VLSI Design Flow: RTL to GDS

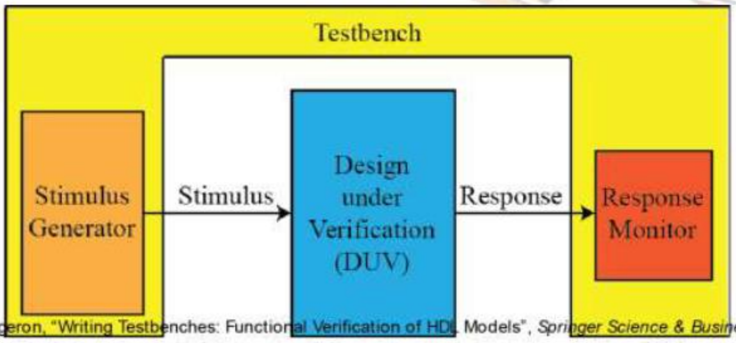
Dr. Sneh Saurabh
Department of Electronics and Communication Engineering
IIIT-Delhi

Lecture 13
Functional Verification using Simulation

Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the 11th lecture. In the earlier lectures we have looked into modeling of hardware using Verilog. We have looked into the various constructs of Verilog and how those constructs can be used to model hardware. So, once we have modeled the hardware, the next step is to verify that the model delivers the required functionality. To accomplish this, we carry out functional verification.

We have also discussed that there are two types of functional verification. First the simulation based and the second using formal methods. So, in this lecture, we will be first looking into functional verification using simulation, and in later lectures, we will be looking into functional verification using formal methods. Specifically, in this lecture, we will be covering these topics.

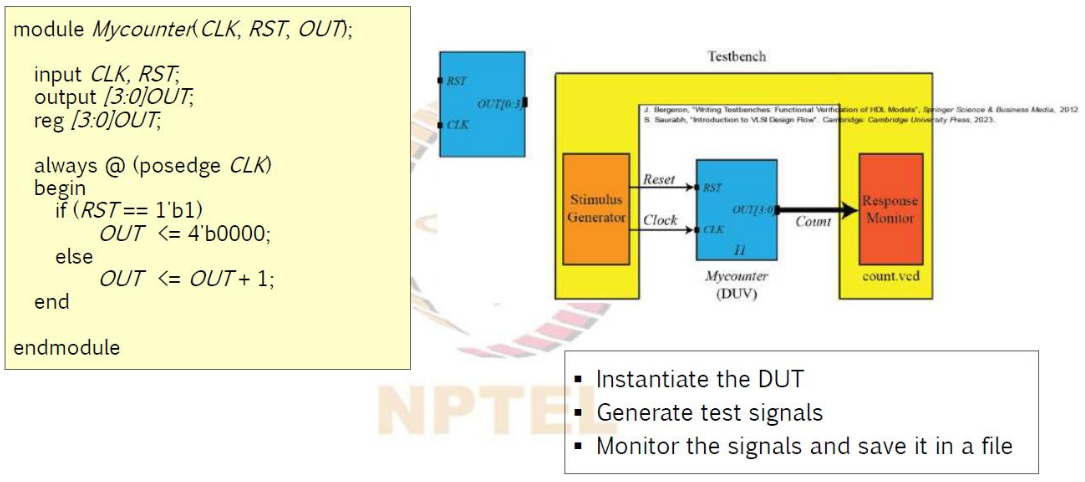
First, we will be looking into the framework of simulation, then we will be looking into test benches and how to create them, and then we will be looking into coverage models, and we will also be looking into the types of simulators and the mechanism of simulation. So, let us recap what does a simulation do. So, simulation is performed to verify that the output generated by the design is in agreement with the desired functionality for a given test stimuli or a set of test stimuli. Now, to perform simulation-based verification, we need to create an environment for doing that verification, and the environment that we create for doing simulation-based verification is known as a test bench. Now, a test bench can be created in many ways, one of the ways we have shown in this figure.



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.

So, this block is basically the design under verification or DUV. This is the model of our hardware, and this is the model we want to verify. Now, to verify this, what does the test bench do? So, test bench actually interacts with a tool which is known as simulator. Now, what does a simulator do? Simulator basically computes mathematically that, given any stimulus and a hardware model, what will be its response. Now, test benches contain two important components: the first component is a stimulus generator, and the second is the response monitor. Now, what does a stimulus generator do? It generates various stimuli; either the test bench can generate those stimuli on its own, or it can read the set of stimuli from some file and then apply those stimuli to the design under verification.

The second part is response monitor, which basically monitors the output response for a given DUV that is generated or that is computed by the simulator and checks whether that response is as per desired functionality or not. Now, this test bench can monitor the response and, decide that the response matches our expectation on its own, or it can just dump out those responses in some file, and in a later stage, it can be ensured, or it can be checked whether the response was meeting our expectation or not. For the sake of illustration, let us take a small Verilog model of a 4-bit counter. Now, in this counter, there are two input signals, RST and CK, and the output counts from 0000 to 1111. So, how it does? So, there is a clock (CLK) signal, and whenever a positive edge comes on the clock signal, then it goes into this always block.



