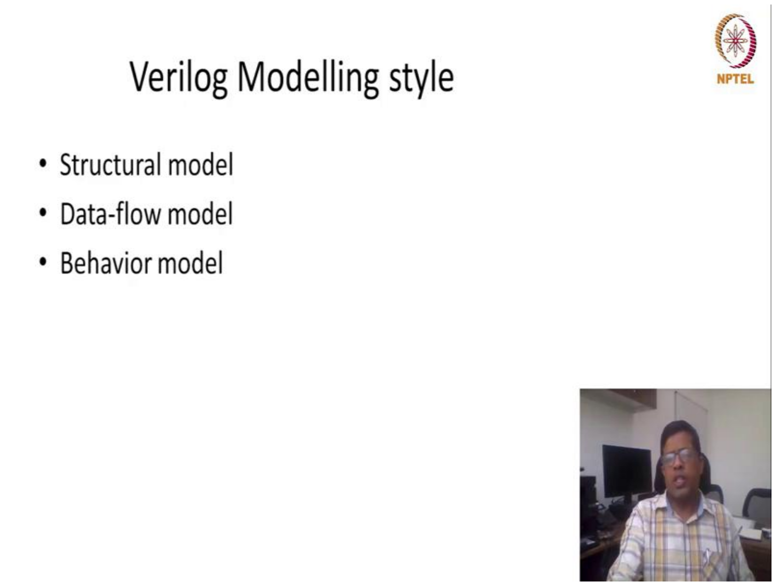
Digital System Design

Professor Neeraj Goel
Department of Computer Science Engineering
Indian Institute of Technology Ropar

Lecture 45
Verilog-Behaviour Model-1

(Refer Slide Time: 0:24)



Verilog Modelling style

- Structural model
- Data-flow model
- Behavior model

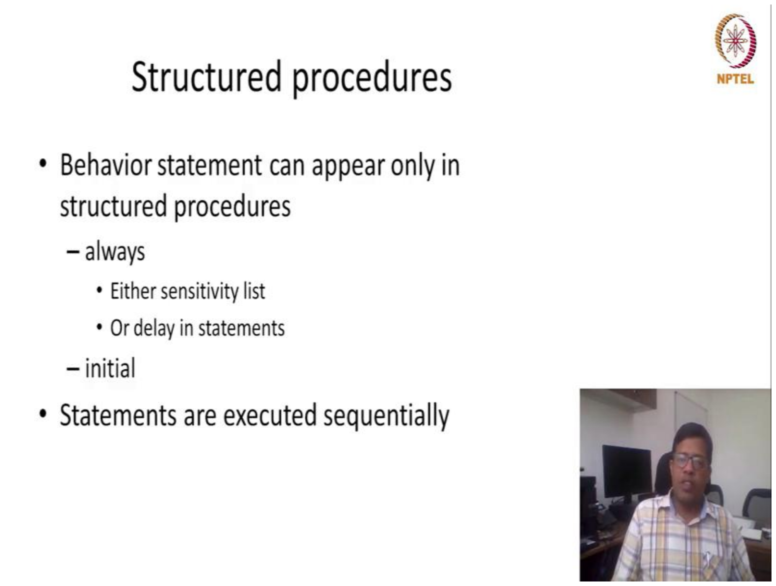
Hello all, today we are going to discuss behaviour modelling in Verilog. So, far we have done couple of assignments and in those lab assignments we have been using structural model or data flow model. Now, let us quickly brief or revise in structural model the way we are modelling were we were instantiating all the sub components or all the components and then connecting them using wires, all the logic would be there only using a pure structure model we will using, we will use primitive gates and using those primitive gates it would create modules and those modules would be used in in some other modules in a hierarchical fashion.

While in data flow model, you can use assign statement or a data flow statement where logic expression can be given as a input. And then we can use mix of data flow and structural model to create a hierarchical model or even more complex models. Although, structural model and data flow model both are extremely good and powerful mechanisms to model any digital circuits.

But as the complexity grows towards the algorithmic level, or when we would like to design something very big maybe 100, 1000s of gates or something like processors or accelerators, in that case, we need to have some other component which where we can directly specify the behaviour or directly specify the functionality in terms of the algorithm.

So, Verilog also provide mechanism, so, that we can specify the model in terms of algorithm and this is called behaviour model. Today, we are going to discuss about behaviour model in Verilog in other hardware description languages like VHDL also we have such kind of a behaviour model where we can specify using sort of a high-level language or CC plus plus kind of syntax,

(Refer Slide Time: 2:52)



Structured procedures

- Behavior statement can appear only in structured procedures
  - always
    - Either sensitivity list
    - Or delay in statements
  - initial
- Statements are executed sequentially

In case of Verilog behaviour statements can appear in structure procedures and there are two structure procedures in Verilog one is always block and another is initial block both of these initial and always block are parallel in nature.

