Lab 6 Report

Noah Conner & Adam Naboulsi
Section ABP

Summary:

In this lab, we explored the capabilities of the NIOS-II processor as the foundation for a System-On-Chip (SOC) project. The addition of a CPU can improve the flexibility, speed and debugging capabilities of many FPGA applications. In our design, the NIOS-II communicated to the DE10-Lite Shield which acted as an interface between the FPGA and USB devices. The goal of this project was to control a bouncing ball with user input from a USB keyboard. To obtain the user input, the DE10-Lite Shield communicated directly with the keyboard to capture the keycodes. These keycodes were then relayed via SPI to the FPGA, where they could then be processed to determine the direction of the ball. The FPGA uses this information to display the ball on a monitor via VGA.

Hardware Component:

A vast majority of the hardware for this lab was created through the Platform Designer. This is an interface that allows for quick instantiation and assembly of different IP Cores offered by Altera. The main component was the NIOS-II processor. This acted as a high-level interface between the usb shield and the FPGA hardware. The NIOS-II has a pure Harvard architecture, meaning it needs separate memories for instructions and data. These two memories include an external SDRAM and a On-Chip Memory. Both are instantiated and routed to the NIO-II in the Platform Designer. These components still need a clock, so a clock source and PLL are instantiated. The PLL is needed to delay the external SDRAM to account for combinational delay. Additionally, a System ID Peripheral is instantiated which gives the hardware configuration an ID so that the software editor can verify it is working with the proper hardware configuration. As far as communicating to the USB shield an SPI module is needed to talk to the MAX3421E chip. Since we are also dealing with software, it is beneficial to have a JTAG interface so that we can inspect what is happening on the FPGA during program execution. Finally, multiple Parallel I/O IPs are instantiated so that the software can relay information from and to the hardware through a memory mapping.

The I/O interface through the NIOS-II processor works as essentially a parameterized version of MEM2IO in lab 5. On platform designer we instantiate a Parallel-Input/Output module (PIO) in which the corresponding device is assigned a memory address. The PIO module bridges from Avalon (our bus architecture) to the FPGA logic either to input new data or to output data from the corresponding I/O. In 6.1 we instantiate a switch PIO and an LED PIO. Within the PIO modules, we have a series of control registers which the CPU (NIOS-II) communicates with and manipulates; the PIO then based on the information provided by the control registers, set signals high/low or read/write some signals. Irrespective of the size of the PIO module, each of the control registers will always be 32 bits (4 Bytes). These addresses are located – in the memory space of the CPU – in order (data, direction, interruptmask, edgecapture, outset, and outclear) starting from the assigned base address of the PIO peripheral,

each offset by 4 bytes since their sizes are 32 bits regardless of the bit-length of the corresponding PIO device. Through the Eclipse IDE, we access the data corresponding to the associated device by assigning a 32-bit pointer to the base address of the corresponding PIO device and dereferencing the pointer to either assign the data register (no offset, first register from the assigned base address) to our desired value or read from the register.

The NIOS-II processor communicates with both the MAX3421E USB chip and the VGA components indirectly through some additional hardware. The USB chip features an SPI communication protocol interface, so we used the SPI IP to read and write to specific registers on the USB chip. This allows us to configure and command the chip. As far as communicating with the VGA component, it provides the VGA hardware with the keycode from the USB device, from which the hardware can determine what needs to be displayed on the VGA monitor.

The SPI protocol used to communicate with the MAX3421E USB chip is a 4-wire synchronous serial communication protocol. This protocol depends on having a master and one or more slaves. In our case we have one slave. The four wires that the devices share include a clock, a slave select, a master-in-slave-out (MISO) and a master-out-slave-in (MOSI). The clock is responsible for synchronizing the communication between the master and slaves. The slave select is a master-controlled line that tells the slaves a transmission is happening. Finally, the MISO and MOSI lines are the data transmission lines. As the names infer, MISO and MOSI are unidirectional and transmit data from the slave to the master and from the master to the slave, respectfully. That describes the physical features of the SPI interface. Now, to actually communicate with a device, the master will pull the slave select line high and send an 8-bit message to the slave. In the case of the MAX3421E USB chip, the higher 5 bits is the address of the register the master is going to read/write to and the 2nd most significant bit is the direction bit. When the direction bit is high the master is writing and when low the master is reading. If the first transaction is for a write, the master will send another with the data to write. If it is for a read, the master will wait while the slave sends the register data via the MISO line. This communication protocol is great for small distance low bandwidth communication as it is very simple and only requires four wires.

