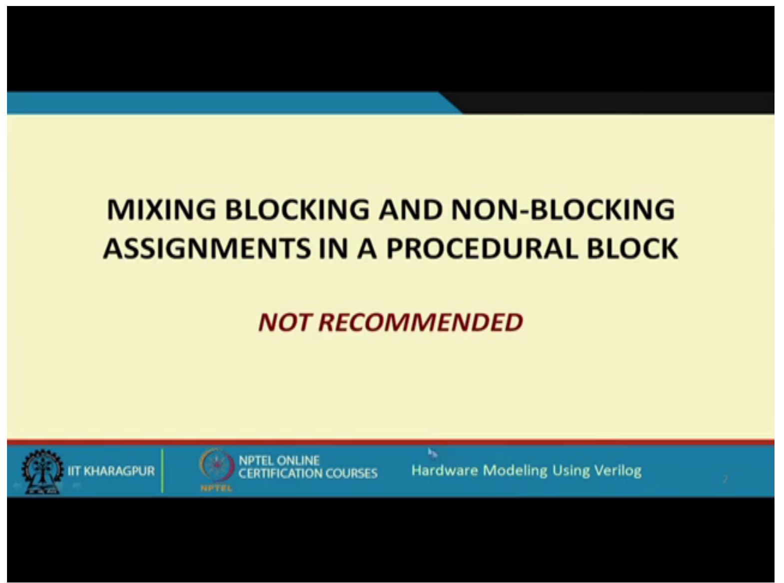
Hardware Modeling using Verilog

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

Lecture - 19
Blocking / Non-Blocking Assignments (Part 4)

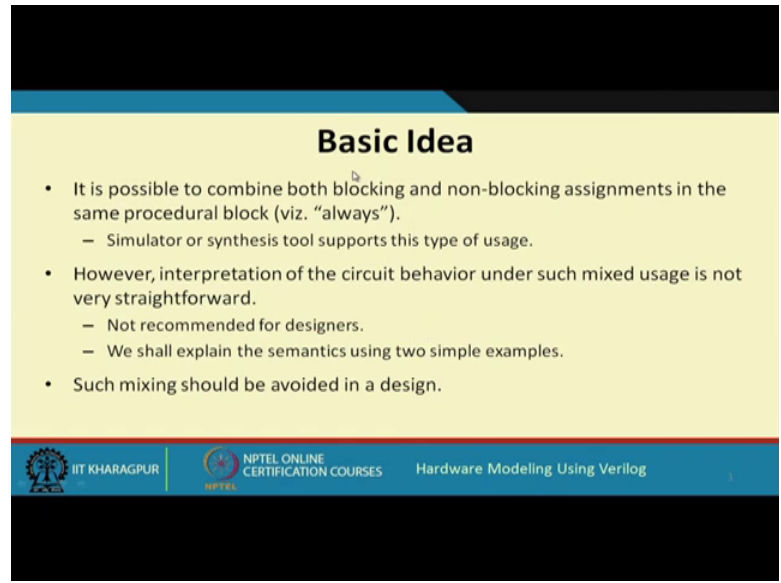
So, in this lecture we shall be looking at some features of Blocking and Non-Blocking Assignments which are not recommended to be used. Like mixing both blocking and non-blocking statements in the same procedural block, this is the first thing. We shall be looking at and then we shall be talking about a feature in a verilog, there is a statement called generate with which you can automatically and dynamically generate code in verilog. So, we shall see these features in this lecture, let us see.

(Refer Slide Time: 01:02)



So, the first thing that we will be looking at very briefly because this is not recommended I shall not go into the detail this is mixing blocking and non-blocking assignments within an insider single procedural block. This I am not going into the detail because this is a very bad design practice ok.

