ECE745 – Assignment 4

Issued on July 8th and due before July 30th at 11:59 PM

The objective is to read a netlist specified in Verilog and to develop a single stuck-at fault simulator and an automatic test pattern generator (ATPG) for full scan circuits.

An example of a circuit netlist in an input file `example.v` is given below:

```
module example ( clock, G0, G1, G2, G3, G17  );

  input  clock, G0, G1, G2, G3;
  output G17;

  reg G5, G6, G7;
  wire n17, n18, n19, n20, n23, n34, n37, n40;

  assign G17 = ~n37;
  assign n17 = ~G0 & G6;
  assign n18 = n23 | n17;
  assign n19 = G3 | n17;
  assign n20 = ~n19 | ~n18;
  assign n34 = G0 & ~n37;
  assign n37 = ~G5 & ~n20;
  assign n23 = ~G1 & ~G7;
  assign n40 = ~G2 & ~n23;

  always @ (posedge clock) begin
    G5 <= n34;
    G6 <= n37;
    G7 <= n40;
  end
endmodule
```

It is fair to assume that after the module's port list, you will have line for declaring all the inputs (using the **`input`** keyword), then a line for declaring all the outputs (using **`output`**), then a line for declaring all the registers (using **`reg`**) and then a line for declaring all the combinational nets (using **`wire`**). Note, in between multiple identifiers, separated by comma, you can have whitespaces that can include a newline. It is worth mentioning that for this ECE745 assignment we will use the 2001 version of the Verilog standard, where the registers can be declared using the **`reg`** keyword and the nets using the **`wire`** keyword. Note also, by default both inputs and outputs are automatically implied to be nets. If, by any chance, an output is registered then it must appear both in the **`reg`** and **`output`** lists. The empty lines between the **`module`** declaration and the **`input`** list are optional, as they are in any other cases in the above example, like between the **`output`** and **`reg`** lists. The names of the signals must be unique, and can be any string with alphanumeric characters plus underscore (except Verilog keywords), however it is acceptable to assume that they will have less than 64 characters.

The combinational netlist consists of a sequence of **`assign`** statements where each **`assign`** statement drives a single bit net that implements a basic logic operation, like `AND` or `OR` between other nets and registers. In addition, a single input is accepted on the right-hand side of an assignment, in which case you can implement either an inverter (like the first line in the above example) or a buffer, e.g., the inversion from the first line can be removed "**`assign`** `G17 = n37;`". For previously optimized netlists, there might be a constant on the right-hand side of an assignment (e.g., `1'b1`). Note, some, but not all, of the inputs of the basic logic gates (`AND` or `OR`) can be inverted. You

can assume that the maximum fan-in of any gate is 64. There is no limit on the fan-out. For the sake of simplifying the parsing of the Verilog input file, it is assumed that the **`always`** block, which is used to describe the transfer of the next state signals into the present state registers, will appear only after *all* the **`assign`** statements have described the combinational logic. Also, a single **`always`** block will be used to describe *all* the single-bit register transfers. A few more input files will be posted for your reference.

Your program should run as follows:

`fault_sim_atpg` `example.v random_vectors abort_time_per_fault`

where: `example.v` is in the input Verilog file (this command line argument must be specified)

`random_vectors` is the number of random vectors that should be fault simulated before ATPG starts to find the input pattern for the hard-to-detect faults (this argument is 0 if not specified, which implies that ATPG starts immediately)

`abort_time_per_fault` is the amount of time in **milliseconds** that your ATPG engine should spend on searching for a test pattern for a fault before aborting the search (this argument is also 0 if not specified, which means that your program should spend an indefinite amount of search time for finding pattern for each *target* fault)

Your program should generate 5 output files in simple text format. Assuming the input Verilog file is called `example.v`, the output files should be called as follows:

`example.faults` is the list of *target* faults that are used by fault simulation and ATPG; the first entry in each line should be a fault like `n23/0`, which means net `n23` stuck-at 0; if the fault is equivalent to other faults that should be discarded, the following entries in the same line should list the equivalent faults; in this case you must list `G1/1` and `G7/1`

`example.stimuli` gives all the input stimuli (in binary) that have been evaluated in the format primary-inputs + pseudo-inputs; for our `example`, the order of the primary-inputs (except `clock`) is `G0,G1,G2,G3` (same order as specified in the line starting with keyword **`input`**), and the order of the pseudo-inputs is `G5,G6,G7` (same order as specified in the line line starting with keyword **`reg`**); if the n^{th} line in `example.stimuli` is `1000110`, this means that the n^{th} input stimuli is `{G0,G1,G2,G3,G5,G6,G7}=7'b1000110`

