ECE 385

Fall 2021

Experiment #6

NIOS II SoC with peripherals

Steven Dimov & Owen Shin

Section ABE

TA: Abigail Wezelis

Introduction & Operation

The NIOS II processor is a System-on-Chip processing unit that we implement onto the FPGA in this lab. To create a minimal NIOS II design, we used the platform designer tool to create an IP (Intellectual Property) block to instantiate SDRAM onto the design and add a Parallel I/O to manipulate LED output and read data from switches. Once this design was completed, c code was compiled and run on the FPGA's NIOS II SoC to perform various operations. The first operation was to simply be able to output a blinking light onto the first LED of the FPGA, on its own. The second operation was to be able to read an 8-bit number from the switches and executing the buttons to accumulate and display onto the LEDs. For the second half of the lab, additional modules are instantiated to accommodate the USB keyboard input, the MAX3421E USB controller, and the VGA display output to be able to run a program onto the NIOS II utilizing these peripherals. This program outputs a moving orange ball on a blue gradient background onto the VGA display also allowing the keyboard input's WASD keys to control the direction of the ball's velocity (W-up, A-left, S-down, D-right). The ball is limited to traveling within the display's borders, flipping directions when bumping into a border. However, the provided lab code was flawed in the sense that the vertical borders could be bypassed when using the keyboard to move vertically. We were able to fix this with the incentive of extra credit.

VGA:

This lab utilized both the VGA and keyboard peripherals. The FPGA has a built in VGA port for displaying graphics to an output. The VGA standard is analog, consisting of the following signals, Ground, Red, Green, Blue, Horizontal-Sync, Vertical-Sync. The RGB signals are driven by voltages that correspond to the amount of that color outputted at the location of that pixel, which gives a range of different colors when all three are modulated. The H-sync signal corresponds to the signal that modulates when the scanline is shifted to the next horizontal line below it. The V-sync signal modulates to the next frame of information, when the last horizontal scanline is drawn.

USB:

In order to utilize and connect to the USB keyboard from the FPGA, an I/O shield is added onto the FPGA's Arduino pins. Although the USB on our FPGA has the same pins, (+5v, GND, D+, D-), as any other standard USB port, we cannot use it for the keyboard beyond reasons of the form factor. This is because the USB type-B port on the FPGA is already used for powering and communicating data between the computer and the FPGA, as well as being only a slave port. Including the Arduino-style shield, we are provided with a female USB port that functions as a host/master, being able to receive scancodes from the keyboard, which then are interpreted to serve the greater purposes of our lab's program. This is done through the SPI protocol, which we use to configure our DE-10 lite and USB keyboard compatibility. The nature of our USB keyboard is an interrupt, as the data of a HID device needs immediate attention and handling since it wouldn't make sense to prioritize it according to other modes of USB protocols. For example, a USB keyboard would not be very functional if it were prioritized after a massive file transfer, which would delay one's keystrokes beyond feasible use. Extensive time priority such as in the Isochronous USB protocol would not be very feasible as well, since that would allow keystrokes to be lost to oblivion if they are delayed by any amount, but would make sense when implemented in playing music or video, where delay or variation in speed is not tolerable and losing data to keep the tempo would be acceptable. So instead, the HID protocol allows a good comprise where no keystrokes are lost and are prioritized over other data by sending interrupt requests and receiving

report descriptors to and from the keyboard and the MAX3421E in the form of scancodes, as listed below, where up to 6 different keycodes would be handled at once:

Byte	Description
0	Modifiers Keys
1	Reserved
2	Keycode 1
3	Keycode 2
4	Keycode 3
5	Keycode 4
6	Keycode 5
7	Keycode 6

The keycodes we are looking for are 26, 4, 22, 7, which correspond to
W, A, S, D, respectively.

