**ECE 385**

Spring 2023

Experiment #6

**SOC With NIOS II In SystemVerilog**

Noah DuVal, Jason Wright

DJ (JZ) / Friday 3:00 pm

Dohun Jeong

**Introduction**

This week's experiment involved working with the NIOS-II processor running on the MAX10 FPGA. The NIOS-II is a 32-bit RISC processor that is highly configurable, making it suitable for a wide range of applications. In this experiment, we utilized the NIOS-II processor to develop a USB/VGA interface that allows for communication between a computer and the FPGA. This interface allows the MAX10 FPGA to communicate with USB and VGA devices, enabling users to interact with the system through a graphical user interface (GUI) on a monitor. The interface operates by using a USB controller and VGA controller, which provide the necessary protocols and signals for USB and VGA communication. Together, these controllers enable the MAX10 FPGA to transfer data to and from USB and VGA devices, making it a useful feature for applications that require user interaction or data visualization.

**Written Description**

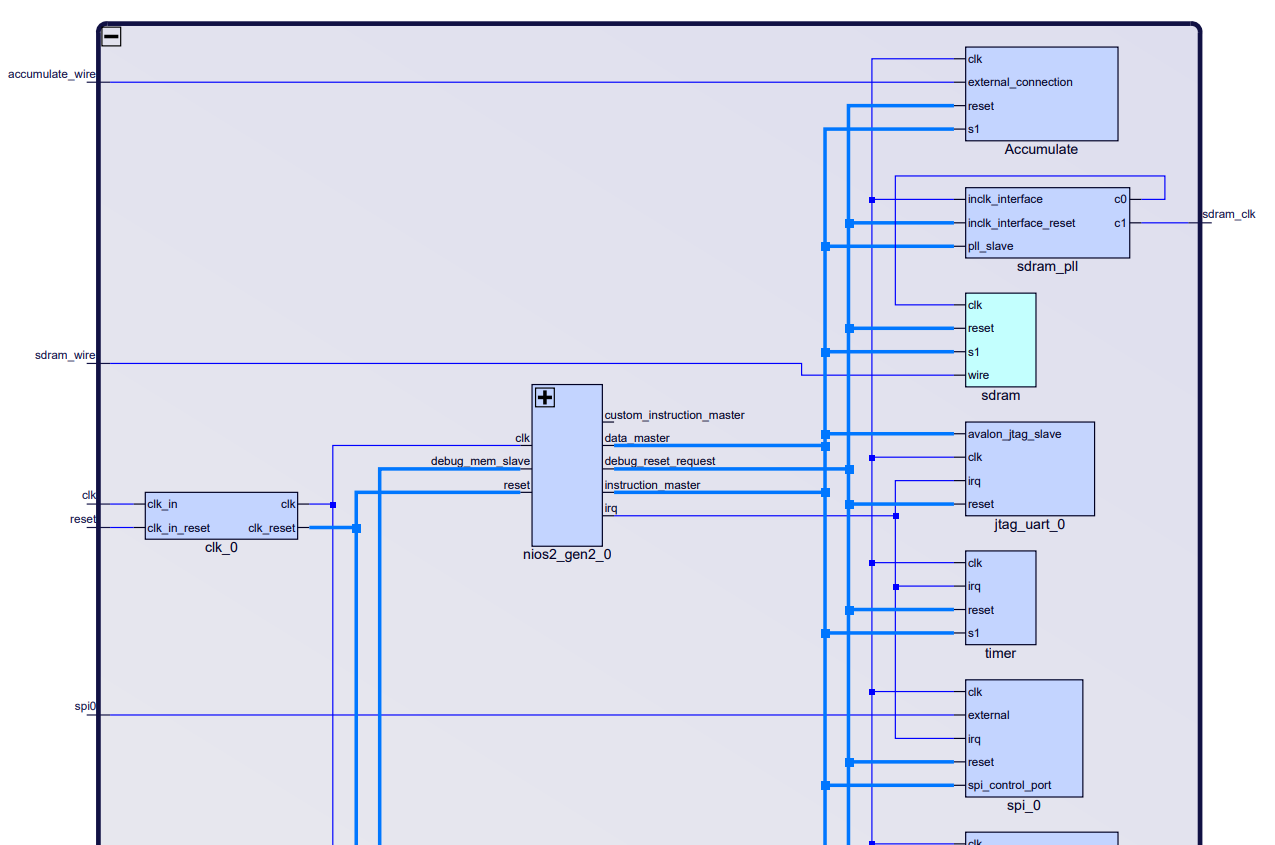
During the development of 6.1 the I/O portion was implemented using a PIO block in which connected the LED’s to the C program that held the accumulator. The user had control over a set of switches and two buttons. One being accumulated to add the binary value set on the switches to the running total. The other button is a reset, which in turn sets the count back to zero. If the system ever hit overflow it would circle back to 0 and add on the leftovers.

