ECE 385

Fall 2021

Experiment #6

SOC with NIOS II in System Verilog

Michael Grawe, Matthew Fang
ABE
Abigail Wezelis

Index

Introduction

The purpose of Lab 6 was to learn about the operations of the NIOS II processor and use this knowledge to interact with memory mapped IO's and on-board switches/buttons, as well as with USB and VGA peripherals. The NIOS II processor is a 32-bit processor with a much higher capacity than the previously used SLC-3 processor. Its instruction set consists of 32 instructions, including a single instruction multiply and divide. In Lab 6.1, a C-program is used to communicate with I/O's on the DE-10 board. This is done through the NIOS II processor. Two different programs are used in Lab 6.1: One program causes an on-board LED to blink on and off and the other accumulates a switch input to be represented in binary on the LEDs. In Lab 6.2, the C-program is used to program a ball on a VGA monitor that takes input to move up, down, left, and right from the WASD keys on a USB keyboard. These peripherals are connected to the C-program through the NIOS II, meaning that the peripherals are extended for this lab.

Written Description and Diagrams of NIOS-II System

Hardware components (platform designer)

Clock source:
Clk_0 is the clock source which clocks all other hardware components at 50 MHz.

NIOS II Processor:
The Nios2_gen2_0 is an embedded processor which runs the C code written.

On-Chip Memory:
The Onchip_memory2_0 is 16 bytes of on-chip RAM.

SDRAM controller:
The SDRAM initializes the memory devices, manages SDRAM banks, and translates read-and-write instructions from the local interface into SDRAM command signals.

ATPLL Intel FPGA IP:
The Sdram_pll creates two 50 MHz clock signals, one of which has a 1ns delay. SDRAM requires precise timings and also time to stabilize, which the two clocks from the SDRAM_pll provide.

System ID Peripheral:
Sysid_qsys_0 assigns a serial number to the hardware and software and compares the two when software is run. This is used to ensure compatibility between the two and prevents loading software onto an FPGA without compatible NIOS II configuration.

JTAG UART:
Jtag_uart_0 implements a method to communicate serial character streams between the terminal of the host computer and the NIOS II chip on the FPGA. Jtag_uart_0 is also

connected to IRQ 1, which means the interrupt is set up at the end of transmission for each buffer, so the CPU is not blocked while waiting for all the data to be transmitted, and is only interrupted whenever the peripheral needs more data.

PIO:
PIOs, or parallel I/Os, are connected to the data bus of the processor. Usb_irq, usb_gpx, and usb_rst are necessary for the connection to the USB chipset, MAX3421E. Hex_digits_pio, Leds_pio, and keycode are used to display information within the data bus. Key is associated with our peripheral keyboard input.

Interval Timer:
The Timer_0 is needed in the USB driver code to keep track of various time-outs that USB requires.

SPI:
Spi_0 is a serial peripheral interface which allows the FPGA to communicate with various peripherals. For the purpose of this lab, only one peripheral device was used, which was connected via the MAX3421E USB peripheral/host controller. The SPI transmits SS, MISO, MOSI, and SCLK signals in order to transmit or receive information from the various peripherals connected to it.

I/O of lab 6.1
Lab 6.1 consisted of an accumulator controlled by two buttons, one reset and one accumulate, and switches. If the run button was pressed, the values on the switches would be read by a C program and then added to the value already in the accumulator. If the reset button was pressed, the value in the accumulator would return to 0. A PIO block to control the LEDs to display the values held within the accumulator.

NIOS interaction with MAX3421E USB chip and VGA components
NIOS interacts with the MAX3421E USB chip and VGA components with four functions: MAXreg_wr, which writes a register to MAX3421E, MAXbytes_wr, which writes multiple bytes, MAXreg_rd, which reads a register from MAX3421E, and MAXbytes_rd, which reads multiple bytes, all via SPI.

The NIOS then relays this information to the top level module on the FPGA and ball module which control use the information to calculate the position and motion of the ball, which is sent to the color_mapper and VGA_controller modules. These modules then control the VGA components via horizontal sync, vertical sync, reg, green, and blue signals.

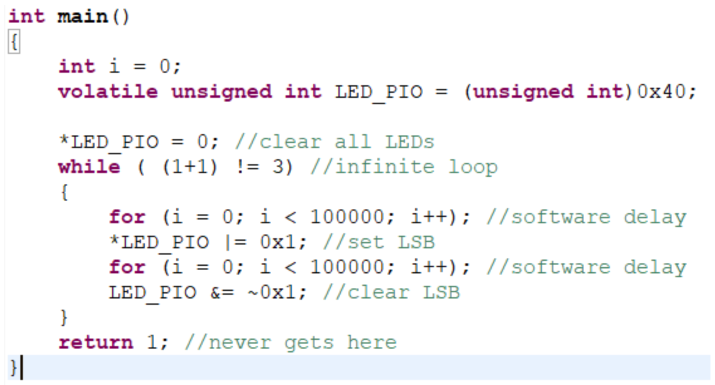
SPI protocol
The Serial port interface controls multiple peripherals with it’s slave select (SS), master in slave out (MISO), master out slave in (MOSI), and SCLK signals. When multiple peripherals are connected, the SS signal chooses which peripheral the FPGA is reading or

writing to. The MISO and MOSI signals determine what is written and read by the FPGA and the chosen peripheral. The MISO signal is what the FPGA reads from the peripheral while the MOSI signal is what the peripheral reads from the FPGA. The SCLK is used to clock the signals and ensure data is read during the window that it is stable.

