Hardware Modeling using Verilog

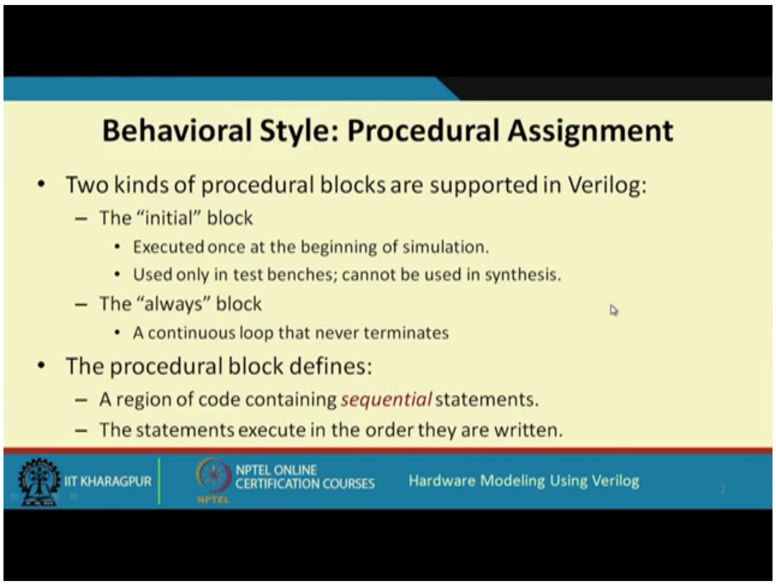
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 13
Procedural Assignment

So, in the last lecture we looked at the data flow or the continuous kind of assignment statement in Verilog, and how they can use to model both combination and sequential circuits. Now, today we shall be looking at the other kind of assignment statement which is the so called procedural assignment statement.

And we shall see the different types and how we can use it in the Verilog language.

(Refer Slide Time: 00:46)



Behavioral Style: Procedural Assignment

- Two kinds of procedural blocks are supported in Verilog:
  - The "initial" block
    - Executed once at the beginning of simulation.
    - Used only in test benches; cannot be used in synthesis.
  - The "always" block
    - A continuous loop that never terminates
- The procedural block defines:
  - A region of code containing *sequential* statements.
  - The statements execute in the order they are written.

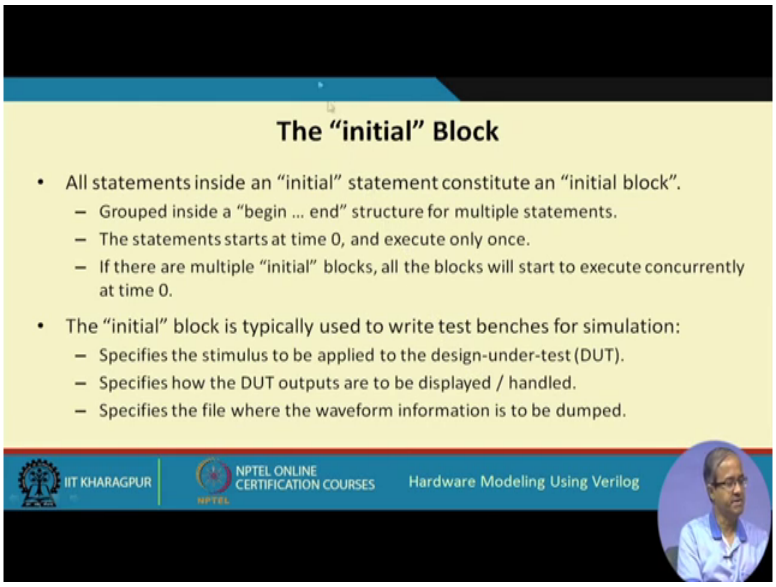
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

The behavioral style or the procedural assignment statement in Verilog comes in two flavors and we have already seen some examples using these. These are sometimes called procedural blocks, they come in two different types, one is the initial block other is the always block. The main difference between these two is the initial block is executed only once; and this is used exclusively for writing test benches or test harnesses and these are not used in synthesis. When your objective is to synthesize the circuit, you should not use the initial construct initial construct is to be used only when you are writing the test benches. It is executed only once at the beginning of the simulation.

And the other type is called always block, well this always block can be used inside the test bench or it can also be used inside your circuit description module. This is like a continuous loop, which never terminates. You see the always block models a hardware circuit in a more natural way whenever we design a hardware that is supposed to work as long as power is switched on. So, we cannot say that this circuit hardware circuit will be working for 10 seconds and then it will automatically stop, it will not stop unless you expressly send a single to stop it. So, the always block is like a piece of hardware which is taking some input generating some output and continuously it is doing it in a repetitive fashion; it never stops it is more like working in a continuous loop.

Now, these are called procedural blocks initial or always. Now, inside the procedural block you have some code which comprises of a set of sequential statements sequential statements means they execute in the order they appear or they are written one by one.

