

# **ECE 385**

Fall 2021  
Experiment 6

## Lab 6: SOC with NIOS II in SystemVerilog

Aditya Jain, Vibhav Adivi  
11/01/2021

## Introduction

In this lab we focused on using the NIOS II processor for System-on-chip projects. This enabled us to compile code written in C on the FPGA to use the interface with the FPGA inputs and outputs through C. In week 2, we added USB functionality so that we could use input from a Keyboard using the extension to the FPGA, and then we could output to the VGA using the port that is on the FPGA. The output to the VGA sends signals that are horizontal sync, vertical sync, and then RGB values to determine the color to put out to the screen.

## Written Description and Diagrams

In Lab 6.1, I/O functionality was achieved through the different PIO modules we added inside platform designer. For example, in order to be able to read from the switches we added a PIO module with a width of 8 so we could read the values of all 8 switches. Inside Platform Designer this module was then assigned a base address, which we could use inside the NIOS II code to reference it and obtain the values. A similar module was also added for the buttons and leds.

In order for the NIOS II SOC to interact with the MAX3412E USB chip and the VGA components we had to initialize different IP modules. For the USB chip we added three different PIO modules named `usb_irq`, `usb_gpx`, and `usb_rst`. However, these were only simple GPIO pins that the NIOS II used to interface with the usb chip. The thing that allowed us to write/read to/from registers was the SPI communication protocol. In order for this to work, we also added a SPI IP on Platform Designer. In terms of VGA, the NIOS II doesn't directly communicate with the monitor, but it does provide the FPGA hardware with the keycodes it receives from the keyboard through the SPI protocol.

The SPI protocol uses four main signals to achieve bidirectional communication. One signal is the Slave Select (SS). This is especially useful in microcontrollers where there's usually one master and multiple slave devices so in order to specify which one is active the SS signal is used. Another signal is the Master Out Slave In (MOSI) signal. This is used to send information from the master to the slave. The third signal is the Master In Slave Out (MISO) signal. This is used to send information from the slave to the master. The last signal is the CLK signal so that all messages are synced to the same clock. This is provided by the master.

## C Functions

**LED Blinker** - This function continuously blinks an LED on the FPGA board

**Accumulator** - This function serves as an accumulator, using the switches as an input and displaying the contents of the accumulator on the leds

