CSE - 312

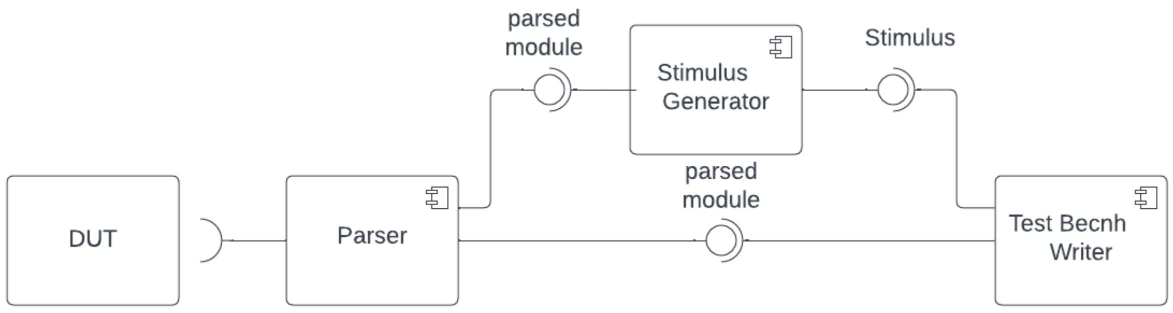
VeriGen

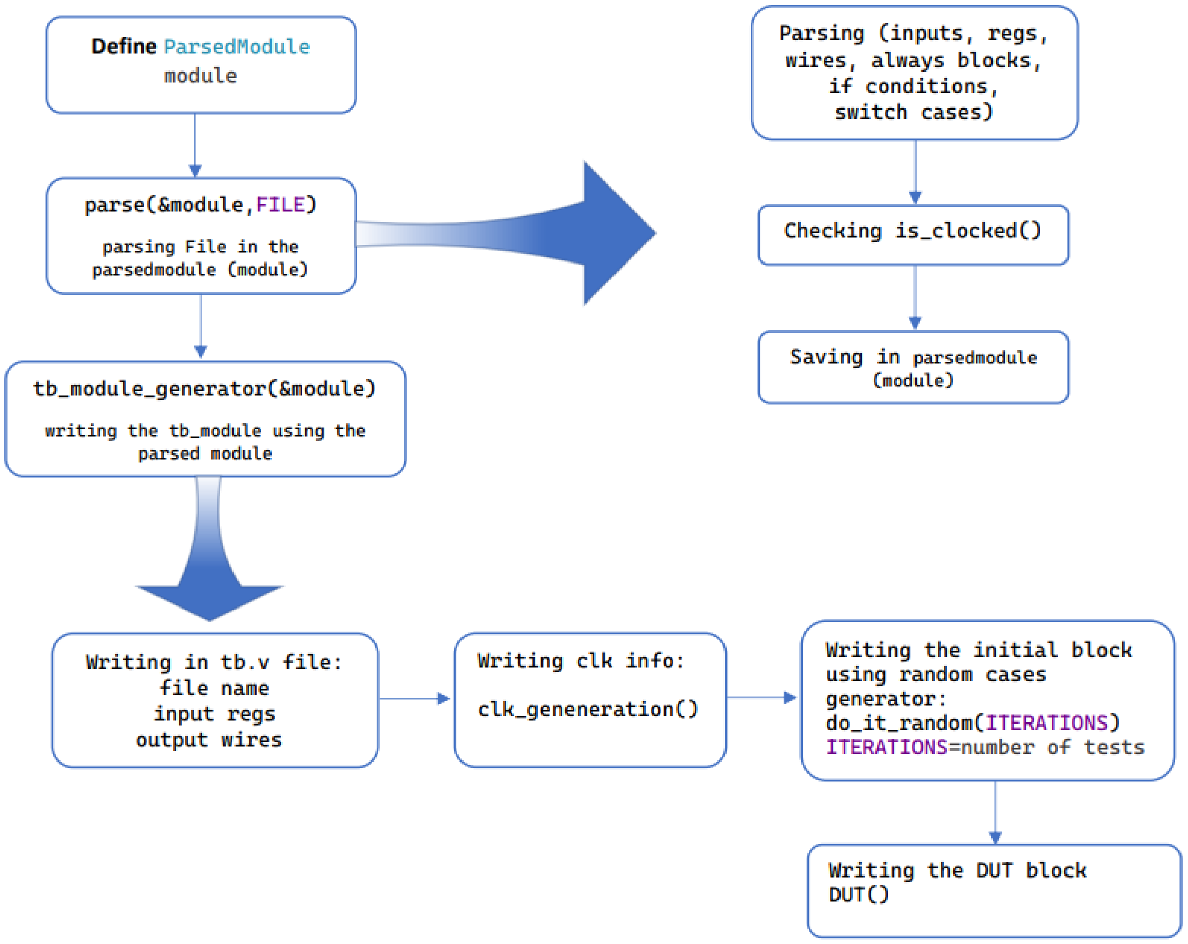
TEAM 7

Mostafa Mohamed Khalil Mahmoud	20P2981
Farah Ahmed AbdelRehim Tharwat	20P1269
Peter Ehab Ibrahim Azmy	20p3995
Mohamed Mostafa Bedair ElMaghraby	20p7732
Osama Saad Farouk kotb	20p3943
Rafik Tamer Magdy	20P1046
Mohamed Ahmed Rizk	21P0364
Abdulrahman Mohammed Sayed Hassan	21P0334

VeriGen

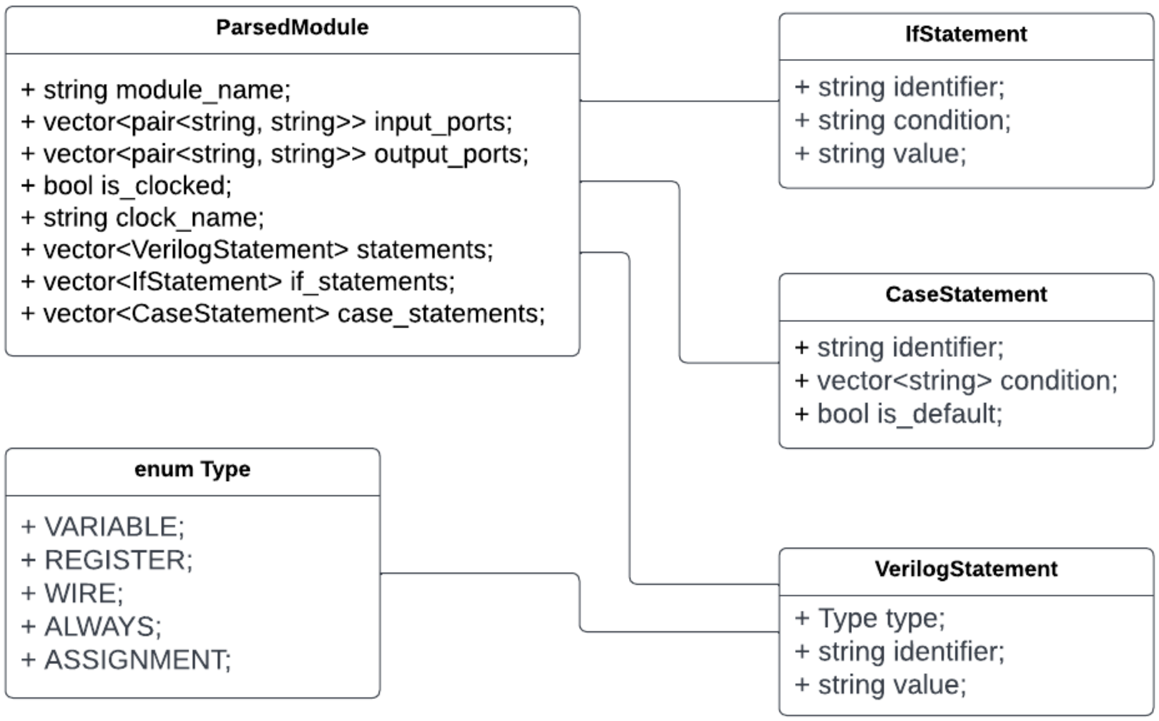
The tool is made up of three components, as shown in the component diagram. The diagram also illustrates the relationships between the components. Another diagram shows the collaboration between the components.





