# ECE385

Fall 2023

Experiment #6

# Lab 6

Haoyu Huang Eitan Tse

Lab section: ZY 10/Day & Time: 10.30

Your TA's Name: Yang Zhou

## 1 Introduction

MicroBlaze is a soft-IP based 32-bit CPU which can be programmed using a high-level language which has 32-bit modified Harvard RISC architecture. In this lab, Microblaze is the system controller and handle tasks which do not need to be high performance (for example, user interface, data input and output) while an accelerator peripheral in the FPGA logic handles the high-performance operations. We have the Clocking Wizard which is used to generate and manage clock signals in FPGA designs. We also have a GPIO (General Purpose I/O). We can connect port to it to control LED. It also has UART (Universal Asynchronous Receiver Transmitter) to give us some basic printf support for debugging. In the first week we design a simple adder which is similar to lab3. We implement adder in software and set port in vivado, we can access the switch value by accessing register.

In the second week, we want to design a simple ball game. We can use keyboard to control the movement of the ball. Besides, we have a canvas where balls can be displayed. If the ball reaches the wall of the screen, it will bounce in the opposite direction. After programming the FPGA and connect it to monitor and keyboard. We will first see a ball located at the center of a gradient screen move right We can press the keyboard which will generate unique keycode for FPGA. We assign W(up), S(down), A(left), D(right) respectively. Users can control the movement of the ball by pressing the keyboard. The position of the ball will be updated in the ball module. We also have a module which generates the two clock signals used by the HDMI.

## 2.Module Description

### a. SV Module

Module: mb_usb_hdmi_top.sv
Input: Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd,
Output: gpio_usb_rst_tri_o,usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n,
[2:0] hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB,
[3:0] hex_gridB
Description: The module functions as the top module of the lab. The input of the module has basic Clk and reset signal. Other signals can be divided into USB signal, HDMI

signal and UART signal. We have instantiated our block design in the in the module. Besides, we instantiated clock wizard, VGA Sync signal generator, Real Digital VGA to HDMI converter, ball instance and color mapper here.
Purpose: The module is the top-level module of this lab which is used to instantiate other modules and connect them to the inputs and outputs needed.

Module: color_mapper.sv
Input: [9:0] BallX, BallY, DrawX, DrawY, Ball_size,
Output: [3:0] Red, Green, Blue
Description: The module has the location of the ball and the location of the pixel as input and has 3 channel color as output. The module is used to draw color according to the current position of pixel. There are two always blocks inside the module. If the pixel is within the ball range, it will be drawn to red. Otherwise we draw it as gradient background.
Purpose: The module is used to update color of pixel on the screen.

Module: ball.sv
Input Reset, frame_clk, [7:0] keycode,
Output [9:0]   BallX, BallY, BallS
Description: The module has reset signal, frame clock, keycode input as input. We use x_motion and y_motion to indicate the velocity of the ball. At first, x_motion is equal to 1 which indictae moving right. When keycode signal is detected, we will update the velocity. For example, if keycode is 8'h1A (W), the ball will clear x_motion and y_motion is set to 1. We also need to check the boundry of the wall after we detect the keycode. If the location of ball adds size of ball excess the boundry we need to move in opposite directions. We need to ensure that we detect keycode first then the boundry which can limit ball within the screen.
Purpose: The module is used to update the movement of the ball. When is keycode is pressed, it will change direction and will bounce when it reaches the wall. When the reset signal is high, it will be back to the center and move right.

Module: vga_controller.sv
Input: pixel_clk,   reset,
Output: hs, vs, active_nblank, sync,   drawX, drawY
Description: The module takes only a reset and pixel_clk as input. The horizontal sync signal is set to high until 640 pixels have been rendered, then it is set low. Vertical sync is set high after the entire frame has been rendered, and low otherwise. This is a holdover from the old CRTs, where some method would be needed to direct the electron beam to the appropriate positions after finishing rendering a line (reset to beginning of next line) or frame (reset to the top left of the screen). The active_nblank signal is there to dictate whether a pixel should be rendered at that coordinate or not (since the actual size of the counters extend beyond the 640x480 pixel screen size, for housekeeping purposes). DrawX and DrawY provide the current coordinates of the pixel being draw.

