ECE 385 Lab Report 6

Experiment #6:

SOC with NIOS II in SystemVerilog

Spring 2022

Trusha Vernekar & Soham Manjrekar

Dohun Jeong (DJ)

tnv2 & sohammm2

Introduction

For Lab 6.1 and 6.2, the main goal was to create a NIOS II based system on the MAX10 FPGA. This is an IP that can be programmed using a high-level language, which was C in our case. Week 1 was to help us get familiar with the eclipse and NIOS II software and how to work with the System on Chip memory.

In week 1, we had to use the LEDs on our FPGA to create an accumulator which would have overflow accounted for. In week 2, we had to extend our previous interaction with the FPGA for it to interact along with the USB and VGA by displaying a ball on a monitor that could be influenced by the W, A, S, and D keys using a C program that was connected to the peripherals via the NIOS II processor.

Written Description and Diagrams of NIOS-II System

clk_0: Clock source, is a 50 MHz clock that is used in other hardware components

nios2_gen2_0: Nios II Processor, The embedded processor which runs the C code

onchip_memory2_0: On-Chip Memory, 16 bytes of RAM

SDRAM: SDRAM Controller, this initializes memory devices, translates read-and-write instructions to command signals, and manages memory banks

sdram_pll: ALTPLL, this device makes two 50 MHz clocks, one of which has a delay of 1 ns. These clocks are used specifically for the SDRAM since its clock needs to be stable and precise.

jtag_uart_0: JTAG UART, this communicates serial character streams between the host (computer and endpoint (NIOS II). It is also connected to Interrupt Request 1 which makes sure the CPU is not blocked between transmissions.

Sysid_qsys_0: System ID Peripheral, this assigns serial numbers to hardware and software that is compared when the software is run to ensure compatibility.

spi0: SPI (3 Wire Serial), the Serial Port Interface has multiple signals that allows the FPGA to communicate with the peripherals attached to it.

timer: Interval Timer, used to keep track of timeouts that are required by USB.

key, keycode, usb_irq, usb_gpx, usb_rst, hex_digits_pio, leds_pio: PIO, Parallel I/O is connected to the processors data bus, the usb PIOs are necessary to connect to the MAX3421, while the other PIOs are necessary to display bus data.

Code Functions

The accumulator portion of Lab 6 has I/O of two buttons, 10 switches, and LEDs. The two buttons represent Accumulate and Reset, the switches are used to represent a value and the LEDs will express the accumulated value from the switches. Every button press of accumulate sums, the value on the LEDs with the value on the switches until overflow. When Reset is pressed, the value on the leds is cleared.

NIOS II has 4 functions through which it interacts with the MAX3421E and VGA components. MAXreg_wr is used to write registers to the MAX3421E chip, MAXreg_rd is used to read registers from the MAX3421E chip, MAXbytes_wr, is used to write more than 1 byte to a register, and finally, MAXbytes_rd is used to read more than 1 byte from a register. To interact with the VGA components the processor relays the 4 functions information to lab62.sv, the top level on the FPGA project, which is used to figure out where the ball is and where the ball is going, which is then passed through other modules until VGA_controller where horizontal sync, vertical sync, red, green, blue signals are passed out.

SPI has multiple signals that let the FPGA communicate with its connected peripherals. The slave select signal chooses a specific peripheral to communicate with. The master-out-slave-in and master-in-slave-out signals decide whether the FPGA is writing or reading from its peripherals respectively. Finally, SCLK is the clock used for the signals to make sure the data is stable during a read.

There were 4 functions that we had to modify for this lab. These included MAXreg_wr(BYTE reg, BYTE val), MAXbytes_wr (BYTE reg, BYTE nbytes, BYTE* data), MAXreg_rd (BYTE reg), and MAXbytes_rd (BYTE reg, BYTE nbytes, BYTE* data).
MAXreg_wr(BYTE reg, BYTE val) is a void function, meaning it does not return anything. This function writes the register in the function argument to the MAX3421E via SPI. MAXbytes_wr (BYTE reg, BYTE nbytes, BYTE* data) on the other hand returns a pointer to a memory address after writing the register. It returns (nbytes + data). MAXreg_rd (BYTE reg) reads the register specified in the function argument from MAX3421E via SPI. This function returns the data that is in the least significant bit of the data that is read by the function. MAXbytes_rd (BYTE reg, BYTE nbytes, BYTE* data) again returns a pointer to a memory address after writing the register. It returns (nbytes + data).