**MAXreg\_wr** - Writes to a register via SPI

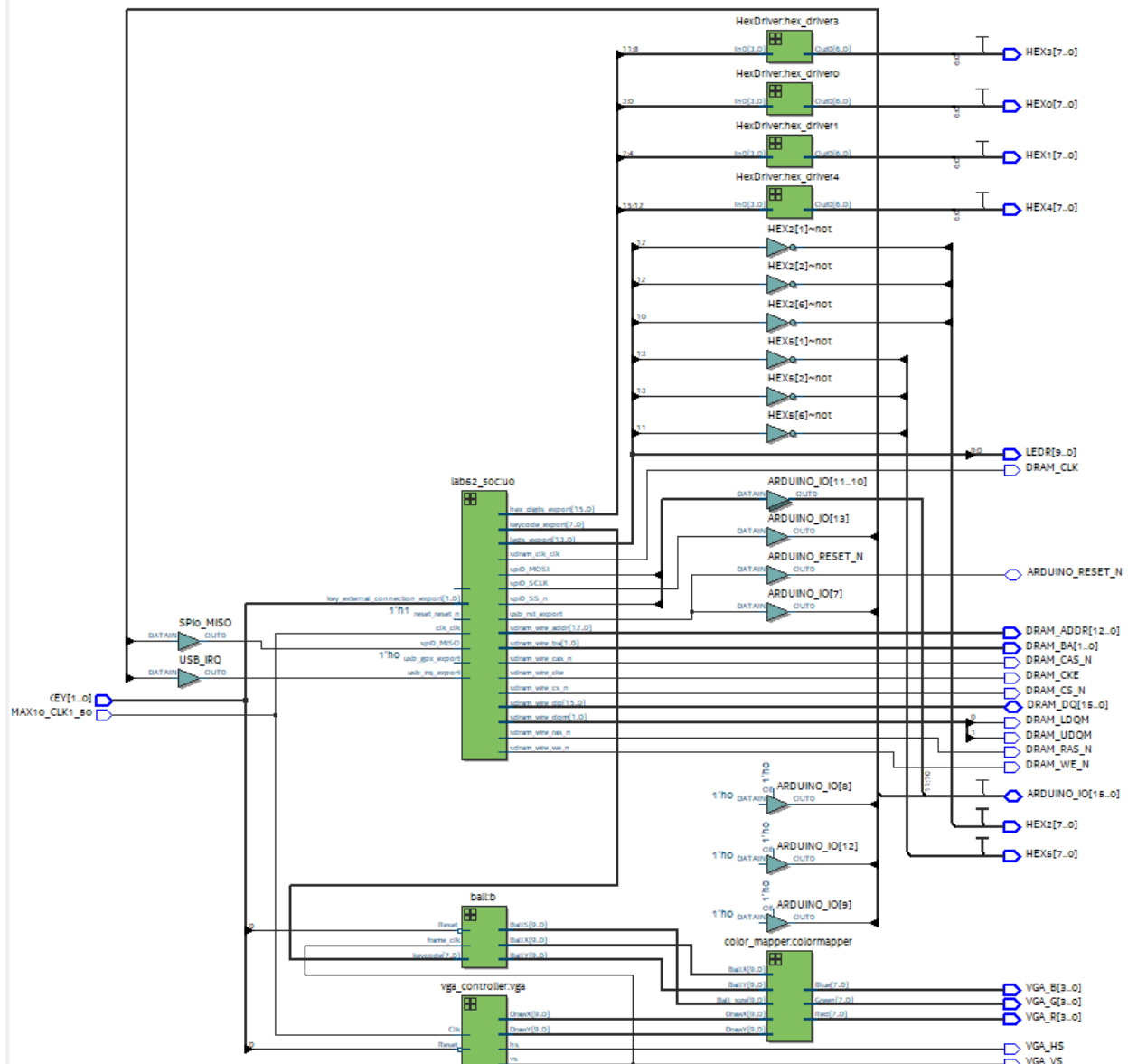
**MAXbytes\_wr** - Writes multiple bytes via SPI

**MAXreg\_rd** - Reads a byte from a register via SPI

**MAXbytes\_rd** - Reads multiple bytes via SPI

VGA monitors linearly display each pixel on the screen fast enough so that humans can't tell that they were individually lit. The electron gun starts at the top left and goes row by row to light the pixels in the correct way. In our code, the VGA controller handles synchronization by providing the Vertical Sync and Horizontal Sync signals as well as determining which pixel the electron gun is currently drawing. The Color Mapper determines what color the pixel should be based on the current state of the electron gun and the position of the ball to determine what should be shown on the foreground vs background. Lastly, the ball module uses the Vertical Sync signal that the VGA controller provides to update the ball position on the rising edge of the frame clock.

### **Top Level Block Diagram**



## Written Description of .sv Modules

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module converts 4-bit binary numbers into hexadecimal.

Purpose: This is necessary for displaying the keyboard code on the HEX displays of the FPGA.

Module: ball.sv

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

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

Description: This module moves a ball on the VGA monitor based on the input, and bounces it back when it hits a corner.

Purpose: This module contains the logic needed to move the ball that has been drawn by the color mapper, fulfilling the final purpose of the project.

Module: Color\_Mapper.sv

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

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

Description: This module determines which color to output to each pixel of the VGA monitor.

Purpose: It is used to draw the ball and background on the monitor that will be used in the ball module.

Module: VGA\_controller.sv

Inputs: Clk, Reset

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

Description: This module synchronizes all the signals and draws the pixels on the monitor.

Purpose: This module is used to define pixels to be able to output to the monitor later on.

