3C7 Digital Systems Design

Lab Session F

Aim:

The purpose of this session is to simulate and explore sequential logic circuits in Vivado and implement them on the Basys-3 Board.

NOTE: This is a longer lab, part-C of which is used in Lab G and Assignment-2 (with minor modifications). Complete this and lab G over weeks 9/10 before moving to Assignment-2. Aim to complete Part A & B in week 9 and advance to Part C when you complete.

Learning Outcomes:

On completing this lab session, you will be able to:

- Write a Verilog module using sequential logic
- Set up a clock and reset in a Verilog testbench
- Construct an LFSR in Verilog
- Write a simple Verilog testbench for a sequential circuit

Required Reading

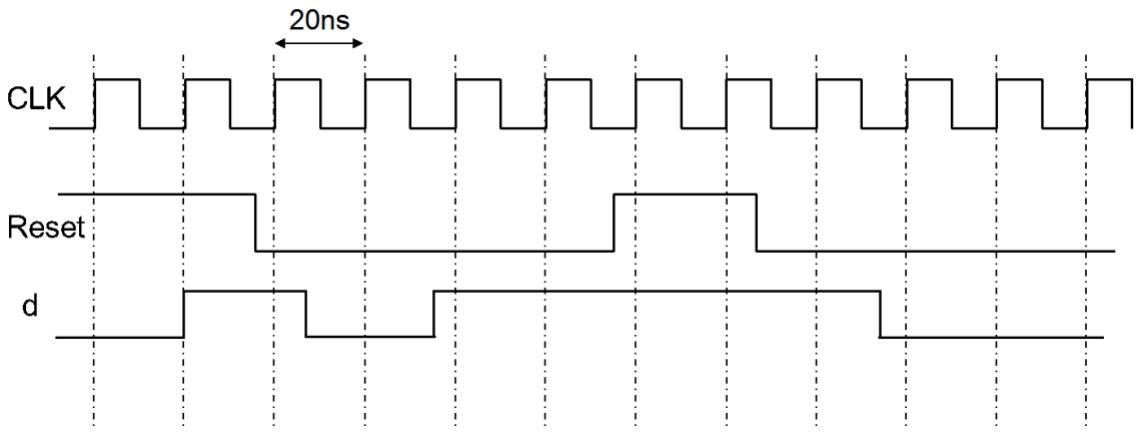
You can refer Section 4.4 "Testbench for Sequential Circuits" in the Chu Course Text. It will be helpful to see how you can set up a clock and reset signal. For other inputs in a sequential circuit, you can choose to use a similar style testbench to those you have written to date, or use this approach where you time on clock edges.

Part A: Clock and Reset

To develop testbenches for sequential circuits, you will need to incorporate a clock and a reset into your generic testbench structure. To practice this, you can use the simple module `d_type_ff.v`. Write a testbench to instantiate this module.

Instructions:

1. Start a new project.
2. Add the file `d_type_ff.v` to your project and create an appropriately named testbench.
3. Look back at Lecture 6 to see examples of clock creation. Also refer to the above section in Chu. Set up a clock with a 20ns period.
4. Using another initial block, create a reset signal which is initially high. The reset should (closely) follow the timing shown in the diagram below.



5. In a similar manner, create a testvector for input **d** which behaves as shown in the timing diagram. Complete your expected value of **q** in the timing diagram above and keep to hand to compare to your waveform after simulation.
6. Run the testbench and monitor all the signals in the module in the waveform viewer. Save a copy of the waveform that clearly shows the changes in the output **q** in response to the inputs you have created.
7. Change the module to make the d-type flip-flop **negative edge** triggered. Re-run the testbench. Save a copy of the waveform that clearly shows the changes in the output **q**. Consider the difference from the previous version of the code.
8. Your module has an asynchronous reset. Now change the module to have a **synchronous** reset (remember to recompile etc. and reload the design).
9. Run the testbench with the changed reset. Monitor all the signals in the module in the waveform viewer. Save a copy of the waveform that clearly shows the changes in the output **q** in response to the inputs you have created.
10. Examine your two waveforms and observe the difference in timing you notice between the **q** output using synchronous and asynchronous reset. (HINT – they should NOT be the same)

Part B: Practical use of DFF

In Lecture 6, we discussed that DFF's are utilised to save the state of the system and can be combined with combinational logic to perform interesting circuits. In this section, you will use a series of DFFs to save the current state of a value (say, temperature) and use the left and right push buttons to increase or decrease value displayed on the seven-segment display on the board. To do this, download the debounce.v and sevenseg.v from blackboard and use it as instructed below.

