ECE 385

Fall 2019

Experiment #7

SOC with NIOS II in SystemVerilog

Pouya Akbarzadeh and Patrick Stach

Section: ABC

Lab TA: Vikram Anjur, Yuhong Li

Index

7.1 Introduction

The point of this experiment was to get familiar and learn the basic capability of the NIOS II processor. This is for the foundation of the System-On-Chips projects. We learned and experimented with memory-mapped I/Os and implement the simple SoC interfacing with peripherals such as the LEDs and switches.

7.2 Written Description of NIOS-II System

7.2.1 Summary of Operation

We wanted to implement the switches and they keys and an output LED. The switches were implemented to accumulate values from 0-255. We implemented two buttons as well. Button-2 was a software reset button while button was the accumulator. We also had to take into account that the system can overflow at the value of 255 in which where we would display 0. This was accomplished with not only the standard hardware that NIOS uses to operate but with some extra implementations. We also implement peripheral input-output blocks. PIOs, peripheral input-output blocks, are used as an interface with other hardwares. We had to interface with the LED outputs and the input switches. We used two 8-bit PIO's to alongside 2 1bit PIO to accomplish this.

The rudimentary C code, starts running when the board is on. When the program detects a key pressed the value of the switches will be added on to the LEDs. After the first iteration there is this while loop which waits until the button is no longer being pressed. This needed to be implemented so we don't keep on adding. The other button, which is reset, is pressed the value saved alongside it's value which is on the hex display resets.

