Zhihao Wang (zhihaow6), Laurenz Nava (lfnava2)

Introduction

The Microblaze processor can be implemented to run code and process data with the hardware and connections of the FPGA board. In this lab, it was used to run a program to interpret the inputs from a connected USB keyboard received by an on board chip.

Written Description and Diagrams of Microblaze

Summary of Operations

The MAX3421E chip takes in the inputs from the USB keyboard, sends it to the Microblaze microcontroller to control a ball on a VGA screen displayed through an HDMI connection.

Written Description of SV files

Module: mb_usb_hdmi_top.sv
Inputs: Clk, reset_rtl_0, [0:0]
gpio_usb_int_tri_1, 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: Takes in signals from MAX3421E and distributes them to Microblaze, as well as send translated HDMI signals out
Purpose: The top level module to integrate all the submodules, interact with physical inputs and outputs

Module: Hex Driver, hex.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: Convert input data into displayable data, then display on 7 segment display
Purpose: Display input data on 7 segment display

Module: nibble to hex, hex.sv
Inputs: [3:0] nibble
Outputs: [7:0] hex
Description: Turns the 4-bit nibble input into an 8-bit output readable by the hex segments to display the hexadecimal value of the input
Purpose: Convert binary input data into displayable on 7 segment display output data

Module: mb_block.bd
Inputs: clk_100MHz, gpio_usb_int_tri_i, reset_rtl_0, uart_rtl_0_rxd
Outputs: [31:0] gpio_usb_keycode_0_tri_o, gpio_usb_rst_0_tri_o, uart_rtl_0_txd,
usb_spi_miso, usb_spi_mosi, usb_spi_sclk, usb_spi_ss

Description: Interprets signals from MAX3421E, manages master-slave relationship between Microblaze and MAX chip
Purpose: To process the USB keycodes from the MAX3421E to be inputs for the other modules

Module: clk_wiz_0.xci
Inputs: Clk, reset
Outputs: clk1_out, clk2_out, locked
Description: clk1_out is a 25 MHz clock, clk2_out is a 125 MHz clock
Purpose: The 25MHz is for the HDMI pixel clock, the 125 MHz is for the HDMI TMDS

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: See below
Purpose: Simulate the VGA system, with additional outputs to help it convert to HDMI

Module: hdmi_tx_0.xci
Inputs: pix_clkx5, pix_clk_locked, rst, ade, vde, [3:0] aux0_din, [3:0] aux1_din, [3:0] aux2_din, [3:0] blue, [3:0] green, [3:0] red, hsync, vsync,
Outputs: TMDS_CLK_N, TMDS_CLK_P, [2:0] TMDS_DATA_N, [2:0] TMDS_DATA_P
Description: See below
Purpose: Convert VGA signals to displayable HDMI signal

Module: ball.sv
Inputs: Reset, frame_clk, [7:0] keycode
Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS
Description: See below, frame_clk run by vsync so ball center only updates as fast as the image of it can
Purpose: Create a ball that can be manipulated with USB keyboard inputs

Module: Color_Mapper.sv
Inputs: [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] Draw, [9:0] Ball_size
Outputs: [3:0] Red, [3:0] Green, [3:0] Blue
Description: See below
Purpose: Based on the current ball position and X position, assign a color to the pixel

Lab 6.1 I/O

The I/O for lab 6.1 was comparatively simpler. The inputs were from the button and switches, and the outputs were the LED. The accumulator is initialized at and resets to 0. Flipping the switches and pressing the accumulate button adds the unsigned binary value of the switches and displays it on the LED. If the value overflows, a message is sent to the console.

Interactions with the MAX3421E chip

The MAX3421E chip is the slave to the Microblaze master. The job of the MAX chip is to connect the Microblaze with its USB peripherals. The program MAX3421E.c sets the FDUPSPI bit to 1 so the MAX chip will run in full duplex mode, meaning one pin is data-in and another pin is data-out to have 8 bits be written and read in 8 clock cycles.

