# ECE 385

Spring 2020

Experiment 7

# SOC with NIOS II in SystemVerilog

Michael Faitz and Zohair Ahmed
Section ABJ: Friday 2:00-4:50
Yuming Wu and Lian Yu

**Introduction**

a. The basic functionality of the NIOS II processor on the Cyclone IV FPGA is to allow us to write higher level code than SystemVerilog. This is called System on Chip, or SoC. We are able to do this by using the platform designer to bridge the NIOS II with the hardware on the FPGA, allowing us to write C code that writes to and reads from various elements on the FPGA. This way, we are able to write C code that makes the rightmost LED on the FPGA blink, as well as a program that accumulates numbers on switches at the press of a button.

**Written Description and Diagrams of NIOS-II System**

a. Summary of Operation
    i. The hardware component of the lab is to 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. We do this in platform designer, creating space for them in memory such that they can interact with both the SystemVerilog code and the C code. We set memory addresses for buttons, LEDs, and switches such that we know how to access them on the NIOS II, allowing us to read from the switches and buttons and write to the LEDs.
    ii. The blinker code is described in detail later in the report, but the basic functionality 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.
    iii. The accumulator takes the value of 8 switches and adds their value to the stored values in the LEDs. The sum is displayed in the LEDs, with the LEDs overflowing at 255. We accumulate when pressing a button on the FPGA. We avoid multiple accumulations in one press by adding a state variable to our code. The state 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. The loop that our code runs is infinite, and the switch, button, and LED variables are volatile as they cannot be run with the same optimization as the local variables.
b. Written Description of all .sv Modules
    i. A guide on how to do all this was shown in the Lab 5 report outline. Do not forget to describe the QSys generated file lab7soc.v.

