Adam Cole
UID: [redacted]
Project 1 Lab Report

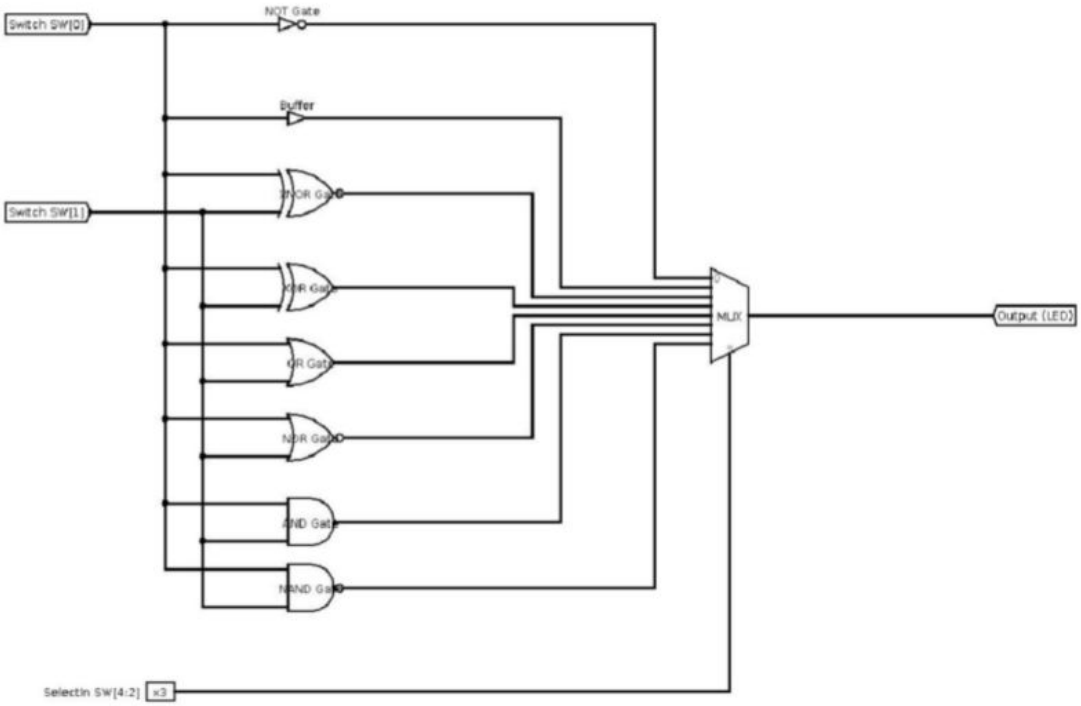
1. Introduction and Requirements

The intention of Project 1 was to become familiar with Xilinx as a hardware development tool as well as coding in Verilog, a hardware description language. In this project we design combinatorial circuits, 4 bit counters using both sequential circuits with D-Flip Flops as well as with a clock and variables. Finally, we use counters to implement a clock divider, which divides a 10kHz frequency clock into a 1kHz clock using a 4 bit counter.

