Joseph Conenna

EEL 5722 – FPGA Design

11/21/16

Yoshi's Nightmare: An FPGA Video Game in Verilog

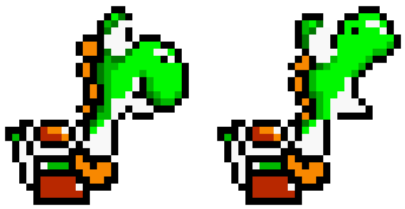
Introduction

Video games are typically designed around a microprocessor that is embedded within a console or computer system. While this is the typical case these days, it is not the way that video games have always been crafted. In its infancy, the first video game experience was that of a fully analog system which implemented a crude form of table tennis on the quaint black and white CRT televisions of the time. With the introduction of TTL and then CMOS logic chips, there existed a new toolset to create simple logic games. The introduction of the microprocessor and its evolution over the many decades has provided a powerful and flexible platform for which to design video game systems and to write software around.

Why would one consider implementing a video game with the reconfigurable logic of an FPGA? There are many ways that FPGA technology can help in the realm of video games. Graphics co-processors can be developed and tested using FPGA technology, providing a more flexible and affordable testing platform than the ASIC. In consideration of small toys and low cost game devices, FPGAs can be used to test standalone digital systems that can later be translated to an ASIC or gate array design and mass manufactured to provide an affordable and fun experience for the consumer. Beyond these possibilities, the intellectual challenge of coordinating a digital system that interprets inputs from a game controller and output graphics to a VGA monitor is very intriguing to me, and I have strived to make the experience as compelling as possible.

The Game: A broad overview

The game that I envisioned when beginning this project revolved around a small green dinosaur named Yoshi, a popular character from Nintendo's Mario series. The first objective was to fully animate his actions, allowing the player to make him run around, and jump up and down from platforms on the screen.

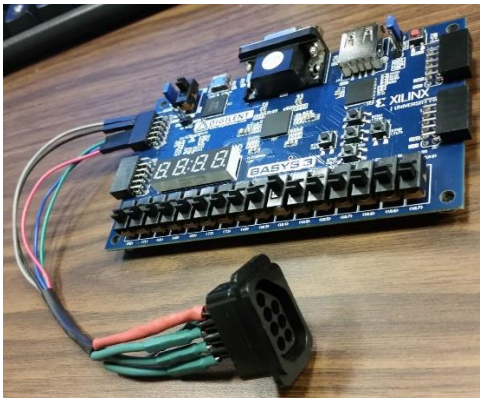


This involved fully implementing a form of 2D Physics for his motion in the x and y dimensions on the screen, keeping in mind conservation of momentum and accelerations, as well as collisions with objects, walls, and platforms.

Next there needed to be reasons for Yoshi to move about, and these are eggs and ghosts. Eggs are objects which Yoshi usually collects in his classic games, so I made his purpose in my game to collect randomly placed eggs on the screen to gain points. Ghosts are traditional enemy characters from the Mario series, and were a fitting object for use as an enemy in the game. These ghosts chase Yoshi around as he collects eggs, and introduce a sense of urgency and challenge to the otherwise simple task at hand.

To create the environment for the game, stationary platforms and an outer wall around the screen are drawn on the monitor, and a separate collision detection circuit is used to determine when Yoshi encounters them. Other assets to the game, such as score display, life hearts display, gameover display, background, as well as title screen were implemented at the end of the design as finishing touches to the game.

To play the game, I decided to use a classic Nintendo NES controller, as this would provide sufficient buttons to control the gameplay and would add to the retro feel of the game. I harvested a female controller port from a scrap Nintendo console and soldered jumper wires from the pins to interface with GPIO on the FPGA board. The controller uses a serial protocol for communicating button states, and therefore a receiver module needed to be implemented to use the controller with the game.



I based the project around the affordable and capable Basys 3 development board which uses a Xilinx Artix-7 series FPGA, with 33k logic cells, and 1800 kbits of block RAM. The board has “p-mod” connectors which allow for outside input, as well as a seven-segment display to display the score on.

