Lab Report #6

Aryan Shah (aryans5), Dev Patel (devdp2)

University of Illinois at Urbana Champaign

ECE 385 - Digital Systems Laboratory

Professor Zuofu Cheng, T.A. Gene Lee

October 30, 2023

Introduction

Summarize the basic functionality of the Microblaze processor running on the Spartan 7 FPGA

The Spartan 7 is an FPGA and Xilinx's 7 Series that allows for a wide-range of prototyping applications, allowing for designing and testing digital circuits while optimizing cost, power efficiency, and performance. The Microblaze processor is classified as soft-core, which means that it can be configured onto our FPGA without the need for a fixed architecture. We were able to utilize C code to program the necessary software so that the processor would run our Lab 6's tasks on the FPGA. We had to instantiate certain components to allow for proper SPI communication between the FPGA and processor through our block design, creating peripherals such as an AXI timer, GPIO ports, and UARTs. We further instantiated a PIO (Parallel Input/Output) block to facilitate parallel data transfer between the processor and our external FPGA components (display (HDMI), communication interfaces (USB), etc).

Briefly summarize the operation of the overall Week 2 design

The Week 2 design involved the integration of the MAX3421E, a USB host controller, which allows for the Microblaze to enable SPI communication protocol. The USB (Universal Serial Bus) is a communication protocol that allows for data transfer via its D- and D+ that hold opposing voltage values during a single time frame, representing 1 bit of data. The MAX3421E constantly sends interrupt signals to our keyboard, which responds with key press information (our up, down, left, right) in the form of report descriptors. Our design included a USB/VGA-HDMI IP interface which allowed us to see our keypresses in real-time using an external monitor connected via HDMI to our FPGA. The VGA consists of horizontal and vertical

syncs outputting analog RGB signals to a display, but we instantiated a VGA-HDMI IP to consider the fact that most current monitors utilize an HDMI interface.

Written Description and Diagrams of Microblaze System

Module Descriptions

Module: mb_usb_hdmi_top.sv

Inputs: Clk, reset_rtl_0, gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd, reset_ah

Outputs: 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, hdmi_tmds_data_n, hdmi_tmds_data_p, hex_segA, hex_gridA, hex_segB, hex_gridB

Description: This module is the top-level wrapper for our week 2 design. It interfaces our hex drivers, the components of our block design (such as CLK, SPI, GPIOs, UARTs), our clocking wizard, VGA Sync signal generator, VGA to HDMI Converter, ball module, and color mapper module.

Purpose: Top-level module for our week 2 design, instantiating necessary components, defining signal logic, as well as inputs and outputs.

Module: mb_intro_top.sv

Inputs: clk, btn, uart_txd

Outputs: led, uart_rxd

Description: This module is the top-level wrapper for our week 1 design. It interfaces our submodule, mb_block, and associated peripherals that we designed for the blinking operation.

Purpose: Top-level module for our week 1 design, instantiating necessary components, defining signal logic, as well as inputs and outputs.

Module: hex.sv

Inputs: Clk, reset, in[4] (4-bit nibble to convert to hex display)

Outputs: hex_seg, hex_grid

Description: This module takes in four 4-bit values, converts them to hex (using a nibble-to-hex submodule), and creates a counter to cycle through the digits to display them onto our 7-segment FPGA display.

Purpose: To be used for our A and B hex drivers, allows for display of proper values on our hex displays.

Module: VGA_controller.sv

Inputs: pixel_clk, reset

Outputs: hs, vs, active_nblank, sync, drawX, drawY

Description: This module generated VGA timing signals. It creates horizontal and vertical sync (hs and vs) pulses that are then used to translate into RGB display values. The code allows for the disabling of the sync signal such that a display does not go outside the borders of the display and resets beforehand The code also allows for proper timing and control of such that a VGA display can output the necessary information in a 640x480 resolution.

Purpose: Allows for the generation of a 640x480 resolution display on an FPGA monitor. Created the logic for sync pulses to translate into RGB values and also handles the logic for when pixels are defined outside the display's borders.

