VLSI DESIGN FLOW: RTL TO GDS

Lecture 11
Functional Verification using Simulation

Sneh Saurabh
Electronics and Communications Engineering
IIIT Delhi

Lecture Plan

Functional Verification using Simulation

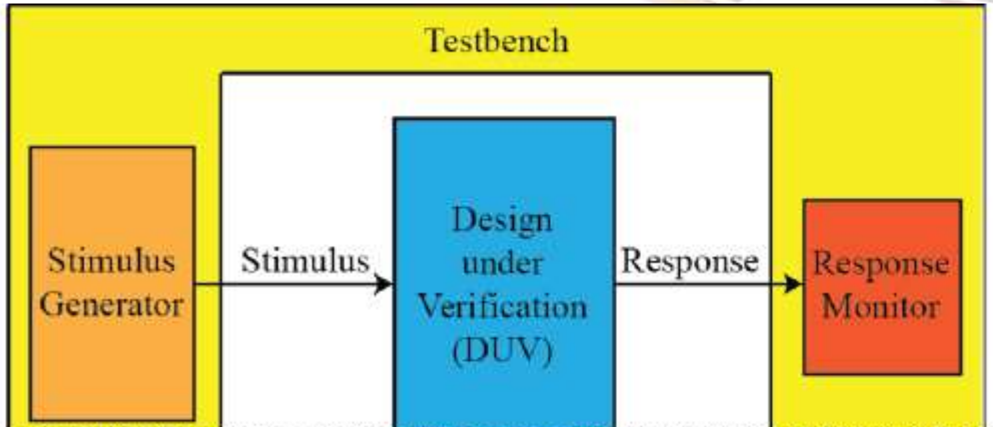
- Framework of Simulation
- Testbench
- Coverage
- Types of Simulators
- Mechanism of Simulation

Simulation

Framework

Simulation: Framework

Simulation is performed to verify that the output generated by the design is in agreement with the desired functionality for the given test stimuli

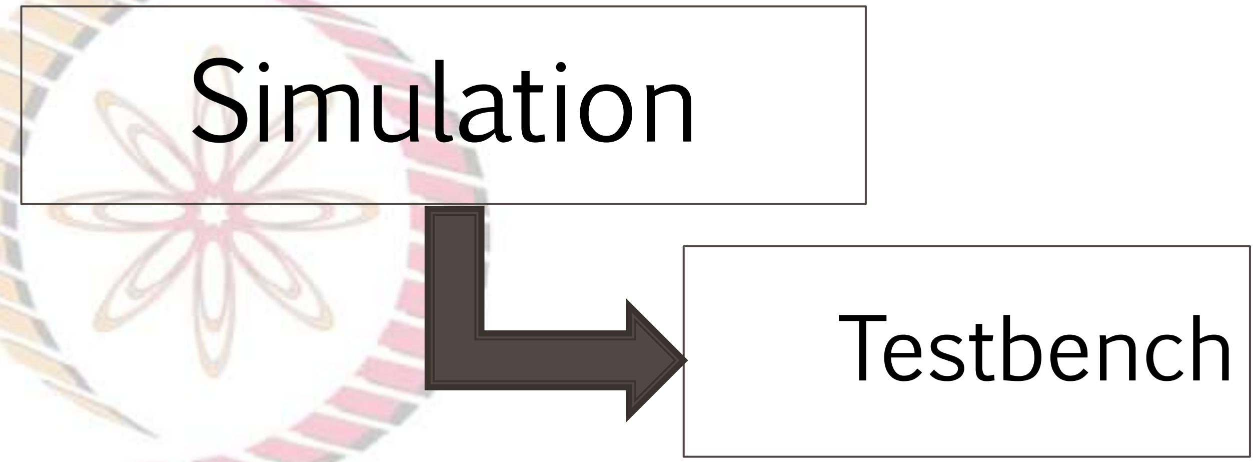


J. Bergeron, "Writing Testbenches: Functional Verification of HDL Models", *Springer Science & Business Media*, 2012.
S. Saurabh, "Introduction to VLSI Design Flow". Cambridge: *Cambridge University Press*, 2023.