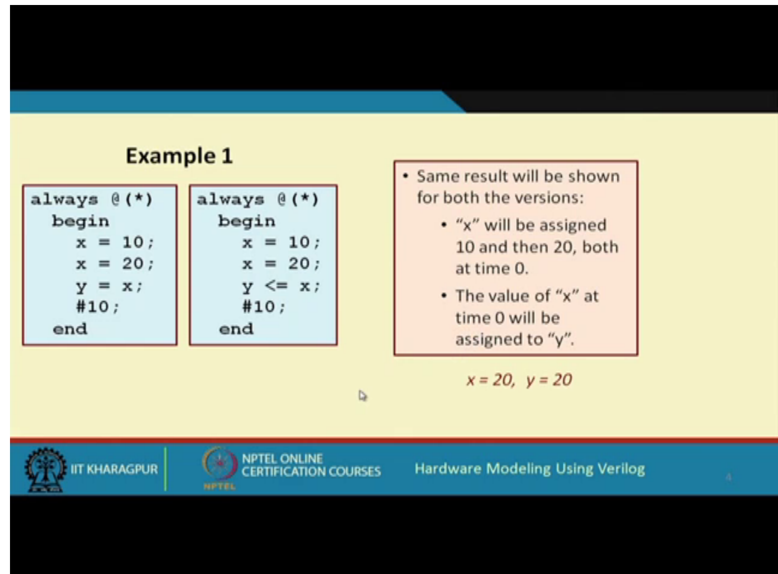
(Refer Slide Time: 01:27)



So, the idea is that the simulation and the synthesis tools that are available, this support using both blocking and non-blocking assignments inside this same always our initial procedural blocks. But again like the examples that you took in the last lecture there we looked at only blocking assignment still then we saw that there are a lot of confusions, the order of the statements that you put that becomes very important which is not So for non-blocking, but if you mix them up the problem becomes even more complicated right.

So, so if you mix them up the interpretation exactly what the circuit is supposed to behave like is not very straightforward and hence it is not recommended. So, we shall be looking at 2 very simple examples, and I am repeating such mixings should be avoided in an actual design ok.

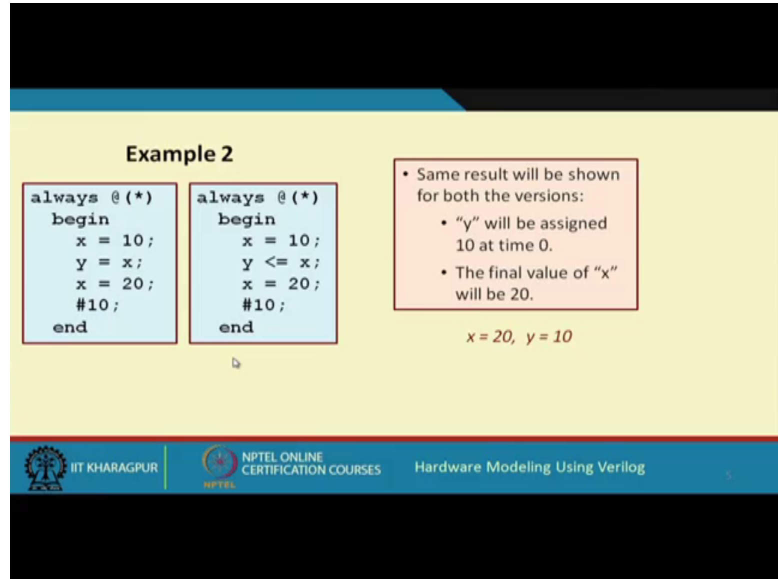
(Refer Slide Time: 02:31)



So, let us take a simple example like this. In the first example we are using only blocking type assignments, always x star x equal to 10 x equal to 20 y equal to x and then a delay of 10. Here and this is repeating, and here the last one we are using a non-blocking assignment. You see for both this k what is the meaning blocking assignment means you first execute the first statement 10 is assigned to x, then execute the second statement 20 is assigned to x, then execute the third statement 20 is assigned to y. So, this x will get 20 and finally, y will also get 20, both x and y will get 20. Now in the second case you see the first 2 are blocking third one is non-blocking. So, the blocking statements because we are not given in a delay, they will be executing at time t equal to 0. So, at time t equal to 0 x will be getting 10, and then x will be getting 20 in this order.