VGA and Ball operation

VGA is an analog signal to display images on a CRT monitor. It controls the RGB values of each pixel with 12-bits, 4-bits each, in raster order, starting from the top left (0,0), completes a row, then drops to the next row beginning column, and ends at the bottom right (639,479) for a 640 x 480 60 Hz display. This is how the vga_controller operates. In addition to that, it generates a horizontal sync signal for pixels 656-752, a vertical sync signal for lines 490-491, and ensures only the pixels within the 640 x 480 resolution are displayed. The horizontal and vertical syncs are used by the VGA to HDMI converter to ensure the timings of the VGA signals match up with the HDMI ones.

The ball module initializes the ball location in the center. It takes the input keycode and moves the center of the ball in the corresponding direction (W = keycode x1A = up, A = keycode x04 = left, S = keycode x16 = down, D = keycode x07 = right). If the ball is inputted to move into a wall, it will bounce off it, or in our implementation, stop moving into the wall. The color mapper colors the current pixel from the VGA the ball color if it fits in the circle equation centered on the ball's center $(\text{drawX} - \text{BallX})^2 + (\text{drawY} - \text{BallY})^2 <= \text{BallR}^2$, so it the pixel is within the circle (ball) it will be colored. Otherwise, the background is colored according to its X position to give it a gradient.

VGA-HDMI IP

Both VGA and HDMI are types of signals to be displayed by a monitor of some type. The VGA to HDMI converter wraps around the analog VGA signal to transform it into a digital HDMI transition minimized differential signaling (TMDS). The VGA RBG of the current pixel is inputted and outputted, until all the 640 x 480 pixels are complete, then the hsync (for completing a row) or vsync (for completing a screen) is applied to keep the VGA in sync with the HDMI.

Software

Accumulator

In an infinite while loop, the program checks if the clear button is pressed, and if so, sets the display data to 0, with another while loop to act as a debouncer. The program also checks if the accumulate button has been pressed, and if so, checks if the new value Overflows, prints an overflow to console if it does, otherwise it just adds the switch value to current value to display, and another debouncer while loop.

SPI protocol for the MAX3421E

The serial peripheral interface (SPI) allows for communication between different chips, with the Quad SPI specifically designed for high speeds, like in the use of flash memory.

This is a master-slave relationship. Microblaze is the master and the MAX3421E is the slave. Communication is done through 1-bit MISO (master input, slave output), MOSI (master output, slave input), SCLK to keep the SPI synchronous with each other, and SS (slave select), though since the MAX3421E is the only slave, it is always low active. The C code configures the MAX3421E to operate in full duplex mode. It sends and receives 1-bit per clock cycle, and the first byte is the command byte, dictates the register address and either read or write, and the next bytes are data.

C functions

Function: void MAXreg_wr(BYTE reg, BYTE val)
Write a value (val) into a register (reg)
Function: BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)
Write to multiple values (value: data, amount: nbytes) registers (from reg)
Function: BYTE MAXreg_rd(BYTE reg)
Read a register (reg) and return the value
Function: BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)
Read registers (from reg) and return multiple values (BYTE*: data, amount: nbytes)
Note:
1. We need to set the slave register at the beginning of the write or read. Using the XSpi_SetSlaveSelect(...) function, its input is a MASK.
2. We need to use the XSpi_Transfer(...) function to transfer(read/write) data.
3. All first data of the writing buffer should be the command which contains the target register and the option of read/write.
4. When we are reading from the read buffer, the first data is always the status code, we do not need it in this Lab.