There were 3 main modules that controlled the VGA operations for us in this lab. These were the Ball module called ball.sv, Color Mapper module called color_mapper.sv and VGA controller module called vga_controller.sv.

In the ball.sv file, the movements of the ball were defined and the constraints for the movement of the ball to make sure that the ball would bounce off a wall when it hits a wall, instead of going through the wall and disappearing. The diagonal movement of the ball was also eliminated in this module.

In the Color mapper module, the shape and size of the ball were created and the colors for the ball and background were decided depending upon the current position of the ball.

In the VGA Controller module, the way the ball moves was defined, rather the signals that make the ball move were defined. There was a sync in the vertical and horizontal directions which made sure that the ball would appear as the ball moved further on the VGA screen.

Top-Level Block Diagram

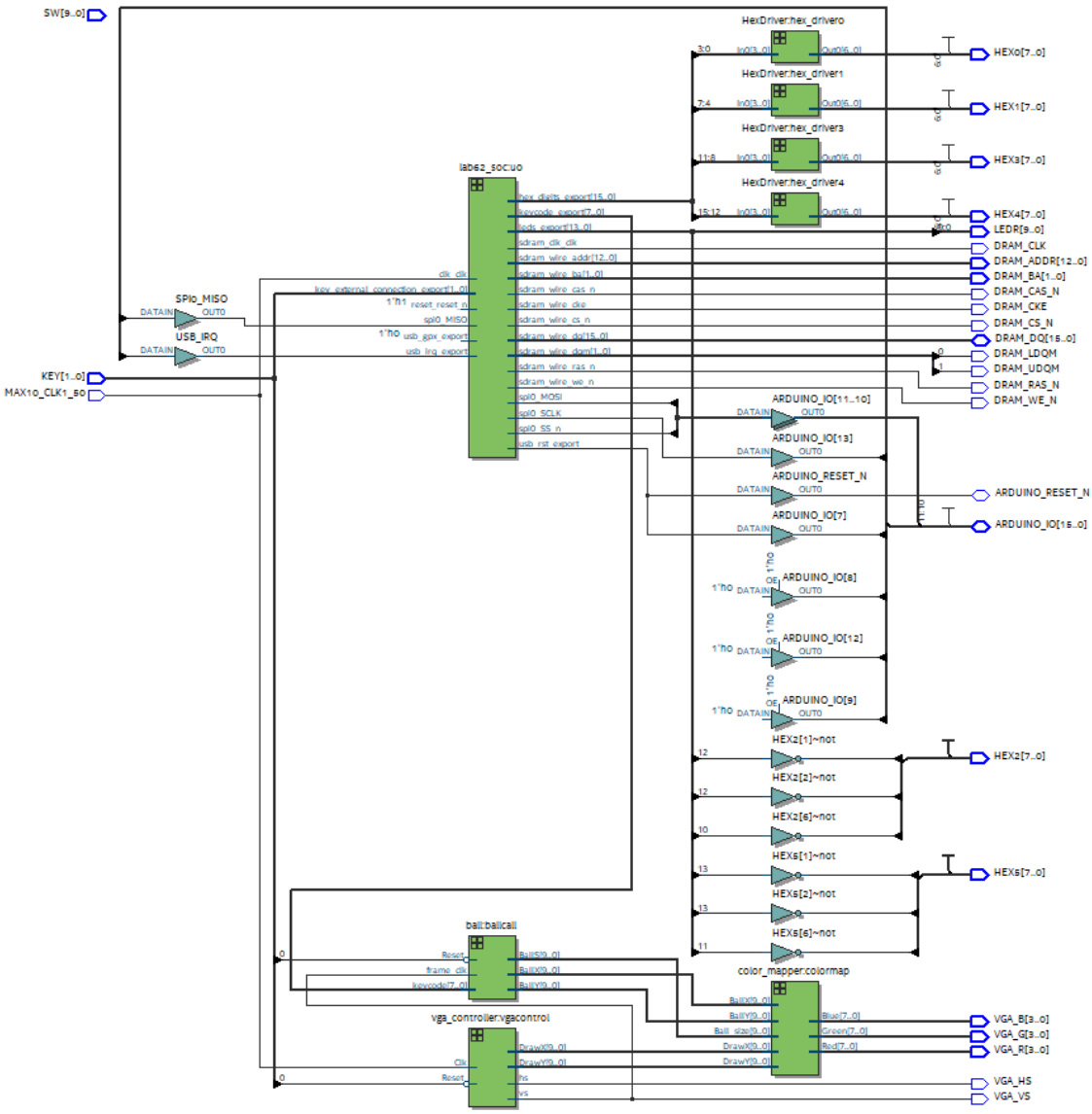

Figure 1: RTL Block Diagram

System Level Block Diagram (this is new for labs 6 & 7)

Use	C...	Name	Description	Export	Clock	Base	End	IRQ
☑		clk_0	Clock Source		*exported*			
☑		nios2_gen2_0	Nios II Processor		clk_0	0x0000_1000	0x0000_17ff	
☑		onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...		clk_0	0x0000_0000	0x0000_000f	
☑		led	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_0070	0x0000_007f	
☑		sdram	SDRAM Controller Intel FPGA IP		sdram_pll_...	0x0800_0000	0x0bff_ffff	
☑		sdram_pll	ALTPLL Intel FPGA IP		clk_0	0x0000_0080	0x0000_008f	
☑		sysid_qsys_0	System ID Peripheral Intel FPGA IP		clk_0	0x0000_0098	0x0000_009f	
☑		switch	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_0060	0x0000_006f	
☑		accumulate	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_0050	0x0000_005f	

