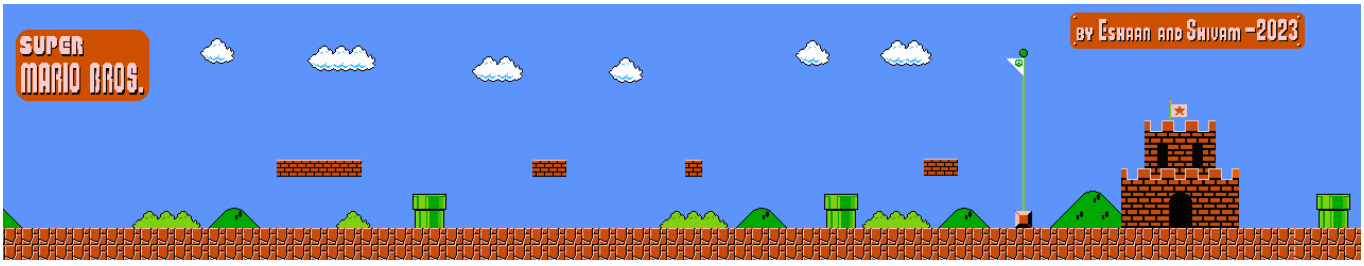
Final Project Report: Super Mario Bros

Spring 2023
Eshaan Tibrewala (eshaant2) and Shivam Patel (shivamp6)
TA: Shitao Liu

Introduction

For our final project, our goal was to design Super Mario Bros, the popular Nintendo game. Super Mario Bros was initially released in 1985 as a platform game for the Nintendo Entertainment System (NES). Since its release, Super Mario Bros has become one of the most popular platform games in the world and is enjoyed by children and adults everywhere. The original Super Mario Bros can be played as single or double player, but for our lab, we only focused on single player functionality. In the single player version, the user controls Mario, and moves him across the map as he fights enemies including Goombas, Koopa Troopas and Piranha Plants. Along the way, Mario can hit regular bricks and mystery boxes, which consist of coins or powerups. Mario has accumulates points as we moves through the levels by collecting coins and powerups, and defeating enemies. Mario starts with three lives, and loses a life every time an enemy kills him. If he loses all three lives, the game is over and the user starts again.

For our version of Mario, we only used one level, and modified the map so that it could fit in memory. We also implemented most of the features and functionality of the original Mario, including Mystery Boxes, score accumulation, and enemy interaction. We also implemented other in game features such as scrolling and proper collision logic for more fluid game play.



Super Mario Bros Map for Our Project

Ultimately, we implemented a very simple version of Super Mario Bros for our project. Due to our time and hardware limitations, a full implementation of Super Mario was infeasible, however we attempted to replicate the core aspects of the popular game.

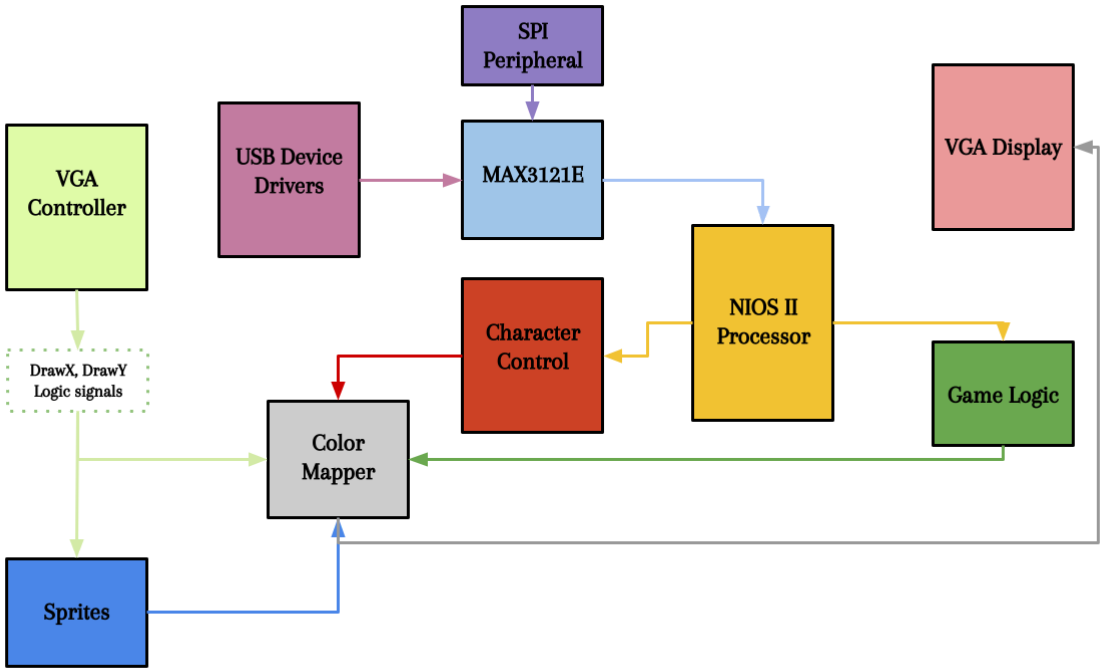
Written Description

Game Functionality

We designed our game to have similar functionality to the original NES game. Our Mario character can run and jump, in both the right and left direction. We also used various sprites to add animation to Mario's movement. Our game uses `W,A,D` keys for movement, where W is jump, A is moving left, and D is moving right. We also use `spacebar` to start the game at the beginning. Our in game functionality includes colliding with the Goomba, were hitting it from the sides results in death. We also added collision logic to objects such as pipes and blocks. We implemented coins and scores, such that whenever Mario collides with a mystery box he may or may not get a coin that increases his score. These coins are randomly assigned to the mystery boxes, and are different each time the game is run. The game ends when Mario reaches the castle at the end of the map.

Design Structure

Our internal design structure is built around the NIOS II processor, which serves as the CPU for our project. Using our lab 6 as a base, we then built upon the structure to fit our game. We kept the same SPI communication structure so that we could feed keyboard inputs into our project via the USB and also built upon the same VGA components so that we could display our game on the monitor.



Simplified Top Level Block Diagram

NIO II

Since the NIOS II processor is ideal for low speed tasks that require large number of states to implement in hardware, it is the ideal choice for a task such as USB enumeration for HID devices, as used in our final project. We can therefore use it with our MAX3421E chip because the speed of the keyboard is very low, but there are only a certain number of states to efficiently handle it in hardware. The USB protocol is handled on the NIOS II software, and the keycode that gets extracted is then sent from the USB keyboard input device to the hardware. The NIOS interacts with the USB host chip via the Serial Peripheral Interface (SPI) protocol, which allows the processor to initiate transactions on the USB chip such as reading and writing data to USB devices.

The Video Graphics Array (VGA) is used to display graphics to a monitor or external display. The NIOS interacts with the VGA by generating the signals and data needed to produce and display images and pixels to the screen. These signals include horizontal and vertical sync, as well as pixel data such as RGB values for the desired image. The NIOS communicates with VGA through the FPGA, which serves as a VGA interface for the processor. In addition to generating the necessary VGA signals, the NIOS also generates the graphic data that is displayed by writing to memory. The FPGA then reads these data bits from memory, and converts it to VGA signals so that the monitor can read and display them.

SPI Protocol

The SPI protocol essentially functions as a communication device that is used to transfer data between microcontrollers and other peripherals. SPI allows two way transfer of data, so data can be received by one device and also sent by the device. SPI operates in a master/slave architecture, where one processor controls all the communication with the other devices. In our project, the NIOS II serves as the master processor, and all the other peripheral devices act as slaves. SPI uses four wires to handle communications; Master Out Slave In (MOSI), Master In Slave Out (MISO), Serial Clock (SCLK), and Slave Select (SS). The MOSI wire is used to send data to the peripheral devices, and the MISO is used by the slaves to send data to the master device. The SCLK wire synchronizes the time during data transfer between the master and slave devices. Finally, the SS wire is used my the master object to select which slave peripheral device to send data to. Data transfer is always initiated by the master device by sending a clock signal on the SLCK line. The data is transferred 1 byte, or 8-bits at a time. MISO and MOSI and then used to send and receive their respective data.

Essentially, the SPI Protocol relates to our game because it is how the keyboard communicates with our design. The keyboard acts as a peripheral device, and communicates with the NIOS II processor through SPI. When a key is pressed, a signal is sent to the NIOS II indicating which key has been pressed, which is then sent to the

computer or display so that the corresponding result is shown on screen. In our project, this is shown by Mario moving in the desired direction based on which key is pressed.

