

Experiment 7 Report

SOC with NIOS II in SystemVerilog

ECE 385 Fall 2020

Homero Vazquez

hvazqu6

Chuangyi Zhang

czhan30

1. Introduction

The purpose of these experiments is to learn the basic capability of the NIOS II processor as the foundation of our System-on-Chip (SoC) projects. We learned the fundamentals of memory-mapped I/Os and implemented a simple SoC interfacing with peripherals such as the onboard switches and LEDs.

a. Summarize the basic functionality of the NIOS-II processor running on the MAX10 FPGA.

The NIOS-II processor is an IP based 32-bit CPU which can be programmed using a high-level language. A typical use case scenario is to have the NIO-II be the system controller and handle tasks which do not need to be high performance such as user interface, data-input, and output. The basic functionality of the NIOS-II processor that we build was to accumulate the value from switch [7:0] and display the result in the LEDs [7:0] from 0-255 unsigned. We implemented PIOs for LEDs [7:0], switches [7:0], and Key[1:0].

When a user pressed Key[0], it would reset the LEDs to zero and clear the value that was stored in the LEDs. When a user pressed Key[1], which would accumulate, it would add the value from Switches [7:0] and display the result value on the LEDs.

2. Written Description and Diagrams of NIOS-II System

A NIOS II based system on the Altera MAX10 device. The NIOS II is an IP based 32-bit CPU which can be programmed using a high-level language (in this class, we'll be using C). A typical use case scenario is to have the NIOS II be the system controller and handle tasks which do not need to be high performance (for example, user interface, data input and output) while an accelerator peripheral in the FPGA logic (designed using SystemVerilog) handles the high-performance operations.

i. Describe in words the hardware component of the lab, in this lab, only the Platform Designer module is here.

 nios2_gen2_0	Nios II Processor			
clk	Clock Input	<i>Double-click to export</i>	clk_0	
reset	Reset Input	<i>Double-click to export</i>	[clk]	
data_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]	
instruction_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]	
irq	Interrupt Receiver	<i>Double-click to export</i>	[clk]	
debug_reset_request	Reset Output	<i>Double-click to export</i>	[clk]	
debug_mem_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	
custom_instruction_m...	Custom Instruction Master	<i>Double-click to export</i>		
				IRQ 0
				0x0000_1000

Nio2_gen2_0 is our CPU, which will process our inputs/outputs from switches or LEDs.

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
reset1	Reset Input	Double-click to export	[clk1]	

Onchip_memory2_0 is our on-chip memory, but we didn't use it at all for this lab.

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_0090
external_connection	Conduit	led_wire		

Led module will output the value of the accumulator

sdram	SDRAM Controller Intel FPGA IP			
clk	Clock Input	Double-click to export	sdram_pll_c0	
reset	Reset Input	Double-click to export	[clk]	
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_0000
wire	Conduit	sdram_wire		

Sdram module is how we store and write our value from/to the Leds and switches.

sdram_pll	ALTPLL Intel FPGA IP			
indk_interface	Clock Input	Double-click to export	clk_0	
indk_interface_reset	Reset Input	Double-click to export	[indk_interface]	
pll_slave	Avalon Memory Mapped Slave	Double-click to export	[indk_interface]	0x0000_00a0
c0	Clock Output	Double-click to export	sdram_pll_c0	
c1	Clock Output	sdram_clk	sdram_pll_c1	

Sdram module generates two clks and connect with the sdram_clk

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

Sysid_qsys_0 module just makes sure that hardware and software is compatible.

sw	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
external_connection	Conduit	sw_wire		

Sw module will take user inputs and wait for the signal to accumulate.