NOTE

We are creating a counter here whose working is discussed in Lecture 6. Refer to the lecture slides to understand the operation.

Instructions:

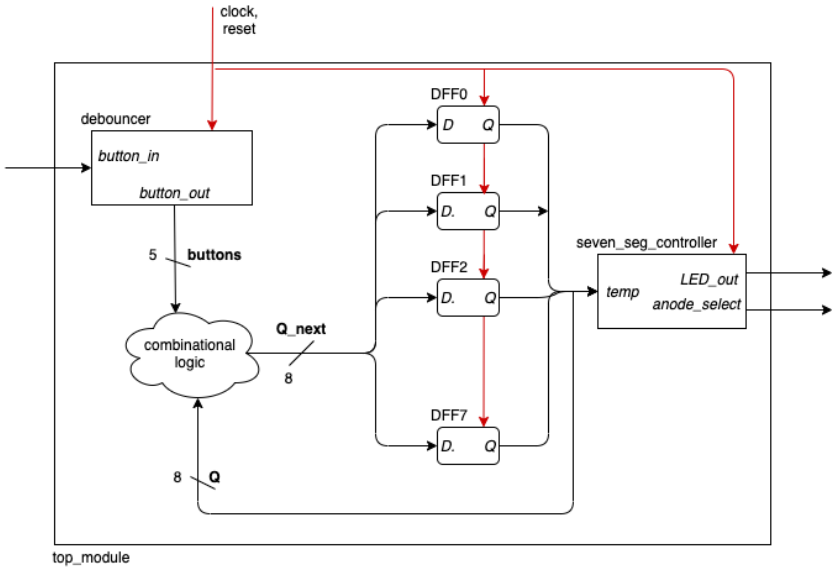
1. Start a new project. Download the practical_dff.zip file from Blackboard and unzip them to your project directory.
2. Add the file **`d_type_ff.v`** (the same DFF file you started with above), **`debouncer.v`** and **`sevenseg.v`** (from the unzipped files) to your project and create an appropriately named top

file. Define signals clock and reset (clk, reset) in the top module along with outputs for the seven segment display (4-bit anode selection to select one of the 4 7-segment modules on the board and a 7-bit LED output vector displays the value) and push button inputs (5-bit vector for the push buttons).

   a. **NOTE**: For easiness, you may use the following names for the signals as these are already mapped in the xdc file you will add in step 12
      i. Clock signal: **`clk`**
      ii. Reset signal: **`reset`**
      iii. Anode Selection (7 Segment): **`anode_sel`** – 4 bit vector
      iv. LED Output: **`led_out`** – 7 bit vector
      v. Push Buttons: **`button_input`** – 5 bit vector

3. Instantiate 8 versions of the DFF (say DFF0 to DFF7), one corresponding to each bit of an 8-bit vector we want to save – this would represent a temperature value from 0 to 127 degrees.
4. Define an 8-bit wire vector ***`Q`*** and connect individual bits of this vector to the output of the DFFs (i.e., bit 0 of Q connects to output of DFF0 and so on). Connect **`Q`** to the 8-bit **`temp`** input of the seven-segment controller module.
5. Instantiate the debouncer module. Define a 5-bit wire vector **`buttons`** and connect this to the **`button_out`** port of the debouncer module. Connect the button input port defined in step 1 in your top module to the **`button_in`** port of the debouncer module.
6. Define an 8-bit signal ***`Q_next`*** which will hold the next state (Increment or Decrement) of the Q value depending on whether the UP/RIGHT or DOWN/LEFT switch was pressed. Use a combinatorial always block to generates **`Q_next`** as an increment/decrement of the current value of **`Q`**, based on the button pushed. We define the button assignment in relation to the **`buttons`** vector as below:
   a. UP – **`buttons[0],`** `LEFT` **`– buttons[1],`** `DOWN` **`– buttons[2],`** `RIGHT` **`– buttons[3] and`** `CENTRE` **`– buttons[4]`**
   b. Condition – `if UP or RIGHT = 1, increment Q to get Q_next; if DOWN or LEFT = 1, decrement Q to get Q_next.` Use a `switch` or `if-else` construct to describe this behaviour.
7. Add a condition in the above statement to load a value of decimal 22, if the CENTRE switch is pressed.
8. Assign the individual bits of the **`Q_next`** vector to the inputs of the DFFs in the corresponding order (i.e., Q_next[0] fed as the D-input of DFF0 and so on). This completes the logic part our counter logic. Check all connections to make sure that they are wired up exactly as described above.
9. Instantiate the `seven_segment_controller` module and connect the **`LED_out`** and **`anode_select`** output ports of this module to the top-level LED and anode selection ports described in step 1. Connect the wire **`Q`** to the **`temp`** input of the seven-segment controller.

