

Lab 7 Report

ECE 385

Friday (ABB)

11:00AM-1:50PM

SOC with NIOS II in SystemVerilog

**Ritvik Avancha and Steven Phan
(rra2 and sphan5)**

Table of Contents

| | |
|---|----------|
| Introduction | 2 |
| Written Description and Diagrams of NIOS-II System | 2 |
| Summary | 2 |
| Module Descriptions | 3 |
| Figure 2.1: Top Level Block Diagram | 3 |
| Figure 2.2: System Level Block Diagram | 4 |
| QSYS Block Descriptions | 4 |
| INQ Questions | 5 |
| Figure 2.3: Sample Program | 6 |
| Postlab | 7 |
| Figure 2.4: Resources and Statistics | 7 |
| Conclusion | 7 |

Lab 7 Report

Introduction

This lab teaches us the fundamentals of the NIOS II Processor (which 32-bit which implies limitations) running on the FPGA board in order to be able to program at a higher level language than SystemVerilog, specifically C. In doing so, we are able to abstract away non high-performance tasks and use the NIOS II system to handle those tasks (user interface, data input/output, etc.) whereas high-performance tasks are left to FPGA logic designed using SystemVerilog. In this lab specifically, we are asked to build an adder: the NIOS II handles inputting data, reset signals, add signals, interfacing with the output LED which are all relatively not high-performance tasks.

Written Description and Diagrams of NIOS-II System

Summary

In the Introduction to NIOS II and Platform Designer (formerly QSYS), we learned how to instantiate IP blocks (including the NIOS II block) and we set up a basic hardware reset button, LED output blocks, SDRAM, and so on. Each block had a set of interconnections that could be connected by clicking on the wire we wanted to connect to other blocks. We also needed to ensure that we assigned memory-mapped addresses to relevant blocks in order to be able to interface them with a higher-level language like C in NIOS II. After the tutorial (which uses given C code to make an LED blink), we added more PIO blocks to make an ADDER with basic functionality (ADD the switches to the current sum, RESET which clears the current sum). In this lab, the hardware consists of the blocks we made on QSYS (Figure 2.2 with descriptions). The software side of this lab was the blinking code and the accumulator code. The way the accumulator code works is that it has pointers mapped to the PIO blocks. Then, we have an infinite while loop. Inside the loop, there are 3 main conditions: if reset is pressed, if accumulate is not pressed, and accumulate is pressed. Since the buttons are active low, a “pressed” state corresponds to “0”. A “stop” variable is introduced to avoid having multiple additions in the case where ACCUM is held for a long time. “RESET” sets sum = 0, “ACCUM” adds the switches to the sum and makes stop = 1 in order to prevent entering that condition again.

Module Descriptions

Module: **lab7.sv**

Inputs: CLOCK_50, [3:0] KEY, [7:0] SW

Inout: [31:0] DRAM_DQ

Outputs: [7:0] LEDG, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK, [3:0] DRAM_DQM

Description: Top level module for lab 7

Purpose: Manages IO between the board and the SOC module

Module: **lab7_soc.sv**

Inputs: accum_export, accum_reset_export, clk_clk, reset_reset_n, [7:0] sw_export

Inout: [15:0] sdram_wire_dq

Outputs: [7:0] led_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, [1:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n

Description: Platform designer generated hardware. It instantiates the NIOS 2 processor, buttons, switches, SDRAM, LEDs, and other off-FPGA parts.

