Dwayne “The Block” Johnson
ECPE 174

Zachariah Abuelhaj
Computer Engineering
Controller and Gameplay
z_abuelhaj@u.pacific.edu

Jujhar Bedi
Bioengineering
Obstacles and Gameplay
j_bedi@u.pacific.edu

Suman Bhardwaj
Computer Engineering
Unicorn Wrangler
s_bardwaj@u.pacific.edu

Paul Vuong
Electrical Engineering
VGA and Gameplay
p_vuong2@u.pacific.edu

December 14, 2016

Table of Contents

Introduction

Field Programmable Gate Arrays, commonly known as FGPAs, allow users to program and configure an integrated logic circuit. When tasked with the responsibility to build a device on a FPGA, we felt inspired by classic Atari game cartridges, which were essentially programmed hardware. With further inspiration from *Pepsi-Man*, an original PlayStation title where the player must avoid streams of obstacles, we derived a much simpler version of the game. With the gameplay of the 90's and the technology of the 70's, we bring to you *Dwayne "The Block" Johnson*.

Problem Description

Our final project is to simulate a game with a FPGA, an Atari analog joystick, and a Video Graphics Array (VGA) monitor. Hardware Description Language (HDL) must be used to develop all aspects of the game. HDL also needs to be used to develop the timing and synchronization signals to drive the images onto the VGA display.

Design

VGA

A critical aspect of this project is driving the images onto the VGA display. To drive the images onto the display, VGA monitors require a combination of analog and digital signals. The analog signals represent color; red, green and blue. The digital signals are known as the synchronization signals, vertical and horizontal synchronization.

First, some background on how a VGA monitor operates. VGA monitors scan from left to right and top to bottom. Proceeding from the top-left corner of the screen to the bottom-right corner of the screen. In order to produce a flicker free image with smooth transitions from one frame to the next, a proper pixel clock speed is needed. For a 640 x 480-pixel resolution and 60Hz frame rate and accounting for the synchronization signals, a 25.175Mhz pixel clock is needed.

The horizontal and vertical synchronization signals determine the start and end of a horizontal and vertical scan. They are also preceded and succeeded with additional synchronization signals known as the front porch and back porch. Before a new horizontal line is scanned it is preceded with the front porch, horizontal synchronization, then back porch as seen in Figure 1. The process is identical for the vertical scan. These synchronization signals are also known as retrace. To start a new horizontal scan, the VGA display requires some time to transition from the end of one pixel line to the start of a new pixel line. The same goes for a vertical scan or vertical retrace. To start a new frame, the VGA display requires some time to transition from the end of a frame to the start of a new frame.

Red, green and blue pixels are then allowed to turn off or on depending on the desired color, but only within this viewable area. Between the end of the back porch and the beginning of the front porch. The color signals are an analog representation, therefore, the twenty-four-bit color signal routed to the onboard DAC of the FPGA, digital-to-analog converter, before output to the monitor connection.

