ECE 385

Spring 2023

Lab 6

SOC with NIOS II in SystemVerilog

Gustavo Fonseca, Hunter Baisden

TA: Hongshuo Zhang

Introduction

In this lab, we used the NIOS-II processor. The main functionality of the processor is to perform low performance tasks that do not require the use of hardware. The processor has a C compiler to perform the basic tasks of a microprocessor.

The USB and VGA interface connects the keyboard inputs to the VGA output. The USB Keyboard connects to the MAX3421E, which interfaces with the FPGA that controls the VGA output.

Hardware Component

I/O

1. **Inputs:**
    a. **KEY[0]**
    Used as the hardware reset for the accumulator
    b. **KEY[1]**
    The button that causes the system to accumulate the value on the switches to the LED
    c. **MAX10_CLK1_50**
    The 50MHz clock that controls the system
    d. **LEDR**
    Controls the LED display to display the accumulator
    e. **SW**
    Switches that control the accumulation

    For more detailed inputs and outputs, refer to the System Level Block Diagram section, at the bottom of which has the overall Platform Designer.

NIOS Interactions

1. **MAX3421E USB**
   This connection connects the keyboard to NIOS II. There are 4 pins in a USB: VDD, D-, D+, GND. The data is transmitted through the D- and D+ pins. In order for the USB device to be enumerated, there are four steps to be performed. Firstly, the MAX3421E chip must be initialized, which is done by the MAX3421E.c program. Secondly, the USB must be reset. Thirdly, the USB address must be assigned. Finally, the device descriptor is gotten. The last three steps are performed by the transfer.c file.
2. **VGA**
   This connection outputs the data from the NIOS II to the monitor. The VGA output has 5 key outputs: Red, Green, Blue, Horizontal Sync (HS), and Vertical Sync (VS). The values from Red, Green, and Blue (RGB) define what color will be output to the selected pixels. The Horizontal Sync is a pulse that tells the screen which pixels on the screen should be colored with the selected RGB value. The Vertical Sync is a pulse that tells the

screen which Lines on the screen should be colored with the selected RGB value. For every value of HS and VS that match, the values defined by RGB are drawn on the pixels defined by those values on the screen.

SPI Protocol

The SPI Protocol is made up of 4 signals: CLK, MOSI, MISO, and SS. The CLK is sent from the Master device, in this case NIOS II, to the be used as the clock for the slave device, MAX3421E. MOSI, Master-Out Slave-In, is used for sending data from NIOS II to the MAX3421E, using the CLK signal as a sync. MISO, Master-In Slave-Out, is used for sending from MAX3421E to NIOS II, again using the CLK signal as a sync.

C Code Functionality

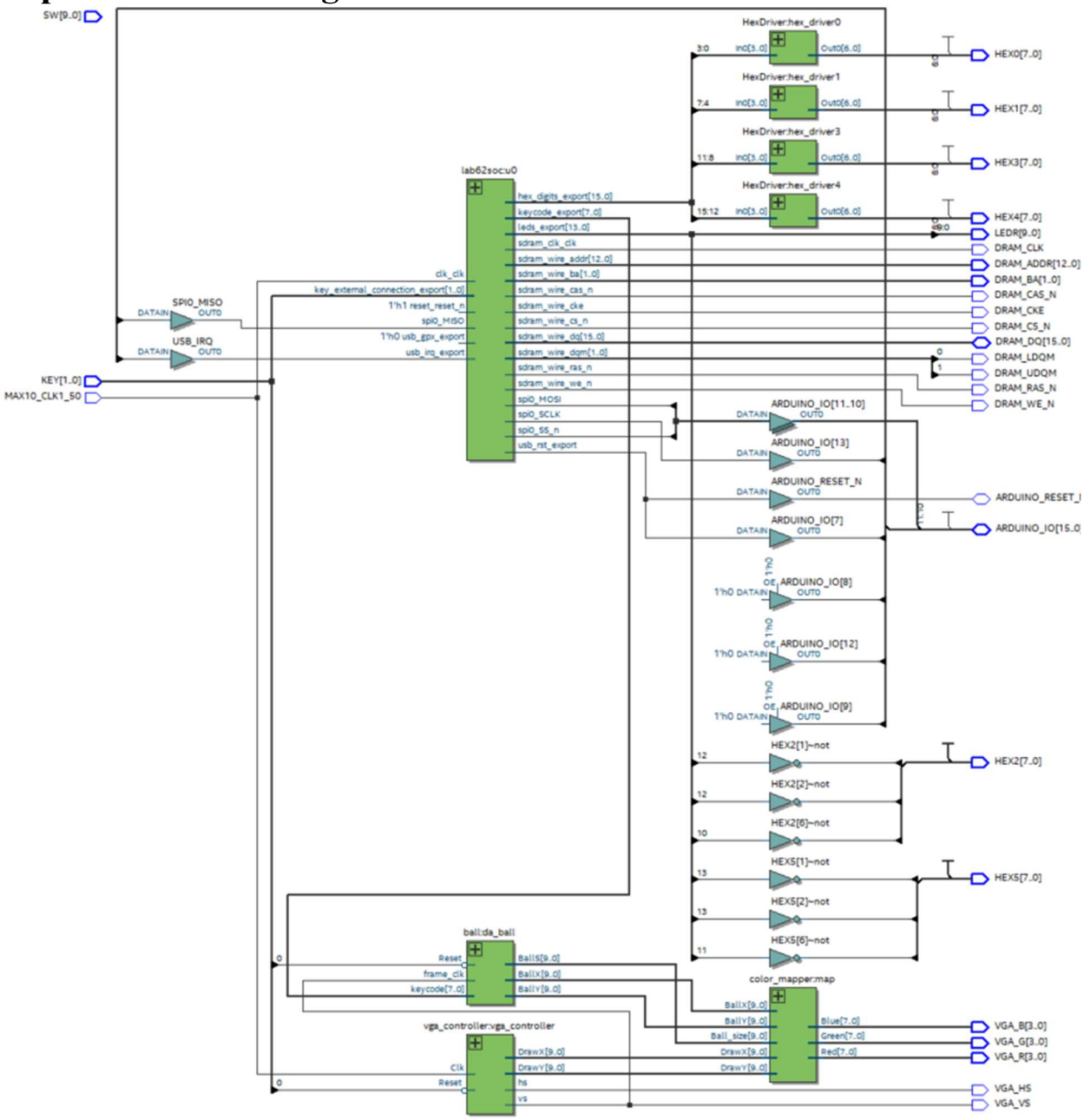
1. **MAXreg_wr**
   The purpose of this function is to write a single byte into a register within the MAX3421E through the SPI Protocol. We select the desired register and value to write into it based on the parameters. We then use the alt_avalon_spi_command function to connect and write to the selected register.
2. **MAXbytes_wr**
   The purpose of this function is to write multiple bytes into a register within the MAX3421E through the SPI Protocol. We select the desired register, the number of bytes, and the actual data to write into the register based on the parameters. We again use the alt_avalon_spi_command function to connect and write into the selected register. The function also returns the pointer of the last byte stored into the register.
3. **MAXreg_rd**
   The purpose of this function is to read a single byte from a register within the MAX3421E through the SPI Protocol. We then select the desired register based on the parameters. We then use the alt_avalon_spi_command function to connect and read from the selected register and save its value. The function will then output this saved value.
4. **MAXbytes_rd**
   The purpose of this function is to read multiple bytes from a register within the MAX3421E through the SPI Protocol. We select the desired register, the number of bytes, and the location where the read data will be stored based on the parameters. We then use the alt_avalon_spi_command function to connect and read from the selected register and save its value. The function will then return the pointer of the last byte stored into memory.

VGA Operation

The VGA works by using an electron beam to paint each pixel, starting from the left side of the monitor until the bottom right side. There are 640 horizonal pixels and 480 vertical lines. The VGA needs three main signals: HS, VS, and RGB.

The main interaction between the vga_controller, ball, and color_mapper modules involve checking if the current pixel should be colored as the ball or the background. To start, the vga_controller orients the system to what is the current pixel we are modifying, sending the current HS and VS to the VGA as well. The ball module then takes in the current keycode from the keyboard, it then outputs the current position of the ball. The color_mapper then checks if the current pixel matches with the ball's current position and desired radius. If they match, then the current pixel will be colored with selected ball's color, if not, the pixel will be colored with the selected background color. Finally, the color_mapper outputs the current RGB values to the VGA.

Top Level Block Diagram



Above is a top-level diagram of the full lab.

.SV Modules

Module: lab62 (in lab62.sv)

Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW

Outputs: [9:0] LEDR, [7:0] HEX (0-5), 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

Inouts: [15:0] DRAM_DQ, ARDUINO_IO, ARDUINO_RESET_N

Description: This is the top-level module for this project. It declares the 4 submodules, as well as the SOC. These include vga_controller, ball, color_mapper, and HexDriver. It takes inputs for the DRAM to be used by the SOC

Purpose: This module serves as the top level for all the interactions with the FPGA. It is used as the top level in compilation when we program the FPGA. A diagram for this function can be seen above.

