ECE385

Fall 2021

Experiment #6

SOC with NIOS II in SystemVerilog

Haozhe Si

haozhes3

Section ABD

1. Introduction

a) Basic Functionality of the NIOS-II Processor

The NIOS II is an IP based 32-bit CPU which can programmed using a high-level language like C. By making use of the NIOS II processor on the MAX10 FPGA, we can have the NIOS II be the system controller and handle tasks which do not need to be high performance while an accelerator peripheral in the FPGA logic handles the high-performance operations.

b) Operation of the USB/VGA Interface

In this lab, we implemented the USB Interface and the VAG interface. The USB interface will take in a key stroke from the keyboard and send it the MAX10 board. The keycode is then send to the NIOS-II Processor for further use. The VGA Interface will determine the color given a pixel and output the RGB information to the monitor. The VGA Interface will scan through all the 800x525 pixels (including the blank pixels) for the 640x480 monitor in series to print one frame. By constantly scanning though the pixels with 25MHz frequency, the monitor can be refreshed at a rate of 60Hz.

2. NIOS-II System Description

a) Hardware Components

The core component in this lab is the NIOS-II/e Processor, which is a resource-optimized 32-bit RISC. In addition to the processor, we have the on-chip memory module, and Key, Switch, LED, and hex digit display PIO module. For the second part of the lab, we further added the USB_IRQ, USB_RST, USB_GPX and keycode PIO modules, along with a timer module and a SPI module to support the USB interface and keyboard input. We also have a JTAG_UART module to get and print out the software outputs. Finally, we have the clock source module, the SDRAM controller module and the system ID Peripheral.

b) I/O for Lab 6.1

The inputs for Lab 6.1 including the pushing buttons and the switches on the MAX10 boards. We will input the adder we want to add through the switches and press one of the push buttons. The software will capture the signal and load adder from the switches and perform the accumulation. The calculate value will be output to the hex display through the hex digit display PIO. If the software captures the clear signal input by another pressing key, it will clear the accumulated value and set the hex digit display to zeros.

c) NIOS-II interaction with USB Chip and VGA Components

We should notice that only the USB Chip interacts with the NOIS-II system, while the VGA components are handled by the FPGA board. For the USB

Chip, it communicates with the NIOS-II System mainly through the SPI ports we created. To be specific, to read data from the USB Chip, NIOS-II system will write the address of the register it wants to read through the MOSI signal and read from the MISO signal. Similarly, for USB write, we first need to write the address of the target register through the MOSI signal, and then write the data through the MISO signal. The communication with the USB Chip through the SPI port is accomplished by calling the SPI API provided by Intel, which provides the functions for reading/writing specific number of bytes from/to specific addresses. In such way we managed the communication between the NIOS-II and the USB Chip.

d) SPI protocol

SPI is a synchronous serial bus, consisting of 4 signals called CLK, MOSI, MISO, and SS. CLK is the clock signal, which is transmitted from the master device (NIOS-II) to the slave devices (MAX3421E) and runs at a frequency of 25MHz. The MOSI signal stands for master-out-slave-in signal, which transmits data from the master device to the slave device synchronously with the CLK. The MISO signal stands for the master-in-slave-out signal, which transmits data from the slave device to the master device also synchronously with the CLK. SS stands for slave select, which allows a specific slave device to be selected when SS is low. As discussed above, the SPI protocol supports an 8-bit read/write operation between the master and slave devices. The address of the target registers for read/write will be written to the slave device first through the MOSI signal bus, which takes 1 byte. Then, we can read multiple bytes from the slave device through the MISO signal bus or write multiple bytes to the slave device through the MOSI signal bus. Reading/Writing one byte will take 8 clock cycles of the CLK signal.

e) C Functions

Method: void Adder()
Purpose: This function will add the value input from the switches to the accumulator each time the push button is pressed and output the result to the LED in binary form.

Method: void MAXreg_wr(BYTE reg, BYTE val)
Purpose: This function will write the one-byte *val* into the register with address *reg+2*.

Method: BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)
Purpose: This function will write *nbytes* data array *data* into the registers start with address *reg+2*.

Method: BYTE MAXreg_rd(BYTE reg)
Purpose: This function will read the one-byte data from the register with

address *reg*.