Module: ball.sv

Inputs: Reset, frame_clk, keycode

Outputs: BallX, BallY, BallS

Description: This module defines the movements of our ball's motion on the display. The ball is updated based on the screen boundaries and if a boundary is hit, it bounces off the edge. We further created logic to control movements (up, down, left, right) based on keycodes (corresponding keyboard arrows).
Purpose: With this module, the ball is able to move based on the keypresses initiated by the user onto the keyboard accordingly. If the ball hits the edge, it will bounce back rather than exit out of the frame of view.

Module: Color_Mapper.sv

Inputs: BallX, BallY, DrawX, DrawY, Ball_size

Outputs: red, green, blue

Description: This module calculates the squared distance between the current pixel using the DrawX and DrawY values, against its center using the BallX and BallY values. If the squared distance is less than or equal to its squared radius, the pixel is therefore in the ball. If the pixel is in the ball, we color the ball with its necessary values. If it is outside of the ball, then the RGB colors are instantiated as a gray gradient with intensity decreasing as DrawX increases.
Purpose: Its purpose is to map our display's pixel colors on the screen based on determining whether the pixel is inside or outside the ball and calculating its distance from DrawX.

Describe every component in the block design

AXI Uartlite - This is a simple UART (Universal Asynchronous Receiver-Transmitter) IP core, allowing for serial communication over our AXI Interface. It allows for the configuration of a baud rate which corresponds to the rate at which data is transferred using the UART protocol. FIFO buffers manage data flow (prevent data loss or

overwriting) and interrupt handling to provide better synchronization of data.

Clocking Wizard - This IP core allows for one or more clock signals to be linked to one common clock signal with shared frequencies and phases. Clock gating is a feature that can enable or disable clocks to other specific block modules. The core also handles reset management and synchronization such that the clocks are shared across one domain.

AXI GPIO (gpio_usb_keycode) - The AXI GPIO (general-purpose input/output) core allows for communication between USB keyboard inputs or keycode information with our peripherals. In the case of gpio_usb_keycode, it means that the USB keyboard, when pressed, sends keycode information that is then received by the GPIO and passed on for processing by the FPGA (detecting key presses and releases).

AXI GPIO (gpio_usb_int) - This is another GPIO core that allows for handling of interrupt signals by our USB device. The USB device generates an interrupt request when data is necessary and this core then detects this and triggers responses from our FPGA.

AXI GPIO (gpio_usb_rst) - This is another GPIO core that allows for handling of interrupt signals by our USB device. The USB device generates an reset request when data has been provided and this core then detects this and triggers responses from our FPGA.

AXI Interrupt Controller - This IP Core manages our interrupt signals that are driven from other IP Cores. This controller allows different block modules to generate interrupt requests and distributes it to the appropriate handlers. It can allow for interrupt prioritization as well, such that different interrupt sources are assigned a higher or lower priority level to ensure more important interrupt signals are taken care of. Lastly, another feature is that it can send out acknowledgement signals so that repeated interrupt signals

are not being sent.

AXI Interconnect - The AXI Interconnect allows for the facilitation of data transfer and communication. It follows the AXI protocol and includes features such as bus arbitration (multiple masters can access shared resources), address decoding to route data and control signals, match data widths across connected components, and has multiple ports so that different masters and slaves can be communicated between each other.

Microblaze_0_local_memory - The local memory IP core stores frequently accessed data or instructions in our on-chip memory. Since it is frequently accessed, there does not need to be a large delay in data retrieval when it is stored in this block. The memory is separated into separate data and instruction buses, but pulling in from the same memory as we follow a modified Harvard memory architecture.

Processor System Reset - This module allows for a total reset protocol for our system. When a signal, typically some sort of hardware “shut-off”, is asserted, the system returns to a full reset state.

AXI Quad SPI - This module allows the FPGA to interact with other SPI devices and peripherals. It allows for bidirectional data transfer, read/write operations, “quad” data lines (MISO0, MISO1, MISO2, MISO3) for data reading which is a higher-performance aspect than a traditional SPI. It includes a memory-mapped interface and allows for configuration of certain aspects such as clock rates and data widths.

Concat - The concat module allows for individual data buses or vectors to be concatenated to form an output signal. In the case of our lab, this includes the interrupt signals sent out from various IP cores, which are then routed to our AXI interrupt controller for further processing.