Purpose: This module generates vertical and horizontal sync signals, a screen blanking signal, and the current X and Y coordinates of the pixel being drawn. This is to generate control signals and logic that other modules can use to draw the pixels as necessary on screen.

## b.Core component:

MicroBlaze：

MicroBlaze is a soft-IP based 32-bit CPU which can be programmed using a high-level language which has 32-bit modified Harvard RISC architecture. In this lab, Microblaze is the system controller and handles tasks which do not need to be high performance. It can be configured with different instruction and data cache sizes, as well as various peripherals to suit the specific application requirements.

Clocking Wizard:

The Clocking Wizard is a clock generating tool that helps in generating clocking structures for the FPGA design. It is used to create clock signals with specific frequencies and phase relationships, which are essential for synchronizing various components in the design.

Processor System Reset:

The Processor System Reset IP block is used to control the reset signals for the MicroBlaze processor and other components in the design. It provides a way to control and synchronize the reset process to ensure the system starts up in a defined state. It is an active low reset.

MicroBlaze Local Memory:

Local memory refers to the on-chip memory resources available to the MicroBlaze processor. The MicroBlaze local memory is an integral part of the processor and is used for storing program code, data, and stack memory.

AXI Interconnect:

The AXI (Advanced extensible Interface) Interconnect is used to connect various IP blocks within an FPGA design. It provides a high-speed and flexible communication infrastructure. The AXI Interconnect can be configured to support different data widths, address ranges, and arbitration schemes to facilitate communication between different components like the MicroBlaze, memory, and peripherals.

The AXI Interconnect follows a bus-based architecture. Various masters (IP cores that initiate transactions) and slaves (IP cores that respond to transactions) are connected through the AXI Interconnect. This architecture allows for multiple masters and slaves to communicate simultaneously.

AXI Interrupt Controller:

The AXI Interrupt Controller controls interrupt signals within the design. It's responsible for handling and distributing interrupt requests from various peripherals to the MicroBlaze processor. It can allow the processor to respond to external inputs.

AXI GPIO:
AXI General Purpose Input/Output (GPIO) is an IP block used for interfacing with external digital I/O devices. It provides a way to read inputs and control outputs from the FPGA, allowing the system to interact with the external world. AXI GPIO can be configured to work with different numbers of pins and can be used for various purposes, such as controlling LEDs, buttons, or other digital devices.

AXI Uartlite:
The UART (Universal Asynchronous Receiver/Transmitter) communication module is designed to work with AXI (Advanced eXtensible Interface) bus architecture. UART communication is a standard method for serial data transfer, often used to establish communication between an FPGA or SoC system and external devices, such as microcontrollers, sensors, or other computers. In this lab we use UART to print some messages in serial terminal.

Concat:
The "Concat" IP core is a simple but important component used in FPGA designs. Its primary purpose is to concatenate or combine multiple input signals into a single output signal. This is particularly useful when merging multiple data or control signals into a single bus or register.

## Lab6.2:

AXI Quad SPI:
The AXI Quad SPI provides an interface between an AXI-4 bus to SPI devices. The SPI device in this case is the MAX3421E, which interfaces with USBs directly and is the slave. This allows the Microblaze to enumerate and read keycodes and use it to control other hardware.

AXI Timer:
The AXI Timer is an IP core designed to provide accurate timing and counting functions within an FPGA or SoC system. AXI Timers can be configured for various counting and timing functions, making them useful for generating time intervals, timeouts, or clocking signals. In this lab, since USB has many timeouts that require timekeeping in milliseconds, the timer allows the MicroBlaze core to keep track of when 10 milliseconds.

## c. Lab6.1 How the I/O works

In lab 6.1, we add AXI General Purpose Input/Output (GPIO) which is an IP block used for interfacing with external digital I/O devices. Thus, we have input and output port. In lab6.1 we want to build an accelerator with software. AXI GPIO has many data registers which can only be read or only written in this lab. We have input <Run>, <LED>, <SW>. <RUN> port is connected to 1 bit data register. <SW>, <LED> are connected to 16 bits data register. Every port indicates a data register in GPIO which can be checked by memory map and datasheet. Besides, we have active low reset input which can reset our IP block including our data register. We first read value from the register which store value of <SW>. Then we add the value of <SW> with local variable sum and use pointer to write back to LED address.

