ECE 385

Fall 2022

Lab 6

SOC with NIOS II in SystemVerilog

Zhuoyuan Li, Shihua Cheng

Tianhao Yu

Introduction

Lab 6.1 aims to create a system-on-chip design using NIOS-II based system on the MAX10 device using a FPGA board. Processing low-performance tasks, NIOS-II is programmed with C and serves as a processor, whose program is run from the SDRAM and shown on LED by wiring them with the PIO module. Lab 6.2 then extended this functionality by using a VGA monitor and USB. The USB serves as a connection and communication between the computer host and the electronic devices. The NIOS-II writes characters to the FPGA board, communicates the USB driver to the SPI, which turns characters to bits and sends them to MAX3421E. The VGA monitor serves as the monitor where the result is displayed.

Written Description and Diagrams of NIOS-II System

module description

clk_0

input:

output: clk, clk_reset

export:clk_in, clk_in_reset

description: clk_o serves as the clock source of the whole design. It provides clk and clk_reset signals for all other modules.

purpose: Provide clk and clk_reset signals for all other modules.

onchip_memory2_0:

input:clk1, s1, reset1

output:

export:

description: The block of synchronized RAM on the FPGA board with a total memory size of 16 bytes.

purpose: Connect the synchronized RAM on the FPGA board to the NIOS-II processor.

sdram

input:clk1, s1, reset1

output:

export: wire

description: Synchronous Dynamic RAM, used to store software programming.

purpose: Store the software programming since the on-chip memory has a very limited memory size.

sdram_pll

input: inclk_interface, inclk_interface_reset, pll_slave

output: c0

export: c1

description: Since the sdram requires precise timing, this module provides the clock signal for SDRAM, and compensates for clock skew in board layout. Its output c0 is the 50MHz control clock which is feeded to the SDRAM controller. This clock is always synchronized. Its c1 is the clock feeded to the SDRAM itself with the same frequency but with a phase shift of 1ns behind the c0, the clock given to the control unit.

purpose: This module provides the clock signal for SDRAM and the SDRAM controller.

sysid_qsys_0

input: clk, reset, control slave

output:

export:

description: This module enables the design to prevent loading software onto an FPGA with an incompatible NIOS II configuration by producing a serial number that's assigned to be 0 which will be checked by the software loader before the software execution.

purpose: A system ID checker to ensure the compatibility between hardware and software

nios2_gen2_0

input: clk, reset, debug_mem_slave

output: data_master, instruction_master, debug_reset_request

export:

description: This module is the NIOS-II processor, used to execute c programs. The data_master is a port connected to the data memory in order to read or write data when the

processor executes a load or write instruction, respectively. The instruction_master fetches instructions to be executed by the NIOS-II processor.

purpose: The processor used to execute C programs.

sw

input: clk, reset, s1

output:

export: external_connection

description: The external connection is connected to the switch on the FPGA board. This module is used to load the switch input.

purpose: Used to load switch input on the FPGA board.

accumulate:

input: clk, reset, s1

output:

export: external_connection

description: Used in lab 6.1. It's externally connected to the key on the FPGA board .Accumulate the input once per press.

purpose: Used to accumulate the input.

led:

input: clk, reset, s1

output:

export: external_connection

description: This synchronized module is externally connected to the led wire that wires to the leds on the FPGA board.

purpose: Module that wire to the leds in order to display the result.

In lab 6.1, input includes the switch input, accumulate signal, and the reset signal. The reset button, or key[0], is connected to the clk_0 module in the platform designer, which will then send the reset signal to other modules. The accumulate module is external connected to the key[1] on the FPGA board, and in main.c, a pointer to the accumulate block is defined, and the c program would use a if statement inside a infinite while loop to increase the value SW_PIO whenever the the key [1] button is pressed. The sw module is externally connected to the switch input, and it could be accessed by the processor by a pointer within the c program that points to the address of the sw block. The output will be displayed through the led module; similarly, a pointer to the address of the Led module is declared within the c program, and the output to the led will update once every time the accumulate button is pressed using an if statement. The led module is externally connected to the actual leds on the FPGA board to show the display.

NIOS-II reads the keycode using the USB driver and transmits that to the FPGA board, then theFPGA uses the keycode to update the ball's position together with the pixel's RGB and return that to the VGA component. The NIOS-II communicates to MAX3421E through SPI and the four functions we implemented in MAX3421E.c.