Logic Design: Introduction

It is not possible to detail each module without mentioning modules with which they are interconnected with. Because of this I will list each module in the entire design here, with concise descriptions:

background_ghost_rom: ROM template for ghost background on screen.

binary2bcd: module to convert binary representation of score to 4 BCD values.

blocks_rom: ROM template for 16x16 block tiles used to form platforms.

display_top: top module for entire design.

eggs: module to control egg placement, collision of Yoshi and egg, score processing, and coordinating RGB data from ROM.

eggs_rom: ROM template for 5 different colored 16x16 egg tiles.

enemy_collision: module receives input of location of Yoshi and Ghosts and outputs a signal HIGH when a collision occurs.

game_logo_display: module to display title screen game logo when in “idle” game state.

game_logo_rom: ROM template for title screen game logo image.

game_state_machine: coordinates game states (“idle”, “playing”, “hit”, “gameover”) depending on various inputs.

gameover_display: module to display gameover image in “gameover” state.

gameover_rom: ROM template for gameover image.

ghost_bottom: module and FSM for “bottom” ghost enemy.

ghost_crazy: module and FSM for “crazy” ghost enemy.

ghost_crazy_rom: ROM template for “crazy” ghost enemy 16x16 tiles.

ghost_normal_rom: ROM template for “bottom” and “top” ghost enemy 16x16 tiles.

ghost_top: module and FSM for “top” ghost enemy.

grounded: module of combinational logic to determine when Yoshi is grounded on a platform or floor.

hearts_display: module to display number of hearts (lives) Yoshi has, out of 3 lives.

hearts_rom: ROM template for hearts tiles.

nes_controller: module and FSM to process data lines from NES controller and signal button presses.

numbers_rom: ROM template for 8x8 tiles for numbers 0-9.

platforms: module to draw tiles from blocks ROM on screen to form platforms and walls.

score_display: module to display the appropriate score on the screen and 7-segment displays.

vga_sync: VGA synchronization module to drive vertical and horizontal sync lines, provide pixel clock, and current x, y pixel location on the screen.

walls_rom: ROM template for 16x16 wall tiles used to form walls.

yoshi_ghost_rom: ROM template for 16x16 composite tiles to form Yoshi, made to have him turn white, when hit by a ghost.

yoshi_rom: ROM template for 16x16 composite tiles to form Yoshi, when idle, walking, jumping up and down.

yoshi_sprite: module and FSM's to control Yoshi's motions: idle, walking, jumping, etc.

Sprites and ROMs:

Having visuals for the ROM images the previous modules coordinate and display will make detailing them further much easier. In my project, I opted to use Verilog synchronous ROM templates, as described in the Xilinx XST Design Manual, to provide a complete set of files that could be easily turned into a working bitstream for the game.

```
module rom
    (
        input wire clk,
        input wire [3:0] row,
        input wire [3:0] col,
        output reg [11:0] color_data
    );

    (* rom_style = "block" *)

    //signal declaration
    reg [3:0] row_reg;
    reg [3:0] col_reg;

    always @(posedge clk)
        begin
        row_reg <= row;
        col_reg <= col;
        end

    always @*
    case ({row_reg, col_reg})
        6'b000000: color_data = 12'b011011011110;

        ...

        default: color_data = 12'b000000000000;
    endcase
endmodule
```