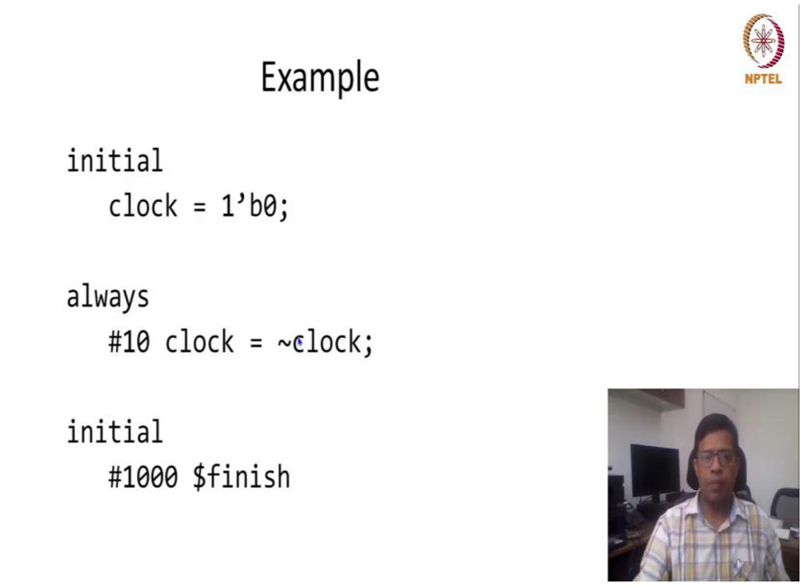
So, for example, if there are multiple initial blocks or multiple always blocks they would run in parallel. So, we have already talked about initial blocks when we were discussing about test benches in Verilog, but today we are going to spend more time on always block or more time on behaviour statements.

So, this behaviour statement which can be present inside any of these initial always block. The difference between initial and always block is in initial block will run only once during the simulation at time 0 or we can say at time of elaboration. So, when simulation is about to start then initial block will be executed only paths, but always block is the block which or the list of statements which are there in always block they can keep on running and till the simulation will end. How will we make sure that these statements will keep on running? So, there would be some sensitivity list or some delay in the statements.

So, whenever any of that event say even in the sensitivity list would happen then this always block will get triggered. So, and whatever behaviour statements are there those behaviour statement can only be present either in the initial block or in the always block. And let us say in always block there are multiple statements. So, all of these statements would execute sequentially. Remember, when we were discussing about data flow statements, data flow statements, we were saying that they were always in parallel, so it does not matter which statement appear first or appear later.

But they would be assigned or they would be, they would execute, or they would look like to be executed at same time, but in behaviour statements, so first statement, if we have written our statement first, then it will be executed. And then once it has finished its execution, then only the next statement will get executed. We will see with a couple of examples.

(Refer Slide Time: 5:43)



Example

```
initial
   clock = 1'b0;

always
   #10 clock = ~clock;

initial
   #1000 $finish
```

So, let us take an very simple example of this initial and always block. Let us say we would like to design a clock; we would like to model a clock. So, we can model a clock because it is it is a repetitive circuit, we can say always with a delay of 10 units clock is equal to naught of clock. So, with this we can create a clock which has a time period of 20 units. After every 10 unit, if the clock was 1 it will become 0. And if it was 0, then it will become 1.

Then the question comes that what would be the initial value, so because always block will keep on running and there is no separate place for initializing the block, initializing any particular variable or we you cannot distinguish whether this particular statement is running first time or the second time or the third time.

So that is why this initial block will be used. And we have seen in other form of test benches also that different test benches, that this initial block is used whenever we would like to initialize some variables whenever we would like to give input at different time intervals. So, this initial block is used for mostly for that purpose.

So, that means initially we said clock would be 0, then after every 10 units of time clock would be inverse or whatever it was earlier. So, eventually it will have a clock period of 20 units, and then it will keep on repeating. When will it stop? So that is the problem, with your always blocked, if it will keep on running, if it will keep on running, then you do not know when it will stop.

So, that also brings us to the one important question that usually when do our Verilog simulation will finish? Verilog simulation usually would be finished either in either of these two cases. First case would be that if you in the simulator, you specify that I am going to execute or I am going to simulate for this much amount of time, let us say 100 nanoseconds or 100 microseconds, or maybe 1second.

