ECE 385

Fall 2023

Experiment 6

System-on-chip (SOC) with Microblaze in SystemVerilog

Cher Rui Tan, Adithya Balaji
Section: Friday 11.30am
TA: Gene Lee

Introduction

We will set up a minimal Microblaze device with an on-chip memory block and a PIO (Parallel I/O) block to blink some LEDs using a C program running on the Microblaze to confirm it is working, and then modify a C program to input data, add as an accumulator and display the data.

The second part of this experiment involved adding a USB host controller, a USB keyboard in this case (MAX3421E) to communicate with the microblaze via SPI protocol and use the keyboard to control a ball that is drawn via VGA. With the finished implementation, a user will be able to hit W,A,S,D on the keyboard to make a ball move up, left, down and right respectively, with the program ensuring the ball does not exceed the bounds of the display monitor.

Written description and diagrams of Microblaze System

Written description of all SV module

Module: mb_usb_hdmi_top.sv:

Inputs & Outputs:

```
input logic Clk,
input logic reset_rtl_0,

//USB signals
input logic [0:0] gpio_usb_int_tri_i,
output logic gpio_usb_rst_tri_o,
input logic usb_spi_miso,
output logic usb_spi_mosi,
output logic usb_spi_sclk,
output logic usb_spi_ss,

//UART
input logic uart_rtl_0_rxd,
output logic uart_rtl_0_txd,

//HDMI
output logic hdmi_tmds_clk_n,
output logic hdmi_tmds_clk_p,
output logic [2:0]hdmi_tmds_data_n,
output logic [2:0]hdmi_tmds_data_p,

//HEX displays
output logic [7:0] hex_segA,
output logic [3:0] hex_gridA,
output logic [7:0] hex_segB,
```

output logic [3:0] hex_gridB

Description: Top level module which instantiates USB signals, HDMI, UART and HEX displays. Does this by instantiating a clocking wizard, VGA sync signal generator, VGA to HDMI converter, as well as the ball and color mapper modules to control the displaying and movement of the ball.

Purpose: Used as a top level to instantiate and link all the relevant modules to one another

Module: hex.sv:

Inputs:	input logic clk,
	input logic reset,
	input logic [3:0] in[4],
Output:	output logic [7:0] hex_seg,
	output logic [3:0] hex_grid

Description: Implements a 4-digit hexadecimal display driver by using a counter to cycle through 4 digits, and selecting the appropriate pattern based on the input nibbles. Also contains a module within for nibble to hex.

Purpose: To display hexadecimal numbers on the 4-digit 7-segment display

Module: nibble_to_hex (.sv)

Inputs:	input logic [3:0] nibble,
Outputs:	output logic [7:0] hex

Description: Takes asynchronous signals and holds them in a flip flop allowing for them to be implemented on a clock signal. This results in the signals being converted from asynchronous to synchronous signals.

Purpose: To eliminate asynchronous components such that the entire circuit remains synchronous to reduce the chance of static hazards arising from timing issues and ensure proper signal handling in FPGA designs.

<u>Module: vga_controller (.sv)</u>

Inputs: pixel_clk,
reset

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

Description: Generates VGA signals to drive display. It manages horizontal and vertical synchronization pulses, provides pixel coordinates for rendering graphics, and controls pixel display. The module uses a 50 MHz clock and includes 10-bit counters to track pixel positions. It also adjusts the timing to match the 25 MHz pixel clock and ensures that only valid pixels are displayed.

Purpose: To output the VGA signal to be used in the display

<u>Module: ball (.sv)</u>

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

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

Description: Controls the motion of ball on VGA display. Handles user input for moving the ball up, down, left, or right, by checking which key (W,A,S,D) is pressed and its hex character. Ball's position is updated based on its current coordinates and boundary limits. Also includes conditionals such that if the ball reaches one of the four sides edges, it will bounce back instead of exiting the screen.

Purpose: Control movement of ball on screen.

<u>Module: mb_intro_top (.sv)</u>

Inputs: input logic clk,
input logic [3:0] btn,
input logic uart_txd,
input logic [15:0] sw);

Outputs: output logic [15:0] led,
output logic uart_rxd,

Description: Interfaces the processor with LEDs, buttons, UART, and switches. Connects signals between MicroBlaze and external peripherals

Purpose: Replacement block diagram wrapper file and top level for use with ECE 385

Written description of modules created for block design

Microblaze

Creates a basic microblaze block diagram with its associated peripherals, with a microcontroller preset which is relevant for this low-level program.