VGA Components

The VGA component of our project is comprised of the color mapper, sprites, objects, and VGA controller. The overall structure is similar to that of lab 6, except that we create all of our objects separately, before finally connecting them to their corresponding sprite so that they can be displayed on the screen.

Objects

Before actually displaying the object, we must design it in a separate module so that we can give it functionality. The main objects in our game are Mario, the Goomba, and the various interactive blocks which consist of the pipes, bricks, mystery boxes, and the flag post block. Each object is given its own functionality, and assigned x and y locations so that the color mapper knows where to draw the object on screen.

Sprites

Each of the objects described above have corresponding sprites. The objects themselves are instantiated in our top level, and their output signals are wired to the color mapper. In the color mapper, we use these signals to call the sprites so that the correct pixels are drawn to the screen when we are interacting with the object. Each of our sprites are 32x32 pixel squares, allowing us to seamlessly integrate them into our design. However not every object is a perfect square, so when we draw the sprite, we use conditional statements so that the color mapper only outputs the sprite we want and not the background. Using a color like hot pink makes this process easier because none of our actual sprites use that color, and so we can make it a uniform background across all of our sprites for the game.

VGA Controller

The VGA controller module is used to produce the necessary timing signals needed by the VGA monitor to draw pixels. It also produces a pixel clock, which operates at half the frequency of our actual clock. As the beam moves through the screen, VGA controller uses horizontal and vertical counters to keep track of the exact position of the beams. So when the correct pixel is found by the beam, the controller checks to make sure it is within the bounds and then assigns it to the `DrawX` and `DrawY` signal so that the pixel can be drawn at the correct position on the screen.

Color Mapper

The Color Mapper module pulls the signals from all of our objects, and instantiates the corresponding sprites so that the objects can be displayed onto the monitor using the VGA controller module. In the color mapper, we establish a hierarchy which determines which object will be drawn first, and creates subsequent levels so it looks like our objects are being drawn onto the level map. Since all of our objects, except for the pipe, are 32x32 pixel squares (the pipe dimensions are 64x64 pixels), we generate them by using the equation of a square and ensuring that their are being drawn within the bounds of their size. Each object also has a corresponding 1-bit signal which tells the color mapper whether or not to draw the object at all, depending on what is currently happening in the game. The defualt layer of our color mapper is the background, so if any pixel on the screen is not a part of an object, it will draw the background at that pixel.

C Functions for USB

```
1  void MAXreg_wr(BYTE reg, BYTE val);
2  BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data);
3  BYTE MAXreg_rd(BYTE reg);
4  BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data);
5
6  int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,
7                             alt_u32 write_length,
8                             const alt_u8* wdata,
9                             alt_u32 read_length,
10                            alt_u8* read_data,
11                            alt_u32 flags);
```

The code block above shows the C functions that drive SPI protocol in our project. This is the same as in lab 6, but we also included it here for reference. `MAXreg_wr` and `MAXbytes_wr` use the SPI protocol to write data to the registers. The `MAXreg_wr` writes the actual register to the MAX3421 USB chip. It does this by first adding 2 to the register BYTE, and then writes the value of the register to the chip. The `MAXbytes_wr` function is used to write bytes to the USB chip, and returns a pointer to the memory location after data is written to memory. All of our write and read functions use the `alt_avalon_spi_command();` function to perform these operations. The function is provided by Intel, and can either perform a read or a write at one time. The function takes in the base address, a read or write signal, and the data to be passed through. The function can also take a flag if needed. The function returns an int, and a return value of `-1` indicates an error in the SPI protocol. The `MAXreg_rd` and `MAXbytes_rd` functions work the same way, where the `MAXreg_rd` function returns register from the USB chip while the `MAXbytes_rd` function returns a pointer to the memory position after the last written data.

Game Logic

Game FSM

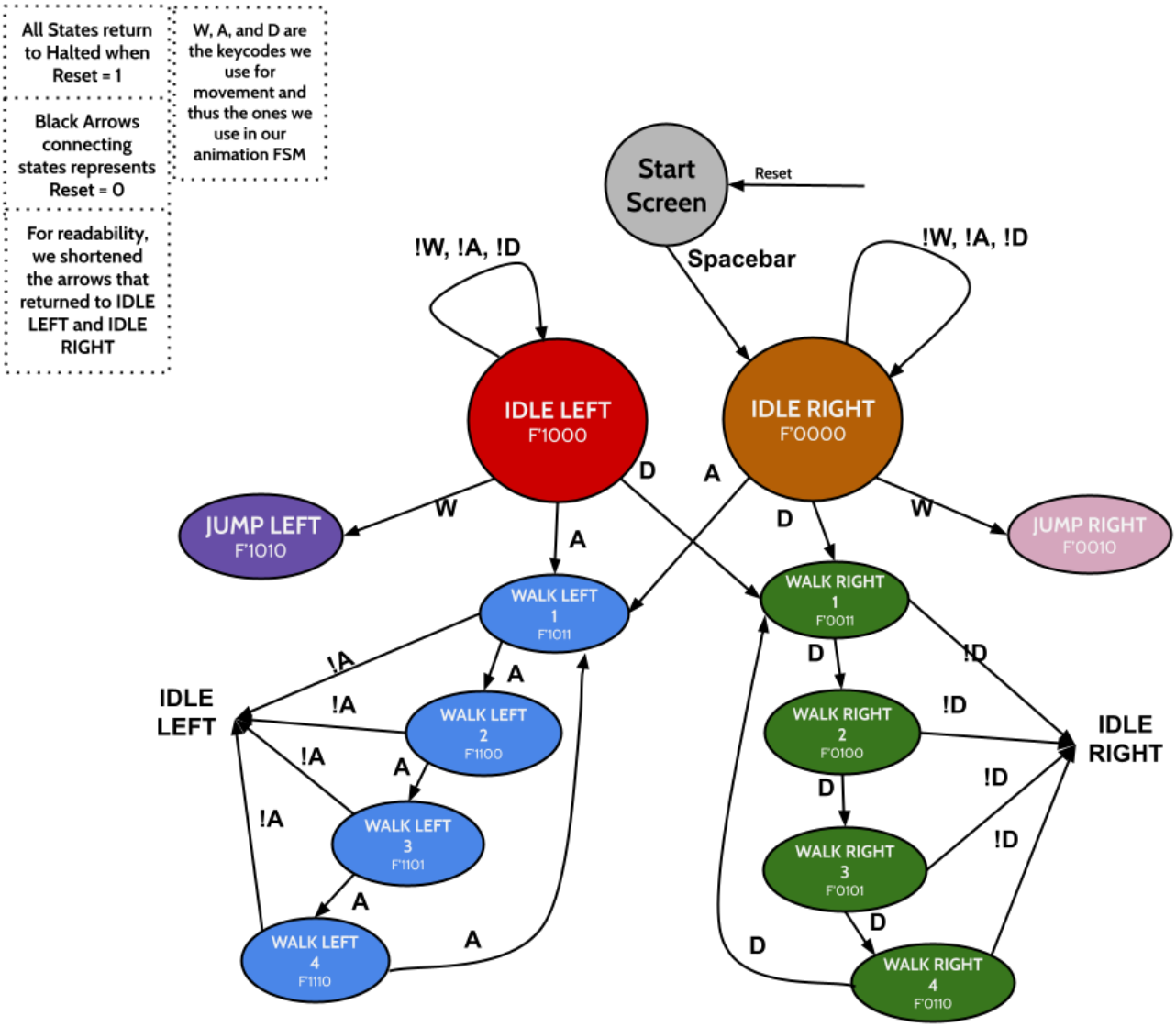
When we were creating our game we wanted to implement game menus like the real Super Mario Bros game and since we only had one level, we realized that we would need a starting menu before we start the game and an ending menu for when we finish the level or if Mario dies. In order to implement this we created a finite state machine such that the game is in the "Start" state when we press reset or first flash our FPGA to the screen. In the start state, we can see Mario but all of our movement keycodes are disabled such that Mario cannot move. When we press the spacebar key, however, we transition to the "Game" state so Mario can freely move left and right and also jump so we can then proceed to play the game. In the game state, we can use the keys to play through the level but there is a hidden sixty-second counter. If Mario dies three times to a Goomba, reaches the castle, or the counter reaches zero, then we transition to the "GameOver" state in which all keycodes are disabled again until we press reset at which point we return back to the "Start" state again.

Mario

The main character in our game had to have several components such that he looks and acts just like Mario does in the real Super Mario Bros game. We first had to make sure his sprite was correct before we moved on to anything else. Most of our level can be broken down into tiles and in our 640x480 pixel screen, each tile was 32x32 pixels. Since we can fit Mario into one of these tiles, we drew his sprite hitbox as exactly one tile and filled it in with a right-facing Mario sprite with a hot pink background. We then removed the pink background in our color mapper module so Mario is cleanly drawn into the game, allowing us to proceed to the animation stage.

