ECE 385

Spring 2024

Experiment 6

SOC with Microblaze in SystemVerilog

By Wendi Wang (wendiw2)

1. Introduction

a) Basic Functionality of the Microblaze Processor on FPGA

The Microblaze is an IP based 32-bit CPU which can be programmed using a high-level language (in Lab 6, it is the C language). In Lab 6, the Microblaze is the system controller and handles 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 (designed using SystemVerilog) handles the high-performance operations. This is the idea behind System-on-Chip concept.

The Microblaze CPU handles the interaction between a list of memory mapped I/O (shown as GPIO IPs in the block design) and the inputs/outputs on FPGA board. The FPGA board controls the status of the input GPIO port. When a change in input status is detected (e.g., a switch is toggled), the FPGA board processes this change and updates the output GPIO port accordingly to reflect the desired behavior on the output peripherals (e.g., lighting up an LED).

b) The Operation of Week 2 Design

In Lab 6.2, we augmented the Week 1 lab with the addition of a USB host controller (MAX3421E) which communicates to the Microblaze via the SPI protocol. In detail, we instantiated some modules to make use of the Urbana board's USB port, enumerated a USB keyboard, and used it to control a ball which is drawn using VGA. We also used a provided IP block to convert the VGA signal to an HDMI signal for displaying the ball on an external HDMI monitor. Finally, we can use 4 keys on the USB keyboard to control the motion of the ball among 4 directions (up, down, left, right but not diagonally). Since we have taken boundary conditions into consideration, the ball will bounce back if it hits the boundary.

2. Written Description and Diagrams of Microblaze System

a. Module Descriptions

a) With postfix .sv files:

Module: mb_usb_hdmi_top.sv (Lab 6.1)

Inputs: clk, [3:0] btn, [15:0] sw, uart_rtl_rxd

Outputs: [15:0] led, uart_rxd

Description: This file contains a top-level module connecting the physical inputs and outputs of the FPGA board (such as buttons, switches, LEDs, and UART pins) to the respective ports of the MicroBlaze block design. It also handles the inversion of the reset signal to match the active high reset requirement of the MicroBlaze. Additionally, it manages the UART communication between the FPGA and the Urbana Board's UART chip, ensuring proper transmission and reception of data.

Purpose: Connecting the physical inputs and outputs of the FPGA board to the respective ports of the MicroBlaze block design.

Module: mb_usb_hdmi_top.sv (Lab 6.2)
Inputs: Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd
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, [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: This file contains a top-level module designed for integrating various peripherals including USB, UART, HDMI, and VGA. It can generate several kinds of clock signals for VGA drawing as well as VGA to HDMI signal conversion. It can also serve as a bridge between ball motion and USB keycode control. This module interfaces with multiple blocks including USB, UART, VGA, HDMI, hex displays, a ball module, and a color mapper.
Purpose: Integrating USB, UART, HDMI, VGA, and additional functionalities within a single top-level file (module).

Module: hex_driver.sv (Lab 6.2)
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [7:0] hex_grid
Description: This file contains a module of visualizing outputs. To be specific, it contains a module of demonstrating output values on LEDs using 7-segment method. An LED digit on the Urbana Board is divided into 7 segments, with different segment combinations we can express numbers from 0 to F in hexadecimal.
Purpose: To translate hex numbers (from 0 to F) into figures appearing on LEDs.

Module: VGA_controller.sv (Lab 6.2)
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: This file contains a module of generating VGA video signals, which requires synchronizing the horizontal and vertical timing of the display as well as managing the drawing coordinates within the screen resolution. It finally outputs several necessary signals for activating the display pixels including x and y pixel coordinates, horizontal and vertical drawing frequencies, etc. It contains internal counters for horizontal and vertical positions to provide proper timing for horizontal and vertical sync pulses. Besides, it also takes reset conditions and display range into consideration.
Purpose: Generating VGA related signals and pixel coordinates for further drawing

as well as displaying on the screen after converting to HDMI signals.

Module: ball.sv (Lab 6.2)
Inputs: frame_clk, Reset, [7:0] keycode
Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS
Description: This file contains a module to simulate the movement of a ball. It receives frame_clk signal from VGA controller as an input signal, which corresponds to vertical update frequency of VGA. Besides, it also contains some logics of moving the ball according to specific keycode received from USB keyboard. To avoid the ball going outside the screen, some limitations and bouncing-back logics are supported in this module. Finally, this module outputs ball position coordinates as well as ball size to Color_mapper module for further drawing.
Purpose: Simulating the ball's motion and position based on input keycode and frame clock signal.

Module: Color_mapper.sv (Lab 6.2)
Inputs: [9:0] BallX, [9:0] BallY, [9:0] BallS, [9:0] drawX, [9:0] drawY
Outputs: [3:0] Red, [3:0] Green, [3:0] Blue
Description: This file contains of module of determining the color to display at each pixel on the screen based on the relative position of the ball and the current drawing coordinates. It also uses a circular boundary check to determine if the pixel falls within the ball's area or falls within the background area to determine what RGB values should be drawn.
Purpose: Mapping locations (positions) to colors drawn represented by RGB values.