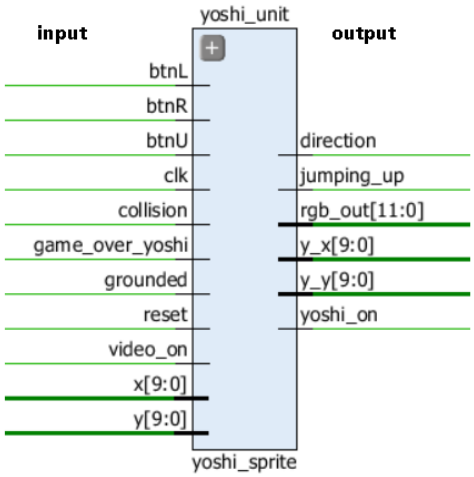
Above is a sample structure of the adjusted ROM template I used to infer each ROM. While it is left up to Vivado how to translate the ROM data into the Block RAM on the chip, a row, column indexing scheme is used to make indexing into the ROM more intuitive. I wrote a Python script that reads in an image and automatically generates

the ROM template for that image, converting the pixel data down to the 12-bit color scheme of the Basys 3 board. I also modified this script to generate a monochromatic representation of the numbers_rom to save space. Below is a diagram showing each ROM that is used in the game.

In some ROMs, there is a specific cyan background color, and is used within the respective display module to know when to display the relevant sprite pixels, or to not display anything and let the background ROM be displayed behind it.

Yoshi:

The module receives inputs from the Up, Left, and Right button signals from the nes_controller module, and collision and grounded signals from the respective modules. These five signals are what control Yoshi's entire motion on the screen. Yoshi's direction (Left, or Right), x/y position on the screen, as well as jumping_up state are outputted from the module and used elsewhere, such as determining the input collision condition.



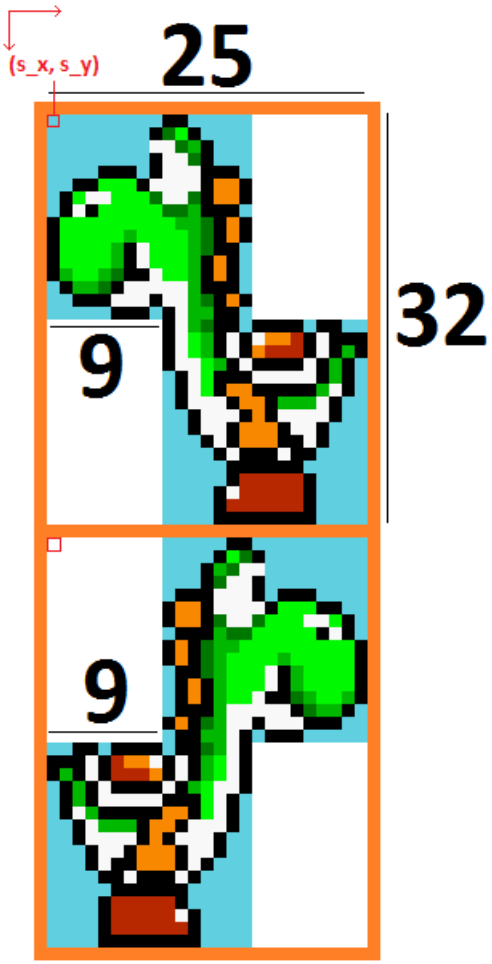
There are two main FSMs in the module that separately control the x motion and momentum, and the y motion and momentum as well as standing/walking states. To simulate the gold standard of 2D video game kinematics (see Super Mario Bros. 3) in the x-dimension, acceleration, terminal velocity, and mid-air change in momentum must be accounted for. In the pixel world, distance is quantized into pixel units. To simulate 2D physics, the time between discrete movements between pixel locations must be a dynamic variable. For instance, acceleration to the right would imply that each successive time delay between moving one pixel to the right would be less and less each time. A register, named x_time is used to provide a dynamic delay between left and right pixel motion. This register is loaded with the contents of the register x_start, the starting value, and counts down to 0, upon which the sprite moves one pixel and the start value is loaded again. By fine tuning the value that is counted down per the system clock, or how fast Yoshi should move, and how this value updates with acceleration and collisions, Yoshi can begin to exhibit a natural motion in the x dimension.