But both will happen at time t equal to 0 only. So, the final value of x will be take will be 20. And this non-blocking assignment will take or evaluate the right hand side at time t equal to 0. And at time t equal to 0 this x has already become 20 because the time has not advanced, time is still t equal to 0 inside this loop. So, that latest value of 20 will be taken and the same 20 will be assigned, you see this is actually quite confusing you should not mix it like this, but if you use this you will see if you simulate this you can see that it is getting, x equal to 20 y equal to 20 same value we are getting right. So, this is one example.

(Refer Slide Time: 04:44)

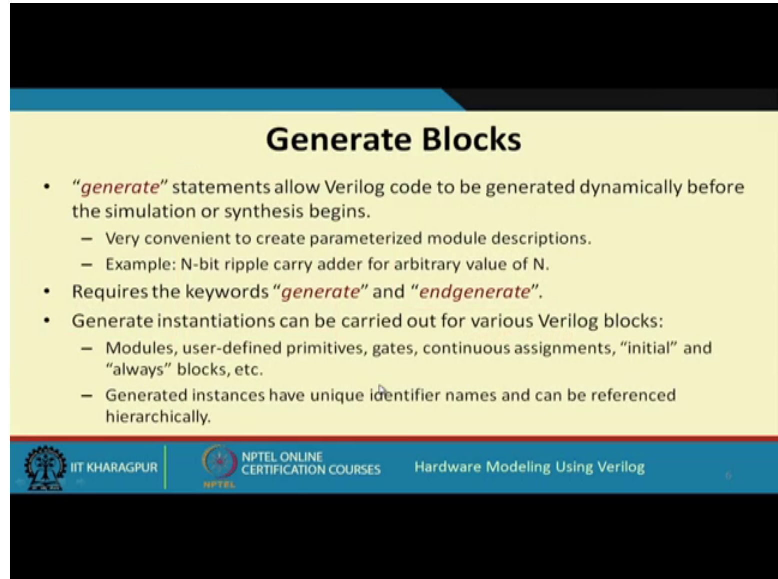


And the other example is that slightly different example x equal to 10 y equal to x, x equal to 20. And here the middle statement we have changed it to a non-blocking assignment, here let us see. The first one let us try to interpret, these are all blocking assignments first x gets 10, then 10 gets assigned to y, then 20 gets assigned to x.

So, the final value of x will be 20 and the value of y will be 10, but here what will happen here you see the order is important. The blocking assignment will interpret is that well once everything before me has finished only then I will start. So, x will get the value 10 then it will wait because there is something before it. So, this 10 will be assigned to y, then this will come to into assign to x. So, this is again very confusing.

Now this simulator interprets it like this if you simulate it using I (Refer Time: 06:02) you will see that you simulate both the designs will be getting the same result x equal to 20 and y equal to 10, but I am repeating you should not use this kind of mixing inside a procedural block. So, you will see some examples later you will see that you can have multiple always blocks. So, in one of the always blocks you can use only non-blocking assignments in some other or always block you can use only blocking assignments. Such things are sometimes required in design. We shall see later when you talk about designing sequential circuits designing finite state machines and so on fine.

(Refer Slide Time: 06:47)



So, we shall not be discussing any more on this. Now we come to a very useful feature that is available in very log which is called generate. Let me try to give you a brief motivation what is this all about. Well, we took an example of a ripple carry order earlier if you recall. So, a ripple carry adder is constructed by simply connecting a number of full adults in cascade. The carry out of a full adder is connected to the carrying of a full adder the carry out of the second is connected to the carrying of the third and so on.