Purpose: Used to connect the FPGA and the other components on the board and instantiate them

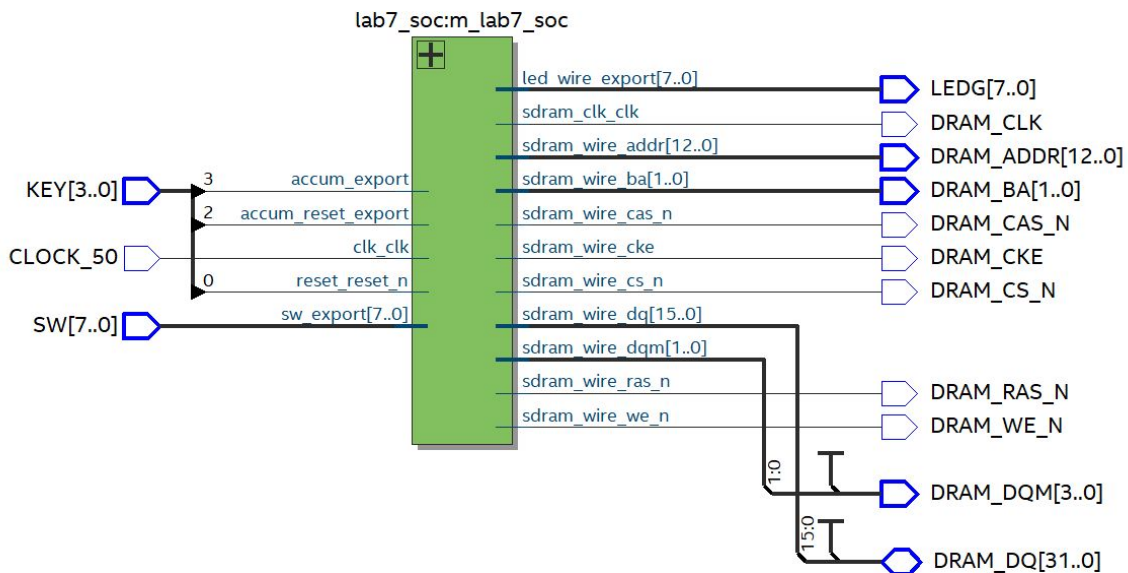


Figure 2.1: Top Level Block Diagram

| Use | Connections | Name | Description | Export | Clock | Base | End |
|-----|-------------|---|--|--|--|-----------------|--------------|
| ✓ | | clk_0 | Clock Source | | | | |
| ✓ | | clk_in clk_in_reset clk clk_reset | Clock Input Reset Input Clock Output Reset Output | clk reset <i>Double-click to</i> <i>Double-click to</i> | exported clk_0 | | |
| ✓ | | nios2_gen2_0 clk reset data_master instruction_m... irq debug_reset_r... debug_mem... custom_instru... onchip_mem... clk1 s1 reset1 | Nios II Processor Clock Input Reset Input Avalon Memory Mapped ... Avalon Memory Mapped ... Interrupt Receiver Reset Output Avalon Memory Mapped ... Custom Instruction Master On-Chip Memory (RAM o... Clock Input Avalon Memory Mapped ... Reset Input | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | clk_0 [clk] [clk] [clk] [clk] [clk] [clk] [clk] | IRQ 0 IRQ 31 | |
| ✓ | | onchip_mem_0 clk s1 reset1 | Clock Input Avalon Memory Mapped ... Reset Input | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | clk_0 [clk1] [clk1] | # 0x0000_0000 | 0x0000_000f |
| ✓ | | led clk reset s1 external_conn... | PIO (Parallel I/O) Intel F... Clock Input Reset Input Avalon Memory Mapped ... Conduit | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | clk_0 [clk] [clk] [clk] | # 0x0000_0090 | 0x0000_009f |
| ✓ | | sdram clk reset s1 wire | SDRAM Controller Intel F... Clock Input Reset Input Avalon Memory Mapped ... Conduit | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | sdram... [clk] [clk] | # 0x0800_0000 | 0x0bfff_ffff |
| ✓ | | sdram_pll indk_interface indk_interface... pll_slave c0 c1 | ALTPLL Intel FPGA IP Clock Input Reset Input Avalon Memory Mapped ... Clock Output Clock Output | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | clk_0 [indk_i...] [indk_i...] sdram... sdram ... | # 0x0000_00a0 | 0x0000_00af |
| ✓ | | sysid_qsys_0 clk reset control_slave | System ID Peripheral Inte... Clock Input Reset Input Avalon Memory Mapped ... | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | clk_0 [clk] [clk] | # 0x0000_00b8 | 0x0000_00bf |
| ✓ | | SW clk reset s1 external_conn... | PIO (Parallel I/O) Intel F... Clock Input Reset Input Avalon Memory Mapped ... Conduit | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | clk_0 [clk] [clk] [clk] | # 0x0000_0080 | 0x0000_008f |
| ✓ | | ACCUM clk reset s1 external_conn... | PIO (Parallel I/O) Intel F... Clock Input Reset Input Avalon Memory Mapped ... Conduit | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | clk_0 [clk] [clk] [clk] | # 0x0000_0070 | 0x0000_007f |
| ✓ | | RESET clk reset s1 external_conn... | PIO (Parallel I/O) Intel F... Clock Input Reset Input Avalon Memory Mapped ... Conduit | <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> | clk_0 [clk] [clk] [clk] | # 0x0000_0060 | 0x0000_006f |

