ECE 385

Fall 2023

Experiment #6

SOC with Microblaze in SystemVerilog

Yunxuan Yang, Yuqi Wen
2023/10/28
TA: Tianhao Yu

1) Introduction

a. The Microblaze processor, designed for Xilinx FPGAs like the Spartan 7, is a soft, configurable processor core. Unlike hard processors, Microblaze is implemented using the FPGA's logic resources and can be tailored to application requirements. It operates on a modified Harvard architecture, combining the benefits of separate instruction and data pathways with the flexibility to access memory efficiently. Moreover, it supports memory-mapped I/O for interfacing with peripherals and is seamlessly integrated with Xilinx's Vivado Design Suite for easy design and deployment.

b. In Week 2's design, the primary focus was on integrating advanced peripherals with the Microblaze-based system-on-chip (SoC) on the Spartan 7 FPGA. Specifically, a USB host controller was interfaced with the Microblaze CPU to communicate with a USB keyboard for user input. Concurrently, a VGA signal was converted to HDMI, enabling the display of graphical content on an HDMI monitor. To demonstrate the system's capabilities, a ball was rendered on the screen, and its movement was controlled by the user through the keyboard, with the ball exhibiting realistic motion dynamics, including boundary interactions.

2) Written Description and Diagrams of Microblaze System

Module descriptions:

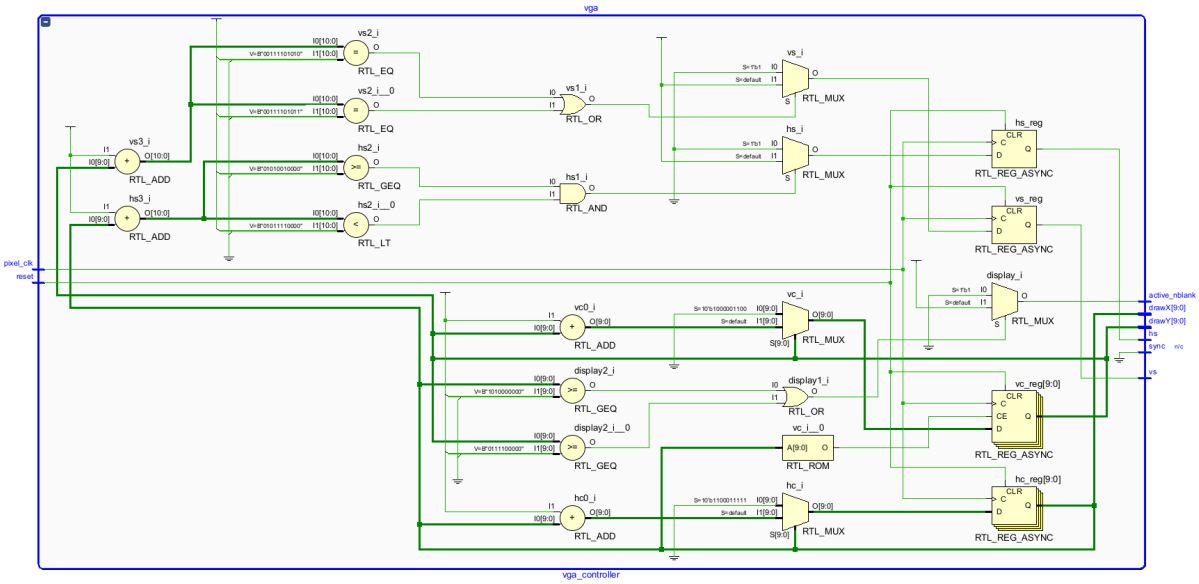
1. Module: VGA_Controller.sv

Inputs: pixel_clk, reset

Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY

Description: This module produces X, Y counters along with H, V sync signals. The blank and sync are left untouched.

Purpose: It's set up to display 640x480 pixels.