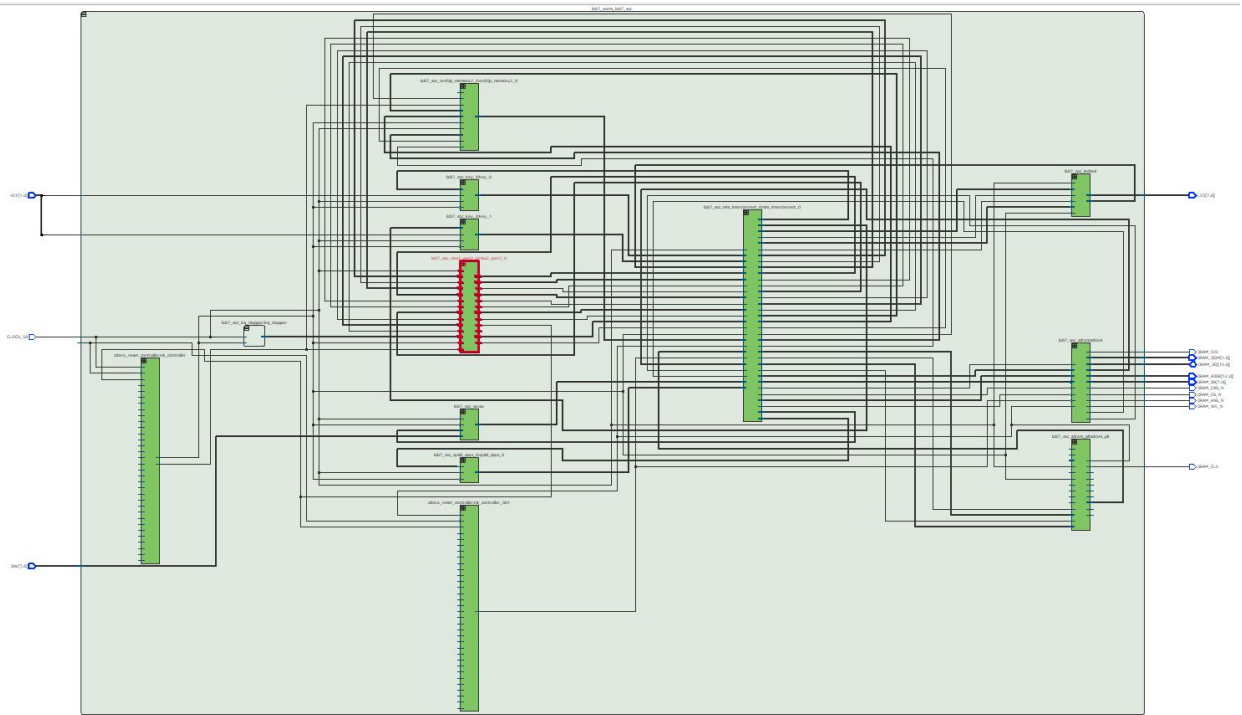
key_1	PIO (Parallel I/O) Intel FPGA IP			
clk	Clock Input	Double-click to export	clk_0	
reset	Reset Input	Double-click to export	[clk]	
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0070
external_connection	Conduit	key1_wire		

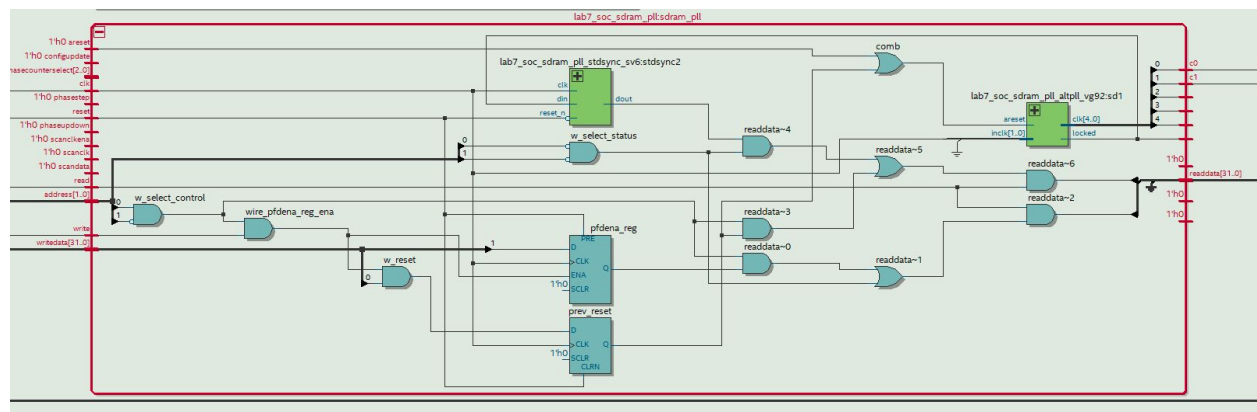
Key_1 takes the user inputs signal to accumulate the values of LEDs and switches

