Boolean Difference Testing

Team Members :
Vishnu Kuncham - 2021102034

Ramgopal - 2021102013

Koundinya - 2021112022

Pradhith - 2021102015

Problem statement:

We need to make a script that performs the boolean difference method and find the test cases that detect the fault in the circuit.

Introduction to Boolean Difference Method

The Boolean Difference method is a pivotal analytical tool in the domain of Design for Testability (DFT), leveraging the principles of Shannon's Expansion Theorem to evaluate and characterize Boolean circuits. This method facilitates the systematic analysis of circuit behavior by expanding an arbitrary Boolean function with respect to specific variables, enabling precise fault detection and test generation.

Mathematical Equations for Boolean Difference

1. **Shannon's Expansion Theorem**

   Any Boolean function F can be expanded about a variable X_i as:

   > $$F = X_i \cdot F_{X_i=1} + X_i{\prime} \cdot F_{X_i=0}$$

   Here, $F_{X_{i=1}}$ and $F_{X_{i=0}}$ represent the function F when $X_i$is set to 1 and 0, respectively.

2. **Condition for Detecting a Stuck-at-0 Fault**

   To detect X_i stuck at 0, the condition is:

   > $$F \oplus F_{X_i=0} = 1$$

   Substituting Shannon's expansion:

> $$\left(X_i \cdot F_{X_i=1} + X_i' \cdot F_{X_i=0}\right) \oplus F_{X_i=0} = 1$$

Simplifying further:

> $$X_i \cdot \left(F_{X_i=1} \oplus F_{X_i=0}\right) = 1$$

3. **Condition for Detecting a Stuck-at-1 Fault**

   To detect X_i stuck at 1, the condition is:

   > $$F \oplus F_{X_i=1} = 1$$

   Substituting Shannon's expansion:

   > $$\left(X_i \cdot F_{X_i=1} + X_i' \cdot F_{X_i=0}\right) \oplus F_{X_i=1} = 1$$

   Simplifying further:

   > $$X_i' \cdot \left(F_{X_i=0} \oplus F_{X_i=1}\right) = 1$$

4. **Boolean Difference Definition**

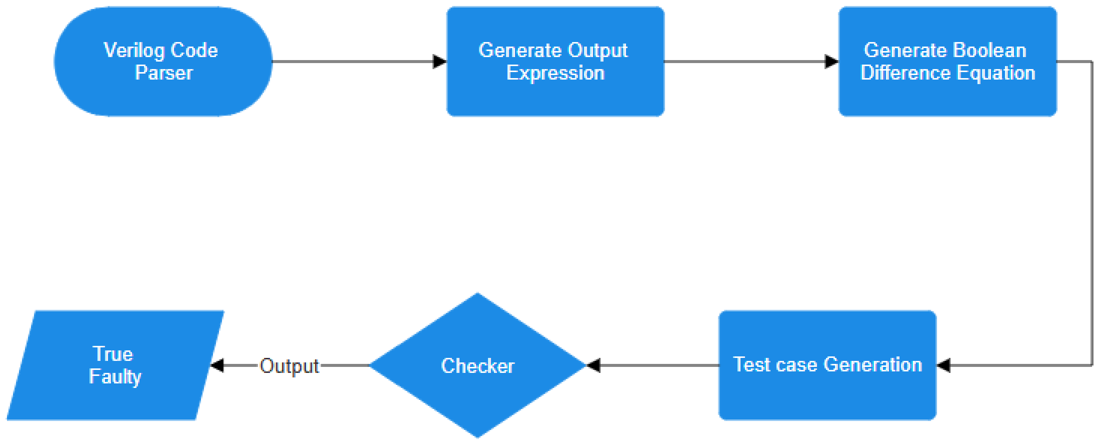
   The Boolean Difference of FF with respect to X_i is defined as:

   > $$\frac{\partial F}{\partial X_i} = F_{X_i=1} \oplus F_{X_i=0}$$

   This equation provides the necessary condition for a fault at X_i to propagate to the output.

Now we use this to implement the boolean difference method.

Methodology & Implementation:



We have done our implementation using python and verilog to take the input circuit.

Given implementation process is divided into 5 stages namely:

```
- Parsing Verilog circuit
- Generating output expression
- Generating boolean difference
- Test case generation
- Checker
```

We will start with all the blocks individually and go through test cases for every block:

Verilog Parser:

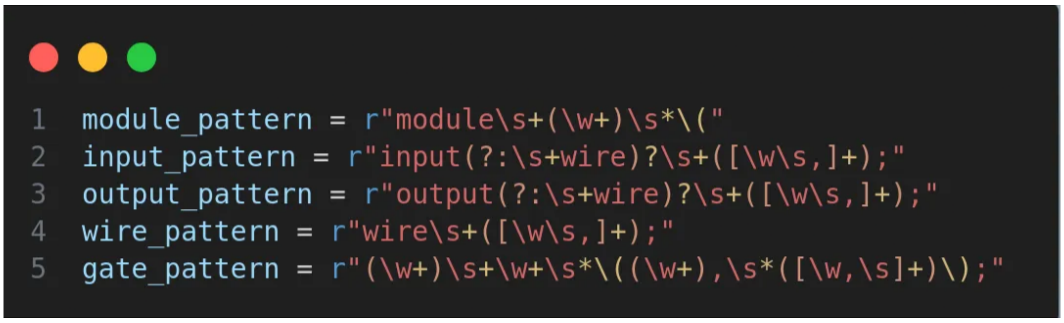
Since we are dealing all the processing part in python we need to represent our verilog code into a python understandable format. Thus we need to make it as clear as possible which can help us in easily finding the expression after this stage.

We started with storing all the gate types (XOR,AND,OR.....) in first row outputs in second row and the other inputs from 3rd row. This would have given us a 2-D array. But there were many issues such as lose of some nodes and hard to access primary inputs and primary outputs which makes the 2-D array unreliable option.
So we started using dictionaries which helps us in mapping primary inputs, primary outputs, internals of the circuit with key and names. Here we can

directly access individual gates directly and we can get the information of them too.

Working of Parser:

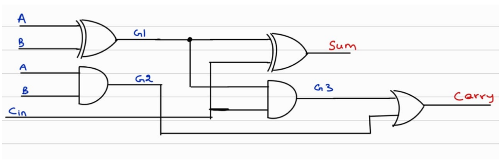
We are assuming that the code is given in structural format. Any verilog code has inputs, outputs, wires and structural information. Now we need to find those in the given verilog code. So we need to find the inputs but finding the string input in verilog and take the nodes after those until we hit ;. Similar things can be done with other wires, output declaration and for gates we need to parse all the loops.

