**ECE 385**

Fall 2019

Experiment #7

SOC with NIOS II in SystemVerilog

Jacqueline Jiang and JiHo Han

AB6 / Wednesday 12:00 - 2:50 p.m.

Nicholas Cebry and Yujie Zhou

# Introduction

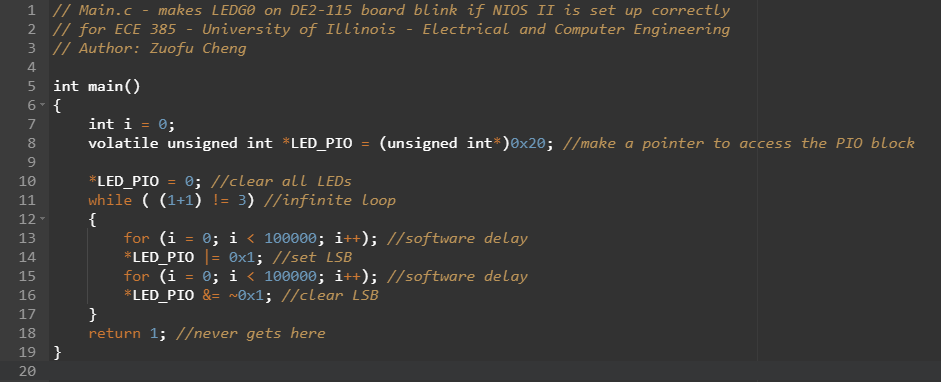
## Summary of the Basic Functionality of the NIOS-II Processor

In this experiment, we created a working NIOS-II based System-On-Chip (SOC) and designed two C programs to run on the NIOS-II. The NIOS-II is a 32-bit CPU that can be programmed to run The first program we produced blinks one green LED, and its main purpose is to test if we properly set up NIOS-II. The second program we produced was an 8-bit accumulator that added a value set by the switches to a total sum, which was displayed on the green LEDs.

# Written Description and Diagram of NIOS-II System

## Summary of Operation

HARDWARE



*Figure $: main.c of LED blink program*

As seen above, In a big frame, the whole software components of the lab were included inside the infinite loop for the blinker code (line 11). The blinker code made the LED to go on and off, i.e. ‘blink’, continuously, after some delay (line 13~16). To be specific, line 14 sets the least significant bit (0x00 | 0x01 = 0x01), whereas line 16 clears the least significant bit (0x01 & 0xFE = 0x00). The whole code is inside the infinite loop, the return value will be 1 since the code is not supposed to return any values (line 18).

For the actual lab,

## 

## 

## Written Description of all .sv Modules

## Module: lab7 from *lab7.sv*

Inputs: CLOCK\_50, [3:0]KEY, [7:0]SW, [31:0]DRAM\_DQ

Outputs: [7:0]LEDG, [12:0]DRAM\_ADDR, [1:0]DRAM\_BA, DRAM\_CAS\_N, DRAM\_CS\_N, DRAM\_CKE, [31:0]DRAM\_DQ, DRAM\_RAS\_N, DRAM\_WE\_N, DRAM\_CLK

Description:

Purpose:

## Module: lab7\_soc from *lab7\_soc.v*

Inputs: clk\_clk, [2:0]keys\_wire\_export, reset\_reset\_n, [31:0]sdram\_wire\_dq, [7:0]switches\_wire\_export

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,

[31:0]sdram\_wire\_dq, [3:0]sdram\_wire\_dqm, sdram\_wire\_ras\_n, sdram\_wire\_we\_n

Description:

Purpose:

## Module: lab7\_soc\_keys from *lab7\_soc\_keys.v*

Inputs: [1:0]address, [1:0] reset

Outputs: debug\_reset\_request

Description :

Purpose:

## Module: lab7\_soc\_led from *lab7\_soc\_led.v*

Inputs: [1:0]address, [1:0]chipselect, [1:0]cls, [1:0]reset\_n, [1:0]write\_n, [31:0] writedata

