ECE 385

Fall 2021

Experiment #7

Lab 7
SOC with NIOS II in SystemVerilog

Guanshujie Fu 3190110666

Xiaomin Qiu 3190110717

Lab Section D231 18:00 – 21:00

TA: Yuchuan Zhu

1. Introduction

a. *Summarize the basic functionality of the NIOS-II processor running on the Cyclone IV FPGA.*

The basic functionality of the NIOS-II processor is that it allows us to compile and write higher level code like C++ other than SV with the help of System on Chip, the SoC. With the help of platform designer, we could bridge the NIOS II with the hardware on the FPGA, which allows us to write C code on the FPGA and interface with parallel input and output devices such as LEDs and switches. Through this way, we could make the LED on the FPGA blink and develop a program that accumulates numbers on switches.

2. Written Description and Diagrams of NIOS-II System

a. *Summary of Operation*

i. *Describe in words the hardware component of the lab (Describe what IP_blocks are used beyond the ones necessary for the NIOS to run. In this lab, the only additional IP blocks used are the PIO blocks).*

The hardware component of the lab includes switches, LEDs, some FPGA pins, the NIOS-II processor, and some other components. We connected the processor to SDRM controller, which describes the dimensions of the on board SDRAM, on-chip memory, which is the placeholder, and a clock phase shifter which could adjust clock on the SDRM. With the help of platform designer, we could connect the switches and LEDs to the rest of the FPGA pins, such that the platform designer can connect them as needed for the NIOS II processor. Then we could create space for those components so they can interact with codes and set memory addresses for switches, buttons, LEDs so we have access to them on the NIOS II. Through this way, we could write to the LEDs and read from the switches as well as buttons.

SDRAM:

SDRAM Controller Intel FPGA IP. Since the on-chip memory has limited storage capacity, we will use the off- chip SDRAM to store the software program. We use a SDRAM controller IP core to interface the SDRAM to the Avalon bus.

SDRAM_PLL:

ALTPLL Intel FPGA IP. The PLL is a phase shifter (-3ns to the clock source) due to the board layout and a second clock with compensation is sent to drive SDRAM.

CLK_0:

It is the clock source with frequency 50MHz that generates clock signals for all modules.

NOIS2_GEN2_0:

NIOSII/e processor

ONCHIP_MEMORY2_0:

On-Chip Memory Intel FPGA IP. A 16-byte writable RAM, which is a small on-chip RAM.

LED:

PIO (Parallel I/O) Intel FPGA IP as 8-bit output.

SWITCHES:

PIO (Parallel I/O) Intel FPGA IP as 8-bit input.

ACCUMULATE:

PIO (Parallel I/O) Intel FPGA IP as 1-bit input.

ACCUMULATE_RESET:

PIO (Parallel I/O) Intel FPGA IP as 1-bit input.

ii. *Describe in words the software component of the lab. One of the INQ questions asks about the blinker code, but you must also describeyour accumulator.*

BLINKER CODE:

The function of this code is that it switches the LSB of the LEDs from high to low with a significant time delay such that we can see the LED change. As a result, the blinker code flips the LED on and off. This is achieved as follows.

First, we defined the address of the LED PIO such that we can communicate with actual hardware on that address. After that, by writing to the memory a 0, we cleared the 8 LEDs. Infinite while loop is used to execute the blinking and the two for loops are used as a software delay.

*"*LED_PIO |= 0x1;"* is used to ensure that the LSB is going to be a 1 which turns on the right most LED through setting the 8-bits data stored at the LED memory location is a binary 1.

*"*LED_PIO &= ~0x1;"* is used to ensure that all the LEDs are turned off through performing a bit wise AND of the LED memory with a binary 0.

ACCUMULATOR CODE:

The function of this code is that when pressing a button on the FPGA, it takes the value of 8 switches and adds their value to the stored values in the LEDs. Then the sum is displayed in the LEDs, with the LEDs overflowing at 255. This is achieved as follows.

First, we declared the switch, accumulator reset and the accumulator to store the memory location of each hardware component. To avoid multiple accumulations in one press, we declared a pause integer to ensure that each button press only executes once. The pause variable is set to high after one accumulation and set to low when the button is released. We do not add if the state variable is high, therefore preventing multiple accumulations. We have 3 if statements in the while loop: The first one means if the accumulator button is pressed, the sum is reset. The second one means if the accumulator button is pressed and the pause integer is 0 then we accumulate the amount of the switch to the sum, we then change the pause integer to a 1. The third one means if we release the accumulator button, we then change the value of the pause integer back to a 0. Finally, we stored the sum into the LED pointer to write the sum to the LED.

