ECE 385

Fall 2022

Experiment #6

SOC with NIOS II in SystemVerilog

Ravi Thakkar, Siraj Khogeer

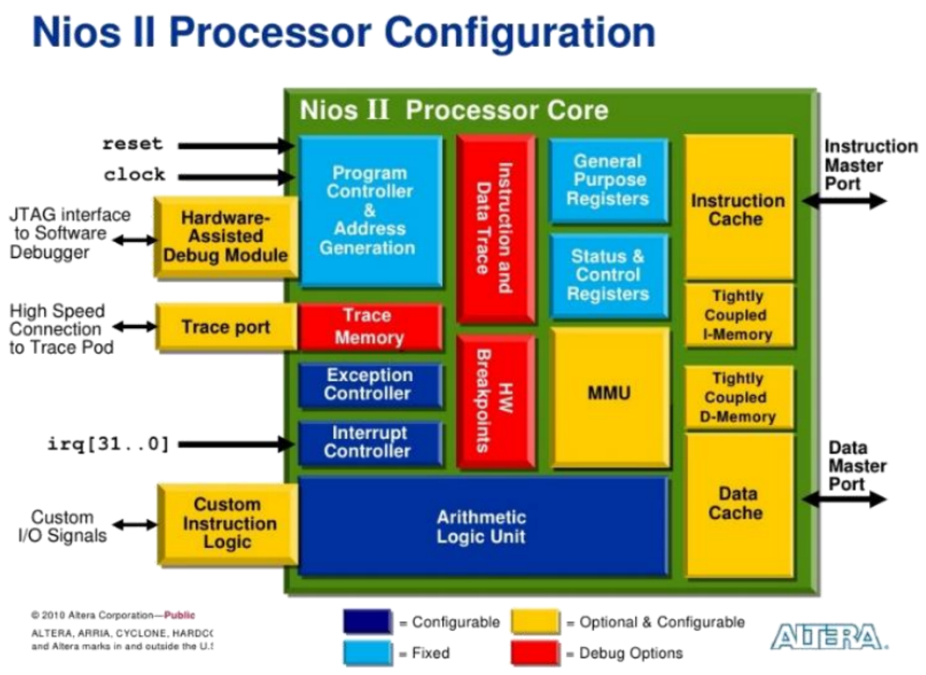
TA: Gavin Wu

10-22-2022

Introduction

In this lab, we use the NIOS-II processor IP through the Platform Designer/QSys tool within Intel Quartus. Using IP cores, we design the processor on Platform Designer and then we generate Verilog files for the SoC so that we can interface, compile, and program the MAX10 FPGA. After programming the FPGA, we are able to run programs written in C on the SoC.

The NIOS-II processor is a Harvard model 32-bit CPU and we use it to handle tasks such as data input and output and user interface. As an IP-based processor, the NIOS-II is a good choice because we are not running any sort of tasks that require high performance but the processor has a full C compiler included and a JTAG interface.



The way that we deal with input and output off of the FPGA itself is through a keyboard that uses a USB connection and a monitor that uses a VGA connection. The I/O requires its own IP components which can be added through Platform Designer onto the SoC. The USB Keyboard, which is for input, reads a keystroke (for Lab 6.2, this is 'W', 'A', 'S', or 'D'), This keystroke signal is sent to the MAX3421E USB host chipset, which communicates with the MAX10 and the SoC that we have created using the SPI protocol, where the MAX10 device is the master and the MAX3421E is the slave device. The VGA connection, which is the output, requires its own peripheral on the SoC. The VGA connection is constantly sending signals to 'draw' pixels, which is done one-by-one by sending an RGB value to the monitor for the respective pixel. This is done for every frame, and the refresh rate for the monitor is 60Hz. This is accomplished by the vga_controller module, which outputs the horizontal and vertical sync signals.

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 Platform Designer module that we created is an SoC centered around the NIOS-II CPU, and this is the main logic being programmed onto the FPGA, other than what the few .sv modules contain. For this component of the lab, we followed the instructions in the *Introduction to NIOS II and QSys* document. Using the IP Catalog on Platform Designer, we added different components such as the CPU itself, the SDRAM, On-Chip Memory, etc. We do this by adding these IP blocks through the catalog, configuring their settings, adding and then adding connections such as clock and reset signals. We also add the JTAG UART so that we are able to debug our C code when we are programming on the SoC. Another important part of this component of the lab is setting up the SPI peripheral to enable communication between the MAX10 device and the MAX3421E chipset. We also add three PIO IP Blocks for the USB keyboard connection. We then used these modules to display the keyboard characters and use it to control a ball on the screen.

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

In Lab 6.1, we created an accumulator circuit which takes in binary values from the switches on the Max10 device, and when the accumulate button is pressed, the LEDs next to the switches light up to represent the total accumulated value in binary. When the clear button is pressed, all of the LEDs turn off, effectively resetting the accumulator. This means that Lab 6.1 required two types of input and one type of output to be enabled and interfaced with.

The way that the I/O functionality was accomplished for Lab 6.1 was through the use of PIO IP Blocks that were added to the SoC on Platform Designer. These PIO blocks act as a bridge between Avalon MM Bus to the FPGA logic. The sw block has an input direction and has a width of eight bits for the eight least-significant switches. The keys block has an input direction and has a width of two bits. Finally, the led block has an output direction and has a width of eight bits. The PIO blocks are memory mapped to base addresses plus offset, which are assigned by Platform Designer.

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

The NIOS interacts with the MAX chip via the SPI protocol. This protocol is also combined with GPIOs to fully interact with the NIOS. The way this was used was by using the FGA pins and mapping them to the respective SPI ports, and then connecting those to the SPI IP in platform designer, and this allows the NIOS to communicate via the avalon bus. The VGA component was much simpler. The NIOS system read the keycode from the keyboard and then exported a wire with the keycode to the top-level module. This happened through platform designer. This wire was then used in the ball module to trace the movement of the ball. The VGA controller was separately instantiated in the top level module to provide the sync signals and the color mapper was used to provide the color. The keycode from the NIOS was used in the ball module which was then used to influence the color mapper.

iv. Written description of the SPI protocol.