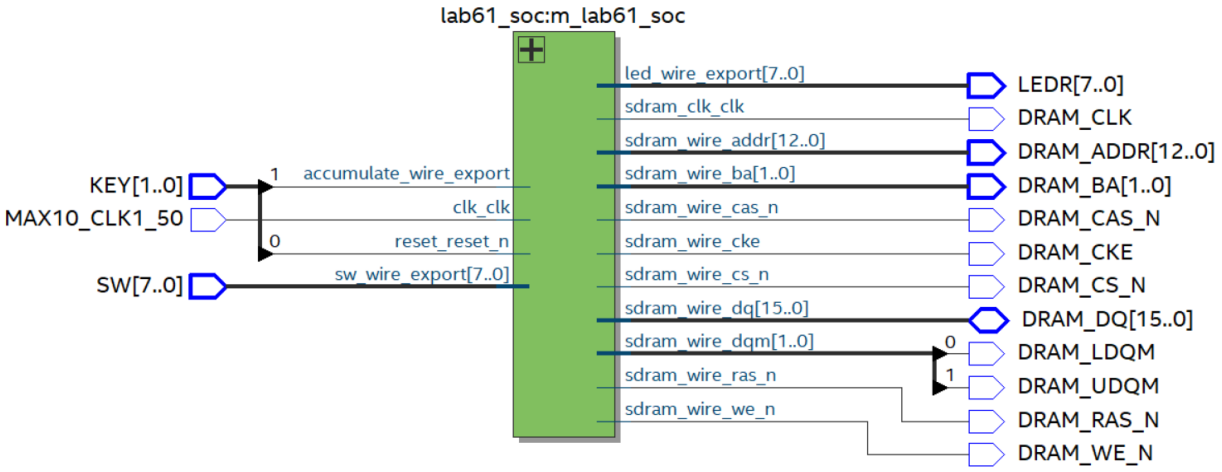
SPI is used for chip to chip communication. In this case, it's used for the communication between the USB driver and the MAX3421E. The SPI port consists of 4 signals: SCK, MOSI, MISO, SS. Whenever a alt_avalon_spi_command is called, the SS, or slave select, will go low, and if a write is operated, the MOSI, or master out slave in signal will start to shift out data from the most significant bit to the least when the master, or FPGA in this case, will toggle CSK for 1 clock cycle for every bit. The data sent out consists of 2 bytes. The first byte is the command byte: its first five bist indicate the register we wanted to SPI write into, the 6th bit is zero, the 7th bit is 1 when doing write, and 0 when performing read, and the 8th bit is not used in this lab. After the command byte is written, the actual data bytle will be written. If a read is being performed, the command byte's 7th bit will be 0, then MISO will send the data to the FPGA, so it's 1 byte read and 1 byte to write command byte.The FPGA will still toggle for 8 clock cycles. In this lab, we only use the chip in half duplex mode, so sending out data and receiving data will not be performed at the same time.

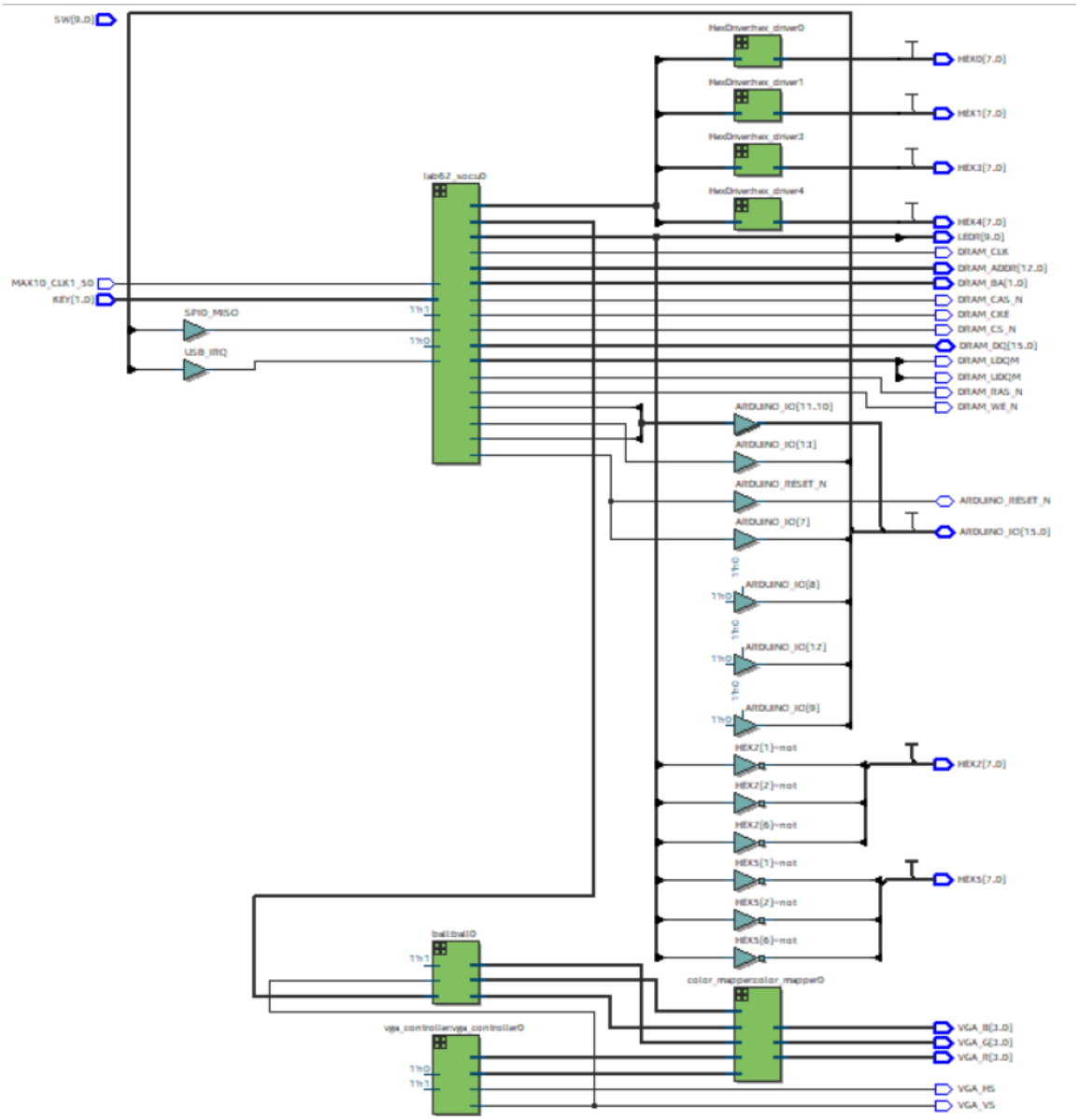
In order for Nios II processor to communicate with the MAX3421E controller, Four functions in total are being implemented, void MAXreg_wr(BYTE reg, BYTE val), BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data), BYTE MAXreg_rd(BYTE reg), and BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data). The first two functions are to write and write multiple bytes to the register of MAX3421E through SPI peripheral, which will be further discussed later in this article. The latter two functions are read and read multiple bytes of the register in MAX3421E. Multiple bytes read and multiple bytes write returns a pointer to the last memory position after last read/written.

