

ECE 385

Fall 2019

Experiment #7

SOC with NIOS II in SystemVerilog

Xinglong Sun Churan He

ABD T 8AM

Mihir Iyer

Introduction

a. Basic functionality of the NIOS-II processor running on the Cyclone IV FPGA

Most systems need lots of low performance tasks such as getting data in and out of system, formatting data, debugging, and user interface. On Cyclone IV FPGA, we leave these tasks to NIOS-II, which is an IP based 32-bit CPU, to handle. In the meantime, an accelerator peripheral in our FPGA logic can handle those high performance tasks.

Written Description and Diagrams of NIOS-II System

a. Summary of Operation

i. Hardware component

In this lab, the main hardware component we include is the NIOS II CPU, which is a 32-bit RISC CPU IP module from Intel. The NIOS II CPU module we used in this lab contains build-in memory, LED, switch and key driver (PIO blocks), PLL clock generator and SDRAM controller. These hardwares make running the C program, reading from the switches, accumulating the value and printing the result to the LEDs possible.

ii. Software component (Blinker + Accumulator)

The software component of this lab contains a blinker LED test program and an accumulator program. The Blinker LED program is explained in detail in the following INQ Question section. The accumulator program will be elaborated as follows:

```
*main.c
1 // Main.c - makes LEDG0 on DE2-115 board blink if NIOS II is set up correctly
4
5 int main()
6 {
7     volatile unsigned int *LED_PIO = (unsigned int*)0x70; //make a pointer to access the PIO block
8     volatile unsigned int *SW_PIO = (unsigned int*)0x60;
9     volatile unsigned int *KEY_PIO = (unsigned int*)0x50;
10
11     *LED_PIO = 0; //clear all LEDs
12     unsigned int total = 0;
13     unsigned int current = 0;
14     while (1) //infinite loop
15     {
16         if(*KEY_PIO == 0x2)
17             total &= 0x0;
18         else if(*KEY_PIO == 0x1){
19             while(*KEY_PIO == 0x1);
20             current = *(SW_PIO);
21             total = (total + current)%256;
22         }
23         *LED_PIO = total;
24     }
25     return 1; //never gets here
26 }
27
```

This program will keep a record of the running total of inputs by user. As the user hits the reset button, the total will be reset back to zero. Otherwise, as the user hits the “Accumulate” button, current switch data will be added to the running total and displayed on the LEDs. One thing to notice is that we use modulo 256 to handle the overflow cases. As the total goes to 255, the next accumulating 1 would cause the total going back to 0.

b. Written Description of all .sv Modules (include lab7soc.v)

Lab7.sv

Inputs:

CLOCK_50,

[3:0] KEY,

[7:0] SW

Outputs:

[7:0] LEDG,

[12:0] DRAM_ADDR,

[3:0] DRAM_DQM,

[1:0] DRAM_BA,

DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, DRAM_RAS_N,
DRAM_WE_N, DRAM_CLK

InOut:

[31:0] DRAM_DQ

Purpose & Description:

This is the top level module used to interface with NIOS II. The module takes the NIOS II inputs and outputs configured in QsYs as the overall inputs and outputs of the system. Specifically, CLOCK_50 is the 50MHz we use for the overall circuit. [3:0] Key are the three press buttons on the FPGA board. [7:0] SW takes in a 8-bit long input data. [7:0] LEDG are the values displayed representing the running total of our accumulator or

for debugging usage. All the DRAM related ports are connected to the DRAM we allocate in NIOS II.

