

**ECE385**

Spring 2020

Experiment #7

## **SOC with NIOS II in SystemVerilog**

Yuantao Lu, Yizhen Lu

Section ABD

TA: Nicholas Cebry, Wenjie Pan

## **Introduction**

In this lab, we used the NIOS-II microprocessor running on the Cyclone IV FPGA to implement softwares controlled simple programs. We first made the processor and necessary components using the platform designer in Quartus. We specified necessary ports using Parallel I/O in this part. Then we generated HDL in the platform designer, which provides us with the specified modules in our design. We then connect the input signals for those modules that form the hardware part of this lab. Finally, we specify the software program we want to run on the hardware with Eclipse. For this lab, we controlled the LED on FPGA to blink. Then we implemented a simple accumulator that takes the buttons and switches as input and then accumulate the binary numbers on the button. The output is shown via the LED.

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

### *Hardware component summary*

The hardware is mainly composed of the NIOS-II microprocessor, the corresponding on-chip-memory, the SDRAM, the PIO, and the Avalon bus that connects all the above components. PLL is also used for better performance of the SDRAM. The NIOS-II microprocessor is the core for computing and processing. The on-chip-memory is the memory that is used for higher performance storage and load, as compared to SDRAM, which is used for relatively slower and larger data storage and load. The parallel I/O are ports connected to the Avalon bus that are used for data input and output. PLL is a clock that activates at several ns prior

to the main clock, resulting in better performance for SDRAM. The main work for this lab is to connect those hardware parts together.

### *Software component summary*

Once the hardware parts are loaded into FPGA, we could use high level programming languages to ask the hardware to perform specified tasks. In this lab, we used C programs that asked the hardware to blink the LED and perform the accumulating tasks. For each of those programs, we first generated BSP that contains the hardware information we just generated. The corresponding makefile was generated then. We could compile(build) the program and then run it on the hardware on FPGA.

For the blinking part, the code was just a main function with an infinite loop. We manually set a software delay for the blinking (iterate an empty for loop many times), after that we just set the led value corresponding to the hardware port address to the value we wanted. In this case, the PIO port corresponding to the led is the address we wanted. We just used a pointer for the port and changed the value of that pointer to control the led.

For the accumulator part, the code was also an infinite loop that checks the status of the accumulating and reset button each time. If the accumulating button was pressed but not released, the value on the switch will be added to the value already on the led and display it out one time. If the value exceeds 255, the value will overflow, so the value will always be the result mod 256. The same method was used like the blinking part to get the address of the value we would like to control. We used a pointer for the port and changed the components to control the hardware.

### *Written Description of all modules*

Module: *lab7.sv*

Inputs: *CLOCK\_50, [3:0] KEY, [7:0] SW, [7:0] LEDR, [7:0] LEDG* .

Outputs: *[12:0] DRAM\_ADDR, [1:0] DRAM\_BA, DRAM\_CAS\_N, DRAM\_CKE, [3:0] DRAM\_DQM, DRAM\_RAS\_N, DRAM\_WE\_N, DRAM\_CLK.*

Inout: *[31:0] DRAM\_DQ.*

Description: This module is the top level of this lab. All necessary inputs and outputs are specified.

Purpose: The module is the top level module for this lab. It not only gives a top level view but also specifies the general input and output for the hardware side of this lab.

Module: *lab7\_soc.v*

Inputs: *accu\_wire\_export, clk\_clk, clr\_wire\_export, [7:0] led\_wire\_export, [7:0] switch\_wire\_export, reset\_reset\_n.*

Outputs: *[7:0] led\_wire\_export, [7:0] ledr\_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, [3:0] sdram\_wire\_dqm, sdram\_wire\_ras\_n, sdram\_wire\_we\_n.*

Inout: *[31:0] sdram\_wire\_dq.*

Description: This module serves as the first layer from the top level. It connects all necessary hardware parts input and output for this lab. This module is also generated by the platform designer after the generating HDL.

Purpose: Inside this module, there will be several other modules that were described by the hardware summary description above. This module generally

connects the wire between those hardware components. It is generally the “top level” for the reset of the modules. However, all the inputs and outputs for this module is specified by the top level lab7.sv.