Module: lab61.sv

Inputs: MAX10\_CLK1\_50, [1:0] KEY, [7:0] SW, [15:0] DRAM\_DQ

Outputs: [7:0] LEDR, [12:0] DRAM\_ADDR, [1:0] DRAM\_BA, DRAM\_CAS\_N, DRAM\_CKE, DRAM\_CS\_N, [15:0] DRAM\_DQ, DRAM\_LDQM, DRAM\_UDQM, DRAM\_RAS\_N, DRAM\_WE\_N, DRAM\_CLK

Description: This module takes in the different SV modules, the platform designer, and C files to run the adder for week 1 on the FPGA.

Purpose: This is used as the top level entity for week 1 of this lab.

Module: lab62.sv

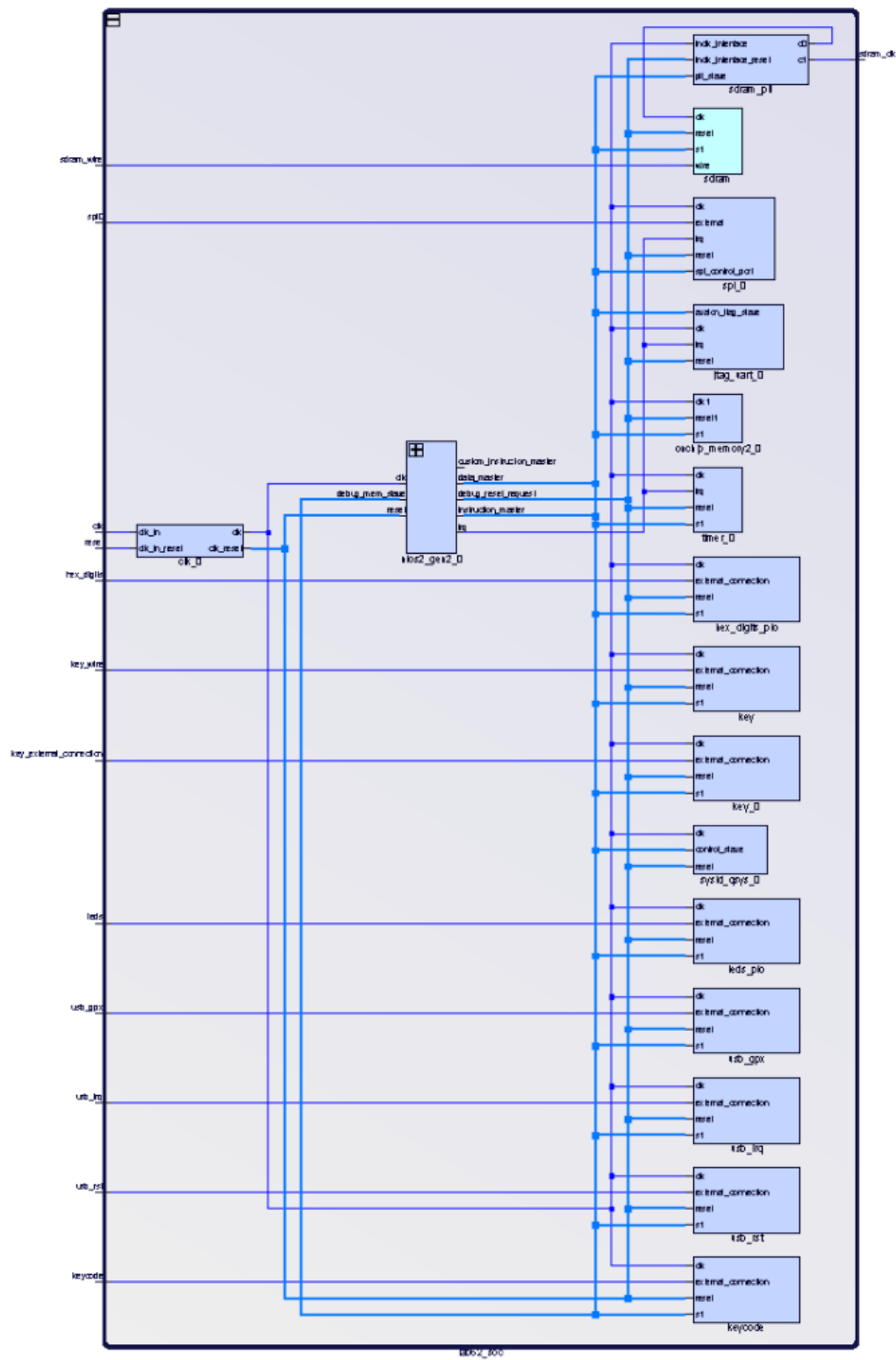
Inputs: MAX10\_CLK1\_50, [1:0] KEY, [7:0] SW, [15:0] DRAM\_DQ, [15:0] ARDUINO\_IO, ARDUINO\_RESET\_N

