ECE 385 Experiment 5: Lab Report

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

Introduction

In this lab we were tasked with implementing a system that uses USB input to the FPGA and VGA output from the FPGA to control the movement of a ball on a screen. Using the WASD keys typed on a keyboard, the ball on the screen will either move up, down, left, or right. In order to accomplish this, we must use the NIOS-II microprocessor with our FPGA and interface our USB and VGA.

The NIOS-II is a microprocessor that can be implemented on the MAX10 FPGA and customized to have a wide range of functionalities: control, processing, and communication. It can be configured to execute custom instructions, making it a suitable choice for various embedded system applications. The NIOS-II processor supports several communication protocols, such as UART, SPI, and Ethernet, allowing us to interface with outside devices. Also, it can be connected to on-chip memory, off-chip memory, and peripherals to support different applications which we do in our labs.

The USB/VGA interface is an embedded system that allows us users to connect a USB keyboard to a VGA monitor so we can input characters through the keyboard and display them on the monitor or use the characters as an input. The system uses a NIOS II processor to handle USB enumeration and communication, and a VGA controller to generate video signals for display on the monitor. The USB keyboard is connected to the system through a SPI interface, while the VGA monitor is connected through a VGA output port. The keyboard input is processed by the Nios II processor and used or displayed on the monitor through the VGA controller. The system also includes a JTAG UART peripheral for debugging and an Interval Timer to keep track of USB time-outs.

Description and NIOS-II System Diagram

Hardware Component Description

For this lab, the only hardware component was the **Platform Designer** modules we created. Platform Designer is used to instantiate IP blocks, including the NIOS II. We will also instantiate a SDRAM control and PIO blocks to handle inputs and outputs to our lab. Table 1 shows all the components we instantiated in our Platform Designer.

Table 1: Platform Designer Components

NAME	DESCRIPTION
clk_0	Clock Source
nios2_gen2_0	NIOS II Processor
onchip_memory2_0	On-Chip Memory
sdram	SDRAM Controller
sdram_pll	ALTPLL Intel FPGA IP
sysid_qsys_0	System ID Peripheral
jtag_uart_0	JTAG UART Intel FPGA
timer_0	Interval Timer Intel FPGA
spi_0	SPI (3 Wire Serial) Intel FPGA
keycode	PIO
usb_irq, usb_gpx, usb_rst	PIO
hex_digits_pio	PIO
leds_pio	PIO
key	PIO

The first thing we do on platform designer is instantiate a clock so that our hardware component operates correctly. For this lab, we used 50MHz clock. Then we add our NIOS II processor. NIOS works as a simple CPU that can be instantiated and then used with our FPGA. We also include a small on-chip RAM to serve as our memory. While this isn't necessary, it allows our program to access registers and memory faster. However our on-chip memory has limited capacity, so our platform designer component also includes off-chip SDRAM to store our software program. Since the SDRAM requires precise timing, we included a PLL component which allows us to compensate for clock skew. The System ID Peripheral component we added allows us to check if our software is compatible with our hardware. We also instantiated multiple PIO blocks, which lets our processor communicate with the inputs and outputs on the physical FPGA.

Lab 6.1 I/O

For week 1 of the lab, our goal was to modify the hardware and software components to perform accumulation on the LED using the values from the switches as inputs. The LEDs would display these values in binary and should be 0 on start up. We also included a `Reset` key which would clear the accumulator and return it to zero. `KEY[1]` on the FPGA was our `Accumulate` signal, and would send the inputs from the switches into the CPU. For each input and output of our program, we had to instantiate a different PIO block on the platform designer. For the LEDs we chose the direction as `Output` since they would be used to display our accumulated values. The data for the led peripheral only needs to access the data bus, as that is where it would get its values from. We also create the necessary connections between the peripherals and the NIOS II through the Avalon bus. We also instantiate the other I/O units the same way, but set their direction to inputs since the data from the switches and the buttons would go into the NIOS II. Once the logic is performed, the data would then be put on the bus and outputted by the LEDs on the FPGA.

NIOS 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 this lab. 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 lab, 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.

C Functions Descriptions

```c
void MAXreg_wr(BYTE reg, BYTE val);
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data);
BYTE MAXreg_rd(BYTE reg);
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data);

int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,
                           alt_u32 write_length,
                           const alt_u8* wdata,
                           alt_u32 read_length,
                           alt_u8* read_data,
                           alt_u32 flags);
```

The code block above shows the C functions we edited in our `usb_kb` folder. `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.