When it came to week 2 we started working with the NIOS and its interactions with the MAX3421E USB chip and the VGA components. The connection to the USB chip relied on four key functions for reading and writing both a single bit and multiple bytes. These were all interconnected using SPI protocols. Furthermore, the NOIS sends these signals to the top level, ball module, color mapper, and VGA controller where the RGB values and screen placement is set using the vertical and horizontal syncs.

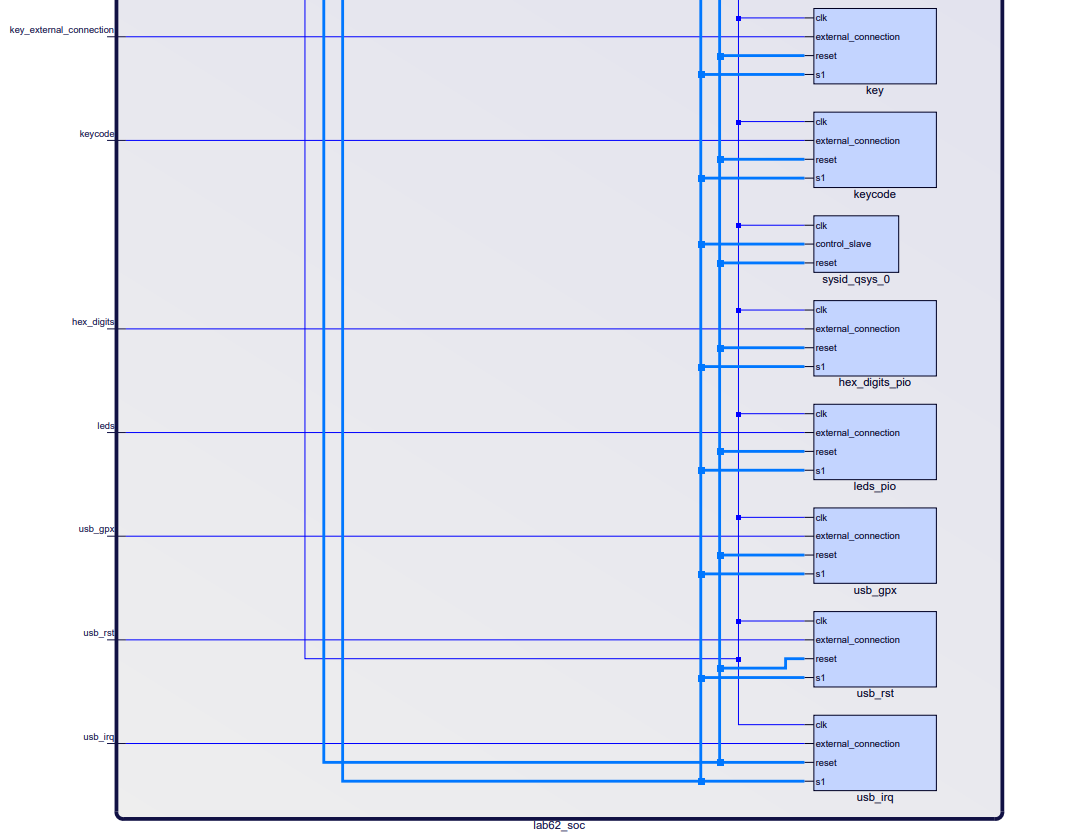
Those four functions include MAXreg\_wr, MAXbytes\_wr, MAXreg\_rd, and MAXbytes\_rd. The two starting as MAXreg are to write and read a single byte from a register. Where the other two MAXbytes are to read and write multiple bytes at the same time.

