ECE 385

Fall 2024

Experiment #6

SOC with Microblaze in SystemVerilog

Krish Gandhi: krishjg2

Aashay Soni: asoni29

Introduction

Summary of Basic Function of Microblaze Processor running on Spartan 7 FPGA

In Lab 6, we familiarized ourselves with the soft-core Microblaze processor within a System-on-chip framework on the FPGA. In week 1, we were tasked with creating a simple adder using the Microblaze processor to configure I/O modules that facilitate communication between the switches and LEDs. The Microblaze will perform a basic accumulation operation, reading 16-bit numbers from switches and summing them to an accumulator displayed on the LEDs.

Summary of Operation of Week 2 Design

In Week 2, we expanded the functionality by adding USB and VGA peripherals that enable interaction with a USB keyboard to control an on-screen ball displayed on an HDMI monitor. Using SPI communication, the Microblaze processor interfaces with the MAX3421E USB host controller to retrieve keypress data, which the Microblaze interprets to direct the ball's movement on the display. The MAX3421E enables this process by facilitating USB communication through SPI, allowing the Microblaze to communicate with the FPGA to manage keyboard inputs and update the ball's position accordingly. The design also includes a VGA-to-HDMI IP block, converting VGA signals into HDMI format, allowing real-time visualization of the ball's motion on an external HDMI monitor in response to user input.

Written Description and Diagrams of Microblaze Systems

Module Descriptions

Module: mb_intro_top.sv
Inputs: clk, [15:0] sw, [3:0] btn, uart_txd
Outputs: [15:0] led, uart_rxd
Description: This is the top-level module for the Week 1 design, connecting the Microblaze processor and handling inputs and outputs such as LEDs and UART. It coordinates components for a simple LED blinking demonstration.
Purpose: This module introduces the Microblaze in Week 1, setting up the environment for basic control over LEDs and serial communication.

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: The VGA controller is responsible for generating timing signals that control the display's synchronization and scanning. It produces horizontal and vertical sync pulses to drive each line and frame refresh, allowing the screen to display images at a resolution of 640x480. The module also includes logic to blank areas outside the display boundaries and reset the sync signals at appropriate intervals.

Purpose: This module is called in the top level module to sync the VGA output. It drives the drawX and drawY signals, which are then sent to ColorMapper.sv.

Module: mb_usb_hdmi_top.sv
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 top-level module integrates the Week 2 design components, including SPI communication, USB control, VGA-HDMI conversion, and hex displays. It manages signals between submodules and connects inputs and outputs like UART, SPI, and HDMI signals.
Purpose: This module acts as the central hub for the Week 2 design, coordinating communication between different components and enabling the SoC to interface with various peripherals.

Name: hex_driver.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: This module converts the four 4-bit input signals in[4] from binary value to an 8-bit binary value that represents a hexadecimal character on the seven segment display. We made no changes to this module.
Purpose: This module is used in the top level module to convert data from hexadecimal representations to seven segment displays. These representations are then outputted to the seven segment displays on the Urbana board. This was useful for showing debug information directly on the FPGA.

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: The Color Mapper determines the RGB value for each pixel based on its position. It calculates the distance from the center of the ball (BallX, BallY) to each pixel's position (DrawX, DrawY) and applies color accordingly. If the pixel is within the ball's radius (Ball_size), the pixel should be orange. If it is not, the background should be a gradient of shades of gray. We made no changes to this module.
Purpose: This module is used in the top level module to assign the RGB values of pixels on the display. The output Red, Green, and Blue signals are fed to the VGA to HDMI converter to be sent to the screen.

Module: ball.sv

Inputs: Reset, frame_clk, [7:0] keycode
Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS
Description: The Ball module controls the position and movement of the ball on the screen. We made changes to implement movement in the X direction. Based on keyboard inputs (keycode), it updates the X and Y coordinates, ensuring the ball moves according to user commands (up, down, left, right) and bounces back upon hitting screen boundaries.
Purpose: This module is called in the top level module. It takes in the keycode from the Microblaze processor and allows the user to control ball movement across the screen.

Block Design
Component: microblaze_0
Inputs: INTERRUPT, DEBUG, Clk, Reset
Outputs: DLMB, ILMB, M_AXI-DP
Description: This is the core processing unit, a 32-bit soft-core Microblaze processor that executes program instructions and manages data flow within the system. It interfaces with connected peripherals through the AXI interconnect.