(Refer Slide Time: 03:15)



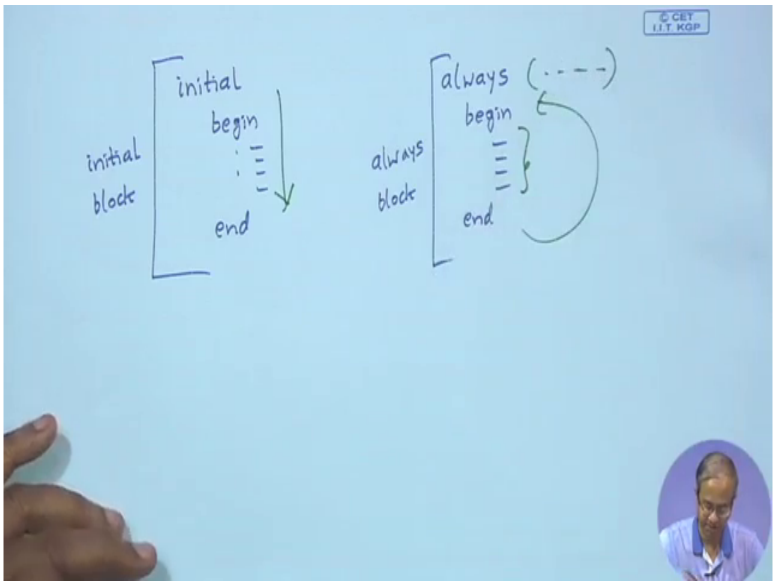
The "initial" Block

- All statements inside an "initial" statement constitute an "initial block".
  - Grouped inside a "begin … end" structure for multiple statements.
  - The statements starts at time 0, and execute only once.
  - If there are multiple "initial" blocks, all the blocks will start to execute concurrently at time 0.
- The "initial" block is typically used to write test benches for simulation:
  - Specifies the stimulus to be applied to the design-under-test (DUT).
  - Specifies how the DUT outputs are to be displayed / handled.
  - Specifies the file where the waveform information is to be dumped.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

So, let us see. First the initial block well we have already seen some examples of the initial block. We have seen some test benches, we shall see some elaborate some more complex test benches later again, but at least whatever we have seen. So, you can appreciate that this statements that appear inside the initial statement that is referred to as initial block, and that typically grouped between begin and end. So, inside and initial block it comes like this.

(Refer Slide Time: 03:56)

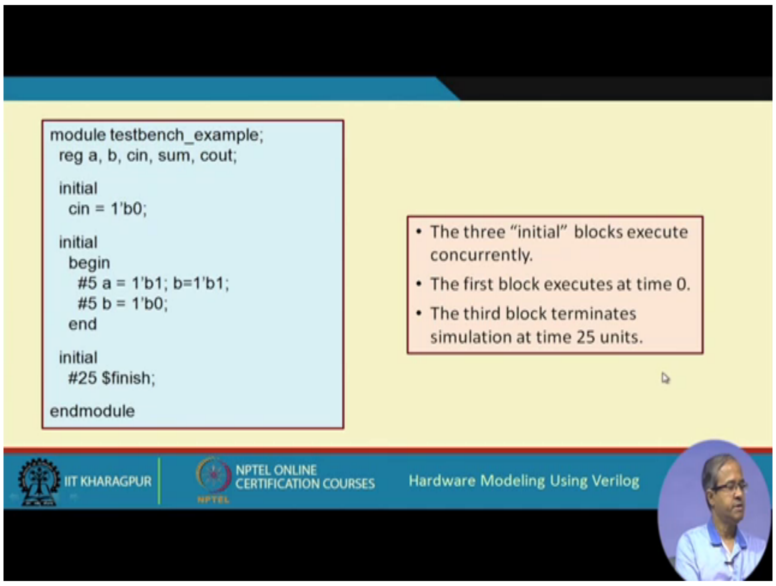


When you give initial, these are statement. So, inside it this statements are grouped in begin and end and this is what is defined as a initial block right. Now, the rule for execution is that the statements that is there inside this begin end, they start executing at time zero and they execute only once, this is important. So, if you do not specify any explicit timing by giving that hash command, it is assumed that the first statement inside that block will start executing at times t equal to 0, and initial means this statements will be executing only once, fine. And in a typical test bench there can be multiple initial blocks. And if there are multiple initial blocks, then all the blocks will be executing in parallel concurrently and all of them will start at time t equal to 0, this is something you should remember.

Now, as I have said repeatedly that this initial block is used to write test benches. But sometimes when you are trying to see how some typical Verilog constructs are working you can also use initial block just for testing, you can do some computation, you can give a display or a monitor statement to print it, just to see whether it is working or not. But not for synthesis, when you are designing and you are target is to generate the hardware, you can forget about initial, initial is something which you should not and cannot use fine.

So, this initial block is used for writing the test benches. The test benches as you have seen they basically specify this stimulus or the inputs that are to be applied to your design whatever you have designed. And also it is specifies how the output of your design are to be handled whether they are to be displayed or they are to be dumped into a file, so that you can display the wave forms later. This test bench specifies all these things and they are typically done inside always blocks always means initial blocks.

(Refer Slide Time: 06:30)



```
module testbench_example;
  reg a, b, cin, sum, cout;

  initial
    cin = 1'b0;

  initial
    begin
      #5 a = 1'b1; b=1'b1;
      #5 b = 1'b0;
    end

  initial
    #25 $finish;

endmodule
```

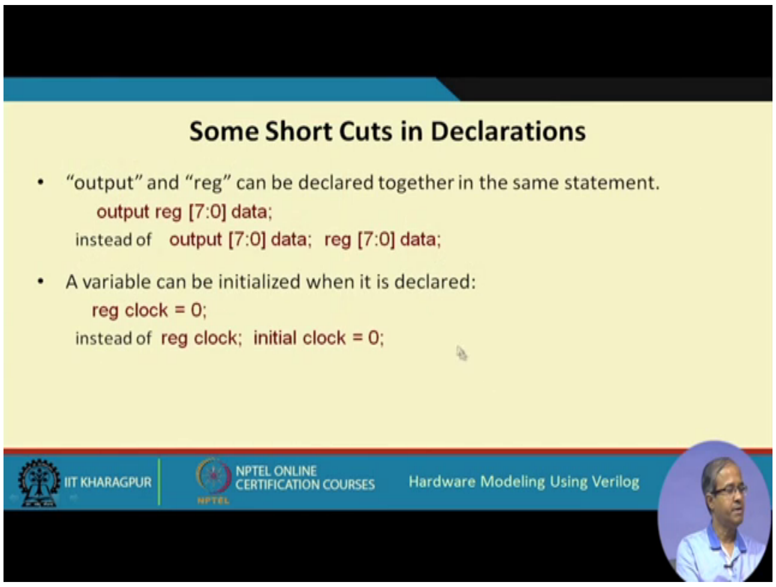
- The three "initial" blocks execute concurrently.
- The first block executes at time 0.
- The third block terminates simulation at time 25 units.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, some simple example, so this is the example of a test bench where I am not instantiating the model because our objective is to just to illustrate the initial blocks. Here you see we have used three initial blocks. And if there are multiple statements inside an initial block, you need this begin and end, but if there is a single statement, you do not need the begin and end, it is optional. So, all the three initial blocks, they start execution at time t equal to 0. You see what is happening at t equal to 0, this is something like a full adder, you see the names are quite familiar a, b, carry in sum and carry out.