The Serial Port Interface (SPI) protocol that was used to make this week's lab work consists of multiple peripherals in forms of Slave Select (SS), Master in Slave Out (MISO), Master Out Slave In (MOSI), and the SCLK. The Slave Select is used to choose which peripheral the MAX10 is going to be interacting with. While the MISO and MOSI signals are used to decide whether to read or write and where to do so. All while the SCLK is making sure that the operations take place at a stable timeline.

The last piece to the puzzle was the VGA modules. These included the VGA Controller, color mapper, and the ball. The VGA Controller being the most important as it used a horizontal and vertical sync to control where the “electron gun” is supposed to point on the screen giving us the values of DrawX and DrawY. Which were used in the other two modules to tell the controller where the ball should be on the screen and what color each pixel should be based on where the “gun” is pointing to.

**Hardware **

**Figure 1: Platform Designer View Part 1**

****

**Figure 2: Platform Designer View Part 2**

Component: clk\_0

Description: This component simply maps the on-board generated clock to the other hardware components defined in system verilog.

Component: nios2\_gen2\_0

Description: This component is the interface between the hardware and the nios2 software CPU. It allows changes in the c code run in nios2 to affect the hardware system and vice versa through its data connections.

Component: Accumulate

Description: This is an I/O component needed for the on-board accumulate key to be read by the software. It is only necessary for week 1, but was not removed for week 2.

Component: sdram\_pll

Description: This pll defines the connections to a system of sdram that can now be accessed by both the hardware and software and generates the clk used by the sdram.

Component: sdram

Description: This is a memory file accessible by the defined hardware and the nios2 software.

Component: jtag\_uart\_0

Description: This component is the interface of the jtag system that connections the user’s computer to the FPGA board through the USB cable.

Component: timer

Description: This component is necessary for the various interrupt signals to function correctly when multiple are received before the first can be resolved.

Component: spi\_0

Description: This is the driver for the USB host code. It allows the port to serially exchange data both ways through the USB between the keyboard and the system.

Component: key

Description: This component is also used only for week 1. It is the key used as the reset for week 1 and was left for week 2 without purpose.

Component: keycode

Description: This PIO is where keypresses from the keyboard are input to the system.

Component: hex\_digits\_pio

Description: This module is needed for the NIOS II software to be able to set values for the hex displays. It provides a connection path between the cpu and the on-board hardware.

Component: leds\_pio

Description: Similar to the previous module, this provides a connection path so that the software can set values for the LEDs.

Component: usb\_gpx

Description: This is the path for the USB’s gpx signal (output from hardware).