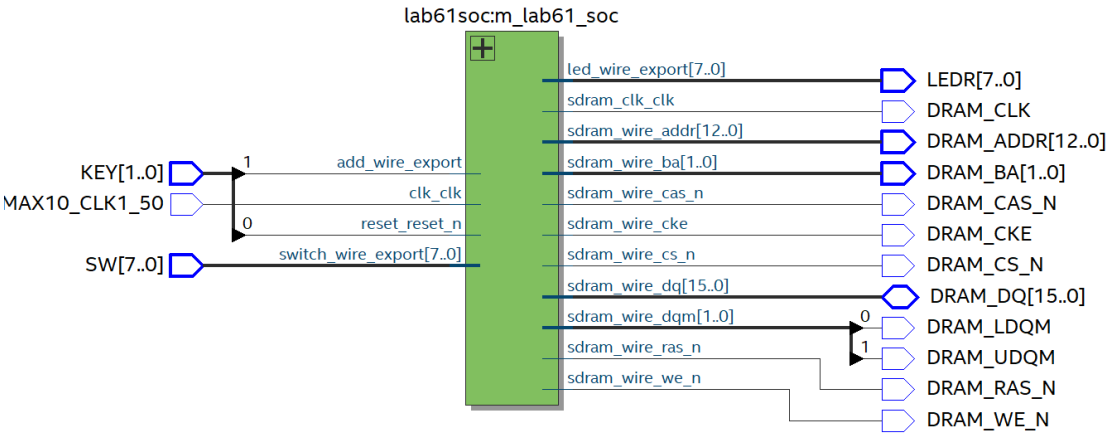
Method: BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)
Purpose: This function will read *nbytes* data from registers start with address *reg*.

f) VGA Operation

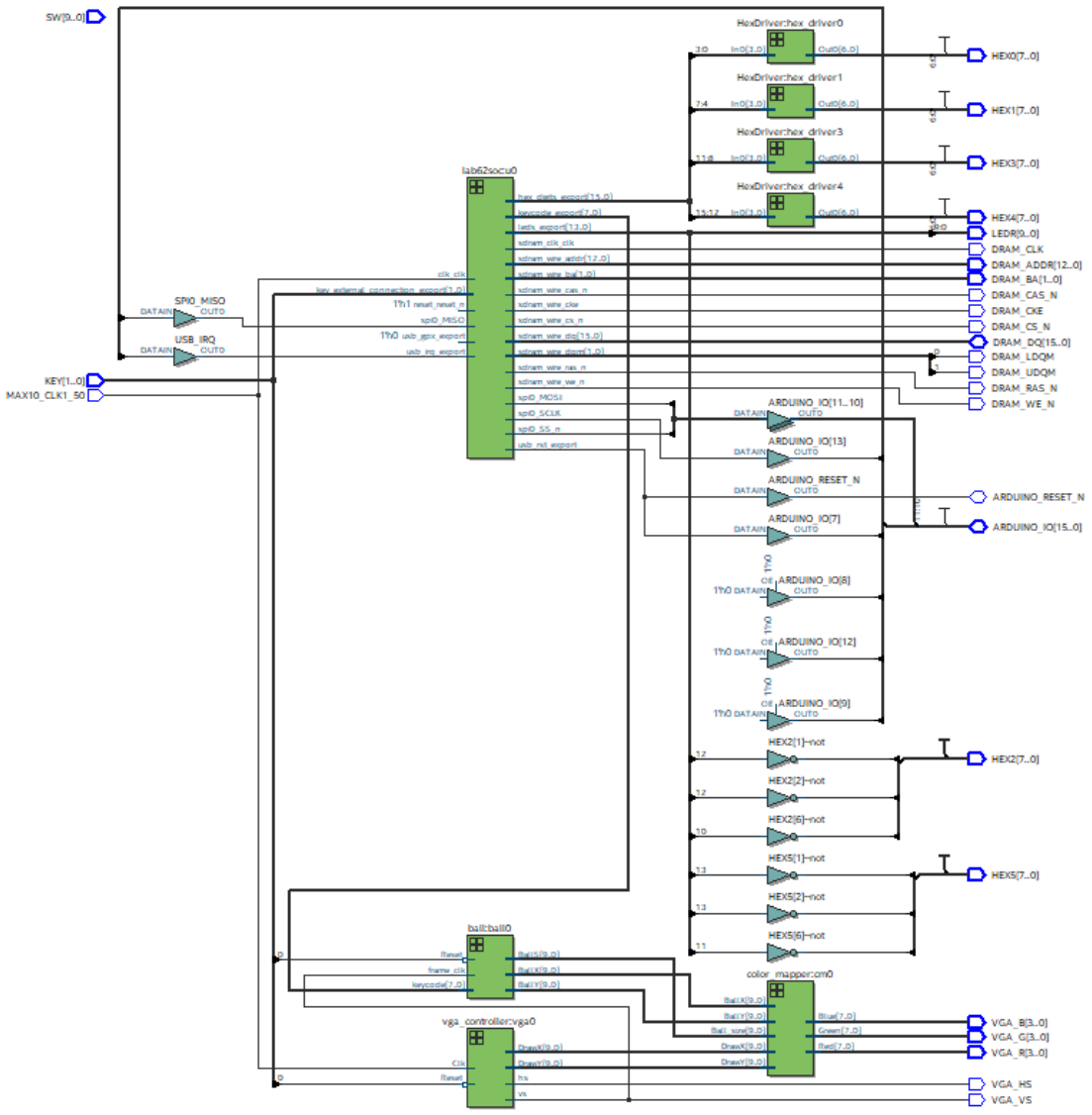
The VGA for a 640x480 pixels monitor will scan a region with size 800x525 pixels in row-major order. The VGA controller module will take charge in the scanning: it will scan each pixel at a 25MHz frequency. Every time the it finishing scanning a row, it will output a horizontal sync signal (*hs*), and every time it finishing all rows, it will output a vertical sync signal (*vs*). Since the actual scanning area is larger than the screen area, when the controller scans the area outside the monitor, it will set the *blank* signal. The controller will also output the current coordination of the pixel it is scanning. The Ball module takes in the keycode we get from the NIOS-II system and determine the motion of the ball. It is synchronous with the frame clock, which is basically the *vs* signal. It will output the coordination of the Ball at the current frame and the Ball Size. The Color Mapper module will take both the pixel location and the ball location, as well as the ball size, and determine the color of the current pixel base on the information above. The output will be the RGB value of the pixel at the input location.