Top-Level diagrams

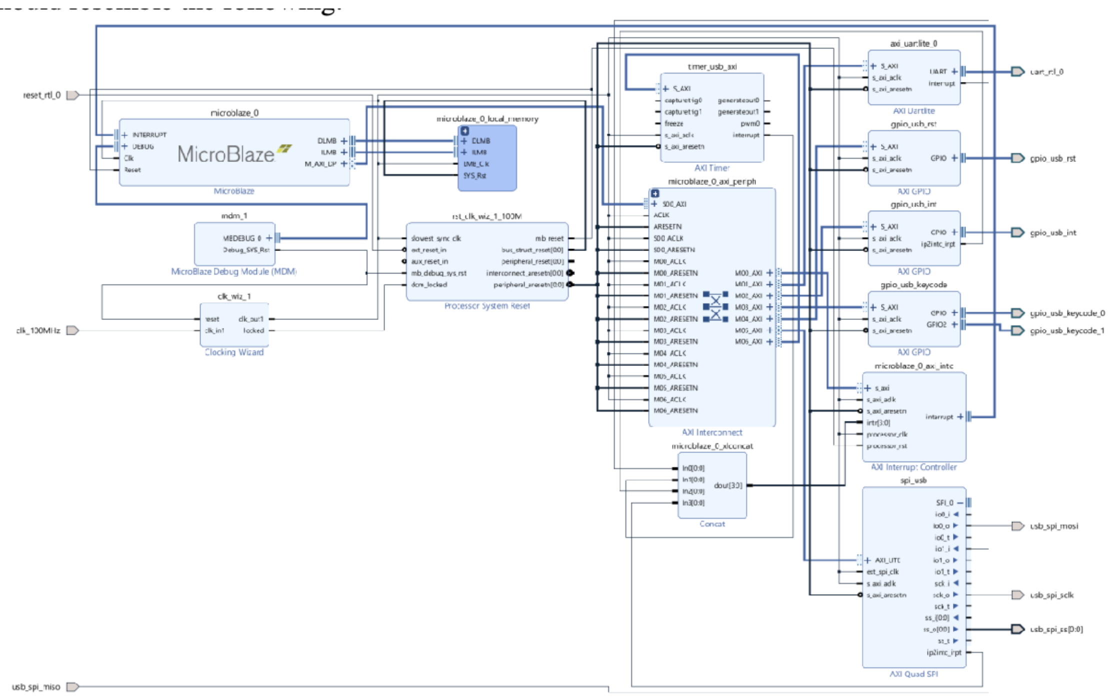
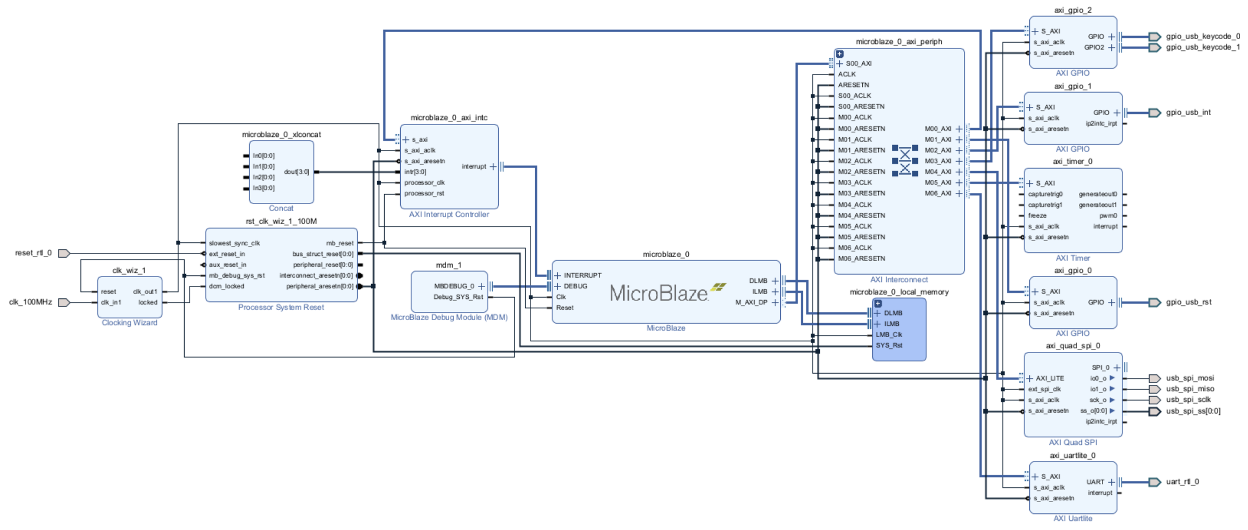
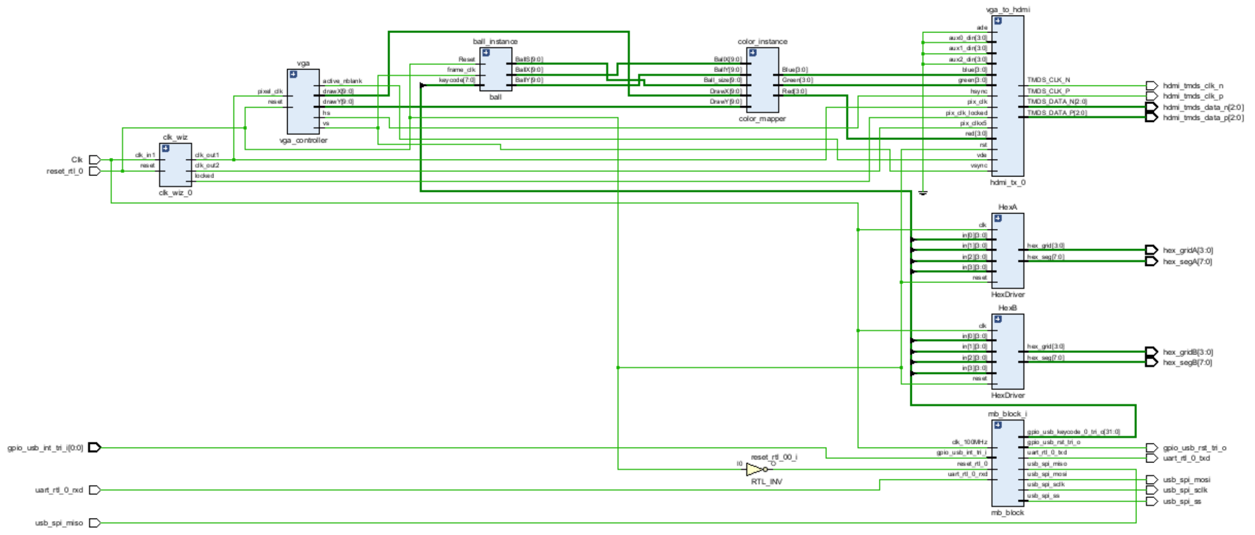