For the VGA operation, the ball.sv calculates the position, or coordinate of the ball, and ensures that the ball moves is controlled by the keyboard and bounces when it hits the wall. The output Ballx, Bally, and Balls all indicate the position of the ball and the size of the ball are given to the color-mapper. The VGA controller controls the actual electron gun that lights

up the display screen, its output drawx and draw indicates the exact coordinate of the beam, and these are given to the color_mapper. The color_mapper will calculate a distance between the beam and the center of the ball x and a distance y using the center coordinate of the ball, Ballx and Bally, and the drawx and drawy, coordinate of the beam. Once the distance is smaller than the radius of the ball, or Balls, the color mapper will output red, green, blue that correspond to the ball's color; otherwise the color mapper output the RGB corresponds to the background color.

Top-level block diagram (Lab 6.1 & Lab 6.2)





Written Description of all .sv modules

Module: lab61.sv

Inputs: [7:0] SW, [1:0] KEY, MAX10_CLK1_50, [15:0] DRAM_DQ

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: This is the top-level module that connects the Nios II processor with the rest of the hardware. The input MAX10_CLK1_50 is the 50MHz clock generated by the FPGA board. SW is the 8-bit input from the on-board switches, KEY is the two keys on the board for reset and accumulate. The output LEDR is connected from the exported LEDR signal from the sdram in the platform designer to the on-board LEDs so that one word inside the

sdram can be visualized. DRAM_DQ is an inout, it is the data between the CPU and the DRAM. DRAM_CLK is the clock for DRAM generated by the PLL for precise timing. The other DRAM_* signals are needed for the operation of the DRAM.

Purpose: This is the top-level module that integrates the Nios II processor with the rest of the hardware.

Module: lab62.sv

Inputs: [9:0] SW, [1:0] KEY, MAX10_CLK1_50, [15:0] DRAM_DQ, [15:0] ARDUINO_IO, ARDUINO_RESET_N

Outputs: [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, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, [9:0] LEDR, VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B

Description: This is the top-level module that integrates the SOC along with the VGA controller module, the color mapper module, and the ball module. The SOC contains the Nios II Processor, the sdram and sdram controller, and some exported ports for I/O. The SOC is used as a USB driver that reads the input from a keyboard connected through a MAX3421E chip to the SOC. The SOC can both read and write to the registers on MAX3421E to obtain keyboard inputs, which will be introduced later. The DRAM_* signals have the same functionality as introduced above in lab61.sv. Six hexdrivers with outputs HEX* are used to visualize the value of the current keypress on the on-board hex displays. The ball module calculates the position and size of the ball according to the keypress, and the color mapper module calculates the color of the current pixel being drawn to the screen according to the position of the ball. The VGA controller generates the outputs VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G and [3:0] VGA_B. VGA_HS is a 31.468kHz clock that is the horizontal sync signal, and VGA_VS is a 60Hz clock that is the vertical sync signal. VGA_R, VGA_G and VGA_B are values that represent the Red, Green and Blue value of the current pixel being drawn to the screen. All the VGA_* signals are connected to the VGA port on the FPGA board and sent to the monitor.