Code functions

Lab 6.1 specific:

Blink

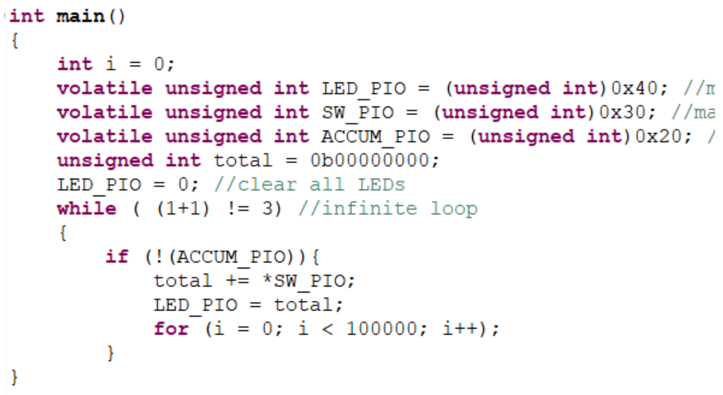


```
int main()
{
    int i = 0;
    volatile unsigned int LED_PIO = (unsigned int)0x40;

    *LED_PIO = 0; //clear all LEDs
    while ( (1+1) != 3) //infinite loop
    {
        for (i = 0; i < 100000; i++); //software delay
        *LED_PIO |= 0x1; //set LSB
        for (i = 0; i < 100000; i++); //software delay
        LED_PIO &= ~0x1; //clear LSB
    }
    return 1; //never gets here
}
```

LED_PIO holds the address where the first LED is stored. By setting *LED_PIO to 0x0, and ~0x1, the LED will turn on or off. This is held in an infinite loop to have the LED continuously blink. The for statements acts as a delay between the times when the LED is turned on and off.

Accumulate

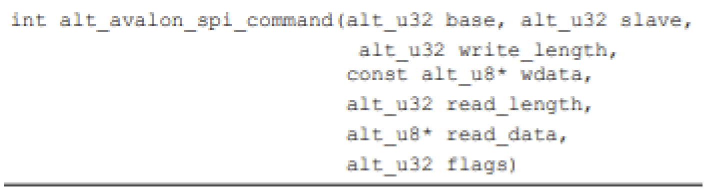


```
int main()
{
    int i = 0;
    volatile unsigned int LED_PIO = (unsigned int)0x40; //m
    volatile unsigned int SW_PIO = (unsigned int)0x30; //ma
    volatile unsigned int ACCUM_PIO = (unsigned int)0x20; /
    unsigned int total = 0b00000000;
    LED_PIO = 0; //clear all LEDs
    while ( (1+1) != 3) //infinite loop
    {
        if (!(ACCUM_PIO)){
            total += *SW_PIO;
            LED_PIO = total;
            for (i = 0; i < 100000; i++);
        }
}
```

This code is also held in an infinite loop which continuously checks whether the accumulate button has been pressed. If the accumulate button is pressed, the values in the switches get added to to total. LED_PIO then takes that values and displays it on the LEDs. Because total is in binary, no additional logic is needed to figure out which LED turns on from a total value that may have been displayed in decimal. The for statement acts as a small delay which accounts for human error when pressing the button and ensure that each press results in one accumulation. The reset is not encoded here and instead created within the hardware.

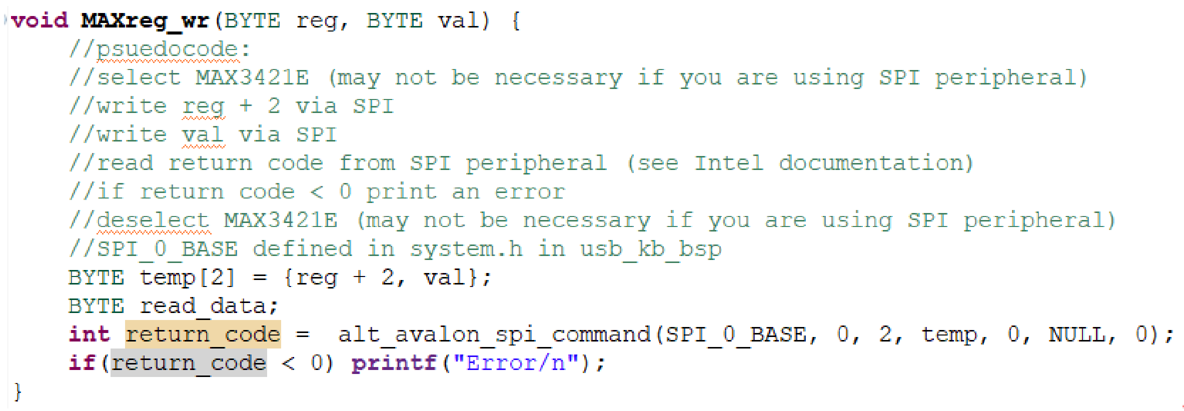
Lab 6.2 specific:

The most essential function used is the alt_avalon_command described below:



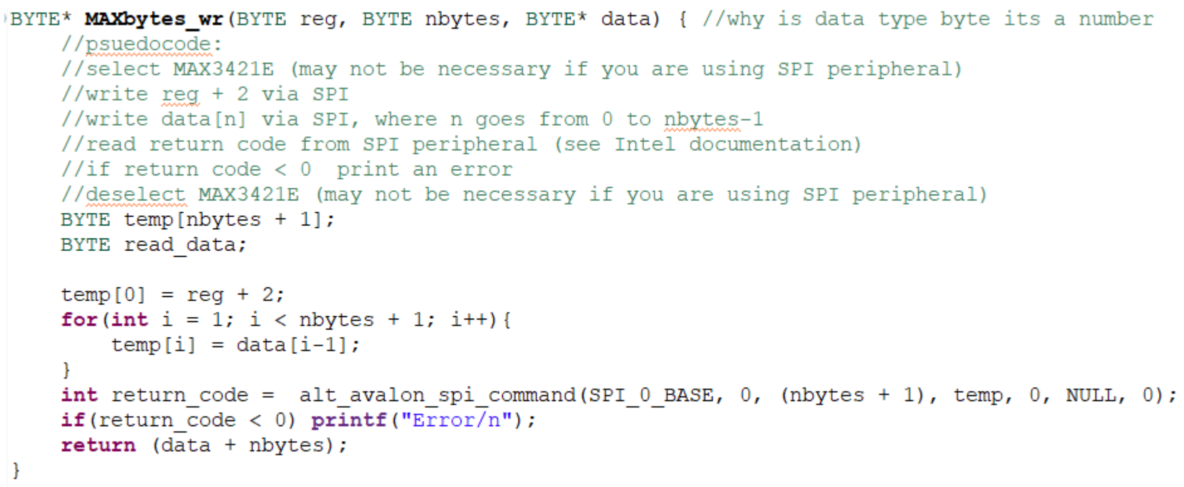
```
int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,
                            alt_u32 write_length,
                           const alt_u8* wdata,
                           alt_u32 read_length,
                           alt_u8* read_data,
                           alt_u32 flags)
```

MAXreg_wr writes a register while MAXbytes_wr writes multiple bytes to MAX3421E. Both write functions are used to enable interrupts, update various bits, check for connections, write reset, start host, and clear and send buffers.



```
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //SPI_0_BASE defined in system.h in usb_kb_bsp
    BYTE temp[2] = {reg + 2, val};
    BYTE read_data;
    int return_code =  alt_avalon_spi_command(SPI_0_BASE, 0, 2, temp, 0, NULL, 0);
    if(return_code < 0) printf("Error/n");
}
```

MAXreg_wr writes a register to MAX3421E. The alt_avalon_command is utilized to write the given value into the given register. The reason the write length is two is because the register and value need to be written. If the write command was valid, the return code would be positive, otherwise, an error would be caught.

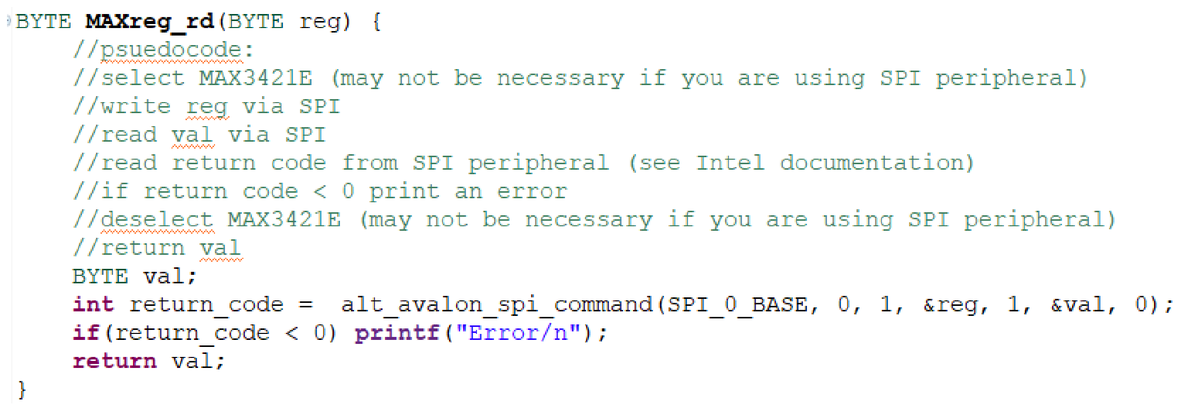


```
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) { //why is data type byte its a number
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write data[n] via SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0  print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    BYTE temp[nbytes + 1];
    BYTE read_data;

    temp[0] = reg + 2;
    for(int i = 1; i < nbytes + 1; i++){
        temp[i] = data[i-1];
    }
    int return_code =  alt_avalon_spi_command(SPI_0_BASE, 0, (nbytes + 1), temp, 0, NULL, 0);
    if(return_code < 0) printf("Error/n");
    return (data + nbytes);
}
```

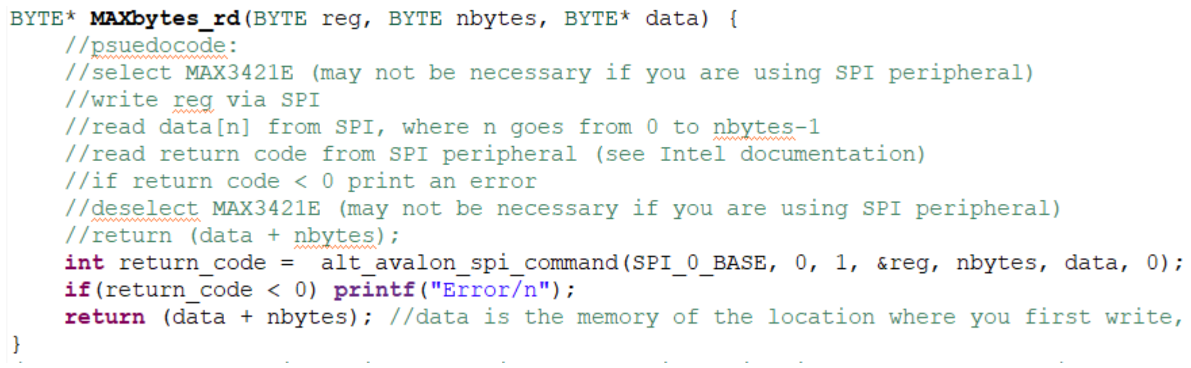
MAXbytes_wr writes multiple bytes to MAX3421E. The difference between this write function and MAXreg_wr is that instead of only writing two values, we write one plus the number of bytes we are writing as well as returning the location last written. Again, this also checks for any errors occurring in the write command.