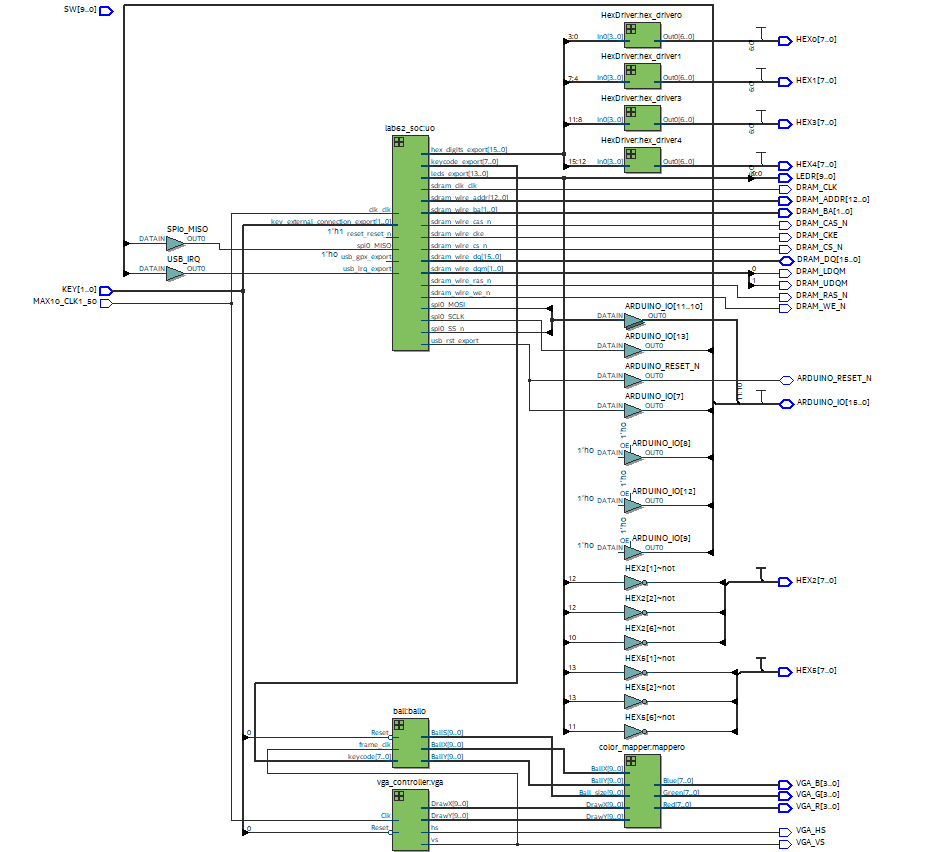
Component: usb\_rst

Description: This is the path for the USB’s reset signal (output to hardware).

Component: usb\_irq

Description: This is the path for the USB’s interrupt signal (output from hardware).

**Module Descriptions**

****

**Figure 3: RTL View Top Level**

Module: vga\_controller.sv

Inputs: Clk, Reset

Outputs: hs, vs, pixel\_clk, blank, sync, [9:0] DrawX, DrawY

Description: This module uses hardcoded values for the number of pixels (800x525) to generate horizontal and vertical sync signals. It also creates a clock half the frequency of the system clock to use for this purpose. This is accomplished by simply incrementing vertical and horizontal pixel counts and comparing the count to the known value of the number of pixels in each row and the number of rows, hs is set high at the end of a row, and vs is set high at the end of the entire screen.

Purpose: This module is needed to generate the pixel clock and sync signals needed for the other modules to understand what part of the display they are interacting with. The vertical and horizontal syncs are also needed to output through the VGA monitor so that the display knows where to place pixels.

Module: lab62.sv

Inputs: MAX10\_CLK1\_50, [1:0] KEY, [9:0] SW, [15:0] DRAM\_DQ

Outputs: [9:0] LEDR, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5; DRAM\_CLK, DRAM\_CKE, [12:0] DRAM\_ADDR; [1:0] DRAM\_BA; DRAM\_LDQM, DRAM\_UDQM, DRAM\_CS\_N, DRAM\_WE\_N, DRAM\_CAS\_N, DRAM\_RAS\_N, VGA\_HS, VGA\_VS, [3:0] VGA\_R, VGA\_G, VGA\_B

Description: This is the top-level module for the system. It contains several interconnections and instantiates all other modules of the system.

Purpose: This module defines how the system is put together at the highest level. It is necessary for the compiler to properly place and connect the other modules together.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module is a single unique case statement that outputs 7 bits of data readable by the hex displays based on the 4 bit input.

Purpose: This allows us to display the number 0-9 and letters a-f needed to show information on the on-board hex displays.