Purpose: This is the top-level module that connects the SOC as a USB driver and the FPGA board that calculates ball position and generates VGA signal output to a monitor. It is able to execute a program that accepts keyboard input to change the motion of a bouncing ball on the monitor.

Module: VGA_controller.sv

Inputs: Clk, Reset

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

Description: This is the module that controls the VGA signal output on the FPGA board, so that contents can be displayed on an external monitor. It has asynchronous reset, and it uses the 50MHz Clk input to generate the 25MHz pixel_clk signal. This 25MHz pixel clock provides 25MHz / 800 (horizontal) / 524 (vertical) = 59.6Hz refresh rate on the external monitor, which is the output vs. HS is the horizontal sync signal, it is equal to 25MHz/800 = 31.25KHz. Blank = 0 when the electron beam is in the blanking interval and the output color should be RGB=000. Sync is not used in this lab. DrawX is a number from 0-639, DrawY is a number from 0-479, they represent the coordinate of the current pixel being drawn.

Purpose: This module is used to control the output VGA signal on the FPGA board so that contents can be displayed on an external monitor.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: HexDriver converts a 4-bit unsigned integer (0-F) to hexadecimal form that is mapped to the LED on the FPGA board.

Purpose: The content of the register can be visualized on the FPGA board.

Module: Color_Mapper.sv

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

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

Description: The Color_Mapper takes DrawX and DrawY from the VGA controller, and calculates the color of the current pixel being drawn. It does so by taking BallX, BallY and Ball_size from the ball instance, which is the position and size of the ball. If (DrawX, DrawY) is on the ball, the Color_Mapper outputs the RGB value for orange, so that the ball is painted in orange. Otherwise, the Color_Mapper outputs the RGB value for blue, so that the background is painted in blue. The output RGB values are used as inputs for the VGA controller to determine the color of the current pixel.

Purpose: This module is used to determine the color of the current pixel being drawn to the screen according to the current position & size of the ball.

Module: ball.sv

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

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

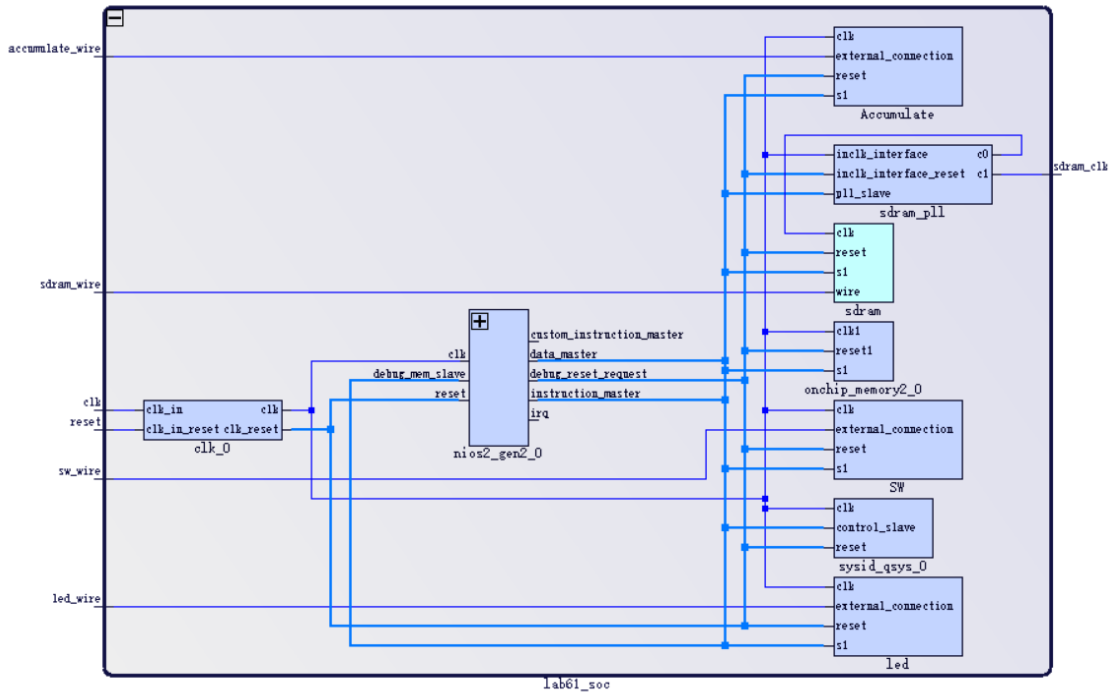
Description: This module is used to calculate the position of the ball. It has asynchronous reset which sets the position of the ball to the center of the screen. The input frame_clk is connected to the output VS of the VGA_controller, which is around 60Hz. This is because the update rate of the ball should be equal to the refresh rate of the screen. Keycode is the output from the Nios II system, which is the code of the key read from the USB input of the keyboard. BallS is a constant predefined value representing the radius of the ball. BallX and BallY represent the position of the ball. It calculates the position by updating one step per frame_clk. The direction of the step is determined by the keycode. It also detects wall contact, so that the ball will bounce (step direction negated) when it reaches a wall.

