ECE 385

Fall 2021

Experiment # 7

SOC with NIOS II in SystemVerilog

Xu Ke / Zhu Xiaohan

LA4/Thursday & 18:00-20:50 Huang Tianhao

1. Introduction

In this lab, we get familiar with NIOs II, we learned how to use C code and PIO to do the modules

for FPGA. We learned how to use NIOS II processor, which is for the foundation of the System-On-Chips projects. The motivation we use SoC is that we want to use software to implement lower performance tasks. We successfully implement the simple SoC interfacing with peripherals such as LEDs and switches in this lab.

2. Written Description and Diagrams of NIOS-II System

Summary of Operation

a) Hardware component

We have switches and LEDs as input and output respectively. The switches are used to give numbers from 0-225 while LED shows the output. We also have two buttons, one for reset and one for accumulate. In order to deal with the overflow, we also implement peripheral input-output blocks, which used as an interface with other hardwares, in this lab, it's switches and LEDs.

b) Software component

We have our C codes to control our FPGA board. One of the codes can control the LED to blink and the other one can used to do adding.

(Code of Blink)

```
//blink
int i = 0;
volatile unsigned int *LED_PIO = (unsigned int*)0x0040;

*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
```

The volatile keyword tells the compiler that this object represents hardware which is accessed outside of the program, thus it might be changed anytime. It can prevent compiler from performing any optimization to that value.

The set function is used to set the first LED, which is the least significant bit to 1. And the clear function is used to set that bit to 0. Thus, after the software delay, the LED will change between 0 and 1, that is, it will blink.

(Code of Accumulator)

```
int i = 0;
volatile unsigned int *LED_PIO = (unsigned int*)0xa0; //
volatile unsigned int *sw = (unsigned int*)0x80;
volatile unsigned int *acc_reset = (unsigned int*)0x60;
volatile unsigned int *acc = (unsigned int*)0x70;

*LED_PIO = 0; //clear all LEDs
int pause = 0;
 volatile unsigned int sum = 0;
while ( (1+1) != 3) //infinite loop
{
    if(*acc_reset == 1){
        sum = 0;
    }
    if(*acc == 1 && pause == 0){
        sum += *sw;
        pause = 1;
    }
    if(*acc == 0){
        pause = 0;
    }
    *LED_PIO = sum;
```

The volatile keywords do the same function as I explained above. In this piece of code, we have 'sw' accessed by switches and 'acc_reset' & 'acc' accessed by two buttons.
Here, if we press reset, the sum will be set to 0. Then if we press acc button, we will add the switches number to sum, until we don't press it anymore. LED will always give the current output when doing accumulation.

Written Description of all .sv Modules

```
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

            );

            // You need to make sure that the port names here are identical to the port names at
            // the interface in lab7_soc.v
            lab7_soc m_lab7_soc (.clk_clk(CLOCK_50),
                                 .reset_reset_n(KEY[0]),
                                 .acc_reset_wire_export(~KEY[2]),
                                 .acc_wire_export(~KEY[3]),
                                 .switches_wire_export(SW),
                                 .led_wire_export(LEDG),
                                 .sdram_wire_addr(DRAM_ADDR),     //  sdram_wire.addr
                                 .sdram_wire_ba(DRAM_BA),         //  .ba
                                 .sdram_wire_cas_n(DRAM_CAS_N),   //  .cas_n
                                 .sdram_wire_cke(DRAM_CKE),       //  .cke
                                 .sdram_wire_cs_n(DRAM_CS_N),     //  .cs_n
                                 .sdram_wire_dq(DRAM_DQ),         //  .dq
                                 .sdram_wire_dqm(DRAM_DQM),       //  .dqm
                                 .sdram_wire_ras_n(DRAM_RAS_N),   //  .ras_n
                                 .sdram_wire_we_n(DRAM_WE_N),     //  .we_n
                                 .sdram_clk_clk(DRAM_CLK)         //  clock out to SDRAM from other PLL port
                                 );

          //Instantiate additional FPGA fabric modules as needed
endmodule
```

Module: lab7.sv
Input & Output: Shown in diagram
Description: This module is the toplevel of our lab7. It assigns all the inputs and outputs to the right place.
Purpose: This module is used to make FPGA and our code in Eclipse interact with each other.

```
module lab7_soc (
      input  wire        acc_reset_wire_export, // acc_reset_wire.export
      input  wire        acc_wire_export,       //       acc_wire.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_wire_export   //  switches_wire.export
   );
```