Module: Color\_Mapper.sv

Inputs: [9:0] BallX, BallY, DrawX, DrawY, Ball\_size

Outputs: [7:0] Red, Green, Blue

Description: This module contains simple logic that takes in the ball’s current position and the desired size of the ball and determines which pixels make up the ball. It makes use of the circle equation x^2 + y^2 = z^2 to easily determine a circular pattern of pixels. The pixels it determines are the ball are then assigned the ball’s orange color. The other pixels are made blue with a gradient effect along the x axis.

Purpose: This module is needed to simply determine which pixels should be assigned which colors based on the ball’s position only.

Module: ball.sv

Inputs: Reset, frame\_clk, [7:0] keycode

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

Description: This module makes use of known parameters (such as the screen borders and ball size) along with the frame\_clk input to determine where the ball should be placed in the next frame. The logic for this is based on two motion vectors, one for each axis, that are set on a keyboard press or reversed on a bounce. The new position can then be determined by adding the motion vector value to the position for each axis.

Purpose: This module contains the game logic that allows the ball to bounce and move. It also maps the keyboard inputs to new motion vector values in order to change direction.

**INQ And Post-Lab Questions**

*What are the differences between the Nios II/e and Nios II/f CPUs?*

Nios II/e is resource optimized while Nios II/f is performance optimized. This means that the Nios II/f has more options when defining hardware, such as using hardware multiply/divide components, tightly coupled-masters, shadow register sets, and more. This makes it more complicated, but worth using if you value ultra-optimized performance.

*What advantage might on-chip memory have for program execution?*

On-chip memory has a performance advantage when the data needs to interact with the hardware because it does not need to be transferred over the USB connection. When valuing performance, this is preferable to executing NIOS II programs from the DRAM.

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

It is a modified Harvard machine because its physical address is spread over many locations. This allows the contents of the instruction memory to be accessed as data. Von neumann machines in contrast only have one physical address while pure harvards only have two.

*Why might this be the case?*

The LED block only needs to display the data sent to it, but the on-chip memory has to be read from and written to, so it needs access to the program bus.

*Why does SDRAM require constant refreshing?*

SDRAM is a system of capacitors used to store data. They must be constantly refreshed because the capacitors are constantly losing charge and the refreshing allows them to stay at the correct values.

*What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?*

NIOS II starts its execution at the beginning of SDRAM, which is 0x0800\_0000 for us. This step is done so the program knows what location to return to after a reset or exception signal.

*Look at the various segments (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code:*

.text is where strings are stored: char z = “anything”;

.bss is where uninitialized data is stored: int z;

.heap is where allocated memory is stored: int pointer = (int)malloc(sizeof(int));

.rodata is a region for read-only data is stored: const int z = 1;

.rwdata is for read/write data to be stored: int z = 1;

.stack is used to store function calls: z = int f(int x, char b);

**Design Statistics**

| **LUT** | 4309 |
| --- | --- |
| **DSP** | 4 |
| **Memory (BRAM)** | 11,264 bits |
| **Flip-Flop** | 2488 |
| **Frequency** | 71.48 MHz |
| **Static Power** | 96.52 mW |
| **Dynamic Power** | 61.22 mW |
| **Total Power** | 178.85 mW |

**Table 1: Design Statistics**

**Conclusion**

Upon completing this week's experiment, everything worked as it should have and the process was smooth going for the most part. There were not too many problems that we had run into this week. However, we did have to do a little catch up on our C programming as there were a few minor mistakes made with using the pointers and address as we forgot how to properly use them. Although, after remembering how to properly address pointers we were able to complete the lab without any other major errors outside of a clock mixup. Other than that the instructions were somewhat straightforward. However, it is definitely starting to get noticeable that we are receiving less and less guidance in starting these labs which makes starting the lab a little more time intensive as it takes a little longer to find the direction you are supposed to take. As well as it leads to you forgetting about a lot more of the little stuff. Especially the thing that you do not know or remember very well.