Rodrigo Becerril Ferreyra
CECS 361 Section 01
Lab 5
29 November 2020

1 Introduction

The purpose of this laboratory assignment was to practice knowledge of faults and fault checking. Specifically, given a faulty module (with a stuck-at-one fault) and the diagram of the circuit it was supposed to implement, the task was to find where the fault was located via two methods: the first is the activation–propagation method, and the second is an equivalence method that was implemented on the Nexys A7 100T FPGA board.

The activation–propagation method consists of finding all the possible places that a stuck-at-one fault could be, either by using gate-oriented fault collapsing or line-oriented fault collapsing. The next step is to activate and propagate the faults. Lastly, if the line being tested is stuck at one, then the activation of the fault will not cause the desired result. The equivalence method, on the other hand, is more simple: the output produced by the faulty circuit using each set of inputs is tested against the expected output from a flawless circuit; the faulty circuit's functionality is tested for equivalence with a circuit that is known to work. This method was implemented on the board: both circuit's inputs are driven by on-board switches, and if any output from both circuits doesn't match, a light flashes with the following color order: red, green, blue.

For reference, the Figure 1 is the desired (non-faulty) circuit that was implemented in this lab.

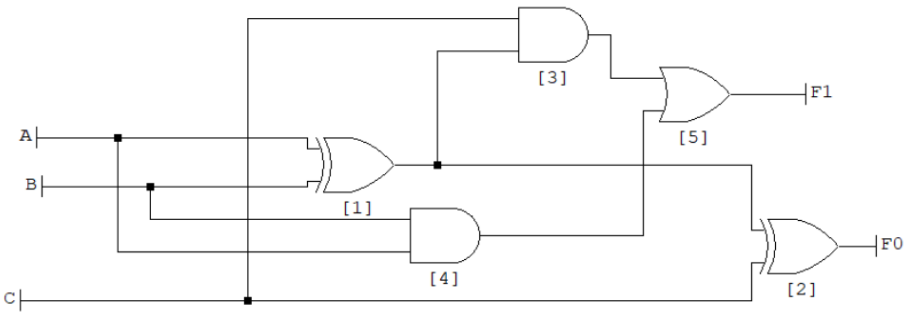


Figure 1: Desired, non-faulty circuit.

2 Activation–Propagation Method

As previously stated, this method involves finding all the places where a stuck-at-one fault can be, and testing them individually. Figure 2 is a diagram of all possible stuck-at-X faults, obtained using the line-oriented fault collapsing method. All of the different places where a stuck-at-one fault can be found are labeled. Note that the two stuck-at-zero faults are not labeled, because we knew prior to testing that the fault was a stuck-at-one fault.

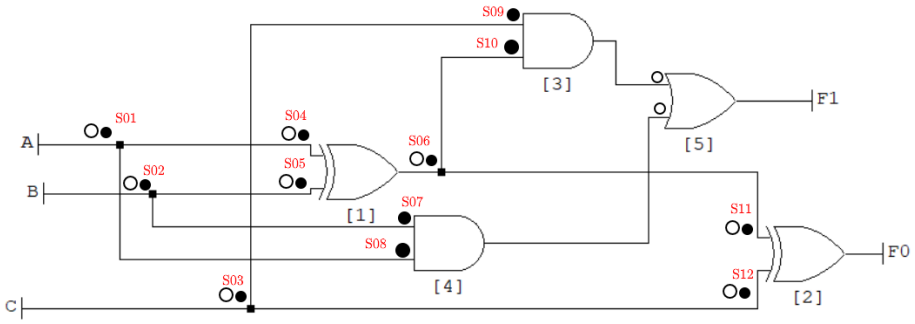


Figure 2: Possible stuck-at-X locations.

Activating the fault means giving the line it's on the opposite value; to activate a stuck-at-one fault, the line being tested for a fault must be set to zero using the correct combination of inputs. Propagating the fault consists of choosing the correct combination of inputs such that no gate on the fault's path has a controlling value; in practice, this means making sure that no AND gate has a zero and no OR gate has a one on its input (an XOR gate does not have a controlling value). To make it easier to see on a waveform, I also flipped the value of the line being tested; if the line is truly "stuck," then there will be no change in the output.

Figure 3 below shows the result of all twelve tests, while Table 1 shows the values used to test each case.

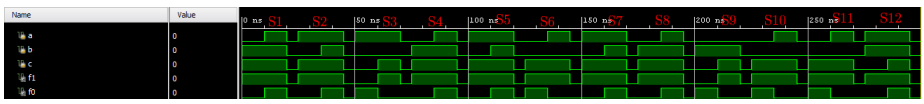


Figure 3: Activation–propagation test results.

Fault	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
{a, b, c}	011	101	100	011	101	001	101	011	010	001	001	101

Table 1: Test cases to activate and propogate a stuck-at-one fault.