Module: *lab7\_soc\_accumulate.v*

Inputs: *[1:0] address, clk, in\_port, reset\_n*

Outputs: *[31:0] readdata*

Description: This module is generated by the platform designer, it is matched with the block *accumulate* in the system-level block diagram below.

Purpose: This module is the representation in verilog for the PIO block in the system-level diagram. We would use this block because we wanted to connect the port with the actual input signal at the top level.

Module: *lab7\_soc\_clear.v*

Inputs: *[1:0] address, clk, in\_port, reset\_n.*

Outputs: *[31:0] readdata.*

Description: This module is generated by the platform designer, it is matched with the block *clear* in the system-level block diagram below.

Purpose: This module is the representation in verilog for the PIO block in the system-level diagram. We would use this block because we wanted to connect the port with the actual input signal at the top level.

Module: *lab7\_soc\_led.v*

Inputs: *[1:0] address, clk, write\_n, reset\_n, chipselect, [31:0] writedata.*

Outputs: *[31:0] readdata, [7:0] out\_port.*

Description: This module is generated by the platform designer, it is matched with the block *led* in the system-level block diagram below.

Purpose: This module is the representation in verilog for the PIO block in the system-level diagram. We would use this block because we wanted to connect the port with the actual output signal at the top level. This PIO generally controls the LEDGs on the Cyclone IV FPGA board.

Module: *lab7\_soc\_nios2\_gen2\_0.v*

Input: *clk, reset\_n, reset\_req, [31:0] d\_readdata, d\_waitrequest, [31:0] i\_readdata, i\_waitrequest, [31:0] irq, [8:0] debug\_mem\_slave\_address, [3:0] debug\_mem\_slave\_byteenable, debug\_mem\_slave\_debugaccess, debug\_mem\_slave\_read, debug\_mem\_slave\_write, [31:0] debug\_mem\_slave\_writedata.*

Output: *[28:0] d\_address, [3:0] d\_byteenable, d\_read, d\_write, [31:0] d\_writedata, debug\_mem\_slave\_debugaccess\_to\_roms, [28:0] i\_address, i\_read, debug\_reset\_request, [31:0] debug\_mem\_slave\_readdata, debug\_mem\_slave\_waitrequest, dummy\_ci\_port.*

Description: This module is generated by the platform designer, it is matched with the block *nios2\_gen2\_0* in the system-level block diagram below.

Purpose: It's the representation in the verilog of the NIOS-II microprocessor which performs the essential processing in this lab.

Module: *lab7\_soc\_sdram\_input\_efifo\_module.v*

Inputs: *clk, wr, reset\_n, wr, [61:0] wr\_data*

Outputs: *almost\_empty, almost\_full, empty, full, [61:0] rd\_data.*

Description: This module is generated by the platform designer, it is matched with the block *sdram* in the system-level block diagram below.

Purpose: This module is the representation in verilog for the SDRAM in the system-level block diagram. It is connected at the top level with other components for this lab. See the system-level block diagram description for detail.

Module: *lab7\_soc\_sdram\_pll\_dffpipe\_l2c.v*

Inputs: *clk, clrn [0:0] d.*

Outputs: *[0:0] q.*

Description: This module is generated by the platform designer, it is matched with the block *sdram\_pll* in the system-level block diagram below.

Purpose: This module is the representation in verilog for the PLL for SDRAM in the system-level block diagram. It is connected at the top level with other components for this lab. See the system-level block diagram description for detail.

Module: *lab7\_soc\_switch.v*

Inputs: *[1:0] address, clk, [7:0] in\_port, reset\_n.*

Outputs: *[31:0] readdata.*

Description: This module is generated by the platform designer, it is matched with the block *switch* in the system-level block diagram below.

Purpose: This module is the representation in verilog for the PIO block in the system-level diagram. We would use this block because we wanted to connect the port with the actual input signal at the top level.