Figure 2: 6.1 Platform Designer

Use	C...	Name	Description	Export	Clock	Base	End	IRQ
☑		clk_0	Clock Source		*exported*			
☑		nios2_gen2_0	Nios II Processor		clk_0	0x0000_1000	0x0000_17ff	
☑		onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...		clk_0	0x0000_0000	0x0000_000f	
☑		sdram	SDRAM Controller Intel FPGA IP		sdram_pll_...	0x0800_0000	0x0bff_ffff	
☑		sdram_pll	ALTPLL Intel FPGA IP		clk_0	0x0000_01c0	0x0000_01cf	
☑		sysid_qsys_0	System ID Peripheral Intel FPGA IP		clk_0	0x0000_01e0	0x0000_01e7	
☑		jtag_uart_0	JTAG UART Intel FPGA IP		clk_0	0x0000_01e8	0x0000_01ef	
☑		keycode	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_01b0	0x0000_01bf	
☑		usb_irq	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_01a0	0x0000_01af	
☑		usb_gpx	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_0190	0x0000_019f	
☑		usb_rst	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_0150	0x0000_015f	
☑		hex_digits_pio	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_0160	0x0000_016f	
☑		leds_pio	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_0170	0x0000_017f	
☑		key	PIO (Parallel I/O) Intel FPGA IP		clk_0	0x0000_0180	0x0000_018f	
☑		timer	Interval Timer Intel FPGA IP		clk_0	0x0000_0080	0x0000_00bf	
☑		spi0	SPI (3 Wire Serial) Intel FPGA IP		clk_0	0x0000_00c0	0x0000_00df	

Figure 3: 6.2 Platform Designer

Platform Designer Modules (Core)

clk_0: This module is a 50 MHz clock created by the FPGA that provides clock and reset inputs to other modules

nios2_gen2_0: This module is the 32-bit processor, NIOS II/e, e for the economy, this processor is an optimized version of the full-fledged NIOS II/f, the processor gives out data and instruction as an output to other modules

onchip_memory2_0: This module is the on-chip memory, it can be used for higher speeds and quicker access to data.