Written Description and Diagrams of NIOS-II System

The SPI protocol is a master-slave digital communication protocol where one byte is transmitted serially at a time. Usually, there are three signals shared between the master device and all slave devices: SCLK, MOSI, and MISO. There is also an SS' signal between the master and each slave device. SCLK is the source clock, provided by the master. The rate at which bits are communicated is based on this clock. The MOSI signal stands for master-out slave-in, and the master device transmits data over this line. The MISO signal stands for master-in slave-out, and the relevant slave device transmits data over this line. The SS' signal for each slave device selects the desired device, and is configured based on whether the device's select is active-high or active-low.

After a slave device is selected, the master device begins cycling the clock and data is transmitted from master to slave over the MOSI line and from slave to master over the MISO line. Though other slave devices may share these lines, those which are not selected through SS' do not read or write any data. One bit per clock cycle is exchanged until the clock signal stops cycling or the slave device is deselected.

In the case of the MAX3421E IC in full duplex mode, the first byte of information exchanged once SS' is low and the clock is cycling is an instruction byte from the master and a status byte from the IC. The instruction bit contains an address for a register to read or write, a read'/write bit, and an acknowledge bit for peripheral mode, which was not used. The status that comes from the IC can also be read from one of the registers and is ignored in this lab. After a register is selected by the master device, if the read'/write signal was low in the instruction then bytes of data from registers are transmitted to the master over the MISO line. If the read'/write signal was high then bytes of data from the master over the MOSI line are written to the relevant registers. If the clock/select signals are not driven inactive after the second byte of data, the address of the register to read or write is incremented unless R20 or R31 is selected.

Purpose of functions:

The first function we made was the MAXreg_wr function, which takes as arguments a register address and a byte to write to that register. The function creates an array of bytes. The first byte is the instruction for the MAX3421, which is the register address with the second bit high to indicate a write. The second byte is the value to be written to that register. The function then calls the alt_avalon_spi_command function for the SPI core in the soc module. This function takes the base address for the SPI core, a byte to represent which slave to select (the MAX3421 in all cases), a write length bit of two in this case, a pointer to the array of two bytes, a read length byte of zero (as we aren't reading anything), an irrelevant read bytes pointer, and a byte for special flags for the instruction, which was null since none are used. The number of bytes read is returned from the SPI core function, and the function we made tests to make sure the number isn't negative, which would indicate some error. After the write, nothing more needs to be done, so the function type is void.

The second function was the MAXbytes_rd function, which takes as arguments a register address, a number of bytes to write, and a pointer to an array of bytes to write. Our function concatenates the register address with the write bit high and the array of bytes into a new array, and then calls the function for the SPI core. The same SPI core base address, slave select byte, read length of zero, don't-care read bytes pointer, and null flag byte are used from the MAXreg_wr function. However, the write length is one greater than the length of the given array length, and the pointer is for the array created for this function. The function checks the return value from the SPI core to make sure it isn't negative and returns void, since only a read was done.

The third function was the MAXreg_rd function, which only takes a register address. It creates a one-byte array with the register address, which has the second bit low to indicate a read. It creates another byte variable, which is empty. It calls the SPI core function with a write length of one, a write pointer pointing to the instruction array, a read length of one, and a read pointer pointing to the new byte variable. All other arguments are the same as in previous functions. Also, the returned read length is checked to make sure it isn't negative. The byte from the read is returned.

The fourth function was the MAXbytes_rd function, which takes a register address, a number of bytes to read, and a pointer to an array of bytes. A one-byte array with the register address and the second bit low is created. The SPI core function is called with a write length of one, a write pointer pointing to the register address array, a read length of the number of bytes to read, and a read pointer of the given pointer to the array of bytes. The other arguments are the same as in previous functions. The read length is checked to make sure it isn't negative, and the address following the end of the given array is returned.

The MAXreg_wr function writes a single byte to the desired register. The MAXbytes_wr function writes to the desired register and the addresses that follow. The MAXreg_rd function reads from the desired register, while the MAXbytes_rd function reads from the desired register and the addresses that follow.

vi. Describe in detail the VGA operation, and how your Ball, Color Mapper, and

the VGA controller modules interact.