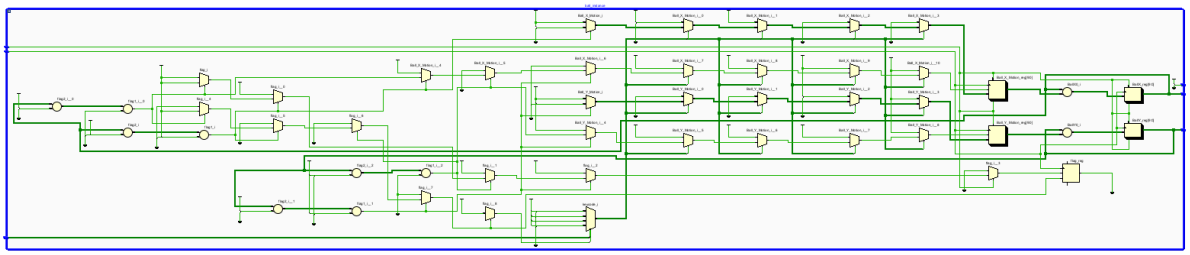
2. Module: ball.sv

Inputs: Reset, frame_clk, [7:0] keycode

Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS

Description: This module updates ball's position in every frame. Reacts to keystrokes W, S, A, D by altering the speed vector.

Purpose: The bouncing ball instance.



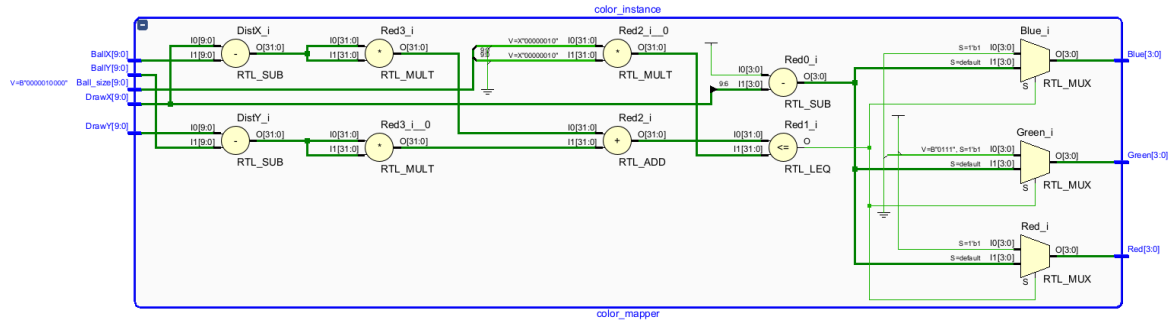
3. Module: Color_Mapper.sv

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

Outputs: [3:0] Red, [3:0] Green, [3:0] Blue

Description: Generates (pixelated) circle by using the standard circle formula.

Purpose: Translates line counters and ball properties to RGB signals.



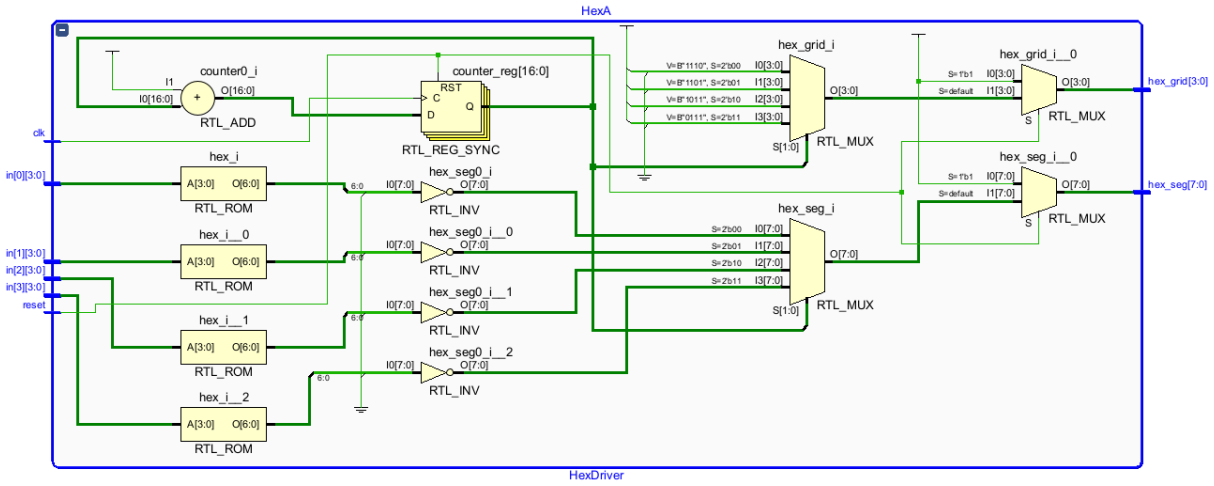
4. Module: HexDriver.sv

Inputs: [3:0] in, clk, reset

Outputs: [7:0] hex_seg, [3:0] hex_grid

Description: This module converts 4bit binary numbers from registers into hexadecimal to display digit on LED.

