ECE 385

Spring 2023

Experiment #6

SOC with NIOS II in SystemVerilog

Dylan Bautista, Matthew Wei

Lab Section NL/ April 1 11:00 AM

Nick Lu

Experiment #6

SOC with NIOS II in System Verilog

I. Introduction/Purpose of Circuit:

Basic functionality of the NIOS-II processor running on the MAX10 FPGA

Experiment 6 implemented a System-On-Chip design, initially with just basic peripherals such as LEDs and switches, and in week 2 with added functionality of USB and VGA. The NIOS-II processor is a 32-Bit CPU that is usually used for low performance tasks such as I/O. NIOSII is instantiated as an IP block using Platform Designer, and then programmed using a high-level language, in our case, C. Our week 1 assignment utilized the NIOSII processor to accept and read in data from switches in order to display an accumulation of bit values on the LED's. The Second week utilized the USB driver on the NIOSII in order to communicate with the MAX3421 via the SPI peripheral, a USB host controller. This would facilitate the overall design of accepting keyboard input to control a virtual ball's movement on a VGA monitor.

Operation of the USB/VGA interface

For week 2, we implemented a USB host controller called the MAX3421E, which communicates with the NIOSII processor using SPI protocol. To accomplish this, we added modules such as SPI and additional PIO ports, then modified the C code to call the SPI device driver and enable register reading and writing. Lastly, the top-level System Verilog entity has to be edited with the proper instantiations of the VGA_controller, Color_mapper, and Ball modules, which will determine the drawing of the balls position.

II. Written description of Circuit

Hardware component of the lab – Platform Designer

For 6.1, the modules we have in our platform designer are the nios2_gen2_0, which is the 32-Bit CPU with Avalon bus, onchip_memory for the accumulator, the SDRAM interface for NIOSII software, sdram_pll and clk_0 for timing, as well as led, SW, and an accumulate block for I/O. For the second week, we got rid of the on-chip memory, added a jtag_uart module for communication with NIOSII to use print and scan statements, systemID peripheral, an interval timer, and SPI port for MAX3421E communication. Additionally there are PIOs key code, key, usb_irq for interrupt, usb_rst for reset, usb_gpx, hex_digits_pio, and leds_pio,.

Lab 6.1- how the I/O works

The NIOSII processor block in Platform designer provides a data and instruction bus for which input data like switches, output data like LED's and program data to fetch and write like the On-Chip memory block are all interconnected. Certain signals that need to be connected externally to the System Verilog code can be exported with a connection name. The NIOSII system can be integrated into a Quartus project, creating a SOC module with corresponding connection names. The Top-level system Verilog file is used to connect the NIOSII to the outside world, with the c program in eclipse utilizing the base addresses of the SOC to read, work with, and output data to PIO ports. Therefore, switch data can be read, interpreted in the C code, with a corresponding response on the LED's.

How the NIOS interacts with the MAX3421E USB chip/ Written description of the SPI protocol

The NIOSII processor interacts with the MAX3421E by using an SPI peripheral, created as a block within platform designer and connected to the nios2 block. 4 functions needed to be completed within the MAX3421 C file in order to facilitate communication between the NIOSII device driver and SPI peripheral. The SPI protocol is a synchronous serial system with USB functionality. It has a clock signal, MISO, MOSI, and SS. MOSI is synchronized to the clock signal and sends data from the NIOSII to MAX3421E. MISO is oppositely the synchronous data from MAX3421E to NIOSII. SS is slave select for multiple devices and allows for communication with a single slave device when its unique SS signal is held low. The protocol and MAX device support half and full duplex mode, which can be configured using SPI operations.

Purpose of each C function filled in/ Code needed to fill in for USB/SPI portion of the Lab 6.2

The register write to MAX3421E via SPI, multi-byte write, register read from MAX3421E via SPI, and multi-byte read functions were created using the alt_avalon_spi_command, which ultimately allows for the acceptance of USB keyboard data. The VGA components are housed at separate System Verilog modules to be instantiated in the top-level. The register write function writes reg+2 and val via SPI and returns an error if the return code from the operation is less than 0. The multibyte function is the same except instead of val, we write a sequence of nbytes of data using a for loop, also returning the position after last written. The read functions operate similarly, with register read returning the read val after writing a register, and multibyte-read reading data into a given array after writing a register.

How the NIOS interacts with the VGA components/ How Ball, Color Mapper, and the VGA controller modules interact

The VGA_controller produces the timing signals for the Horizontal and vertical sync, controlling the "electron beam" and thus outputting the current pixel being edited at a given time. The Color Mapper module is given the coordinates and size of the ball and will return RGB values depending on what DrawX and DrawY are. Finally, the ball module takes in the current keycode input and interprets it to determine where the Ball is moving within the clock cycle, returning the ball's coordinates. Within the top level, one instance of these three were formed, with Color Mapper receiving the current pixel coordinates from the VGA controller and the current ball location from the ball module, in order to output the RGB values as output for the top-level. Additionally, the remaining clocks, inputs and outputs are connected to these instances.