`example.responses` gives all the fault-free output responses that have been obtained after zero-delay logic simulation in the format primary-outputs + pseudo-outputs; for our `example`, the order of the primary-outputs is `G17` (although there is only one output in this `example`, we follow the same order as specified in the line starting with keyword **`output`**), and the order of the pseudo-outputs, i.e., flip-flop inputs, is `n34,n37,n40` (same order as the corresponding flip-flop outputs that are specified in the line starting with keyword **`reg`**); if the n^{th} line in `example.responses` is `1100` this means that the n^{th} output response is `{G17,n34,n37,n40}=4'b1100` (as obtained when simulating the n^{th} stimuli `1000110` from `example.stimuli`)

`example.detected` gives in line *n* all the stuck-at faults that are detected by the corresponding n^{th} input stimuli; for example if the n^{th} pattern from `example.stimuli` detects faults `G0/0`, `G1/1`, `G7/1`, `G17/0`, `n23/0`, `n34/0`, `n37/1` and `n40/1`, then these faults should be listed in line *n* from `example.detected`. Note also, for the faults that have been marked as equivalent, like `G1/1`, `G7/1` and `n23/0`, it is sufficient to list only one of them.

`example.undetected` gives all the *target* faults that have not been detected; the first entry in each line should be the undetected fault, for example `a/0`, and the second entry should be the reason for not finding an input pattern for this fault (`aborted` or `untestable`)

At the end of its execution, your program should also report in the terminal, the runtime, the number of *untestable* faults and the fault coverage of the *testable* faults.

This is a complex assignment that needs to be broken into several steps, each of them relying on some simplifying assumptions that make the implementation feasible within the given timeframe. Below is the sequence of steps that is recommended that you go through (further details will be provided in class, as needed).

i) Read the netlist and represent it as a directed acyclic graph (DAG). To ensure that the netlist is correctly represented in your internal data structures, you should perform zero-delay logic simulation using a set of random values and confirm that the results match the ones that you can obtain through Modelsim. You should also create a full-scan model of your circuit, where all the flip-flop outputs/inputs are considered as pseudo-inputs/outputs to the combinational circuit block. This significantly simplifies both the fault simulation and ATPG. The full scan circuit can also be simulated, and the by-products of this simulation are the `example.stimuli` and `example.responses` files.

ii) By default, the total number of stuck-at faults in a netlist is equal to 2 `x` (#**`wires`** + #**`inputs`** + #**`outputs`** + #**`regs`**) - with a few simplifying assumptions. The purpose of this step is to reduce the fault space by performing fault collapsing based only on fault equivalence of the basic logic gates. The non-collapsed faults are labeled as *target* faults and they should be listed as the first entry in each line in the `example.faults` file. The *target* faults, together with the equivalent faults (listed as the additional entries in each line from `example.faults`) will form the entire stuck-at fault space. Note also, although we treat the faults of the stem and branches of a fan-out point as identical, we are not permitted to collapse the outputs of the gates from a fan-out point to the faults from the stem.

iii) Fault simulation should be performed for a total of `random_vectors` input patterns. Subsequently, each time when a new input pattern is generated by the ATPG engine for an unprocessed *target* fault, you should perform fault simulation for this input pattern as well. As a golden model, you should first implement a "slow" serial fault simulator, which should be enabled only through a compile-time directive (`#define … #ifdef … #endif`) for debugging purposes. Subsequently, you should implement a more efficient deductive fault simulator in order to fault simulate at least ten patterns per second.

iv) After the fault simulation of `random_vectors`, the ATPG engine inspects how many *target* faults have not been detected so far by fault simulation. Subsequently, one or multiple unprocessed faults will be targeted at-a-time by the ATPG engine (to be discussed in class in more detail), in order to generate an input pattern that will sensitize the fault site and propagate its effect to an observable output. If no input pattern can be generated when a single fault is targeted this is an indication of a redundancy in the circuit and the fault is `untestable`, and recorded in the `example.undetected` file. If no input pattern can be generated within `abort_time_per_fault` amount of time, the fault is `aborted`, and recorded as such in the `example.undetected` file. If an input pattern was successfully generated, this pattern will be fault simulated and *all* the faults detected by it will be recorded in `example.detected`. Subsequently, ATPG will continue until all the faults have been processed.

As a final point, it is worth clarifying that it is strongly recommended to use a Boolean satisfiability solver (SAT solver) as the ATPG engine. Nonetheless, one should not underestimate the magnitude of the task at hand, which, considering the more advanced stage in the course, will pose both implementation and intellectual challenges (especially when pattern count reduction matters … as it is, naturally, expected).