Purpose: Set the 7-segment LED display on board.



5. Module: mb_usb_hdmi_top.sv

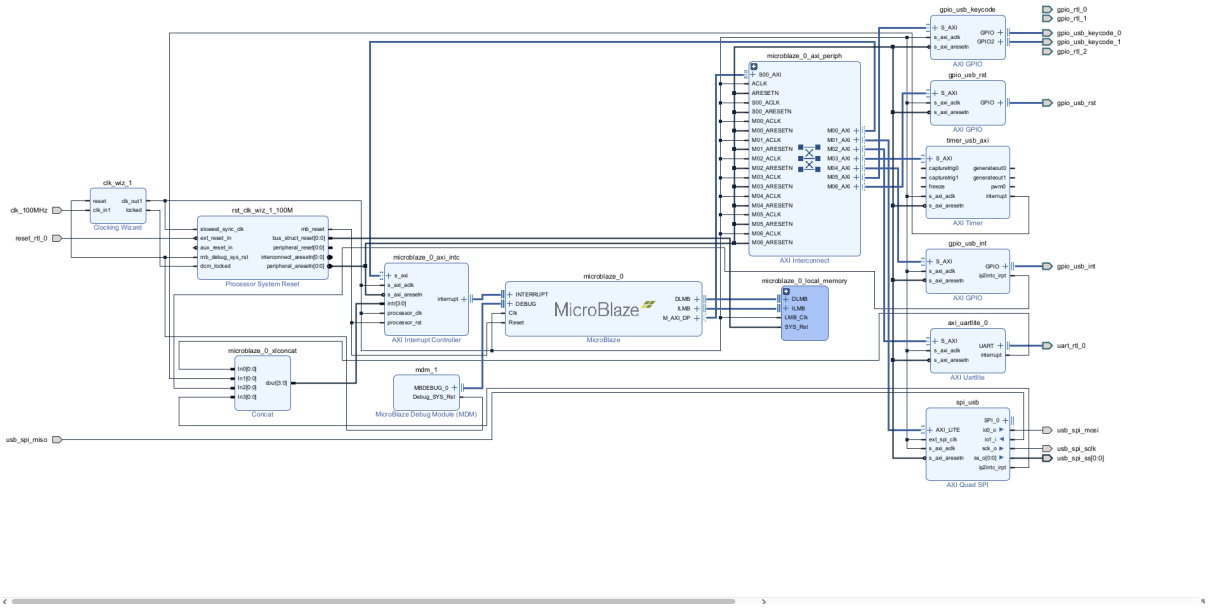
Inputs: Clk, reset_rtl_0: 1-bit, gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd

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, [3:0] hdmi_tmds_data_n, [3:0]

hdmi_tmds_data_p, [8:0] hex_segA, [8:0] hex_segB, [4:0] hex_gridA, [4:0] hex_gridB

Description: Top level for mb_lwusb test project.

Purpose: The module is used as a top-level module to connect each module in this project, receive input signal, compute it with adder, change it to hex representation, and display it on the board.



AXI Interconnect:

The AXI interconnect, a component of ARM's Advanced eXtensible Interface (AXI) within the AMBA framework, is a pivotal routing mechanism used in FPGAs and SoCs. It efficiently manages and routes data between multiple master and slave components, ensuring smooth data flow and handling arbitration during concurrent access. This interconnect is essential for coordinating high-bandwidth, low-latency communications in complex digital systems, seamlessly integrating various subsystems in modern electronic designs.

AXI Interrupt Controller:

The AXI Interrupt Controller is a crucial component in FPGA and SoC designs, serving as an intermediary between peripheral devices and the main processor or processing units. It consolidates multiple interrupt sources from AXI peripherals into a single or a reduced number of interrupt outputs, enabling the processor to efficiently handle and prioritize interrupt requests. By managing these interruptions in an organized manner, it ensures timely responses to urgent tasks and maintains smooth operation in complex digital systems.