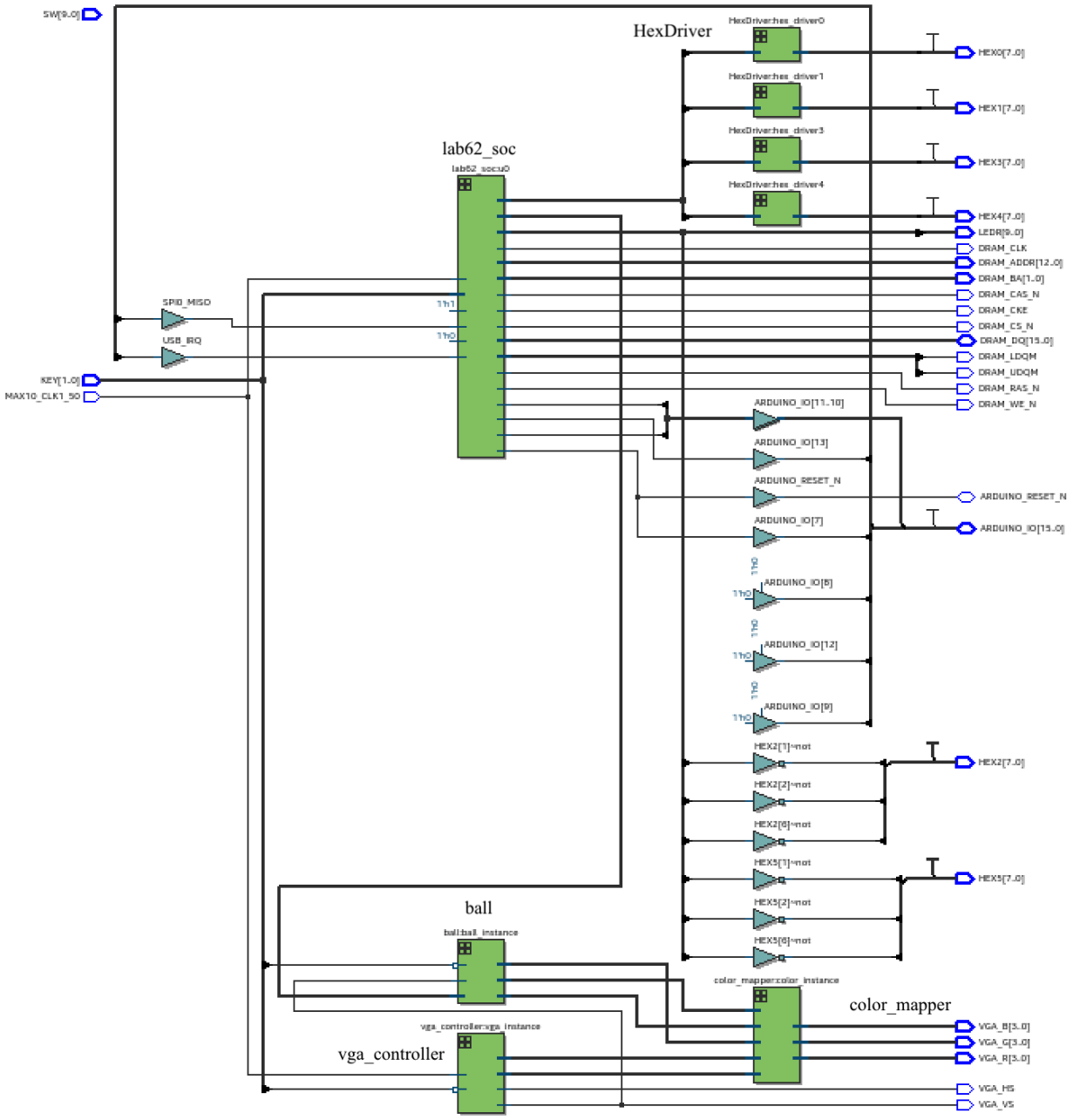
The main function in lab 6.1's purpose initially was to continuously make the LSB LED on our FPGA turn on and off; after adding modifying the code such that the LEDs acted as an accumulator, its purpose was to add the switch values of our FPGA to the value represented on the LEDs everytime the Accumulator button was pressed. In lab 6.2, we modified the MAX3421E.c file's MAXreg_wr, MAXbytes_wr, MAXreg_rd, and MAXbytes_rd functions. The purpose of the MAXreg_wr function was to write a given single BYTE value to the given register (of the SPI); the purpose of the MAXbytes_wr function was to right a given arbitrary number of BYTEs (and corresponding value) to a given register (of the SPI) and return a pointer to a memory position after the last written BYTE; the purpose of the MAXreg_rd function was to read a single BYTE of data stored in a given register (of the SPI) and return it; lastly, the purpose of the MAXbytes_rd function was to read an arbitrary number of BYTEs from a given register (of the SPI) and return a pointer to the memory position of the last read BYTE.