So, you specify for how (duration), what is the duration of your simulation, what would be the duration of your simulation in the simulation framework. So, let us say you are using IAC simulators or vivado simulator or maybe you are using iverilog then it do specify that for how long you are going to run it.

If you are using EDA playground, then also by default it get simulated only for certain duration of time. Now, otherwise, the simulation will end when there is no further event. So, that also you would have observed in your some of the previous lab assignments that when then let us say you have given 2 or 3 stimulus values, after those 3 stimulus values, there is no further value because you are not going to introduce any other input.

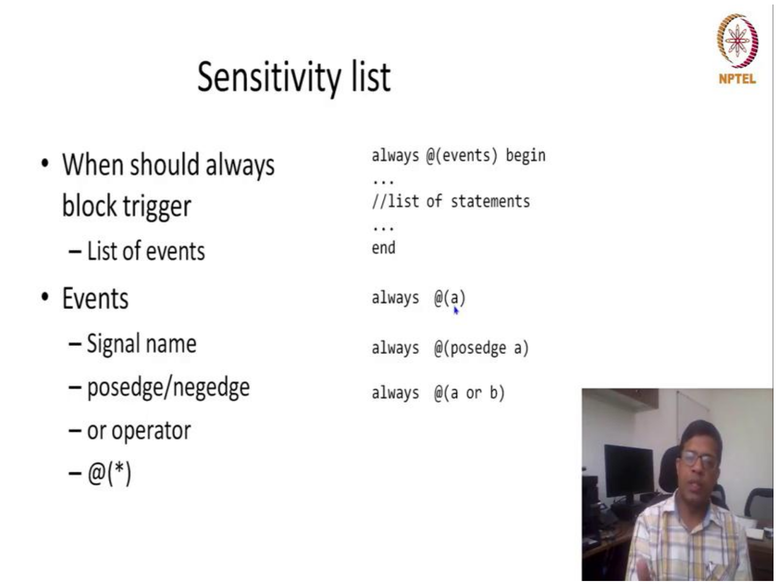
So, that means they would, there will not be any other change in the output. Because no other event is happening then simulation will cease or simulation will stop itself. But in this case, when you are having an always statement that means event it will keep on happening. So, every time there would be a clock so after every 10 units of time clock is going to be the reverse of clock. So, that means the list of event will never finish.

In those cases, in the Verilog file itself, we should specifically specify or specifically say that when do we want our simulation to finish. Then we can specify initial let us say some time, hash 1000 and then say dollar finish, dollar finish means the simulation would finish whenever 1000 units of time will happen.

So, also remember that because these two initial blocks are in parallel, so, they this would be although executed first and after 1,000 units of time this would execute. So, we could have even combined both of these initial blocks in this particular case. Another thing to notice is that here the always block has no sensitivity list. So, we are not it is getting called automatically because it depends on the clock and whenever there is a 10 units of time it will get automatically triggered by itself. So, let us take more example to understand that when this always block will get triggered.

So, now try to see this way that you have an always block, you have a list of statement in this. Now, this list of statement will get executed sequentially. Now, when should we start executing this list of sequential statements because they are all sequential statements, so, there has to be some particular time when we have to say that yes, now they should get executed.

(Refer Slide Time: 11:23)



Sensitivity list

- When should always block trigger
  - List of events
- Events
  - Signal name
  - posedge/negedge
  - or operator
  - @(*)

```
always @(events) begin
...
//list of statements
...
end

always  @(a)

always  @(posedge a)

always  @(a or b)
```

So, the list of, when this always block will get triggered. So, there has to be some list of events, when any of these events will occur then this always block should get triggered. Now, what is this list of events, the possibility of lists of events could be could be good. So, you can see that we have an always blocked and we have to say that at the rate of event at the rate means, we are specifying that whenever this event will occur, then the list of statement will get executed.

So, what is this event? This event could be you can specify a signal name. So, whenever when you specify a signal name that means for example, we say at always at the rate a. So, whenever this signal a will change, will change means whether it will become 0 to 1, 1 to 0 or maybe it changes from X to 1, X to 0 or may be 1 to Z, when any value from one value to another value, if this particular signal, this particular wire this particular port, if that changes its value, then this list of statement can get triggered. So, this is one particular way.

Now, the other is we do not want to trigger this particular always block at any event change or any particular value change, but we would like to say that whenever there is a positive edge so, whenever the value of a is changing from 0 to 1 that means positive edge whenever the value is changing from 1 to 0, then it would be a negative edge.

