

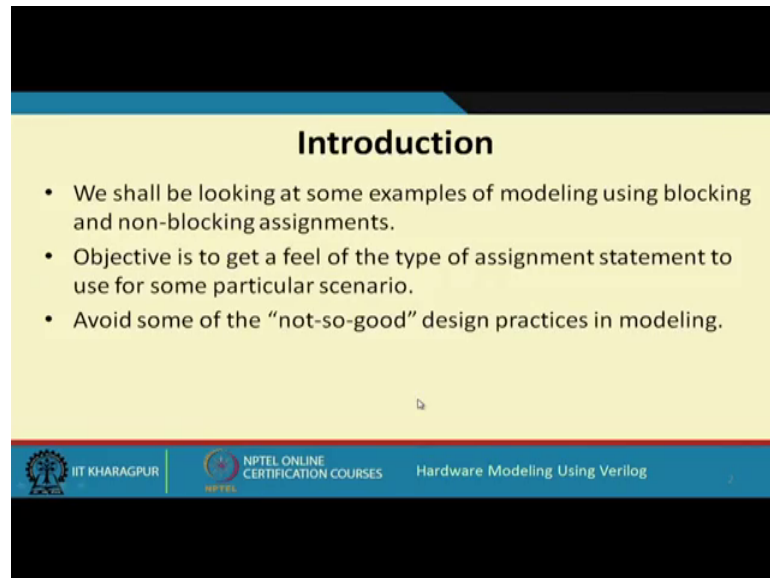
**Hardware Modeling using Verilog**  
**Prof. Indranil Sengupta**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 17**  
**Blocking / Non-Blocking Assignments (Part 2)**

So, in this lecture, we shall be looking at some of the examples where we will be using both Blocking and the Non-Blocking styles. And we shall see some of the features and some of the issues that might that might happen or that might arise.

So, this is our part 2 of this lecture, fine.

(Refer Slide Time: 00:40)



**Introduction**

- We shall be looking at some examples of modeling using blocking and non-blocking assignments.
- Objective is to get a feel of the type of assignment statement to use for some particular scenario.
- Avoid some of the “not-so-good” design practices in modeling.

b

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

So, as I said, we shall be looking at some actually small examples. Now why we are doing that our objective is to try to give you a feel that for what kind of design which kind of assignment statements would be better suited and there are a few practices which might lead to some errors which are best avoided these are or this may be regarded as not so good design practices.

So, while we discuss the various styles and we look at the different examples, we shall see that some of the design styles may lead to some constructs which are very easily confused by the designer that the designer may very easily insert some error means in advertently in the design. It is very easy to do. So, if you are not extremely careful. So,

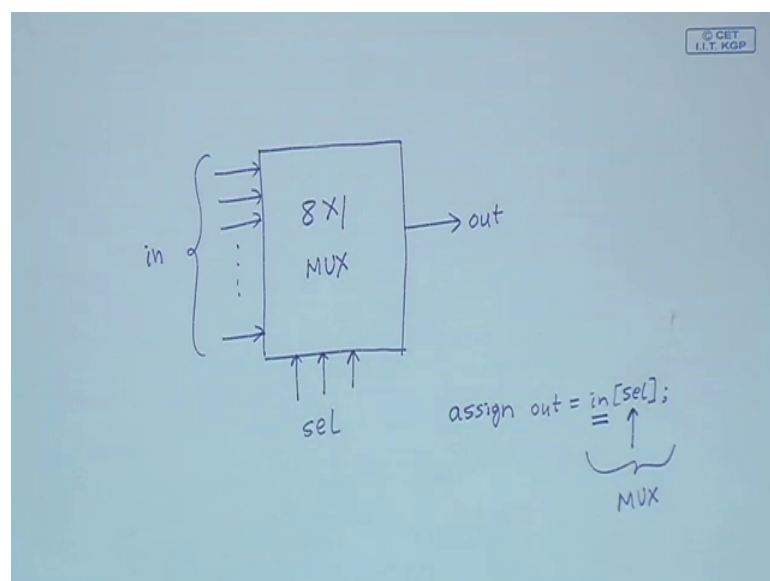
there are some constructs which are best avoided like one, I have already told you in the last lecture that you should avoid using both blocking and non-blocking assignment within the same always or the same initial block.