7.2.2 Written Description of all .sv Modules

Module: lab7.sv

```
module lab7(      input         CLOCK_50,
                  input  [3:0]  KEY,
                  input  [7:0]  SW,
                  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:

It is the top level module of this project. It brings all the different elements of the project together and allows for smooth operation. It combines the FPGA with Altera and NIOS II.

Purpose:

The purpose of it is to create interaction between the FPGA and the code written in Eclipse.

Module: lab_7.v

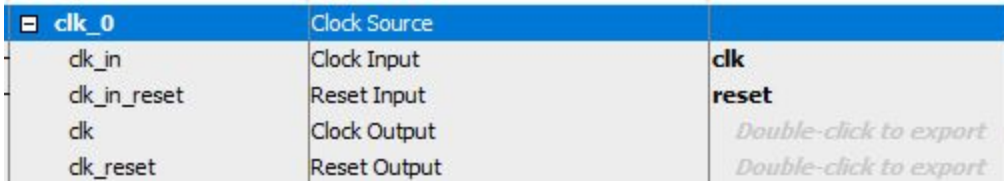
```
module lab7_soc (
        input  wire [1:0]  buttons_wires_export, // buttons_wires.export
        input  wire        clk_clk,              //           clk.clk
        output wire [7:0]  led_wire_export,      //      led_wire.export
        input  wire        reset_reset_n,        //         reset.reset_n
        output wire        sdram_clk_clk,        //     sdram_clk.clk
        output wire [12:0] sdram_wire_addr,      //    sdram_wire.addr
        output wire [1:0]  sdram_wire_ba,        //              .ba
        output wire        sdram_wire_cas_n,     //              .cas_n
        output wire        sdram_wire_cke,       //              .cke
        output wire        sdram_wire_cs_n,      //              .cs_n
        inout  wire [31:0] sdram_wire_dq,        //              .dq
        output wire [3:0]  sdram_wire_dqm,       //              .dqm
        output wire        sdram_wire_ras_n,     //              .ras_n
        output wire        sdram_wire_we_n,      //              .we_n
        input  wire [7:0]  switch_wire_export    //   switch_wire.export
    );
```

Description:
The lab_7.v file is what is generated when you press “Generate HDL” in platform designer. It instantiates all the modules from what is designed in platform designer too.
Purpose:
We need the modules and exported wires to interface with the top level, and we needed platform designer to generate in HDL so we can upload to FPGA

Platform Designer Module Descriptions
In order, we get

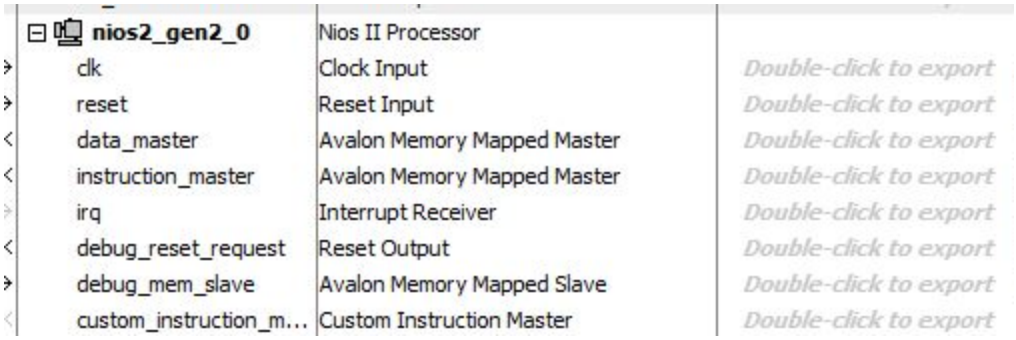
1) ***Input name***
2) ***Input Description***

3) Outputs



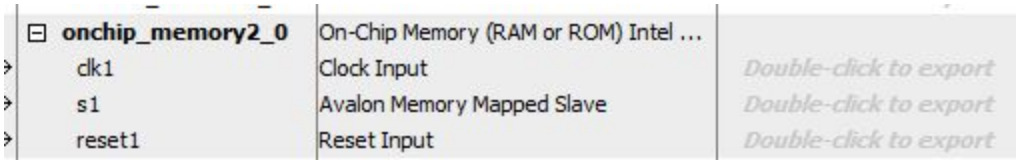
clk_0	Clock Source	
clk_in	Clock Input	**clk**
clk_in_reset	Reset Input	**reset**
clk	Clock Output	Double-click to export
clk_reset	Reset Output	Double-click to export

This module is the clock module- it is simply the 50Mhz generated by the FPGA. The clk from here goes to all the other clocks inputs



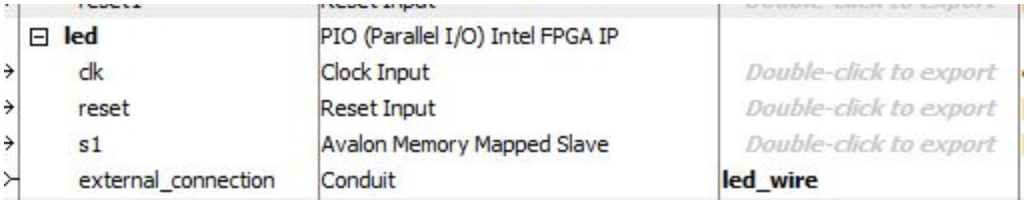
nios2_gen2_0	Nios II Processor	
clk	Clock Input	Double-click to export
reset	Reset Input	Double-click to export
data_master	Avalon Memory Mapped Master	Double-click to export
instruction_master	Avalon Memory Mapped Master	Double-click to export
irq	Interrupt Receiver	Double-click to export
debug_reset_request	Reset Output	Double-click to export
debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export
custom_instruction_m...	Custom Instruction Master	Double-click to export

This is the actual processor- in our case we have the Nios II/e, the economy version of the Nios II/f. This is the module responsible for processing all the instructions. The NIOS II is a 32-bit modified Harvard RISC architecture.



onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...	
clk1	Clock Input	Double-click to export
s1	Avalon Memory Mapped Slave	Double-click to export
reset1	Reset Input	Double-click to export

This module is our on-chip memory, which is often smaller than SRAM in size but is faster and is actually on the chip. The data width is 32 bits and the total memory size is 16 bytes

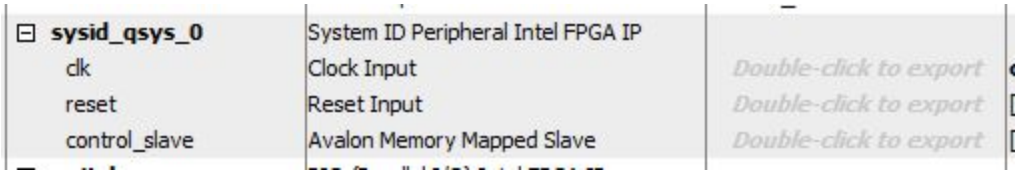


led	PIO (Parallel I/O) Intel FPGA IP	
clk	Clock Input	Double-click to export
reset	Reset Input	Double-click to export
s1	Avalon Memory Mapped Slave	Double-click to export
external_connection	Conduit	**led_wire**

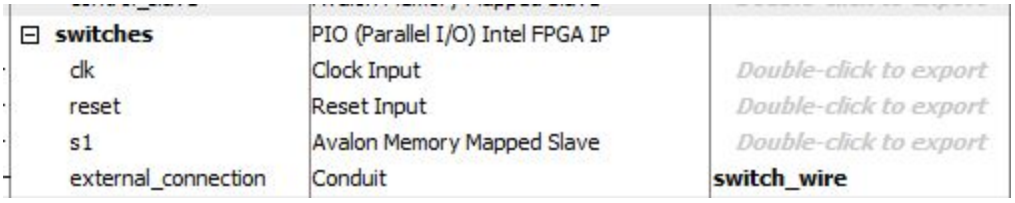
This module is a simple PIO block for our LEDs. The width is 8 since we have 8 LEDs and we have it set to output since we will be writing to the LEDs to control them.

This module is our SDRAM that we are interfacing with. We use SDRAM to store the software program because the on-chip memory is limited. We have to use an SDRAM controller to interface with the bus since we have row/column addressing and constantly needs to refresh in order to retain data.

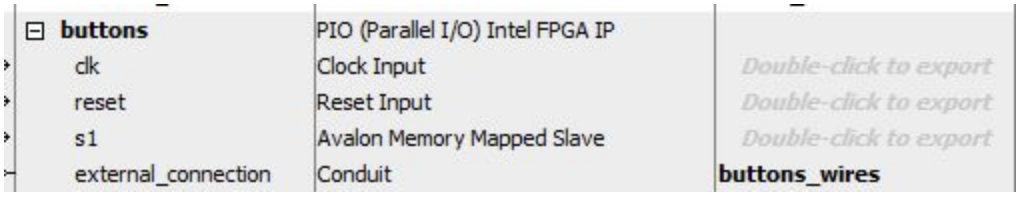
This module generates the clock that goes into the SDRAM. The PLL allows us to account for delays, specifically 3ns in order to have the SDRAM wait for the outputs to stabilize.



This is a system ID checker- this is a preventative measure to prevent any sort of incompatibilities between hardware and software. It will prevent any incompatible NIOS II configurations to be uploaded to the FPGA.



This module is a simple PIO block for our switches. The data width is 8 bits since we have 8 switches, and we have this set as input since we are inputting the values into the system and reading them.



This module is a simple PIO block for our switches-ie. Accumulate and Reset Sum for our accumulate program. The data width is 2 bits wide because we are interfacing with 2 switches, and it is set as input because we are inputting them into the systems and reading the values.

C Code:

(Blink)

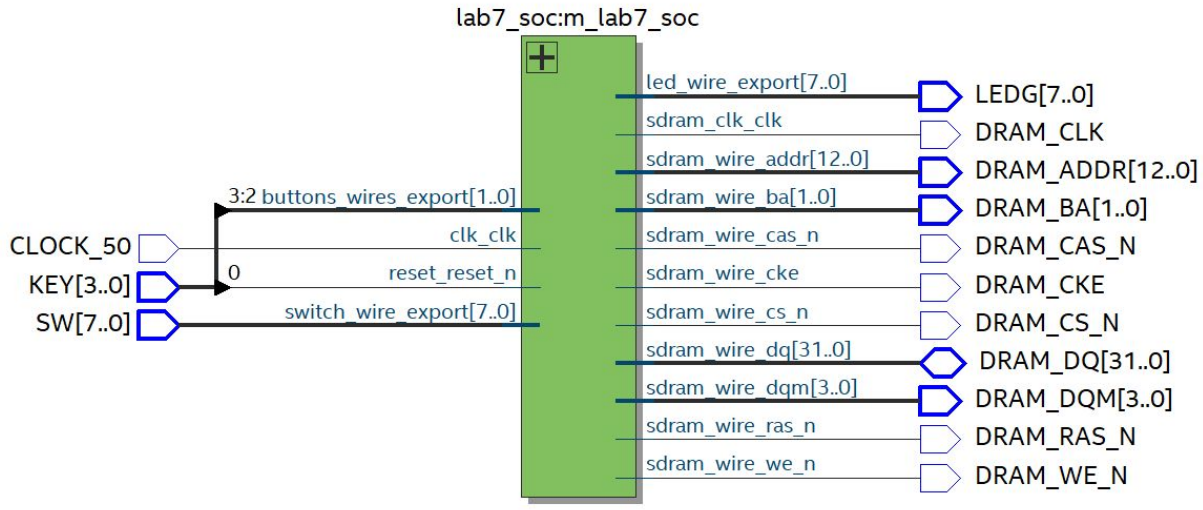
```
int i = 0;
volatile unsigned int *LED_PIO = (unsigned int*)0x40; //make a pointer to access the PIO block

