ECE 385

Spring 22

Experiment #6

SOC with NIOS II in SystemVerilog

Siddarth Iyer, Geitanksha Tandon

TA: Ruihao Yao (RY)

**Includes the extra credit for Lab 6.

1. Introduction:

---

i. Summarize the basic functionality of the NIOS-II processor running on the MAX10 FPGA.

In our lab, we learned how to create a SoC (System-On-Chip projects) to compile C code on our FPGA along with input and output functionalities via the integration of the NIOS-II Processor.
The NIOS-II Processor (IP-based 32-bit CPU) is the foundation of all of our (SoC) System-On-Chip projects. It uses C for programming, and acts as the controller for tasks that are low-performance, especially those that act as data inputs and outputs for user interfaces, leaving the FPGA to handle the operations which require higher performance. The NIOS-II instantiates IP blocks which actually perform the input and output functions. Our week 1 circuit had basic controllers for functions like Synchronous Dynamic RAM (SDRAM), Parallel Input/Output (PIO) for LEDs, Keys and Switches.
In week 1, we needed to accept and sum up / accumulate the values inputted from our Switch PIOs (8-bit values) and display the values on our LED PIOs (8-bit values). The keys served to reset LEDs and the sum (Key[0]), while Key[1] served to accumulate the values from the switches.

---

ii. Briefly summarize the operation of the USB/VGA interface.

In week 2, we were asked to extend this functionality to integrate the peripheral extensions of a USB (keyboard), as well as a VGA Monitor to our DE-10 Lite board to display a ball moving on the screen using WASD keys (2 bits for these) on the keyboard. We did this by crafting drivers that would be able to receive and send data from these peripherals. Our USB port on the FPGA was used to access our Keyboard and VGA port was used to control the Monitor. The 2 bits we used to represent the WASD keys were represented in ASCII and used by our C program to maneuver the ball, which was displayed via the VGA module which used horizontal and vertical sync to display pixels.

---

2. Written Description and Diagrams of NIOS-II System

i. Describe in words the hardware component of the lab, in this lab, only the Platform Designer module is here.

The hardware component of the lab was designed on the Platform Designer. This is a GUI that would help us set up our hardware components, meaning that we crafted our System Verilog hardware files via a GUI instead of creating .sv files and instantiating modules, which the platform designer did for us directly. We then used these while writing our software code in Eclipse. In the first week, we used the Platform Designer to communicate with the "peripherals" on our FPGA - LEDs, Switches, Clock signals, SDRAM and so on, using available IPs. In the second week, we used the SPI protocol to create drivers (IPs) that would interact with our USB Component and our VGA output. The use of the platform designer lets us connect the clocks and the data instructions / master arrays to all necessary components, and this generates a .qsys file that would hold the CPU and relevant peripherals.

A more thorough description of all components employed in our platform designer is given in section 4 whilst describing the .v files for the socs resulted by the Platform Designer.

---

ii. Describe in Lab 6.1 how the I/O works.

In Lab 6.1, we were supposed to connect our NIOS-II to our input / output peripherals. In this part of the lab, our peripherals were the LEDs and Switches on the FPGA. The way we connected these peripherals to our processor was via the Avalon Memory Mapped Bus. It ensured that each peripheral and device listed above had an assigned base address that was used to read and write onto the peripheral as necessary.
In 6.1, we had the Switches and 2 key buttons (key_0, key_1) being instantiated as input PIOs, and the LEDs and Hex Displays being our output PIOs. Using the platform designer, we connected these components to our bus and ensured all peripherals were given access to the data and instruction bus as necessary.

{Note: 2. iv. before 2. iii.}

iv. Written description of the SPI protocol.

We consider how the NIOS is a processor that needs to access the MAX3421E USB Chip because it is an external peripheral - we need to hence use a Serial Peripheral Interface (SPI) Protocol. The SPI consists of 4 aspects:

- Clk: It sends the clock signal from the main device (master) to the dependent devices (slaves)
- Master Out Slave In (MOSI): This transfers data from the MAX3421E to the NIOS.
- Master In Slave Out (MISO): This transfers data from the NIOS to the MAX3421E.

- Slave Select (SS): This selectes between different slaves that are connected and chooses which one to transfer data to / receive data from.

---

iii. Describe in words how the NIOS interacts with both the MAX3421E USB chip and the VGA components.

Using the SPI Protocol, we can interact with our USB and our VGA.

USB Interaction: Using the 4 signals (Clk, MOSI, MISO, SS) that are received from the USB, we can employ the use of the IntelFPGA SPI Peripheral driver function whilst coding in order to receive and transmit signals to our external USB Component. In our c program, we employ this function in order to perform 4 main functions that actually execute the interaction between the USB signals and the SPI Driver:

int alt_avalon_spi_command(alt_u32 base, alt_u32 slave, alt_u32 write_length, const alt_u8 wdata, alt_u32 read_length, alt_u8* read_data, alt_u32 flags)*

VGA Interaction: We have a VGA module which handles the transition of the electron beam over the screen, and how it color maps the required colors and prints the required pixel on the screen. The color mapper and a controller were 2 sv modules we employed to interact with the MAX3421E.

---

v. Describe the purpose of each function you filled in the C code (you do not need to describe the functions you did not modify.

Week 1: We had been given a main.c file which used to make the LED peripheral on the FPGA light up. We altered this to create an accumulator which would sum the values on the LEDs with the ones on the switch peripherals. It also could reset the values on the LEDs to start over.

Week 2: We had to ensure that our USB peripheral could work with the MAX3421 and so we had to create 4 functions that could either write to or read from the master to slave and vice versa. We used the *alt_avalon_spi_command* in order to perform these functions as described below.

Functions:

```
void MAXreg_wr(BYTE reg, BYTE val;
```

This function writes the register value from to MAX3421E via SPI. This was done by first writing the value of the reg + 2 and concatenating it with val, executing the spi function and returning the return code given by the function.

```
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data);
```

This function writes the data array through the SPI function and returns the return code given by function. The entire function returns the pointer to the memory address after the last write function was executed.

```
BYTE MAXreg_rd(BYTE reg);
```

This function reads the register value from the MAX3421E and returns the value that was read.

```
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data); r
```

This function reads the values from the data and returns the pointer to the memory position where the last data was written.

---

vi. Describe in detail the VGA operation, and how your Ball, Color Mapper, and the VGA controller modules interact.

The Video Graphics Array (VGA) is the video type where the screen has a matrix of pixels of size 640 x 480. The electron beam scans and prints the pixel color and type from in row major order.

The implementation of this VGA involved the working with 3 main components:

VGA Controller: The VGA Controller creates a new clock of speed 25MHz using the given 50MHz clock, and controlls the X, Y coordinates of the pixels that are being printed. It controls their movement and resetting. Aside from this, it handles the logic for blanking of the screen to orient the colors, and generate the sync pulses for the horizontal and vertical coordinates.