(Refer Slide Time: 02:04)

```
// 8-to-1 multiplexer: behavioral description
module mux_8to1 (in, sel, out);
  input [7:0] in;   input [2:0] sel;
  output reg out;
  always @(*)
  begin
    case (sel)
      3'b000: out = in[0];
      3'b001: out = in[1];
      3'b010: out = in[2];
      3'b011: out = in[3];
      3'b100: out = in[4];
      3'b101: out = in[5];
      3'b110: out = in[6];
      3'b111: out = in[7];
      default: out = 1'bx;
    endcase
  end
endmodule
```

So, the first example, we take is that of a multiplexer, this is a 8 line to 1 line multiplexer, this is the behavioral description of it. Now just to tell you 1 by 8 line to 1 line multiplexer what is this schematic we are looking at?

(Refer Slide Time: 02:27)



Yes we are looking at a 8 to 1 multiplexer whose output is out there are 3 select lines which we call is SeL and there are 8 input lines, the input lines are called in. So, depending on the select line; one of the input will be selected. Now earlier we had seen that we can very easily model a multiplexer using an assign statement like we can write assigns out equal to in SeL.

So, if we use a vector on the right hand side with a variable as the index this will generate a multiplexer this is what we mentioned, but here we are going a little bit into the behavior of it, but instead of using assign statement we are trying to use a procedural block to model a multiplexer. So, how we do it we are using a; we are using an always block. So, let us see here the parameters are in SeL and out in is the input there are 8 lines select line there are 3 bits and output see this output we are declaring as reg, because inside this always block we are assigning some value to this out.

So, I mentioned inside the procedural block, the left hand side has to be either reg or an integer or a real or a time variable. So, here we have defined it as type reg. Now here we are saying that always at the reg star whenever some in values change input values change either in or SeL, you do this; it is a simple case statement on SeL, here we are just enumerating all 8 binary combinations 0 0 0 up to 1 1 1 if it is 0 0 0, then in 0 is selected, it will be assign to out if 0 0 1, then in 1 will be assign to out and so on.


If it is 1 1 1 in 7 will be assign to out. Now you see at the end, we have given a default case which says that we are initializing some undefined value to out. Now you may ask that we will we have already defined all possible 8 combinations here in the case. So, why do we need to specify default? So, you remember that in Verilog, we mentioned when we talked about variables and their values that Verilog is actually a 4 valued logic modeling system.

So, every variable can assume a value not only 0 and 1, but also x and z due to some problem in the test bench or the circuit, which is driving maybe you have not initialized some variable in a proper way this select line may be coming here as an x value. So, what will happen. So, it will not match with any one of 0 0 0 0 1 up to 1 1 1. So, it is a different value x. So, for those cases it will go to the default option. So, it will get matched to the default and the output will also be said to undefined x in that case, right.


(Refer Slide Time: 06:15)

```
// Up-down counter (synchronous clear)
module counter (mode, clr, ld, d_in, clk, count);
input mode, clr, ld, clk;
input [0:7] d_in;
output reg [0:7] count;

always @ (posedge clk)
    if (ld)        count <= d_in;
    else if (clr)  count <= 0;
    else if (mode) count <= count + 1;
    else          count <= count - 1;
endmodule
```