Component: gpio_usb_keycode
Inputs: S_AXI, s_axi_aclk, s_axi_aresetn
Outputs: GPIO, GPIO2
Description: This is an AXI GPIO module that provides IO for the keyboard. In this design, it drives the 2 32-bit keycode inputs.

Component: gpio_usb_rst
Inputs: S_AXI, s_axi_aclk, s_axi_aresetn
Outputs: GPIO
Description: This is an AXI GPIO module that provides IO for the USB reset. In this design, it drives the 1-bit gpio_usb_rst input.

Component: gpio_usb_int
Inputs: S_AXI, s_axi_aclk, s_axi_aresetn
Outputs: GPIO, ip2intc_irpt
Description: This is an AXI GPIO module that provides IO for the USB interrupts. In this design, it drives the 1-bit gpio_usb_int input.

Component: Microblaze_0_axi_intc
Inputs: s_axi, s_axi_aclk, s_axi_aresetn, intr[3:0], processor_clk, processor_rst
Outputs: interrupt

Description: An AXI Interrupt Controller that manages and prioritizes interrupt signals from various peripherals, ensuring the Microblaze processor can handle multiple external events efficiently by routing and handling interrupt signals.

Component: axi_uartlite_0
Inputs: S_AXI, s_axi_aclk, s_axi_aresetn
Outputs: UART, interrupt
Description: An AXI Uartlite module for basic serial communication, facilitating data transmission and reception over a serial interface. It is useful for debugging or for simple data exchange between the system and external devices.

Component: spi_usb
Inputs: AXI_LITE, ext_spi_clk, s_axi_aclk, s_aci_aresetn, io1_i
Outputs: SPI_0, io0_o, sck_o, ss_o[0:0], ip2intc_irpt
Description: An AXI Quad SPI module used to implement the SPI protocol. This component drives the usb_spi_mosi, usb_spi_sclk, and usb_spi_ss[0:0] signals.

Component: microblaze_0_local_memory
Inputs: BLMB, ILMB, LMB_Clk, SYS_Rst
Outputs: None
Description: This component is used as the local memory for the Microblaze processor.

Component: timer_usb_axi
Inputs: S_AXI, s_axi_aclk_ s_aci_aresetn
Outputs: interrupt
Description: This component is an AXI Timer. It's interrupt signal is fed to the concat component, which is then fed into the AXI Interrupt Controller.

Component: clk_wiz_1
Inputs: reset, clk_in1
Outputs: clk_out1, locked
Description: This component is a Clocking Wizard. It is used as the clock for the entire block diagram.

Component: rst_clk_wiz_1_100M
Inputs: slowest_sync_clk, ext_reset_in, mb_debug_sys_rst, dcm_lcoked
Outputs: mb_reset, bus_struct_reset[0:0], peripheral_aresetn[0:0]
Description: This component is a Processor System Reset. It is used to reset the other components in the block diagram, such as the MicroBlaze, the Interrupt Controller, the Interconnect and more.

Component: xlconcat_0
Inputs: in0[0:0], in1[0:0], in2[0:0], in3[0:0]
Outputs: dout[3:0]
Description: This is a concat component. In the block diagram, it is used to concatenate interrupt signals and then feed them to the Interrupt Controller.

Component: mdm_1
Inputs: None
Outputs: MDDEBUG_0, Debug_SYS_Rst
Description: This is a MicroBlaze Debug Module. The output of this is fed into the MicroBlaze debug input, and the Clocking Wizard reset.

Component: microblaze_0_axi_periph
Inputs: S00_AXI, ACLK, ARESETN, S00_ACLK, S00_ARESETN, M00_ACLK, M00_ARESETN, M01_ACLK, M01_ARESETN, M02_ACLK, M02_ARESETN, M03_ACLK, M03_ARESETN, M04_ACLK, M04_ARESETN, M05_ACLK, M05_ARESETN, M06_ACLK, M06_ARESETN,
Outputs: M00_AXI, M01_AXI, M02_AXI, M03_AXI, M04_AXI, M05_AXI, M06_AXI
Description: This component is the AXI Interconnect. Essentially, this component connects one master device to multiple slave devices. In our block diagram, the outputs are fed into the S_AXI inputs for the AXI GPIOs, AXI Timer, and AXI Uartlite.