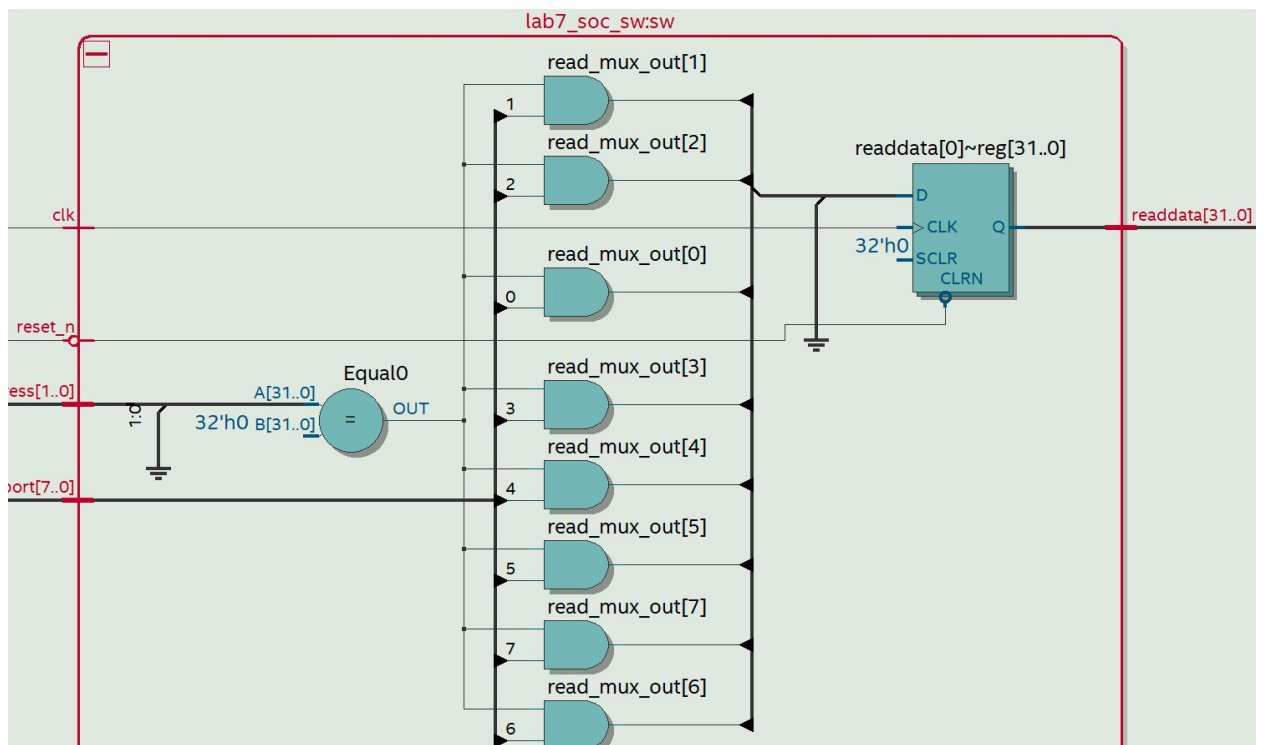
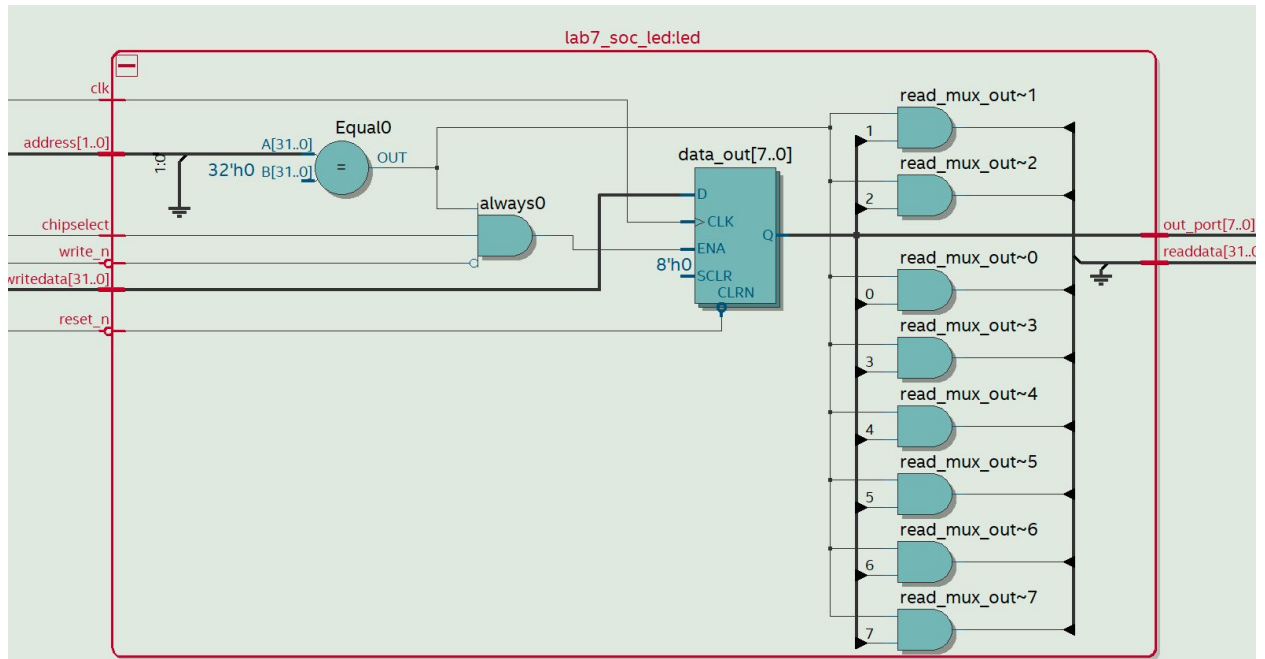
key_0	PIO (Parallel I/O) Intel FPGA IP			
clk	Clock Input	Double-click to export	clk_0	
reset	Reset Input	Double-click to export	[clk]	
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0060
external_connection	Conduit	key0_wire		