Purpose: This module is used to calculate the position coordinates of the ball, which is controlled by a keyboard and bounces when it reaches the edge of the screen. The output is used by the color_mapper to map the correct colors.

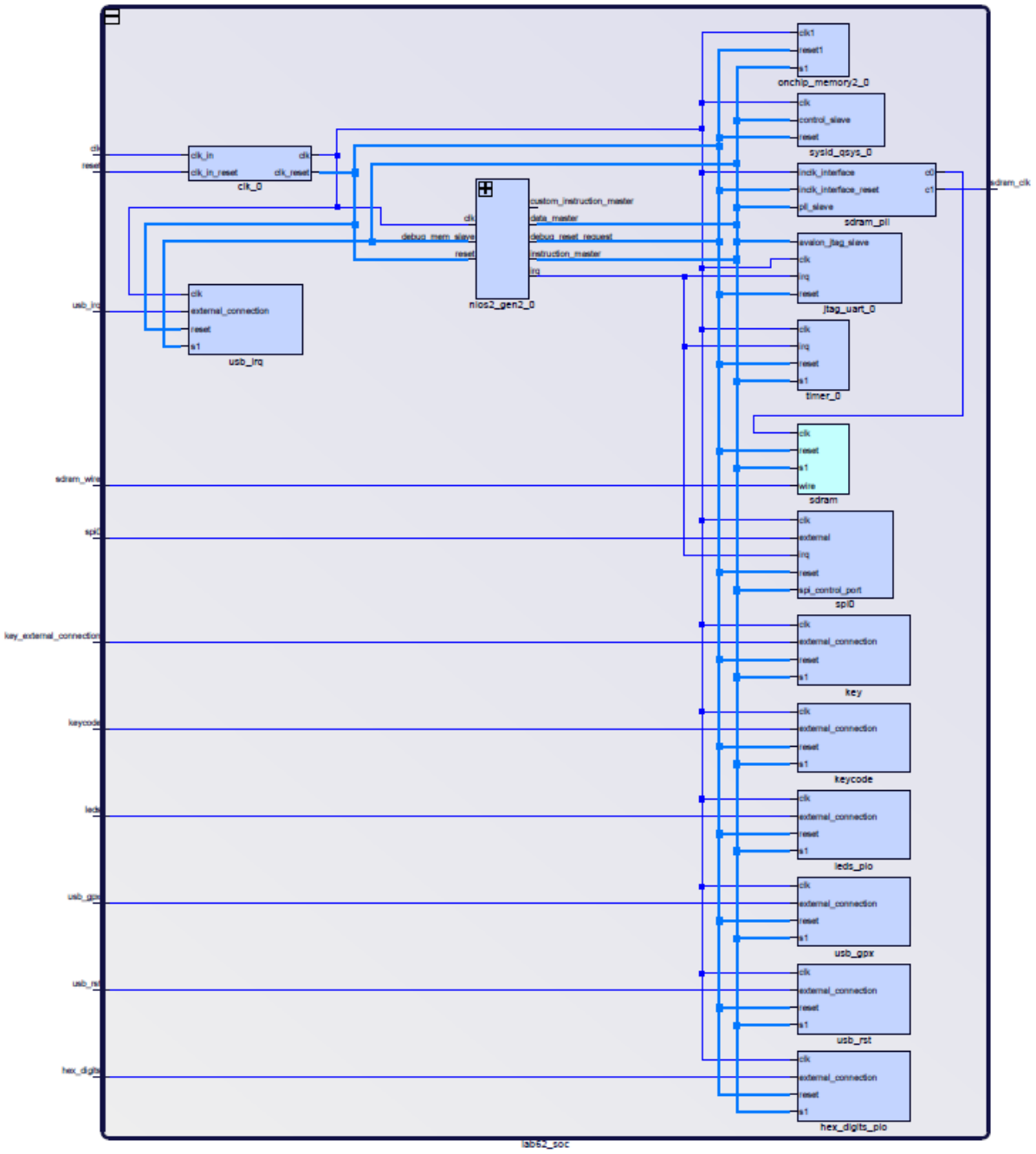
The PIO blocks are introduced in the next section.