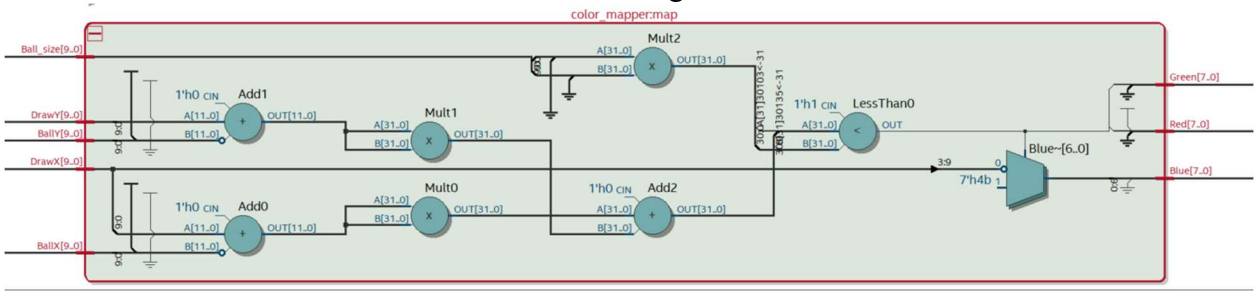
Module: color_mapper (in Color_Mapper.sv)

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

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

Description: This module takes in all the information about the ball generated from the ball module, as well as the draw information from the vga_controller. It then combines this information to output the correct values of the red, green, and blue to the screen.

Purpose: This module tells the VGA what to output given the location of the draw functions and the ball. Without this module, no colors would change.



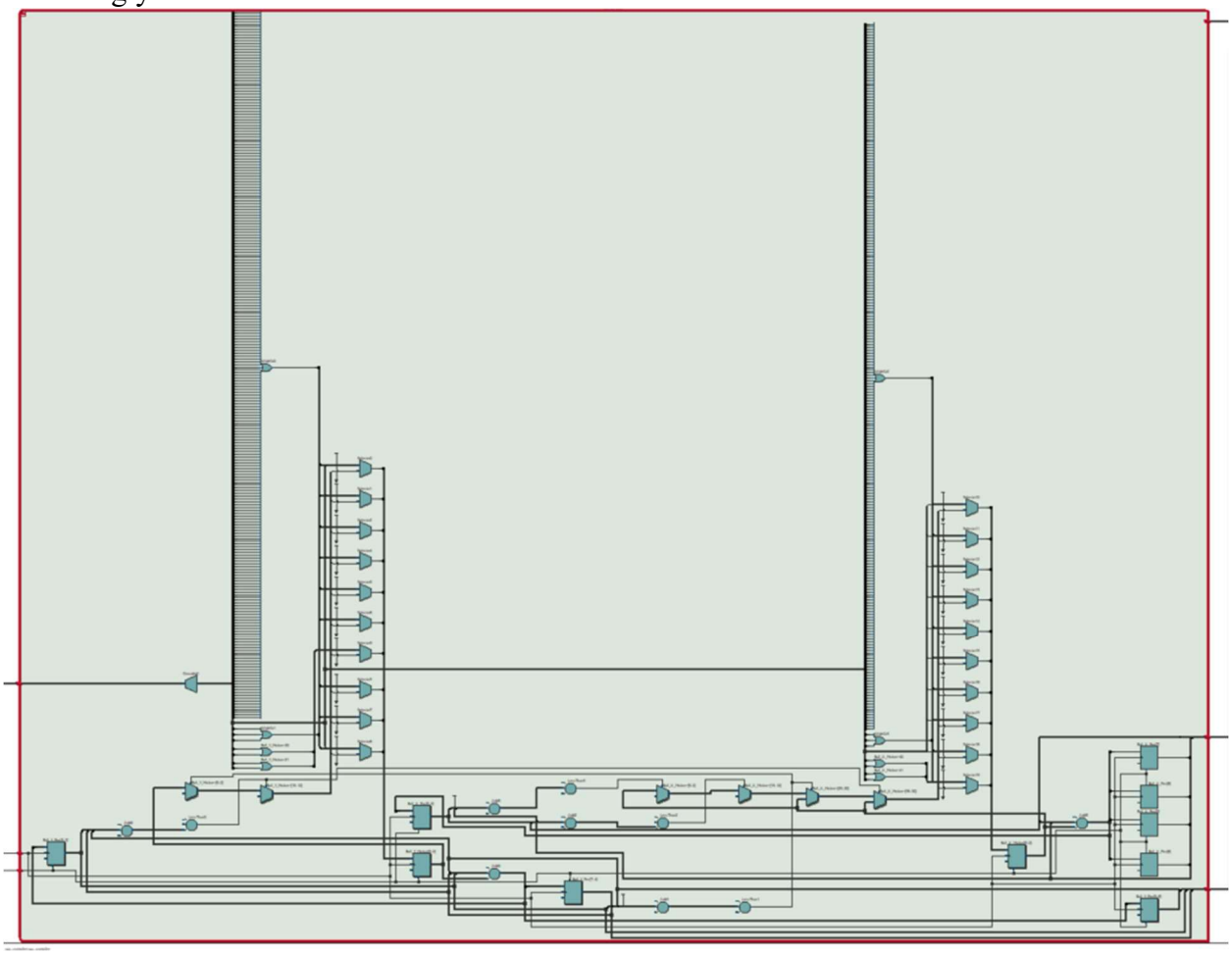
Module: ball (in ball.sv)

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

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

Description: This module controls the motion of the ball on the timing of the frame_clk. The given keycode tells the ball which direction it should move to if given an input. This module then outputs the corresponding x and y coordinates as well as the balls size to be used in the color mapper.

Purpose: This module is imperative for making the ball move along the screen and respond accordingly to the sides of the screen.



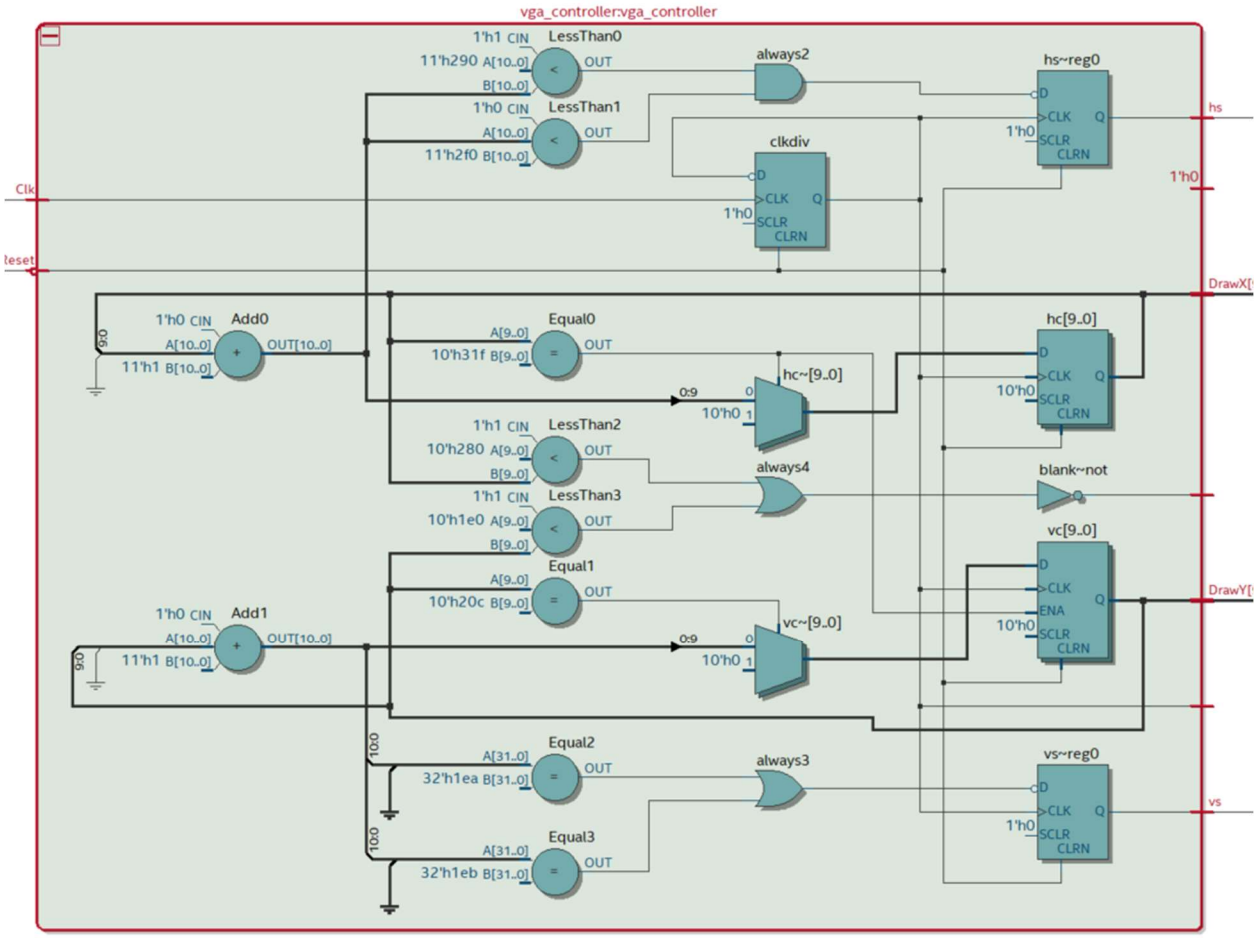
Module: vga_controller (in vga_controller.sv)

Inputs: Clk, Reset

Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY

Description: This module controls the timings and locations on the vga output itself. It simply takes in the Clk and Reset and outputs the vsync and hsync, as well as the pixel_clk, sync, blank, and the coordinates DrawX and DrawY.

Purpose: This module is important to making sure the vga output it properly timed and also not blurring and displaying incorrect pixels.