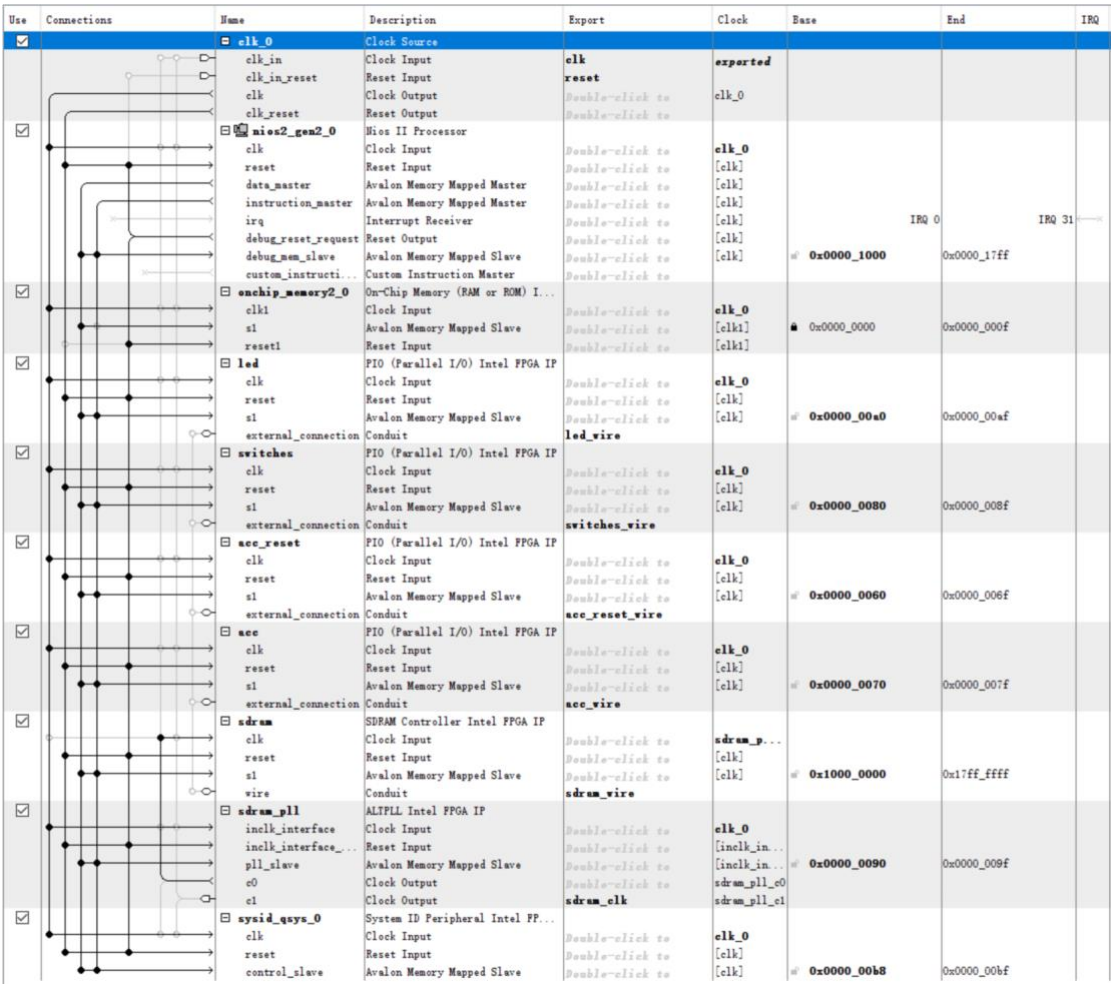
Module: lab_7.v

Input & Output: Shown in diagram

Description: This document is automatically generated by NIOS II. It can instantiate modules from NIOS II platform.

Purpose: The modules are used to interface those wires with top level, so that FPGA can execute them.

For the module in Platform Designer:



Module: clk_0

Input & Output: Shown in diagram

Description: This module is a 50Mhz clock generated by FPGA.

Purpose: It will be used in other modules as clock signal.

Module: nios2_gen2_0
Input & Output: Shown in diagram
Description: This is the actual processor (we use the Nios II/e).
Purpose: This module is responsible for processing all the instructions.

Module: onchip_memory2_0
Input & Output: Shown in diagram
Description: This module is the on-chip memory, which have data width of 32 bits and total memory size of 16 bytes.
Purpose: This module is used to for our on-chip memory.

Module: led (PIO)
Input & Output: Shown in diagram
Description: This module is a simple PIO block for our LEDs of data width of 8.
Purpose: This module is used for LED outputs.

Module: switches (PIO)
Input & Output: Shown in diagram
Description: This module is a simple PIO block for our switches of data width of 8.
Purpose: This module is used for switches inputs.