lab7soc.v

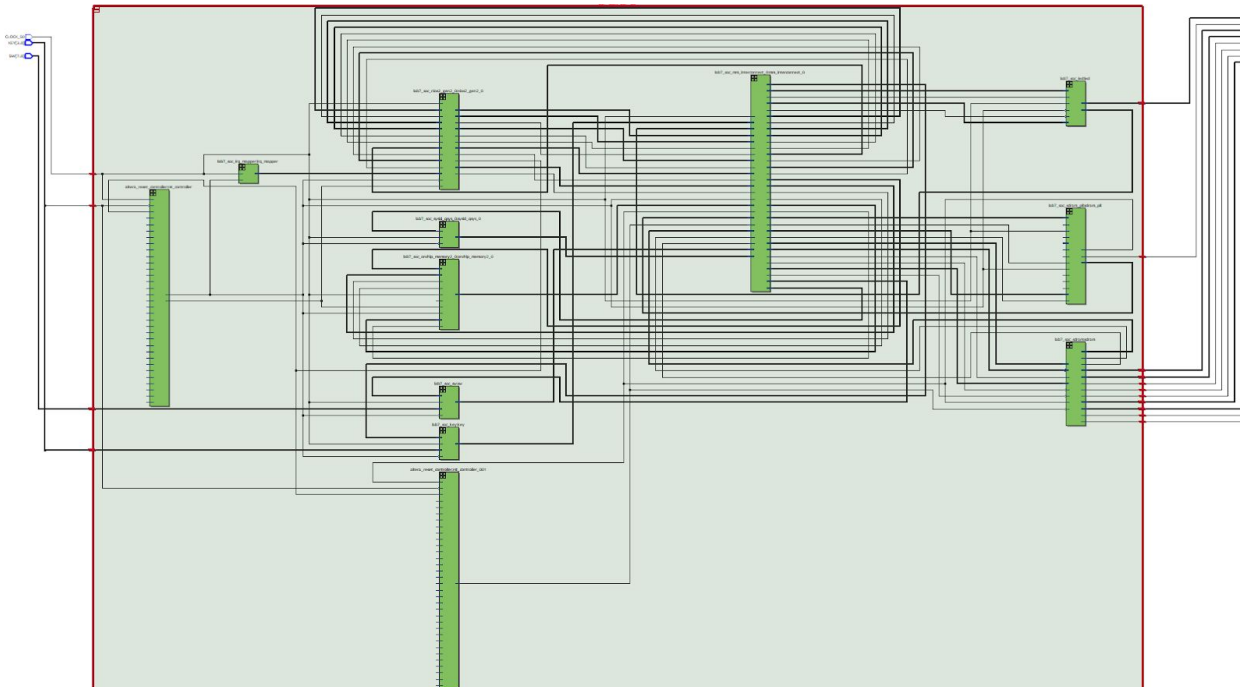
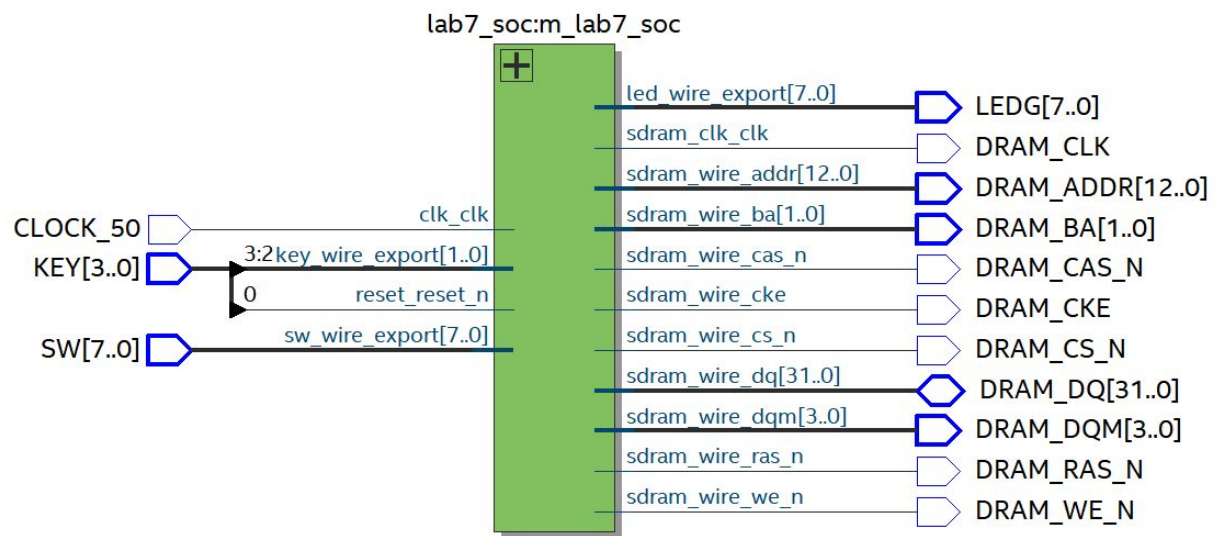
Contents:

clk_0, nios2_gen2_0, onchip_memory2_0, led, sdram, sdram_pll, sysid_qsys_0,
sw, key