Testbench

- Interacts with simulator
- Applies stimulus
- Observes response

Robustness of verification strongly depends on the testbench


Simulation
Testbench

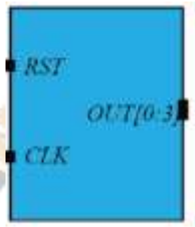
Testbench: Illustration (1)

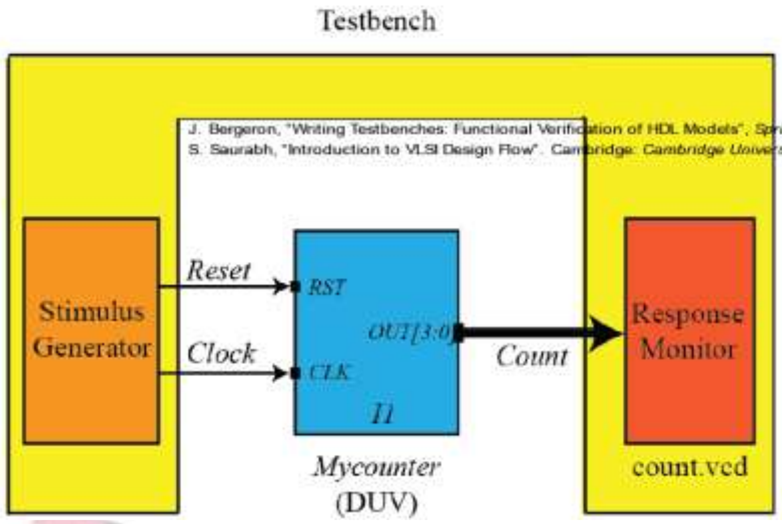
```
module Mycounter(CLK, RST, OUT);

   input CLK, RST;
   output [3:0]OUT;
   reg [3:0]OUT;

   always @ (posedge CLK)
   begin
      if (RST == 1'b1)
            OUT  <= 4'b0000;
      else
            OUT  <= OUT + 1;
   end

endmodule
```

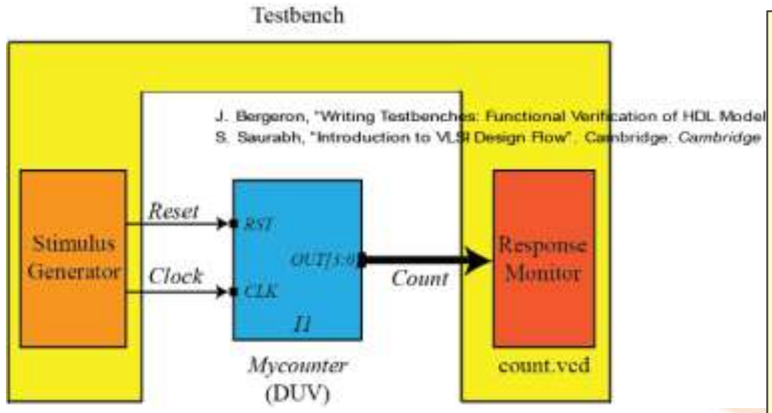




J. Bergeron, "Writing Testbenches: Functional Verification of HDL Models", Springer Science & Business Media, 2012.
S. Saurabh, "Introduction to VLSI Design Flow". Cambridge: Cambridge University Press, 2023.