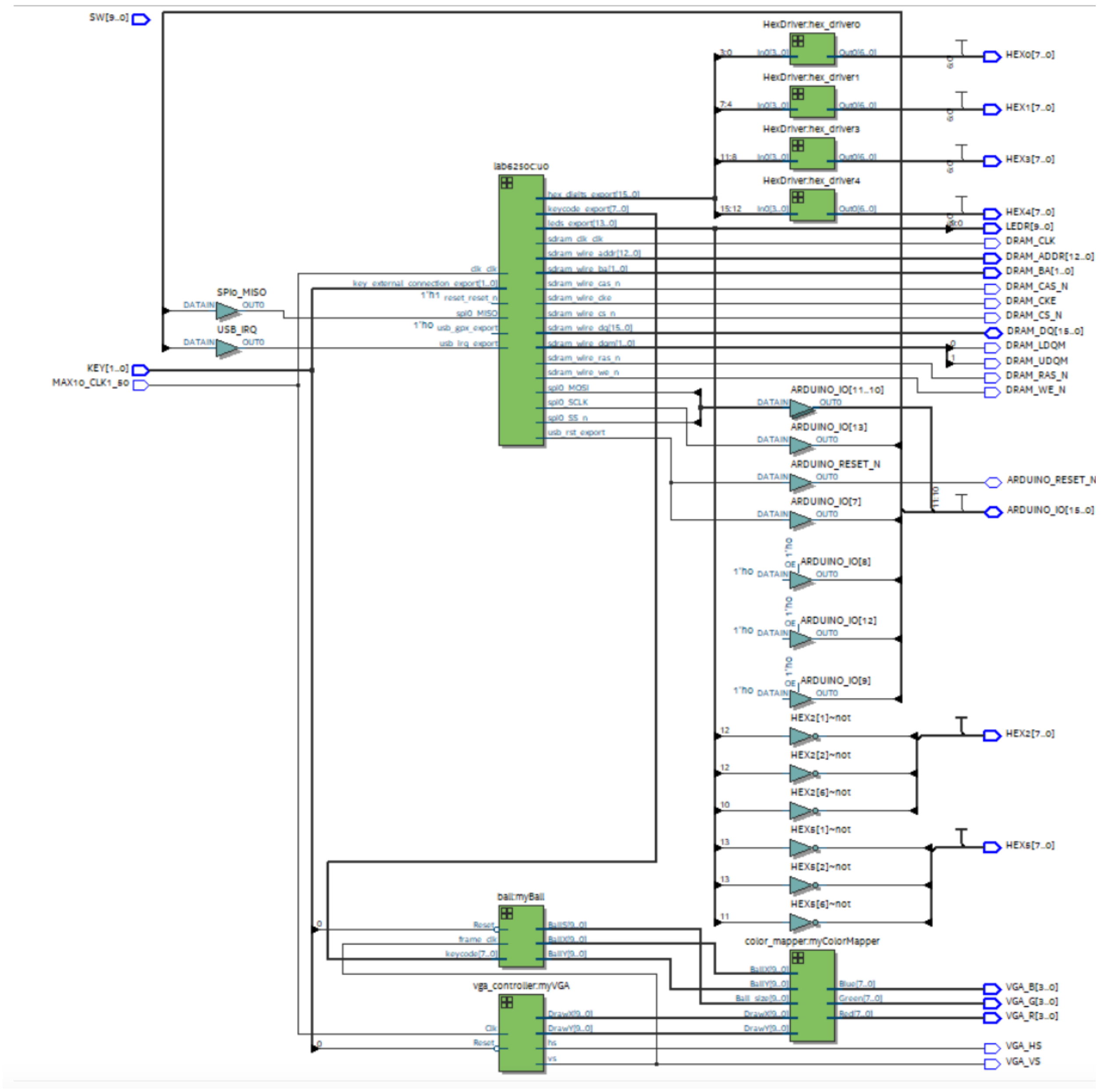
Ball Module: This module controls the movement of the ball that will be displayed on the screen, and ensures that the movement of the ball along the X, Y coordinates is handled correctly, depending on the input from the USB monitor (or mouse, depending on peripheral used).

Color Mapper: This module checks if one must paint the front (foreground) or the back (background) of the module and uses the VGA Controller module for the X, Y coordinates of the pixel.

---

3. Top Level Block Diagram

i. This diagram should represent the placement of all your modules in the top level. Please only include the top-level diagram and not the RTL view of every module.



4. Written Description of all .sv Modules

a. A guide on how to do this was shown in the Lab 5 report outline. Do not forget to describe the Platform Designer-generated file for your Nios II system! When describing the generated file, you should describe the PIO blocks added beyond those just needed to make the NIOS system run (i.e. the ones needed to communicate with the USB chip and other components). The Platform Designer view of the Nios II system is helpful here.

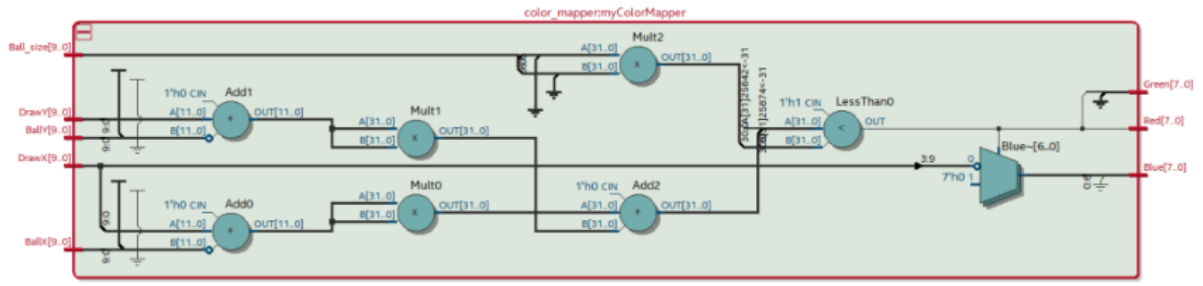
Module: Color_Mapper.sv

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

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

Description: This module checks whether the ball is supposed to be drawn, and also checks the RGB color values of the pixel depending on the input. It uses an if-else statement to check if the ball is supposed to be drawn - if yes, it draws the color of the ball, otherwise it draws the color of the background

Purpose: This module decides the color of the pixels given by DrawX, DrawY, controlling the color palette of the screen.



