University of Illinois at Urbana-Champaign

ECE 385

LAB 6

LAB REPORT

by Shubham Gupta & Devul Nahar (sg49 & danahar2)

3/04/2022

Introduction:

For Lab 6.1, we used the NIOS II/e processor to perform functions like reading and writing to our memory mapped peripherals like the switches and the LEDs. These memory mapped locations were established using the Parallel I/O IPs provided by Intel. We had written C code that was compiled to NIOS II's ISA and stored to SDRAM. This is how we implemented our adder via the NIOS II.

For Lab 6.2, we used the NIOS II/e processor to perform functions like reading and writing to the registers in the USB shield. This was done by establishing the SPI protocol. Further, we used NIOS II to display the keycodes on the hex displays via PIOs. The remaining hardware was used to generate the VGA signals.

The Universal Serial Bus interface was used to connect the keyboard to the FPGA. This wasn't done directly, we established SPI protocol to communicate between the USB shield and the NIOS II. A USB has four signals: VDD, D+, D-, and GND. The Video Graphic Array is organized as a matrix of 640 x 480 pixels. The VGA signal has a vertical sync pulse, horizontal sync pulse, and a blank signal along with RGB values of the pixel. The electron beam swipes from left to right and top to bottom to paint one frame of the video. We use a 25 MHz clock signal to drive the VGA signal generated by the VGA controller.

Written Description and Diagrams of NIOS-II System:

Platform Designer:

This is the tool we used to generate our IP modules and link the inputs and outputs together. We have instantiated the on-chip memory of the FPGA and the NIOS II initially. This is the heart of our design. Then we have added PIOs which are parallel input outputs for our peripherals. Now, to store our C code we then add a SDRAM. Since SDRAM is

physically away from the chip to account for the delay of 1ns in the clock cycle we have added a PLL.

- **NIOS II:** This is the System on Chip that we are using for this lab. It has a 32-bit word size and allows us to write C code which the compiler converts to the ISA for NIOS II. It is an IP provided by Intel that is run on the FPGA.
- **SDRAM:** This is the Synchronous Dynamic Random Access Memory which stores our compiled code that is to be executed by the NIOS II.
- **PIO Blocks:** It is the memory mapped IO we use to read and write data from the CPU to the external peripherals like the switches and LEDS.
- **PLL:** To account for the delay in the clock cycle due to physical path difference between OCM and the SDRAM, we use a -1ns delay via this IP.
- **SPI:** We use Serial Protocol Interface in lab 6.2 to communicate with the MAX3421E chip that connects the Keyboard.

Description of Lab 6.1 I/O:

In Lab 6.1, we used only Parallel I/O blocks which is an IP given by Intel. We use these PIO blocks to create memory mapped addresses so that the CPU can perform direct reads and writes. This is provisioned by the platform designer as we assign the base addresses to the PIOs. We have connected the data bus to the PIO from the NIOS II. The LEDs and the Switches are the two PIO blocks that we have instantiated.

Description of NIOS II interactions with both the MAX3421E USB & VGA:

We will now see how we interact with the MAX3421E chip and the VGA via the NIOS II:

- **USB:** We have written the C code for the USB drivers. The NIOS II interacts with the USB chip via the SPI protocol. The provided MAX3421E chip is the SPI slave to the

NIOS II which is further connected via USB to the keyboard. The driver code is discussed in the further sections.

- **VGA:** The VGA is Video Graphics Array; it is a 2-dimensional array of 640 x 480 pixels. The VGA signal has a vertical sync pulse, horizontal sync pulse, and a blank signal along with RGB values of the pixel. The electron beam swipes from left to right and top to bottom to paint one frame of the video. We use a 25 MHz clock signal to drive the VGA signal generated by the VGA controller.

SPI Protocol:

As discussed earlier, we use the Serial Peripheral Interface protocol to exchange data between the NIOS II and the MAX2431E chip, to establish the USB connection. The SPI has a synchronous bus having the following signals:

- **CLK**: This is the signal that the master sends to the slave upon reading/writing every bit.
- **MOSI**: Master out slave in, in our case the data is sent from NIOS II (Master) to MAX2431E (Slave).
- **MISO**: Master in slave out, in our case the data is sent from MAX2431E (Slave) to NIOS II (Master).
- **SS**: It is an active low signal used to select the salve device.

SPI protocol has a full duplex capability where both read and write can happen at the same time. However, for this lab we are using half duplex. We are reading and writing data to MAX2431E chip by the functions that we wrote in the MAX2431E.c file using the alt_avalon_spi_command function provided by Intel.

Description of C code:

In Lab 6.1, we just modified the main.c file to write the code for the accumulator. We accessed the PIOs via the pointer to the base address.

In Lab 6.2, we modified the MAX3421E.c file to complete the SPI driver implementations for the USB protocol. We did this using by calling the helper function alt_avalon_spi_command.