Define ParsedModule module
parse(&module,FILE)
parsing File in the parsedmodule (module)
tb_module_generator(&module)
writing the tb_module using the parsed module
Parsing (inputs, regs, wires, always blocks, if conditions, switch cases)
Checking is_clocked()
Saving in parsedmodule (module)
Writing in tb.v file: file name input regs output wires
Writing clk info: clk_geneneration()
Writing the initial block using random cases generator: do_it_random(ITERATIONS) ITERATIONS=number of tests
Writing the DUT block DUT()

We have agreed to unify our work by each working on the ParsedModule structure independently, as shown in the following diagram, allowing us to work on our respective tasks without interference.



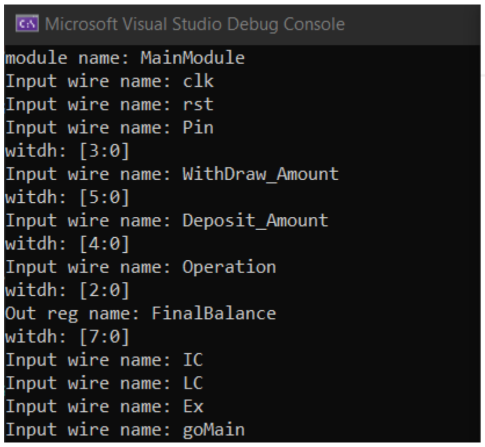
Verilog Parser

With the help of regular expressions, we managed to input a .v file, read it and parse everything we need from it. At first we did parse everything then print it, in the second phase we stored them in some data structures to be used next.

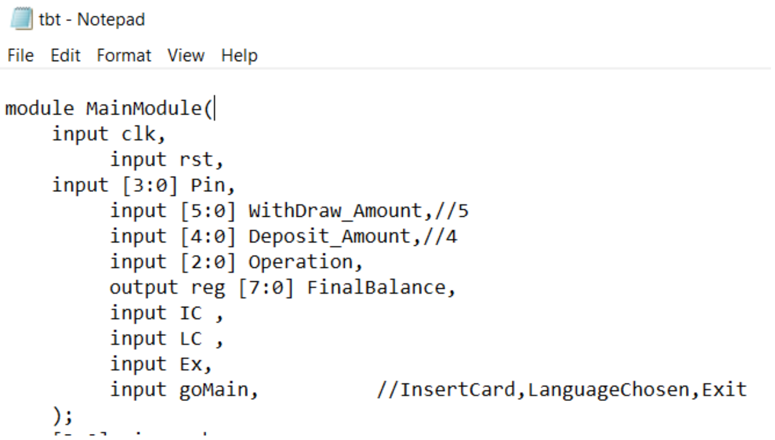
The structure of the code was mainly reading the code line by line and then detect if there is one of the Verilog reserved words, or any type of statements that we need to parse.

We were able to parse all of the following:

Module name, inputs and outputs. The following snippet is the Verilog code we tested and the corresponding parsed output of our program.



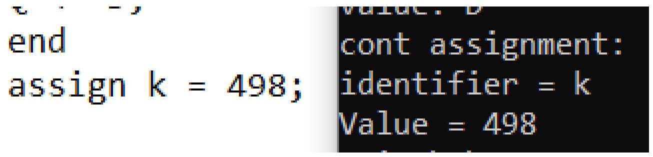
```
module name: MainModule
Input wire name: clk
Input wire name: rst
Input wire name: Pin
witdh: [3:0]
Input wire name: WithDraw_Amount
witdh: [5:0]
Input wire name: Deposit_Amount
witdh: [4:0]
Input wire name: Operation
witdh: [2:0]
Out reg name: FinalBalance
witdh: [7:0]
Input wire name: IC
Input wire name: LC
Input wire name: Ex
Input wire name: goMain
```



```
module MainModule(
    input clk,
         input rst,
    input [3:0] Pin,
         input [5:0] WithDraw_Amount,//5
         input [4:0] Deposit_Amount,//4
         input [2:0] Operation,
         output reg [7:0] FinalBalance,
         input IC ,
         input LC ,
         input Ex,
         input goMain,          //InsertCard,LanguageChosen,Exit
    );
```

Note that we specify the type of inputs and outputs whether they are wires or registers. Also note how we capture the width of each.

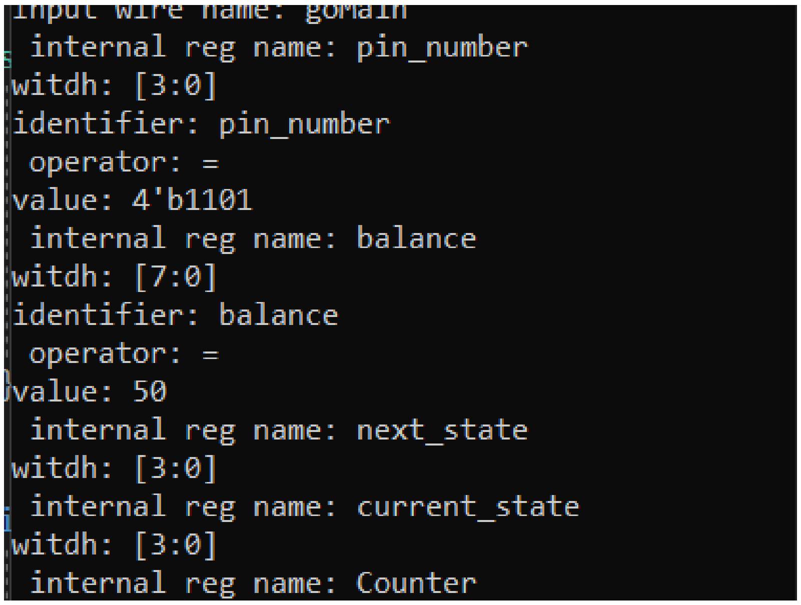
Continuous assignment is also considered:



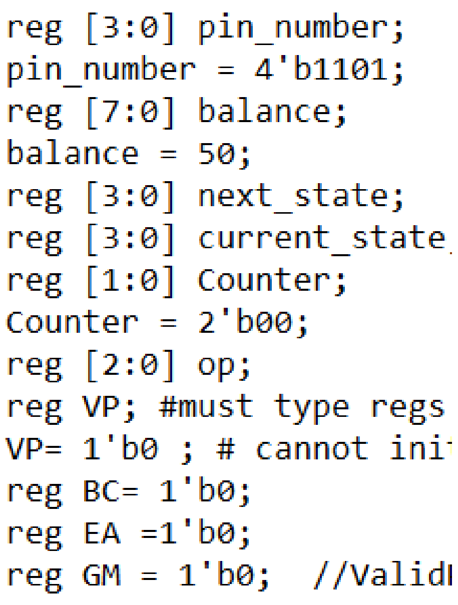
```
end
assign k = 498;
```

```
cont assignment:
identifier = k
Value = 498
```

We then take a separate variables to indicate any additional internal wires or registers that are defined anywhere else. Also notice how we read a line such as “balance = 50;”, See the following:



```
Input wire name: goMain
 internal reg name: pin_number
witdh: [3:0]
identifier: pin_number
 operator: =
value: 4'b1101
 internal reg name: balance
witdh: [7:0]
identifier: balance
 operator: =
value: 50
 internal reg name: next_state
witdh: [3:0]
 internal reg name: current_state
witdh: [3:0]
 internal reg name: Counter
```

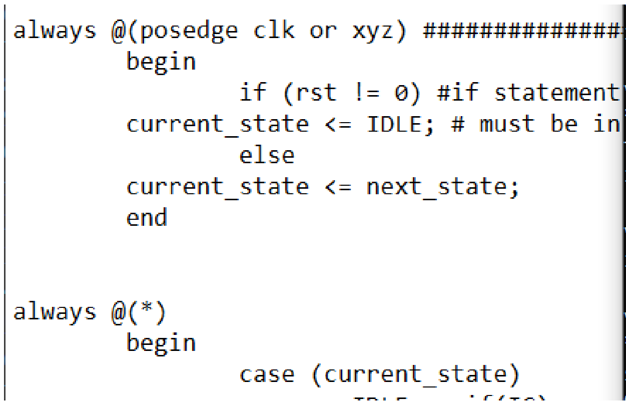


```
reg [3:0] pin_number;
pin_number = 4'b1101;
reg [7:0] balance;
balance = 50;
reg [3:0] next_state;
reg [3:0] current_state
reg [1:0] Counter;
Counter = 2'b00;
reg [2:0] op;
reg VP; #must type regs
VP= 1'b0 ; # cannot ini
reg BC= 1'b0;
reg EA =1'b0;
reg GM = 1'b0;  //ValidI
```

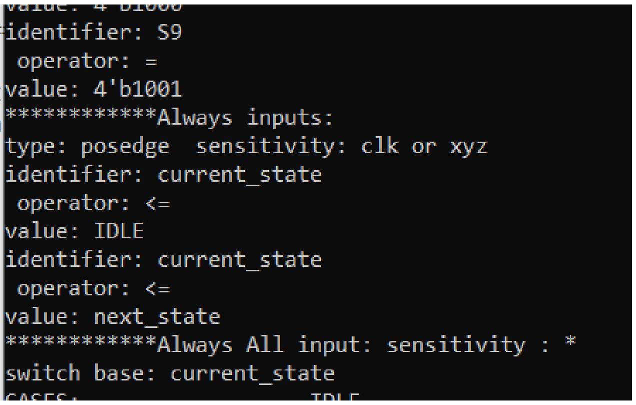
Here the “balance = 50;” is read divided into three variables:

First the identifier which is the LHS, the operator which is the assignment operator and then the value assigned to the identifier. Note that the value can be a number written in either the normal formatting like 50 or line 2’b00 or can be another variable. We tested it on every possible form and it does exactly like it should.

Always block is also considered:

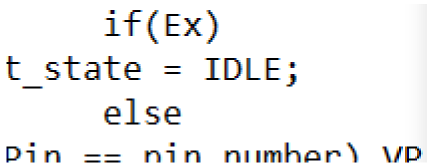


```
always @(posedge clk or xyz) ##############
        begin
                if (rst != 0) #if statement
        current_state <= IDLE; # must be in
                else
        current_state <= next_state;
        end


always @(*)
        begin
                case (current_state)
                        IDLE    if(IC)
```

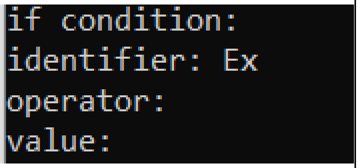


```
value: 4'b1000
identifier: S9
 operator: =
value: 4'b1001
************Always inputs:
type: posedge  sensitivity: clk or xyz
identifier: current_state
 operator: <=
value: IDLE
identifier: current_state
 operator: <=
value: next_state
************Always All input: sensitivity : *
switch base: current_state
CASES:                 IDLE
```

We also read if conditions like the following:



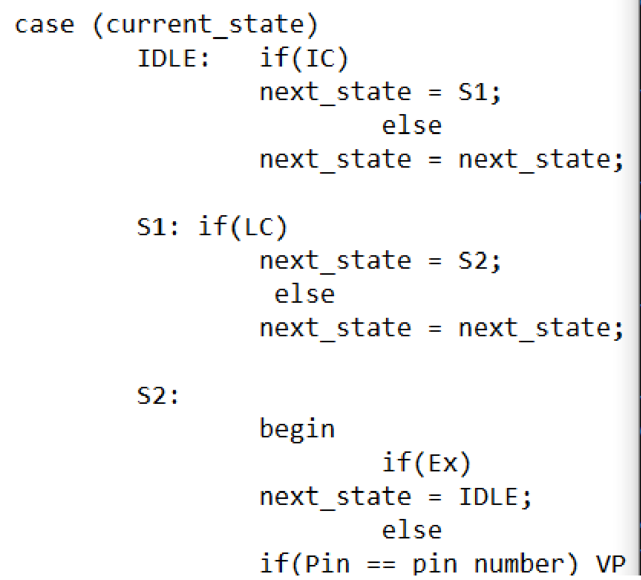
```
       if(Ex)
t_state = IDLE;
       else
Pin == pin number) VP
```



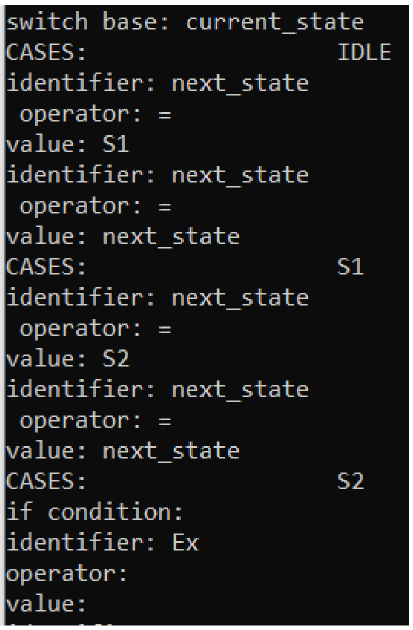
```
if condition:
identifier: Ex
operator:
value:
```

Every if statement has it’s condition, it can be something like x == 5 and then we set the identifier to x, the operator to ==, and the value to 5. Or it can be something like the above where we only have an identifier.