MAXreg_rd reads a register while MAXbytes_rd reads multiple bytes from MAX3421E. Both read functions are used to read irq registers, wait for the pll to stabilize when resetting, and checking j and k statuses.



```
BYTE MAXreg_rd(BYTE reg) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return val
    BYTE val;
    int return_code =  alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, 1, &val, 0);
    if(return_code < 0) printf("Error/n");
    return val;
}
```

MAXreg_rd reads a register from MAX3421E. The register to be read is passed in the alt_avalon_command, and different from the read write functions, read length is no longer 0 and the read_data no longer is stored in null. Instead, the register is stored in the pointer val.



```
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read data[n] from SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);
    int return_code =  alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, nbytes, data, 0);
    if(return_code < 0) printf("Error/n");
    return (data + nbytes); //data is the memory of the location where you first write,
}
```

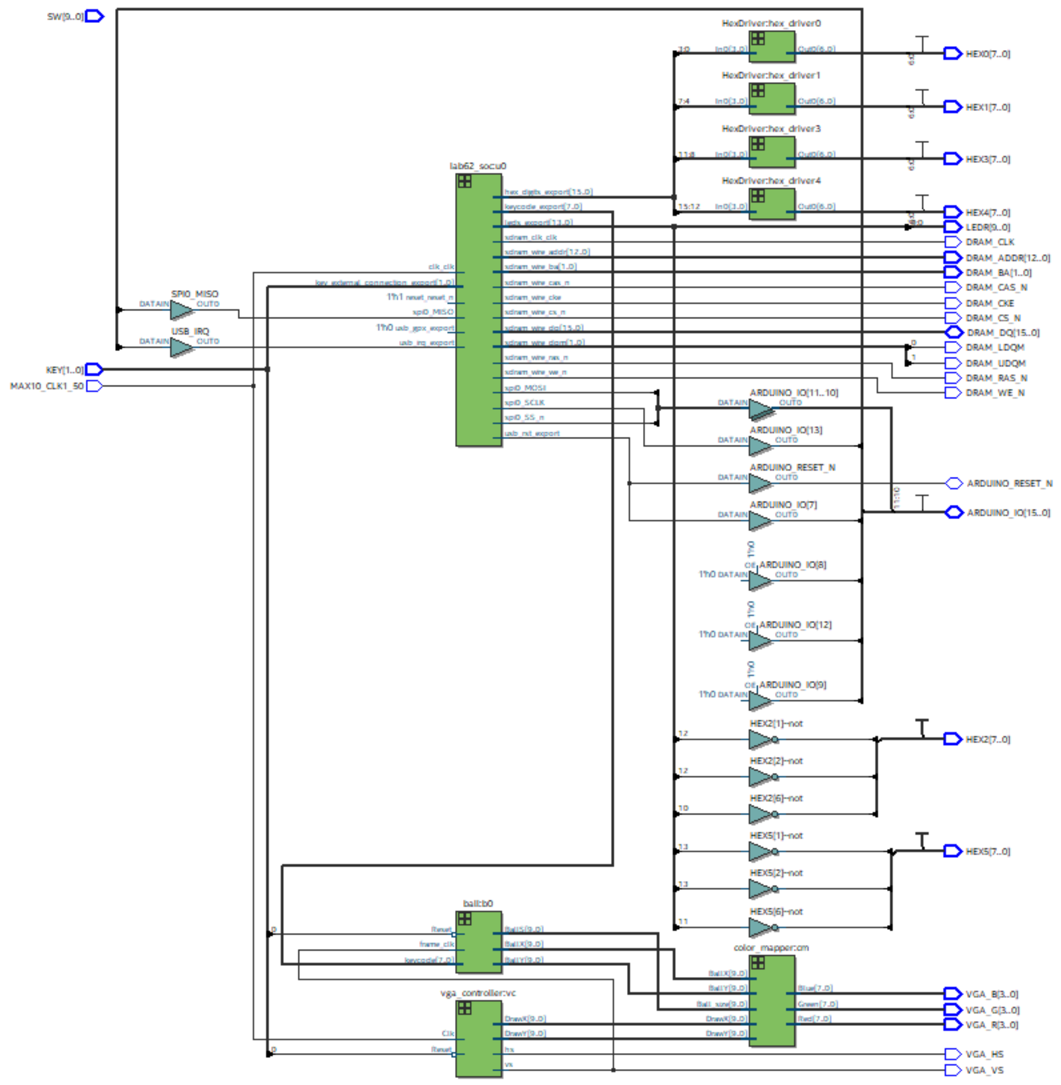
MAXbytes_rd reads multiple bytes. Unlike writing multiple bytes, write_length is only one to access the desired register while read_length is nbytes, to access the multiple bytes desired. The function returns the memory location after last written, similar to MAXreg_wr. Similar to the rest of the functions, an error check exists for MAXbytes as well.