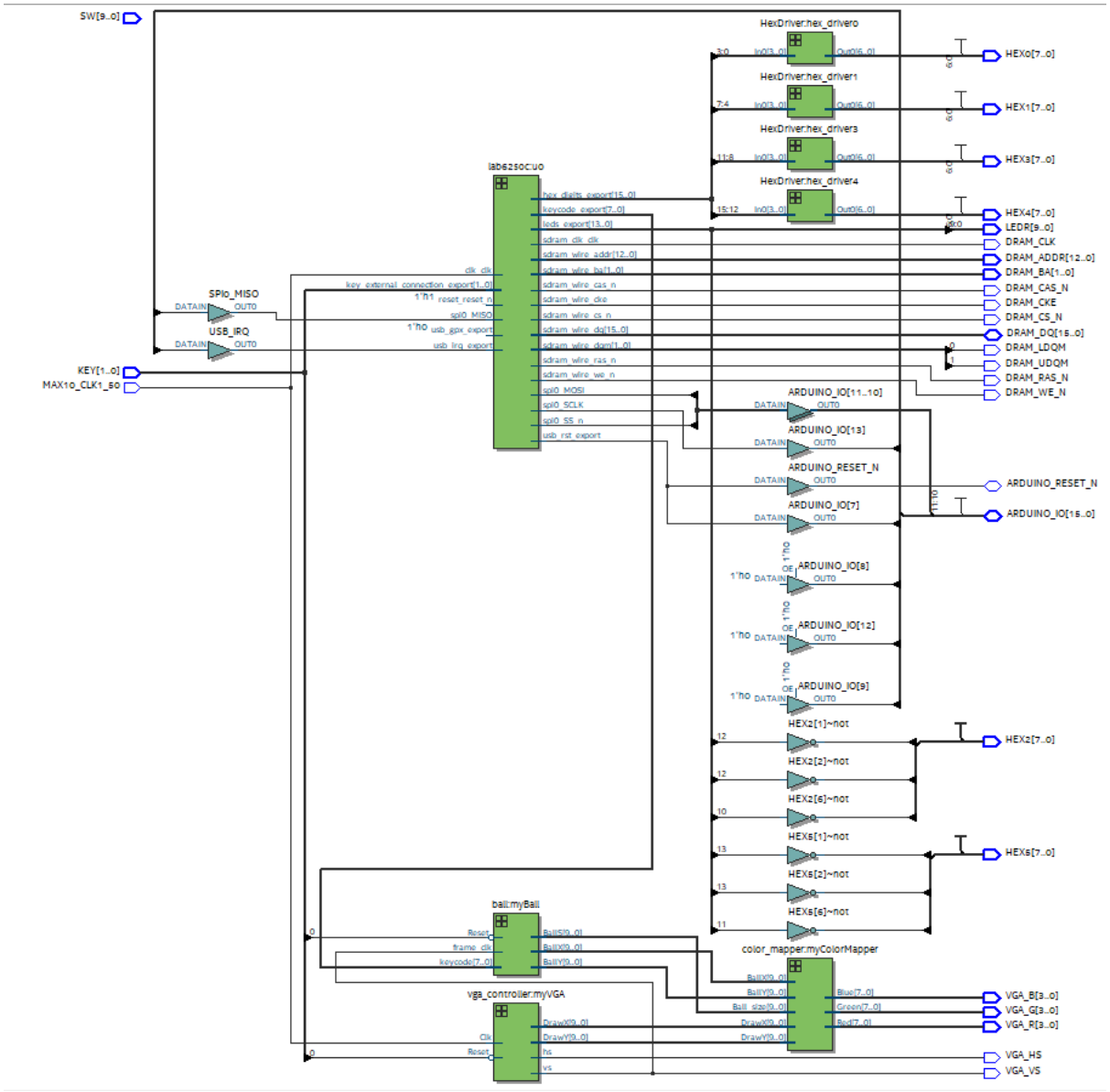
Description of VGA Controller, Ball and Color Mapper Module:

The VGA controller handles the generation of the VGA signals. One of its key features is to make a 25 MHz clock from the 50 MHz clock. Then this module keeps track of the horizontal and vertical pixel coordinates. It also resets them if they reach the end of horizontal and vertical pixel bounds. Another function is to generate the horizontal and vertical pixel sync pulse along with blanking signal.

The Ball module receives the frame clock and XY coordinates from the VGA controller and decides the new location (XY coordinates) of the ball based on the keycode pressed and the boundary conditions.

The color mapper takes in the XY coordinates of the VGA controller and as well as the Ball to decide whether to paint the background or the foreground.

