ECE 385

2021 FA

LAB 7
SOC with NIOS II in SystemVerilog

Zhou Qishen, Xie Mu

Introduction

In Lab7, we create a basic SoC based on the NIOS II system on the Altera Cyclone IV. The simple system allow the FPGA compile and execute C code with parallel input and output system. With the basic SoC, we blink the LEDs on the board and develop an accumulator in C code, that can keep adding numbers until it overflows.

Description & Diagram

Hardware Description

The hardware component of this lab is an NIOS II processor, and a SDRAM. In the lab, we connect the processor with the controller of the SDRAM. The controller can help decide the dimension of the SRAM on the chip, and the phase shifter provide the SDRAM with correct clock that the IO can be synchronous with the processor. For the PIO part, we create parallel IO system for the LEDs and the Switches to display and modify the C program to input data, add and display them with faster speed.

Software Description

The LED Blinker First, we define the actual address of the LEDPIO to connect the software with the hardware. Second, we clear the LED by writing 0s to the register. Then we create infinite loops to keep the LED blinking. In the loop, we will let LEDPIO |= 1 loop for several to make the LED memory to and b'1 to high the LSB for some time, then we will keep the LEDPIO &= 0 to make the LED memory and b'0 to lower the LSB.

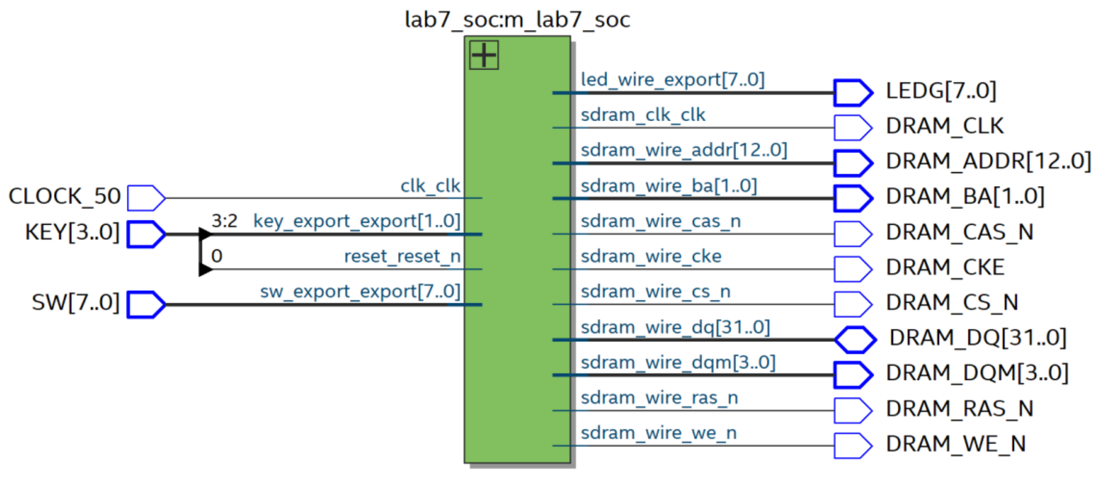
The Accumulator First, we define the actual address to the pointers that the program can access and directly write value to the actual hardware. Then we clear the LED and the values that record the status of the bottom and the value of the accumulator. Then we create an infinite loop to keep the program running. In the loop, we check if the Reset bottom is pressed. If so, the program will reset the LED and the value of the accumulator, and change the status of the reset bottom inside the program until it is released. Then we check if the accumulate bottom is pressed. If so, we will add the value in the switches with the stored value. If overflow happened, it will loop back to smaller value. Then the status of the accumulated bottom will return to 0 when the bottom is released. Finally at the end of the loop we will display the current value of the accumulator on the LED.

Module Description

Module: lab7
Inputs: Clock_50, [3:0] KEY, [7:0] SW
Output: [7:0] LEDG, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [31:0] DRAM_DQ, [3:0] DRAM_DQM, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK.
Description: The top-level file of LAB7.
Purpose: The top-level file declares, contents, and connects all the department together. For SoC, the top-level connects it with the hardware like LEDs, Switches and SDRAM, and connect the DRAM with the board memory.

