

University of Illinois at Urbana-Champaign

ECE385

Digital Systems Laboratory

Lab6: SOC with NIOS II in SystemVerilog

TA: Abigail Wezelis, Harris Mohamed

Yan Miao, Guangxun Zhai
yanmiao2, gzhai5

March 2021

Contents

1	Introduction	1
1.1	Basic Functionality of the NIOS-II Processor	1
1.2	Operation of the USB/VGA Interface	1
2	Written Description and Diagrams of NIOS-II System	1
2.1	Hardware Component	1
2.2	How the I/O Works	1
2.3	NIOS-II Interaction	1
2.4	Written Description of the SPI Protocol	2
2.5	Functions of C Codes	2
2.6	VGA Operation	2
3	Top Level Block Diagram	2
4	Description of all .SV modules	3
5	Lab 6.2 Extra Credit	5
6	System Level Block Diagram	6
6.1	Modules shared in Lab 6.1 and 6.2	6
6.2	Modules Only Included in Lab 6.1	7
6.3	Modules Only Included in Lab 6.2	7
7	Software Description	9
7.1	Accumulator	9
7.2	The USB/SPI in Lab 6.2	10
8	INQ Questions	11
9	Design Resources and Statistics	13
9.1	Statistics for Lab 6.1	13
9.2	Statistics for Lab 6.2	14
10	Conclusion	14
10.1	Bugs Encountered	14
10.2	Functionality of Our Design	14
10.3	Suggestions to Lab-Manual and Given Document	15

1 Introduction

1.1 Basic Functionality of the NIOS-II Processor

The NIOS II is an IP-based 32-bit CPU which can be programmed using a high-level language. We used C codes for the software part and made the NIOS II be the system controller and handle tasks which do not require much high performance. The major functionality of our processor in this lab is letting the led blink and doing accumulations.

1.2 Operation of the USB/VGA Interface

In Lab 6.2, the USB protocol is handled in the software on the Nios II, and the extracted keycode from the USB keyboard is then sent to the hardware. During week 2, we used the keyboard (majorly w, a,s,d) for inputs to control the ball to move in the monitor. The keyboard sends data to FGPA depending on the USB interface, and our VGA interface is the monitor and the graph shown on the screen.

2 Written Description and Diagrams of NIOS-II System

2.1 Hardware Component

Our hardware component consists of a NIOS-II Processor, an on-chip memory, an off-chip memory, PIOs like keys, HEX Driver and LEDs, PLL, ID checker, timer, and several USB's PIOs(E.g., usb_irq). All detailed description for these components can be found in Section 6 of the report.

2.2 How the I/O Works

In lab 6.1, the inputs are the two 1 bit keys that control the reset operation and accumulate operation. The output is the led that only receives data to do the on/off operations. We connected them to the same clock in the hardware designer and write a c code to perform the complicated operation of the blinking.

2.3 NIOS-II Interaction

Lab 6.1 and lab 6.2 almost share similar NIOS II interactions. They both have a clock that connects all the components, a NIOS II economic version processor that reads and runs the instructions, an on-chip memory, an SDRAM as an off-chip memory, and a clock that has some delay to be adapted to the SDRAM. For lab 6.2, we also introduced the JTAG UART peripheral so that we can use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c).

2.4 Written Description of the SPI Protocol

According to the lab documents, 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). MOSI is a data signal which stands for master-out slave-in, which transmits data from the Nios II -> MAX3421E – synchronous to the CLK. MISO is a data signal which stands for master-in slave-out, which transmits data from the MAX3421E -> Nios II – also synchronous to the CLK. SS stands for slave select, which allows a specific slave device to be selected when SS is low. Multiple slave devices may share the same SPI bus by sharing MOSI, MISO and CLK lines, but SS must be a unique connection between the master device and each slave device.