In the real game, Mario can run and jump at the same time but based on our provided lab code we could only register and use one key at a time and every time we would try to do both Mario would stop running and jump. To fix this, we added a PIO block allowing for another keycode and wherever we used keycode in our code we added "keycode1" such that we can both move left/right and also jump.

Animation FSM

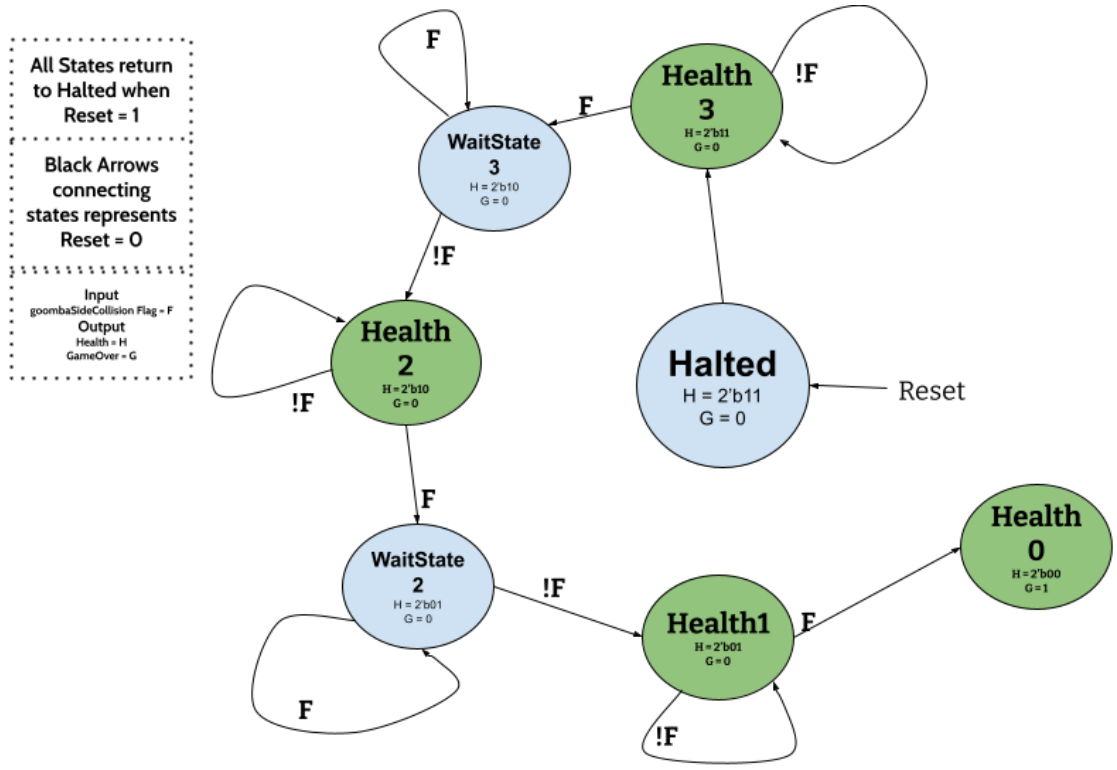


Simplified Mario Animation FSM Diagram

To create animations for Mario jumping and running left and right, we decided to implement a finite state machine. We transition between states by using keycodes and each state outputs a four-bit "frame" value that we wire to the colormapper module to tell the colormapper what frame to draw in Mario's hitbox. When we first press the spacebar key to first start our playthrough of the level, Mario starts in the "Idle Right" state which means he is standing still facing the right direction. If we want to run, we have to use his three running sprites: right hand forward, switching hands, and left hand forward. When we hold down the "D" key to run to the right, we use the three running sprites in the right direction and cycle through those states until we let go of that key, returning us back to the "Idle Right" state. If we hold down the "A" key, however, we run to the left and must cycle through the three running sprites in the left direction until we let go of that key, instead returning us to the "Idle Left" state. Finally, our jump sprite will appear jumping right or left when we press the "W" key depending

on if our previous state was “Idle Right” or “Idle Left” and return back to that idle state once Mario is done jumping. Once we implemented our fsm for Mario’s movement animation, we realized that transitioning between states was incredibly fast as they would occur every clock cycle and we would have to reduce the speed of the next state transitions. As a solution, we created a temporary variable that gets incremented every clock cycle and when the temp variable modulus 15 is 0 then we transition to the next state. This meant that we would transition between frames every 15 positive edges of the clock.

Jump FSM



Simplified Mario Jump FSM Diagram

After we completed our animations, we had to actually implement gravitational jumping such that Mario would jump on his own off of one tap of the “W” key. We move Mario in the y-direction by adjusting his y-direction position by doing $YPos = YPos + YMotion$ so we created a jumping FSM module that has a

“Velocity” output that we wire to our ball module to represent YMotion. For every state a velocity value is outputted from the jump fsm and after all of the jump states are over Mario moves up by 132 pixels and then after the fall states he moves back down 132 pixels. With this logic, we were not able to account for falling off of blocks in the air or jumping off of a block that is in the air. To fix this, we added many extra fall states to ensure that Mario reaches the floor no matter where he jumps off.

Goomba

Additionally, goombas are a big part of our game as they determine how Mario can lose his lives and we need to implement their movement along with Mario’s collision with them. When we first start our level, the goomba begins moving to the right and if it collides with a pipe its direction reverses and it then walks in the other direction. We were able to use the ball.sv logic which we had from a previous lab for this implementation. For the collision, if Mario runs into the goomba from the left or right side, he displays a death sprite and his life counter decrements, sending him back to the starting position if he still has remaining lives. If Mario jumps onto the goombas, however, then the goomba would die and Mario’s score would increment.

Blocks

In our game, we included various objects and non-playable characters for Mario to interact with. For example, we have bricks, mystery boxes, pipes, coins, and goombas. Just as Mario has a 32x32 hitbox, so too do the obstacles. We were able to represent all of the obstacles with either a 32x32 pixel hitbox or a 64x64 hitbox. We could check for collision when the border of Mario’s hitbox overlaps in any way with the obstacle hitboxes. Since we ran into some bugs, we were able to test our collision flags by wiring them to our Hexdrivers to display on the Hex LEDs and fix our collision module’s code where necessary. When Mario collides with each obstacle, different things would occur in the game or to Mario.

Firstly, the bricks that are scattered around the map and the two pipes should block Mario’s movement when he collides with them from any direction. We needed to create logic such that Mario’s movement in the x-direction is 0 whenever the collision flags between Mario and the bricks or pipes from the left and right are high. Furthermore, we needed to interrupt our jump state machine when Mario encounters an obstacle from the top or bottom. If the top of Mario overlaps with a brick then we want to set that flag high and move from whichever jump state we are in the FSM directly to the ARCTOP state. This will stop upward movement immediately and immediately move on to falling downwards.

Scrolling

```
  always_comb begin
    if(marioBoundsFlag)
      Ball_X_Max = 10'd639;
    else
      Ball_X_Max = 10'd319;
  end

  always_ff @ (posedge frame_clk or posedge Reset) begin
    if (Reset)
      ScrollX<= 0;
    else begin
      //SCREEN SCROLLING LOGIC
      if((Ball_X_Pos + Ball_Size) >= Ball_X_Max && keycode == 8'h07 &&
PipeCollisionRight == 1'b0 && Brick1CollisionRight == 1'b0 &&
Pipe2CollisionRight == 1'b0 && StairCollisionRight == 1'b0)
        ScrollX <= ScrollX + 3;
      else
        ScrollX <= ScrollX;
    end
  end
```

One of the most challenging parts of our final project was implementing scrolling. Since the monitor only displays 640x480 pixels, and the level map of Mario is much larger in the x direction, it is obviously impossible to fit the entire map into one screen. So in order to implement the map seamlessly, we had the option of choosing between room logic or scrolling. Scrolling was much more challenging to implement, but more realistic in terms of the original Mario game, so we went with that option in our pursuit of building the original NES game. We accomplished this challenge by creating a `ScrollX` variable in our Mario module. This was a 12-bit signal so that it could represent unsigned decimal numbers up to 4096.

The code block above shows how we instantiated the scrolling signal, and connected it to Mario's position on the screen. The `marioBoundsFlag` is raised when Mario reaches the end of the map. While Mario is moving through the map, his maximum x position is set to `10'd319`, which is halfway between the monitor 640px wide monitor. So when Mario reaches this maximum and tries to move forward on the keypress, his x position does not change but the map "scrolls" in the forward direction by increments of his step size, giving the appearance that Mario is still moving right.