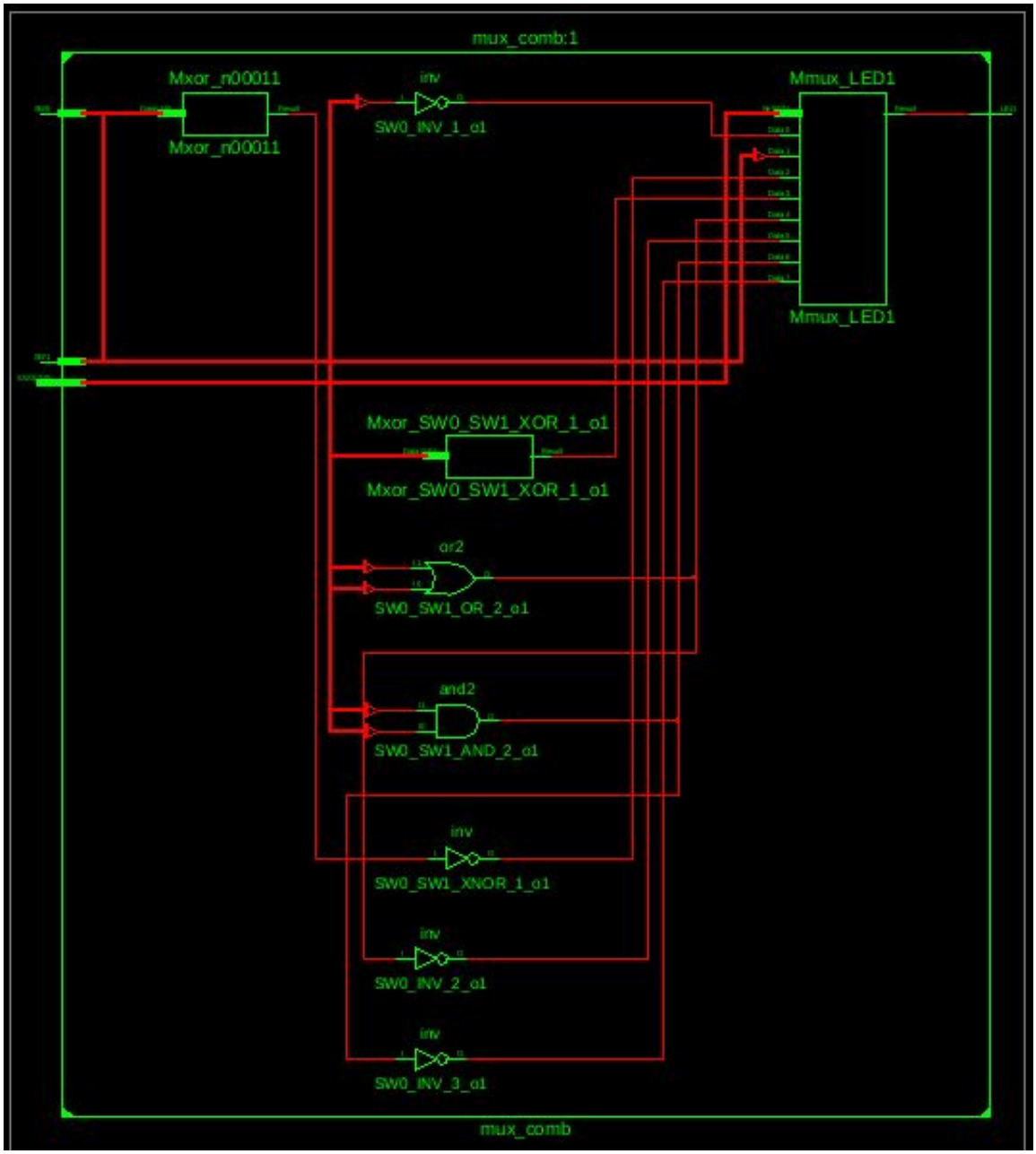
2. Design Description

2.1 Combinatorial Circuit

For the Combinatorial Circuit, the spec gave us a drawn circuit for us to implement in Verilog through Xilinx and simulate its waveforms. The given diagram is shown below.