So, if I want to design a 4 bit ripple carry adder I will be connecting 4 such full adders. Now if I want an 8 bit can you look at adder I will be connecting 8 such, suppose I tell you I wanted to design a 32 bit ripple carry adder then means using the way which you have learned. So, far you will have to instantiate the full adder 32 times. Isn't it? 32 copies of full adders are required and they will be connected like that you will have to mentioning the carry values also, carry out will be carrying of the next one.

So, 32 lines of code you have to write. Now you may think, so isn't it quite logical that the language should provide a feature of some kind of alteration? Or loop so I can say that I want 32 copies of full adder. I specified only once and 32 copies will be generated. So, this generate block does exactly that it dynamically creates multiple copies of your designer specification whatever written in verilog to suit a particular design. So, we shall be looking at some examples, generate blocks. So, there is a statement called generate is a keyword. General statements as I mentioned, this allows this simulator or the

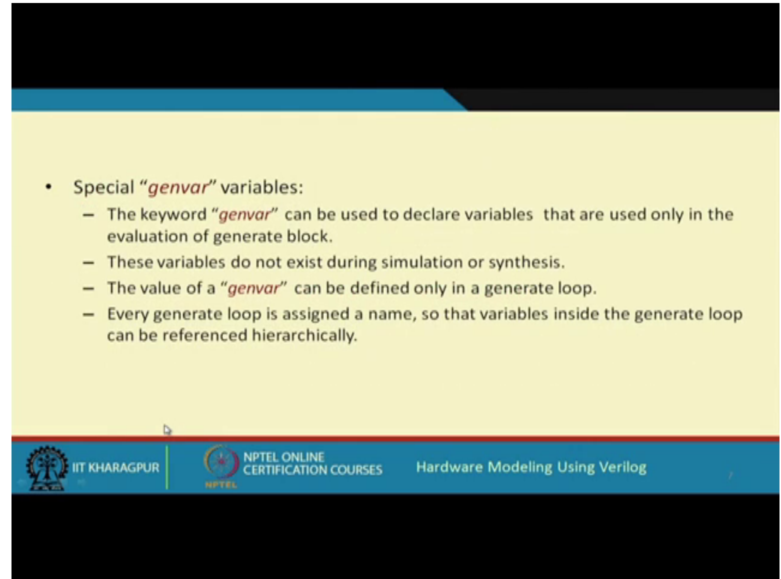
synthesizer to generate cerilog code dynamically before they can be used. This is a very convenient feature to create so called parameterised module descriptions.

So, I mentioned this parameters earlier you can define a constant. So, you can define an N-bit ripple carry adder for arbitrary value of n. This is an example of a pediment of a parameterised design. N is a parameter you change the value of n you get a 4 bit adder 8 bit adder or 16 bit adder, whatever you want just change the value of n, but the verilog code remains the same you need not have to change the code. The generate block will automatically do it for you right. So, there are 2 keywords you require generate and endgenerate.

So, what generate block does is it creates some kind of instantiation of verilog blocks, number of times as you want. Now this verilog blocks that you can repeat or you can generate, they can consist of modules user driven primitives, gates, continuous assignments, initial block always block anything almost any kind of blocks you can use for this kind of repetitive generation. And also shall see through some examples that when you create say for example, multiple copies of the full adder.

So, each copy of the full adder will be given some name, So that you can refer to the signals in for example, full adder number 5 the sum of the full adder number 5 you can refer to by a name that name dot sum which will mean that will be sum of that particular full adder we will see how to do it. So, the generated instances will have some unique identifier names which you can refer. And these names are typical typically hierarchical.