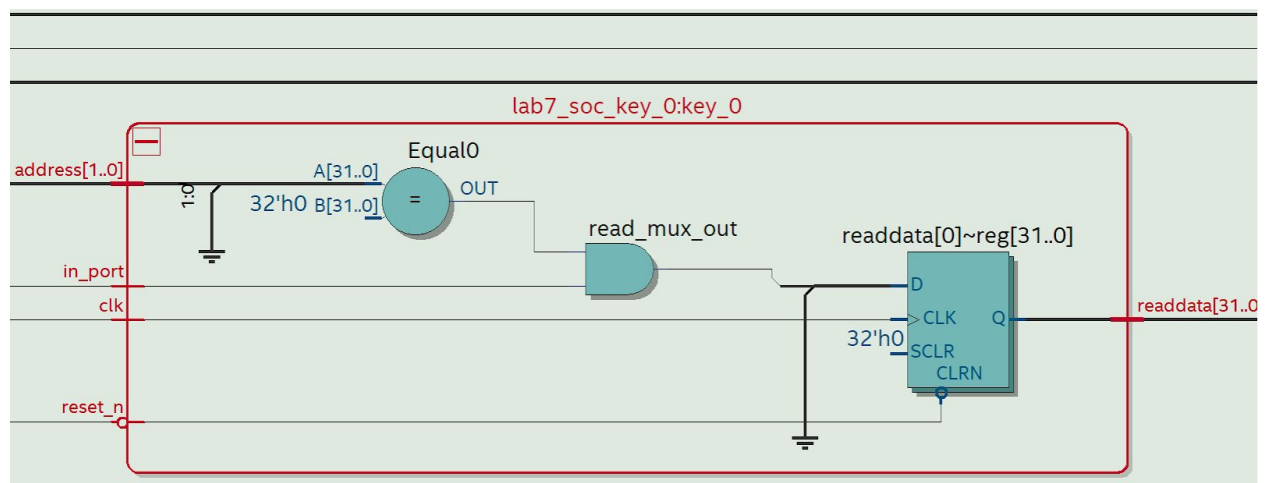
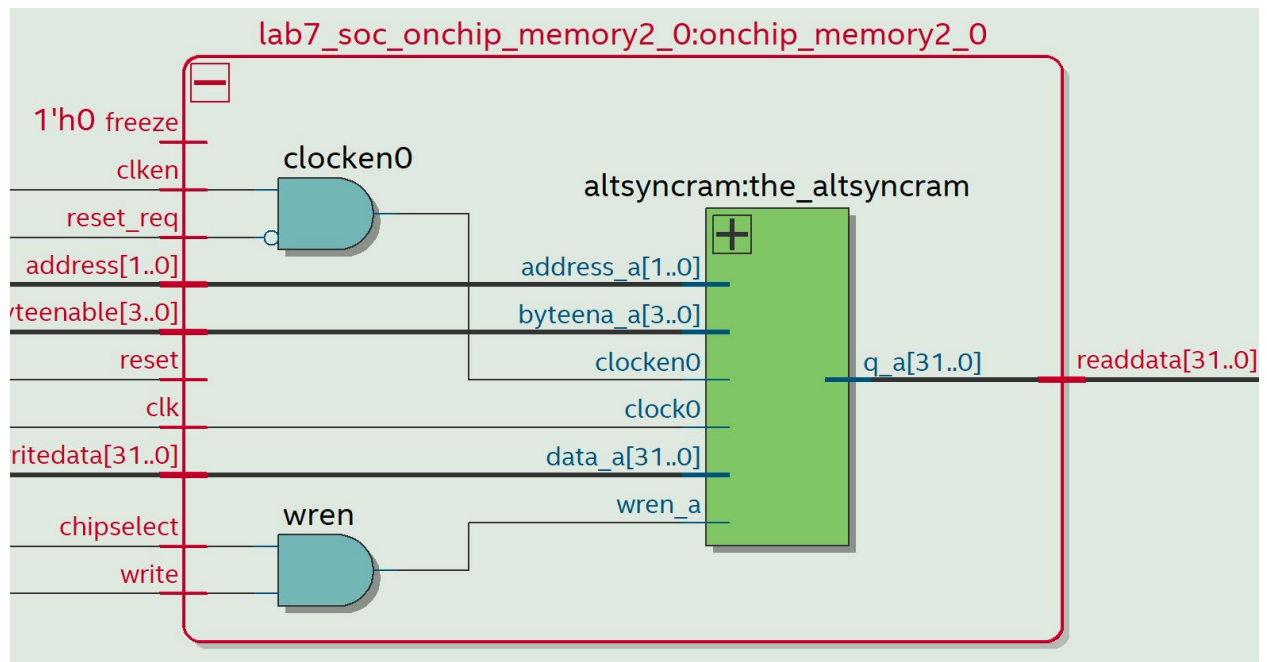
Key_0 takes the user input signal to reset the value of the LEDs