Figure 2.2: System Level Block Diagram

QSYS Block Descriptions

clk_50: A 50 MHz clock used to clock all the synchronous components of the design

nios2_gen_2_0: NIOS 2 processor that executes the C code stored in SDRAM

onchip_memory2_0: NIOS 2's 16 byte memory

led: A PIO block that maps NIOS 2 processor to the green led on the board (allows code to drive LED pins)

sdram: Memory space used to store instructions and general storage

sdram_pll: Generates a 2nd clock that is out of phase with the main clock in order to ensure that when SDRAM refreshes, the input ports have time to stabilize

sysid_qsys_0: verification module for interfacing code with the board. It ensures that the code that is trying to get on the is meant for that board

SW: A PIO block that maps NIOS 2 processor to the slider switches on the board (allows switches to interface with the code directly, i.e. to indicate how much to add to current sum)

ACCUM: A PIO block that maps NIOS 2 processor to a push button on the board (allows button to directly affect code, i.e add to the current sum the values of the switches)

RESET: A PIO block that maps NIOS 2 processor to a push button on the board (allows button to directly affect code, i.e. clear the current sum on the LEDs)

INQ Questions

1. $f \rightarrow$ fast variant: runs faster and has more support for industry standard protocols
 $e \rightarrow$ economy variant: runs cooler and slower

In addition to performance, E is meant to be more efficient compared to F (which also has three multipliers and cache implemented internally, among other things as well).

2. The on-chip memory is physically closer so inherently there's lower latency.
3. Modified Harvard because the program code was preloaded and treated as data.

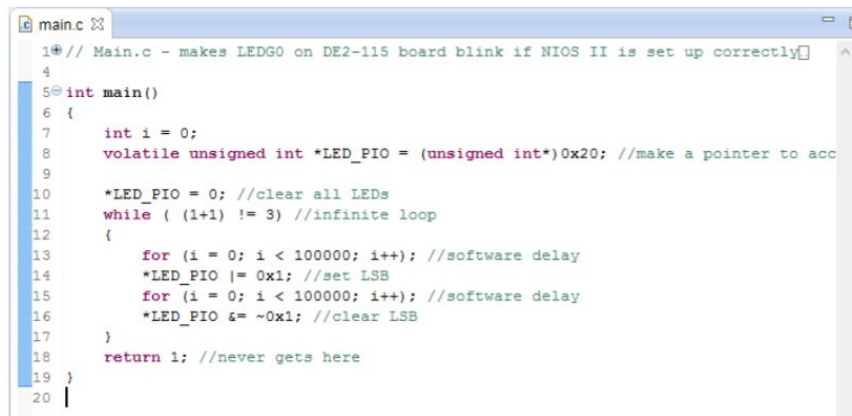
Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?