The SPI protocol is a way for a master and a slave to communicate. SPI stands for Serial Peripheral Interface. There are 4 signals, a clock signal, a Master Input Slave Output, a Master Output Slave Input, and a slave select. The clock signal and slave select depend on the host and slave connection and are usually constant. MISO is the output of the slave periphiral to the master input, and MOSI is the opposite. SPI works by having sycronous serial signals that communicate between master and slave. In this lab we implemented two main things, a write and a read. In this situation, the SPI starts with a command byte which sets a specific register on the master to read or write to. To write, we use the MISO and to read we use the MOSI pins. These were implemented in NIOS using C, and using the avalon interface.

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

The VGA is split into three parts. The VGA controller, the Ball, and the color mapper. The VGA controller works by providing two signals, a HS and a VS signal. This is essentially a counter that increments horizontally for 800 pixels, outputting HS at 656-752, and a vertical counter for 525 pixel, with VS at 490-491 lines. The VGA controller also provides a blank and pixel_clk that is used in other parts. These VS and HS are directly connected to the FPGA and outputted to the monitor which is used to synchronize it with the VGA port. The controller also provides a DrawX and a DrawY signal which are essentially the coordinates on the screen that is currently being drawn. The second part is the Ball module. This module defines the parameters of a ball using math, and sets its coordinates in relation to the screen. The module then takes a keycode input to impact the motion of the ball. The ball is initially in the center which initializes its coordinates. The module then defines its size and X and Y coordinates. These are then exported to the color mapper. The ball also uses math to determine the motion of the ball, if the right arrow is pressed, then the next coordinate of the ball goes to the right and continues until further input or it reaches a wall. If it reaches a wall it then bounces back by using math to see if the coordinates and its size are within the confines or the screen. The ball module is a synchronous module and uses a frame clock which we mapped to the VS signal. This ensures that the ball runs at 60hz, which matches the screen frame rate. The DrawX,Y BallX,Y and Ball Size then go into the color mapper to be drawn on the screen. The color mapper looks at the current screen coordinates, and the ball coordinates taking account its size, and if the ball coordinates match the current coordinates, then red green and blue signals are set to be the color of the ball. If not, the background is set as a gradient dependent on the horizontal position. These red green and blue signals are outputted to the top level module which maps it to the RGB pins of the FPGA port which then goes into the display to show the ball and the screen. Note: The keycode to control the ball motion comes from the NIOS system through the SOC on the top level.

Purpose of C functions (Only the modified functions)

Lab 6.1

Function: **main** in main.c

Parameters: None

Return: Returns a constant integer, the return doesn't matter because there is an infinite loop within main().

Purpose: This function contains the code for the blinker program as well as the accumulator program. Using an infinite loop, signals from the inputs (switches and buttons) and signals to the output (LEDs) can be manipulated. The blinker program makes the LSB LED blink on and off, and the accumulator program takes input from the switches and adds them to a sum which is displayed on the LEDs.

Lab 6.2

Function: MAXreg_rd

Parameters: BYTE reg

Return: BYTE val

Purpose: This function is used to read one byte of data from a target register specified by the byte 'reg'.

Function: MAXbytes_rd

Parameters: BYTE reg, BYTE nbytes, BYTE* data

Return: BYTE* data+nbytes

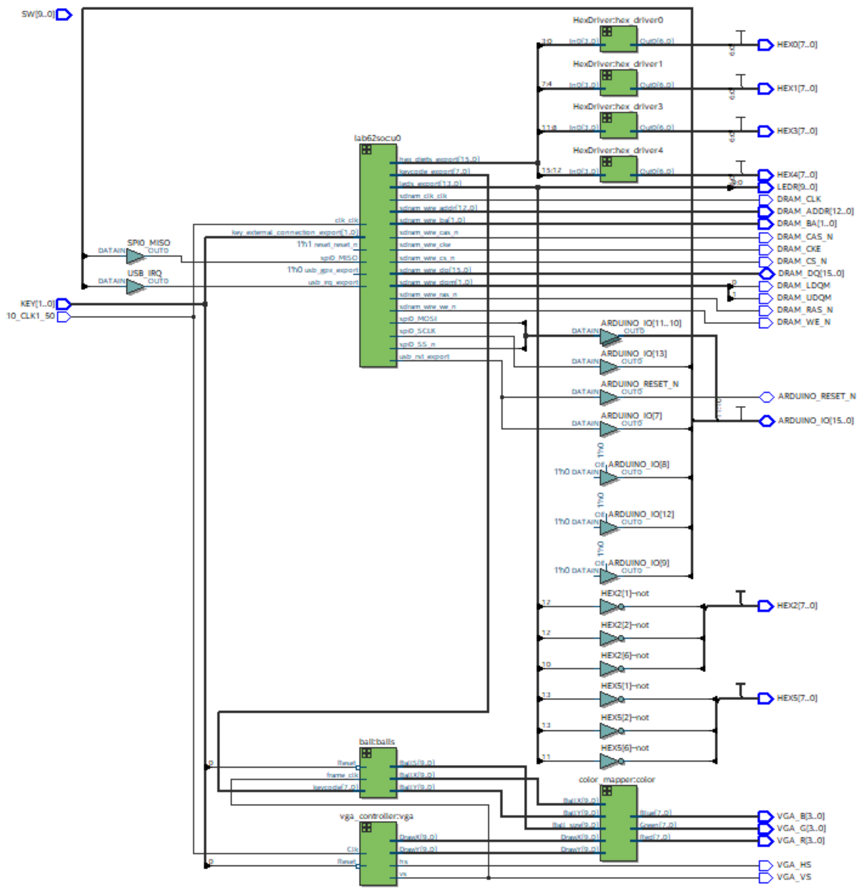
Purpose: Using a target register specified by the byte 'reg' and a byte specifying the number of bytes to be read, a read operation is performed on a register using a pointer that is provided. The pointer plus the number of bytes read is then returned.