So, I am initializing with the carry in as 0 and in the other initial block which also starts at time t equal to 0, it encounters a delay of 5. So, it waits for 5 units and then it then it applies a equal to 1, and b equal to 1. Then it again waits another 5 units and then applies b equal to 0 and that is it this is a single execution. And the third initial block this also starts at time t equal to 0, it waits for 25 time units and then calls the system function finish which means the end of simulation. So, at time t equal to 25, this simulation will stop. So, such multiple means initial blocks a very typical inside a test bench. Now, inside a test bench you can also have always blocks as you shall see later with some examples. So, as I have set to summarize the three initial blocks execute concurrently; the first block because there is no delays, this statement executes at time t 0, the third block calls finish. So, it terminates simulation at time 25.

(Refer Slide Time: 08:37)



Some Short Cuts in Declarations

- "output" and "reg" can be declared together in the same statement.
  output reg [7:0] data;
  instead of output [7:0] data; reg [7:0] data;
- A variable can be initialized when it is declared:
  reg clock = 0;
  instead of reg clock; initial clock = 0;

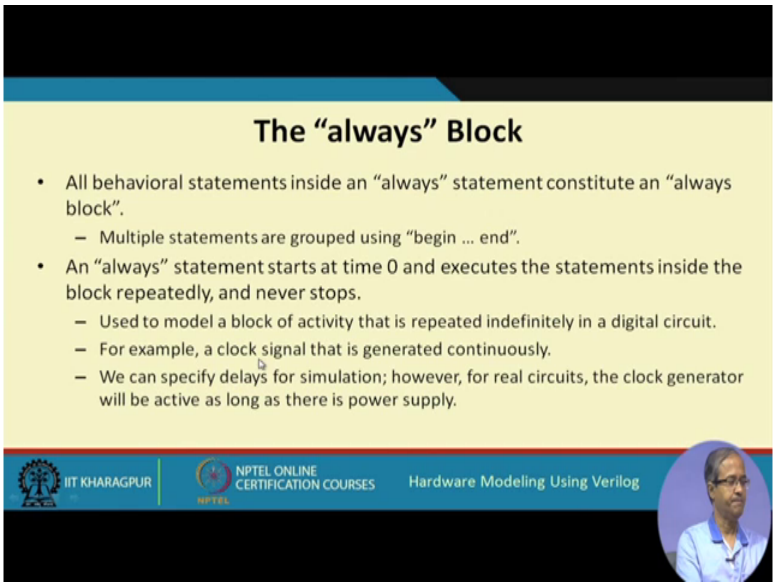
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

Now, just to note that there are some short cuts and declarations, which are supported by Verilog means at least the recent version of Verilog, which are available now. See normally whenever you have some parameter in a module which is an output and which is assigned inside a procedural statement which has to be a reg, you declare it like this output, let us say it is a vector 7 to 0 data. And again you have to give reg 7 to 0, a data to indicate that this data is also a registered type, so that it can be assigned inside a procedural block later on in the code.

But Verilog now permits us to combine this two statements in a single statement like this you can straight away right output reg 7 0 data. So, you are specifying two things together, the data is an output and also it is a registered type variable. Suddenly, see you use to write something like this say a clock, you declare a clock as a register type variable then in initial block, you initialize clock to 0. Suppose, this is the single statement inside the initial block, now this initialization and declaration can be done together in a single statement now you can write something like this reg clock equal to 0. Because here also you see clock equal to 0, initialization will take place at time t equal to

0, because no delay is specified; same thing will happen here whenever you are declaring clock at time t equal to 0, it is also initialized fine.

(Refer Slide Time: 10:26)



The "always" Block

- All behavioral statements inside an "always" statement constitute an "always block".
  - Multiple statements are grouped using "begin ... end".
- An "always" statement starts at time 0 and executes the statements inside the block repeatedly, and never stops.
  - Used to model a block of activity that is repeated indefinitely in a digital circuit.
  - For example, a clock signal that is generated continuously.
  - We can specify delays for simulation; however, for real circuits, the clock generator will be active as long as there is power supply.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

Now, let us come to the always block the other kind of procedural block now there is a statement the always statement just like we said that there is an initial statement and it comprises as an initial block in exactly the same way there is an always statement inside the always statement typically you have begin end again. So, begin end can group several statements together and this whole thing is called a always block right. So, the set of statements which are behavioral statements inside they always they constituent, they always block. And as I have shown in that example, these block is grouped using begin and end.

