FAN ATPG (project)

`Brahad Kokad  - 2201112004`
`Akshit Gureja - 2020112004`

Objective

The objective of this project is to implement FAN ATPG algorithm for combinational circuits. This includes:

- Reading a circuit's hardware description from a verilog file, parsing it to c++ to describe the circuit using arrays and vectors for our data structures.
- Running FAN ATPG algorithm. This includes:
  - Activation and propagation of a given fault.
  - Backtracing to assign values wherever necessary.
  - Backtracking in case of a conflict (overspecified values).
  - Justifying headlines in the end.
  - Asserting if a fault is testable or not, providing a test pattern in case it is.
- Using the test pattern generated above to run a verilog testbench on the circuit.

Implementation specifics

Parsing verilog to cpp

The benchmark circuits provided follow a standardized method of writing verilog files and this structured approach has been exploited while writing the parser from scratch for this project.

The parser code opens the verilog file and read every line sequentially. For every line that it reads from the verilog file, the string of inputs, gets divided into words. Different actions are performed by the code depending on what the first word of the line is.

If the first word happens to be “input”, then the wires following are all assigned as headlines by default. If the word is output or wires, then all the wires following it will be marked as ‘free’ as default.

If the first token matches with none of the keywords, then it represents a gate in the netlist, which is then processed into an n-length array which is stored in the code for the FAN to be implemented upon later. Then the wires are marked if they are head line, bound line, or stem branches or free lines accordingly.

FAN algorithm

Fault activation

To introduce a fault in the CUT, we need to activate it, for this we backtrace the fault line until we reach a headline (which can be justified in the end), in this process we shall assign appropriate values to some wires wherever necessary.

Propagation of fault

To propagate the the fault, we essentially propagate its effect. We are emulating the 5-valued D algebra by specially handling propagation through all kinds of gates exhaustively. So, for propagating a fault from the input of a gate to its output, we assign the other inputs of the gate to non-controlling values (for instance 1 for an OR gate, 0 for AND gate, etc).

Backtracing

In this process of propagating a fault we end up assigning certain lines to specific values so as to make them non-controlling with respect to the gate in consideration. Thus, for each such assignment we need to justify it. If this line happens to be a headline then we leave to be justified in the end.

> 💡 headlines are ends of isolated sub-circuits (cones), thus the circuit inputs preceding a headline are guaranteed to have an input pattern that gives us the required value at that headline.

If this line is not a headline then we justify it by backtracing until we reach a headline.

This procedure of the algorithm is implemented recursively.

Backtracking

If while backtracing we encounter a conflict- i.e. a certain line being overspecified, then we backtrack to the previous assignment and change it if possible, we do this for a fixed number of times, say the ‘backtracking depth’. If the conflict isn’t resolved, we declare that the fault is not testable.

Justifying headlines

In the very end, we backtrace from the headlines to get values of all the lines.

At this point we shall successfully get values for all the inputs, and thus generating a test pattern for the given fault in the CUT.

Implementing the FAN algorithm in cpp

Back Propagate

The function `back_propagate_to_head()` is a recursive function for backtracing or back propagation in FAN ATPG for combinational circuits. The purpose is to propagate logical values '0' or '1' through the circuit's netlist based on specified conditions.

Here's an overview:

- We traverse the netlist and identifies gates (like NAND, AND, OR, NOR, XOR) connected to a particular wire.
- For each gate type, we check the required value ('0' or '1') and set the inputs accordingly to propagate this value backward through the circuit (justify the value).
- For instance, for NAND gates, we try to set the inputs to produce either '0' or '1' based on the desired output based on cases of stuck-at-0 and stuck-at-1 faults by modifying the inputs appropriately.

This function is a crucial part of a larger FAN ATPG algorithm.

Forward progagation (propagating single stuck-at fault)

The function `front_propagate()` is used to propagate a fault along a path until it is effective on a primary output.

Here's a breakdown of what the code does:

- It iterates through the netlist to find gates connected to the specified wire.
- Depending on the gate type (NAND, OR, NOR, AND, XOR), it sets the other input of the gate to the non-controlling input.

For instance:

- For NAND gates, it sets the input to '1' and computes the output based on the value of the other input.
- For OR gates, it sets the input to '0' and computes the output based on the value of the other input.

It then continues this propagation recursively for the output of the gate to further propagate the fault value through the circuit.

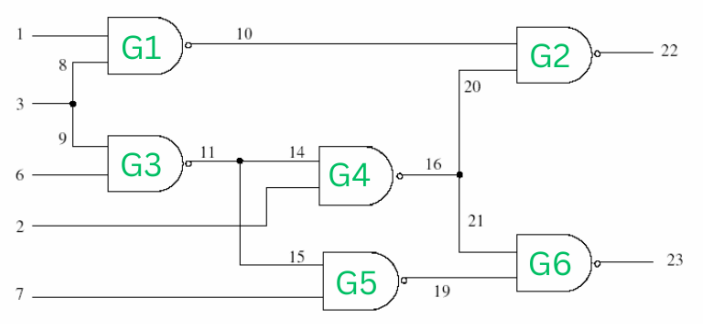
Justifying headlines in the end

```
int justify_head1(string wire){

    int n = netlist.size();
    int is_input = 0;

    for(int i = 0; i < n; i++){
        if(netlist[i][0] == wire){
            // just give input of gate, output wire value, other input value
            values_for_fault[netlist[i][0]] = wire_val(netlist[i][3], values_for_fault[netlist[i][2]], values_for_fault[netlist[i][1]]);
        }

        if(netlist[i][1] == wire){
            values_for_fault[netlist[i][1]] = wire_val(netlist[i][3], values_for_fault[netlist[i][2]], values_for_fault[netlist[i][0]]);
        }
    }
    return 0;
}
```

Here we essentially justify the headlines in the end after the fault has been propagated to a primary output (and rest of the lines get justified in the procedure anyhow). The beauty of this idea is that headlines are guaranteed to get justified as they are heads of isolated cones in the circuit that are not dependent on the rest of the circuit.