Top Level Block Diagram:



Written Description of all .sv modules:

i. **Module**: VGA_controller.sv

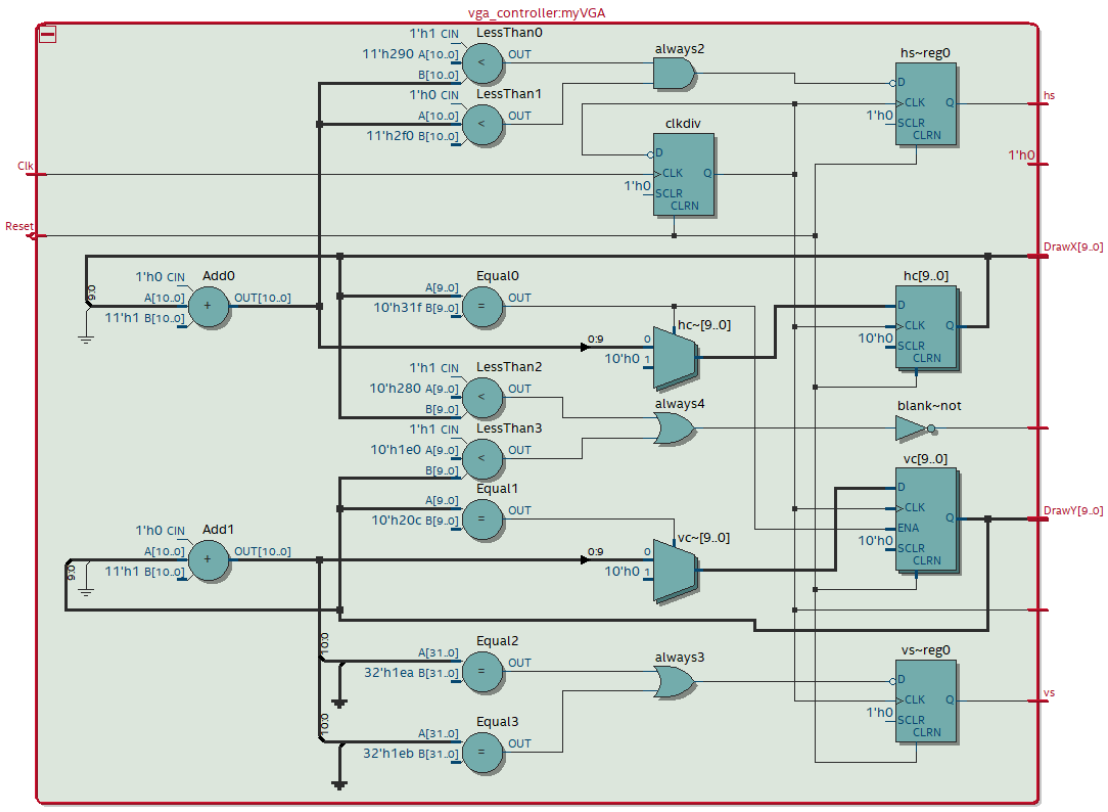
Inputs: Clk, Reset

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

Description: This module handles the generation of the VGA signals. One of its key features is to make a 25 Mhz clock from the 50 Mhz clock. This is done by using a always_ff block. Then this module keeps track of the horizontal and vertical pixel

coordinates. It also resets them if they reach the end of horizontal and vertical pixel bounds. Another function is to generate the horizontal and vertical pixel sync pulse. Lastly, we use if else block to generate an active low blank signal.

Purpose: It is the key component that generates the VGA signals. It provides us the DrawX and DrawY cordinates which are extremely important to draw anything on the screen. This module also generates a 25 MHz pixel clock. It provides info for the blanking interval as well, so that we do not draw anything.



ii. **Module**: Color_Mapper.sv

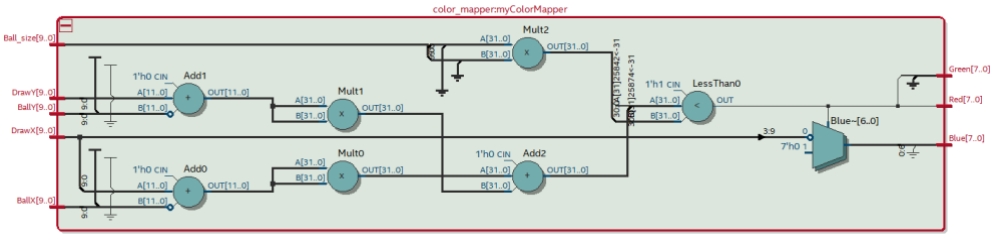
Inputs: [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] DrawY, [9:0] Ball_size

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

Description: This module generates the appropriate color values (Red, Green, Blue) based on the DrawX and DrawY along with BallX and BallY. We use if else blocks to check if the ball is supposed to be drawn on the given DrawX and DrawY

coordinates, based on ball size, BallY, and BallX inputs. If the ball is not supposed to be drawn, then we simply draw the background.

Purpose: It is primarily used to generate the color (RGB) for the pixel depending on X and Y location of the pixel.