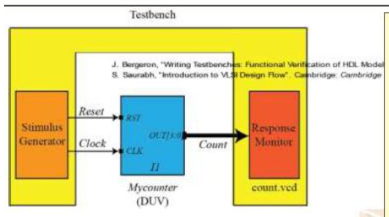
```
module Mycounter(CLK, RST, OUT);

   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

Now, if the reset signal is 1, in that case the OUT gets a value of 0000 else OUT gets a value of OUT+1, that is, it is incremented by 1. So, first, if we start with 0000 in the next clock cycle, it will get a value of 0001 and so on. Now, suppose this is our design or hardware model for which we want to create a test bench. Now, how will we create a test bench? So, this is our design under verification that is Mycounter, and the input signals are the RST and the CLK, and these will be driven by the test benches, or the stimulus will be applied on these signals, and the output that we get from this hardware model is at the port OUT, and that output will be monitored by the test bench. Now, let us look into how we can create a Verilog model for the test bench.

So, first, we need to instantiate the design under test, then we need to generate test signals from the test bench, which goes into design, to the counter, and then we will read the output from this counter and save it in a file to check whether it is behaving correctly or not. Now, these are the steps which needs to be carried out inside our test bench and we will be using Verilog constructs to perform these 3 tasks that we have listed. Now, let us understand how we can do that using Verilog language. So, we create a module. We have given its name as Testbench and we created 2 reg signals or 2 variables Clock and Teset and another wire which is Count. Then, the first task is to instantiate the design under test or design under verification.


Testbench
J. Bergeron, "Writing Testbenches: Functional Verification of HDL Model
S. Saurabh, "Introduction to VLSI Design Flow". Cambridge: Cambridge
Stimulus Generator
Reset
Clock
RST
CLK
OUT[3:0]
I1
Count
Mycounter
(DUV)
Response Monitor
count.vcd

- 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
```

So, we instantiate the Mycounter model that this is our design under verification. We instantiated Mycounter, we give an arbitrary name I1 or arbitrary instance name to this instantiated counter and then connect the signals or the input ports and the output ports of the instance with the signals that are there in the test. So, the Clock, Reset and Count these were the signals of the test bench that are applied at the input pins of the instance and the output pin of the instance. Then, the next step is to generate the test signal and initialize the test bench. So, how do we initialize it? We initialize it by first making the clock as 0, we say that clock signal is initially 0 and we say that reset is 1 meaning that we are resetting our counter to 0000. And then after a time or delay of 100 time units we say that reset is equal to 0.

So, what happens is that at t=0, the clock will be 0, and the reset will be 1, and then what will happen is that it will reset the counter to 0, and at 100 time units, after 100 time units, we make reset=0 meaning that now we have de-asserted or removed the reset. Now, when we apply the clock signal, it will be able to count and increment the count. And we say that we want to finish the simulation after a time unit of 2000. So, this is the initialization of the test bench. Now, the next step is to generate the test signals.

Now, in this case, the test signal is only the clock signal. So, there are 2 input signals for this design under verification that is reset and clock. Now we must de-assert the reset then only the counter or the actual operation of the counter will start. And then the input signal that must be given to the counter is the clock signal. Now, we have to generate that clock signal inside our test bench.

So, we generate the stimulus, in this case, clock signal by an statement always # (which means a delay of) 50 time units, clock becomes ~clock (not clock). So, suppose we had clock=0 at t=0, then after 50 time units it will change to ~clock meaning it becomes 1. And then after 50 time units again it will go down, it will become 0.Then again it will go up at 150 time units, and so on. So, a continuous clock signal will be generated of time period 100.

So, we have done the generation of the test signal. Now the next part is monitoring the signal, monitoring the output response. So, for monitoring the output response, what we do is that we open a VCD file. Now what is VCD file? VCD stands for Value Change Dump.

So, this is a format in which the output of simulations can be saved. So, we open a file by the name count.vcd. So, once we open a file count.vcd, we can dump all the information of the results of the simulation in this file count.vcd file. Why do we want to dump the information in a VCD file? Because later on after the simulation is performed we can load this VCD file in a waveform viewer and see that what is the output response and by seeing the output response we can ascertain that the output response is as per expectation. So, we dump in the VCD file all the variables, and we also monitor various signals at the clock, reset, and count. These are the three signals that are there in the test bench. We monitor them and we also keep a track of at what time using $time which is basically a system call which keeps track of the simulation time. So, it will keep or the VCD file will contain information that at what simulation time or at a given simulation time what were the values of clock, reset and count. So, these quantities, these variables or values of these signals will be monitored and using this, by monitoring the signal we will be able to verify that our design is working as per expectation. Now, we can run this test bench in the simulator. So, there are some simulators, there are some commercial simulators, and there are also freely available simulators for Verilog.

