ECE 385

Fall 2019

Experiment #7

SOC with NIOS II in SystemVerilog

Eric Dong, Yifu Guo
ericd3, yifuguo3
Section AB3, Thursday 2pm
Gene Shiue

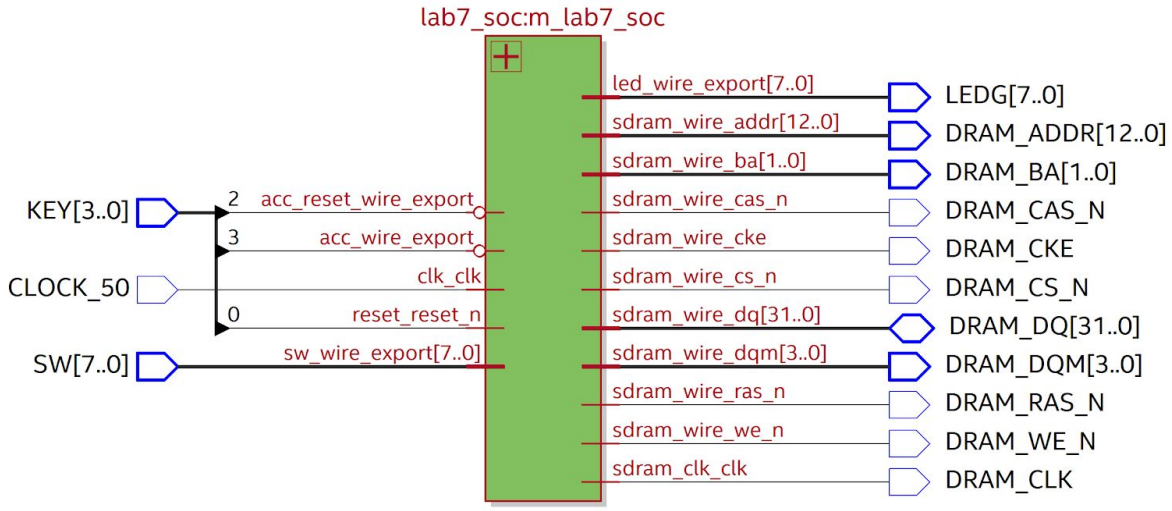
Introduction

In this lab, we created a SoC using the NIOS-II processor which allows us to write and compile C code on the FPGA and interface with parallel input and output devices such as LEDs as well as switches. As a result, we were able to make an on board LED blink as well as develop an accumulator program so that we can keep on adding numbers to a sum until it overflows.

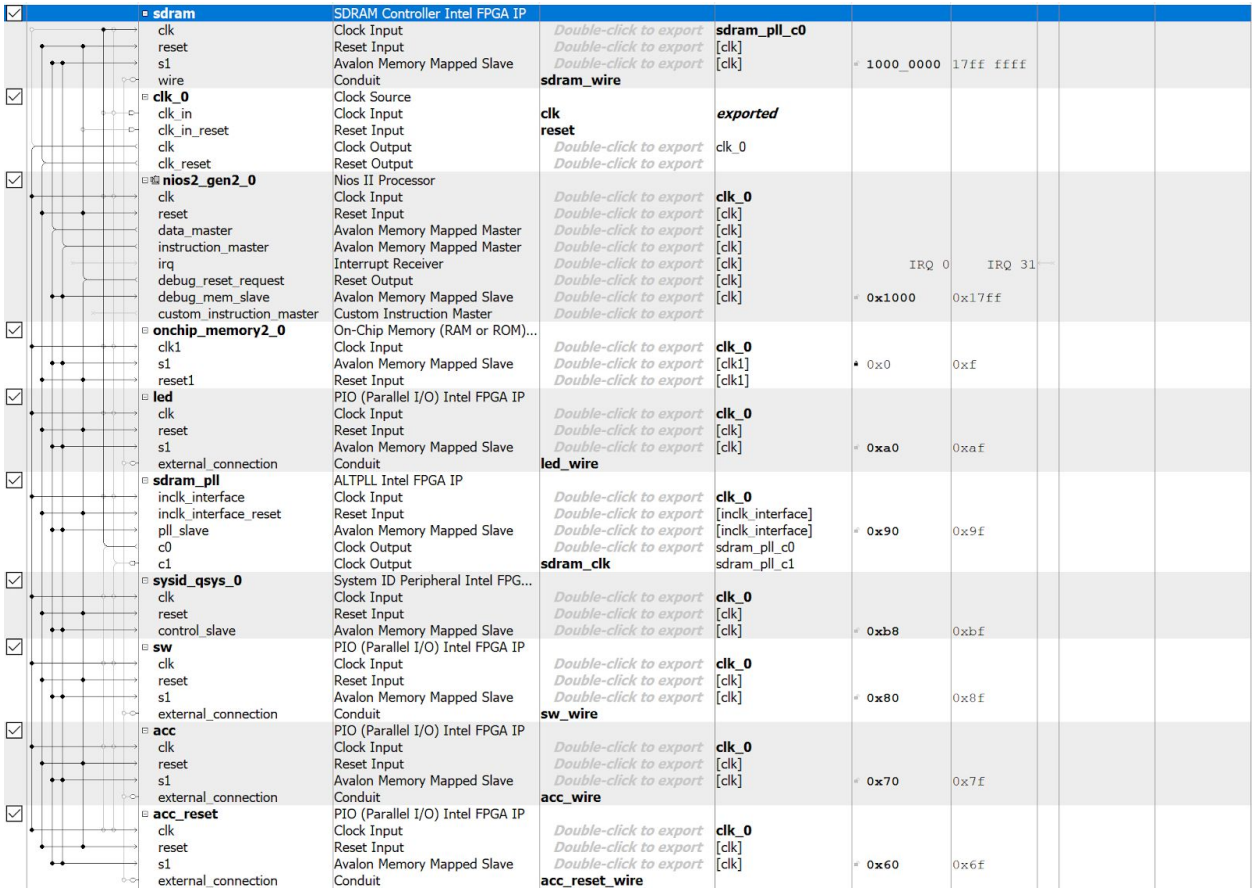
Written Description and Diagrams of NIOS-II System