b) Components inside the Block Design

mb_block Module:
This module is likely the MicroBlaze block, responsible for implementing the MicroBlaze processor in the FPGA design.
Interface signals:
clk_100MHz: Clock signal at 100 MHz provided to the MicroBlaze block.
gpio_usb_int_tri_i: Tri-state input signal for USB interrupt from the USB controller.
gpio_usb_keycode_0_tri_o, gpio_usb_keycode_1_tri_o: Tri-state output signals for keyboard scan codes received from the USB controller.
gpio_usb_rst_tri_o: Tri-state output signal for USB reset control.
reset_rtl_0: Active low reset signal for the MicroBlaze block.

uart_rtl_0_rxd, uart_rtl_0_txd: UART receive and transmit signals.
usb_spi_miso, usb_spi_mosi, usb_spi_sclk, usb_spi_ss: SPI interface signals for communication with the USB controller.

clk_wiz_0 Module:
This module is likely a clock wizard or clock generator block, responsible for generating clock signals with specific frequencies required by the HDMI interface.
Interface signals:
clk_out1: Output clock at 25 MHz, used for VGA timing.
clk_out2: Output clock at 125 MHz, used for HDMI timing.
reset: Active high reset signal for the clock wizard.
locked: Output signal indicating whether the generated clocks are locked and stable.
clk_in1: Input clock signal provided as the reference clock input to the clock wizard.

hdmi_tx_0 Module:
This module is likely the HDMI transmitter block, responsible for converting VGA signals to HDMI-compatible signals for display on an HDMI monitor.
Interface signals:
Various clocking and reset signals for synchronization.
Color and synchronization signals (red, green, blue, hsync, vsync, vde) for HDMI output.
Differential outputs (TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N) for HDMI TMDS signaling.

AXI GPIO:
It is a fundamental interface commonly found in microcontrollers and FPGAs, allowing them to interact with the external world by controlling and monitoring digital signals. In this context, GPIO signals are often used to interface with various peripherals, such as buttons, switches, LEDs, and other devices.

AXI UART:
It is a commonly used serial communication interface found in microcontrollers, FPGAs, and other electronic devices. It enables asynchronous serial communication between devices, allowing them to exchange data over serial links.

AXI TIMER:

The AXI Timer module represents a programmable timer component designed to generate periodic or one-shot timing events in FPGA designs. AXI Timer modules are commonly used for tasks such as scheduling tasks, measuring time intervals, and triggering events at specific time intervals.

AXI Quad SPI:

The AXI Quad SPI module represents the Advanced eXtensible Interface (AXI) Quad Serial Peripheral Interface (SPI) controller, commonly used in FPGA designs to interface with SPI flash memories, sensors, and other SPI-compatible devices. The AXI Quad SPI controller provides a high-speed serial communication interface between the FPGA and external SPI devices, supporting quad data lines for increased data transfer rates.

AXI INTC:

The AXI INTC serves as the central hub for managing interrupts within the MicroBlaze system, facilitating communication between the processor and peripheral devices. It handles interrupt requests generated by various peripherals connected to the MicroBlaze processor, prioritizing and dispatching them to the processor for handling.

AXI Peripheral Interface:

This block serves as the bridge between the MicroBlaze processor and the AXI interconnect, facilitating communication between the processor and various peripheral devices in the system. The AXI Peripheral Interface block typically supports multiple AXI interfaces, allowing it to connect to multiple peripherals simultaneously.

b. I/O Workflow in Lab 6.1

In Lab 6.1, we used three GPIOs (General Purpose I/O) to communicate information between the input (switches and buttons on the board) and the output (LEDs on the board). All GPIOs are single-ported, and one GPIO is “all inputs” with width 16 (16 switches), the other “all inputs” GPIO is with width 1 (“Accumulate” button), and another one is the other one is “all outputs” with width 16 (16 LEDs).

We connected on-board switches’ ports, button ports, and on-board LEDs’ ports to corresponding inputs and outputs of the GPIOs. When a switch is toggled or pressed on the board, it changes the state of the corresponding input pin of the GPIO port. When we pressed “Accumulate” button, the accumulator will increment by the value on the switches. Based on the input signals received from the input GPIO port, the output GPIO port controls the state of the LEDs by turning them on or off accordingly.

The FPGA board controls the status of the input GPIO port. When a change in input status is detected (e.g., a switch is toggled), the FPGA board processes this change and updates the output GPIO port accordingly to reflect the desired behavior on the output peripherals (e.g., lighting up an LED).

c. Interaction of Microblaze with USB and Ball Motion

The MicroBlaze communicates with the MAX3421E USB chip using the SPI (Serial Peripheral Interface) protocol. SPI allows for serial communication between devices and is commonly used for interfacing with peripherals like the MAX3421E. The MicroBlaze sends commands to the MAX3421E to control its operation, such as initializing USB communication, sending, and receiving data packets, and handling USB device enumeration. The MAX3421E, in turn, communicates with the USB keyboard, constantly polling for key press information. When a key press is detected, the MAX3421E sends the corresponding scan code to the MicroBlaze. The MicroBlaze receives the scan code from the MAX3421E and processes it to determine the direction key pressed by the user (W, A, S, or D).

The MicroBlaze controls the motion of the ball displayed on the HDMI monitor screen. It receives input from the USB keyboard via the MAX3421E to determine the direction in which the ball should move. Initially, when the program starts, the MicroBlaze ensures that a stationary ball is displayed in the center of the screen. It continuously monitors for keyboard input from the user. Upon detecting a direction key press (W, A, S, or D), the MicroBlaze updates the motion parameters of the ball accordingly. For example, if the user presses the 'W' key for upward movement, the MicroBlaze adjusts the ball's vertical position to move it upwards.

