Stony Brook University

Project 2: Fault Simulation

Issued: 2/23/16, Due: 3/11/16 5:00PM

1. Introduction

The goal of Project 2 is to extend your logic simulator (Project 1) to perform single-stuck-at ***fault simulation***. You will do this by building new functionality on top of your Project 1 solution (or my reference solution). Additionally, you will be using new versions of the Circuit and Gate classes, which have been updated in a few key ways to make this project possible.

As before, you will use the computers in the Graduate CAD Lab (Light Engineering room 183). You can either work in the lab, or connect to these machines from home using SSH. (Please see the Lab Computer Overview for information on this.) Also as in Project 1, you may work on your own computer, but I will be evaluating it using the CAD Lab Linux computers, so it must function properly there.

You will be working alone on this project. Please do not discuss details with each other or look at each other's code.

We will again be using Piazza for discussion related to this project. I have added a "proj2" folder for us to use.

2. Getting Started

I am providing you with the following things:

- This project description
- Baseline software. This software will take care of all file input/output, and will setup all the data structures for you. You can find this on the lab computers at `/home/home4/pmilder/ese549/proj2`
- API documentation for the updated Circuit and Gate classes. You can view this documentation as a PDF (apidoc_v2.pdf), or as a nice interactive webpage (which you can download as apidochtml_v2.zip, unzip, and then browse). Both are available on Blackboard. (This documentation is automatically generated from the source using a tool called Doxygen.)

Please remember to work within the private work directory you created last time. Please see the Project 1 handout for directions for setting this up and copying our starting code base there.

3. Overview

Your system will take as input:

- a description of a circuit at the Boolean gate level (in `.bench` format, as in Proj. 1)
- a text file with a list of desired faults to simulate (each giving a fault location, and a fault type, stuck-at 0, or stuck-at 1)
- a text file with a list of desired inputs to simulate (same as in Project 1, but with no D or D' inputs, just 0, 1, and X)

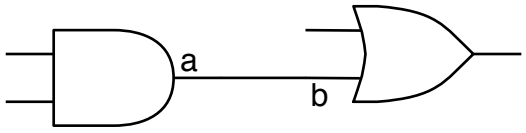
The system will loop over all faults in the fault file; for each fault, it will simulate all of the input vectors. (That is, if you give 15 faults to simulate and 10 vectors, it will run a total of 150 simulations.) As output it will produce a text file giving the circuit's outputs for each simulation.

As in Project 1, you are given code to parse in the circuit, setup the data structures, and read in the desired inputs. I have added this code to read the fault file. Then, you will write the code to compute all gate output values (starting from your project 1 solution) taking into account the newly added faults.

4. Faults and Fanout

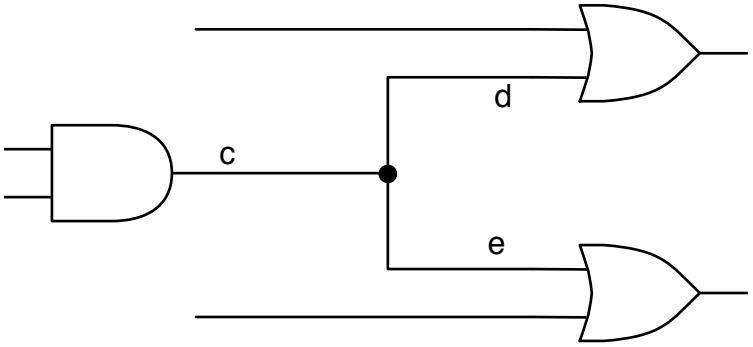
For convenience, in this project we will assume that stuck-at faults will occur only at the *output* of gates (including PI gates).