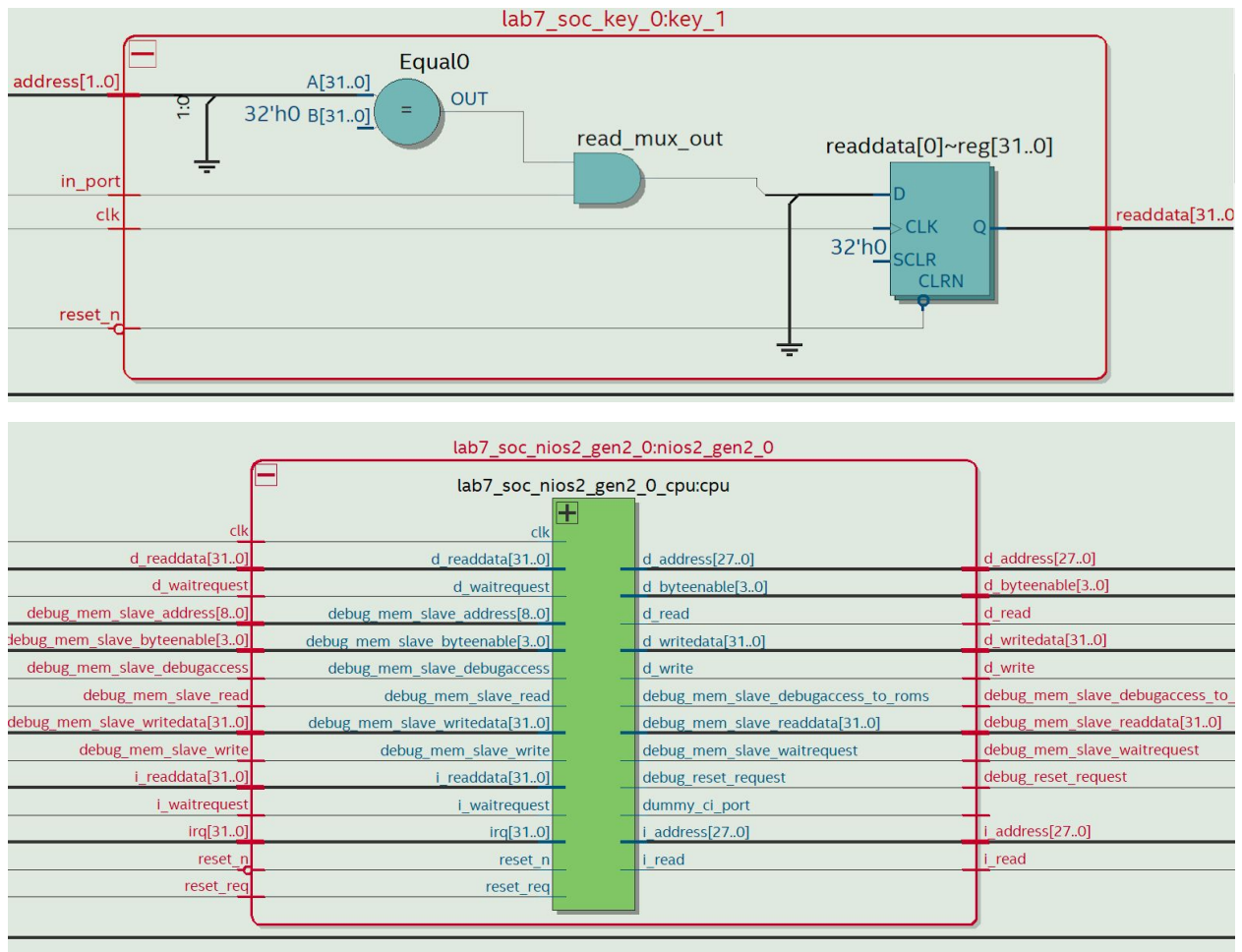
3. Top Level Block Diagram











i. For this lab, this may be trivial, since there is only the Platform Designer module.