AXI GPIO:

The AXI GPIO (General-Purpose Input/Output) module provides a bridge between the AXI interface and general-purpose I/O ports on FPGAs or SoCs. It allows designers to read from or write to pins directly via the AXI bus, enabling flexible, software-controlled interaction with external components or on-chip signals. This modularity is essential for tasks ranging from simple LED control to interfacing with custom hardware, ensuring that the broader system's communication standards are seamlessly maintained.

AXI Timer:

The AXI Timer module, integrated within FPGA and SoC designs, offers precise timing and control functionalities accessible through the AXI interface. By generating interrupts at specified intervals or counting external events, it enables timed operations, pulse width modulation, and event measurement. Serving as a crucial tool for system tasks like scheduling, time-outs, or frequency measurements, the AXI Timer ensures synchronization and time-driven actions in complex digital architectures.

AXI UARTlite:

The AXI UARTlite is a lightweight, compact module designed for communication between FPGA or SoC systems and external devices using the Universal Asynchronous Receiver-Transmitter protocol. Accessible through the AXI interface, it provides basic UART functionality, facilitating serial data transmission and reception with minimal resource overhead. Ideal for applications where advanced UART features aren't essential, the UARTlite offers a balance between simplicity and performance in digital communication systems.

AXI Quad SPI:

The AXI Quad SPI (Serial Peripheral Interface) module facilitates high-speed serial communication between FPGAs or SoCs and SPI devices, leveraging the ability to transmit data on four channels simultaneously (quad mode). Through the AXI interface, this module supports both standard SPI and Quad SPI modes, allowing for efficient data transfers, especially for applications like flash memory access or high-speed data streaming. Its integration ensures rapid data exchange and enhanced throughput in digital systems requiring swift SPI-based interactions.

Describe in Lab 6.1 how the I/O works.

Input: The primary inputs in this lab are the FPGA board switches (SW [15:0]). These switches represent a 16-bit unsigned binary number, where the up position indicates a binary '1' and the down position indicates '0'. Additionally, there are push buttons which provide control input: 'Reset' (btn[0]) and 'Accumulate' (btn[1]).

Output: The output is presented using the on-board LEDs. The accumulated value, post any processing, is displayed in binary on the LEDs. For example, if the accumulator value is 5 (binary: 0101), the corresponding LEDs will light up to represent this value. If the number overflow 0xffff, then the LEDs will be clear and the debug console output overflow message.

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

Interacting with the MAX3421E uses the SPI protocol, which encompasses four main pins: SS, SCLK, MOSI, and MISO. Each slave device has its unique Slave Select (SS) to indicate when it's being addressed, while the Slave Clock (SCLK) sets the pace for data exchange. To start a data exchange, the master dispatches a control byte via the Master-Out Slave-In (MOSI) line. In this byte, bits [7:3] specify the desired register, and bit [1] denotes the direction of data transfer (either 'Read' or 'Write'). Depending on this, the master either sends data using MOSI or retrieves data from the slave through the Master-In Slave-Out (MISO) line. Meanwhile, it controls the ball motion components through a combination of hardware interfaces in the top-level SystemVerilog design and specific behaviors defined in the ball.sv file.

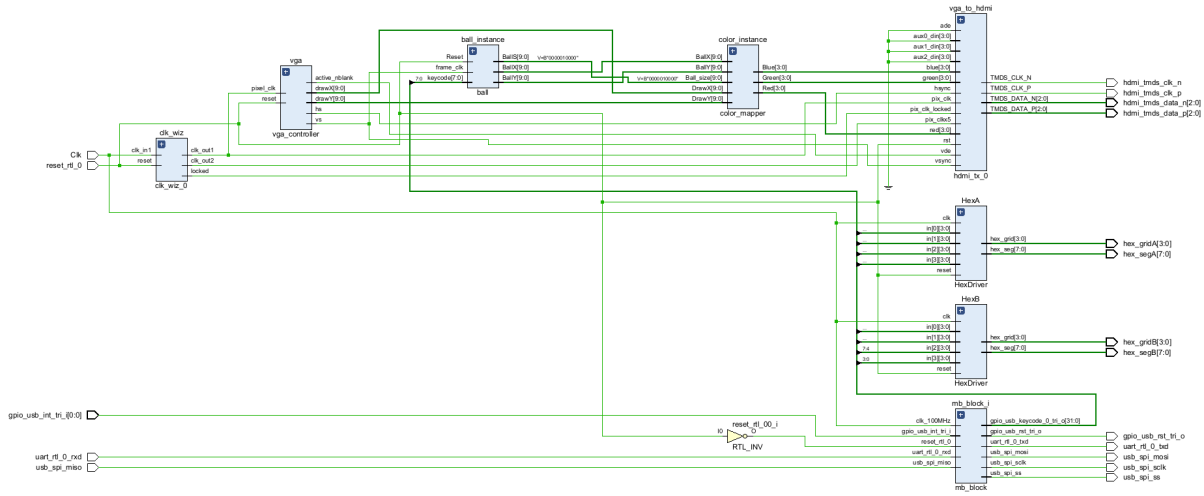
Describe in detail the VGA operation, and how your Ball, Color Mapper, and the VGA controller modules interact.