VGA Components

The VGA component of our lab is comprised of three main modules, `Ball.sv, VGA_controller.sv, and Color_Mapper.sv`. We instantiated all of our VGA files in our top level so that we could use our local variables to connect the modules and display the lab to our monitor.

The Ball module is used to draw the actual ball object on the VGA. It instantiates a center pixel for the ball based on the x and y axis, the step size of the ball based on pixels in both the x and y direction, and the edges of the screen, which stretch from 0 to 639px in the x direction and 0 to 479 in the y direction. The Ball module first uses 4 if statements to check if the ball is at the edge of the screen, by comparing the position with the edge pixels. If the ball is in the middle somewhere, then the ball does not bounce and continues in motion based on the case statements. The case statements are based on the `keycode` value, which represent either `W,A,S,D` based on the direction the user wants the ball to move.

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.

The Color Mapper module pulls the signals from the Ball and VGA controller module, and brings them together to output the correct pixel colors to the screen. The ball is first generated by using the standard equation of a circle. It also instantiates a 1-bit signal that tells the program to draw the pixel to the ball or the background. If the `ball_on` signal is high, the outputted color will be a reddish shade. However if the signal is low, the pixel will reflect a bluish color for the background.

Ultimately, these three modules come together in the top level module to actually control the FPGA and draw the pixels to the screen. They also handle cases from input devices such as a keyboard in our case, and can control movement of generated objects.