The next-state logic for this FSM is very complex, and is heavily commented to document the details in the source code. The goals achieved by it are that when Yoshi is on a platform or ground, and he moves left or right, he accelerates quickly to a top speed. When the direction of motion changes on the ground he is nimble and can immediately begin moving in the opposite direction. If the left or right button is pressed Yoshi will be running on the ground, and when it is let go, he will immediately stop. Contrary to this, while Yoshi is not grounded and is midair his momentum will be conserved as he follows a natural projectile motion. The only exception to this is allowing the player to slowly adjust his x direction of motion midair by pressing the opposite directional buttons.

The second FSM coordinates accelerations in the y-direction from jumping, as well as Yoshi's appearance in the jumping, walking, and standing states. If the Up button is pressed when Yoshi is grounded he will begin to jump up and accelerate in positive y direction. There is a separate register that counts how long the Up button is held when jumping and determines how high Yoshi will go until reaching the peak of his y motion, upon which he will begin to move in the negative y direction, until reaching a terminal speed. While Yoshi is jumping up, he can move through platforms, and only falls onto a platform and lands when his feet contact them and the "grounded" input signal for the module is asserted.

The "jump_up", "jump_down", "standing", and "walking" states of this FSM also adjust how the bottom tile for Yoshi's torso is indexed from within the ROM. To save space, the head tile of Yoshi remains constant, while his torso is adjusted depending on what he is doing. The offset of the head and torso tile are constant, but depend on

which direction he is facing as well. When the left button is pressed, the directional dir_reg is updated to LEFT, and the sprite is drawn as given in the ROM with the torso ROM drawn 16 down and 9 to the right of the upper left corner of the head tile. When the right button is pressed, the directional dir_reg is updated to RIGHT, and the sprite tiles are drawn mirrored on the screen. The location for Yoshi is held in the s_x and s_y registers and is always the upper left pixel of the composite Yoshi image on the screen.

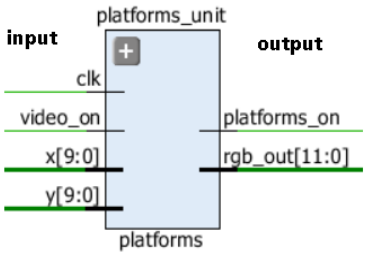


The collision signal that is inputted into the module is also used to determine which Yoshi ROM to index into, either the normal ROM for when he hasn't been touched by a ghost, or the yoshi_ghost ROM for when he is hit by a ghost. He maintains the ghost appearance for 2 seconds, while he is invincible, as described later.

The game_over_yoshi input signal is used to set when Yoshi can move, making him stationary in the gameover state.

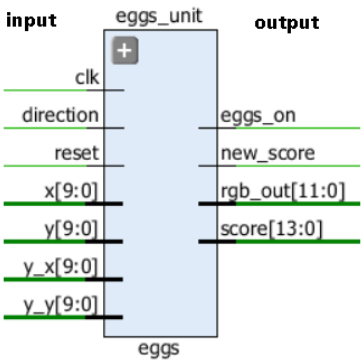
Platforms and Eggs:

The platforms module contains combinational logic that determines where platforms or walls are drawn on the screen.



It receives the x/y pixel location of the current pixel being drawn on the monitor, and outputs the corresponding rgb data for when the pixel is located within a platform. There are 6 “platforms” that Yoshi can stand on and that eggs can spawn onto. There is also an outer wall that contains Yoshi within the monitor screen.

The eggs module receives input of both Yoshi’s location on the screen as well as direction. This data is used to determine when Yoshi collides with the current egg on the screen. The collision logic takes into account that Yoshi’s location is given as the upper left corner of his composite image, and that this composite image contains two areas of empty space where a collision should not occur.



The location of the first egg is constant, but the location of subsequent eggs on the screen is pseudorandom and depends on when Yoshi collects the previous egg. The 8-bit platform_select register increments each clock cycle and is used to determine which platform the egg will spawn at when Yoshi collides with the previous egg. Each platform then has a separate register that continuously cycles between each platform's x range. When Yoshi collides with an egg, the location of the next egg to spawn is at the platform determined by the platform_select register, and specifically along that platform depending on its own location register value. Because the aforementioned registers are clocked at 100 MHz, it is practically impossible to know where the next egg will appear.

