Formal Verification & Property Based Verification

Formal Verification

- Process of constructing a proof that a target system will behave in accordance with its specification
- Mathematical reasoning to prove that an implementation satisfies a specification
- Prove: LHS = RHS

$(a+b)^2 = a^2+2ab+b^2$ ---- (1)

$(a+b)(a+b) = (a+b)a + (a+b)b$ ---- (2), Distributive

$(a+b)^2 = (a+b)a + (a+b)b$ ---- (3), Substitute 2 in 1

$(a+b)a = a.a+a.b$ ---- (4)

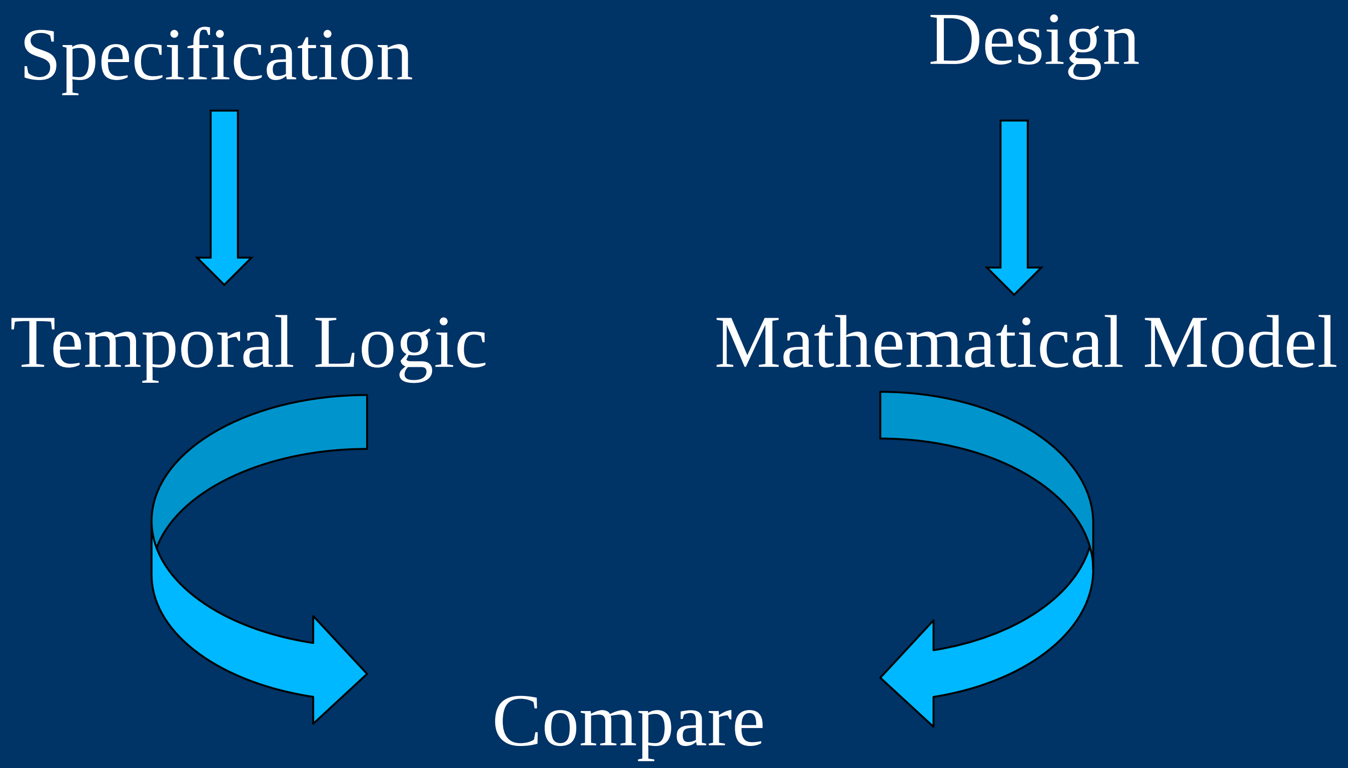
$(a+b)b = a.b+a.a$ ---- (5)

$(a+b)^2 = a.a+a.b + a.b+b.b$ ---- (6), Substitute 4 and 5 in 3

$(a+b)^2 = a^2+2ab+b^2$ ---- (7), Using definition of square and addition

Formal Verification

- Process of constructing a proof that a target system will behave in accordance



- Techniques
- Deductive Verification
  - Use of axioms
  - Time consuming and difficult
- Model Checking
  - Use of BDD
- Equivalence Checking
  - Check if 2 circuits are equivalent

What is being verified?

- Equivalence Checking
  - Mathematically proves that the origin and output are logically equivalent and that the transformation preserved its functionality
  - Compares two netlists
    - Ensure that scan chain insertion, clock-tree synthesis or manual modification, did not change the functionality of the circuit
  - Can also verify that two RTL descriptions are logically identical

What is being verified?

- Property Checking
  - Assertions - characteristics of a design are formally proven or disproved
  - Like, a particular state be reachable or not, if EN signal asserted then either STOP or VALID signal asserted eventually

Functional Verification

- Purpose is to ensure that a design implements intended functionality
- Can show that a design meets the intent of its specification, but it cannot prove it

Functional Verification Approaches

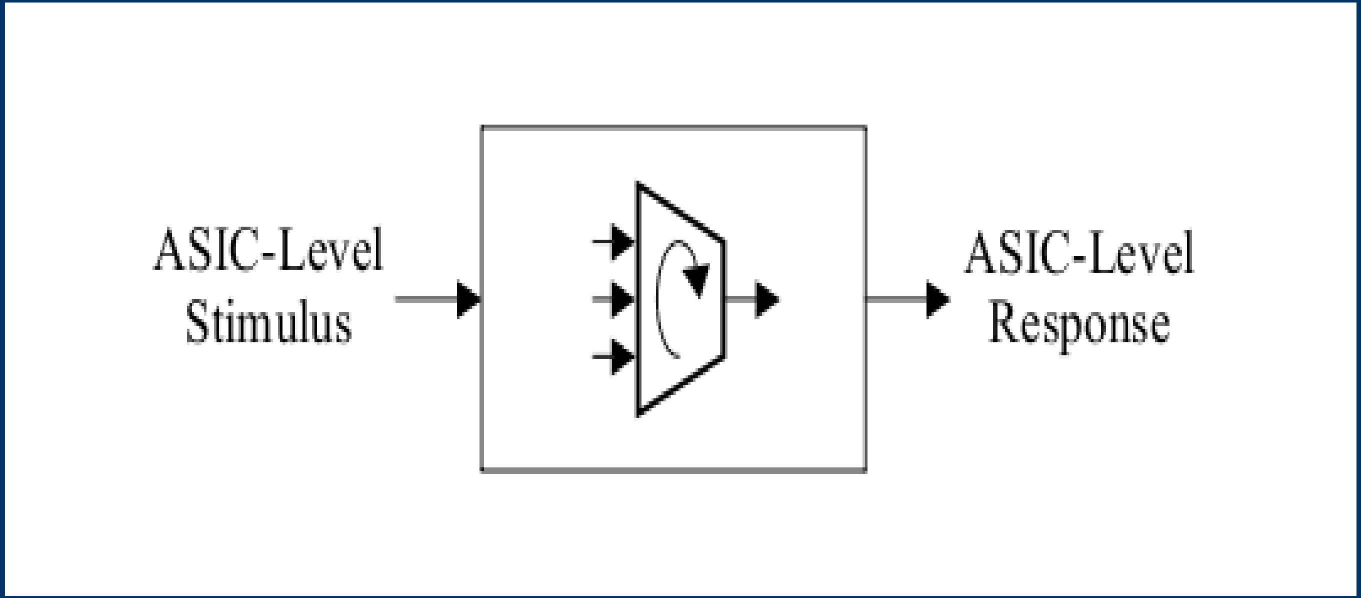
- FV can be accomplished using 3 complementary approaches