III. Software Incorporation

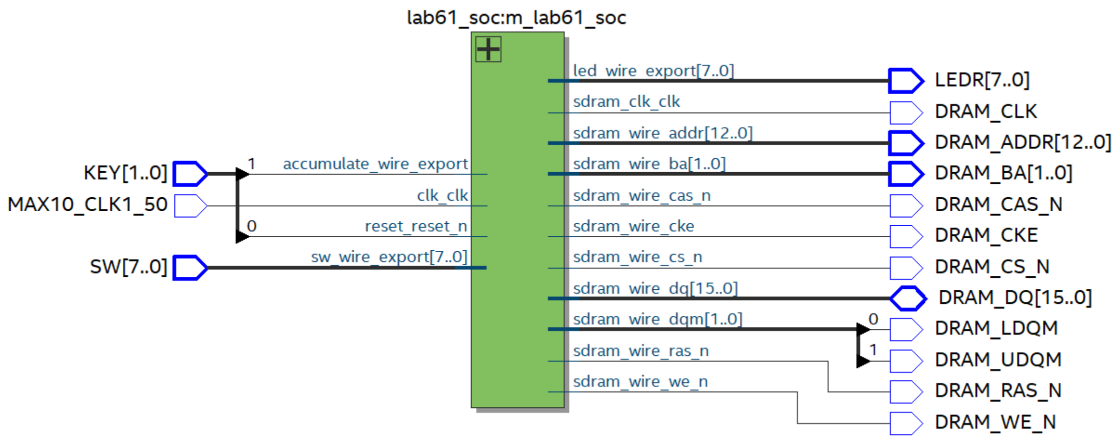
Description of the accumulator

The accumulator code written in C, was written in eclipse to control the data sent in from input and determine the data sent to output addresses. Our program first creates pointers to access the LED, switch and reset blocks. Then after clearing the sum variable and LEDs, an infinite while loop checks to see if the sum needs to be cleared and the current sum displayed on LED's. Next, after the end of the key press, if conditions are met, we add switch values to sum and check for overflow. In the case of overflow, we subtract the overflow value. After this, the LED's display the value of sum.

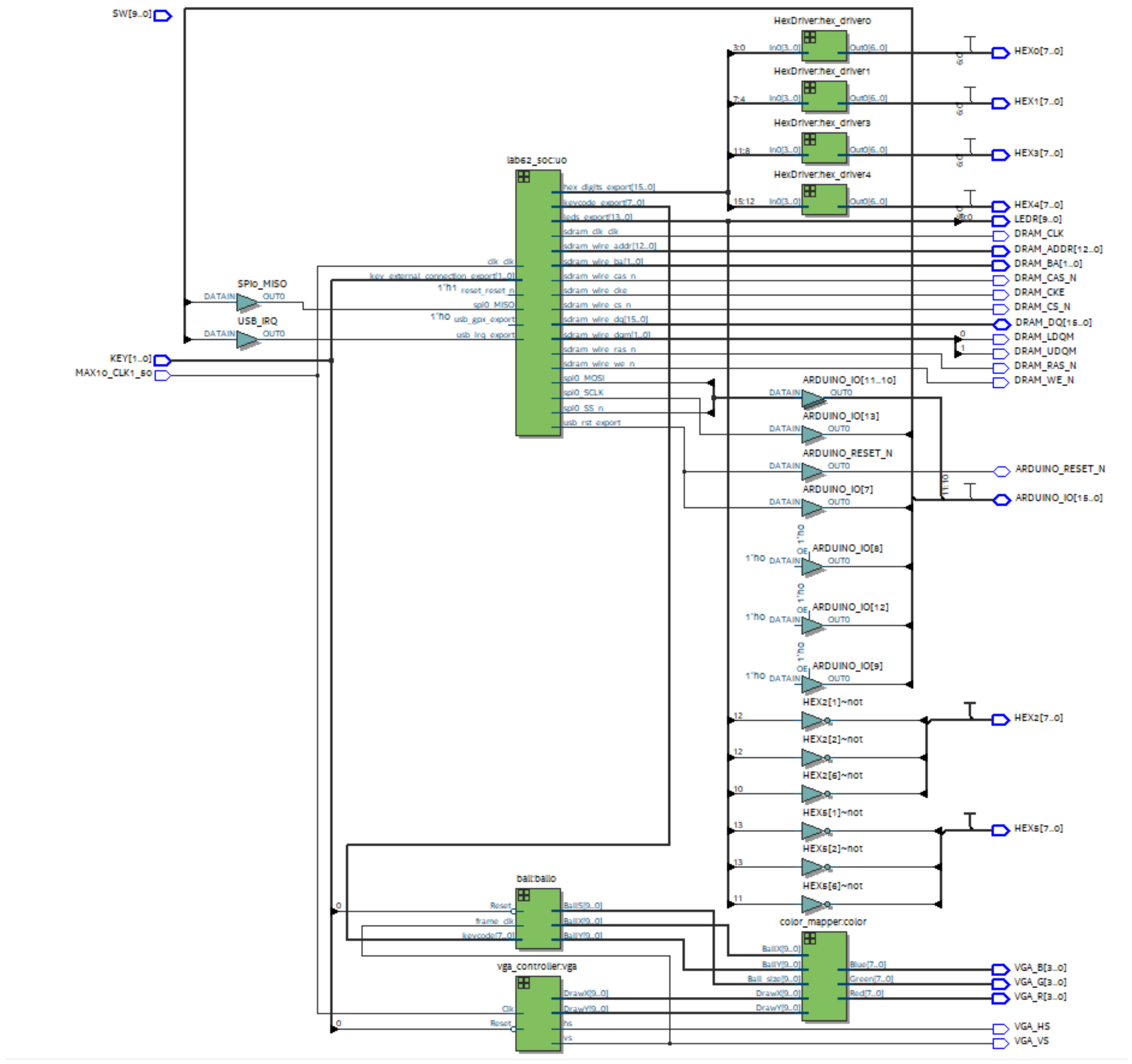
For the second part, four functions had to be implemented all utilizing the alt_avalon_spi_command. If any of the functions return a number less than 0, then an error occurred. Otherwise, the function executed properly. For MAXreg_wr, the function writes a register to the MAX3421E. We did this by using an array to store the register and the value to write. Afterwards, we simply filled in the parameters accordingly. For MAXbytes_wr, it does the same thing as the function before except it writes multiple bytes. A similar thing was done incorporate this except that we used a longer array. The 0th index holds the register and the rest of the array holds the values. Since the bytes to be written is passed as a parameter, we were able to do some math to figure out other values required in the parameters. For MAXreg_rd, the function reads a single register from the MAX3431E. Instead of an array, we created a local BYTE variable to store the read value in. The rest of the arguments to the Avalon function were filled in accordingly (same as previous functions) and the read value is returned from the function. Finally for MAXbytes_rd, this function is the multiple byte read version of the previous function. Like the write function, we are given the number of bytes to be read so using this value and doing some simple math, we can calculate our unknown parameters to the Avalon function. All in all, the four functions have a similar implementation where the parameters need to be filled in. Since the values are given to use, only some math and a few lines of code are needed to incorporate each of the functions.