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

Description:

Purpose:

## Module: lab\_soc\_nios2\_gen2\_0 from *lab\_soc\_nios2\_gen2\_0.v*

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

Outputs: [28:0]d\_address, [3:0]d\_byteenable, d\_read, d\_write, [31:0]d\_writedata, debug\_mem\_slave\_debugaccess\_to\_rams, [28:0]i\_address, i\_read, debug\_reset\_request, [31:0]debug\_mem\_slave\_readdata, debug\_mem\_slave\_writerequest, dummy\_ci\_port

Description: Runs the instructions from the C.

Purpose:

## Module: lab7\_soc\_onchip\_memory2\_0 from *lab7\_soc\_onchip\_memory2\_0.v*

Inputs: [1:0] add

Outputs:

Description:

Purpose:

## Module: lab7\_soc\_sdram from *lab7\_soc\_sdram.v*

Inputs: clk, reset

Outputs:

Description:

Purpose:

## Module: lab7\_soc\_sdram\_pll from *lab7\_soc\_sdram\_pll.v*

Inputs: indk\_interface, indk\_interface\_reset

Outputs:

Description:

Purpose:

## Module: lab7\_soc\_switches from *lab7\_soc\_switches.v*

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

Outputs: [31:0] readdata:

Description:

Purpose:

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

Inputs:

Outputs:

Description:

Purpose:

## Top Level Block Diagram

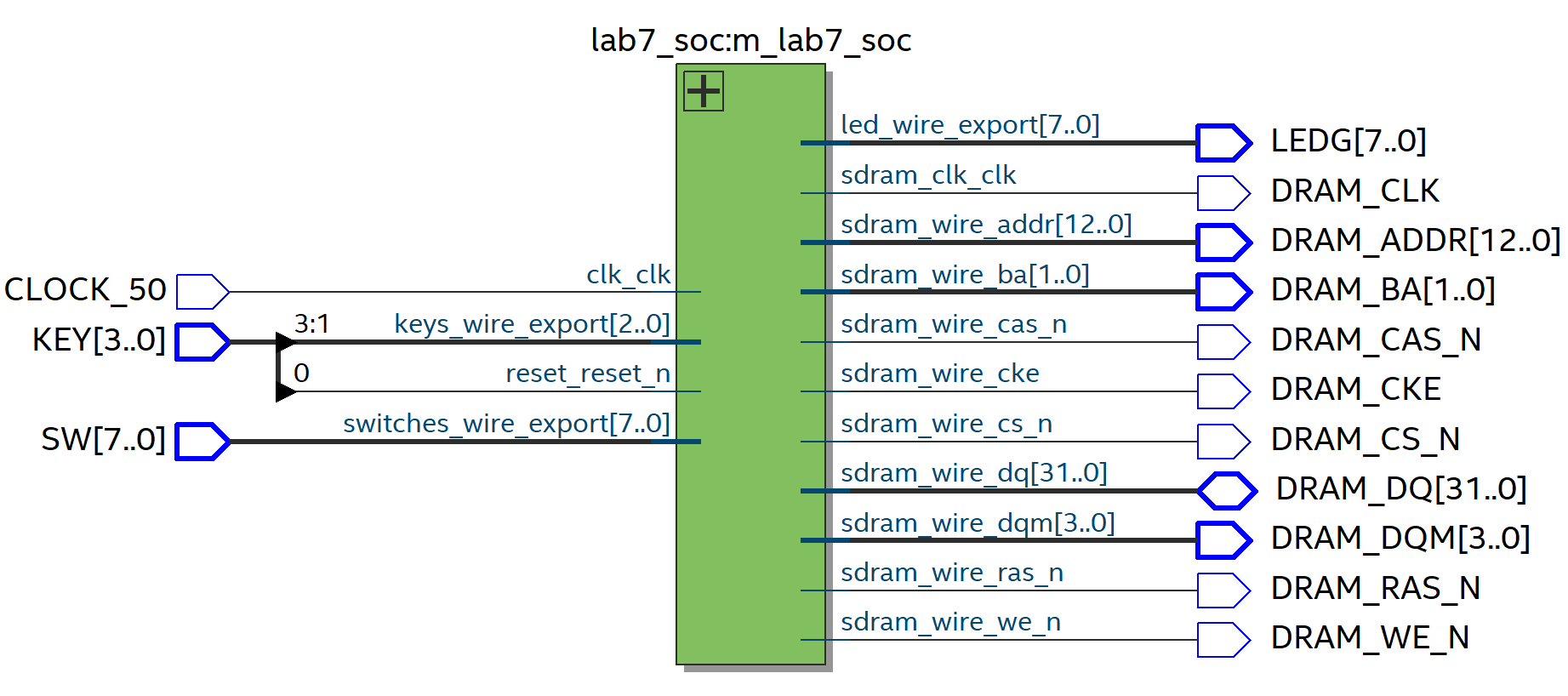


Figure $+1: Top-Level Block Diagram of the module

## 

## System Level Diagram

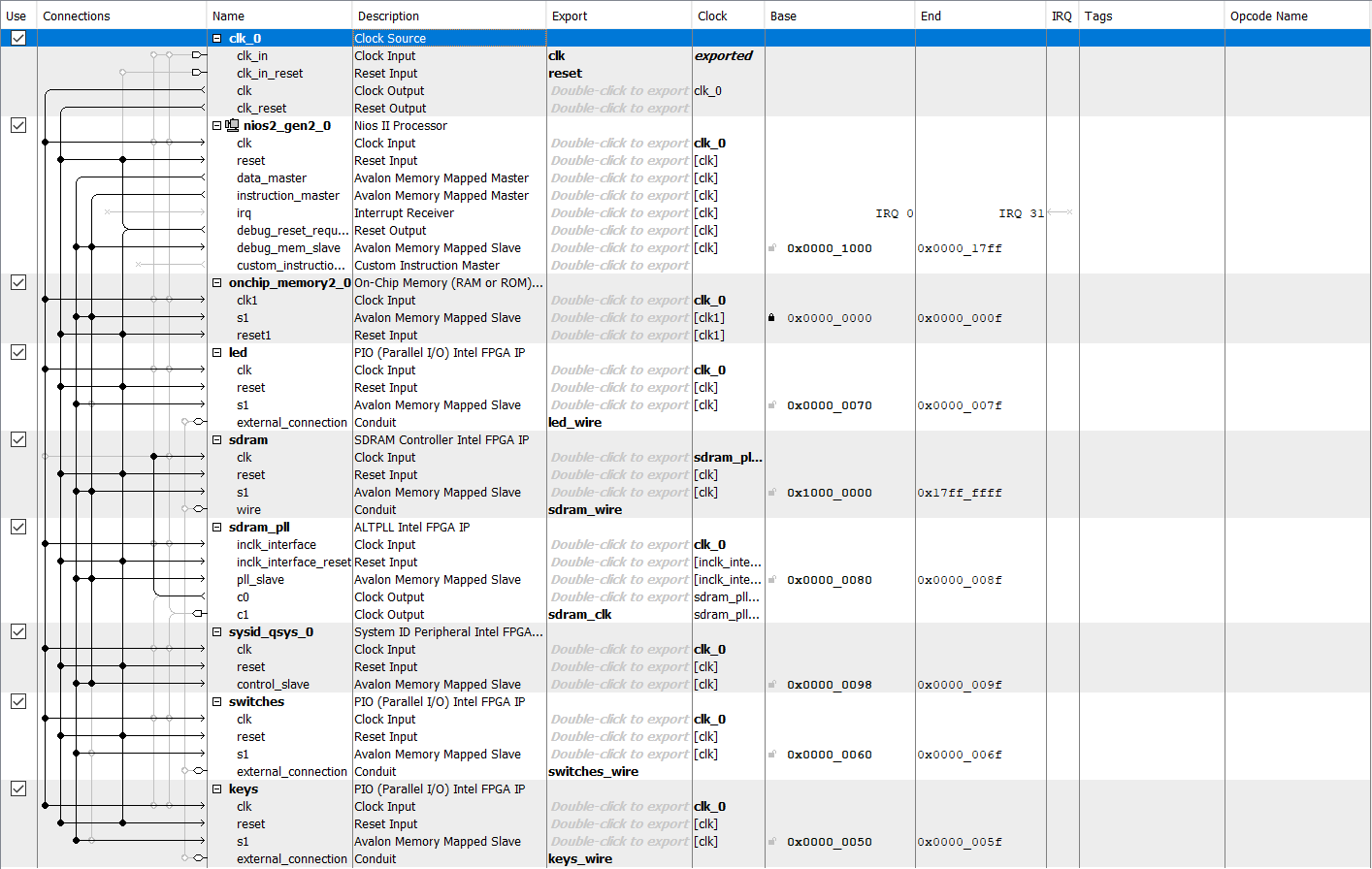


Figure $+2: QSys view of the SoC module

Component: clk\_0

Description:

Component: nios2\_gen2\_0

Description:

Block: onchip\_memory2\_0

Description:

Component: led

Description:

Component: sdram

Description:

Component: sdram\_pll

Description:

Component: sysid\_qsys\_0

Description:

Component: switches

Description:

Component: keys

Description:

## Answers to all INQ Question

INQ 1: What are the differences between the Nios II/e and Nios II/f CPUs?

e stands for Economy, and f stands for Fast. Economy is the free version that lacks a few features.

|  |  |  |
| --- | --- | --- |
|  | **Nios II/e** | **Nios II/f** |
| **Summary** | Resource-optimized 32-bit RISC | Performance-optimized 32-bit RISC |
| **Featues** | JTAG Debug Module  ECC RAM Protection | JTAG Debug Module  Hardware Multiply/Divide Option  Separate Instruction/Data Caches  Optional Tightly-Coupled Master  ECC RAM Protection  External Interrupt Controller  Shadow Register Sets  Optional MPU or MMU |
| **RAM Usage** | 2+ Options | 2+ Options |

INQ 2: What advantage might on-chip memory have for program execution?

It has the highest throughput and lowest latency. Having the latency of only one clock cycle, it makes program execution faster because it does not take that long to get instructions.