Top Level Module:

```
module lab7(    input       CLOCK_50,
                            input  [3:0]  KEY,
                            input  [7:0]  SW,  //added the switches *****checkme
                            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: Holds all variables and instantiates the lab7_soc.

Purpose: Used to initialize lab7_soc and run the program.

Qsys generated soc module:

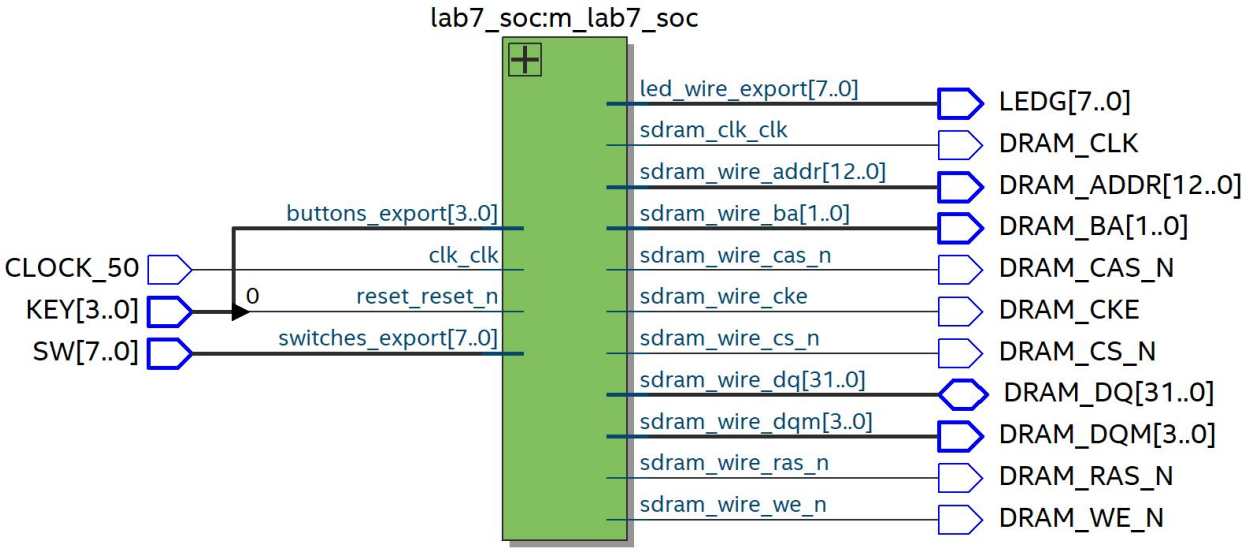
```
module lab7_soc (
        input  wire [3:0]  buttons_export,   //    buttons.export
        input  wire        clk_clk,          //        clk.clk
        output wire [7:0]  led_wire_export,  //   led_wire.export
        input  wire        reset_reset_n,    //      reset.reset_n
        output wire        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]  switches_export   //   switches.export
    );
```

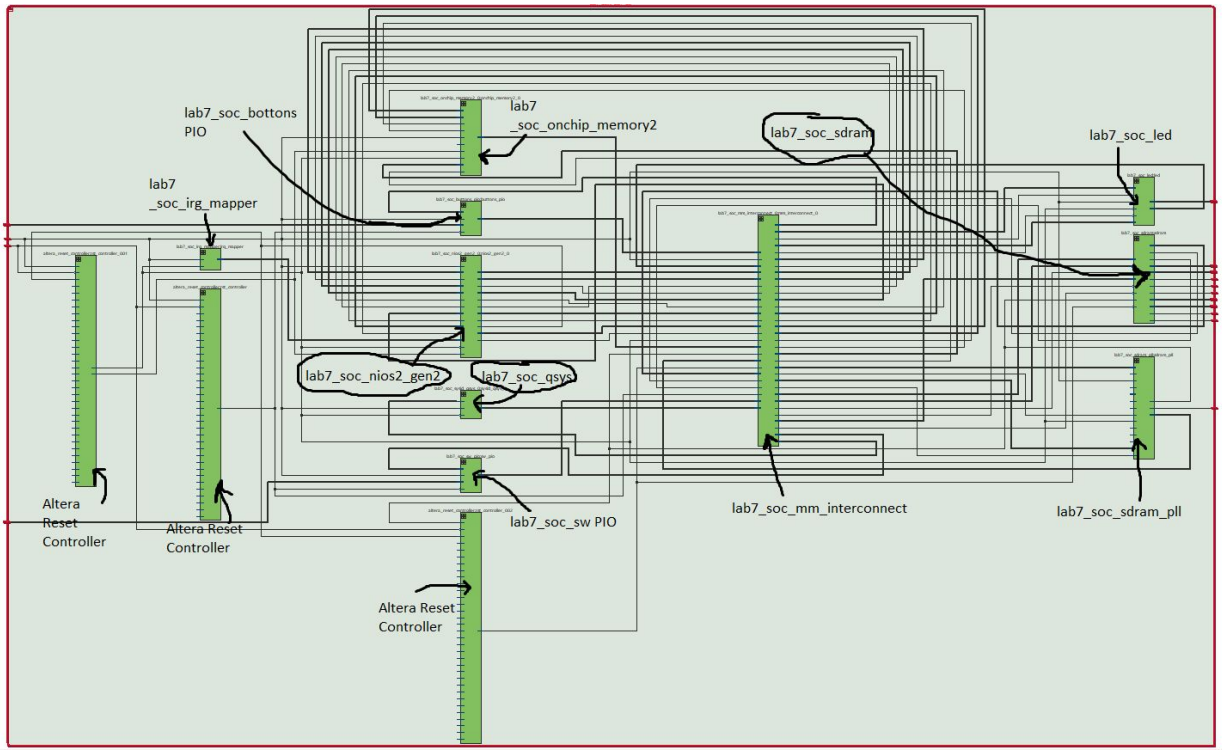
Description: Holds all modules and interfaces with Platform Designer such that the C code can read/write the variables located here.

Purpose: Used to read/write the FPGA switches, buttons and LEDs. The interface with Platform Designer allows the C code to read the FPGA switches and buttons and modify the LEDs.

c. Top Level Block Diagram

i. 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.




d. System Level Block Diagram

i. The QSys 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).

| Use | Connections | Name | Description | Export | Clock | Base | End | IRQ | Tags | Opcode Name |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 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_requ... | Reset Output | Double-click to export | [clk] | | | | | |
| | | debug_mem_slave | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_1000 | 0x0000_17ff | | | |
| | | custom_instructio... | Custom Instruction Master | Double-click to export | | | | | | |
| ☑ | | **onchip_memory2_0** | On-Chip Memory (RAM or ROM)... | | | | | | | |
| | | 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_0070 | 0x0000_007f | | | |
| | | external_connection | Conduit | **led_wire** | | | | | | |
| ☑ | | **sdram** | SDRAM Controller Intel FPGA IP | | | | | | | |
| | | clk | Clock Input | Double-click to export | **sdram_pl...** | | | | | |
| | | 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_inte... | | | | | |
| | | pll_slave | Avalon Memory Mapped Slave | Double-click to export | [inclk_inte... | 0x0000_0080 | 0x0000_008f | | | |
| | | c0 | Clock Output | Double-click to export | sdram_pll... | | | | | |
| | | c1 | Clock Output | **sdram_clk** | sdram_pll... | | | | | |
| ☑ | | **sysid_qsys_0** | System ID Peripheral Intel FPGA... | | | | | | | |
| | | 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_0098 | 0x0000_009f | | | |
| ☑ | | **sw_pio** | 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 | **switches** | | | | | | |
| ☑ | | **buttons_pio** | 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_0050 | 0x0000_005f | | | |
| | | external_connection | Conduit | **buttons** | | | | | | |

nios2_gen2_0 Block: The nios2 block handles the C code conversion over to system verilog which then is able  to be executed within the hardware FPGA board.

Onchip_memory2_0: The memory block acts as available hardware memory for variables needed to execute the C code onto the board. This effectively acts as a compliment to the nios2 processor.

Clk_0 (not shown in screenshot): Serves as the functional clock for the entire system and acts as a reference for the other modules.

led: This block serves as a PIO which allows for data to be updated from the C code and pushed to the board in a one directional manner so that they can be displayed on the FPGA.

sdram: This block allows us to get available SDRAM on the FPGA board since on-chip memory is too small for the available program to be stored and updated successfully.  SDRAM is useful due to its quick processing time and low update and output delay.

sdram_pll: This block is the method to account for the small delays within the transfer of the data in and out of the SDRAM and acts as a separate clock for the system.

sysid_qsys_0:  This block verifies the correct transfer of the software and hardware by looking back and forth between the C code and SystemVerilog to assure the data transfer is done in the correct format.

sw_pio: This block functions similarly to the led module as it functions as a direct variable transfer between the software and the hardware for the switches on the FPGA board. Unlike the led module, however, this PIO allows for bidirectional data transfer from the FPGA to software and visa versa.

buttons_pio: Similar in function to the switches mentioned above except the inputs are the buttons on the FPGA device.

e. Answers to all INQ Questions
    i. What are the differences between the Nios II/e and Nios II/f CPUs? Economy is focused on minimizing resources while f is focused on performance at the cost of using more resources.