AXI Timer - The AXI timer allows for accurate timing for our events (interrupts, resets, data transfers, etc.) within certain time intervals. Multiple timer channels is a feature so that channels can operate independently if needed. One purpose of this would be to generate interrupts at specific time intervals.

Microblaze - The Microblaze is a soft-core processor that allows for custom programmed operations in C to be configured onto our FPGA.

Microblaze Debug Processor - The debug processor allows for tools such as Xilinix's IDE or other debuggers such as GDB to allow us to observe and modify the behavior of a program running in real-time, with the purpose of fixing software issues. Features include stepping line by line, setting watchpoints or breakpoints, halting program execution, so we can make sure the program is running as intended or if it needs modifications as necessary.

Describe in Lab 6.1 how the I/O works

The Week 1 portion of our lab involved the creation of an accumulator that is managed by reset and accumulate buttons. Switches are specified such that the FPGA knows what value is to be added on to. After pressing the run button, our C program from the Microblaze processor reads values from the switches and adds them to the stored value in our accumulator. The reset button simply sets the stored value back to 0. A PIO block allows the values in the accumulator to be displayed onto our LEDs.

Describe in words how the MicroBlaze interacts with both the MAX3421E USB chip and the ball motion components

The Microblaze is responsible for allowing for the motion of the ball to output onto the screen. Using input signals based on the ball's position, the Microblaze outputs signals to the display to update the ball's position and motion on the screen. It also receives input from the user via a keyboard to control the ball's movements, sending in keycode signals and receiving the correct response back to output onto the screen. The MAX3421E chip manages the USB portion of the communication, allowing for the Microblaze to send and receive data via the MAX3421E. When data transfer is initiated, necessary read and write buffers of the MAX3421E allow for the data to be usable by the processor and therefore output what is necessary to the display.

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

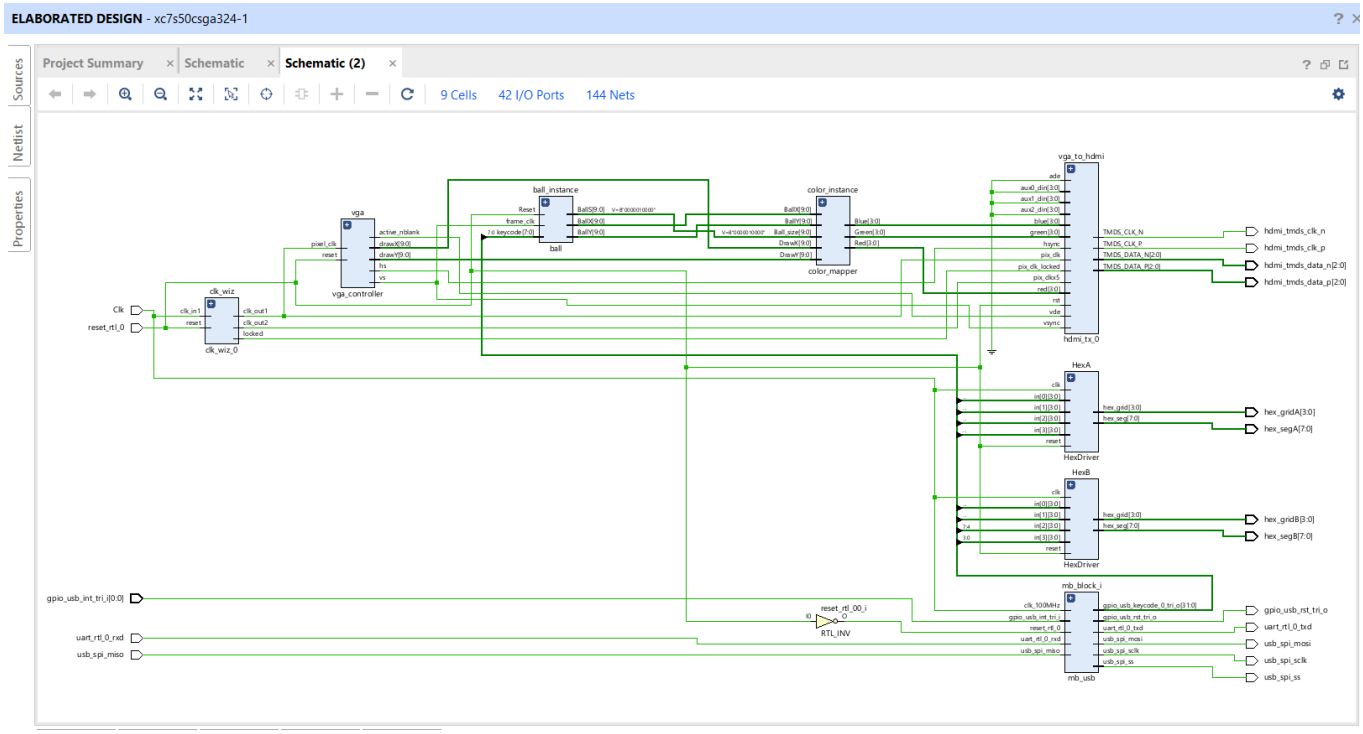
The VGA module instantiates the timing signals, determines the coordinates of our pixels, and allows it to be displayed on a device in the 640x480 resolution. A display enable signal, the active_nblank, is also used to determine what color should be used for each pixel (based on if it is within the ball, or outside the ball and how far it is away from the DrawX variable). The color mapper module has the pixel coordinate data (DrawX and DrawY) from the VGA. It determines whether the output pixel should be orange if it is within the regions of the ball, or what gradient level of gray based on its position from DrawX. The ball module controls the motion of our ball on the screen. It provides the VGA with the X and Y coordinates of our ball's position. It ensures that our ball cannot move outside the boundaries of the display, rather bounces off. It also takes in keycode values and determines which X and Y coordinate to move (up +1Y, down -1Y, right