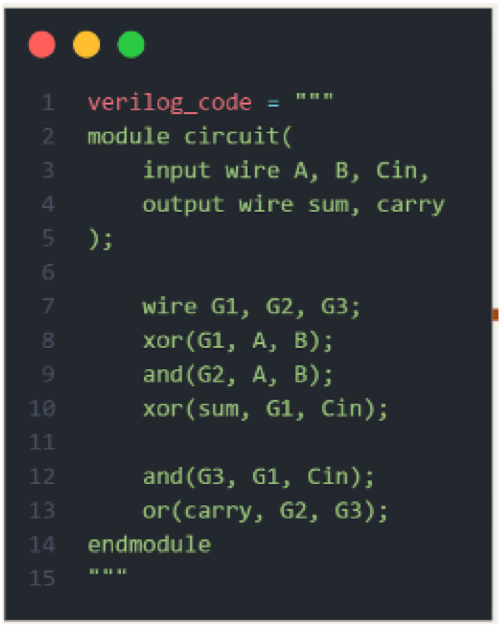


```python
module_pattern = r"module\s+(\w+)\s*\("
input_pattern = r"input(?:\s+wire)?\s+([\w\s,]+);"
output_pattern = r"output(?:\s+wire)?\s+([\w\s,]+);"
wire_pattern = r"wire\s+([\w\s,]+);"
gate_pattern = r"(\w+)\s+\w+\s*\((\w+),\s*([\w,\s]+)\);"
```

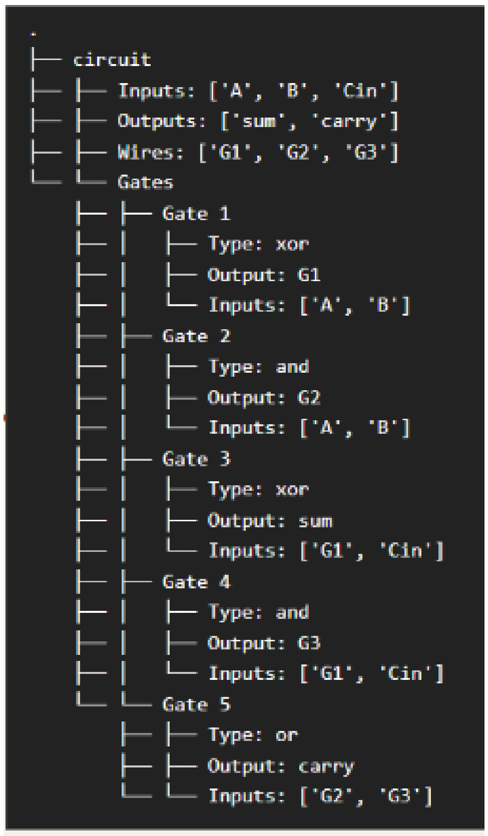
As we can see in the above code snippet we kept patterns to find the modules,inputs,outputs,wires,gates and their inputs and outputs.

Once searching is done the verilog code is modified as a dictionary in python that can be used for further stages.
The mapping of verilog code to parser code output is as follows;





```
verilog_code = """
module circuit(
    input wire A, B, Cin,
    output wire sum, carry
);

    wire G1, G2, G3;
    xor(G1, A, B);
    and(G2, A, B);
    xor(sum, G1, Cin);

    and(G3, G1, Cin);
    or(carry, G2, G3);
endmodule
"""
```



```
.
├── circuit
├── ├── Inputs: ['A', 'B', 'Cin']
├── ├── Outputs: ['sum', 'carry']
├── ├── Wires: ['G1', 'G2', 'G3']
└── └── Gates
    ├── ├── Gate 1
    ├── │   ├── Type: xor
    ├── │   ├── Output: G1
    ├── │   └── Inputs: ['A', 'B']
    ├── ├── Gate 2
    ├── │   ├── Type: and
    ├── │   ├── Output: G2
    ├── │   └── Inputs: ['A', 'B']
    ├── ├── Gate 3
    ├── │   ├── Type: xor
    ├── │   ├── Output: sum
    ├── │   └── Inputs: ['G1', 'Cin']
    ├── ├── Gate 4
    ├── │   ├── Type: and
    ├── │   ├── Output: G3
    ├── │   └── Inputs: ['G1', 'Cin']
    └── └── Gate 5
            ├── ├── Type: or
            ├── ├── Output: carry
            └── └── Inputs: ['G2', 'G3']
```

Corresponding dictionary that contains the information about the complete circuit

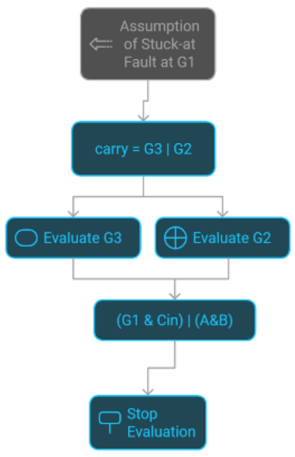
Output expression generator:

Once the parser gives the output we need to generate the boolean expression for all the outputs.
But we should not parse the whole circuit because that would give us an end to end expression where the output expression has only primary inputs. Now we need to modify this algorithm such that the final expression should be in terms of primary inputs and fault_nodes only.
Lets see an example of this

Consider the same adder expression considered earlier. We go through a recursion process where we start with the primary outputs. Now for each output we need to go towards the primary inputs.

Flow diagram

For example consider G1-SA1 in the given adder circuit earlier. We have two outputs sum and carry. Lets consider the case of carry now. The algorithm works as shown in the Image. Here we start with outputs expression and it is evaluated to the recent inputs of it carry was derived from an "OR" gate where G3 and G2 are the inputs. Now comes the recursion part. First check G3 and G2 are primary inputs or not and then check if they are stuck_nodes or not. Now if they don't follow such expression we need to evaluate those expression. Here evaluation G3 we get G1 & Cin where G1 is the stuck nodes and Cin is the primary inputs. Similarly for G2 and the final expression is found. The base condition for recursion is the values in the expression should be either primary inputs or the stuck node.



Finally the same procedure is done to all the other output pins too. The final output for a C432 circuit for the stuck_node N282 is as follows;

```
N223 Expression:  ~(~(~N1 & N4) & ~(~N11 & N17) & ~(~N24 &
N30) & ~(~N37 & N43)............& ~(~N50 & N56) & ~(~N63 &
N69) & ~(~N76 & N82) & ~(~N89 & N95) & ~(~N102 & N108))

N329 Expression:  ~(~((~(~(~N1 & N4) & ~(~N11 & N17) & ~(~N
24 & N30)..................... ~(~N89 & N95) & ~(~N102 & N1
08)) ^ ~(~N102 & N108)) & ~(N112 | ~N108)))

N370 Expression:  ~(~((~(~((~(~(~N1 & N4) & ~(~N11 & N17) &
~(~N24 & N30).......... ~(~N89 & N95) & ~(~N102 & N108)) ^
~(~N102 & N108)) & ~(N115 | ~N108))))

N421 Expression:  ~(~~(N4 & ~(N1 & ~(~(~N1 & N4) & ~(~N11 &
N17) & ~(~N24 & N30)...~(~N102 & N108)) & ~(N115 | ~N10
8)))) & N115) & N108)))

N430 Expression:  ~(~(~(~(~(~N1 & N4) & ~(~N11 & N17) & ~(~
N24 & N30)................ ~(~N50 & N56) & ~(~N63 & N69) &
~(~N76 & N82) & ~(~N89 & N95) & ~(~N102 & N108)) ^
~(~N102 & N108)) & ~(N115 | ~N108)))) & N66) & N56))

N431 Expression:  ~(~(~(~(~(~N1 & N4) & ~(~N11 & N17) & ~(~
N24 & N30)..............& ~(~N102 & N108)) ^ ~(~N102 & N10
```

