Lab 1

Information

The goal of lab #1 is to get you familiarized with creating, simulating and synthesizing VHDL projects using XIlinx ISE Webpack. It spans week 3 and week 4. It will get you ready for the real stuff - implementing the processor on FPGA.

Lab 1 requires you to invest disproportionate amount of time even though it carries only 10% of your lab marks. The idea behind Lab 1 is to let you learn the 'hard way' what works in hardware and what doesn't (for certain things, you can't get a better teacher than experience). It might take a bit of time, effort and frustration before you are able to write good synthesizable code. The best bet is to go through the notes meticulously and have the hardware in mind while writing the code. Once you get the hang of it, Lab 2 and Lab 3 would be easier. It is somewhat like learning swimming - you have to get into the water, struggle a bit, but once you get going, it's fun.

Kindly note that **Lab 1 homework is an individual exercise** though only one FPGA board will be issued per team.

Here is the lab 1 **Assessment Schedule**.

Tasks

1. Program the FPGA using the simple_count.bit given in this zip file. See the counter in action.
2. Go through the VHDL+FPGA manual, and implement the half adder on FPGA.
3. Implement the Classwork problem (see below for the description) on the FPGA [credits : 2 marks].
4. Realize the Homework problem on FPGA. [credits : 8 marks].
5. Get familiarized with the MIPS simulator/assembler. Write an assembly language program to compute the first N numbers in the Fibonacci sequence and store it in a memory location of your choice. Emulate it using MARS (week 4).

Tasks 1-3 are expected to be completed by the first lab session. Get started on task 4, and clarify any doubts you may have about the problem before you leave the lab. All 5 tasks have to be completed by your designated lab session in week 4.

Classwork Problem

Augment the sample counter (in step 1) as follows :

When a button DIR is pressed, the counter should toggle the count direction. Upon pressing the RESET button, the counter value should be reset (synchronously or asynchronously) to "0000" when counting up, and to "1111" when counting down.

Optional (for enthusiastic folks only, no extra credit) : The counter has 4 different speeds - 1x (the default speed), 2x, 3x, and 4x. A button SPEED is used to change the speed of the counter - each press of SPEED increases the speed, and if the current speed is 4x, it wraps around to 1x.

Note : You will need to debounce DIR and SPEED buttons to observe the desired results (why?). However, debouncing is left as an optional task. Priority of buttons (i.e., what should happen when two or more buttons are pressed together) is left to your discretion.

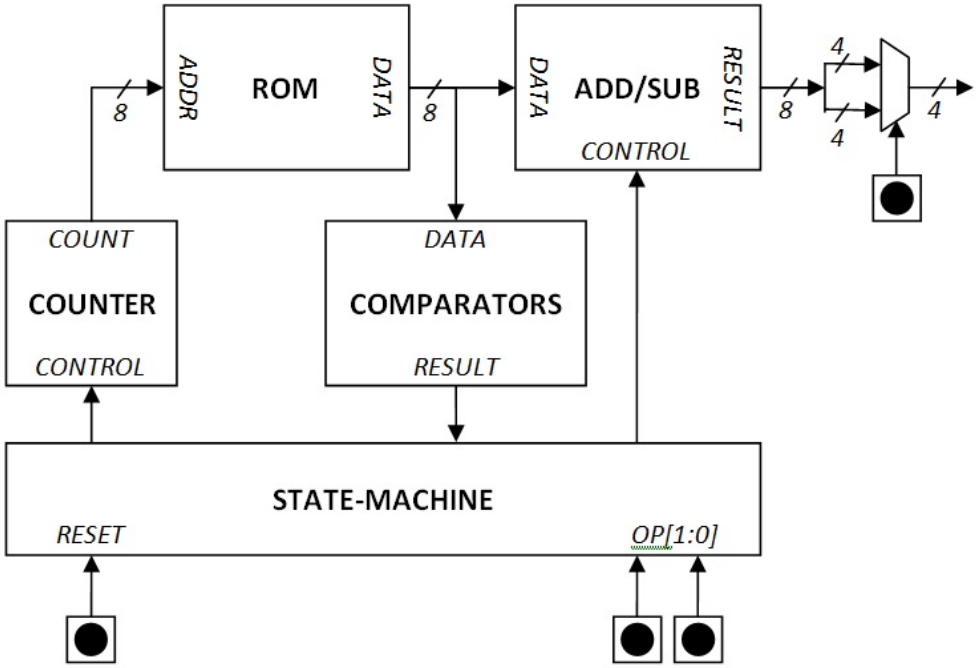
Homework Problem

Please implement the problem in a **modular / structural style**.

- Modularize your system into blocks (entities / components) of appropriate size, with well-defined interfaces.
  - Bigger blocks are harder to test / debug.
  - Smaller blocks limit the amount of optimization done by the synthesis tool (do not go to the extent of implementing gates / very simple stuff as separate entities).
- Test each entity thoroughly.
  - Simulate each entity thoroughly using well designed test-cases.
  - Synthesize each entity by setting it as the top-level module.
- Connect the entities together to form higher level entities. Test them thoroughly as explained in the step above.
- Most warnings are NOT safe to ignore. Identify the reasons for the warnings.

