ECE 385 Lab 7 Report

Yangkai Du, 3170112258

Chengting Yu, 3170111707

❑ Introduction

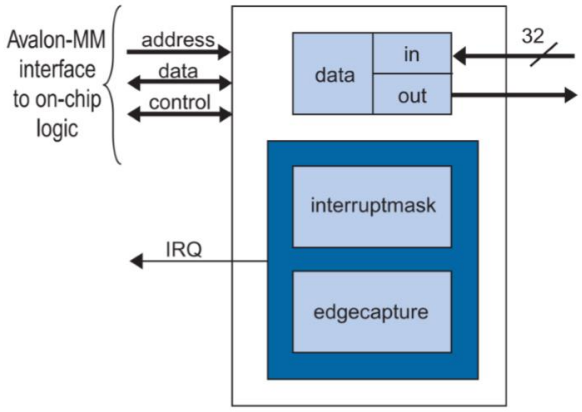
❑ Summarize the basic functionality of the NIOS-II processor running on the Cyclone IV FPGA.

In this lab, we created a NIOSII/e, SDRAM and PIO based SoC(System on Chip) which performs addition from switches into LEDs. The Behavior of design is similar to lab4, but using software(written in C) instead of hardware. Pressing 'Reset' (KEY[2]) at any time clears the accumulator to 0 and updates the display accordingly (turns all the LEDs off). Pressing 'Accumulate' (KEY[3]) loads the number represented by the switches into the CPU, adding it to the accumulator. Push buttons only react once to a single actuation.

❑ Written Description and Diagrams of NIOS-II System

❑ Summary of Operation

❑ Describe in words the hardware component of the lab (Describe what IP blocks are used beyond the ones necessary for the NIOS to run. In this lab, the only additional IP blocks used are the PIO blocks).



The PIO (Parallel I/O) Module is used to control registers memory mapped to addresses assigned by QSYS. In this experiment, we used following three IP blocks in our design:

- led [OUTPUT]: output the value of accumulate to LED pin to display
- switch [INPUT]: Input the values waited to be accumulated
- push [INPUT]: Input of 2-bit vector corresponding to KEY[3:2], with (push&0x10 -- KEY[3]) representing to the "execute" of accumulator, and (push&0x01 -- KEY[2]) representing to the "reset" of accumulator.

❑ Describe in words the software component of the lab. One of the INQ questions asks about the blinker code, but you must also describe your accumulator.

- NOIS II Processor: 32-bit modified Harvard RISC architecture processor, used with Avalon bus. Code written in C can be complied to Instruction that can be executed on NOIS II processor, serve as the CPU for software side.
- On-Chip Memory: On-Chip cache for CPU execution.
- SDRAM: Synchronous Dynamic Random-Access Memory, serve as the outside memory for CPU execution.

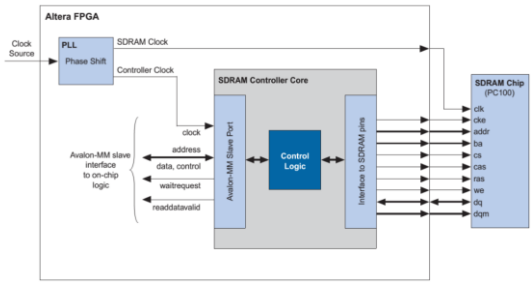
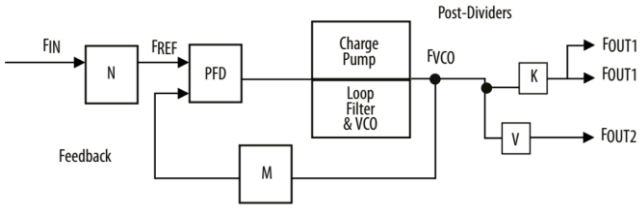


Figure 11

- Accumulator: Program written in C. It implements a accumulator which forms to the behavior written bellow:
  - pressing ‘Reset’ (KEY[2]) at any time clears the accumulator to 0 and updates the display accordingly (turns all the LEDs off). Pressing ‘Accumulate’ (KEY[3]) loads the number represented by the switches into the CPU, adding it to the accumulator. If the accumulator is overflow(above 255), clear to zero and then accumulate the remainder.
- SDRAM_PLL: To execute phase shift for external clock, and give SDRAM a separate clock.



❏ Written Description of all .sv Modules

❏ A guide on how to do this was shown in the Lab 5 report outline. Do not forget to describe the Qsys generated file lab7_soc.v!