VGA operation: Ball, Color Mapper, and VGA controller modules

The VGA controller module handles the synchronization of vertical and horizontal sync signals. This controls where the “electron gun” is pointing on the screen and also determines when to turn it on or off. The Ball module updates the position and motion of the ball. If a key is pressed, or a wall is hit, the motion changes accordingly. The color mapper module determines the shape of the ball and position from the ball module to determine which pixels should be which colors. Color mapper also uses the DrawX and DrawY signals from the VGA controller to determine position on screen and what RGB signals to send to actually draw the pixel.

Top Level Block Diagram

The following is the top-level block diagram for Lab 6.



Written Description of all .sv Modules

Module: lab62

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

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

Inout: [15:0] DRAM_DQ, ARDUINO_IO,
ARDUINO_RESET_N

Description: This is the top-level module used. Within this module, color_mapper, VGA_controller, ball, and lab62_soc are instantiated .

Purpose: The purpose of lab62 is to take user input and

Module: lab62_soc/synthesis/lab62_soc
Description: These are the hardware components created in the platform designer.

Purpose: These files allow the interaction with the keyboard and monitor peripherals.

Module: color_mapper
Inputs: [9:0] BallX, BallY, DrawX, DrawY, Ball_size
Outputs: [7:0] Red, Green, Blue

Description: Color_mapper determines the shape and size of the ball, from the ball module, and then determines whether the pixels on screen represent the ball or the background.

Purpose: The purpose of color mapper is to determine what color each pixel should be on screen.

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

Description: VGA_controller handles vertical and horizontal sync while outputting DrawX and DrawY signals to represent the part of the screen to be drawn, which is analyzed by other modules to determine what pixel should be drawn.

Purpose: The purpose of VGA_controller is to move the virtual electron gun across the screen and handle the vertical and horizontal sync in order to draw pixels.

Module: hexdriver
Inputs: [3:0] In0
Outputs: [6:0] Out0

Description: lab62 module receives the keycodes from the keyboard which is then passed to the hexdriver module in order to convert into a visual display for users.

Purpose: The purpose of hexdriver is to display the keycode of the most recent key pressed

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

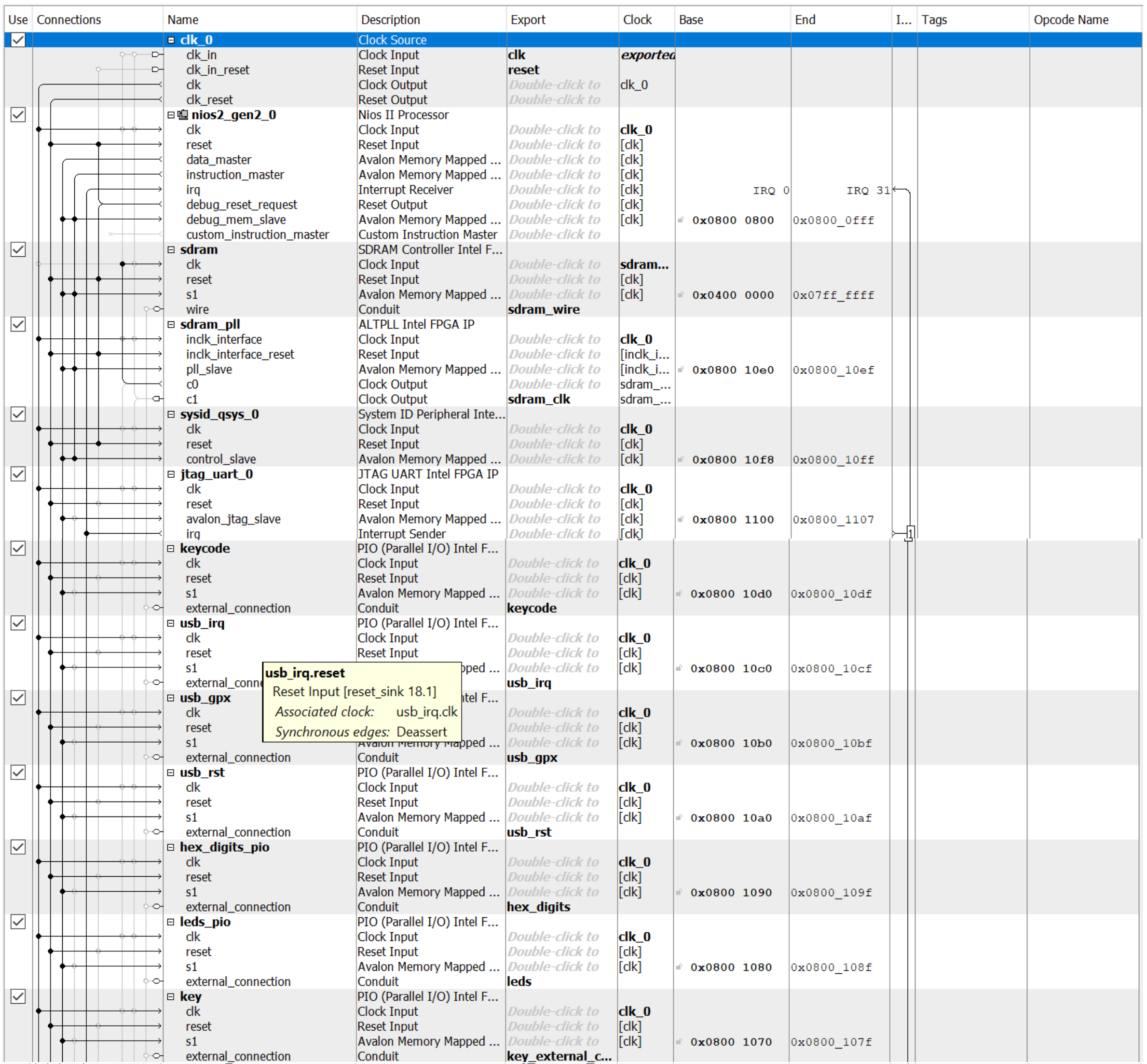
Description: The ball module updates the motion of the ball whenever a key is pressed. From the motion, BallX and BallY are calculated to determine the location of the center of the ball which is passed to color mapper to be drawn.