4. Written Description of all .sv Modules

- i. For this lab, this may be trivial, since there is only the Platform Designer module.
- ii. A guide on how to do this was shown in the Lab 5 report outline. Do not forget to describe the Qsys generated file lab7soc.v!

Module: lab7.sv

Inputs: CLOCK_50, [1:0] KEY, [15:0] DRAM_DQ

Outputs: [7:0] LED, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [1:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK

Descriptions: This file is the top level file for the entire lab and it just connects the SOC file to some external outputs and inputs

Purpose: The purpose of this module is to connect the SOC files to external inputs and outputs.

Module : lab7_soc.sv

Inputs: clk_clk, key)_wire_export, key1_wire_export, reset_reset_n, [7:0] sw_wire_export

Outputs: [7:0] led_wire_export, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cke, sdram_wire_cas_n, sdram_wire_cs_n, [1:0] sdram_wire_dqm, sdram_wire_we_n, sdram_wire_ras_n


I/O: [15:0] sdram_wire_dq

Descriptions: This module connects every submodule of the SoC part of the lab and connects different blocks like the NIOS II, the PIOs the SDRAM and SDRAM Controller and other files form the platform designer.

Purpose: The purpose of this file is to connect all the blocks from platform designers that were made into hardware HDL files.

5. System Level Block Diagram (this is new for labs 7-9)

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).

 nios2_gen2_0	Nios II Processor			
clk	Clock Input	<i>Double-click to export</i>	clk_0	
reset	Reset Input	<i>Double-click to export</i>	[clk]	
data_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]	
instruction_master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]	
irq	Interrupt Receiver	<i>Double-click to export</i>	[clk]	
debug_reset_request	Reset Output	<i>Double-click to export</i>	[clk]	
debug_mem_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	
custom_instruction_m...	Custom Instruction Master	<i>Double-click to export</i>		
				IRQ 0
				0x0000_1000

Nio2_gen2_0 module is the NIOS-II Processor. It has clk, reset, data_master, instruction_master, and instruction_master, irq, bebug_reset, debug_slave, and custom_instruction_master. With those all I/O ports which allows the processor to interrupt, interface, and execute the instructions.

Nio2_gen2_0 is our SoC for this lab, and it has the base address at 0x00001000, which allows us to process all the signals, instructions, and PIOs interfacing

led	PIO (Parallel I/O) Intel FPGA IP			
clk	Clock Input	<i>Double-click to export</i>	clk_0	
reset	Reset Input	<i>Double-click to export</i>	[clk]	
s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	
external_connection	Conduit	led_wire		0x0000_0090

Led module is the PIOs for LED, it has clk, reset, s1, and external_connection. It allows the led_wire to access the data from the avalon bus at each clock rising edge. It has the base address at 0x00000090

onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...			
clk1	Clock Input	<i>Double-click to export</i>	clk_0	
s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk1]	
reset1	Reset Input	<i>Double-click to export</i>	[clk1]	0x0000_0000

Onchip_memory2 is the memory module that we allocate our on-chip memory. It has the clock input, avalon memory slave, and reset_input. It allows data to be read or written into the on-chip memory.

Onchip_memory2_0 is our on-chip memory module and which allows us to access the data. This on-chip memory has 16-bit data_width, 13 rows, 14 columns, and 1 chip selects, and 4 banks, which it has 32M*16 and has a total amount of 512 MBits.

sdram	SDRAM Controller Intel FPGA IP			
clk	Clock Input	<i>Double-click to export</i>	sdram_pll_c0	
reset	Reset Input	<i>Double-click to export</i>	[clk]	
s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	
wire	Conduit	sdram_wire		0x0800_0000

Sdram is the module that we are interfacing with LEDs and switches. Sdram has clk, reset, and avalon memory, and it has the external wire connected with sdram_clk that it generates by the sdram_pll to control how the data to be read and written into the sdram. It has the base address at 0x08000000

sdram_pll	ALTPLL Intel FPGA IP			
indk_interface	Clock Input	<i>Double-click to export</i>	clk_0	
indk_interface_reset	Reset Input	<i>Double-click to export</i>	[indk_interface]	
pll_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[indk_interface]	
c0	Clock Output	<i>Double-click to export</i>	sdram_pll_c0	
c1	Clock Output	sdram_clk	sdram_pll_c1	0x0000_00a0