Figure 5 - MicroBlaze with USB Peripherals

From Document



From Block Diagram



Post-lab and INMB Questions

LUT	2860
DSP	9
Memory (BRAM)	8
Flip-Flop	2665
Latches	0
Frequency	123.304 MHz (WNS 1.890 ns)
Static Power	0.075 W
Dynamic Power	0.384 W
Total Power	0.46 W

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

Microcontroller is smallest, then its real time, and largest is application
BRAM - instruction and data caches, DSP48E - Integer and floating point multiplication
Microcontroller 0 BRAM 3 DSP48E, smallest and lightest
Real time 6 BRAM 3 DSP48E , specialized for real time operations
Application 20 BRAM 6 DSP48E , most performant preset, most resources

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

> Modified Harvard, since it has instruction and data caches like a harvard, but can still get instruction from memory like Von Neumann, and in the local memory block, the memory for instruction and data use the same address space, but different buses.

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?

> Data is transmitted without a shared clock signal. Instead, data is sent in discrete packets with a start bit, data bits, optional parity bit, and stop bits. Asynchronous can work between modules with different clock cycles, and require less components than a synchronous equivalent making it simpler, whereas synchronous does not have start and end bits and has timing precision for accuracy.

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.

> Since the UART takes longer to transmit data compared to the CPU, the interrupt is important so other incoming signals don't interfere with the transmission of the UART until it has completed.

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

> Because they are only taking in data, like UART-rxd, or outputting based on the data, like LED and UART-txd.

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)

> Volatile means the variable value may change unexpectedly, typically used for hardware external to the computer. It forces the computer to load from memory location rather than the register, to avoid the compiler doing aggressive optimizations. It does not apply to outside CPU cores.
> Line 30 Logical OR with 1, it sets first bit to 1, and changes nothing else
> Line 33 Logical AND with NOT 1, it sets first bit to 0, and changes nothing else

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

> Text is stuff that goes in flash memory, won't be changed as the program runs
> Data is the variables that can be changed as the program runs, note that the initial data is stored in flash and data stores where to access it in ram and not counted in text
> BSS, or Block Started by Symbol, is uninitialized variables, stored in ram
> Dec is the sum of text, data, and BSS

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.

> The printf statements take up a lot of space since it needs to access the c standard library, which will cause dynamic linking, which will consume lots of time.

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

> 0x4000_0008

Conclusions

> The project functioned as specified. The lab manual and other given materials were sufficient in completing and understanding the lab.