Function: MAXreg_wr

Parameters: BYTE reg, BYTE val

Return: None Purpose: A target register which is reg+2 is written into with a byte of data labeled by ‘val’.
Function: MAXbytes_wr Parameters: BYTE reg, BYTE nbytes, BYTE* data Return: BYTE* data+nbytes Purpose: Using a target register + 2 specified by the byte ‘reg’ and a byte specifying the number of bytes, a write operation is performed on a register using a data pointer and that pointer plus the number of bytes is returned.

Top Level Block Diagram



Written Description of .sv Modules

Module: lab61.sv

Lab 6.1 or 6.2: 6.1

```
input         MAX10_CLK1_50,
                input  [1:0]  KEY,
                input         [7:0]   SW,
                output [7:0]  LEDR,
                output [12:0] DRAM_ADDR,
                output [1:0]  DRAM_BA,
                output        DRAM_CAS_N,
                output            DRAM_CKE,
                output            DRAM_CS_N,
                inout  [15:0] DRAM_DQ,
                output                DRAM_LDQM,
                output                DRAM_UDQM,
                output            DRAM_RAS_N,
                output            DRAM_WE_N,
                output            DRAM_CLK
```

Description: This module acts as the top-level for week 1. It instantiates the SOC made to allow the NIOS system to work in conjunction with the FGA and memory. Also allows the LEDS to be displayed on the FPGA. Also allows the KEYS to be connected to the SOC and the NIOS system to reset and accumulate. Also allows the switches to be imputed into the SOC to be accumulated.

Purpose: Top level module for the first week of lab 6.

Module: lab62.sv

Lab 6.1 or 6.2: 6.2

```
///////// Clocks /////////
input     MAX10_CLK1_50,

///////// KEY /////////
input    [ 1: 0]   KEY,

///////// SW /////////
input    [ 9: 0]   SW,

///////// LEDR /////////
output   [ 9: 0]   LEDR,

///////// HEX /////////
output   [ 7: 0]   HEX0,
output   [ 7: 0]   HEX1,
output   [ 7: 0]   HEX2,
output   [ 7: 0]   HEX3,
output   [ 7: 0]   HEX4,
output   [ 7: 0]   HEX5,

///////// SDRAM /////////
output             DRAM_CLK,
output             DRAM_CKE,
output   [12: 0]   DRAM_ADDR,
output   [ 1: 0]   DRAM_BA,
inout    [15: 0]   DRAM_DQ,
output             DRAM_LDQM,
output             DRAM_UDQM,
output             DRAM_CS_N,
output             DRAM_WE_N,
output             DRAM_CAS_N,
output             DRAM_RAS_N,

///////// VGA /////////
output             VGA_HS,
output             VGA_VS,
output   [ 3: 0]   VGA_R,
output   [ 3: 0]   VGA_G,
output   [ 3: 0]   VGA_B,


///////// ARDUINO /////////
inout    [15: 0]   ARDUINO_IO,
inout              ARDUINO_RESET_N
```

Description: This is the top level module that controls the entire system. It instantiates the SOC, VGA controller, Ball, and Color Mapper. The inputs and outputs to the FPGA are also set here, such as the VGA port pins and the memory and the USB pins (denoted by Arduino).

Maps the inputs and outputs of each module and SOC to allow them to communicate together and with the FPGA.

Purpose: Top_Level Module for week 2 of lab 6. Used to instantiate modules and SOC.

Module: VGA_controller.sv

Lab 6.1 or 6.2: 6.2

```
input        Clk,       // 50 MHz clock
             Reset,     // reset signal
output logic hs,        // Horizontal sync pulse.  Active low
             vs,        // Vertical sync pulse.  Active low
             pixel_clk, // 25 MHz pixel clock output
             blank,     // Blanking interval indicator.  Active low.
             sync,      // Composite Sync signal.  Active low.  We don't use it in this lab,
                        //   but the video DAC on the DE2 board requires an input for it.
output [9:0] DrawX,     // horizontal coordinate
             DrawY );   // vertical coordinate
```

Description: This module is the controller which controls the behavior of the VGA port and provides signals to the monitor and the rest of the modules. This module has two counters, a vertical and horizontal one which provide a HS and VS signal to sync the monitor with the FPGA. The module also provides a DrawX and Y signal which is the position of the current pixel being drawn. A blank signal is also provided to show nothing should be drawn. The pixel_clock was not used in this lab, as well as the sync signal.

Purpose: Module used to control and Sync the VGA port of the FPGA with the monitor.

Module: ball.sv

Lab 6.1 or 6.2: 6.2

```
input Reset, frame_clk,
input [7:0] keycode,
output [9:0]  BallX, BallY, BallS
```

Description: This module is used to define what the ball is, its coordinates and its size. This module also defines the motion of the ball, as to how it changes its coordinates with time. Note that the frame_clk is actually VS which appears at 60hz frequency. The ball first defines the ball in the center of the screen with 0 motion and a set size. It then waits for a keycode signal and then it begins moving in that direction until it hits the wall in which it bounces back. If another command is given while it moves it changes direction to match that direction. Only one direction is accepted at a time.

Purpose: Module used to define the structure of the Ball, its size, and its coordinates. Used to control the motion of the ball by using the keycode and to ensure it doesn't pass through the wall.

Module: Color_Mapper.sv

Lab 6.1 or 6.2: 6.2

```
input        [9:0] BallX, BallY, DrawX, DrawY, Ball_size,
output logic [7:0]  Red, Green, Blue );
```

Description: This module takes in the coordinates of the current pixel being drawn as well as the ball coordinates and size and if the current pixel is within the ball then the RGB colors are set to the color of the ball. Otherwise the mapper draws the background color which depends on the horizontal position. This color mapper actually defines what a ball is by using x^2+y^2<=r^2. Here r = ball size and x and y are the ball coordinates. If the current pixel is within that inequality then it knows to draw it as a ball.