So, we can specify that when any particular edge is there, then this list of statement should get triggered. So, we can say always at the rate posedge positive edge of a and then the we can start the block we can say begin and end and then we can write the list of statements and those lists of statement will get executed whenever there is a positive edge of a.

Now, let us say that we want to get this always block triggered not only at 1 event, but any of the multiple events. So, then what we can do is we can have a OR operator we can specify that let us say always at the rate A or B so when value you have a will change or value of b will change, then this block should get triggered.

So, what are we saying here, we are trying to say that in case of data flow statements, data flow assignments, they were continuous assignment. So, that means whenever input would change, output would change automatically no sensitivity list required. No list of events required. They were automatically taken whenever any of the right-hand side variable will change, data flow assignment will be computed and would be assigned.

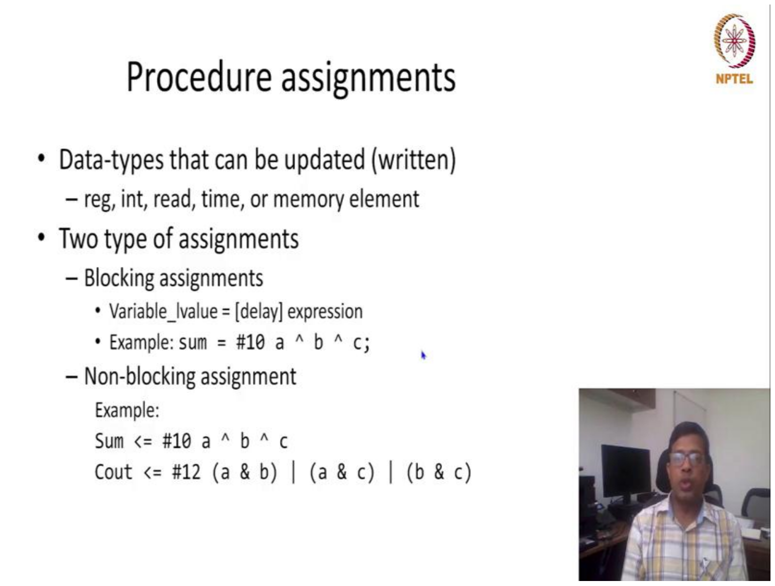
But in case of procedural statements, in case of always block, because always block itself is not a parallel. This is not the list of statements are not parallel but sequential in nature. Because they are sequential in nature, we would like them to get triggered only when we would like them to get triggered.

So, there has to be some specific event, so we have to specify that when do we want them to get triggered, so that is why we are specifying this list of sensitivity or list of signals to which my always block is sensitive. Now, let us say I want to design some kind of combinational logic. In combinational logic my output depends on the input.

Now, if the list of input is huge in a combinational logic and I would like to design or model my combinational logic using behaviour statements, if I am trying to design a combinational logic using behaviour statement, then list sensitivity list could be huge. So, what sometimes we do is we simply write at the rate wildcard star, which means that any of the input or any of the right-hand side variable in the always log can trigger the always block.

So, your Verilog compiler is doing the hard work for you in that case, it is trying to identify what all variables are using the right hand side of any expression and all of those right hand side expressions are automatically taken by a compiler to automatically taken by a compiler as a sensitive sensitivity list or as a list of events which could trigger them. Here in those cases, it would be mostly the, this particular level triggering, so whenever there is a value change in any of those variables, then it will get triggered just like a data flow statements. So, I can understand that some of the things may not be understood here, but we will, in this lecture, we will take a couple of examples, at least one for all of these cases, so we would be able to understand using them.

(Refer Slide Time: 17:06)