Top Level Block Diagram

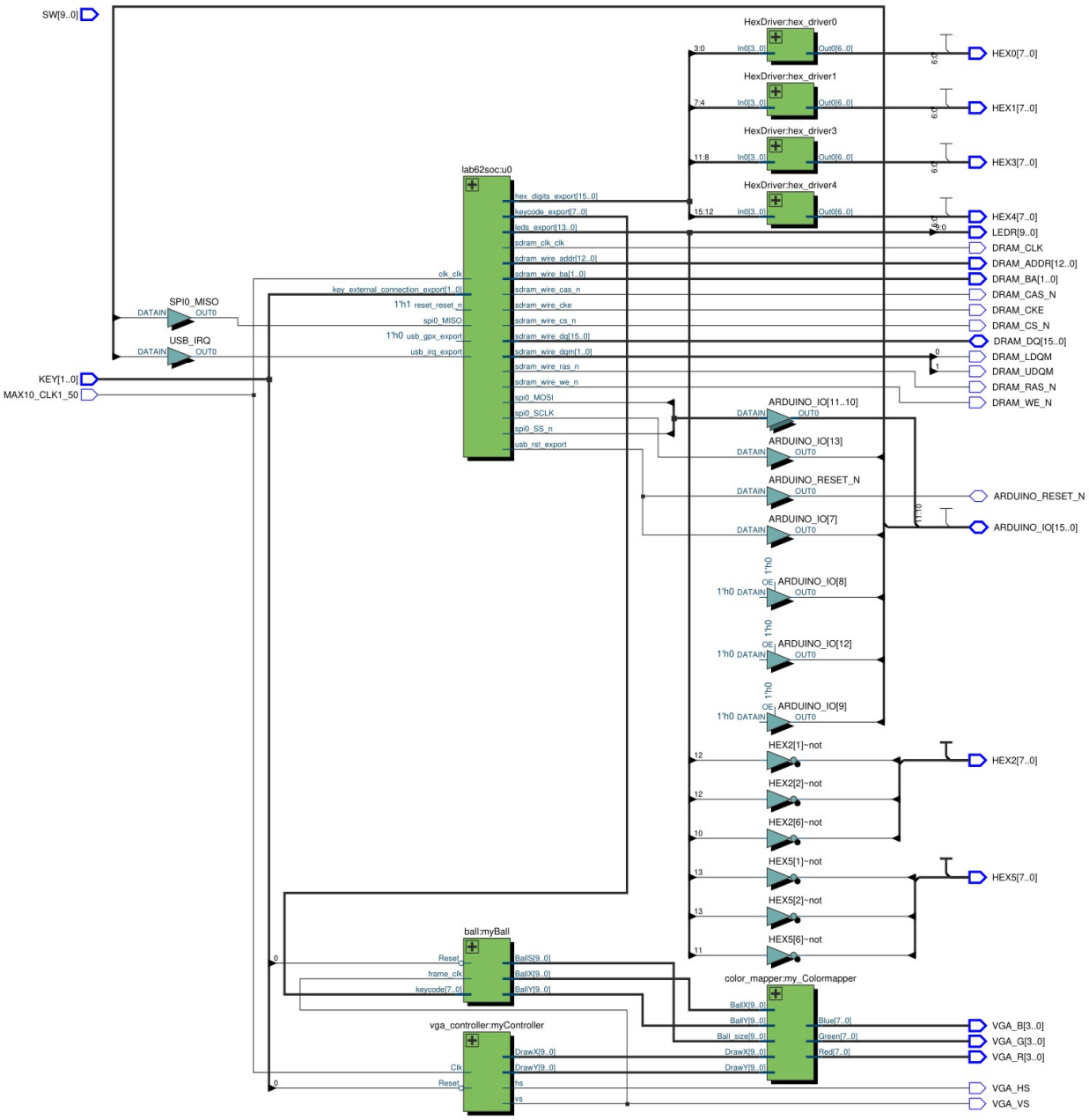

Figure : Top-level RTL diagram of lab 6, part 2. Notice that aside from sharing the same clock and reset signals, the only connection between the NIOS II module and the VGA/ball related modules is that a keycode can be sent from the NIOS II module to the ball module. Notice also that the ball, VGA controller, and color mapper modules control the outgoing VGA signals, while the NIOS II module receives the SPI signals from the IC on the FPGA's Arduino hat.