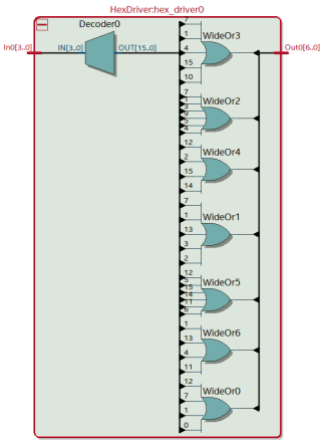
iii. **Module**: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: We use a UNIQUE switch case to map the binary values to the hexadecimal display. Given to us from previous labs.

Purpose: It maps the binary values to the hexadecimal LED display. Used for displaying values stored in registers. For user interface on the FPGA.



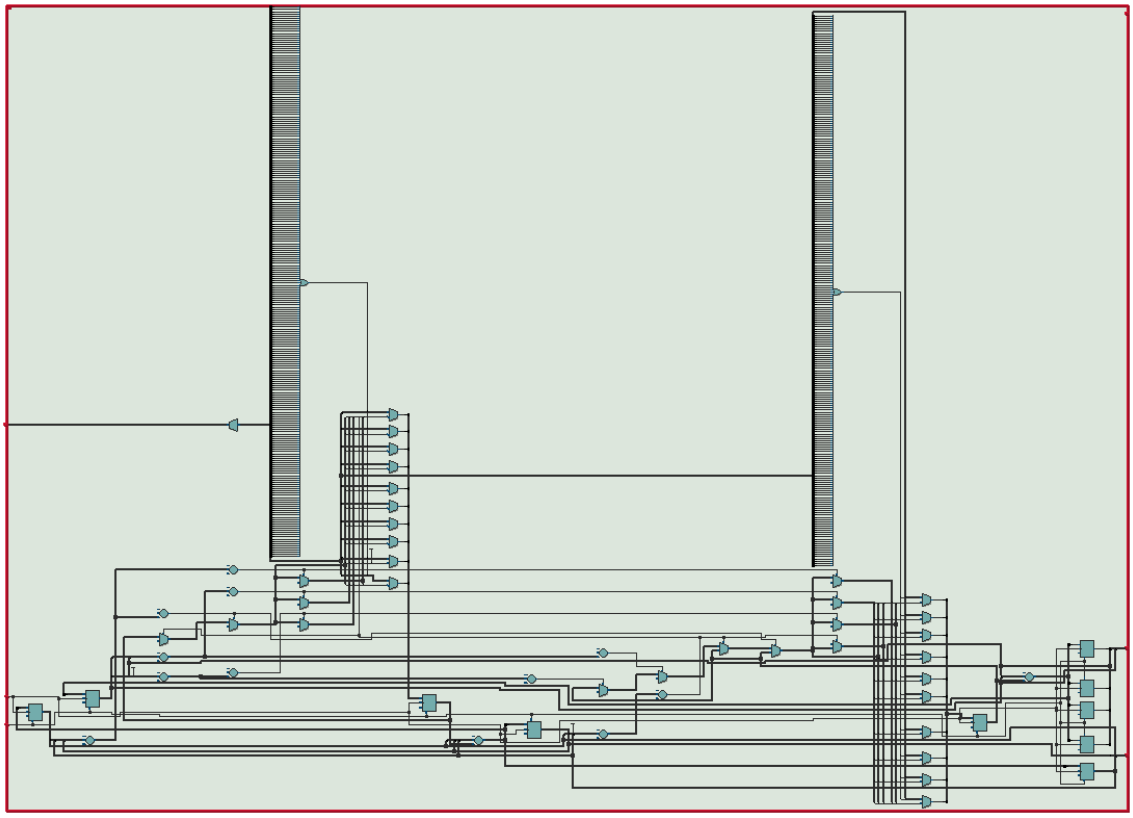
iv. **Module**: Ball.sv

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

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

Description: This is the module that generates the X and Y coordinates of the ball. We used always_ff to calculate new positions of the ball based on the boundary conditons and update the motion in X and Y direction. Followed by updating the X and Y coordinates based on motion X and Y, along with step X and Y. It is dependent on the keycode. We have also taken care of the edge cases where the keycode is continuously pressed and the ball should not escape the screen.

Purpose: Its primary purpose is to generate the new position (X and Y) for the ball based on the keycode and boundary conditions.



v. **Module**: Lab62soc.v

Inputs: clk_clk, [1:0] key_external_connection_export, reset_reset_n, [15:0] sdram_wire_dq, spi0_MISO, usb_gpx_export, usb_irq_export

Outputs: [15:0] hex_digits_export, [7:0] keycode_export, [13:0] leds_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, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export

Description: We have instantiated the following IPs in the platform designer which make up this module. This top-level module is so that we can connect inputs and outputs of these IPs. The IPs include: NIOS ll/e, OCM, PIOs, JTAG UART, SDRAM, PPL for SDRAM, SPI, TIMER, and USB INTERRUPTS. NIOS ll/e is our processor. JTAG UART is for debugging purposes that we use to debug the NIOS ll. SDRAM is the primary memory where our compiled code is stored as instructions. PLL is to accommodate the physical time offset due to the path difference between the SDRAM and the OCM. SPI is the IP we used to communicate between the USB shield and NIOS ll. USB interrupts are so that we can send interrupt requests whenever keys are pressed on the keyboard. PIOs are used to display the content to HexDrivers as memory mapped I/O.