Module: lab7_soc
Input: clk_clk, acc_reset_wire_export, acc_wire_export, reset_reset_n, [7:0]sw_wire_export.
Output: [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 SoC module that generated by the Platform Designer.
Purpose: The SoC module contents all the hardware information and the connection. It is generated from the PD based on what we done with the UI system.

System Level Block Diagram



Block Diagram

Hardware Module Description

Module: sdram
Input: clk, reset, s1
Output: wire
Description: This module is a 32-bit dynamic memory of SOC, which contains 1 chip, 4 banks, 13 rows and 10 columns, with the size of 1GB data.
Purpose: It is used for store data and address.

Module: onchip_memory2_0
Input: clk, reset, s1, wire
Output: readdata
Description: This module is writable RAM with memory size of 16 byte and datawidth of 32.
Purpose: This module is the on chip memory of NOIS II processor.

Module: led
Input: clk, reset, s1
Output: external_connection
Description: The module of LED on DE2 board with 8-bit datawidth.
Purpose: This module is used for control the LED on board to output data.

Module: sdram_pll
Input: inclk_interface, inclk_interface_reset, pll_slave, c0
Output: c1
Description: Clock module with the offset of -3 ns.
Purpose: This module is used for adjusting the clock input and I/O of sdram.

Module: sysid_qsys_0
Input: clk, reset, control_slave
Output: almost_empty, almost_full, empty, full, rd_data
Description: This module is system ID checker to ensure the compatibility between hardware and software.
Purpose: It can prevents us from loading software onto an FPGA which
has an incompatible NIOS II configuration.

Module: nios2_gen2_0
Input: clk, reset, data_master, instruction_master, irq, debug_deset_request, debug_mem_slave
Output: custrom_instruction_master
Description: The build-in NIOS II/e processor.
Purpose: The main processor of the SOC system.

Module: sw
Input: clk, reset, s1
Output: external_connection

Description: It is an 8 bit I/O module.
Purpose: This module is used to import 8-bit data from switch on board.

Module: key
Input: clk, reset, s1
Output: external_connection
Description: It is a 2 bit I/O module.
Purpose: This module is used to import 2-bit data from button on board.

Post-Lab

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

Due to shorter transmission distance, on-chip memory enable to reduce the I/O delay time, leading to the acceleration of the system.

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

Modified Havard, since it allows the contents of the instruction memory to be accessed as data while the addresses are changeable.

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, which is the component does not have the capability to process, while memory needs access to both the data and program bus, for it needs to acquire the source and target location of data.

Why does SDRAM require constant refreshing?

DRAM is a kind of RAM made of capacitors, which means it will constantly upcharge. That is to say, if it isn't refreshed, then the data might decay in the ram modules

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 (BITS)	[width]	32

# OF ROWS	[nrows]	13
# OF COLUMN	[ncols]	10
# OF CHIP SELECTS	[ncs]	2
# OF BANKS	[nbanks]	2

Calculation Process:

$$[\text{bits per address}] \times [\text{number of rows per bank}] \times [\text{number of columns per bank}] \\ \times [\text{number of banks per chip}] \times [\text{number of chips}]$$

$$32 \times 2^{13} \times 2^{10} \times 2^{2} \times 2 \text{ bits} = 2^{30} \text{ bits} = 1 \text{ Gbits}$$

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.

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

$$\text{Access Time} = \frac{1}{5.5 \text{ s}} \times 32 \text{ M} = 720 \text{ MB/s}$$