Purpose & Description

This is the actual SOC we use in this lab, which runs our C program. The IP for CPU is NIOS II/e, which is the efficient version in NIOS II 32-bit RISC CPU family. It has an on-chip memory module working as a data buffer/cache, led controller (PIO) for LED display, sdram controller to read/write data from off-chip SDRAM, a sdram_pll clock generator for the shifted sdram clock, sysid module for software verification, and finally sw and key drivers handling inputs. These modules would be discussed in detail afterwards. In FPGA design, the CPU IP can significantly reduce the workload for a project, and it can handle slower tasks, and leave more specific and performance required tasks to logics on FPGA.

c. Top Level Block Diagram



d. System Level Block Diagram

Use	Connections	Name	Description	Export	Clock
<input checked="" type="checkbox"/>		<div>clk_0</div> <div>clk_in</div> <div>clk_in_reset</div> <div>clk</div> <div>clk_reset</div>	<div>Clock Source</div> <div>Clock Input</div> <div>Reset Input</div> <div>Clock Output</div> <div>Reset Output</div>	<div>clk</div> <div>reset</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>exported</div> <div>clk_0</div>
<input checked="" type="checkbox"/>		<div>nios2_gen2_0</div> <div>clk</div> <div>reset</div> <div>data_master</div> <div>instruction_master</div> <div>irq</div> <div>debug_reset_request</div> <div>debug_mem_slave</div> <div>custom_instruction_m...</div>	<div>Nios II Processor</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Master</div> <div>Avalon Memory Mapped Master</div> <div>Interrupt Receiver</div> <div>Reset Output</div> <div>Avalon Memory Mapped Slave</div> <div>Custom Instruction Master</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>clk_0</div> <div>[clk]</div> <div>[clk]</div> <div>[clk]</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>onchip_memory2_0</div> <div>clk1</div> <div>s1</div> <div>reset1</div>	<div>On-Chip Memory (RAM or ROM) Intel ...</div> <div>Clock Input</div> <div>Avalon Memory Mapped Slave</div> <div>Reset Input</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>clk_0</div> <div>[clk1]</div> <div>[clk1]</div>
<input checked="" type="checkbox"/>		<div>led</div> <div>clk</div> <div>reset</div> <div>s1</div> <div>external_connection</div>	<div>PIO (Parallel I/O) Intel FPGA IP</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div> <div>Conduit</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>led_wire</div>	<div>clk_0</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>sdram</div> <div>clk</div> <div>reset</div> <div>s1</div> <div>wire</div>	<div>SDRAM Controller Intel FPGA IP</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div> <div>Conduit</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>sdram_wire</div>	<div>sdram_pll_...</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>sdram_pll</div> <div>indk_interface</div> <div>indk_interface_reset</div> <div>pll_slave</div> <div>c0</div> <div>c1</div>	<div>ALTPLL Intel FPGA IP</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div> <div>Clock Output</div> <div>Clock Output</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>sdram_clk</div>	<div>clk_0</div> <div>[indk_interf...</div> <div>[indk_interf...</div> <div>sdram_pll_c0</div> <div>sdram_pll_c1</div>
<input checked="" type="checkbox"/>		<div>sysid_qsys_0</div> <div>clk</div> <div>reset</div> <div>control_slave</div>	<div>System ID Peripheral Intel FPGA IP</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div>	<div>clk_0</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>sw</div> <div>clk</div> <div>reset</div> <div>s1</div> <div>external_connection</div>	<div>PIO (Parallel I/O) Intel FPGA IP</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div> <div>Conduit</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>sw_wire</div>	<div>clk_0</div> <div>[clk]</div> <div>[clk]</div>
<input checked="" type="checkbox"/>		<div>key</div> <div>clk</div> <div>reset</div> <div>s1</div> <div>external_connection</div>	<div>PIO (Parallel I/O) Intel FPGA IP</div> <div>Clock Input</div> <div>Reset Input</div> <div>Avalon Memory Mapped Slave</div> <div>Conduit</div>	<div>Double-click to export</div> <div>Double-click to export</div> <div>Double-click to export</div> <div>key_wire</div>	<div>clk_0</div> <div>[clk]</div> <div>[clk]</div>

i. SoC Module Description

Module: nios2_gen2_0

Exports:

Description:

This is an economy version processor we allocate for NIOS II. It handles all the logical operations and computations our program might use. One more thing to notice is that this processor also supports JTAG Debug, which means we can debug using “printf” in Eclipse.