Purpose: This is the top-level module exported from the Platform designer for Lab 6.2 and implements all the IPs generated in the Platform designer to the FPGAs. These include the NIOS ll/e, PIOs, JTAG UART, SDRAM, PPL for SDRAM, SPI, TIMER, and USB INTERRUPTS.

vi. **Module**: Lab61soc.v

Inputs: [1:0] button_wire_export, clk_clk, reset_reset_n, [15:0] sdram_wire_dq, [7:0] switch_wire_export

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

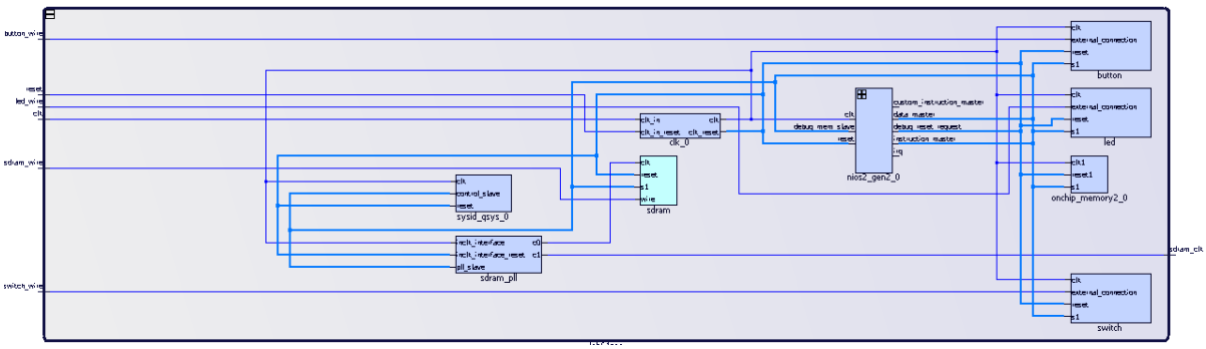
Description: We have instantiated the following IPs in the platform designer which make up this module. This top-level module is so that we can connect inputs and

outputs of these IPs. The IPs include: NIOS ll/e, PIOs, OCM, SDRAM, and PPL for SDRAM. NIOS ll/e is our processor. SDRAM is the primary memory where our compiled code is stored as instructions. PLL is to accommodate the physical time offset due to the path difference between the SDRAM and the OCM (1ns). PIOs are used to display the content to LEDs as memory mapped I/O. Similarly, a PIO for the switches is used as well.

Purpose: This is the top-level module exported from the Platform designer for Lab 6.1 and implements all the IPs generated in the Platform designer to the FPGAs. These include the NIOS ll/e, PIOs, OCM, SDRAM, and PPL for SDRAM.

System Level Block Diagram:

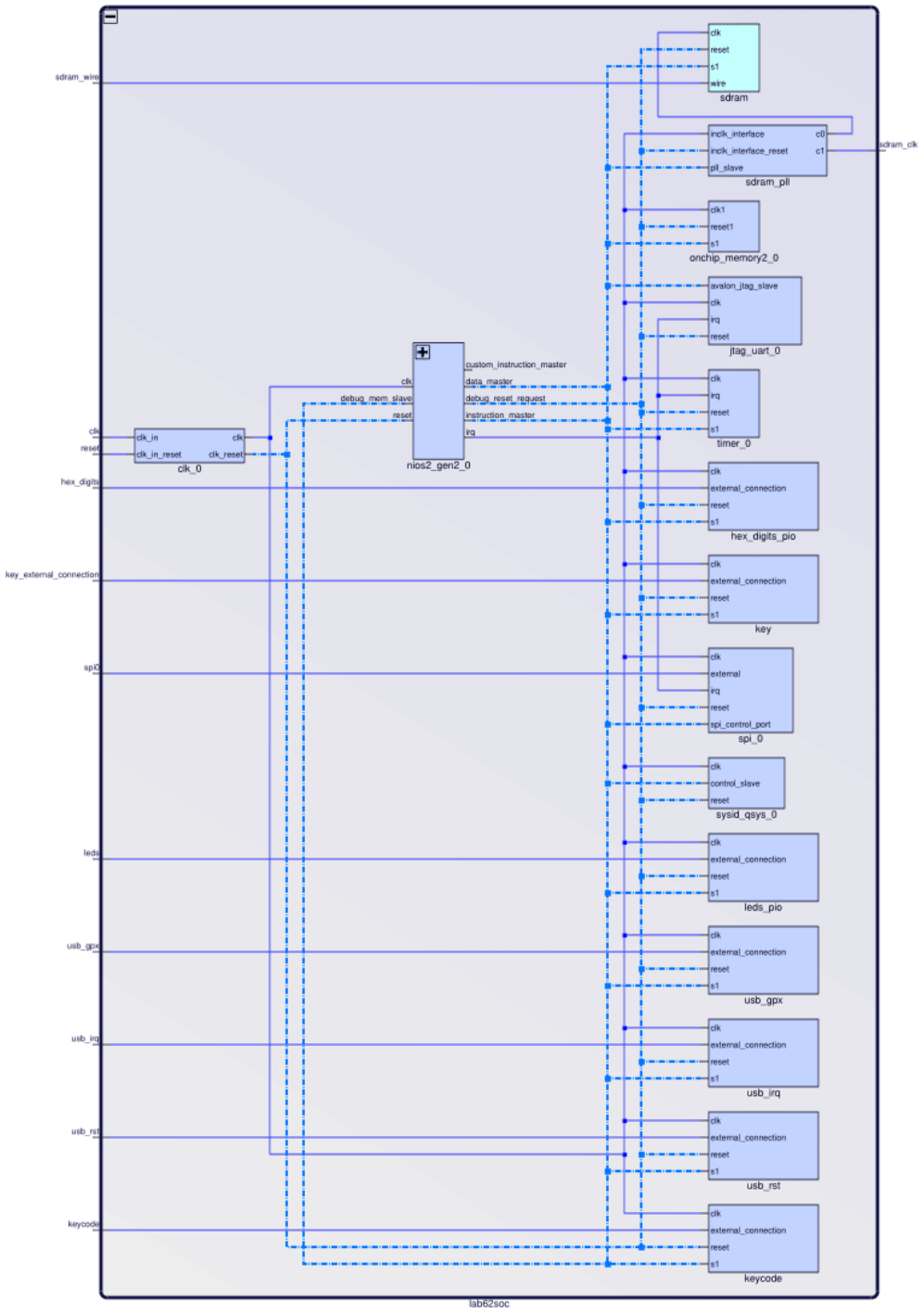
Lab 6.1:



Components	Description
Clk_0	Used to generate clock for the FPGA, SDRM, as well as the MAX3421E.
Nios2_gen2_0	The CPU layer that behaves like slc3 but is more robust. This CPU controls all functionality within the machine. It is responsible for parsing and acting on instructions that it fetches from memory.

Sdram	SDRAM is the external memory chip connected to the FPGA board. The NIOS II is stored here instead of the onchip memory. Instructions as well as data is fetched from here.
Sdram_pll	Since sdram is further away from the onchip memory and because it requires a clock signal, the PLL handles the timing offsets that are caused due to board layout. This component is very important as the SDRAM requires precise timings due to the fact that it is made up of capacitors.
Onchip_memory2_0	Onchip memory is a small memory block that resides within the FPGA chip. Since onchip memory is very close to the actual FPGA, it is faster access times. We don't use this in this lab, but we could for future labs.
Sysid_qsys_0	Used to check the system ID so that it can maintain the compatibility between software and hardware. It functions by giving a serial number which the software cross checks with the software.
led	Used to display the accumulated sum as well as the counter on the LED displays. Assigned an address.
Switch	Used to enter numbers in binary to add to the accumulated sum. It is mapped to the switches on the FPGA.
Button	This block is for the 2 buttons on the FPGA and allows us to use these buttons in the software. The first button allows us to reset our accumulator and set it to 0. The second allows us to add the number on the switches to our sum.

Lab 6.2:



Components	Description
Clk_0	Used to generate clock for the FPGA, SDRM, as well as the MAX3421E.

Nios2_gen2_0	The CPU layer that behaves like slc3 but is more robust. This CPU controls all functionality within the machine. It is responsible for parsing and acting on instructions that it fetches from memory.
Sdram	SDRAM is the external memory chip connected to the FPGA board. The NIOS II is stored here instead of the onchip memory. Instructions as well as data is fetched from here.
Sdram_pll	Since sdram is further away from the onchip memory and because it requires a clock signal, the PLL handles the timing offsets that are caused due to board layout. This component is very important as the SDRAM requires precise timings due to the fact that it is made up of capacitors.
Jtag_uart_0	JTAG UART allows serial character communications streams between a host PC and the FPGA. In addition, the JTAG UART core is also connected to the Interrupt controller as transferring text over the console is slow and we don't want to slow down the CPU.
Timer_0	This block handles the timer for the USB Port. The timer is essential to the USB driver in order to keep track any time-out that the USB needs.
Spi_0	SPI is known as the Serial Port Interface, which allows us to read and write to the register on the MAX3421E chip.
Key	Key is the input to FPGA. It specifies the key entered on the keyboard. Key is 2 bits to specify the 4 different relevant keys the user can enter (WSAD).