In fanout-free circuits, this is not a problem. For example:



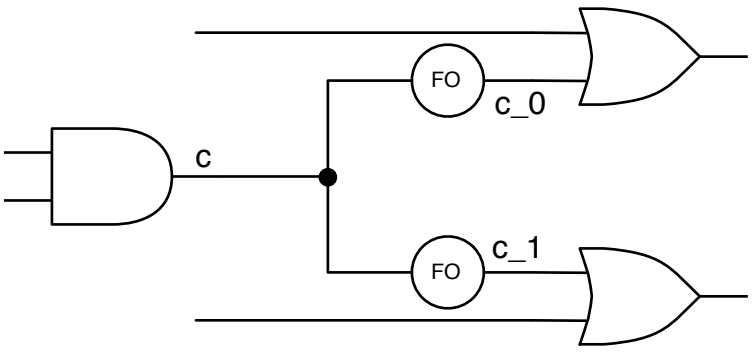
Any fault on a is equivalent to the corresponding fault on b. So, in examples like this, there is no need to worry about faults on gate inputs.

However, fanout branches present a difficulty here, because faults on the stem and the branches all behave differently. For example, in this circuit:



a fault on c is not equivalent to a fault on d or e. However, our circuit file format doesn't give us a good way to distinguish between these; only gate outputs are given names in our file format. So, we have a problem: how can we simulate a fault on a fanout branch such as d or e?

We will address this problem by first adding a new "gate" type to represent fanout branches. The new gate type (called GATE_FANOUT) essentially works as a buffer: it takes in one input and produces the same value as output, with no logical change. The key is that by inserting these gates on every fanout branch, it allows us to specify faults separately on the stems and branches. For example, we would modify the circuit above to this form:



Now, the fanout gates (FO) are added to every fanout branch, and their outputs are automatically named c_0, c_1, etc, representing braches of c. Now, we can represent the faults on the fanout branches by placing faults on the output of the fanout "gates."

Manually modifying the .bench files to do this would be very time consuming and annoying. ***Instead, I have modified the circuit setup code to do this automatically for you.*** Whenever there is any fanout branch from the output of a gate, FO gates are inserted for each stem, and names are automatically given as above. (That is, the stems of a branch named `a` get names `a_0`, `a_1`, and so on.)

In order to adapt your Project 1 logic simulation code to work with these gates, you will need to add a small amount of code. If your simulator encounters a gate of type GATE_FANOUT, it should simply treat it like a buffer (set its output equal to its input).

5. Simulation and File Formats

Like before, you can compile your code by typing `make`. This builds the executable called `faultsim`.

You can then run the tool with the following format:

```
./faultsim [bench_file] [output_loc] [fault_file] [input_vectors]
```

`bench_file`: the target circuit in .bench format

`output_loc`: location for output file

`fault_file`: faults to be simulated

`input_vectors`: list of input vectors to simulate

The system will simulate each vector in `input_vectors` for each `fault_file`. For fault free simulation, enter -1 for the fault location and fault type in the `fault_file`. The simulation output will be stored in `output_loc`.

File Formats:

- The circuit is described in Bench format, as in Project 1. (Please refer to previous documentation if needed.)

- The fault file is a list of faults we would like to simulate or generate tests for. The format of this file is simple: one line gives the name of the signal where we would like the fault, and the next line gives a 0 or 1 indicating a stuck-at-0 or stuck-at-1 fault. For example:

```
A
0
B_1
1
```

  specifies two faults to simulate or generate tests for: A stuck at 0, and B_1 stuck at 1. Note that as in the second example we may target a fanout branch.

- The input vectors are provided in the same format as Project 1. Each line contains *k* characters, where *k* is the number of circuit PIs. The characters may be 0, 1, or X.

- The system's output file will be in the following format: each line of the output file will contain a number of characters, representing the POs of the circuit (similar to Project 1). However, there is one very important difference. In Project 1, your system simulated each given input vector and printed the result. Here, you are simulating each input vector for *each fault specified in the fault list*. So, if you ask for 150 vectors and 100 faults, your output file will have 15,000 lines of outputs. The system will simulate all vectors for each fault before going on to the next fault. The file will print "--" on a line to between each fault.