  a. Black Box

  b. White Box

  c. Grey Box

Black Box Verification

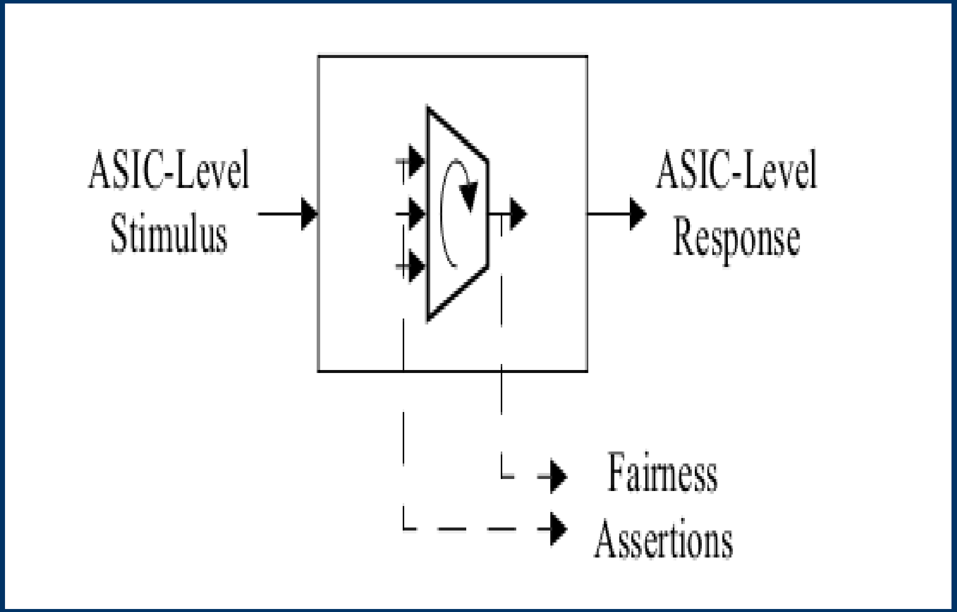


- Black box verification cannot look at or know about the inside of a design
- All the verification is accomplished using available interfaces
- Suffers from lack of visibility and controllability
- Difficult to locate the source of the problem
- Black Box testbenches can be used as Golden Testbenches

Grey Box Verification

- Grey box approach controls and observes a design entirely through its top level interfaces.
- A typical Grey box strategy is to include some non functional modification to provide additional visibility and controllability.
- Addition of observability or controllability features for the design is called Design for Verification
- Grey and Black box verification can be done in parallel with design

White Box Verification



- White box verification has intimate knowledge and control of the internals of design
- This is tied up to a specific implementation
- Used to verify the correctness of the functionality

Verification Technologies

Reference: Writing Test Benches using SystemVerilog

Verification Technologies

- Linting
- Simulation
  - *Stimulus and Response*
  - *Event-Driven Simulation*
  - *Cycle-Based Simulation*
  - *Co-Simulators*
- Verification Intellectual Property
- Code Coverage
  - *Block Coverage*
  - *Branch Coverage*
  - *Statement Coverage*
  - *Path Coverage*
  - *Expression Coverage*
  - *FSM Coverage*

Linting

- Finds common programmer mistakes
- Does not require stimulus
- Only identify a certain class of problems
- Lint the code as it is being written
- Can detect race conditions

Example

```
module saturated_counter(output done, input rst, input clk);
        byte counter;

always_ff @(posedge clk)
        begin
            if (rst) counter <= 0;
            else if (counter < 255) counter++;
        end
        assign done = (counter == 255);
endmodule
```

Example

```
begin
    integer i;
    ...
    fork
    i = 1;
    i = 0;
    join
    ...
end
```

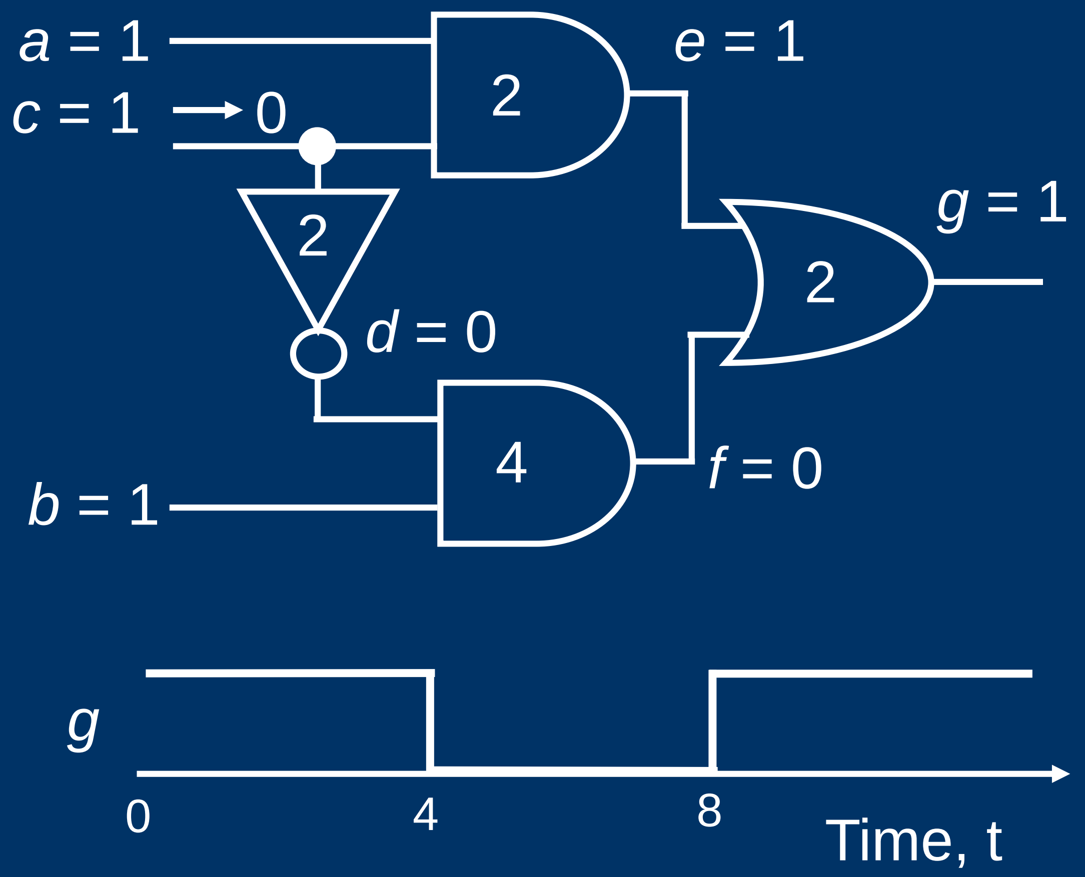
Simulation

- Most common and familiar verification technology
- Allows the designers interact with the design before it is manufactured, and correct flaws and problems earlier
- A simulation is never the final goal of a project
- Goal of all hardware design projects to create real physical designs that can be sold and generate profits

Simulation

- Stimulus and Response
- Event Driven and Cycle based simulation
- Co-simulators
  - Most of the cycle based simulators are integrated with the event driven simulators

Event Driven Simulation (Example)