+1X, left -1X). They are all essential for both controlling the ball, its movements, and what is outputted onto our display.

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

The VGA-HDMI IP was an IP core that was utilized to translate the VGA signals into HDMI video signals, because it is what is more commonly used in today's display monitors. It includes features such as video signal processing, format conversion, and encoding/decoding to allow for this transition across VGA and HDMI signals. HDMI, which is more commonly used, can support higher resolution signals than VGA. Displays using HDMI today can be seen to have quality up to 4K and higher, as compared to a maximum of 640x480 pixels on the VGA. The VGA uses analog signals to transmit its video data, which can have interference at longer cable lengths. HDMI, instead, utilizes digital signals which are less susceptible to signal interference and also can translate both audio and video in a single cable. VGA, on the other hand, does not typically support audio transfer. When it comes to their similarities, both are used for display onto external devices (such as monitors, TVs, etc)

Top Level Block Diagram



Describe in words the software component of the lab (you must also describe your accumulator)

In week 1, we designed the accumulator portion of the lab. We included a main function with an infinite while loop so that the accumulator constantly checks for input signal (by pressing the accumulate button) to add its existing value to the value of the switches. If a sum overflows, we read out a value to the console and reset the accumulator back to 0. The LED[0] is set to high to indicate an overflow.

For week 2, we displayed a gray gradient on a screen with a ball that would move based on keypresses on a keyboard, outputting it onto an HDMI display. For this, we had to fill in several given functions in our MAX3421E.c file. Our first MAXreg_wr function writes a single bit of data into a register. Our second MAXbytes_wr function writes multiple bytes into a register. Our

MAXreg_rd function reads a single bit of data from a register. Our MAXbytes_rd function reads multiple bytes of data from a register.

Written description of the SPI Protocol and how it operates in the context of the MAX3421E

The SPI protocol is a communication method that allows for data exchange between a master and one or more slaves. It utilizes a serial clock (clock signal to synchronize data transfer), MOSI (master out slave in, master data output used to send data to the slave), MISO (master in slave out, slave data output used to send data to a master), slave select (indicate which slave device), and chip select (enable or disable transfer). The MAX3421E needs to be instantiated using the XSpi_SetSlaveSelect function with an instance pointer (base address which is always &SpiInstance in our case) and offset (1 to turn it on, 0 to turn it off). A master device asserts the slave select and chip select values for the MAX3421E, indicating a transfer is requested. Data is then transmitted across the MOSI line while the MAX3421E reads data from the MISO line. The SCLK ensures that this transfer is synchronized between the master and slave. The MOSI data includes commands to control the USB operations in our code. We must deactivate the MAX3421E as well at the end of each program.