- Instantiate the DUT
- Generate test signals
- Monitor the signals and save it in a file

Testbench: Illustration (2)



J. Bergeron, "Writing Testbenches: Functional Verification of HDL Model
S. Saurabh, "Introduction to VLSI Design Flow". Cambridge: Cambridge

- Instantiate the DUT
- Generate test signals
- Monitor the signals and save it in a file

```
module Testbench();
   reg Clock, Reset;
   wire [3:0] Count;

   // instantiate the DUV and make connections
   Mycounter I1(.CLK(Clock), .RST(Reset), .OUT(Count));
   // initialize the testbench
   initial begin
         Clock = 1'b0; Reset = 1'b1; // reset the counter at t=0
         #100 Reset =1'b0; // remove reset at t=100
         #2000 $finish; // end the simulation after t=2000
   end
   // generate stimulus (in this case clock signal)
   always #50 Clock = ~Clock; // clock period = 100
   // monitor the response and save it in a file
   initial begin
         $dumpfile ("count.vcd"); // specifies the VCD file
         $dumpvars; // dumps all variables
         $monitor("%d,%b,%b,%d",$time, Clock, Reset, Count);
   end

endmodule
```

Code Coverage

A good testbench:

- Performs comprehensive design verification without wasting resources on dispensable tasks
- Contains a sufficiently diversified set of stimuli
  - Adding more test stimuli for already tested features may be redundant

Code Coverage:

- Identifies sections of DUV's source code that did not execute during verification.
- Helps monitor the progress of verification effort

- **Line coverage:** number of times each RTL statement is executed during simulation
- **Branch coverage:** whether all branches of the code (as in if–else and case statements) were exercised in simulation

- **State coverage:** whether all the states of an FSM have been activated and all the state transitions traversed.
- **Toggle coverage:** whether all variables or bits in variables have risen and fallen.

Code Coverage vs. Functional Coverage

Code coverage:

- Measures coverage of an existing code
- An empty module will have a 100% code coverage, while complete functionality is missing

Functional coverage:

- Measures whether the specified design features have been exercised during simulation
- We need to provide design features using coverage model
  - Not automatically inferred by the tool
- Coverage model is independent of the design implementation.
  - Can detect missing features also in the implementation

Simulation in Verilog

$\rightarrow$ Mechanism

Verilog Simulation: Definitions (1)

Verilog considers a design as consisting of connected processes.

Processes: design objects that a simulator can evaluate and produce a response to a given stimulus.

Examples: gate primitives, initial block, always block, continuous assignment, and procedural assignment statements.

- **Event:** Anything that requires simulator to take some action.
- Events are of two types:
  1. **Update Event**: change in the value of a net or a variable.
  2. **Evaluation Event**: evaluation of processes when an update event has occurred

```
module top(in1, in2, out1, out2);
        input in1, in2;
        output out1, out2;
        reg out1, tt;

        and A1(out2, tt, in1); // gate primitive

        assign out1 = tt; // contn. assignment

        always @(in1 or in2) // always block
        begin
                tt = in1 & in2;
        end
endmodule
```