*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
```

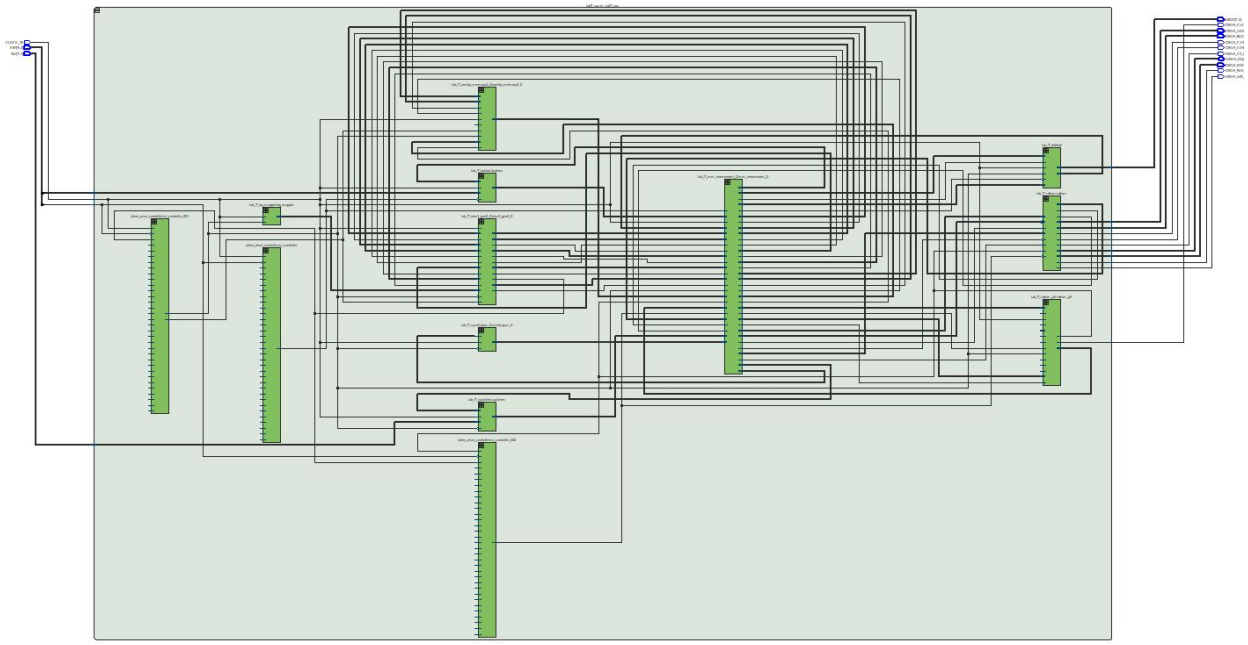
(Accumulator)

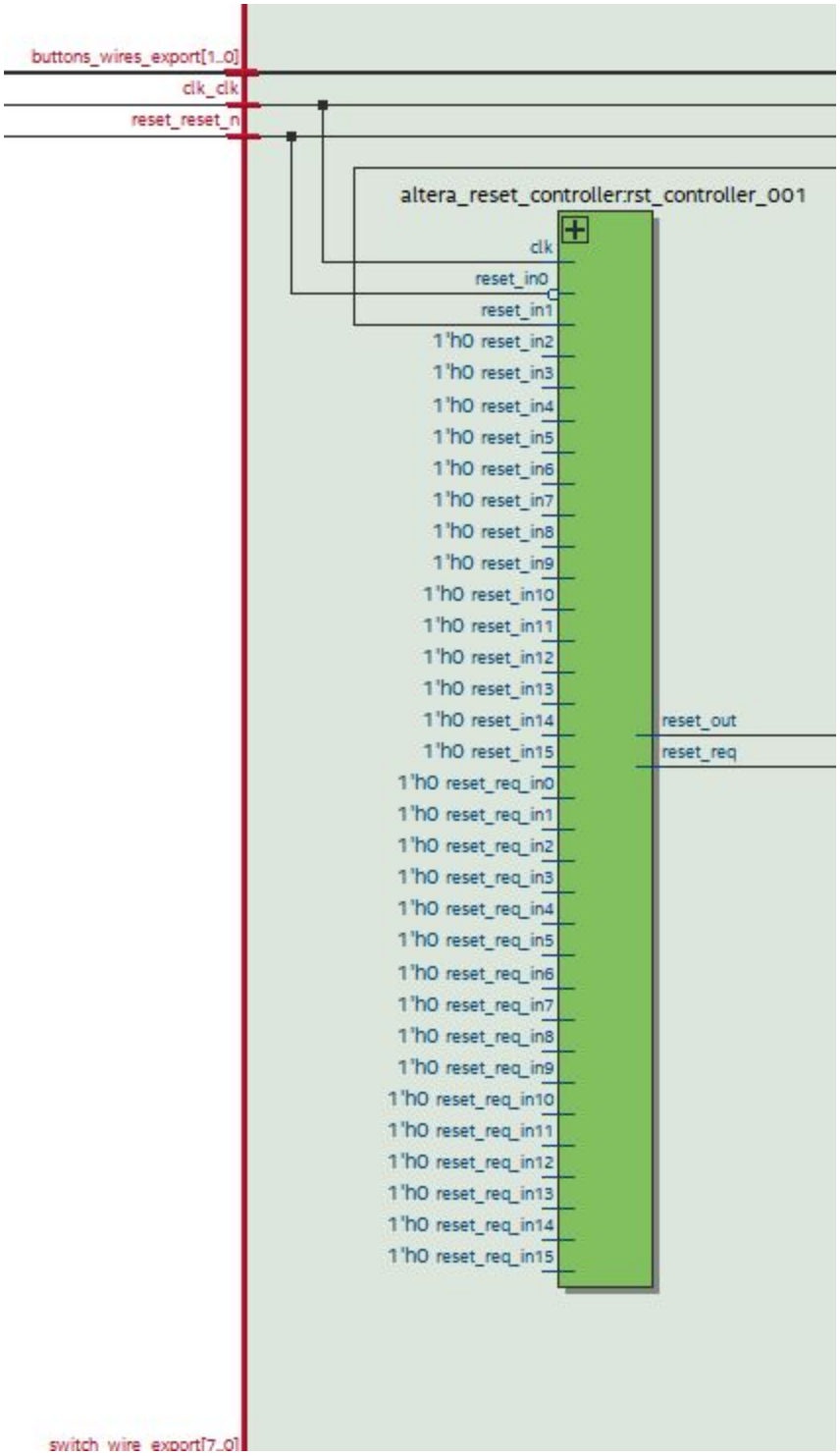
```c
//Accumulator
//
int main()
{
    //KEY[3] = Accumulate =2
    //KEY[2] = RESET = 1
    int buttons_pressed=0;
    volatile unsigned int *LED_PIO = (unsigned int*)0x40; //make a pointer to access the PIO block
    volatile unsigned int *SWITCHES = (unsigned int*)0x30; //make a pointer to access the PIO block
    volatile unsigned int *BUTTONS = (unsigned int*)0x50; //make a pointer to access the PIO block
    int sum =0;
    *LED_PIO = 0; //clear all LEDs
    while(1)
    {
        buttons_pressed = (~(*BUTTONS)& 3);
        //*LED_PIO = buttons_pressed;
        if (buttons_pressed == 1)
        {
            *LED_PIO=0;
            sum = 0;
        }
        else if (buttons_pressed == 2)
        {
            sum = sum+ *SWITCHES;
            while (buttons_pressed ==2)
            {
                buttons_pressed = (~(*BUTTONS)& 3);
            }
            *LED_PIO=sum;
        }
    }
}
```

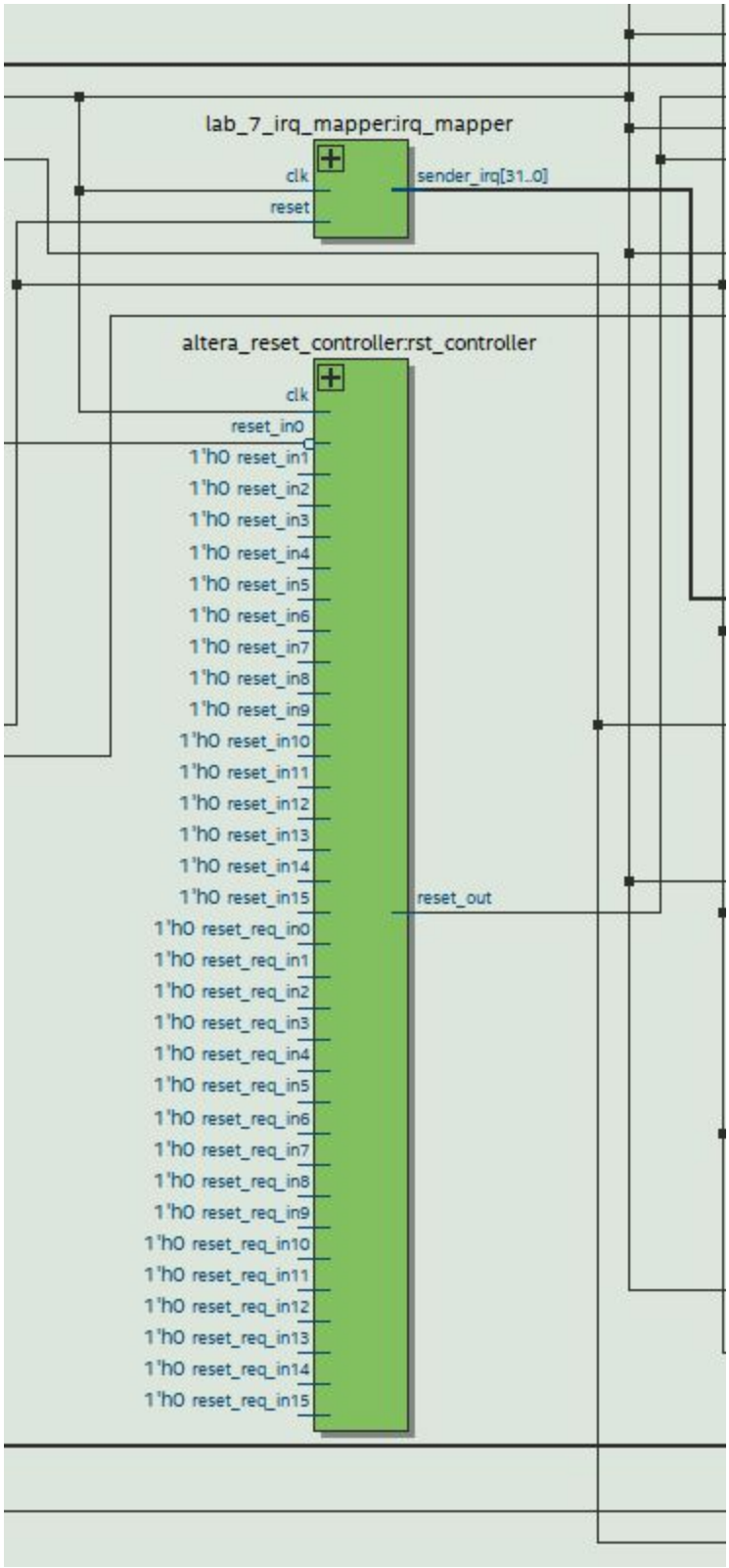
7.2.3 Top Level Block Diagram

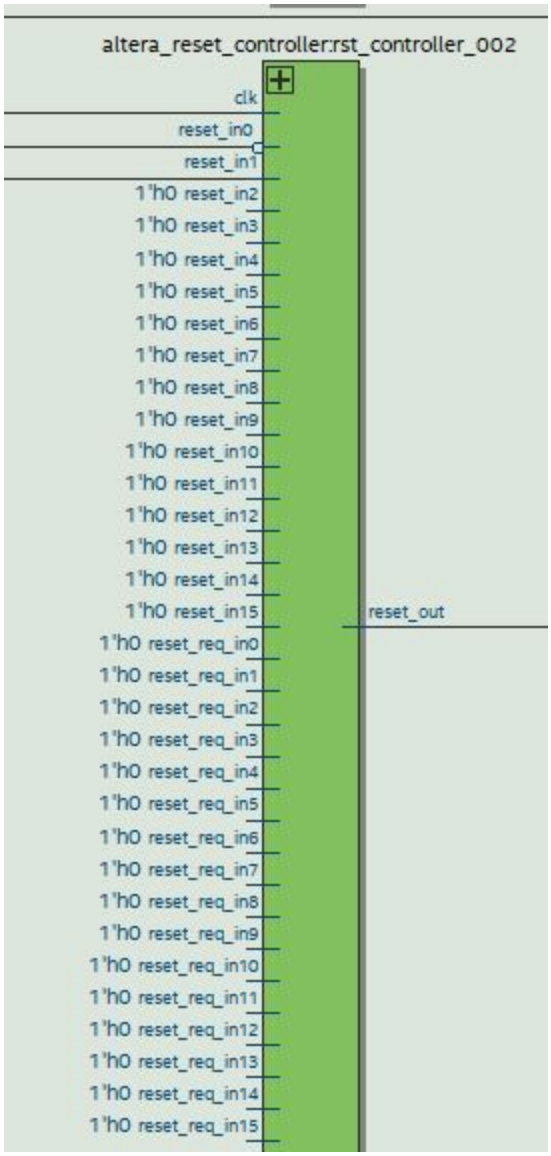