Module: lab7.sv

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, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK

Inout: [31:0] DRAM_DQM

Description: This module implements the top-level entity for this lab, which connect all the port that is necessary for NIOS II system from outside world, and instantiate a SoC module built by qsys.

Purpose: As the top level entity, this module simply instantiates the SOC and thus all the other modules generated by .qsys file and connects the NIOS II system with the FPGA hardware.

Module: lab7_soc.v

Inputs: clk_clk, [3:0] push_wire_export, reset_reset_n, input [7:0] switch_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, [3:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n.

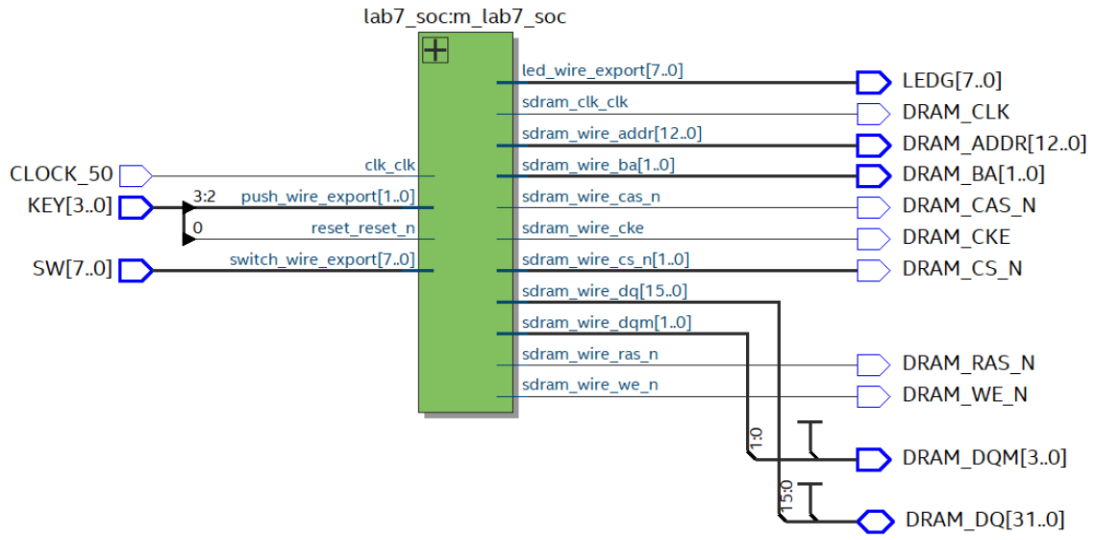
Inout: [15:0] sdram_wire_dq

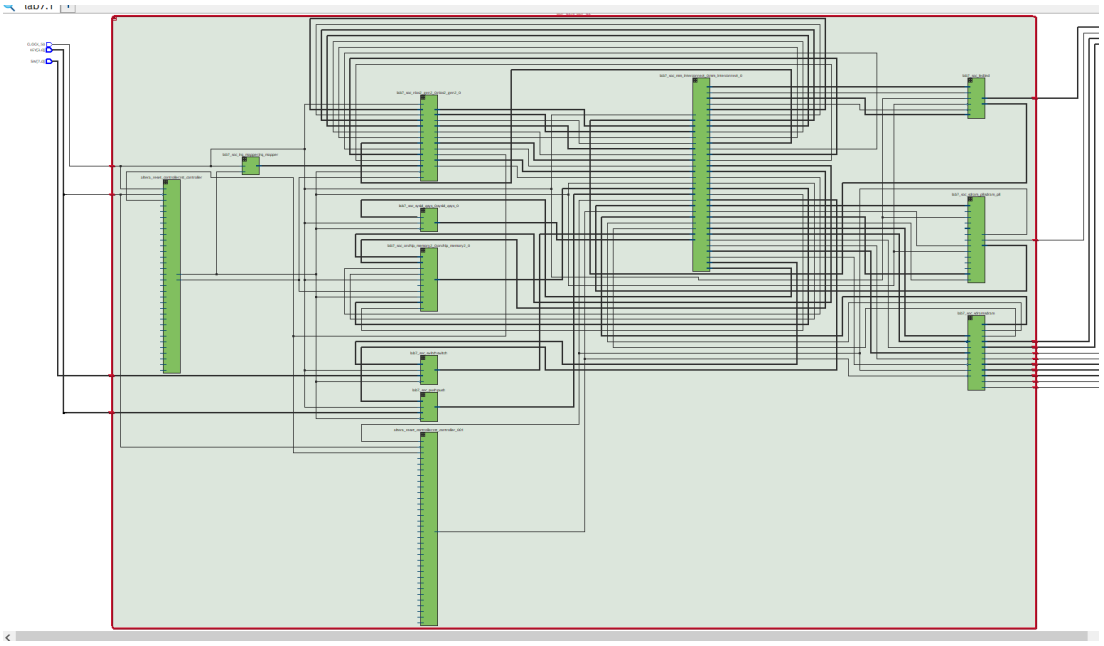
Description: This module connect all the IP blocks built by the platform designer, and provide Inputs/Outputs for the SoC.