Purpose: The ball module controls the position and motion of the ball.

Module: lab6.sdc
Description: This module sets output delays for precise time signals coming from dram and false paths for the less time sensitive signals from the switches, keys, and leds.

Purpose: The purpose of this timing file is to ensure external signals are stable before measurement.

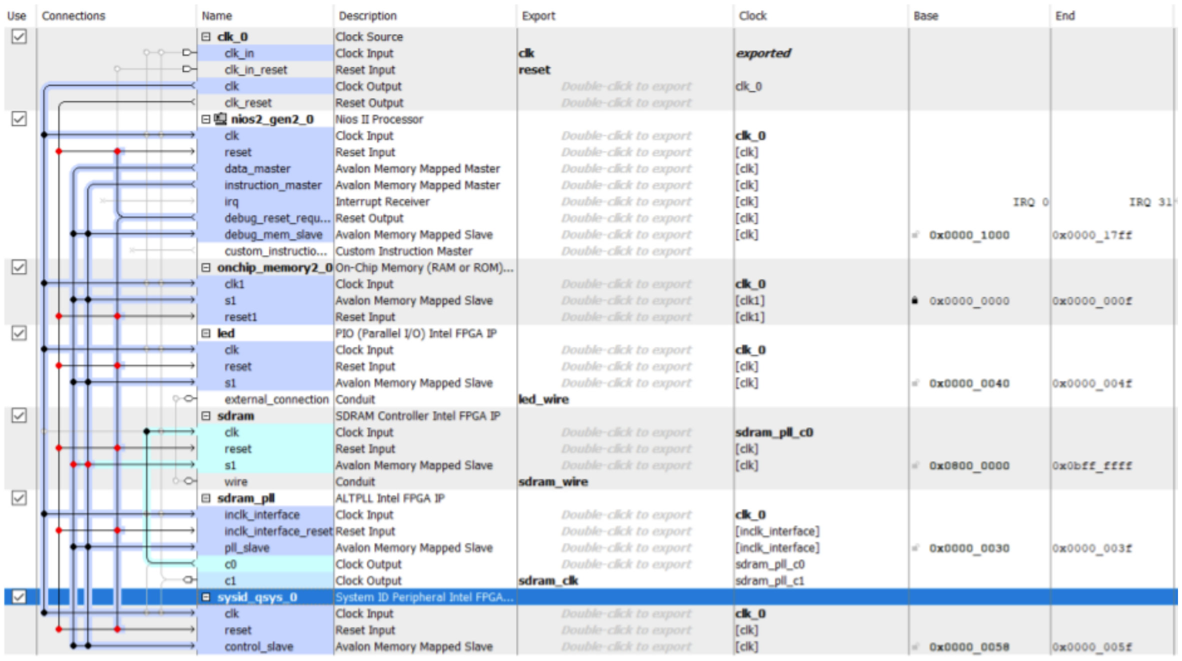
Diagrams of Platform Designer



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*	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 ...	*Double-click to*	[clk]					
		instruction_master	Avalon Memory Mapped ...	*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 ...	*Double-click to*	[clk]	0x0800 0800	0x0800_0fff			
		custom_instruction_master	Custom Instruction Master	*Double-click to*						
✓		⊟ **sdram**	SDRAM Controller Intel F...							
		clk	Clock Input	*Double-click to*	**sdram...**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0400 0000	0x07ff_ffff			
		wire	Conduit	**sdram_wire**						
✓		⊟ **sdram_pll**	ALTPLL Intel FPGA IP							
		inclk_interface	Clock Input	*Double-click to*	**clk_0**					
		inclk_interface_reset	Reset Input	*Double-click to*	[inclk_i...					
		pll_slave	Avalon Memory Mapped ...	*Double-click to*	[inclk_i...	0x0800 10e0	0x0800_10ef			
		c0	Clock Output	*Double-click to*	sdram_...					
		c1	Clock Output	**sdram_clk**	sdram_...					
✓		⊟ **sysid_qsys_0**	System ID Peripheral Inte...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		control_slave	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 10f8	0x0800_10ff			
✓		⊟ **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 ...	*Double-click to*	[clk]	0x0800 1100	0x0800_1107			
		irq	Interrupt Sender	*Double-click to*	[clk]			1		
✓		⊟ **keycode**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 10d0	0x0800_10df			
		external_connection	Conduit	**keycode**						
✓		⊟ **usb_irq**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	pped ...	*Double-click to*	[clk]	0x0800 10c0	0x0800_10cf			
		external_conn		**usb_irq**						
✓		⊟ **usb_gpx**	ntel F...							
		clk		*Double-click to*	**clk_0**					
		reset		*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 10b0	0x0800_10bf			
		external_connection	Conduit	**usb_gpx**						
✓		⊟ **usb_rst**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 10a0	0x0800_10af			
		external_connection	Conduit	**usb_rst**						
✓		⊟ **hex_digits_pio**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 1090	0x0800_109f			
		external_connection	Conduit	**hex_digits**						
✓		⊟ **leds_pio**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 1080	0x0800_108f			
		external_connection	Conduit	**leds**						
✓		⊟ **key**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 1070	0x0800_107f			
		external_connection	Conduit	**key_external_c...**						

usb_irq.reset
Reset Input [reset_sink 18.1]
Associated clock: usb_irq.clk
Synchronous edges: Deassert

Above is the Platform Designer view for Lab 6.2.