10. Connect the clock and reset signals to the `debouncer`, DFF and `seven_segment_controller` modules. When completed, the block diagram will be similar to the one shown below.



11. **NOTE:** We are not using a test bench here to verify the functionality, which is not ideal – this is primarily because of the debounce module reduces the sampling time of the switches drastically to ensure that we do not get multiple pulses corresponding to one button activation (due to the spring contact mechanism).

12. Add the **`practicaldff.xdc`** from the unzipped folder (step-1) into the project. Map the top-level port names from your top module to the signal names in the xdc file (see comments in the file to understand which port to assign to the corresponding pin).

13. Save the project, run the synthesis, implementation and bitstream generation steps. Program the board with the generated bitstream. Set SW0 (switch 0 on the board) to high for one second and then move it back to low – SW0 is mapped as the active high reset in this exercise.

14. Does the implementation work as expected? Does the counter increment, decrement and reset to 22 when the different buttons are pressed? Record your observations for submission.

If we click too fast it doesnt register

Part C: LFSR Implementation

In Lecture 6, we looked at the design of LFSRs. You are required to implement a maximal length, i.e. 2^N-1, LFSR counter in Verilog. The document **lfsr_setup.pdf** gives the value of N that each student must use for their module and the feedback (XOR or XNOR). In this lecture, you received a handout of tap values for implementations of maximal length LFSRs (remember to watch out for the numbering convention used though). Using this information and the sample code from the lecture, construct your LFSR.

Instructions:

1. Start a new project.
2. Draw a block diagram of your module clearly showing the taps you are taking the feedback from and how you generate the feedback. (example in Lecture 6)
3. Write the Verilog for the module, using an asynchronous active-high reset (i.e. just like in the example). You may use the template in Lecture 6.
4. Set up a local parameter to hold the seed value for the LFSR.
5. Generate an extra single bit **registered** output called **`max_tick_reg`** which goes high for a single clock cycle when the full count of your LFSR is reached (i.e. after 2^N-1 cycles when it reaches the seed value once more).
6. Construct a testbench to test your module. Generate a clock signal with a suitable period. The testbench should have a reset high for about 10 clock cycles at the start of the test and then remain low for the remainder of the test. (hint – you can mostly reuse the previous testbench format)
7. You will find it useful to work out how long to run the simulation for, based on the clock period and the size of your LFSR.
8. Using the waveform viewer, show that your LFSR asserts the *max_tick* signal after 2^N-1 cycles (after the reset is low) and then starts again. You might use multiple waveforms to do this. Also recall the use of multiple cursors to show the time between events on a signal in the waveform window in the Vivado tutorial.
9. Set your seed value to a "forbidden" value for your LFSR, (refer to lecture 6). Examine the behaviour of the LFSR in this case.
10. Include your module and testbench in your submission. You can now attempt the advanced part that follows.