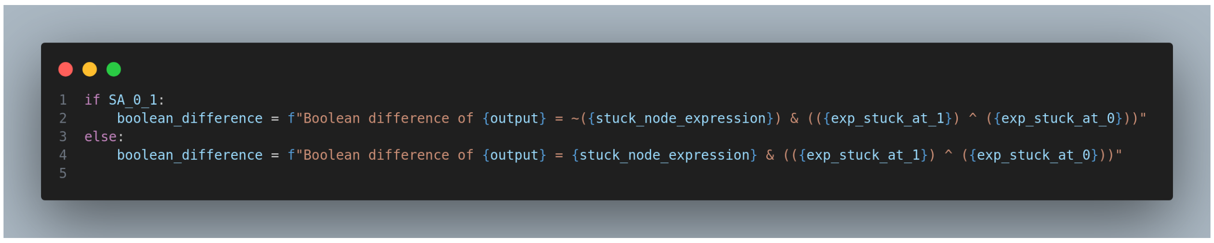
```
8)) & ~(N115 | ~N108)))) & N92) & N82)))

N432 Expression:  ~(~(~(~(~(~N1 & N4) & ~(~N11 & N17) & ~(~
N24 & N30)..............~(~N102 & N108)) & ~(N115 | ~N10
8)))) & N105) & N95)))

Stuck Node N282 expression:
~((~(~(~(~N1 & N4) & ~(~N11 & N17) & ~(~N24 & N30) & ~(~N37 &
N43) & ~(~N50 & N56)... & ~(~N63 & N69) & ~(~N76 & N82) & ~
(~N89 & N95) & ~(~N102 & N108)) ^ ~(~N89 & N95)) & ~(N99 |
~N95))
```

These are the output expression that helps in finding the final boolean difference expression.

Boolean Difference Generation:

Once the output expression are generated we need to generate the boolean difference expression based on the fault type (SA1 , SA0). we need to find the final expression through which we extract the final test cases. The logic for the expression is as follows;
Initially replace the stuck_node with value 0 which gives us $f(x=0)$ and replacing 1 with stuck_node gives us $f(x=1)$. The boolean difference defined as $\frac{dy}{dx} = f(x=0) \oplus f(x=1)$

Now using this boolean difference expression we need to and it with stuck_node if the fault is stuck at 0 else we need to complement our stuck_node and then applied to and logic as earlier. The code snippet for this shown below.



```python
if SA_0_1:
    boolean_difference = f"Boolean difference of {output} = ~({stuck_node_expression}) & (({exp_stuck_at_1}) ^ ({exp_stuck_at_0}))"
else:
    boolean_difference = f"Boolean difference of {output} = {stuck_node_expression} & (({exp_stuck_at_1}) ^ ({exp_stuck_at_0}))"
```

We get final boolean difference expression after running this function.

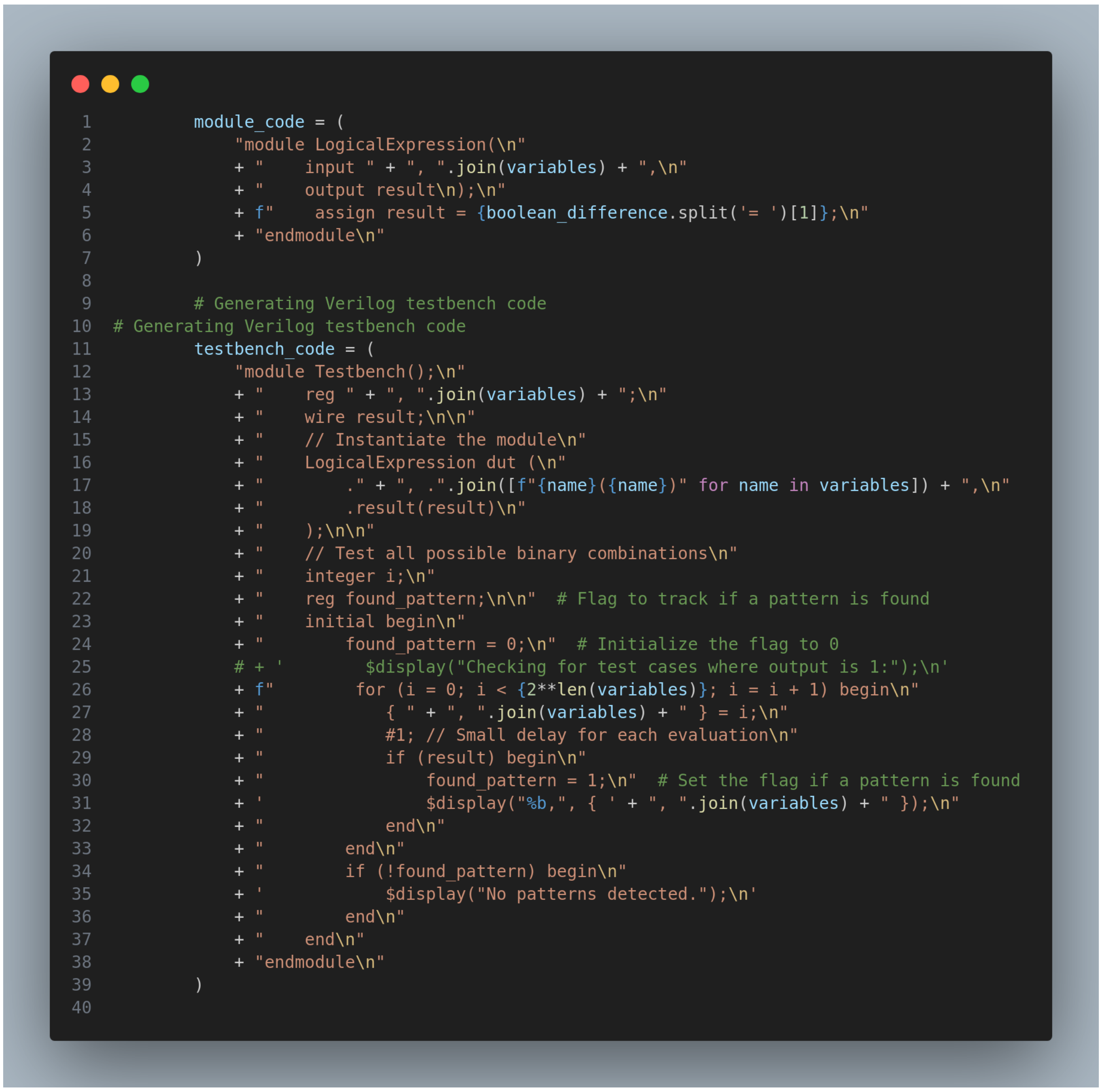
Test Pattern generator:

Once the boolean difference expressions are evaluated we need to find the test patterns that would result 1 when substituted. So here lets consider the circuit of C432 which has 36 inputs. To compute and check 2^36 inputs it would be a large computation which takes days to run the code. So we tried to minimise this by just calculating the count of inputs that the output expression depended on. Let's say there is an output expression which contains only 4 primary input parameters now we need to only check for 2^4 cases and remaining inputs are X (Don't care) anyways.