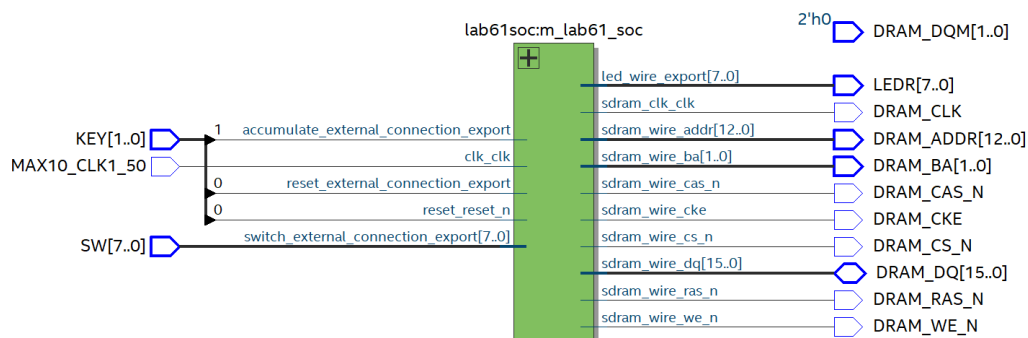
2.5 Functions of C Codes

C codes during this lab serve as the functionality of the LED and the ball movement. More details of what we have added in the c codes are written below in report section 7.

2.6 VGA Operation

The screen is just a 640x480 pixels matrix. During this lab, we used a simple color mapper and VGA controller to draw the shapes on the screen. These shapes include the ball and the background. Moreover, the color mapper needs to have as inputs the horizontal and vertical position counters, and maps output color either to foreground color (e.g. red) or background color (e.g. white).

3 Top Level Block Diagram



Lab 6 Week 1 Top Level Block Diagram

3. Module: VGA_controller.sv

- Inputs: Clk, Reset
- Outputs: hs, vs, pixel_clk, blank, sync, DrawX, DrawY
- Description: This module generate the Horizontal Sync(HS) and the Vertical Sync(VS) signals.
- Purpose: Generate the coordinates corresponding to HS and VS, so that over modules like Color_Mapper() can draw to the coordinate and display it on screen.

4. Module: lab62.sv

- Inputs: MAX10_CLK1_50, KEY, SW, DRAM_DQ, ARDUINO_IO, ARDUINO_RESET_N
- Outputs: LEDR, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM_CLK, DRAM_CKE, DRAM_ADDR, DRAM_BA, DRAM_LDQM, DRAMUDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGS_VS, VGA_R, VGA_G, VGA_B, ARDUINO_IO, ARDUINO_RESET_N
- Description: this module is the top-level file of lab 6.2. It basically connects all the components together to run the design.
- Purpose: serves as an interface for connecting FGPA and the c code.

5. Module: lab62soc.v

- Inputs: clk_clk, key_external_connection_export, reset_reset_n, sdram_wire_dq, spi0_MISO, usb_gpx_export, usb_irq_export
- Outputs: hex_digits_export, keycode_export, leds_export, sdram_clk_clk, sdram_wire_addr, sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export
- Description: this file is generated by the platform designer for all the hardware components used in the lab 6.2 design. It includes all the connections of the wires that we made in the platform designer.
- Purpose: provide all the wires and hardware for the top-level module to use and interact.

6. Module: HexDriver.sv

- Inputs: In0
- Outputs: Out0
- Description: Each hex value has its unique corresponding 7-bit binary value. (7 bits because we modify the HEX driver to not display decimal point by adding another bit).
- Purpose: Hex module is used to output the hex value on the board screen.

7. Module: Color_Mapper.sv

- Inputs: BallX, BallY, DrawX, DrawY, Ball_size
- Outputs: Red, Green, Blue
- Description: This module determines the RGB values for the ball and the background.
- Purpose: serves as a color setting for the ball and the background shown on the screen.

8. ball.sv

- Inputs: Reset, frame_clk, keycode
- Outputs: BallX, BallY, BallS
- Description: this module controls the movement of the ball after pressing the keyboard as well as the ball bouncing when reaching a boundary.
- Purpose: Basically, this is the major operation that we are going to demo through our lab6.2 that the ball moves horizontally and vertically through our press (w, a, s, d) on the keyboard.