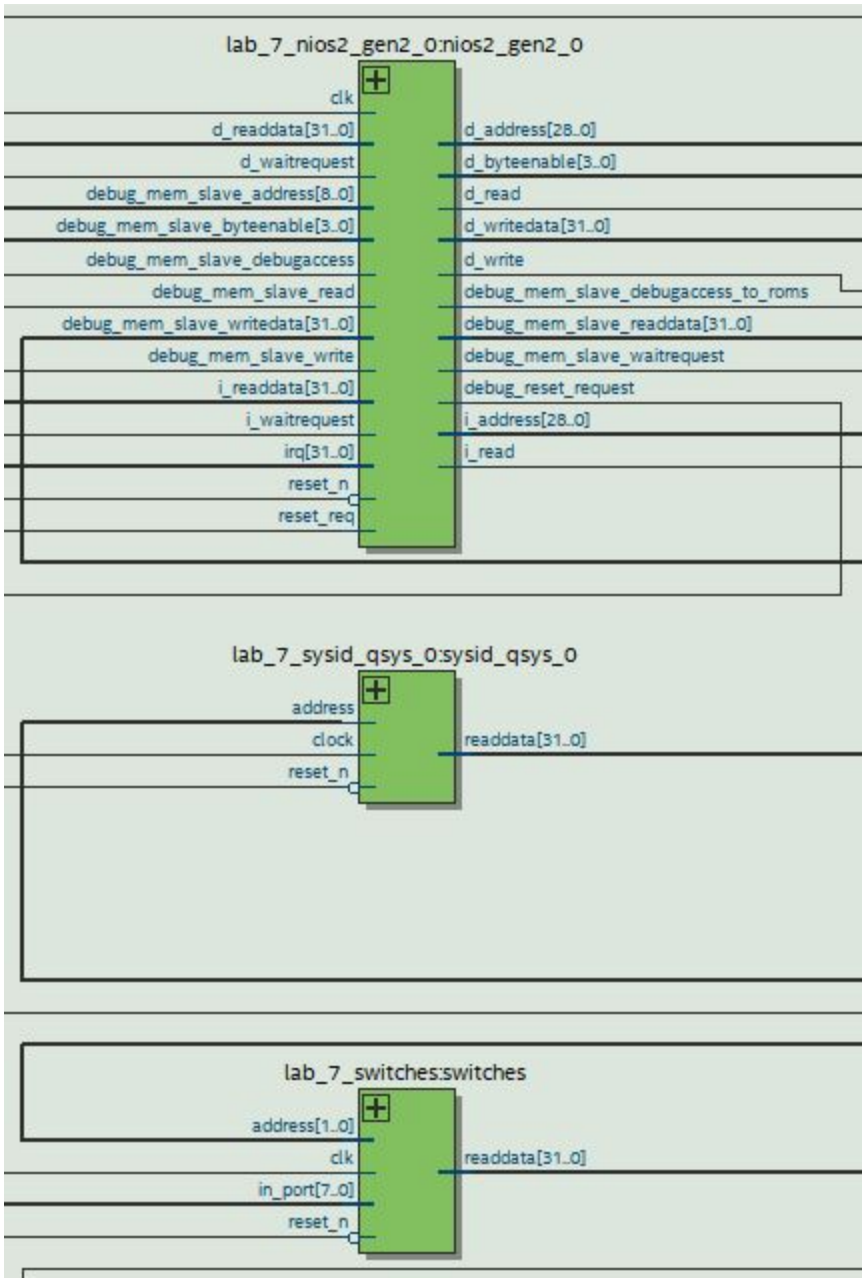
7.2.4 System Level Block Diagram

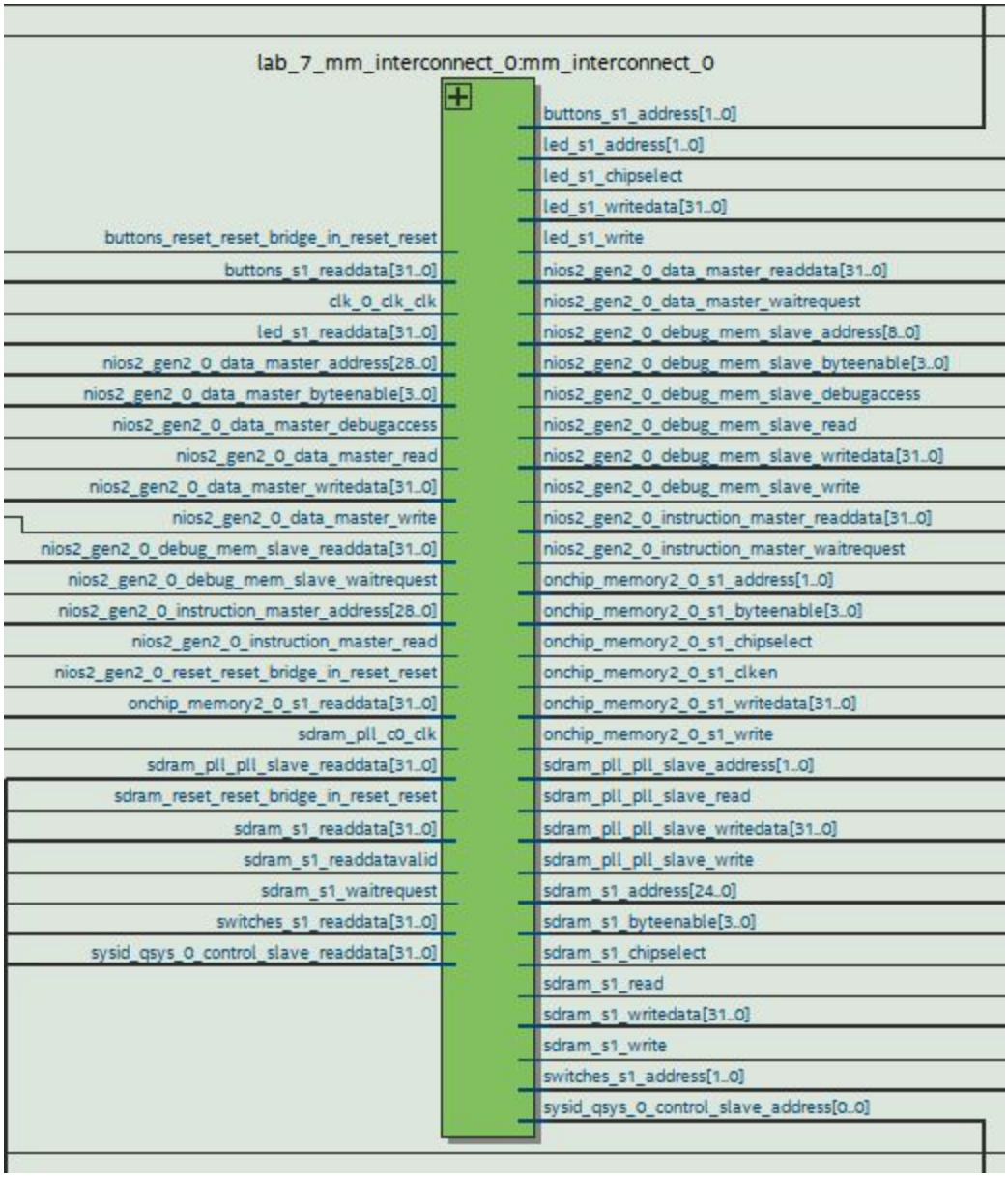


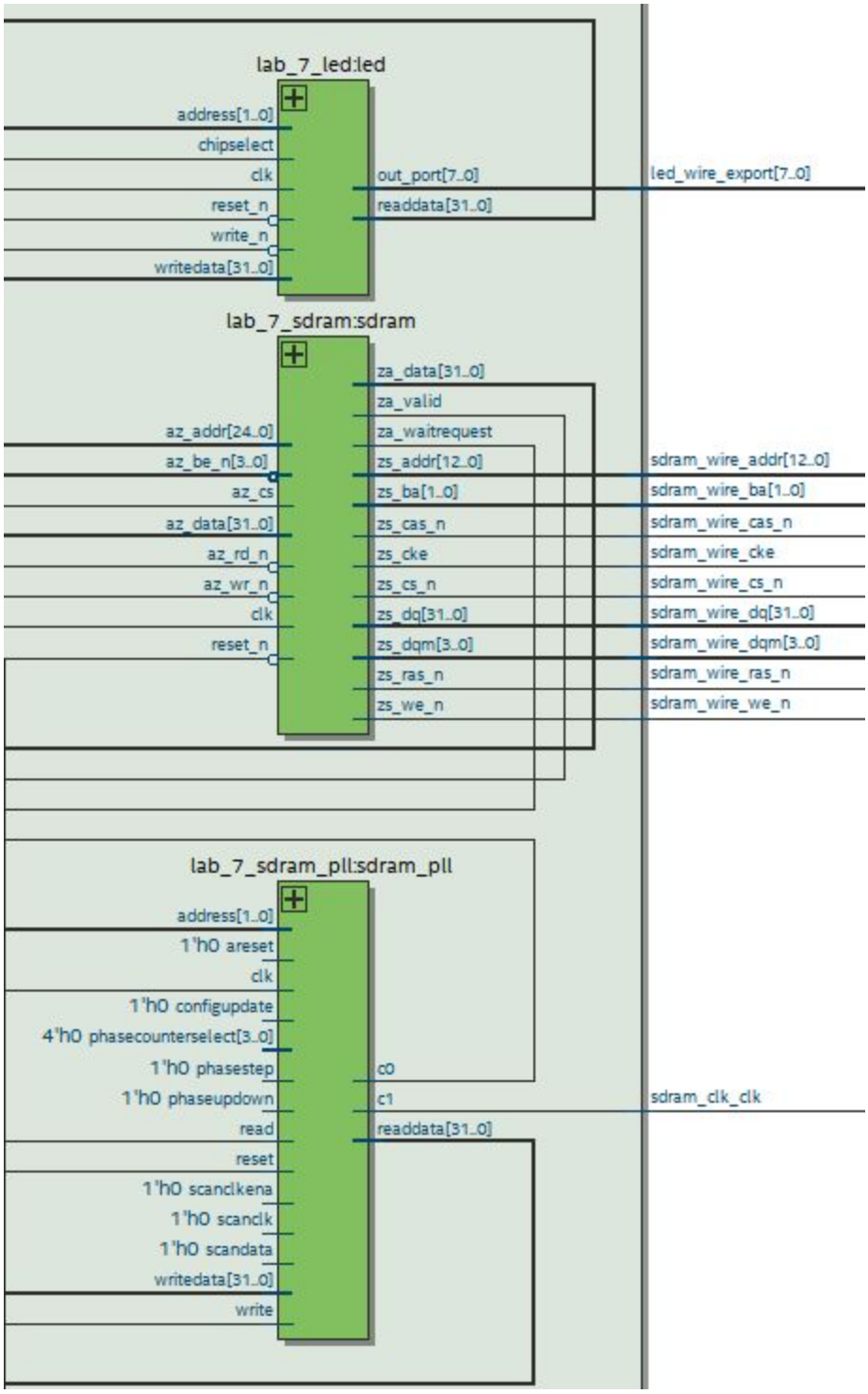

buttons_wires_export[1..0]
clk_clk
reset_reset_n
altera_reset_controller:rst_controller_001
clk
reset_in0
reset_in1
1'h0 reset_in2
1'h0 reset_in3
1'h0 reset_in4
1'h0 reset_in5
1'h0 reset_in6
1'h0 reset_in7
1'h0 reset_in8
1'h0 reset_in9
1'h0 reset_in10
1'h0 reset_in11
1'h0 reset_in12
1'h0 reset_in13
1'h0 reset_in14
1'h0 reset_in15
1'h0 reset_req_in0
1'h0 reset_req_in1
1'h0 reset_req_in2
1'h0 reset_req_in3
1'h0 reset_req_in4
1'h0 reset_req_in5
1'h0 reset_req_in6
1'h0 reset_req_in7
1'h0 reset_req_in8
1'h0 reset_req_in9
1'h0 reset_req_in10
1'h0 reset_req_in11
1'h0 reset_req_in12
1'h0 reset_req_in13
1'h0 reset_req_in14
1'h0 reset_req_in15
reset_out
reset_req
switch_wire_export[7..0]


lab_7_irq_mapper:irq_mapper
clk
reset
sender_irq[31..0]
altera_reset_controller:rst_controller
clk
reset_in0
1'h0 reset_in1
1'h0 reset_in2
1'h0 reset_in3
1'h0 reset_in4
1'h0 reset_in5
1'h0 reset_in6
1'h0 reset_in7
1'h0 reset_in8
1'h0 reset_in9
1'h0 reset_in10
1'h0 reset_in11
1'h0 reset_in12
1'h0 reset_in13
1'h0 reset_in14
1'h0 reset_in15
reset_out
1'h0 reset_req_in0