Use	Connections	Name	Description	Export	Clock	Base	End
✓		⊟ 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_0040	0x0000_004f
		external_connection	Conduit	led_wire			
✓		⊟ sdram	SDRAM Controller Intel FPGA IP				
		clk	Clock Input	Double-click to export	sdram_pll_c0		
		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_interface]		
		pll_slave	Avalon Memory Mapped Slave	Double-click to export	[inclk_interface]	0x0000_0030	0x0000_003f
		c0	Clock Output	Double-click to export	sdram_pll_c0		
		c1	Clock Output	sdram_clk	sdram_pll_c1		
✓		⊟ 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_0058	0x0000_005f

Above is the Platform Designer view for Lab 6.1.

Description of Platform Designer Modules
(Also found on page 2)

Between Lab 6.1 and 6.2, there are many shared modules within the platform designer. These modules are described below.

Clk_0 is a module that functions as the 50 MHz clock from the FPGA. This module outputs clk and clk_reset which are used as clock and reset inputs for most of the other modules

Nios2_gen2_0 is the 32 bit processor that is the centerpiece of the design. In this case, the version of NIOS II that is used is the NIOS II/e which is the resource optimized version of the processor. The processor sends a data output and instruction output to other modules.

SDRAM is the off-chip memory that is used in both labs. It has a size of 512 MBits and takes input from the data and instruction masters outputted by the NIOS II. An interesting thing to note about the sdram is that the input clock comes from the sdram_pll modules. This clock is phase shifted to allow the SDRAM to perform a data read or write in the correct window with respect to the master.

SDRAM_PLL creates the clock that is outputted to the sdram. This clock is delayed by 1ns to compensate for the difference between the master and slave clock phase.

Sys_id_qsys_0 checks for incompatibilities between the hardware and the software that are being connected. This is used as a preventative measure so that the FPGA is not overloaded and possibly ruined by the software that it is connected to.

Led and Leds_pio both perform the same function and act as pio to provide a 8-bit LED output for both the 6.1 and 6.2 labs

The following modules are specific to the Lab 6.1

SW this module acts as a 8-bit pio input. It is necessary to implement the accumulator function of lab 6.1 and comes from the switches on the DE-10 board.

Key is a 2-bit pio input to lab 6.1 and is necessary as the accumulate signal of the C accumulator function.

Onchip_memory2_0 is a module that is not used but included in the lab 6.1 design to demonstrate the possible benefits of using an on-chip memory for increased speed and ease of access for frequently accessed data.

The following modules are specific to Lab 6.2.

Jtag_uart_0 provides a way for the PC and FPGA to communicate a serial bit stream.

Keycode is an 8-bit PIO input to the processor that is necessary to read the code of the key pressed on the USB keyboard.

Usb_irq, usb_gpx, and usb_rst are all 1-bit PIO modules that are necessary to utilize the USB keyboard in conjunction with the FPGA.

Hex_digits_pio is a 16-bit PIO that allows for the keycode of the current pressed key to be outputted on the hex display on the DE-10 board.

Timer_0 is a 1-bit module that sends a signal every millisecond so that the NIOS II is able to keep track of how much time has passed.

Spi_0 is a module that allows for data transfer between master and slave and vice versa through a MISO and MOSI channel. This can be done simultaneously depending on the mode of the SPI. The SPI also has a slave select that is active low. This is the part where our ‘four function’ controlling reading and writing bits was implemented.

Software Description
(Also found with pictures starting on page 4)

In Lab 6.1, the first function was called the blinker. This function implemented a single blinking on the least significant LED on the DE-10. This was done by using a pointer to the address in memory of the led PIO previously described. The contents of the pointer (i.e. the LED data) is OR-ed with x1 within a for loop until i=100000. This is done so that the LSB of the LED data will be 1 for the time delay specified by the loop. Then, the contents of the LED address pointer is AND-ed with x0 within a for loop until i=100000. This is done to return the LSB of the LED data to 0 for the same time delay of the for loop. All of this is placed inside an infinite loop to repeat this process indefinitely and results in a blinking LED that is on and off for equal amounts of time.

The second function in Lab 6.1 is the accumulator. This function uses pointers to the addresses of the LED, SW, and Key PIO's described above. These pointers are necessary to access the data stored at these locations. To start the function, the contents of the LED pointer are set to 0 to reset all the LEDs to off. Then, a local variable called total is initialized to 0. If the contents of the Key pointer is 0, then the contents of the SW pointer (the values on the DE-10 Switches) is added to total, and the contents of the LED pointer is connected directly to the total variable. This is because the contents of Key pointer will be 0 only when the Accumulate button is pressed on the DE-10, since the buttons are active low. Then an empty for loop spanning from i=0 to 100000 is written to implement a software delay. All of this is placed in an infinite floor loop so that results can compound upon each other and be constantly tested on the DE-10 board.

For lab 6.2, the four functions that are implemented are the Maxreg_wr, the Maxreg_read, the Maxbytes_wr, and the Maxbytes_read. To implement all of these, the alt_avalon_spi_command function is used. For simplicity, assume this is included in all the functions as the write/read function.