So, in those simulators we can load the design and we can load this test bench and ask the simulator to simulate and once that the simulator will simulate then these VCD file will be created and all these signals will be monitored and then we can load the VCD file in a waveform viewer and see the output response or the values of the signal at various time instances and by observing the response, we can make sure that whether our design in this case, the counter is working as per expectation or not. So, in this case, the design under verification was a very simple hardware model, just a 4-bit counter, but in real industrial designs, the design will be much more complicated, and in those cases, creating test benches will require a lot of effort. Now, another important thing is when we create a test bench, how do we measure the quality of the test bench? When we say that we have created a test bench, now, test bench contains lot of stimulus.

Now, for a small design, it is easy to ensure that we are creating sufficient stimulus. Now, for a large design how can we be sure that we have applied sufficient stimulus to our design and we have done a thorough verification. To do these things, there are various various metrics and one of them is code coverage. So, a good test bench should perform comprehensive design verification without wasting resources on dispensable task. What it means is that we want to have a kind of test bench which covers the scenario or our design exhaustively or comprehensively.

Now, if we have tested a feature in our design, now again if we apply another stimulus which is testing the same thing then probably the second stimulus is kind of redundant. So, just creating lots of stimulus is not helpful when we create a test bench. So, the test bench should create or should have a diversified set of stimulus. So, why we want to have diversified set of stimuli because adding more test stimuli to already tested features those are kind of redundant and may only waste resources in terms of run time and effort. But how do we measure that we have done a comprehensive design verification.

So, to measure that, we use code coverage, and code coverage basically identifies, a section of design under verification source code, meaning the Verilog model that we created for the hardware, identifies the section of the Verilog code that did not execute during verification. If there are some portions of our Verilog code or hardware model that did not get executed while we ran our test bench, then probably that particular portion of the design was not verified, and therefore, our testing is not comprehensive. So, a code coverage helps us monitor the progress of verification effort because if we see that we have created a test bench, we run simulation and also we compute the code coverage. If we find that, say, 10 percent or 20 percent of our code is not covered, then we will direct our effort in creating the test bench to cover those 10-20 percent of the code that were not covered earlier. So, it will help us monitor the progress of the verification effort because initially may be our coverage will be, say, 50 percent, then as we write more test cases, it may go to, say, 70 percent, 80 percent, 85 percent, and so on.

So, as more effort is put into the verification and the test bench becomes more exhaustive or more comprehensive, and we can measure that using code coverage metrics. So, there are various types of code coverage metrics, one of them is line coverage. So, the line coverage measures the number of times each RTL statement is executed during simulation. So, we can identify that in the RTL code, these lines were not executed at all. And probably then those lines can be covered later on by creating a more comprehensive test bench or creating stimuli which executes that portion of the RTL statement.

Now, the other type of code coverage measure is the branch coverage. It measures whether all branches of the code, as in if-else or case statements, were exercised in simulation. So, if we write a statement like if (condition c), then a set of statements, else a set of statements, then branch coverage will ensure that c takes both a value of true as well as false. If it is not, then it will report that this branch was not covered similarly for case statements. There is another measure which is the state coverage it measures whether all the states of an FSM have been activated and all the state transitions have been traversed.

So, we can use state coverage measures for measuring that whether we are comprehensively testing our FSM in our RTL code or not. Then there is another measure of code coverage, which is known as toggle coverage. It measures whether all variables or bits in the variables have both risen and fallen. It should take both the values, and if it does not, then it will report the variables that didn’t toggle or which did not go from 0 to 1 or 1 to 0. So, toggle coverage can also be used to measure whether our verification is thorough or not. So, a point to notice that the code coverage measures coverage of an existing RTL code.

We already have written a code, now, in that code, which line is covered and not covered, the code coverage can only measure that. Now if we give an empty module and the module does not contain any functionality for example, we just discussed a counter. If we just write an empty module and ask the code coverage tool that what is the coverage the tool may report that it has got 100 percent coverage because there was no line inside it. But the complete functionality of the counter is missing in that case. So, we should interpret the result of code coverage more prudently, meaning that it only measures whether the given RTL code is covered during simulation or not.

If the functionality is missing, the code coverage cannot identify it, and it will not report. So, in those cases, another measure of the quality of the test bench or thoroughness of testing is used, and that is known as functional coverage. So, what is functional coverage? It measures whether the specified design feature has been exercised during simulation. So, the design features are not extracted directly from our RTL code. What is done is that we need to provide design features using a coverage model.

Suppose this is our RTL code of our hardware. Now, we create a separate model, which is known as a functional coverage model, which contains a description of the various features in our design. So, the features of our design are contained in a coverage model. It is not automatically inferred by the tool. What does the tool do? The tool will report that whether the features specified in the coverage model were actually executed by the test bench or not.

If not, then it will report that those features were not tested . Since we are creating a separate model for a coverage model which is independent or separate/segregated from the hardware model, if we are creating a good coverage model, it will be able to flag the cases where the features are missing in our design. Therefore, in that sense, the functional coverage model is much better than the code coverage model. But what is the downside? The downside is that, in this case, we have to create, as a designer, a separate functional coverage model.