3. Block Diagram

a) Lab 6.1



b) Lab 6.2



4. Module Description

a) Lab 6.1

Module: Lab61.sv
Inputs: [1:0] KEY, [7:0] SW, MAX10_CLK1_50
Outputs: [7:0] LEDR, [12:0] DRAM_ADDR, DRAM_CKE, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CS_N, DRAM_LDQM, DRAM_UDQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
I**nout:** [15:0] DRAM_DQ
Description: This module instantiated the module for NIOS-II system.
Purpose: This is the top-level module for interfacing with the NIOS-II system module. It connects the IOs and DRAM on the MAX10 board to the NIOS-II system module.

Module: Lab61soc.sv
Components: clk_0, nios2_gen2_0, onchip_memory2_0, led, sdram, sdram_pll, sysid_qsys_0, sw, add
Description: This module instantiated the components used in NIOS-II system. Among all the components, the PIO blocks are the *led* block, which is used for displaying the results of operation; the *sw* block, which is the input to the SOC, and the *add* block, which is one push button that used to signify the software to perform addition.
Purpose: This is the System Verilog code for the SOC we defined in the QsYs. The components are connected according to the code we have here.

b) Lab 6.2

Module: Lab62.sv
Inputs: [1:0] KEY, [7:0] SW, MAX10_CLK1_50
Outputs: [7:0] LEDR,
[7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B,
[12:0] DRAM_ADDR, DRAM_CKE, [1:0] DRAM_BA, DRAM_CLK, DRAM_CAS_N, DRAM_CS_N, DRAM_LDQM, DRAM_UDQM, DRAM_RAS_N, DRAM_WE_N
I**nout:** [15:0] DRAM_DQ, [15:0] ARDUINO_IO, ARDUINO_RESET_N
Description: This module instantiated the module for NIOS-II system.
Purpose: This is the top-level module for interfacing with the NIOS-II system module. It connects the IOs and DRAM on the MAX10 board to the NIOS-II system module.

Module: Lab62soc.sv
Components: clk_0, nios2_gen2_0, onchip_memory2_0, leds_pio, sdram, sdram_pll, sysid_qsys_0, keycode, hex_digits_pio, spi_0, jtag_uart_0, timer_0, usb_irq, usb_gpx, usb_rst, key

Description: This module instantiated the components used in NIOS-II system. Among all the components, the PIO blocks are the *leds_pio* and the *hex_digits_pio* block, which are used for displaying the results of operation; the *keycode* block, which outputs the keycode of the key stroke to the FPGA, the *key* block, which reads the push button signal to the NIOS-II system, and the *usb_irq, usb_gpx, usb_rst* signals, which communicate with the USB chip.
Purpose: This is the System Verilog code for the SOC we defined in the QsYs. The components are connected according to the code we have here

Module: ball
Inputs: Reset, frame_clk, [7:0] keycode
Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS
Description: This module decides the location of the ball by first deciding its motion. The motion is primely controlled by the *keycode*. If the key is not pressed, it will move in the direction of the last key stroke until it hits the edge of the screen. Once hits the edge, the direction of motion will be inversed. The module will run synchronously with the *frame_clk*. *Reset* signal will reset the ball location to the center of the screen. This module returns the location of the ball and the size of it.
Purpose: The purpose of this module is to decide the location of the ball at the current frame. This information will be later used to determine the color of the pixels.