5 Lab 6.2 Extra Credit

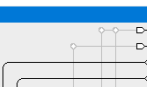
```
/******debugging code for EC*****  
/******fix the bug that the ball will go out of boundary when spamming keycodes*****  
/******3/23/2021*****  
  
    if (Ball_Y_Pos + Ball_Y_Motion >= Ball_Y_Max)  
        Ball_Y_Pos <= Ball_Y_Max-Ball_Size;  
    else if (Ball_Y_Pos + Ball_Y_Motion <= Ball_Y_Min)  
        Ball_Y_Pos <= Ball_Y_Min+Ball_Size;  
    else  
        Ball_Y_Pos <= (Ball_Y_Pos + Ball_Y_Motion); // Update ball position  
  
    if (Ball_X_Pos + Ball_Size + Ball_X_Motion >= Ball_X_Max)  
        Ball_X_Pos <= Ball_X_Max - Ball_Size;  
    else if (Ball_X_Pos + Ball_X_Motion <= Ball_X_Min)  
        Ball_X_Pos <= Ball_X_Min + Ball_Size;  
    else  
        // Ball_Y_Pos <= (Ball_Y_Pos + Ball_Y_Motion); // Update ball position  
        Ball_X_Pos <= (Ball_X_Pos + Ball_X_Motion);
```

Before these lines of codes, we found that the ball acted incorrectly when reaching the bound. For instance, if we keep pressing the "D" button and the ball already reaches the rightmost bound, the ball ought to keep hitting the wall and rebounding. However, the origin code lets the ball immerse into the wall and go through the wall after a few seconds. This happens because when the ball keeps move right, its x coordinate will keep increasing until it overflows. As a result of the overflow, the x coordinate value will restart from 0, which means it would come out of the left on the display.

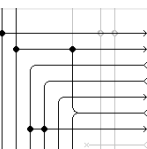
In order to fix this bug, we adjusted the code and add more if conditions. Once the ball reaches the boundary of the screen, we just forcefully reset its position to just inside the screen. Through this way of checking of hard-coding coordinates, we successfully showed the TA during demo that we have solved this boundary bug.

6 System Level Block Diagram


6.1 Modules shared in Lab 6.1 and 6.2

<input checked="" type="checkbox"/>		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			

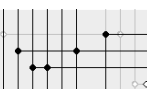
This is a 50MHz clock and it is connected to each other modules to ensure them working.

<input checked="" type="checkbox"/>		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]		
		debug_reset_request	Reset Output	Double-click to	[clk]		
		debug_mem_slave	Avalon Memory Mapped ...	Double-click to	[clk]		
		custom instruction master	Custom Instruction Ma...	Double-click to	[clk]		
						IRQ 0	IRQ 31
						# 0x0000_1000	0x0000_17ff

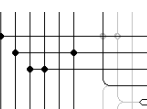
This is an economic version of the processor that we used in this lab. This module handles all the instructions.

<input checked="" type="checkbox"/>		onchip_memory2_0	On-Chip Memory (RAM o...				
		clk1	Clock Input	Double-click to	clk_0		
		s1	Avalon Memory Mapped ...	Double-click to	[clk1]	# 0x0000_0000	0x0000_000f
		reset1	Reset Input	Double-click to	[clk1]		

This module serves as a 16-byte RAM. For most designs, we will save valuable on-chip memory and instead execute NIOS II programs from the DRAM, however, we are instantiating a small on-chip RAM as a placeholder block.

<input checked="" type="checkbox"/>		sdram	SDRAM Controller Inte...				
		clk	Clock Input	Double-click to	sdram...		
		reset	Reset Input	Double-click to	[clk]		
		s1	Avalon Memory Mapped ...	Double-click to	[clk]	# 0x0800_0000	0x0bff_ffff
		wire	Conduit	sdram_wire			

We will use the off-chip SDRAM to store the software program since the on-chip memory has limited storage capacity. SDRAM cannot be interfaced to the bus directly, as it has a complex row/column addressing scheme and requires constant refreshing to retain data. We will use an SDRAM controller IP core to interface the SDRAM to the Avalon bus.