Module: *lab7\_soc\_sysid\_qsys\_0.v*

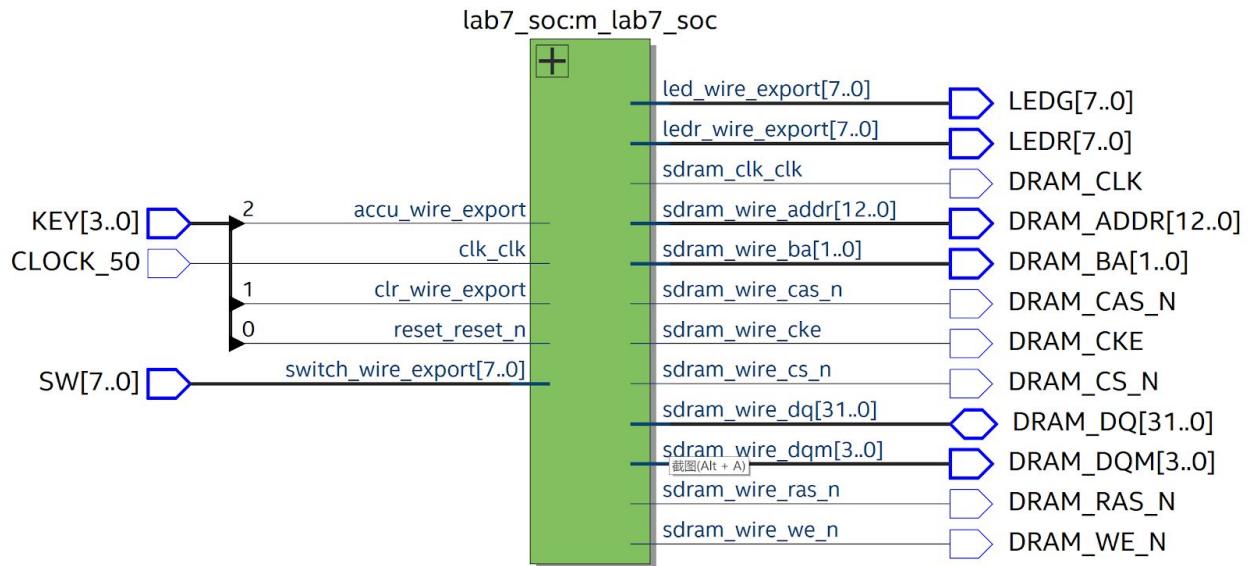
Inputs: *address, clk, reset\_n.*

Outputs: *[31:0] readdata.*

Description: This module is generated by the platform designer, it is matched with the block *sysid\_qsys\_0* in the system-level block diagram below.

Purpose: This module is the representation in verilog for the system id checker in the system-level block diagram. It is connected at the top level with other components for this lab. See the system-level block diagram description for detail.

*Top-level block diagram*



*Figure 1. Top level block diagram*

## System-level block diagram



Figure 2, 3. System level block diagram

Functions of each block in the System-level block diagram.

clk\_0:

This block generates the main clock signal for all other modules.

nios2\_gen2\_0:

This block represents the NIOS-II economic version. It is the microprocessor we used in this lab, which performs the computing and processing job in the hardware.

*onchip\_memory2\_0:*

It is the memory corresponding to the NIOS-II for loading and storing data that requires high-level performance.

*led:*

This block represents the parallel I/O module for LEDGs signals on the FPGA board. It is connected at the top level. Software could use the address of the port of this module to control the LEDGs.

*sdram:*

This block represents the sdram module. It makes a connection with the processor, so the system can use the memory specified by the sdram.

*sdram\_pll:*

This block represents the PLL for the system. It controls a clock with a phase shift of 3ns prior to the main clock. In this way, the sdram will start processing before the clock edge, and the date would be faster processed after the main clock edge.

*sysid\_qsys\_0:*

This block represents the system id checker. The purpose of this block is to check the configuration of the hardware part and the software part to make sure the two matches.

*switch:*

This block represents the parallel I/O port for switch input for FPGA. The switch is connected at the top level. Software could use the address of the port of this block to read the switch input from the board.

*ledr:*

This block is similar to led, but it controls LEDRs on the Cyclone IV FPGA board instead. This is also a PIO module for output, which can be controlled by software through the address of the port.