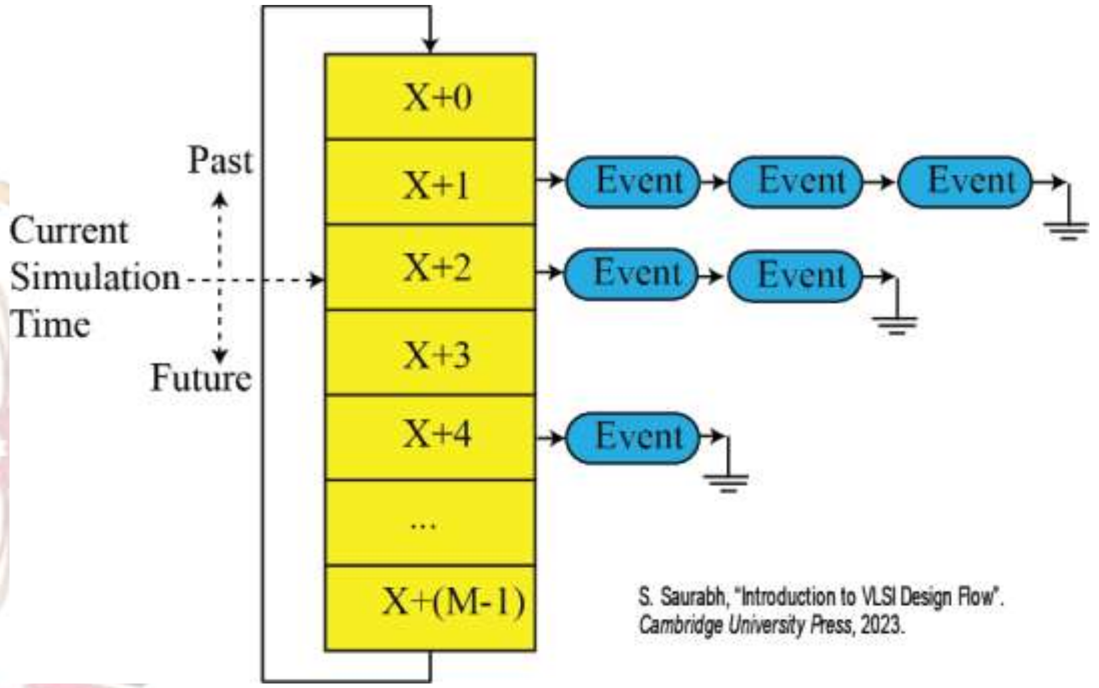
S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

- Processes: gate instance, continuous assignment, always block
- Assume: *in1* makes a 0→1 transition
  - Update event: for the net *in1*
  - Evaluation event: in gate instance and always block (because sensitive)

Verilog Simulation: Definitions (2)

- **Simulation time:** time value maintained by a simulator to model the actual time in the simulated circuit.
  - ➢ Simulation time depends on test-bench and the design being simulated
  - ➢ Simulation time is different from the time taken by the simulator to simulate.

- Events have simulation time associated with it
- Simulators need to ensure events are processed in the increasing order of time
- **Event Queue**: internal queue maintained by a simulator ordered by simulation time



S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

- **Timing wheel**: an efficient data structure to implement event queue
- An array of M slots indexed by simulation time $T\%M$

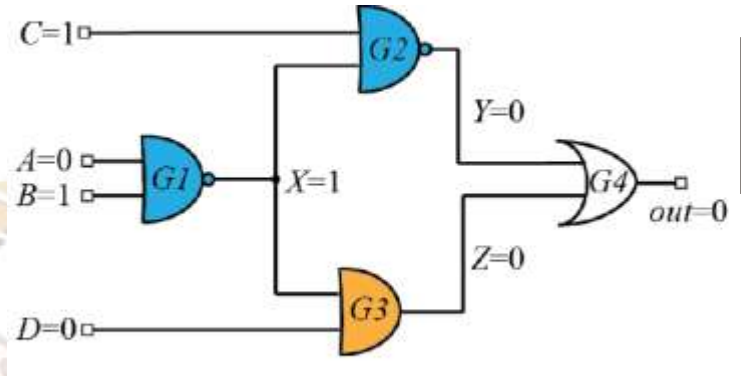
Processing of Events

- Applying stimulus creates update events
- Update events can trigger evaluation events
- Evaluation events can further change values of nets/signals and lead to new update events
- Sequence of:

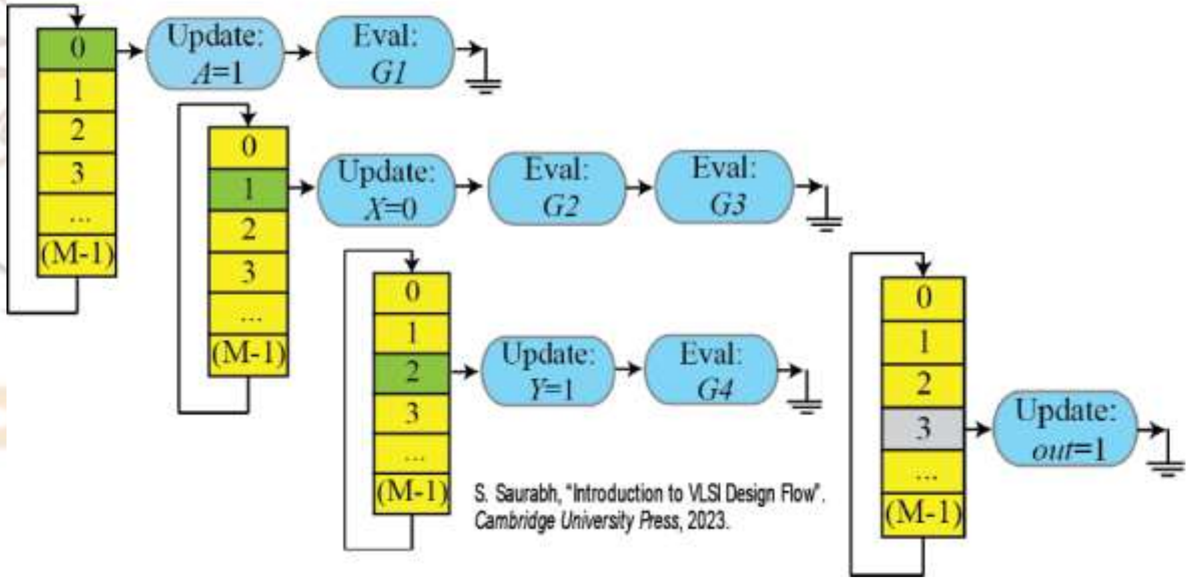
update → evaluate → update → evaluate …

goes on until the end of simulation time

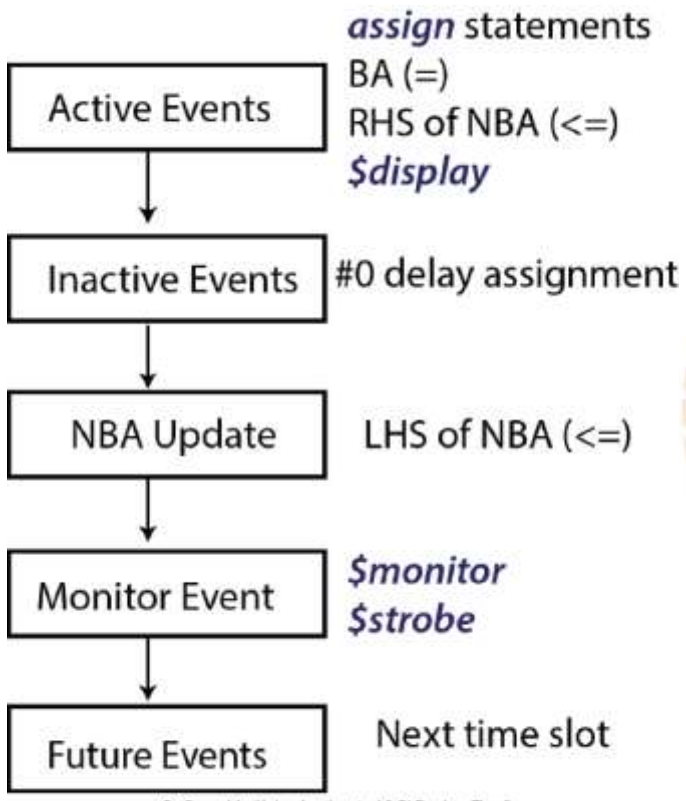
Assume that the circuit is as shown above. Assume that delay of each gate is 1 time unit.