The MicroBlaze also ensures that the ball follows the specified behavior, such as bouncing off the screen edges when it reaches them and changing direction immediately upon receiving a new key press without returning to the center of the screen.

To achieve motion in both the X (horizontal) and Y (vertical) directions, the MicroBlaze updates the position of the ball based on the user's input and ensures that the ball's motion is synchronized with the VGA controller's timing signals.

d. VGA Operation and Corresponding Interaction

The operation of VGA involves generating horizontal and vertical synchronization signals along with pixel color information to produce a video signal that can be displayed on a VGA monitor or compatible display device. The VGA controller module generates timing signals for horizontal sync (hs), vertical sync (vs), and active/negative blanking (vde). These signals define the timing and structure of the video signal.

The pixel_clk input provides the pixel clock signal, typically running at a frequency of 25 MHz, which synchronizes the timing of pixel data transmission.

The reset input resets the VGA controller.

The drawX and drawY outputs provide the current pixel position within the active display area. Then the HDMI converter module takes the VGA signal generated by the VGA controller and converts it into an HDMI-compatible signal. It receives the pixel clock (pix_clk), horizontal sync (hsync), vertical sync (vsync), and active/negative blanking (vde) signals from the VGA controller. It also receives the red, green, and blue color signals (red, green, blue) representing the pixel colors. These signals are then processed and converted into TMDS (Transition Minimized Differential Signaling) signals (TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N) suitable for transmission over an HDMI cable.

The Ball module handles the animation and movement of a ball on the VGA display. It receives the reset signal (reset_ah) for initialization. It receives the frame clock signal (vsync) from the VGA controller to synchronize its animation with the display frame rate. It also receives the keycode representing the direction of movement for the ball (keycode0_gpio[7:0]). The module calculates the position and size of the ball based on its internal logic and outputs signals (BallX, BallY, BallS) representing the ball's position and size.

The Color Mapper module maps the color of the ball onto the VGA display. It receives signals representing the ball's position (BallX, BallY) and size (Ball_size). It also receives signals representing the current pixel position on the display (DrawX, DrawY). Based on the ball's position and size, the Color Mapper module modifies the color signals (Red, Green, Blue) to display the ball at the correct location with the correct color.

Overall, the VGA controller generates timing signals, the HDMI converter converts VGA signals to HDMI, the Ball module handles ball animation, and the Color Mapper module maps the ball's color onto the display, working together to display the animated ball on the VGA monitor.

e. VGA-HDMI IP and Differences & Similarities between HDMI and VGA

The VGA-HDMI IP block takes in VGA signals generated by a VGA controller, which typically include horizontal sync, vertical sync, and pixel color information. It then processes these signals and outputs an HDMI-compatible signal containing the same visual information. This conversion allows VGA-based video sources to be displayed on HDMI-compatible displays.

Differences:

- ✧ VGA is an analog interface, transmitting video signals as analog electrical signals. HDMI, on the other hand, is a digital interface, transmitting video and audio signals as digital data.
- ✧ HDMI supports higher resolutions and better image quality compared to VGA. And HDMI can transmit both video and audio signals over a single cable, making it convenient for connecting devices such as Blu-ray players, gaming consoles, and

set-top boxes to TVs and monitors. VGA, on the other hand, only supports video transmission, requiring separate audio cables for audio output.

- ✧ HDMI connectors are smaller and more compact compared to VGA connectors, making HDMI cables easier to manage and connect. VGA connectors, by contrast, are larger and bulkier, and VGA cables do not feature a locking mechanism, which may result in less secure connections.

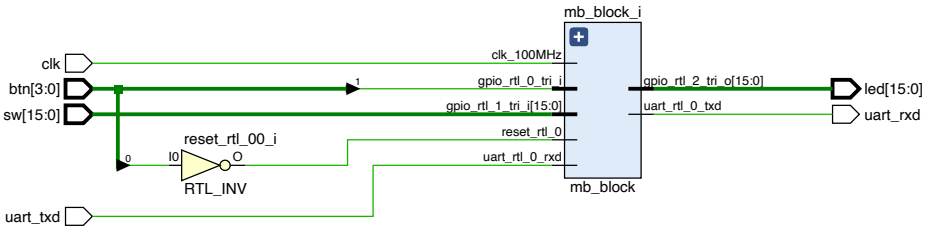
Similarities:

- ✧ Both HDMI and VGA are used to connect devices to displays such as monitors, TVs, and projectors. They serve as interfaces for transmitting video signals from the source device (such as a computer, DVD player, or gaming console) to the display device.
- ✧ Both HDMI and VGA offer compatibility with a range of devices, ensuring that users can connect their devices to a variety of displays.