IIT KHARAGPUR



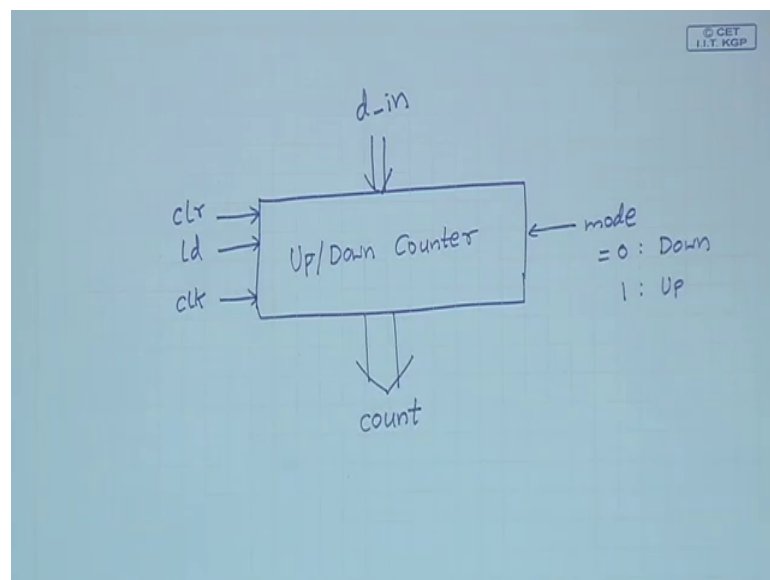
NPTEL ONLINE  
CERTIFICATION COURSES

Hardware Modeling Using Verilog



So, this is a multiplexer the next example we take is a synchronous up down counter. Now let us again look at what kind of a circuit we are talking about.

(Refer Slide Time: 06:30)



Here we are talking about a circuit which is a up down counter which can count up or which can also count down. Here the count value which is the output, this we are calling as count. Now we can initialize the value of the counter. There is another input called d in, there is an input called clr, I can clear the counter to 0, I can load the counter with this input value d in, if I want and of course, there is a clock, these are my signal values in

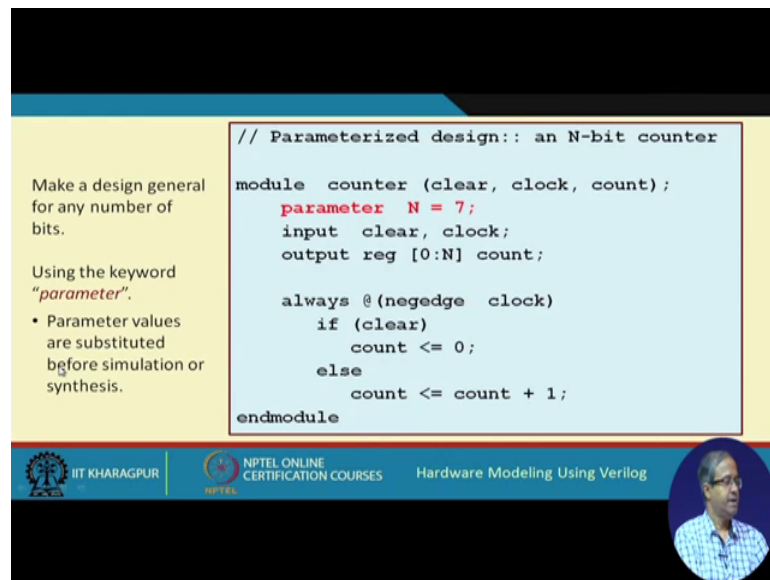
this counter. So, I can clear it to all 0s. So, I can load it with any value I want and if I apply a clock it will count.

And there of course, there is another input called mode; mode will tell whether I am going to count up or count down if mode equal to 0 it means I want to count down if mode equal to one it means I want to count up right this is the circuit which you want to design or model in Verilog, let us see how we have done it. So, the first thing we have done it is that we have defined all the parameters mode clear load data in clock and count this 4 are the input signals d in also input, but d in is a vector 8 bit counter, I am assuming 8 bits and count is also an output which also the reg because we are assigning count we have defined as reg also 8 bits.

And here we are doing everything in a synchronous way load clear everything is synchronous this will happen at the positive edge of the clock. So, whenever there is a positive edge of the clock we first check whether the load input is one or not if the load input is one then the value is loaded from d in. Next we check if clear is 1; if clear is 1, then count value is initialized to 0 or else we check if mode is 1 or 0. If mode is 1, we increment the count by 1 up count, if mode is 0, we decrement it by one down count. So, this shows you the behavior of the up down counter. So, you see using this kind of non-blocking statement we can model this counter in a very convenient way well.

Well of course, here the statements are not executing concurrently because of the; if than else, exactly one of them will be executing, but there could have been more statements also.

(Refer Slide Time: 09:32)



Make a design general for any number of bits.


Using the keyword "parameter".