Top Level Block Diagram

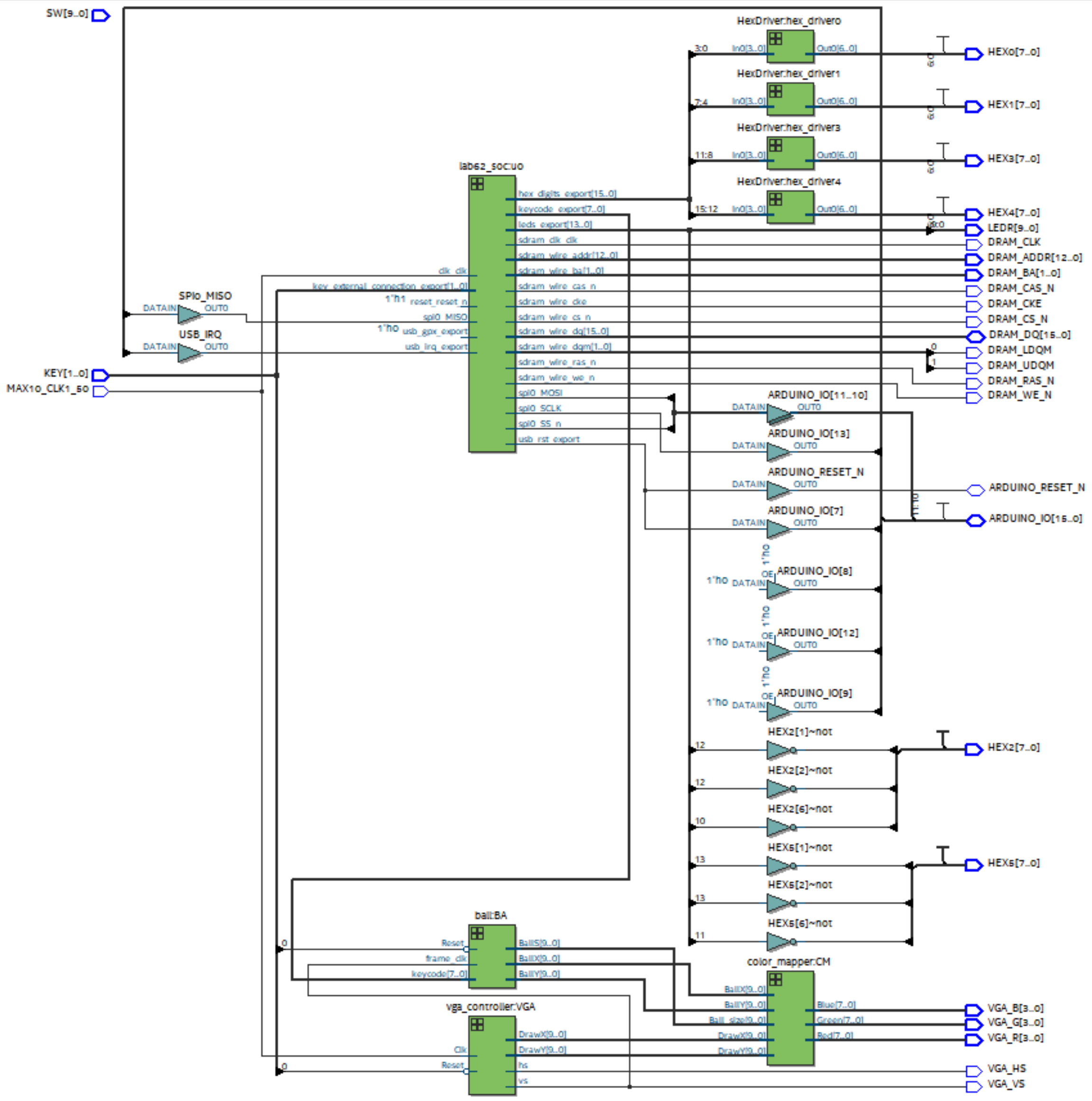


Figure 1: Top Level RTL Diagram

SV Modules

lab62_soc.v

```
module lab62_soc (
    input  wire        clk_clk,                        //
              clk.clk
    output wire [15:0] hex_digits_export,              //
       hex_digits.export
    input  wire [1:0]  key_external_connection_export, //
key_external_connection.export
    output wire [7:0]  keycode_export,                 //
          keycode.export
    output wire [13:0] leds_export,                    //
             leds.export
    input  wire        reset_reset_n,                  //
            reset.reset_n
    output wire        sdram_clk_clk,                  //
        sdram_clk.clk
    output wire [12:0] sdram_wire_addr,                //
       sdram_wire.addr
    output wire [1:0]  sdram_wire_ba,                  //
                 .ba
    output wire        sdram_wire_cas_n,               //
                 .cas_n
    output wire        sdram_wire_cke,                 //
                 .cke
    output wire        sdram_wire_cs_n,                //
                 .cs_n
    inout  wire [15:0] sdram_wire_dq,                  //
                 .dq
    output wire [1:0]  sdram_wire_dqm,                 //
                 .dqm
    output wire        sdram_wire_ras_n,               //
                 .ras_n
    output wire        sdram_wire_we_n,                //
                 .we_n
    input  wire        spi0_MISO,                      //
             spi0.MISO
    output wire        spi0_MOSI,                      //
                 .MOSI
    output wire        spi0_SCLK,                  // .SCLK
    output wire        spi0_SS_n,                  // .SS_n
    input  wire        usb_gpx_export,             // usb_gpx.export
    input  wire        usb_irq_export,             // usb_irq.export
    output wire        usb_rst_export              // usb_rst.export
  );
```

The module represents the blocks we created in platform designer. It instantiates the platform designer so that it can be used with the rest of the hardware code. The blocks we created in the Platform Designer are further explained in the *System Level Block Diagram* section of our lab report.

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.

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,
	output				DRAM_WE_N,
	output				DRAM_CAS_N,
	output				DRAM_RAS_N,

	///////// VGA /////////
	output				VGA_HS,
	output				VGA_VS,
	output	[ 3: 0]		VGA_R,
	output	[ 3: 0]		VGA_G,
	output	[ 3: 0]		VGA_B,


	///////// ARDUINO /////////
	inout	[15: 0]		ARDUINO_IO,
	inout				ARDUINO_RESET_N

);
```

`lab62.sv` is our top level module where we 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.

HexDriver.sv

```
module HexDriver (input   [3:0]   In0,
                  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.sv

```
module  color_mapper ( input        [9:0] BallX, BallY, DrawX, DrawY, Ball_size,
                       output logic [7:0]  Red, Green, Blue );
```