IV. <u>Block Diagram:</u>

Top Level Diagram

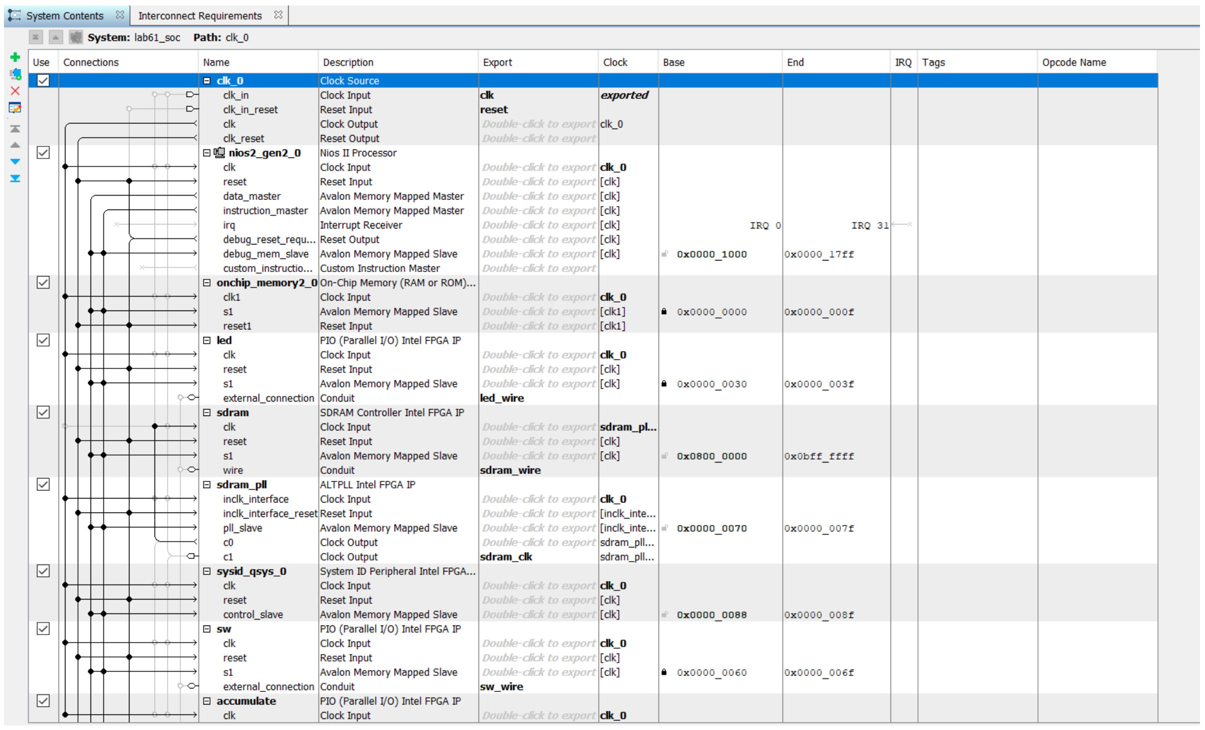


Lab 6.1 Top-Level Diagram



Lab 6.2 Top-Level Diagram

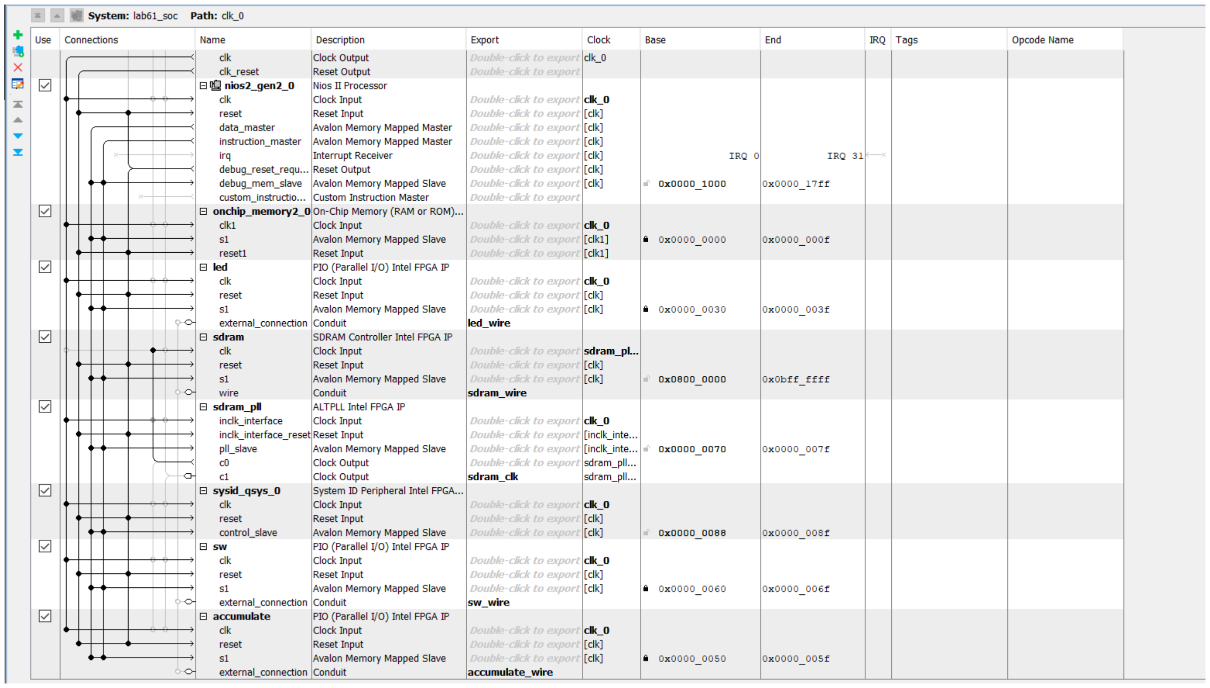
System Level Diagram



System Contents | Interconnect Requirements

System: lab61_soc **Path:** clk_0

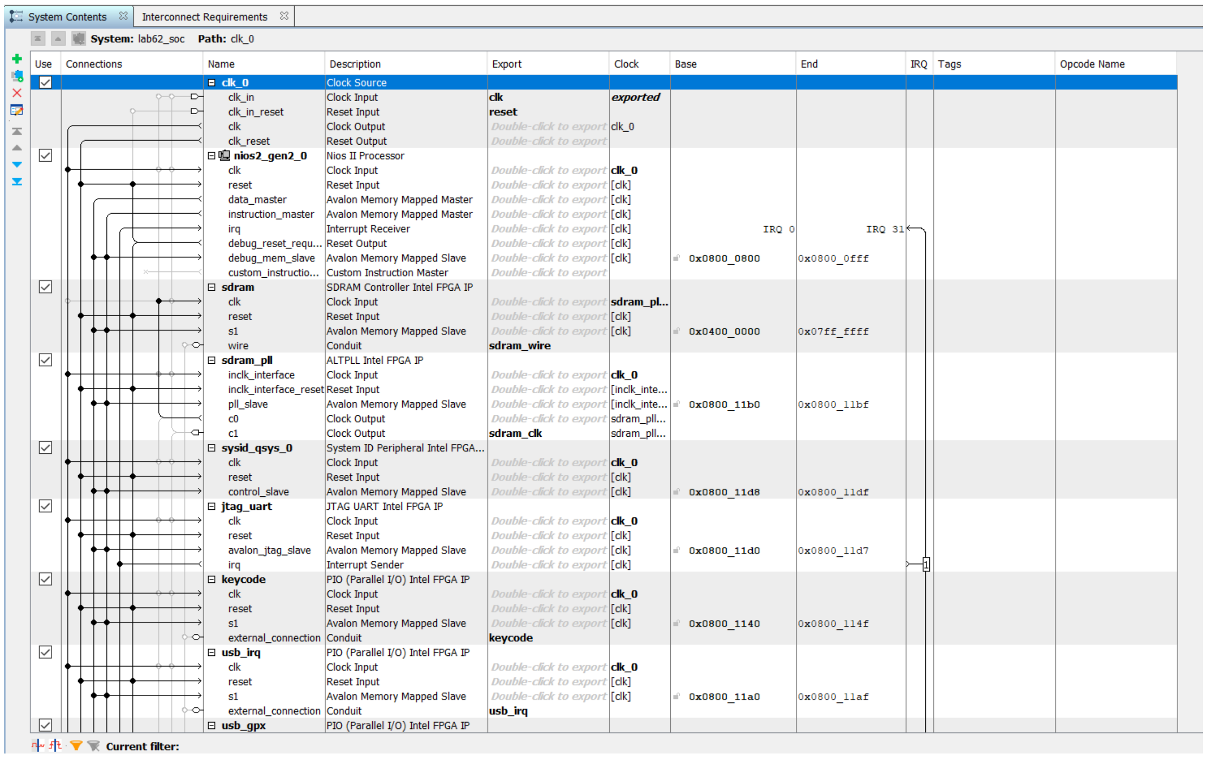
Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
✓		**clk_0**	Clock Source							
		clk_in	Clock Input	**clk**	*exported*					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	*Double-click to export*	clk_0					
		clk_reset	Reset Output	*Double-click to export*						
✓		**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 Master	*Double-click to export*	[clk]					
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]					
		irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ 31			
		debug_reset_requ...	Reset Output	*Double-click to export*	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_1000	0x0000_17ff			
		custom_instructio...	Custom Instruction Master	*Double-click to export*						
✓		**onchip_memory2_0**	On-Chip Memory (RAM or ROM)...							
		clk1	Clock Input	*Double-click to export*	**clk_0**					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0000_0000	0x0000_000f			
		reset1	Reset Input	*Double-click to export*	[clk1]					
✓		**led**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0030	0x0000_003f			
		external_connection	Conduit	**led_wire**						
✓		**sdram**	SDRAM Controller Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**sdram_pl...**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*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_inte...					
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_inte...	0x0000_0070	0x0000_007f			
		c0	Clock Output	*Double-click to export*	sdram_pll...					
		c1	Clock Output	**sdram_clk**	sdram_pll...					
✓		**sysid_qsys_0**	System ID Peripheral Intel FPGA...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0088	0x0000_008f			