AXI GPIO

Serves as a link between the AXI bus as well as input/output pins on the FPGA, used to read and write data from/to the FPGA. Allocate and maps memory in the Microblaze for different peripherals

AXI Uartlite

Universal asynchronous receiver transmitter, allows for asynchronous UART communication via the AXI bus, handling data transmission and reception. Used for communication with microcontrollers.

AXI Interconnect

Allows for connection and transmitting data between multiple AXI modules, by managing data traffic by handling routing and latency control. Serves to make the system more efficient and flexible.

AXI QUAD SPI

Handles high speed data transfer involving memory mapped access to external devices (in this case, the USB) but with 4 data lines instead of the usual 1. For week 2 we can see that the keycodes were obtained through the SPI then transmitted to the ball for movement decoding.

Clocking wizard
Takes a 100Mhz Clk input and outputs two clocks of 25Mhz and 125Mhz to support the TMDS portion of the VGA to HDMI converter IP.

AXI Interrupt Controller
Converts signals into a single signal to be sent to the processor

Processor Sytem Reset
Allows for enabling or disabling based on user, such as active high

Concat
Feeds concataneted interrupt signals into interrupt controller

Microblaze Debug module
Allows for JTAG based debugging

AXI Timer
Timer to the interface that provides reset and clk signals

Description of IO:

The GPIO acts as a bridge from the Microblaze to the FPGA logic. They are used as data transmitters that map to specific points in memory that can be accessed by software. They can be uni or bi-directional depending on how they are defined and have a specified data bit width but a total of 32 bits. The GPIOs like LED are an output which transmits the overflows bit and Switches GPIO takes in the value of the switches. There is also the Run accumulate GPIO which takes in the value of the push button and the Reset GPIO which takes in the value of the reset button.

Interaction of MicroBlaze with MAX3421 and ball components

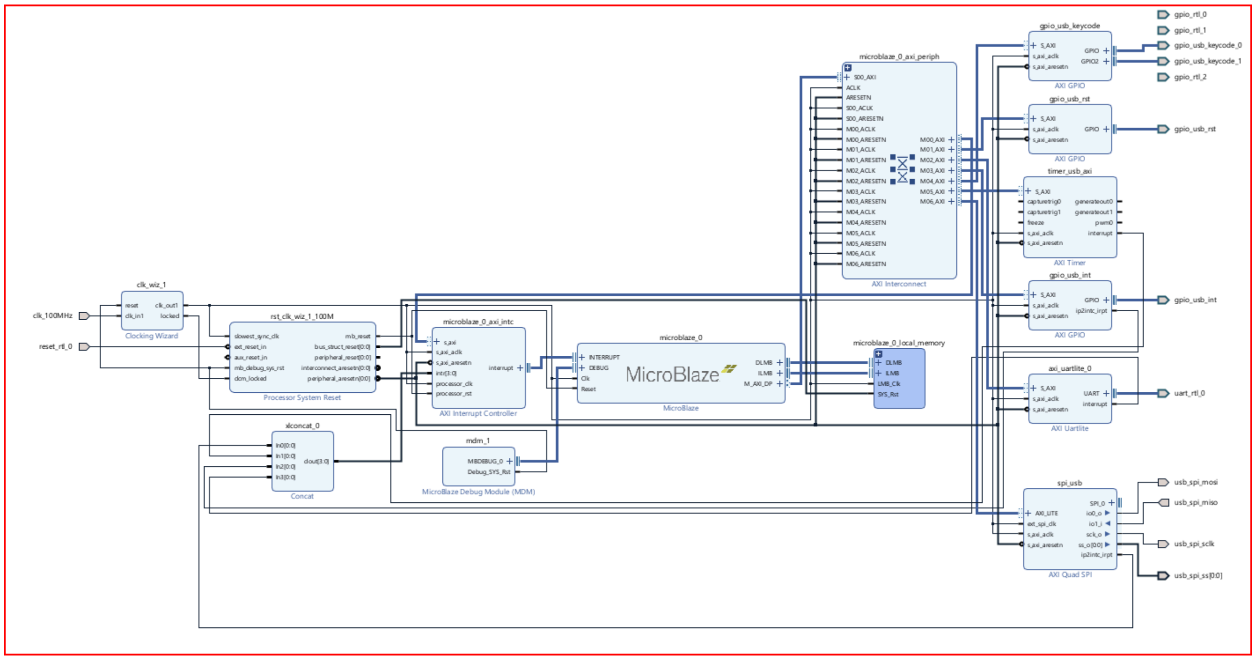
d. Describe in words how the MicroBlaze interacts with both the MAX3421E USB chip
and the ball motion components.

