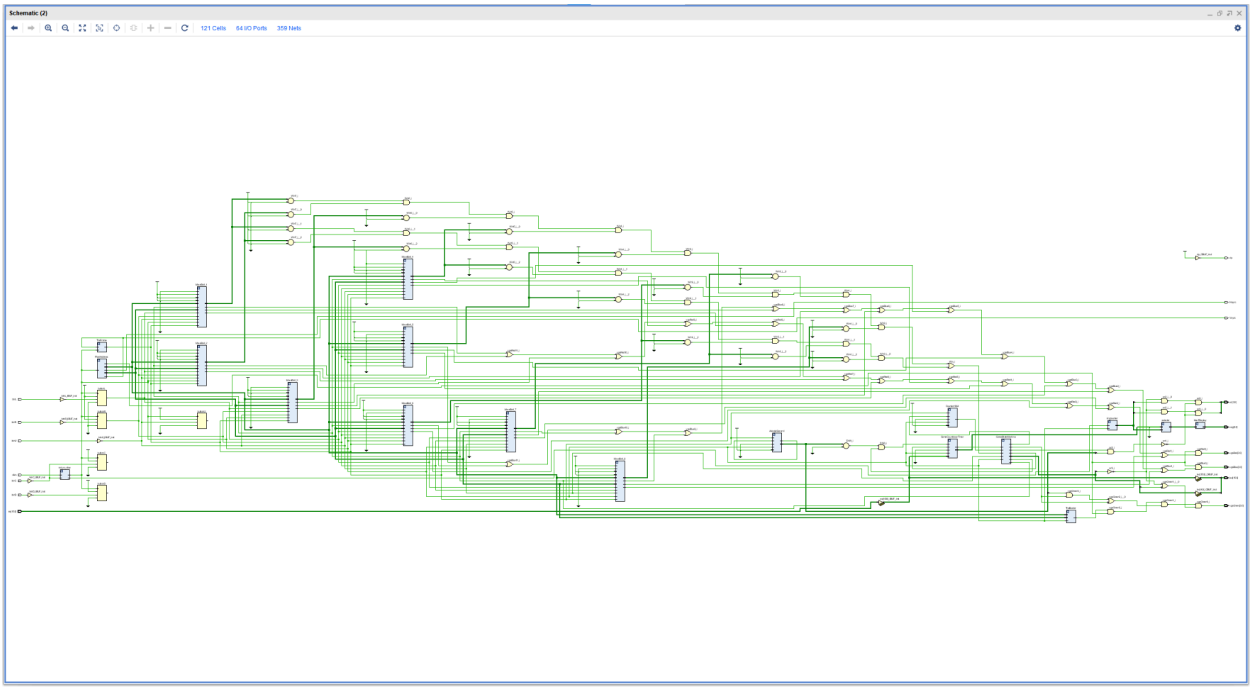


Fig. 3 Schematic

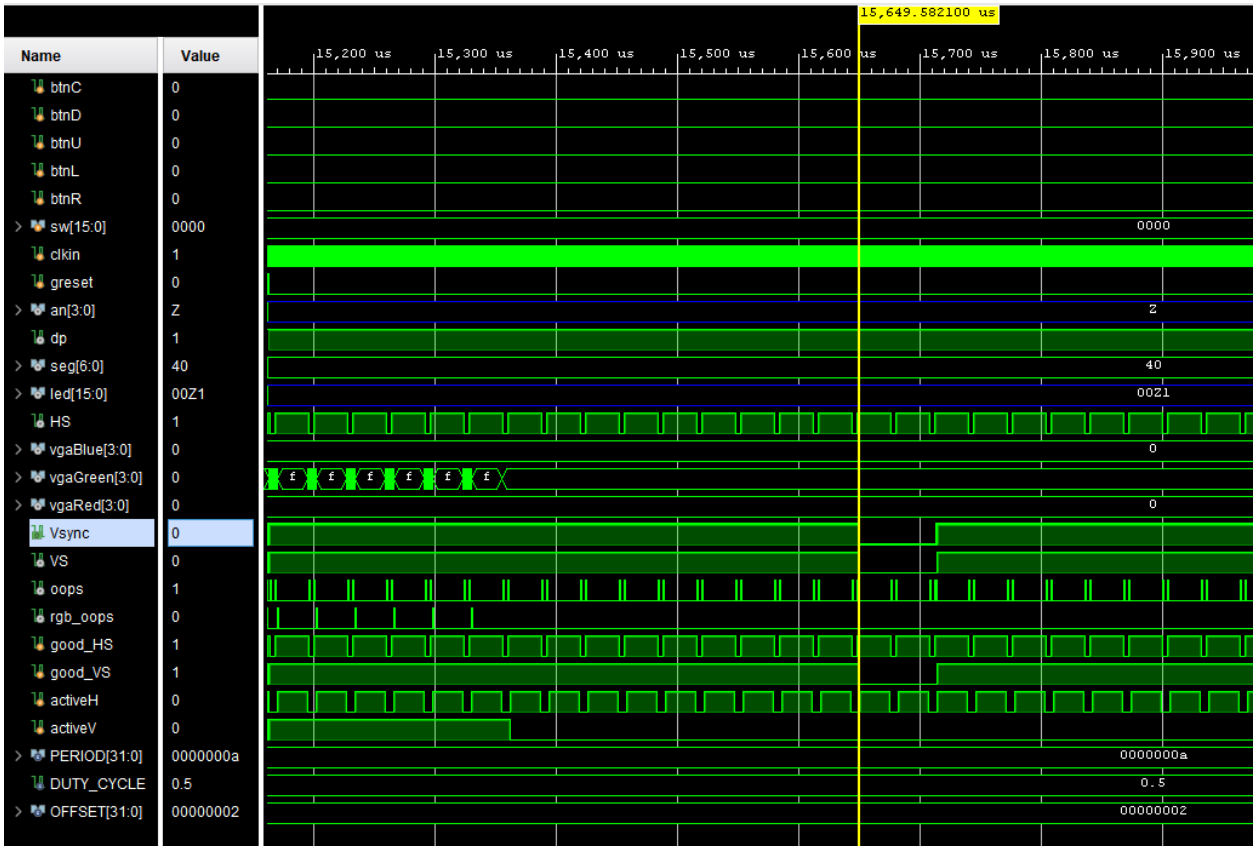


Fig. 4 Simulation in which the Vsync (vertical sync signal) goes low

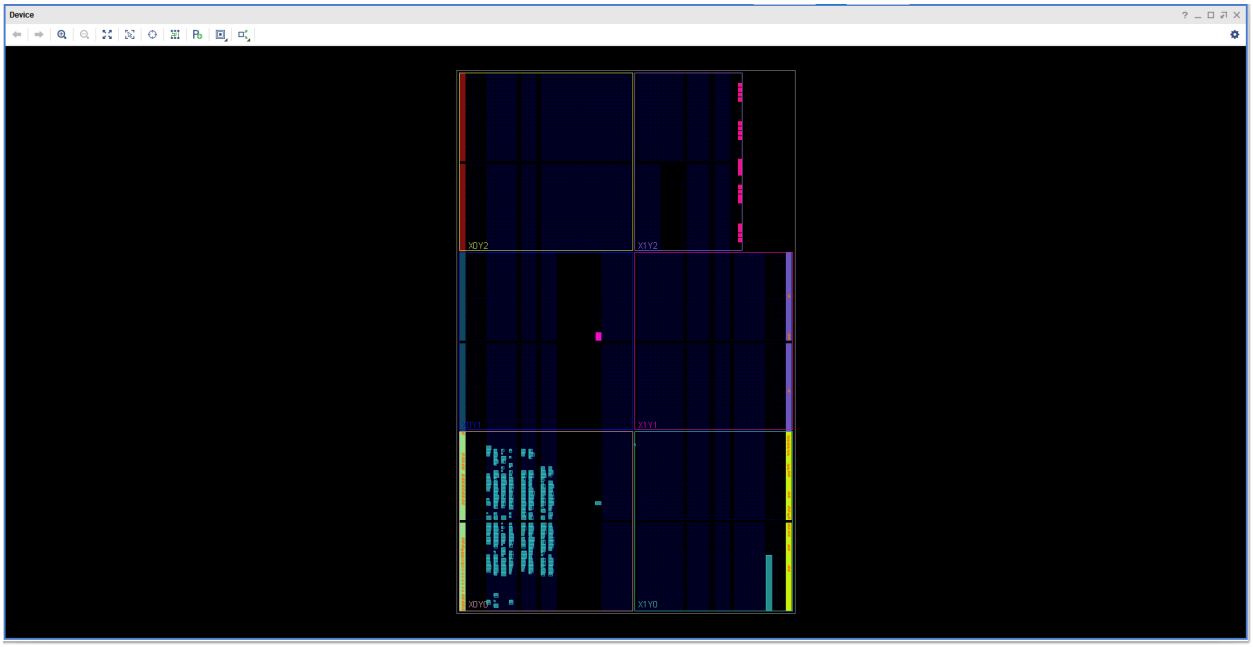


Fig. 5 Implemented Design