Outputs: [7:0] LEDR, [12:0] DRAM\_ADDR, [1:0] DRAM\_BA, DRAM\_CAS\_N, DRAM\_CKE, DRAM\_CS\_N, [15:0] DRAM\_DQ, DRAM\_LDQM, DRAM\_UDQM, DRAM\_RAS\_N, DRAM\_WE\_N, DRAM\_CLK, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, VGA\_HS, VGA\_VS, [3:0] VGA\_R, [3:0] VGA\_G, [3:0] VGA\_B, [15:0] ARDUINO\_IO, ARDUINO\_RESET\_N

Description: This module takes in the different SV modules, the platform designer, and C files to run the ball for week 2 on the FPGA.

Purpose: This is used as the top level entity for week 2 of this lab.


### System Level Block Diagram



### Components in both weeks

Name	Description
[-] <b>clk_0</b>	Clock Source
clk_in	Clock Input
clk_in_reset	Reset Input
clk	Clock Output
clk_reset	Reset Output

This is the clock module with 50 MHz frequency. This is the main clock from the FPGA that influences the actions of the other components.

[-]  <b>nios2_gen2_0</b>	Nios II Processor
clk	Clock Input
reset	Reset Input
data_master	Avalon Memory Mapped Master
instruction_master	Avalon Memory Mapped Master
irq	Interrupt Receiver
debug_reset_requ...	Reset Output
debug_mem_slave	Avalon Memory Mapped Slave
custom_instructio...	Custom Instruction Master

This is the main processor of this experiment that processes the instructions to take the C program and run it on the FPGA.

[-] <b>onchip_memory2_0</b>	On-Chip Memory (RAM or ROM)...
clk1	Clock Input
s1	Avalon Memory Mapped Slave
reset1	Reset Input

This is the on-chip memory which is the memory that is directly on the FPGA. It is used for faster processing in some cases because memory can work right on the device without having to fetch information to memory off of the chip.

[-] <b>sdram</b>	SDRAM Controller Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
wire	Conduit

This is the module for the SDRAM, where we can store the software for the FPGA to run, because there is limited space on the chip itself. This is constantly refreshing.

▣ <b>sdram_pll</b>	ALTPLL Intel FPGA IP
inclk_interface	Clock Input
inclk_interface_reset	Reset Input
pll_slave	Avalon Memory Mapped Slave
c0	Clock Output
c1	Clock Output

This module runs the clock for the SDRAM to allow it to constantly refresh and delay the use of the data if it is unstable.

▣ <b>sysid_qsys_0</b>	System ID Peripheral Intel FPGA...
clk	Clock Input
reset	Reset Input
control_slave	Avalon Memory Mapped Slave

The System ID prevents any communication errors between the different components by checking what is uploaded to the FPGA.

▣ <b>key</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is for the button on the FPGA which serves as an input, used to accumulate the sum in the Week 1 lab.

### Components in Week 1 Only

▣ <b>switch</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This corresponds to the switches on the FPGA to choose the numbers to add.

▣ <b>LED</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is the LED on the FPGA, which is used in the blinker portion of the week 1 lab.