Purpose: This module is used to draw the ball on the screen as well as the background colors,

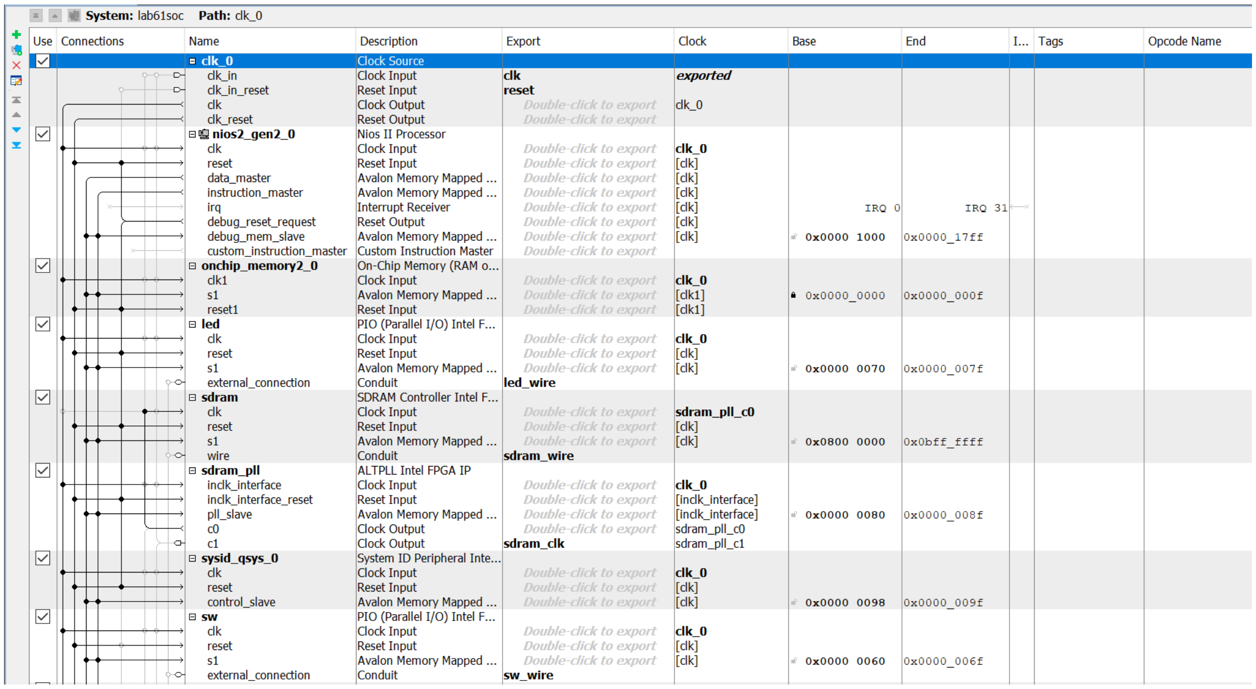
System Level Block Diagram

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).

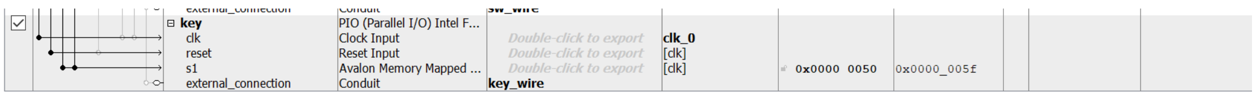
b. Note that this is not trivial, as there are many components within the Platform Designer view.

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.

Lab 6.1 Platform Designer View:

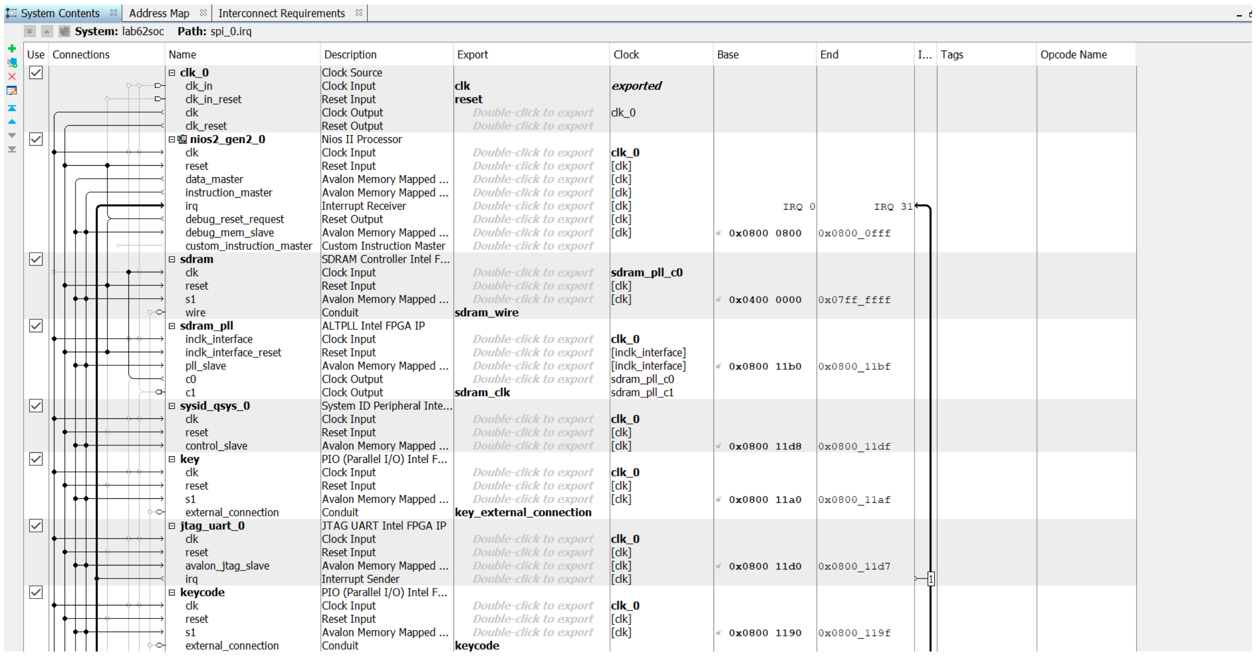
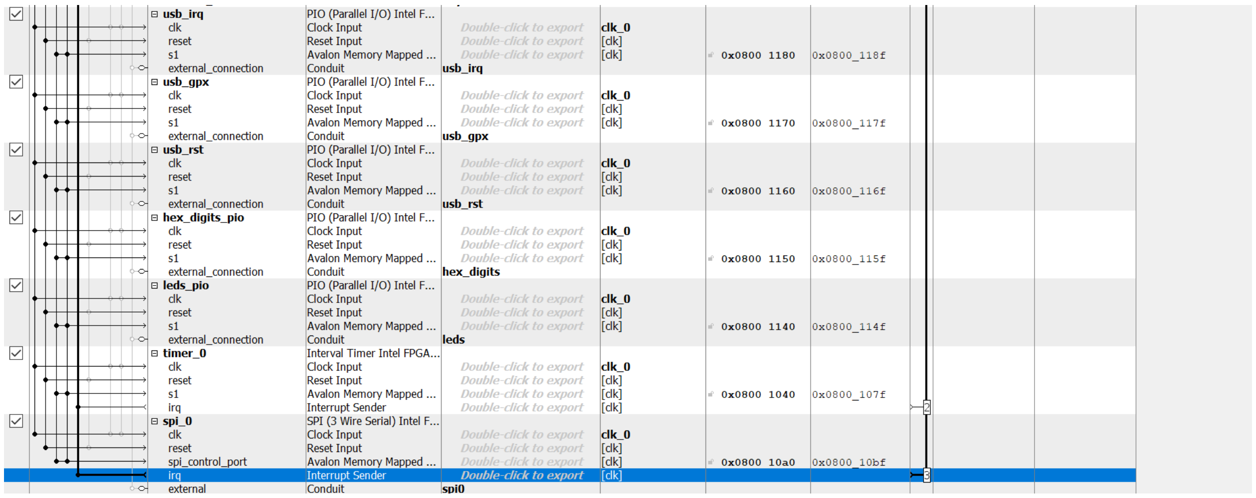

System: lab61soc Path: clk_0

Use	Connections	Name	Description	Export	Clock	Base	End	I...	Tags	Opcode Name
✓		clk_0	Clock Source							
		clk_in	Clock Input	**clk**	*exported*					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	*Double-click to export*	clk_0					
		clk_reset	Reset Output	*Double-click to export*						
✓		**nios2_gen2_0**	Nios II Processor							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		data_master	Avalon Memory Mapped ...	*Double-click to export*	[clk]					
		instruction_master	Avalon Memory Mapped ...	*Double-click to export*	[clk]					
		irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ 31			
		debug_reset_request	Reset Output	*Double-click to export*	[clk]					
		debug_mem_slave	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0000 1000	0x0000_17ff			
		custom_instruction_master	Custom Instruction Master	*Double-click to export*						
✓		**onchip_memory2_0**	On-Chip Memory (RAM o...							
		clk1	Clock Input	*Double-click to export*	**clk_0**					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk1]	0x0000_0000	0x0000_000f			
		reset1	Reset Input	*Double-click to export*	[clk1]					
✓		**led**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0000 0070	0x0000_007f			
		external_connection	Conduit	**led_wire**						
✓		**sdram**	SDRAM Controller Intel F...							
		clk	Clock Input	*Double-click to export*	**sdram_pll_c0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 0000	0x0bff_ffff			
		wire	Conduit	**sdram_wire**						
✓		**sdram_pll**	ALTPLL Intel FPGA IP							
		inclk_interface	Clock Input	*Double-click to export*	**clk_0**					
		inclk_interface_reset	Reset Input	*Double-click to export*	[inclk_interface]					
		pll_slave	Avalon Memory Mapped ...	*Double-click to export*	[inclk_interface]	0x0000 0080	0x0000_008f			
		c0	Clock Output	*Double-click to export*	sdram_pll_c0					
		c1	Clock Output	**sdram_clk**	sdram_pll_c1					
✓		**sysid_qsys_0**	System ID Peripheral Inte...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		control_slave	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0000 0098	0x0000_009f			
✓		**sw**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0000 0060	0x0000_006f			
		external_connection	Conduit	**sw_wire**						



Use	Connections	Name	Description	Export	Clock	Base	End	I...	Tags	Opcode Name
✓		**key**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0000 0050	0x0000_005f			
		external_connection	Conduit	**key_wire**						