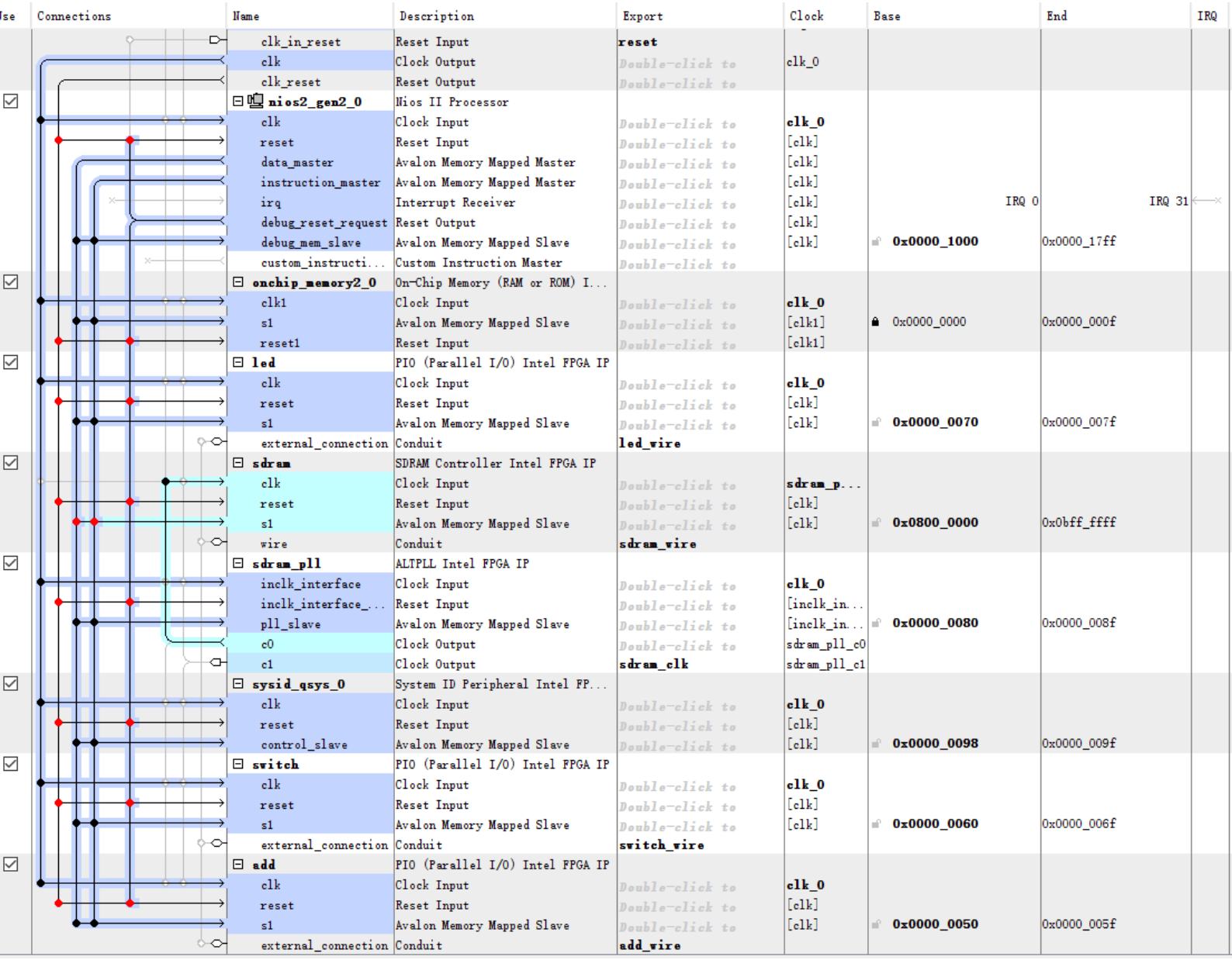
Module: VGA_controller
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, sync, blank, [9:0] DrawX, [9:0] DrawY
Description: This module will first divide the *Clk* speed in half and create the *pixel_clk*. Then module is then synchronized with the *pixel_clk*. The VGA_controller module will scan the pixel array of the monitor. Every time the it finishing scanning the length of a row, it will set the *hs* signal, and every time it finishing all pixels, it will set the *vs* signal. Since the actual scanning area is larger than the screen area, when the controller scans the area outside the monitor, it will set the *blank* signal. The module will also output the coordination of the pixels it is scanning at the current clock cycle.
Purpose: The purpose of this module is to decide the location of the pixel it is scanning. It also outputs the synchronization signals of the VGA hardware. This information will be later used to determine the color of the pixels.