  For example, here we have a circuit with two POs. We are simulating it with three test vectors (t0, t1, t2), and we are simulating it for two faults (f0 and f1). Our output will look like:

```
--
XX
11
D1
--
XX
11
11
```

  First, the first three lines of outputs (XX, 11, D1) correspond to the circuit's outputs for each of the three test vectors when the first fault specified in the fault list is active. (That is, when f0 is active, then t0 produces output XX, t1 produces output 11, and t2 produces output D1). Then, the next three data lines show the circuit's outputs for the same three tests when the second fault f1 is active.

6. Fault Simulation

In order to add fault simulation capabilities you will start from project 1 code (either yours, or my reference solution if you prefer), which does fault-free logic simulation on five-valued logic (0, 1, X, D, D'/B). Then, you will extend it to work with stuck-at faults (i.e., you will check if a gate has a fault and set its value accordingly).

To start, first look at `ClassGate`. Note specifically the new `set_faultType()` and `get_faultType()` functions. Each Gate now can have a stuck-at fault on its output.

Now, read through the code in the `main()` function. In this code, the system will read a fault out of the given fault file, which specifies a fault location (i.e. the gate whose output is faulty) and the fault type (stuck-at 0 or 1). The code then places it on appropriate gate output (by calling `set_faultType(faultType))`. Then, the code sets up the PI input values (as in Project 1). After that, you should run your simulation code to simulate the values on all other gates.

What is new is that whenever you need to set a gate's output value, you also need to check if it is faulty. For example, if you are going to set the value of a gate's output to be LOGIC_ONE, but that gate has a stuck-at-0 fault on the output, then instead you would set its output value to be LOGIC_D. You can use the get_faultType() function to check if a Gate has a fault on its output.

7. What's New in the API Code?

The following changes have been made to the API since Project 1:

- A `Gate` can now have a stuck-at fault on its output. See the functions `Gate::set_faultType()` and `Gate::get_faultType()`.

- The `Circuit` class only has one major change: when the parser initializes the circuit, it will add "fanout gates" on every fanout branch (see section 4).

Please do not change or edit the ClassGate and ClassCircuit files at all. If you think anything useful is missing from these classes, please let me know and potentially we can add it for everyone.

8. Main function

The main() function in main.cc contains all of the code for input/output and setting up the data structures. Please look at this function carefully. Just like before, you will see a blank space in the middle labeled "Your code here" for you to write your function. Write your code in this place. If you choose to use any extra functions (which is probably a good idea), write the prototypes in the section labeled at the top of the file, and put the functions themselves in the section at the bottom of the file. ***Please keep your code in the requested regions, because it will make it much easier for me to read and understand what you did.***

9. Running and testing your code

We are using a Makefile to make it very easy to compile your code. To compile, simply type:

```
make
```

This will produce an executable called faultsim. This executable takes in four parameters at the command line, as specified above in Section 5. For example, you can run:

```
./faultsim tests/c17.bench myc17.out tests/c17.faults tests/c17.refinput
```

This will simulate the circuit in tests/c17.bench with the faults in tests/c17.faults and theinputs in tests/c17.refinput. It will store the output values in myc17.out. I have provided a few reference circuits with test inputs and correct outputs (.refout files).

You can check that your output file (e.g., myc17.out) matches the reference output by typing:

```
diff myc17.out tests/c17.refout
```

This command will show you any differences between the two files. (If it reports nothing, the files are identical.)

The tests/ subdirectory contains a few circuits to work with, but you can easily construct your own in the proper format (this is especially important when getting started).

10. Report

Write a short (~1 page) report describing:

- How your solution works.
- Any problems you had and how you fixed them.
- How you tested your solution.

You may submit this report as a hard copy in class, or as a PDF on Blackboard (see below). Please do not submit other file formats—if you are not able to make a PDF, then print it out and submit in class.

11. Code and Report Submission

Following the directions above, your code should be entirely contained in the main.cc file. ***Please submit only this file***—do not include the other source files from the project, since you did not modify them. Please make sure you include a comment at the top of the file with your name.

You can submit your code (and optionally the PDF of your report) on Blackboard. On Blackboard go to Assignments → Project 2.