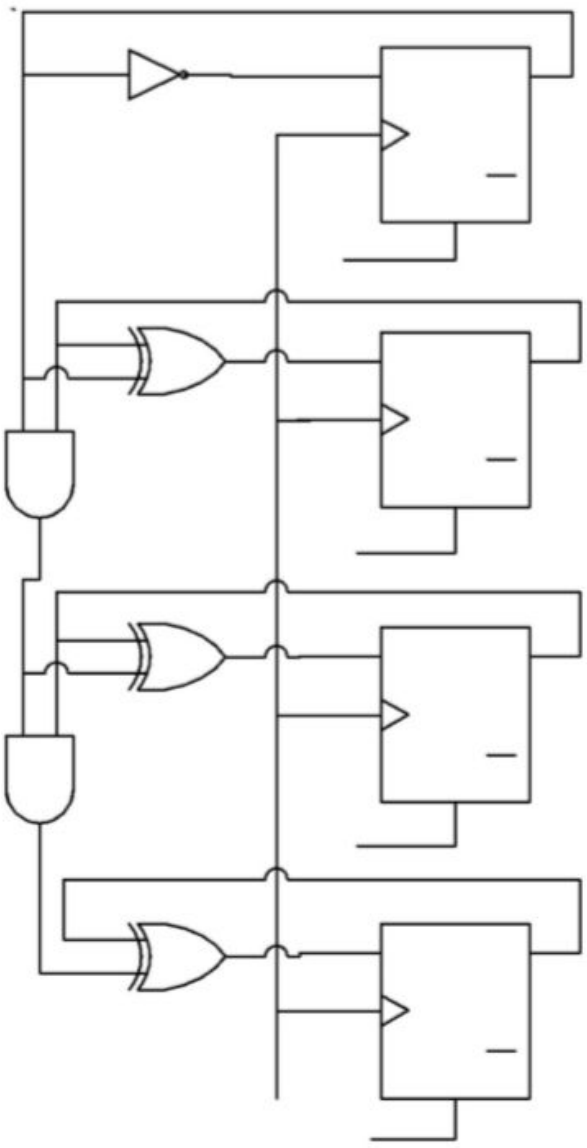
In order to implement this, I created only one module for the multiplexer (8:1 MUX), and hard-coded the included combinatorial logic into the wires of the multiplexer. Each of the gates used in the MUX could simply be implemented through bitwise operators. The inputs to my module were the 5 Switch values shown in the diagram - SW0, the top input bit, SW1, the second input bit, and finally SW2[2:0], which represent 3 MUX input wires to decide which logical path to send to the output LED. By simply turning SW2 into a register, a case statement in verilog easily completed the MUX module. The Xilinx RTL schematic of my module is shown below.



While Xilinx creates some ‘black-boxes’ to encapsulate some of the details of my module, the RTL matches the circuit given to us in the spec diagram.