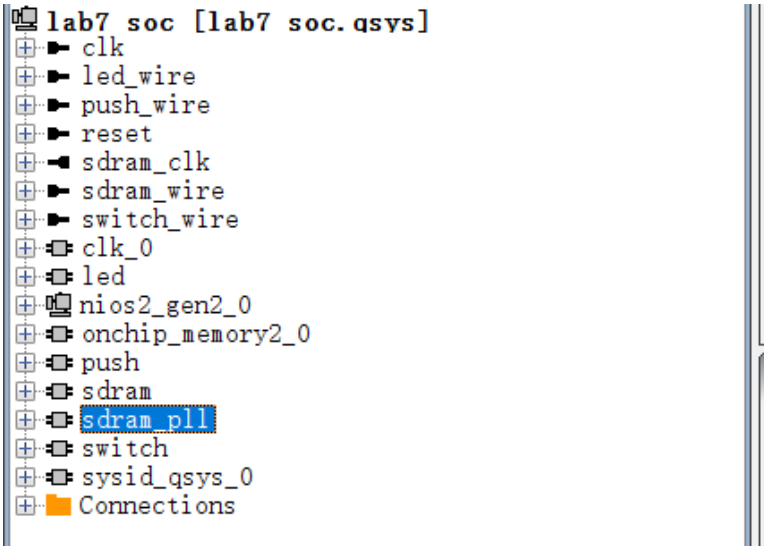
Purpose: Connect the NIOS II Processor, SDRAM and PIO modules created by the platform designer.

❑ Top Level Block Diagram

❑ This diagram should represent the placement of all your modules in the top level. *Please only include the top level diagram and not the RTL view of every module*. The Qsys view of the NIOS processor is not necessary for this portion.


lab7_soc:m_lab7_soc
CLOCK_50
KEY[3..0]
SW[7..0]
clk_clk
3:2
push_wire_export[1..0]
0
reset_reset_n
switch_wire_export[7..0]
led_wire_export[7..0]
sdram_clk_clk
sdram_wire_addr[12..0]
sdram_wire_ba[1..0]
sdram_wire_cas_n
sdram_wire_cke
sdram_wire_cs_n[1..0]
sdram_wire_dq[15..0]
sdram_wire_dqm[1..0]
sdram_wire_ras_n
sdram_wire_we_n
LEDG[7..0]
DRAM_CLK
DRAM_ADDR[12..0]
DRAM_BA[1..0]
DRAM_CAS_N
DRAM_CKE
DRAM_CS_N
DRAM_RAS_N
DRAM_WE_N
1:0
DRAM_DQM[3..0]
15:0
DRAM_DQ[31..0]




lab7_soc [lab7_soc.qsys]
clk
led_wire
push_wire
reset
sdram_clk
sdram_wire
switch_wire
clk_0
led
nios2_gen2_0
onchip_memory2_0
push
sdram
sdram_pll
switch
sysid_qsys_0
Connections

❑ Answers to all 11 INQ Questions

❑ The Prelab question (part A) is one of these 11 questions. The questions are reproduced at the end of this document for convenience.

See Appendix

❑ Postlab Question

❑ Document the Design Resources and Statistics in following table.

LUT	2281
DSP	No
Memory (BRAM)	36864bits
Flip-Flop	1825
Frequency	88.14MHZ
Static Power	102.06mw
Dynamic Power	41.01 mw
Total Power	204.14 mw

❑ Conclusion

❑ Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it

In this expriment, we designed the SoC(System on Chip) with a NIOSII/e, SDRAM and PIO, which performs "accumulation" from switches and output the result into LEDs to display. Behavior is similor to lab4, but using software(written in C) instead of hardware. Pressing 'Reset' (KEY[2]) at any time clears the accumulator to 0 and updates the display accordingly (turns all the LEDs off). Pressing 'Accumulate' (KEY[3]) loads the number represented by the switches into the CPU, adding it to the

accumulator, then detecting whether "overflow" occurs. Push buttons only react once to a single actuation as requested.