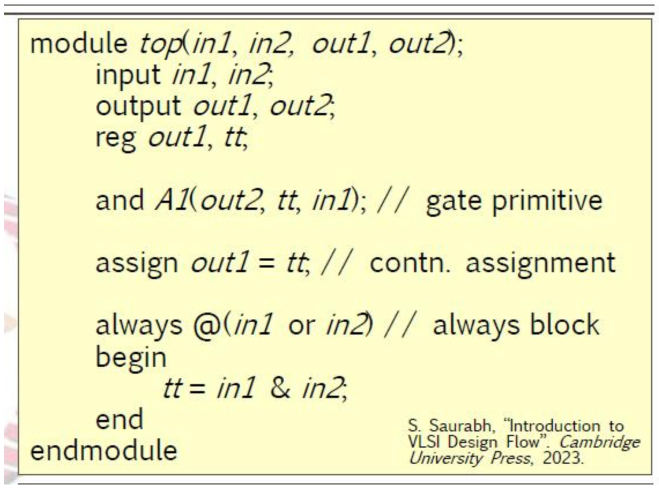
It is not automatically inferred by the tool. So, the coverage model is independent of the design implementation. It can detect missing features also in the implementation. But a designer needs to make effort to create the coverage model. Now, let us look into the simulation in Verilog, the mechanism of simulation in Verilog. Why is it important? Because in earlier discussion we have said that the Verilog language was designed or was invented for the purpose of verification. So, the verification was the primary reason for why Verilog was invented. So, all other uses of Verilog, for example, synthesis and others, follow the syntax or semantics, which is followed by simulation. So, if we understand the simulation of Verilog thoroughly, then it will help us understand the

Verilog language and its application for synthesis and other purposes very well. And therefore, we can write very good hardware model using Verilog.

So, understanding the mechanism of simulation is very important for a digital hardware designer. So, before going into the mechanism of simulation, let us look into a few important definitions. So, Verilog considers a design as consisting of connected processes and what are processes? Processes are design objects that a simulator can evaluate and produce a response to a given stimulus. For example, the gate primitives, initial block, always block, continuous assignment and procedural assignment statements these are examples of processes. Why? Because for these kind of Verilog constructs the simulator can evaluate them and produce a response or can compute the response for a given stimulus.

Now, what is an event? Anything that requires simulator to take some action that is known as event. For example, if there is a change in value of a net or a variable, there is a design in which the value of a net changes. Why? Because we might have applied a stimulus. Now, that change in the value of a signal or a net or a variable is known as an update event, and there is another type of event, which is known as an evaluation event. So, whenever an update event happens then because some other processes can be sensitive to this update event.

And whenever other processes are sensitive to an update event, then what do we need to do is or what the simulator needs to do? It needs to evaluate that process. So, evaluation of processes is required when an update event has occurred for the processes which are sensitive to that update event. Now, let us look into an example. Suppose this is a module, we have a instance, primitive gate instance, then we have a continuous assignment statement and then there is an always block. Now, what are the processes in this module? So, the processes are the gate primitive/ gate instance, the first process, then the second process that is the continuous assignment and the third is this always block, these are three processes.



```
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
```

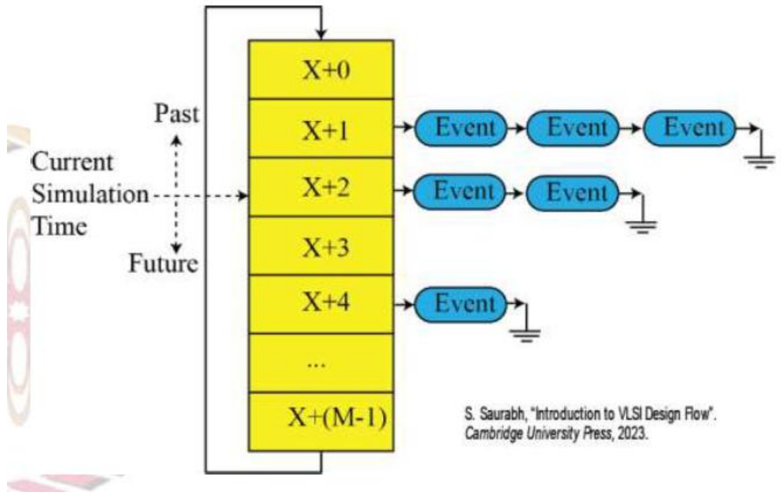
S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

And suppose that the signal in1 makes a transition from 0 to 1. So, in this case what will be the events create? Now in1 changes a value from 0 to 1. So, it creates an update event because the value has changed from 0 to 1, and therefore, an update event is created for the net in1. And then an evaluation event. Whenever there is a change, there is a update event on an net, in this case in1, then the processes which are sensitive to it, is this always block because in1 is coming in the sensitivity list and also this gate primitive because on the input there is in1. So, these two processes are sensitive to the changes in in1 and therefore there will be an evaluation event in the gate instance and the always block in these two processes, there will be an evaluation event.