- Summary of Operation
    - The hardware component of this lab consists of the NIOS-II processor itself as well as several other peripherals. We connected the processor to an sdram controller, on chip memory, as well as a clock phase shifter for the sdram. The sdram controller allows us to describe the dimensions of the on board sdram, such specs such as the number of rows/ columns, and the number of banks. The on chip memory is used to store valuable on chip memory, but here it is just a placeholder. Finally, the phase shifter is used to adjust the clock on the SDRAM so that the sdram chip can read data at the correct times.
    - Software Component:
      Blinker Code:
        - For the LED blinker code, we first define where the address of the LED PIO is located, there we can communicate with actual hardware. Then, we first clear the LED by writing to the memory a 0 to clear all 8 LEDs. Following, we declare an infinite while loop that executes the blinking. The two for loops are used as a software delay. The code "*LED_PIO |= 0x1;" means we or the 8 bit data stored at the LED memory location with a binary 1 to ensure that the LSB is going to be a 1 which turns on the right most LED. Similarly, the code "*LED_PIO &= ~0x1;" does a bit wise AND of the LED memory with a binary 0 to ensure that all the LEDs are turned off.
    - Accumulator Code:
        - For the accumulator code, we declared more volatile unsigned integer pointers such as the switch, accumulator reset as well as the accumulator to store the memory location of each hardware component. Next we declared a pause integer to ensure that each button press only executes once. In the while loop, we have 3 if statements. The first if statement means if the accumulator button is pressed, the sum is reset. The second if statement means if the accumulator button is pressed and the pause integer is 0 then we accumulate the amount of the switch to the sum, we then change the pause integer to a 1. The final if statement means if we release the accumulator button, we then change the value of the pause integer back to a 0. At the end of the code, we would then store the sum into the LED pointer to write the sum to the LED.
- Written Description of all .sv Modules
    - Module: lab7
        - Inputs: Clock_50, [3:0] KEY, [7:0] SW,

        - Outputs: [7:0] LEDG, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [31:0] DRAM_DQ, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
        - Description: This is the top level of the entire lab.
        - Purpose: This module declares a SoC module and connects all the hardware led, switches, as well as buttons with the SoC module. It also connects the DRAM with the on chip memory.
    - Module: lab7_soc
        - Inputs: clk_clk, acc_reset_wire_export, acc_wire_export, reset_reset_n, [7:0] sw_wire_export
        - Outputs: [7:0] led_wire_export, reset_reset_n, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [31:0] sdram_wire_dq, [3:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n,
        - Description: This is the System on Chip module that connects all the hardware together.
        - Purpose: This SoC module connects all the individual hardware components such as the LED, buttons, switches, processor, on chip memory, sdram controller, sdram clock phase shifter together.
- Top Level Block Diagram



- System Level Block Diagram



  - Sdram
    - This block controls the sdram module and tells the system the configuration of the sdram modules.
  - Clk_0
    - This is the clock source, it generates the clock for all the other modules.
  - Nios2_gen2_0
    - This is the Nios 2 processor which handles compiling the C code and interfacing with the hardware input outputs.
  - Onchip_memory2_0
    - This is the on chip memory which is supposed to be used to store valuable data / variables, but in this case it's only a placeholder.
  - Led
    - This is a PIO module which controls the on board LEDs.
  - Sdram_pll
    - This is a phase shifter for the sdram, it shifts the clock -3ns so that the sdram can read in the data at a better time.
  - Sysid_asys_0