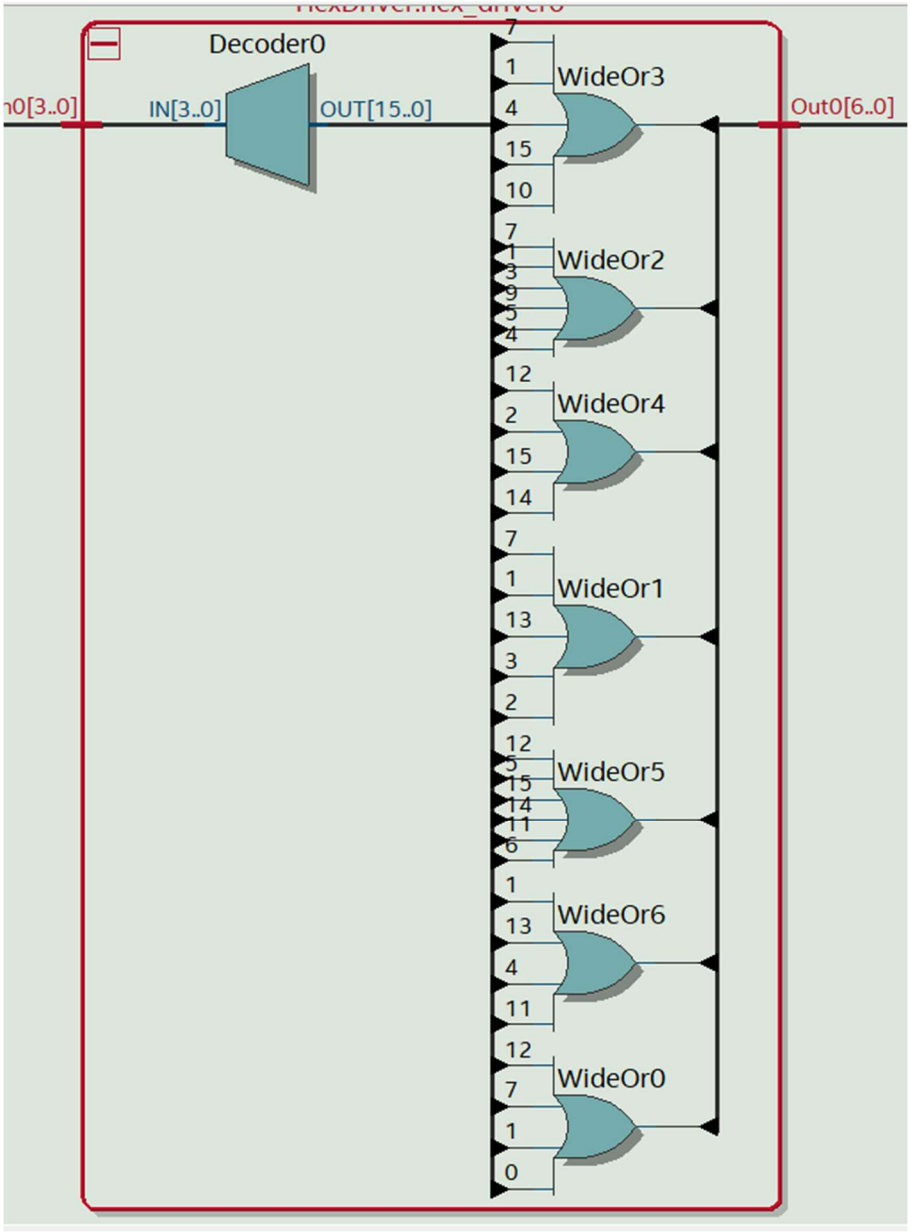
Module: HexDriver (in HexDriver.sv)

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module pairs every unique case of 4-bits (16 in total) into a 7-bit value used to illuminate the hex displays.

Purpose: This was used to output the values we were working with to the hex displays, both for debugging and for the demo itself.



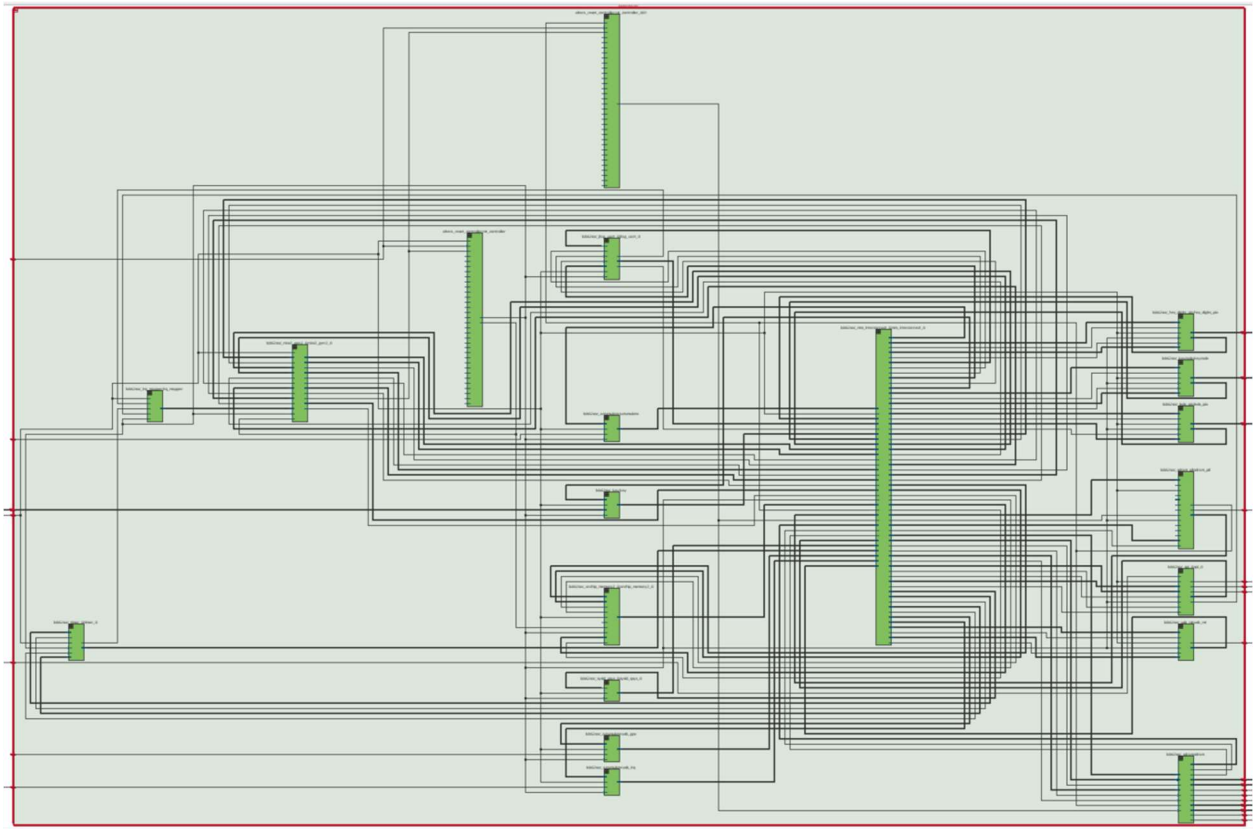
Modules: lab62soc (in lab62soc.qip)

Description: Below I will describe the PIO blocks added in addition to the ones needed for the NIOS II processor in order to complete the platform designer.

Purpose: All of these modules come together to make a working NIOS II processor with memory and USB abilities.

System Level Block Diagram

If not explicitly mentioned, the SOC component was used in both parts of the lab. An overview of the SOC is shown below:



The rest of this section goes into further detail about the different IP's within the platform designer, generated within the SOC.

Module: nios2_gen2_0 (in lab62soc.qip)

Connections: clk, reset, data_master, instruction_master, irq, debug_reset, debug_mem_slave

Exports: None

Description: This is the central processor that everything is connected to. It is a NIOS II Processor from Intel.

Purpose: This module is what everything else is connect to. The block diagram for this module is far to large to fit into a page.

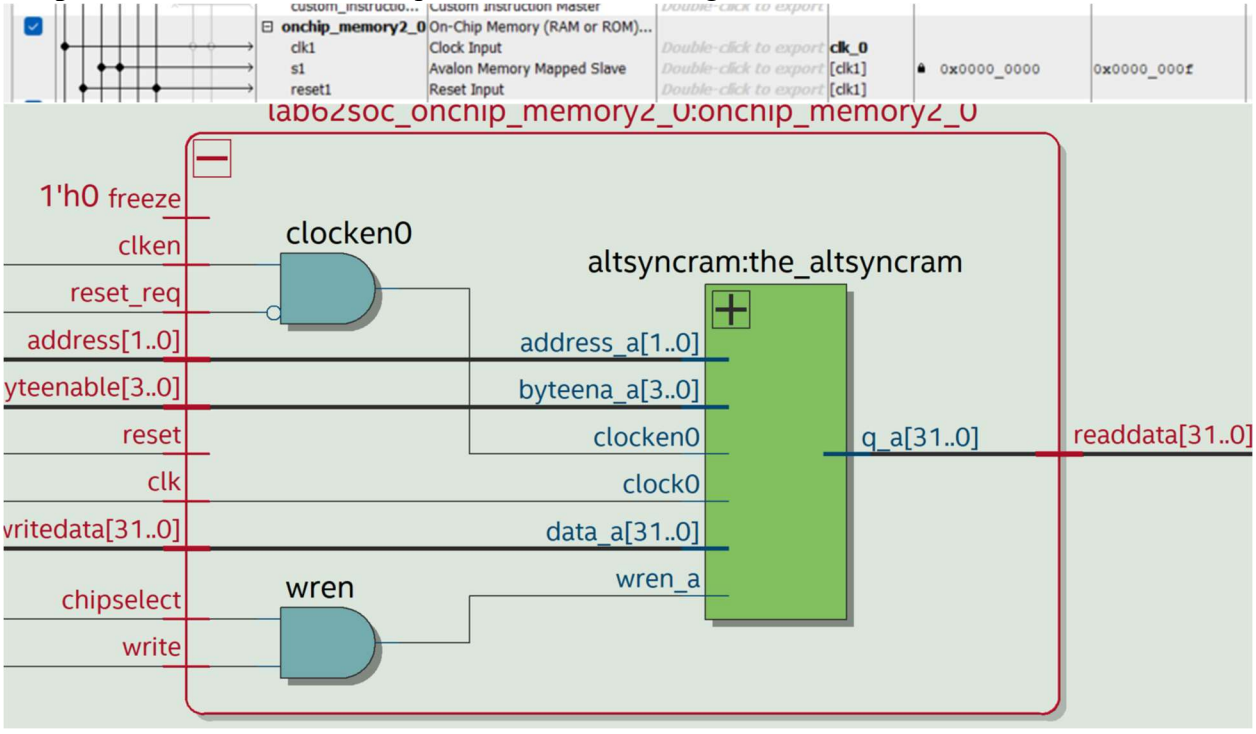
Module: onchip_memory2_0 (in lab62soc.qip)