4. LED peripheral only needs access to data bus because it's an output, so it doesn't need program bus access.
5. SDRAM needs to be constantly refreshed because data is stored in capacitors. So, inherently capacitors discharge and leak data. Without refreshing, this data is lost indefinitely, hence why SDRAM is a short term memory.

6.

| 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 |

7. $(1/(5.5\text{ns})) * (32\text{M(bits/s)}/8(\text{bits/byte})) = 727 \text{ MB/s}$
8. The SDRAM should run at the same frequency as the SDRAM controller. If the SDRAM frequency is too slow, the functionality of the system can be affected. One reason for this loss in functionality is due to the way SDRAM stores data (losing data if refresh rate is too low). If the frequency is too high, there is unnecessary refreshing and if its too slow data is lost.
9. The delay is placed in order to account for real time PCB trace delay. -3ns was determined by various on board testing by the board manufacturer.
10. 0x10000000. We do this step after assigning the addresses in order to make sure the processor can handle requests such as reset once it begins execution and because SDRAM only maps to a part of the address space.



```

1 // Main.c - makes LEDG0 on DE2-115 board blink if NIOS II is set up correctly
4
5 int main()
6 {
7     int i = 0;
8     volatile unsigned int *LED_PIO = (unsigned int*)0x20; //make a pointer to acc
9
10    *LED_PIO = 0; //clear all LEDs
11    while ( (1+1) != 3) //infinite loop
12    {
13        for (i = 0; i < 100000; i++); //software delay
14        *LED_PIO |= 0x1; //set LSB
15        for (i = 0; i < 100000; i++); //software delay
16        *LED_PIO &= ~0x1; //clear LSB
17    }
18    return 1; //never gets here
19 }
20

```

Figure 2.3: Sample Program

11. Volatile - tells the compiler to not worry about optimizing this assignment since other drivers might change its values besides the actual program itself.
Set - ORs the LSB of LED with 1 to make it 1 since anything OR 1 is 1.
Clear - ANDs the LSB of LED with 0 to make it 0 since anything AND 0 is 0.
12. **.bss** is for uninitialized data. It contains all global and static variables that are initially 0 or not declared explicitly in source code. Ex: static int var;. This places var into the .bss data segment.
.heap is the system's memory space and can be managed in C using calls like malloc, calloc, realloc, and free. It can be accessed directly via pointers from various functions,

libraries, threads, etc. Ex: `int *myIntPtr = new Int();` //new keyword means heap. This places the new `Int()` into the `.heap` data segment.

.rodata is reserved for constant variables (read-only, no access after declaration). Ex: `const int my_constant[3] = {1,2,3};`. This places 1,2,3 into `.rodata` data segment

.rwdata is reserved for read-write variables (read and write access upon initial declaration). Ex: `int myInt = 4;`. This places 4 with into `.rwdata` segment

.stack is a LIFO structure and is used to keep track of bookkeeping information for functions/function calls and local variables. Ex: `int myInt1 = 1, myInt2 = 2;` places `myInt1` into the stack data segment first, then `myInt2` onto the stack data segment. The same idea can be applied to the order functions are called. Main is always the base of the stack in a C program.

.text is the portion of the object file or the section of the program's address space that has the executable instructions stored. `int x = myInt1 + myInt2;` stores the instruction of adding two integers into the `.text` data segment.

Postlab

| | |
|--------------------|--------|
| LUT | 2,312 |
| DSP | 0 |
| Memory | 36,864 |
| Flip-Flop | 1,754 |
| Frequency (MHz) | 68.45 |
| Static Power (mW) | 102.02 |
| Dynamic Power (mW) | 38.48 |
| Total Power (mW) | 193.60 |

Figure 2.4: Resources and Statistics

Conclusion

The NIOS II adder built in this lab functioned as expected as it allowed for data, reset, and add functionality to interface with the output LEDs. The general use of the NIOS II system is to allow for lower ermonace tasks to be abstracted through the use of C code.