*clear:*

This block represents a PIO module for KEY input on the Cyclone IV FPGA board. It was connected on the top level for input KEY[1] from the board. The software could know the input from KEYS by reading the port address. This input was intended to clear the led output.

*accumulate:*

This block represents a PIO module for KEY input on the Cyclone IV FPGA board. It was connected on the top level for input KEY[2] from the board. The software could know the input from KEYS by reading the port address. This input was intended to accumulate the switch to the led.

*Answer to INQ questions*

*Q: What are the differences between the Nios II/e and Nios II/f CPUs?*

- “E” stands for the economic version of Nios II, and “F” stands for fast. Nios II/e is simpler and slower than Nios II/f CPU with minimal logic and memory.

*Q: What advantage might on-chip memory have for program execution?*

- It is closer to the processor, so it supports faster read and write.

*Q: Note the bus connections coming from the NIOS II; is it a Von Neumann, “pure Harvard”, or “modified Harvard” machine and why?*

- It is a modified Harvard. In modified harvard, the data and instructions are separated in memory, but instructions can be accessed as data.

*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?*

- On-chip memory not only stores data, but also stores the program. It will need to access the program bus to run the program, it will also need to access the data bus to read and write data. The led is just a PIO which only receives data from the board input or from the program output. It does not need any information about the program.

*Q: Why does SDRAM require constant refreshing?*

- A SDRAM uses capacitors, whose charge decays by time. So it will need constant refreshing.

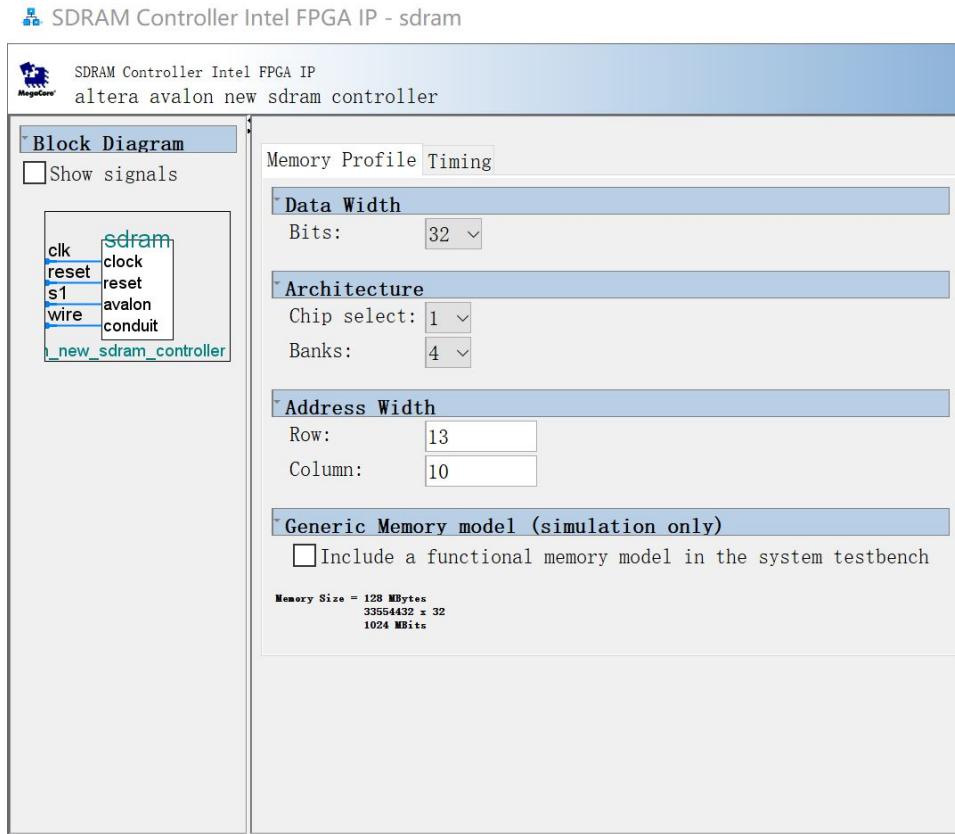
*Q: What is the maximum theoretical transfer rate to the SDRAM according to the timings given?*

- $1 / 5.5\text{ns} * 32\text{bit}/8 = 727.27\text{MB/s}$

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

- Because we need to constantly refresh the SDRAM or data will be lost.