Connections: clk1, reset1, Nios II Data bus

Exports: None

Description: This is an On chip memory module used for the on chip memory on the FPGA

Purpose: This module is not required, but makes storage easier



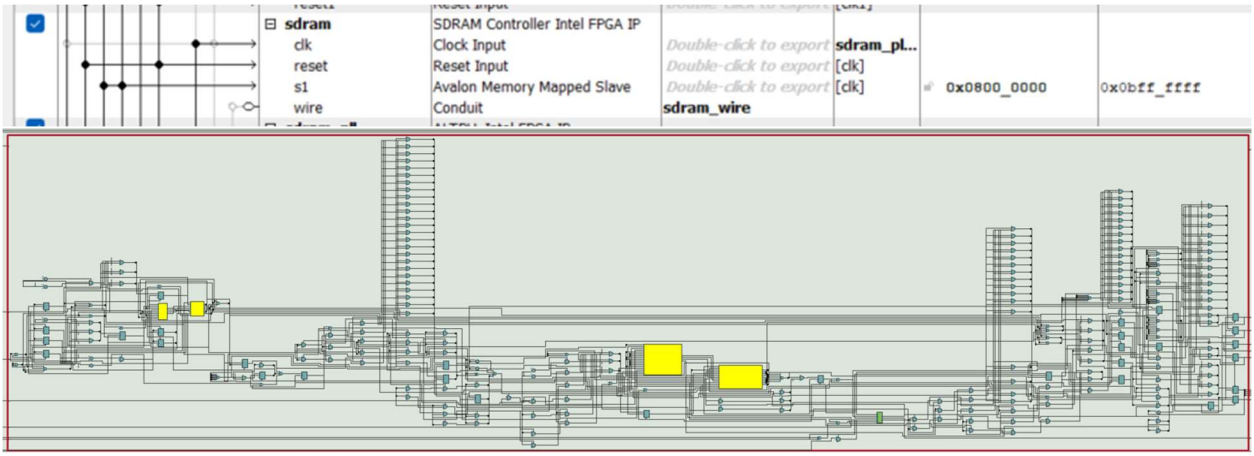
Module: sdram (in lab62soc.qip)

Connections: clk1, reset1, Nios II Data bus

Exports: sdram_wire

Description: This is the functional synchronous memory for the NIOS II processor.

Purpose: Without this, the processor would not be able to store any data.



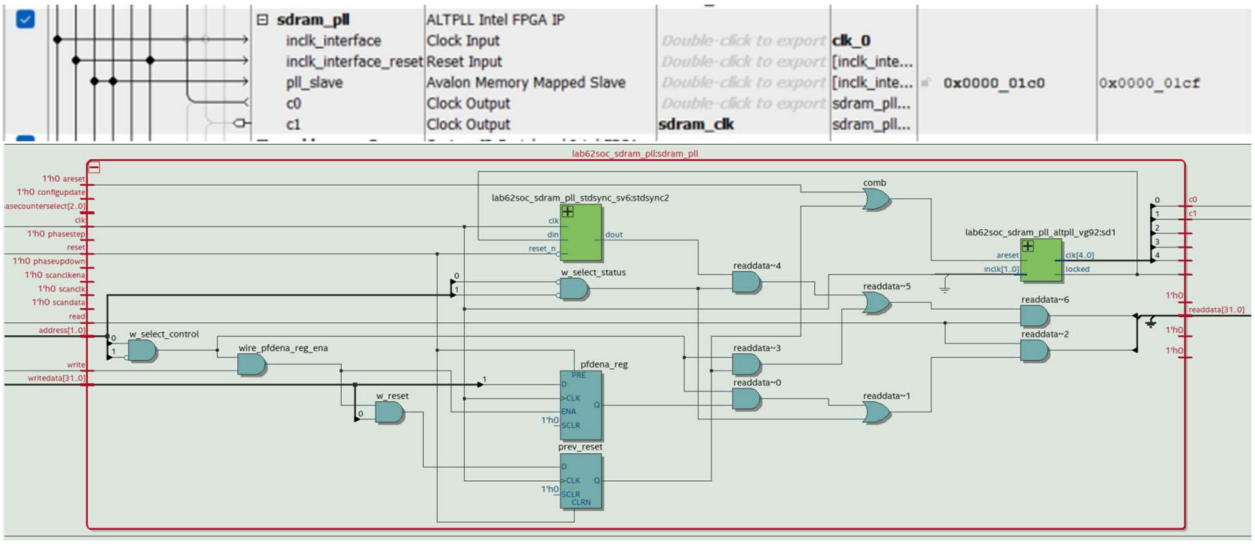
Module: sdram_pll (in lab62soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: sdram_clk

Description: This is the module that is used to control the SDRAM, specifically clocking it at the right frequency.

Purpose: Without this module, the SDRAM would not work well, if at all, because of asynchronous clocking times.



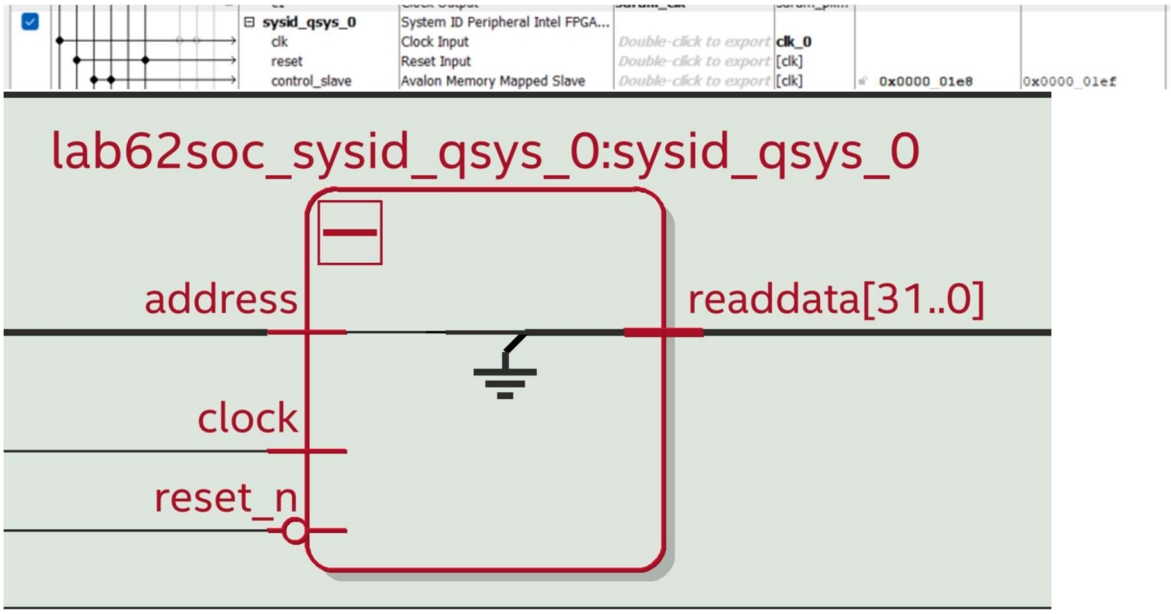
Module: sysid_qsys_0 (in lab62soc.qip)

Connections: clk, reset, control_slave

Exports: None

Description: This module acts as a manager between the hardware and the BSP that is generated.

Purpose: Without this module, the BSP would not work, and in turn the c code could not be run on the FPGA .



