CSE125 Lab1: Behavioral Verilog

Due Date: Wednesday 1/25/2020 11:59pm

Before starting, create an account at git.ucsc.edu and submit your username to this form. This will provide you access to testbenches for this lab.

I. Linear Feedback Shift Register (10 Points)

Figure 1 shows a linear feedback shift register (LSFR). LSFRs are frequently used to produce pseudo-random sequences via XOR gates with low overhead in hardware. For instance, the circuit below computes 8-bit numbers which are completely predetermined but which appear random as they only repeat after a very long cycle.

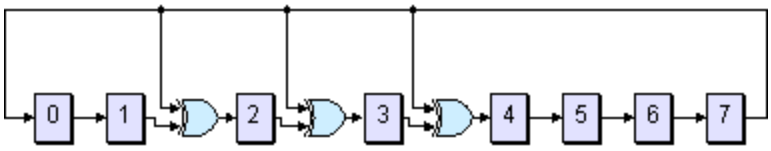

Figure 1: Linear Feedback Shift Register

Implement the LSFR described in Figure 1 in two versions, one using structural and the other using behavioral Verilog. The code needs to be implemented as part of a module with the signature below. You need to develop a testbench to test your module. You need to synthesize your modules and report the consumed resources and the maximum operating frequency.

```
module lfsr(input clk, input res_n, input [7:0] data_in,
            output [7:0] data_out );
           /* Your code here */
endmodule
```

The Vivado documentation linked below has useful information regarding Verilog and available structural elements.

- Vivado Design Suite User Guide - Synthesis
- Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide

Submit via Git the following files. (No Canvas submission)

a) A Verilog file named lfsr_structural.v that implements the structural lfsr.
b) A Verilog file named lfsr_behavioral.v that implements the behavioral lfsr
c) A screenshot of a waveform stored as waveform.png showing all inputs and outputs over the duration of the 10 clock cycles (in addition to reset time etc.)
d) A write up that lists the 10 outputs generated given the input defined in c). The write up should also contain the synthesis results (resources and Fmax).

II. Build a Multiplier using an Accumulator (10 Points)

Construct a multiplier by accumulating the results of an adder. Below is the Verilog template.

```
module multiplier(input clk, input res_n, input start, output done,
                  input [15:0] arg1, input [15:0] arg2,
                  output [31:0] product);
                /* Your code here */
endmodule
```

You can assume that the start input is pulsed for one clock cycle and after it is pulsed the arg1 and arg2 inputs are held stable until the done output goes high. Also, assume that the logic driving the multiplier is well behaved (it does not toggle start until the multiplier is done with the last computation).
To decrease cycles per instruction (CPI) develop a second implementation that accumulates up to four times the input per clock cycle (Use multiple additions per cycle but no multiplication). Ensure that both implementations compute the correct result, also for inputs that are not a multiple of four.
Synthesize both designs and to determine resource consumption and Fmax. Compute the total runtime in seconds for computing the result of “67*43”.

Submit via Git the following files (No Canvas submission)

a) A Verilog file named multiplier_1.v that implements the multiplier using an accumulator.
b) A Verilog file named multiplier_4.v that accumulates an input up to four times
c) A screenshot of a waveform stored as waveform_mul.png showing all inputs and outputs for the multiplication in h)
d) A write up containing the synthesis results including resources (FFs and LUTs) and Fmax for both implementations. Furthermore, list the total runtime in seconds for computing the result of “67*43”.