Lab 6.1 - How the I/O Works

The Microblaze processor within our FPGA configuration communicates with external components through memory-mapped I/O that allow software running on the Microblaze to access I/O peripherals as if they were memory locations. In lab 6.1, we established connections to onboard I/O using GPIOs for interfacing with switches and LEDs, as well as serial communication interfaces using UART for external data transmission. Externally, we used the Block Design to create an accumulator that is controlled by an execute and reset button. Our C program allows the Microblaze to read the values directed by the switches and add it to the stored value which is then displayed onto the FPGA using LEDs.

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

The Microblaze processor manages the ball's motion on the display by connecting the between user inputs and the FPGA. The FPGA uses the ball's current coordinate signals to continuously update the display and correctly render the next position of the ball in real time. Each key press from a connected keyboard is processed through the

Microblaze and sent to the FPGA, so it can perform logic and control the ball's movement across the screen. The Microblaze translates these inputs into keycode signals, which are then sent to the FPGA. The MAX3421E USB host controller enables the Microblaze to send and receive data through the USB keyboard. When a data transfer is initiated, the MAX3421E's internal read and write buffers temporarily store the data, making it accessible for the Microblaze to process. This buffer system allows the Microblaze to update the display based on immediate user inputs. The Microblaze can continuously manage and synchronize input from the keyboard with the ball's motion output on the display.

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

The VGA module generates timing signals, determines pixel coordinates, and enables the display at a 640x480 resolution. It uses a display enable signal to decide the color for each pixel which distinguishes between areas within the ball's boundaries and other screen regions based on their distance from a defined DrawX position. The Color Mapper module takes the pixel coordinates (DrawX and DrawY) from the VGA and returns the RGB value for that pixel. Pixels within the ball are displayed in orange, while areas further from DrawX vary in gradient levels of gray. The Ball module manages the ball's motion by updating its X and Y coordinates in response to keypress inputs—moving up or down by adjusting Y, or left or right by adjusting X. It ensures the ball bounces back upon reaching display boundaries. The Ball, Color Mapper, and VGA modules work in sync to control and display the ball's movement and appearance on the screen. As the VGA controller scans the display pixel by pixel, it collaborates with the Ball module to identify the ball's location and with the Color Mapper to retrieve the correct color which creates a real-time representation of the ball's movement and position on the screen. This setup enables dynamic and responsive interaction with visual feedback on the VGA display.

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

The VGA-HDMI IP converts analog VGA signals into digital HDMI output since modern computers don't use VGA. VGA transmits color information in the RGB format and has different channels for audio and video while HDMI transmits both audio and video in packets, allowing for better resolution and synchronization capabilities. HDMI supports higher resolutions compared to VGA's 640x480 resolution because of the use of analog signals in VGA controllers. Analog signals are much slower and more susceptible to noise compared to digital signals which reduces the speed and quality of signals in VGA.