System Level Block Diagram

Lab 6.1:



Lab 6.2:



Common blocks:

nios2_gen2_0: This is the Nios II/e processor used to execute C programs. The inputs Clk and reset are from the FPGA. The data_master is connected to the SDRAM to write and read data needed for the program. The instruction_master is connected to the SDRAM to retrieve instructions.

onchip_memory2_0: This is the on-chip memory. It can only store 16 bits in our design and is used as a placeholder. This block is not used. It is a synchronized RAM on the FPGA board with faster access speed than SDRAM.

sdram: This is the Synchronous Dynamic RAM used by the Nios II processor to store data and instructions to execute the program. It has 512 Mbits of size.

sdram_pll: This is the sdram controller. It adds a 1ns phase to the input clock from FPGA and feeds the output clock that lags the FPGA clock to the SDRAM. This is needed to compensate for clock skew in board layout. The 1ns delay guarantees that all synchronous signals are stabilized at the rising clock of the SDRAM clock.

sysid_qsys_0: This module produces a serial number which will be checked by the software loader before software execution. It prevents loading software onto an FPGA with incompatible Nios II configuration.

PIO blocks for Lab 6.1:

SW: This is a 8-bit parallel I/O block. It is assigned a base address of 0x0060 to 0x006f. It is externally exported to the FPGA, and connected to the on-board switches. The Nios II processor can read the value from the switches by accessing memory location 0x0060.

LED: This is a 8-bit parallel I/O block. It is assigned a base address of 0x0070 to 0x007f. It is externally exported to the FPGA, and connected to the on-board LEDs. The Nios II processor can write a value to the LEDs by accessing memory location 0x0070.

Accumulate: This is a 1-bit parallel I/O block. It is assigned a base address of 0x0050 to 0x005f. It is externally exported to the FPGA, and connected to the on-board key. The Nios II processor can read the value from the accumulate key by accessing memory location 0x0050.

PIO blocks for Lab 6.2:

usb_irq, usb_gpx, usb_rst: These are 1-bit parallel I/O blocks that are necessary for the USB keyboard to function correctly.

keycode: This is a 8-bit parallel I/O block. It is assigned a base address of 0x0180 to 0x018f. It is the input of the key read from the USB keyboard represented in 8 bits. It is externally exported to the FPGA for the ball.sv module to calculate ball position.

hex_digits_pio: This is a 16-bit parallel I/O block. It is assigned a base address of 0x0170 to 0x017f. Each 4 bit in the 16-bit value represents a hexadecimal number from 0-F. This block is externally exported to the hexdrivers in the FPGA top module so that the current keycode can be displayed on the on-board hex displays.

leds_pio: Same as Lab 6.1, used to produce on-board LED output.

key: This is a 2-bit parallel I/O block, used for Lab 6.1 and not used in 6.2.

Other blocks for Lab 6.2:

timer_0: This is a timer that sends out a signal per millisecond. This is used by the Nios II processor for USB timeout.

spi0: This is an SPI block. It is used for the data transfer in master mode. The Nios II (master) can send data to the MAX3421E (slave) through MOSI, and it can read data from the slave through MISO. SS (slave select) is active low for this lab. This block can be controlled by the Nios II using the alt_avalon_spi_command() function.

Description of the software component

For Lab 6.1, we wrote two programs: the LED blinking program and the accumulator program. For the LED blinking program, we first looked at the Platform Designer to determine the base address of the LED PIO. It is 0x40, so we declared a pointer to a volatile unsigned integer named LED_PIO and set its value to 0x40. Then we were able to change the value of the on-board LEDs by writing an 8-bit word to the base address 0x40 in the program because the PIO connects the RAM with a wire exported to the on-board LEDs. First, we did a clear by running *LED_PIO = 0. Then, inside an infinite while-loop, two for-loops that have 100,000 iterations each are used to create some delay for the blinking. Between the delays, the least significant bit is set by ORing *LED_PIO with 16'h0001 so that all upper 15 bits are not changed (x OR 0 = x) while the lowest bit is set to 1 (x OR 1 = 1). It is reset by ANDing *LED_PIO with 16'hFFFE, so that the upper 15 bits are not changed (x AND 1 = x) while the lowest bit is set to 0 (x AND 0 = 0). Therefore, the sequence is delay→set→delay→reset→delay→set→…, and the LED will keep blinking. The meaning of the volatile keyword is explained in the next section.

For the accumulator program, we assigned an extra PIO named accumulate, exported to an on-board key. We first determined the base addresses for LED, SW and accumulate:

```
volatile unsigned int *LED_PIO = (unsigned int*)0x70; //make a pointer to access the PIO block
volatile unsigned int *SW_PIO = (unsigned int*)0x60;
volatile unsigned int *Accumulate_PIO = (unsigned int*)0x50;
```

Then the LEDs are cleared using *LED_PIO = 0. Following is the main loop:

```
15  while ( (1+1) != 3) //infinite loop
16  { // during one button press
17   if ( (*Accumulate_PIO) == 0 ) {
18    // accumulate from switch
19    i = ( i + (*SW_PIO) ) % 256 ;
20
21    // run once
22    while ( (*Accumulate_PIO == 0) ){}
23   }
24
25   // output to LED
26   *LED_PIO = i;
27  }
28
29  return 1; //never gets here
30 }
```