A standard VGA acts as an interface for controlling analog monitors where the computing side of the interface provides the monitor with color magnitudes, ground references, and horizontal/vertical sync signals. The vertical/horizontal sync signals are digital waveforms provided by our FPGA which synchronize the signal timing with the monitor. Colors are produced through analog signals sent over Red, Green, and Blue wires. VGA Monitors consists of an organized matrix of 640 horizontal pixels and 480 vertical lines that make up the screen. Images are displayed on the VGA through an electron beam which "paints" each pixel from left to right in each row from the top of the matrix/screen to the bottom. When the electron beam reaches the bottom right of the screen, it resets for the next frame and returns to repeat the process from the top left of the screen where each sweep from top to bottom represents a single frame on our screen and the screen refresh rate is 60Hz (one frame = 16.67 ms). In order for our VGA to properly display our intended images (in lab 6.2's case, the ball and the background), we need to keep track of the electron beam's position; this is done through the use of counters in our VGA_controller module. By keeping track of the beam position, we know the coordinates of the beam at each pixel time tick. In the VGA_controller module, the coordinates are labeled as DrawX and DrawY and are available to both the Color_mapper and Ball modules of our design for lab 6.2.

Our Color_mapper module performs object/shape coloring and rendering by putting the RGB signals to the monitor. The module is given the Ball's coordinates (from Ball.sv described below) and sets the RGB densities for the pixels at the ball's location and surrounding the ball (corresponding to the color orange); when the electron beam's coordinates coincide with a point within the circle of pixels for the ball, the module outputs the color orange (ball's color in our lab) for that pixel. When the beam coordinates are outside of the ball's circle of pixels, the module outputs a blue background color that darkens from left to right.

The ball module in our design computes the position of the ball in every frame. We keep track of the frames through the vertical sync signal provided by the vga_controller; when the signal goes from low to high, we know that we have started a new frame. We create the motion of the ball by updating its coordinates with each frame (ballx + stepx, bally + stepy). Positive stepx values produce rightward motion of the ball, negative values of stepx produce leftward motion of the ball, positive stepy values produce upward motion of the ball, and negative values of stepy produce downward motion of the ball. Increasing the magnitude of the step values increases the speed of the ball. The ball module takes in commands from the keyboard and sets the corresponding direction of movement for the ball through a case statement with the WASD keycodes.

Top-level Diagram



Module Descriptions:

Module: ball.sv

Inputs: Reset, frame_clk, [7:0]keycode
Outputs: [9:0]BallX, [9:0]BallY, [9:0]BallS

Description: In an always_ff block, we check if the asynchronous reset is high and if so, set the ball motions to 0 and the ball position to the center. Else, we check if the ball is at the bottom edge, the top edge, the right edge, or the left edge, and set the ball X/Y motion accordingly such that the ball bounces off of the edge (by x or y step, corresponding to the edge direction). If the ball is not at any of the edges, we go into a case statement that checks the key code (AWSD, LUDR) and set the ball X/Y motion accordingly. At the end of the always_ff code, we update the ball Y and X positions to the previous X/Y positions + X/Y ball motion.
Purpose: The purpose of this module is to compute the ball's position in every frame based on the keyboard input direction.

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: At the beginning of the module, we set DistX and DistY variables to the distance between the current pixel (DrawX, DrawY) and the old ball position (BallX, BallY). Next, in an always_comb block, we check if the current pixel is within the ball using the standard circle formula along with the ball size, and mark the ball as on otherwise, we mark it off. In another always_comb block, we set the current pixel's RGB values if the ball is on to the intended ball color, otherwise, if the ball is off, we set the current pixel's RGB values to the indented background color (in our case a blue gradient based on the pixel position).
Purpose: The purpose of this module is to render and color the objects/shapes of our design (ball and the background) to display through the RGB color domain to the monitor.

Module: VGA_controller.sv

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

Description: In the beginning of the module we set the horizontal pixels and vertical pixels parameters and instantiate the line counters (horizontal and vertical, to keep track of the electron beam) and disable composite sync. Next, we cut the 50MHz clock in half

(assigning it to clkdiv) in an always_ff block synchronous with the posedge of the clk. In another always_ff block synchronous with clkdiv, we check if reset is high, and if so, set the horizontal and vertical counters to 0. Otherwise, if the counters have reached the end of the pixel count (they equal the horizontal and vertical parameters) we set them to 0, else, we increment the horizontal and vertical counters. Next, we assign the current pixel (DrawX, DrawY) to the horizontal and vertical counters. In another always_ff block synchronous with clkdiv, we check if reset is high and if so, set the horizontal sync low; otherwise, if we are in the overscan region (656-752), we set the horizontal sync low to ensure a clean waveform, else we set the horizontal sync high. In another always_ff block synchronous with clkdiv, we check if reset is high and if so, set the vertical sync low; otherwise, if we are in the overscan region (490-491), we set the vertical sync low to ensure a clean waveform, else we set the vertical sync high. Lastly, in an always_comb block, we display pixels if they are between horizontal 0-639 and vertical 0-479 (640x480) by correspondingly checking and setting display low or high.
Purpose: The purpose of this module is to keep track of the electron beam's position and provide its coordinates to the Color_Mapper, Ball, and top-level modules.