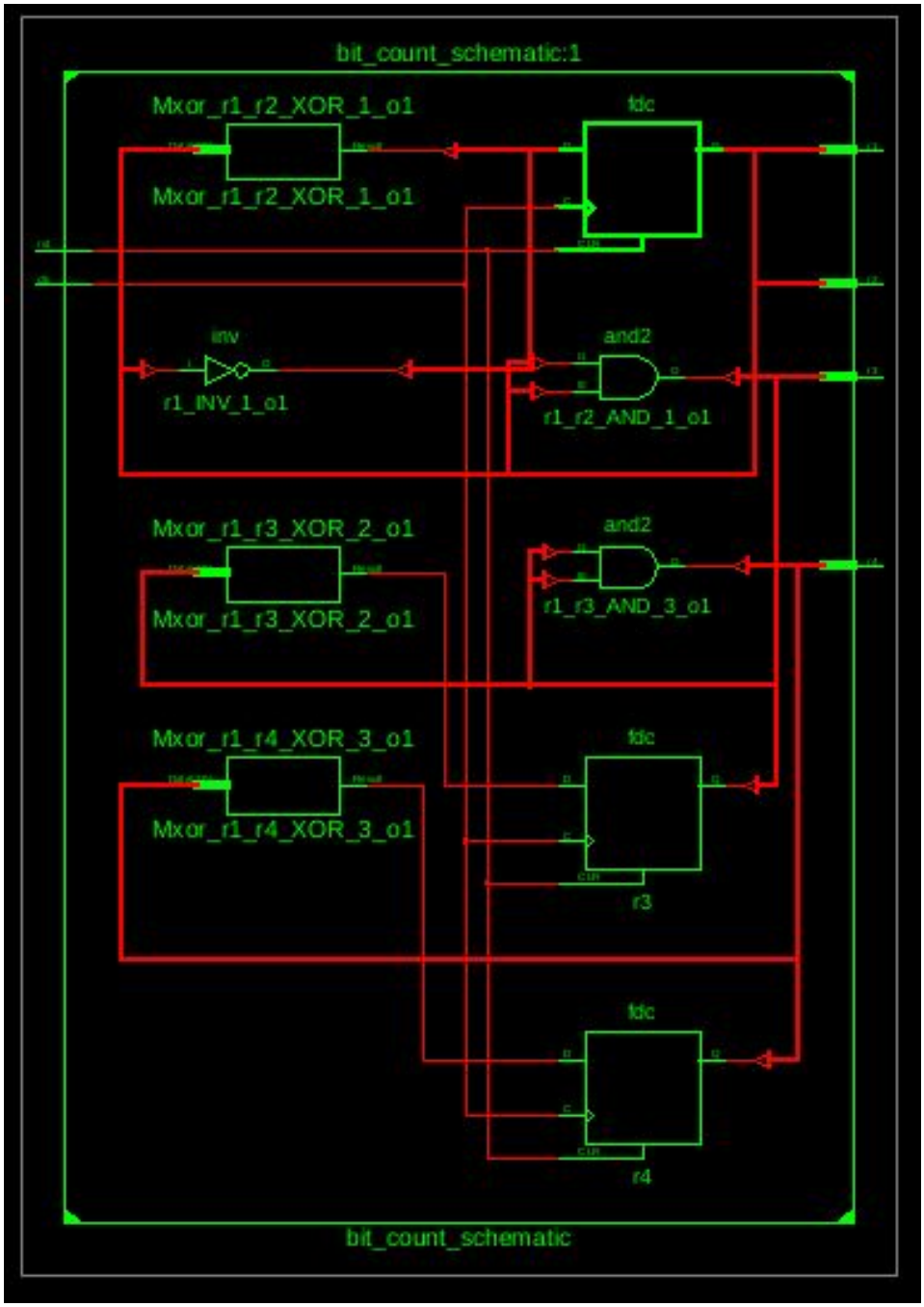
2.2 Sequential Circuit Bit Counter

For the second question, our task was to implement a 4 bit counter through the translation of a sequential circuit using logic gates and D-Flip Flops. The diagram of the circuit is given below.



Here, each of the D-Flip Flops store the value of a bit in the four bit counter, which is then displayed through the output. The flip flops hold the current state of the counter until the next clock cycle toggles, given by the center wire that connects to all the flip

flops. Therefore, I translated this circuit to Verilog code by creating 4 registers to model the 4 flip flops. Registers r1, r2, r3, and r4 correspond to the flip flops on the circuit from top to bottom (least significant to most significant). I updated the registers at every clock cycle or every reset, where the value of the registers depended on the combinatorial logic given in the diagram and the previous state of the register. The Xilinx RTL schematic of my module is shown below.



Here you can see three D-Flip Flops that Xilinx interprets from my module. The fourth flip flop, corresponding to the least significant bit of the counter, does not have a flip flop because the bit always toggles. Therefore, Xilinx optimized the register away entirely.

2.3 Modern Bit Counter

For the third question, our task was to create another 4 bit counter, but this time without translating a sequential circuit. This “modern” bit counter should be created with solely verilog logic, following the format given to us in the project specification below.

```
// Example Verilog code for the counter

reg [3:0] a;

always @ (posedge clk)

  if (rst)

    a <= 4'b0000;

  else

    a <= a + 1'b1;
```

My implementation followed the instructions given, and my module took this form and is given below.

```
//////////////////////////////////////////////
module counter(
    input clk,    //clock input
    input rst,    //reset
    output reg [3:0] out  //current count
    );

//run on clock posedge or clock reset
always @ (posedge clk or posedge rst)
begin
   if (rst)    //if reset
      out <= 4'b0000;  //set counter to 0
   else    //else increment counter by 1 bit
      out <= out + 1'b1;
end

endmodule
```

2.4 10kHz Clock Divider

For our last question, our task was to use our new skill in creating 4 bit counters to help us create a 10kHz clock divider. A clock divider is a module that takes in an input clock at a certain frequency, and divides the clock into an output clock of a slower frequency. In our case, our module needed to divide a 10kHz clock into a 1kHz clock.