The type of egg that spawns is also pseudorandom, and depends on the lower 6-bits of the platforms x-location register. This value is stored in the egg_type register when the egg is spawned, and determines the offset into the ROM for the color of the egg being displayed, as well as it's associated score value. The probability of each egg being spawned is different, with green eggs being most common and least valuable, and the rainbow egg having a 1/64 chance of dropping and with the highest value.

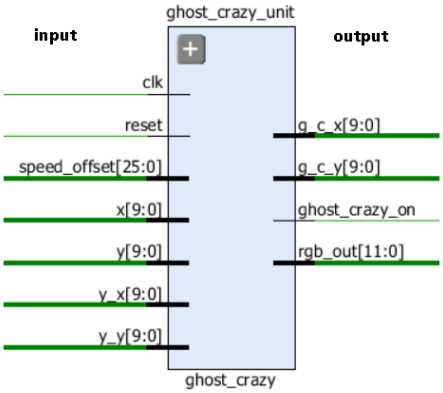
The score for this game is between 0 and 9999, and this value is held in the score register within the eggs module. This score is updated when a collision between Yoshi and the current egg occurs, and the new_score output is asserted, letting the binary2bcd module know when to convert the new score for display.

Ghosts, Collisions, Grounded:

There are three ghost enemies on the screen that track Yoshi's location and chase him around the screen. Two ghosts are relegated to the top and bottom sections of the screen and only chase Yoshi when he is in their respective regions. They have the "normal" ghost appearance as shown above, and track Yoshi at a speed that is proportional to the score. There is one "crazy" ghost that chases Yoshi anywhere in the screen, and tracks Yoshi in proportion to the score, but at a more rapid speed than the other two ghosts. I will detail the "crazy" ghost module, as the other two are very similar in design.

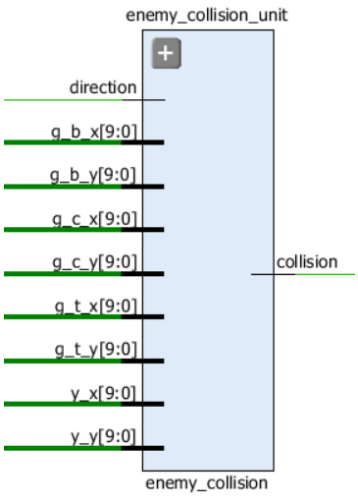
The module receives inputs for Yoshi's location on the screen which are directly used to determine the Ghost's next move after a certain delay. The delay is determined by the time_reg, which loads the value to countdown to 0 before updating the ghost's position. The input "speed_offset" is determined from the score outside the module, and is subtracted from a max value given to time_reg, such that as the score increases, the speed_offset increases, and the delay between movements becomes shorter. The values of the starting countdown value and offset

calculation were tuned to make the game easier in the beginning and to cap out at a very challenging pace to ensure the player would never make it to a 9999 score.

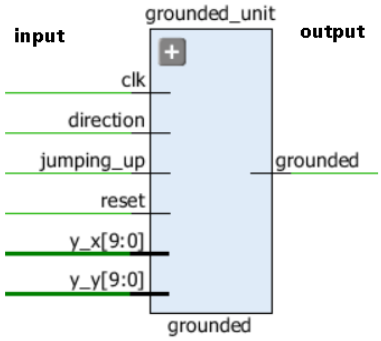


The ghost's x and y location is stored in two separate registers, and this value is compared to Yoshi's location to determine what move to make next. The ghost always moves in the x and y direction that will decrease the distance between itself and Yoshi, until the ghost's location on the screen is the exact same as the location of Yoshi. Keep in mind, the location of these sprites is stored as the upper left corner of their image on the screen. By comparing the x location of the ghost and Yoshi, the ghost is also made to face towards Yoshi as it moves towards him.