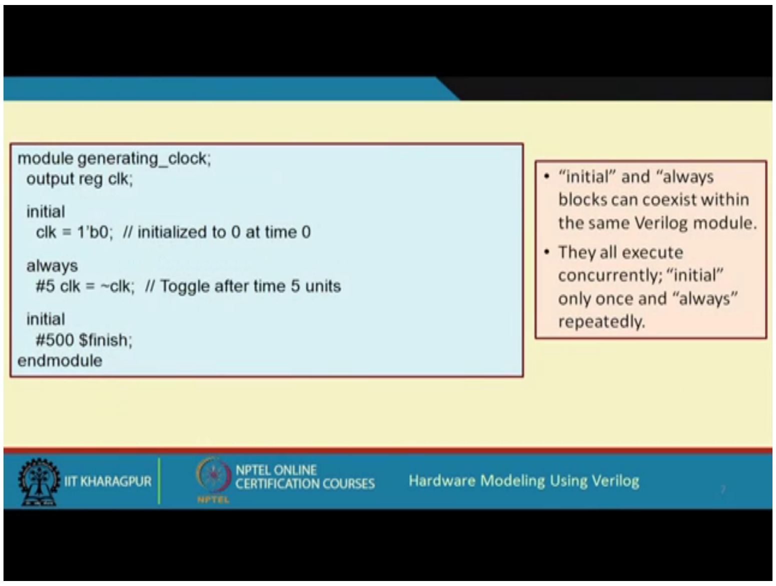
Now, just like initial this always block this statements inside it also starts at time t equal to 0 and executes this statement inside the block repeatedly and never stops. What I mean is that here inside this always block I have this begin end statement, there can be several statements here. Now, these statements are executed repeatedly in an infinite loop kind of a thing and it never stops. Initial executes only once, it executes once and stops, but always will never stop, it will execute the block repeatedly in response to some event condition that I will mention very shortly, this is the difference.

So, the always block is typical used to model some activity which is repeated indefinitely in a digital circuit which is characteristic of a digital circuit. So, as long as power is switched on it will continue. For example, I have clock generator as long as there is power the clock signal will go up and down up and down indefinitely it will never stop. So, only when I forcibly switch off the clock generator or switch off the power only then the clock will stop, fine. So, this is the example I have also given here the clock signal that is generated continuously.

Well for simulation purposes, we can specify delays even with the always block, but again when we are actually generating a hardware the clock generator this delays does not mean does not carry any meaning. So, as long as there is the power supply, it will go on of course, in simulation you can specify some delay we can say that after some delay it will stop, but in an actual hardware there is certain things that we cannot do that will see later on.

Because you see you have looked at so many different constructs in the language something which you can do only for simulation something you do typically for synthesis. So, we shall see later we will talk specifically about it, there is a subset of the language which of course, is a little subjective it can vary from one software or synthesis tool to another, there is a subset which is called synthesizable subset. There is a set of statements or set of features, which are only allowed by the synthesis tool. If you allow or use any other construct of the language this synthesis tool will simply ignore them it will not take them fine.

(Refer Slide Time: 14:28)



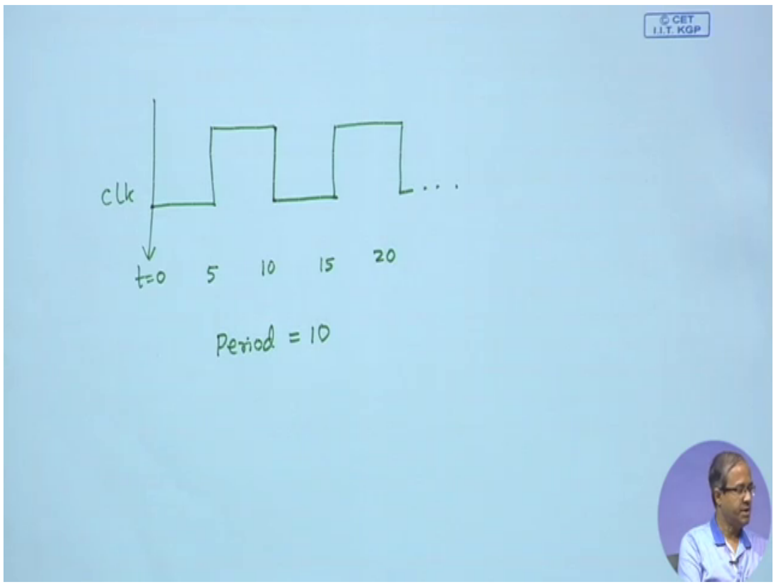
```
module generating_clock;
  output reg clk;

  initial
    clk = 1'b0;  // initialized to 0 at time 0

  always
    #5 clk = ~clk;  // Toggle after time 5 units

  initial
    #500 $finish;
endmodule
```

- "initial" and "always blocks can coexist within the same Verilog module.
- They all execute concurrently; "initial" only once and "always" repeatedly.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, here is an example just a clock generator. So, this is just a module just like a test bench which is generating a clock signal, this an output reg. You see this initial block at time t equal to 0 is initializing the clock signal to 0. So, initialize to 0 at time 0. And in the test bench I can also give an always block. I have given a always block which says that with the delay of 5, you use clock equal to not clock. What does this mean?

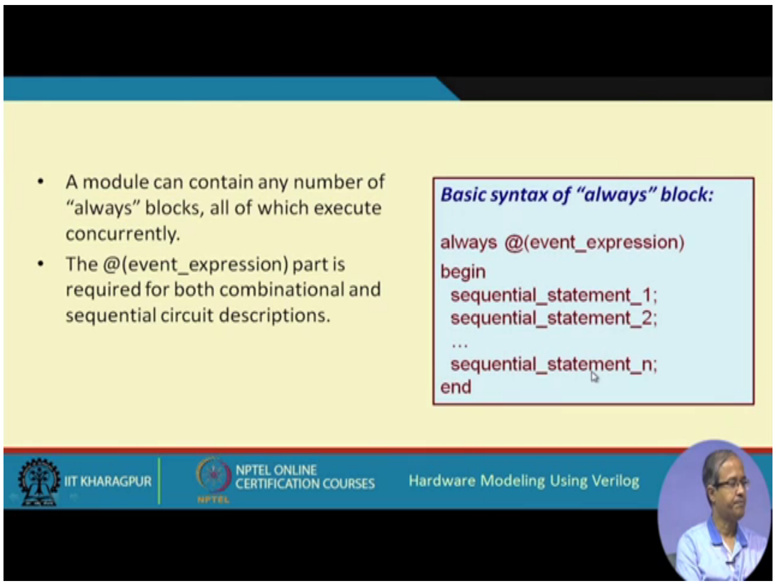
(Refer Slide Time: 15:13)