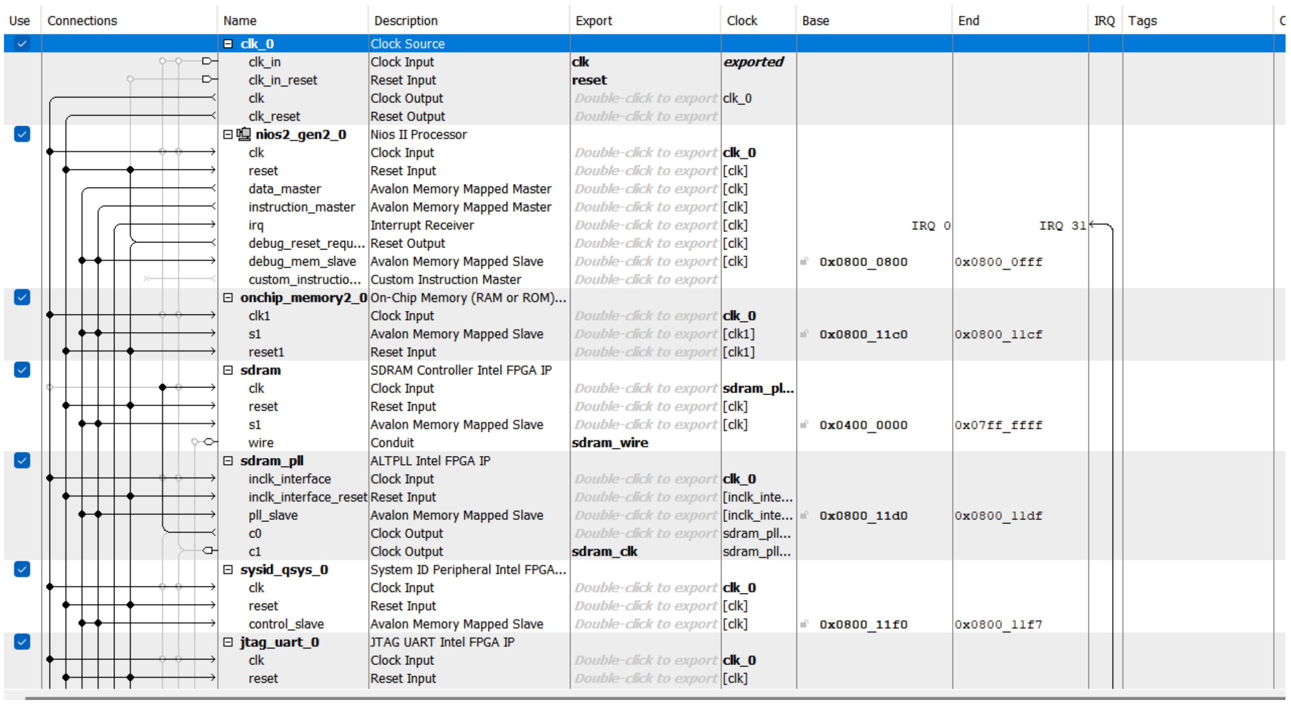
This is the color mapper module, and takes in the ball coordinates, size, and the pixel coordinate and outs the color to be use in the foreground or background. It declares both a `DistX` and `DistY` using the inputs, and uses it to draw the actual shape of the ball using the equation of a circle. It then checks if the pixel is within the bounds of the shape, and assigns the respective color. The overall purpose is to create the ball shape that is used in the lab and to assign pixel colors so that it can be seen on the monitor.

ball.sv

```
module  ball ( input Reset, frame_clk,
          input [7:0] keycode,
               output [9:0]  BallX, BallY, BallS );
```