But though this minimisation helps in some of the cases there would be many cases where we can't check for all the test cases. This would be the drawback of boolean expression method too since for small circuits it works efficiently but as we progress towards large circuits evaluating and test pattern generation would become hard and computationally expensive.

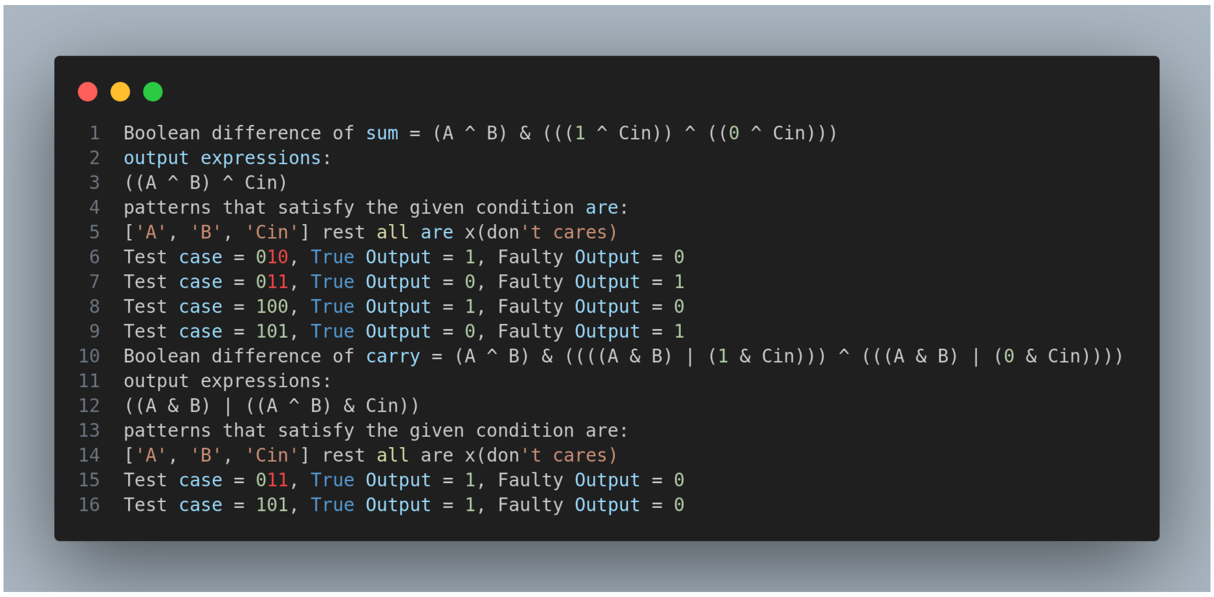
Now Python was giving some late responses when sequentially code is computed. So This test pattern generation process includes developing a verilog script that takes the evaluated expressions from python and when the verilog code is run it would compute the test cases.

The verilog code snippet is given in python it self this is would create a verilog code run it and then take the sub process output from verilog and returns it to the python again for computation.



```python
        module_code = (
            "module LogicalExpression(\n"
            + "    input " + ", ".join(variables) + ",\n"
            + "    output result\n);\n"
            + f"    assign result = {boolean_difference.split('= ')[1]};\n"
            + "endmodule\n"
        )

        # Generating Verilog testbench code
# Generating Verilog testbench code
        testbench_code = (
            "module Testbench();\n"
            + "    reg " + ", ".join(variables) + ";\n"
            + "    wire result;\n\n"
            + "    // Instantiate the module\n"
            + "    LogicalExpression dut (\n"
            + "        ." + ", .".join([f"{name}({name})" for name in variables]) + ",\n"
            + "        .result(result)\n"
            + "    );\n\n"
            + "    // Test all possible binary combinations\n"
            + "    integer i;\n"
            + "    reg found_pattern;\n\n"  # Flag to track if a pattern is found
            + "    initial begin\n"
            + "        found_pattern = 0;\n"  # Initialize the flag to 0
            # + '        $display("Checking for test cases where output is 1:");\n'
            + f"        for (i = 0; i < {2**len(variables)}; i = i + 1) begin\n"
            + "            { " + ", ".join(variables) + " } = i;\n"
            + "            #1; // Small delay for each evaluation\n"
            + "            if (result) begin\n"
            + "                found_pattern = 1;\n"  # Set the flag if a pattern is found
            + '                $display("%b,", { ' + ", ".join(variables) + " });\n"
            + "            end\n"
            + "        end\n"
            + "        if (!found_pattern) begin\n"
            + '            $display("No patterns detected.");\n'
            + "        end\n"
            + "    end\n"
            + "endmodule\n"
        )

```

Example output is as follows:

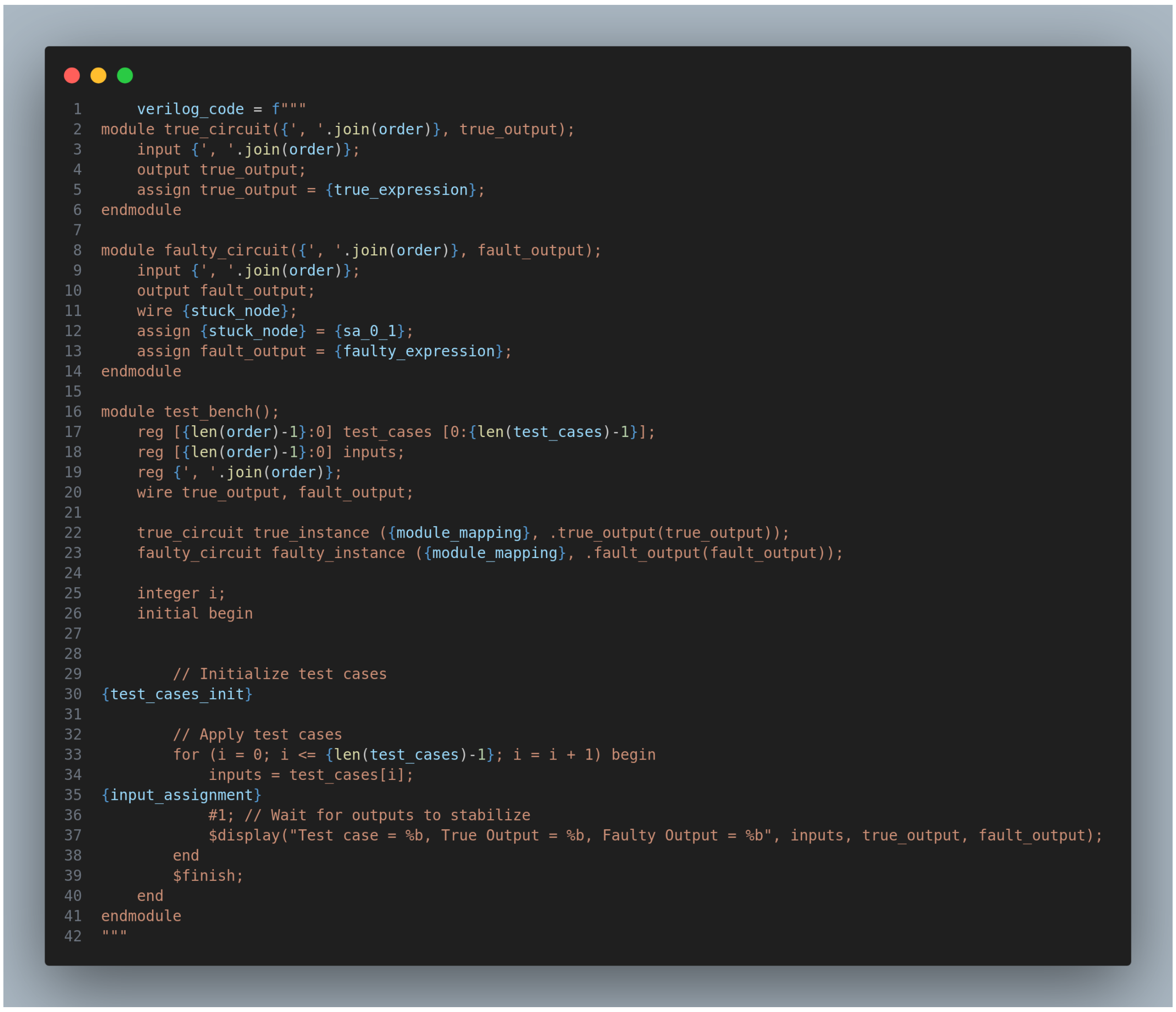

```
Boolean difference of sum = (A ^ B) & (((1 ^ Cin)) ^ ((0 ^ Cin)))
output expressions:
((A ^ B) ^ Cin)
patterns that satisfy the given condition are:
['A', 'B', 'Cin'] rest all are x(don't cares)
Test case = 010, True Output = 1, Faulty Output = 0
Test case = 011, True Output = 0, Faulty Output = 1
Test case = 100, True Output = 1, Faulty Output = 0
Test case = 101, True Output = 0, Faulty Output = 1
Boolean difference of carry = (A ^ B) & ((((A & B) | (1 & Cin))) ^ (((A & B) | (0 & Cin))))
output expressions:
((A & B) | ((A ^ B) & Cin))
patterns that satisfy the given condition are:
['A', 'B', 'Cin'] rest all are x(don't cares)
Test case = 011, True Output = 1, Faulty Output = 0
Test case = 101, True Output = 1, Faulty Output = 0
```

For the given sum expression the patterns generated are 001,010,101 and 110 that produce 1 when substituted in the boolean difference sum expression. The true output and faulty output for that case will be discussed in the next part.

This works fine until we have 2^20 inputs if that is increases out computation time increases so we have computed the test cases only when the output is dependent on 20 or less than 20 primary inputs. If the count exceeds I'm printing that the computation of that output is expensive because our system are limited to the RAM and also time of computation. So lets say computing 2^36 inputs for every output would take days to run the code. We tried by dividing the task into chunks by sub process but that too didn't work properly. So Given if we have a higher computation systems we can try to run 2^42 test cases too but that would be computationally hard. To solve this we can also restrict the test case checking let's say whatever the task we do we can restrict our input computation to that values lets say I only need 1/2 test cases then restricting the computation for every output to 20 also works.

Checker script for test patterns:

Checker script check the test patterns generated for example It will use the parser to find the true expression and anyways we have a faulty expression from our previous computation. We just need to build 2 modules one for true circuit, one for faulty circuit. Once they are done the test bench need to have all the test patterns generated earlier and the outputs will be checked based on that. The checking is also done in verilog where the generalised script is loaded into python as follows;



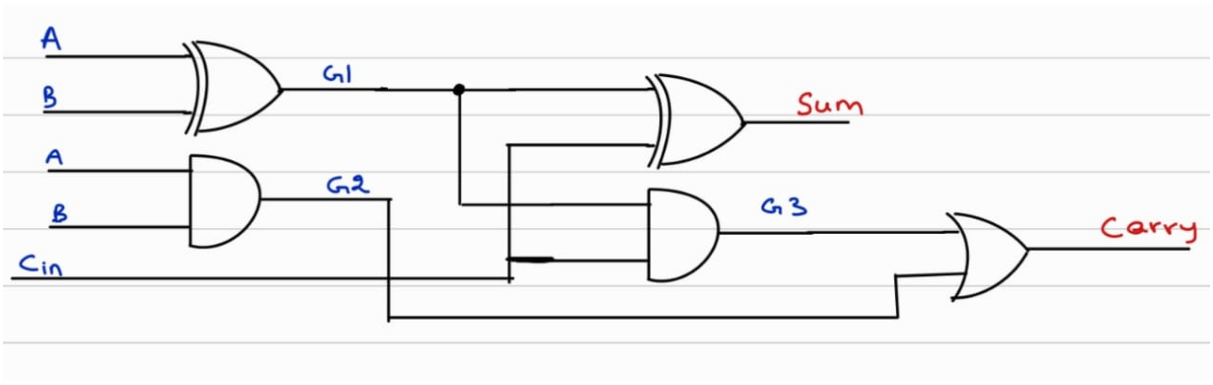
```
    verilog_code = f"""
module true_circuit({', '.join(order)}, true_output);
    input {', '.join(order)};
    output true_output;
    assign true_output = {true_expression};
endmodule

module faulty_circuit({', '.join(order)}, fault_output);
    input {', '.join(order)};
    output fault_output;
    wire {stuck_node};
    assign {stuck_node} = {sa_0_1};
    assign fault_output = {faulty_expression};
endmodule

module test_bench();
    reg [{len(order)-1}:0] test_cases [0:{len(test_cases)-1}];
    reg [{len(order)-1}:0] inputs;
    reg {', '.join(order)};
    wire true_output, fault_output;

    true_circuit true_instance ({module_mapping}, .true_output(true_output));
    faulty_circuit faulty_instance ({module_mapping}, .fault_output(fault_output));

    integer i;
    initial begin


        // Initialize test cases
{test_cases_init}

        // Apply test cases
        for (i = 0; i <= {len(test_cases)-1}; i = i + 1) begin
            inputs = test_cases[i];
{input_assignment}
            #1; // Wait for outputs to stabilize
            $display("Test case = %b, True Output = %b, Faulty Output = %b", inputs, true_output, fault_output);
        end
        $finish;
    end
endmodule
"""
```

This schematic is used to generated a verilog file and when the file is run that will give the true and faulty outputs. Now our test patterns should detect the fault thus the faulty and non-faulty outputs should be complementary.