But there will not be an evaluation event in this continuous assignment statement because it is not sensitive to in1. What is a simulation time? Simulation time is the time value maintained by simulator to model the actual time in the simulated circuit and the simulation time will depend on the test bench and the design being simulated. And we should not confuse simulation time with the time taken by the simulator to do the simulation. So, the simulation time will depend on the test bench and the design, and if the design and the test benches are well-coded, then any simulator will give the same simulation time for the same events. But the time taken by the simulator to simulate is dependent on the processor on which we are running our simulator, and it also depends on the algorithm of the simulation.

So, different simulators can have different time of simulation, but the simulation time for a given event should be ideally same for any simulator. Now, events have got simulation time associated with it. So, each event that is there in the simulator, for each event there is an unique simulation time associated with it. Now, simulator should ensure that events are processed in the increasing order of the time. So, for example, if an event is at say 10 time unit then it should be processed after say it has processed events of 9 time unit and before it has processed the time events of 11 time unit.

So, the event should always be processed in increasing order of the simulation time. To do this, what the simulators do is they keep an event queue, and what is an event queue? It is an internal queue maintained by simulator, ordered by simulation time and it is ordered so that the processing can happen easily in increasing order of simulation time. Now, tools can keep various kinds of data structures for maintaining the event queue, for example, a priority queue or other kind of sophisticated data structure. One common or one popular data structure for keeping the event queue is the timing wheel, which is shown in the slide. So, timing wheel is an efficient data structure to implement event queue.

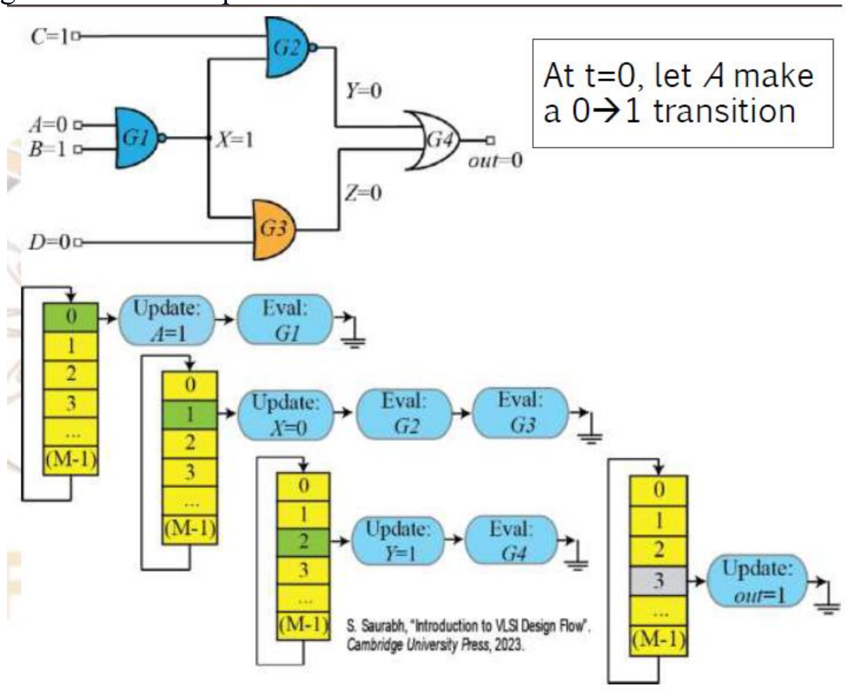

X+0
X+1
X+2
X+3
X+4
...
X+(M-1)
Past
Current Simulation Time
Future
Event
S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

Now, how are events kept in a timing wheel? So, in a timing wheel we have time slots and each slot has associated with it a time or for each simulation time we have a slot on the timing wheel and an event which is occurring at a given simulation time it is appended in a linked list associated with the corresponding timing slot in the timing wheel for that given simulation time. So, the timing wheel contains basically an array of M slot. So, M may be say 100. So, let us take an example of 100. So, an array of 100 slots indexed by simulation time where for a given simulation time, the slot is T%M meaning that remainder of or the modulo division of T, T is the simulation time and M is the number of slots.

For example, if we consider a time of 58. So, what will be the slot corresponding to that simulation time 58, it will be 58%100= 58. So, it will be going to the slot number 58 in this timing wheel. Similarly, suppose there is another simulation time T = 163. Now, where will this simulation time map to the timing wheel it will be 163 %100 and it will be mapping to the timing slot 63. So, for each simulation time, there will be a defined slot on the timing wheel, and the events are basically appended to the corresponding time slot.