Now we have to parse the case statement:



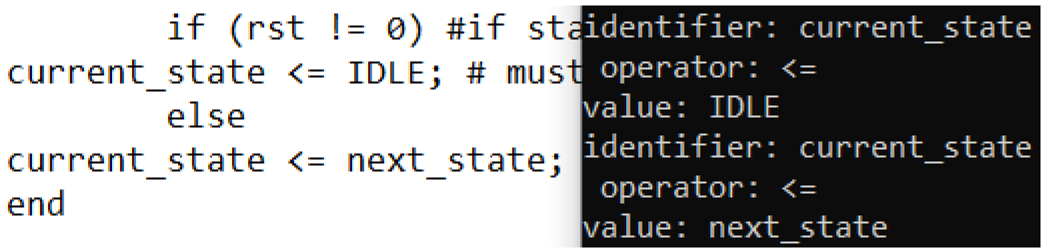
```
case (current_state)
        IDLE:   if(IC)
                next_state = S1;
                        else
                next_state = next_state;

        S1: if(LC)
                next_state = S2;
                 else
                next_state = next_state;

        S2:
                begin
                        if(Ex)
                next_state = IDLE;
                        else
                if(Pin == pin number) VP
```



```
switch base: current_state
CASES:                 IDLE
identifier: next_state
 operator: =
value: S1
identifier: next_state
 operator: =
value: next_state
CASES:                 S1
identifier: next_state
 operator: =
value: S2
identifier: next_state
 operator: =
value: next_state
CASES:                 S2
if condition:
identifier: Ex
operator:
value:
```

Notice that the variable current_state is the switch base which we compare each time with the cases, we then read each case like the IDLE, SI, S2. As you see.

We then read the non-blocking assignment exactly as we explained earlier with the (identifier, operator, value) shape as follows:



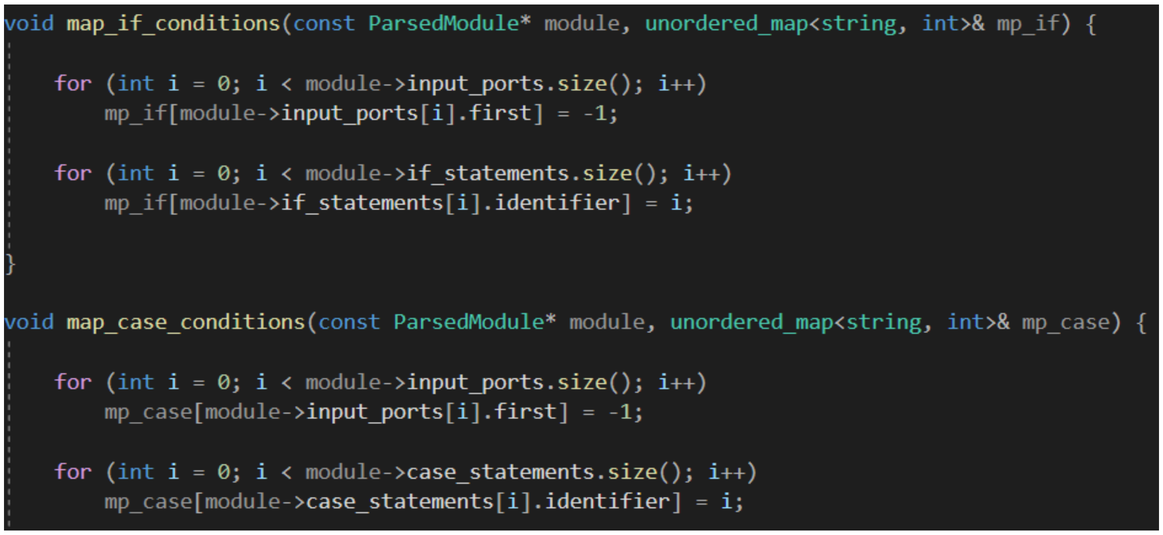
```
        if (rst != 0) #if sta
current_state <= IDLE; # must
        else
current_state <= next_state;
end
```

```
identifier: current_state
 operator: <=
value: IDLE
identifier: current_state
 operator: <=
value: next_state
```

Taking into account the logical operator that has been read throughout the entire code. Hence covering all the required items to be parsed from the Verilog code, enabling others to proceed from this point in their part of the project.

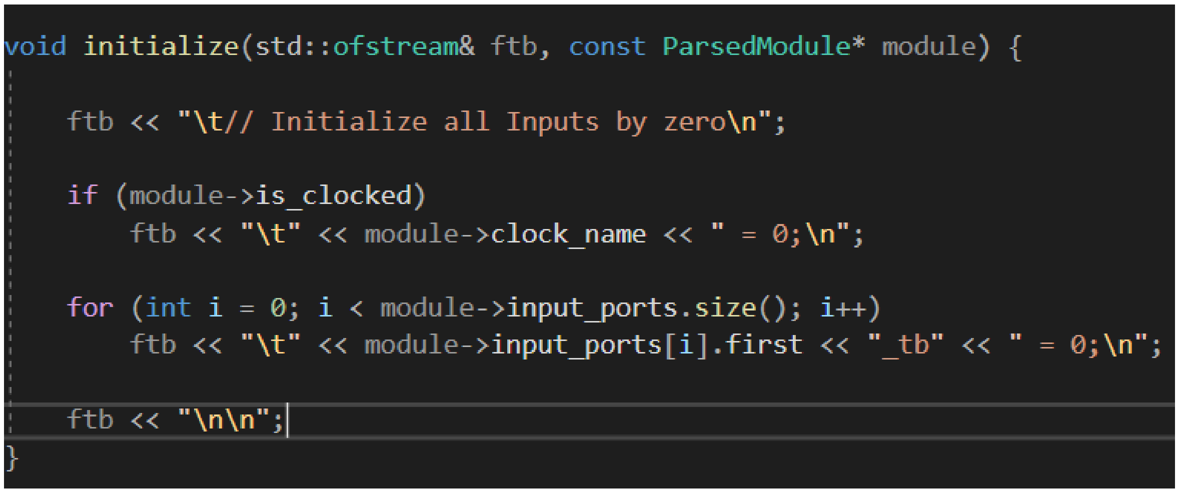
Stimulus Generator:

First, this component maps the if and case conditions to see if it has a certain condition that can be randomized within certain constraints. This is done through the functions "map_if_conditions()" and "map_case_conditions()".