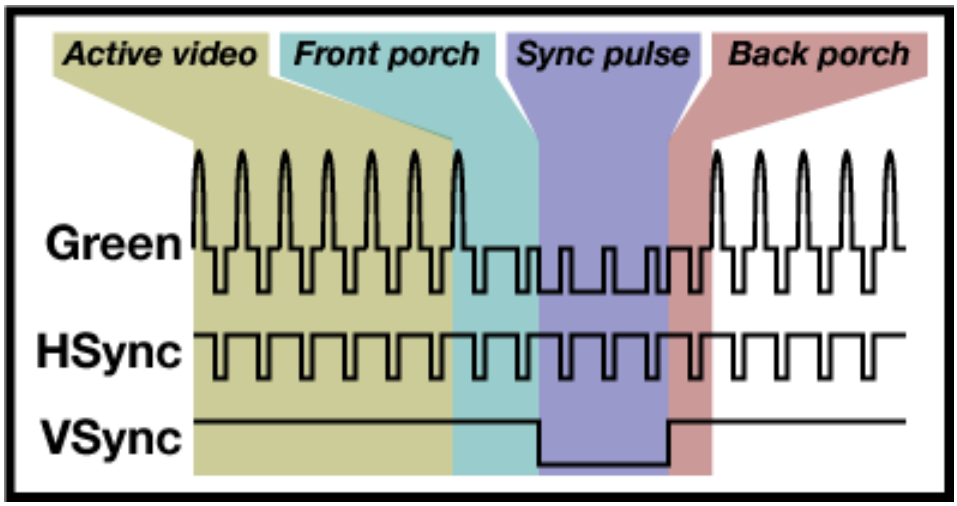

Figure 1: Horizontal and Vertical Synchronization signals

The onboard DAC also requires some additional signals to enable and disable color production. Color reproduction should not occur during horizontal and vertical retrace. All of which occur in our synchronization module; horizontal and vertical synchronization, front and back porch for both horizontal and vertical synchronization, horizontal and vertical enable signals for the DAC and rerouting twenty-four-bit color values to the DAC based on the status of the enable signals.

Proceeding on, since one DAC is available for color reproduction, there could only be one module to coordinate the stream of color data, twenty-four-bit data. This *road* module as we've termed it, sets the color parameters to the specified colors based on limits and enable signals routed to it from preceding modules. The limits are tripped with a feedback signal from the synchronization module, horizontal and vertical synchronization signals. They are in actuality counters operating on the pixel clock. The signals sending the limits and enable signals are produced by the game-play module. These signals coordinate where the character are and how the obstacles progress down the screen towards the character.

To produce the required pixel clock, Quartus' built in tool Qsys was used to generate a PLL. The Qsys tool generated required design files along with support files to drive the pixel clock.

Character

To move the purple character block left and right on the screen, we integrated an Atari 2600 joystick to the system. This specific controller was chosen for application due to its simplicity; all directional and button inputs were active-low analog signals output directly from pins. Because of the active-low output signals, application was easy for us to integrate, as the left, right, and ground pinouts were connected directly to the GPIO pins on the Cyclone IV board.

Since the signals were active low from button pads within the controller, a level-detection synchronizer circuit was coded for both the left and right directions of the joystick. For the gameplay, a level-detection circuit was chosen for ease of difficulty due to the 4 Hz clock speed of the obstacles streaming down the screen. If we had not coded the character block to default back to the center of the screen, moving for every position would be too tedious or frustrating for the player.

In order to actually move the character across the screen, a point-system was applied. At the start of the game, a Moore finite state machine sets the character at a default position in the center of the screen. Depending on the input signals from the joystick, the character block moves left or right, respectively. Shown in Figure 2 is the player state machine, where *idle* is the center position of the screen.

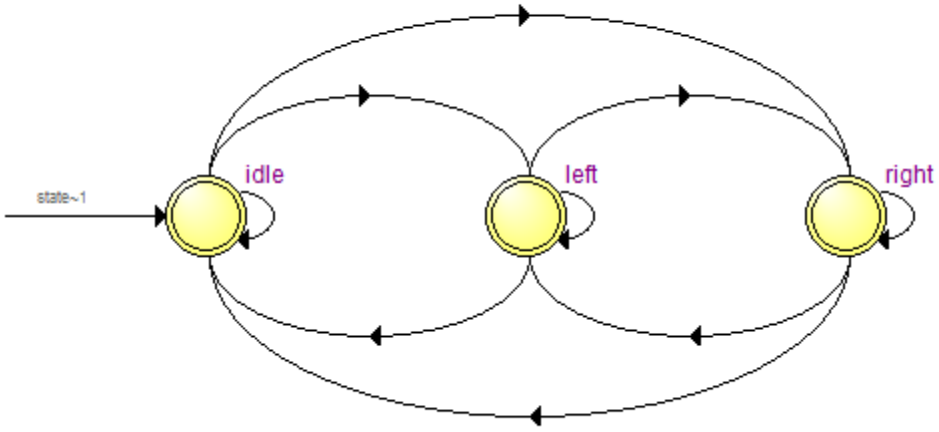

Figure 2: Moore FSM for the character module

Obstacles

Unlike the character module, the orange block obstacles are controlled by counters rather than a controller. Initially, two position signals, x1 and x2, are instantiated to a preset value, all of which can only be 170, 270, or 370. The y1

and y2 coordinates, however, both start at zero; as the counter increases, these points begin to move down the screen 100 pixels at a time.