SDRAM: This module is the off-chip memory, it is 512 MBits and takes instructions and data from the processor. This module unlike the others gets its clock from the sdram_pll module which accounts for the difference in phase between the master and slave clocks.

sdram_pll: This module is a clock created for the SDRAM, that has a 1ns delay

sysid_qsys_0: This module is used to verify that the loaded software on the FPGA is allowed to run on the hardware provided, ensuring that the FPGA is not incompatible with the code.

Platform Designer Modules (6.2)

jtag_uart_0: This module is used to communicate a serial bit stream between the FPGA and connected computer

keycode: This module is a Parallel I/O (PIO) that is used to identify which key on the keyboard has been pressed, it is 8 bits wide

key: This module is a 2-bit PIO that is used to represent the two buttons on the FPGA

usb_irq, usb_gpx, usb_rst: This module is a 1-bit PIO that is used to connect the keyboard to the FPGA

hex_digits_pio: This module is a 16-bit PIO that is used to output the keycode on the Hex Display

leds_pio: This module is an 8-bit PIO that is used to output to the leds above the switches on the FPGA

timer: This module is used to track time for NIOS II

spi0: This module is used to transfer data between the attached peripherals and the FPGA, and controls which peripheral is being read from or written to or if the FPGA is being read from or written to

Platform Designer Modules (6.1)

led: This module is an 8-bit PIO that is used to output the accumulated sum to the leds above the switches on the FPGA

switch: This module is a Parallel I/O (PIO) that is used to identify which switches are set, it is 10 bits wide

accumulate: This module is a 2-bit PIO that is used to represent the button on the FPGA for Run/Accumulate, that will add the led value with the value loaded on the switches

Describe in words the software component of the lab.

For Lab 6.1 we were asked to first create an SOC, after creating one that provided a clock, a processor, memory, and IO, our C code took advantage of the hardware to make a simple code that built on the existing blinker code to accumulate the value of the switches to the LEDs and reset them when needed. The reset was a functionality that existed already so we had to implement two new IO, switches and the accumulate button, and then using the same syntax by which the LEDs were called and made into useable variables for the lab, we made variables for the switches and the accumulate button, after that, we wrote code that would sum the value of the LEDs with the value of the switches until overflow at a value of 255 + 1 at which point it would continue counting where 255 + 1 = 0 and 255 + 2 = 1. We also had to include wait states to account for how long the button press lasts so that one press wouldn't be interpreted as multiple.

For 6.2 we were applying one function: alt_avalon_spi_command(args) in multiple ways to perform register and multiple byte value reads and writes. For MAXreg_wr we simply had to set the flags for write length to 2 and read length to 0, setting wdata to an array with values reg + 2, and val. For MAXbytes_wr we again are only writing data and reading nothing, and again setting wdata to an array of length nbytes + 1, since this array will be filled with the values of the array data (of size nbytes) and reg + 2, we set the first value of wdata to be reg + 2 then we use a for loop to store the values of data in the array then call the function to write the array wdata and the write length to be nbytes + 1, the same size of the array we are writing. For MAXreg_rd we set write length and read length to 1, set rdata to &reg, and set the value of wdata to &val (a created variable to store the write out), then val is returned out. Finally, for MAXbytes_rd we do similar to MAXbytes_wr, except we set read length to 1 and write length to nbytes, rdata is &reg, and wdata is set to the data array.