To implement Maxreg_wr, the write/read command is called to write 2 bytes: the value of the register passed into the function +2 and the val to be written to that register. These bytes are passed to the write/read command putting them in an array and passing the starting address of the array.

To implement Maxreg_read, the write/read command is called to write 1 byte and read 2 bytes. In one command, the value of the reg to be read from is written and the value at that register is stored in a variable named val and returned from the function.

To implement Maxbytes_wr, the write/read command is called to write nbytes + 1 bytes. The value of the register to be written into +2 is one of those bytes, and the data pointer is input to the function is used to access the nbytes of data through a for loop. These nbytes and the register + 2 are combined in an array and passed into the write/read command. A pointer to the end of the data set is returned.

To implement Maxbytes_read, the write/read command is called to write 1 bytes and read n bytes. The value of the register to be read from is written, and nbytes are read from that location and stored in a data array. A pointer to the end of the array is returned.

Post Lab Questions

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

The NiosII/e CPU is slower than the Nios II/f. This is because the Nios II/e uses less resources.

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

Off-chip memory takes more access time because it has to travel through a longer datapath. If there is a need to access memory fast, then on-chip memory can be very helpful.

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 modified Harvard. This is because it has a common memory for instructions and data, but it has a different bus for both.

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 LEDs only need to access the data stream to set the LEDs as 1's or 0's. They don't need to access the instructions, since they don't store any program data other than the LED vector.

Why does SDRAM require constant refreshing?

The SDRAM is implemented with transistors and capacitors. If the SDRAM is not constantly refreshed, the memory value can decay to not be seen as a 1 or a zero due to the decaying nature of the capacitor.

You will need to determine the following parameters to instantiate the SDRAM controller. Refer to the DE10-Lite schematic and the IS42S16320D SDRAM (datasheet (PDF). Make sure you are looking at the correct part of the datasheet:

SDRAM parameter	Short name	Parameter Value
Data Width	width	16 bits
# of Rows	nrows	13
# of Columns	ncols	10
# of Chip Selects	ncs	1
# of Banks	nbanks	4

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

Data width/Access time = 16/(5.4ns) = 353 Mbytes/second.

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

There is a window where the SDRAM can communicate with the master with the correct data. If the SDRAM is run too slowly, this window will not always line up, and some data will be transferred incorrectly.

This puts the clock going out to the SDRAM chip (clk c1) 1ns behind of the controller clock (clk c0). Why do we need to do this?

This is necessary to line up the transfer window between the controller and the SDRAM. Without this line up, incorrect data would be transferred.

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

NIOS II starts executing at first SDRAM address, in our case 0x04000000. Assigning the vectors after addresses is necessary to avoid overlap between the addresses and the vector jumps.

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.

.bss is a region with uninitialized data (int x). .heap contains allocated memory (pointer=(int)malloc(sizeof(int)). .rodata contains read only data, or static constants (const int x=10). .rw data contains readable and writable memory, or a local variable (int x = 10). .stack contains function call data (int function(a,b) call). .text holds strings in memory (char x= “hello”).

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).

This is almost all explained in the Software Description section. The volatile keyword specifies that the variable is outside the program. This is necessary so that the program will not optimize the variable.

Design Resources and Statistics

LUT	3667
DSP	4
Memory (BRAM)	11264 bits
Flip-Flop	2430
Frequency	82.03 Mhz
Static-Power	96.5 mW
Dynamic Power	56.89 mW
Total Power	174.68 mW

Hidden Question:

Note that Ball_y_motion in the above statement may have been changed at the same clock edge that is causing the assignment of Ball_Y_pos. Will the new value of Ball_Y_Motion be used, or the old? How will this impact behavior of the ball during a bounce, and how might that interact with a response to a keypress? Can you fix it?

The old value is used in the motion calculations due to parallel assignments. This means that for one clock cycle, even though the bounce or key conditional was met, the ball would continue in its original motion before changing to the correct motion the next clock cycle.
This response delay is very short as it consists only in one frame. While this cannot be fixed for keypresses, this could be accounted for in bounces by setting new bounce conditions so that the bounce signal will occur one frame before it should actually bounce. This will cause the ball to continue its motion for one frame and reach the edge, but get the bounce command the next frame, so the ball would not appear to move into the wall at all.

Extra credit:

It is possible to glitch the ball off the top and left of the screen. If this happens, the ball would appear at the bottom and right of the screen respectively after a short delay. This is due to how the screen is formatted, with the top being the 0th row and the left being the 0th column. Due to the extra rows and columns at the bottom and right sides of the screen for blanking, this exact glitch doesn't occur there. To fix these bugs, one possible solution is to check the position of the ball when pressing a key. If the key is at its respective edge, it will bounce regardless of whether or not the key is pressed. Example shown below.

```
8'h1A : begin
         if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min )  // Ball is at the top edge, BOUNCE!
         begin
            Ball_Y_Motion <= 1;
            Ball_X_Motion <= 0;
         end
         else
         begin
            Ball_Y_Motion <= -1;//W
            Ball_X_Motion <= 0;
         end
       end
```

Conclusion

By the end of this lab, we had created a fully functioning ball that would change directions based off of key presses and bounce off the corners of the screen. The lab was straightforward and the only minor bugs were fixed by reading the FAQ about the "reset timeout!" and the program freezing after one keycode.

There was nothing ambiguous about the lab as the tutorials for both weeks for the lab clearly laid out how to create and edit many of the components needed for the lab. One thing that may be improved on is the explanation of alt_avalon_command, especially the reg + 2 for MAXreg_wr, because the intel documentation was rather difficult to understand. Otherwise, all the lab documentation was extremely helpful.