❑ Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right so it doesn't get changed.

INQ Question is so hard, and we need to learn most of them by ourselves. Maybe more hits/info about those questions.

APPENDIX

INQ Questions

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

	Nios II/e	Nios II/f
Summary	Resource-optimized 32-bit RISC	Performance-optimized 32-bit RISC
Features	JTAG Debug ECC RAM Protection	JTAG Debug Hardware Multiply/Divide Instruction/Data Caches Tightly-Coupled Masters ECC RAM Protection External Interrupt Controller Shadow Register Sets MPU MMU
RAM Usage	2 + Options	2 + Options

Nios II/e use less resources(LEs) but support fewer features and lower performance, Nios II/f use more Les but support more features and better performance.

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

It provide high-speed cache for CPU execution, which is much faster compared to get data from outside hardware like DRAM. For small program, on-chip memory can boost the performance of the system

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". Since it supports seperate data buses and instruction buses, which is the feature of the Harvard architecture. However both of data and instructions are stored can be accessed through one address space, and allows the content of instruction to be accessed as data, this is the feature for the modified Harvard.

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?

Since the led peripheral only needs the data to display as request, it does not need to access to the program bus to get instructions. But on-chip memory needs to access for both instruction and data for cache.

5. Why does SDRAM require constant refreshing?

In order to hold the data, SDRAM uses capacitor and transistor storage, so it must be refreshed every once in a while. If the storage unit is not refreshed, the electricity in the capacitor is discharged, and the storage information is lost.

6. 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	[width]	32 [bits]
# of Rows	[nrows]	2^13[bits] = 8192
# of Columns	[ncols]	2^10[bits] = 1024
# of Chip Selects	[ncs]	1
# of Banks	[nbanks]	4

Together: 4*8192*1024*32 [bits] = 1 [Gbits]

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

The access time is 5.5 ns, and the data width is 32bit, so we can get:

32bits/(5.5ns * 8 bits/byte) = 727[MB/s]

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

   Given a SDRAM running at a slow frequcency, the refresh frequency of the SDRAM would also be slow. Once the refresh rate is not high enough, the data in the memory might be lost due to the charge lack of storage units.

9. 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.

   Since it takes longer time for SDRAM to read/write data compared to the execution of controller, so if we set the controller clock 3ns behind, it will result in a better synchronous design and avoid possible glitches.

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

    We start the execution from address 0x1000_0000. This address is for the reset vector, which indicates is where the reset program starts. In this case, we need to run the reset code(boot loader) at first before we start to run any other program.

    Since this address should not be overlapped with address of other modules(PIOs) we created, so we first assign the addresses for other modules and then assign the reset vector to avoid conflict. Thus, the reset vector should be assigned until all memory components in the system are assigned properly.

11. 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.*

[line 8]: To tell the compiler that any assignment and change to this address, need to be executed and can not be optimized, in this way, all the assignment for LED will not be neglected by compiler, so alternating the value in LED can works correctly.

[line 10]: Clear the LED

[line 11:]: Create a infinitely loop, make the blinker never stop

[line 12]: Make a software delay, wait for the value in LED hardware stabilize

[line 13]: Light the LED

[line 14]: Make a software delay, wait for the value in LED hardware stabilize

[line 15]: Clear the LED, make LED dark

12. 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, which stores the uninitialized global variables.
e.g. **[static int i;]**, then [variable i] will be saved in this segment.
2. **.heap**: The segment used for standard heap.
e.g. **[ptr = (int*) malloc(sizeof(int))]**, then the block with int size would be in this segment.
using **[*ptr = 0;]** for example to store 0 into this segment
3. **.rodata**: The segment used for read-only data, which should be constants.
e.g. **[const int i = 0;]**, then [variable i] (with value 0) will be saved in this segment.
4. **.rwdata** : The segment used for read-write data, which is able to both read and write.
e.g. **[int i = 0;]**, then [variable i] (with value 0) will be saved in this segment.
5. **.stack** The segment used for standard stack.

e.g. **[int ret = someFunc(x,y,z);]** , the segment (stack) would be used each time we invoke a subfunction, and the necessary info like local variables, %EBP, %EIP etc would be put into this segment.

6. **.text:** The segment is used to store text (string/char).

e.g **[char c = 'T';]**, then [variable c] (with value 'T') will be saved in this segment.