Describe the purpose of each function you filled in the C Code

Main.c - In lab 6.1, we imported the “main.c” file from the given file list. In this code, we included a non-terminating while loop that enables the accumulator to run continuously, checking constantly for the accumulator button to send a signal by being pressed. If the accumulate button was pressed, a variable detects an active high and thus adds the value of the switches to the previous sum added. If the sum overflows, then our accumulator resets.

void MAXreg_wr (BYTE reg, BYTE val) - This function writes data into a register. We first

select the MAX3421E chip using the XSpi_SetSlaveSelect function. We then create our write buffer array with the values reg+2 and the passed val. We set a return variable to be the value of XSpi_Transfer function with our writebuffer array, read set to NULL, and byte size of 2. If the return value is 0, it indicates a failure and we print an error. We then call slave select with a value of 0 to deselect the MAX3421E.

BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) - We select the MAX3421E chip using the same method as above. We initialize an array of length 2 and then instantiate the first element to be reg+2. We loop through the amount of nbytes, setting the first element of our array at the corresponding data index of the loop iteration we are one. We then set a return variable to be the value of our SPI transfer function with the write set to our array, read set to NULL, and byte size set to 2. If our return value is not 0, we print out an error. After error is checked, we deselect the MAX3421E chip and then return data+nbytes.

BYTE MAXreg_rd(BYTE reg) - We first select our MAX3421E chip. We instantiate separate read and write buffers using arrays of length 2. We set the first element of our write buffer to be reg and the second element to be 0. We set a return value to be the output of our SPI Transfer function using our write and read buffer arrays, and a byte size of 2. If the return value is unsuccessful, we output an error. We deselect our MAX3421E chip and finally return the second element of our read buffer.

BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) - We select the MAX3421E chip. We then instantiate separate arrays for our read and write buffer, both of length 2. We set the first element of our write and read buffer arrays both to reg. We set the second element of our write buffer to be 0. We loop through the amount of nbytes, and inside the loop we set a return variable value to be the output of our transfer function with our write and read buffers, and a byte size of

2. Inside the loop, as well, we instantiate the data value at the index of our loop counter to be the value of the second element of our read buffer. If our return value is not 0, we output an error (also under the for loop). Outside the for loop, we deselect the MAX3421E chip using XSpi_SetSlaveSelect(&SpiInstance, 0). Finally, we return data+nbytes.

Answers to all INMB (italicized) questions

1) **You should do some research and figure out what are some primary differences between the various presets which are available**

Preset:
Microcontroller - Microcontroller preset suitable for microcontroller designs. Area optimized, with no caches and debug enabled.
Real-time - Real-time preset geared towards real-time control. Performance optimized, small caches and debug enabled, most execution units.
Application - Application preset design for high-performance applications. Performance optimized, large caches and debug enabled, and all execution units including floating-point.

2) **Note the bus connections coming from the MicroBlaze; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?**

A Von Neumann contains a single bus based architecture for both data and instructions. A pure Harvard design contains separate buses for data and instructions. A modified Harvard architecture has separate buses but the memory for data and instructions are shared. In our case, the Microblaze is implemented with a modified Harvard memory architecture; instruction and data accesses are done in separate address spaces but can overlap, which means that the processor maps both instruction and data memories to the same physical memory space.

3) **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?**

Asynchronous is the method of data transmission where in the UART, data is transmitted without a shared clock signal between the sender (transmitter) and receiver (receiver).
Steps:
Start bit for transmission of data is logical low
Actual data bits follow start bit
After data bits, one or more stop bits are transmitted at a logical high to indicate end of data byte
On the other hand, in synchronous communication, both the sender and receiver rely on a common clock signal to synchronize the transmission and reception of data.