<input checked="" type="checkbox"/>		sdram_pll	ALTPLL Intel FPGA IP				
		inclk_interface	Clock Input	Double-click to	clk_0		
		inclk_interface_reset	Reset Input	Double-click to	[inclk...		
		pll_slave	Avalon Memory Mapped ...	Double-click to	[inclk...	# 0x0000_01c0	0x0000_01cf
		c0	Clock Output	Double-click to	sdram...		
		c1	Clock Output	sdram_clk	sdram...		

This module is a PLL component that provides the required clock signal for the SDRAM chip because the SDRAM requires precise timings, and the PLL allows us to compensate for clock skew due to the board layout.

<input checked="" type="checkbox"/>		sysid_qsys_0	System ID Peripheral ... Clock Input Reset Input Avalon Memory Mapped ...	Double-click to Double-click to Double-click to	clk_0 [clk] [clk]	# 0x0000_01e8	0x0000_01ef
-------------------------------------	--	---------------------	--	---	-------------------------	---------------	-------------

we add a system ID checker to ensure the compatibility between hardware and software. This module gives us a serial number (we will assign it to 0), which the software loader checks against when we start the software. This prevents us from loading software onto an FPGA which has an incompatible NIOS II configuration (or an FPGA without a NIOS II at all). For example, if we had added a new NIOS II peripheral and forgot the regenerate/reprogram the FPGA, this block would prevent us from trying to load software from the old configuration onto our incompatible NIOS II.

6.2 Modules Only Included in Lab 6.1

<input checked="" type="checkbox"/>		led	PIO (Parallel I/O) In... Clock Input Reset Input Avalon Memory Mapped ... Conduit	Double-click to Double-click to Double-click to	clk_0 [clk] [clk]	#	
-------------------------------------	--	------------	---	---	-------------------------	---	--

This is for the led blinking and accumulating. The module has 8 bits to control exactly 8 LEDs on our board. The led peripheral only needs access to the data bus because it is the output and only needs to be determined on/off by the data.

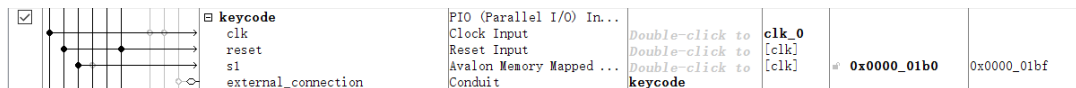
<input checked="" type="checkbox"/>		switch	PIO (Parallel I/O) In... Clock Input Reset Input Avalon Memory Mapped ... Conduit	Double-click to Double-click to Double-click to	clk_0 [clk] [clk]	# 0x0000_0080	0x0000_008f
<input checked="" type="checkbox"/>		reset	PIO (Parallel I/O) In... Clock Input Reset Input Avalon Memory Mapped ... Conduit	Double-click to Double-click to Double-click to	clk_0 [clk] [clk]	# 0x0000_0070	0x0000_007f
<input checked="" type="checkbox"/>		accumulate	PIO (Parallel I/O) In... Clock Input Reset Input Avalon Memory Mapped ... Conduit	Double-click to Double-click to Double-click to	clk_0 [clk] [clk]	# 0x0000_0060	0x0000_006f

These three are all simple PIO blocks that can control the switches and two buttons on the board. The switch has 8 bits since there are 8 switches on the board. Either the reset or the accumulate button has one bit since they only control high and low for reset and accumulation.

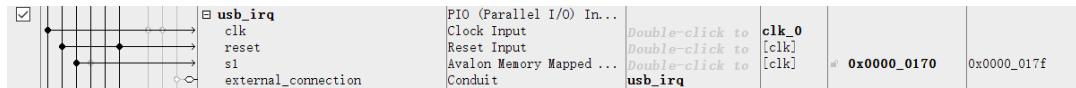
6.3 Modules Only Included in Lab 6.2

<input checked="" type="checkbox"/>		jtag_uart_0	JTAG UART Intel FPGA IP Clock Input Reset Input Avalon Memory Mapped ... Interrupt Sender	Double-click to Double-click to Double-click to	clk_0 [clk] [clk]	# 0x0000_01e0	0x0000_01e7
-------------------------------------	--	--------------------	---	---	-------------------------	---------------	-------------

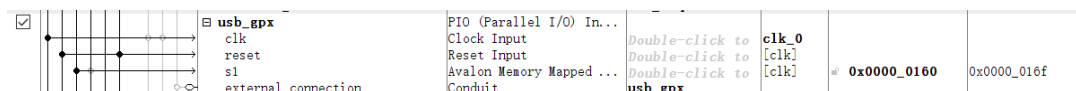
The JTAG UART peripheral helps us to use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c). It gives us the ability to use console (printf) commands from the Nios II which go through the programming cable via USB.