For example, the event on suppose, take time instant of 1. So, it will map to the slot 1 and the events which are occurring at T=1 those will be appended at this location. Now, as the simulation time progresses, the events are processed in the increasing order of the time slots, and then it is wrapped back. And once we process an event for a given time slot, then we remove all the events associated with that, and that time slot becomes empty, and therefore, it can be reused again and again. So, that is how we can use the same time slot throughout our simulation. In some cases if the time slot or the time that we are processing is still occupied by some other older, some other time and there is a clash, in that case, we can put the event in some secondary data structure in a linked list and later on when that time is being being considered then the event from that secondary data structure can be loaded and again put on the time slot and then processing can be done.

Now, how does the processing of events is done on the timing wheel? When we apply stimulus, it creates update events. Now, update events can trigger evaluation events for the nets or signals which are sensitive to it. So, once the evaluation events are processed, then evaluation can further trigger update events and can lead to new update events. And therefore, there is a sequence of update, evaluate, update, evaluate and so on it goes on until the end of the simulation time. So, let us take an example and understand how this processing of events takes place.



Suppose this is our circuit and the initial value at the inputs are C=1, A=0, B=1 and D=0 and the corresponding initial value, I have written here. Now, assume that the delay of each gate is 1 time units like each gate delay is having 1 time unit. And we assume that at T=0, let A make a transition from A =0 to 1. So, this is making a transition from 0 to 1.

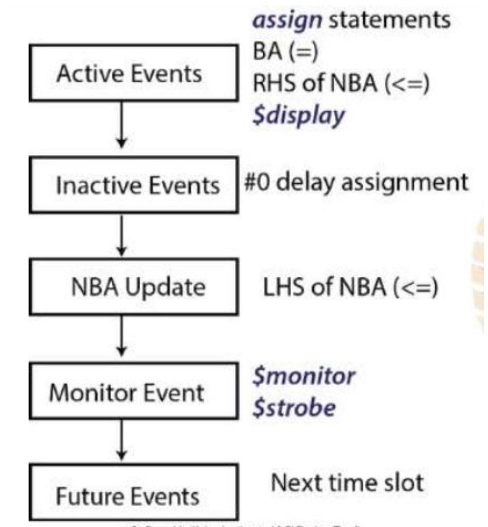
Now, how will the processing of events be done? So, in the timing wheel, we will have an update event created for it at T=0, meaning there will be an update event A=1. Now, when there is an update event at a net or at a signal A, the gate G1 is sensitive to it. And therefore, we must evaluate or an evaluation event must be created at G1. Once this evaluation is done on G1. So, initially the value of this gate was or the value of X was 1, why because initially A was 0, B was 1 and therefore, this is a NAND gate and the output was 1.

Now, what has happened is that update event was created and A became 1, when A became 1, this is a NAND gate, both the inputs are 1 and therefore, X will become 0. So, when we evaluate, when the evaluation event on G1 is processed, and update event on X will be created because the value of X was 1,. Now it has changed to 0. So, update X is equal to 0 is created. Now, when X is changing, these 2 gates are in the immediate fan out, and therefore, an update in evaluation event will be created at G2 and G3.

Now, once we evaluate G3, the value at D was 0 now X is equal to 0. So, Z will remain 0. It will not change, but what will happen to this G2? Initially, it was 1, C was 1, and output Y was 0, but now X has become 0. So, Y will become 1. So, evaluation of G2 will create an update event, but evaluation of G3 will not create any new event. So, we have update on Y is equal to 1. Now, in the fan out of Y, we have G4 and therefore, there will be an evaluation at G4 and once the evaluation is done on G4 because Y has become 1 then the out will become 1 instead of 0 it will become 1.

So, there will be an update at out to 1 and then this update of out is equal to 1 will not create any new evaluation event because there is nothing in its fan out. So, this is how the simulation will be done in Verilog and how the timing wheel will be used in maintaining and processing the event in an increasing order of time. So, in this example, I have shown just primitive gates to illustrate how simulation is done. For other types of processes like always block and other kind of assignments and so on, the mechanics remains the same but only the constructs changes, the evaluation and update and other things, the mechanics of those things remains the same, only how the evaluation will be done individually for a given construct that might change. Now, what we have discussed in the previous slide is how the processing of events will be done when they are in different time slots.

Now, if the events are in the same time slot, say at T is equal to 1, there are multiple events. Now, out of those events, which one will be processed first, which one will be processed next, and so on? So, those kind of things are defined by the language using what is known as stratified Verilog event queue. Now, what is a stratified Verilog event queue? It is a conceptual model that explains how different Verilog constructs are simulated at a given time slot. So, the events at a given simulation time or a given time slot of a timing wheel is divided into 5 layers and processed top to down and what are these these 5 layers? The first one is active event, the next one is inactive, third is non-blocking assignment update or NBA update, the fourth is monitor event and the fifth is the future events.