In order to do this, I created a module with an input clock and an output clock. In my module, I use a 4 bit counter to count the number of clock cycles in the 10kHz clock. Once this counter reaches 5, the output clock toggles. Once the counter reaches the 11th cycle, the output clock toggles again and the counter is reset to 0. In this way, the output clock will have 1 complete frequency for every 10 clock cycles of the 10kHz clock.

3. Simulation Documentation

3.1 Combinatorial Circuit

Running my combinatorial module through simulation validated my implementation. To test my module, I created a testbench verilog program to initialize SW0 and SW1 to both be constant at 1, and then used a while loop to increment the SW2 MUX selector value. The output waveform associated with this testbench is shown below:

In the waveform linked to SW2, the selector bits, we can see the value increment from 0 to 7 and back to 0. Since both SW0 and SW1 were held constant throughout the simulation, the associated LED output value corresponds to the logical operation placed on two inputs of value 1. When synthesizing the module, the report presented the following statistics.

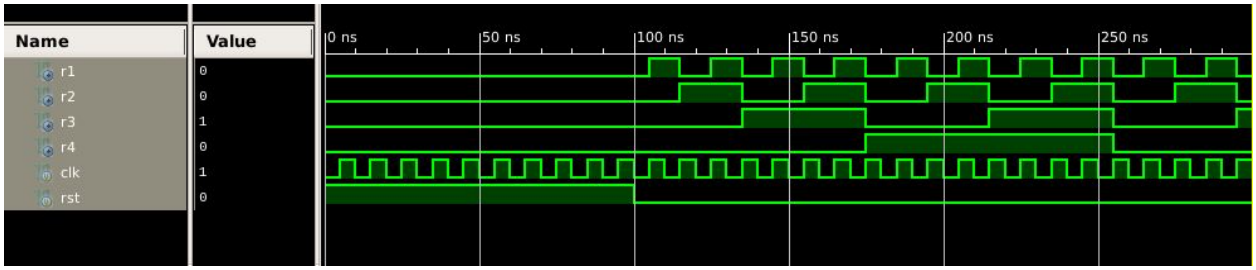
==========
=

```
HDL Synthesis Report

Macro Statistics
# Multiplexers                                          : 1
 1-bit 8-to-1 multiplexer                               : 1
# Xors                                                  : 2
 1-bit xor2                                             : 2
=========================================================
=
```

Here, we can see that Xilinx uses the 8:1 MUX hardware gate and XOR gates to implement my design, validating my module as a successful 8:1 MUX implementation.

3.2 Sequential Circuit Bit Counter

Running my module through simulation validated my implementation. To test my module, I created a testbench verilog program to have a continuous clk_in that toggles in order to drive my sequential circuit implementation. After initialization I reset the counter, and let it count away. The output waveform for the simulation is given below.



Here, we can see the clock cycling, and once the counter reset is toggled, the counter begins to increase. While there is no variable that outputs the count numerically, that would not be a correct translation of the sequential circuit given above. This waveform represents which registers hold 1 and 0 at any given time, and if you hover sequentially from left to right, the registers count up from 0 to 15 in binary as they should. When synthesizing the module, the report presented the following statistics.