Written Description of all .sv Modules

a. A guide on how to do this was shown in the Lab 5 report outline. Do not forget to describe the Platform Designer generated file for your Nios II system! When describing the generated file, you should describe the PIO blocks added beyond those just needed to make the NIOS system run (i.e. the ones needed to communicate with the USB chip and other components). The Platform Designer view of the Nios II system is helpful here.

5. System Level Block Diagram (this is new for labs 6 & 7)

    a. The Platform Designer view of the SoC module should be found here, describe the functionality of each block (including those which are part of the SoC, such as the memories).

    b. Note that this is not trivial, as there are many components within the Platform Designer view.

    c. You can separately describe the 'core' components common to Lab 6.1 and 6.2 first, and then describe the specific modules for week 1 and week 2.

6. Describe in words the software component of the lab.

    a. One of the INQ questions asks about the blinker code, but you must also describe your accumulator.

    b. Describe the code you needed to fill in for the USB/SPI portion of the lab for Lab 6.2

Answers to all INQ & Post lab questions

Q1 (INQ.5) What are the differences between the Nios II/e and Nios II/f CPUs?

Both are 32-bit RISC CPUs, but the Nios II/e is resource-optimized, while Nios II/f is performance-optimized. Both have JTAG debugging and RAM protections, but Nios II/f has many more features, including caches, memory protection/management, and an external interrupt controller.

Q2 (INQ.7) What advantage might on-chip memory have for program execution?

It has faster access time than external memory, so you could use it as a cache (Embedded Peripherals IP User Guide 25.1). Also, non-MRAM blocks can be initialized on startup to hold constraints or boot code (Embedded Peripherals IP User Guide 25.2.1).

Q3 (INQ.7) Note the bus connections coming from the NIO II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?

It's a modified Harvard machine because data and instructions have their own busses but share an address space (Wikipedia, "Modified Harvard Architecture," "Comparisons").

Q4 (INQ.8) Why does the PIO only need the data bus and not the instruction bus?

It does not need to interpret instructions or send data; it only receives data. The on-chip memory, by comparison, may need to take a read/write instruction to know whether to read from or write to the data bus.

Q5 (INQ.8) Why does SDRAM require constant refreshing?

SDRAM uses capacitors to maintain a voltage which represents data, so if the charge in these capacitors isn't refreshed for too long it could "leak" and information will be lost. SRAM does not need refreshing (allaboutcircuits.com, "Introduction to DRAM").

Q6 (INQ.9) Why is the SDRAM 512 megabits large?

8,000 rows * 1,000 columns per bank = 8 million bits per bank

8 million bits per bank * 4 banks = 32 million bits

32 million bits * width of 16 = 512 million bits

Q7 (INQ.9) What is the maximum theoretical transfer rate to the SDRAM according to the timings given?

5.4 nanoseconds per access + 20 nanoseconds per precharge = 25.4 nanoseconds per read

15.625 microseconds active / 25.4 nanoseconds per read = 615.16 reads

615.16 reads / (15.625 microseconds active + 70 nanoseconds refreshing) = 39.19 megareads per second (or megahertz)

5.4 nanoseconds per access + 14 nanoseconds per write recovery = 25.4 nanoseconds per write

15.625 microseconds active / 19.4 nanoseconds per write = 805.41 writes

805.41 writes / (15.625 microseconds active + 70 nanoseconds refreshing) = 51.31 megawrites per second (or megahertz)

Q8 (INQ.9) Why can't the SDRAM have a clock frequency of less than 50 MHz?