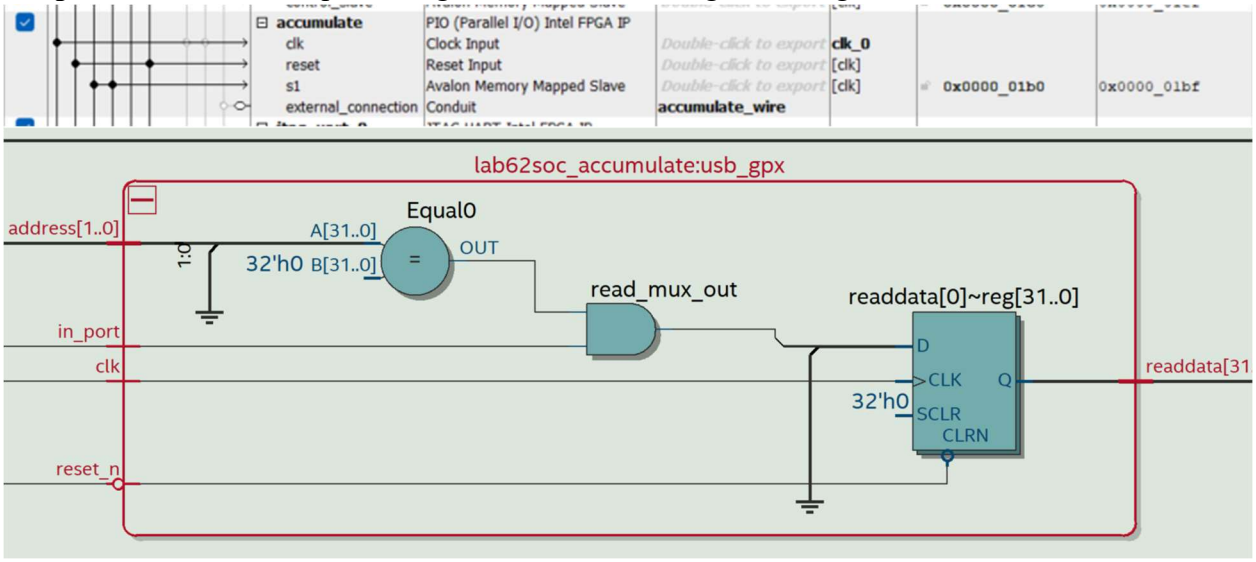
Module: accumulate(in lab62soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: accumulate_wire

Description: This is the accumulate button used in part 1 of this lab.

Purpose: This PIO is required to get the accumulate signal through into the C code.



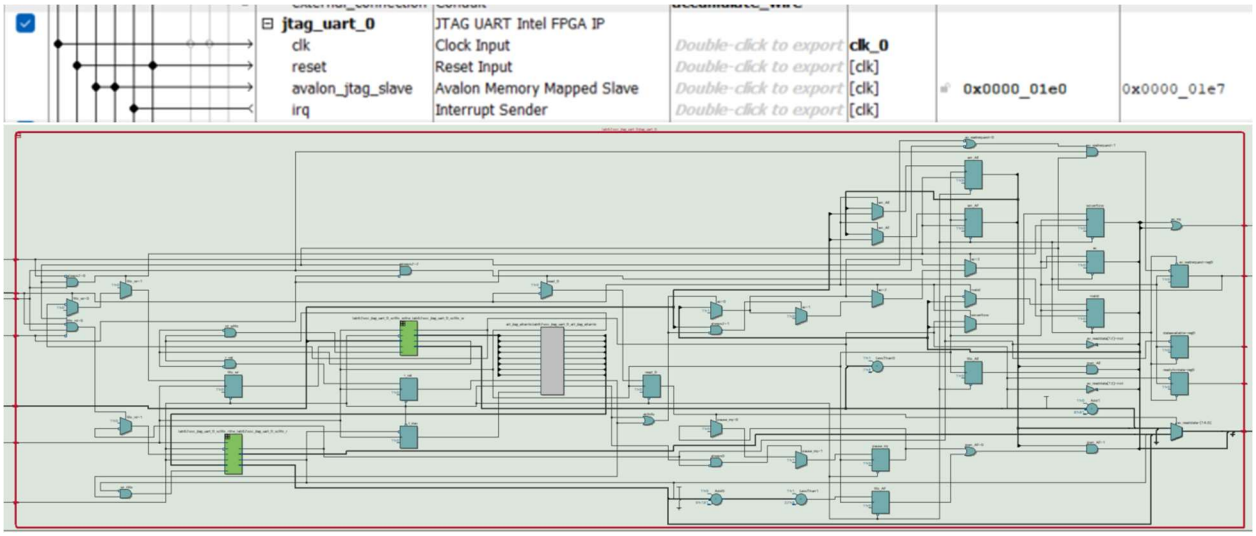
Module: JTAG UART Intel FPGA(in lab62soc.qip)

Connections: clk, reset, Nios II Data bus, IRQ

Exports: None

Description: This is a module within the platform design that is specifically used for USB connections. This was for Part 2 of the lab.

Purpose: This allows for the printf() command to be used in the eclipse IDE.



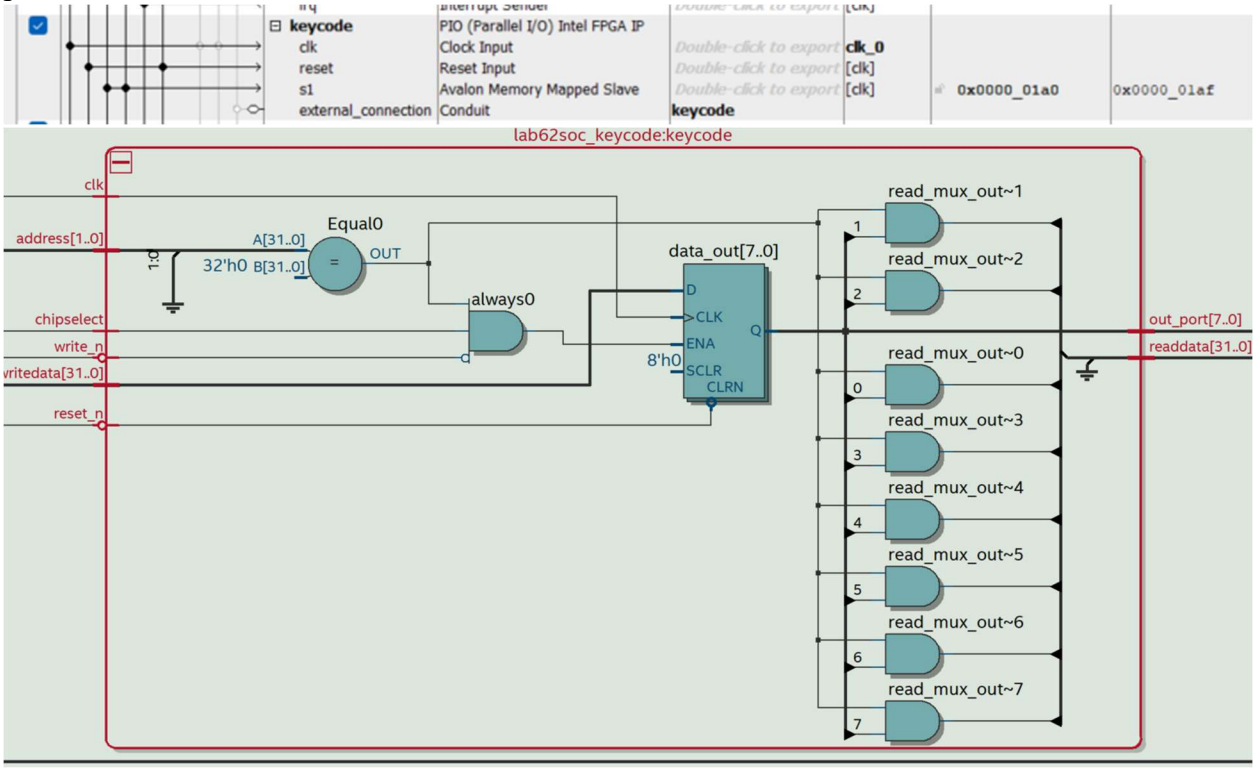
Module: keycode (in lab62soc.qip)

Connections: clk, reset, Nios II Data bus, IRQ

Exports: keycode

Description: This is a PIO designed to take in a keycode from a peripheral and send it to the NIOS II processor. This was for part 2 of the lab.

Purpose: This PIO is imperative to get the proper signal from the keyboard to the NIOS II processor.



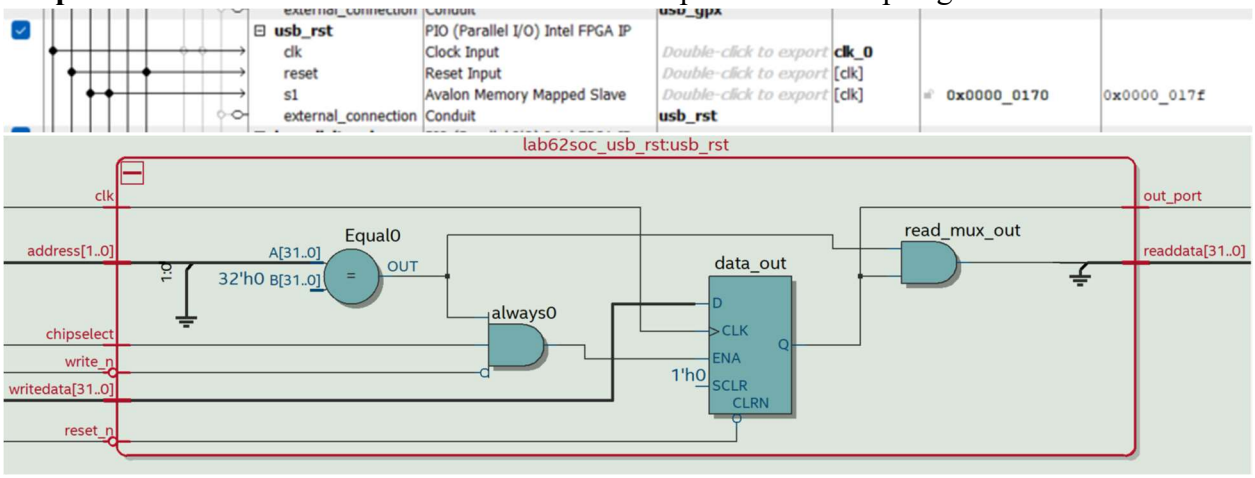
Module: usb_rst (in lab62soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: usb_irq

Description: This is a PIO used within the USB. This was for part 2 of the lab.