Problem Statement

Design a system that can read-in characters from a ROM (Read-Only Memory) and convert lower-case characters into upper-case character and vice-versa. The block diagram of such a system is shown in figure below.



Details

Each character in a string (array of characters) is stored in 8-bit value according to an ASCII table. For example if we want to store "Hello World" (without quotes), then the array would contain {0x48, 0x45, 0x4C, 0x4C, 0x4F, 0x20, ...}. 0x48 (in HEX) corresponds to 'H', 0x45 to 'E' and so on. Please visit http://www.asciitable.com/ for a list of ASCII codes.

A ROM contains the string that we want to change case. Use the *Language Template\VHDL\Synthesis Constructs\Coding Examples\ROM* in ISE to instantiate a ROM.

The counter generates the address for the ROM. The data from the ROM goes to the comparators. If the data is between 0x41 and 0x5A, then the character is upper-case. If the data is between 0x61 and 0x7A, then the character is lower-case. If it is out of these ranges, it is not an alphabet.

The comparator compares the data whether the data is either upper-case letter or lower-case letter or neither. The result is provided to the state-machine.

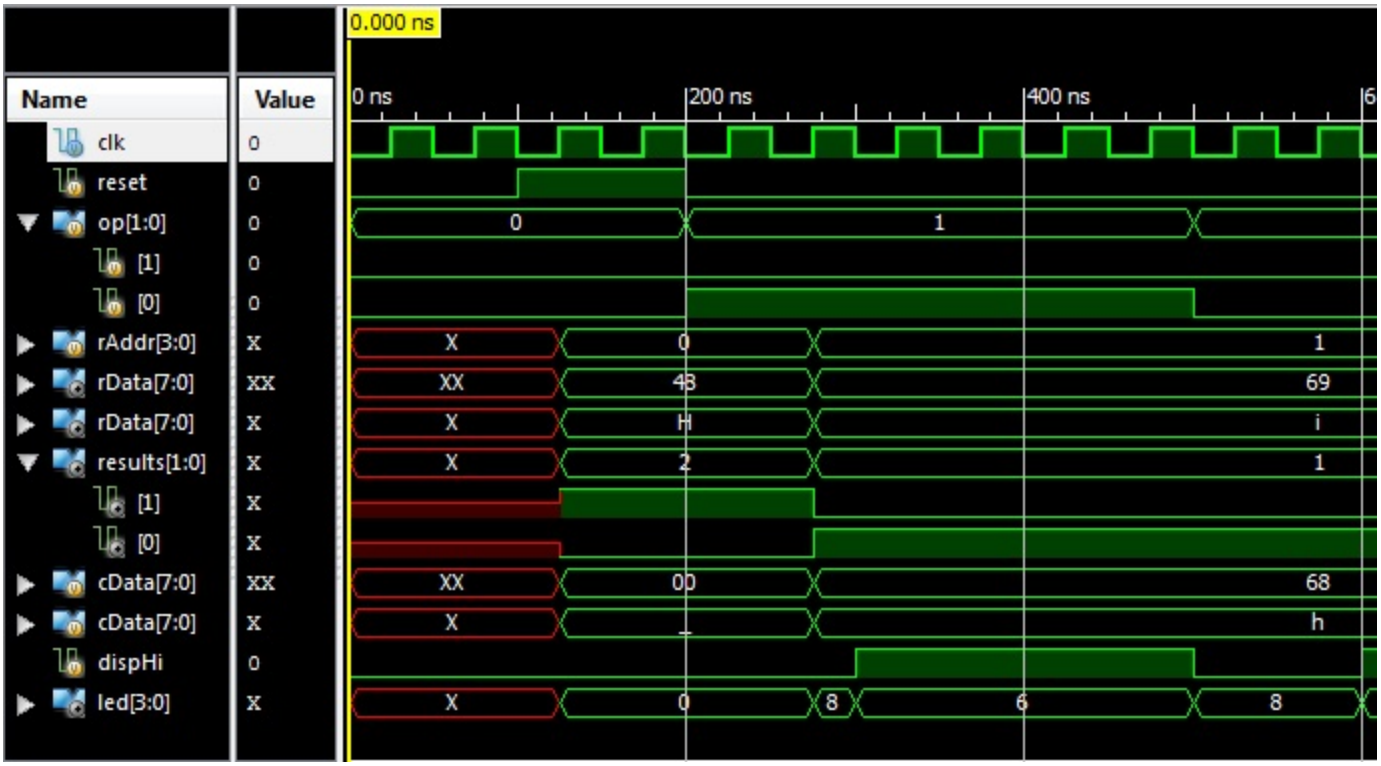
The Adder/Subtractor can convert the case of the data from the ROM to lower by adding 0x20. Similarly, it can also convert to upper-case by subtracting 0x20. If the data is not an alphabet, the data can remain unchanged. The resulting data is placed on the output of the ADD/SUB module. The operation performed by the ADD/SUB module (i.e. (1) ADD 0x20, (2) SUB 0x20, and (3) NO OPERATION) is under the control of the state-machine.