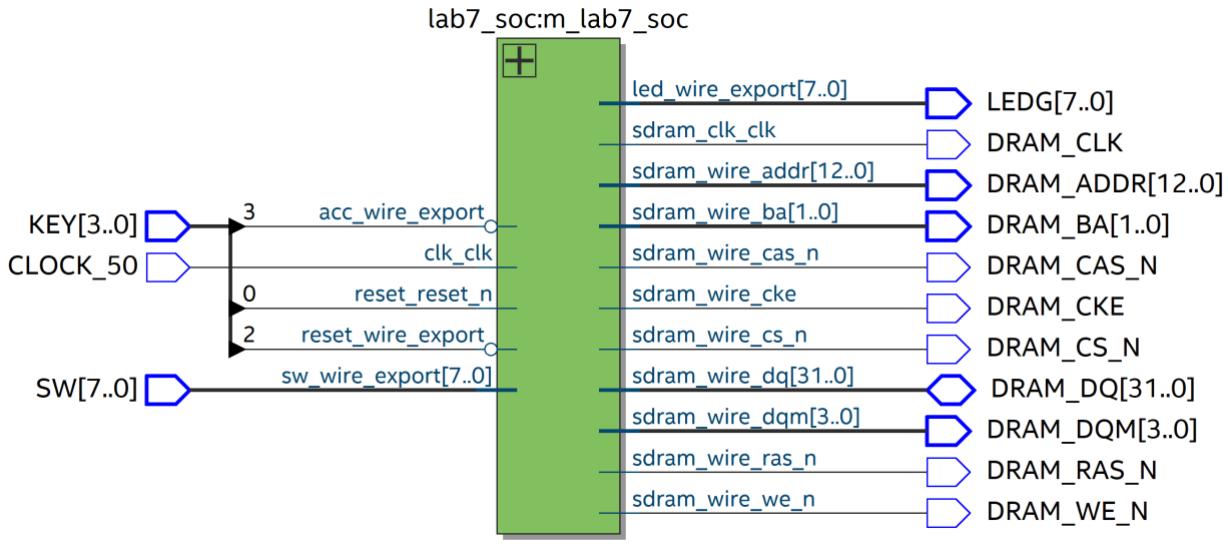
iii. *Written Description of all .sv Modules (A guide on how to do this was shown in the Lab 5 report outline. Do not forget to describe the Qsys generated file lab7_soc.v!)*

Modules	**Lab7.sv**
Inputs	input CLOCK_50, input [3:0] KEY, input [7:0] SW,
Outputs	output [7:0] LEDG, output [12:0] DRAM_ADDR, output [1:0] DRAM_BA, output DRAM_CAS_N, output DRAM_CKE, output DRAM_CS_N, inout [31:0] DRAM_DQ, output [3:0] DRAM_DQM, output DRAM_RAS_N, output DRAM_WE_N, output DRAM_CLK
Description	This is the top level of the entire lab which holds all variables and instantiates the lab7_soc.
Purpose	It is used to initialize lab7_soc and run the program and declare a SoC module and connect all the hardware led, switches, as well as buttons with the SoC module. It also connects the DRAM with the on chip memory.