Module: onchip_memory2_0

Exports:

Description:

This is the onchip_memory module for our system, which is always used as cache or data buffer. On-chip memory is relatively “expensive” since it required a lot more logic elements, but it can significantly improve the memory operation speed compared on external memory chips.

Module: led

Exports: led_wire

Description:

This is a PIO(Parallel I/O) block used to handle our led output. By exporting the led_wire to the physical led pins on the FPGA board, we can control the display of the LEDs by directly changing the memory-mapped value in our NIOS II software program.

Module: sdram

Exports: sdram_wire

Description:

This is the SDRAM controller for NIOS II CPU. SDRAM need to be refreshed every time as we do memory operations, we don't need to write a driver by ourselves, because this existing SDRAM IP module can be used in this case.

Module: sdram_pll

Exports: sdram_clk

Description:

Pll stands for “Phase Lock Loop.” As suggested by the name, it’s used to generate a second clock signal shifted by an amount. As explained in detail in the following subsection(INQ Question), this block is necessary to prevent skew operation.

Module: sysid_qsys_0

Exports:

Description:

This system ID checker is to ensure the compatibility between hardware and software. This module will give us a serial number, which the software loader checks against when we start the program. This prevents us from loading software onto an FPGA which has an incompatible NIOS II configuration.

Module: sw

Exports: sw_wire

Description:

This is a PIO block used to handle our switch inputs. By connecting the export “sw_wire” to the physical switches on board, we can directly receive the user input switch data in NIOS II software program.

Module: key

Exports: key_wire

Description:

This is a PIO block used to handle our key inputs. By connecting the export “key_wire” to the physical keys on board, we can directly receive the user input key data in NIOS II software program.

e. Answers to all INQ Questions(include Prelab question (part A))

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

NIOS II provides us the option of two kinds of embedded processor. Particularly, Nios II/e CPU is an economy version of the processor and is the one we use in this lab. It is resource-optimized and saves more power compared to the Nios II/f CPU, which is performance optimized. Generally, Nios II/f CPU also supports more features than its counterpart.

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

Operations with on-chip memory would be significantly faster than off-chip memories like the SRAM chip or the SDRAM chip. Writing or reading to on-chip memory don't need to transfer data through MUXes and the tristate buffer. For small and simple program (which wouldn't use a lot of memory), on-chip memory can boost the overall system performance.

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. In a Von-Neumann-Architecture Computer, data and instruction share the same memory address and datapath. Therefore, they couldn't be accessed at the same time. For pure Harvard Machine, there are separate storage and signal pathways for data and instructions. Modified Harvard Machine, as a variation of Pure Harvard Machine, allows the content of instruction to be accessed as data. By observation of the connections, we can see that Data and Instruction have two separate paths, indicating its Harvard architecture. Furthermore, we notice that both Data and Instruction go into the "s1" port of our on-chip memory, indicating that it's a Modified Harvard Machine.

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 NIOS II System is a modified Harvard machine, so it has a data bus and a program bus. The on-chip memory (as a cache) needs to store the program as well as the data during the software operation, therefore it connects to both data and program bus. The LED is a PIO, which only takes the data from data bus to display, therefore it only needs to connect to data bus.

Why does SDRAM require constant refreshing?