    ii. What advantage might on-chip memory have for program execution? Processing data to and from memory on chip requires less time than going off chip and through the additional logic elements and buffers needed to ensure correct read/writes.
    iii. Note the bus connections coming from the NIOS II; is it a Von Neumann, “pure Harvard”, or “modified Harvard” machine and why? As we have multiple buses, the machine cannot be using a Von Neumann architecture. As the memory is all stored either on chip or in the SDRAM, we know there are two memory address spaces, meaning we use a pure Harvard Machine.
    iv. 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 on-chip memory is read write, while the LED peripheral is only write. The LED does not need to alter or store memory anywhere else in the program.
    v. Why does SDRAM require constant refreshing?
    The SDRAM needs constant refreshing in order to maintain its contents. This is due to the method the SDRAM uses to hold data, which will corrupt over time unless it is constantly refreshed.
    vi. What is the maximum theoretical transfer rate to the SDRAM according to the timings given? Access time = 5.5 ns. 1/(access time) = (transfer rate)/(bit) = 181.2 MHz/bit. Theoretical transfer rate = 32 bits * 181.2 MHz per bit = 5.8 GHz.
    vii. The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case? 50 MHz is the lowest frequency that the SDRAM can be refreshed without data corruption.
    viii. 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? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller. The 3ns window is enough to ensure that data reads from the bus are correct and not subject to corruption by data writes to the bus. We need this due to data stability issues with the SDRAM.
    ix. What address does the NIOS II start execution from? Why do we do this step after assigning the addresses? Our NIOS II begins execution at x10000000. This

step is done to ensure the program starts where the instructions are located, and that the program knows where to start after a reset.

x. 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, e.g. the code: const int my_constant[4] = {1, 2, 3, 4} will place 1, 2, 3, 4 into the .rodata segment.
.bss handles static variables that are not initialized to a value, static int x;
.heap handles memory allocations, int* m = (int *)malloc(1);
.rodata is read only data, const int i = 0;
.rwdata is read and write data, int * ptr;
.stack is used when a stack is involved, such as a function call on the program stack
.text is the segment of memory that handles executable instructions

xi. You will need to justify how you came up with 1Gbit to your TA.
$2^5$(width)*$2^{13}$(rows)*$2^{10}$(columns)*$2^0$(chip selects)*$2^2$(banks)*2(chips) = 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 |

xii. Explain what every line of the main.c code given does to your TA, specifically the volatile keyword (line 8), and the set and clear functions (lines 13 and 16).

1. Line 7 creates variable i, used as the counter, and puts it on the runtime stack.
2. Line 8 creates the volatile unsigned int * that stores the value to go to the LEDs. The volatile keyword tells the compiler not to make optimizations on this variable, as it may be changed outside the scope of the function.
3. Line 10 resets the LED pointer so that it's value is 0 when dereferenced.
4. Line 11 creates a while loop that is infinite given its conditions.
5. Line 13 creates a software delay such that we can see the LED switch from off to on.
6. Line 14 turns the rightmost LED on.
7. Line 15 creates a software delay such that we can see the LED switch from on to off.
8. Line 16 turns the leftmost LED off.
9. Line 18 holds the return statement for the function. It is never reached as we have an infinite loop, but is necessary for compilation.

f. The only problem we ran into was setting up the project, but that occurred because we misread the total memory size when setting up the platform designer. After we fixed that, we ran into no issues. The code we wrote was slightly tricky as we were not used to writing code that would be tested with hardware, but we were able to find a way to make our button act once with a pause/state variable.
g. Document the Design Resources and Statistics in table provided in the lab.

| | |
|---|---|
| LUT | 2267 |
| DSP | 0 |
| Memory (BRAM) | 36864 |
| Flip-Flop | 1985 |
| Frequency | 97.15 MHz |
| Static Power | 102.06 mW |
| Dynamic Power | 43.31 mW |
| Total Power | 204.78 mW |

**Conclusion**

a. We treated our buttons as a 3:0 logic element, which made the C code a bit more difficult to write. We used a pause variable in our C code so that we would only accumulate the switches once, setting it high after the accumulation and resetting it when the accumulation stopped.