Top Level Block Diagram

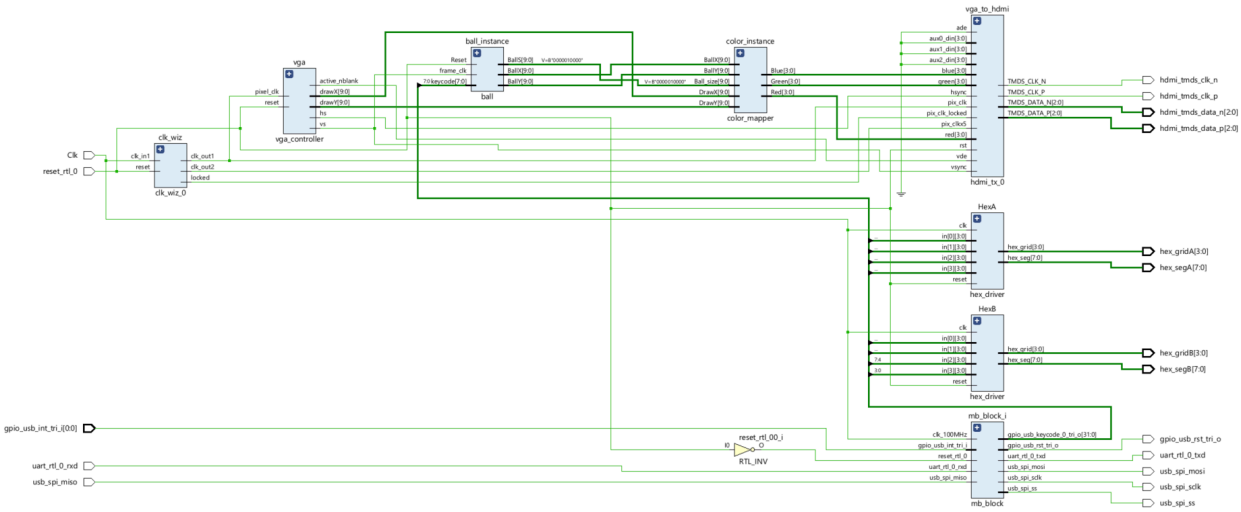

Figure 1: Top Level Block Diagram of Circuit

Describe in words the software component of the lab.

Written Description of SPI Protocol

SPI protocol is a communication method used for communication between microcontrollers and peripherals. In the protocol, data is usually transferred in a full duplex, meaning 8 bits are read and written during the same 8 clock cycles through MISO and MOSI. The protocol relies on four main signals: CLK, MOSI, MISO and SS.
CLK is controlled by the master. MOSI and MISO carry data between the master and slave. SS selects the wanted slave device, which allows multiple devices to use the same bus.

In the context of the MAX3421E, SPI facilitates the communication between the Microblaze and the FPGA. The MAX3421E manages USB communications for the connected keyboard. This means the MAX3421E uses SPI to write to and read from specific registers. This is crucial for the design to work as intended.

Purpose of Each Implemented Function

In this lab, we had to implement 4 functions in the MAX3421E.c file. Since the MAX3421E is the USB peripheral/host controller, this file has the implementation for low-level functions, such as reading from registers, writing to registers, reset, host transfer, etc. We implemented the following functions: MAXreg_wr(BYTE reg, BYTE val), MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data), BYTE MAXreg_rd(BYTE reg), MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data).

MAXreg_wr(BYTE reg, BYTE val):

This function is used as a single host register write. It writes the byte value val via SPI. This function will print an error statement if the status code from the SPI peripheral is not 0. This function has no return value.

MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data):
This function is used as a multiple byte write. It writes nbytes of data starting at the memory address data via SPI. This function will print an error statement if the status code from the SPI peripheral is not 0. This function returns a pointer to the memory address after the last written byte.

MAXreg_rd(BYTE reg):
This function is used as a single host register read. It reads the value from register reg via SPI. It does this by first writing the register it wants via SPI and then reading the value via SPI. This function will print an error statement if the status code from the SPI peripheral is not 0. This function returns the value that was read from the specified register.

MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data):
This function is used to read multiple bytes from registers. It reads nbytes values from register reg via SPI. It does this by first writing the register it wants via SPI and then reading the values via SPI. This function will print an error statement if the status code from the SPI peripheral is not 0. This function returns a pointer to the memory address after the last read byte.

Answers to all INMB (italicized) & Post lab questions

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

The microcontroller preset is suitable for microcontroller designs and it optimizes the area needed for a functionality, does not have caches, and has debug enabled. Furthermore, the real-time preset is suitable for real-time control so it optimizes performance, has a small cache, and debug is enabled. Lastly, the application preset is used for high performance applications so it optimized performance, has a large cache, debug is enabled and all executions include floating point

Note the bus connections coming from the MicroBlaze; is it a Von Neumann, “pure Harvard”, or “modified Harvard” machine and why?

The bus connections coming from the Microblaze are a modified Harvard machine because we have different buses for instruction and data but reserve the functionality to overlap their address spaces. The two buses can perform parallel access to memory. The Von Neumann bus has shared instruction and data in a single bus architecture. On

the other hand, the Pure Harvard bus has separate bus architecture for instruction and data. Lastly, the modified Harvard bus has separate buses for data and instruction that share the same address space.