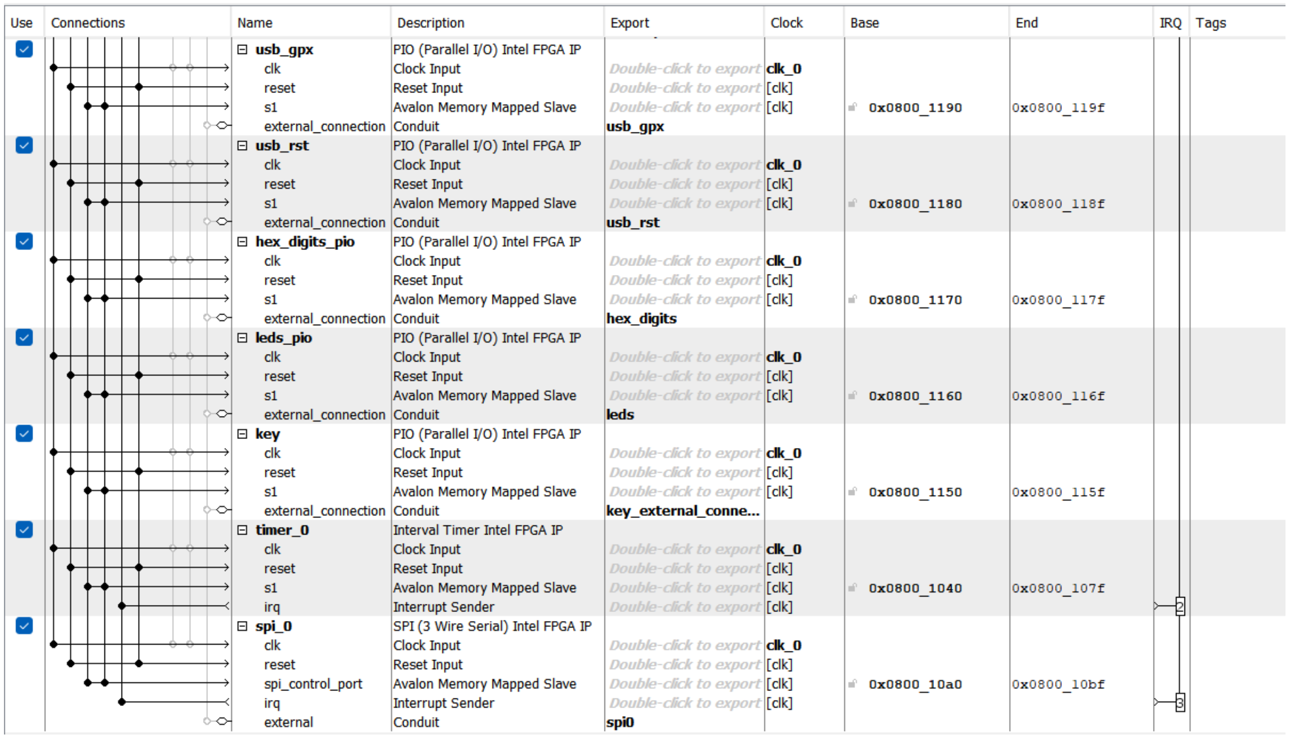
This is the ball module, and takes the keycodes from the keyboard as input along with the frame clock and reset signal. The reset signal is generate from the buttons on the FPGA, and the frame clock is the VGA vertical sync signal which allows a synchronous time for the ball parameters to be defined. While this module does not actually draw the ball shape, it does assign important variables such as the center coordinates, the size, position, and ball motion parameters. It also assigns movement directions using a case statement with the keycode depending on which key is pressed. The purpose of this module is to essentially output the signals used by the `Color_Mapper` module to draw the ball shape to be displayed on the screen.

System Level Block Diagram



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
✓		⊟ **clk_0**	Clock Source						
		clk_in	Clock Input	**clk**	*exported*				
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click to export*	clk_0				
		clk_reset	Reset Output	*Double-click to export*					
✓		⊟ **nios2_gen2_0**	Nios II Processor						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ 31		
		debug_reset_requ...	Reset Output	*Double-click to export*	[clk]				
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_0800	0x0800_0fff		
		custom_instructio...	Custom Instruction Master	*Double-click to export*					
✓		⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM)...						
		clk1	Clock Input	*Double-click to export*	**clk_0**				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0800_11c0	0x0800_11cf		
		reset1	Reset Input	*Double-click to export*	[clk1]				
✓		⊟ **sdram**	SDRAM Controller Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**sdram_pl...**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0400_0000	0x07ff_ffff		
		wire	Conduit	**sdram_wire**					
✓		⊟ **sdram_pll**	ALTPLL Intel FPGA IP						
		inclk_interface	Clock Input	*Double-click to export*	**clk_0**				
		inclk_interface_reset	Reset Input	*Double-click to export*	[inclk_inte...				
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_inte...	0x0800_11d0	0x0800_11df		
		c0	Clock Output	*Double-click to export*	sdram_pll...				
		c1	Clock Output	**sdram_clk**	sdram_pll...				
✓		⊟ **sysid_qsys_0**	System ID Peripheral Intel FPGA...						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_11f0	0x0800_11f7		
✓		⊟ **jtag_uart_0**	JTAG UART Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				

Figure 2: Platform Designer View (clk _0 to jtag_uart_0)



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
✓		⊟ **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_1190	0x0800_119f		
		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_1180	0x0800_118f		
		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_1170	0x0800_117f		
		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_1160	0x0800_116f		
		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_1150	0x0800_115f		
		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**					

Figure 3: Platform Designer View (usb_gpx to spi_0)

Week 1 Blocks

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.