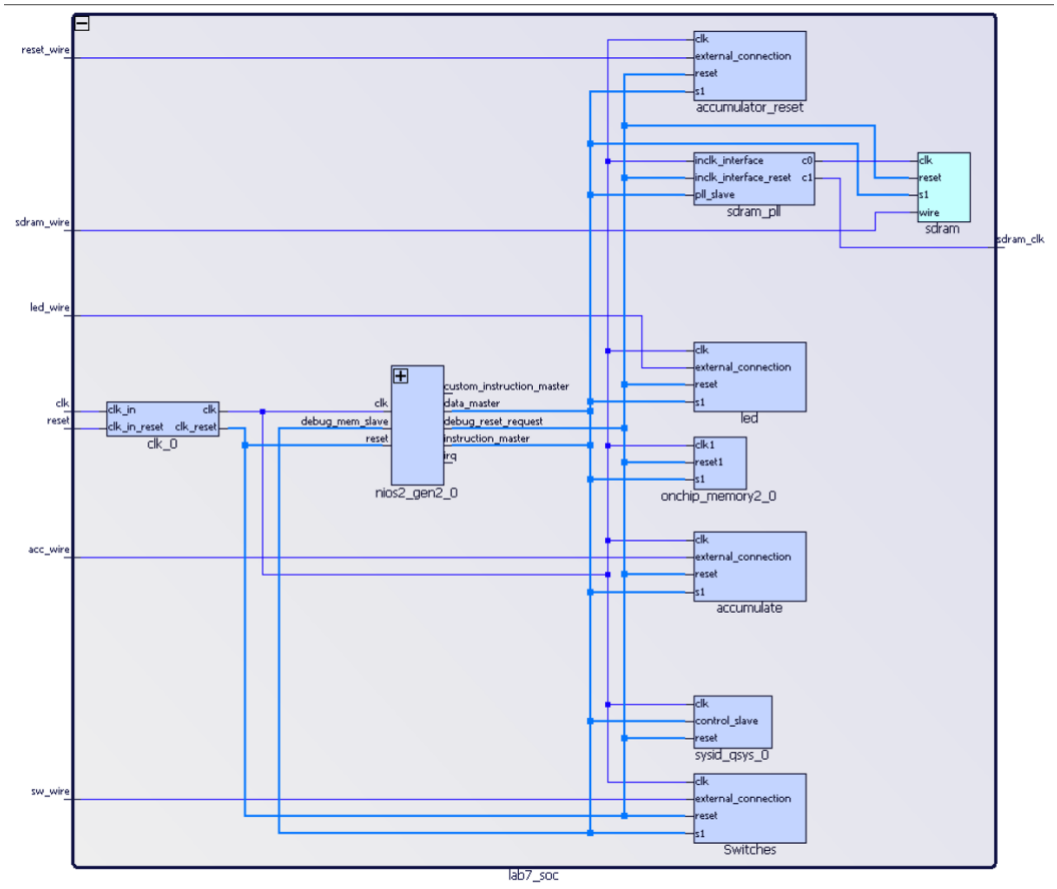
Modules	**lab7_soc.v**
Inputs	input wire acc_wire_export, input wire clk_clk, input wire reset_reset_n, input wire reset_wire_export input wire [7:0] sw_wire_export
Outputs	output wire [7:0] led_wire_export output wire sdram_clk_clk, output wire [12:0] sdram_wire_addr, output wire [1:0] sdram_wire_ba, output wire sdram_wire_cas_n, output wire sdram_wire_cke,

	output wire sdram_wire_cs_n, inout wire [31:0] sdram_wire_dq, output wire [3:0] sdram_wire_dqm, output wire sdram_wire_ras_n, output wire sdram_wire_we_n,
Description	This is the System on Chip module that connects all the hardware together such that the C code can read/write the variables located here.
Purpose	It connects all the individual hardware components such as the LED, buttons, switches, processor to read/write from them. The interface with Platform Designer allows the C code to read the FPGA switches and buttons and modify the LEDs.

iv. *Top Level Block Diagram (This diagram should represent the placement of all your modules in the top level. Please only include the top-level diagram and not the RTL view of every module. The Qsys view of the NIOS processor is not necessary for this portion.)*



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
☑		**clk_0**	Clock Source							
		clk_in	Clock Input	**clk**	***exported***					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	Double-click to export	clk_0					
		clk_reset	Reset Output	Double-click to export						
☑		**nios2_gen2_0**	Nios II Processor							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]					
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]					
		irq	Interrupt Receiver	Double-click to export	[clk]		IRQ 0	IRQ 31		
		debug_reset_request	Reset Output	Double-click to export	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_1000	0x0000_17ff			
		custom_instruction_m...	Custom Instruction Master	Double-click to export						
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) Intel ...							
		clk1	Clock Input	Double-click to export	**clk_0**					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]	0x0000_0000	0x0000_000f			
		reset1	Reset Input	Double-click to export	[clk1]					
☑		**led**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0060	0x0000_006f			
		external_connection	Conduit	**led_wire**						
☑		**sdram**	SDRAM Controller Intel FPGA IP							
		clk	Clock Input	Double-click to export	**sdram_pll_...**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x1000_0000	0x17ff_ffff			
		wire	Conduit	**sdram_wire**						
☑		**sdram_pll**	ALTPLL Intel FPGA IP							
		inclk_interface	Clock Input	Double-click to export	**clk_0**					
		inclk_interface_reset	Reset Input	Double-click to export	[inclk_interf...					
		pll_slave	Avalon Memory Mapped Slave	Double-click to export	[inclk_interf...	0x0000_00a0	0x0000_00af			
		c0	Clock Output	Double-click to export	sdram_pll_c0					
		c1	Clock Output	**sdram_clk**	sdram_pll_c1					
☑		**sysid_qsys_0**	System ID Peripheral Intel FPGA IP							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_00b8	0x0000_00bf			
☑		**Switches**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0090	0x0000_009f			
		external_connection	Conduit	**sw_wire**						
☑		**accumulate**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0080	0x0000_008f			
		external_connection	Conduit	**acc_wire**						
☑		**accumulator_reset**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	**clk_0**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0070	0x0000_007f			
		external_connection	Conduit	**reset_wire**						



3. Answers to all 11 INQ Questions

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

It reduces the time needed for going off chip or going through the additional logic elements and reduces buffers needed to ensure correct read/writes. Also, it is physically closer to the chip and hence is more efficient for execution.