Module: lab62_soc.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: To the designer, this is a back box containing the HDL for the NIOS-II, memory and other Platform Designer components. It is instantiated in the top-level module, lab62.sv.
Purpose: The purpose of this module is to abstract the SoC created in Platform Designer. More specifically, it contains all of the functionality for the NIOS-II processor, on-chip memory, SDRAM interface, JTAG interface, SPI interface and parallel I/O components. These are necessary for the SoC to record user input. The SPI interface and parallel I/O components specifically interact with the MAX3421E chip to get information about the USB devices, such as a keyboard or mouse. The USB I/Os are for the NIOS-II to communicate with the SPI interface on the FPGA so that it can do something useful with that information. Further information is available in a later section.

Module: lab62.sv

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

Outputs: [9:0]LEDR, [7:0]HEX0, [7:0]HEX1, [7:0]HEX2, [7:0]HEX3, [7:0]HEX4, [7:0]HEX5, DRAM_CLK, DRAM_CKE, [12:0]DRAM_ADDR, [1:0]DRAM_BA, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [3:0]VGA_R, [3:0]VGA_G, [3:0]VGA_B
Inout: [15:0]DRAM_DQ, [15:0]ARDUINO_IO, ARDUINO_RESET_N

Description: This is the top-level module and is used only to instantiate submodules and connect interconnect wires. The lab62_soc.v module is instantiated and connected to the required SDRAM pins, clock, keycode export, etc. The vga_controller, ball and color_mapper are also instantiated and connected accordingly.
Purpose: The purpose of this module is to instantiate and allow the VGA_Controller, Color_Mapper, and Ball modules to interact with one another such that the frames of the ball and background (and their various colors) are displayed correctly on the VGA screen through the electron beam. It also instantiates the lab62_soc, establishing the clk, SDRAM, USB SPI, USB GPIO, and HEX/LED keycode data. Lastly, it instantiates HEX drivers and outputs the keycodes of the inputted keyboard keys.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0

Description: Based on the inputs, a switch statement is used to map the outputs to the corresponding LED segments (7 total) on the HEX display of the FPGA board.
Purpose: Used to output bits of registers in the register unit to the HEX displays on the FPGA board.

Platform Designer Description:

A large portion of the hardware in this lab was created using the Platform Designer. This allowed us to quickly develop the SoC for user input control. Below are the Platform Designer schematics for both lab 6.1 and lab 6.2. As you can see, all the components used in lab 6.1 are used in lab 6.2. This is because both labs take advantage of the NIOS-II IP which requires certain components to operate. Each IP core is described below.

(6.1 & 6.2) clk_0:
This is a clock source interface that provides the rest of the IP cores with a synchronous clock to the rest of the hardware. It has two exported signals, clk and reset, which can be connected in the instantiation of the Platform Designer module.

(6.1 & 6.2) nios2_gen2_0:

This is a NIOS-II processor IP which is the actual CPU that will control all user input and other low speed operations. It has inputs clk and reset from the clock source so that it is synchronous with the reset of the hardware and inputs irq and debug_mem_slave for debugging purposes. Outputs include data_master and instruction_master which are the two data buses in the harvard architecture. The output debug_reset_request is used for debugging purposes.

(6.1 & 6.2) onchip_memory2_0:

This is an on-chip memory IP that creates an on-chip RAM. While normally, programs are executed from DRAM due to memory size, the on-chip RAM is used as a placeholder block.

(6.1 & 6.2) sdram:

This is an SDRAM controller IP which will communicate with the external SDRAM chip on the FPGA development board. This memory is large enough to store the software program for the NIOS-II processor.

(6.1 & 6.2) sdram_pll:

This is a PLL IP which means phase-locked loop. This can essentially create a clock signal of any frequency or phase with respect to the input clock, which in this case is the clock source module. The reason for needing this module is because the external SDRAM needs a slightly delayed clock signal due to the delay of the FPGA's combinational logic. This puts the rising edge, or the sampling point, right in the middle of the eye pattern data-in signal.

(6.1 & 6.2) sysid_qsys_0:

This is a system ID peripheral IP which gives the module a unique identification number that the software IDE can use to confirm it has the correct configuration when running the software program.

(6.1 & 6.2) leds_pio:

This is a parallel I/O IP that enables communication between the hardware and NIO-II processor via memory-mapping. Specifically, this allows the software to control the state of the LEDs.

(6.2) keycode, usb_irq, usb_gpx, usb_rst, hex_digits_pio, key:

These are parallel I/O IPs that enable communication between the hardware and NIO-II processor via memory-mapping. The keycode is an output to the hardware so that the VGA hardware can determine the motion of the ball. The USB I/Os are for USB communication with the USB devices. The hex is an output to the hardware so that it can display on the hex displays on the FPGA development board. Finally, the key is an input for the two buttons on the board. They are not being used with the NIOS-II in this lab. All of these parallel I/Os are mapped to specific memory locations for the software to access them.