✓		**sw**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0060	0x0000_006f			
		external_connection	Conduit	**sw_wire**						
✓		**accumulate**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					



System: lab61_soc **Path:** clk_0

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
		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 Master	*Double-click to export*	[clk]					
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]					
		irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ 31			
		debug_reset_requ...	Reset Output	*Double-click to export*	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_1000	0x0000_17ff			
		custom_instructio...	Custom Instruction Master	*Double-click to export*						
✓		**onchip_memory2_0**	On-Chip Memory (RAM or ROM)...							
		clk1	Clock Input	*Double-click to export*	**clk_0**					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0000_0000	0x0000_000f			
		reset1	Reset Input	*Double-click to export*	[clk1]					
✓		**led**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0030	0x0000_003f			
		external_connection	Conduit	**led_wire**						
✓		**sdram**	SDRAM Controller Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**sdram_pl...**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*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_inte...					
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_inte...	0x0000_0070	0x0000_007f			
		c0	Clock Output	*Double-click to export*	sdram_pll...					
		c1	Clock Output	**sdram_clk**	sdram_pll...					
✓		**sysid_qsys_0**	System ID Peripheral Intel FPGA...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0088	0x0000_008f			
✓		**sw**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0060	0x0000_006f			
		external_connection	Conduit	**sw_wire**						
✓		**accumulate**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0050	0x0000_005f			
		external_connection	Conduit	**accumulate_wire**						