## d. MicroBlaze, MAX3421E USB chip and ball motion components

The MicroBlaze has the ability to read keyboard data to set the ball moving in the appropriate directions. Keycodes are read from the MAX3421E through SPI transactions done in the C code. These keycodes are then sent to the ball.sv module and HEX displays. The HEX displays allow visualization of what keycode is currently being read, while ball.sv reads one of the two keycodes outputted. It sets the ball in motion in the direction described by the key immediately, unless it runs into a wall or some other boundary where its behavior must change from the usual directional movement.

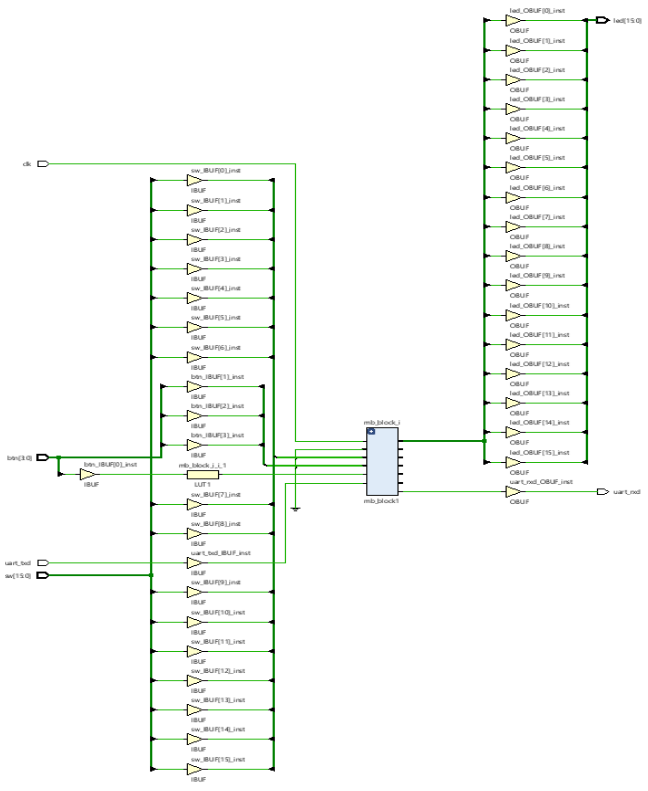
## e. Describe in detail the VGA operation, and how your Ball, Color Mapper, and the VGA controller modules interact.

The Ball module implements logic for moving the ball and resetting its position. It will execute checks based on its position, and bounces the ball or moves it in the direction based on the keystroke immediately. It synchronizes itself to the VSYNC pulse, generated from the VGA controller module. This means the Ball module updates itself after the frame is fully rendered. It also tracks the ball position and its size and provides them as outputs which are used by the color mapper. The VGA controller module also provides DrawX and DrawY coordinates, which are used to describe the current coordinate of the pixel being drawn. The ball position, size, DrawX, and DrawY are taken in by the ColorMapper module. It checks if the current pixel being drawn is within the radius of the ball (defined by ball_size). If so, the pixel is colored orange. Otherwise, it is colored a shade of gray. The background, where the current pixel is not within the radius of the ball, is a series of vertical segments that start at white on the left and get increasingly darker towards the right.

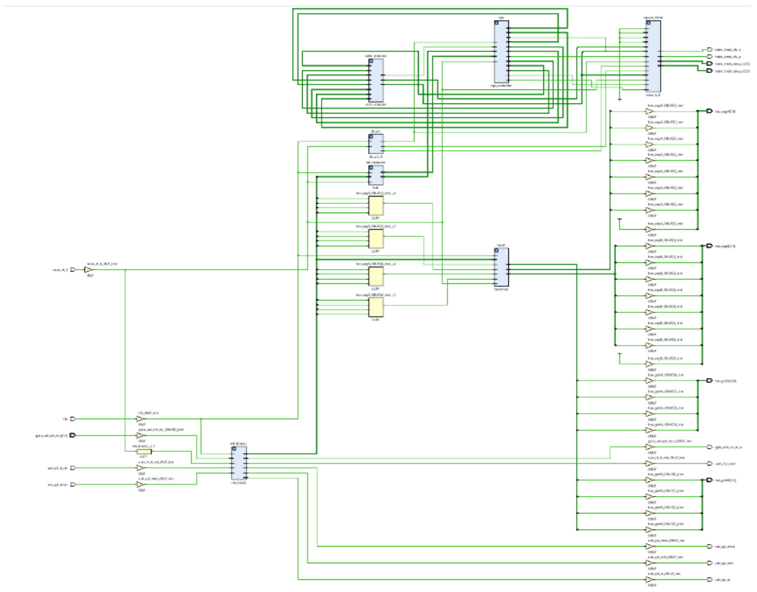
## f. Describe the VGA-HDMI IP, how does HDMI differ from VGA, how are they similar?