Purpose: This PIO is used to make sure the USB responds to interrupt signals.



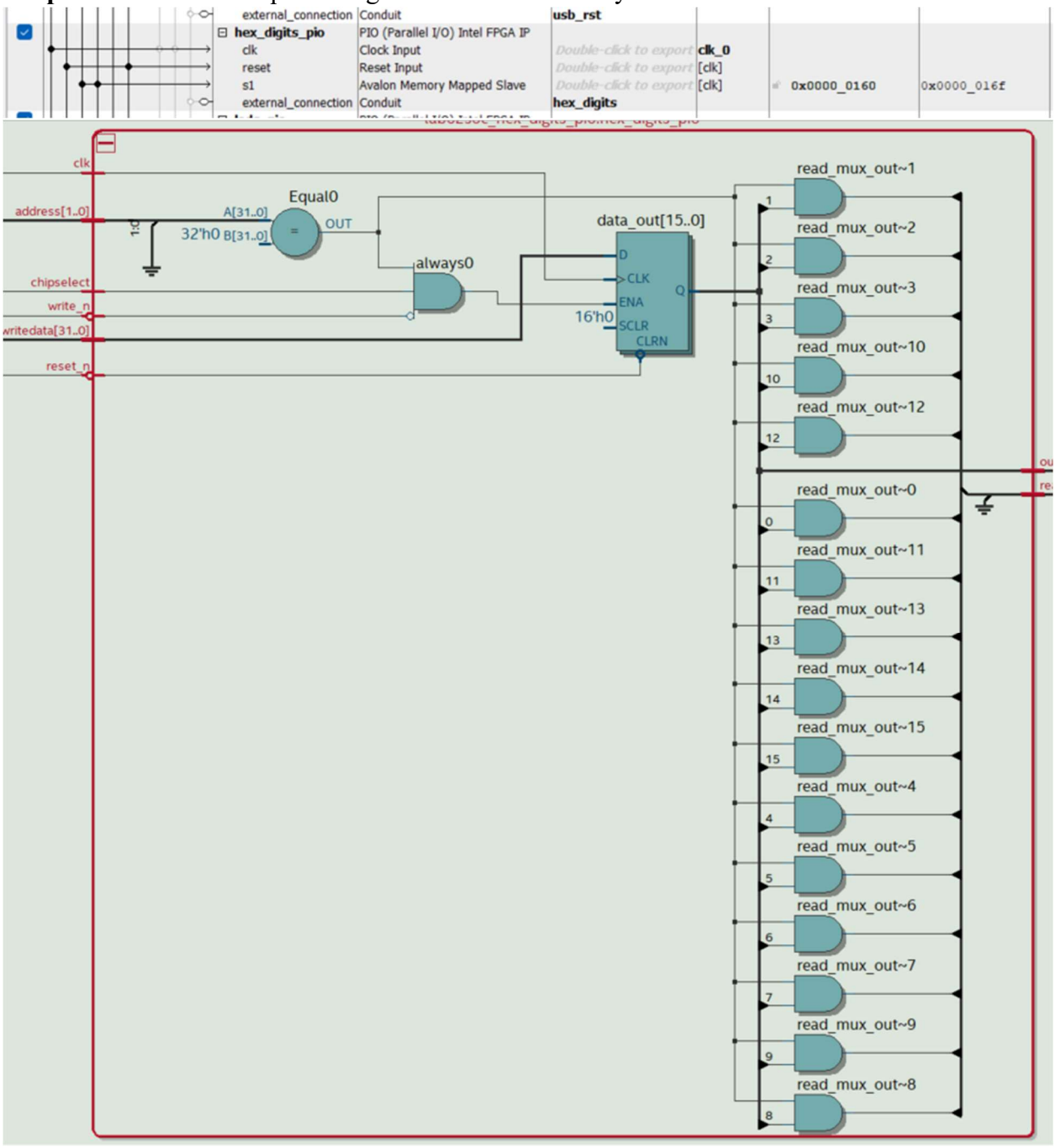
Module: hex_digits_pio (in lab62soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: hex_digits

Description: This is the PIO used to print values to the hex displays coming from the keyboard. This was for part 2 of the lab.

Purpose: This PIO is required to get the hex from the keycodes to the Hex Drivers.



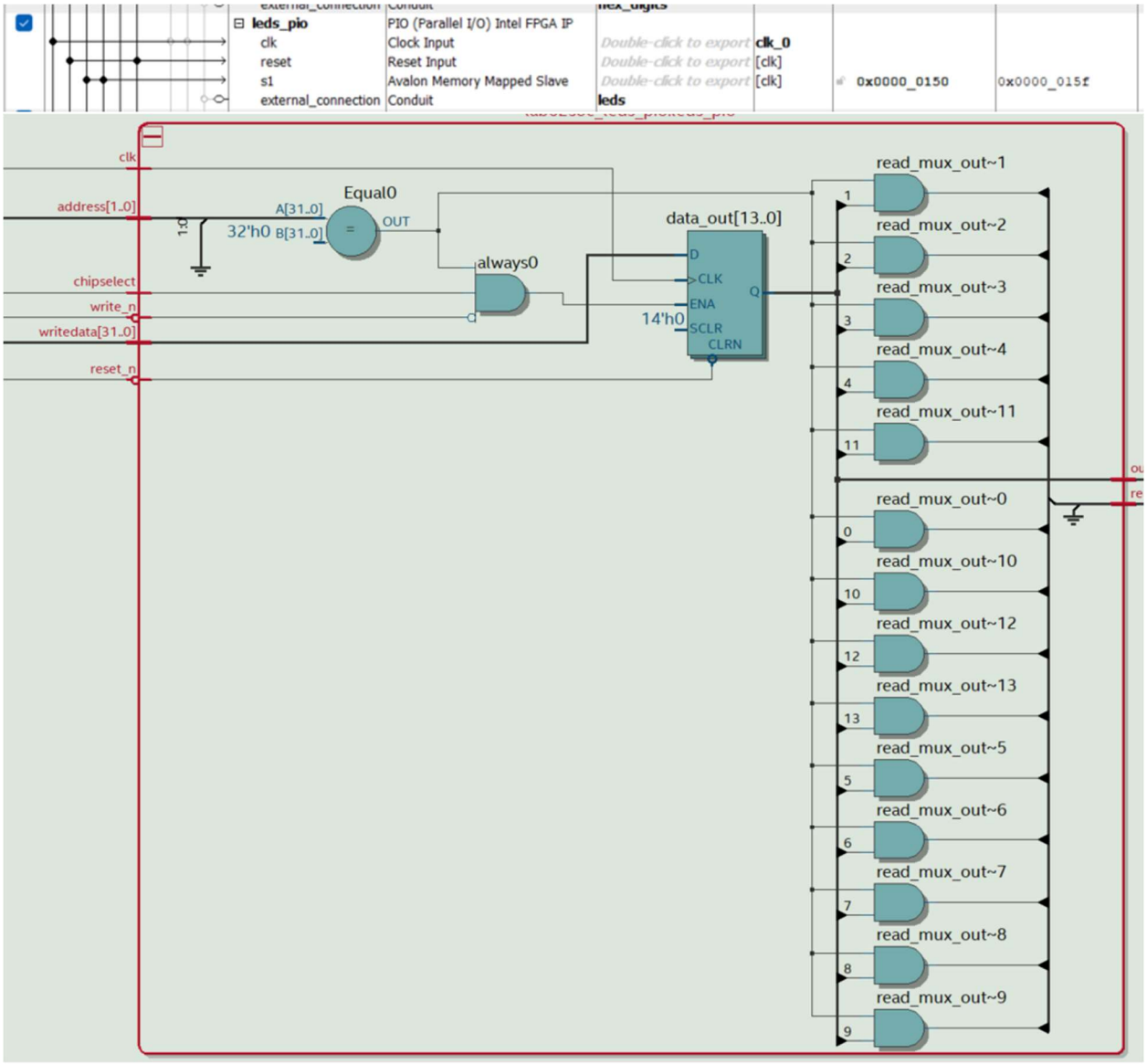
Module: leds_pio (in lab62soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: leds

Description: This is the PIO used to display onto the LEDs. This was for Part 1 of the lab.

Purpose: Without this PIO, it would not be possible to display the running sum from part one on the LEDs.