Week 2 Blocks

Week 2 also utilizes most of the blocks declared in week 1 such as the clock, memories, and NIOS II processor. However we did add some new blocks to handle the MAX3421E USB chip.

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

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.

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 Component

Question 1

One of the INQ questions asks about the blinker code, but you must also describe your accumulator.

In our code for the blinker, we had an infinite while loop in which we would wait, change the LSB of the LED_PIO, wait again, and change the LSB of the LED_PIO. In doing this, we would first keep our LED on for however long we waited, then changing the LSB would turn it off and we wait again until we turn it back on. Our accumulator, however, also needed the LED_PIO so we had to comment out our blinker code so we could implement a new function. We have four volatile unsigned int values, two of which are pointers. We want to store our values in switches in *SW and our button for run in *run_acc. Since we want to store our sum after each press of the run button in our LEDs, we have a temporary variable for sum and we also have a wait state variable to ensure that we only add on each press and not by holding down the run button. To do this, we had in if statement that would make the wait state signal low if the button was not pressed along with another if statement that would only add the value of our switches into sum if wait state was 0 and the button was pressed. In the second if statement we also store our sum in the LED_PIO and set wait state high so we can not only see our accumulated value but also not keep adding until the run button is pressed again.

Question 2

Describe the code you needed to fill in for the USB/SPI portion of the lab for Lab 6.2.

The `MAXreg_wr` function is used to write data into a register. An array is created with 2 elements, reg+2 and the value. The element ‘reg+2’ is used to specify the register to be written to and the element 'val' is the value to be written into that register. The function writes both the register and the value with a write length of two. It then checks if the write command is valid and returns a return code. No reading is done in this function.

The `MAXbytes_wr` function is used to write several bytes of data. An array of size nbytes+1 is created to hold the register to write to and the data to be written. A for-loop is used to assign data to the array. The function then writes the data using the `alt_avalon_spi`command function with a write length of the number of bytes plus one to account for the register. The function returns the data+nbytes.

The `MAXreg_rd` function is used to read data from a register. The variable 'val' is created to hold the value read from the register. The function writes to the register with a write length of 1 and then reads from the same register with a read length of 1. It then checks if the read command is valid and if it is, returns the value read.

The `MAXbytes_rd` function is used to read multiple bytes of data. The function writes to the register with a write length of 1 and then reads the specified number of bytes with a read length of `nbytes`. It then checks if the read command is valid and if it is, returns the `data+nbytes`.

Post Lab Questions

INQ5 -- What are the differences between the Nios II/e and Nios II/f CPUs?

One of the main differences is that the NIOS II/f CPU is faster than the NIOS II/e CPU because it has a higher bit instruction format. Since the NIOS II/f has 32-bit instructions where the NIOS II/e has 16-bit instructions, the NIOS II/f can execute instructions in one clock cycle where the NIOS II/e would need two. For more instructions, the faster the NIOS II/f will be in comparison. Furthermore, the NIOS II/f has some instructions that are not available on the NIOS II/e. NIOS II/e CPU is designed to use the least resources and logic elements thus it is slower because it can take multiple clock cycles before an instruction is executed. NIOS II/f CPU is designed to work fast and have as high of performance as possible.

INQ7 -- What advantage might on-chip memory have for program execution?

On-chip memory can provide faster access to data and instructions compared to off-chip memory. This is because on-chip memory is located on the same chip as the processor, which means that it has a shorter access time and lower latency. This can result in faster program execution and improved overall system performance. Additionally, on-chip memory can be used for code that needs to be executed quickly and without the delays of accessing external memory. The off-chip memory takes more time because to retrieve memory, it has to go through more datapaths.

INQ7 -- Note the bus connections coming from the NIOS II; is it a Von Neumann, “pure Harvard”, or “modified Harvard” machine and why?

The NIOS II In a modified Harvard architecture, there are separate buses for instruction and data, but the memory system is connected, allowing for flexibility when accessing the memory. In this case, the NIOS II has separate data and instruction buses, as well as an on-chip memory that can be accessed by both buses. This design allows for simultaneous fetching of instructions and data, which can result in faster program execution.

INQ8 -- Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?