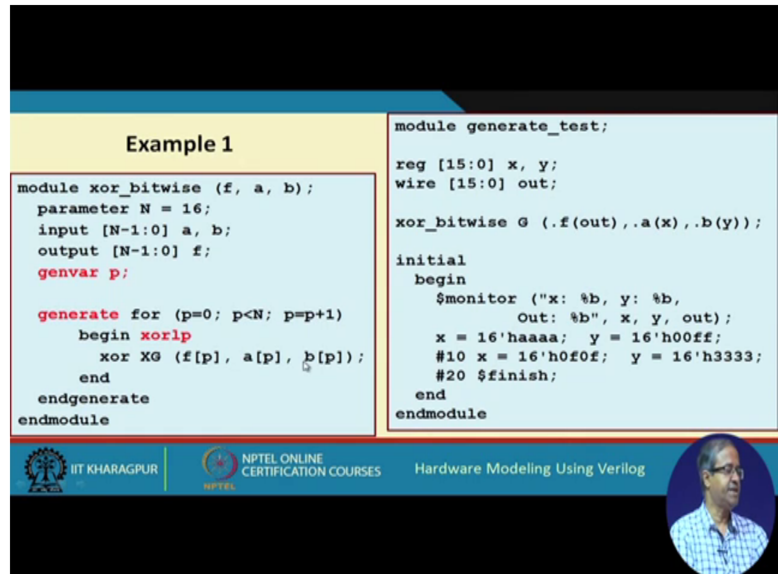
(Refer Slide Time: 11:22)



So, in order to use generate you need some special variables which tell you how many times to repeat or generate, these are called genvar type variables. The keyword genvar is used for the purpose.

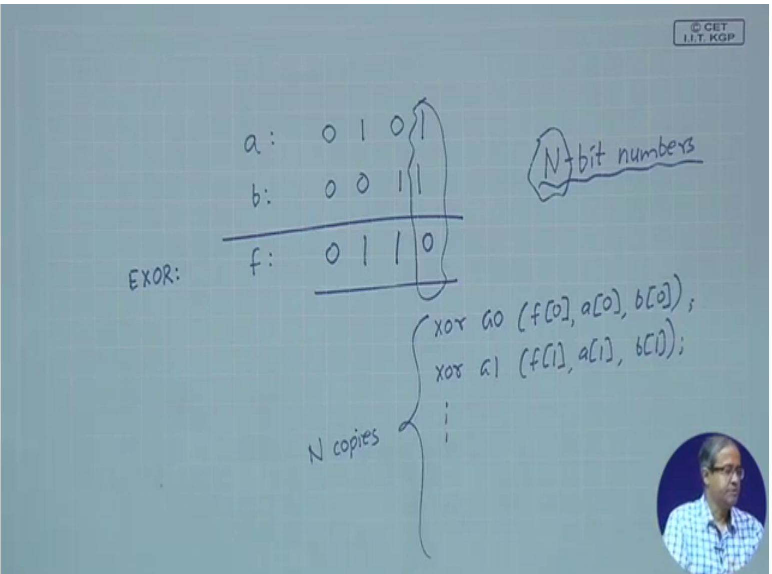
So, you can use this keyword to declare such variables, which are used only for the generation of the blocks. These variables will be ignored during simulation or synthesis. These variables will only be required or used during the generation of the code right. And these general variables they are defined only within our generate loop and can be used there and means, I mentioned earlier. That you can have a hierarchical naming convention, that every generate loop you define you can assign a name to it. And that name by default can be used to refer to the variables of the generated copies. We shall see some examples how to do it, but these are the basic ideas.

(Refer Slide Time: 12:44)



So, let us straight a look at an example this is the best way to think. Here we are taking a very simple example a very simple example of a bitwise exclusive or. Like let us say what I am saying I am taking a problem.

(Refer Slide Time: 03:11)



Let us say I have a variable a, I have a variable b, which is multi bit variable let us say this is 0 1, 0 1 this value is 0 0 1 1, let us say and suppose I am wanting to carry out bit by bit exclusive or of these bits. And the result let us call it f. So, 1 and 1 if you take an exclusive r it is 0 0 1 1 will be 1 1 and 0 will be 1 0 and 0 will be 0. So, this bitwise

EXOR I want to do a parameterize design using genvar using generate, where I can use it for any arbitrary n for any arbitrary N-bit numbers.