- Parameter values are substituted before simulation or synthesis.

```
// Parameterized design:: an N-bit counter
module counter (clear, clock, count);
    parameter N = 7;
    input clear, clock;
    output reg [0:N] count;

    always @(negedge clock)
        if (clear)
            count <= 0;
        else
            count <= count + 1;
endmodule
```

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



Next, let us look at a parameterized design parameterized design we mentioned earlier we can define some kind of a constant call a parameter like here we define say here we are trying to design an N bit counter, but N can be anything we are specifying the value of N by this parameter N equal to 7, but inside my program I am using N everywhere well here just in one place. So, the count register I am declaring of size N. So, if N equal to 7 means I am actually declaring a 8 bit register.

One more than this; 0 to 7 because N will be 1 less than that; so, actually; it will be N plus 1, not exactly N 1 more than that. So, the declaration is very simple. So, always just assuming that the counter will be count at the negative edge of the clock at the negative edge of the clock, there is a clear, if clear you clear the count or otherwise you increase the count by 1, right. So, using this parameter, you can create a general design where you can only change this one single line and your entire design can become a instead of an 8 bit counter, it can become a sixteen bit counter, right.

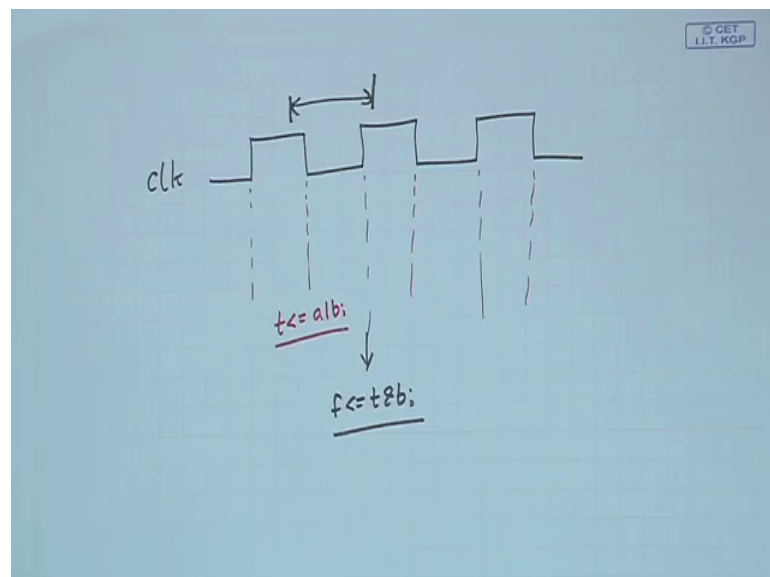
(Refer Slide Time: 11:00)

```
// Using multiple edges of the same clock
module multi_edge_clk (a, b, f, clk);
  input a, b, clk;
  output reg f; reg t;
  always @(posedge clk)
    f <= t & b;
  always @(negedge clk)
    t <= a | b;
endmodule
```

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

This example shows that you can have a module where you can use more than 1 clocks; there can be very complex designs, I means in such complex designs you may be using not only 1 clock.

(Refer Slide Time: 13:29)



But more than 1 clock. So, 1 clock signal may be used to control one part of the circuit the other clock signal may be used to control some other part of the circuit and those 2 parts may be executing concurrently under the control of the 2 clocks, the 2 clocks may be of different frequencies also, right. So, in Verilog even inside a single module you can

do this kind of modeling very easily, here I am assuming that the 2 clocks are the clock 1 and clock 2 and a, b, c, these are inputs and f 1, f 2 are 2 outputs declared as reg. So, I can use and this is just a very simple example for illustration I can use 2 concurrent always block, let us say one of them is triggering at the positive edge of clock one other one is triggering at the negative edge of clock 2.

So, these are working concurrently; one of them is assigning the value a and b to f 1 and the other one is a; it is assigning b x or c; the exclusive or 2 f 2 and they are happening concurrently in synchronism with 2 different clocks the clocks may be of different phases different frequencies. So, there is no restriction there. So, you can have a very general design like this where multiple signals can be used to synchronize the operation of your design; this is possible not only that you can also use the 2 edges of the same clock multiple edges of the same clock you can use to carry out some operations like in this example.