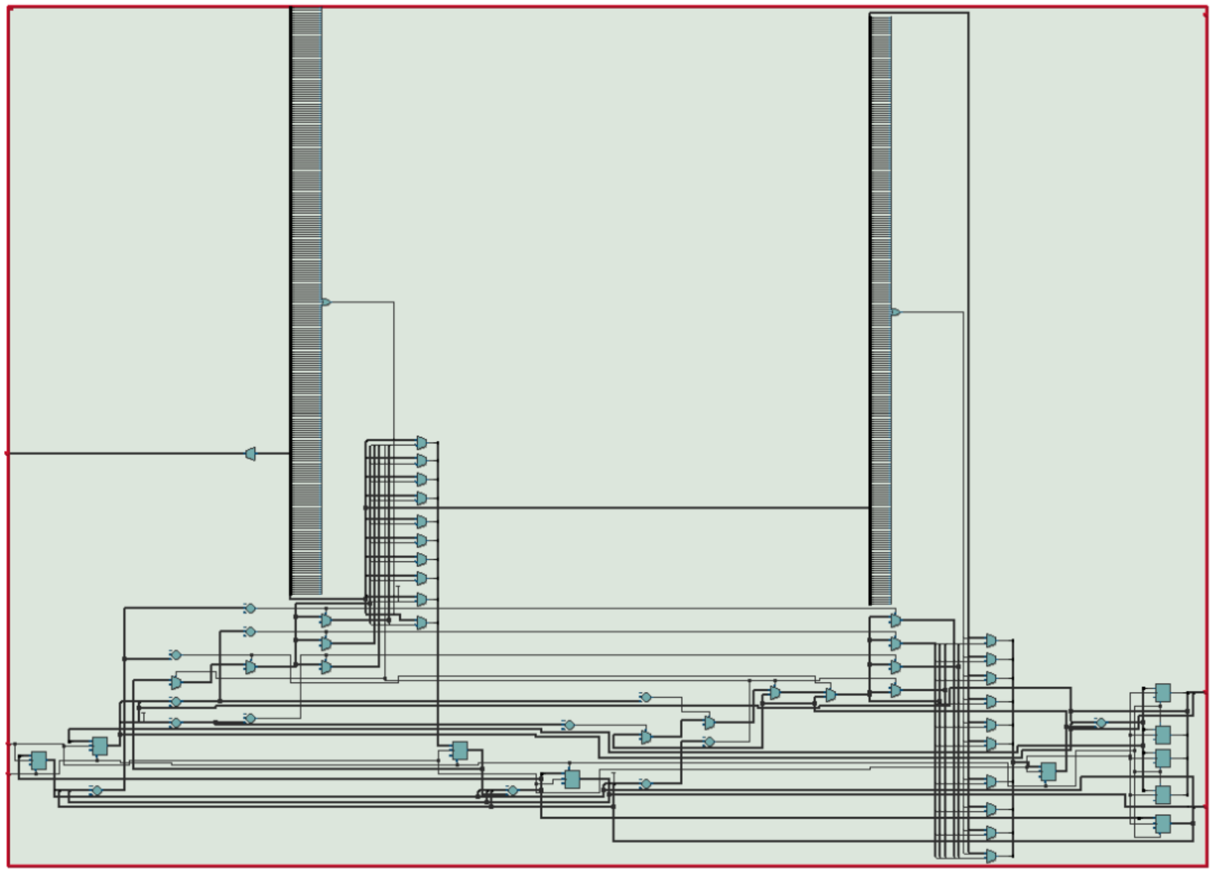
Module: ball.sv

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

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

Description: This module lets us describe the movement of the ball by defining the X, Y coordinates and then uses a synchronous always_ff block to update the direction of the movement of the ball based on the key (WASD) clicked on the keyboard. This code was also altered in order to define how the ball was supposed to stay at the corner instead of disappearing and leaving the screen if they key was pressed when it reached the edge of the screen.

Purpose: This module helps describe the next direction and movement of the ball based on the input provided by the keyboard, and handles how the ball reacts when it reaches the edge of the screen.



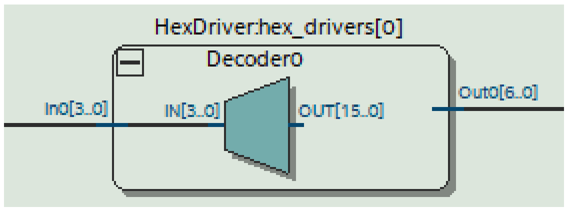
Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This makes use of a unique case switch to set the 7 bits of Out0 to the corresponding 4 bit In0, to display the 4 bit hex number on the 7-segment LED.

Purpose: This module maps a hex number to its corresponding value so it can be displayed on a 7-segment LED.



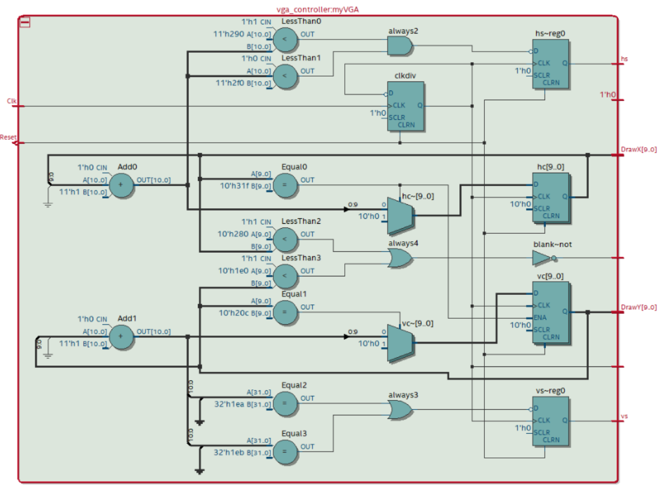
Module: VGA_controller.sv

Inputs: Clk, Reset

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

Description: This module generates the VGA signals by creating a 25MHz clock from the 50MHz clock given to using the always_ff block. If reset is clicked, the ball is moved to the center. Otherwise, it helps keep track of the coordinates of the pixel (X, Y coordinates) and describes the horizontal and vertical sync signals to ensure appropriate movement of the ball. It handles the blanking logic which helps set the value of the black color on the VGA.

Purpose: This module decides the movement and the direction of the ball on the screen using the input given by the keyboard. It decides the movement of the ball when it reaches one end of the screen. It handles the sync of the coordinates, and handles the blanking logic.



Module: lab61soc.v *****_The table given below this section highlights in detail the components that went into the Platform Designer for Weeks 1, 2 and their functionality._

Inputs: clk_clk, reset_reset_n, [7:0] switch_wire_export, [1:0] button_wire_export

Outputs: [7:0] led_wire_export, sdram_clk_clk, [12:0] sdram_wire_addr,
[1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [1:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n

Inouts: [15:0] sdram_wire_dq

Description: This file is the resultant .sv file that is generated by the Platform designer tool for the first week. It includes the creation of our Peripherals - NIOS-II, On-Chip Memory, SDRAM, I/O Blocks and so on. The NIOS-II is our CPU, the SDRAM is the main memory where our instructions are stored, the SDRAM_pll is the extra module that handles the time delay while instructions transfer from the FPGA to the SDRAM (delay of 1ns), the PIOs are used to display the values on LEDs / Hexes and receive data from the switches.

Purpose: This file is the top-level of our entire week 1 implementation. It executes the descriptions we made via GUI in our Platform Designer.

**Module: lab62soc.v: ***_The table given below this section highlights in detail the components that went into the Platform Designer for Weeks 1, 2 and their functionality._

Inputs: clk_clk, reset_reset_n, usb_irq_export, usb_gpx_export, [1:0] key_external_connection_export

Outputs: [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, [7:0] sdram_clk_clk, [12:0] sdram_wire_addr, [7:0] keycode_export, usb_rst_export, [15:0] hex_digits_export, [13:0] leds_export,spi_0_MISO, spi_0_MOSI, spi_0_SCLK, spi_0_SS_n