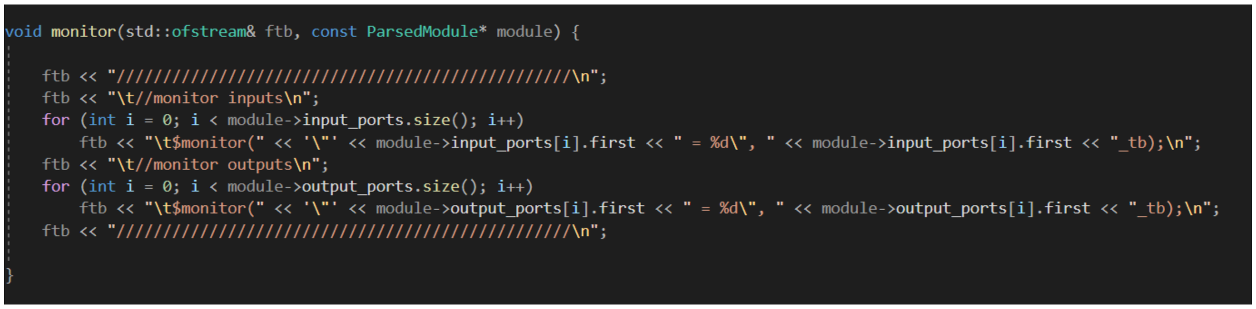
```
void map_if_conditions(const ParsedModule* module, unordered_map<string, int>& mp_if) {

    for (int i = 0; i < module->input_ports.size(); i++)
        mp_if[module->input_ports[i].first] = -1;

    for (int i = 0; i < module->if_statements.size(); i++)
        mp_if[module->if_statements[i].identifier] = i;

}

void map_case_conditions(const ParsedModule* module, unordered_map<string, int>& mp_case) {

    for (int i = 0; i < module->input_ports.size(); i++)
        mp_case[module->input_ports[i].first] = -1;

    for (int i = 0; i < module->case_statements.size(); i++)
        mp_case[module->case_statements[i].identifier] = i;
```

Then stimulus starts by initializing all the input ports by zero, calling "initialize()":



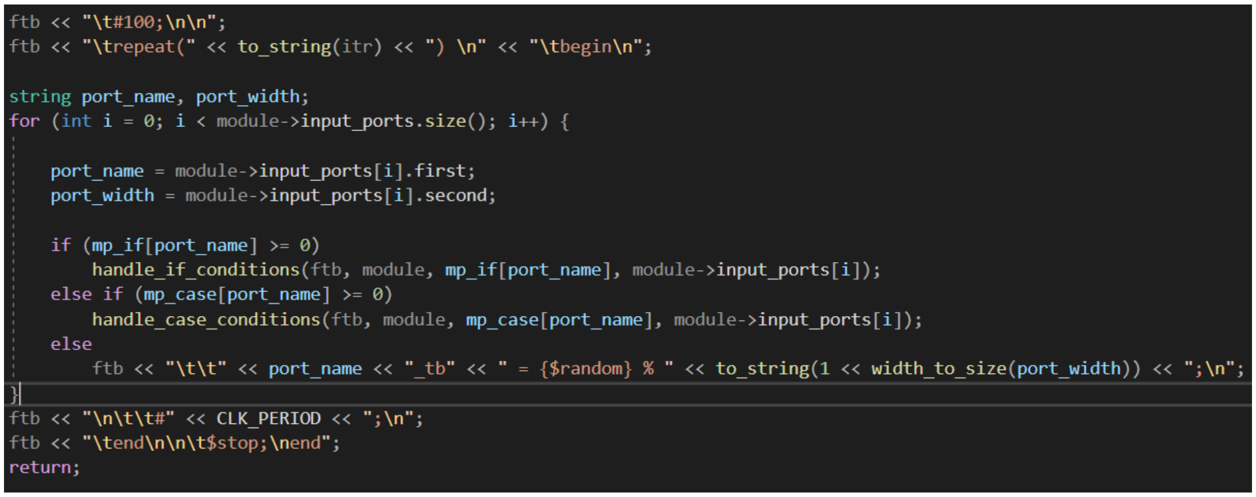
```
void initialize(std::ofstream& ftb, const ParsedModule* module) {

    ftb << "\t// Initialize all Inputs by zero\n";

    if (module->is_clocked)
        ftb << "\t" << module->clock_name << " = 0;\n";

    for (int i = 0; i < module->input_ports.size(); i++)
        ftb << "\t" << module->input_ports[i].first << "_tb" << " = 0;\n";

    ftb << "\n\n";
}
```

After that, the generation of monitoring:



```
void monitor(std::ofstream& ftb, const ParsedModule* module) {

    ftb << "/////////////////////////////////////////////////\n";
    ftb << "\t//monitor inputs\n";
    for (int i = 0; i < module->input_ports.size(); i++)
        ftb << "\t$monitor(" << '\"' << module->input_ports[i].first << " = %d\", " << module->input_ports[i].first << "_tb);\n";
    ftb << "\t//monitor outputs\n";
    for (int i = 0; i < module->output_ports.size(); i++)
        ftb << "\t$monitor(" << '\"' << module->output_ports[i].first << " = %d\", " << module->output_ports[i].first << "_tb);\n";
    ftb << "/////////////////////////////////////////////////\n";

}
```

The generator prints a repeat statement that is based on the number of iterations defined in the "CONFIG.h" file. It then processes each input port. The input is handled in one of three ways, depending on the input module and port: it is either included in a case statement, an if statement, or neither. At the end, it puts the delay corresponding to the CLK_PERIOD, which is defined in the "CONFIG.h" file too.

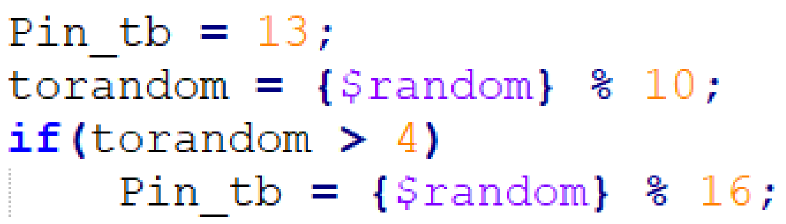


```
ftb << "\t#100;\n\n";
ftb << "\trepeat(" << to_string(itr) << ") \n" << "\tbegin\n";

string port_name, port_width;
for (int i = 0; i < module->input_ports.size(); i++) {

    port_name = module->input_ports[i].first;
    port_width = module->input_ports[i].second;

    if (mp_if[port_name] >= 0)
        handle_if_conditions(ftb, module, mp_if[port_name], module->input_ports[i]);
    else if (mp_case[port_name] >= 0)
        handle_case_conditions(ftb, module, mp_case[port_name], module->input_ports[i]);
    else
        ftb << "\t\t" << port_name << "_tb" << " = {$random} % " << to_string(1 << width_to_size(port_width)) << ";\n";
}
ftb << "\n\t\t#" << CLK_PERIOD << ";\n";
ftb << "\tend\n\n\t$stop;\nend";
return;
```

Handling the randomization:

1. Handling an if condition:

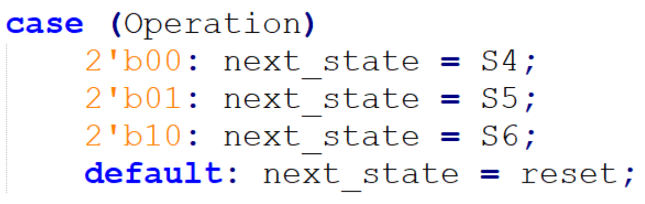
   I. We first check if the port was compared with a number rather than an internal register or something else.
   II. If the port was not compared with a number, we randomize it from zero to its limit based on its size.
   III. If the port was compared with a number, we look for the logical operator used.
   IV. If the operator is "==", there is a 50% chance that the value will be set to equal the number and a 50% chance that it will be randomized within the whole range.
   V. If any other operator is used, there is a 50% chance that the value will be set to be less than the number and a 50% chance that it will be set to be greater than or equal to the number.