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

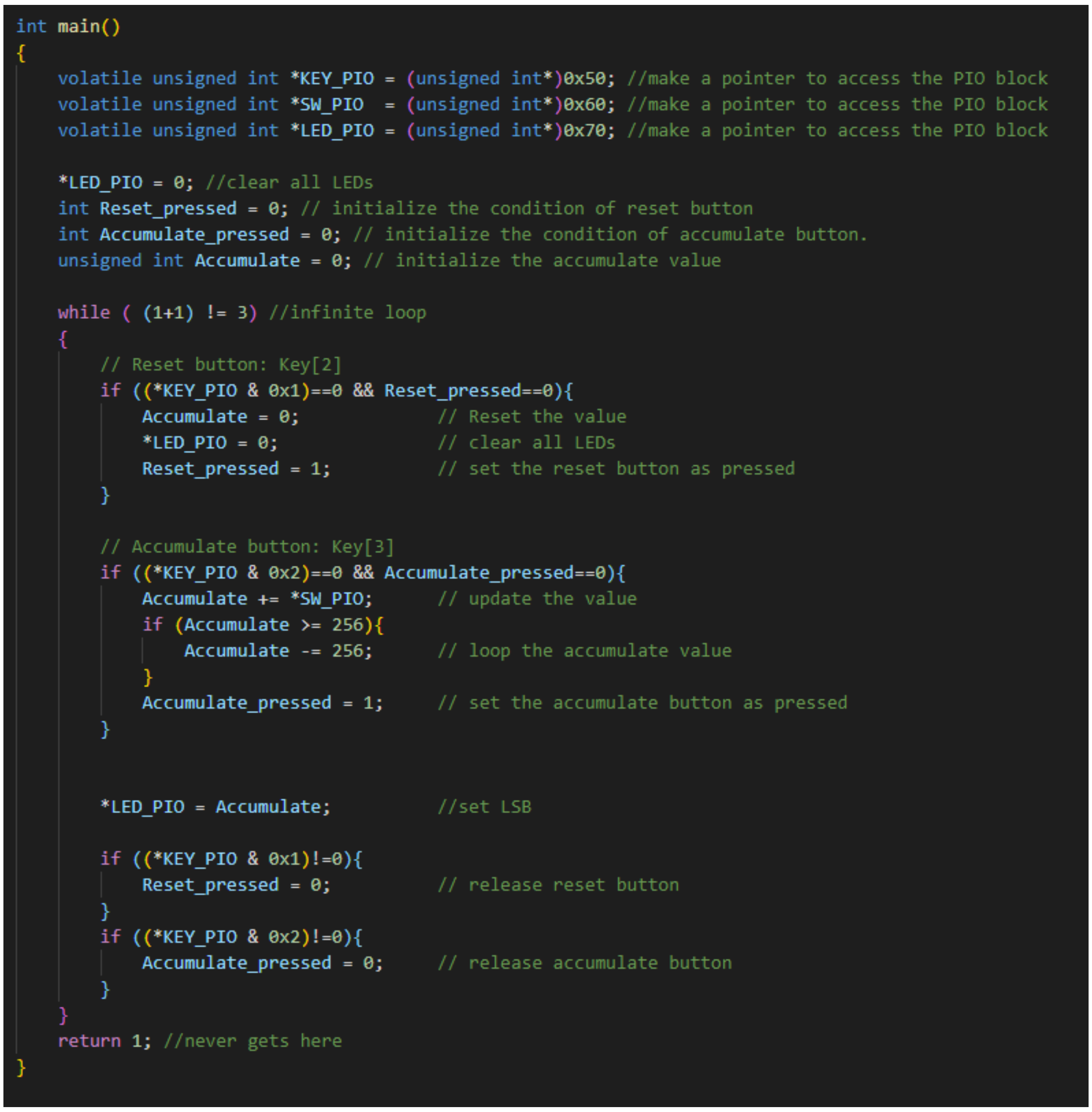
Latency issues occurs when it comes to I/O such as the SDRAM CAS latency, so we need to shift the clock ahead to eliminate all the latency to read in the correct value.