INQ 3: 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 machine. There are separate sections storing the data and the instructions, which makes it like a Havard machine. However, both the data and the instructions use the same memory bus to the SDRAM and on-chip memory, which transforms this into a modified Harvard machine.

INQ 4: 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 peripheral only needs access to the data bus because it is only used to display data. There is no need for it to access the program bus because it is not storing or executing any code, just displaying data when it needs to.

INQ 5: Why does SDRAM require constant refreshing?

Because it is dynamic, it needs constant refreshing to maintain its contents. If it did not refresh, then the stored memory could be lost as the charge in the memory cell leaks away. Additionally, reads for each row in dynamic memory can be destructive, so the rows refresh after each read. However, since this is not dependable because not all rows will be refreshed with a read, a constant refresh is needed.

INQ 6:

|  |  |  |
| --- | --- | --- |
| **SDRAM parameter** | **Short name** | **Parameter Value** |
| **Data Width** | [width] | 16 \* 2 = 32 in total |
| **# of Rows** | [nrows] | 8192 for each bank (13 address pins) |
| **# of Columns** | [ncols] | 1024 for each bank (10 address pins) |
| **# of Chip Selects** | [ncs] | 1 for each chip |
| **# of Banks** | [nbanks] | 4 for each chip |

INQ 7: 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.