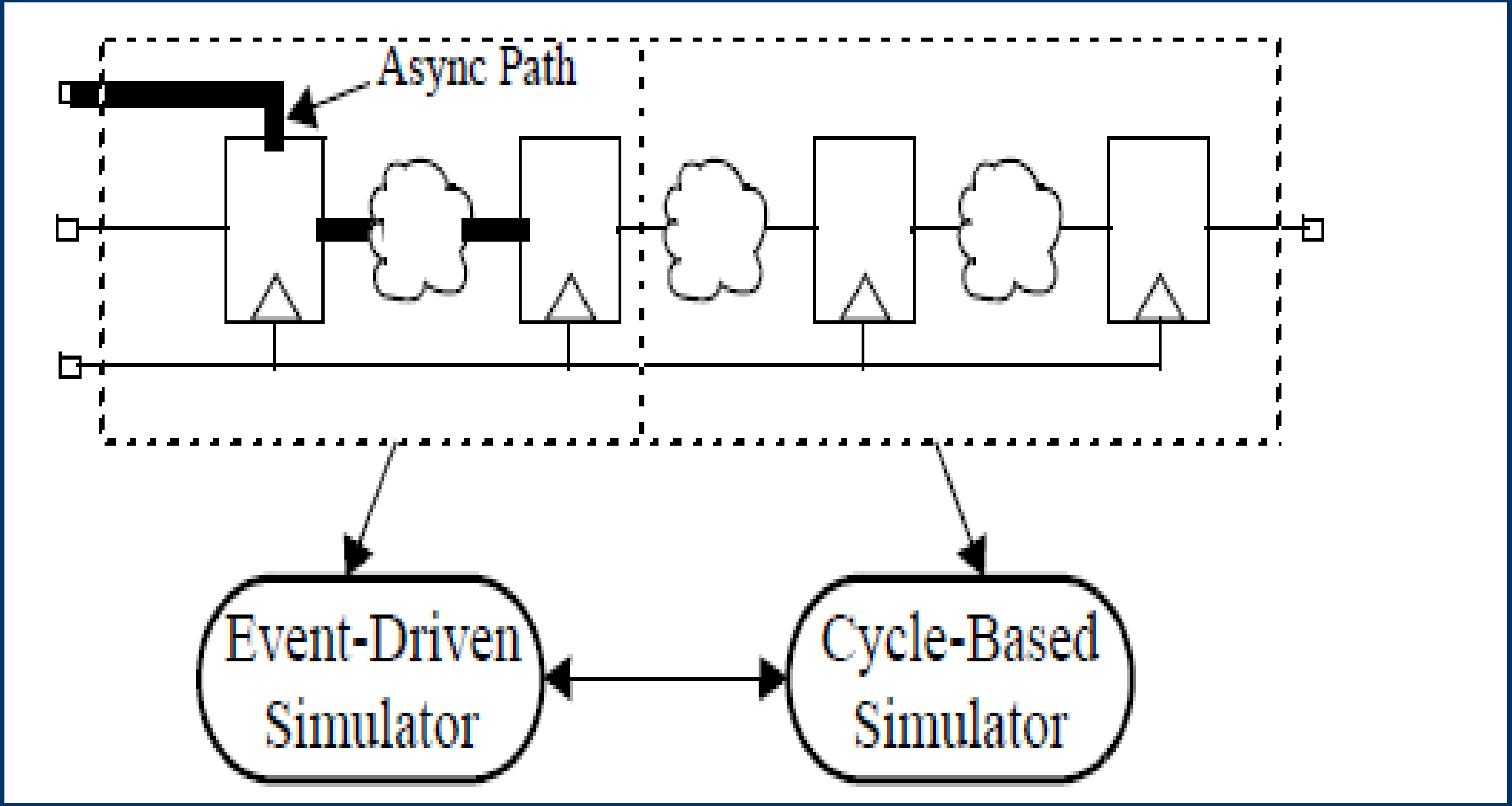


Time stack	Scheduled events	Activity list
t = 0	$c = 0$	d, e
1		
2	$d = 1, e = 0$	f, g
3		
4	$g = 0$	
5		
6	$f = 1$	
7		g
8	$g = 1$	

Cycle-Based Simulation

- Cycle simulation is a technique for simulating circuits that do not take into account the detailed circuit timing
- Cycle simulation computes the steady state response of the circuit at each clock cycle boundary
- Need for another tool for timing analysis

Simulation



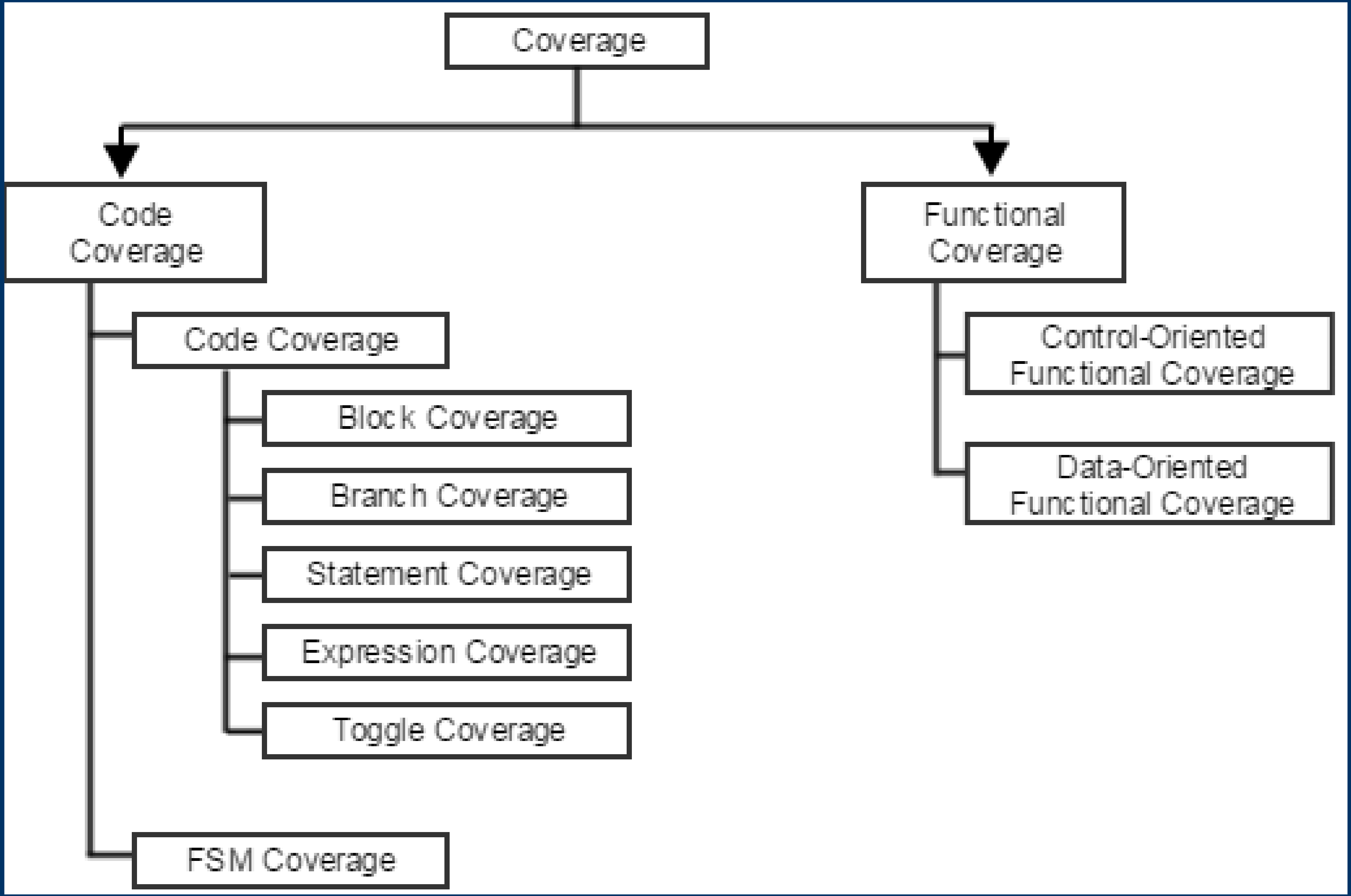
Verification Intellectual Property

- If you want to verify your design, it is necessary to have models for all the parts included in a simulation
- Models for RAMs , Standard Interfaces and so on
  - Could contain stimulus sequences, coverage model, and a particular block in the design, such as a USB interface
- It is cheaper to buy models than write them yourself