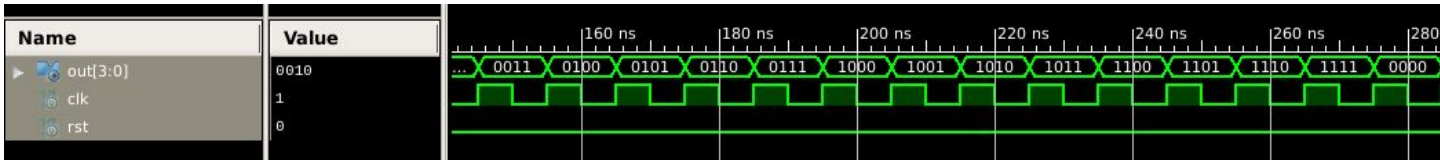
```
=========================================================
=
HDL Synthesis Report
```

```
Macro Statistics
# Registers                                          : 4
 1-bit register                                      : 4
# Xors                                               : 3
 1-bit xor2                                          : 3
=========================================================
=
```

Here, we can see that Xilinx uses 4 hardware registers and XOR gates to implement my design, validating my module as a successful translation of the sequential circuit given in the project specification.

3.3 Modern Bit Counter

Running my module through simulation validated my implementation. To test my module, I again created a testbench verilog program to have a continuous clk_in which ensured my counter would continue to increment. After initialization, I reset the counter and let my module run. The output waveform associated with the simulation is given below.



Here, we can see the clock cycling which increments my 4 bit counter at every positive edge. The counter continues until incrementing past 1111, and resets back to 0. When synthesizing the module, the report presented the following statistics.

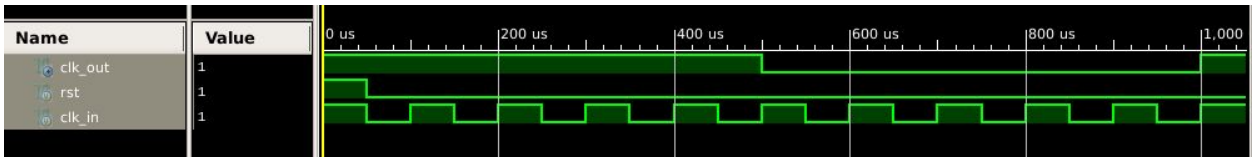
```
=========================================================
=
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                 : 1
 4-bit adder                                         : 1
# Registers                                          : 1
 4-bit register                                      : 1
```

```
=========================================================
=
```

Here, we can see that Xilinx uses 1 hardware adder and 1 four-bit register to implement my design, validating my module as a successful simplified version of the previous sequential four bit counter using D-flip flops.

3.4 10kHz Clock Divider

Running my module through simulation validated my implementation. To test my module, I again created a testbench verilog program to have a continuous clk_in that matched a 10kHz clock. In order to do this, I made the clk_in cycle 10 times every 1ms. This meant my input clock had to toggle every 50 microseconds. The waveform associated with my module simulation is shown below.



Here we can see the 10kHz clock on the bottom row, cycling 10 times every 1ms. On the top, we see the output clock of my module, cycling once every 1ms. This waveform validates my module as a correct clock divider because it takes in the 10kHz clock, and uses a counter to divide it into a 1kHz output clock. When synthesizing the module, the report presented the following statistics.

```
=========================================================
=
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 1
 4-bit adder                                           : 1
# Registers                                            : 1
 4-bit register                                        : 1
# Comparators                                          : 1
 4-bit comparator lessequal                            : 1
=========================================================
=
```

4. Conclusion

A .ucf file is a user constraints file, and if we took our code and implemented it onto a Nexys3 board, this file would be used as an interface between the Xilinx ports and the pins on the machine. In conclusion, the lab was rather straightforward as the problem solving goes. Some difficulty that I encountered was first becoming familiar with the Xilinx process, as well as debugging in Verilog. I also encountered issues with understanding the spec at times. For the clock divider problem, the instructions were vague and I was not sure whether I was implementing a 4kHz clock divider or a 10kHz clock divider in my implementation and testbench.