Module: switches (PIO)
Input & Output: Shown in diagram
Description: This module is a simple PIO block for our switches of data width of 8.
Purpose: This module is used for switches inputs.

Module: acc_reset (PIO)
Input & Output: Shown in diagram
Description: This module is a simple PIO block for our reset button of data width of 1.
Purpose: This module is used for reset button input.

Module: acc (PIO)
Input & Output: Shown in diagram
Description: This module is a simple PIO block for our accumulate button of data width of 1.
Purpose: This module is used for accumulate button input.

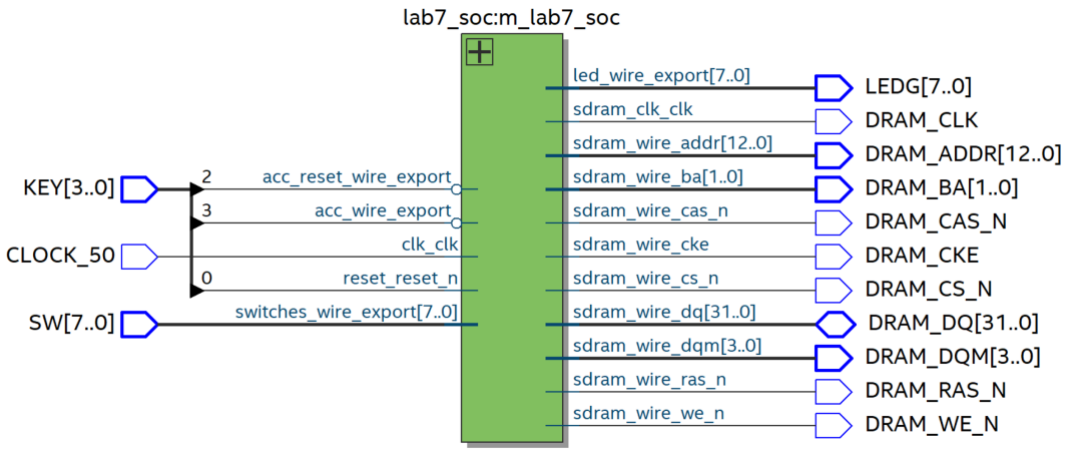
Module: sdram
Input & Output: Shown in diagram
Description: This module is our SDRAM that we are interfacing with.
Purpose: This module is used as SDRAM to store the software program since we don’t have enough space in the on-chip memory.

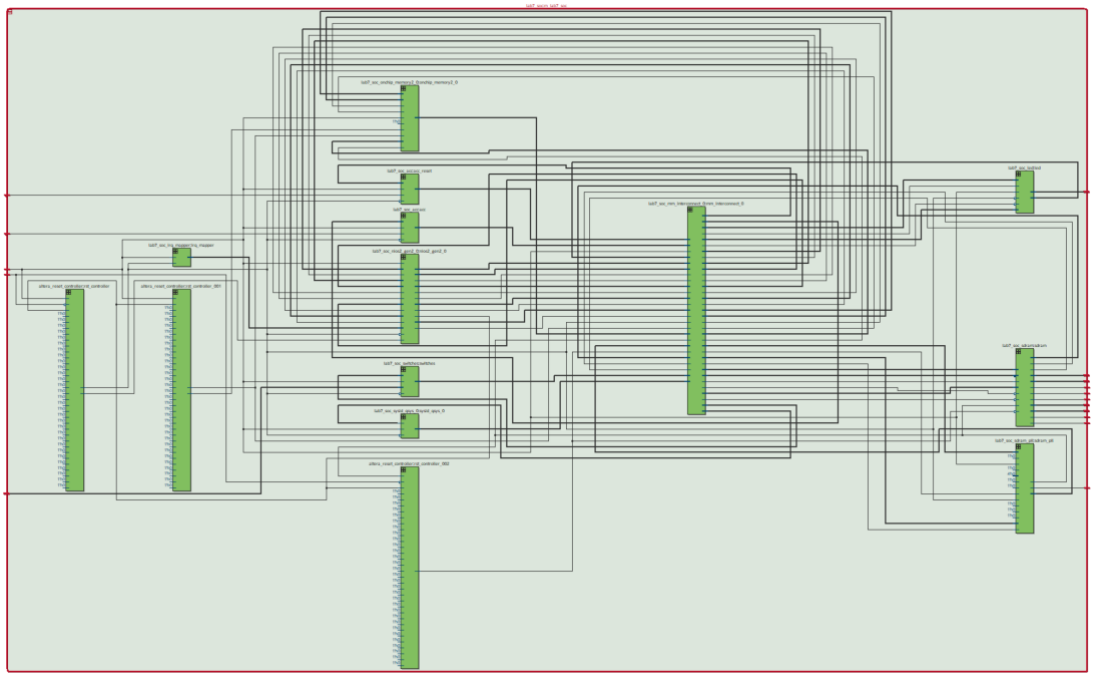
Module: sdram_pll

Input & Output: Shown in diagram
Description: This module generates the clock that goes into the SDRAM.
Purpose: This module is used PLL that allows for delays of 3ns so that SDRAM can wait for the outputs to stabilize.