Lab 6.1 Modules

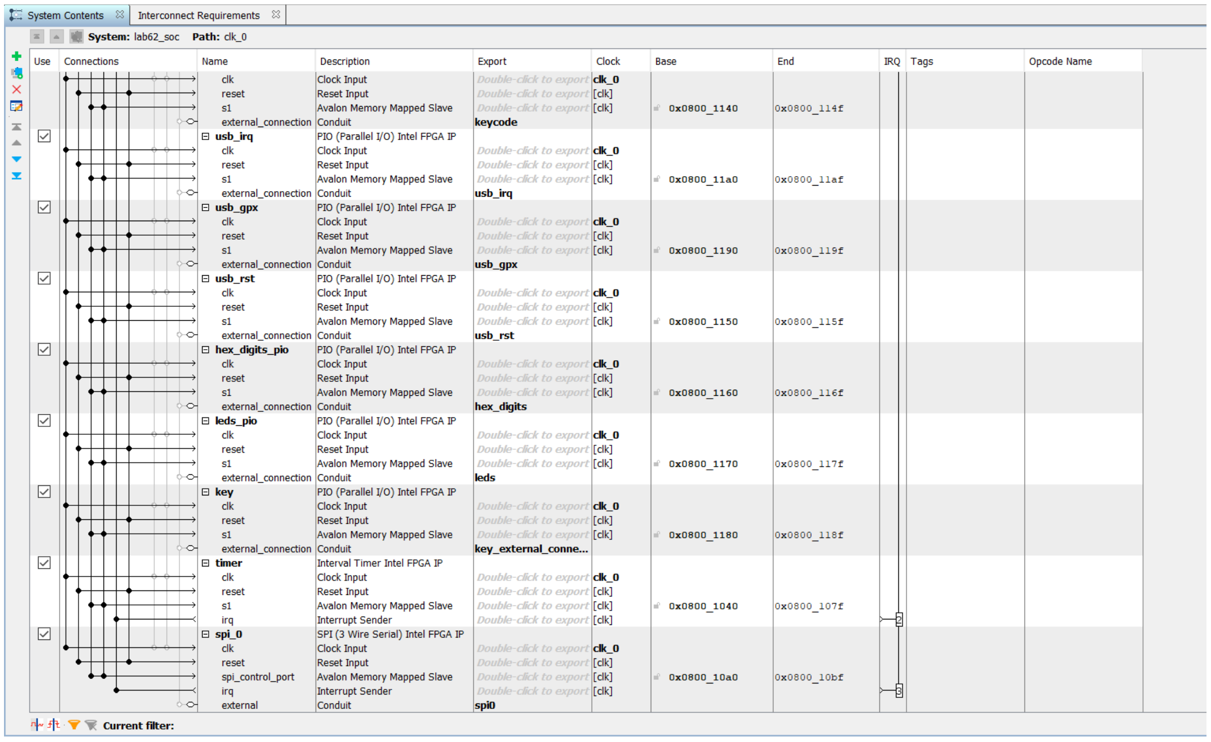


System Contents | Interconnect Requirements

System: lab62_soc **Path:** clk_0

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	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 Master	*Double-click to export*	[clk]					
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]					
		irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ 31			
		debug_reset_requ...	Reset Output	*Double-click to export*	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_0800	0x0800_0fff			
		custom_instructio...	Custom Instruction Master	*Double-click to export*						
✓		**sdram**	SDRAM Controller Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**sdram_pl...**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*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_inte...					
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_inte...	0x0800_11b0	0x0800_11bf			
		c0	Clock Output	*Double-click to export*	sdram_pll...					
		c1	Clock Output	**sdram_clk**	sdram_pll...					
✓		**sysid_qsys_0**	System ID Peripheral Intel FPGA...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_11d8	0x0800_11df			
✓		**jtag_uart**	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 Slave	*Double-click to export*	[clk]	0x0800_11d0	0x0800_11d7			
		irq	Interrupt Sender	*Double-click to export*	[clk]			1		
✓		**keycode**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_1140	0x0800_114f			
		external_connection	Conduit	**keycode**						
✓		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_11a0	0x0800_11af			
		external_connection	Conduit	**usb_irq**						
✓		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP							

Current filter:



System Contents | Interconnect Requirements

System: lab62_soc **Path:** clk_0

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_1140	0x0800_114f			
		external_connection	Conduit	**keycode**						
✓		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_11a0	0x0800_11af			
		external_connection	Conduit	**usb_irq**						
✓		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_1190	0x0800_119f			
		external_connection	Conduit	**usb_gpx**						
✓		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_1150	0x0800_115f			
		external_connection	Conduit	**usb_rst**						
✓		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_1160	0x0800_116f			
		external_connection	Conduit	**hex_digits**						
✓		**leds_pio**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_1170	0x0800_117f			
		external_connection	Conduit	**leds**						
✓		**key**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_1180	0x0800_118f			
		external_connection	Conduit	**key_external_conne...**						
✓		**timer**	Interval Timer Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*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 FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		spi_control_port	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_10a0	0x0800_10bf			
		irq	Interrupt Sender	*Double-click to export*	[clk]			3		
		external	Conduit	**spi0**						

Current filter:

Lab 6.2 Modules

V. Modules and Blocks:

SystemVerilog Modules

ball.sv

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

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

Description: Module that holds information on where the ball should be on the screen.

Purpose: Allows for the ball to obtain a position.

Color_Mapper.sv

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

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

Description: Module that determines color each pixel should be, either for the ball or background.

Purpose: Used to determine what color each pixel is.

HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: Module that holds information about the Hex LED displays on the FPGA board.

Purpose: Used so the correct output can be displayed onto the FPGAHEX displays.

lab61.sv

Inputs: MAX10_CLK1_50, [1:0] KEY, [7:0] SW

Outputs: [7:0] LEDR, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [15:0] DRAM_DQ, DRAM_LDQM, DRAM_UDQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK

Description: Instantiates all the modules required for our design.

Purpose: Top-level for our 6.1 design.

lab62.sv

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

Outputs: [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

Description: Instantiates all the modules required for our design.