I am using a single clock signal clk a b at the inputs this output reg, this f is an output and t is another temporary; I am assuming just see what you are doing here, here let us say I have a clock signal a clock signal coming like this let us say these are the positive edges and the negative edges, I am showing were different color; these are the negative edges. Now let us look at this specification this specification says that always at the positive edge of the clock you do f assign t and b. So, at the positive edge let us say here you are doing f assign t and b this statement is executing and at the negative edge you are doing t assigned a or b; let us say here at the negative edge you are doing t assigned a or b.

So, you see what is happening here in this in this f statement f assigned t and b; you are using the value of t and t is getting assigned by this statement which is happening at this clock. So, you can say that I am doing some computation which is starting here and it is continuing till here I am using 2 edges. So, at the first edge; I am computing a value t and at the second edge, I am using that value t to compute some other final value f and this will go on repeating; right. So, this is one way or one technique using which we can actually carry out 2 operations within a single clock period we can do something in the rising edge of the clock something else in the falling edge of the clock.



(Refer Slide Time: 15:34)

The slide contains a Verilog code block and a list of bullet points. The code defines a module `multi_edge_clk` with inputs `a, b, c, d, f, clk` and an output register `f`. It uses two `always` blocks: one triggered by the positive edge of `clk` to assign `c <= a + b;` and another triggered by the negative edge of `clk` to assign `f <= c - d;`. The bullet points explain that two operations are carried out every clock cycle: `c` is assigned at the rising edge and `f` is assigned at the falling edge. It also notes that it is assumed that addition or subtraction can be completed in half a clock cycle.

```
// Another example
module multi_edge_clk (a, b, c, d, f, clk);
  input clk;
  input [7:0] a,b,c,d;
  output reg [7:0] f;

  always @(posedge clk)
    c <= a + b;

  always @(negedge clk)
    f <= c - d;

endmodule
```

- Two operations are carried out every clock cycle.
  - "c" is assigned at the rising edge.
  - "f" is assigned at the falling edge.
- It is assumed that addition or subtraction can be completed in half a clock cycle.

And one of the values may be fed to the input of the other computation like in the example I showed just now, right. So, this is quite possible. So, let us take another example this is an example we shows some addition and subtraction operations similar with this multiple edge clock that same kind of a thing. Here we are using let us say this a plus b, a minus b, here of course, the input description is not completed, let us just make it complete, let us say input a b clock, let us make another declaration, let us say we define input, let us say we have 8 bits and we define a, b, c and d all of them, let us declare like this a, b, c, d, all are 8 bit inputs; let us say and this f is a reg t is also reg.

T is not required here of course, c let us make it t now finds fine. This t is not required. So, t you can forget, fine. So, here you see at the positive edge you are doing a plus b is assigning to c. So, whatever value was there in c that gets modified and at the negative edge of the clock you are using that value of c subtracting with d and you are storing it into another value f, right. So, in this f will also be a vector sorry this will also be a vector like this fine. So, here at the rising edge you are doing some computation at the falling edge you are do some computation.

So, you can do 2 computation in the same clock cycle; one addition and one subtraction, if your clock cycle time is large enough. So, basically; what I am saying is that you are carrying out 2 operations in every clock cycle in these always loops in the first positive edge of the clock c is assigned a value the sum of a and b and at the other one at the


falling edge of the clock, f is assigned a value. This is the subtraction of c and d. So, here we assume that our clock cycle time is large enough. So, that this addition and subtraction can be completed within half a clock cycle for this kind of a scenario we can use these multiple edges of the same clock to activate 2 operations, all right.