### Components in Week 2 Only



[-] <b>jtag_uart_0</b>	JTAG UART Intel FPGA IP
clk	Clock Input
reset	Reset Input
avalon_jtag_slave	Avalon Memory Mapped Slave
irq	Interrupt Sender

This allows Eclipse to communicate with the NIOS II, and is a PIO module.

[-] <b>keycode</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is the output of the keyboard. When a key on the keyboard is pressed, the 8 bit output of that action is output through this module.

[-] <b>usb_irq</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a 1 bit input that sends an interrupt request from the USB device to the CPU.

[-] <b>usb_gpx</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is the USB gpx input, from the MAX3421E chip's gpx pins.

[-] <b>usb_rst</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is the USB reset output, controlling the reset pin of the MAX3421E chip.

[-] <b>hex_digits_pio</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is the output to the hex displays, allowing the keycode values to show up on the hex displays of the FPGA.

▣ <b>leds_pio</b>	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This component is an output for all 14 LEDs on the FPGA.

▣ <b>timer_0</b>	Interval Timer Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
irq	Interrupt Sender

This module serves as an interval timer with a period of 1 ms to keep track of the time-outs needed by the USB.

▣ <b>spi_0</b>	SPI (3 Wire Serial) Intel FPGA IP
clk	Clock Input
reset	Reset Input
spi_control_port	Avalon Memory Mapped Slave
irq	Interrupt Sender
external	Conduit

This is the SPI driver with three wires: SS, MISO, and MOSI. It allows data transfer between the MAX3421E device where the USB is connected, and the NIOS II.

### Description of Software Component

```
int main()
{
    int i = 0;
    volatile unsigned int *LED_PIO = (unsigned int*)0x70; //make a pointer to access the PIO block
    volatile unsigned int *SW_PIO = (unsigned int*)0x60; //make a pointer to access the PIO block
    volatile unsigned int *KEY_PIO = (unsigned int*)0x50; //make a pointer to access the PIO block

    *LED_PIO = 0; //clear all LEDs
    unsigned int sum = 0;
    while ( (1+1) != 3) //infinite loop
    {
        if (*KEY_PIO == 0){
            sum = *SW_PIO + *LED_PIO;
            if (sum > 0xFF){
                *LED_PIO = sum - 0x100;
            }
            else {
                *LED_PIO = sum;
            }
        }
        while(*KEY_PIO == 0){}
    }
    return 1; //never gets here
}
```

This program reads 8-bit numbers from the switches and adds it to an accumulator which is displayed on the leds. It uses the Switch PIO to read the data and the LED PIO to display it. The Key PIO is used for the push button operations.

```
int main()
{
    int i = 0;
    volatile unsigned int *LED_PIO = (unsigned int*)0x70; //make a pointer to access the PIO block
    volatile unsigned int *SW_PIO = (unsigned int*)0x60; //make a pointer to access the PIO block
    volatile unsigned int *KEY_PIO = (unsigned int*)0x50; //make a pointer to access the PIO block

    *LED_PIO = 0; //clear all LEDs
    unsigned int sum = 0;
    while ( (1+1) != 3) //infinite loop
    {
        for (i = 0; i < 100000; i++); //software delay
        *LED_PIO |= 0x1; //set LSB
        for (i = 0; i < 100000; i++); //software delay
        *LED_PIO &= ~0x1; //clear LSB
    }
    return 1; //never gets here
}
```

This program uses an infinite loop and for loops preset to a certain range to continuously blink an LED on the FPGA.

```

//writes register to MAX3421E via SPI
void MAXreg_wr(BYTE reg, BYTE val) {
    BYTE data[2] = {reg + 2, val};
    //alt_u8 readData = 0;
    int return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 2, data, 0, NULL, 0);
    if (return_code < 0) printf("Error");
}

//multiple-byte write
//returns a pointer to a memory position after last written
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {

    BYTE data2[nbytes+1];
    for (int i = 0; i < nbytes+1; ++i) {
        if (i == 0) {
            data2[i] = reg + 2;
        }
        else {
            data2[i] = data[i-1];
        }
    }

    //alt_avalon_spi_command(0x00c0, 0, 1, &reg, 1, &readData, 0);
    int return_code = alt_avalon_spi_command(SPI_0_BASE, 0, nbytes+1, data2, 0, NULL, 0);
    if (return_code < 0) printf("Error");
    return (data + nbytes);
}

//reads register from MAX3421E via SPI
BYTE MAXreg_rd(BYTE reg) {

    alt_u8 readData;
    int return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, 1, (&readData), 0);
    if (return_code < 0) printf("Error");
    return readData;
}

//multiple-byte write
//returns a pointer to a memory position after last written
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    int return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, nbytes, data, 0);
    if (return_code < 0) printf("Error");
    return (data + nbytes);
}