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” in the UART refers to the clockless transmission of data between the sender and receiver. The UART uses the start bit in the data to start synchronization between the sender and receiver. An asynchronous UART is helpful because it’s simpler to program, doesn’t cost much memory, and its data rate can be flexible. On the other hand, a synchronous signal refers to the transmission of data between a receiver and sender with the help of a clock. A synchronous clock is costly in terms of data consumption but it is less prone to error and faster.

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.

Interrupts are needed so that the system is not constantly checking for inputs from your keyboard. An interrupt stops the program so that the system can read the inputs from the keyboard which allows the processor to efficiently handle UART data because it will check the inputs when they’re ready rather than constantly polling for them. Furthermore, since the UART doesn’t have a clock, it doesn’t know when the inputs are ready.

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

The UART and LED peripherals are connected only to the data bus because they require simple data transfers rather than complex memory operations. In a typical setup, the address bus is used to specify memory locations, while the data bus is responsible for carrying the actual data values. Since the UART and LED peripherals operate as simple I/O devices, they don’t need addressable memory locations

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 volatile signal casts a variable so that the variable can change at any time without any interruption. Since the LED status may change due to hardware actions or external

events, volatile ensures that each read and write occurs exactly as written in the code, preventing unexpected behaviors caused by compiler optimizations.

In lines 30 and 33, the while loop runs infinitely and sets the first LED to high, prints “LED on” and ORs the current value with the high. After, it sets the LED to 0 by using AND with ~1, then printing “LED off”. The cleanup_platform signal disabled the hardware, disables caches, and ensures all data is returned to memory

Answer to 4a: The accumulator works by continuously summing values read from the switches and displaying the result on the LEDs. The accumulator starts out at zero until a binary value is inputted and added to the total after which each consecutive binary number inputted from the switches is added onto the total. Each switch that is turned high counts as 1 bit in a 16 bit integer which is then added to the current value of the accumulator. This updated total is sent to the LEDs which display the sum as a 16 bit integer. After a reset button is pressed, the accumulator is set back to zero and the LEDs are all turned low. The program runs in an infinite loop, repeatedly reading inputs, updating the accumulator, and displaying the results, allowing the accumulator to maintain a running total of all values entered until reset.

Look at the various segments (text, data, bss), what does each segment mean? What kind of code elements are stored in each segment?
Text -> executable code
Data -> global and static variables that are initialized
BSS -> uninitializes block and the global and static variables. Starts at the end of the segment

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.
Simple programs consume significant memory space due to multiple reasons: startup code and included libraries needed for initialization, overhead storage associated with setting up and utilizing hardware peripherals, and compiler and linker configurations that might not optimize by default, such as debug information and inclusions of unused code

Make sure you understand the register map on page 10. If
the base address is 0x40000000, how would you access GPIO2_DATA (for example?)
We would add the specified offset to the base to access GPIO2_DATA
volatile uint32_t* led_gpio_data = 0x40000000

Design Resources and Statistics Table

LUT	2796
DSP	9
Memory (BRAM)	8
Flip-Flop	2608
Latches	0
Frequency	126.89 MHz
Static Power	0.075 W
Dynamic Power	0.383 W
Total Power	0.459 W

Table 1: Design Resources and Statistics in IVT

Table 1 shows the Design Resources and Statistics for our design. Frequency was calculated using the formula $f = \frac{1}{\mathrm{T} - \mathrm{WNS}}$, where the period T is known to be $10\mathrm{ns}$. The Worst Negative Slack WNS of our design was found to be $2.119\mathrm{ns}$. Therefore, the frequency was calculated to be 126.89 MHz.

Conclusion

Functionality of our Design

In this lab, we demonstrated the capabilities of the Microblaze processor in creating an FPGA-based System-on-Chip (SoC). Week 1 focused on basic I/O operations using GPIOs to control LEDs and switches, which introduced how the Microblaze manages simple peripherals. In Week 2, we expanded by integrating a USB keyboard and HDMI display. This setup used SPI for USB communication and a VGA-to-HDMI converter for real-time feedback on an external monitor. Modules like the ball and color mapper with the VGA controller enabled interactive responses to keyboard inputs and highlighted the importance of efficient data flow and timing.

Improvements for next year

The only suggestion we have is to make the to-do list more clear for this lab. We spent a few hours wondering why the ball was not moving, only to realize the frame_clk input of the ball module needed to be set.