Purpose: Top-level for our 6.2 design.

lab61_soc.qip

Description: Module that holds information about the hardware created in Platform Designer.

Purpose: Used so outside hardware can be interacted with from user.

lab62_soc.qip

Description: Module that holds information about the hardware created in Platform Designer.

Purpose: Used so outside hardware can be interacted with from user.

VGA_controller.sv

Inputs: Clk, Reset,

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

Description: Module that sets the pixel value onto the screen using vertical and horizontal syncs.

Purpose: Used to draw or set pixel values on the 640x480 VGA screen.

System Blocks

Core Blocks

clk_0

Function: Main 50 MHz clock used for the entire design and all components

nios2_gen2_0

Function: NIOS II/e processor block. It sends instructions and data to other blocks.

sdram

Function: Off-chip 512 MB memory utilizing sdram_pll as a clock to read and write operations correctly.

sdram_pll

Function: Clock delayed 1ns from main clock to accurately have sdram work.

sysid_qsys_0

Function: Ensures no mismatch or incompatibilities between hardware and software incorporation.

Week 1 Blocks

onchip_memory2_0

Function: Increases speed of data access for blocks.

sw

Function: 8-bit input. SW is the switches on the FPGA board.

led

Function: 8-bit output. LED is the LEDs on the FPGA board.

accumulate

Function: 2-bit input. Accumulate are the two buttons on the FPGA board (run and reset).

Week 2 Blocks

jtag_uart

Function: Allows software to communicate with FPGA by outputting onto terminal.

keycode

Function: 8-bit input value that holds which keyboard key was pressed.

usb_irq/usb_gpx/usb_rst

Function: 1-bit PIOs that allows keyboard to communicate with FPGA.

hex_digits_pio

Function: 16-bit output. Hex_digits_pio is the Hexdigit displays on the FPGA board.

key

Function: 2-bit input value. Key is the of the 2 buttons on the FPGA board (run and reset).

timer

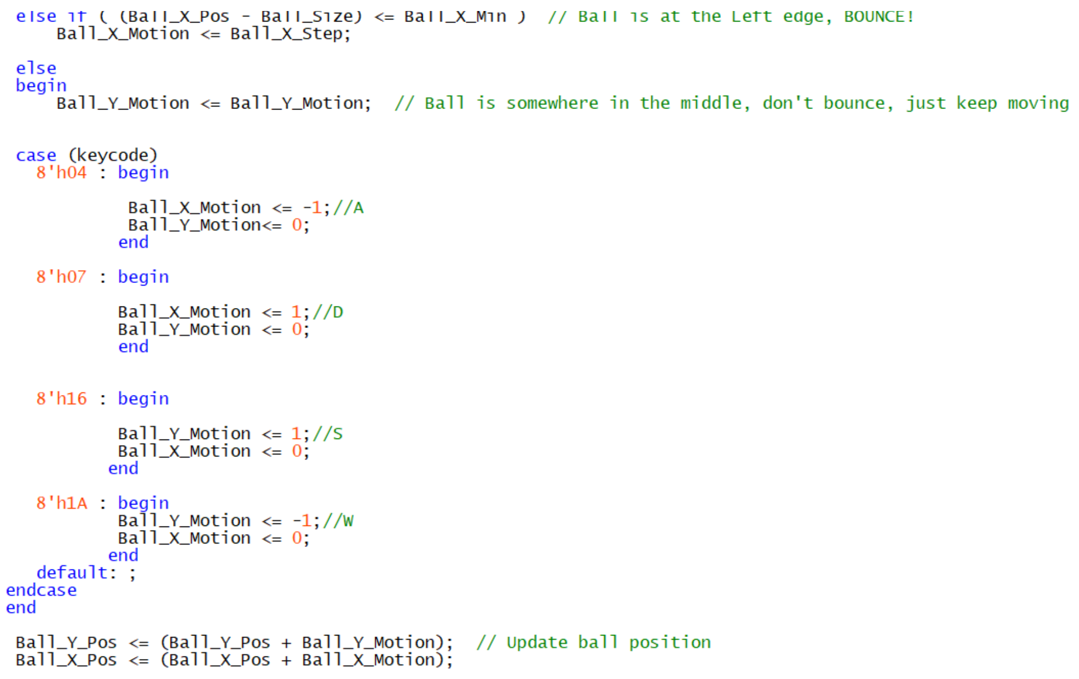
Function: Clock to keep track of how much time has passed.

spi_0

Function: Allows data transfer between master and slave components.

VI. Extra Credit

In the given code, there would be a glitch in which if the ball reaches a boundary while the user is pressing the key which the ball is traveling (ball moving up and user holding up when it reaches the border), the ball would glitch through the border and disappear, reappearing in a spot after some time has passed. This is due to incorrect placement of a specific chunk of code in the ball.sv module. The keycode case should be placed with the else statement where the ball is in the middle and not at the border. Putting the keycode cases outside of the statement will cause the keycode statement to always be checked, therefore updating the motion when it shouldn't need to be updated when the border is reached. Having it inside will only cause the keycode to be read if the ball is moving inside the borders.