Thus as seen in the earlier adder example the true output and fault output are printed there for every test case.

Testing the code (Outputs):

Full Adder :



Stuck node = G1 SA0

The outputs are as follows;

```
Boolean difference of sum = (A ^ B) & (((1 ^ Cin)) ^ ((0 ^ Ci
output expressions:
((A ^ B) ^ Cin)
patterns that satisfy the given condition are:
['A', 'B', 'Cin'] rest all are x(don't cares)
Test case = 010, True Output = 1, Faulty Output = 0
Test case = 011, True Output = 0, Faulty Output = 1
Test case = 100, True Output = 1, Faulty Output = 0
Test case = 101, True Output = 0, Faulty Output = 1
Boolean difference of carry = (A ^ B) & ((((A & B) | (1 & Cin
(((A & B) | (0 & Cin))))
output expressions:
((A & B) | ((A ^ B) & Cin))
patterns that satisfy the given condition are:
['A', 'B', 'Cin'] rest all are x(don't cares)
Test case = 011, True Output = 1, Faulty Output = 0
Test case = 101, True Output = 1, Faulty Output = 0
```

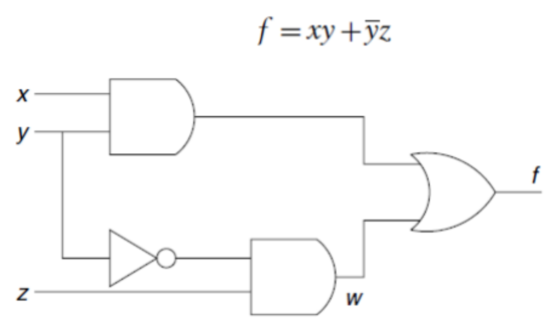
stuck node = G3 SA0

```
Boolean difference of carry = ((A ^ B) & Cin) & ((((A & B) |
(((A & B) | 0)))
output expressions:
((A & B) | ((A ^ B) & Cin))
patterns that satisfy the given condition are:
['A', 'B', 'Cin'] rest all are x(don't cares)
```

```
Test case = 011, True Output = 1, Faulty Output = 0
Test case = 101, True Output = 1, Faulty Output = 0
```

Example code given in class:

We need to find the test patterns for y SA0 and also W SA0.



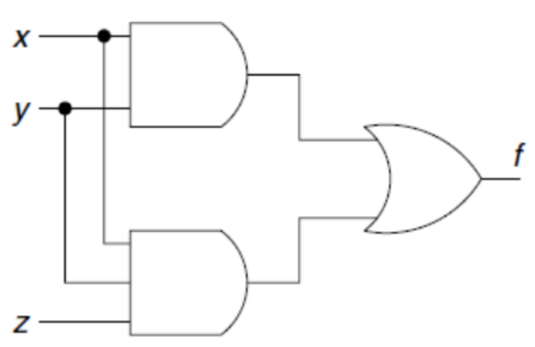
- Consider the fault y stuck at zero.
- Any test vector that satisfies y.df/dy =1 will detect the fault.
- df/dy = f(y=1) ⊕ f(y=0) = x ⊕ z
- y.df/dy = 1 → y.(xz' + x'z) = 1 → xyz'+x'yz=1
- Test vectors = 110, 011

- Consider the fault w stuck at zero.
- Any test vector that satisfies w.df/dw =1 will detect the fault.
- Test vectors = 001, 101

The output generated by the code is as follows;

```
Result for Y - SA0
Boolean difference of f = y & ((((x & 1) | (~1 & z))) ^
(((x & 0) | (~0 & z))))
patterns that satisfy the given condition are:
['z', 'y', 'x'] rest all are x(don't cares)
Test case = 011, True Output = 1, Faulty Output = 0
Test case = 110, True Output = 0, Faulty Output = 1

Result for W - SA0
Boolean difference of f = (~y & z) & ((((x & y) | 1)) ^
(((x & y) | 0)))
patterns that satisfy the given condition are:
['z', 'y', 'x'] rest all are x(don't cares)
Test case = 100, True Output = 1, Faulty Output = 0
Test case = 101, True Output = 1, Faulty Output = 0
```

Example for no test patterns generation:



- Consider the fault z stuck at zero.
- Any test vector that satisfies z.df/dz =1 will detect the fault.

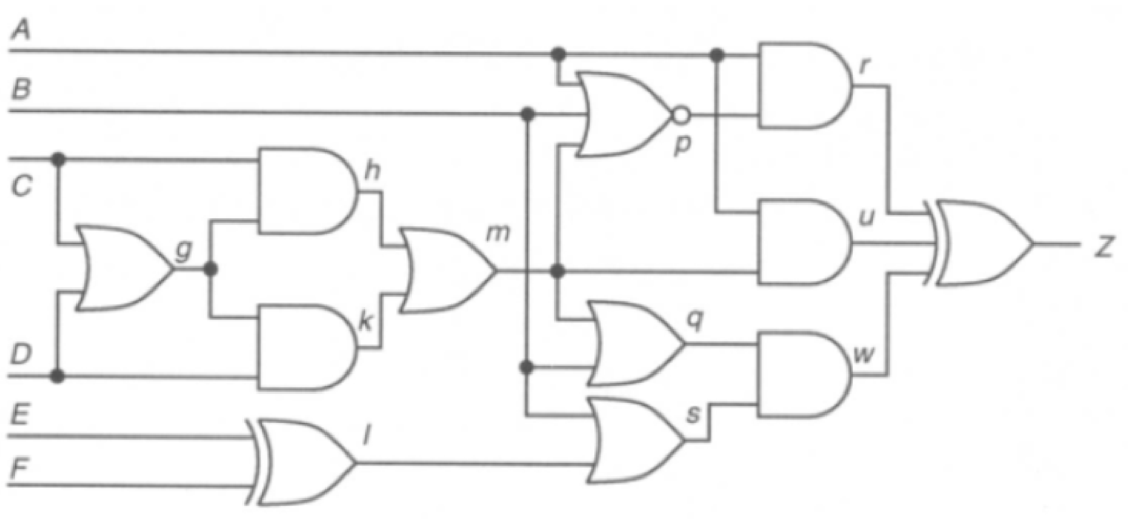
- No test vector satisfies z.df/dz =1.
- Z stuck at zero is not testable.

Here there should be no pattern that satisfies the given condition.