Module: sysid_qsys_0
Input & Output: Shown in diagram
Description: This module is a system ID checker.
Purpose: This module can make hardware and software compatible. So that it can prevent incompatibilities when uploading NIOS II configurations to the FPGA.

Top Level Block Diagram





3. Answers to all 11 INQ Questions

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

On-chip memory has shorter wires so that it can read and write more quickly than using tristates and MUXs. So, it will be more efficiency.

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

It's not a Von Neumann machine since it can only read or write, while our can do both at the same time.

3. 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?

That is because peripheral only needs to receive data to display, it doesn't need to deal with data transferred on the on-chip memory.

4. Why does SDRAM require constant refreshing?

Because SDRAM use capacitors to store data, however, capacitor will dissipate over time, the data need to be refreshed to make sure it holds the correct data.

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

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

Memory of the chip = data width * 2^(# of Rows) * 2^(# of Columns) * (# of Chip Selects) * (# of Banks) = 32 * 2^13 *2^10 * 1 * 4 = 128Mbytes = 1Gbit

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

Since the SDRAM is 32 bit wide, we read the data from timings in Platform Designer that the access time is 5.5ns.

1/(5.5NS) = 181.81Mhz

So, 181.81Mhz * 32 = 5.818Gb/s

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

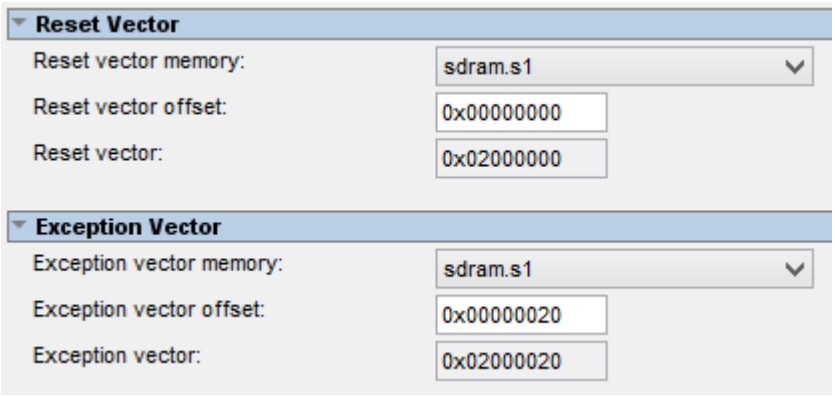
The SDRAM need to be refreshed at certain frequency, if the frequency is too slow, SDRAM will not be able to hold the data correctly.

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 ahead of the controller clock (clk c0). Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.

That is because it takes time for controller to get address, data, and control signals to be valid at SDRAM, we do need a clock that is few time delay of the controller clock.

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

NIOS II start execution from the start of the SDRAM, which we can see in the Reset Vector. For example, in this situation, we start at 0x02000000.



We need to assign the address first so that the processor knows where to start with when a hardware reset happen, so that it can make sure there is no memory overlap.

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). This question is referring to the blinker code.

We have already explained this question in previous part of report.

11. 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.

int counter;	(.bss is used when a variable is not specifically initialized)
malloc arr[]	(.heap is used when memory is allocated on the heap)
int value = 1;	(.rodata is used when a variable is initialized)
	(.rwdata is used for read and write data, where the linker compress data)
	(.stack is used when stack segment is define)
	(.text is used for the actual machine instructions)

4. Postlab Question

LUT	2242
DSP	0
Memory (BRAM)	36864
Flip-Flop	1999
Frequency	97.14MHz
Static Power	102.06mW
Dynamic Power	41.55mW
Total Power	204.99mW

5. Conclusion

Our design can either make LED blinks or do simple accumulation, it works well.
However, in the lab manual, it doesn't explain well about the specific function of each part of NIOs II, I cannot get the point of what this part doing until I finished the lab. The manual can add more pictures and explain some reasons why we need to do those steps, it basically only tell us how to complete those steps.