(6.2) jtag_uart_0:

This is a JTAG UART IP used to aid in the debugging process. It allows the use of print statements in the software program.

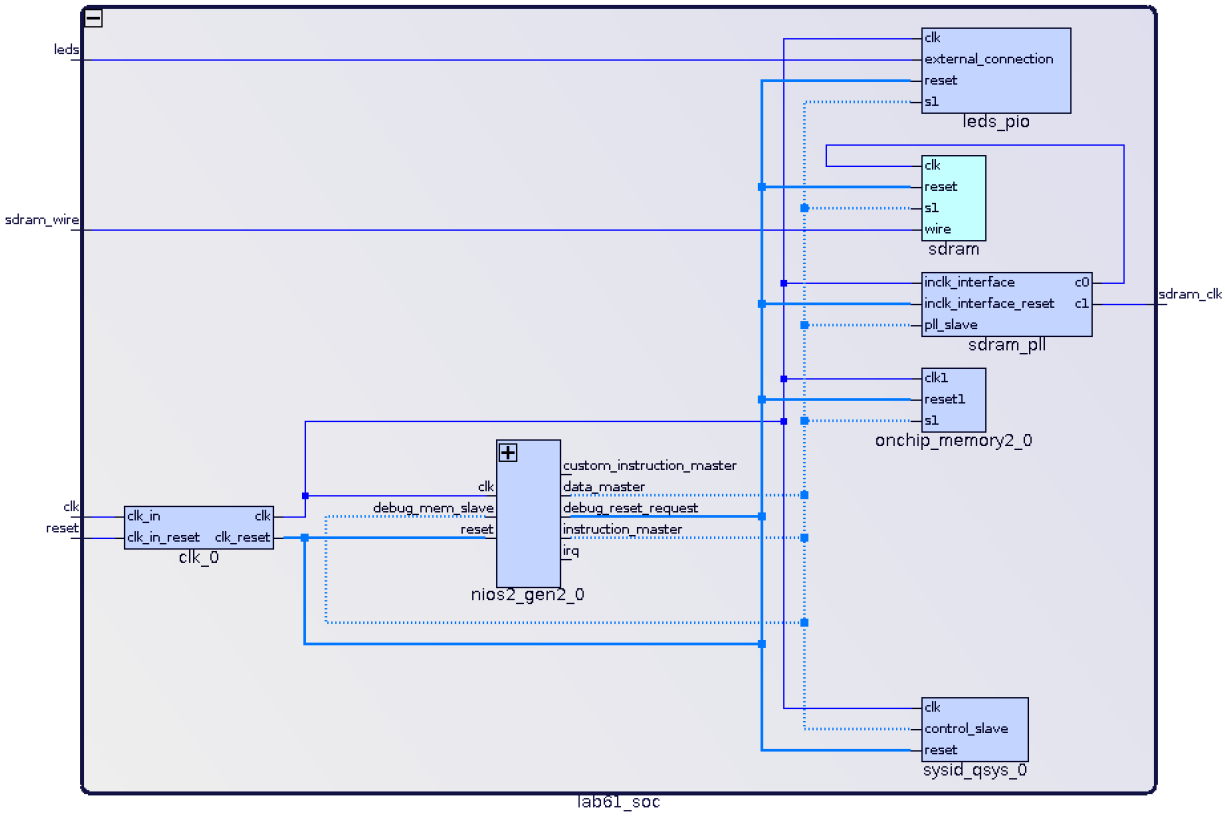
(6.2) timer_0:

This is an interval timer IP that creates a sort of counter for the software to base delays off of. It is configured to 1ms. The software is able to create delays as small as 1ms with this.