<u>Lab 6.2 Platform Designer View:</u>

System Contents | Address Map | Interconnect Requirements

System: lab62soc **Path:** spi_0.irq

Use	Connections	Name	Description	Export	Clock	Base	End	I...	Tags	Opcode Name
✓		**clk_0**	Clock Source							
		clk_in	Clock Input	**clk**	*exported*					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	Double-click to export	clk_0					
		clk_reset	Reset Output	Double-click to export						
✓		**nios2_gen2_0**	Nios II Processor							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		data_master	Avalon Memory Mapped ...	Double-click to export	[clk]					
		instruction_master	Avalon Memory Mapped ...	Double-click to export	[clk]					
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31			
		debug_reset_request	Reset Output	Double-click to export	[clk]					
		debug_mem_slave	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 0800	0x0800_0fff			
		custom_instruction_master	Custom Instruction Master	Double-click to export						
✓		**sdram**	SDRAM Controller Intel F...							
		clk	Clock Input	Double-click to export	**sdram_pll_c0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0400 0000	0x07ff_ffff			
		wire	Conduit	**sdram_wire**						
✓		**sdram_pll**	ALTPLL Intel FPGA IP							
		inclk_interface	Clock Input	Double-click to export	**clk_0**					
		inclk_interface_reset	Reset Input	Double-click to export	[inclk_interface]					
		pll_slave	Avalon Memory Mapped ...	Double-click to export	[inclk_interface]	0x0800 11b0	0x0800_11bf			
		c0	Clock Output	Double-click to export	sdram_pll_c0					
		c1	Clock Output	**sdram_clk**	sdram_pll_c1					
✓		**sysid_qsys_0**	System ID Peripheral Inte...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		control_slave	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 11d8	0x0800_11df			
✓		**key**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 11a0	0x0800_11af			
		external_connection	Conduit	**key_external_connection**						
✓		**jtag_uart_0**	JTAG UART Intel FPGA IP							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		avalon_jtag_slave	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 11d0	0x0800_11d7			
		irq	Interrupt Sender	Double-click to export	[clk]			1		
✓		**keycode**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 1190	0x0800_119f			
		external_connection	Conduit	**keycode**						
✓		**usb_irq**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 1180	0x0800_118f			
		external_connection	Conduit	**usb_irq**						
✓		**usb_gpx**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 1170	0x0800_117f			
		external_connection	Conduit	**usb_gpx**						
✓		**usb_rst**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 1160	0x0800_116f			
		external_connection	Conduit	**usb_rst**						
✓		**hex_digits_pio**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 1150	0x0800_115f			
		external_connection	Conduit	**hex_digits**						
✓		**leds_pio**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 1140	0x0800_114f			
		external_connection	Conduit	**leds**						
✓		**timer_0**	Interval Timer Intel FPGA...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 1040	0x0800_107f			
		irq	Interrupt Sender	Double-click to export	[clk]			2		
✓		**spi_0**	SPI (3 Wire Serial) Intel F...							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		spi_control_port	Avalon Memory Mapped ...	Double-click to export	[clk]	0x0800 10a0	0x0800_10bf			
		irq	Interrupt Sender	Double-click to export	[clk]			3		
		external	Conduit	**spi0**						

Functionality of IP Blocks

IP Block: nios2_gen2_0 (NIOS II Processor) Lab 6.1 or Lab 6.2: Both Functionality: This is the CPU that is used in this lab. It is in charge of the accumulation circuit and the USB/SPI communication. Main CPU where the C code runs on. Also used to access different parts of the Avalon Bus through the C code.
IP Block: sdram Lab 6.1 or Lab 6.2: both Functionality: Memory used to store information for the CPU. Maps to the SDRAM on the FPGA. Used as high capacity memory for the CPU to store and run programs.

IP Block: sdram_pll
Lab 6.1 or Lab 6.2: both
Functionality: This module is used to create a clock signal that is out of phase from the main clock at a specific angle. This is used to drive the SDRAM controller and the FPGA SDRAM at different clocks. This skew is needed due to the precise timing of the SDRAM and is used to compensate for the delay and inaccuracies between the SDRAM and the controller.

IP Block: sysid_qsys
Lab 6.1 or Lab 6.2: both
Functionality: This is used to ensure compatibility between the hardware and software components, for example the NIOS system and the loaded program on the FPGA. Gives a warning when there is a mismatch of the serial number provided. Used for warning and debugging purposes.

IP Block: key
Lab 6.1 or Lab 6.2: both
Functionality: This module is a PIO, a parallel IO module. Specifically this is a 2-bit input module that is mapped to the buttons on the FPGA. This is used for the accumulation circuit, where run is needed to be pressed to see when to accumulate, and reset in some other cases.

IP Block: jtag_uart
Lab 6.1 or Lab 6.2: 6.2
Functionality: This module is used to communicate with the NIOS system while it's running through the computer terminal. This allows statements such as print and scan to work in C, and is used mainly for debugging.

IP Block: keycode
Lab 6.1 or Lab 6.2: 6.2
Functionality: This is a PIO block. This is a 8-bit output port that is a part of the SOC. This is used to export the read character from the keyboard to the top-level sv module which then goes into the ball module.

IP Block: usb_irq
Lab 6.1 or Lab 6.2: 6.2
Functionality: This is a 1-bit PIO block. It is in input mode. This is used as an interrupt port that the MAX chip uses. Used for interrupt functionality in some cases. This is mapped to the chip via the top sv module and the FPGA pin assignments.

IP Block: usb_gpx
Lab 6.1 or Lab 6.2: 6.2
Functionality: This is a 1-bit input PIO block. It is required for the MAX chip, and is a general purpose push pull output. Functionality is decided by the chip and depends on different factors. This is mapped to the chip via the top sv module and the FPGA pin assignments.

IP Block: usb_rst
Lab 6.1 or Lab 6.2: 6.2

Functionality: This is a 1-bit output PIO block. It is required for the MAX chip. This is mapped to the chip via the top sv module and the FPGA pin assignments.
IP Block: hex_digits Lab 6.1 or Lab 6.2: both Functionality: This is a 16-bit output PIO block. This is used to output the hex of the keycode. Used to map the FPGA ports to the SOC in the top-level to display on the HEX display. The NIOS system sets the output according to the current keyboard presses.
IP Block: led_pio Lab 6.1 or Lab 6.2: 6.1 Functionality: This is a 14-bit output PIO block. This is used to display the LED accumulation on the FPGA leds. Mapped to the ports of the FPGA from the top-level SOC.
IP Block: timer Lab 6.1 or Lab 6.2: 6.2 Functionality: This is a timer that keeps track of the timeouts that the USB requires. This is required for the USB SPI implementation on the MAX chip. An interrupt is assigned which is connected to the NIOS system.
IP Block: spi_0 Lab 6.1 or Lab 6.2: 6.2 Functionality: This is the SPI port peripheral that is required to use SPI between the NIOS systema and the MAX chip. In this case the NIOS system is the host and MAX is the slave. The SPI ports are connected via the top-level to the MAX chip via the FPGA.
IP Block: sw Lab 6.1 or Lab 6.2: 6.1 Functionality: This is a 10-bit input PIO block. This is used as an input to the AVALON bus which allows the NIOS system to read an input from the FPGA/USER and use that as the input to the accumulation program.
IP Block: onchip_memory Lab 6.1 or Lab 6.2: 6.1 Functionality: This block is on chip memory that is instantiated. This can be used as memory to run the NIOS program or to hold other types of information. This was not used in our lab and was used as a placeholder for use in the future.