Keycode	The 2 bits generated by the Key block is received by the keycode, which generates a 8 bit ASCII code. This ascii code is then used to change the direction of the ball on the screen.
Onchip_memory2_0	Onchip memory is a small memory block that resides within the FPGA chip. Since onchip memory is very close to the actual FPGA, it is faster access times. We don't use this in this lab, but we could for future labs.
Sysid_qsys_0	Used to check the system ID so that it can maintain the compatibility between software and hardware. It functions by giving a serial number which the software cross checks with the software.
Hex_digits_pio	The PIO address for Hex displays. A set of addresses are assigned for this PIO, so that we can use them in software.
leds_pio	The PIO address for LED displays. A set of addresses are assigned for this PIO, so that we can use them in software.
Usb_gpx	Connects the circuit to the USB port, which is connected to the keyboard.
Usb_rst	Provides the resetting signal from the USB port.
Usb_irq	Handles the interrupt controller for the USB port.

Description of software:

Blinker Code:

```c
int main()
{
    int i = 0;
    volatile unsigned int *LED_PIO = (unsigned int*)0x40; //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
}
```

The code above gets the LEDs on the FPGA to start blinking. To do this we first make a pointer to access the PIO block as assigned by the platform designer. It is important that this address is correct otherwise, the LEDs will not work as expected. Note that we use a volatile keyword to refer to the LED_PIO pointer. In C, declaring a variable as volatile informs the compiler that the variable's value can change at time without the source code specifying it. Now to make the LEDs blink, we use an infinite while loop that turns the LEDs on and off with some delay in between so we can see the changes happening in real life.

Accumulator Code:

```c
int main()
{
    volatile unsigned int *LED_PIO = (unsigned int*)0x50;
    volatile unsigned int *SW_PIO = (unsigned int*)0x40;
    volatile unsigned int *RESET_PIO = (unsigned int*)0x30;
    volatile unsigned int *ACC_PIO = (unsigned int*)0x20;

    *LED_PIO = 0x0; //clear all LEDs

//start 1
    while (1) //infinite loop
    {
        if(*ACC_PIO == 0x0) {
            while(*ACC_PIO != 0x1) continue;
            *LED_PIO = *LED_PIO + *SW_PIO;
        } else if(*RESET_PIO == 0x0) {
            *LED_PIO = 0x0;
        }
    }
//end 1
```

The accumulator simply adds values from the switches on to whatever is displayed by the LEDs and displays it back into the LEDs. To do this we connect the LEDs, switches, reset button and the accumulator button to its respective PIOs. We use an infinite while loop that simply accumulates the value on the LEDs (note that the accumulator is active low). Finally, if the reset is pressed, we reset the value stored and displayed by the LEDs.

USB/SPI functions:

```
//writes register to MAX3421E via SPI
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)

    int return_code;
    BYTE writeData[2] = {reg + 2, val};

    return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 2, &writeData, 0, 0, 0);

    if(return_code < 0) alt_printf("There was an error in MAXreg_wr\n");
}
```

In this function we write to a register in MAX3421E. To do this we use the alt_avalon_spi_command function as described in the driver documentation. This function takes in the following arguments:

```
int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,
                           alt_u32 write_length, const alt_u8 * write_data,
                           alt_u32 read_length, alt_u8 * read_data,
                           alt_u32 flags);
```

To write we set the base to SPI base address; slave is set to x0 as that is the address of slave; write_length is 2 as we are writing in 2 bytes (register address + offset of 2, as well as the value that we want to store in the register); read_length and read_data is 0 as we are not trying to read anything; flags is also set to 0 as we don't use any flags. Note that the output of

this function is an integer that tells us whether there was an error in the function or not (happens when it is negative).

```
//multiple-byte write
//returns a pointer to a memory position after last written
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //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
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);

    BYTE writeData[nbytes + 1];
    writeData[0] = reg + 2;
    for (int i = 1; i < nbytes + 1; i++) writeData[i] = data[i-1];

    int return_code = alt_avalon_spi_command(SPI_0_BASE, 0 , nbytes + 1, &writeData, 0, 0, 0);

    if(return_code < 0) alt_printf("There was an error in MAXbytes_wr\n");

    return (data + nbytes);
}
```

In this function we are writing multiple bytes into the memory that exists on the MAX3421E. To do this we first create an array called writeData of size nBytes + 1, and then pass in the data as well as the starting register address to the alt_avalon_spi_command (this is achieved using a forloop). Finally, we print if there was an error using the return code and return a pointer to the to a memory position last written.

```
//reads register from MAX3421E via SPI
BYTE MAXreg_rd(BYTE reg) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return val

    BYTE val;
    int return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, 1, &val, 0);
    if(return_code < 0) alt_printf("There was an error in MAXreg_rd\n");
    return val;
}
```

In this function we are reading a register from MAX3421E and returning the data found at that register. To do this we first write the register we want to read from, and then we read whatever value is stored at that register address and set val to that value. Finally, we print if there was an error using the return code and return the value found at that register.

```
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read data[n] from 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
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);

    int return_code = alt_avalon_spi_command(SPI_0_BASE, 0 , 1, &reg, nbytes, data, 0);
    if(return_code < 0) alt_printf("There was an error in MAXbytes_rd\n");
    return (data + nbytes);
}
```

In this function we are reading multiple bytes from memory on MAX3421E given the starting register address in memory. To do this we first write the register we want to read from, and then we read the number of bytes we want to read and set it to data. Finally, we

print if there was an error using the return code and return a pointer to the to a memory position last written.