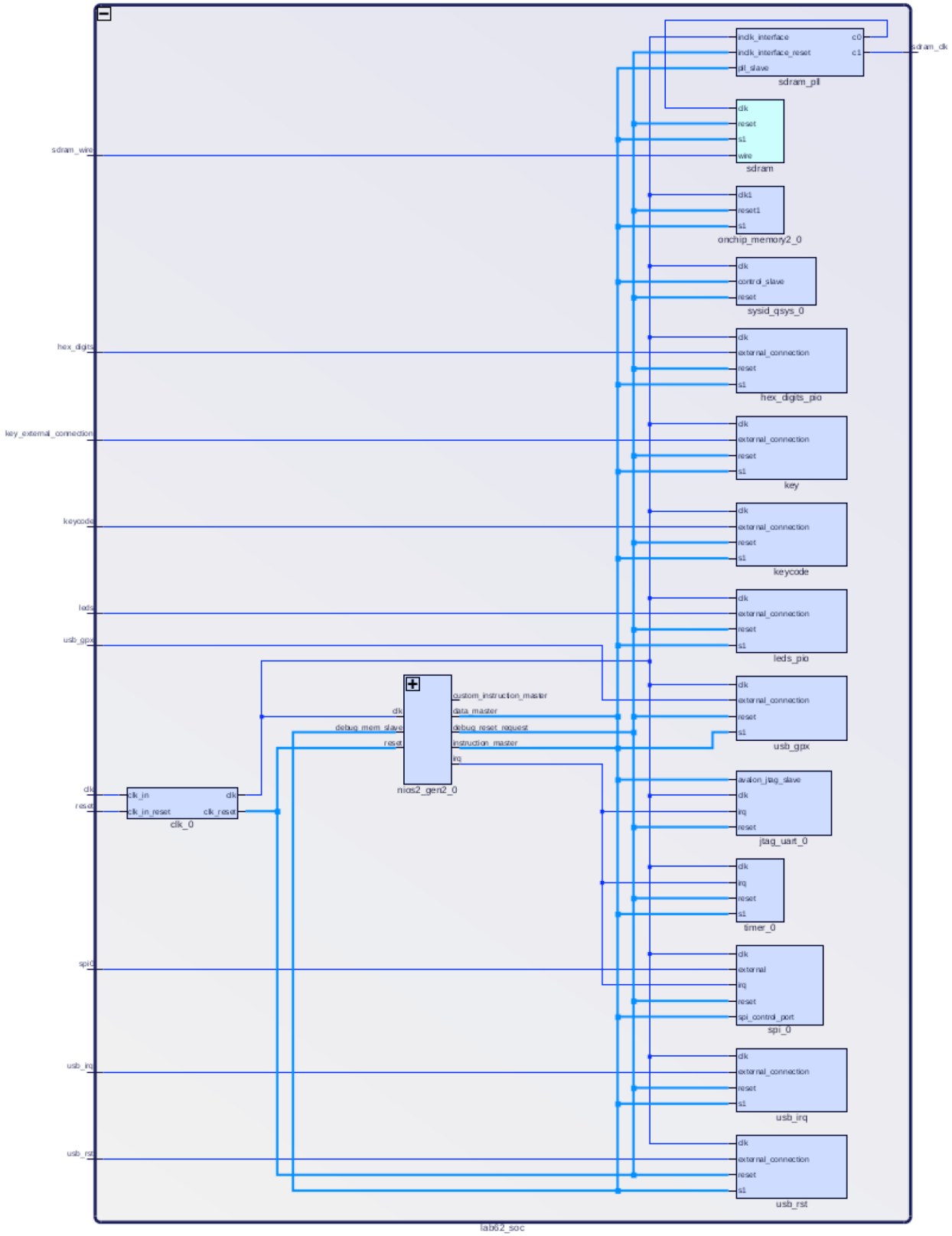
(6.2) spi_0:

This is an SPI IP used to communicate with the MAX3421E USB chip. It abstracts all of the SPI processes such as the buffer, acknowledgements and transmission of data. The software program can use this component to communicate with the USB chip without the distraction of hardware.

Lab 6.1 Platform Designer Schematic:



Lab 6.2 Platform Designer Schematic:



Software Component:

In the main function for lab 6.1, we assigned a volatile unsigned int (32-bit) pointer to the base address of our LED PIO (LED_PIO), a volatile unsigned int (32-bit) pointer to the base address of our Accumulator button PIO (ACC_PIO), and a volatile unsigned int (32-bit) pointer to the base address of our switches PIO (SW_PIO). We then cleared the LED PIO's data control register by assigning the dereferenced LED_PIO pointer to 0. In an infinite while loop (constantly true condition), if the Accumulator button was active – the data control register of the accumulator PIO was 0x0 since it is active-low, checked by notting dereferenced ACC_PIO and bitwise anding it with 0x1 and checking if the result is 1 – we enter a while loop that checks the same condition (thus waiting until it is false, equivalently until the button is released), then once out of the while loop, we set the dereferenced LED_PIO (its data register) to the previous value of the LED's data control register, *LED_PIO, + the current value of the switches (dereferenced switch pointer, *SW_PIO) and took its modulus with 256 (the max value that could be represented on the LEDs of our FPGA). The extra while loop after the if condition is met in our code is used to ensure that LED adds its previous value only once (the addition statement will continuously happen without the while loop if the button is held for longer than one iteration of the outer infinite while loop).