The loop detects if the accumulate key is pressed by accessing *Accumulate_PIO because it is exported to the on-board key. If the key is not pressed, then the value should be 1. If the key is pressed, the current value i is incremented by the input from the switches read from *SW_PIO. The value should overflow when it exceeds 256 since there are only 8 LEDs, so it is set to the remainder divided by 256. In order for the loop to only run once for each keypress, we used another while loop to ensure that the outer loop does not reach the next iteration until the key is released. Finally, the updated value is written to *LED_PIO to show on the on-board LEDs. If no key is pressed, then the LEDs will maintain the current value.

For Lab 6.2, we needed to fill in four register write & read functions in MAX3421E.c. The input BYTE reg for each function is an 8-bit value, with its lower 3 bits = 0 and upper 5 bits representing a register from 0 to 31. For the Nios II processor to communicate with the MAX3421E controller, a command byte must first be written to the MAX3421E. The upper 5 bits of the command byte indicate the target register, and the 2nd lowest bit is W/!R, which is 1 when doing a write and 0 when doing a read. The remaining 2 bits are kept low. Therefore, for the MAXreg_wr function and the MAXbytes_wr function, the command byte is calculated as reg + 2. Then, we created an BYTE array “arr” to store the data to be sent to the MAX3421E, with its first element being the command byte. For MAXreg_wr, the length of the array is 2. The second element is the input val, being written to MAX3421E. For MAXbytes_wr, the length of the array is 1+nbytes. All the elements after the command byte are copied from the input array BYTE* data. After this is done, the function alt_avalon_spi_command() is called:

```
int ret_code = alt_avalon_spi_command(SPI0_BASE, 0,
                                      1+nbytes, arr,
                                      0, NULL,
                                      0);
```

SPI0_BASE is a macro representing the base address of the SPI block, and 0 represents the slave number (we only had one slave). For MAXreg_wr, nbytes = 1. The second line of arguments is the length of data being written and the data being written. Since we don’t want

to read any data for write functions, the third line is 0 and NULL. The last argument is flags, but we don't need any. ret_code should be the number of bytes being read, so it should be zero in this function. If it is less than zero, an error is thrown. MAXreg_wr does not have a return value, and MAXbytes_wr returns a pointer to the memory location after last written.

For the read functions, the command byte is just the input reg because the W/!R bit should be zero. The command byte should be first sent, then we can start to read feedback from MAX3421E. To achieve this, we used this function call:

```
ret_code = alt_avalon_spi_command(SPI0_BASE, 0,
                                  1, &reg,
                                  nbytes, data,
                                  0);
```

Only one function call is needed because the chip is run in half-duplex mode, which means that when write and read bytes are both larger than zero, then write will happen first, and read will happen second. In this case, the command byte will be written first, then the value read from MAX3421E will be stored in the pointer BYTE* data. The argument nbytes = 1 in MAXreg_rd, and is provided as a function input in MAXbytes_rd. The ret_code is then checked to see if there are any errors. The MAXreg_rd function will return a BYTE, which is the byte read from MAX3421E. The MAXbytes_rd function will return a BYTE* pointer pointing to the next position of the last element read from MAX3421E.

Answers to all INQ & Post lab questions

During Lab 6.2, we encountered a problem that caused the USB driver to be stuck in state 4. We tried using different keyboards and confirmed that it was not the keyboard's problem. We originally used two alt_avalon_spi_command() calls in the write and read functions, the first one sending the command byte and the second one sending/receiving the data. We found out that there might be problems with this implementation. Instead, using only one alt_avalon_spi_command() call is enough and efficient. We then added the command byte and the data bytes to a BYTE* array and send the array using one function call, and the problem was solved.

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

"e" stands for "economic", "f" stands for "fast". Nios II/f is designed for faster performance at the cost of larger area and power. It supports features such as hardware multiply/divide. Nios II/e is smaller in area and has lower power consumption and lower performance. It is fast enough for the USB driver function.

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

On-chip memory has lower latency, so it has faster access time compared to off-chip memory. Also, certain memory blocks can have initialized contents when the FPGA powers up, so data

constants can be stored during boot up. It also supports dual port accesses, while the off-chip memory doesn't.

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

The Nios II is a "modified Harvard" machine. This is because it uses separated instruction and data buses, but the instructions and data are stored in the same memory.

The led peripheral only needs access to the data bus. Why might this be the case?

The SDRAM or on-chip memory is used to store both the program (the instructions) and the data needed for the program, so it must be connected to both the data bus and the instruction bus. However, the Nios II processor does not need to access instructions from the led peripheral. The only function of the led peripheral is to read a word of data (instead of instruction) stored in the SDRAM, and connect the data to the on-board LEDs. Therefore, the instruction bus is not needed by the led peripheral, only the data bus is needed.