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

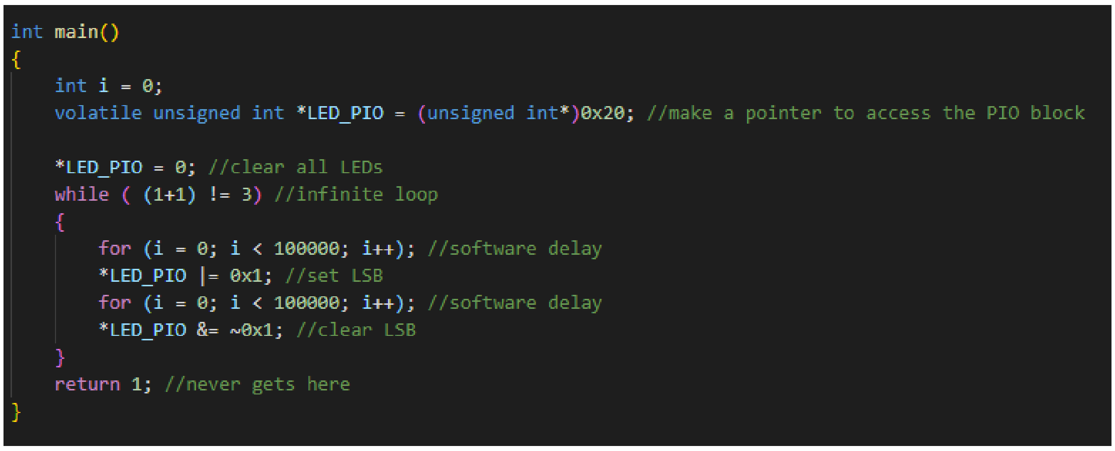
Execution starts from memory location 0x10000000. This step is done after address assigning since the processor need to specify where to start the program as well where to restore when the program is reset.

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.



```c
int main()
{
    volatile unsigned int *KEY_PIO = (unsigned int*)0x50; //make a pointer to access the PIO block
    volatile unsigned int *SW_PIO  = (unsigned int*)0x60; //make a pointer to access the PIO block
    volatile unsigned int *LED_PIO = (unsigned int*)0x70; //make a pointer to access the PIO block

    *LED_PIO = 0; //clear all LEDs
    int Reset_pressed = 0; // initialize the condition of reset button
    int Accumulate_pressed = 0; // initialize the condition of accumulate button.
    unsigned int Accumulate = 0; // initialize the accumulate value

    while ( (1+1) != 3) //infinite loop
    {
        // Reset button: Key[2]
        if ((*KEY_PIO & 0x1)==0 && Reset_pressed==0){
            Accumulate = 0;             // Reset the value
            *LED_PIO = 0;               // clear all LEDs
            Reset_pressed = 1;          // set the reset button as pressed
        }

        // Accumulate button: Key[3]
        if ((*KEY_PIO & 0x2)==0 && Accumulate_pressed==0){
            Accumulate += *SW_PIO;      // update the value
            if (Accumulate >= 256){
                Accumulate -= 256;      // loop the accumulate value
            }
            Accumulate_pressed = 1;     // set the accumulate button as pressed
        }


        *LED_PIO = Accumulate;          //set LSB

        if ((*KEY_PIO & 0x1)!=0){
            Reset_pressed = 0;          // release reset button
        }
        if ((*KEY_PIO & 0x2)!=0){
            Accumulate_pressed = 0;     // release accumulate button
        }
    }
    return 1; //never gets here
}
```



```c
int main()
{
    int i = 0;
    volatile unsigned int *LED_PIO = (unsigned int*)0x20; //make a pointer to access the PIO block

    *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
}
```

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.

SEGMENT	EXAMPLE	MEANING
.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();	function
.TEXT	char[] str = “myString”	string

Post-Lab

Property	Value
LUT	3156
DSP	0
Memory	10368
Flip-Flop	1903
Frequency	50 MHz
Static Power	40.34 mW
Dynamic Power	102.03 mW
Total Power	195.79 mW

Conclusion

1. Our design transforms the FPGA board into a simple MCU that it can compile and execute simple C software. During the lab, we meet Netlist error when we try to compile the project. This was cause by several undefined interfaces during the design. Next time we could check them twice before exit the design.

2. The lab manual is pretty good since it proves detailed steps for new learners who touch the SoC at the first time. However, I think you can bookmark the lab manual to make it easier to look up when we meet some problems.