Module: Colop_Mapper
Inputs: [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] DrawY, [9:0] Ball_size,
Outputs: [7:0] Red, [7:0] Green, [7:0] Blue
Description: This module will first decide whether the current pixel location is ball or background. Then it will output the color of the current pixel location.
Purpose: The module is used to set color to the screen given the ball size and

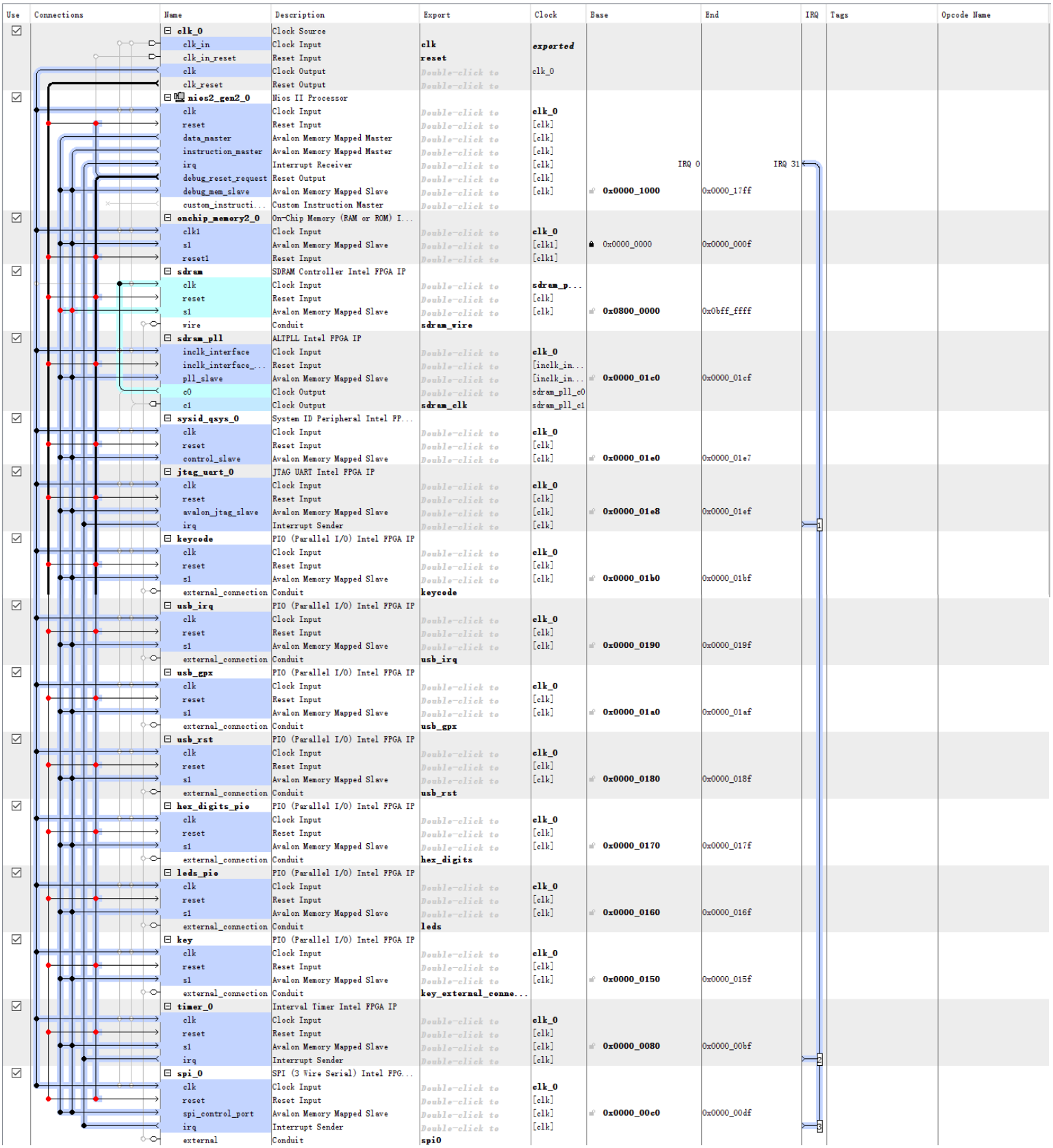
location.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: The module will output the code for seven-segment display depending on the input 4-bit data.
Purpose: This module is used to translate the binary data into the code for hex display on the board.