So, the idea is that if you want to do one EXOR one bit EXOR you need to make one instantiation of an xor gate. Like for example, a b or that you can write xor you can give the name say a name z 0 that you can write f 0 a 0 b 0, like this then for the second bit you can generate another xor gate xor g 1 f 1 a 1 b 1 and so on. So, what I am saying?

Now, is that we want to define a parameter n. So, that N copies of such xor's will be automatically generated. So, I do not have to write so many xors. This n can be 32, 64 whatever right. So, this is the example let us see this is my verilog code and this is the test bench this is the verilog code. So, whatever written let us see this is the name of the module, 3 parameters f, a and b. And here you have defined a parameter n which for the sake of example we have taken it to be 16.

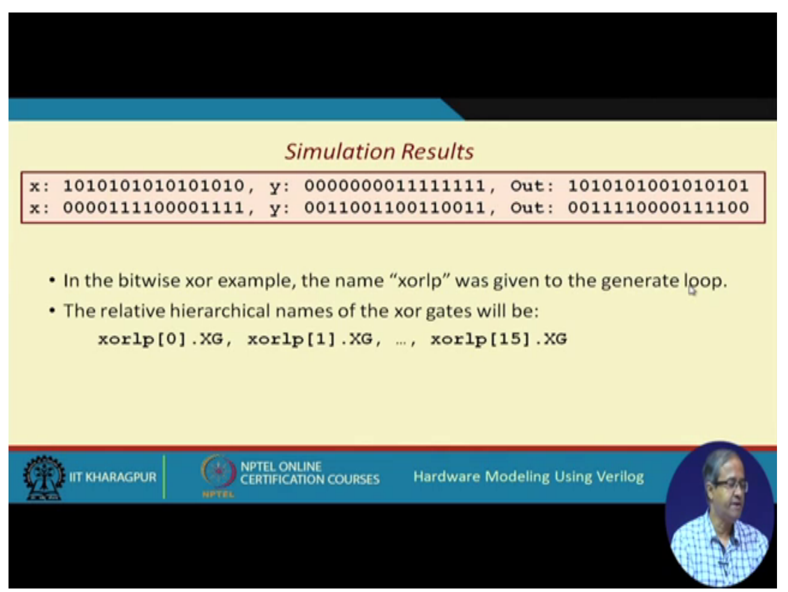
So, a and b these are the inputs we are declaring it to be inputs from bit number 0 up to n minus 1 a and b, and similarly f is the output similarly as a vector 0 to n minus 1. And we are declaring a genvar p which p is a variable which we will be using inside this loop here you see use p. And this is how we can use a generate loop, generate a for loop. Generate for p equal to 0 p less than n p equal to p plus 1 it is exactly like a for loop, but it starts with a keyword generate and it ends with endgenerate right.

So, what it will mean is that whatever is there inside this begin end this will be instantiated n times 0 up to n minus 1. So, what I given inside this loop, you see I have defined a label. Xor l p is the label, I have given a label of this loop. And inside this I have instantiated one xor gate xor I have given the name as x g f this p a p b p, you see here we are given f 0 is 0 b 0 f 1 a 1 b 1. So, generally I am calling it as p right. So, I call it b p a p f p this is all. So, is. So, so if I just write this and I give it to the verilog simulator or the synthesizer. It will be generating N copies of this xor gates. So, here I have written it only once. So, suppose n is 16 16 such xor lines will be generated automatically.

So, we have a means we can basically test this code out, using a test bench like this I have also showed a test bench you can try it out here. We have instantiated this xor bitwise function we just use this named convention of passing arguments. This f a b was here, and here in this test bench we are calling them x y and out. X y are 16 bits out of

the 16 bits. Initial we are monitoring x y and out, and we are just applying 2 sample inputs. The first input is that first input x you are applying a 16 bit (Refer Time: 18:18) number a a a a and why you are applying 0 0 f f. And after a delay of 10 we are applying x equal to 0 f 0 f and y is 3 3 3 3 finish.