Inouts: [15:0] sdram_wire_dq

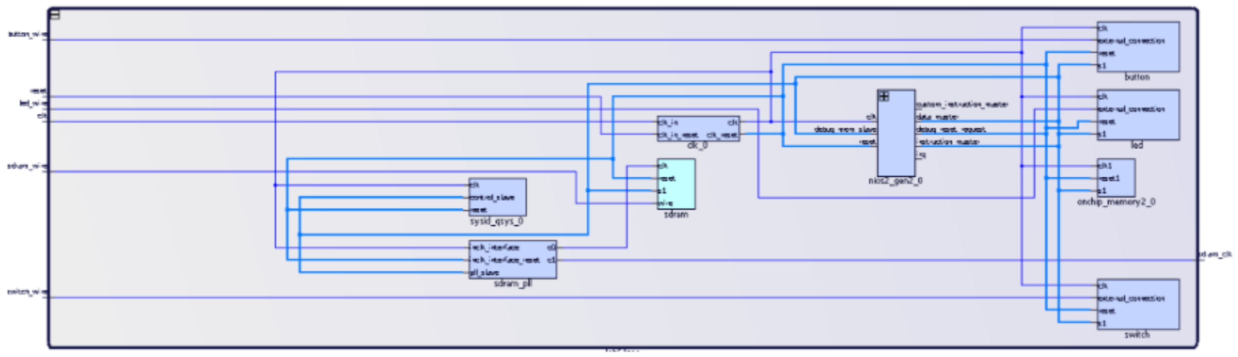
Description: This file is the resultant .sv file that is generated by the Platform designer tool for the first week. It includes the creation of our Peripherals - NIOS-II, On-Chip Memory, SDRAM, I/O Blocks and additional SPI blocks, USB interface, JTAG_UART, Timer, USB Interrupts.. The NIOS-II is our CPU, the SDRAM is the main memory where our instructions are stored, the SDRAM_pll is the extra module that handles the time delay while instructions transfer from the FPGA to the SDRAM (delay of 1ns), the PIOs are used to display the values on LEDs / Hexes and receive data from the switches. The newer components are the SPIO block (Serial Port Interface) which enables interaction between the USB Shield and the NIOS-II and hence enables reading and writing to the registers available on the MAX3421E Chip; the JTAG UART component allows debugging by performing the text transfer functionality; the USB Interrupt helps interrupt functionality when the keyboard keys are pressed, and the PIOs are used for displaying content on the HexDrivers.

<u>Purpose</u>: This file is the top-level of our entire week 2 implementation. It executes the descriptions we made via GUI in our Platform Designer.

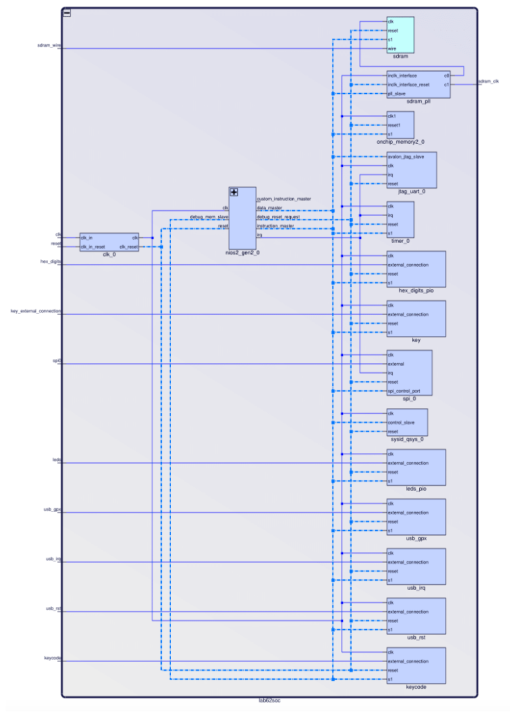
<u>5. System Level Block Diagram</u>

a. The Platform Designer view of the SoC module should be found here, describe the functionality of each block (including those which are part of the SoC, such as the memories).
b. Note that this is not trivial, as there are many components within the Platform Designer view.
c. You can separately describe the 'core' components common to Lab 6.1 and 6.2 first, and then describe the specific modules for week 1 and week 2.

1. Lab 6 Week 1 System Level Block Diagram:



2. Lab 6 Week 2 System Level Block Diagram:



<u>Common components and variables and their purposes in the execution of Lab 6:</u>

- Clk_0: Main clock for the FPGA, MAX3421E, SDRAM.
- Sysid_qsys_0: returns whether the system ID is correct so it correctly executes the software onto the hardware.
- [component]_pio: the address for the PIO displays is given by these which all the files refer to when trying to use them in our software.
- Usb_[fn]: handles the usb functions like the circuit from the keyboard, reset, and the interrupt.

Component	Name	Description
Nios-II Processor	nios2_gen2_0	This is our 32-bit CPU - the primary controller that is controlled by C, a high-level programming language. It is used to control the peripherals that primarily handle data and do not require fast processing times, and only need to transmit data.
SDRAM	sdram	The SDRAM (Synchronous Dynamic RAM) is used because of the limited availability of On-Chip memory. In week 1, we used it to write our value onto the LEDs and read values from the Switches.
SDRAM PLL	sdram_pll	The SDRAM is not on the FPGA and so needs a synchronous clock which is used in this module.

On Chip Memory	onchip_memory 2_0	The on chip memory was unused for the lab, but can be employed as it has much faster access times compared to the SDRAM because it is physically much closer on the FPGA than the SDRAM.
LEDs	led	This is a PIO Block (Parallel I/O Block) that outputs the accumulator during the first week of Lab 6.
Switches	sw	This is a PIO Block that took in user inputs for the accumulator in the first week of our lab.
Keys	Key_1, key_0	key_1 is the signal to accumulate LEDs and switches. key_0 indicated that the value of the LEDs needed to be reset.
SPI (W2)	spi_0	The SPI (System Peripheral Interface) lets us interact with USB Peripherals like the Keyboard, and mouse if necessary.
JTAG UART (W2)	jtag_uart	This allows character movement between the Computer and the FPGA, enabling transferring of text which avoids the slowing down of the CPU for tasks that do not require high processing powers.

6. 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.

Blinker:

The blinker code contains a pointer to the PIO block which accesses the LEDs set to outputs. It is initially set to zero to turn the LEDs off. It then enters an infinite loop. Within the while loop, there is a delay after which the LED pointer is ORed with 1 setting the Least Significant Bit to 1. This turns on the first LED on the board. Following another delay, it is ANDed with 0, turning the LED on. Since this portion of the code is within an infinite loop, the LED turns on and off at periodic intervals determined by the for loops.

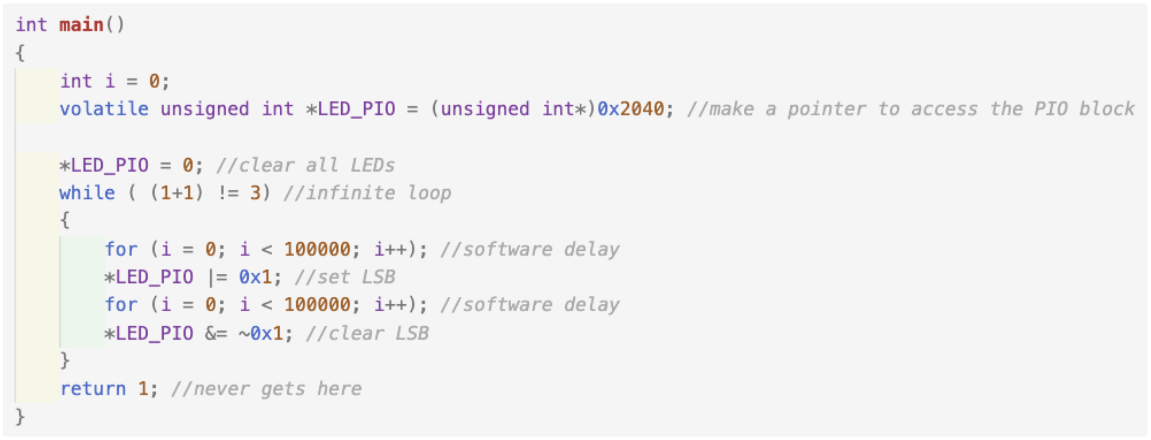
Accumulator:

First, pointers to PIO blocks are initialized allowing the C code to read from inputs Switches, and Reset, Accumulate keys and accordingly alter outputs LED. Once LEDs are initialized to zeros, the code enters an infinite loop. It first checks if the Reset Key is 0(Note that keys are active low) and accordingly sets the LEDs to 0 thereby resetting it. If the reset key is not pressed, then it checks whether the accumulate key is pressed. If pressed, the code loops till the key is depressed. The LEDs are then updated by adding the value set by switches to it.

b. Describe the code you needed to fill in for the xUSB/SPI portion of the lab for Lab 6.2

Week 1: We had been given a main.c file which used to make the LED peripheral on the FPGA light up. We altered this to create an accumulator which would sum the values on the LEDs with the ones on the switch peripherals. It also could reset the values on the LEDs to start over.

Main.c for the blinker code:

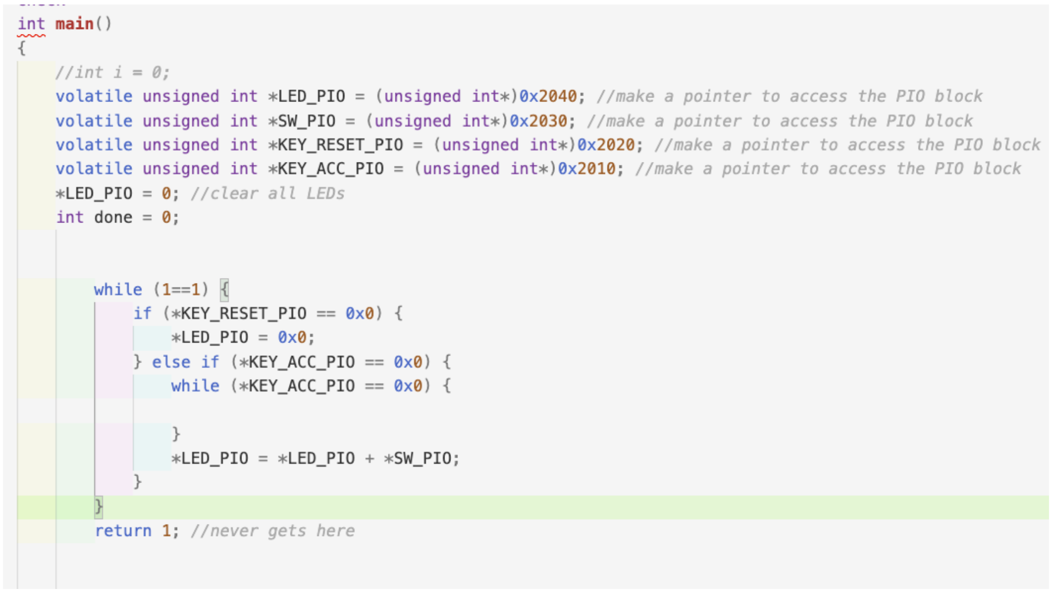


```c
int main()
{
    int i = 0;
    volatile unsigned int *LED_PIO = (unsigned int*)0x2040; //make a pointer to access the PIO block

    *LED_PIO = 0; //clear all LEDs
    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
}
```

Week 2: We had to ensure that our USB peripheral could work with the MAX3421 and so we had to create 4 functions that could either write to or read from the master to slave and vice versa. We used the *alt_avalon_spi_command* in order to perform these functions as described below.

Main.c code for the accumulator:

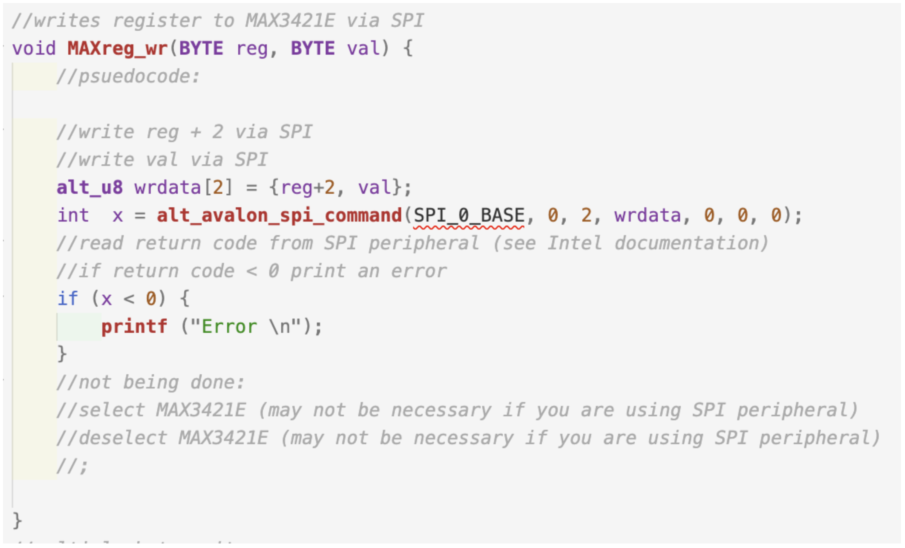


```
int main()
{
    //int i = 0;
    volatile unsigned int *LED_PIO = (unsigned int*)0x2040; //make a pointer to access the PIO block
    volatile unsigned int *SW_PIO = (unsigned int*)0x2030; //make a pointer to access the PIO block
    volatile unsigned int *KEY_RESET_PIO = (unsigned int*)0x2020; //make a pointer to access the PIO block
    volatile unsigned int *KEY_ACC_PIO = (unsigned int*)0x2010; //make a pointer to access the PIO block
    *LED_PIO = 0; //clear all LEDs
    int done = 0;


        while (1==1) {
            if (*KEY_RESET_PIO == 0x0) {
                *LED_PIO = 0x0;
            } else if (*KEY_ACC_PIO == 0x0) {
                while (*KEY_ACC_PIO == 0x0) {

                }
                *LED_PIO = *LED_PIO + *SW_PIO;
            }
        }
        return 1; //never gets here
```