However, the map eventually comes to an end, and we wanted to our scrolling logic so that Mario can move freely past his bounds and reach the castle, which is placed beyond the 319 pixel bound. This logic is controlled directly in our `spriteram.sv` where we draw the map, and raises the `marioBoundsFlag` when the end condition is met. Once the flag is raised, Mario's maximum x position updates, allowing him to move to the end of the screen, where the castle stands.

In addition to drawing the background and driving Mario's motion, the `ScrollX` signal was also vital for our object placement. It allowed us to place objects beyond the bounds of the 640 pixel screen, so that it would only draw the object when Mario encounters it further down the map. We were able to do this by using the equation $objectXpos = objectXstart - ScrollX$ where "object" can be anything from bricks to the pipes and the start position can range up to 1028, which is the width of our map. Essentially this made it so that every time Mario attempted to move past his bounds, `ScrollX` would increment, and eventually $objectXpos$ would reach a value that is in the range of 0 and 640 such that it can be visible to the user on the monitor. This also allows us to use overflow logic, such that when the value of `ScrollX` becomes larger then the start coordinate, the position becomes negative and falls of the screen such that it is not drawn again once Mario moves past it.

Collisions

The score in our game can be incremented in two main ways: killing a goomba or collecting a coin. We wired the `goombaTopCollision` and `coinCollision` flags from the collision module to a score module that increments a “Score” variable every clock cycle with the collision flags. For most of the game, the score will remain the same since the collision flags are low but when they become high, then we want to increment the score. We then wired our “Score” variable to the hex drivers which we were able to use to display our total score during and after the game on the Hex LEDs.

Mystery Boxes and Coin Generation

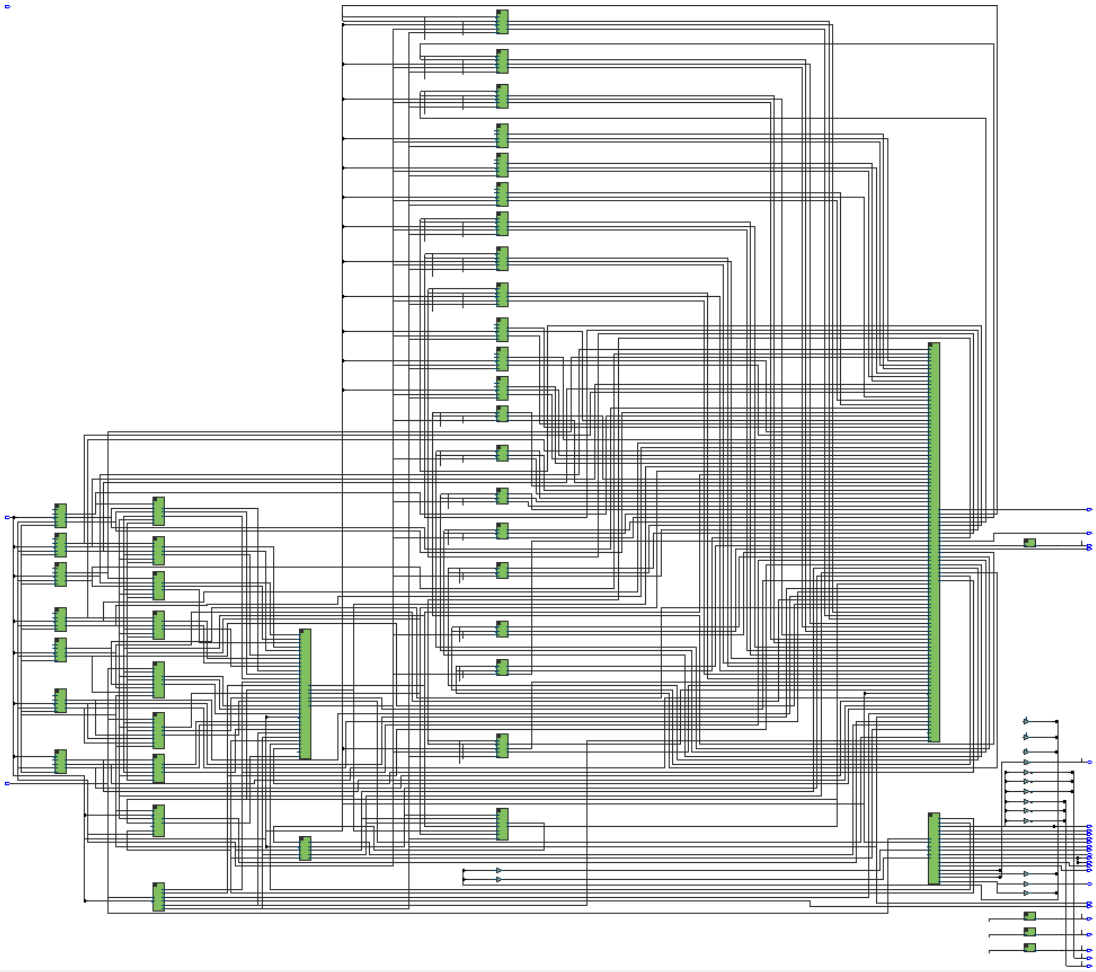
Finally, we have the mystery boxes and coins. We wanted to add a twist to our game and change our mystery boxes to trick boxes. When we are at the start of the game, we have a counter, and right when we press the spacebar key to begin the game, a temporary variable is loaded with the value of the counter modulus 10. Then, based on that temporary variable’s value, we choose which mystery boxes in the game will hold coins and which ones will not. Each mystery box has a coin at the nearest ground level below it but it cannot be collided with until the “mBoxOn” flag is low. Mario’s collision from any direction with the mystery box raises a flag and if that flag and the mBoxOn flags are both high, then this means we have collided with a box and want it to disappear. We also want to then determine based on the temporary value if we turn its corresponding coin

on by making the coinOn flag high. If the coin is on, then Mario can go ahead and collect it. If the coin is not on then we will not collect it if we walk over its position since we can only collect it if the collision flag between Mario and the coin is high along with the coinOn flag. Once we collect the coin, then we move on to incrementing the score.

Health

Last, but not least, we have lives that we implemented in our game, which we did through an FSM. At the start of the game, Mario has 3 lives which we represent through "health = 2'b11" and we go to the Health3 state. To change between health states, we wired in the goombaSidecollision which signifies that Mario should lose a life since he collided with a goomba. Every time Mario collides with a goomba we want to transition to the next state where he has less health and when his health is 0, then he has 0 lives and we raise the gameOver flag. The gameOver flag is then wired to our game fsm which ends the game. We ran into the error where Mario would lose all three lives upon one collision with the goomba and this was because if Mario collided with the goomba for three clock cycles, he would lose a life in each clock cycle. To fix this, we created wait states after each "Health#" state such that if Mario collides with the goomba he loses a life but cannot lose another one until he stops colliding (leaving the wait state) and collides with the goomba again. With all of this logic and the rest mentioned above, we were able to successfully implement a Super Mario Bros level that we made.

Top Level Block Diagram



RTL Top Level Block Diagram

System Verilog Module Descriptions

Hex Driver

HexDriver.sv

```
1 module HexDriver (input  [3:0]  In0,
2                   output logic [6:0]  Out0);
```

This is the HexDriver module that takes a 3-bit input and extends it to 7-bits so that it can output the corresponding hex digit. Its purpose is to assign the hex outputs that are displayed to the user on the FPGA.