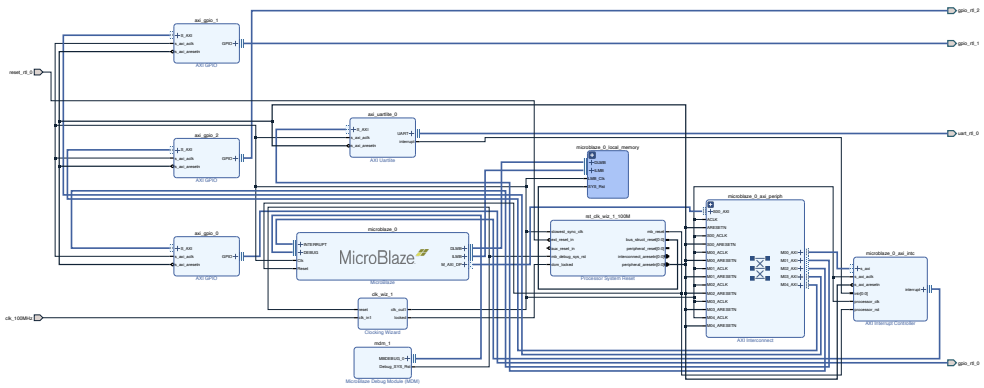
3. Top Level Block Diagram

a) Lab 6.1

i. RTL Schematic

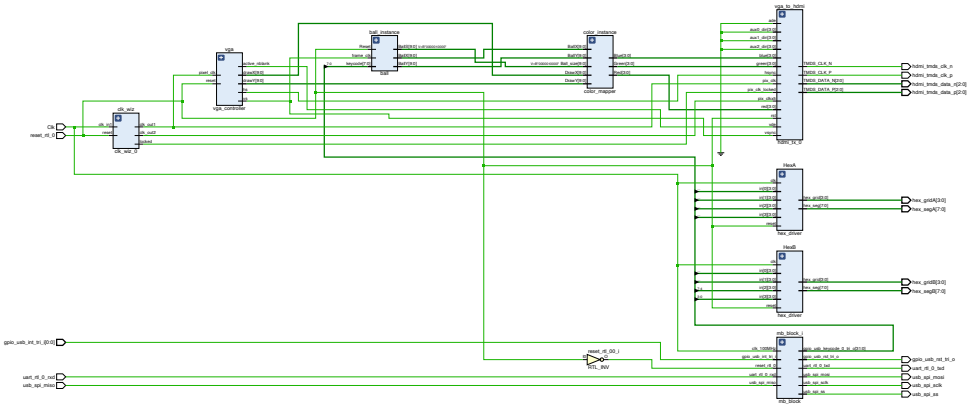


ii. The Microblaze Block Diagram

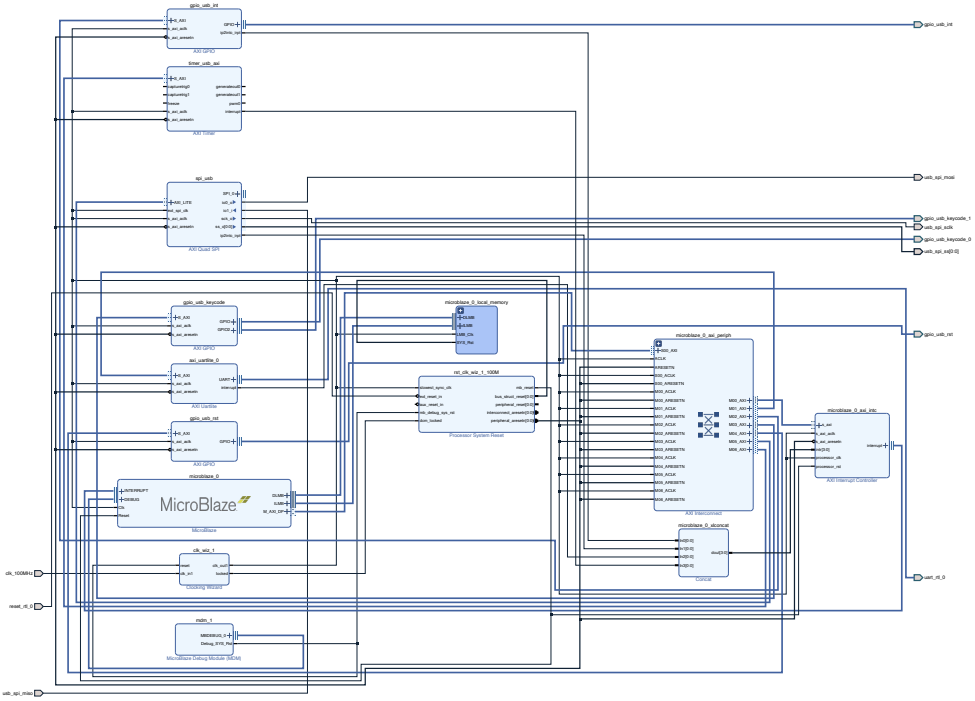


b) Lab 6.2

i. RTL Schematic



ii. The Microblaze Block Diagram



4. Description of Software Components

a. Written Description of the SPI Protocol and Its Operation

The Serial Peripheral Interface (SPI) protocol is a synchronous serial communication protocol used to transfer data between devices. It typically involves one master device controlling one or more slave devices. In Lab 6.2, SPI is used to communicate with the microcontroller or FPGA controlling the MAX3421E. Below is how SPI operates in this lab:

Master-Slave Configuration: The microcontroller or FPGA acts as the master device, while the MAX3421E serves as the slave device. The master device initiates communication by generating a clock signal and selecting the slave device with a chip select (CS) signal.

Clock Synchronization: SPI communication is synchronous, meaning both the master and

slave devices share a common clock signal. The master device generates the clock signal, and data is sampled on the rising or falling edge of the clock pulse, depending on the specific SPI mode configured.