`1. void MAXreg_wr(BYTE reg, BYTE val;`



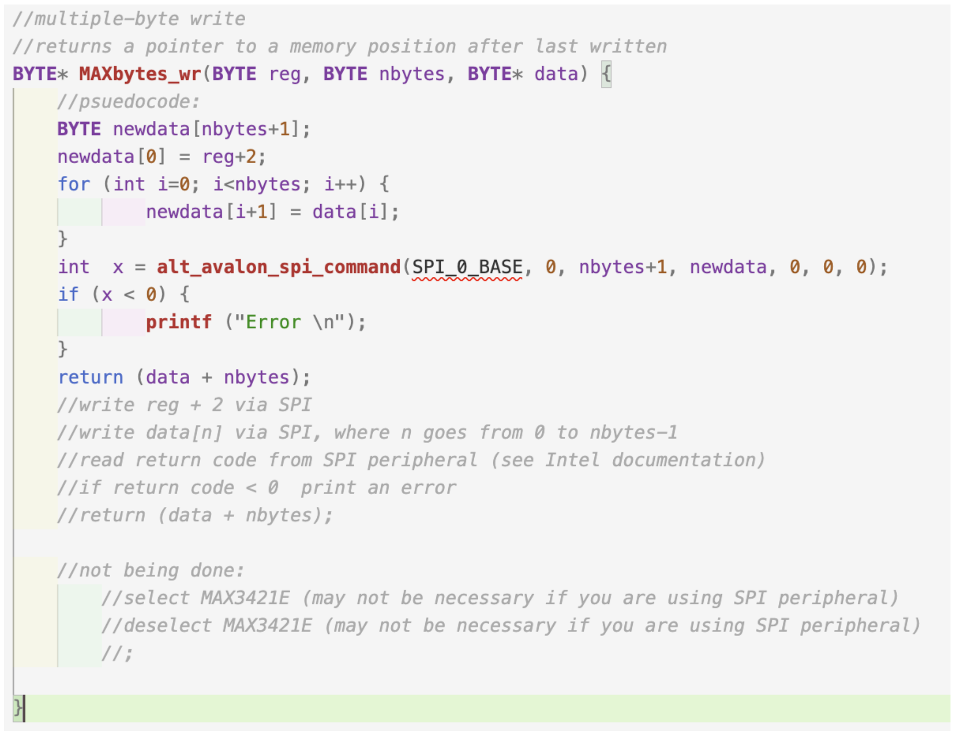
```
//writes register to MAX3421E via SPI
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:

    //write reg + 2 via SPI
    //write val via SPI
    alt_u8 wrdata[2] = {reg+2, val};
    int  x = alt_avalon_spi_command(SPI_0_BASE, 0, 2, wrdata, 0, 0, 0);
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    if (x < 0) {
        printf ("Error \n");
    }
    //not being done:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //;

}
```

This function writes a 1 BYTE value ‘val’ to a register in the MAX3421E via SPI. This is done by first writing the value of the reg + 2 to specify the address, and the actual value to be written. The SPI function is then called, the writedata is passed and is set to perform a write operation.

2. BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data);



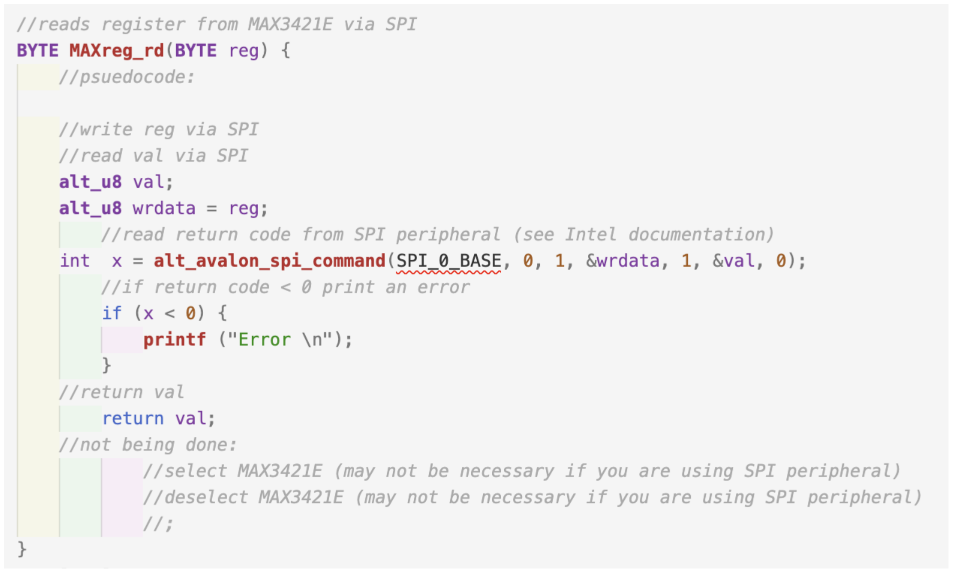
```
//multiple-byte write
//returns a pointer to a memory position after last written
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    BYTE newdata[nbytes+1];
    newdata[0] = reg+2;
    for (int i=0; i<nbytes; i++) {
        newdata[i+1] = data[i];
    }
    int  x = alt_avalon_spi_command(SPI_0_BASE, 0, nbytes+1, newdata, 0, 0, 0);
    if (x < 0) {
        printf ("Error \n");
    }
    return (data + nbytes);
    //write reg + 2 via SPI
    //write data[n] via SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0  print an error
    //return (data + nbytes);

    //not being done:
        //select MAX3421E (may not be necessary if you are using SPI peripheral)
        //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
        //;

}
```

This function writes a data array of length nbytes given by the pointer data. First a new array is created consisting of the data to be written. First, the register address reg + 2 is written followed by the N bytes of data. The SPI command is then invoked, the new array is passed, and it is set to write. The entire function returns the pointer to the memory address after the last write function was executed.

3. BYTE MAXreg_rd(BYTE reg);

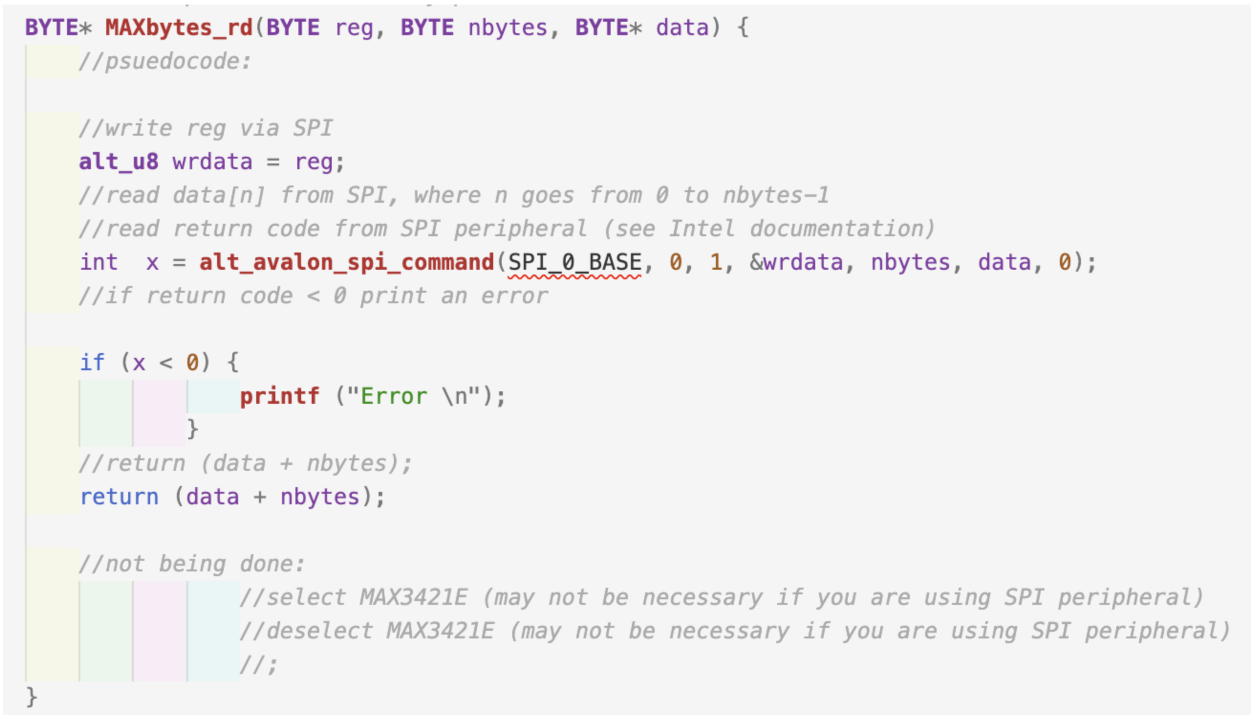


```
//reads register from MAX3421E via SPI
BYTE MAXreg_rd(BYTE reg) {
    //psuedocode:

    //write reg via SPI
    //read val via SPI
    alt_u8 val;
    alt_u8 wrdata = reg;
        //read return code from SPI peripheral (see Intel documentation)
    int  x = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &wrdata, 1, &val, 0);
        //if return code < 0 print an error
        if (x < 0) {
            printf ("Error \n");
        }
    //return val
        return val;
    //not being done:
            //select MAX3421E (may not be necessary if you are using SPI peripheral)
            //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
            //;
}
```