Sdram_pll is the module that has one clk input and two clk outputs. It generates the clock that can connect to the sdram. This module allows us to add some clock delay for sdram in order to get the right result for read and write. It has the base address at 0x00000a0

[-] sysid_qsys_0	System ID Peripheral Intel FPGA IP			
clk	Clock Input	<i>Double-click to export</i>	clk_0	
reset	Reset Input	<i>Double-click to export</i>	[clk]	
control_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	0x0000_00b8

Sysid_qsys_0 is the system ID checker, which can prevent any incompatibility between hardware and software. It has the base address at 0x00000b8

[-] sw	PIO (Parallel I/O) Intel FPGA IP			
clk	Clock Input	<i>Double-click to export</i>	clk_0	
reset	Reset Input	<i>Double-click to export</i>	[clk]	
s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	0x0000_0080
external_connection	Conduit	sw_wire		

Sw is the PIO module for SW[7:0], and it has clk, reset, and s1, and external_connection(sw_wire), which NIOS_II can take the input value of the switch[7:0] at every rising edge. It has the base address at 0x0000080

[-] key_1	PIO (Parallel I/O) Intel FPGA IP			
clk	Clock Input	<i>Double-click to export</i>	clk_0	
reset	Reset Input	<i>Double-click to export</i>	[clk]	
s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	0x0000_0070
external_connection	Conduit	key1_wire		

key_1 is the PIO module for Key[1], and it has clk, reset, and s1, and external_connection(sw_wire), which NIOS_II can take the input value of the Key[1] at every rising edge. It has the base address at 0x0000070

[-] key_0	PIO (Parallel I/O) Intel FPGA IP			
clk	Clock Input	<i>Double-click to export</i>	clk_0	
reset	Reset Input	<i>Double-click to export</i>	[clk]	
s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	0x0000_0060
external_connection	Conduit	key0_wire		

key_0 is the PIO module for Key[0], and it has clk, reset, and s1, and external_connection(sw_wire), which NIOS_II can take the input value of the Key[0] at every rising edge. It has the base address at 0x0000060

- Describe in words the software component of the lab. One of the INQ questions asks about the blinker code, but you must also describe your accumulator.

Blinker code

```
1 // Main.c - makes LEDG0 on DE2-115 board blink if NIOS II is set up correctly
4
5 int main()
6 {
7     int i = 0;
8     volatile unsigned int *LED_PIO = (unsigned int*)0x90; //make a pointer to access the PIO block
9
10    *LED_PIO = 0; //clear all LEDs
11    volatile unsigned int sum = 0;
12    int pause = 0;
13
14    while ( (1+1) != 3) //infinite loop
15    {
16        for (i = 0; i < 100000; i++); //software delay
17        *LED_PIO |= 0x1; //set LSB
18        for (i = 0; i < 100000; i++); //software delay
19        *LED_PIO &= ~0x1; //clear LSB
20    }
21    return 1; //never gets here
22 }
```

Accumulator code

```
1 // Main.c - makes LEDG0 on DE2-115 board blink if NIOS II is set up correctly
4
5 int main()
6 {
7     int i = 0;
8     volatile unsigned int *LED_PIO = (unsigned int*)0x90; //make a pointer to access the LED PIO block
9     volatile unsigned int *SW_PIO = (unsigned int*)0x80; //make a pointer to access the SW PIO block
10    volatile unsigned int *Reset_PIO = (unsigned int*)0x60; //make a pointer to access the RESET PIO block
11    volatile unsigned int *Accumulate_PIO = (unsigned int*)0x70; //make a pointer to access the Accumulate PIO block
12
13
14    *LED_PIO = 0; //clear all LEDs
15    volatile unsigned int sum = 0; // declare sum variable to hold sum value
16    int pause = 0; // make a pause
17
18    while ( (1+1) != 3) //infinite loop
19    {
20
21        if(*Reset_PIO == 0x0 ) // if reset is pressed
22            sum = 0; // LEDs will be cleared
23
24
25
26        if(*Accumulate_PIO == 0x1 && pause == 0){ //if the Accumulate is Pressed and Pause is off
27            sum += *SW_PIO; //sum will adds the value of the switches
28            pause = 1; // Pause will be on
29        }
30
31        if(*Accumulate_PIO == 0x0) // If the accumulate is not prssed
32            pause = 0; // Pause will be off
33
34        *LED_PIO = sum; // LEDs display the sum
35    }
36    return 1; //never gets here
37 }
38 }
```