INQ & Post Lab Questions:

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

NIOS II/e is the economy version of the processor, and it is designed to use as minimal resources and logic elements as possible. This makes this version not only much cheaper but also much slower than its NIOS II/f counterpart.

NIOS II/f on the other side is the *fast* version of the processor, and it is having more built-in hardware that make operations much faster. For instance, it has physical hardware optimized to do multiply/divide operations. In addition, it also has very fast cache, which means it doesn't have to go to memory to fetch instructions every time.

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

The key difference between on-chip memory and the SDRAM is that the on-chip memory is far closer in proximity to the CPU as compared to the SDRAM. This results in faster access times and thus decreases latency for any data that is to be extracted from the memory. In general, reduced latency means a better performing CPU as it has to go through a smaller fraction of the datapath to access instructions.

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

The main difference between the three is how instructions and data are stored in memory and accessed through the bus. In the Von Neumann model, instructions and data are both stored in the main memory and is accessed through the same shared bus. In the Pure Harvard model, instructions and data are not only stored in distinct memory units but they are

also accessed through distinct buses. For this lab, we used the Modified Harvard machine, which is where the instructions and data have their own distinct buses, but they have a shared memory (SDRAM).

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?

Since we are using PIO (Parallel I/O) modules for this lab, we have assigned certain memory addresses through which the LEDs can access data. Additionally, note that LEDs are just output devices which don't need to have any knowledge of the current instructions being executed. They simply go to the base address specified by the PIO and read whatever data is inside those memory addresses. If for instance, we ever wanted to show the instructions on the LEDs, we would still not need the LEDs connected to the instruction bus as we can store the instruction at the base address specified by the PIO and then access the instruction through the data bus.

Why does SDRAM require constant refreshing?

SDRAM is made up of transistors and capacitors. Since the charge inside the capacitors decay over time, they need to be refreshed constantly to their original charge to avoid any data loss.

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

The maximum theoretical transfer rate is given by:

$$\frac{Data\ Width\ in\ bits}{Access\ Time} * \frac{1\ byte}{8\ bits} = \frac{32\ bits}{5.4 * 10^{-9}s} * \frac{1\ Byte}{8\ bits} = 740.74\ MB/s$$

Next, we add a PLL component to provide the required clock signal for the SDRAM chip. This is because the SDRAM requires precise timings, and the PLL allows us to compensate for clock skew due to the board layout. The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?

Since the SDRAM is made up of capacitors, there is a brief period of time in which cells in the SDRAM retain the correct value before becoming discharged. Therefore, the SDRAM cannot be too slow and needs to be refreshed fast enough so that the values it outputs are valid.

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?*

Since the SDRAM module is further away from on chip memory, a particular segment of a signal is relevant at a later time because it takes more time for the signal to go to the SDRAM as compared to the on chip memory. This is why we need to set the clk c1 1ns behind clk c0. For instance, suppose it takes 1ns for the clock to go from on chip memory to SDRAM. In order to take care of this delay, we shift it by -1ns to account for extra delay taken due to the distance.

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

The NIOS II starts execution from address x08000000. This step is completed after we assign addresses to let the processor know where to start the execution from, as well as where to go in case there is an exception or a reset.

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}

Segment	Meaning
.bss	Region where uninitialized data (variables and constants) is stored. Global variables are stored here. Ex : int number;
.heap	Region where dynamic memory allocation usually takes place. Ex: int number **= (int**) malloc(sizeof(int));
.rodata	Region of static constants - read only data Ex: const int number = 1;
.rwdata	Region of variables that can be read/written data Ex: int number = 1;
.stack	Region which stores function activation record, and its subsequent data (parameters, return information, etc). Also stores local variables for the function. Ex: int add(int a, int b){}
.text	Region which stores text/strings Ex: char num = "num";

Design Resources and Statistics:

LUT	3094
DSP	10
Memory (BRAM)	55296
Flip-Flop	2442
Frequency	85.23 MHz
Static Power	96.51 mW
Dynamic Power	58.63 mW
Total Power	176.80 mW

Conclusion:

Our design for both weeks was fully functional. However, we had some trouble figuring out how to use the 4 SPI/USB functions.

Though there was nothing particularly difficult or ambiguous about this lab. It was quite daunting at first to go through the entire documentation to find the relevant functions, especially since they were not explained in the lecture. Perhaps for future labs, we would prefer if there was more help as it what the functions meant and how they operated. Something the lab did well was give us a lot of real-world experience of how we can use various resources to figure out how to operate between different IO devices and use NIOS II as well as external C compiler to run our code on the FPGA.