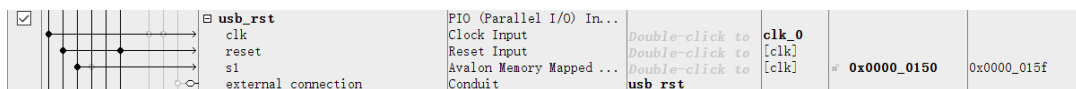
This is an output of 8 bits which outputs the keycodes from the keyboard.



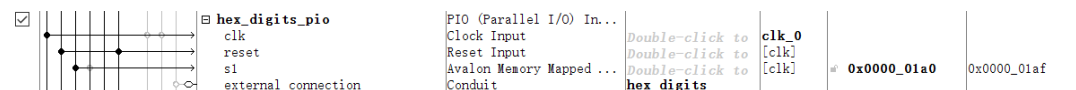
This is one of the PIO(Interrupt Request) of the USB component, which is necessary to connect to the USB chipset(MAX3421E). [Details referred to the MAX3421E Datasheet\(Page 5\)](#)



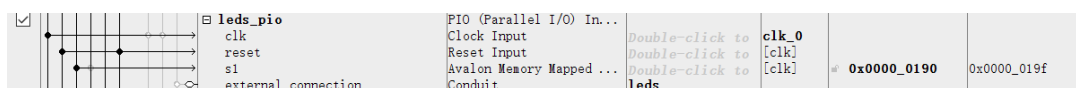
This is one of the PIO of the USB component(Genral Purpose Multiplex Output), which is necessary to connect to the USB chipset(MAX3421E).



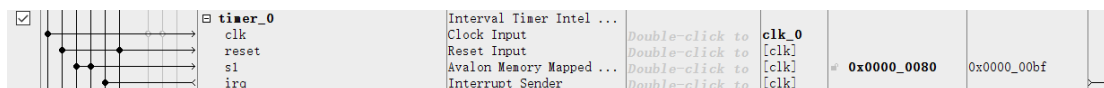
This is one of the PIO(Reset) of the USB component, which is necessary to connect to the USB chipset(MAX3421E). [Details referred to the MAX3421E Datasheet\(Page 4\)](#)



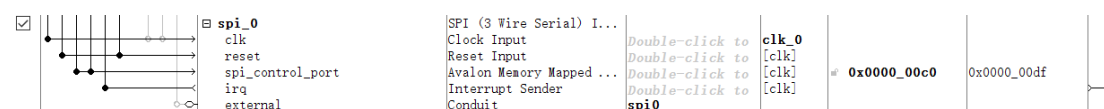
This is our HEX driver on the DE10 board.



This is the LEDs on the DE10 board.



This module is the Interval Timer Intel FPGA IP. The timeout period is 1ms and has a timer counter size of 64.



The SPI port peripheral connects the provided USB host code to the SPI driver.

7 Software Description

7.1 Accumulator

```
int main()
{
    volatile unsigned int *LED_PIO = (unsigned int*)0x90; //make a pointer to access the PIO block
    volatile unsigned int *sw = (unsigned int*)0x80; //make a pointer to access the sw block
    volatile unsigned int *reset = (unsigned int*)0x70;
    volatile unsigned int *accumulate = (unsigned int*)0x60;

    volatile int flag = 1;
    int i;
    *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

        // Remember buttons are active low
        if(*reset == 0){
            *LED_PIO = 0;
            flag = 1;
        }
        // Flag ensures only adding once when pressed down
        // if (*accumulate == 0){*LED_PIO += *sw;}
        else if(*accumulate == 0 && flag == 1){
            *LED_PIO += *sw;
            flag = 0;
        }
        else if (*accumulate == 1 && flag == 0){
            flag = 1;
        }
    }
    return 1; //never gets here
}
```

This function will reset(turn off all LEDs) when the reset button(active low) has been pressed. And it will accumulate when the accumulate button(active low) is pressed. The "flag" is just to ensure that each press will only lead to one addition. The overflow problem is automatically handle by the length of LED_PIO(8 bits) variable defined in the hardware designer.

