ECE 385

Spring 2019

Experiment #7

SOC with NIOS II in System Verilog

Rob Audino and Soham Karanjikar

Section ABJ, Friday 2:00-4:50

TA: David Zhang

Introduction:

The intention of Lab 7 was very similar to that of Lab 4 in that we were given a gentle introduction into a new way of generating a digital design. Specifically, Lab 7 introduced us to using Qsys and NIOS II in conjunction with Quartus and the FPGA, which we were already familiar with. Lab 7 showed us that we could use the FPGA as a platform upon which to run code that we had written in C. This is useful because it allows us to program more based off of software than the hardware-based System Verilog that we had previously used. The end goal of this lab was to use the first eight switches switches on the FPGA as the inputs to an accumulator, and for that accumulator to roll over once it summed past 255 (essentially carrying out the operation 'sum modulus 256'). In addition, the running sum in the accumulator would be displayed on the Hex display. For example, if the sum ended up being 257, the display would show a numerical value of 1.

Written Description and Diagrams of NIOS-II System:

Summary of Operation:

Our design began with the few basic inputs from the FPGA. We knew that we needed to have an accumulator (adding) button and a reset button, as well as the 8 switches as the numerical inputs to the accumulator. Since 2^8 = 256, having these 8 switches would allow the user to input any value from 0 to 255. We also had to use several PIO blocks in order for our design to work, since this was the only way to get the NIOS code to work with the FPGA. The switches and the LED's each required an 8 bit wide PIO and the Reset and Accumulate buttons each required a 1 bit wide PIO.

In C, we created a main.c file in order to poll for any pressing of keys. When a key is pressed, the value on the switches is added to the value currently displayed on the LED's, and the final sum is also displayed on the LED's. When they key is pressed down, the code runs in a loop in order to preserve the contents of the sum that is displayed on the LED's. This loop is exited when the accumulate or reset keys are pressed. If the accumulate button is pressed, the value presently in the switches is added onto the value currently in the LED's, and the new sum is displayed in the LED's. If the reset button is pressed, the LED's would be cleared. In order to account for overflow, we simply used the modulus operator for 256. For example, if our sum was 258, our end result would be 258%256 = 2.

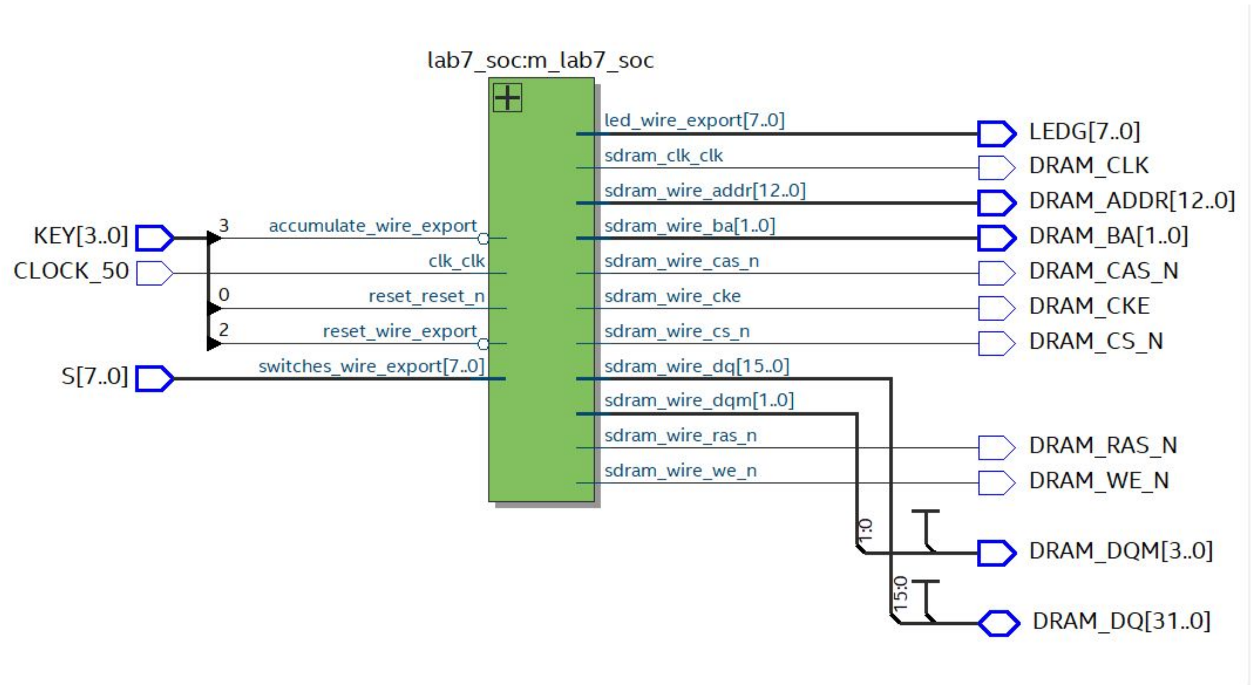
Written Description of all .sv Modules:

Module: lab7
Inputs: CLOCK_50, [3:0] Key, [7:0] S
Outputs: [7:0] LEDG, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
Description: This module is the top level module for the lab 7 project. It integrates NIOS II with Quartus and the FPGA

Purpose: We need this module because it allows us to adapt the FPGA to be able to use the C code that we wrote in Eclipse.

Module: lab7_soc
Inputs: accumulate_wire_export, clk_clk, reset_reset_n, reset_wire_export, [7:0] switches_wire_export
Outputs: [7:0] led_wire_export, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [1:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n
Description: This module completes the connections from platform designer wires to other modules.
Purpose: Generated by platform designer, this module creates an interface between peripherals, software and hardware. Anything added in platform designer is available in this module.

Top Level Block Diagram



Answers to all 11 INQ Questions:

1. Q:What advantage might on-chip memory have for program execution?
A: Accessing on-chip memory doesn't require processing through all of the buffers, multiplexers and other components that off-chip memory does. This means that on-chip memory has much faster read and write times than off-chip memory.

2. Q: Note the bus connections coming from the NIOS II; is it a Von Neumann, “pure Harvard”, or “modified Harvard” machine and why?
A: Since the fetch and store operations in a Von Neumann machine share a common bus, they cannot operate at the same time, meaning that this machine is not a Von Neumann. In addition, it cannot be a Pure Harvard machine since the Pure Harvard machine stores instructions and data separately. Because our machine can fetch and store at the same time, and stores instructions and data together, it must be a Modified Harvard machine.

3. Q: 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?
A: Since the LED is completely separate from anything that is going on in the program, it only needs the data bus. The LED’s simply display the data result of the operation. However, the on-chip memory requires knowledge of what data to send where, what data to receive, and what to do with the data.

4. Q: Why does SDRAM require constant refreshing?
A: SDRAM requires constant refreshing because of the nature of its storage. SDRAM is made up of capacitors that slowly discharge over time, and as a result need to be periodically refreshed with charge so that they maintain the correct values. If they weren’t getting refreshed, all of the charge (and therefore data) would be lost.

5. Make sure this is consistent with your above numbers; you will need to justify how you came up with 1 Gbit to your TA.

SDRAM Parameter	Short Name	Parameter Value
Data Width	[width]	16
# of Rows	[nrows]	13
# of Columns	[ncols]	10
# of Chip Selects	[ncs]	1
# of Banks	[nbanks]	4

6. Q: What is the maximum theoretical transfer rate to the SDRAM according to the timings given?
A: The maximum theoretical transfer rate is 720 MB/s.

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

A: If the SDRAM is run too slowly, the refreshing of the capacitors doesn't occur often enough to prevent any data loss or corruption.