Extra Credit Assignment (1 point) for Lab Report

Yes, in its unedited state the identified bug in the ball.sv file is that when long-pressing the A or W keys on the keyboard the ball will vanish off of the top or left side of the screen, after disappearing the ball then reappears at the opposite side where it disappeared off of. This occurs on the top and left side of the screen and not the right and bottom side because the top and left of the screen are the 0th row and column respectively, there are more columns and rows on the right and bottom they are simply blanked out with the VGA_controller. To solve this specific glitch

we decided to include the check of which key is being pressed within the else conditional when the ball is in the middle of the screen so that only when the ball is in the middle of the screen a keypress and subsequent direction is declared, otherwise the ball will bounce off the edges regardless of if the keyboard is being pressed or not.

```
else
begin
     if ( (Ball_Y_Pos + Ball_Size) >= Ball_Y_Max )  // Ball is at the bottom edge, BOUNCE!
         Ball_Y_Motion <= (~ (Ball_Y_Step) + 1'b1);  // 2's complement.
     else if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min )  // Ball is at the top edge, BOUNCE!
         //Ball_Y_Motion <= (~ (Ball_Y_Step) + 1'b1);
         Ball_Y_Motion <= Ball_Y_Step;
     else if ( (Ball_X_Pos + Ball_Size) >= Ball_X_Max )  // Ball is at the Right edge, BOUNCE!
         Ball_X_Motion <= (~ (Ball_X_Step) + 1'b1);  // 2's complement.
     else if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min )  // Ball is at the Left edge, BOUNCE
         Ball_X_Motion <= Ball_X_Step;
     else
       begin
         Ball_Y_Motion <= Ball_Y_Motion;  // Ball is somewhere in the middle, don't bounce, just keep moving
     case (keycode)
       8'h04 : begin

                Ball_X_Motion <= -1;//A
                Ball_Y_Motion<= 0;
               end
       8'h07 : begin

                Ball_X_Motion <= 1;//D
                Ball_Y_Motion <= 0;
                end
       8'h16 : begin

                Ball_Y_Motion <= 1;//S
                Ball_X_Motion <= 0;
               end

       8'h1A : begin
                Ball_Y_Motion <= -1;//W
                Ball_X_Motion <= 0;
               end
         default: ;
     endcase
     end
```

Figure 4: Ball.sv edit

Answers to all INQ & Post lab questions

Question 1] What are the differences between the Nios II/e and Nios II/f CPUs?

Answer] In the two names, the e suggests that it is economic and the f suggests that it is fast. The reason why Nios II/e is economic is that it uses the least number of logic elements and resources. The reason why the Nios II/f is faster is that it has a faster on-chip processor, which provides higher performance.

Question 2] What advantage might on-chip memory have for program execution?

Answer] One of the advantages is that it acts as storage on the chip processor. This makes it more efficient for program execution because it reduces the time taken for data more frequently accessed.

Question 3] Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?

Answer] The NIOS II is a "modified Harvard" machine because the memory between the instruction and the data master buses is shared.

Question 4] 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?

Answer] The LED peripheral only needs access to the data bus because it requires the data to decide which LED must be on or off depending on the data. It does not require access to the program bus because it does not need to perform any operation, so access to program information is not required.

Question 5] Why does SDRAM require constant refreshing?

Answer] The SDRAM requires constant refreshing because it consists of multiple capacitors and transistors which have bits of information. Due to the nature of capacitors, the charge decays as time passes, which means that valuable information about the program or data is lost. Due to this, the information is not always guaranteed to be correct. Refreshing ensures that the information is always up to date.

Question 6] What is the maximum theoretical transfer rate to the SDRAM according to the timings given?

Answer] The access time is 5.4 ns. Our data width was 32 bits. So, the access time was (32/5.4)/8 = 740.74 MB/s.

Question 7] The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?

Answer] The SDRAM has a specific time frame in which the transactions are valid. If the clock is slower than this time frame, then the SDRAM can have incorrect values due to the delays due to the capacitors (as mentioned in the prior questions).

Question 8] This puts the clock going out to the SDRAM chip (clk c1) 1ns behind the controller clock (clk c0). Why do we need to do this?

Answer] We need this time delay because it takes time for the address, control and data signals to be valid at the pins of the SDRAM. The time delay gives it enough time to get the correct values in a specific time frame, be it to read or to write.

Question 9] What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?