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

It is Modified Havard as it allows the contents of the instruction memory to be accessed as data. For Pure Havard, data and instructions come in separate busses. For Von Neumann, data and instructions come in same bus.

As we have multiple buses, the machine cannot use a Von Neumann architecture. Since there are two memory address spaces, the memory is all stored either on chip or in the SDRAM and the address are changeable, it is more likely to be a Modified Havard.

c. *Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?*

The LED peripheral only needs to be written and receive data from the board as it is an output peripheral. It does not need to alter or store memory anywhere else in the program. However, the on-chip memory needs to perform read and write. It is part of the program and must know where to read data and write data.

d. *Why does SDRAM require constant refreshing?*

Constant refreshing is used to maintain its contents. The data in SRAM will corrupt over time unless it is constantly refreshed as data are stored with capacitors and the charge might leak out without constant refreshing.

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

2^5(bits per address/width)$\cdot 2^{13}$(rows) $\cdot 2^{10}$(columns) $\cdot 2^1$(chip) $\cdot 2^2$(banks) = 128MB = 1Gbit.

SDRAM Parameter	Short Name	Parameter Value
Data Width	[width]	32 bits
# of Rows	[nrows]	13 rows
# of Columns	[ncols]	10
# of Chip Selects	[ncs]	1
# of Banks	[nbanks]	4

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

Access time = 5.5 ns. Theoretical transfer rate = 32 bits ·1/(access time) = 1 / 5.5s ·32M = 720MB/s

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

Because the chip cannot refresh often enough to prevent data corruption under the lowest frequency 50M

If the SDRAM runs too slow, the charge in the capacitor will leak off which could result in data corrupting. Also, as the processor will run at 50MHz, if the SDRAM runs slower, it will take more time to read or write which will affect the performance.

h. *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.*

This is to ensure that data reads from the bus are correct without corruption by data writes to the bus. SDRAM has data stability issues such as the SDRAM CAS latency, this might cause the SDRAM to read in data from the bus when the data is about to change, so we need to shift the clock to read in the correct value.

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

NIOS II starts execution at x10000000. We do this step after assigning the addresses is to ensure the program starts where the instructions are located and make the processor know where to start the program as well as where to go back to when the program is reset, then system can start in a definite state without changing the addresses of the PIO blocks.

j. *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.*

Blinker Code: This has been explained in the previous section.

More detailed,

Line 7 creates variable *i*, used as the counter, and puts it on the runtime stack.

Line 8 creates the volatile unsigned *int* * that stores the value to go to the LEDs. The Volatile in the code is used to tell the compiler that this data represents some hardware, something that can be accessed outside of the program, such that the C program can write to that address instead of only reads from the cache.

Line 10 clear all LEDs to reset the LED pointer so that its value is 0 when dereferenced.

Line 11 creates a while loop that is infinite.

Line 13 creates a software delay such that we can see the LED switch from off to on.

Line 14 turns the rightmost LED on.

Line 15 creates a software delay such that we can see the LED switch from on to off.

Line 16 turns the rightmost LED off.

Line 18 holds the return statement for the function. It is never reached.

```
5   int main()
6   {
7       int i = 0;
8       volatile unsigned int *LED_PIO = (unsigned int*)0xa0; //make a pointer to access the PIO block
9
10      *LED_PIO = 0; //clear all LEDs
11      volatile unsigned int accumulator = 0;
12      int Pause = 0;
13      while ( (1+1) != 3) //infinite loop
14      {
15          for (i = 0; i < 100000; i++); //software delay
16          *LED_PIO |= 0x1; //set LSB
17          for (i = 0; i < 100000; i++); //software delay
18          *LED_PIO &= ~0x1; //clear LSB
19      }
20      return 1; //never gets here
21  }
```

k. *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.*

.bss	static int x;	static variables that are not initialized to a value
.heap	int* m = (int *) malloc(1);	memory allocations using heap
.rodata	const int i = 0;	read only data
.rwdata	int i = 0;	read/write data
.stack	int x = getInt();	Stack used, function call
.text	char[] str = “myString”;	segment of memory that handles executable instructions

4. Post Lab Question

a. *Document the Design Resources and Statistics in following table.*

LUT	2291
DSP	0
Memory (BRAM)	10368bits
Flip-Flop	1991
Frequency	90.56MHz
Static Power	102.06mW
Dynamic Power	41.38mW
Total Power	204.80mW

5. Conclusion

a. *Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it*

In this experiment, we used NIOS II to realize a blink program and an accumulator program. Our design works well and there are no terrible bugs encountered, except for the instability of Eclipse, which always generates errors in run configuration while there are no bugs.

b. *Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right so it doesn't get change.*

Everything is clear and fine