Verification Technologies

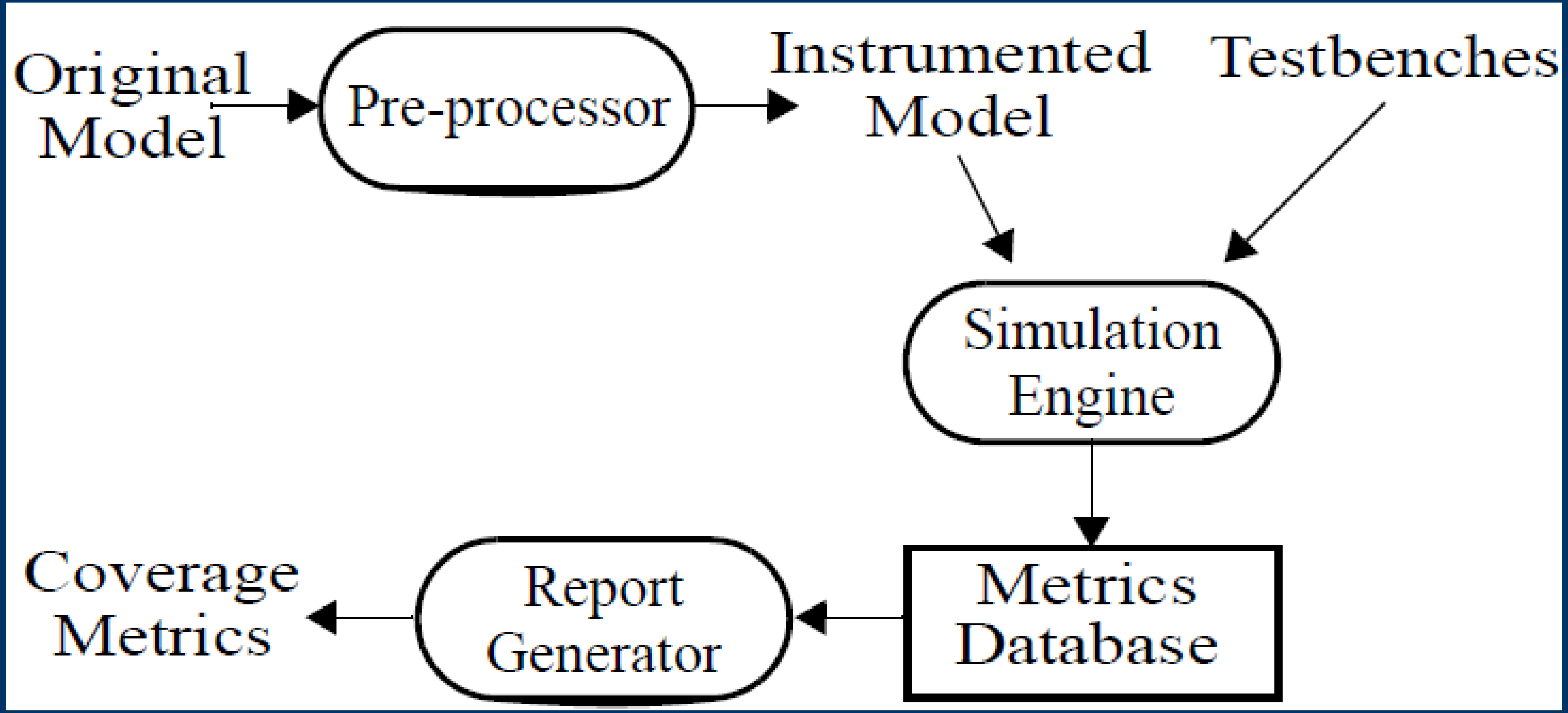
- Linting
- Simulation
  - *Stimulus and Response*
  - *Event-Driven Simulation*
  - *Cycle-Based Simulation*
  - *Co-Simulators*
- Verification Intellectual Property
- Code Coverage
  - *Block Coverage*
  - *Branch Coverage*
  - *Statement Coverage*
  - *Path Coverage*
  - *Expression Coverage*
  - *FSM Coverage*

Overview of Coverage Types



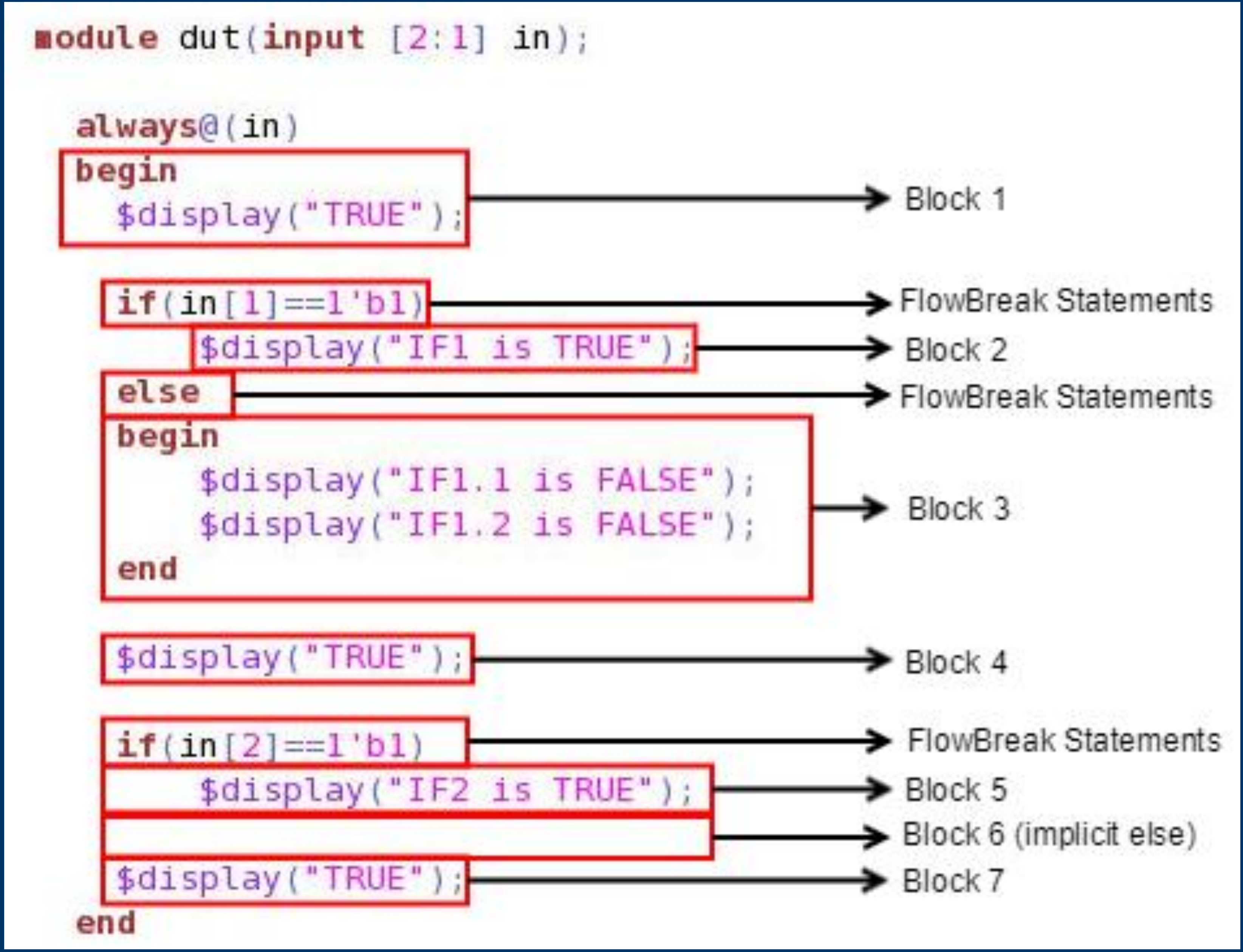
Code Coverage

- Identify the portion of the code has been/not been executed in the DUV
- Are there any sections of the RTL code not triggered a functional error?
- Tool adds checkpoints at strategic locations of the source code



Block Coverage

- Example of a Block
- Block 1 starts with the begin statement and ends when if is encountered
- After if, a new block "Block 2" starts and it ends when else is encountered
- Block 3 starts with the begin statement and ends with an end statement as there is no flowbreak statement
- Both Block 4 and Block 5 end with if
- Block 6 ends with an implicit else



```
module dut(input [2:1] in);

 always@(in)
 begin
   $display("TRUE");

  if(in[1]==1'b1)
      $display("IF1 is TRUE");
  else
  begin
      $display("IF1.1 is FALSE");
      $display("IF1.2 is FALSE");
  end

  $display("TRUE");

  if(in[2]==1'b1)
      $display("IF2 is TRUE");

  $display("TRUE");
 end
```

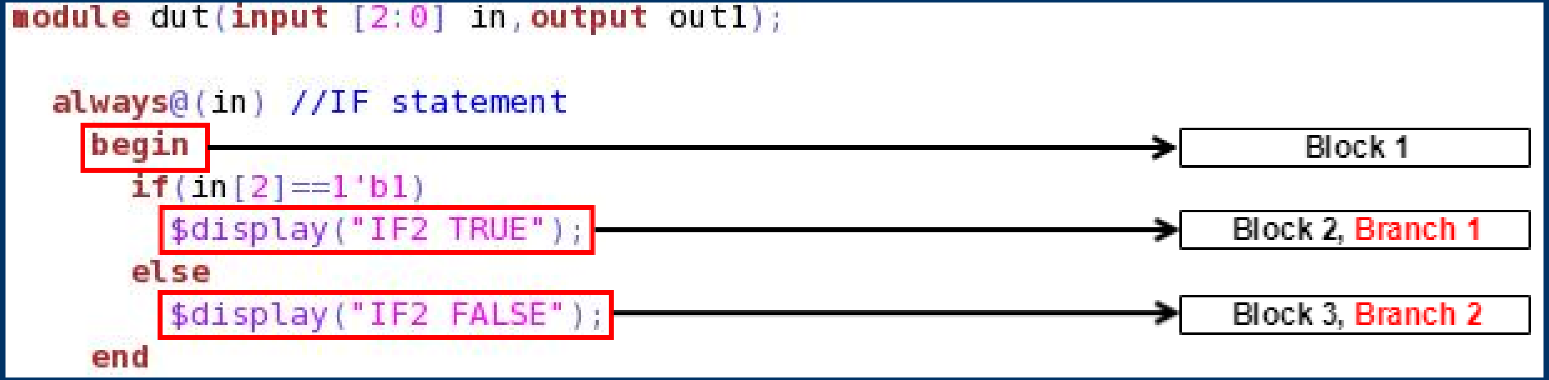
Block Coverage

- Limitations
- Blocks are not scored for concurrent assignment statements
- Blocks are not defined for primitives
- Blocks are not scored for a function defined inside a package when the function is called from inside a generate block (VHDL)