The program bus is used for accessing instructions stored in memory, while the data bus is used for transferring data between memory and peripherals. In the case of the led peripheral, it only needs to receive data to control the LED lights, and doesn't require access to the program bus since it does not execute any instructions. Therefore, it only needs access to the data bus to receive data from the on-chip memory or any other peripheral that provides data. This reduces the number of bus cycles required for communication and frees up the program bus for accessing instructions, which is crucial for the proper functioning of the processor.

INQ8 -- Why does SDRAM require constant refreshing?

The SDRAM is composed of a transistor and capacitor charge per bit of information. Since the charge of the capacitors decay over time, the information is not guaranteed to be correct and therefore, must be refreshed often so that data is guaranteed to be correct.

INQ9 -- What is the maximum theoretical transfer rate to the SDRAM according to the timings given?

Data Width: 32 Bits.

Access Time: 5.5 ns

$\frac{32\ Bits}{5.5\ ns} * \frac{1\ Byte}{8\ Bits}$ = 727 MB/s

INQ9 -- The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?

The SDRAM cannot run too slow because there is a window of time in which the SDRAM transactions are valid and therefore, the SDRAM clock must toggle within that time period to capture the correct values after being refreshed. If the clock is too slow, it will capture incorrect values. Therefore, the SDRAM needs to be refreshed constantly to maintain the stored data which can be done by reading it and writing it back to the same memory location. If the SDRAM is run too slowly, the refresh cycle may not complete before the next refresh cycle is required, which can result in data loss. The minimum frequency requirement of 50 MHz ensures that the SDRAM is refreshed frequently enough to prevent data loss.

INQ11 -- You must now make a second clock, which goes out to the SDRAM chip itself, as recommended by Figure 11. Make another output by clicking clk c1, and verify it has the same settings, except that the phase shift should be -1ns. This puts the clock going out to the SDRAM chip (clk c1) 1ns behind of the controller clock (clk c0). Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.

The reason why we want a clock going to the SDRAM chip 1ns behind its controller clock is because the controller takes time for address, data, and control signals to be valid at the SDRAM pins for the chip which is why we need to delay the window of time of the clock where we capture those values within that time frame when the values are correct. The delay allows the SDRAM to receive and process commands before data is requested and if we did not have a delay there would be timing issues with the commands and data requests, leading to incorrect data storage and retrieval.

INQ14 -- What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?

The NIOS II starts executing from the start of the SDRAM which is address **x10000000**. This is done after assigning the addresses so the processor knows where to go in cases where there may be an exception or a reset in which the processor returns to guarantee that there is no memory overlap.

INQ21 -- Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code: const int my_constant[4] = {1, 2, 3, 4} will place 1, 2, 3, 4 into the .rodata segment.

.bss: region with uninitialized data (variables and constants). Example: int a;

.heap: region where memory is allocated. Example: int ptr = (int)malloc(sizeof(int));

.rodata: region of static constants (not variables) - read only data. Example: const int a = 0;

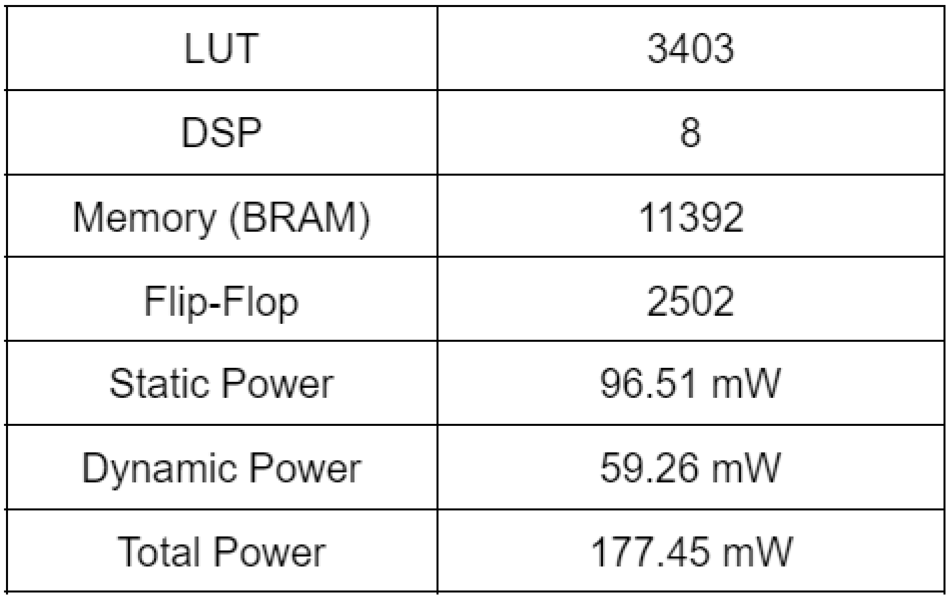
.rwdata: region of read/write data that can be changed. Example: int a = 1;