Advanced section

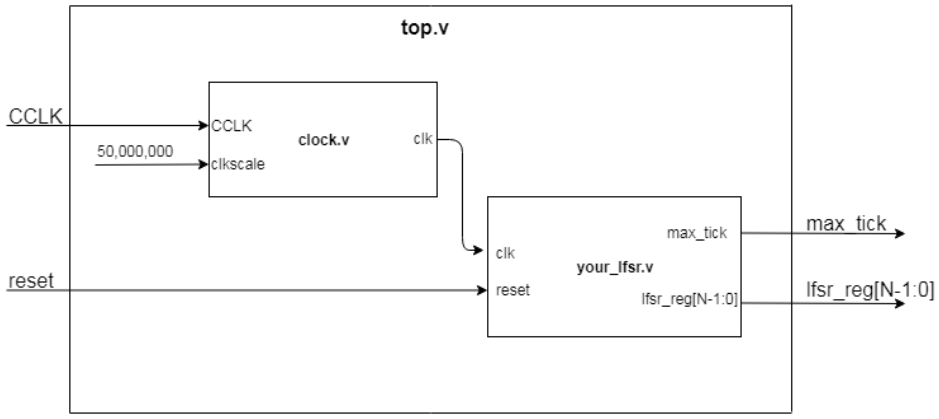
11. In the example in Lecture 6, the MSB of the LFSR is output from the module.
12. Write an extra module that takes this bit as an input and counts how often the bit is 1, and how often the bit is 0 within a full 2^N-1 cycles of the LFSR. Comment on what you find from this.
13. The counting process must restart whenever the LFSR does.
14. Draw a block diagram of the module showing the inputs and outputs you need and the high-level elements of the module.
15. Show a waveform to demonstrate how many 1s and 0s you count.
16. Show another waveform that shows the counting starting after a reset.
17. Show another waveform showing the counter restarting after 2^N-1 cycles (i.e. when the *max_tick* is reached)

Part D: LFSR - Target to Board

You will now use the clock on the FPGA for the first time to observe your LFSR in action. Read the handout "Using Clocks" before you attempt this.

Instructions:

1. Create a top module which includes a clock divider and your LFSR, to enable you to run your LFSR with a slowed down 1Hz clock. You should use the output of your clock divider module as your clock input to the LFSR.
2. The inputs and outputs for the module should look like this:



3. Create an XDC file for your project using the I/O Planning view in Vivado. You should tie the CCLK (crystal oscillator clock) input to pin W5 of the board. You should tie each bit of the LFSR's value to an LED, including an additional LED for max_tick_reg to alert you when a full 2^{N-1} cycles have been completed. The I/O standard for all ports will be LVCMOS33.
4. (Note: if your LFSR has a large number of bits, you may not want to wait for the full number of cycles!)
5. Your clock constraint should look something like this when you view your constraints file:

```
set_property PACKAGE_PIN W5 [get_ports CCLK]
set_property IOSTANDARD LVCMOS33 [get_ports CCLK]
```

6. Run bitstream generation and target your design to the Basys 3 board. You should be able to see your LFSR cycling through pseudorandom values at a rate of 1 per second.

Submission:

Capture the waveform of DFF, LFSR working and the advanced optional part. Capture the block diagram into the **short report**.

You must submit the following items via Blackboard in a **single zip file:**

- All code and testbenches from this assignment
- All code MUST be suitably commented
- Short report capturing the waveform and block diagram.

Deadline:

Group A (Tuesday labs): Tuesday 28th March 2023, 11 PM.
Group B (Friday labs): Friday 31st March 2023, 11 PM.
All submissions are via blackboard. This report is worth 10% of your CA.

Plagiarism

Plagiarism is interpreted by the University as the act of presenting the work of others as one's own work, without acknowledgement. Plagiarism is considered as academically fraudulent, and an offence against University discipline. The University considers plagiarism to be a major offence, and subject to the disciplinary procedures of the University.

Visit http://tcd-ie.libguides.com/plagiarism

All students must complete the TCD Ready Steady Write plagiarism tutorial and sign a declaration when submitting course work, whether in hard or soft copy or via Blackboard, confirming that you understand what plagiarism is and have completed the tutorial. If you read the information on plagiarism, complete the tutorial and still have difficulty understanding what plagiarism is and how to avoid it, please seek advice from your College tutor, your Course Director, your supervisor, or from Student Learning Development.

e.g. having exactly the **same testvectors or module** as another person **is plagiarism**.

Plagiarism will result in loss of **ALL** marks for this assignment.