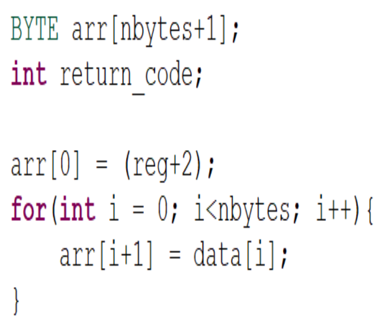
Software Component

a. One of the INQ questions asks about the blinker code, but you must also describe your accumulator.

The accumulator code is very simple. The Main C file starts by initializing a variable called sum and setting the LEDS to 0. The function then goes into an infinite loop. There are then two if statements. If run is pressed, then the switches are read and added to sum. After that, if the sum is greater than 255 then sum = 0. Note that this might be changed depending on the desired overflow behavior. Another part is that a wait statement is included in the while loop at the end. This is because when run is pressed it accumulates multiple times which is not desired. This was solved by adding a for loop that waits a set amount which allows the button to go back to 0. The PIO blocks were accessed using pointers that are mapped to their respective positions on the AVALON bus. Note that a volatile structure was used for this. This is because the switches and buttons can be changed while the software is unaware.

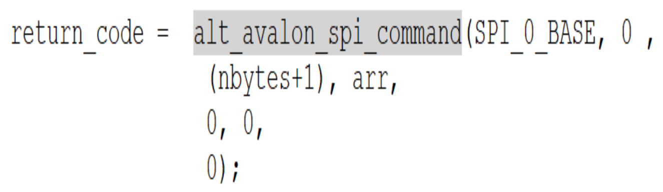
b. Describe the code you needed to fill in for the USB/SPI portion of the lab for Lab 6.2

In lab 6.2 we needed to fill in 4 functions for the USB/SPI to work properly. These were the read and write functions of the SPI from the NIOS system to the MAX chip. To implement these functions we used the alt_avalon_spi_command and followed the manual for the instructions. To implement write, we first wrote the command byte then the value we wanted to write. This needed to be implemented using pointers so we created an array that had reg+2 and val, with reg+2 being the command byte and val being what we wanted to write. We then did a similar thing for writing nbytes, but instead of val we had data[i] which we copied using a for loop. This could have been done using multiple statements instead but this was more compact. To implement the read function we first wrote the command byte using a pointer to reg and then we used a pointer to where we wanted to store the data we read. We then either returned the value read or a pointer to the end of the data pointer. Some examples can be seen below:

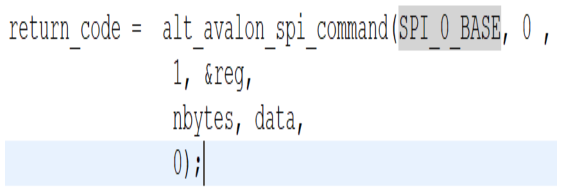


```
BYTE arr[nbytes+1];
int return_code;

arr[0] = (reg+2);
for(int i = 0; i<nbytes; i++){
    arr[i+1] = data[i];
}
```



```
return_code =  alt_avalon_spi_command(SPI_0_BASE, 0 ,
                (nbytes+1), arr,
                0, 0,
                0);
```



```
return_code =  alt_avalon_spi_command(SPI_0_BASE, 0 ,
               1, &reg,
               nbytes, data,
               0);
```

Post Lab Questions/INQ Questions

1. Refer to the Design Resources and Statistics in IQT.17-19 and complete the following design statistics table

LUT	3181
DSP	10
BRAM	11264
Flip-Flop	2458
Frequency Mhz	73.83
Static Power mW	96.52 mW
Dynamic Power mW	61.84 mW
Total Power mW	180.05 mW

Answers to INQ Questions

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

The ‘e’ stands for the economical version of the NIOS-II CPU and the ‘f’ stands for the functional version of the CPU. For someone programming an FPGA with the CPU, the two versions of the same CPU provide them with the ability to make a decision to either use a CPU that uses less logical components or a CPU that has higher performance. In this lab, we use NIOS-II/e because we would prefer less logical utilization and the tasks that we are performing do not require high performance.

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

Access speeds are greater for data that is utilized more often because of its shorter connection distance than other methods of memory. This could make a considerable difference when it comes to program execution speeds.

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

The NIOS-II CPU is a “modified Harvard” CPU. The major difference between a Harvard and Von-Neumann machine is that a Von-Neumann machine does not separate data and instructions, and both are stored in the same block of memory. A Harvard model completely divides the data and instructions into two sets of memory. The NIOS-II CPU has two different datapaths for the data and the instructions but they can be stored in the same on-chip memory.

4. Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?

The LED peripheral is just an output, so it does not need to have access to the program bus. The on-chip memory needs access to the data and program bus because it's constantly fetching and writing data.

5. Why does SDRAM require constant refreshing?

SDRAM uses capacitors and transistors for every bit of information, and the charge held in the capacitors will slowly discharge over time, meaning if you would like to keep the data stored in SDRAM, you will need to constantly refresh.

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

The data width is 4 bytes, and you divide that value by 5.5ns to get 727MB/s.