```

These 4 functions are used to perform read/write operations to the usb chip. Specifically, there are 2 read and write functions each. One variant performs the operation on a single byte while the other variant performs a multiple byte operation.

### Post-Lab Questions

LUT	4071
-----	------

DSP	0
Memory (BRAM)	11,392
Flip-Flop	2542
Frequency	157.08 MHz
Static Power	96.51 mW
Dynamic Power	60.82 mW
Total Power	178.41 mW

### INQ Questions

- What are the differences between the Nios II/e and Nios II/f CPUs?
  - Nios II/f (f for fast) is designed for better performance so it has more resources and logic elements while the Nios II/e (e for economy) is designed to use the least amount of resources possible.
- What advantage might on-chip memory have for program execution?
  - Because on-chip memory is physically closer to the processor, it decreases the amount of time needed to fetch information as compared to a off-chip memory.
- Note the bus connections coming from the NIOS II; is it a Von Neumann, “pure Harvard”, or “modified Harvard” machine and why?
  - This processor is a Modified Harvard machine because there are separate paths for instructions and data and because instruction memory can be accessed as if it was regular data.
- 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 Led peripheral only needs to display values provided to it on the data bus while the on-chip memory needs access to both so that it can be read/written to when a fetch or other data instruction is being performed.
- Why does SDRAM require constant refreshing?
  - SDRAMs use capacitors to store information and because they discharge they need to constantly be refreshed to preserve their value.
- Note that there is one 32M\*16 chips, so the total amount of memory should be 512Mbits (64 Mbytes), make sure this is consistent with your above numbers; you will need to justify how you came up with 512 Mbit to your TA.
  - $2^{\text{rows}} * 2^{\text{columns}} * 2^{\text{(bank pins)}} * \text{width} = 2^{29} = 512\text{Mbits}$

7. What is the maximum theoretical transfer rate to the SDRAM according to the timings given?
  - a.  $(32 \text{ bits} / 5.4\text{ns}) = 5.92\text{e}9 \text{ bits/sec}$
8. The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?
  - a. The SDRAM timings are very precise and narrow so it must capture data within a certain period or else the values might be wrong.
9. 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? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.
  - a. It takes time for signals like the control and data signals to be valid due to flip flop delay so the SDRAM chip should be 1ns behind the controller clock so it captures data in the middle of the valid window.
10. What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?
  - a. Exception vector address
11. 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).
  - a. Explained in previous section
12. Look at the various segment (.bss, .heap, .rodata, .rdata, .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 - uninitialized data, heap - dynamic memory allocation, rodata - constants, rdata - read/write data, stack - activation record/function calls, text - text/strings

### **Conclusion**

Both parts of the lab were functional, and they were fairly straightforward save for a few bugs that we were stuck on. In week one, we did not realize that the buttons on the FPGA are active low, so we programmed in C for them to be active high, which resulted in some bugs. After figuring this out, the week one project worked perfectly. For week 2, we spent some time trying to figure out what to put in for read signals in write functions and vice versa, but again, were able to find our problems eventually, and both the LEDs from week 1 and the ball from week 2 worked well. The lab manual was helpful in guiding us through both parts of the lab, and

I like how most of the lab was run. One part that was confusing was the pictures and explanation of the platform designer and where to draw the connections. It would be great if these could be made clearer. Other than that, this lab went well.