Color Mapper

Color_Mapper.sv

```
module  color_mapper ( input         Reset, frame_clk,
                       input        [11:0] BallX, BallY, Ball_size,
                       input        [9:0] DrawX, DrawY,
                       input        MarioFlag,
                       input     [11:0] GoombaX, GoombaY, Goomba_size,

                       input     [11:0] SB1X, SB1Y, SB1_size,

                       input     [11:0] QB1X, QB1Y, QB1_size,
                       input     [11:0] QB4X, QB4Y, QB4_size,
                       input     [11:0] QB5X, QB5Y, QB5_size,
                       input     [11:0] QB6X, QB6Y, QB6_size,
                       input     [11:0] QB7X, QB7Y, QB7_size,
                       input     [11:0] QB8X, QB8Y, QB8_size,
                       input     [11:0] QB9X, QB9Y, QB9_size,

                       input     [11:0] BB1X, BB1Y, BB1_size,
                       input     [11:0] BB2X, BB2Y, BB2_size,
                       input     [11:0] BB3X, BB3Y, BB3_size,
                       input     [11:0] BB4X, BB4Y, BB4_size,
                       input     [11:0] BB5X, BB5Y, BB5_size,
                       input     [11:0] BB6X, BB6Y, BB6_size,
                       input     [11:0] BB7X, BB7Y, BB7_size,
                       input     [11:0] BB8X, BB8Y, BB8_size,
                       input     [11:0] BB9X, BB9Y, BB9_size,
                       input     [11:0] BB10X, BB10Y, BB10_size,

                       input     [11:0] CB1X, CB1Y, CB1_size,
                       input     [11:0] CB2X, CB2Y, CB2_size,
                       input     [11:0] CB3X, CB3Y, CB3_size,
                       input     [11:0] CB4X, CB4Y, CB4_size,
                       input     [11:0] CB5X, CB5Y, CB5_size,
                       input     [11:0] CB6X, CB6Y, CB6_size,
                       input     [11:0] CB7X, CB7Y, CB7_size,

                       input     [11:0] PBX, PBY, PB_size,
                       input     [11:0] PB2X, PB2Y, PB2_size,

                       input     [3:0] marioLevel, MARIO_FSM,
                       input       [11:0] ScrollX,
                       input       marioDirection, //mario direction flag,
```

```
42                  input logic vga_clk, blank, goombaAnimationFlag,
    gameovertemp,
43                  output logic marioBoundsFlag,
44
45                  output logic COINCOLLISION,
46                  output logic COINCOLLISION2,
47                  output logic COINCOLLISION3,
48                  output logic COINCOLLISION4,
49                  output logic COINCOLLISION5,
50                  output logic COINCOLLISION6,
51                  output logic COINCOLLISION7,
52
53                  output logic SCORECOLLISION,
54                  output logic SCORECOLLISION2,
55                  output logic SCORECOLLISION3,
56                  output logic SCORECOLLISION4,
57                  output logic SCORECOLLISION5,
58                  output logic SCORECOLLISION6,
59                  output logic SCORECOLLISION7,
60
61                  output logic [3:0] MARIOSCORE,
62
63                  output logic marioBrickFlag,
64                         output logic [7:0]  Red, Green, Blue);
```

The color mapper module takes multiple 12-bit inputs which represent the X and Y coordinates along with the sizes of all the sprites we draw on the monitor. It also takes the 10-bit input `DrawX` and `DrawY` which are used to draw the pixels to the screen. The 4-bit inputs `marioLevel` and `Mario_FSM` are used to drive our Mario sprite animations so that it looks like Mario is running and jumping on screen. The 1-bit input `MarioFlag` tells the color mapper whether or not to draw Mario. The signal is high when Mario is alive and has not collided with an enemy, and low when he collides with a Goomba, signaling to the color mapper to not draw Mario. The 12-bit `ScrollX` signal is the scrolling signal that tells color mapper where to draw the sprites based on Mario's location throughout the level. The `ScrollX` signal increments as Mario moves left to right. The 1-bit `marioDirection` signal tells the color mapper which set of sprites to use for drawing Mario. One set shows all the sprites facing right and a flipped copy that shows Mario facing left. The `vga_clk` and `blank` signals provide the clock and draw signal for overall color mapper functionality. The `goombaAnimation` signal is similar to the `Mario_FSM` except controls the animations for the Goomba. Since the Goomba animation only uses 2 sprites, we were able to use a 1-bit signal. The `COINCOLLISION` and `SCORECOLLISION` signals are all 1-bit outputs to the our other modules since we instantiated all the coin and collision signals for each mystery box and coin collection in

color mapper module itself. The 4-bit `MARIOSCORE` is the actual signal that holds Mario's score and is wired to our top level so that it can be outputted to the hex displays. The 8-bit `Red/Green/Blue` signals are the color signals outputted to the other modules so that the pixels can be drawn to the screen.

The purpose of the color mapper module was to draw all of our sprites to the display. We collected all the signals in this and instantiated all the sprites so that we could assign the pixel colors that draw our game to the screen in the correct order and orientation.

VGA Controller

VGA_controller.sv

```
module  vga_controller ( input         Clk,       // 50 MHz clock
                                       Reset,     // reset signal
                         output logic hs,        // Horizontal sync pulse.  Active low
                                      vs,        // Vertical sync pulse.  Active low
                                      pixel_clk, // 25 MHz pixel clock output
                                      blank,     // Blanking interval indicator.  Active low.
                                      sync,      // Composite Sync signal.  Active low.  We don't use it in this lab,
                                                 //   but the video DAC on the DE2 board requires an input for it.
                                      output [9:0] DrawX,     // horizontal coordinate
                                                   DrawY );   // vertical coordinate
```

This is the VGA controller and connects all the VGA signals together so that the pixels can be outputted to the monitor or display. Based on the vertical and horizontal count the `DrawX` and `DrawY` signals are assigned so that the correct pixel can be drawn at the correct spot.

The purpose is to use the pixel beams to find the correct coordinates to draw the pixel.

Background Helper

Color_Mapper.sv

```
module drawBackground(input logic [3:0] red, green, blue,
                      output logic [3:0] red_out, green_out, blue_out);
```

This was essentially a helper module we used for drawing our sprites. The 4-bit inputs were the pixel colors that will be drawn to the monitor, and the 4-bit outputs are the actual colors of the pixel to be drawn to the screen.

Essentially, all of our sprites had a pink background in the empty spaces so that they would all be the size of a 32x32 pixel box. However we obviously don't want the pink drawn, so if the current pixel being drawn was pink the output would instead be the background pixel. If not pink, then the output would just be the input. We used a separate module for this because it just made our code a little more organized and easier to debug.

Top Level

lab62.sv

```
module lab62 (

    ///////// Clocks /////////
    input     MAX10_CLK1_50,

    ///////// KEY /////////
    input    [ 1: 0]   KEY,

    ///////// SW /////////
    input    [ 9: 0]   SW,

    ///////// LEDR /////////
    output   [ 9: 0]   LEDR,

    ///////// HEX /////////
    output   [ 7: 0]   HEX0,
    output   [ 7: 0]   HEX1,
    output   [ 7: 0]   HEX2,
    output   [ 7: 0]   HEX3,
    output   [ 7: 0]   HEX4,
    output   [ 7: 0]   HEX5,

    ///////// SDRAM /////////
    output             DRAM_CLK,
    output             DRAM_CKE,
    output   [12: 0]   DRAM_ADDR,
    output   [ 1: 0]   DRAM_BA,
    inout    [15: 0]   DRAM_DQ,
    output             DRAM_LDQM,
    output             DRAM_UDQM,
    output             DRAM_CS_N,
```

```
32      output              DRAM_WE_N,
33      output              DRAM_CAS_N,
34      output              DRAM_RAS_N,
35
36      ///////// VGA /////////
37      output              VGA_HS,
38      output              VGA_VS,
39      output    [ 3: 0]   VGA_R,
40      output    [ 3: 0]   VGA_G,
41      output    [ 3: 0]   VGA_B,
42
43
44      ///////// ARDUINO /////////
45      inout     [15: 0]   ARDUINO_IO,
46      inout               ARDUINO_RESET_N
47
48  );
```

We copied our top level module directly from lab62 and used to call all of our other VGA functions. All the output signals are labeled accordingly, and serve as inputs for our .soc module to instantiate all the blocks we created on Platform Designer. The VGA signals are used by VGA controller, and the outputs are assigned to the corresponding pixel color the is outputted by the `Color_Mapper` module. The overall purpose is to bring all the other hardware components together so that the VGA can output a display to the screen. We also used this module to instantiate all of our outer modules and objects such as Mario, Goomba, Question Blocks, Mystery Boxes, and some collision modules such as Mario and Goomba collision and collisions with the pipes and blocks.

The purpose of this module is to instantiate all of our objects and collision logic in one place so that we can wire all the signals from a central place. This makes debugging and following our wire path easier.