7. 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?

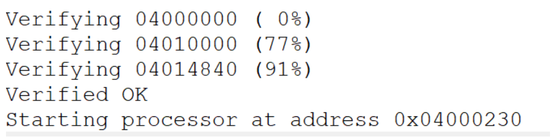
Incorrect values will be reported from the SDRAM if it is run too slowly. This is due to the possibility of metastability which only allows the SDRAM to run in a specific interval of frequencies without metastability occurring.

8. You must now make a second clock, which goes out to the SDRAM chip itself, as recommended by Figure 11. Make another output by clicking clk c1, and verify it has the same settings, except that the phase shift should be -1ns. This puts the clock going out to the SDRAM chip (clk c1) 1ns behind the controller clock (clk c0). Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.

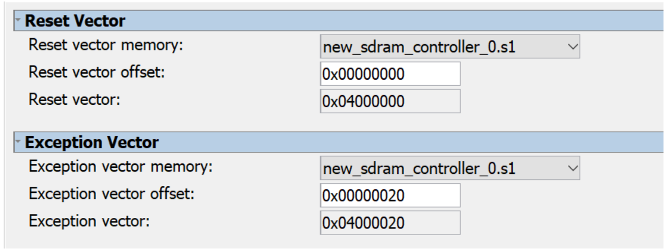
We do this to wait until the values that are being stored and refreshed in the SDRAM are stable enough. If we don't delay the clock of the SDRAM we could have inaccurate reportings of memory data from the SDRAM.

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

For Lab 6.1, Platform Designer had assigned the base address of x0800 0000 for the SDRAM IP Block. This corresponds to the address that the NIOS-II starts its execution from for the first part of the lab. For Lab 6.2, Platform Designer had assigned the base address of x0400 0000 for the SDRAM, so for the second part of the lab the NIOS-II started execution from that memory address. This is done after assigning the address in the case that there's a reset or exception.



```
Verifying 04000000 ( 0%)
Verifying 04010000 (77%)
Verifying 04014840 (91%)
Verified OK
Starting processor at address 0x04000230
```



Reset Vector

Reset vector memory:	new_sdram_controller_0.s1
Reset vector offset:	0x00000000
Reset vector:	0x04000000

Exception Vector

Exception vector memory:	new_sdram_controller_0.s1
Exception vector offset:	0x00000020
Exception vector:	0x04000020

10. You must be able to explain what each line of this (very short) program does to your TA. Specifically, you must be able to explain what the volatile keyword does (line 8), and how the set and clear functions work by working out an example on paper (lines 13 and 16).

The 'volatile' keyword indicates that the variable that is created may change after the execution of the program. 'Clear' sets the LSB LED to a low state and 'Set' does the opposite, where the LSB LED is set to high.

11. Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code:

Segment	**Meaning**	**Example**
.bss	Uninitialized region	int x;
.heap	Heap memory region	int * heap = (int*) malloc (sizeof(int))
.rodata	Read-only data	Const int x = 0;
.rwdata	Read-write data	int x = 0;
.stack	Last-In-First-Out structure	int lifo(a,b,c,d);
.text	String	string x = “string”;

<u>Conclusion</u>

Overall, both of the parts of the lab were pretty straightforward, and that is mostly due to the detailed *Introduction to NIOS-II and Qsys* document, which included a step-by-step procedure for setting up and editing/configuring the SoC for both parts of the lab. Our implementation of the lab ended up being completely functional but we were experiencing some problems when we connected the MAX10 to the monitor. One of our FPGAs was cutting the leftmost columns of the pixels and making them black. We had that FPGA connected to a Macbook and for that Macbook we had a VGA-to-HDMI adapter that was connected to the monitor. We had identical code programmed on another FPGA and the problem did not occur, so we believe that the problem was due to the VGA-to-HDMI adapter.

We didn't find anything in the Lab manual overly ambiguous or difficult to understand, especially due to the fact that there were a lot of supplemental resources that were provided.

<u>Extra Credit:</u>

Problem: The problem is that it is possible to make the ball go outside the screen by holding a direction such as up while it is bouncing causing it to go through the wall.

Solution(Code Below): This was solved by creating two flags, one for the X and one for the Y direction. If the ball is currently on any of the edges then the flag signal goes high. This tells the program to then ignore any key presses that come in. If the ball is not on the edge it then sets the flags = 0 and loops back(always_ff). This causes the ball to essentially bounce whenever it hits the wall regardless of keypress. Whenever the ball is not contacting a wall the keypress resumes.

```
        begin
             if ( (Ball_Y_Pos + Ball_Size) >= Ball_Y_Max )  begin // Ball is at the bottom edge, B
                 Ball_Y_Motion <= (~ (Ball_Y_Step) + 1'b1);  // 2's complement.
                 flagy = 1;
                 end
             else if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min ) begin  // Ball is at the top edge,
                 Ball_Y_Motion <= Ball_Y_Step;
                                flagy = 1;
end
              else if ( (Ball_X_Pos + Ball_Size) >= Ball_X_Max ) begin  // Ball is at the Right ed
                 Ball_X_Motion <= (~ (Ball_X_Step) + 1'b1);  // 2's complement.
                                flagx = 1;
end
             else if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min ) begin  // Ball is at the Left edge
                 Ball_X_Motion <= Ball_X_Step;
                                flagx = 1;

end
             else begin
                 Ball_Y_Motion <= Ball_Y_Motion;  // Ball is somewhere in the middle, don't bounce
                 flagx = 0;
                 flagy = 0;
             end
             if (flagx==0) begin
               if(flagy==0) begin
             case (keycode)
               8'h04 : begin
                        if(flagx!=1)
                        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
             end
             Ball_Y_Pos <= (Ball_Y_Pos + Ball_Y_Motion);  // Update ball position
             Ball_X_Pos <= (Ball_X_Pos + Ball_X_Motion);
                flagx = 0;
                flagy = 0;
```