5. System Level Block Diagram



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
		clk_in_reset	Reset Input	**reset**				
		clk	Clock Output	Double-click to	clk_0			
		clk_reset	Reset Output	Double-click to				
☑		**nios2_gen2_0**	Nios II Processor					
		clk	Clock Input	Double-click to	**clk_0**			
		reset	Reset Input	Double-click to	[clk]			
		data_master	Avalon Memory Mapped Master	Double-click to	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double-click to	[clk]			
		irq	Interrupt Receiver	Double-click to	[clk]	IRQ 0		IRQ 31
		debug_reset_request	Reset Output	Double-click to	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_1000**	0x0000_17ff	
		custom_instructi...	Custom Instruction Master	Double-click to				
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) I...					
		clk1	Clock Input	Double-click to	**clk_0**			
		s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0000_0000	0x0000_000f	
		reset1	Reset Input	Double-click to	[clk1]			
☑		**led**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to	**clk_0**			
		reset	Reset Input	Double-click to	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0070**	0x0000_007f	
		external_connection	Conduit	**led_wire**				
☑		**sdram**	SDRAM Controller Intel FPGA IP					
		clk	Clock Input	Double-click to	**sdram_p...**			
		reset	Reset Input	Double-click to	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0800_0000**	0x0bff_ffff	
		wire	Conduit	**sdram_wire**				
☑		**sdram_pll**	ALTPLL Intel FPGA IP					
		inclk_interface	Clock Input	Double-click to	**clk_0**			
		inclk_interface_...	Reset Input	Double-click to	[inclk_in...			
		pll_slave	Avalon Memory Mapped Slave	Double-click to	[inclk_in...	**0x0000_0080**	0x0000_008f	
		c0	Clock Output	Double-click to	sdram_pll_c0			
		c1	Clock Output	**sdram_clk**	sdram_pll_c1			
☑		**sysid_qsys_0**	System ID Peripheral Intel FP...					
		clk	Clock Input	Double-click to	**clk_0**			
		reset	Reset Input	Double-click to	[clk]			
		control_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0098**	0x0000_009f	
☑		**switch**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to	**clk_0**			
		reset	Reset Input	Double-click to	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0060**	0x0000_006f	
		external_connection	Conduit	**switch_wire**				
☑		**add**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to	**clk_0**			
		reset	Reset Input	Double-click to	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0050**	0x0000_005f	
		external_connection	Conduit	**add_wire**				



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*	clk_0					
		clk_reset	Reset Output	*Double-click to*						
☑		**nios2_gen2_0**	Nios II Processor							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		data_master	Avalon Memory Mapped Master	*Double-click to*	[clk]					
		instruction_master	Avalon Memory Mapped Master	*Double-click to*	[clk]					
		irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 31			
		debug_reset_request	Reset Output	*Double-click to*	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_1000**	0x0000_17ff			
		custom_instructi...	Custom Instruction Master	*Double-click to*						
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) I...							
		clk1	Clock Input	*Double-click to*	**clk_0**					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk1]	0x0000_0000	0x0000_000f			
		reset1	Reset Input	*Double-click to*	[clk1]					
☑		**sdram**	SDRAM Controller Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**sdram_p...**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0800_0000**	0x0bff_ffff			
		wire	Conduit	**sdram_wire**						
☑		**sdram_pll**	ALTPLL Intel FPGA IP							
		inclk_interface	Clock Input	*Double-click to*	**clk_0**					
		inclk_interface_...	Reset Input	*Double-click to*	[inclk_in...					
		pll_slave	Avalon Memory Mapped Slave	*Double-click to*	[inclk_in...	**0x0000_01c0**	0x0000_01cf			
		c0	Clock Output	*Double-click to*	sdram_pll_c0					
		c1	Clock Output	**sdram_clk**	sdram_pll_c1					
☑		**sysid_qsys_0**	System ID Peripheral Intel FP...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		control_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_01e0**	0x0000_01e7			
☑		**jtag_uart_0**	JTAG UART Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_01e8**	0x0000_01ef			
		irq	Interrupt Sender	*Double-click to*	[clk]			1		
☑		**keycode**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_01b0**	0x0000_01bf			
		external_connection	Conduit	**keycode**						
☑		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_0190**	0x0000_019f			
		external_connection	Conduit	**usb_irq**						
☑		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_01a0**	0x0000_01af			
		external_connection	Conduit	**usb_gpx**						
☑		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_0180**	0x0000_018f			
		external_connection	Conduit	**usb_rst**						
☑		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_0170**	0x0000_017f			
		external_connection	Conduit	**hex_digits**						
☑		**leds_pio**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_0160**	0x0000_016f			
		external_connection	Conduit	**leds**						
☑		**key**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_0150**	0x0000_015f			
		external_connection	Conduit	**key_external_conne...**						
☑		**timer_0**	Interval Timer Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_0080**	0x0000_00bf			
		irq	Interrupt Sender	*Double-click to*	[clk]			2		
☑		**spi_0**	SPI (3 Wire Serial) Intel FPG...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		spi_control_port	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_00c0**	0x0000_00df			
		irq	Interrupt Sender	*Double-click to*	[clk]			3		
		external	Conduit	**spi0**						

Core Components:
Module: clk_0
Exports: clk, reset
Description: This module is the clock source for the NIOS-II system.

Module: nios2_gen2_0
Exports:
Description: The economy version processor of the NIOS-II system. It handles all the logical operations and computations the programs might use.

Module: onchip_memory2_0
Exports:
Description: The on-chip memory of the NIOS-II system. It is "expensive" to use since it requires more logical elements than DRAM, but it can increase the operation speed of memory I/O. Therefore, we use it as the cache or data buffer.