1'h0 reset_req_in1
1'h0 reset_req_in2
1'h0 reset_req_in3
1'h0 reset_req_in4
1'h0 reset_req_in5
1'h0 reset_req_in6
1'h0 reset_req_in7
1'h0 reset_req_in8
1'h0 reset_req_in9
1'h0 reset_req_in10
1'h0 reset_req_in11
1'h0 reset_req_in12
1'h0 reset_req_in13
1'h0 reset_req_in14
1'h0 reset_req_in15


altera_reset_controller:rst_controller_002
clk
reset_in0
reset_in1
1'h0 reset_in2
1'h0 reset_in3
1'h0 reset_in4
1'h0 reset_in5
1'h0 reset_in6
1'h0 reset_in7
1'h0 reset_in8
1'h0 reset_in9
1'h0 reset_in10
1'h0 reset_in11
1'h0 reset_in12
1'h0 reset_in13
1'h0 reset_in14
1'h0 reset_in15
1'h0 reset_req_in0
1'h0 reset_req_in1
1'h0 reset_req_in2
1'h0 reset_req_in3
1'h0 reset_req_in4
1'h0 reset_req_in5
1'h0 reset_req_in6
1'h0 reset_req_in7
1'h0 reset_req_in8
1'h0 reset_req_in9
1'h0 reset_req_in10
1'h0 reset_req_in11
1'h0 reset_req_in12
1'h0 reset_req_in13
1'h0 reset_req_in14
1'h0 reset_req_in15
reset_out


lab_7_nios2_gen2_0:nios2_gen2_0
clk
d_readdata[31..0]
d_waitrequest
debug_mem_slave_address[8..0]
debug_mem_slave_byteenable[3..0]
debug_mem_slave_debugaccess
debug_mem_slave_read
debug_mem_slave_writedata[31..0]
debug_mem_slave_write
i_readdata[31..0]
i_waitrequest
irq[31..0]
reset_n
reset_req
d_address[28..0]