Suppose this is my time t equal to 0, and this is my signal clock, my clock was 0 at time t. At time 5 clock equal to not clock my clock becomes high; at time 10, after gap of another five again clock equal to not clock, it will become 0. At time 15, it will again become 1; at time twenty it will again become 0 and so on. So, with the gap of five the clock will continually toggle and the time period will be double the delay, it will be 10; 5 off period and 5 on period. So, this is how we specify that after every five time the clock will toggle from 0 to 1, and 1 to 0 and just for simulation we can specify this that at time at time 500 this simulation should finish. So, the clock will go on toggling up to time 500.

So, inside a Verilog module initial and always blocks can coexist you have a combination of initial and always blocks, but of course, within the test bench only not within the main Verilog module. And they all execute concurrently. The difference I mentioned that initial will execute only once, but the always block will execute indefinitely repeatedly.

(Refer Slide Time: 16:49)



- A module can contain any number of "always" blocks, all of which execute concurrently.
- The @(event_expression) part is required for both combinational and sequential circuit descriptions.

Basic syntax of "always" block:

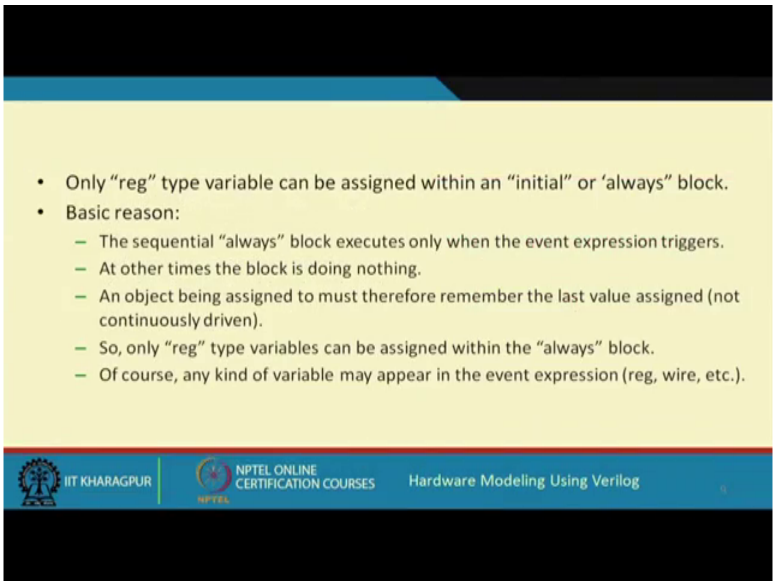
```
always @(event_expression)
begin
  sequential_statement_1;
  sequential_statement_2;
  ...
  sequential_statement_n;
end
```

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

So, a module in general can contain any number of always blocks I mean I mean a module where I am describing the hardware and these blocks are all executing concurrently. Now, I told you just sometime back, this always block will be executing in response to some event, you see this is the general syntax of an always block, after always you give this at the rate symbol and within bracket you give an event expression.

So, whenever this event expression occurs or takes place, this statements inside the begin end block will execute sequentially. And this event expression this is required for modelling both combinational sequential circuits as we will see through examples slowly. So, this is the general syntax. There are a number of sequential statements inside the always block and every time this block will get executed whenever this event expression triggers. So, let say this event expression can be a clock. So, every time a clock is coming it will execute; again next time clock comes again it will execute something like that.

(Refer Slide Time: 18:07)

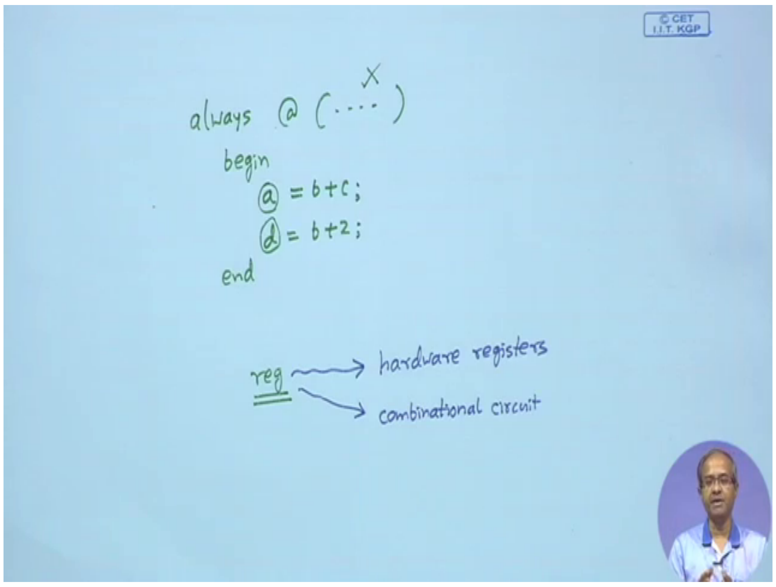


- Only "reg" type variable can be assigned within an "initial" or 'always" block.
- Basic reason:
  - The sequential "always" block executes only when the event expression triggers.
  - At other times the block is doing nothing.
  - An object being assigned to must therefore remember the last value assigned (not continuously driven).
  - So, only "reg" type variables can be assigned within the "always" block.
  - Of course, any kind of variable may appear in the event expression (reg, wire, etc.).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

And there is some restrictions here. So, inside the initial or always block, whenever you are assigning some variables to some values only reg type variables can appear on the left hand side. You can assign values only to reg type variables. The basic reason is that you see the sequential always block as I had said it will execute only when the event expression triggers like.