Module: sdram
Exports: sdram_wire
Description: The SDRAM controller of the NIOS-II system. It specifies the size the of the SDRAM and the SDRAM IP module will be the driver for memory, performing operations like refreshing every time we do memory operations.

Module: sdram_pll
Exports: sdram_clk
Description: Pll means "Phase Lock Loop". This module will generate a second clock signal with a phase shift. This block is necessary to prevent skew operation.

Module: sysid_qsys_0
Exports:
Description: The system ID checker is used to ensure the compatibility between the hardware and software we defined. This module will provide a serial number, and the software loader will check this number at the beginning of the program. This prevents us from loading software with incompatible NIOS-II configuration to FPGA.

Components for Lab6.1:
Module: led
Exports: led_wire
Description: This is a PIO block that handles the led output. The exported led_wire will be connected to the physical led pins on the FPGA board so that we can control the led through software using NIOS-II system.

Module: switch
Exports: switch_wire
Description: This is a PIO block that handles the switch input. The exported

switch_wire will be connected to the physical switch pins on the FPGA board so that we can read the switch inputs through software using NIOS-II system.

Module: key
Exports: key_wire
Description: This is a PIO block that handles the push button input. The exported key_wire will be connected to the physical push button pins on the FPGA board so that we can read the push button inputs through software using NIOS-II system.

Additional Components for Lab6.2:
Module: keycode
Exports: keycode
Description: This is a PIO block that reads the keycode of the key stroke from the keyboard. The exported keycode will be connected to the physical hex digit display pins on the FPGA board so that we can display the presses keycode on the hex digit displays. This signal will also be used to determine the moving direction of the ball.

Module: jtag_uart_0
IRQ: IRQ1
Description: This JTAG_UART block is used for debug. It has an interrupt connection with the processor. With this block, we can use “printf” in Ellipse.

Module: timer_0
IRQ: IRQ2
Description: This is a timer block that counts on millisecond intervals. It has an interrupt connection with the processor. The block is necessary in the USB driver code in order to keep track of the various time-outs that USB requires.

Module: spi_0
IRQ: IRQ3
Description: This is a SPI block that communicate with the USB controller with four buses: SS, MOSI, MISO, SCLK. The functionality of the SPI block is discussed above. The four signals of SPI will connect to the corresponding ports of the MAX3421E USB Peripheral/Host Controller. It will also have an interruption with the processor.

Module: usb_irq
Export: usb_irq
Description: This is a PIO block which connects to the interrupt output of the MAX3421E USB Peripheral/Host Controller as required by datasheet.

Module: usb_rst
Export: usb_irst
Description: This is a PIO block which connects to the device reset of the

MAX3421E USB Peripheral/Host Controller as required by datasheet.

Module: usb_gpx
Export: usb_gpx
Description: This is a PIO block which connects to the General-Purpose Multiplexed Push-Pull output of the MAX3421E USB Peripheral/Host Controller as required by datasheet.

6. Software Components

Method: void Blinker()
Description: The function has a PIO signal for the LED signals as the output, this signal will be cleared to zero initially. The main functions are inside an infinite loop, while inside the loop, it will first wait on a delay. After the delay, the function will OR the LED signal with 0x1, which will light up the last LED on the board. Then the function will hang on another delay. After that delay, the function will AND 0x3FE with the LED signal to turn off the LED. The LED will blink by repeating the light up and turn off operation inside the infinite loop.

Method: void Adder()
Description: This function will read the *add* signal from the push button and the switches signals from the PIO. The function also has a PIO signal for the LED, this signal will be cleared to zero initially. The main functions are inside an infinite loop, while inside the loop, it will wait on the *add* signal. After the *add* is set, the function will take in the number from the switches and add it to the output signal. Then the LED will be set to the binary result of the accumulation. The function will hang at a waiting state if the push button for *add* is not released.

Method: void MAXreg_wr(BYTE reg, BYTE val)
Description: This function will write the one-byte *val* into the register with address *reg+2*. This is done by first write *reg+2* then write *val.*

Method: BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)
Description: This function will write *nbytes* data array *data* into the registers start with address *reg+2*. This is done by first write *reg+2* then write array of *data.* The function will return the pointer to the memory position after last written.

Method: BYTE MAXreg_rd(BYTE reg)
Description: This function will read the one-byte data from the register with address *reg*. This is done by first write *reg* then read the data and save it to a local variable *val.* The function will return the one-byte *val.*

Method: BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)
Description: This function will read *nbytes* data from registers start with address *reg*.

This is done by first write *reg* then read *nbytes* data and save it to the array *data.* The function will return the pointer to the memory position after last written.