The x and y location of the ghost is outputted to feed into the collision detection circuit. This is the case for all three of the ghosts. The collision detection module receives the x/y locations for Yoshi and all ghosts, and implements combinational logic to determine if any ghost is drawn within Yoshi's body. The collision logic was finetuned to assure that a collision only occurs if a ghost touches Yoshi's body, not the empty space in the composite rectangular image. There is also some slack which allows the two images to touch to a certain extent to not count a few pixel overlap as a collision, in order to make the game more forgiving to the player.



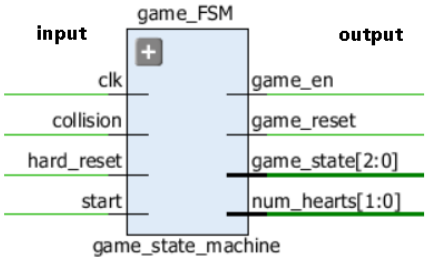
The grounded module is like enemy_collision in that it employs combinational logic to assert a grounded signal when Yoshi is on a platform or has dropped down onto one. This signal is used by Yoshi's module to determine when he can start falling off a platform, or when he stops falling and lands on one.



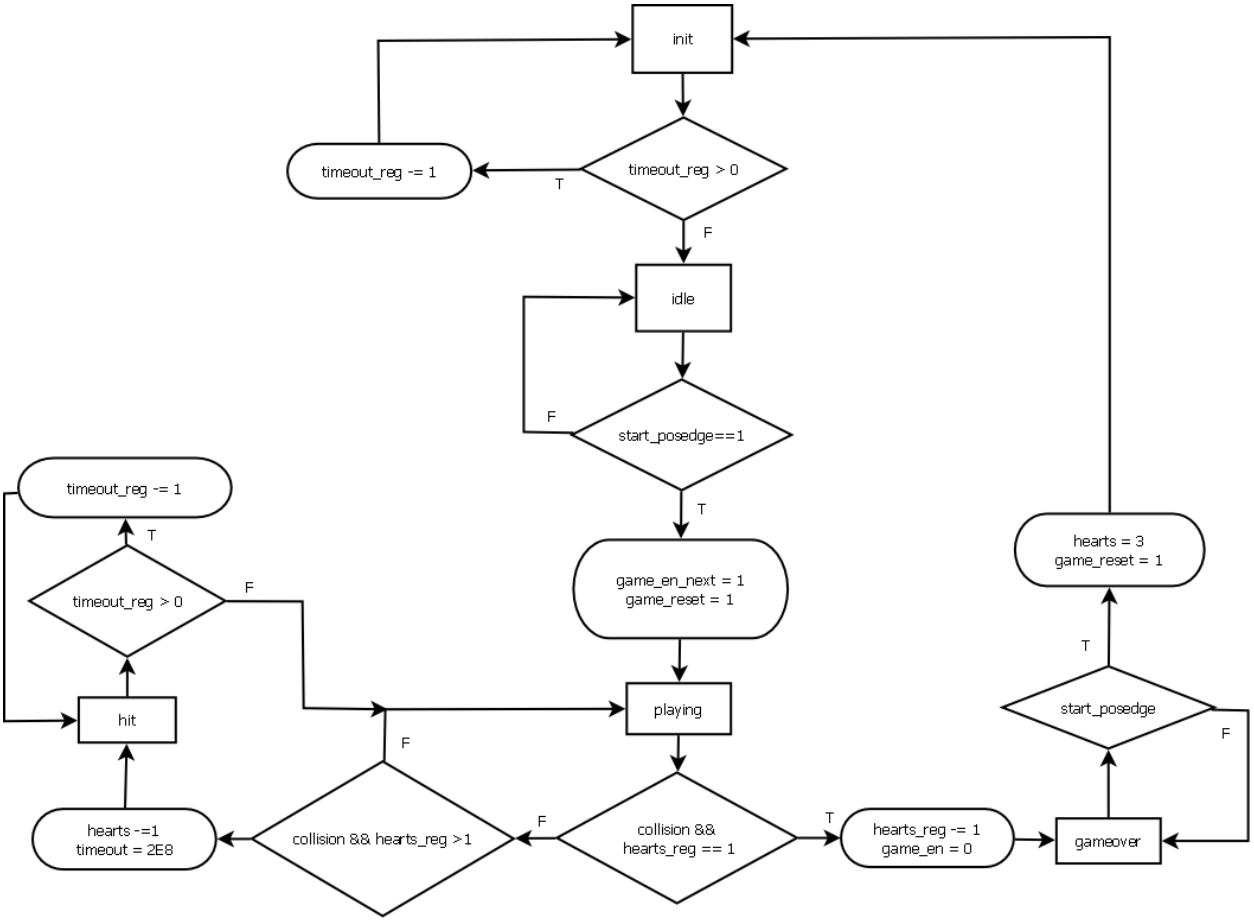
Game State Machine:

With the implementation of the above modules, the core structure of the gameplay has been designed. The game_state_machine module takes care of the overall states of gameplay, such as the idle screen, playing, hit by a ghost, and gameover events. The start input is the signal from the start button on the NES controller and is used to transition from "idle" to "playing", and from "gameover" to "idle" states. The collision signal lets the FSM know when to transition from the "playing" state to the "hit" state. The hit state is meant to provide a delay where Yoshi gets 2 seconds of invincibility from ghosts after being hit by one. This invincibility is controlled and the "hit" state detected for in the top module.

Yoshi has 3 hearts, or lives, and getting hit will decrement the number of hearts he has. This number of hearts is output from the module and used to display the correct number of hearts on the screen by the hearts_display module. The game_en output signal is used to determine when Yoshi can be controlled by the NES controller, and stops him from being controlled in the gameover state. The game_reset signal is used to reset registers in other modules to start the game assets from their initial conditions.



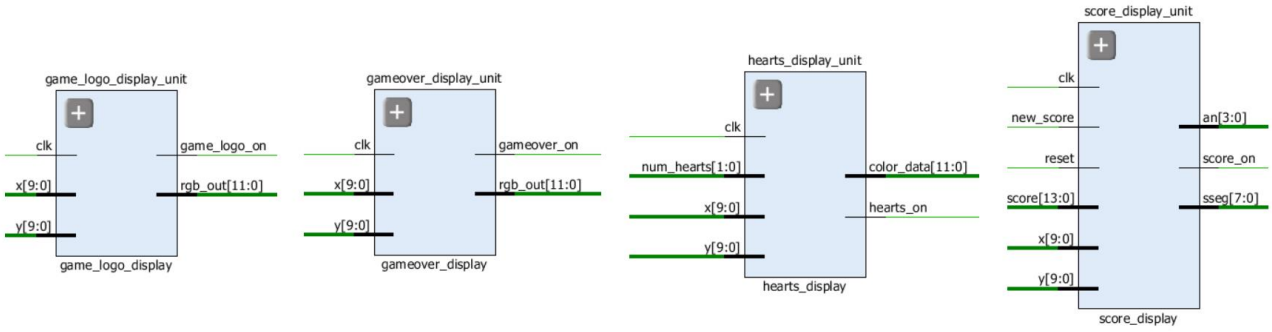
Below is the FSM state diagram for the game, with squares representing states, triangles representing conditions, and ovals representing actions based upon conditions being met or unmet. The “init” state is a short delay that allows the NES controller to initialize without affecting the game state.



Game Logo, Gameover, Hearts, Score Displays:

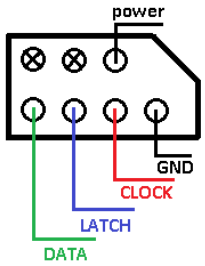
In the “idle” state of the game, the game logo is displayed on the top of the screen, while in the “gameover” state the gameover image is displayed. When these images are displayed is determined in the section of the top module where the various elements are drawn to the screen.