Answer] The NIOS II starts executing from address x10000000. The execution happens after assigning the address because in case there is an exception or we hit the reset case, the address is assigned correctly to avoid memory overlap.

Question 10] 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.

Segment type	Description	Example
.bss	A region when a variable is declared but not initialized.	int i;
.heap	A region when the memory for the variable is allocated on the heap	int *i = (int)malloc(sizeof(int));
.rodata	A region when a variable is initialized and are static constants.	const int i = 0;
.rwdata	A region when a variable is used for both reading and writing	int i = 0;
.stack	A place in the stack where information about the function such as activation record is stored	int temp(int a, int b){ }
.text	A region for strings	char y = “string”;

Table 1: Segment type with description and example

Question 11] You will need to determine the following parameters to instantiate the SDRAM controller. Refer to the DE10-Lite schematic and the IS42S16320D SDRAM (datasheet (PDF). Make sure you are looking at the correct part of the datasheet

SDRAM Parameter	Short Name	Parameter Value
Data Width	width	16 bits
# of rows	Nrows	13
# of columns	ncols	10
# of chip selects	ncs	1
# of banks	nbanks	4

Table 2: SDRAM parameter with short name and parameter value

Document the Design Resources and Statistics in a table provided in the lab.

LUT	3415
DSP	4
Memory (BRAM)	55,296 bits
Flip-Flop	2443
Frequency	126.58 MHz
Static Power	96.41 mW
Dynamic Power	0.71 mW
Total Power	156.15 mW

Table 3: Design resources and statistics table

Conclusion

For both weeks, our circuit design was fully functioning. For week 1, the accumulator worked correctly and it accounted for the overflow correctly as well. As for week 2, the ball movement was correctly seen after implementing the read and write functions in NIOS II. One of the main issues we faced was in week two. After writing all the code and making sure everything was correct, there was a “reset timeout”. The way we fixed this was by rubbing a capacitor behind the IO shield of the FPGA. The error would magically disappear everything we did this. We realized that this was because the capacitance from our finger can recouple the PLL. This was a very interesting bug fix.

Overall, this lab was a really good way to get us familiar with the NIOS II software and the integration between hardware, software, and the VGA memory. This was a great introduction lab which will help us a lot in our final project.

Appendix A

Module: lab62.sv

Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW,

Outputs: [9:0] LEDR, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM_CLK, DRAM_CKE, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS [3:0] VGA_R, VGA_G, VGA_B, [12:0] DRAM_ADDR, [1:0] DRAM_BA

Inout: [15:0] DRAM_DQ, ARDUINO_IO, ARDUINO_RESET_N

Description: This is the top-level module used.

Purpose: The purpose of lab62 is to take input and instantiate modules

Module: color_mapper.sv

Inputs: [9:0] BallX, BallY, DrawX, DrawY, Ball_size

Outputs: [7:0] Red, Green, Blue

Description: determines the balls shape and size and decides what part of the screen should be ball or background

Purpose: Determine color of each pixel on screen

Module: VGA_controller.sv

Inputs: Clk, Reset

Outputs: hs, vs, pixel_clk, blank, sync

[9:0] DrawX, DrawY

Description: Handle vertical and horizontal sync while setting a DrawX and DrawY

Purpose: The purpose of VGA_controller is to shoot the “electron gun” across the screen and determine the hs and vs to draw pixels accordingly

Module: hexdriver.sv

Inputs: [3:0] In0,

Outputs: [6:0] Out0

Description: Takes a nibble or 4 bits and converts it to a 7 bit equivalent that is displayable on the HEX# output LEDs of the FPGA.
Purpose: To allow the seven segment displays to display what the input and output of our code are.

Module: ball.sv
Inputs: Reset, frame_clk, [7:0] keycode
Outputs: [9:0] BallX, BallY, BallS
Description: This module determines the position and motion of the ball depending on which key on the keyboard is pressed. The ball module updates the motion of the ball whenever a key is pressed.
Purpose: The ball module controls how the ball moves.