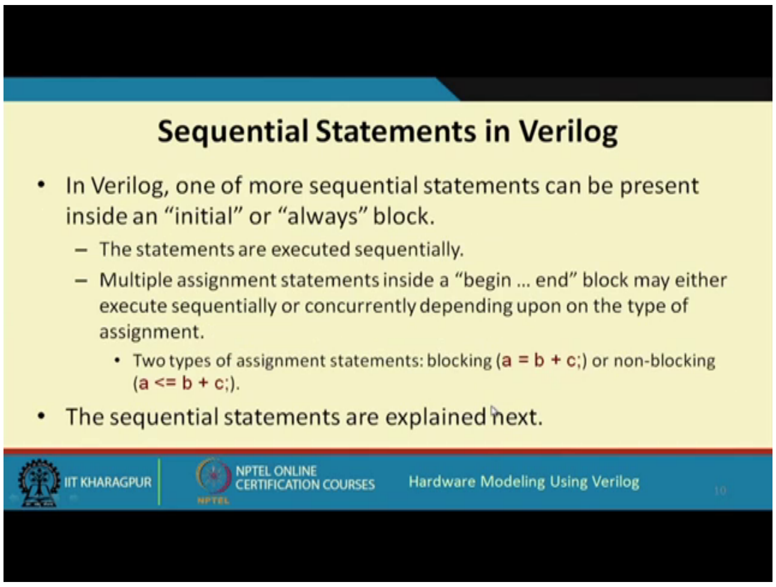
(Refer Slide Time: 18:40)



What I have said is that always at the rate some event expression I do not care what it is what I am saying is that inside the begin end block let say I am writing some statement like this a equal to b plus c, I write d equal to b plus 2 something like this. So, whenever this event expression will appear or will trigger, these two statements will execute, but now think what will happen during the time when this triggering condition is not true. So, what will happen to the statements whatever you have assigned to a and d should remain in a and d that is why we say that the left hand side must naturally be reg type variables. So that if required this reg type variables can be mapped to hardware registers by the synthesizer.

But it is not necessarily true, sometimes you will see that the variable may be reg type variable, but whatever you are mapping, whenever you are mapping it, so it is actually mapping to a combinational circuit. So, no registers or storage elements are required, flip-flops latches are not required. So, a reg can map to either a sequential element or a combinational circuit, but if we use a net variable net variable can never be sequential it will always be combinational right. So, this is what I mentioned the object which you are assigning must remember the last file you assigned this is not continuously driven. So, you must have reg type variables in the left hand side, but on the right hand side or in the event expression you have a any combination of variables reg, wire etcetera.

(Refer Slide Time: 20:50)



Sequential Statements in Verilog

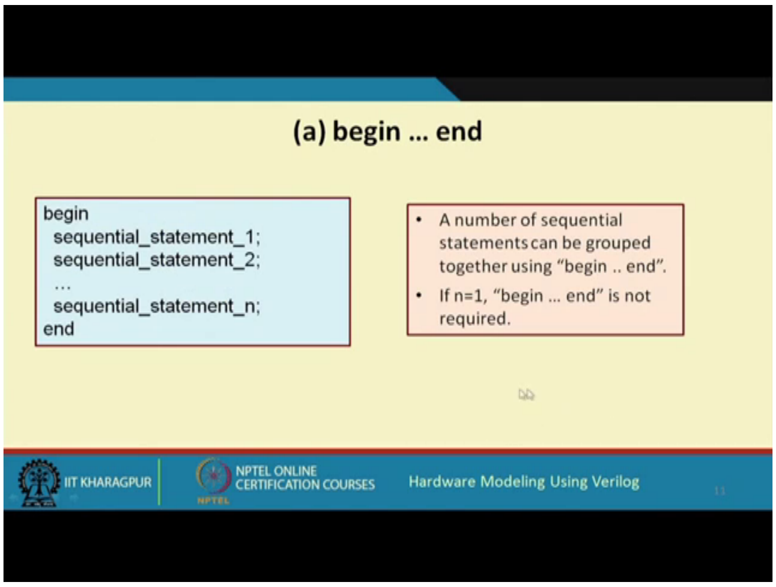
- In Verilog, one of more sequential statements can be present inside an "initial" or "always" block.
  - The statements are executed sequentially.
  - Multiple assignment statements inside a "begin ... end" block may either execute sequentially or concurrently depending upon on the type of assignment.
    - Two types of assignment statements: blocking (a = b + c;) or non-blocking (a <= b + c;).
- The sequential statements are explained next.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

Now, let us come to the sequential statements in Verilog. So, we have said that inside Verilog or in the always block, you can have a number of sequential statements such sequential statements can also be there inside the initial block. They are supposed to be executing sequentially one by one. So, what is the idea, there are multiple assignments statements inside a begin end block may be there, which may execute sequentially or concurrently depending upon the type of assignment, this is something we will be discussing later. What you are saying is that inside a begin end block, there can be some assignment statements, these assignment statements may execute one by one sequentially or they may be executing concurrently, depending on what kind of assignment we are making, this we shall be seeing later.

There are two type of assignment statement - one is called blocking, which is denoted by the equal to symbol; and there is another type called non blocking, which is denoted by less than equal to or the arrow symbol. Now, the sequential statements, whatever is available in the Verilog language will be explained now one by one.