d_byteenable[3..0]
d_read
d_writedata[31..0]
d_write
debug_mem_slave_debugaccess_to_roms
debug_mem_slave_readdata[31..0]
debug_mem_slave_waitrequest
debug_reset_request
i_address[28..0]
i_read
lab_7_sysid_qsys_0:sysid_qsys_0
address
clock
reset_n
readdata[31..0]
lab_7_switches:switches
address[1..0]
clk
in_port[7..0]
reset_n
readdata[31..0]


lab_7_mm_interconnect_0:mm_interconnect_0
buttons_reset_reset_bridge_in_reset_reset
buttons_s1_readdata[31..0]
clk_0_clk_clk
led_s1_readdata[31..0]
nios2_gen2_0_data_master_address[28..0]
nios2_gen2_0_data_master_byteenable[3..0]
nios2_gen2_0_data_master_debugaccess
nios2_gen2_0_data_master_read
nios2_gen2_0_data_master_writedata[31..0]
nios2_gen2_0_data_master_write
nios2_gen2_0_debug_mem_slave_readdata[31..0]
nios2_gen2_0_debug_mem_slave_waitrequest
nios2_gen2_0_instruction_master_address[28..0]
nios2_gen2_0_instruction_master_read
nios2_gen2_0_reset_reset_bridge_in_reset_reset
onchip_memory2_0_s1_readdata[31..0]
sdram_pll_c0_clk
sdram_pll_pll_slave_readdata[31..0]
sdram_reset_reset_bridge_in_reset_reset
sdram_s1_readdata[31..0]
sdram_s1_readdatavalid
sdram_s1_waitrequest
switches_s1_readdata[31..0]
sysid_qsys_0_control_slave_readdata[31..0]