To achieve the visual appearance of multiple obstacles scrolling down the screen at once, we delayed one obstacle on screen by 3 clock cycles. We integrated this clock system to the obstacles so that the second object would not scroll down the screen to the character until after the player avoids the first one. If not implemented, the game could become too difficult as there may be possible situations where the player cannot maneuver to safety.

Where *Pepsi-Man* included level design with an end-goal, ours did not. This was solely done to avoid hard-coding too many obstacle positions in the game. We instead ran a counter from 0 to 20, holding the x1 and x2 coordinates for 3 clock cycles. Once the counter reached 20, the logic resets the counter back to 0, allowing the player to conquest through the orange blocks until collision.

Collisions

The game only ends when the character collides with an obstacle; until a collision occurs, it repeats forever. In order to detect these collisions, we connected the output signals of the FSM in Figure 2 and the obstacle positions into combinational logic. Shown in Figure 3 is visual aid on how the gameplay logic determines a collision. For a collision to be detected, the obstacle moving down the screen must meet both of these criteria: it must be in the collision column and collision row. Once a collision is detected, a flag signal is set high and the game ends; otherwise a collision is checked on the next clock cycle.

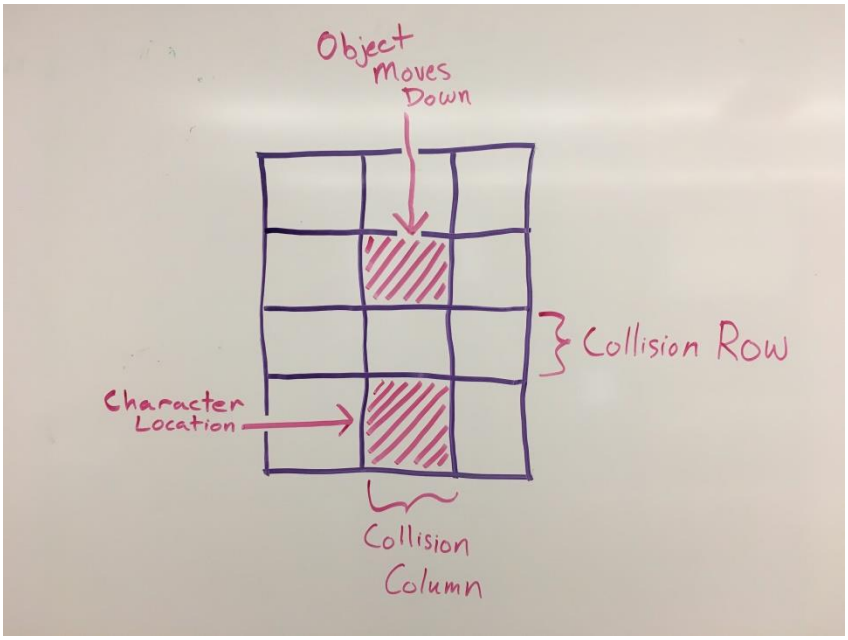

Figure 3: Character and object location diagram

Results

Final outcome of the design was a success. Synchronization module created the necessary digital and analog signals to drive the images on the display. Digital signals, horizontal and vertical synchronization signals feedback to the *road* module and served as a running counter and coordinated twenty-four-bit color signals to feed into the synchronization module. The game play module created the enable signals and limits to help define the size and color of the character and the obstacles. Also, the collision signal from game play module dictated if the game was still in play or had come to an end. Reset and start signals, initiated the game or if the game had concluded would reset the game. The pixel clock provided the necessary clock speed for the synchronization module. The two clock dividers within the game play module gave the game itself some variety by providing slower and faster version of the game. Controlling the character with the analog joystick also functioned as anticipated. Included in Figure 4 is the complete RTL viewer of the design.