Results

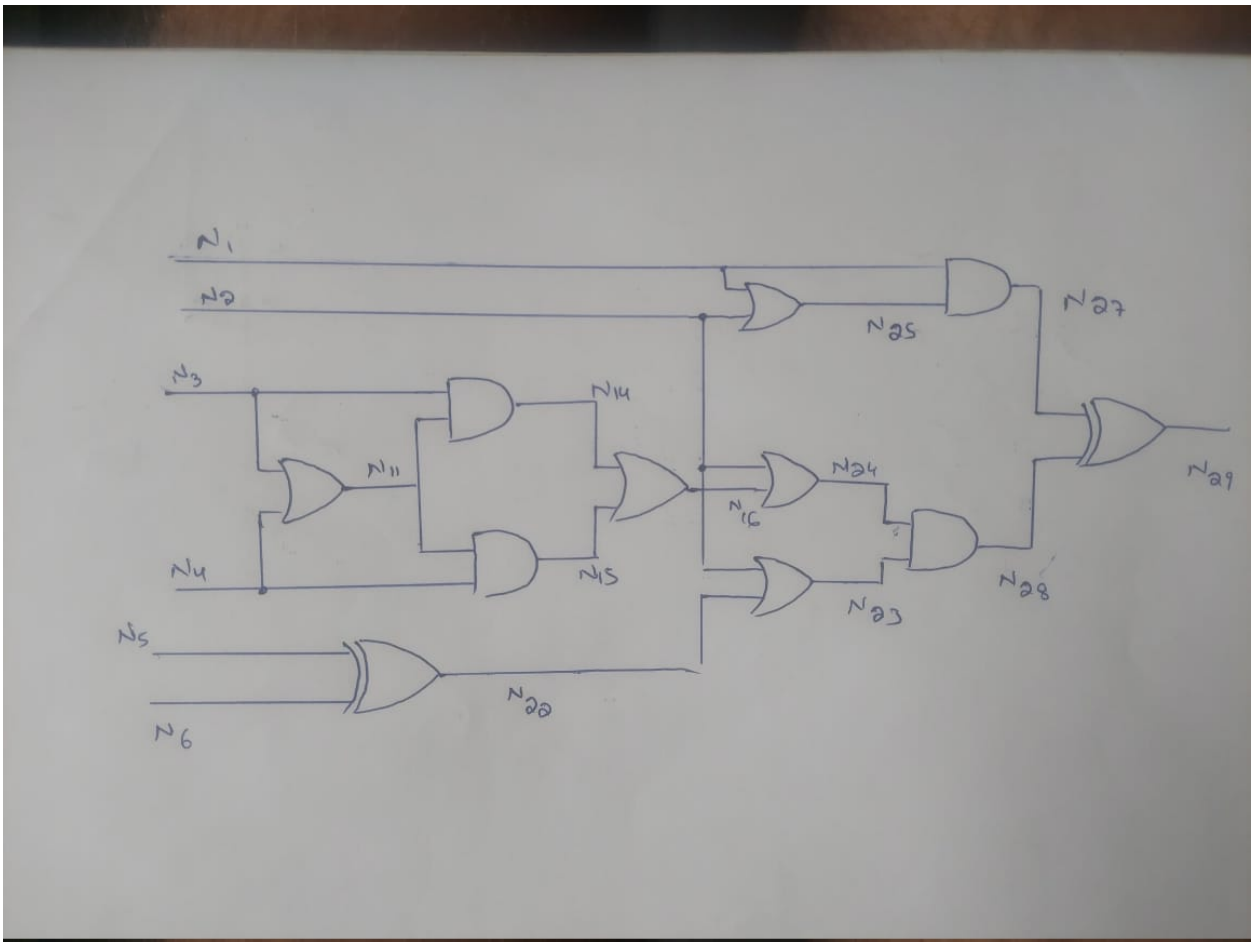
Faults vs tests on C17



Fault	N1 (PI)	N3 (PI)	N6 (PI)	N2 (PI)	N7 (PI)	N22 (PO)	N23 (PO)
N10 sa0	0	1	1	x	x	0	x
N10 sa1	1	1	1	x	x	1	x
N11 sa0	0	x	0	1	x	1	x
N11 sa1	0	1	1	1	x	0	x
N16 sa0	0	1	1	x	x	0	x
N16 sa1	0	x	0	1	x	1	x
N19 sa0	x	1	1	x	x	x	0
N19 sa1	x	x	0	0	1	x	1

As seen in the table above, our implementation of the FAN ATPG algorithm successfully generates test patterns for all faults (on internal nodes) in the c17 circuit. We have manually verified the correctness of these patterns.

Faults vs tests on a larger circuit with XOR and NOR gates



Fault	N1 (PI)	N2 (PI)	N3 (PI)	N4 (PI)	N5 (PI)	N6 (PI)	N29 (PO)
N15 sa0	0	0	1	1	0	1	1
N22 sa1	0	0	1	1	x	x	0
N24 sa0	0	x	1	1	0	1	1
N28 sa1	0	0	x	x	0	0	0
N16 sa0	0	0	1	1	0	1	1
N25 sa1	1	0	x	x	0	0	0
N11 sa0	0	0	1	0	0	1	1

The test circuit shows that our code works for all gates including XOR