(Refer Slide Time: 22:20)



(a) begin ... end

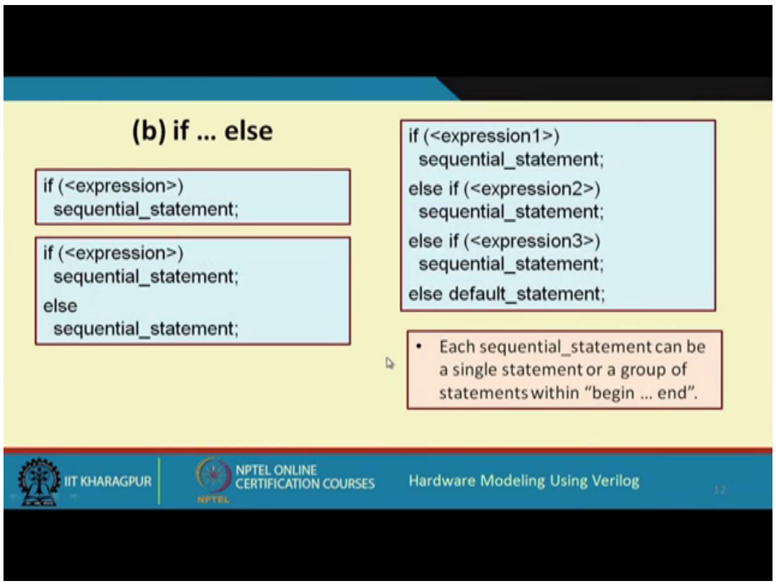
```
begin
  sequential_statement_1;
  sequential_statement_2;
  ...
  sequential_statement_n;
end
```

- A number of sequential statements can be grouped together using "begin .. end".
- If n=1, "begin ... end" is not required.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

The most important and the basic which you have already seen is the begin end. Begin end is the basic block, which combines a number of sequential statements in to one composite statement. So, a number of sequential statements can be grouped together. Now, if the number statement is 1, then the begin end keyword is not required, you can simply write this statement right.

(Refer Slide Time: 22:51)


(b) if ... else

```
if (<expression>)
  sequential_statement;
```

```
if (<expression>)
  sequential_statement;
else
  sequential_statement;
```

```
if (<expression1>)
  sequential_statement;
else if (<expression2>)
  sequential_statement;
else if (<expression3>)
  sequential_statement;
else default_statement;
```

- Each sequential_statement can be a single statement or a group of statements within "begin ... end".

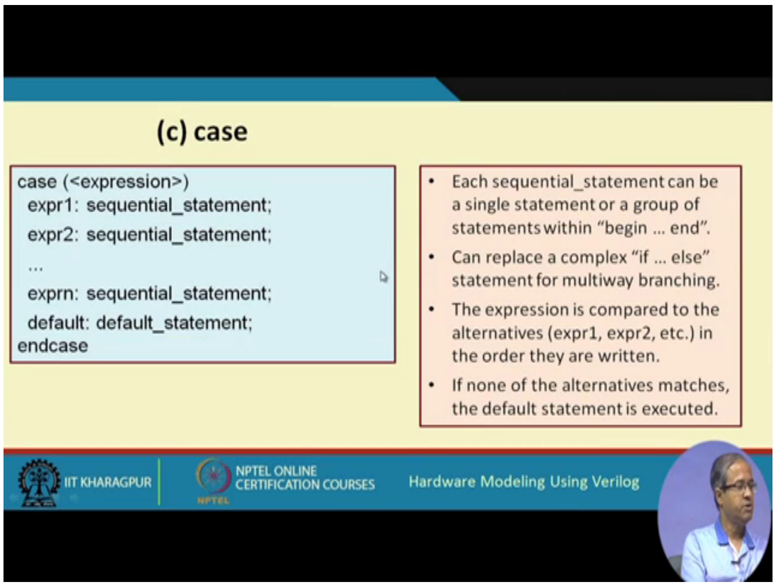
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Now, if else you see these constructs are very similar to the one, which are available in high languages like C. So, I am showing three different types of if else construct, one is a simple leaf without an else part. If some logical expression it can be true or false; if it is true then execute this sequential statement. Now, you see one thing, I have written just sequential statement. And just in the previous one I said begin end is also sequential statement, so that sequential statement itself can be this begin end block. So, although I have written a single sequential statement, os there can be begin and end inside if and there can be five statements there. Several statements can be here with some begin end.

So, in the second version there can be an else. If this logical expression is true, then this block or sequential statement will execute it, else this block will be executed. So, you can have a begin end here, you can again have a begin end here. In a third version we are showing nested if then else if else. It says if expression 1 is true, then you execute this sequential statement, else if expression 2 is true then execute this, else if this is true then execute this otherwise default statement. So, you see the expressions are checked in a particular order.

The first match whatever you get that particular statement will be executing. So, inside this block exactly one of the statements will execute either this one or this one or this is one and if none of them match the default one. And here as I had said that each sequential statement can be either a single statement on its own or they can be grouped using begin end. Now, we can have a more concise version of this if then because you see if then else nested one, it becomes quite complex in terms of the structure.

(Refer Slide Time: 25:18)



(c) case

```
case (<expression>)
  expr1: sequential_statement;
  expr2: sequential_statement;
  ...
  exprn: sequential_statement;
  default: default_statement;
endcase
```

- Each sequential_statement can be a single statement or a group of statements within "begin ... end".
- Can replace a complex "if ... else" statement for multiway branching.
- The expression is compared to the alternatives (expr1, expr2, etc.) in the order they are written.
- If none of the alternatives matches, the default statement is executed.