Data Transfer: SPI transfers data in full-duplex mode, allowing simultaneous transmission and reception of data. The master device sends data to the slave device through a MOSI (Master Out Slave In) line, while the slave device sends data back to the master device through a MISO (Master In Slave Out) line.

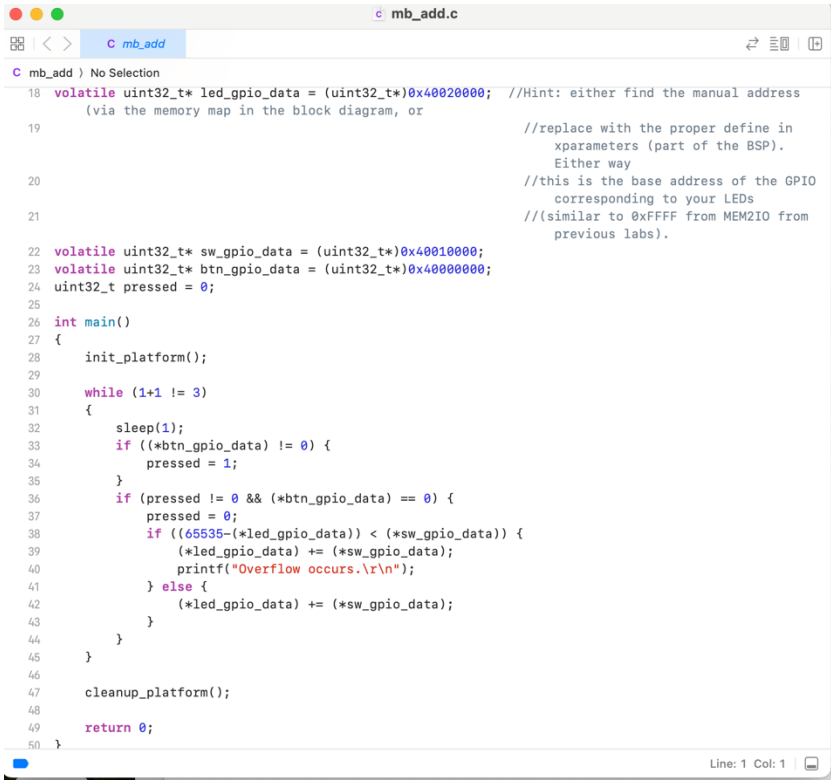
Serial Communication: Data is transferred serially, one bit at a time, over the MOSI and MISO lines. Each byte of data is typically composed of 8 bits, but the data frame size can be configured to support different bit lengths.

Chip Select (CS): The master device selects the slave device with a dedicated chip select (CS) signal. When the CS signal is asserted (set to a logic low), the slave device becomes active and ready to communicate. When the CS signal is de-asserted (set to a logic high), the slave device stops listening to the SPI bus.

b. Description of Modified Functions in the C Code

a) Lab 6.1

int main(): Its purpose is using an infinite loop to continuously execute the following accumulation code which can handle both normal addition as well as overflow situations. This function checks if any button is pressed by reading the value from the GPIO peripheral connected to the buttons. If a button is pressed (non-zero value), sets the pressed variable to 1. Then it checks if a button was previously pressed (indicated by pressed being non-zero) and is now released (indicated by GPIO value being 0). After two steps of checking, it resets the pressed variable to 0. Next, it checks if adding the value of switches to the LED GPIO data would cause overflow (LED value exceeds 65535). If overflow would occur, adds the switch value to the LED value, prints a message indicating overflow, and increments the LED value.



b) Lab 6.2

void MAXreg_wr(BYTE reg, BYTE val): This function writes a single byte value to a register in the MAX3421E USB host controller. It selects the MAX3421E device, sends the register address and data byte via SPI (Serial Peripheral Interface), and checks for any errors during the transfer. If an error occurs, it prints an error message. This function is used to configure and control the MAX3421E by writing to its registers.

```
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:
    //write val via SPI
    //read return code from SPI peripheral (see Xilinx examples)
    //if return code != 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)

    //select MAX3421E
    SpiInstance.SlaveSelectReg = 0;

    BYTE data[2] = {reg + 2, val};

    BYTE recv[2] = {0, 0};

    int ret_code = XSpi_Transfer(&SpiInstance, data, recv, 2);

    if (ret_code != 0) {

        xil_printf("Error!");

    }
}
```

BYTE MAXreg_rd(BYTE reg): This function reads a single byte value from a register in the MAX3421E USB host controller. It selects the MAX3421E device, sends the register address via SPI, receives the data byte from the register, checks

for any errors during the transfer, and returns the read byte. If an error occurs, it prints an error message. This function is used to retrieve information or status from the MAX3421E registers.

```
/* Single host register read        */
BYTE MAXreg_rd(BYTE reg) {
    //psuedocode:
    //select MAX3421E (
    //write reg via SPI
    //read val via SPI
    //read return code from SPI peripheral
    //if return code != 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return val
    SpiInstance.SlaveSelectReg = 0;

    BYTE data[2] = {reg, 0};

    BYTE recv[2] = {0, 0};

    int ret_code = XSpi_Transfer(&SpiInstance, data, recv, 2);

    if (ret_code != 0) {

        xil_printf("Error!");

    }

    return recv[1];
}
```