This function reads a register and returns the value stored. To do this we invoke the SPI command. The address of the register is written, following which the 1 BYTE value is read and returned.

4. `BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data);`



```
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:

    //write reg via SPI
    alt_u8 wrdata = reg;
    //read data[n] from SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    int  x = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &wrdata, nbytes, data, 0);
    //if return code < 0 print an error

    if (x < 0) {
            printf ("Error \n");
        }
    //return (data + nbytes);
    return (data + nbytes);

    //not being done:
            //select MAX3421E (may not be necessary if you are using SPI peripheral)
            //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
            //;
}
```

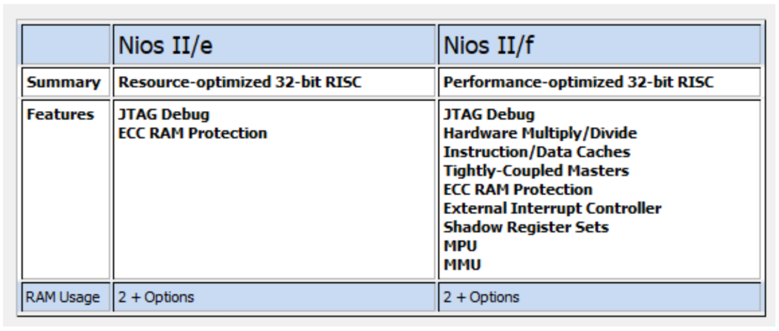
This function reads nbytes bytes of data and stores it in an array data. It invokes the SPI command. The starting register address is first written and then nbytes of data is read and stored in ‘data’. It then returns the pointer to the memory position where the last data was written.

---

7. Answers to all INQ & Post lab questions

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

A. The NIOS II/e is the 32-bit **Resource-optimized** RISC, whereas the NIOS II/f is the **Performance-optimized** RISC. This means that the 'e' version is the version that employs lesser functions, and uses lesser number of resources and LEs. It is slower than its counterpart and offers lesser features. One of the main differences is how the 'f' version enables multiplication / division operations, and has an incredibly fast cache, so it does not go to memory to access instructions.



	Nios II/e	Nios II/f
Summary	Resource-optimized 32-bit RISC	Performance-optimized 32-bit RISC
Features	JTAG Debug ECC RAM Protection	JTAG Debug Hardware Multiply/Divide Instruction/Data Caches Tightly-Coupled Masters ECC RAM Protection External Interrupt Controller Shadow Register Sets MPU MMU
RAM Usage	2 + Options	2 + Options

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

A. On chip memory is closer to the CPU compared to the SDRAM. This means that the access time for the on-chip memory is much lesser and so it is much faster and has lesser latency. Hence, the entire CPU can run much faster than if the memory instructions were being fetched from the On-Chip Memory instead of the SDRAM.

Q. Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?

A. A 'Vonn Neumann' machine is one where the instructions and the data are stored in the primary memory and there is a singular bus which handles the transferring of this data and the addresses. Hence, only one-way data manipulation and fetching is possible. In the 'Pure Harvard' machine, the instructions are in a separate memory location from the data, and different buses are used to access these. It is a fully duplex model because fetching and writing can be performed simultaneously. Our function uses the 'Modified Harvard' machine, because in our situation, we use the property of the Vonn Neumann model where the data and instructions are both stored in the same memory location (SDRAM) but we gave a separate data bus and a separate instruction bus.

Q. 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?

A. The OCM requires access to the data and program bus because it needs to fetch / write instructions and needs to access data and push data in. However, for our LEDs, they have assigned base addresses that are used to access them in the C code. Since they are peripherals that just act as output, they do not need the information on which instructions are being executed and hence only need access to the data bus that contains the data to be displayed. This is done by writing that data to the base address which is memory mapped and thus gets displayed to the LEDs.

Q. Why does SDRAM require constant refreshing?

The SDRAM is volatile - it means that it loses its data because it is made of capacitors which lose their charge and thus data over time. If we refresh the SDRAM, we ensure that the data is not lost in memory.

Q. What is the maximum theoretical transfer rate to the SDRAM according to the timings given?

The maximum theoretical transfer rate is:

$$\frac{\text{Data width (bits)}}{\text{Access time}} * \frac{1 \text{ byte}}{8 \text{ bits}} = \frac{32}{5.4\times10^{-9}} \times \frac{1}{8}$$

$$= 740.74 \text{ Mb/s}$$

Q. The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?

The SDRAM consists of capacitors that eventually lose the charge collected across them over time (discharging) and so to prevent the data from being lost, we need to ensure that the capacitors get charged before they lose their data and hence the value of refresh rate cannot be below 50MHz in order to ensure the values are refreshed often enough that the data is never lost and the access of the data results in valid values.

Q. 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? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.

The SDRAM is not on the chip, it is located further away. Hence, attempting to access data on it takes more time. This means that our access time will have a delay in it (1ns) that needs to be offset. Thus, when we access the data from the SDRAM we need to make sure that there is a delay across the clocks for accessing time, making sure that the SDRAM information is collected 1ns behind to match the access time.