buttons_s1_address[1..0]
led_s1_address[1..0]
led_s1_chipselect
led_s1_writedata[31..0]
led_s1_write
nios2_gen2_0_data_master_readdata[31..0]
nios2_gen2_0_data_master_waitrequest
nios2_gen2_0_debug_mem_slave_address[8..0]
nios2_gen2_0_debug_mem_slave_byteenable[3..0]
nios2_gen2_0_debug_mem_slave_debugaccess
nios2_gen2_0_debug_mem_slave_read
nios2_gen2_0_debug_mem_slave_writedata[31..0]
nios2_gen2_0_debug_mem_slave_write
nios2_gen2_0_instruction_master_readdata[31..0]
nios2_gen2_0_instruction_master_waitrequest
onchip_memory2_0_s1_address[1..0]
onchip_memory2_0_s1_byteenable[3..0]
onchip_memory2_0_s1_chipselect
onchip_memory2_0_s1_clken
onchip_memory2_0_s1_writedata[31..0]
onchip_memory2_0_s1_write
sdram_pll_pll_slave_address[1..0]
sdram_pll_pll_slave_read
sdram_pll_pll_slave_writedata[31..0]
sdram_pll_pll_slave_write
sdram_s1_address[24..0]
sdram_s1_byteenable[3..0]
sdram_s1_chipselect
sdram_s1_read
sdram_s1_writedata[31..0]
sdram_s1_write
switches_s1_address[1..0]
sysid_qsys_0_control_slave_address[0..0]



7.2.5 Answers to all INQ QuestionS

SDRAM Parameter	Value
Data width	32
# of Rows	13
# of Columns	10
# of Chip Selects	1
# of Banks	4

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