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

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


case (keycode)
  8'h04 : begin

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

  8'h07 : begin

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


  8'h16 : begin

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

  8'h1A : begin
          Ball_Y_Motion <= -1;//W
          Ball_X_Motion <= 0;
         end
  default: ;
endcase
end

Ball_Y_Pos <= (Ball_Y_Pos + Ball_Y_Motion);  // Update ball position
Ball_X_Pos <= (Ball_X_Pos + Ball_X_Motion);
```

Change in ball.sv

VII. Conclusion

Overall, our design worked functionally well for both checkpoints and we were able to implement them successfully. The document for setting up the appropriate PIO modules for both checkpoints was detailed and easy to follow. No issues regarding confusion on a specific step or how to do something arose. Any problems that we did encounter was easily found in the FAQs on the main wiki or resolved through a post on Campuswire.

For the first checkpoint, no main bugs were encountered. However, we had a lot of issues during the second checkpoint. Our first main bug came during when we were trying to run our C code. Though it compiled fine, it gave errors. Some of these errors were fixed when we checked two boxes (solution from Campuswire), but the code gave a different error. Through the FAQs, we determined that the pin assignments assigned to the incorrect module and after recompiling, it gave no errors. Now, it gave a different error where it wasn’t initiating the drivers. We found out this bug was due to errors in our C code where we used the wrong variable names when declaring the BYTE variables to be read and written. This took a long time to figure out because we never assumed that the variable name would affect the commands (both were unsigned 8-bit integers). After that, we had a reset interrupt loop. Through a TA, we found out that the FPGA was just malfunctioning and rubbing it at a specific spot would fix the error. Finally, our keyboard worked and could read the keycodes and our VGA displayed correctly, but our ball would not move. This took us a long time to debug as well, but it turns out we set the address of the keycode incorrectly. We never guessed to check this since the keycodes could be read correctly, so we assumed it was getting the right information from the right address. After fixing that, our code ran perfectly and was able to operate correctly.

Post-Lab Questions

	SOC
LUT	3921
DSP	10
Memory (BRAM)	46080
Flip-Flip	2500
Frequency (MHz)	71.9
Static Power (mW)	96.52
Dynamic Power (mW)	61.67
Total Power (mW)	179.32

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

The NIOS II/e processor has less logical elements and resources, but lower performance. The NIOS II/f on the other hand has higher logical elements and a higher performance.

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

On-chip memory provides faster memory and data access as it will require less time to access the data in memory as compared to off-chip memory.

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

The NIOS II is a modified Harvard because instruction and data have their own distinct buses, but memory is shared. A Von Neumann has data and instruction share both bus and memory while a pure Harvard has data and instruction each have separate buses and memory. Modified Harvard has same buses, but different memory. Therefore, the NIOS II is a modified Harvard machine.

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 only needs the data bus because it is an output. Its only function is to display the data, so only data is needed. On-chip memory needs access to both because it is writing and fetching from memory, requiring the data and program bus.

Why does SDRAM require constant refreshing?

The SDRAM operates with capacitors and transistors. Due to the behavior of capacitors, the charge across them will decrease over time. If you want to keep the information accurate, then the SDRAM will need to be refreshed to keep the capacitor charged.

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.

32*(2^13)*(2^10)*1*4 = 64 Mbytes

SDRAM Parameter	Short name	Parameter value
Data Width	[width]	32
# of Rows	[nrows]	13
# of Columns	[ncols]	10
# of Chip Select	[ncs]	1
# of Banks	[nbanks]	4

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

With data width being 32 bits and access time at 5.4 ns, convert 32 bits to bytes by dividing by 8 to get 4 bytes. Now divide that by 5.5 ns to get 740 MB/s.

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

If the SDRAM is run too slow, then the information will be incorrect. This is because if the refresh rate is too slow, the capacitor will have a longer time to discharge potentially taking it outside of the valid range.

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?

The second clock is needed because the controller needs time for the address data and control signals to be stable at the SDRAM. The delay will cause the design to fall within the correct window when the signals are stable.

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

The NIOS II will start execution from the SDRAM base address. In our case, it is x08000000. This step is done after assigning the addresses so that the processor knows where to return if a reset or exception were to occur. It prevents any overlaps of memory.

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 marks the variable as a variable that can change value throughout the program's execution. This prevents the program from optimizing for that variable and the program will always check if the variable has changed. Set works by setting LED_PIO to a high value (turning it on) and clear works by setting LED_PIO to a low value (turning it off).

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:

.bss - Uninitialized region

  Example: int x;

.heap - The heap portion of memory (allocated memory)

  Example: int x = (int)malloc(sizeof(int));

.rodata - Region containing the read only variables or static variables

  Example: const int x = 0;

.rwdata - Region containing the read and write variables

  Example: int x = 0;

.stack - The stack portion of memory (function-called variables)

  Example: int func(int a, int b);

.text - Region containing strings

  Example: char s = “string”;