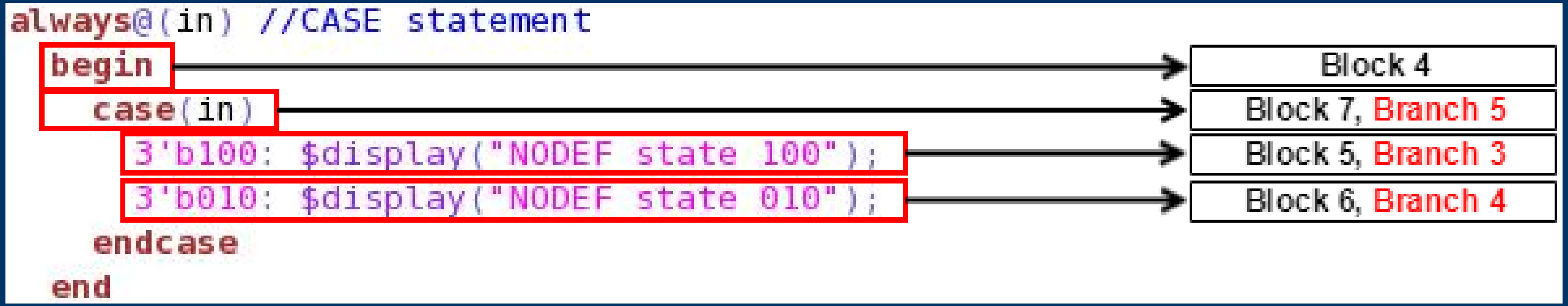
Branch Coverage

- Complements block coverage by providing more precise coverage results for reporting coverage numbers for various branches individually
- A piece of code is considered 100% covered when each branch of a conditional statement has been executed
- A branch is a statement that is executed based on evaluation of a condition used in a conditional statement. Ex:
  - assign out2 = (cond1)? 1'b1:1'b0;
    - If cond1 evaluates to true, out2 is assigned 1’b1.
    - If cond1 evaluates to false, out2 is assigned 1'b0.
    - In block coverage, this statement is considered a single block and is considered covered if cond1 is either true or false.
    - With branch coverage, this statement is considered 100% covered when each branch of the conditional statement has executed.
- if, else if, else, and implicit else (also scored as blocks),
- case blocks with implicit or explicit defaults (also scored as blocks), Separate block or branch is formed corresponding to each case item expression
- Conditional operator's expressions

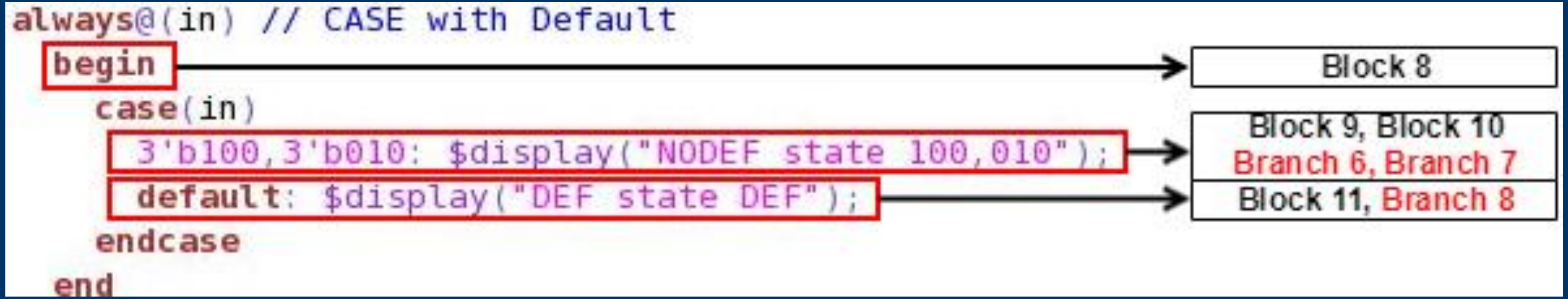
Branch Coverage - Example



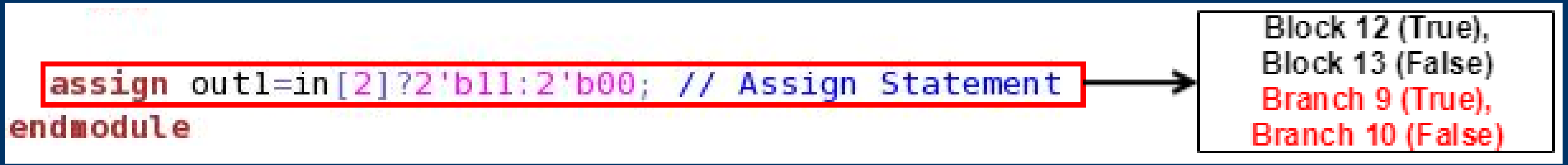
```
module dut(input [2:0] in,output out1);

  always@(in) //IF statement
    begin
      if(in[2]==1'b1)
        $display("IF2 TRUE");
      else
        $display("IF2 FALSE");
    end
```



```
always@(in) //CASE statement
  begin
    case(in)
      3'b100: $display("NODEF state 100");
      3'b010: $display("NODEF state 010");
    endcase
  end
```



```
always@(in) // CASE with Default
  begin
    case(in)
      3'b100,3'b010: $display("NODEF state 100,010");
      default: $display("DEF state DEF");
    endcase
  end
```



```
  assign out1=in[2]?2'b11:2'b00; // Assign Statement
endmodule
```