The VGA operates by transmitting analog signals, specifically Red, Green, Blue, and synchronization signals (HSYNC and VSYNC), to produce images on a screen. In this system, the Ball module provides the position and attributes of a ball on the display. The Color Mapper assigns colors to specific screen objects or regions, determining the ball's hue based on factors like its position. As the VGA Controller scans the screen pixel by pixel, it collaborates with the Ball and Color Mapper modules: it identifies the ball's location and fetches its color, ensuring accurate representation of the ball's movement and appearance on the VGA output.

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

The VGA-HDMI IP facilitates the transition between the analog VGA and digital HDMI signals, allowing them to interface with one another. VGA, an older analog interface, primarily transmits RGB video signals with separate synchronization, whereas HDMI is a digital interface that conveys both video and audio data, offering superior quality and resolution capabilities. Despite their differences in transmission nature and quality, both aim to provide a means of video connection between devices, albeit through different technological methods and standards.

3) Top Level Block Diagram



4) Software component description

Blink:

Every second, it sets the LED GPIO data to turn on all the LEDs, indicated by the message "Led On!", and then waits another second to turn them all off, printing the message "Led Off!". This blinking pattern repeats indefinitely until the system is interrupted or powered off.

```c
int main()
{
    init_platform();
    volatile int result = 0;
    int flag=1;
    while (1)
    {
        sleep(1);
        *led_gpio_data =  0x000000FF;
        printf("Led On!\r\n");
        sleep(1);
        *led_gpio_data &= ~0x000000FF; //blinks LED
        printf("Led Off!\r\n");
    }

    cleanup_platform();

    return 0;
}
```

Accumulator:

The code continuously checks the status of a GPIO input (acc_gpio_data). When accumulate button is pressed and a flag is set, the value from switch_gpio_data is accumulated into the result, also setting flag to zero. This accumulation is indicated by the message "accumulated". If the accumulated result exceeds 65535, it resets to zero, signaling an "overflow", and updates LEDs to reflect the current result. Additionally, if accumulate button is not pressed and the flag is not set, the flag is then set to 1, preparing for the next accumulation. This loop continues indefinitely until the system is externally interrupted or powered off, after which the platform resources are cleaned up.

```
int main()
{
    init_platform();
    volatile int result = 0;
    int flag=1;
    while (1)
    {
        if(((*acc_gpio_data)&1) == 1&&flag){
            result += *switch_gpio_data;
            printf("accumulated\r\n");
            flag=0;
            if(result>=65535){
                result=0;
                printf("overflow!\r\n");
            }
            *led_gpio_data = result;
        }

        else if(((*acc_gpio_data)&1) == 0&&flag==0)flag=1;
    }

    cleanup_platform();

    return 0;
}
```

MAXreg_wr:

The function MAXreg_wr is designed to write a byte value to a specific register of the MAX3421E device using the SPI protocol. Initially, the MAX3421E is selected via the XSpi_SetSlaveSelect function. The target register is determined by adding 2 to the provided reg parameter, and this, along with the value val, is stored in the send_buffer. A SPI transfer is then initiated using the XSpi_Transfer function, which sends the contents of the send_buffer and receives data into the receive_buffer. If the transfer encounters an error (indicated by a non-zero status), an "Error!" message is printed. Finally, the MAX3421E is deselected, completing the write operation.

```
/* Functions    */
/* Single host register write   */
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:
    //select MAX3421E
    //write reg + 2 via SPI
    //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)
    int status;
    BYTE send_buffer[2], receive_buffer[2];

    XSpi_SetSlaveSelect(&SpiInstance, 0x01);    //select MAX3421E
    send_buffer[0] = reg + 2;
    send_buffer[1] = val;

    status = XSpi_Transfer(&SpiInstance, send_buffer, receive_buffer, 2);

    if(status != 0){
        xil_printf ("Error!");
    }

    XSpi_SetSlaveSelect(&SpiInstance, 0x00);    //deselect MAX3421E
}
```

MAXBytes_wr**:**

The MAXBytes_wr function facilitates multi-byte writing to the MAX3421E device using the SPI protocol. After selecting the MAX3421E, the function sets the target register by adding 2 to the input reg and places it in the beginning of the send_buffer.