(Refer Slide Time: 18:36)

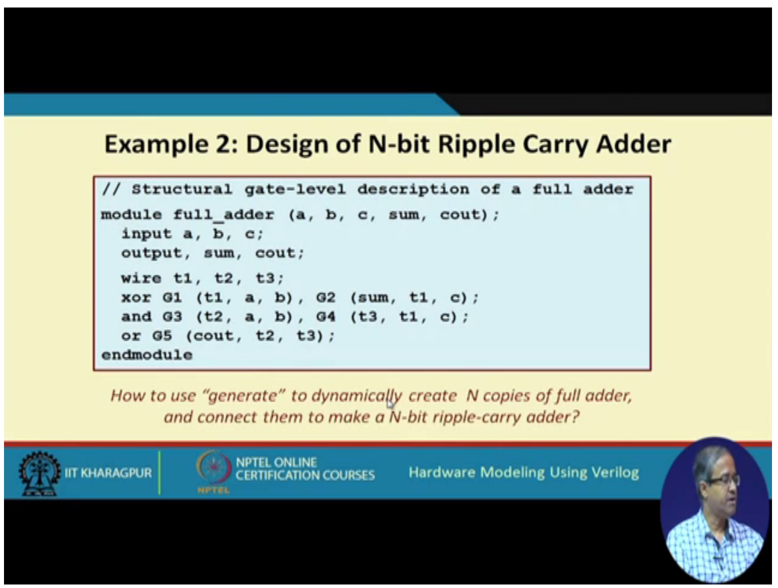


So, if you actually run this you get an output like this. You see a a a a and 0 0 f f. This a is 1 0 1 0, you see 1 0 0 a 1 0 1 0 1 0 1 0 1 0 0 and 0 0 if f. Bit by bit xor, 1 and 0 is 1 0 and 0 is 0. So, the first 8 bits would be like this 1 0 1 0, and the last one will all be inverted. 1 and 0 x or is 1 1 and 1 xor is 0 and so on.

So, this is one verify. And in the second test case we have given 0 f 0 f and 3 3 3 3. So, 0 and 3 if we xor it will be 3 f and 3 if we xor if you become c 1 1 0 0, then again 0 0 1 1 again 1 1 0 0. So, this is the simulation result. Now you see this xor l p and this xor. Now the point is that xor lp this was the name we had given to the generate loop right. So, these 16 copies of the xor gate that will be generated, we have given the name as xg. But there will be 16 such gates what will be the names. Because in this example I trying to write I was writing g 0 g 1 g 2 like this. But here I wrote it only once xg. So, what will be the name that will be generated for the 16 copies of these xor gates? They will be like this xor l p 0 dot xg xor l p 1 dot x g dot dot dot up to xor lp 15 dot x g.

So, whatever loop variable you have defined that will be treated as a vector and you can use it with an index. Put a dot and whatever variable name you have defined you can use it after that. So, like this you can uniquely refer to the 16 xor gates right.

(Refer Slide Time: 20:50)



Let us take another example, just like the ripple carry adder I talked about. Well, we start by giving you a structural description of a full adder. This is a gate level description of full adder that consists of 2 xor gates, 2 end gates and one or gate I showed this circuit earlier you can just check it out. This is a compact gate level representation of a full adder. So, the first 3 parameters are the inputs then some and carry out. So, what do you want to do now?

So, we want to use it using generate to create an N-bit ripple carry adder.