It may not be obvious from the waveform, but this test shows that there is a stuck-at-one fault at S06, S10, and S11. Flipping S06 was expected to change both outputs, but F1 did not change. The same happened while testing S10 and S11, which is to be expected, as these lines are connected to each other.

Since S06 does not connect to anything other than S10 and S11, the possible combinations of faults are as follows: fault at S06 only, fault at S10 and S11 only, or fault at S06, S10, and S11. These three cases are indistinguishable.

3 Equivalence Method

This method consists of testing the faulty outputs with the expected ones. To do this, I created the expected functionality of the circuit in Figure 1, structurally, in Verilog. The resulting circuit diagram can be seen in Figure 4.

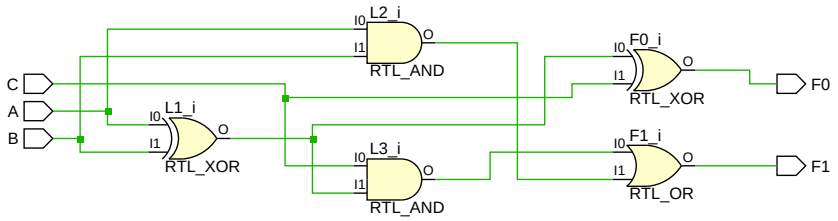


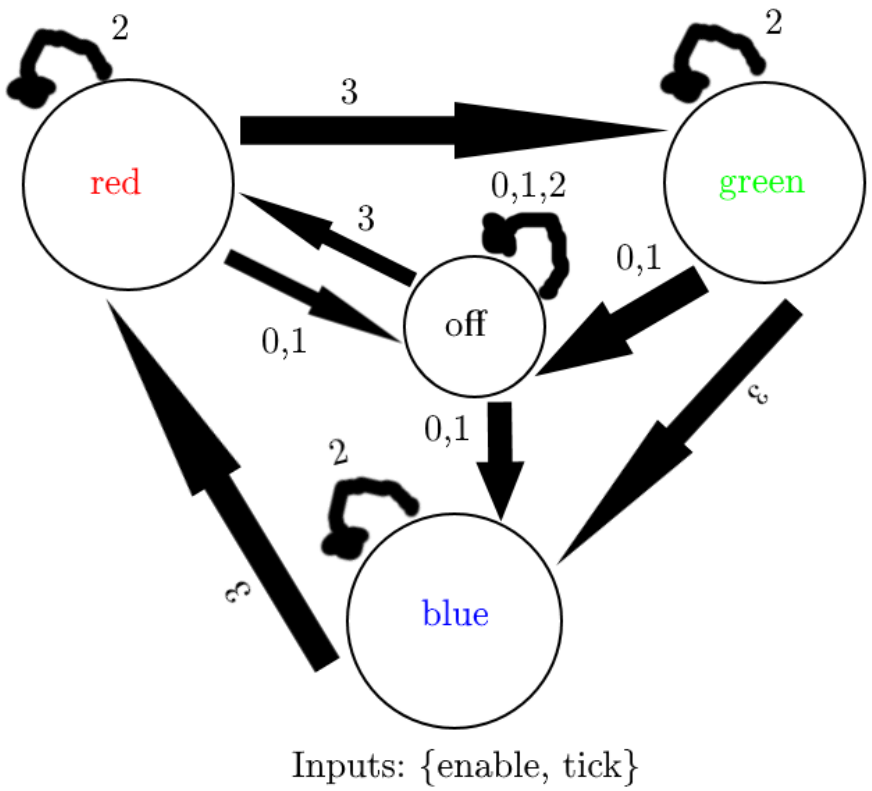
Figure 4: Verilog schematic diagram of expected (non-faulty) circuit.

The respective `F0` and `F1` outputs of both (faulty and non-faulty) modules

were then XOR'd together, and the OR of these XOR outputs activated the RGB LED flash mechanism.

3.1 RGB LED Flash Mechanism

The flashing was implemented using a Moore Finite State Machine model. It takes two inputs and outputs three signals: its inputs are `enable` and `tick`, and its outputs are `red_on`, `green_on`, and `blue_on`. The `enable` input is driven by the XOR/OR described above, and the `tick` input is driven by a 0.5 s, one-clock-wide pulse. The outputs are meant to control the red, green, and blue LEDs on the board, respectively. The state transition diagram, table, and outputs are shown in Figure 5 below.



Input = {enable, tick}	Next state			
state	0	1	2	3
off	off	off	off	red
red	off	off	red	green
green	off	off	green	blue
blue	off	off	blue	red

State	Outputs = {red, green, blue}
off	000
red	100
green	010
blue	001

Figure 5: FSM state transition diagram, table, and outputs (Moore).