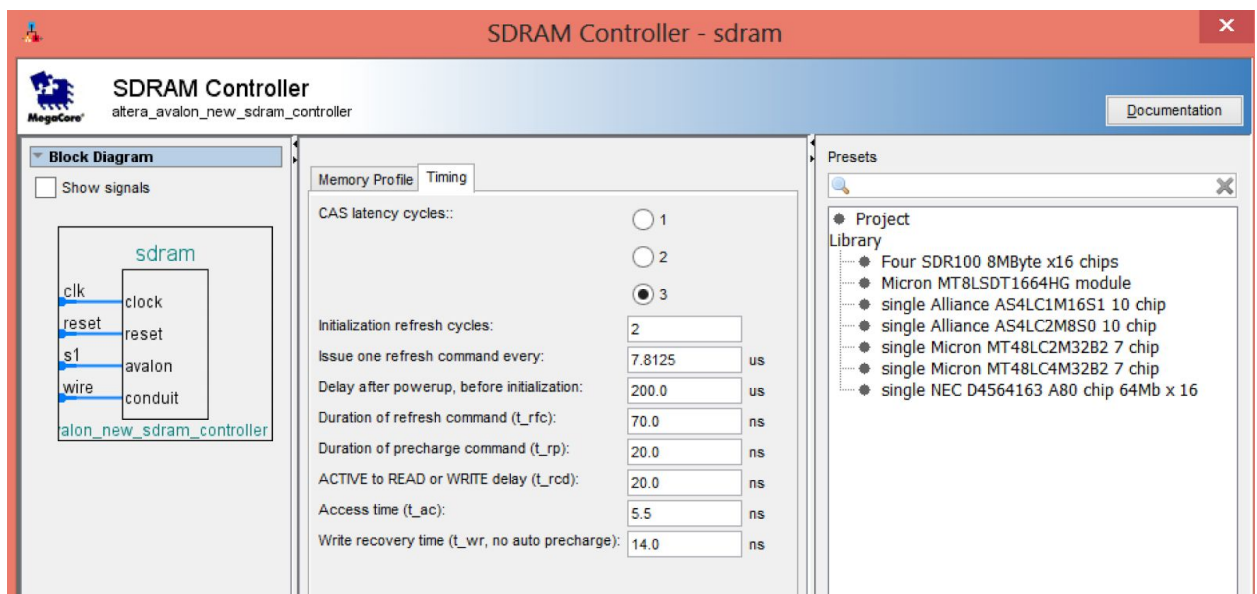
Referring to the name, D means “Dynamic”. DRAM uses a single driving transistor and capacitor to contain a bit, which is represented as the charge on the capacitor. However, the capacitor can discharge, which will eventually lead to a loss of the data if we do not refresh it. Therefore, we need to periodically make the driving transistor charge the capacitor to the same level in order to secure the data.

*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 two 32M * 16 chips (16320D), so the total amount of memory is:

$$2 \text{ (chips)} * 8\text{M (words)} * 4 \text{ (banks)} * 16 \text{ bits} = 1 \text{ Gbit} = 128 \text{ Mbytes}$$

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



We notice that the access time is 5.5ns. Therefore, the maximum transfer rate in bits per second should be:

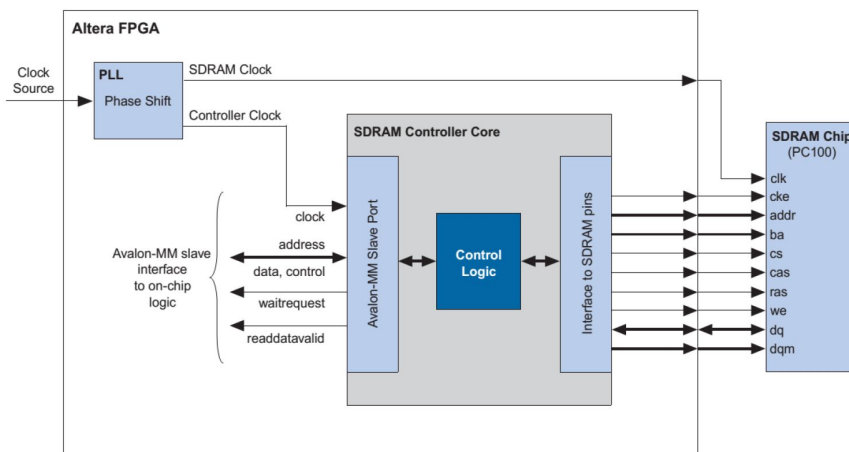
$$\begin{aligned} & \# \text{ Access times per second} * \text{Bits data width} \\ &= (10^9 / 5.5) * 32 \\ &= 5.82 * 10^9 \\ &= 5820 \text{ Mbits} \end{aligned}$$

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

50 MHz is the frequency of the overall system clock, and our NIOS II cpu runs at that speed. If the SDRAM runs slower than 50 MHz, it's not able to provide new and refreshed data to the data bus for NIOS cpu to use and compute.

Why do we need to generate a second clock by PLL?

We generate a shifted second clock because our SDRAM requires precise timings, and the PLL allows us to compensate for the clock skew due to the board layout. Specifically, given the diagram as follows:



We generate a clock going out to the SDRAM Chip 3ns behind of the controller clock. In this way, we can secure the data transfer because at the rising edge of the SDRAM clock, the control signals coming from SDRAM controller have already been stable.