The Microblaze accesses the AXI4 bus which is a memory mapped interface. This much like MEM2IO, intercepts data from specific locations and sends them to peripherals and receives data and puts them in specific memory locations. The Quad SPI takes the keycode values and passes them to the memory locations where the Microblaze is able to access and decodes them before inputting them to the ball motion sv modules, which result in a different type of motion depending on the keycodes.

Interaction of VGA controller with Ball and Color mapper

VGA controller is the main file for the drawing of pixels on the screen, it handles the total number of bits on the screen and the drawing process with DrawX and DrawY as outputs. These values are then passed to the colormapper.sv module. The ball.sv module purely defines how the ball moves and interacts with the rest of the screen, including edge detection and keycode decoding for extra movement with the keyboard. This information is then passed to colormapper.sv where the RGB values are determined based on if the pixel is within the ball or not within the ball.

Top Level Block Diagram



<u>Figure 1: RTL Block diagram</u>

Describe in words the software component of the lab

a. One of the INQ questions asks about the blinker code, but you must also describe your accumulator.

Mb_blink.c handles the logic for accumulation as well as turning on and off the LED. It assigns a volatile pointer to the base address of the GPIO corresponding to the LEDs, with the volatile keyword indicating that the object is a memory mapped hardware object, which can be changed anytime so the compiler should not optimize code. Volatile pointers to the base address of the GPIOs corresponding to the switches and buttons. To implement the accumulator, we have a while loop that is infinitely true, and a flag that is assigned for checking if the button is being pressed. This flag is set to high when the pointer to the run button is equal to zero, so that eventually when the pointer to the run button is high, the value at the pointer to the LEDs and the switches are added. There is also a check to ensure there is no overflow, in which case an error message is printed. Following a successful addition, the flag is set back to 0 for the next cycle of the run being pressed.

b. Written description of the SPI protocol and how it operates in the context of the MAX3421E + Describe the purpose of each function you filled in in the C code

For 6.2, we had low-level C code to implement the SPI protocol. At a broad level, this involves a synchronous serial communication protocol between the master device (in this case, the CPU) and the slave device (MAX3421E). There are 4 key signals that this protocol works on, which are the SClk, MOSI, MISO as well as the CS (Chip select). Data is transmitted one bit at a time over multiple clock cycles.

For both reading and writing, the MAX3421E waits for the CS signal to be lowered for a command to be passed to the slave. Based on the !R/W signal in the 8 bit string, if reading the master sends a command with the register being read from, and the data is returned from the slave (in which case the MISO is high), and if writing the master sends a command to write followed by as many 8-bit long data strings as it wants to write until the CS goes back to high.

The functions we modified and their purposes are as follows:

<u>void MAXreg_wr(BYTE reg, BYTE val):</u>
Used for single host register writing. Writes the value of the register being written two which is offset by 2 to set the write bit to 1 subsequently. The value is written via the SPI and the return code is read to check if successful, before deselecting the MAX3421E

<u>BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)</u>
Used for multiple byte writing to different registers. The same algorithm as single host writing, but the main difference is having an array to store data of size (nbytes+1) to write. Additionally, returns a pointer to the memory location after the last written data.

<u>BYTE MAXreg_rd(BYTE reg)</u>
Used for single host register reading. Create a BYTE array to store the register that it wants to read from and send this information through the XSpi_transfer function, and the data being read is also assigned into another BYTE array, with an error message printed if unsuccessful, before deselecting the MAX3241E

<u>BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)</u>

Used for multiple byte reading. Similar algorithm as single host reading, but uses a BYTE array of size (nbytes + 1) to send in the register values being read from as well as an output BYTE array of size (nbytes + 1) to support the multiple data bytes being read, which are subsequently reassigned back to the BYTE* data, skipping the first bit of the output BYTE array which represents the register data.