8. Q: Make another output by clicking clk c1, and verify it has the same settings, except that the phase shift should be -3ns. This puts the clock going out to the SDRAM chip (clk c1) 3ns ahead of the controller clock (clk c0). Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.
A: Having the clock for the output of the SDRAM be slightly behind the clock for the input of the SDRAM allows for stabilization of the synchronous inputs to the SDRAM, and therefore helps ward off glitches.

9. Q: What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?
A: NIOS II starts execution from x10000000. The reason we do this step after assigning the addresses is twofold. Firstly, this step prevents overlap of memory. Secondly, this step assures that the processor knows exactly where to go after a reset or another special case, and also makes sure it knows where each item is compared to its original address.

10. Q: 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 question is referring to the blinker code.
A: Volatile is used in C to describe a variable whose value could change randomly, and is used to make sure the variable can handle any sudden changes (like compiler optimization or interrupts) and still operate correctly. The blinker code is simply two extremely long for loops nested inside an infinite while loop. The first for loop (line 13) sets the LSB (the first LED) to on by ORing the value of *LED_PIO with x1 (which is always 1), and the program remains inside this for loop for 100,000 iterations of 'i'. The next for loop sets the LSB to off by ANDing the value of *LED_PIO with 0 (which is always 0), and the program remains inside this for loop for another 100,000 iterations of 'i'. Having 100,000 iterations of 'i' within each of these for loops creates the visual delay we see between the LED turning on and turning off, essentially creating the blinking effect. The while loop with the input of 1 + 1 != 3 will run forever, since 1 + 1 = 2.

11. Q: 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, e.g. the code: const int my_constant[4] = {1, 2, 3, 4} will place 1, 2, 3, 4 into the .rodata segment.
A:

.bss	Static double var;	(similar to a global variable)
.heap	ptr (int*)malloc(sizeof(float));	(similar to a standard heap)
.rodata	const int a = 7;	(This is a constant because it is read only)
.rwdata	int a = 7;	(This is normal data, as it can be changed)
.stack	char b = randfunc();	(just a standard stack)
.text	char message = "Write anything"	(Just stores text)

12. Q: What are the differences between the NIOS II/e and NIOS II/f CPU?
A: The biggest difference between the two is that ‘e’ is resource optimized and ‘f’ is performance optimized. In addition, since ‘e’ is the resource optimized, stripped down version, ‘f’ has a lot more features than ‘e’. ‘f’ includes a hardware multiply/divide, instruction/data caches, shadow register sets, and myriad other features that ‘e’ lacks. In fact, the only features that ‘e’ has are JTAG debug and ECC RAM protection.

Post Lab Questions:

LUT	2181
DSP	0
Memory (BRAM)	10368 bits
Flip-Flop	1772
Frequency	87.09 MHz
Static Power	102.04 mW
Dynamic Power	38.12 mW
Total Power	203.38 mW

Conclusion:

This lab was a similar to lab 7 in that it introduced us to a new way of creating digital designs, this time through writing C code in NIOS and using the platform designer to integrate this C code with the FPGA itself. It was a relatively simple introduction into NIOS, but it did require a close following of a long and intricate tutorial on setting up Platform Designer and NIOS. Thankfully, we were able to get the light blinking on the first run through of the tutorial, and never had to circle back to see where or if we went wrong.

The structure of the lab didn’t leave much room for individual design variation, as a lot of the lab was made for us. Most of what was necessary for platform designer was given to us in the tutorial, and we only needed to add PIO blocks for the switches and for the accumulate and reset buttons. The modulus operator served to prevent the sum from exceeding the value of 255. The issues dealing with matching system ID’s plagued us for several days but a TA ended up helping us solve this issue in office hours. We needed to restart Quartus, NIOS, and the FPGA, and then try everything again. Thankfully, we were able to get the lab working by the second hour in the lab section. Overall, this lab was a good introduction, but I wish we were offered more tools to figure out the matching system ID’s on our own.