At t=0, let *A* make a 0→1 transition



Stratified Verilog Event Queue



S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

Stratified Verilog Event Queue:

- Conceptual model that explains how different Verilog constructs are simulated
- Events at a given simulation time is divided into five layers and processed top-down

- There is no priority among *Active Events*
  - ➢ Execution order among Active Events is arbitrary
  - ➢ Source of indeterminism in simulation of Verilog code

- Statements in a sequential block (within begin-end) are executed as they appear in the block: deterministic
- Different blocks can be put in Active Events Queue in any order: source of indeterminism

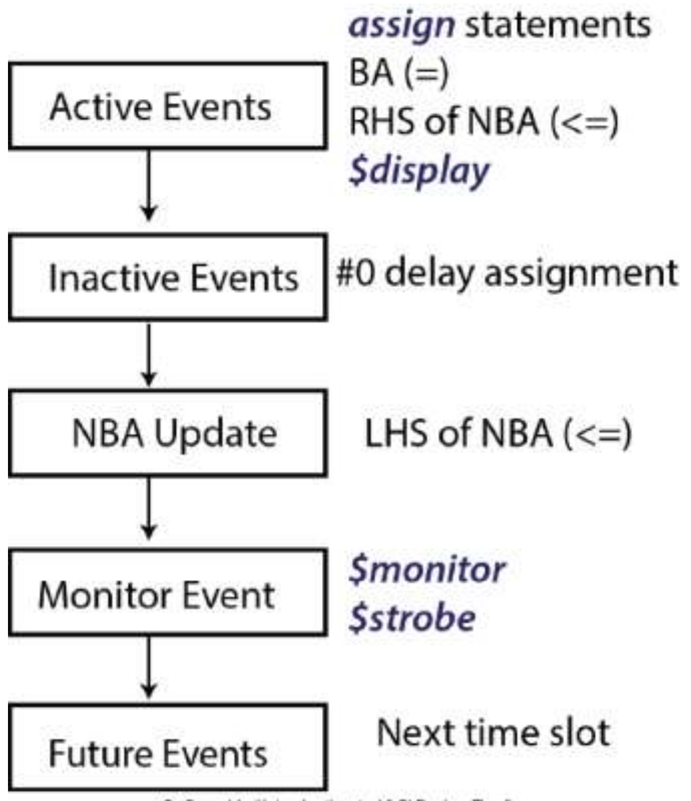
- NBA Updates can trigger further Active Events

Mechanism of Simulation: Illustration

For the Verilog code shown along side, let us determine the output of the $display function.

```
initial begin
        a = 1'b0;
        a <= 1'b1;
        $display("\nValue of a is :%b", a);
end
```

- *a* = 1'b0; put in Active Events Queue
- *a* <= 1'b1; RHS put in Active Events Queue and LHS put in NBA Update Queue
- $display: put in Active Events Queue
- Active Event Queue is processed:
    - *a* gets value of 1’b0
    - $display produces “0” for *a*
- NBA Update Queue is processed:
    - *a* gets value of 1’b1



S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

Race conditions in Verilog : Definition

- Verilog language specification (IEEE standard) defines which statements have a guaranteed order of execution and which statements have no guaranteed (indeterminate) order of execution

- *Definition:* When two or more statements that are scheduled to execute in the same simulation time, and would give different results when the order of execution of the statements are changed (as permitted by IEEE standard), then race condition is said to exist

Race conditions in Verilog : Illustration

```
module race();

        reg a, b;

        initial begin
                a  = 0;
                b = 1;
        end

        initial begin
                a  = 1;
                b = 0;
        end

endmodule
```

S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

- Race conditions can occur in many situations
- Race conditions are difficult to debug
- Some guidelines or good coding practice can be followed to avoid race conditions

References

- S. Saurabh, "Introduction to VLSI Design Flow". Cambridge: *Cambridge University Press*, 2023.
- J. Bergeron, "Writing Testbenches: Functional Verification of HDL Models", *Springer Science & Business Media,* 2012.
- "IEEE standard Verilog hardware description language." IEEE Std 1364-2001 (2001), pp. 1–792.