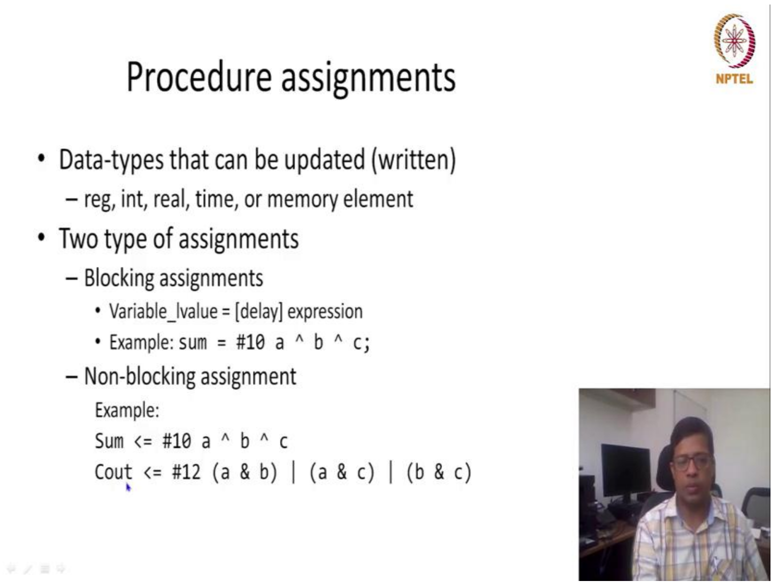


Procedure assignments

- Data-types that can be updated (written)
  - reg, int, read, time, or memory element
- Two type of assignments
  - Blocking assignments
    - Variable_lvalue = [delay] expression
    - Example: `sum = #10 a ^ b ^ c;`
  - Non-blocking assignment

    Example:

```
Sum <= #10 a ^ b ^ c
Cout <= #12 (a & b) | (a & c) | (b & c)
```



Procedure assignments

- Data-types that can be updated (written)
  - reg, int, real, time, or memory element
- Two type of assignments
  - Blocking assignments
    - Variable_lvalue = [delay] expression
    - Example: `sum = #10 a ^ b ^ c;`
  - Non-blocking assignment

    Example:

```
Sum <= #10 a ^ b ^ c
Cout <= #12 (a & b) | (a & c) | (b & c)
```

So, the other thing before going ahead should be understood here that should be understood that in any procedural statement, what are these procedural statements that also come in the next slide, but in any other procedural statement, there would be assignment so, there would be left hand side or right hand side.

So, your right-hand side means the expression and left-hand side means the variable where you are going to assign these values. So, wherever we are going to assign these values, these values can be assigned only the variable type has to be register. So, that means r e g reg should be the datatype of whenever we are we are assigning or we are writing any expression to a variable. So, that variable has to be of reg type

Now, you might have seen or you will see that when you will do experiment. So, when you do experiment, you will see that instead of reg it is integer, real or time then also you can write these you can write to those variables. So, why because internally int integer real money float or time all of these are also internally of registered type, are taken as a register type by Verilog compiler.

So, in other words, we can say that you can write a variable in a procedural statement only if the datatype of that particular variable is reg in real time. All of these are otherwise of type registers internally or it could be a memory element. Memory element is also defined by either reg or int type. So, that is about the right, left hand side or l value.

Now, the two kinds of assignments in procedural assignments, one is called blocking, another is called non-blocking. So, in blocking, you can whenever so, as we said earlier, in procedural block like always block or initial block, when one statement gets executed then only the second statement gets executed. And that would happen because the assignment is called a blocking assignment.

So, blocking assignment means that you will write the l value and then equal to sign and then you can delay is option and then you can write expression. So, for example, you can say that a sum is equal to hash 10 a XOR b XOR c. Now, when this statement will get executed, then only the next statement can get executed.

What does it mean? It means that this sum will get evaluated only after these 10 units of time and the if the next statement will start that next statement will start only after 10 units of time when this first statement were or started executing. This may not be this may be a desirable functional T at many places, but if you are involving delays, so, it may not be desirable at other places.

So, desirable for example, whenever you are writing test benches, we write test bench in the same way. So, we write hash 10 then the first condition, then, we write another hash time and then the second a second stimulus value then third stimulus value. So, all of those things, we are sure that we would like to stimulate, we would like to assign only after that delay, but if we are using these behaviour, behavioral statements or procedural statements to model a data flow or to model a combinational logic, so, then we would like to have a non-blocking statement

In a non-blocking statement, what happens that all these statements would execute at the same time just very similar to data flow statements. So, you whenever you are executing the next statement, you will not wait that whether this particular statement would finish or not. So, for example, you are designing this, we are modelling a full adder.

So, you can say some non-blocking statements are characterized by this less than and then equal to this particular symbol. So, you are saying sum less than equal to then hash 10 a XOR b XOR c and then Cout less than or equal to a hash 12 a and b OR a and c OR b and c. Now, in this case, I would see my sum would be available at hash 10 time and Cout would be a available at hash 12 time.

Because this statement would be executed at the same time. So, this the next statement the second statement has not waited till my first statement would finish, while in the same in the other hand, if we were using blocking statement, that means this less than sign was not there. In that case sum would be available at 10 units of time and C out would be available at 22 units of time we see the change of a and b.

So, from the change of a and b it will take 22 units of time to assign, to change the, to see the change in Cout. So, we have to judiciously use when to use blocking statement when to use non-blocking statement. Now, after understanding these basic fundamentals, let us see what kind of procedural statements we have.