Mario

ball.sv

```
1  module  ball ( input Reset, frame_clk, VGA_Clk,
2            input [7:0] keycode, keycode1,
3            input       MarioFlag,
4            input [9:0] DrawX, DrawY,
5            input marioBrickFlag,
6            input marioBoundsFlag,
7
8            input PipeCollisionLeft,
9            input PipeCollisionRight,
```

```
10          input PipeCollisionTop,
11
12          input Pipe2CollisionLeft,
13          input Pipe2CollisionRight,
14          input Pipe2CollisionTop,
15
16          input Brick1CollisionLeft,
17          input Brick1CollisionTop,
18          input Brick1CollisionRight,
19
20          input Brick2CollisionLeft,
21          input Brick2CollisionTop,
22          input Brick2CollisionRight,
23
24          input Brick3CollisionLeft,
25          input Brick3CollisionTop,
26          input Brick3CollisionRight,
27
28          input Brick4CollisionLeft,
29          input Brick4CollisionTop,
30          input Brick4CollisionRight,
31
32          input StairCollisionLeft,
33          input StairCollisionTop,
34          input StairCollisionRight,
35
36
37          output MarioDirection,
38          output [3:0] FSM_OUT,
39               output [11:0]  BallX, BallY, BallS,
40           output [11:0] ScrollXOut);
```

The Mario module is where we instantiate all the signals related to the user controlled character for our game. The 1-bit `Reset` and `frame_clk` signals are used throughout the module to reset the signals and provide a clock for our `always_ff` blocks. The 8-bit `keycode` and `keycode1` inputs are passed in from the keyboard and used to control Mario based on the key press. We passed in two different keycode signals so that the user can pass in parallel keystrokes to the game. This is useful for example if the user wants Mario to move right as he is jumping in the air. The 1-bit `MarioFlag` is used to signal to the module whether or not Mario is dead or alive. The 1-bit signals `marioBrickFlag` and `marioBoundsFlag` are used to signal if Mario has collided with a brick and if he is within the scrolling bounds. All the 1-bit inputs for the multiple `PipeCollision`, `BrickCollision`, and `StairCollision` signals are used to send collision signals for the different objects on the map. Each object has its own collision

signals, so if Mario collides with that object from a specific direction, the given flag with send a signal to the Mario module to restrict movement in that direction. For example, if Mario collides with Pipe 1 from the left, then `PipeCollisionLeft` will turn high, and Mario will not be able move in that direction. This leads to the 1-bit `MarioDirection` is used just for left and right movement, and is wired to the top level to be used by other modules. When the signal is high, Mario moves left and when the signal is low Mario moves right. The 4-bit `FSM_OUT` signal is a concatenated signal that tells the color mapper with sprite to show, based on Mario's movement. If the right or left key is pressed, then Mario will cycle through three sprites which represent a complete running animation in his respective direction. If Mario is jumping, then the signal will show the same animation depending on which direction he is moving while in the air. The 12-bit signals `BallX`, `BallY`, and `BallS` hold Mario's x and y coordinates, as well as his size, which is a constant 32 pixels. The size of the signals was originally 10 bits, but we had to extend it to account for our larger map. Finally, this module outputs a 12-bit `ScrollXOut` signal, which drives the scrolling. This signal is based on Mario's movement, and allows us to implement flawless scrolling based on Mario's position on the map at any given map. Since the map is 1280 pixels wide, we had to make the signal size big enough such that we could represent Mario's position anywhere on the map in terms of our `ScrollX` signal.

The purpose of this module is to create Mario. This is where we created the FSM for his animations, used the keycodes to drive his movement, and designed the jump FSM which allows him to jump on blocks and over the pipes placed throughout our map. We also create the `ScrollX` signal in this module which is then wired to our top level and used by all of our other modules to place objects throughout the map. We declared that signal in our Mario module because the signal increments when Mario moves, so it would be easiest to implement here since the signal directly depended on Mario and his movement along the x-axis.

Goomba

goomba.sv

```
module  goomba ( input Reset, frame_clk,
          input [11:0] ScrollX,
          input  GoombaFlag,
               output [11:0]  goombaX, goombaY, goombaS,
          output goombaAnimationFlag,
          output overflowFlag);
```

This module is similar to the Mario module, except it instantiates the Goomba object. The 12-bit `ScrollX` signal is used to position the Goomba in the map and prevent overflow as Mario moves. The 1-bit `GoombaFlag` signal represents whether it is alive or not. The 12-bit `goombaX`, `goombaY`, and `goombaS` represent the Goomba's x and y position and the size respectively. The `goombaAnimationFlag` tells the color mapper which Goomba sprite to draw so that it looks like the Goomba is walking. Finally, the `overflowFlag` gets switched to high when the Goomba has moved out of the screen, and tells the module when to stop drawing the Goomba to the screen once Mario has moved past it.

The purpose of the module is to put all of the Goomba's animations and movement in one place. This module is called in the top level so that the signals can be wired to the other modules that need the Goomba signals for fluid gameplay.

Sprites

spriteram.sv

```
module mariomap(
  input logic [9:0] DrawX, DrawY,
  input logic [11:0] ScrollX,
  input logic vga_clk, blank,
  output logic marioBoundsFlag,
  output logic [3:0] bg1_red, bg1_green, bg1_blue
);

module goomba_sprite (
  input logic [9:0] DrawX, DrawY,
  input logic [11:0] GoombaX, GoombaY,
  input logic vga_clk, blank,
  output logic [3:0] goomba_red, goomba_green, goomba_blue
);

module idle_mario_1 (
  input logic [9:0] DrawX, DrawY,
  input logic [11:0] MarioX, MarioY,
  input logic vga_clk, blank,
  output logic [3:0] idle_mario_1_red, idle_mario_1_green,
idle_mario_1_blue
);

module brickBlockSprite (
  input logic [9:0] DrawX, DrawY,
  input logic [11:0] BBX, BBY,
  input logic vga_clk, blank,
```

```
  output logic [3:0] bb_red, bb_green, bb_blue
);

module pipeBlockSprite(
  input logic [9:0] DrawX, DrawY,
  input logic [11:0] PBX, PBY,
  input logic vga_clk, blank,
  output logic [3:0] pb_red, pb_green, pb_blue
);

module coin (
  input logic [9:0] DrawX, DrawY,
  input logic [11:0] COIN_X, COIN_Y,
  input logic vga_clk, blank,
  output logic [3:0] coin_red, coin_green, coin_blue
);
```

This module declares all the sprites we used in our game. All of the modules are pretty much the same, and have a 10-bit `DrawX` and `DrawY` input, along with the `vga_clk` and `blank` signals. They also have some kind of 12-bit x and y position signal, which tells each individual module where to place the sprite on the screen. Finally, each sprite module outputs three 4-bit signals which give the red, green, and blue color value of the pixel. These modules also call the palette and the rom files for each individual module so that it can get the right color to draw to the screen. The only unique sprite module is `mariomap`, which also has `ScrollX` and `marioBoundsFlag` as input and output respectively. The `ScrollX` signal is passed so that the map gets continuously drawn as Mario moves forward. This makes it seem like the map is scrolling as the map is being drawn. The `marioBoundsFlag` is used to signal when Mario has reached the end of the map so that scrolling turns off. This signal is necessary because Mario eventually arrives in the end of the map, and we don't want to draw more since Mario is within the range of the end of the map. So when the flag is raised the background becomes static so that Mario can reach the castle and finish the game.

The purpose of this module is to call all the sprites in one file. Each module instantiates a different sprite for our game. We put all of them in one file so that our code was more organized and easy to debug if we run into issues.

Collisions

collision.sv

```
module marioGoombaCollision(input frame_clk,
                    input [11:0] MarioX, MarioY, MarioS,
                    input [11:0] GoombaX, GoombaY, GoombaS,
```

```
                    output MarioFlag, GoombaFlag);

module marioPipeCollision(input [11:0] MarioX, MarioY, MarioS,
                    input [11:0] PipeX, PipeY, PipeS,
                    output PIPE_FLAGLEFT,
                    output PIPE_FLAGRIGHT,
                    output PIPE_FLAGTOP);

module Mario5BricksCollision(input [11:0] MarioX, MarioY, MarioS,
                   input [11:0] BlockX, BlockY, BlockS,
                   output BRICK_FLAGLEFT,
                   output BRICK_FLAGRIGHT,
                   output BRICK_FLAGTOP);
```

The collisions module is where we instantiate all of our collisions between Mario and the objects on the map. We have a different module for collisions with Goomba, the pipes, and the mystery and brick blocks. Each one takes 12-bit signals for the coordinates and size of Mario and the object he is colliding with. The `marioGoombaCollision` module outputs two separate 1-bit signals, each one representing the alive status of Mario and and Goomba based on the collision type. However all the modules that involve collisions with objects output 3 1-bit signals, each one representing a collision flag from the three possible collision directions. This can be collision from the right, left, or the top.

The purpose of these modules is to control the collision signals, and is called in the top level so that we can easily wire the output signals to the necessary modules so that collisions are represented on screen when Mario runs into Goombas or objects.