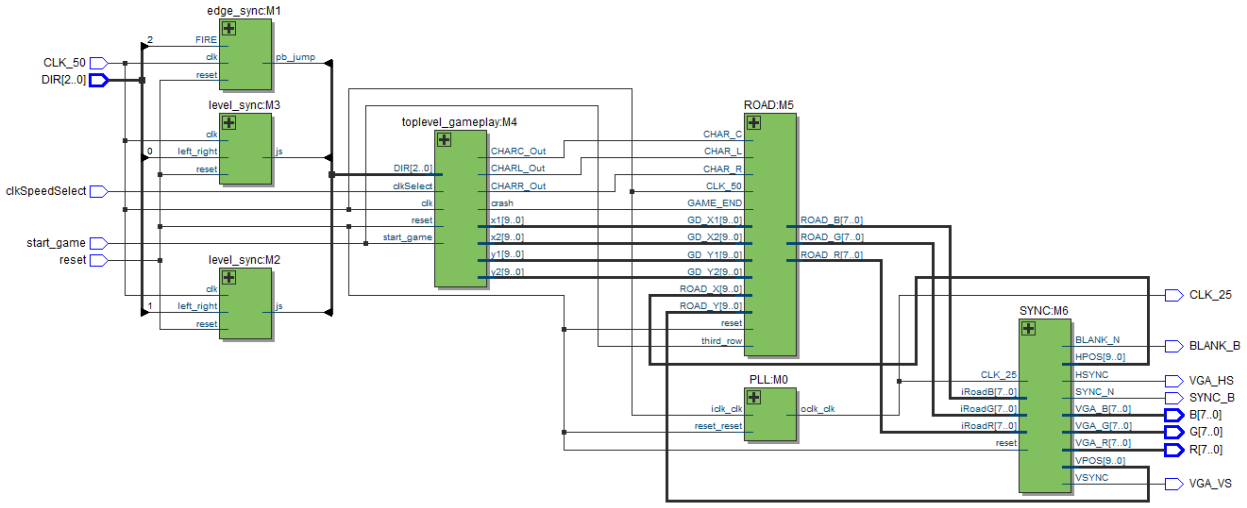

Figure 4: RTL viewer of the complete design

Testbenches

The first module that was simulated using ModelSim was the *obs* module. Due to the fact that the only signal input to the module is a clock signal, the testbench was straightforward as we just initialized and instantiated the input and output signals. The results of the simulation can be seen in Figure 5.