There are 4 banks on one chip according to the datasheet. Each bank has 13 row pins and 10 column pins. This amounts to 8,192 (8K) possible rows and 1,024 (1K) possible columns in a bank. In total, there are 2^23, or 8,388,608 (8M), possible row-column combinations for each bank. Accounting for all 4 banks, this gives you 2^25, or 33,554,432 (32M), possible addresses. Because we are using 2 chips, each with a width of 16 bits, if both chips use the same addressing, each address can correspond to a total for 32 bits, with one chip holding the upper bits and the other chip holding the lower bits. With both chips, you get a grand total of 2^30, or 1,073,741,824 (1G), bits of data within memory. This corresponds to 2^27, or 134,217,728 (128M), bytes of data in memory. ((32 \* (2^13) \* (2^10) \* 4 = 1,073,741,824)).

INQ 8: What is the maximum theoretical transfer rate to the SDRAM according to the timings given?

Access time is 5.5ns, and since the units for theoretical transfer rate is in bytes/second

(1 / (5.5 \* 10 ^ -9 seconds)) \* (32 bits) \* ((1 byte) / (8 bits)) = .727272 \* 10 ^ 9 bytes/second = 727.272 \* 10 ^ 6 bytes/second = 727.272 Mbytes/second

INQ 9: The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?

If the SDRAM is ran too slowly, it will not refresh properly. In addition, our system is running at 50MHz, and since we want our SDRAM to be synced up with our system, the clock for SDRAM should also be running at 50MHz in order to ensure that our accesses are valid and not misaligned.

INQ 10: 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 the SDRAM controller.

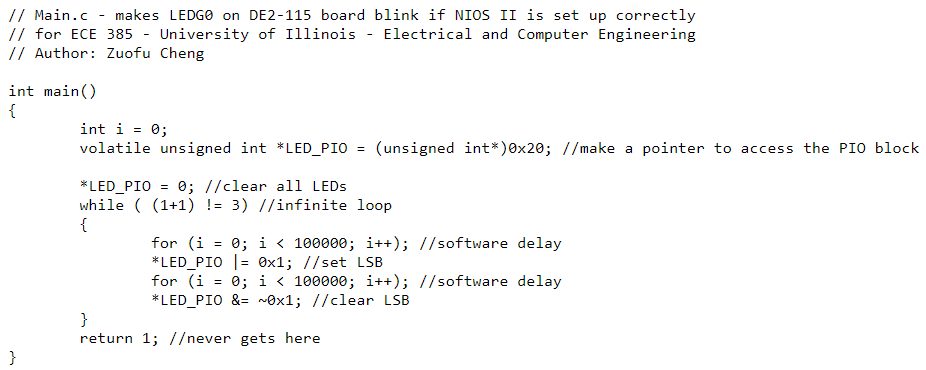
The address, data, and control signals are only valid in a certain time frame. As such, in order to access these signals from the SDRAM, we must toggle the SDRAM clock appropriately. When this delay is calculated, it turns out that the SDRAM clock needs a 3ns delay behind the controller clock in order for the address, data, and control signals to be valid and stable when we access them.

INQ 11: What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?

NIOS start execution starts from the reset vector, which in our code is 0x10000000. We assign the reset and exception vectors after assigning the addresses because these vector addresses will change after assigning the base addresses and we want the most updated addresses when assigning the vectors.