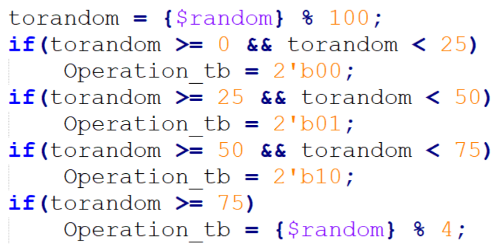
```
Pin_tb = 13;
torandom = {$random} % 10;
if(torandom > 4)
    Pin_tb = {$random} % 16;
```

2. Handling a Case statement: assume we have n conditions and m case with numbers.

   I. For each case value with a number, the expression will have 1/n probability to be equal to that number.
   II. Any other case, including the default, the expression will have a (n-m)/n probability to be randomized within the whole range.

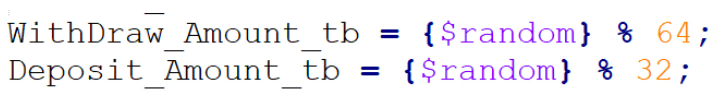


```
case (Operation)
    2'b00: next_state = S4;
    2'b01: next_state = S5;
    2'b10: next_state = S6;
    default: next_state = reset;
```



```
torandom = {$random} % 100;
if(torandom >= 0 && torandom < 25)
    Operation_tb = 2'b00;
if(torandom >= 25 && torandom < 50)
    Operation_tb = 2'b01;
if(torandom >= 50 && torandom < 75)
    Operation_tb = 2'b10;
if(torandom >= 75)
    Operation_tb = {$random} % 4;
```

3. Neither If nor Case: the input will be randomized within the whole range.

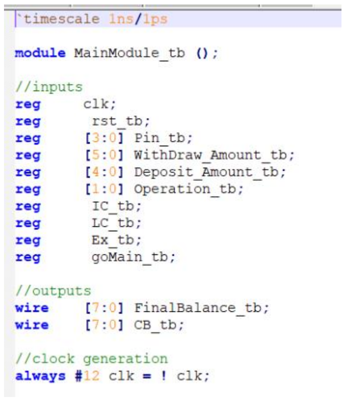


```
WithDraw_Amount_tb = {$random} % 64;
Deposit_Amount_tb = {$random} % 32;
```

Test Bench Writer

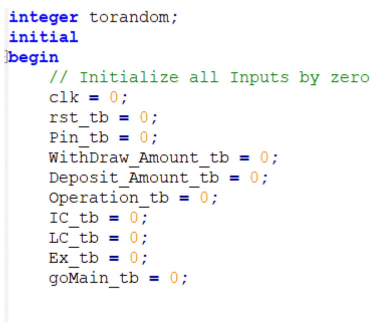
The tb_module_generator function generates a Verilog testbench file for a given ParsedModule object. It does this by opening an output file stream and writing the necessary testbench code to it.

The function starts by opening an output file stream for a file named according to the module name. It then writes the necessary preamble for a Verilog module definition, including the module name and port declarations for the input and output ports of the ParsedModule object. If the ParsedModule object has a non-zero is_clocked field, the function also declares a clock signal port in the testbench module. Next, the clk_generation function is called to generate a clock signal for the testbench if necessary. For example:



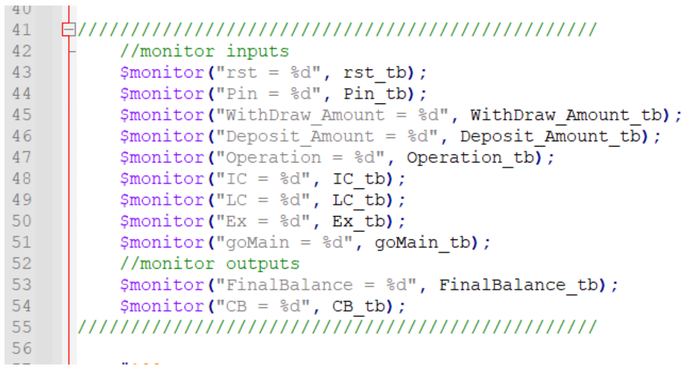
```
`timescale 1ns/1ps

module MainModule_tb ();

//inputs
reg      clk;
reg       rst_tb;
reg      [3:0] Pin_tb;
reg      [5:0] WithDraw_Amount_tb;
reg      [4:0] Deposit_Amount_tb;
reg      [1:0] Operation_tb;
reg       IC_tb;
reg       LC_tb;
reg       Ex_tb;
reg       goMain_tb;

//outputs
wire     [7:0] FinalBalance_tb;
wire     [7:0] CB_tb;

//clock generation
always #12 clk = ! clk;
```

The do_it_random function is then called to generate random stimuli for the input ports of the design module. By first adding the initial values of the registers as seen in the following figure:



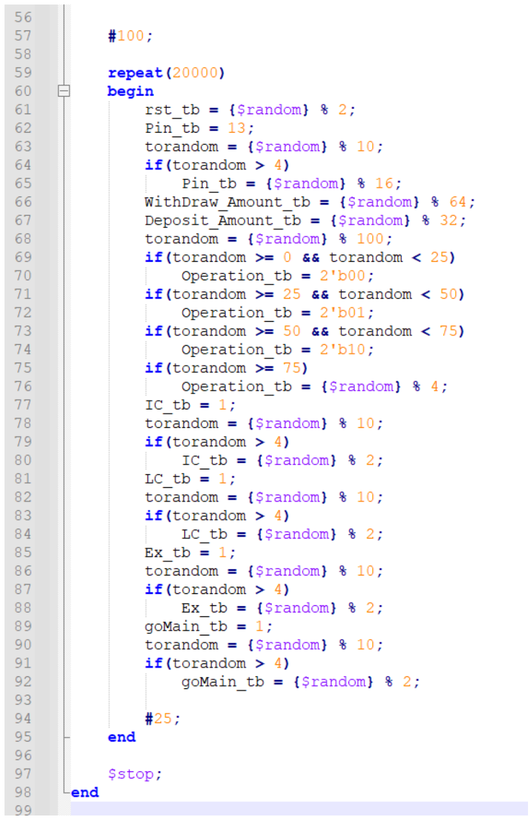
```
integer torandom;
initial
begin
    // Initialize all Inputs by zero
    clk = 0;
    rst_tb = 0;
    Pin_tb = 0;
    WithDraw_Amount_tb = 0;
    Deposit_Amount_tb = 0;
    Operation_tb = 0;
    IC_tb = 0;
    LC_tb = 0;
    Ex_tb = 0;
    goMain_tb = 0;
```

Then monitoring the input and outputs of the module:



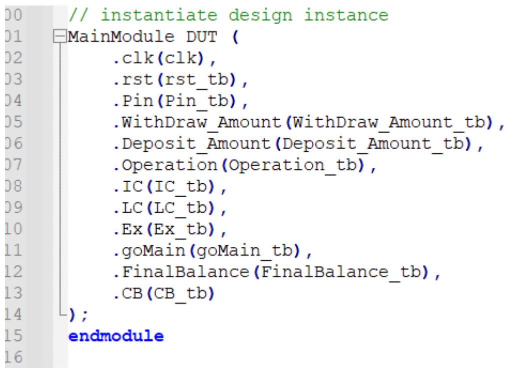
```
///////////////////////////////////////////////////
    //monitor inputs
    $monitor("rst = %d", rst_tb);
    $monitor("Pin = %d", Pin_tb);
    $monitor("WithDraw_Amount = %d", WithDraw_Amount_tb);
    $monitor("Deposit_Amount = %d", Deposit_Amount_tb);
    $monitor("Operation = %d", Operation_tb);
    $monitor("IC = %d", IC_tb);
    $monitor("LC = %d", LC_tb);
    $monitor("Ex = %d", Ex_tb);
    $monitor("goMain = %d", goMain_tb);
    //monitor outputs
    $monitor("FinalBalance = %d", FinalBalance_tb);
    $monitor("CB = %d", CB_tb);
///////////////////////////////////////////////////
```

Then the do_it_random function generate the random values accordingly as shown in the following example:



```

    #100;

    repeat(20000)
    begin
        rst_tb = {$random} % 2;
        Pin_tb = 13;
        torandom = {$random} % 10;
        if(torandom > 4)
            Pin_tb = {$random} % 16;
        WithDraw_Amount_tb = {$random} % 64;
        Deposit_Amount_tb = {$random} % 32;
        torandom = {$random} % 100;
        if(torandom >= 0 && torandom < 25)
            Operation_tb = 2'b00;
        if(torandom >= 25 && torandom < 50)
            Operation_tb = 2'b01;
        if(torandom >= 50 && torandom < 75)
            Operation_tb = 2'b10;
        if(torandom >= 75)
            Operation_tb = {$random} % 4;
        IC_tb = 1;
        torandom = {$random} % 10;
        if(torandom > 4)
            IC_tb = {$random} % 2;
        LC_tb = 1;
        torandom = {$random} % 10;
        if(torandom > 4)
            LC_tb = {$random} % 2;
        Ex_tb = 1;
        torandom = {$random} % 10;
        if(torandom > 4)
            Ex_tb = {$random} % 2;
        goMain_tb = 1;
        torandom = {$random} % 10;
        if(torandom > 4)
            goMain_tb = {$random} % 2;

        #25;
    end

    $stop;
end

```

Finally, the DUT function is called to instantiate the design module and connect the input and output ports of the testbench and design modules.



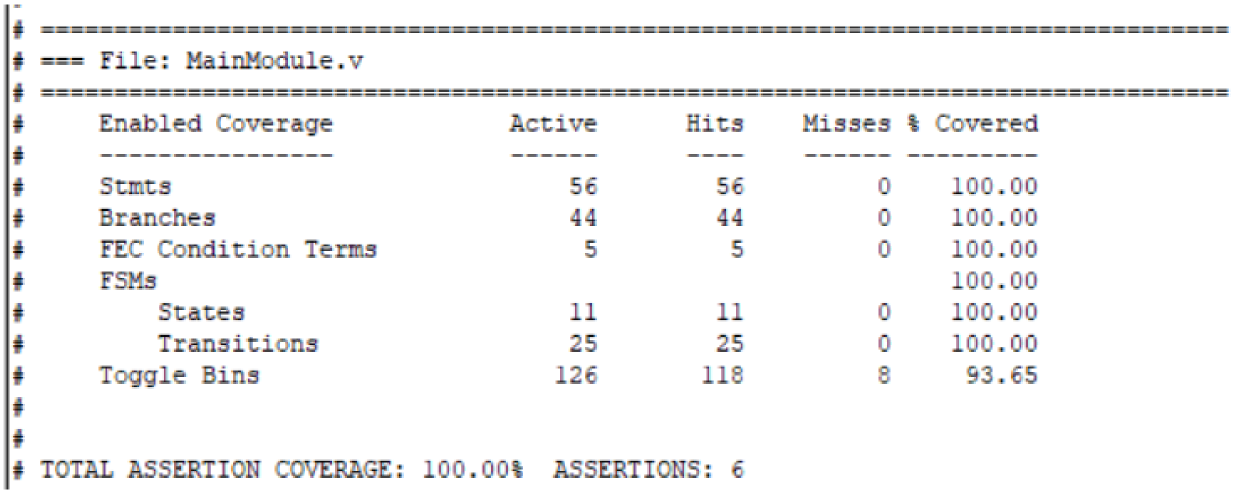
```
// instantiate design instance
MainModule DUT (
    .clk(clk),
    .rst(rst_tb),
    .Pin(Pin_tb),
    .WithDraw_Amount(WithDraw_Amount_tb),
    .Deposit_Amount(Deposit_Amount_tb),
    .Operation(Operation_tb),
    .IC(IC_tb),
    .LC(LC_tb),
    .Ex(Ex_tb),
    .goMain(goMain_tb),
    .FinalBalance(FinalBalance_tb),
    .CB(CB_tb)
);
endmodule

```

The function then closes the output file stream, completing the generation of the testbench file.

Testing

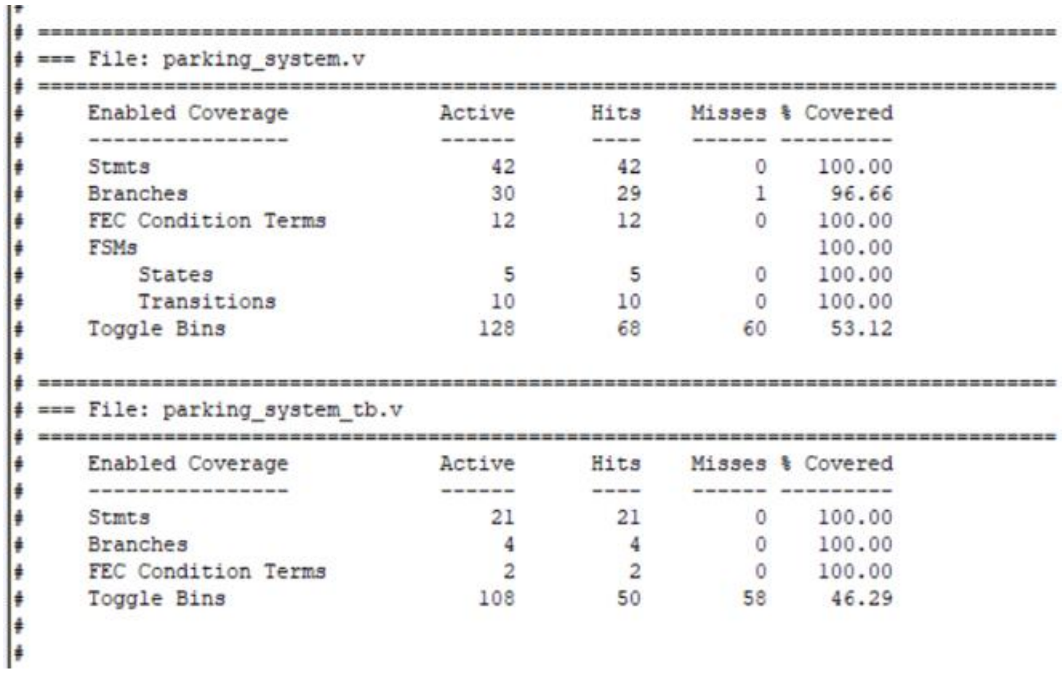
We tested the tool on three Verilog cases to verify its functionality and effectiveness. We set the number of iterations to be 20000 for the 3 tests and used Questa Sim to calculate the coverage:

Test 1 on the ATM_Module:



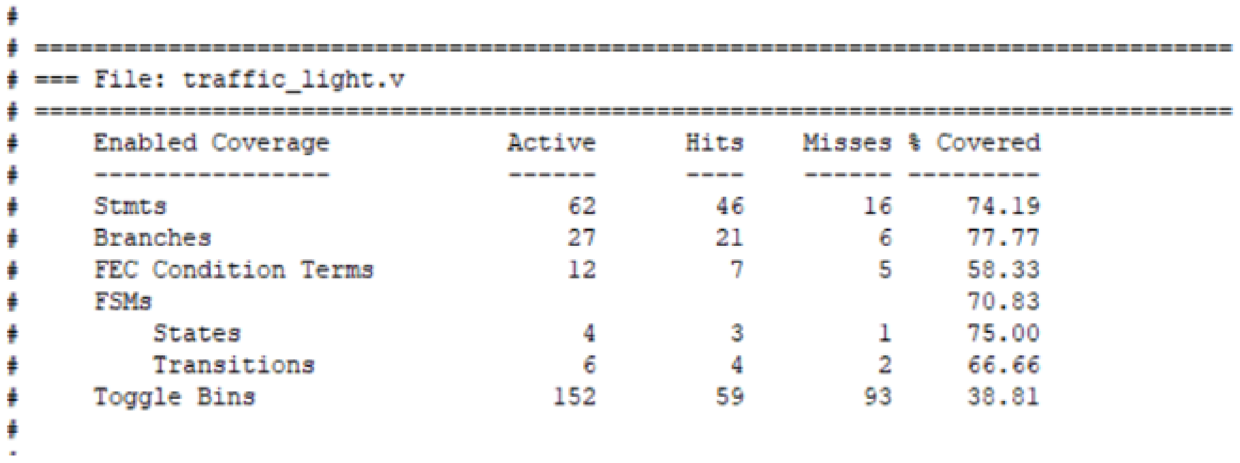
```
# ================================================================================
# === File: MainModule.v
# ================================================================================
#     Enabled Coverage            Active      Hits    Misses % Covered
#     ----------------            ------      ----    ------ ---------
#     Stmts                           56        56         0    100.00
#     Branches                        44        44         0    100.00
#     FEC Condition Terms              5         5         0    100.00
#     FSMs                                                      100.00
#         States                      11        11         0    100.00
#         Transitions                 25        25         0    100.00
#     Toggle Bins                    126       118         8     93.65
#
#
# TOTAL ASSERTION COVERAGE: 100.00%  ASSERTIONS: 6
```

Test 2 on the parking_system:



```
# ================================================================================
# === File: parking_system.v
# ================================================================================
#     Enabled Coverage            Active      Hits    Misses % Covered
#     ----------------            ------      ----    ------ ---------
#     Stmts                           42        42         0    100.00
#     Branches                        30        29         1     96.66
#     FEC Condition Terms             12        12         0    100.00
#     FSMs                                                      100.00
#         States                       5         5         0    100.00
#         Transitions                 10        10         0    100.00
#     Toggle Bins                    128        68        60     53.12
#
# ================================================================================
# === File: parking_system_tb.v
# ================================================================================
#     Enabled Coverage            Active      Hits    Misses % Covered
#     ----------------            ------      ----    ------ ---------
#     Stmts                           21        21         0    100.00
#     Branches                         4         4         0    100.00
#     FEC Condition Terms              2         2         0    100.00
#     Toggle Bins                    108        50        58     46.29
#
#
```

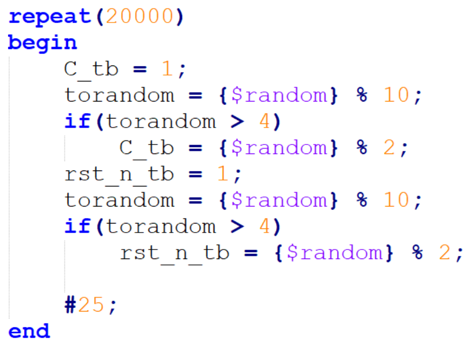
Test 3 on Traffic_light module:



```
#
# ==================================================================================
# === File: traffic_light.v
# ==================================================================================
#     Enabled Coverage              Active      Hits    Misses % Covered
#     ----------------              ------      ----    ------ ---------
#     Stmts                             62        46        16     74.19
#     Branches                          27        21         6     77.77
#     FEC Condition Terms               12         7         5     58.33
#     FSMs                                                          70.83
#         States                         4         3         1     75.00
#         Transitions                    6         4         2     66.66
#     Toggle Bins                      152        59        93     38.81
#
```

Limitations:

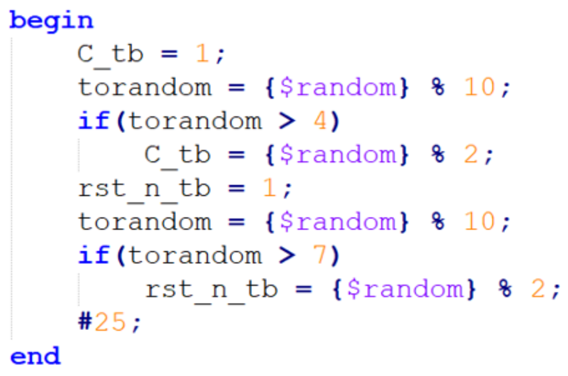
traffic light achieved low coverage, Although it has just two input ports. This is because we randomize any if statement with 50%.



```
repeat(20000)
begin
    C_tb = 1;
    torandom = {$random} % 10;
    if(torandom > 4)
        C_tb = {$random} % 2;
    rst_n_tb = 1;
    torandom = {$random} % 10;
    if(torandom > 4)
        rst_n_tb = {$random} % 2;

    #25;
end
```

To fix this issue, we will have to parse and build an Abstract Syntax tree and randomize with respect to the size of each sub-tree not with equal percentage.

when we randomized the rst with 80% to be true it results in 99% coverage with the same 20000 iteration.



```
begin
    C_tb = 1;
    torandom = {$random} % 10;
    if(torandom > 4)
        C_tb = {$random} % 2;
    rst_n_tb = 1;
    torandom = {$random} % 10;
    if(torandom > 7)
        rst_n_tb = {$random} % 2;
    #25;
end
```