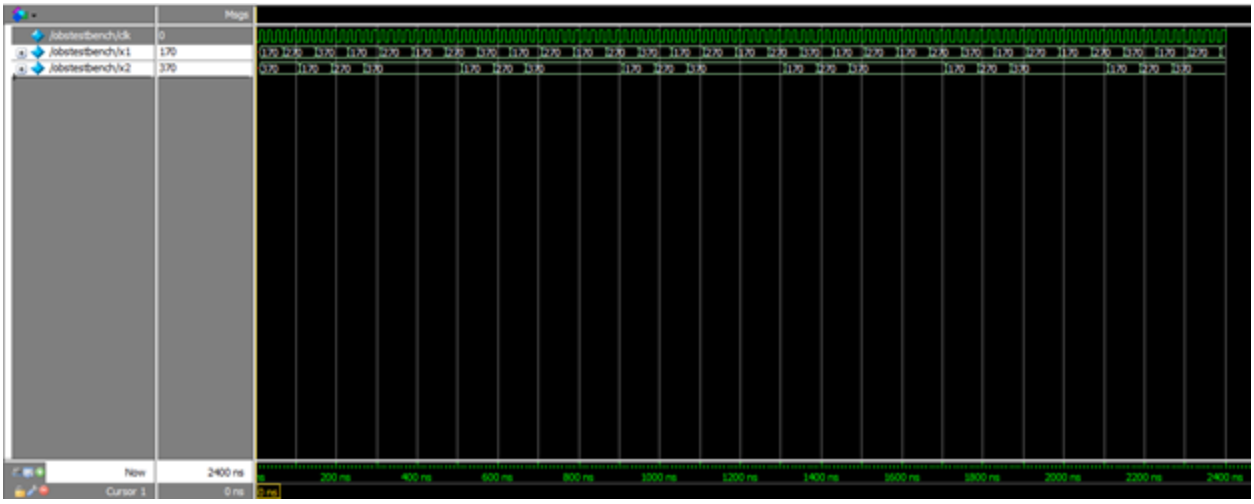

Figure 5: Simulation of the *obs* module

The next module that was simulated was the movement module, *movement*. Initially, both y positions start at zero, but then the y2 signal does not increment until the y1 signal reaches 200 by periodic displacements of 100 pixels. As mentioned previously, while one signal increments, the other remains zero. The results of the ModelSim simulation are provided in Figure 6.

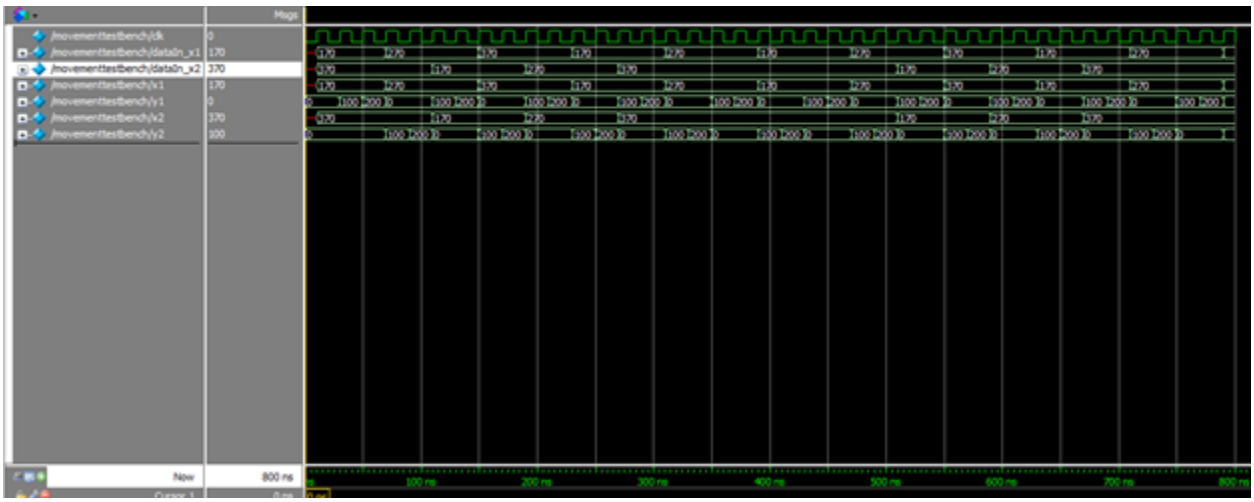

Figure 6: Simulation of the *movement* module

The last simulated module was *character*; the simulation results can be seen in Figure 7. For this testbench, the directional input from the controller move the character position either left or right, where 170 is left, 270 is center, and 370 is right.