Blocks

blocks.sv

```
module questionBlock(input [11:0] ScrollX,
                input [11:0] QB_X_Start, QB_Y_Start,
                output [11:0] QB_X, QB_Y, QB_S);

module brickBlock(input [11:0] ScrollX,
                input [11:0] BB_X_Start, BB_Y_Start,
                output [11:0] BB_X, BB_Y, BB_S);

module stairBlock(input [11:0] ScrollX,
                input [11:0] SB_X_Start, SB_Y_Start,
                output [11:0] SB_X, SB_Y, SB_S);

module pipeBlock(input [11:0] ScrollX,
             input [11:0] PB_X_Start, PB_Y_Start,
```

```
15              output [11:0] PB_X, PB_Y, PB_S);
16
17  module coinBlock(input [11:0] ScrollX,
18                   input [11:0] CB_X_Start, CB_Y_Start,
19                   output [11:0] CB_X, CB_Y, CB_S);
```

The blocks module is used to instantiate the actual objects that are then represented by the sprites on the display. Each module in `blocks.sv` has three 12-bit inputs, `ScrollX` and the x and y starting positions. The modules also output three 12-bit signals, representing the start coordinates accounting for scrolling as well as the size. The size is assigned as a constant 32 pixels, and is consistent throughout all of our objects for simplicity. The y coordinate also stays the same as the value passed in, but the outputted x coordinate is different then the signal passed in because it accounts for the scroll value generated by Mario.

The purpose of this module is to declare the start position of all the blocks before we connect them to sprites so that we can organize all the declarations in one place. This makes our program more organized and easier to debug when we are repositioning the blocks.

Coin Animations

coins.sv

```
1  module coinAnimation(input Reset, frame_clk, output [1:0] COINSIGNAL);
```

The coins module is used to create the animations for the mystery box which generates the coins. The 1-bit inputs are used to generate the counter in an `always_ff` block, and the 2-bit `COINSIGNAL` output is the current count, depending on which the color mapper draws the corresponding map. The sprite changes on a constant cycle, giving the appearance of a block animation.

The purpose of this module is to create the animations for the mystery boxes. We call this module in the top level so that we can connect the signals to our color mapper and output the sprites to the screen.

Game

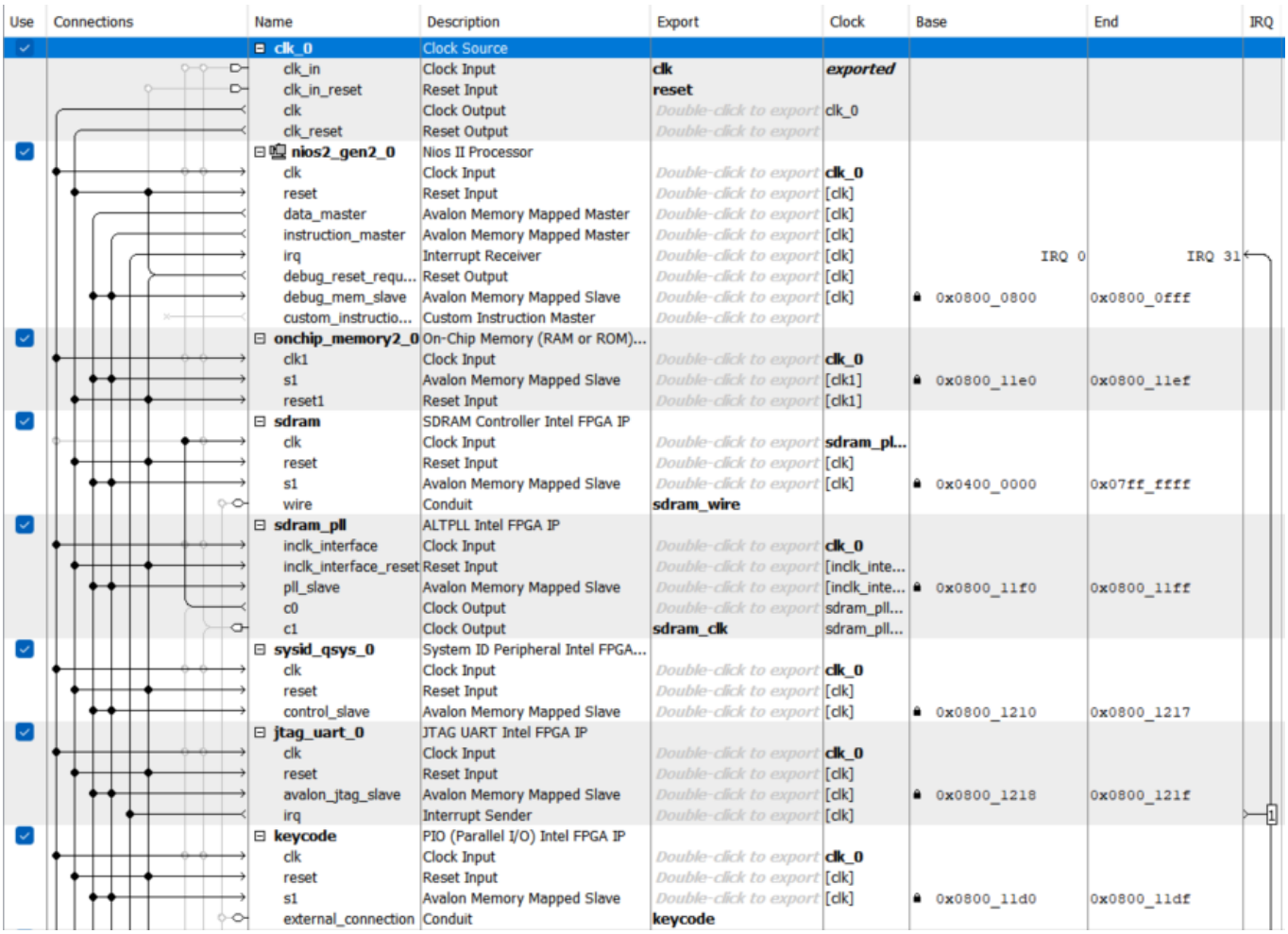
fsm.sv

```
module game(input logic Clk,
                        input logic Reset,
                        input logic collisionFlag,
                input [7:0]keycode,
                input logic [11:0]ballX,
                input logic [11:0]scrollX,
                        output logic gameOver, playerFlag);
```

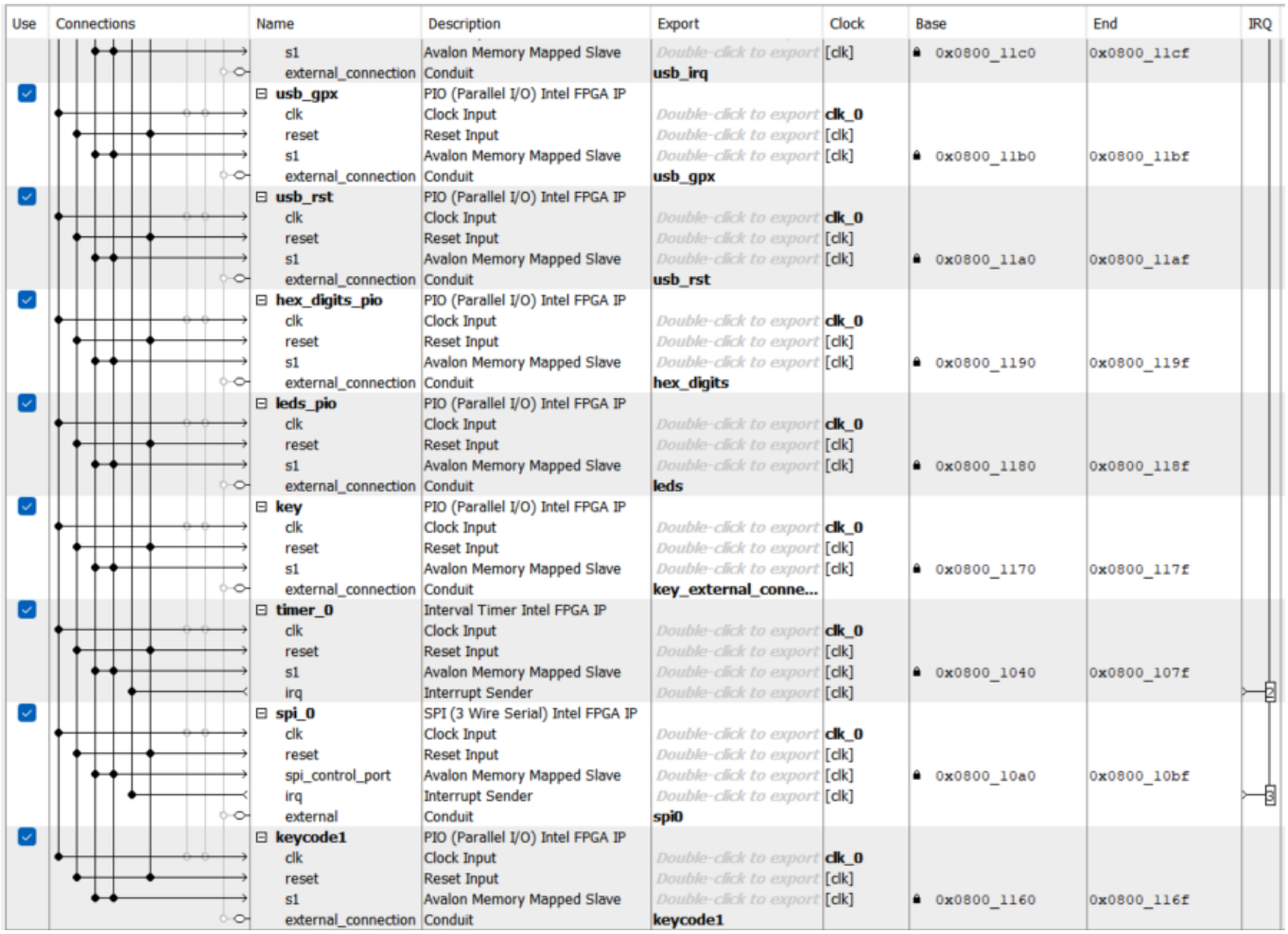
The game module takes in three 1-bit inputs which drive the overall game fsm. The 8-bit `keycode` signal is used to start the overall game when the user presses spacebar. The 12-bit signals `ballX` and `scrollX` represent Mario's position on the map based on the value of scroll. The `gameOver` flag is raised when collision is detected, and the `playerFlag` represents whether or not Mario is alive based on the current state of the game.