The hearts_display unit receives a signal for num_hearts (0-3) and displays three hearts or empty heart containers depending on the value. The score_display module receives the score and new_score signal. Upon receiving a new score, it uses the binary2bcd module to convert the value to four BCD values to display the four digits of the score on the screen, as well as the seven-segment displays on the Basys 3 board.

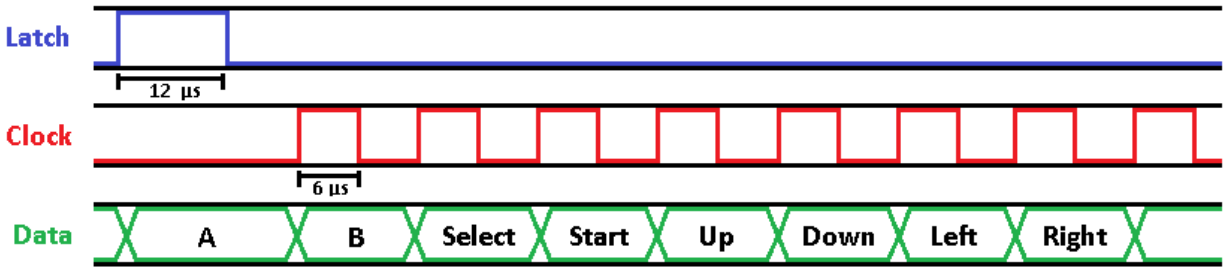


NES Controller:

The NES Controller uses a serial data protocol that is communicated over three lines: latch, clock, and data.



The host controls the Latch line, and holds it high for 12 µs to latch the button states of the controller to an 8-bit shift register to sample them. After the 12 µs latch pulse, the latch line is held low thereafter, the data line is sampled by the host as the 'A' button state, and 6 µs delay elapses. After this delay the host sends eight 6 µs pulses over the clock line, with 6 µs delays between each pulse. The data line is sampled at the end of each clock pulse, and read in sequentially as the 'B', "Select", "Start", "Up", "Down", "Left", and "Right" button states.



The nes_controller module contains a FSM that continuously latches, samples, and outputs the button states. The button states are registered, so previous states are held until they are updated. The only buttons used from the controllers in the game are Start, Left, Right, and A, where A is used as the Up signal to make Yoshi jump.

Top Module, Usage Report, and Conclusion:

The top module instantiates all working modules and routes signals between them according to the design. The speed_offset used by the ghosts to adjust their speed with the score value is calculated as the top 11 bits of the score multiplied by 4096, up until the offset reaches the max value to avoid the game becoming impossibly hard. The rgb signals that are output from each game element module are drawn to the screen in an if-else-if chain that gives priority to elements higher up in the chain being drawn first on the screen. Any element at the same pixel location with a lower priority does not show up at that pixel. This allows objects to overlap and cover each other in the game.

The design utilizes 2122 LUTs for logic, none for memory, and 996 kbits of block RAM for the image ROMs. The background image being 256x256 and logo image being 35x368 cause the synthesis optimization process to nearly exceed an hour. Because of this, these ROMs were omitted during the design and testing stage in order to speed up testing new iterations of the design.

The end result is a game that feels quite good to play and offers a challenge for those who play it. The objective is to score as high a score as possible and therefore survive as long as possible. An improvement to the design would be to interface with an EEPROM in order to keep a log of top scores for the game. Another improvement would be to interface with a sound module in order to generate sound effects for the game.

Implementing a simple video game in HDL forces one to consider every aspect of the game while designing it. The benefit to implementing a game with an FPGA is that a single chip is used to hold the video ROM, gameplay logic, control interface, and video driver circuit. Typical retro consoles instead used a processor, video RAM, working RAM, ROM, and separate controller interface chips to achieve the same results. While this project is by no means up to par with modern video games, it still provides an insight into what it takes to implement a simple 2D game with basic physics and rules.