INQ 12: 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. *(int i …)* Make a variable for the loops for software delay
  2. *(volatile unsigned int …)* Make a pointer to the LEDs; volatile means that the system will always reload the variable because it can be changed elsewhere
  3. *(\*LED\_PIO = 0 …)* Initialize the LEDs
  4. *(while …)* Loop infinitely so that the code is always run on the board
  5. *(for …)* Software delay
  6. *(\*LED\_PIO |= …)* Set the least significant bit (0x00 | 0x01 = 0x01)
  7. *(for …)* Software delay
  8. *(\*LED\_PIO &= …)* Clear the least significant bit (0x01 & 0xFE = 0x00)
  9. *(return …)* Should never return because of infinite loop



INQ 13: 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

* 1. .bss (block started by symbol) is used for uninitialized statically allocated objects (file scope variables and static local variables)
     1. *static int counter;* allocates space in the .bss segment for an integer and sets it to zero
  2. .heap is used for a dynamically allocated memory
     1. *int\* ptr = malloc(sizeof(int));* allocates an integer in the .heap segment
  3. .rodata is used for constant data
     1. *const int bitmask = 0x0001;* will store the value of bitmask in the .rodata segment
  4. .rwdata is used for initialized static variables (global variables and static local variables)
     1. *static int num = 42;* will store the value of num in the .rwdata segment
  5. .stack is used for storing local variables and stack frames for functions
     1. *int i = 0;* at the start of a function will make room for an integer in the .stack segment and initialize it to 0
  6. .text is used for storing code that runs on the machine
     1. *a = b + c;* is stored in the .text segment

## 

## Documentation of the Design Resources and Statistics

|  |  |
| --- | --- |
| **LUT** | 2,160 |
| **DSP** | 0 |
| **Memory (BRAM)** | 36,864 |
| **Flip-Flop** | 1,952 |
| **Frequency (MHz)** | 91.85 |
| **Static Power (mW)** | 102.05 |
| **Dynamic Power (mW)** | 40.95 |
| **Total Power (mW)** | 204.39 |

# Conclusion

## The Functionality of the Design

## The main bug that we encountered in this experiment was that after compiling our C code, the ELF file was not properly generated. This occurred when we were trying to compile the code for the blinking LED, so we were unfamiliar with the Eclipse environment. We tried searching for information in the console to see if we had some type of compilation error, but the console said that it was built correctly. We also tried regenerating the BSP, and building again, but this still did not generate an ELF file for us. Finally, we tried retracing our steps from the start of the Introduction to NIOS II and Platform Designer tutorial to see if we for some reason missed a step. After making sure that we follow all of the steps in the tutorial properly, we took one last look at the C code we wrote and realized that we forgot to save our main.c file before building. After saving the file and attempting to compile once more, we found that we were finally able to generate an ELF file. On a side note, we found it interesting that the Eclipse interface asks if we want to save our changes when we try to run the program but does not do the same when we are building the code.

Ultimately, the design worked as expected. Accumulator successfully incremented by the value signified by the switches when pressed Accumulate, and cleared when Reset is pressed. Also, when the value exceeds 255, the accumulator has successfully overflowed as expected.

## Ambiguity and Incorrectness in the Lab Manual

The main issue we had with the given documentation is the figures in the Introduction to NIOS II and Platform Designer (formerly Qsys) documentation. This is due to the fact that the figures are inconsistent and do not always properly reflect the items discussed in the documentation. For example, page INQ.10 references Figure 17 in the documentation as an example of how to configure the Inputs/Lock page of the SDRAM PLL. However, upon further inspection, we find that Figure 17 demonstrates how the reset and exception vectors are supposed to be set, and the figure that the documentation was actually trying to reference did not actually exist. A similar issue also exists on page INQ.18. This page references Figure 26, which is supposed to display a pop-up window for creating a new project. However, Figure 26 is about how to run configurations on the board, and the real figure that page INQ.18 was referencing is nowhere to be found. Inside this documentation, Figure 16 is also wrong. It shows that the slave for the led peripheral is wired to both the data master and the instruction master. This is inconsistent with how it was displayed in Figure 9, which only shows the slave of the led peripheral connect to the data master. Additionally, Figure 16 does not support the question asked earlier in the documentation on why led peripheral only needs access to the data bus and not the program bus. Although this does not fundamentally change how the led peripheral will respond inside the system, this does lead to confusion among students that are trying to check over all of their connections.