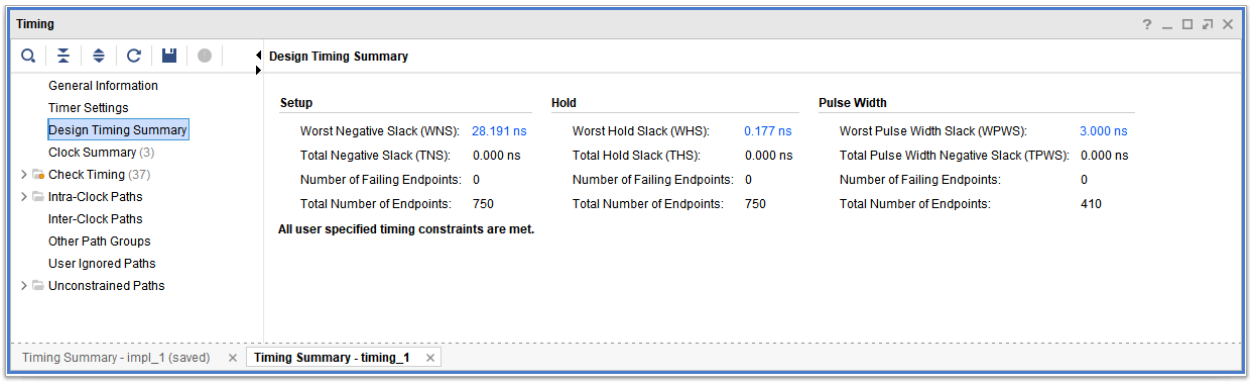


Fig. 6 Timing Summary pt. 1

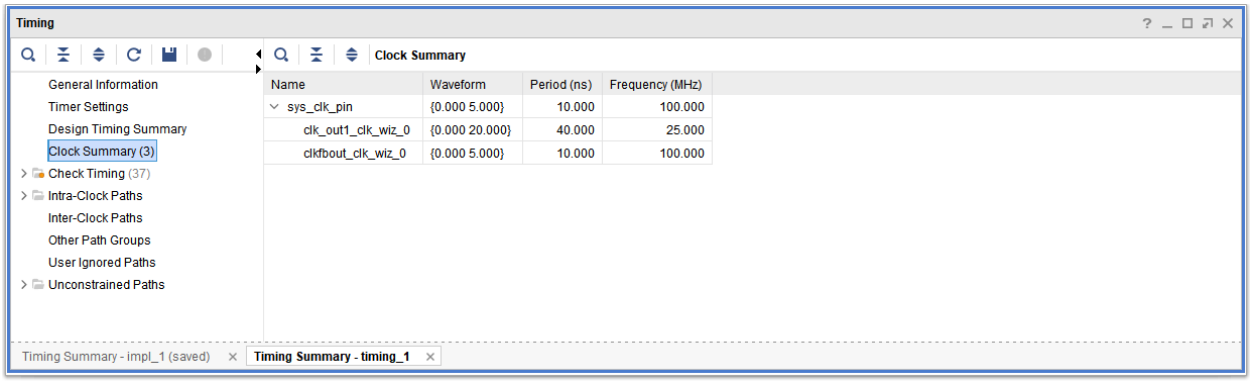


Fig. 7 Timing Summary pt. 2

As you can see from the timing summaries above, there are no red values and the Worst Negative Slack and Worst Hold Slack are both positive indicating that I am meeting the constraint. Fig. 7 shows that the external clock 100MHz ‘sys_clk_pin’ from the Basys 3 board is within range and the same goes for my 25MHz clock ‘clk_out1_clk_wiz_0’ which comes from the ‘clk’ port of the ‘labVGA_clks’ module.