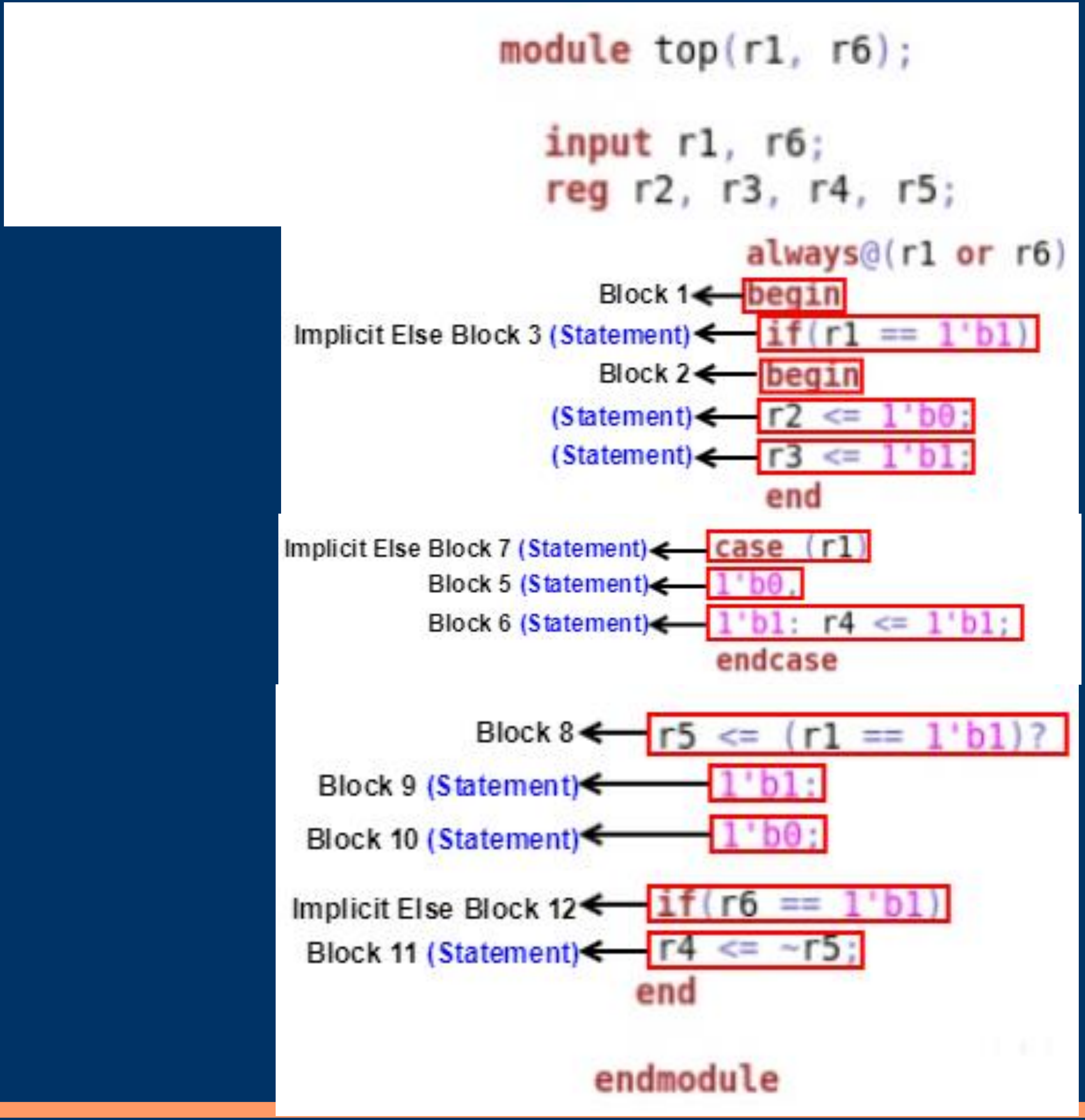
Statement Coverage

- Also called as block coverage
- Statement, line or block coverage measures how much of the total lines of code were executed by the verification suite

Statement Coverage

- An unverified block with more statements is likely to have more errors than the one with a single statement
  - Ensure verification completeness by prioritizing blocks based on the number of statements in a block.
- Statement coverage is used to ascertain the number of statements in each block and ensure that each statement is executed at least once.
- By default, a block coverage report does not include information on the number of statements within a block.
  - Include this information in a block coverage report using the set_statement_scoring command in the coverage configuration file during elaboration.

Statement Coverage



```
module top(r1, r6);

  input r1, r6;
  reg r2, r3, r4, r5;

      always@(r1 or r6)
      begin
       if(r1 == 1'b1)
       begin
       r2 <= 1'b0;
       r3 <= 1'b1;
       end
     case (r1)
     1'b0,
     1'b1: r4 <= 1'b1;
     endcase

   r5 <= (r1 == 1'b1)?
         1'b1:
         1'b0;

   if(r6 == 1'b1)
   r4 <= ~r5;
   end

  endmodule
```

Statement Coverage

- The block named *acked* is executed entirely whenever the expression in the if statement evaluates to true.
- Counting the execution of that block is equivalent to counting the execution of the four individual statements within that block.

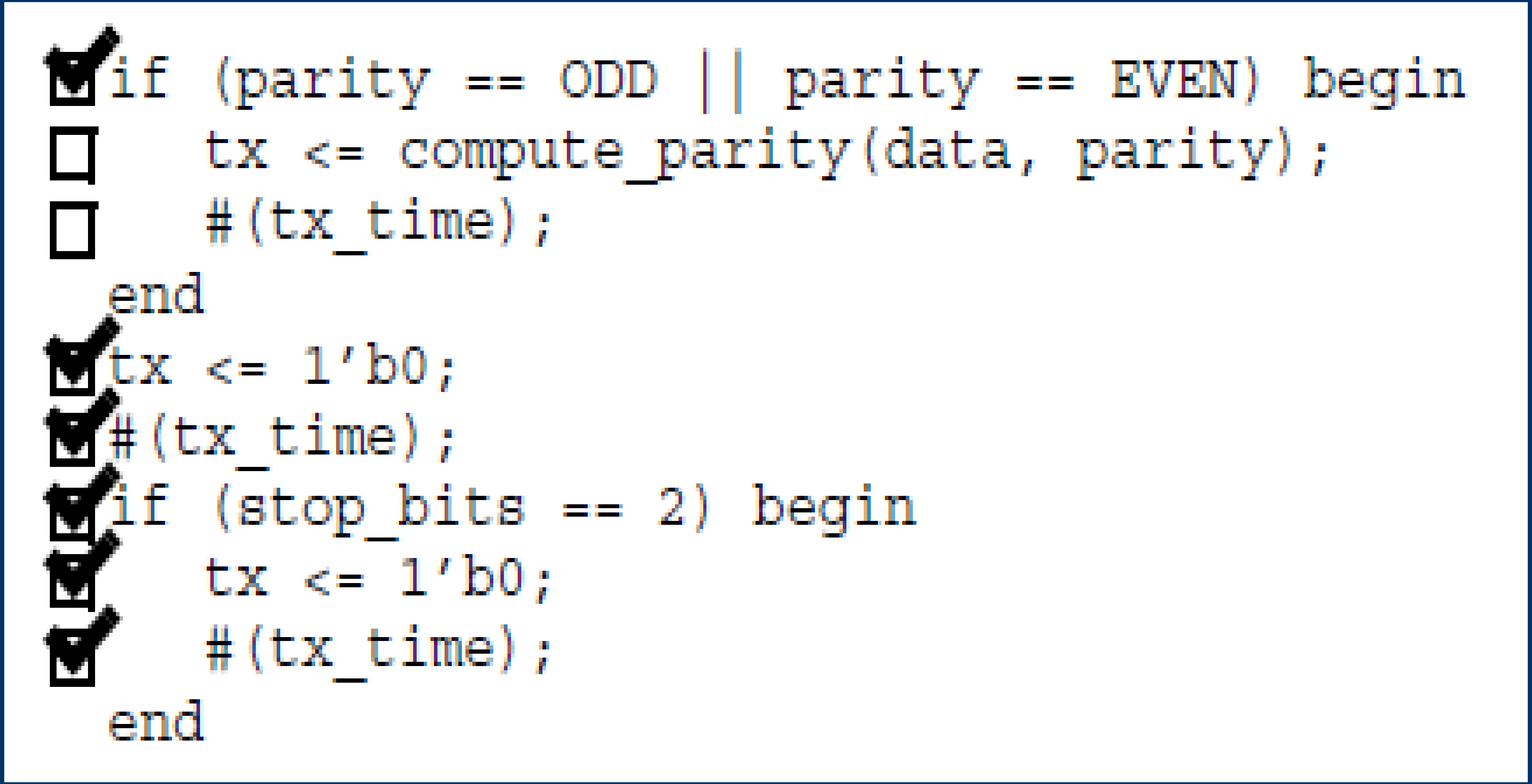
```
if (dtack == 1’b1)
   begin: acked
            as        <= 1’b0;
            data      <= 16’hZZZZ;
            bus_rq    <= 1’b0;
            state     <= IDLE;
    end: acked
```

Statement Coverage

- Subsequent sequential statements after wait statement may not have executed - form a separate statement block.

```
address      <= 16’hFFED;
ale          <= 1’b1;
rw           <= 1’b1;
wait (dtack == 1’b1);
 read_data = data;
 ale        <= 1’b0;
```

Statement Coverage



```
☑ if (parity == ODD || parity == EVEN) begin
☐      tx <= compute_parity(data, parity);
☐      #(tx_time);
   end
☑ tx <= 1'b0;
☑ #(tx_time);
☑ if (stop_bits == 2) begin
☑      tx <= 1'b0;
☑      #(tx_time);
   end
```