The VGA-HDMI IP shares similarities by rendering everything with RGB signals, horizontal and vertical sync, much like VGA does. However, HDMI has options for auxiliary data which includes audio, among a few different things. The outputs are also different; HDMI uses differential outputs for the HDMI port. VGA in contrast just produces an RGB output, hsync, vsync, and video blanking outputs to control the monitor. The VGA-HDMI IP uses much more complex logic to generate its outputs, creating control signals for the red and green channel and using the blue channel to encode the hsync and vsync signals, generating specific hardware to encode that. There are also multiple serializers to convert the data to appropriate TMDS_DATA and TMDS_CLK outputs, which have their own output buffers.

## RTL Diagram: Lab6.1



## RTL diagram: lab6.2



## 3. Software Component

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

The blinker code functions by blinking on the LEDs according to its current value, starting at 0. The LEDs are also reset and held to 0 as long as the reset button is pressed. This is accomplished by ANDing the LED_GPIO_DATA value with 0. The accumulator works by simply adding the value of the switches to the LED_GPIO_DATA whenever the accumulate button is pressed. To prevent button bounce from conducting multiple adds, a sleep() command is added immediately after updating the LED_GPIO_DATA values. Overflow is accounted for by seeing if the added value causes the stored data to change signs. If that does occur, the appropriate message is printed out (through UART) to the console.

### b. Written description of the SPI protocol and how it operates in the context of the MAX3421E.

SPI is a full-duplex communication protocol that typically involves two devices: a master and one or more slave devices.

Serial Clock (SCK) which is clock signal generated by master device.

Master Out, Slave In (MOSI): This is the data line on which the master sends data to the slave.

Master In, Slave Out (MISO): This is the data line on which the slave sends data to the master.

Slave Select (SS): This signal is used to select a specific slave device for communication. We have only one slave in the lab, thus we have ss [0:0].

<u>Operation:</u>

Initialization: The microcontroller configures the SPI communication parameters, such as clock speed, data format.
Select slave-device: SPI will select slave device according to SS signal. SPI has slave select register. Slaves select register at most have one bit that is high since slaves register is active low. As we only have one slave device, we only need the least significant bit in the register. When the bit is 0, it means MAX3421E is de-selected. Otherwise, we select MAX3421E.
Data Exchange: SPI sends commands and requests to the MAX3421E, specifying tasks like USB device enumeration, data transfers, or device control. For the request command, if we want to read value. We should write the register address to the slave first. If we want to write, we should first write register address plus two. It is because the second bit of command indicates the read/write mode. If the second is 1, we will write value to the slave.
Data Response: SPI will receive a response which indicates whether to read or write data successfully.

## c. Describe the purpose of each function you filled in in the C code (you do not need to describe the functions you did not modify) --- what does this function do in USB ?

Void MAXreg_wr (BYTE reg, BYTE val):
It serves the crucial purpose of configuring and controlling the MAX3421E USB host controller during USB communication. By writing values to specific registers within the MAX3421E, this function enables to set up the host controller for USB tasks, manage USB device interactions, and ensure proper operation of the USB communication protocol. We add 2 to the register value to signal that it is a write operation, because the second LSB signals whether it is a read or a write operation. W also instantiate a 2-byte buffer. The first byte contains the destination registers, which we have modified as above to signal the operation type. The second byte contains the value to write. Then, the slave_select (ss) signal is asserted high and the first slave is selected to ready the MAX3421E to write. Our sendbuffer is then transferred, with a status code recorded. If the status code is anything but 0, which signals a successful transfer, an error is printed out. Then, we set slave_select signal to low and disconnect from the MAX3421E.

BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data):
The function is to enable the efficient transmission of multiple bytes of data to a specific register within the MAX3421E USB host controller. This is useful for transmitting larger chunks of information that doesn't fit in a single byte, because not all data packets will be small. We add 2 to the register value to signal that it is a write signal. The function of the signal is very similar to the MAXreg_wr, except the send buffer is extended to nbytes+1 to include the multiple bytes of transmission instead of just 2

bytes, and the number of bytes specified to send in the XSpi_Transfer() function is also changed.

BYTE MAXreg_rd(BYTE reg):
This function enables reading from a target register. This is useful for grabbing new bytes of data from the device, such as reading keystrokes. To signal a read operation, the second LSB is left alone since it defaults to 0 and that signals a read operation. Similarly to the MAXreg_wr, we instantiate a 2-byte buffer except the second byte is left initialized to 0 to receive the data from the registers. Slave_select is set high to select the MAX3421E, and the first slave (the MAX3421E) is selected to read from. Then, using XSpi_Transfer(), the data from the MAX3421E is loaded into the second byte of the buffer. An error code is printed if something goes wrong with the operation and the status code returned does not signal a successful operation. Then, we set slave_select to low and disconnect from the MAX3421E, and returns the second byte of the buffer.

BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data):
This function reads multiple bytes from a target register. This is useful for obtaining larger bits of information from the device. It could be large chunks of data, or detailed bits of information about the device, such as serial numbers, name, and manufacturer of the device, among other things. This functions quite similarly to the MAXreg_rd, except the buffer is extended to nbytes+1, and the multiple-byte reads are transferred to the corresponding bytes of the buffer. The number of bytes to be transmitted is also updated in XSpi_Transfer().

## 4.INQ Question

<u>*Microblaze Preset:*</u> Microcontroller. There are quite a few provided, but the microcontroller preset is well-suited for embedded systems because of lower resources and power usage. The others are microprocessors and application processors. Microprocessors are more suited for general computing, and application processors run an OS as part of a full system-on-chip.

<u>*Bus connections:*</u> In this lab, we use a modified Harvard machine.
The major difference between a Harvard and Von-Neumann machine is that a Von-Neumann machine does not separate data and instructions, and both are stored in the same block of memory. A Harvard model completely divides the data and instructions into two sets of memory. In this lab, Microblaze has two different datapaths for the data and the instructions but they can be stored in the same on-chip memory.

<u>*What does the "asynchronous" in UART refer to regarding the data transmission method? What are some advantages and disadvantages of an asynchronous protocol vs. a synchronous protocol?*</u>

Asynchronous refers to the lack of the clock pin used in UART protocol. Instead, BAUD rates must be matched within a certain range and the transmission of data is synchronized by both the BAUD rates and start and stop bits. USART (the synchronous version) has a clock line to synchronize the two. Hence, it is asynchronous because there is no clock pin. Asynchronous means you don't need a clock line and can synchronize (assuming matched BAUD rates) with just start and stop bits to define the beginning and end of data transmissions. However, it is also low speed and the size of the data you can transmit is limited. USART has the benefits of higher bandwidth and being faster, though needs a clock line.

*You should have learned about interrupts in ECE 220, and it is obvious why interrupts are useful for inputs. However, even devices which transmit data benefit from interrupts; explain why.*
Data-transmitting devices can benefit from interrupts by using interrupts to signal the CPU when it is ready to be polled or has new information or something of the like. Otherwise, the CPU would be constantly spending time checking for new data when nothing is changed. This frees up the CPU to do other tasks and comes back when the interrupt is triggered to get the new data. By doing this, the system overall can run faster because the CPU is no longer wasting time checking something when it does not need to.

*Why are the UART and LED peripherals only connected to the data bus?*
UART (Universal Asynchronous Receiver/Transmitter): UART is a peripheral used for serial communication, which primarily involves the exchange of data and print message in serial terminal. It doesn't execute instructions; its primary purpose is to receive and transmit data. Therefore, it makes sense to connect it to the data bus, as data is what it processes.
LED (Light Emitting Diode): LEDs are used for visual feedback and don't execute instructions. They are controlled by sending data rather than executing instructions. Hence, connecting them to the data bus is sufficient.