.stack: region of stack where the activation record and function calls are stored. Example: `int func(int a, int b)`

.text: region of text/strings. char a = "text";

SDRAM PARAMETER	NAME	PARAMETER VALUE
Data Width	[width]	16 bits
# Rows	[norws]	13 rows
# Columns	[ncols]	10 columns
# Chip Sets	[ncs]	1 cs
# Banks	[nbanks]	4 banks

Design Resources and Statistics



LUT	3403
DSP	8
Memory (BRAM)	11392
Flip-Flop	2502
Static Power	96.51 mW
Dynamic Power	59.26 mW
Total Power	177.45 mW

Figure 4: Design Resources and Statistics

From our design statistics table we can notice a few changes from the previous labs and see just how much more we are processing in this lab. Starting in the lookup tables, we are using an astounding 3403 which is almost three times of the LUTs used in lab 5 (1128). In this experiment, we began to use a lot more our FPGA by using the VGA and connecting hardware code in SystemVerilog with software code in C. Our memory usage is lower since we are not doing as many operations in this lab as in lab 5 but we are just doing more complex computations which do not take as much memory. Although the static power is similar to the previous lab, the dynamic power is much higher, specifically twenty times higher. This is because we have to use more power when we switch as we have much more activity in this lab requiring more power to switch. Using over 3000 lookup tables and more dynamic power in this lab allows us to complete a lot more with our FPGA.

Conclusion

In the first week of the lab, we ran into trouble with the accumulator code not working. We would have garbage values displayed on our FPGA LEDs and this was because our addresses were wrong for our PIO and for each of our new volatile unsigned int values. After we fixed this, we were then able to proceed with debugging our code to ensure functionality. In week two of the lab, we had finished each part but when we attempted to test, we would attempt to program to the FPGA and rather there being a ball and a blue background we had an empty screen. After further debugging, we had realized that our code was almost correct but our instantiation of the VGA controller module was incorrect leading to a blank screen. After we fixed that, we ran into the problem of our keys not functioning correctly. On the press of each key, garbage values would appear on our FPGA

Hex LEDs and the ball would not move. To fix this, we looked again at our mistake from week 1 and it was very similar since we had not used the correct memory addresses for the keys to be used therefore they did nothing. After finishing this, we had full functionality of our lab.

Any doubts we may have had after reading over the manual were resolved by attending lectures and watching past Q/A video streams as we were able to grasp a better visual understanding of the lab and how we can start implementing it. The manual provided us with knowledge of what was required of the design in the lab and what it must accomplish whereas the lectures and Q&A videos provided us with an understanding of how we may get started and with hints on designing our program. For next semester there is not anything that is unnecessarily difficult which can be improved upon for next semester.

In the end, we were able to successfully implement the lab we were tasked with since we could use keyboard inputs that we typed to move a ball in four directions. Using this knowledge, we can implement keyboard input on a screen in our final project, whatever that may be. This lab taught us the simple task of moving a ball in four directions but the implementation can be changed to be the keys of a piano, activate a power up, or even type a word to the console.

Extra Credit

For the extra credit, we were able to fix the glitch by adding the bounds check code into the case statements of the keycode press. So each time a keycode is read, the if statement checks if the ball is at the edge of the dimensions before the ball moves in the respective direction so that it does not move past the edge and bounces back into the screen.

Before

```
case (keycode)
          8'h04 : begin
                Ball_X_Motion <= -1;//A
                Ball_Y_Motion<= 0;
                end
```

After

```
          8'h04 : begin
          if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min )  // Ball is at the Left edge, BOUNCE!
            Ball_X_Motion <= Ball_X_Step;
              else
                Ball_X_Motion <= -1;//A
                Ball_Y_Motion<= 0;
                end
```

The before code block shows the case statements that were causing the bug in lab, and the after code block shows the case statement once we added the if state to check if the ball is at the edge. These code blocks only show one case for the upward movement, but apply to cases in all four directions. As a result, we implemented the if statement for all cases so that we could fix the direction throughout the whole border of the screen.