Statement Coverage

- Once the conditions have been determined

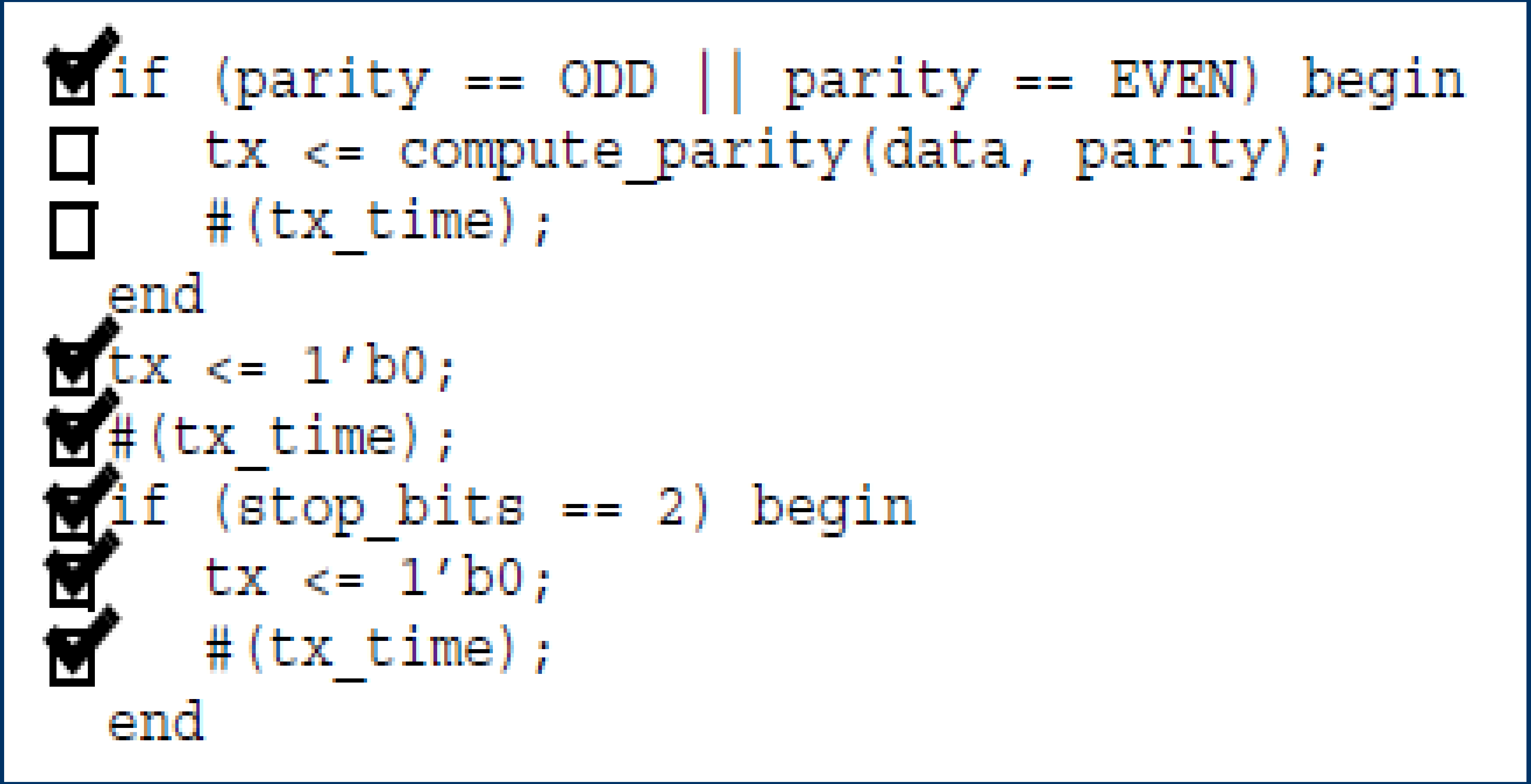
1. One must understand why they never occurred in the first place.
2. Is it a condition that can never occur?
3. Is it a condition that should have been verified by the existing verification suite? (OR)
4. Is it a condition that was forgotten?

Path Coverage

- Path coverage measures all possible ways one can execute a sequence of statements.
- Number of paths will depend on the control-flow Statements

Path Coverage

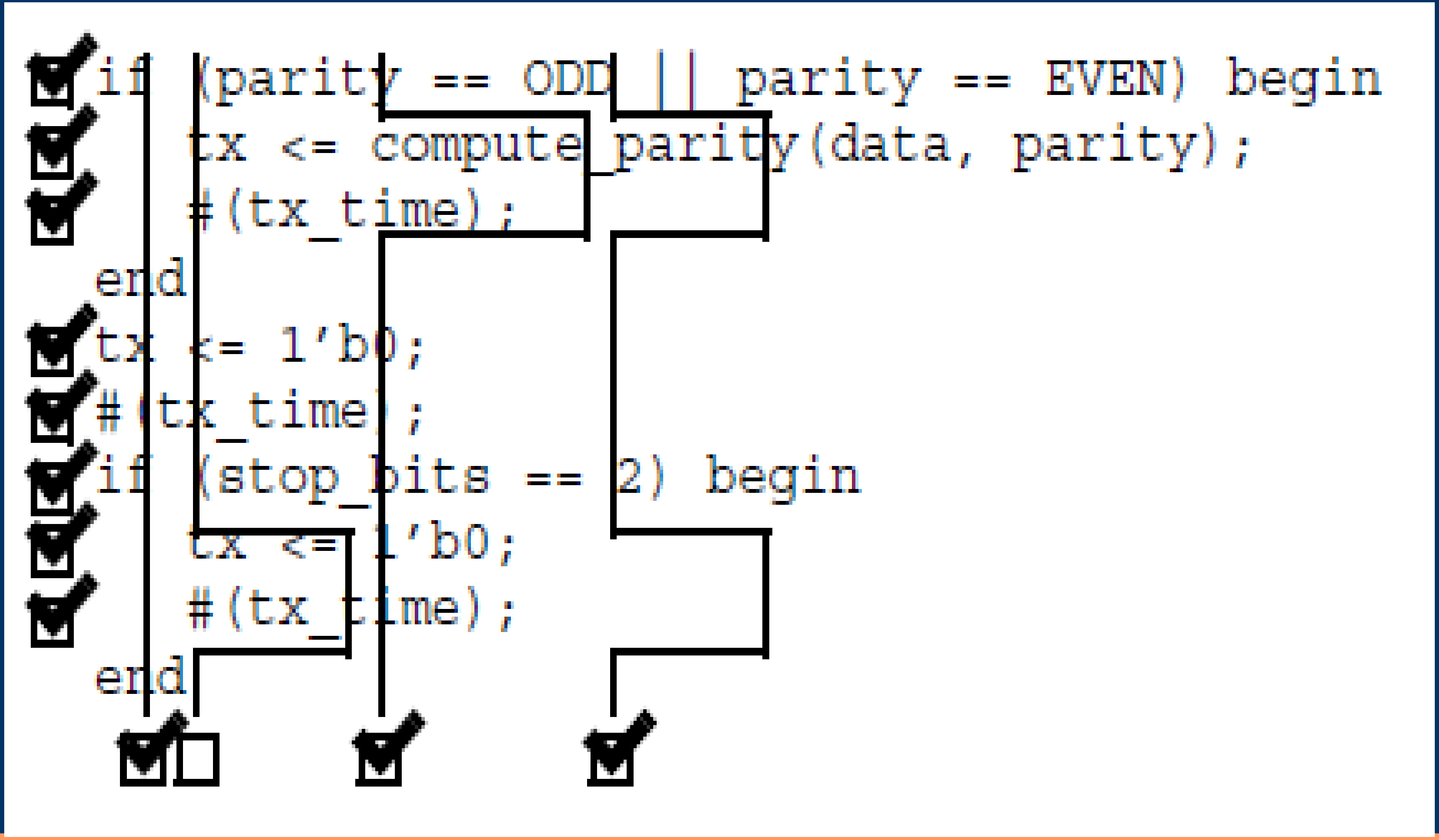
- How many paths are there??



```
☑ if (parity == ODD || parity == EVEN) begin
☐     tx <= compute_parity(data, parity);
☐     #(tx_time);
  end
☑ tx <= 1'b0;
☑ #(tx_time);
☑ if (stop_bits == 2) begin
☑     tx <= 1'b0;
☑     #(tx_time);
  end
```

Path Coverage

- Answer: 4 paths



```
if (parity == ODD || parity == EVEN) begin
    tx <= compute_parity(data, parity);
    #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
    tx <= 1'b0;
    #(tx_time);
end
```