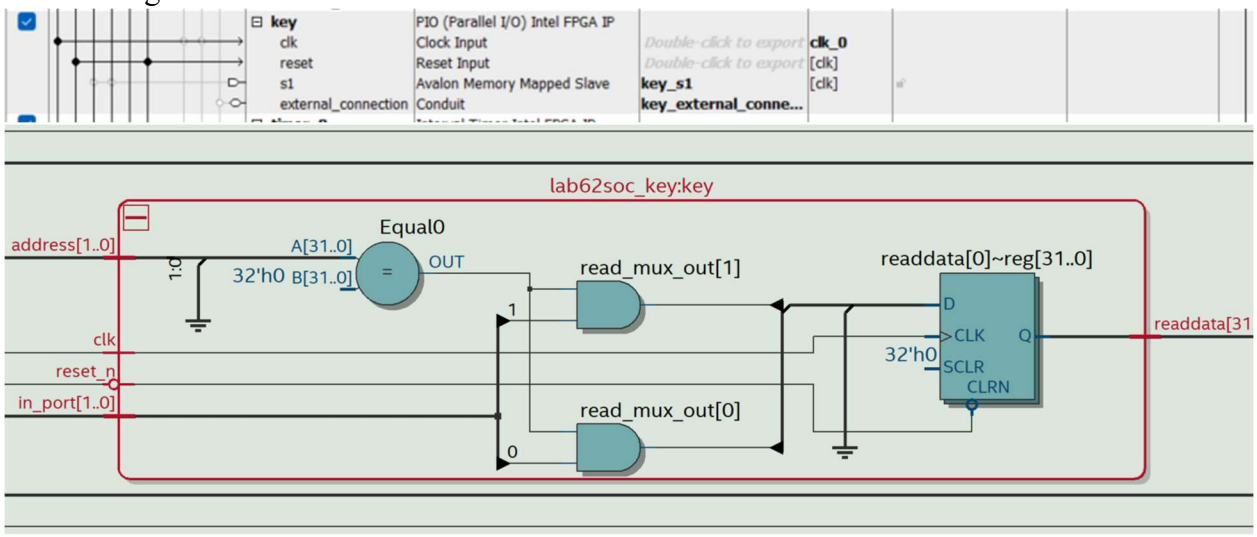
Module: key (in lab62soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: key_s1, key_external_connection

Description: This is the PIO used to detect key presses on the FPGA. Both the key[0] and key[1] are assigned to different functions/keys on the FPGA. Used for part 2 if the lab.

Purpose: This module was required to make sure the 2 buttons were usable in part 2, particularly for resetting the ball.



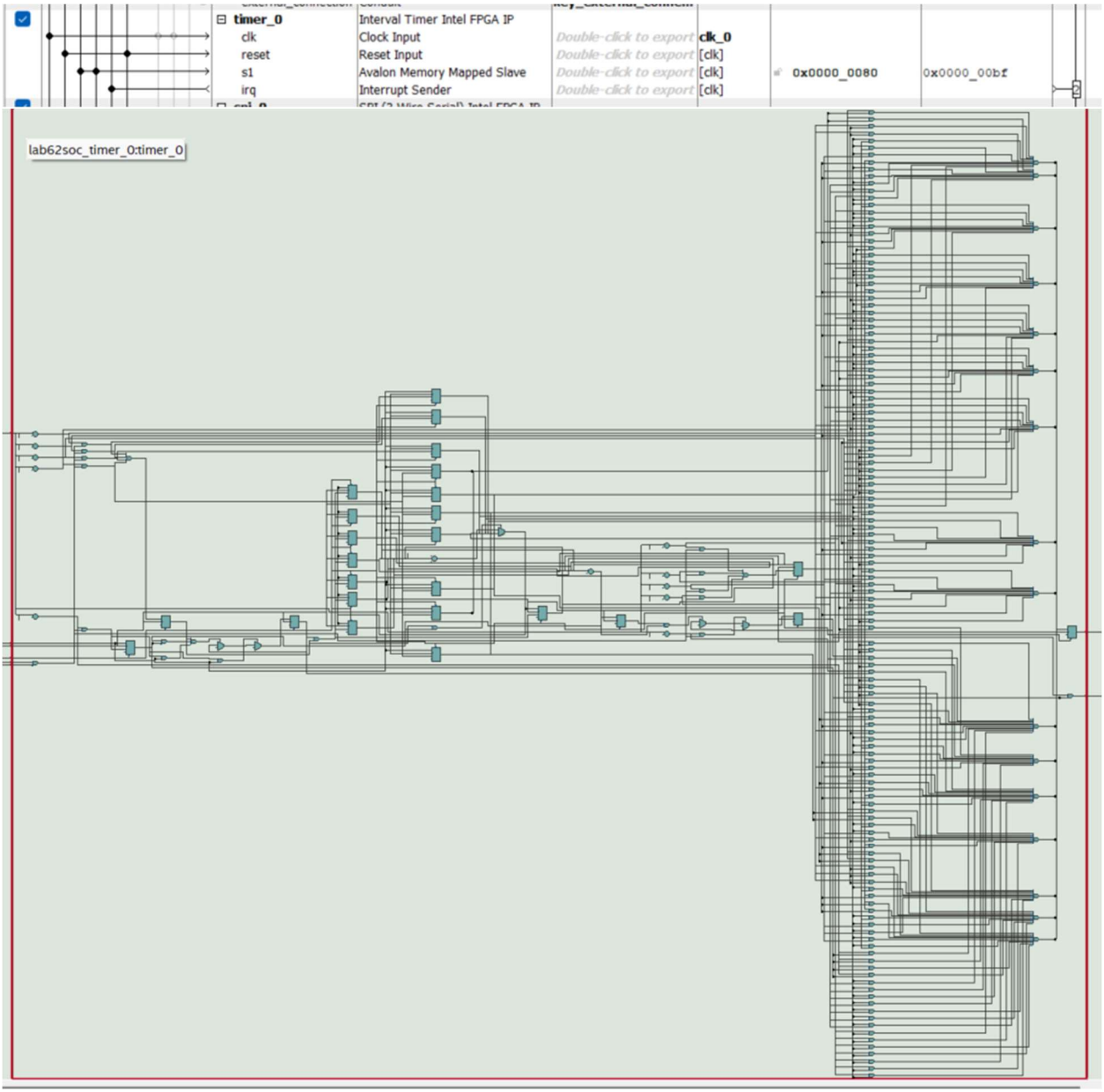
Module: timer_0 (in lab62soc.qip)

Connections: clk, reset, Nios II Data bus, IRQ

Exports: None

Description: This Intel timer IP is used to keep the timings in order. It is required for USB timings to be exact. This IP was added for part 2 of the lab.

Purpose: This time is used to line up the timings with the IRQ as well as with the SPI.



Module: spi_0 (in lab62soc.qip)

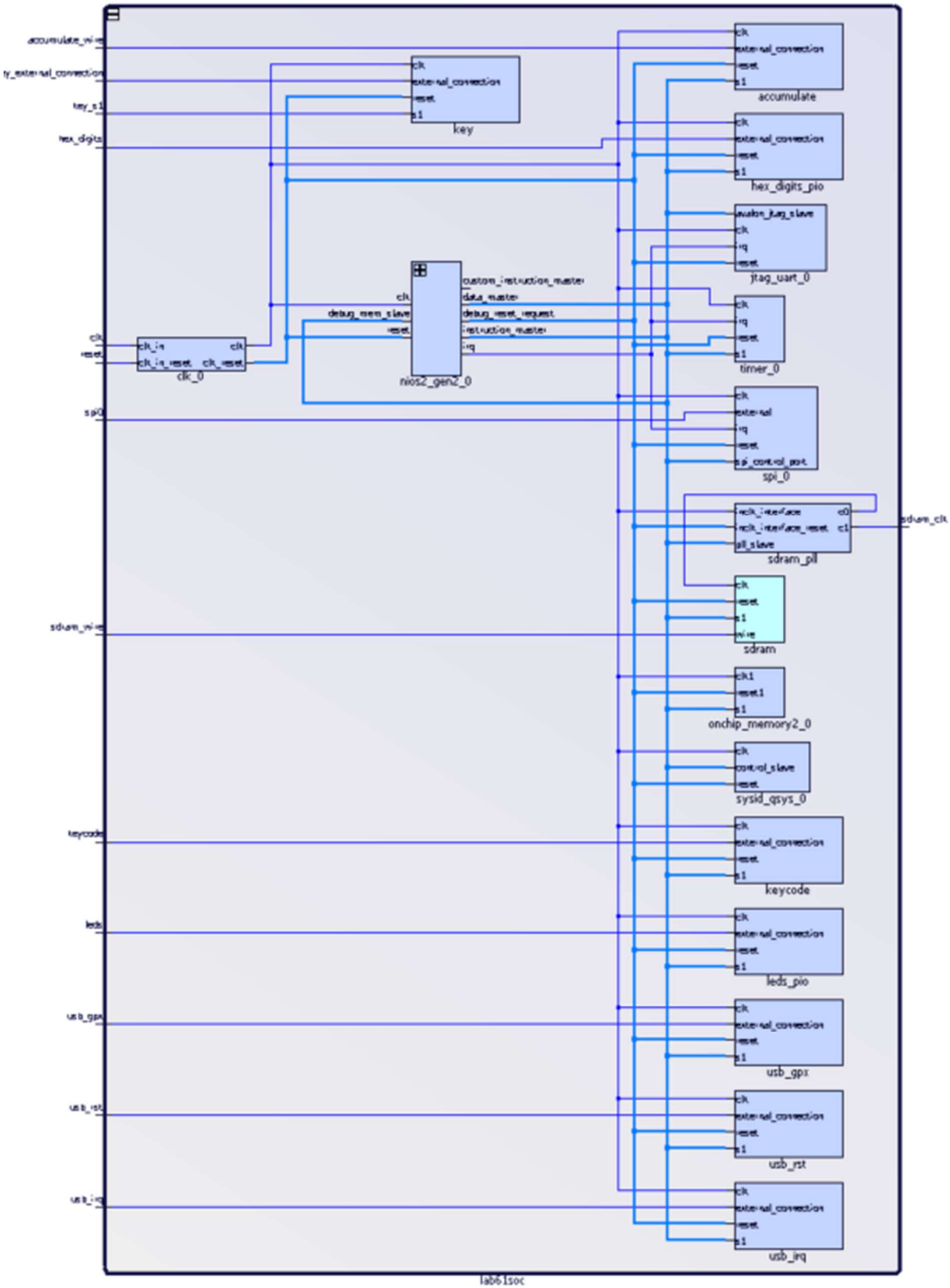
Connections: clk, reset, Nios II Data bus, IRQ

Exports: spi0

Description: This is a SPI(3 Wire Serial) Intel IP and is responsible for the majority of the driver work between the USB and the NIOS II. This was added for part 2 of the lab.

