Verilog & FPGA Advantages

ENEE 245: Digital Circuits and Systems Laboratory
Lab 6

Objectives

The objectives of this laboratory are the following:

- To understand just how powerful are the tools of behavioral Verilog and logic synthesis
- To get a taste of what real test scenarios look like

As mentioned before, Verilog is a powerful language, and writing code that produces hardware makes design and debugging of that hardware far simpler than using a breadboard. As a result, you will find that using Verilog makes is easy to design circuits of incredible complexity. Software design of hardware (i.e., CAD, or computer-aided design) is the only way that modern computer chips could be built. To try to design and test these things by hand would be nothing short of impossible.

In this lab you will create several extremely large circuits, including a fairly powerful state machine, using a minimum of Verilog code. The focus will be on the testing and verification processes.

Pre-Lab Preparation

64-bit ALU Components

Each of your components must be in a *separate module*, so that you can get separate timing numbers for each. If all code is in the same module, you will not be able to differentiate between components.

Design a 64-bit adder, using behavioral Verilog. It should produce a 65-bit result, where the 65th bit represents an overflow scenario.

Design a 64-bit subtractor, using behavioral Verilog. It should produce a 65-bit result, where the 65th bit represents an overflow scenario.

Design a 64-bit ANDer, using behavioral Verilog. It should produce a 64-bit result, representing the bitwise AND of its two input values.

Design a 64-bit ORer, using behavioral Verilog. It should produce a 64-bit result, representing the bitwise OR of its two input values.

If done right, these modules should be relatively small; the adder and subtractor in particular can be written in just a few lines of code.

Linear Feedback Shift Register

Design a 64-bit linear feedback shift register, using behavioral Verilog. This should be clocked on the rising edge of a clock input. A 64-bit LFSR has taps at bits 64, 63, 61, and 60. For instance, here is an 8-bit Galois LFSR with taps at bits 8, 6, 5, and 4:

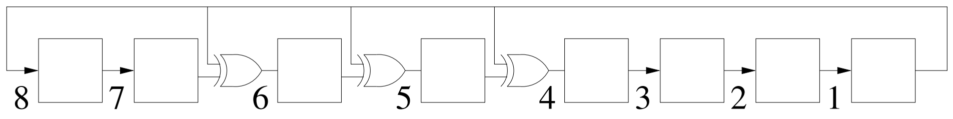


Figure 1: An 8-stage Galois LFSR with cycle size 255. This LFSR has taps at positions 8,6,5 and 4.

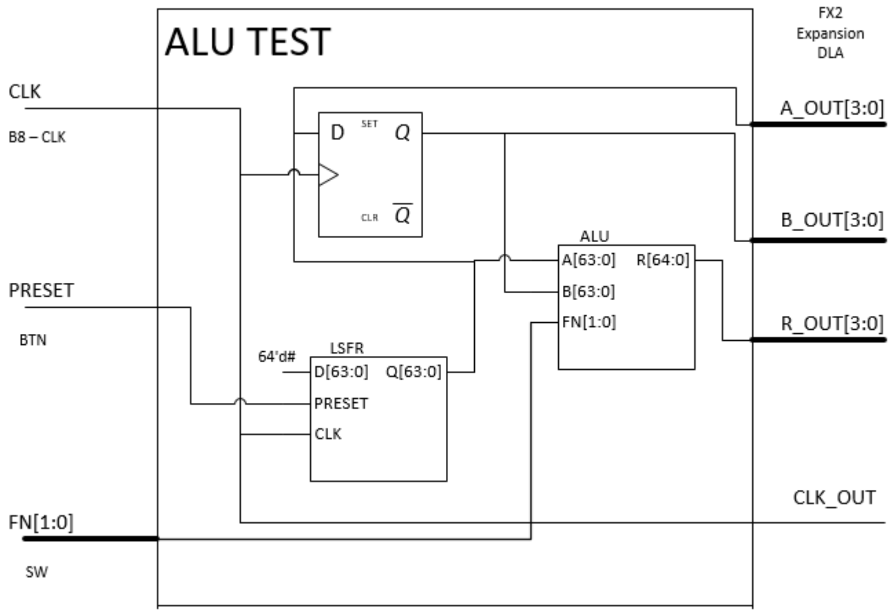
And here is the Verilog code for that 8-bit Galois LFSR (bits renumbered 0 through 7):

```
module LFSR10 (dbus, clk, preset, qbus);
      input [7:0] dbus;
      input clk;
      input preset;
      output [7:0] qbus;
      reg [7:0] qbus;

      always @ (posedge clk) begin

            if (preset)
                  begin
                        qbus <= dbus;
                  end
            else
                  begin
                        qbus[7] <= qbus[0];
                        qbus[6] <= qbus[7];
                        qbus[5] <= qbus[6] ^ qbus[0];
                        qbus[4] <= qbus[5] ^ qbus[0];
                        qbus[3] <= qbus[4] ^ qbus[0];
                        qbus[2] <= qbus[3];
                        qbus[1] <= qbus[2];
                        qbus[0] <= qbus[1];
                  end

      end
endmodule
```

Note that the all-zeroes state for the shift register is not allowed (it will cycle forever), so be sure that your initialization mechanism sets it to some non-zero value.

Test Harness

The following figure shows an example setup for a test harness, using the ALU and LFSR.



Design a test harness using your LFSR to drive different sets of inputs to your ALU components and verify that, for each input set, each ALU component produces the correct output. On every cycle, transfer the contents of the LFSR to a 64-bit register. Use these two values—the output of the LFSR and the contents of the register, which is the previous value of the LFSR—as your ALU's test inputs. Test your circuits on the first 100,000 input values. Note that, in an industrial setting, you would want to test every possible combination of inputs whenever possible.

Simulate your code and bring a pre-lab write-up showing the code and simulation results.

Your ALU components and LFSR should be synthesizable, but the test harness that compares the output and verifies its correctness should be in a separate file that will not be synthesized and will not be put onto the FPGA. That functionality will instead be done with the DLA and your eyes.

Simple Performance Analysis

Use the design tools to extract the expected latency for each component. What is the fastest clock speed you should expect to be able to achieve?

In-Lab Procedure

Bring flash drives to store your data.

Ask the TA questions regarding any procedures about which you are uncertain.

Complete the following tasks:

- **Program your system onto the FPGA board.**
- Connect the FPGA to the break-out board.
- Run the system and use the DLA to observe the bottom 4 bits of the various components. Save the DLA's output for your post-lab report.
- Look at the detailed RTL schematics produced by the software; save it for your post-lab report.
- Look at the timing report that gives the pin-to-pin delays for input/output combinations of every pin. Save these reports and tables for your post-lab report.

Post-Lab Report

Write up your code, schematics, and lab procedures. Demonstrate the correctness of your designs through your pre-lab simulations and note any differences between what you simulated and how the circuits behaved in the lab.

Regarding the RTL schematics produced by the software—how did the design software synthesize your code? Where did it choose poorly, and how could it have done better? Could you have better specified your design to get more efficient results?

Regarding the timing report that gives the pin-to-pin delays for input/output combinations of every pin—what does the design software say for the timing? How fast is each component?