Path Coverage

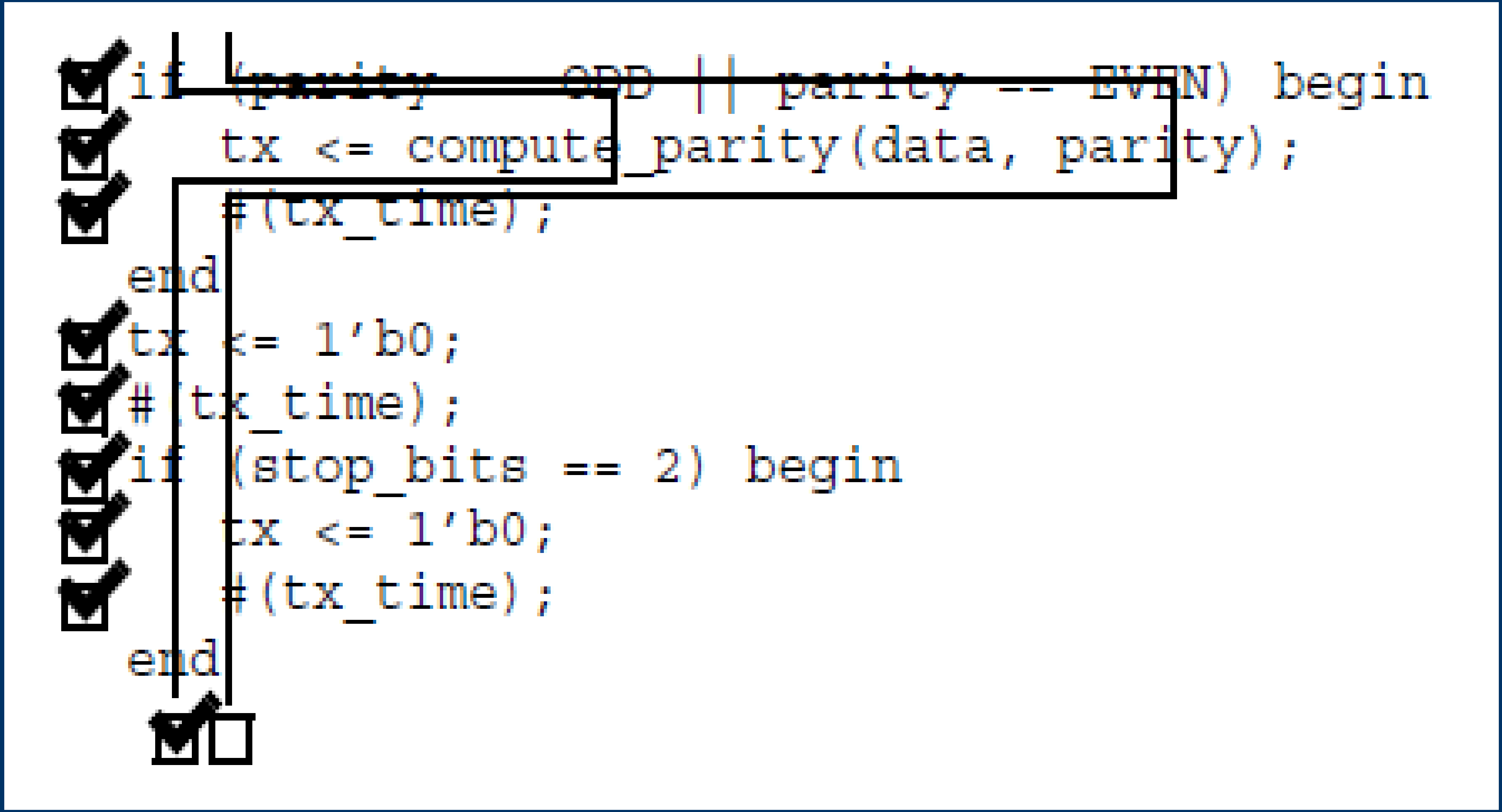
- The number of paths in a sequence of statements grows exponentially with the number of control-flow statements.
- Code coverage tools give up measuring path coverage if their number is too large in a given code sequence.
- To avoid this situation, keep all sequential code constructs (*always and initial blocks*, *tasks and functions*) to under 100 lines.

Reaching 100 percent path coverage is very difficult

Expression Coverage

- Is a mechanism that factorizes logical expressions and monitors them during simulation run.
- It measures how thoroughly a testbench exercises expressions in assignments and procedural control constructs (if/case conditions).
- This is done by identifying each input condition that makes the expression true or false, and further checking whether that condition happened during simulation.
- Expression coverage provides a finer granularity of coverage metrics than other code coverages, such as block and branch coverage.
- Therefore, it is standard practice to define high goals for expression coverage

Expression Coverage

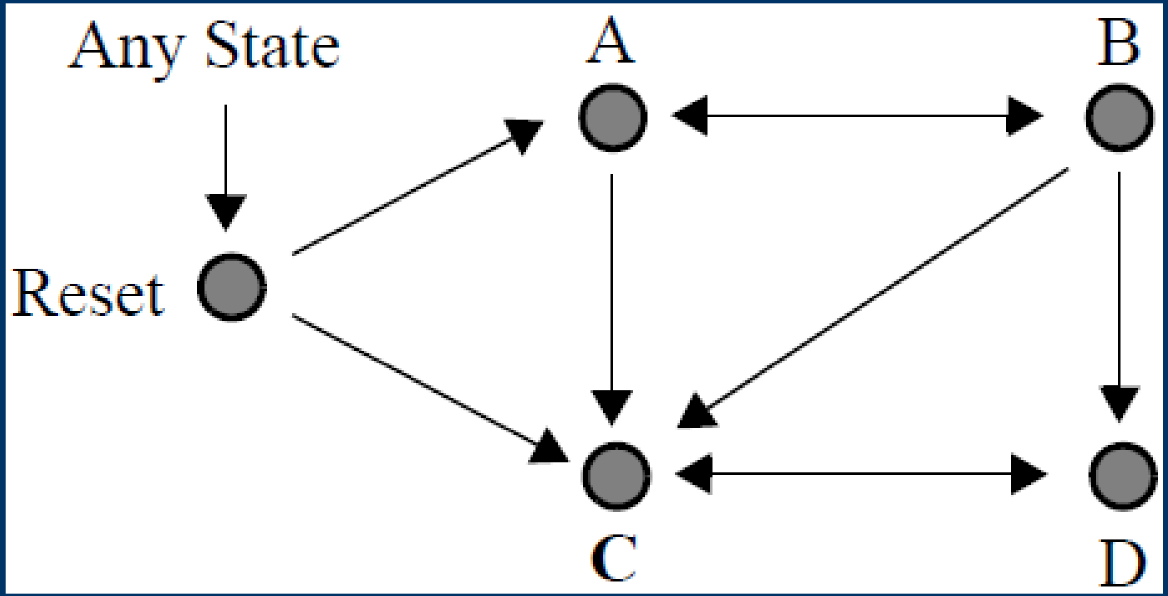


```
if (parity == ODD || parity == EVEN) begin
    tx <= compute_parity(data, parity);
    #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
    tx <= 1'b0;
    #(tx_time);
end
```

Reaching 100 percent expression coverage is very difficult

FSM Coverage

- FSM is explicitly coded using a choice in a *case* statement
- The state corresponding to an uncovered *case* statement choice is not visited during verification



- Fourteen possible transition
- State coverage will be 100% if transition occurs from Reset ->A->B->D->C
- Transition coverage, 5/14 = 0.36

FSM Coverage

- Statement coverage detects unvisited states.
- FSM coverage identifies state transitions.
- FSM coverage cannot identify unintended or missing transitions.
- Formal verification may be better suited for FSM verification

Functional Coverage

- Ensure that a bad design is not hiding behind passing testbenches
- Code coverage - how much of the implementation has been exercised
- Functional coverage - how much of the original design specification has been exercised
- Functional coverage records relevant metrics (e.g., packet length, instruction opcode, buffer occupancy level)

Functional Coverage

```
enum {ADD, SUB, JMP, RTS, NOP} opcode;
...
    case (opcode)
        ADD: ...
        SUB: ...
        JMP: ...
        default: ...
    endcase
```

- Code coverage might be 100% but functional coverage is not 100%
- Functional coverage does not rely on actual code.
- It will report gaps in the recorded values whether the code to process them is there or not

Transition Coverage

- Transition coverage helps answer questions like
  - “Did I perform all combinations of back-to-back read and write cycles?”
  - “Did we execute all combinations of arithmetic opcodes followed by test opcodes?”
  - “Did this state machine traverse all significant paths?”