3.2 Truth Table

Of course, this task can also be achieved by using a truth table, and comparing the results of the expected outputs to the faulty outputs.

	Actual		Expected	
{a, b, c}	F1	F0	F1	F0
000	0	0	0	0
001	**1**	1	**0**	1
010	0	1	0	1
011	1	0	1	0
100	0	1	0	1
101	1	0	1	0
110	1	0	1	0
111	1	1	1	1

Table 2: Expected and faulty outputs

It is evident that in the input combination {`a, b, c} = 3'b001`, the output value of F1 differs between the actual (faulty) circuit and the expected circuit. This is also where the hardware-implemented RGB LED flashes.

3.3 Top-Level Diagram

Figure 6 displays the schematic diagram of the top-level module, which is implemented on the board (feel free to zoom in).

From this diagram, it is clear how both circuits share the same inputs, and how the differences in corresponding outputs from both circuits are detected. A two-input XOR gate's output is high whenever its inputs are different, so if there is any difference between either set of inputs (or both), the OR gate will go high and activate the RGB flash mechanism.

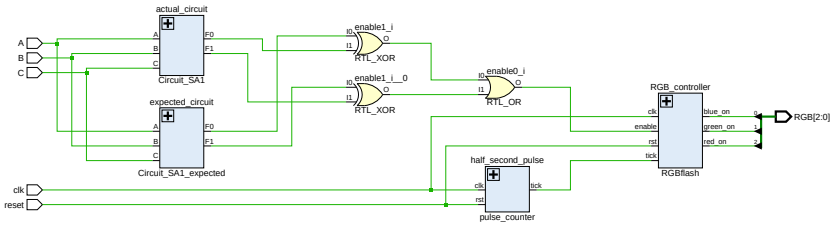


Figure 6: Top-level board implementation.

4 Waveforms and Testing

The truth table in Table 2 was achieved by testing the given circuit with all $2^3 = 8$ permutations of inputs. In addition, the expected circuit shown in Figure 4 is also displayed for comparison. Figure 7 shows the combinations of inputs and their respective outputs for both circuits.

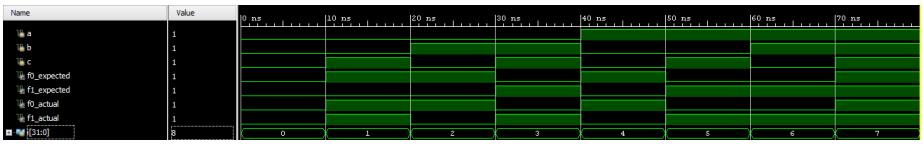


Figure 7: Waveform for given (faulty) circuit.

The differences between these two pairs of inputs can be seen clearly when $i = 1$ (or equivalently when `{a, b, c} = 3'b001`).

Figure 8 shows the board implementation when either of the outputs from both circuits don't match, and when all corresponding outputs match.

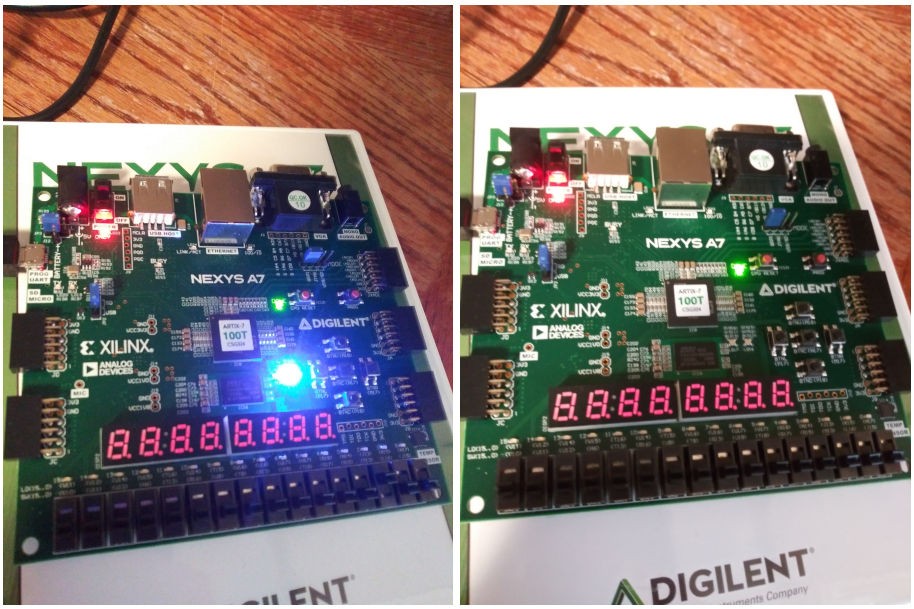


Figure 8: Mismatched and matching outputs on the board.