(Refer Slide Time: 18:24)

```
// A ring counter
module ring_counter (clk, init, count);
input clk, init;
output reg [7:0] count;
always @ (posedge clk)
begin
if (init) count = 8'b10000000;
else begin
count = count << 1;
count[0] = count[7];
end
end
endmodule
```

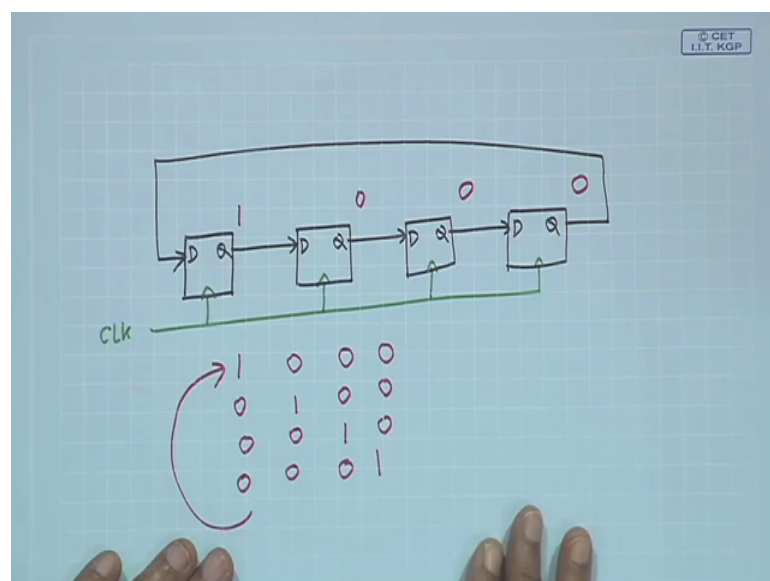
- This solution is wrong.
- count[7] will get overwritten in the first statement.
- Rotation of the bits will not happen.

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



So, let us now look at the example of a ring counter. Now I mean what is a ring encounter just to recall ring counter is nothing, but a shift register.

(Refer Slide Time: 18:32)

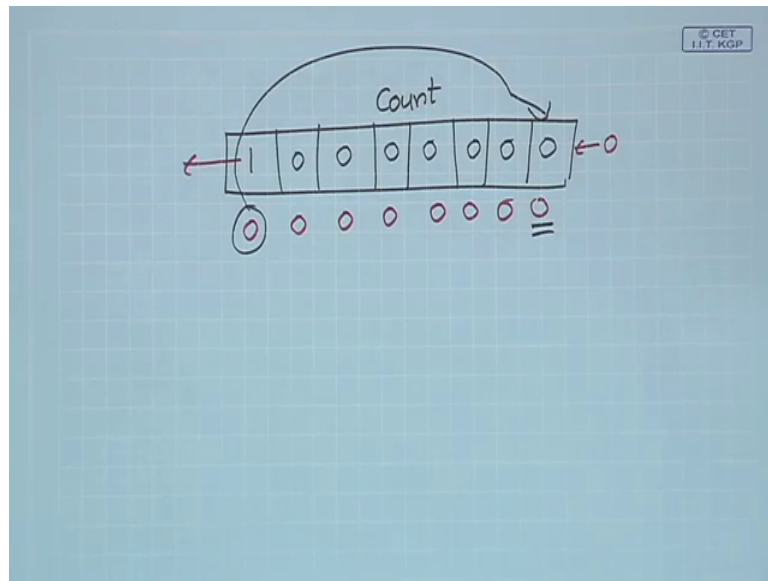


So, I am showing a 4 bit ring counter, it is just a shift register comprising of several deep flip flops. So, the output of one flip flop is connected to the input of the other ring counter means they are connected as a ring as a chain and this counter is initialized typically to the state 1 0 0 0 and the clock signal is applied to all the flip flops together this is the clock. So, if you initialize this shift register with 1 0 0 0 and if you go on applying the clock. So, what will happen initially it was 1 0 0 0 after 1 clock, it will be rotating right by 1 place.

So, this 1 will be shifted here. So, this 0 will come back 0 will come here next clock this one will come here this 0 will again come back 0 0 1 0, next clock 0 0 0 1, then again it will go back to 1 0 0 0. So, this will repeat, right, this is the function of the ring counter. So, here in this design we are showing you an 8 bit ring counter. So, we have a clock, we have a init; init signal which will be initializing the ring counter to a single one and all 0s and the output of the counter we just count. So, clock and init are the inputs and count is the output which is reg.