BYTE MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data):* This function writes multiple bytes of data to consecutive registers starting from the specified register address in the MAX3421E USB host controller. It selects the MAX3421E device, sends the register address followed by the data bytes via SPI, and checks for any errors during the transfer. If an error occurs, it prints an error message. This function is used for bulk data transfer or configuration of the MAX3421E.

```
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    SpiInstance.SlaveSelectReg = 0;

    int i;

    BYTE send[nbytes+1];

    BYTE recv[nbytes+1];

    for (i = 0; i < nbytes+1; i++) {

        if (i == 0) {

            send[i] = reg + 2;

        } else {

            send[i] = data[i-1];

            recv[i] = 0;

        }

    }

    int ret_code = XSpi_Transfer(&SpiInstance, send, recv, nbytes+1);

    if (ret_code != 0) {

        xil_printf("Error!");

    }

    return (data + nbytes);
}
```

BYTE MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data):* This function reads multiple bytes of data from consecutive registers starting from the specified register address in the MAX3421E USB host controller. It selects the MAX3421E device,

sends the register address via SPI, receives the data bytes from the registers, checks for any errors during the transfer, and stores the read data in the provided data array. If an error occurs, it prints an error message. This function is used for bulk data retrieval from the MAX3421E.

```
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    SpiInstance.SlaveSelectReg = 0;

    int i;

    BYTE send[nbytes+1];

    BYTE recv[nbytes+1];

    for (i = 0; i < nbytes+1; i++) {

        if (i == 0) {

            send[i] = reg;

        } else {

            send[i] = 0;

        }

    }

    int ret_code = XSpi_Transfer(&SpiInstance, send, recv, nbytes+1);

    for (i = 0; i < nbytes; i++) {

        data[i] = recv[i+1];

    }
    if (ret_code != 0) {

        xil_printf("Error!");

    }
    return (data + nbytes);
}
```

5. Answers to All INMB & Post Lab Questions

a) Answers to All INMB Questions

i. You should do some research and figure out what are some primary differences between the various presets which are available in the Microblaze Preset?

There're 10 presets available in the Microblaze Preset, namely Microcontroller, Real-time, Application, Minimum area, Maximum performance, Maximum frequency, Linux with MMU, Low-end with MMU, Typical, Frequency-optimized, and their differences are stated as below.

The **Microcontroller** preset is optimized for embedded control tasks, emphasizing low resource usage and simplicity. In contrast, the **Real-time** preset prioritizes deterministic response times, making it suitable for time-sensitive applications. The **Application** preset strikes a balance between performance and resource utilization, catering to general-purpose computing needs. The **Minimum area** preset sacrifices performance for minimal resource utilization, ideal for space-constrained environments. Conversely, the **Maximum performance** preset maximizes computational capabilities at the expense of resource consumption. The **Maximum frequency** preset aims to achieve the highest clock frequency possible

for speed-critical applications. **Linux with MMU** preset enables running Linux-based operating systems, providing memory management capabilities for complex applications. The **Low-end with MMU** preset offers a balance between resource utilization and MMU support for cost-sensitive designs. The **Typical** preset represents a standard configuration suitable for most applications, while the **Frequency-optimized** preset focuses on achieving the highest clock frequency within given resource constraints, targeting performance-critical applications.

ii. **Note that the bus connections coming from the Microblaze; is it a Von Neumann, 'pure Harvard', or 'modified Harvard' machine and why?**

It is a "modified Harvard" machine because it combines elements of Von Neumann and Harvard architectures. Like the pure Harvard architecture, our machine utilizes separate memory units for instructions and data. However, it also allows for some degree of flexibility by permitting instructions to be treated as data under certain conditions.

iii. **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?**

The term "asynchronous" in UART (Universal Asynchronous Receiver/Transmitter) refers to the method of data transmission where the sender and receiver do not share a common clock signal. Instead, they rely on agreed-upon timing parameters, such as baud rate, to synchronize the transmission and reception of data.

The advantages of asynchronous protocols are as follows. Asynchronous protocols usually make designs simpler and less expensive. Besides, since there's no need for strict synchronization, asynchronous protocols can accommodate variations in clock speeds between sender and receiver.

However, asynchronous protocols also have several disadvantages. For example, they typically have lower data transfer rates compared to synchronous protocols due to the additional overhead required for start and stop bits. Furthermore, asynchronous protocols may have limited distance capabilities compared to synchronous protocols because timing constraints become more stringent with longer transmission distances.

iv. **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.**

By using interrupts, the UART can signal the CPU when it's ready to transmit data or when data has been received. When an interrupt occurs, the CPU suspends its

current task, handles the interrupt, and then resumes its previous task. This allows the CPU to be more efficient by performing other tasks while waiting for the UART to complete its data transmission or reception. Since the UART takes a long time to transmit a single byte, if without interrupts, the CPU would need to continuously poll the UART, leading to inefficient CPU utilization and potentially slower overall system performance.

v. **Why are the UART and LED peripherals only connected to the data bus?**

First, by connecting these peripherals only to the data bus, the overall system design becomes simpler. It reduces the number of connections required between the peripherals and the CPU or memory, streamlining the design process, and reducing complexity.

Besides, both UART and LED peripherals typically do not require direct control signals from the CPU or memory. Instead, they are usually driven by data sent over the data bus. Connecting them directly to the data bus allows for efficient data transfer without the need for additional control signals.