Purpose: This IP is the powerhouse between the NIOS II and the USB, and without it, a connection between the 2 would be infeasible. The block diagram for this section is far too large and complicated to fit into this lab report.

Putting these together we can obtain a schematic view of the whole platform designer:



Ball.sv Extra Credit

Within the Ball.sv file, there is an error that allows the user to manipulate the ball off the screen. To avoid this, some modifications needed to be made to ball.sv. The simplest way to do this is to hard code all the conditions into the ball.sv file. This can be modeled in pseudocode below:

```
if(ball is at top of screen && balll is moving up)begin
  Pause
  Make Ball move down
end
```

And transfering it to actual code we get:

```
always_comb begin
       if((Ball_Y_Pos + Ball_Size) >= Ball_Y_Max && Ball_Y_Motion == 1)begin
           Ball_Y_Motion = -1
       end
   end
```

Within the always_comb, we start with an if statement. *(Ball_Y_Pos + Ball_Size) >= Ball_Y_Max* checks if the ball is at the top of the screen. *Ball_Y_Motion == 1* checks if the ball is traveling in the upwards Y direction. If both of these are true, then *Ball_Y_Motion = -1* will be executed, switching the balls motion to be downward. This can be applied to any edge using the same technique and adding the code within an always_comb in ball.sv.

Software Component

1. **Blinker Code**
   Blinker code makes LED0 blink indefinitely. This is performed by creating a while loop that never ends and is always true. Within this while loop, there are two for loops. The first loop takes the current value of the LEDs and performs a bitwise OR with 0x1. This forces LED0 to become high, while all other LEDs become low. The second while loop takes the current value of the LEDs and performs a bitwise AND with inverse of 0x1. This forces LED0 to become low, as all other LEDs were low they continue at this value. We use a large number for the for-loop's iterator, 100000, to slow down the blinking.
2. **Accumulator Code**
   The code adds the value of the switches to the current value of the LEDs. We accomplished this by creating a while loop that is always true and never ends. Within this while loop, we define a dummy variable as the value that accumulate_PIO points to, this value is active low, when the button is not being pressed the value is high. We then embedded a while loop that does nothing while the dummy variable is equal to the value that accumulate_PIO points to. This while loop creates an edge detection for when the value of accumulate button changes, as the value of dummy will not change after being defined before. We then used an if statement with the condition of checking the value of dummy. This if statement creates a negative edge detection, in other words, the if

statement is only true when the accumulate button is pressed. Finally, within the if statement, we update the value of the LEDs to the addition of the current value of the switches with the current values of the LEDs.

3. **MAXreg_wr**
   This function writes a single bit to the MAX3421E register via the SPI. We created this function by creating a 2-length array X: {register + 2, value to written}. We then use the function "alt_avalon_spi_command(base, slave, write_length, write_data, read_length, read_data, flags)" to connect the MAX3421 to NIOS II. We use the address of the SPI, SPI_0_BASE, as the base, length of the write data is 2, the write data itself is the address of X, the other parameters are all 0. We then check if the return_code is not 0, if there was an issue with the connection, the return_code would be 0, thus we print an error message if this happens.
4. **MAXbytes_wr**
   This function writes multiple bits to the MAX3421E register via the SPI. We created this function by creating a (1 + number of bytes to write)-length array X. To create this array, we set the first value as register + 2, then we created a for loop to take each value from the data array and input into the array. The final X array looks as follows: {register + 2, data[0], data[1], … data[number of bytes – 1]}. We then use the function "alt_avalon_spi_command" to connect the MAX3421 to NIOS II. We use the address of the SPI, SPI_0_BASE, as the base, length of the write data is the number of bytes to write plus one, the write data itself is the address of X, the other parameters are all 0. We then check if the return_code is not 0, if there was an issue with the connection, the return_code would be 0, thus we print an error message if this happens. We then return the pointer of the last byte stored into the register, (data + number of bytes to write).
5. **MAXreg_rd**
   This function reads a single byte from the MAX3421E register via the SPI. We created a local variable to store the read value, val. We then use the function "alt_avalon_spi_command" to connect the MAX3421 to NIOS II. We use the address of the SPI, SPI_0_BASE, as the base, length of the write data is 1, the write data itself is the address of register, the read length is 1, and the place to store the read data is the address of val. We then check if the return_code is not 0, if there was an issue with the connection, the return_code would be 0, thus we print an error message if this happens. We then return the value read from the register, val.
6. **MAXbtyes_rd**
   This function reads multiple bytes from the MAX3421E register via the SPI. We created a local variable to store the read value, val. We then use the function "alt_avalon_spi_command" to connect the MAX3421 to NIOS II. We use the address of the SPI, SPI_0_BASE, as the base, length of the write data is 1, the write data itself is the address of register, the read length is the number of bytes to be read, and the place to store the read data is the address of the data arry. We then check if the return_code is not 0, if there was an issue with the connection, the return_code would be 0, thus we print an error message if this happens. We then return the pointer of the last byte stored into memory, (data + number of bytes to write).

INQ Questions

1. **What are the differences between the Nios II/e and Nios II/f CPUs?**
   The e cpu is the free one(economy), the economy is the bare minimum and the f cpu is the paid version (fast) is a full cpu.
2. **What advantage might on-chip memory have for program execution?**
   On chip memory is much faster than off chip memory. In addition it is not dependent on data going back and forth from the connected pc, but rather just power from the connected pc.
3. **Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?**
   a. Von Neumann: Program Instructions and Data share the same memory.
   b. Pure Harvard: Separate Storage and signal pathways for program instruction and data.
   c. Modified Harvard: Variation of Pure Havard that allows contents of the program instruction to accessed as data.
4. **Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?**
   The on-chip memory needs access to both the instruction list and data, while the LED only needs to read from the data bus, thus it would be useless to connect it to the program bus.
5. **Why does SDRAM require constant refreshing?**
   SDRAM stands for synchronous DRAM. This means that it refreshes with the clock, allowing for more useful cpu cycle.
6. **On the Timing tab, enter the numbers according to Figure 10. What is the maximum theoretical transfer rate to the SDRAM according to the timings given?**
   Data_Width/(Access_Time + RoW_Delay) = 32 bits / (25.4ns) * 1 byte / 8 bits = 157.48 Mb/s.
7. **The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?**
   The clock for the cpu is at 50MHz. If we reduce the clock for the SDRAM, it must be in sync with the clock of the cpu or else it will capture incorrect values.
8. **You must now make a second clock, which goes out to the SDRAM chip itself, as recommended by Figure 11. Make another output by clicking clk c1, and verify it has the same settings, except that the phase shift should be -1ns. This puts the clock going out to the SDRAM chip (clk c1) 1ns behind of the controller clock (clk c0). Why do we need to do this?**
   The cpu needs a FRACTIONAL amount of time to find the address it needs, and to send it to the sdram, so we make sure that the sdram is slightly behind the first clock.
9. **What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?**
   The NIOS II starts at 0x10000 because we wanted the memory to start at 0x0, and everything else to be above it in memory. So we locked the memory at 0x0 and then assigned the addresses automatically from there.
10. **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 8), and how the set and clear functions work by working out an example on paper (lines 13 and 16).**

The volatile keyword a way for us to declare a variable in a way that the C compiler does not complain when we change it outside of the main.c. The volatile keyword signals to the program to read the value from the address every time instead of storing the value within the a temporary register.

11. **Look at the various segment (.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:**
    a. Bss: just a normal data entry like int, char, i.e
    b. Heap: is memory allocation, i.e malloc and free
    c. Rodata: is a read only variable, i.e. const
    d. Rwdata: is data that can be read and written too
    e. Stack: is a data type that is first in last out
    f. Text: is a character or set of characters to read or display

Design Resources and Statistics

LUT	4,099
DSP	0
Memory(B-Ram)	0
Flip-Flop	280
Frequency	50 Mhz
Static Power	96.83mW
Dynamic Power	64.33mW
Total Power	181.80mW

Conclusion

1. **Functionality**
   There were not that many bugs within our design. In Lab 6.1, the main bug that we struggled was with the edge detection of the accumulate button. The main issue was that we could not differentiate between pressing and depressing the accumulate button. We fixed this by creating an if statement that would only change the LEDs if the accumulate dummy variable was high before changing the value of accumulate button. As this value is active low, this creates a negative edge detector and the intended functionality. After fixing this edge detection bug our Lab 6.1 design had full functionality. Regarding Lab 6.2 design, one major bug was with our address of the keyboard input into the MAX3421E. We developed the functions of USB/SPI communication correctly, but the program continued not recognize the keyboard inputs. We then discovered that issue was that we set the address of the keyboard incorrectly, after fixing this issue the ball was able to be controlled by the keyboard as intended. After fixing this bug our Lab 6.2 design had full functionality.
2. **Improvements**
   Overall, this lab was presented very well. However, our main complaint is the lack of guidance on the C functions on 6.2. We feel that if there was a concrete example of how

they wanted us to use the given function then we would be able to understand the code better. Regarding the positives, we really thought that the explanation of the Platform Designer was very helpful in our implementation.