7. Post-Lab Questions

a) *What are the differences between the Nios II/e and Nios II/f CPUs?*
Nios II/e is the resource-optimized version of the Nios II processor while the Nios II/f is the performance-optimized version of the Nios II processor. Therefore, the Nios II/f processor supports more features than the Nios II/e processor, which the Nios II/e processor saves more power.

b) *What advantage might on-chip memory have for program execution?*
Operations on on-chip memory can be significantly faster than on off-chip memories. This is because on-chip memory has a faster read/write speed than the off-chip memories like SRAM/SDRAM chips.

c) *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 Machine. This is because in Modified Harvard Machine, data and instructions can be saved in the same place, but need two separate paths. By observing the design of our processor, we can see that the data path and instruction path are separated, meanwhile they go into the same port of the on-chip memory. Therefore, it is Modified Harvard Machine.

d) *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 on-chip memory needs to store the program instructions as well as the data for program operations, therefore, it needs to connect both the data and the instruction path. On the other hand, the LED, as a PIO, only takes the data from data path to display, therefore, only the data path is connected.

e) *Why does SDRAM require constant refreshing?*
The DRAM uses a single driving transistor and capacitor to contain a bit, which is represented as the charge on the capacitor. Given the property of the capacitor, it can discharge, which will lead to a loss of data eventually. Therefore, in order to preserve the data, we need to constantly make the driving transistor charge the capacitor to the same level, in another word, refreshing it.

f) *Justify how to set the SDRAM controller to came up with a total memory of 512 Mbit.*

There is one 32M x 16 Chip, and the total amount of memory gives by:

8K rows * 1K columns * 16bits * 4 banks = 512Mbits

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

Given the access time is 5.4ns, and the maximum transfer rate in bit/sec is given by:

#Access/Sec * Data Width

= 1 sec / 5.4 ns/access * 16 bits

≈2.759Gbit/sec

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

50MHz is the frequency of the over system clock and the NIOS II processor runs at that speed as well. If the SRDRAM is slower than 50MHz, it cannot provide new data for the system to use on time.

i) *Why do we need generate a second clock by PLL?*

We need a second clock because the SDRAM requires precise timings, and the PLL can compensate for the clock skew causing by the board layout. To be specific, by generating a clock to the SDRAM chip 1 ns slower than the controller clock, we can ensure the data transfer since at the rising edge of the SDRAM clock, the control signals from the SDRAM controller have already been stabilized during the 1ns delay.

j) *What address does the NIOS II start execution from? Why do we assign the reset and exception vectors for the Nios II processor after assigning the addresses?*

It starts execution from the base address of the *sdram*, which is 0x0800 0000. We need to assign reset and exception vectors after assigning the base address because we need to tell the rest and exception vectors where they should go for proper system executions.

k) *For the Blinker program, 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 keywords means that the variable may change during the execution unexpectedly. Here, the keyword is used to manipulate peripheral registers, so that the pattern of the LED can change during the program.

The address of the *led_pio* on our Avalon bus is 0x70. Therefore, we will first assign a pointer pointing that address, and dereferencing the address will give

us the value of the LEDs at that point of time. We can also modify the value of the LEDs in the same way. By dereferencing the address, we can set the value to zero to clear the LED output or performing logical operations with the value to light up specific LEDs. The details for achieving blinking has been discussed above.

l) *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.*

Segment: .bss
Meaning: Uninitialized static data
Example: int val;

Segment: .heap
Meaning: Dynamically allocated memory
Example: int *val = malloc(sizeof(int));

Segment: .rodata
Meaning: Read-only initialized static variables
Example: constant int val = 1;

Segment: .rwdata
Meaning: Initialized static variables
Example: int val = 1;

Segment: .stack
Meaning: Call stack
Example:
```
int retVal(int valA) {
        ValB = ValA;
        return ValB;
}; // will put the call stack to the .stack segment
```

Segment: .text
Meaning: Executable code and is generally read-only and fixed size
Example:
```
int retVal(int valA) {
        ValB = ValA;
        return ValB;
}; // will put the code to the .text segment
```

8. Design Analysis

1) Lab 6.1

LUT	2269
DSP	0
Memory (BRAM)	10368
Flip-Flop	478
Frequency	83.7MHz
Static Power	96.44mW
Dynamic Power	47.87mW
Total Power	161.70mW

2) Lab 6.2

LUT	3811
DSP	0
Memory (BRAM)	12544
Flip-Flop	752
Frequency	73.02MHz
Static Power	96.52mW
Dynamic Power	61.51mW
Total Power	179.69mW

9. Conclusion

a) Functionality

The first part of the lab implements the functionality of enabling a blinking led and performing accumulation using the switch input. The second part of the lab implements a bouncing ball that can be controlled by the keyboard and the bouncing ball will be shown on the screen via VGA ports. Both parts are achieved by using NIOS II processor and they functioning well.

b) Comment on Lab Manual

Everything is good.