What address does the NIOS II start execution from? Why do we need to assign the reset and exception vectors after assigning the addresses?

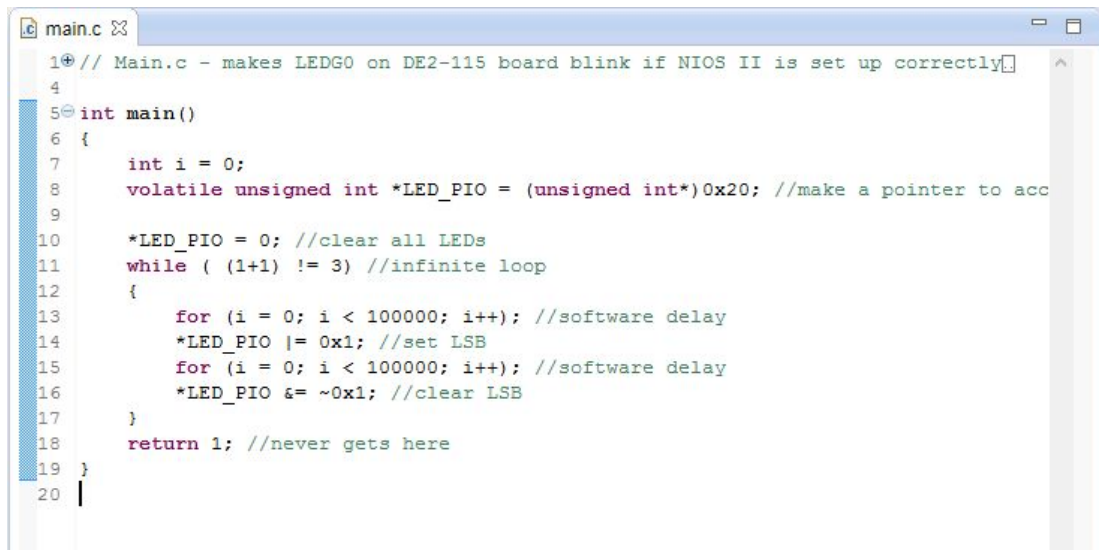
The NIOS II start execution from `0x10000000`, which is the base address of sdram. We need to assign reset and exception vectors after assigning the base addresses because the vectors need to know where are their destinations. Without assigning base addresses, the reset and exception vectors wouldn't be able to lead the system to execute correctly.

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).

The volatile qualifier means that the variable could change at any time unexpectedly. Basically, three types of variables that could change are :

- 1.Memory-mapped peripheral registers
- 2.Global variables modified by an interrupt service routine
- 3.Global variables accessed by multiple tasks within a multi-threaded application

In our case, we use the volatile keyword to manipulate peripheral registers.



```
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
```

Test Program we use

The address of the LED_PIO on our avalon bus is 0x20. Therefore, after assigning a pointer LED_PIO of this address, dereferencing LED_PIO would give us directly the value of LED value right now. We first dereference to access the LED value and then set it to “0” to achieve the “clear”. Later, the program enters an infinite loop which is basically like a pooling I/O, constantly changing the LED value. The software delay in between allows human eyes to physically capture the blinking.

Look at the various segment (.bss, .heap, .rodata, .rdata, .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 val0 = 0;
- .heap: double *val1 = malloc(sizeof(double));
- .rodata: const int val2 = 0;
- .rdata: int val3 = 0; //outside function
- .stack: int retVal(int valA) {

```

        valB = valA + 1;
        return valB;
    }
    .text: char text[] = "Text";

```

f. Design Resources and Statistics

LUT	2266
DSP	0
Memory (BRAM)	36864 bits
Flip-Flop	1969
Frequency	75.77 MHz
Static Power	102.03 mW
Dynamic Power	40.16 mW
Total Power	195.60 mW

Conclusion

a. Functionality of the design

Our design can work correctly. It can accumulate the value entered from the switch and show the value on the LED. If the value hit 255, the value could be restored to 0 and start to accumulate again ($255 + 1 = 0$). However, when we transfer the project to another computer, the Eclipse encounters trouble with the workspace directory and misses some files. So we need to recreate the Eclipse workspace, and paste our code to that new workspace. In conclusion, this lab is quite simple, but it really helps us know how to use IP modules and multiple peripherals in Quartus.