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

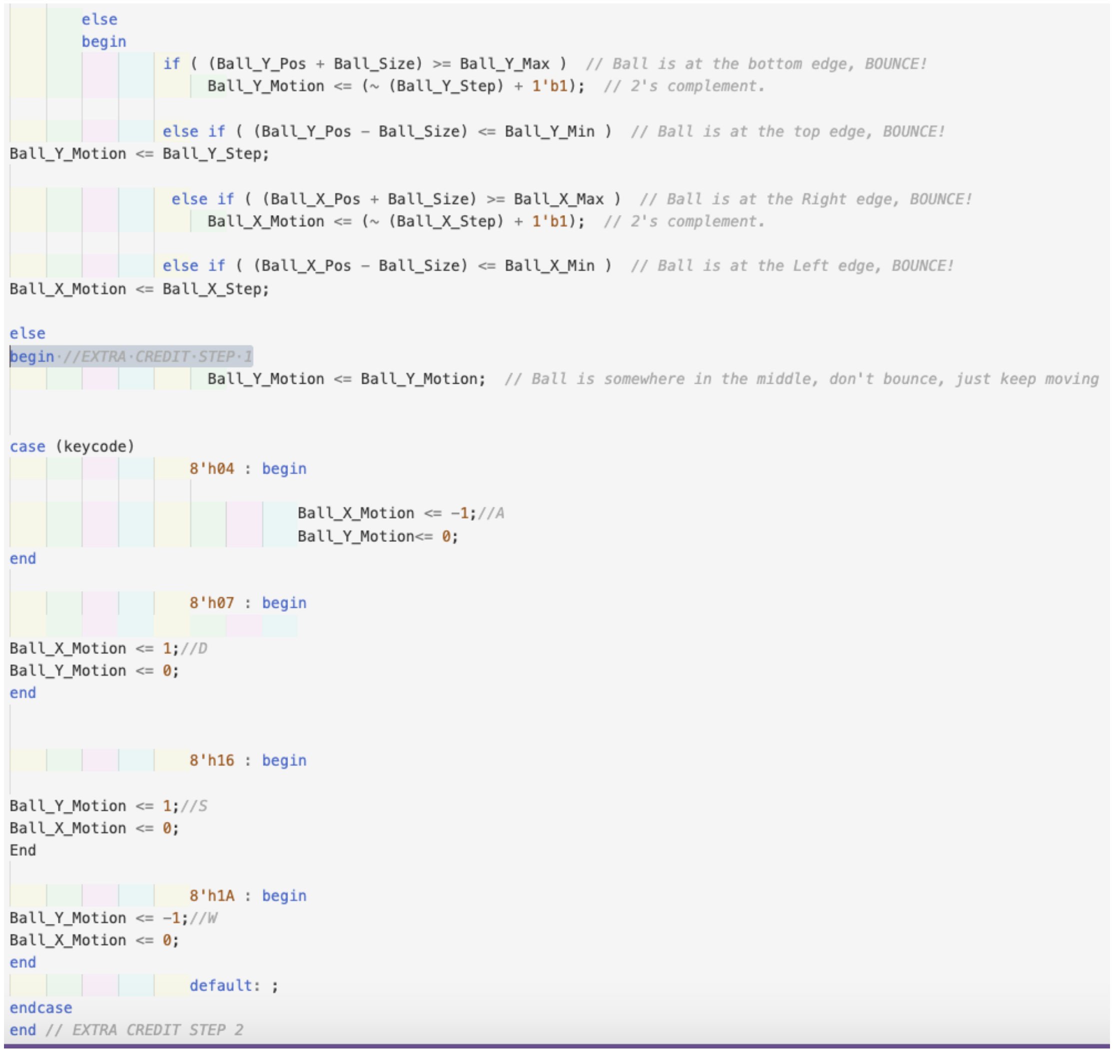
The address instruction starts from x0800 0000. The first block of memory is saved for the OCM (On-Chip-Memory) because memory assigned in the Platform Designer is contiguous and so since we do not use OCM, it needs to have the first few address blocks. We assign this address because in this situation, the NIOS-II knows where to start its execution from in cases of interruptions, exceptions, or resets.

Q. 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:
Const int my_constant[4] = {1, 2, 3, 4}

File extension name	***Meaning***
.bss (Block Starting Symbol)	The file in which allocated variables that have not been initialized are stored, including global variables. Ex: double val;
.heap	The place in memory where the dynamic memory allocated variables are placed, and this dynamic allocation takes place. Ex: double* ptr = (double*) malloc(sizeof(double));
.rodata	This contains static constant values, and is read-only by most architectures. Ex: const double val = 3.0;
.rwdata	This segment stores static and global variables. It is non-executable and is of read-write type. Ex: double val = 3.0;
.stack	The place in memory where local variables are stored contiguously. It has the functions record, and the data of that function (parameters, return types etc). Ex: double subtr(double a, double b){};
.text	This segment is in the memory which stores executable instructions and texts / strings. Ex: String str = “string”.

<u>Extra Credit:</u>

We were asked to modify our ball.sv module to fix a glitch that occurred - our ball would disappear off the screen if we pressed and held the W (Up) key for too long as the ball traveled to the top edge. We modified the ball.sv file as follows: we added a begin and end instruction code for our code's else part (the begin is highlighted, and the end is the last line of our code) The 4 if statements check whether the ball is at the edges of the screen. Initially, the case(keycode) was placed outside of the else condition. Since the code is in an always_ff procedure, pressing a key would continue making the ball move in the same direction, rather than checking if it was at the edge and making it bounce back. We only want the ball to respond to key presses if the ball is not at the edge, which is why we add the case statement as a part of the overall else statement, by adding begin and end statements.
(The extra credit lines are labeled as STEP 1, STEP 2).



```
        else
        begin
              if ( (Ball_Y_Pos + Ball_Size) >= Ball_Y_Max )  // Ball is at the bottom edge, BOUNCE!
                  Ball_Y_Motion <= (~ (Ball_Y_Step) + 1'b1);  // 2's complement.

              else if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min )  // Ball is at the top edge, BOUNCE!
Ball_Y_Motion <= Ball_Y_Step;

               else if ( (Ball_X_Pos + Ball_Size) >= Ball_X_Max )  // Ball is at the Right edge, BOUNCE!
                  Ball_X_Motion <= (~ (Ball_X_Step) + 1'b1);  // 2's complement.

              else if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min )  // Ball is at the Left edge, BOUNCE!
Ball_X_Motion <= Ball_X_Step;

else
begin //EXTRA CREDIT STEP 1
                  Ball_Y_Motion <= Ball_Y_Motion;  // Ball is somewhere in the middle, don't bounce, just keep moving


case (keycode)
                 8'h04 : begin

                                  Ball_X_Motion <= -1;//A
                                  Ball_Y_Motion<= 0;
end

                 8'h07 : begin

Ball_X_Motion <= 1;//D
Ball_Y_Motion <= 0;
end


                 8'h16 : begin

Ball_Y_Motion <= 1;//S
Ball_X_Motion <= 0;
End

                 8'h1A : begin
Ball_Y_Motion <= -1;//W
Ball_X_Motion <= 0;
end
                 default: ;
endcase
end // EXTRA CREDIT STEP 2
```

8. Design Resources and Statistics

LUT	3096
DSP	10
Memory (BRAM)	50520
Flip-Flop	2449
Frequency	85.3 MHz
Static Power	96.51 mW
Dynamic Power	59.43 mW
Total Power	175.90 mW

---

9. Conclusion

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it?

Our design was enabling the integration of external peripherals to our FPGA in order to access data and memory from external sources in order to use it in our design. It also taught us how to use these peripherals in C code so we could use these in our NIOS-II processor to perform functions with the data received from these devices (ike the USB Keyboard/Mouse) and display the output of the information from the code to other peripherals (like the VGA Monitor). This lab enables us to successfully learn how to perform appropriate tasks that can be extended to further functions and explores the boundary of hardware and software and make them work in unison to extend a ball movement on the screen, which we will work on expanding into more difficult tasks in the future labs and projects.

b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester?

There was not much difficulty in comprehending the material provided to us, however, a simpler version of the SPI Core explanation along with documentation on how to understand the purposes of the various parameters (which was explained in the lectures) being in document form would be helpful. In addition, the timing section of this lab was more difficult - if we could instead be provided more information on MISO / MOSI / SS and how they work with the timer, this lab would be more informative and could extend to future understanding of data and instruction transfer.