```
module lab7_soc (
        input  wire [1:0]  buttons_wires_export, // buttons_wires.export
        input  wire        clk_clk,              //           clk.clk
        output wire [7:0]  led_wire_export,      //      led_wire.export
        input  wire        reset_reset_n,        //         reset.reset_n
        output wire        sdram_clk_clk,        //     sdram_clk.clk
        output wire [12:0] sdram_wire_addr,      //    sdram_wire.addr
        output wire [1:0]  sdram_wire_ba,        //              .ba
        output wire        sdram_wire_cas_n,     //              .cas_n
        output wire        sdram_wire_cke,       //              .cke
        output wire        sdram_wire_cs_n,      //              .cs_n
        inout  wire [31:0] sdram_wire_dq,        //              .dq
        output wire [3:0]  sdram_wire_dqm,       //              .dqm
        output wire        sdram_wire_ras_n,     //              .ras_n
        output wire        sdram_wire_we_n,      //              .we_n
        input  wire [7:0]  switch_wire_export    //   switch_wire.export
    );
```

Nios II/e is the economy version of the Nios II/f processor. Nios II/f has additional things such as separate instruction/ data caches while the Nios II/e doesn't

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

On chip memory is more efficient. It will have higher read and write time due to shorter wires and compared to when compared to when data has to be processed through tristates and MUXs.

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

The design of a von Neumann architecture machine is simpler than a Harvard architecture machine which has one dedicated set of address and data buses for reading and writing to memory, and another set of address and data buses to fetch instructions. Von Neuman can only read OR write at once while ours can do both.

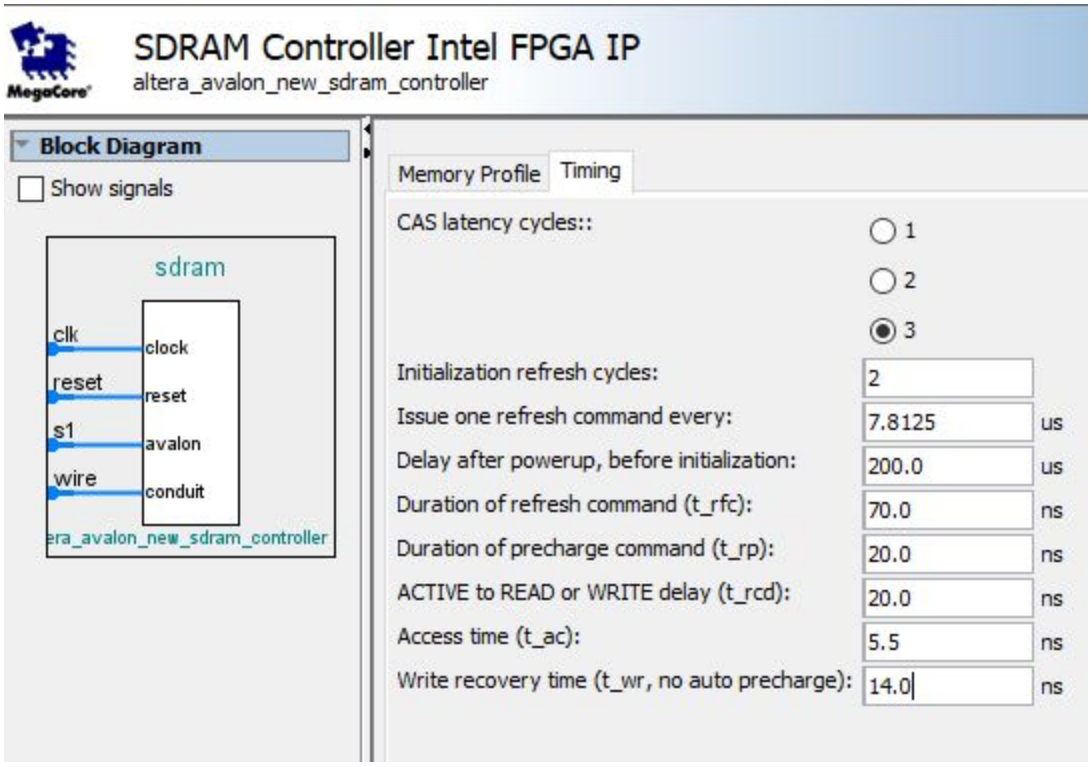
4. *Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?*

The peripheral only needs to receive data to display. It simply does not care about the data transferred on the on-chip memory.

5. *Why does SDRAM require constant refreshing?*

The data is stored within a capacitor by an electric charge. With time the charge goes away as the capacitor dissipates. So it needs to be refreshed so it holds the data.

6. *What is the maximum theoretical transfer rate to the SDRAM according to the timings given?*
   We know the SDRAM is 32 bits wide. We can extract the data from the timings in Platform Designer



For the theoretical max transfer rate we look at the access time- 5.5ns. 1/(5.5ns) = 181.81 Mhz, so 181.81Mhz*32 bits = 5.818Gb/s

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

The SDRAM needs to be refreshed and if the clock cycle is below 50MHz it will corrupt the data since it wasn't refreshed fast enough.

8. *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 behind of the controller clock (clk c0). Why do we need to do this?*

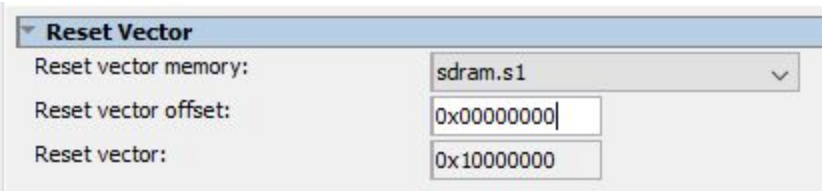
This 3ns "lag" allows glitches not to take place by allowing the inputs to be stabilized.

9. *Right click on nios2_qsys_0 and click Edit. For both reset and exception vectors, choose sdram.s1, as shown in Figure 17. Do not change default settings for offsets. Click Finish.*

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

The Nios II starts execution from the reset vector. We need to do this step after assigning the addresses because we need to dictate in a hardware reset what memory location we go back to - aka start.

Specifically, our reset vector is 0x10000000



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

.bss is used when a variable is not initialized to a value

int counter;

.heap is used when memory is allocated on the heap

malloc arr[ ]

.rodata is used when a variable is initialized

int value = 10;

.rwdata is used when for read/write data, which has a lot of zeros by default. The linker compresses data here

.stack is used when a stack segment is defined

.text is used when for the actual machine instructions

7.3 Post Lab Questions

7.3.1

Refer to the Design Resources and Statistics in IQT.30-32 and complete the following design statistics table.

LUT	2260
DSP	0
BRAM	36,864
Flip-Flop	1973
Frequency	96.96mW
Static Power	102.02mW
Dynamic Power	38.98mW
Total Power	53.42mW

7.3.2 ANY problems?

The main issues we had were simply a result of not being familiar with using this new interface and running into silly mistakes by missing an instruction or two which would cascade into more issues. Also another issue was matching port names throughout our files, ie. when we would press “Generate HDL” in platform designer we would have an instance type that did not match the instance type we defined in our lab7.sv. Also, we were initially confused at the .sdc file modifications because of all the confusing syntax but we eventually figured it out.

7.4 Conclusion

In conclusion this was an introduction to NIOs II, how to implement PIOs, and how to implement C code with the FPGA. One of the parts that I believe needed more explanation was perhaps the part where we had to fix the Timequest Timing Analyzer with the modification of

the SDC file. Perhaps some explanation of syntax would be nice. Other than that this lab was fairly easy and a great transition for the next segment of this course.