(Refer Slide Time: 21:41)

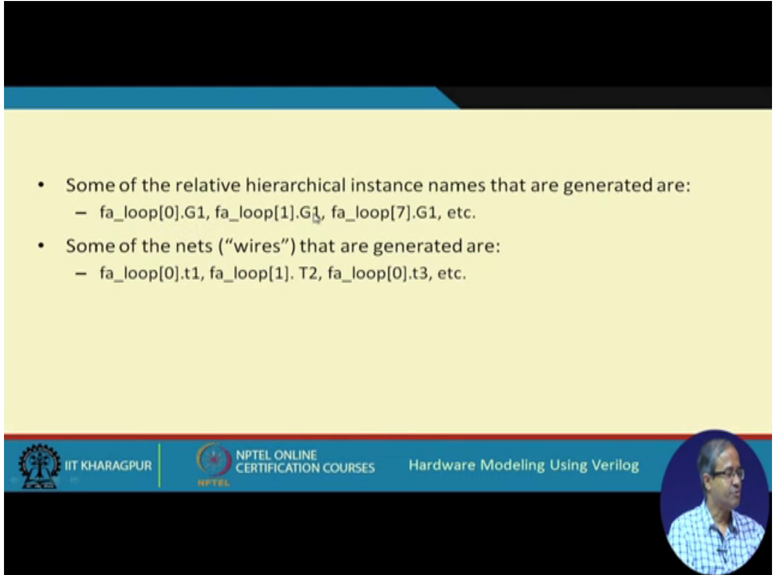
```
module RCA (carry_out, sum, a, b, carry_in);
  parameter N = 8;
  input [N-1:0] a, b;    input carry_in;
  output [N-1:0] sum,    output carry_out;
  wire [N:0] carry;  // carry[N] is carry out
  assign carry[0]   = carry_in;
  assign carry_out = carry[N];
  genvar i;
  generate for (i=0; i<N; i++)
    begin  fa_loop
      wire t1, t2, t3;
      xor G1 (t1, a[i], b[i]), G2 (sum[i], t1, carry[i]);
      and G3 (t2, a[i], b[i]), G4 (t3, t1, carry[i]);
      or G5 (carry[i+1], t2, t3);
    end
  endgenerate
endmodule
```

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

Let us see how. This is the code you see the code is so compact. See this means we are not in stretching this we are just picking this code directly. You see these 3 wire and his xor and or. So, whatever was this we have directly picked up these 4 lines of code. So, these 3 temporary lines and these 3 getting xor and an or so here you see we are using a parameter n let us say 8 we are creating an 8 bit adder. So, a and b are 0 to n minus 1, sum is also 0 to n minus 1. Carry there will be one carry means that the output of every ripple carry adders (Refer Time: 22:30). So, it will be n minus 1 plus 1 carry out. So, that is why I am defining 0 up to n. And this last bit carry in that is nothing but the carry out, I am just doing a continuous assignment using a sign. Similarly the carry in of the adder that is your carries 0, the rest will be generated by the full adders.

So, I have generated a variable genvar I/, generate this loop again 0 to n minus 1 this is the name of the loop f a loop and the description just the description which is given same description here right. Now you see this generate will be creating 8 copies of these 4 statements are there 4 lines. So, total 32 lines will be generated. Now you see there are so many variables here there are gates g 1, g 2, g 3, g 4, g 5. There are some wires t one t 2 t 3. So, how do you refer them for the copies? Same way this f a loop 0 dot t 1 f a loop 0 dot t 2 f a loop 0 dot g 1 f a loop 1 dot t 1 and so on in the same way.

(Refer Slide Time: 23:48)



So, this kind of hierarchical looking conventions you can write. So, f a loop this index you can use to in the copy number, you can indicate the gate number, the intermediate net number anything you want right. So, this is a very convenient to this t will be small right. So, you see using this generate you can very conveniently represent some designs which are which are inherently high creative in nature.

Like this means adder is a very natural example. So, means whatever design comes to your mind which can be created by replicating or repeating the same building block a number of times, you can use these kind of general statements very conveniently. So, it makes your module description very small. And you are putting the responsibility on the synthesis or the simulation tool to expand them right. So, this we come to the end of this lecture.

So, we shall be continuing with the discussion in the next lecture.

Thank you.