The state-machine has a reset input that causes the system to start reading from the start. The OP[1:0] is two-bit input that has the following meaning:

- 00: No Operation
- 01: Read and Convert to lower-case
- 10: Read and Convert to upper-case
- 11: No Operation

The output of the ADD/SUB is 8-bit. To display it on 4 LEDs, we need to display multiplex the LEDs using the fourth push button such that when the button is pressed, the LEDs shown the higher 4-bits and when not, it shows the lower 4-bits of the output of the ADD/SUB module.

Timing Diagram



Legend:

clk: Clock Signal	**reset:** Reset Signal	**rData:** Read Data from ROM (shown both in Hex and ASCII)	**results[1:0]:** Comparison results from comparator (MSB indicates an Upper-case character while LSB indicates a Lower-case character)
op[1:0]: OP Signals	**rAddr:** ROM Address	**cData:** Converted Data from ADD/SUB module (Also shown in both Hex and ASCII)	**dispHi:** Push button to select upper/lower bits
led: Output LEDs			

Initially, reset is asserted (from 100ns to 200ns). This resets the complete system to a predefined state. The 'X's denote don't cares. In this case, synchronous reset is demonstrated although asynchronous is also acceptable.

At 200ns, OP[0] button is pressed (to convert to lower-case). Action for this is taken at 275ns by incrementing rAddr (and thus reading the next ROM location) and updating cData with the converted character corresponding to the previous rData. It is important to note that although the OP[0] is still pressed, no further action is taken until it is release and one of the OP buttons is pressed again. Similarly, OP[1] is pressed at 800ns. Similar action for it is taken at 875ns.

The dispHi is also shown to select the upper/lower bits of cData on the LEDs.

Submission Info

Please upload the LAB 1 latest by **5 Sep 2014**. Those who are doing the demo in week 6 should upload by 19 Sep 2014.
Please include the **RTL sources, testbench(es), .txt files** (if you have done textio), the **.ucf** and **.bit files** to the IVLE in the format **<Matric No>_Lab1.zip**.

Tips

- Read all the inputs synchronously. Use only synthesizable process templates I (purely combinational) and III (purely synchronous).
- ROM need not have a clock. Please see one of the comments below this page for a discussion on this.
- Do Not use 'event attribute for signals other than clock. i.e., do not use something like tick'event. If you need to detect activation of tick (i.e., to detect a rising edge of an external input), read it inside a synchronous process (without tick being in the sensitivity list), store the previous value in a variable, and use tick='1' and tick_prev = '0'.
- Debouncing inputs will help you make the circuit response predictable, making sure that you read only one character per button

press.Typically, you have to wait for about 10-100ms for debouncing. This can be implemented using a simple down-counter. The number of bits can be calculated using the formula D=(2^n)/f (where D is delay in seconds, n is the width of the counter in bits and f is the operating frequency). Once the input changes (previous value = 0 and current value = 1), start the counter. After the counter reaches terminal count i.e. zero, copy the value of the input to the output of the debouncer (i.e., the press is valid if the current value = 1 even after the delay).

- While there is no guarantee that a design which can be simulated it synthesizable and works in hardware, a design which cannot be simulate correctly is more or less guaranteed not to work on hardware.
- DO NOT dump all your code into a single entity - while this could be slightly more efficient with respect to hardware usage (you are passing the onus to synthesis tool), this is not a good design practice. Most of you are aware of software engineering principles - many of which apply to writing HDL code too. The right way to go about it is to split the hardware into reasonably sized blocks based on functionality (but not to the extent of having individual gates as separate entities), implement each block as a separate entity in a separate file, simulate and synthesize them independently (by setting them as lop level entities), and use them in a higher level entity as components. This hierarchical design practice is essential for future lab sessions.

FAQs

1. Are the values stored in ROM?
   YES. Minimum size of ROM should be 16 characters.

2. When we press either the OP buttons, how the system should react?
   When you press either OP buttons, it should read in ONE character from the ROM and convert it appropriately. Even if you have kept the OP button pressed, it should still read only ONE character. To read the next one, release the OP buttons and press it again.

3. How to implement debouncing?
   See the tips above.

4. What should be displayed when we press no operation?
   The previous value should be held until OP buttons are pressed again.

5. Can I use variables to store flags etc in my state machine code?

   Ans. Ideally, the SM process should need only those flip-flops required to encode the states, and nothing else. If you need stuff like flags, you haven't identified the required states properly (i.e., you are implicitly introducing states). Yes, the same task can probably be done with different SM structures, considering flag as a separate module (flip-flop) controlled by controller SM will be more consistent with the spirit of modularity (chances are that it will be less efficient than splitting the state where you need the flag into two separate states).