Output of the code is as follows;

```
Boolean difference of f = z & ((((x & y) | (x & y & 1))) ^
(((x & y) | (x & y & 0))))
patterns that satisfy the given condition are:
['z', 'y', 'x'] rest all are x(don't cares)
no pattern detected
```

Circuit Given for mid eval:



Result for u-SA1:

```
Boolean difference of z = ~((a & ((c & (c | d)) | ((c | d)
& d)))) &
```

```
((((a & ~(a | b | ((c & (c | d)) | ((c | d) & d)))) ^ 1 ^
((((c & (c | d)) |
((c | d) & d)) | b) & (b | (e ^ f))))) ^ (((a & ~(a | b |
((c & (c | d)) |
((c | d) & d)))) ^ 0 ^ ((((c & (c | d)) | ((c | d) & d)) |
b) & (b | (e ^ f))))))
patterns that satisfy the given condition are:
['c', 'd', 'f', 'a', 'e', 'b'] rest all are x(don't cares)
Test case = 000000, True Output = 0, Faulty Output = 1
Test case = 000001, True Output = 1, Faulty Output = 0
Test case = 000010, True Output = 0, Faulty Output = 1
Test case = 000011, True Output = 1, Faulty Output = 0
Test case = 000100, True Output = 0, Faulty Output = 1
Test case = 000101, True Output = 1, Faulty Output = 0
Test case = 000110, True Output = 0, Faulty Output = 1
Test case = 000111, True Output = 1, Faulty Output = 0
Test case = 001000, True Output = 0, Faulty Output = 1
There are some more too I truncated them whole output can b
e seen in code.
```

Result for z-SA1:

```
Boolean difference of z = ~(((a & ~(a | b | ((c & (c | d))
| ((c | d) & d)))) ^
(a & ((c & (c | d)) | ((c | d) & d))) ^ ((((c & (c | d)) |
((c | d) & d)) | b) &
(b | (e ^ f))))) & ((1) ^ (0))
output expressions:
((a & ~(a | b | ((c & (c | d)) | ((c | d) & d)))) ^ (a &
((c & (c | d)) |
((c | d) & d))) ^ ((((c & (c | d)) | ((c | d) & d)) | b) &
(b | (e ^ f))))
patterns that satisfy the given condition are:
['c', 'd', 'f', 'a', 'e', 'b'] rest all are x(don't cares)
Test case = 000000, True Output = 0, Faulty Output = 1
Test case = 000010, True Output = 0, Faulty Output = 1
Test case = 000100, True Output = 0, Faulty Output = 1
Test case = 000110, True Output = 0, Faulty Output = 1
Test case = 001000, True Output = 0, Faulty Output = 1
Test case = 001010, True Output = 0, Faulty Output = 1
Test case = 001100, True Output = 0, Faulty Output = 1
Test case = 001110, True Output = 0, Faulty Output = 1
Test case = 010000, True Output = 0, Faulty Output = 1
Test case = 010101, True Output = 0, Faulty Output = 1
Test case = 010110, True Output = 0, Faulty Output = 1
....
```

C432:

Output for N1-SA1

```
Boolean difference of N223 = ~(N1) & ((~(~(~1 & N4) & ~(~N1
1 & N17) & ~(~N24 & N30)& ~(~N37 & N43) & ~(~N50 & N56) & ~
(~N63 & N69) & ~(~N76 & N82) & ~(~N89 & N95) & ~(~N102 & N1
08))) ^ (~(~(~0 & N4) & ~(~N11 & N17) & ~(~N24 & N30) & ~(~
N37 & N43)& ~(~N50 & N56) & ~(~N63 & N69) & ~(~N76 & N82) &
~(~N89 & N95) & ~(~N102 & N108))))

output expressions:
~(~(~N1 & N4) & ~(~N11 & N17) & ~(~N24 & N30) & ~(~N37 & N4
3) & ~(~N50 & N56) &
~(~N63 & N69) & ~(~N76 & N82) & ~(~N89 & N95) & ~(~N102 & N
108))
patterns that satisfy the given condition are:
['N43', 'N63', 'N17', 'N89', 'N37', 'N11', 'N82', 'N76', 'N
1', 'N69', 'N108',
'N102', 'N56', 'N95', 'N24', 'N50', 'N4', 'N30'] rest all a
re x(don't cares)
Test case = 000000000000000010, True Output = 1, Faulty Out
put = 0
Test case = 000000000000000110, True Output = 1, Faulty Out
put = 0
Test case = 000000000000001010, True Output = 1, Faulty Out
put = 0
Test case = 000000000000001011, True Output = 1, Faulty Out
put = 0
Test case = 000000000000001110, True Output = 1, Faulty Out
```

```
put = 0
Test case = 000000000000001111, True Output = 1, Faulty Out
put = 0
Test case = 000000000000100110, True Output = 1, Faulty Out
put = 0
Test case = 000000000000101110, True Output = 1, Faulty Out
put = 0
Test case = 000000000000101111, True Output = 1, Faulty Out
put = 0
Test case = 000000000001000010, True Output = 1, Faulty Out
put = 0
Test case = 000000000001000110, True Output = 1, Faulty Out
put = 0
Test case = 000000000001001010, True Output = 1, Faulty Out
put = 0
Test case = 000000000001001011, True Output = 1, Faulty Out
put = 0
Test case = 000000000001001110, True Output = 1, Faulty Out
put = 0
Test case = 000000000001001111, True Output = 1, Faulty Out
put = 0
Test case = 000000000001100110, True Output = 1, Faulty Out
put = 0
Test case = 000000000001101110, True Output = 1, Faulty Out
put = 0
Test case = 000000000001101111, True Output = 1, Faulty Out
put = 0
Test case = 000000000011000010, True Output = 1, Faulty Out
put = 0
Test case = 000000000011000110, True Output = 1, Faulty Out
put = 0
Test case = 000000000011001010, True Output = 1, Faulty Out
put = 0
Test case = 000000000011001011, True Output = 1, Faulty Out
put = 0
.....
Each one of the output has its own test cases. When the exp
ression is
```

```
dependent on more than 20 inputs we are not calculating tes
t cases since
it would be computationally expensive.
```