        - This is a system ID checker which prevents us to load the software if the configuration is not compatible. It checks the compatibility between the software and the hardware.
    - Sw
        - This is a PIO module which connects to the switches and reads in the data from the switches.
    - Acc
        - This is a PIO module which connects to a button (KEY[3]). When the button is pressed, the system accumulates the sum and writes it to the LEDs.
    - Acc_reset
        - This is a PIO module which connects to a button(KEY[2]). When the button is pressed, the LEDs all clear and turn off.
- Answers

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

- The e variant is the small variant which is slower. It needs multiple clocks for 1 instruction, not fully pipeline, quite slow. It takes the least amount of resources, only 700 logic elements.

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

- The memory is much more efficient and read and write times are way faster since it is physically closer to the processor.

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

- Our design is Modified Havard since it allows the contents of the instruction memory to be accessed as data and also the addresses are changeable.
- Pure Havard: Data and instructions come in separate busses
- Von Neumann: Data and instructions come in same bus
- Modified Havard: allows the contents of the instruction memory to be accessed as data, or memory can access each other, or caches are different

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?

- The LED needs access to only the data bus because it is only an output peripheral, it has nothing to do with the program whereas the on-chip memory is part of the program and has to know where to read data and write data. The LED only receives data from the board.

Q: Why does SDRAM require constant refreshing?

- If it isn’t refreshed, then the data might decay in the ram modules since they are capacitors and the charge might leak out

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.

- 32(bits per address) * 2^{13}(num of rows/bank) * 2^{10}(num of columns/bank) * 2(chips) * 2^{2}(banks/chip)
- 32Mbits * 16addresses * 2 chips = 1Gbit

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

- 1/(5.5s(access time)) * 32M = 720MB/s

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

- When the SDRAM is run too slowly, the chip is not able to refresh often enough to prevent data corruption

Q: This puts the clock going out to the SDRAM chip (clk c1) 3ns behind the controller clock (clk c0). Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.

- There are many latency issues such as the SDRAM CAS latency, this might cause the SDRAM to read in data from the bus when the data is about to change, so we need to shift the clock so it reads in the correct value.

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

- Execution starts from memory location 0x10000000. We do this step after assigning the addresses since the processor has to know where to start the program as well where to go back to when the program is reset. We set it so that we don't change the addresses of the PIO blocks. The system can start in a definite state.

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).

- Blinker Code:
    - For the LED blinker code, we first define where the address of the LED PIO is located, there we can communicate with actual hardware. Then, we first clear the LED by writing to the memory a 0 to clear all 8 LEDs. Following, we declare an infinite while loop that executes the blinking. The two for loops are used as a software delay. The code "*LED_PIO |= 0x1;" means we or the 8 bit data stored at the LED memory location with a binary 1 to ensure that the LSB is going to be a 1 which turns on the right most LED. Similarly, the code "*LED_PIO &= ~0x1;" does a bit wise AND of the LED memory with a binary 0 to ensure that all the LEDs are turned off.
- Volatile - this tells the compiler that this data represents some hardware, something that can be accessed outside of the program. Otherwise, the program sees that we are only writing to that location without ever reading from it, then it might optimize it and never actually write. The volatile makes the C program actually write to that address. In the other case, it might never actually read from that address, and only reads from the cache. Since the value can change based on the hardware, we want the compiler to actually read from the memory location.

Q: 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.

.bss	static int x = 0;	Global variable
.heap	int* ptr = (int*)malloc(sizeOf(int) * 2);	heap
.rodata	const int x = 0;	Read only data
.rwdata	int x = 0;	r/w data
.stack	int x = getInt();	Stack implementation/ function call
.text	char[] str = “myString”	text/strings

- Post Lab Question

LUT	3,186
DSP	0
Memory	10,368
Flip-Flop	1983
Frequency	73.88 MHz
Static Power	102.03 mW
Dynamic Power	40.11 mW
Total Power	195.56 mW

- Design Resources and Statistics

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

Conclusion

- The lab was oriented around trying to integrate a SoC onto our FPGA and write software to the SoC. We did not run into many troubles during the lab, everything went well as we just followed the instructions on the manual.
- This lab was really straight forward and there weren't too many confusing parts. I do think that the lab manual should tell us what each part of the platform designer was and how it correlates to all other parts which would make us learn even more instead of trying to let us figure that out.