If the SDRAM operates too slowly, it won't refresh frequently enough, so information will be lost.

Q9 (INQ.11) Why do we need a PLL to phase shift the clock?

This is so clock cycles for different components are in phase, so hardware delays won't cause bitwise errors or data transfers to be in different phases between components. The SDRAM clock can lead or lag the controller clock by a read/write lead or lag time (Embedded Peripherals IP User Guide 33.7.3).

Q10 (INQ.14) What address does the NIOS II start execution from? Why do we edit vectors after assigning the addresses?

The processor will start execution from the first SDRAM address (Base of SDRAM + offset of 0). We configure this after assigning addresses so we can set both reset and exception to point to the SDRAM instead of a different peripheral.

```
int main()
{
    int i = 0;
    volatile unsigned int *LED_PIO = (unsigned int*)0x40; //make a pointer to access the PIO block

    *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
}
```

Figure 22

Q11 (INQ.20) Explain what the given main.c code for lab 6 part 1 does. Specifically, what does "volatile" in line 8 do and how the set and clear functions on lines 14 and 16 work.

This code toggles the rightmost LED on the FPGA and uses for loops to delay by a set period after turning an LED on or off. The word "volatile" on line 8 indicates that the variable LED_PIO is going to be read or manipulated frequently. Lines 14 and 16 are or-equals and and-equals expressions, which set the variable on the left side to its original value or'ed or and'ed with its original value. Line 14 or-equals the LED peripheral register with the least significant bit, setting that bit high and leaving all other bits alone. Line 16 and-equals the LED peripheral register with the all but the least significant bit high (~0x01), resetting the least significant bit low and leaving the rest alone.

Q12 (INQ.21) What does each section of the SDRAM memory mean? Give an example of C code which places data into each segment.

The .bss section of SDRAM holds global and static (global or local) variables. One example of this may be a variable that is declared with the type "static int" or has been declared outside of main.c.

The .heap section can be used to store temporary information of different sizes. One example of this may be creating a binary search tree using malloc().

The .rodata section stores read-only constant variables or strings used in our program, like a variable declared with the type "const int."

We are not sure what .rwdata does.

The stack in .stack holds information relevant to function calls so the data can be popped once it is no longer relevant. This may include any variables declared in a function call or arguments used.

The .text section holds the program that the microcontroller runs, including functions and interrupt service routines. This section is read-only.

POST-LAB

LUT	3071
DSP	10
Memory (BRAM)	55,296
Flip-Flop	2355
Frequency	75.92 MHz
Static Power	96.50 mW
Dynamic Power	56.67 mW
Total Power	175.01 mW (Including I/O Power Dissipation)

Conclusion

We were eventually able to complete this week's lab with full operation by the end of the second week. However, we struggled with some issues in lab 6.1, specifically the set-up of the NIOS II processor and its memory instantiation. When we got to testing our design, the visible problem was that the LED was not blinking or turning on at all when running the code, along with having mismatched timestamps. Having thought we followed the manual word for word, we came across the same issue regardless of any changes we made to our code or to the platform design. The first area we looked at was the pin assignment, as the introduction to NIOS II tutorial uses a couple of different pins from what is provided in the .qsf file, the pins being the DRAM_DQM [1:0], seen as DRAM_LDQM and DRAM_UDQM. After realizing that the names match up in the code anyway, we went to the next possible issue which was the LED address in the platform design and that of the c code in eclipse. After seeing that they were the same address, and later fixing the timestamp issue we exhausted all possible solutions and restarted our design. Upon reconfiguration, we were successful for the first part of lab6. After we made a few changes to progress through lab 6.2, we ran into the same issue again and went to office hours several times with no insight on the issue. It was only when we discovered an intel forum online mentioning that the RESET signal is actually active low, which happened to be the root of all of our problems, as nothing was being outputted with a constantly high reset signal.