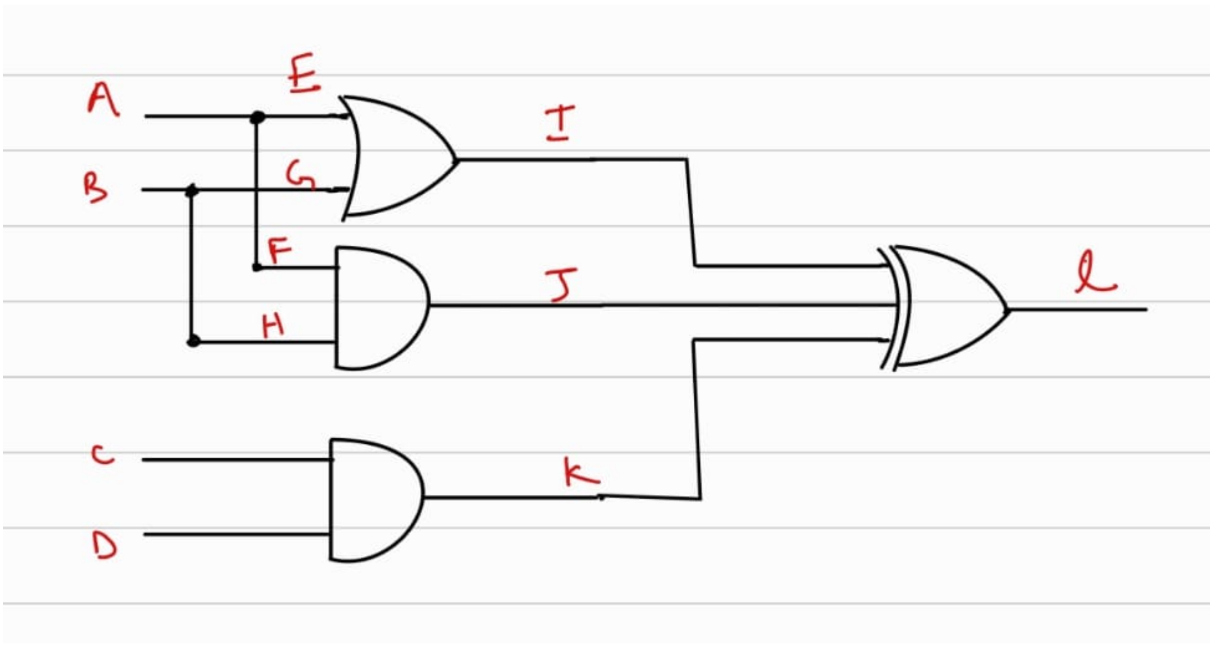
output for N309 - SA0:

```
Boolean difference of N370 = ~(~((~(~(~N1 & N4) & ~(~N11 &
N17) & ~(~N24 & N30)
...............~(~N76 & N82) & ~(~N89 & N95) & ~(~N102 & N1
08)) ^ ~(~N76 & N82))& ~(N92 | ~N82)))~(~N102 & N108)) ^ ~
(~N102 & N108)) & ~(N115 | ~N108))))))
running this code is computationally expensive since the ou
tput is dependent on 36 input parameters

Boolean difference of N421 = ~(~((~(~(~N1 & N4) & ~(~N11 &
N17) & ~(~N24 & N30)
& ~(~N37 & N43) & ~(~N50 & N56) & ~(~N63 & N69) & ~(~N76 &
N82) & ~(~N89 & N95)
& ~(~N102 & N108)) ^ ~(~N1 & N4)) & ~(N8 | ~N
4))..................& ~(~N89 & N95) & ~(~N102 & N108)) ^
~(~N102 & N108)) &~(N115 | ~N108)))) & N115) & N108)))))
running this code is computationally expensive since the ou
tput is dependent on 36 input parameters

Boolean difference of N430 = ~(~((~(~(~N1 & N4) & ~(~N11 &
N17) & ~(~N24 & N30)
& ~(~N37 & N43) & ~(~N50 & N56) & ~(~N63 & N69) & ~(~N76 &
N82) & ~(~N89 & N95)
& ~(~N102 & N108)) ^ ~(~N1 & N4)) & ~(N8 | ~N4)) & ~((~(~(~
N1 & N4) & ~(~N11 & N17)& ~(~N24 & N30) & ~(~N37 & N43) & ~
(~N50 & N56) & ~(~N63 & N69) & ~(~N76 & N82) &......... ~(~
N50 & N56) & ~(~N63 & N69) & ~(~N76 & N82) & ~(~N89 & N95)
& ~(~N102 & N108)) ^ ~(~N102 & N108))& ~(N115 | ~N108)))) &
N66) & N56))))
running this code is computationally expensive since the ou
tput is dependent on 36 input parameters
```

```
Boolean difference of N431 = ~(~((~(~(~N1 & N4) & ~(~N11 &
N17) & ~(~N24 & N30)
.............& ~(~N37 & N43) & ~(~N50 & N56) & ~(~N63 & N6
9)
& ~(~N76 & N82) & ~(~N89 & N95) & ~(~N102 & N108)) ^ ~(~N10
2 & N108))
& ~(N115 | ~N108)))) & N92) & N82)))))
running this code is computationally expensive since the ou
tput is dependent on 36 input parameters

Boolean difference of N432 = ~(~((~(~(~N1 & N4) & ~(~N11 &
N17) & ~(~N24 & N30).............~(~N24 & N30) & ~(~N37 &
N43) & ~(~N50 & N56) & ~(~N63 & N69) &
~(~N76 & N82) & ~(~N89 & N95) & ~(~N102 & N108)) ^ ~(~N102
& N108))
& ~(N115 | ~N108)))) & N105) & N95)))))
running this code is computationally expensive since the ou
tput is dependent on 36 input parameters
```

Handling Fanout’s :

We need to check for **Fanout nodes** too since. For Boolean difference method the major thing is when we backtrack the expression for fanout and stem would be same. Now how to handle the stuck at fault at stem is taken care by the stuck node expression. As per expression the output would be same thus it is automatically considered.



example of FAN out circuit.

output for (e SA0):

```
output expressions of l:
(((a) | (b)) | ((a) & (b)) | (c & d))
patterns that satisfy the given condition are:
['d', 'b', 'c', 'a'] rest all are x(don't cares)
Test case = 0001, True Output = 1, Faulty Output = 0
Test case = 0011, True Output = 1, Faulty Output = 0
Test case = 1001, True Output = 1, Faulty Output = 0
```

output for (f SA1):

```
output expressions of l:
(((a) | (b)) | ((a) & (b)) | (c & d))
patterns that satisfy the given condition are:
['d', 'b', 'c', 'a'] rest all are x(don't cares)
no pattern detected
```

Thus this code handles FAN out nodes too and efficient in finding the test patterns.

Hurdles:

As a part of this project we learnt about the Boolean difference method. The major problems we encountered are

Parsing: To convert the Verilog circuit into a python circuit we need to make a parser. We started by making 2-D array and some array for inputs and outputs. Everything was scattered so we were unable to index properly and access the required elements. To cover one variable many variables came into play.
In order to solve this we made a dictionary format as shown in the parsing circuit implementation.

Large circuits many outputs:

Since our project was Boolean difference the output expression were very large and in many cases the output expressions does not contain the stuck node so x^x = 0 so no pattern used to satisfy those conditions. In that case it was waste of computation for that output.

In order to reduce that computation we have made a verification such that when the expression of outputs are generated if the outputs doesn't contain the expression of stuck_node then the output would be removed and not sent to test case calculation.

Test case analysis:

The output expression sometimes depends upon 30 inputs and as we increase the complexity of circuit that complexity would be increases. Now we can't compute 2^30 values for all the outputs thus we were skipping those outputs which were having more than 20 dependent inputs. Here dependent inputs are nothing but the expression is affected by 30 primary inputs.

Checker:

Smaller circuits can be checked easily by taking the circuit logic. But as we go to larger circuits the output expression maybe more than 200 lines sometimes so we can't manually evaluate them so we are required to make a checker that tests the outputs generated.
In order to do this we have made a
checker.py first then we integrated it in our main code. So the check would take the test case and give it to true and faulty circuit and show the output.

Individual Contribution:

Vishnu & Ram Gopal - Made parser, output expression generator and boolean difference equation generator.

Koundinya & Pradhith - Dealt with verilog codes that are generated by the script, made checkers

Everyone contributed equally in documenting this as a report.

References:

Slides of boolean algebra from the course.