7.2 The USB/SPI in Lab 6.2

```
void MAXreg_wr(BYTE reg, BYTE val) {
    alt_u8 wdata_ptr[2];
    wdata_ptr[0] = reg + 2;
    wdata_ptr[1] = val;
    alt_avalon_spi_command(SPI_0_BASE, 0, 2, wdata_ptr, 0, 0, 0);
}

BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    alt_u8 wdata_ptr[nbytes+1];
    wdata_ptr[0] = reg + 2;
    for(int i=0; i < nbytes; i++){
        wdata_ptr[i+1] = data[i];
    }
    alt_avalon_spi_command(SPI_0_BASE, 0, nbytes+1, wdata_ptr, 0, 0, 0);
}

BYTE MAXreg_rd(BYTE reg){
    alt_u8 reg_val_ptr[1];
    reg_val_ptr[0] = reg;
    alt_u8 read_val[1];
    alt_avalon_spi_command(SPI_0_BASE, 0, 1, reg_val_ptr, 1, read_val, 0);
    return read_val[0];
}

BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    alt_u8 reg_val_ptr[1];
    reg_val_ptr[0] = reg;
    alt_avalon_spi_command(SPI_0_BASE, 0, 1, reg_val_ptr, nbytes, data, 0);
    return (data + nbytes);
}
```

To enable the communication between the NOIS-II processor and the USB-chip, we wrote 4 functions with the help of the provided SPI API function. Note that in the 2 write() functions, we did "reg + 2" to change the second LSB from 0(read mode) to 1(write mode). and since the passed in "reg" argument already has the pre-shifted register index, we can directly use its value.

```
1  int alt_avalon_spi_command(
2      alt_u32 base,           // Base SPI Device
3      alt_u32 slave,         // Slave SPI Device
4      alt_u32 write_length,
5      const alt_u8 * write_data, // ptr to data being written
6      alt_u32 read_length,
7      alt_u8 * read_data,      // ptr to data being read
8      alt_u32 flags
9  );
```

8 INQ Questions

(a) **What are the differences between the Nios II/e and Nios II/f CPUs?**

E stands for "economics", so the f version has more features than the e-version, but if we use e-version for a simple project, it can save loads of unnecessary energy consumption.

(b) **What advantage might on-chip memory have for program execution?**

The main advantage on-chip memory has is faster access compared to the off-chip memory. Since it's on the same board as the CPU, there's less latency for the CPU required to fetch data from memory. On-chip memory is usually even faster than cache, but one trade-off is that it has relatively small storage size compared to the off-chip memory.

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

Von-Neumann architecture uses the same bus and the same memory for both data and instruction. Pure Harvard architecture uses separate bus and separate memory for data and instruction. Modified Harvard architecture uses separate bus but the same memory for data and instruction. NOIS-II is a modified Harvard since it has separated data and instruction buses while only one memory unit type, which can be viewed using the platform designer.

(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?**

Because the led is an output that only needs to be on/off based on the data (1/0). However, on-chip memory always needs to do some complicated operations, so it also has to access the program bus as well as the data bus.

(e) **Why does SDRAM require constant refreshing?**

SDRAM has a single capacitor and a single transistor. The capacitor will leak off over a period of time, so it needs constant refreshing to prevent data loss.

(f) **Justify how you came up with 512 Mbit**

Data Width: 16;

Number of Rows: 13;

Number of Columns: 10;

Number of Chip Selects: 1;

Number of Banks: 4.

So $16 \times (2^{13}) \times (2^{10}) \times 1 \times 4 / (2^{20}) = 512 \text{ Mbit}$.

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

Each data transfer can transfer 16 bits because data width is 16 bits for our SDRAM.