Furthermore, the CPU can easily communicate with these peripherals by reading from or writing to memory-mapped addresses associated with the data bus, allowing for seamless integration into software routines.

vi. **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).**

For the provided blink function:

The volatile keyword indicates to the compiler that the value of led_gpio_data may change unexpectedly (such as through hardware interrupts), so it should not perform optimizations like caching or reordering accesses to this variable. The working example of the set and clear functions are as follows:

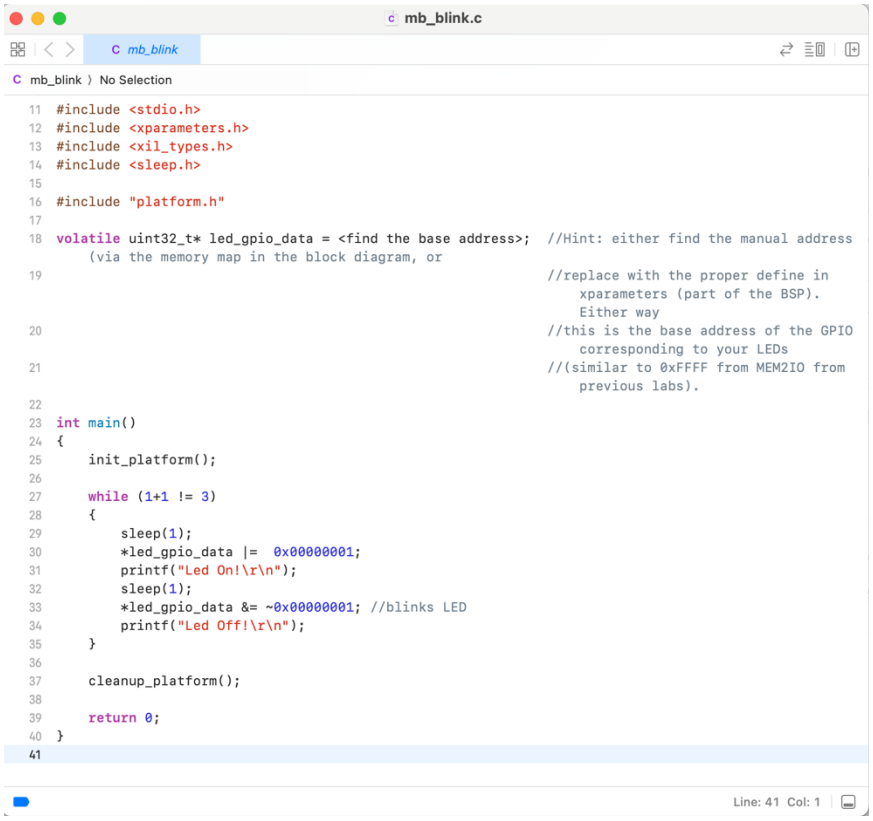
*led_gpio_data |= 0x00000001;

This line sets the least significant bit of the value pointed to by led_gpio_data to 1, effectively turning on the corresponding LED. It does this by performing a bitwise OR operation between the current value at the memory address pointed to by led_gpio_data and the value 0x00000001.

*led_gpio_data &= ~0x00000001;

This line clears the least significant bit of the value pointed to by led_gpio_data, effectively turning off the corresponding LED. It does this by performing a bitwise AND operation between the current value at the memory address pointed to by led_gpio_data and the bitwise complement of 0x00000001 (which is 0xFFFFFFFE). This clears the least significant bit

while leaving the rest of the bits unchanged.

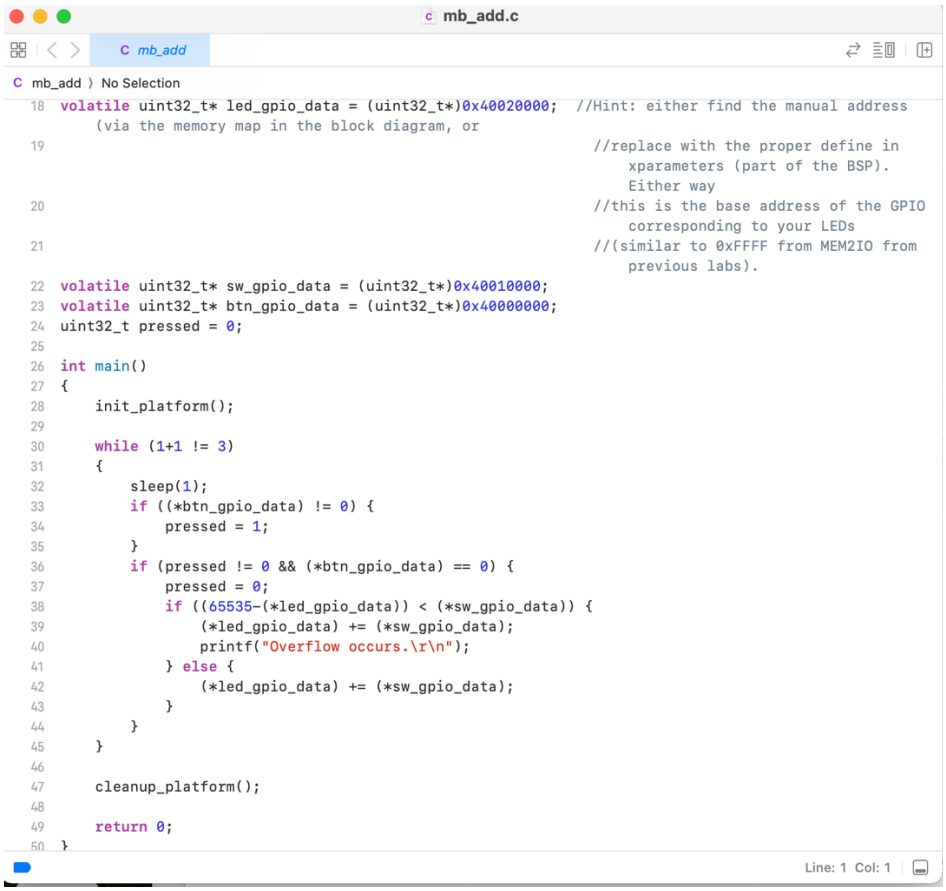


For our accumulator:

The volatile keyword indicates to the compiler that the values of led_gpio_data, sw_gpio_data, and btn_gpio_data may change unexpectedly (such as through hardware interrupts), so it should not perform optimizations like caching or reordering accesses to this variable. The working example of the set and clear functions are as follows:

(*led_gpio_data) += (*sw_gpio_data);

Checks if adding the value of switches to the LED GPIO data would cause overflow (LED value exceeds 65535). If overflow would occur, adds the switch value to the LED value, prints a message indicating overflow, and increments the LED value. If no overflow would occur, adds the switch value to the LED value. Therefore, the clear function are implicitly included in the set function as a role of overflow.



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

The **text segment**, also known as the code segment, contains the executable code of the program. This segment stores the compiled machine code instructions of the program's functions and procedures. It is typically read-only and fixed in size, as it contains the actual instructions that the CPU will execute. The text segment is where the program's logic resides, including function definitions, control structures, and executable statements.

The **data segment** contains initialized global and static variables. This segment stores data that has an initial value specified at compile time. Variables declared with an explicit initializer (such as int x = 10;) are stored in the data segment. Initialized arrays, strings, and other static data structures also reside in the data segment.

The **BSS (Block Started by Symbol) segment** contains uninitialized global and static variables. Unlike the data segment, the BSS segment does not store the initial value of variables. Instead, it allocates memory space for variables that will be initialized to zero at program startup. Variables declared without an explicit initializer (such as int y;) are stored in the BSS segment.

viii. **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.

For both the provided “blink” function and our accumulator, the function “printf” takes much of the program memory. When commenting out “printf” lines of code, the size of the program memory significantly decreases. It is because the “printf” function is part of the standard input/output (stdio) library in C. This library includes various functions for input and output operations, such as printing formatted data to the console. When you include “printf” in your code, the entire stdio library is often linked to your program, even if you're not using other functions from that library. This can result in a larger program size because the entire library is included, even if only a small portion of it is utilized.

Besides, the “printf” function is quite versatile and can handle a wide range of formatted output, including integers, floating-point numbers, strings, and more. To support this flexibility, “printf” requires additional code to handle various data types and formatting options. As a result, the function itself and its associated code can be relatively large, contributing to the overall program size.

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

The offset of GPIO2_DATA is 0x00000008. Therefore, we could access GPIO2_DATA at address 0x40000008.

b) Answers to Post-lab Questions

i. Design Statistics Table

	Lab 6.1	Lab 6.2
LUT	250	2799
DSP	6	9
Memory (BRAM)	0	8
Flip-Flop	79	2608
Latches	0	0
Frequency	135.7 MHz	122.399 MHz
Static Power	0.035 W	0.075 W
Dynamic Power	0.103 W	0.385 W
Total Power	0.138 W	0.46 W

6. Documentation of Design Sources and Statistics

	Lab 6.1	Lab 6.2
LUT	250	2799
DSP	6	9
Memory (BRAM)	0	8
Flip-Flop	79	2608
Latches	0	0
Frequency	135.7 MHz	122.399 MHz
Static Power	0.035 W	0.075 W
Dynamic Power	0.103 W	0.385 W
Total Power	0.138 W	0.46 W

7. Conclusion

a. Functionality of This Design

All parts of my design (both Lab 6.1 and Lab 6.2) work well with required functionalities. In the first week's lab, the focus lies on configuring the basic SoC setup with Microblaze CPU, memory, and rudimentary peripherals such as LEDs and switches. Through memory mapped I/O, my final version of Lab 6.1 design supports the Microblaze communicating with these peripherals, facilitating tasks like data input from switches, accumulation, and display on LEDs.

In the second week, my design is expanded to incorporate advanced peripherals like USB and VGA. The USB host controller communicates with the Microblaze via SPI, enabling interaction with USB devices such as keyboards. The keyboard input is polled constantly, allowing the Microblaze to receive key press information, which can be displayed via serial terminal. Moreover, VGA functionality is leveraged to draw and control a ball on an HDMI monitor. The ball's motion, governed by keyboard inputs, is managed seamlessly, with the ball bouncing off edges and changing directions instantly upon receiving direction commands.

b. Summary

The documentation and instructions of Lab 6.1 are clear and detailed, with some inspiring questions during the process of completing the block design, we can easily understand and catch up with using the new software. However, I personally think the documentation of SPI protocol can be more detailed with more annotations on provided SPI working diagrams. Simply viewing a timing diagram for the MAX3421E with 4 SPI modes is kind of ambiguous, since we have no idea of what the values mean and why the values are themselves. Maybe a more detailed and analyzed timing diagram will help us realize what we need to do in writing 4 SPI read/write functions.