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

So, the events are processed top-down in this manner. Now, what are the events or what are the types of events in the active events? So, in the active events, we put the continuous assignment, assign statement, we keep the blocking assignments and then the

RHS of non-blocking assignment. So, you might remember from the previous lecture we said that if we write something A <=B this is a kind of non-blocking assignment and we had discussed in the previous lectures that non-blocking assignments are processed in two steps. So, in the first step, the RHS is evaluated, and an event is scheduled, and then in the second step, the LHS is updated. So, in the active event queue, we only put the RHS of NBA and the LHS of the non-blocking assignment are put in the lower hierarchy in the stratified Verilog event queue; this is processed later, and the $display statement is in the active events queue. After active events queues, all of them are processed, then inactive events are processed, and inactive events are #0 delay assignment. The language allows #0 delay assignment, but typically we should avoid #0 delay assignment in our designs or in test benches because there are more efficient methods to model the effects of #0 delay.

Then comes the non-blocking assignment, update the LHS part, and then comes the monitor statement, the strobe statement, and then it moves to the next time slot. So, this is how the processing will happen. Now, there is no priority among active events. So, if there are multiple events in the active event queue, there is no priority, meaning execution order among active events is arbitrary. Why this kind of arbitrariness is there in the Verilog simulation, the reason is that we want to model a kind of parallel operation in hardware. So, if there are multiple gates and there is a signal, we do not know whether this signal will reach first, this gate or this gate. There are uncertainties involved in that and therefore, either this one can be processed first or the next one can be processed later on.

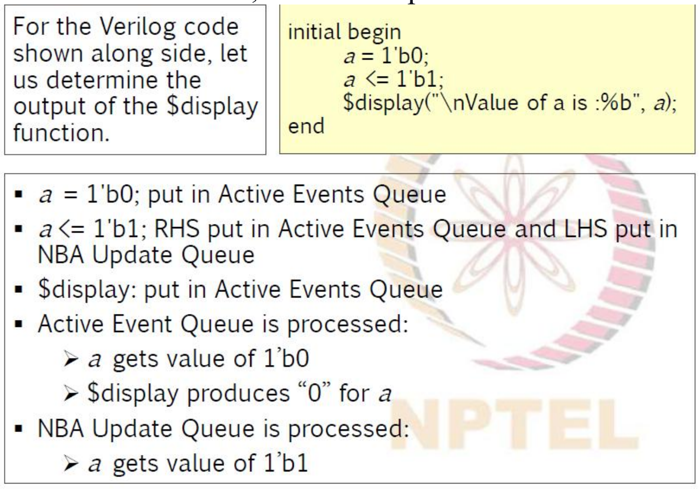
Ideally, our functionality should not depend on which one is processed first, and therefore, the tool or the simulator is free to process any of them. So, this is a source of indeterminism in simulation of Verilog code. Now, this indeterminism is not a bad thing, but it is basically inherent to our hardware, and in that sense, this indeterminism in the simulation of Verilog code is trying to capture that indeterminism in the hardware also, and therefore, it is more modeling the hardware in a realistic manner. So, this is about the events that are in the active events queue, the various events; suppose there was an always block, and in this always block, there was a blocking assignment, and there was another always block, and inside that, there was another blocking assignment. Now these two blocking assignment both of them can be put in this active event queue and either of them can be processed in any sequence as desired by the simulator, we have no control.

So, this is what we mean by the source of indeterminism in simulation of Verilog code. So, ideally, our functionality should not be dependent on which of them is processed earlier or later. But there are some deterministic things also. So, statements in a sequential block are executed as they appear in the block. For example, if we have an always block and we have begin and end, we have various statements, statement 1, statement 2, statement 3 all are say blocking assignments.

Then, the language says that the statements in a sequential block are executed as they appear in the block. So, this one will be executed first and then this and then this, there is no indeterminism here. So, if there is a sequential block meaning that we have within a begin and end block then the statements will be processed as defined in the, as given in the hardware model, there is no indeterminism. So, different blocks can be put in the active events queue in any order that is what I have mentioned here. If there are different blocks, they can be put in any arbitrary order, but if the statements are within a begin and end block, then those must be processed in a sequential manner only.

So, now when the processing is going from active event to inactive event and NBA event, it may happen that when an NBA update happens because of LHS of the NBA, a non-blocking assignment happens later in the stratified queue whenever then update is made to the LHS of NBA, non blocking assignment, then it can trigger further active event. It can trigger more events because if we have an update event then it can lead to evaluation events and those evaluation events can go into again your active event queue and then those needs to be processed. So, it is not processed in this manner that is only in this manner. What may happen is that once this is done, then again, it must be processed, and so on.

So, there may be iterations in this kind of simulation. Now, let us take an example that how this stratified event queue of Verilog can be used or will be used by the simulator to give the result. Consider this block . So, in this block, we have an initial block, and in this, we have begin and end statement. So, this is a sequential statement.