Answers to all INQ & Post lab questions

1) Select the “Microcontroller” Preset and then modify the “Local Memory” to 32KB (see Figure) You should do some research and figure out what are some primary differences between the various presets which are available.

   The microcontroller preset is used for bare metal, low-level code. The real time processor preset is used for lasks with low latency responses or strict timing requirements. The application processor is more suitable for multitasking and running programs that do not require real time constraints.

2) . Note 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 as it allows instructions and data to be fetched from the same physical memory (unlike a pure Harvard that has two separate memory spaces for this. It is also not a Von Neumann as it has separate instruction and data caches.

3) What does the “asynchronous” in UART refer to regarding the data transmission method? What are some advantages and disadvantages of an asynchronous protocol vs. a synchronous protocol?

   The UART interface does not use a clock signal to synchronize the transmitter and receiver devices; it transmits data asynchronously. Instead of a clock signal, the transmitter generates a bitstream based on its clock signal while the receiver is using its internal clock signal to sample the incoming data. This allows the benefit of needing no synchronization across the devices, allowing for a different clock speed between the Microblaze and its peripherals. This however might result in data discrepancies if the baud rate is not aligned resulting in corrupted data. This was an error we experienced.

4) You should have learned about interrupts in ECE 220, and it is obvious why interrupts are useful for inputs. However, even devices which transmit data benefit from interrupts; explain why. Hint: the UART takes a long time (relative to the CPU) to transmit a single byte.

Interrupts are useful for receiving devices as it allows for the CPU to receive data without constantly polling, freeing up the CPU to perform other tasks more efficiently.

The transmitting device, or in this case the UART which passes through the data from the sending to receiving device, because the UART handles less signals for interrupt than polling, since the UART takes a long time relative to the CPU.

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

The UART and LED are both time independent signals which are only accessed when the CPU deems it necessary; this allows there to be no need for it to be a synchronous input to the system.

6) 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 program creates a blinking LED by taking the pointer to the address of the LEDs and ANDing it with x00000000 or x00000001. The volatile keyword means that the pointer points to a memory-mapped hardware register that can change without the compiler's knowledge, so the compiler should not be allowed to optimize read/write operations for this memory locations

7) 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 contains the program's executable code, such as functions and logic. The data segment stores the initialized global and static variables with stated initial values from the source code. The BSS segment stores uninitialized or initialized variables that are not assigned a value.

8) 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 line of code that takes up a significantly large memory is the xil_printf. This is because the xil_printf is a modified version of the printf, but customized to work with Xilinx's specific hardware, with a different library from the standard C libraries, and this library contains significantly more functions/drivers/optimization to support and facilitate the low-level hardware interactions, unlike a standard printf.

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

Access the base address by mapping the pointer either directly to the base address (ie 0x40010000) or by static casting the memory location GPIO2_DATA located in XParameters.h and assigning it as a pointer.

Design Resources and Statistics

The frequency is calculated by doing 1/(10ns-wns);

LUT	2788
DSP	9
Memory (BRAM)	8
Flip-Flop	2592
Latches*	0
Frequency	0.123Mhz
Static Power	0.075W
Dynamic Power	0.383W
Total Power	0.458W

Conclusion

The design that we implement managed to work successfully for both parts of the lab, including blinking an LED as well as an accumulator. As an end result, we were also able to control the movement of the ball with the WASD keys on a USB keyboard while ensuring that the ball does not move out of the bounds of the screen.

In conclusion, this lab allowed us to understand the integration of hardware on Vivado with Software on Vitis to create a program with user I/O, as well as an introduction to the Microblaze processor on Vivado. For future experimentation, one possibility is to look into integrating controls for multiple keys being pressed at once, to make the ball move diagonally, or to extend this to a USB mouse instead of a keyboard, where additional logic would be needed to handle the fact that a mouse only has two buttons and we would need it to work for more than 2 directions.

We did feel there was a small amount of ambiguity in the guide as to which Vivado files we actually needed to edit for the ball movement portion of the lab, as we needed to understand this better from the TAs, thus it would be better if it was more explicitly stated (unless that was the point where we are less being told what to do and expected to figure it out ourselves.)