Following this, the provided data bytes are loaded into the buffer sequentially. The SPI transfer is then initiated, sending the data from the buffer. If any error arises during the transfer (evidenced by a non-zero status), an "Error!" message is displayed. Post-transfer, the MAX3421E is deselected, and the function returns a pointer pointing to the position after the last written byte in the input data.

```
/* multiple-byte write */
/* returns a pointer to a memory position after last written */
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write data[n] via SPI, where n goes from 0 to nbytes-1
    //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 (data + nbytes);
    int status;
    BYTE send_buffer[nbytes+1], receive_buffer[nbytes+1];
                        <error-type> SpiInstance
    XSpi_SetSlaveSelect(&SpiInstance, 0x01);    //select MAX3421E
    send_buffer[0] = (reg + 2) ;
    for(int i = 0; i < nbytes; i++){
        send_buffer[i + 1] = data[i];
    }

    status = XSpi_Transfer(&SpiInstance, send_buffer, receive_buffer, nbytes+1);

    if(status != 0){
        xil_printf ("Error!");
    }

    XSpi_SetSlaveSelect(&SpiInstance, 0x00);    //deselect MAX3421E
    return data+nbytes;

}
```

MAXReg_rd**:**

The MAXReg_rd function is designed to read a single register from the MAX3421E device using the SPI protocol. Initially, it selects the MAX3421E and prepares the send_buffer with the desired register value and a placeholder byte. Through the SPI transfer, it sends the target register and receives the corresponding value in the receive_buffer. If the SPI transfer encounters an issue (as indicated by a non-zero status), an "Error!" message is printed. After reading, the MAX3421E device is deselected. The function then returns the value read from the specified register, which is stored in the second byte of the receive_buffer.

```c
/* 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
    int status;
    BYTE send_buffer[2], receive_buffer[2];

    XSpi_SetSlaveSelect(&SpiInstance, 0x01);    //select MAX3421E
    send_buffer[0] = reg ;
    send_buffer[1] = 0;

    status = XSpi_Transfer(&SpiInstance, send_buffer, receive_buffer, 2);

    if(status != 0){
        xil_printf ("Error!");
    }

    XSpi_SetSlaveSelect(&SpiInstance, 0x00);    //deselect MAX3421E
    return receive_buffer[1];
}
```

MAXbytes_rd**:**

The MAXbytes_rd function facilitates the reading of multiple bytes from a specified register of the MAX3421E device via the SPI protocol. Upon selecting the device, it initializes the send_buffer with the targeted register followed by placeholder bytes. An SPI transfer then reads the desired bytes into the receive_buffer. If any errors arise during this transfer, denoted by a non-zero status, an "Error!" message is displayed. After reading, the acquired data is transferred from the receive_buffer to the provided data array. Finally, the MAX3421E device is deselected, and the function returns a pointer to the memory position after the last byte read.

```
/* multiple-bytes register read                              */
/* returns a pointer to a memory position after last read    */
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read data[n] from SPI, where n goes from 0 to nbytes-1
    //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 (data + nbytes);
    int status;
    BYTE send_buffer[nbytes+1], receive_buffer[nbytes+1];

    XSpi_SetSlaveSelect(&SpiInstance, 0x01);    //select MAX3421E
    send_buffer[0] = reg ;
    for(int i = 0; i < nbytes; i++){
        send_buffer[i + 1] = 0;
    }

    status = XSpi_Transfer(&SpiInstance, send_buffer, receive_buffer, nbytes+1);

    if(status != 0){
        xil_printf ("Error!");
    }
    for(int i = 0; i < nbytes; i++){
        data[i] = receive_buffer[i+1];
    }
    XSpi_SetSlaveSelect(&SpiInstance, 0x00);    //deselect MAX3421E
    return data+nbytes;
}
```

Written description of the SPI protocol:

The XSpi_SetSlaveSelect function is an SPI interface function that manages the selection of a specific slave device on the SPI bus. By manipulating Slave Select (SS) line, this function determines which slave device is active and ready for communication. When invoked, it receives parameters that indicate which slave device to activate, ensuring that only the desired peripheral listens to the master's communication, while other devices remain inactive or in a standby state. It's utilized to select or deselect the MAX3421E device for data transfer operations.