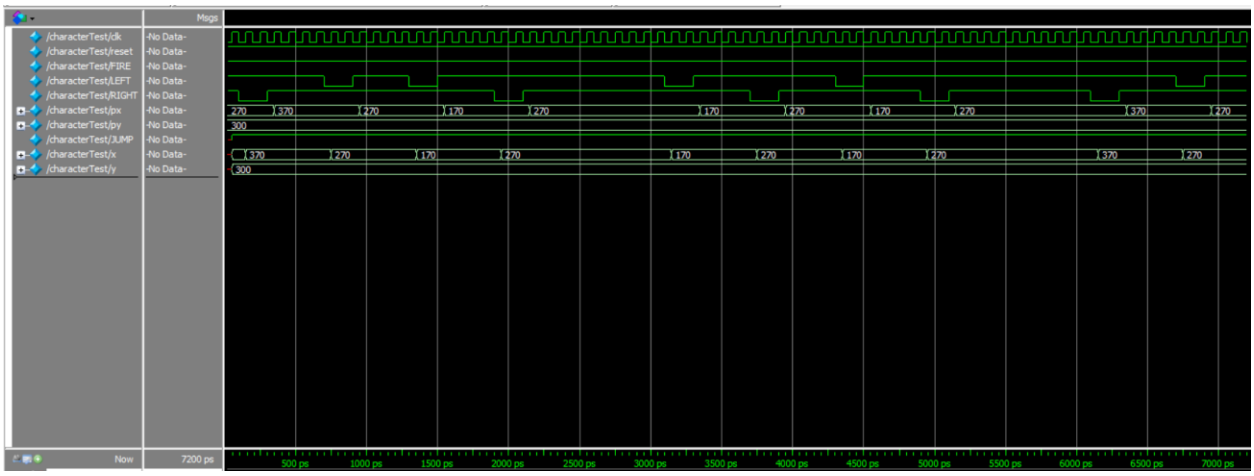

Figure 7: Simulation of the *character* module

Prior to final simulation, the *collision* module was tested. If a collision is detected, then the *collision_occured* signal goes high. After a collision occurs and once the game ends, reset inputs to the module as high, starting afresh for another play-through. The simulation of this module can be seen in Figure 8.