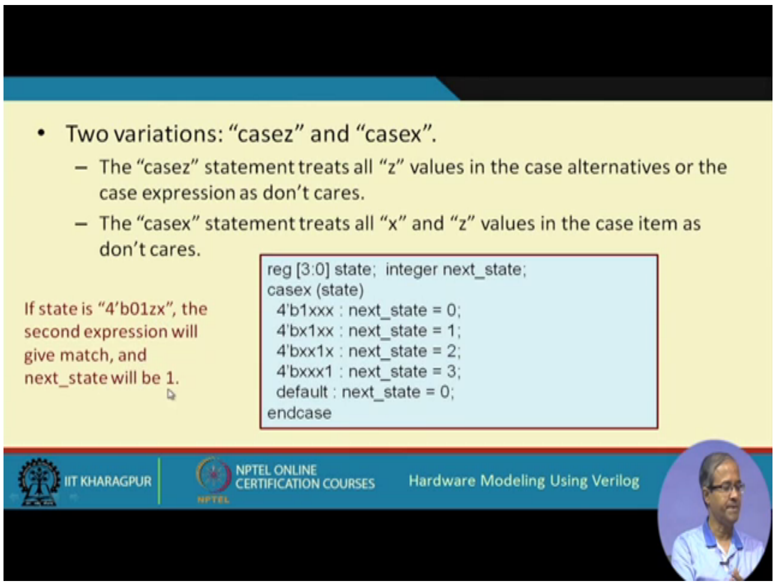
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, you can have a concise one well in C language you have a similar construct called switch case; in Verilog, we have case. Here you say that we give an arithmetic expression inside this case with in bracket. So, it evaluates to a value and that values compared with these expressions you just specified before the colon. So, it compares with this expression sequentially one by one, it first compares with expression one, then expression two, expression three up to expression n.

And whichever it is matching that corresponding sequential statement will be executing. And if none of them matches then there can be a default then this default statement will be executing and case will end with end case. So, each of this sequential statements like the earlier ones, they can be either a single statement or it can be a group of statement inside begin else. So, this becomes much more simpler as compared to a complex if else kind of a construct.

Now, the order in which you list this expressions, they will actually give you the order in which you are checking because you see the same expression the way you are comparing, it can match with more than one condition, there can be some don't care values also; so the first one that matches will be the one that will be taken.

(Refer Slide Time: 26:57)



- Two variations: "casez" and "casex".
  - The "casez" statement treats all "z" values in the case alternatives or the case expression as don't cares.
  - The "casex" statement treats all "x" and "z" values in the case item as don't cares.

If state is "4'b01zx", the second expression will give match, and next_state will be 1.

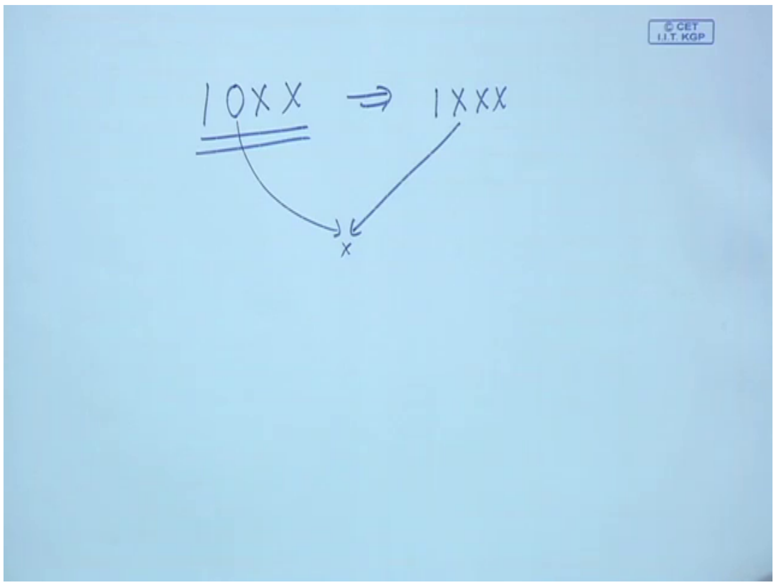
```
reg [3:0] state;  integer next_state;
casex (state)
  4'b1xxx : next_state = 0;
  4'bx1xx : next_state = 1;
  4'bxx1x : next_state = 2;
  4'bxxx1 : next_state = 3;
  default : next_state = 0;
endcase
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

Like there are two variations of case you can have in Verilog, one is called case z, others called case x. In case z statement, here the high impedance or the z values in the inputs and the expressions are treated as don't cares, but in case x not only z also this x they are treated as don't cares. Let us take a small example. Suppose I have declared a state vector four bit next state is an integer I am using a case x on the state four bit. And you see the values of the expressions I am specifying they have don't cares, they are not crisp values I am saying one don't care, don't care, don't care, x 1 x x, x x 1 x and here last one is one.

So, depending on which one here I am saying that if the first bit is one then next state is zero you do this; if the second bit is one, you do this. But if none of this matches; that means, none of the bits are one then default something you can say here I am saying next state zero.

(Refer Slide Time: 28:31)



You see if your state has a value let say 1 0 x x, this will match with the first condition which is 1 x x x because 0 and x matches x is don’t care, and 0 is one possible value of the don’t care. But if you give only case then this will not match, but if you give case x then only the first bit is in the state is one or not other three can be anything need not be x x x need be zero 1 x 0 1 z anything. So, this is how you can give. So, there are several other kind of constructs also. So, we have only seen a few begin end, if then else, case. So, in our this what I am saying this state is this, this has an example 0 1 z xm this second expression will be matching 0 1 z x because x matches with anything and next state will one.

So, in the next lecture we shall be continuing with this. We will see that in Verilog, there are some other kind of sequential statements also available because we have so far talked about begin end, if then else, if else, nested if else and case and some versions of case. We have not talked about the different looping constructs for, while and some others. So, in the next lecture, we shall be talking about those and also we shall be seeing some examples.

Thank you.