According to the datasheet, each access time would be 5.4ns. To calculate the transfer rate, simply divide the 2 numbers will get us the correct value.

$$\frac{16bits}{5.4ns} = 2.963Gbps.$$

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

When SDRAM runs too slowly, it meets the same issues as not doing constant refreshing.

This low frequency will cost data loss.

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

Because we want to ensure the meta-stability. This -1ns new clock enables the SDRAM to read the high value from the clock correctly and do the right operation. Otherwise, it might try to read the value right on the clock edge(E.g. during T_hold), which will not get the correct data. The amount of the delay(1ns) is determined by the latency of SDRAM CAS, according to the Intel manual.

- (j) **What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?**

It starts from 0x08000000. We do this step to prevent the base address got changed. This is very important because here is the starter of our program, and we want to ensure a starting point if we want to debug.

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

The "volatile" tells the compiler always to assume this value might change. The while statements are for an infinite loop, since 1 + 1 will never reach 3. The two for loops are just for the software delay, since the program runs so fast, and if we don't set any delay between the led operations, we cannot observe the blinking. Line 13 is making LED to be on while line 16 is making it to be off. Combining these two code together, we are able to see the led blinking.

(l) Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean?

- **.bss**: it contains all global variables and static variables. Those variables are initialized to zero or do not have any specific value in the code. E.g. `int a;`
- **.heap**: the heap memory is allocated by the programmer that a new object goes into the heap. e.g. `malloc Cube();`
- **.rodata**: a variable is initialized but is read-only. E.g. `const a = 10;`
- **.rwdata**: a variable is initialized and it can either read or write. E.g. `int a = 7;`
- **.stack**: stack has a LIFO structure. Every time a function is called, the machine allocates some stack memory for it. When a new local variable is declared, more stack memory is allocated for that function to store the variable. Such allocations make the stack grow downwards. After the function returns, the stack memory of this function is deallocated, which means all local variables become invalid. The allocation and deallocation for stack memory are automatically done. E.g. `int c = 12345;`
- **.text**: it contains executable instructions and is generally read-only with a fixed size. An example would be a segment of code.

9 Design Resources and Statistics

9.1 Statistics for Lab 6.1

LUT	2757
DSP	0
Memory (BRAM)	36,864 / 1,677,312 (2 %)
Flip-Flop	1815
Frequency	74.08MHz
Static Power	96.61mW
Dynamic Power	46.66mW
Total Power	197.94mW

9.2 Statistics for Lab 6.2

LUT	4481
DSP	0
Memory (BRAM)	46,080 / 1,677,312 (3 %)
Flip-Flop	2987
Frequency	70.45MHz
Static Power	96.51mW
Dynamic Power	59.85mW
Total Power	177.24mW

10 Conclusion

10.1 Bugs Encountered

We have encountered the case whether the whole program just froze after we pressed and sending one keycode. According to Professor Chen, this might happen because some keyboards mis-report their maximum reporting rate, and the USB driver doesn't handle the 'NACK' condition very gracefully. We solved this using the below trouble shooting method.

Troubleshooting

Stuck in "reset timeout!" loop:

touch around on the DE10 Lite Shield (NOT THE FPGA) until it works

Program freezes after printing keycode:

add "usleep(1000)" between MAX3421E_Task and USB_Task inside main.c::main()
add "usleep(1000)" between MAXreg_wr and XferInTransfer inside of HID.c::kbdPoll()

Ball won't respond to WSAD presses:

change 0x8002000 to KEYCODE_BASE inside main.c::setKeycode

10.2 Functionality of Our Design

Our design either on the accumulating LED or the ball movements on VGA is fully functional and we got all points during the demo in Week1 and Week2. We also fixed the bug of the ball when it bounces from the wall mentioned in the lecture. All provided programs can be executed with correct behavior, details of which can be found in the above report.

10.3 Suggestions to Lab-Manual and Given Document

Honestly, we spent hours correcting the pin assignments during these two-week-lab. There was some name mismatching from the given file, and since we started the lab very early, we don't know how to import the pin assignments at first. When doing the complication on the Quarters, it keeps warning us to change the voltage from 3.3 to 2.5 or 2.5 to 3.3 for different pins, which consumed us loads of time.