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

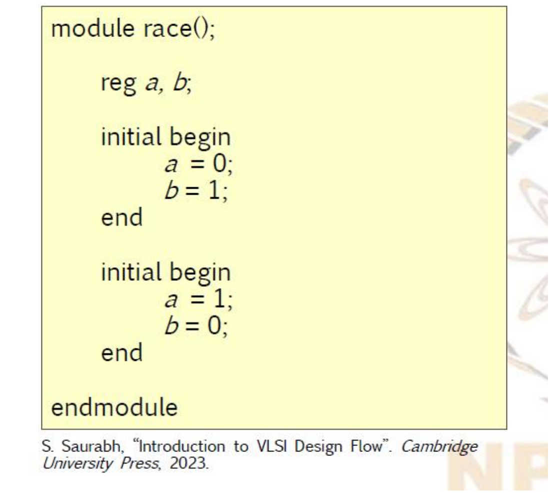
So, these statements will be processed in a sequential manner. Now, let us understand that what will be displayed by this statement display. So, this is the stratified queue that we just discussed and let us understand how the processing will be done. So, the first statement a = 1'b0, this statement will be put in the active event queue. So, here we have put a = 1'b0, then the next statement comes.

And here we have a non blocking assignment. So, in a non blocking assignment, in the active event queue, we have to put only the RHS part. So, the RHS part is 1'b1. So, whenever this RHS is processed then it says that the LHS will become this value whenever it will be processed. So, this RHS will not be reevaluated later on. Now, whenever it will be processed, the RHS will be put in the active event queue and LHS part in the NBA update queue.

So, here we will say that a is scheduled to get a value of 1’b1 that was on the RHS of the NBA. Then comes the third statement, which is $display, and that will be put here. And since these are sequential statements, these will be processed in this manner. There is no ambiguity here. Now, active event is processed a gets a value of 1’b0, then the $display will produce the value of a and the current value of a is 0 because when this was processed, a was assigned a value of 0. Note that the LHS of NBA that is a, is not yet updated, but the display statement will be executed before that.

So, the display will produce a value of a, and then the NBA update queue is processed, and then a will get a value of 1’b 1. So, in this case what will happen is that display will show a value of 0, while the final value of a will be 1 when the LHS of NBA will be processed. So, this is how the processing of events for a given time slot occurs. Note that all these things are occurring at time T =0, and the time slot is the same, within a time slot how these events are processed. Now, because the events in an active event queue those can be processed in any arbitrary order if they are not belonging to a sequential statements then they can be processed in any arbitrary order, it can lead to a situation in which the results depend on the order in which the processing happens. So, Verilog language specification defines which statements have a guaranteed order of execution. For example, in a begin-end block, the statement will be executed as they appear in the RTL code, and which statements have no guaranteed or indeterminate order of execution.

For example, if there are multiple always block in our module then which of these always block will be processed first, we do not know, all will be put in the active event queue and it can be processed in any arbitrary order. So, in those cases, there is no guaranteed order of execution. So, when two or more statement that are scheduled to execute in the same simulation time and would give different results when the order of execution of statements are changed as permitted by IEEE standard then we say a race condition has happened. So, what is a race condition? This is an important concept in simulation-based verification that sometimes it may happen that the result of the simulation will depend on the order in which the events are processed and if we change the order as permitted by the IEEE standard and we get different results in the simulation, then we say that there is a race in simulation.



```
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.

Now let us take an example to explain this what we mean by race. Now consider this situation in which we have two initial blocks. Now, as per IEEE standards, any of these initial blocks can be executed first. Now if this one is executed first and next this one is processed then the final value of a and b will be a is equal to 1 and b is equal to 0, but what if this one is processed first and then this one then the final value of a will be 0 and b is equal to 1. And therefore, the result of the simulation will be dependent on the order in which these initial blocks are processed and therefore, we say that there is a race condition in this. So, the race condition can occur in many situations, in this case is a very simple case or rather a trivial case in which race condition is there and we can easily find out, but many a times what happens is that the race conditions happen and it is very difficult to find out.

And more so because the simulator will give you one answer , but the race condition will be discovered later on, maybe when the synthesis assumes some other behavior, and at the gate level simulation, you get another result. And then probably we go back and trace that the result produced between the gate level simulation and the RTL simulation are varying because there was a race condition and the simulator assume this behavior and the synthesis tool this behavior and therefore, the gate level simulation give different result. So, this race condition can be very problematic and it often becomes difficult to debug and therefore, people have adopted some guidelines to avoid the race conditions. For example, there can be guidelines or some of the guidelines to avoid the race conditions, such as that we use a blocking assignment in combinational circuits and non blocking assignment for sequential circuits. Another guideline can be that we should not write to a variable in more than one always block. In that case, there can be a race condition.

So, there are other guidelines which can help us in avoiding the race conditions. So, some guidelines and good coding practices can be followed to avoid race conditions. So, these are some of the references that you can look into to understand more about functional verification using simulation. Finally, to summarize, in this lecture, we have looked into simulation-based verification.

We discussed how we can create test benches and use them for simulation based verification. We also looked into the mechanism of Verilog based simulation. In the next lecture, we will be looking into that given an RTL model, how those are synthesized, and how those models are used further down the design. That is the end of the lecture. Thank you very much.