The XSpi_Transfer function facilitates the transmission and reception of data between the master and the selected slave device over the SPI bus. When invoked, it takes parameters specifying the data buffers for both sending and receiving data, as well as the length of data to be transferred. The function initiates an SPI communication cycle, sending data from the master to the slave and simultaneously receiving data from the slave to the master. Once the data transfer is complete, the function returns a

status indicating the success or failure of the operation. This function is crucial for reading from and writing to the MAX3421E device.

The first byte is command byte, indicating the register address and !R/W. The given register is already shifted 3 bits so if for write functions, we add 2 to the reg indicating the second last bit is 1 while for read is 0 for this bit so no add is needed. ACKSTAT is considered 0 for this implementation.

5) Answers to all questions

1. **Note the bus connections coming from the MicroBlaze; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?**
   The MicroBlaze soft processor core, designed for Xilinx FPGAs, employs a Modified Harvard Architecture. It has separate instruction (LMB) and data (DLMB) interfaces, allowing simultaneous instruction and data fetches. Although it has distinct buses, they can connect to same memory. Thus, MicroBlaze is classified as a "modified Harvard" machine due to its hybrid bus and memory configuration approach.

2. **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 indicates that data transmission occurs without a shared clock signal between sender and receiver. Instead, the receiver synchronizes based on the transmitted data's start bit. Asynchronous protocols, like UART, don't require a dedicated clock line, making them simpler and more adaptable to devices with varying clock sources. However, the overhead of start and stop bits can reduce efficiency, especially with large data sets. Conversely, synchronous protocols offer higher data rates due to continuous streams without start and stop bits but necessitate a shared clock, adding to the complexity and potential susceptibility to noise interference.

3. **Why are the UART and LED peripherals only connected to the data bus?**
   The UART and LED peripherals typically only require data transfers, meaning they read or write data but don't execute instructions. The UART handles serialized communication, so it needs to send or receive data bytes. Similarly, LED peripherals are generally driven by data values representing their on/off state or brightness. They don't need to fetch executable instructions from memory. Thus, connecting these peripherals to only the data bus simplifies design and reduces unnecessary bus traffic, focusing solely on the data they need to process or represent.

4. **Look at the various segments (text, data, bss), what does each segment mean? What kind of code elements are stored in each segment?**
   The memory of a computer program is divided into segments to manage and organize its data and instructions. The Text Segment contains a program's executable instructions and is typically read-only. The Data Segment holds initialized global and static variables, while the BSS Segment, a subset of the Data Segment, is reserved for uninitialized global and static variables which are auto-initialized to zero. These segments ensure efficient memory usage, proper organization, and safe execution of programs by keeping executable and data portions separate.

5. **Why does the provided code, which does very little, take up so much program memory?**
   Even simple code can consume substantial program memory due to several factors. Firstly, overhead from libraries and initialization routines might be included in the final binary. Secondly, the binary might contain debugging information, adding to its size. Additionally, compiler optimizations for speed can result in larger code, while data structures and static variables further increase memory usage. A single line of high-level code can also translate into multiple machine instructions, and the inclusion of runtime environment support can further bloat the program's footprint.

6. **If the base address is 0x40000000, how would you access GPIO2_DATA (for example?).**
   To access GPIO2_DATA from a base address of 0x40000000, we'd add the offset of GPIO2_DATA to this base.

6) Design Resources and Statistics in table

LUT	2762
DSP	6
Memory	8
Flip-Flop	2777
Latches	0
Frequency	512.3 MHz
Static Power	75 mW

Dynamic Power	388 mW
Total Power	463 mW

7) Conclusion

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

The design integrates multiple interfaces including USB, UART, HDMI, and HEX displays, all orchestrated by the MicroBlaze processor. Our design works as expected.

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

The lab manual explains what we need along with the two provided tutorial files.