All the description of the code annotated in the screenshot.

a. Answers to all INQ Questions

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

<p>Nios II/f[edit]</p> <p>The Nios II/f core is designed for maximum performance at the expense of core size. Features of Nios II/f include:</p> <ul style="list-style-type: none">• Separate instruction and data caches (512 B to 64 kB)• Optional MMU or MPU• Access to up to 2 GB of external address space• Optional tightly coupled memory for instructions and data• Six-stage pipeline to achieve maximum DMIPS/MHz• Single-cycle hardware multiply and barrel shifter• Optional hardware divide option• Dynamic branch prediction• Up to 256 custom instructions and unlimited hardware accelerators• JTAG debug module• Optional JTAG debug module enhancements, including hardware breakpoints, data triggers, and real-time trace	<p>Nios II/e[edit]</p> <p>The Nios II/e core is designed for smallest possible logic utilization of FPGAs. This is especially efficient for low-cost Cyclone II FPGA applications. Features of Nios II/e include:</p> <ul style="list-style-type: none">• Up to 2 GB of external address space• JTAG debug module• Complete systems in fewer than 700 LEs• Optional debug enhancements• Up to 256 custom instructions• Free, no license required
--	--

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

The advantage of on-chip memory is highest throughput and lowest latency memory, usually taking around 1 cycle. We also don't need additional board space and it can directly implement the FPGA.

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

NIOS-II is a harvard machine, because there are separate data and instruction busses. It is not a Von Neumann architecture, because it doesn't combine all the busses into one.

4. **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 LEDs only display the value of the data and it is not any of the system verilog file or modules, therefore we don't need to access the data or program bus.

5. **Why does SDRAM require constant refreshing?**

DRAM memory is held by using a transistor and a capacitor. And capacitor leaks charge over time, so we need to be refreshed in order to not lose the data.

6. **For the SDRAM table on the bottom of page 8, how did we come up with the numbers?**

$$(2^{13})(2^{10}) \cdot 1 \cdot 4 \cdot 16 = 512 \text{Mbits}$$

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

The maximum theoretical transfer rate to the SDRAM is 39.4ns, because the active to read, or active to write delay is 20ns, access time is 5.4ns, write recovery time is 14ns.

8. **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? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.**

According to the Altera Embedded Peripheral IP datasheet, the data-in hold time is 1ns. That means data-in has to be right when it is read. Therefore clk c1 is 1ns behind of the controller clk, so data-in can fall within the valid window.

9. **What address does the NIOS II start execution from? Why do we do this step after assigning the addresses? (Page 14)**

NIOS II starts executing from 0x00000020 because the memory occupies the addresses from 0x00000000 to 0x0000000f.

Also the exception vector causes the processor to skip the normal operation and jump to a prefixed location. For the NIOS II we preassigned the exception vector to 0x00000020 that's why it starts at that point

10. **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). (Page 20)

The volatile keyword means that a variable can change at any time without any action taken from the code. Which is useful for hardware since we change variables without using code.

11. **Look at the various segments (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment. (Page 21)**

.bss	Uninitialized global/static variables	Static int x; Static int x =0;
.heap	Heap memory which is managed by malloc,	malloc();

	calloc, realloc and free	
.rodata	Read only data, is when we define a value	#define x = 5;
.rwdta	Read- write data, they can be modified so like regular data variables	int x =0; x = 10;
.stack	A LIFO structure, variables that go into the stack	function (x, y, z)
.text	Part of the code that contain the object file, program file and others	int main(printf("I love ECE 385!");)

- Post-lab Question
- Document the Design Resources and Statistics in a table provided in the lab.

LUT	2796
DSP	0
Memory(BRAM)	10368
Flip_flop	1823
Frequency	84.23 mHz
Static Power	96.44mW
Dynamic Power	48.20mW
Total Power	162.02mW

7. Conclusion

- Overall our design performed as expected. We were able to accomplish the specified tasks for the demo. It was really interesting to be able to learn about NIOS II and how we can write

some software programs into the processor on the fpga to accomplish different tasks.

Something that we had trouble with was with some of the errors that we would get from eclipse and that was where we had the most trouble with but at the end we were able to fix it and it worked out.

- I think that having a list of possible errors from the platform designer, eclipse and how to correctly compile would help a lot because we spent a lot of time figuring out why our code wouldn't be able to create the run configurations or why would we get random errors that have nothing to do with our programming or the hardware that we build.