*You must be able to explain what the volatile keyword does (line 18), and how the set and clear functions work by working out an example on paper (lines 30 and 33).*
When used with the C and C++ programming languages, the volatile keyword is used to indicate that a variable may be changed at any time by an external source (e.g., a hardware peripheral), and therefore the compiler should not optimize or cache access to this variable. In this code, it serves to tell the compiler that the led_gpio_data pointer may be changed by external register update. Without the volatile keyword, the compiler may optimize, or cache reads and writes to led_gpio_data, which may result in incorrect behavior in the event of hardware changes.
Function:
The function sets the LED by using logic expression LED OR 1. It ORs a binary 1

with itself, turning on the LSB of LED. This is reflected in the board proper by the rightmost LED turning on and off.
Ex. Start with LED = 0x00000000. LED | 0x00000001 = 0x00000001. Since the operator |= means to OR the left side with the value to the right of the operator, and store it in the left side, this means that LED gets changed to 0x00000001.
Clear function is implemented by the logic expression LED AND 0. This ANDs itself with 0, resetting the entire LED bank to 0 regardless of what value it was before. This is accomplished with the &= operator, which functions similarly to the |= operator specified above except with an AND instead of OR operation.
Ex. Start with LED = 0x00000001. LED & 0x00000000 = 0x00000000, which is stored back in LED and functionally turns off the LED that we see.

*Look at the various segments (text, data, bss), what does each segment mean? What kind of code elements are stored in each segment? Also note the size of the executable in bytes. Remember that we configured 32Kbytes of on-chip memory to use as our program memory. Why does the provided code, which does very little, take up so much program memory? Hint: try commenting out some lines of code and see how the size changes.*

Text Segment
A text segment, also known as a code segment, is the memory section where executable instructions live. It includes the actual code that the program executes.
It is usually read-only and contains the program's instructions. The size of the text segment depends on the number of instructions in your program and the size of any constants or literals. In the lab since we have included many headers file, it will become very large.

Data Segment:
This segment contains initialized data and global variables. It includes variables that have initial values assigned in the source code. The size of the data segment depends on the size and number of global variables and initialized data structures. A data segment (.data) starts right after .text.
BSS Segment
This segment contains uninitialized data and global variables. Variables declared in the code without an initial value are placed in this segment. Uninitialized data segment or bss segment, named after an ancient assembler operator, that stood for “block started by symbol.” This segment starts at the end of the data segment and contains all global and static variables that do not have explicit initialization. bss also ends up in RAM. Let’s add another global variable without initialization:

Though the provided code is little, we have included many headers file which will take up so much program memory. These header files import various C standard libraries, which explains the space they take up.

*Make sure you understand the register map on page 10. If the base address is 0x40000000, how would you access GPIO2_DATA (for example?).*

GPIO2_DATA: 0x400000008 offset address: 0x0008

## Design Resource Usage Statistics

### Lab 6.1

| | |
|---|---|
| LUT | 1792 |
| DSP | 3 |
| Memory (BRAM) | 8 |
| Flip-Flop | 2319 |
| Latches | 0 |
| Frequency | 123.107MHz |
| Static Power | 0.071W |
| Dynamic Power | 0.128W |
| Total Power | 0.198W |

### Lab 6.2

| | |
|---|---|
| LUT | 2776 |
| DSP | 9 |
| Memory (BRAM) | 8 |
| Flip-Flop | 2605 |
| Latches | 0 |
| Frequency | 53.242MHz |
| Static Power | 0.075W |
| Dynamic Power | 0.396W |

|  |  |
|---|---|
| Total Power | 0.471W |

## 5. Conclusion

This lab is a good practice to combine hardware and software. In the first week, we have to consider hardware logic into software programming. I make a mistake when I want to detect when the run button is released. I fixed the bug by adding another loop to detect the signal. In the second week, we try to program USB drivers. It is good practice to look through many provided code and figure which is useful. I can get correct revision read at first. Then I found I used the wrong function to write select register. Besides, I also found a bug when the ball is at the edge of the wall. If we press the right button at right corner, part of the ball will go into the wall. I fixed the bug by changing the order of detecting bounce and keyboard. I check keyboard signal first then for bouncing condition.

Overall, the lab helps us get familiar with microblaze and other IP modules. We know how to implement USB drivers. It will be helpful in the final project.