Asynchronous:
Advantages - Flexible for devices with different clock rates, simpler to implement, fewer bits transmitted (less overhead)
Disadvantages - lower data throughput, more susceptible to errors from noise or interference
Synchronous:
Advantages - higher data throughput, lower error rates, better for continuous data streams without interruption of start/stop bits
Disadvantages - devices need to be synchronized on a shared clock, more complex implementation

4) **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. Hint: the UART takes a long time (relative to the CPU) to transmit a single byte.**

Even devices which transmit data benefit from interrupts because it allows for efficient handling of data transmission (no continuous polling, triggering of interrupts to notify the system for transmission), allow for asynchronous operation (device can initiate a transfer at any time), allows for multitasking, and reduced CPU utilization without constant polling.

5) **You must be able to explain what each line of this (very short) program does to your TA. Specifically, 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).**

The first couple #include commands is a preprocessor directive allowing us to include contents of other header files into the mb_blink.c file. We reuse some instantiated global variables to make our code simpler.

Volatile signals to the C compiler that a variable's value may change unexpectedly, and thus should not optimize or assume the variable's value. It always reads from memory and is useful for memory-mapped I/O and multithreaded programs.

The line 18 itself declares a pointer to a 32-bit unsigned integer (variable name 'led_gpio_data) with memory address 0x40000000 (since it is volatile it may change unexpectedly so the compiler does not optimize its value).

The main() program includes a loop with the intention to toggle the LED on and off with a 1 second delay. The program initializes its configurations with init_platform, enters an infinite loop (1+1 != 3 always true), sets the least significant bit of the value pointed to by 'led_gpio_data' to 1 (turns on LED), prints to the console "Led On!", sleep(1); pauses the program for 1 second, clears the least significant bit of the value pointed to 'led_gpio_data' using *led_gpio_data &= ~0x00000001 (turns off LED), prints to the console "Led Off!", continues looping and then technically cleans up resources in the platform with cleanup_platform (never reached), returns 0 to indicate program termination in C guidelines (never reached).

6) **Look at the various segments (text, data, bss), what does each segment mean? What kind of code elements are stored in each segment?**

   Text - Contains executable code of the program. Represents the size of the program's instructions or code stored in non-modifiable memory.
   Data - initialized global or static variables. Includes variables that have an initial value specified in the source code.
   Bss (block started by symbol) - represents uninitialized or zero-initialized variables. Also includes global or static variables declared without an initial value.
   Dec (decimal) - shows the size of each respective section in decimal (base 10) representation
   Hex (hexadecimal) shows the size of each respective section in hexadecimal (base 16) representation
   Filename - name of the file being analyzed "mb_blink.elf"

7) **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.**

There are two reasons that the code can take up so much program memory. The first of which (also the most likely) comes from our header files. Each included file bring additional code and variable declarations which increase the size of our program in memory even though mb_blink.c looks so small. The second reason is that the printf() function has a large implementation, which can contribute to the code taking up larger memory. By using xil_printf() we can use less memory.

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

You can access it using the code line: volatile uint32_t* led_gpio_data = 0x40000000;
Creating a pointer to point to the value at the 0x40000000 base address and then initializing it as an unsigned 32-bit integer, indicating that it is volatile (can be changed and therefore compiler should not optimize). GPIO2_DATA would point to 0x40002000, as the second GPIO block would have a base ADDR offset by FFFF (for both GPIO_1 and GPIO_0), which would account for the 32 bit maximum width of the GPIO module.

Answers to all Post-lab questions

None

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

LUT	2721
DSP	9

Memory(BRAM)	8
Flip-Flop	2560
Latches	0
Frequency	123.076923 MHz
Static Power	0.075 W
Dynamic Power	0.383 W
Total Power	0.458 W

Conclusion

Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it?

During the course of our design, we had several minor issues. The first problem began when trying to merge our two block designs from Week 1 to Week 2. We noticed that instead of starting the project from its entirety we could simply copy the block design using Ctrl A and Ctrl V and then disabling all the project Week 1 sources. Another issue encountered was in Week 2, when we utilized a keyboard that we did not know would not properly work with our lab. We had working code for a couple of hours and it wasn't until a decision was made to test the ECEB equipment that we figured out our design worked, but just was not working on that specific keyboard.

Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester?

N/A