*Q: Note that there are two 32M\*16 chips, so the total amount of memory should be 1Gbit (128 Mbytes), make sure this is consistent with your above numbers; you will need to justify how you came up with 1 Gbit to your TA.*



- $2^{13}(row) * 2^{10}(col) * 4(bank) * 32(bit) = 1024Mbits = 1Gbits$

*Q: 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.*

- Because latency caused by other signals to SDRAM is different from the main clock. To avoid the case that data was readed while it is not stable, we need 3ns for the data to stabilize.

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

- The execution starts from 0x10000000. We need to assign the ports before setting exceptions and reset vectors because the program has to know the address of the ports to start, or it will start at a nondeterministic state, and the address of PIO ports might be changed.

*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).*

```
1 // Main.c - makes LEDG0 on DE2-115 board blink if NIOS II is set up correctly
2
3 int main()
4 {
5     int i = 0;
6     volatile unsigned int *LED_PIO = (unsigned int*)0x20; //make a pointer to acc
7
8     *LED_PIO = 0; //clear all LEDs
9     while ( (1+1) != 3) //infinite loop
10    {
11        for (i = 0; i < 100000; i++); //software delay
12        *LED_PIO |= 0x1; //set LSB
13        for (i = 0; i < 100000; i++); //software delay
14        *LED_PIO &= ~0x1; //clear LSB
15    }
16    return 1; //never gets here
17 }
```

- The volatile keyword tells the program this variable value could be changed by other tasks outside of the current program at any time. In the compilation, optimization for this variable is omitted. It means that the value of this variable would always be read/write from/to the actual address without being read/write from/to temporary registers.

- LED\_PIO is the pointer to the actual led PIO port. Changing the value of this pointer would change the actual output of led on the board. The program is simply an infinite loop because we never want it to halt. Inside, we have software delays(the for-loops with nothing inside) to make blinking of LED human-readable. The value \*LED\_PIO was masked by the mask 0x01, which only changes the last bit of the value. When a value OR with 0x01, the last bit will be set to 1. When a value AND with ~0x01, the last bit will be set to 0. This is how the led blinks.

*Q: Look at the various segments (.bss, .heap, .rodata, .rwdta, .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 allocated variable with no initialization,

```
static int a
```

- .heap : heap, dynamically allocated memory,

```
int * ptr = (int*)malloc(sizeof(int))
```

- .rodata : read only data,

```
const int a = 0
```

- .rwdta : read and write data,

```
int a = 0
```

- .stack : stack frame for function calls,

```
int main(a){
```

```
    if (a <= 0) return a;
```

```
    main(a-1);
```

```

}

main(5);

//the information for main will be on the stack frame 5 times when main is
//called with 5.

```

- .text : code segment,

```

int main(a){
    if (a <= 0) return a;
    main(a-1);
}

```

//function int main(a){} will be stored as .text

### *Design Resources and Statistics(Postlab)*

|                   |            |
|-------------------|------------|
| LUT               | 2358       |
| DSP               | N/A        |
| Memory(BRAM)      | 36864 bits |
| Flip-Flop         | 2013       |
| Frequency         | 66.24MHz   |
| Static Power      | 102.06mW   |
| Dynamic Power     | 40.56mW    |
| I/O Thermal Power | 58.28mW    |
| Total Power       | 200.89mW   |

*Table 1. Design Resources and Statistics*

### **Conclusion**

In this lab, we explored Soc using quartus. We used the platform designer to design the hardware for this lab, and then used a software program written in C to control it. It is different from previous labs that are all hardwares. More interesting tasks would be achieved using Soc. For this lab, we made a led blinker and an accumulator with some additional features. For the accumulator, when the switches on the FPGA board are switched on, the corresponding LEDR will light up. When the accumulate button is pressed, the number on the switch will be added to the LED. When the reset button is pressed, the LED will be set to zero.