Why does SDRAM require constant refreshing?

In SDRAM, data bits are stored in capacitors, and the capacitors will lose data (charge leak) over a short period of time. If we want to keep the data for longer than that, then refreshing is needed to save the data by keeping the electrical charge in the capacitors.

How did we come up with 512 Mbit?

Data width = 16 bits, Row address bits = 13, Column address bits = 10, number of banks = 4, number of chips = 1.

Total number of bits = num of rows * num of cols * data width * num of banks * chip selects

$$= 2^{13} * 2^{10} * 2^{4} * 2^{2} * 1 = 2^{29}\, bits = 2^{9} Mbits = 512 Mbits$$

What is the maximum theoretical transfer rate to the SDRAM?

Transfer rate = 16 bits / access time = 2 Bytes / 5.4ns = 370 MB/s.

The SDRAM cannot be run below 50MHz. Why might this be the case?

The Nios II processor runs at 50MHz. If the SDRAM runs below 50MHz, then the clock window of the processor and the SDRAM will not align, data may be transferred incorrectly.

This puts the clock going out to the SDRAM chip 1ns behind the controller clock. Why do we need to do this?

The PLL must be tuned to introduce a clock phase shift so that SDRAM clock edges arrive after synchronous signals have stabilized. In our case, 1ns is enough for the synchronous signals to stabilize.

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

0x0800 0000. This is the base address of SDRAM. We need to set the vectors for the Nios II to find the first instruction based on the SDRAM base address, so that it does not corrupt or retrieve anything irrelevant.

Meaning of the volatile keyword, and how set and clear functions work.

The *volatile* keyword is used to declare a variable, and it tells the compiler not to make any optimizations for it because the value may be changed by external sources even if it is not changed within the code. In lab 6.1, the volatile keyword implies that the value LED_PIO in the SDRAM may be changed even if it is not explicitly changed in the code because the SDRAM is exported to FPGA. Set and clear have been explained in a previous section.

What does each section (.bss, .heap, .rodata, .rwdata, .stack, .text) mean? Give an example of C code: const int my_constant[4] = {1, 2, 3, 4}

.text: this section contains executable instructions of the compiled program. This section is usually read-only to avoid modifying the program.
Example: any compiled C program puts its machine code into the .text segment.

.bss: this section contains all global variables and static variables that are initialized to zero or do not have explicit initialization.
Example: int num; will put the variable num into .bss.

.heap: this section contains variables that are dynamically allocated. It is managed by functions such as malloc, realloc and free.
Example: int* a_ptr = (int*)malloc(sizeof(int)); will create an int variable on the heap that can be accessed using *a_ptr.

.stack: this section contains the program stack. Function call pushes data onto the stack as an “activation record”, including function arguments, return address/return value, and local variables. Function return pops the activation record.
Example: int f(int arg) {int a = 5; return a;}; f(6); will push the argument 6, the return address, the space for return value, and the local variable a into .stack. After the function returns, this data will be popped.

.rodata: this section stores all initialized constant data, and is read-only. The data can be global or static. String literals and const declared variables are stored in this section.
Example: const int num = 1; will place 1 into the .rodata segment.

.rwdata: this section stores all initialized global and static data. The data can be modified. Example: int num = 1; will place 1 into the .rwdata segment.

Description of the bug in ball.sv

In ball.sv, there is a bug that causes the ball to move through the wall or move diagonally. This is because keycode and wall contact are detected in the same cycle, and the motion of keycode is prioritized (it overwrites the wall bounce motion). However, in the situation where wall contact and key press are both detected, the expected behavior is to prioritize wall bounce and ignore the keypress for that frame. In order to fix the bug, we used an if-statement to choose between the wall and the keypress, so that only one can be executed at each cycle. First, wall contact will be checked. If there is wall contact, then the motion change will be only due to the wall, and the keypress is ignored. If not, the motion change will be decided by the keypress. Finally, the bug was fixed.

Design Resources and Statistics

Week 1:

LUT	2,266
DSP	0
BRAM	10,368 bits
Flip-Flop	1,807
Max Frequency	83.67 MHz
Static Power	96.43 mW
Dynamic Power	45.67 mW
Total Power	159.49 mW

Week 2:

LUT	3,239
DSP	0
BRAM	11,392 bits
Flip-Flop	2,465
Max Frequency	72.48 MHz

Static Power	96.45 mW
Dynamic Power	47.10 mW
Total Power	164.00 mW

Conclusion

For Lab 6.1, our design successfully executed the LED blinking program as well as the accumulator program using the NIOS II processor. For Lab 6.2, we successfully wrote the USB driver and ran it on the NIOS II processor so that keyboard inputs can be recognized and used to move the ball on the screen. All parts worked as expected. There wasn't anything ambiguous in the lab manual.