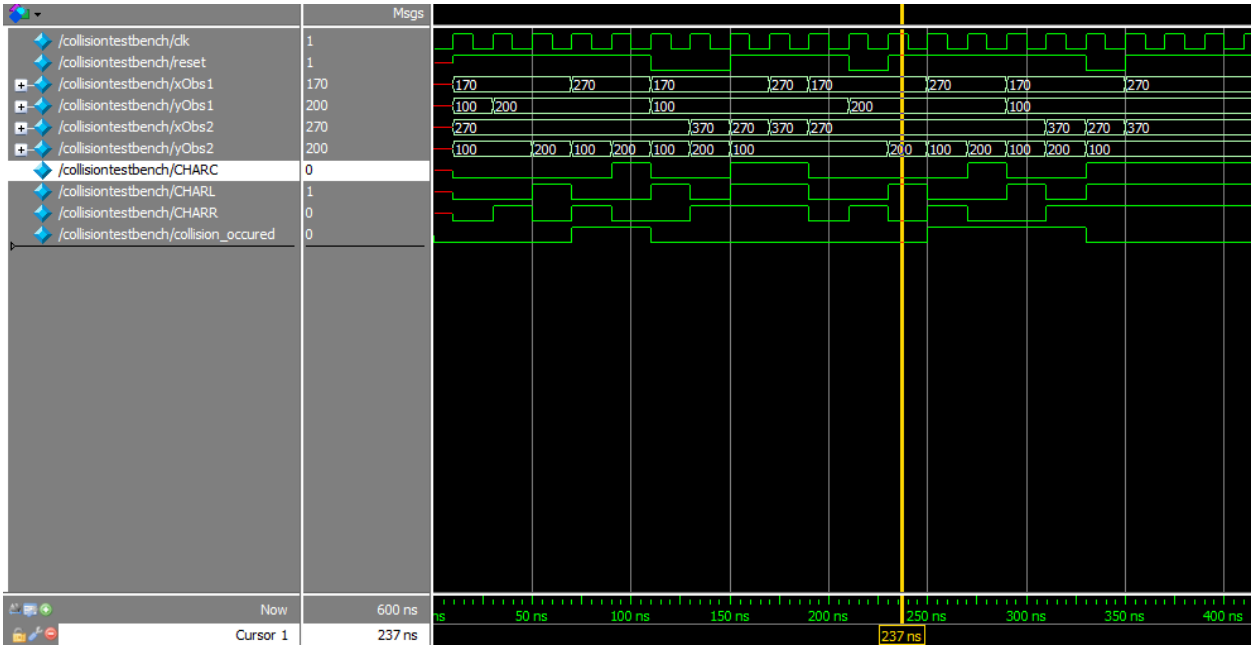

Figure 8: Simulation of the *collision* module

Lastly, all the aforementioned modules were tested together in a *toplevel_gameplay* module. The testbench for this randomized the *reset*, *start_game*, *clkSelect*, and *DIR* signals. Since all the other modules had already been tested beforehand, there was no difficulty in simulating this design. The results of this simulation are shown in Figure 9.

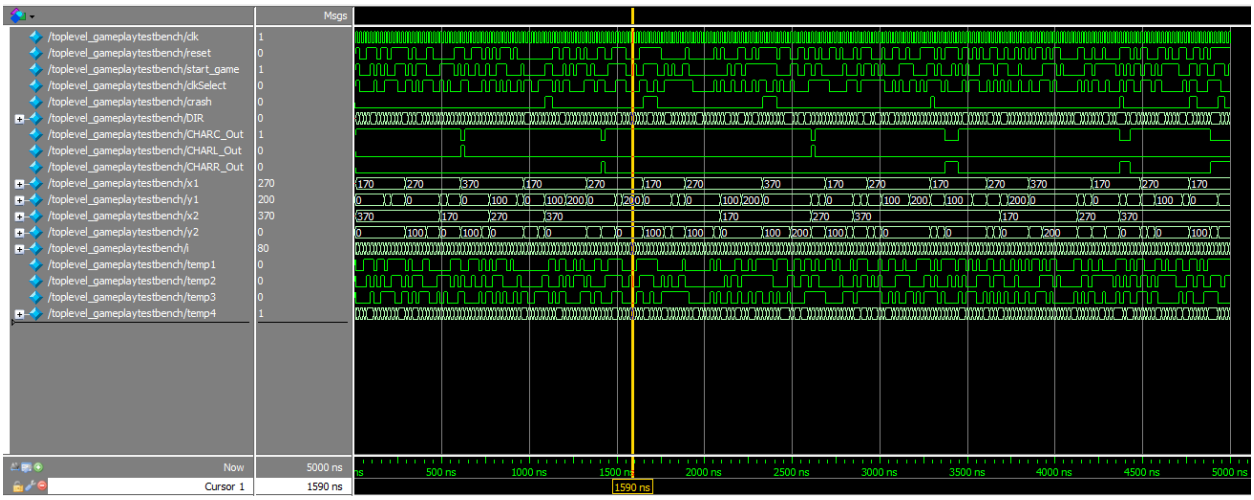

Figure 9: Simulation of the *toplevel_gameplay* module

System Analysis

Discussion

Through successful simulation of all gameplay logic, we were capable of creating a fully functional game with a controller and display. The end result additionally consisted of an on/off switch, a game-start switch, and two game modes for more fast-paced play. That being said, there was much more intended for this project. Originally, there was to be push-button input from the controller to jump over obstacles as well as an over-the-shoulder perspective. The initial plan with over-the-shoulder gameplay included obstacles being more elaborate and scalable; as objects moved closer to the player they would increase in size. Obstacles were intended to be completely randomized, but due to time constraints, this feature was chosen to be left out. Given the opportunity to do so, randomization for less predictable gameplay would be a top-priority to implement.

Task Breakdown

Name	Design	Testing	Poster	Report
Jujhar	*obs, movement, collision, toplevel_gameplay*	*obs, movement, toplevel_gameplay*	Not Applicable	Testbenches, Results, Obstacles, Collisions
Zach	*level_sync, character,* VGA assistance	*level_sync, character*	Not Applicable	Introduction, Character, Obstacles, Collisions, Testbenches, Discussion
Paul	*road, pll, all things VGA*	Not Applicable	Not Applicable	VGA, Results, Problem Description
Suman	VGA assistance	Not Applicable	Everything related to poster	Not Applicable