For lab 6.2 (the USB/SPI portion of the lab), we filled in code for the following functions: MAXreg_wr, MAXbytes_wr, MAXreg_rd, and MAXbytes_rd. In the MAXreg_rd function, we began by initializing an int return_code assigned to the output of the alt_avalon_spi called later. We then initialize an alt_u8 array of size 1 byte (unsigned 8 bit value) named wdata. Next we set wdata's [0] byte to the inputted (pre-shifted) reg value of the function. Next, we initialized another alt_u8 array of size 1 byte labeled rdata. Afterwards, we set the return_code integer equal to the called alt_avalon_spi_command function with the following parameters (base = SPI_0_BASE, slave = 0, writelength = 1, write_data* = wdata, readlength = 1, read_data* = rdata, flags = 0). This call of the function accesses the SPI, writes 1 byte from wdata (pre-shifted reg value) which specifies which register to read from and reads 1 byte from that register, setting dereferenced rdata to the read data. Next, we check if the return_code is less than 0 which indicates an error and print "ERROR" if it is less than 0. Lastly we return dereferenced *rdata which returns the read value stored in the passed in register index.

In the MAXreg_wr function, we began by initializing an int return_code assigned to the output of the alt_avalon_spi called later. We then initialize an alt_u8 array of size 2 bytes (unsigned 8 bit value) named wdata. Next we set wdata's [0] byte to the passed in (pre-shifted) reg index value (+2 to indicate the direction, a write). Next, we set the second byte of the wdata array to the passed in byte val (the value intended to be written to reg). Next, we initialized another alt_u8 variable named rdata. Afterwards, we set the return_code integer equal to the called alt_avalon_spi_command function with the following parameters (base = SPI_0_BASE, slave = 0, writelength = 2, write_data* = wdata, readlength = 1, read_data* = &rdata, flags = 0).

This call of the function accesses the SPI, writes 2 bytes from wdata (pre-shifted reg value to write to, then the value to be written) which specifies which register to write to and what data should be written, reading nothing to rdata. Next, we check if the return_code is less than 0 which indicates an error and print "ERROR" if it is less than 0. Since this function is void, we do not need to return anything.

In the MAXbytes_rd function, we began by initializing an int return_code assigned to the output of the alt_avalon_spi called later. We then initialize an alt_u8 array of size 1 byte (unsigned 8 bit value) named wdata. Next we set wdata's [0] byte to the inputted (pre-shifted) reg value of the function. Afterwards, we set the return_code integer equal to the called alt_avalon_spi_command function with the following parameters (base = SPI_0_BASE, slave = 0, writelength = 1, write_data* = wdata, readlength = nbytes, read_data* = data, flags = 0) where nbytes is a parameter representing the number bytes meant to be read, and data is a parameter that is meant to take in the read bytes. This call of the function accesses the SPI, writes 1 byte from wdata (pre-shifted reg value) which specifies which register to read from and reads nbytes from that register, setting dereferenced data to the read data. Next, we check if the return_code is less than 0 which indicates an error and print "ERROR" if it is less than 0. Lastly, we return data+nbytes which returns a pointer to the last read-in value from the register.

In the MAXbytes_wr function, we began by initializing an int return_code assigned to the output of the alt_avalon_spi called later. We then initialize an alt_u8 (unsigned 8 bit value) array of size nbytes+1 named wdata. Next we set wdata's [0] byte to the passed in (pre-shifted) reg index value (+2 to indicate the direction, a write). Next, in a for loop (from byte n=0 to nbytes-1, set the following bytes of the wdata array to the parameter's BYTE array, "data"'s values (the value intended to be written to reg) – wdata[n+1] = data[n]. Next, we initialized another alt_u8 variable named rdata. Afterwards, we set the return_code integer equal to the called alt_avalon_spi_command function with the following parameters (base = SPI_0_BASE, slave = 0, writelength = nbytes+1, write_data* = wdata, readlength = 0, read_data* = &rdata, flags = 0). This call of the function accesses the SPI, writes nbytes+1 bytes from wdata (pre-shifted reg value to write to, then the value to be written) which specifies which register to write to and what data should be written (the bytes following the first bytes in the array which store the values of the data array parameter), reading nothing to rdata. Next, we check if the return_code is less than 0 which indicates an error and print "ERROR" if it is less than 0. Lastly, we return data+nbytes which returns a pointer to the memory position of the last written BYTE to reg.

Post Lab Questions:

What are the differences between the Nios II/e and Nios II/f CPUs? P5

The e is the economy version, which has only the processor and JTAG components (plus ECC). The f stands for fast and has many more options and components premade, such as hardware multiply/divide, caches, MMU, MPU.

What advantage might on-chip memory have for program execution? P7

On-chip memory is the highest throughput, lowest latency memory possible in an FPGA-based embedded system. It typically has a latency of only one clock cycle. Some variations of on-chip memory can be accessed in dual-port mode, with separate ports for read and write transactions. Dual-port mode effectively doubles the potential bandwidth of the memory. Another advantage of on-chip memory is that it requires no additional board space or circuit-board wiring because it is implemented on the FPGA directly. Using on-chip memory can often save development time and cost. Finally, most variations of on-chip memory can be automatically initialized with custom content during FPGA configuration.

Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why? P7

The NIOS-II has a Pure Harvard architecture which means there are separate storages and data paths for instructions and data. This architecture can allow for quicker processing since the instructions and data can be accessed in parallel. However, the number of connections within the FPGA is increased due to the separate data paths.

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? P8

The LEDs only display data, on or off, like wires. They aren't components that can make sense of instructions like the NIOS-II processor. The reason the on-chip memory needs access to both buses is because it acts as an instruction/data buffer for the NIOS-II.

Why does SDRAM require constant refreshing? P8

SDRAM memory is held using a transistor and a capacitor. Capacitors leak charge over time, so they need to be refreshed to their initial levels occasionally so that data isn't lost. In contrast to SRAM, which uses several transistors together in a loop to avoid refresh at the cost of space.

*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. P9*

width * **nrows** * ncols * **ncs** * nbanks = total memory

16 * **8k** * 1k * **1** * 4 = 512000000 bits = 64000000 Bytes = 64 MBytes

What is the maximum theoretical transfer rate to the SDRAM according to the timings given? P9

The maximum theoretical transfer rate is 400 MB/s. The datasheet says the maximum operating frequency is 200 Mhz, so using pipelining, it is possible to read or write 16 bits, or 2 bytes, of data every clock cycle. The product of the operating frequency and 2 bytes gives us the maximum theoretical transfer rate.

The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case? P9

Since SDRAM is volatile, it needs to be refreshed. If the clock is too slow then it will not be refreshed fast enough and data may be lost.

Why do we need to add a second clock with phase shift -1ns that goes out to the SDRAM chip itself? P11

We need to add this delay so that the clock and the data coming from the SDRAM controller are synchronized. Since the control signals have to go through FPGA logic and the clock is directly connected to the SDRAM, a propagation delay occurs that makes the clock sample the SDRAM data very close to when the SDRAM data becomes available. A -1ns delay provides the perfect amount of phase shift so that the clock samples the data in the middle of its eye pattern.

What address does the NIOS II start execution from? Why do we do this step after assigning the addresses? P14

0x080000000; You have to have memory to know where your parts of the program will be located.

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). P20

In the main.c file for lab 6.1, we first initialize a variable i to 0 used later in the program and assign a volatile unsigned int (4 bytes/32 bits) pointer (LED_PIO) to the base address of the LED PIO block created in platform designer – such that we can access the block. C's volatile keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time--without any action being taken by the code the compiler finds nearby. After initialization, we clear all the LEDs by setting the dereferenced LED_PIO variable to 0 which sets the 32-bit data register of the LED PIO to 0, thus causing the LEDs on the physical FPGA to output 0. Afterwards, we enter an infinite while loop (constantly true condition) and create a software delay through a for loop that counts up to 100000 then bitwise ORs the dereferenced LED_PIO with 0x1 (again accessing the data register of the PIO and setting the lowest significant bit to 1 such that the physical LEDs of our FPGA light up the LSB) and again go into a software delay through a for loop, then finally bitwise AND the dereferenced LED_PIO with 0x1 (again accessing the data register of the PIO and clearing its bits to 0 such that the physical LEDs on FPGA are all off). This constant loop allows the LED representing the LSB to continuously turn on and off. The last line of code (return 1) outside of the infinite for loop is never reached but since the function (main) needs a return integer value, we return 1.

Lines 13:16
LEDS initially cleared = 0000000000

Line 13: 100000 iteration Delay
Line 14: 0000000000 ||
0000000001
= 0000000001
Line 15: 100000 iteration Delay
Line 14: 0000000001 &&
0000000000
= 0000000000

Look at the various segments (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code: const int my_constant[4] = {1, 2, 3, 4} P21

.bss	read-write zero initialized data; uninitialized global/static variables e.g. “int var;”
.heap	reserved section for heap
.rodata	read-only data; global/static constants e.g. “const int var = 4;”
.rwdata	read-write initialized data; initialized global/static variables e.g. “int var = 2;”
.stack	reserved section for stack
.text	program code; functions, executable code e.g. “int add(x, y) return x + y;”

Design Resources and Statistics:

LUT	4037
DSP	2
Memory (BRAM)	11392 bits
Flip-Flop	2528
Frequency	72.28 Mhz
Static Power	96.51 mW
Dynamic Power	60.00 mW
Total Power	178.23 mW

Conclusion:

Our design allows for simple user interface via a USB device and VGA monitor. At 60fps and a very low-latency user input, this design can be used to build a very responsive game. This minimal setup is the foundation on which many platforms were created and serves an important role in learning about FPGA design. All in all, our design functions as expected, and we encountered no problems and finished the lab relatively quickly. As far as understanding how the design was supposed to operate, there was nothing ambiguous, incorrect or unnecessarily difficult. A few odd problems occurred such as having to physically touch the IO shield to get it working properly and adding arbitrary delays in the software to keep it from freezing. Overall, these problems created a difficult debugging environment, but we were able to complete the lab in time.