The purpose of this module is to control our overall game so that it starts when spacebar is pressed, and ends when Mario either collides with the Goomba, or reaches the end of the map. We also use the output signals to control the black game over screen so that the user knows when the game ends.

System Level Descriptions and Diagram



Platform Designer Blocks 1/2



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_11c0	0x0800_11cf	
		external_connection	Conduit	**usb_irq**				
☑		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	**clk_0**			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_11b0	0x0800_11bf	
		external_connection	Conduit	**usb_gpx**				
☑		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	**clk_0**			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_11a0	0x0800_11af	
		external_connection	Conduit	**usb_rst**				
☑		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	**clk_0**			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_1190	0x0800_119f	
		external_connection	Conduit	**hex_digits**				
☑		**leds_pio**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	**clk_0**			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_1180	0x0800_118f	
		external_connection	Conduit	**leds**				
☑		**key**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	**clk_0**			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_1170	0x0800_117f	
		external_connection	Conduit	**key_external_conne...**				
☑		**timer_0**	Interval Timer Intel FPGA IP					
		clk	Clock Input	Double-click to export	**clk_0**			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_1040	0x0800_107f	
		irq	Interrupt Sender	Double-click to export	[clk]			2
☑		**spi_0**	SPI (3 Wire Serial) Intel FPGA IP					
		clk	Clock Input	Double-click to export	**clk_0**			
		reset	Reset Input	Double-click to export	[clk]			
		spi_control_port	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_10a0	0x0800_10bf	
		irq	Interrupt Sender	Double-click to export	[clk]			3
		external	Conduit	**spi0**				
☑		**keycode1**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	**clk_0**			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_1160	0x0800_116f	
		external_connection	Conduit	**keycode1**				

Platform Designer Blocks 2/2

Descriptions

clk_0

This block is the Clock Source and generates the main clock that was used in our lab. It also produces the reset output which will be used throughout our lab when the `Reset` signal is connected. The clock and reset outputs are connected to all the other blocks so that the program can function in sync.

nios2_gen2_0

This is our NIOS II Processor block, and instantiates the processor we will use. It includes the master connections for data and instructions which are then driven to the slave peripherals used throughout our lab. Once configured with the clock and memory, it can be used to manage data communication between the different components of the lab.

onchip_memory2_0

This block instantiates the On-Chip Memory we need in the lab. It serves as one of the slave peripherals of the NIOS II processor so that it can pull data from the on-chip memory module.

sdram, sdram_pll

However our on-chip memory only has limited storage, so we also needed to instantiate an external off chip memory. The SDRAM Controller accomplishes this goal. Since the off-chip memory has complex accessing method, it cannot interact with the data bus directly. Therefore, we also declare the ALTPLL IP block which provides a separate clock signal for the sdram block. This separate timing method essentially allows our off-chip memory to interact with the Avalon bus, where the actual data transfer happens.

sysid_qsys_0

This block simply serves as a system ID checker, and its purpose is to ensure the compatibility between hardware and software. It prevents the user from accidently loading software onto the FPGA that has an older or incompatible NIOS II configuration.

Switches and LEDs

We also instantiated PIO blocks for both the switches and the display LEDs on the FPGA. Their purpose was to act as IO signals for week 1 of the lab, which the accumulation data values entering the bus as data inputs and then the correct sum being reflected on the LEDs as outputs.

jtag_uart_0

The JTAG UART peripheral allows for communication between the terminal of the host device and the NIOS II processor. It essentially allows us to use `printf` statements that we can use to debug our software while developing. However it is important to note that this is not considered good user interface because it requires a programming cable to be connected and Altera installed on the user's computer.

usb_irq, usb_gpx, usb_rst

These blocks were necessary on Platform Designer for proper connection to the USB chipset. The `irq` is the interrupt signal that is generated by the USB, and indicates a change in USB status, so whether or not it is plugged in. The `gpx` signal is used to control the power to a USB device. It controls an external power supply that provides additional power to the USB device in case the power provided by the port is not significant enough. The `rst` serves as a reset signal, when asserted, tells the device to return to its default or previous state.

keycode/keycode1

This block is responsible for handling the keycodes that represent which key is pressed on the keyboard. The data is outputted to the user via the hex display on the FPGA. We instantiated two different blocks for keycodes so that we could have "ghost keys", which essentially allows for multiple keyboard keycodes to be passed to the NIOS II processor at the same time.

hex_digits_pio, leds_pio

These function as output peripherals that get data from the Avalon bus when signaled by the master processor and output it using the displays on the FPGA.

key

Serves as an input for the program whose signal comes from the FPGA buttons. It communicates with the processor as another slave peripheral.

timer_0

This block provides a timer function that is used by the FPGA. It provides a timer that can be used by the rest of the program as a slave peripheral and allows for better communication and performance. It can generate periodic time signals and includes control signals for the timer.

spi_0

This is the SPI peripheral block is instantiated on platform designer to function with the rest of the program and the Avalon bus. This component also includes the interrupt signal and is integrated with the overall program so that all the other devices can communicate with one another.

Software Description

Since implemented all of our game design and logic in hardware, we did not make any changes many changes to our lab 6 software. We just edited it to account for our additional keycode, so that two keycodes could be received by our processor instead of one.

Design Resources and Statistics

LUT	Memory(BRAM)	Flip-Flop
9,481 elements	1,005,696 bits	2754 registers
Static Power	**Dynamic Power**	**Total Power**
96.75 mW	110.79 mW	228.91 mW

In our final project, we used an astounding 9,481 lookup tables. This is significantly more than any lab that We have ever done in this class but the staggering amount is justifiable. In our lab, we not only combined multiple large labs from this semester, but we also built a lot upon it. The same goes for our other statistics as well. For example, we used over an astounding one million BRAM bits which significantly lengthened the compilation time of our lab. One major change that may not immediately be noticed, however, is the spike in dynamic power. In previous labs, our dynamic power would be between 1 and two mW but in this final project, it is 110.79 mW. Since we are not always using the many, many blocks in our program, the system's blocks can be placed in low-power sleep modes when inactive, again justifying the large changes in the design resources and statistics.

Conclusion

With the knowledge from the course this semester and what we gleaned from alternate sources, we were able to successfully create a level of the popular NES game: Super Mario Bros. We were able to modify the maps to implement lives, randomized boxes, exponential jumping, and so much more. We could accumulate scores through interactions with NPCs and obstacles and also use multiple finite state machines to represent lives, the gameplay, and animations of Mario. Due to hardware limitations and a crunch on time, we were able to compile our knowledge of SystemVerilog and use it all in recreating the single level over the entire game. We could complete every feature that we promised for game functionality and we did not stop there. Although we missed a few features that may have increased the difficulty score, we were able to use our extra time to include many other unique features that make the game much better for the player. When we began this course, it seemed like it was Goliath and we were David; however, we rose up to the task and successfully chose and completed a project which was quite hard to accomplish.