So,. So, every time when there is a clock edge; posedge, we are checking that if init is active init is one well if init is one we are initializing count to this 1 0 0 0 0; you see here, we are using blocking assignment statement else begin count equal to count shift left by one then count 0 equal to count 7. Now tell me whether this is a correct description of a ring encounter. So, these 2 statements do they actually do the shifting correctly; just see count is an bit variable; right.

(Refer Slide Time: 21:25)



So, let me just show you; count is an 8 bit register; there are 8 flip flops. So, I have 1 here and there are 7 0s. This is my count, now in this code; what we are doing in this begin end we are first shifting count left by 1.

So, if you shift it left by 1; this one will go out and one will disappear and shift left by default will feed as 0 on the right. So, the counter will become all. So, after the shift like this one will disappear and then you are saying count 0 equal to count 7 to make the rotator, but this count 7 has already become 0. So, even if you store this 0, here this value will still remain a 0 this one will not come back. So, means after a rotate operation after this begin end block the count value will become all 0 and after that it will remain all 0.


So, this is not a correct description of a ring counter. So, as I said, this solution is wrong, this count 7 will get overwritten in the first statement itself, it will become 0 and the rotation will not happen.

(Refer Slide Time: 22:56)

```
// A ring counter (Modified version 1)
module ring_counter (clk, init, count);
  input clk, init;
  output reg [7:0] count;
  always @ (posedge clk)
  begin
    if (init) count = 8'b10000000;
    else begin
      count <= count << 1;
      count[0] <= count[7];
    end
  end
endmodule
```

- This is the correct version.
- Since non-blocking assignments are used, rotation will take place correctly.

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



So, how I can rectify this error; it is very easy, I change the blocking to non-blocking see here what will happen here we are doing the same thing, but we are using a non-blocking at sum.

(Refer Slide Time: 23:17)

© CET I.I.T. KGP

count <= count << 1;      00000000  
count [0] <= count [7];      1

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0

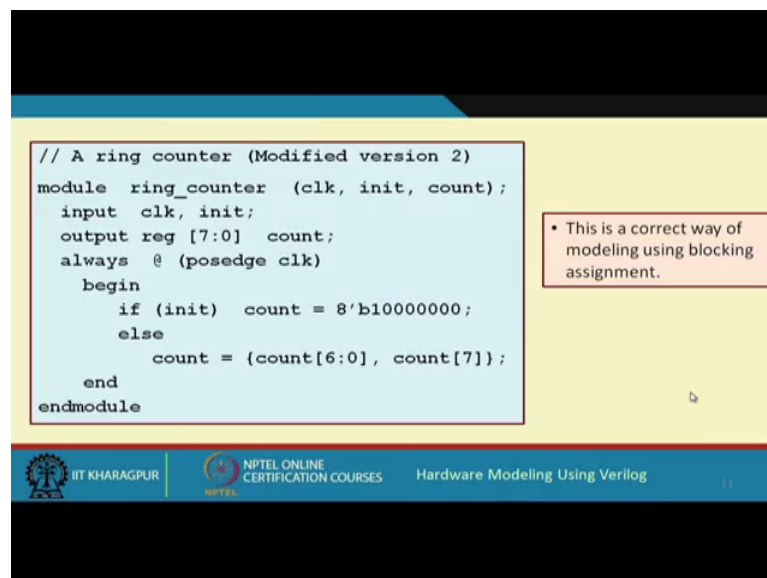
000000, 1

Let us see; what will happen. So, now, our statements are the first statement says count non-blocking count shift left by one and the second statement says count 0 assign count 7. Now again let us look at the count, there is an 8 bit register what will happen let us see suppose initially my register contains this, this is my count and these are the index values

0, 1, 2, 3, 4, 5, 6 and 7. Now according to the rule of the non-blocking assignments, the right hand sides will be evaluated first concurrently.

So, if you do a count less than less than 1; what will happen if you do a count less than 1? This was the count. So, it will become all 0, but count 0; this count 7; count 7 is 1. Now you are assigning them together. So, when you are just assigning them, these counts 7 1; this is also bit getting stored. This will be assigned to count 0. So, this 0 will become 1 and the remaining bits will become 0. This is how it is interpreted. So, actually the rotation will take place correctly here, fine. So, here this is the correct version rotation will actually take place.

(Refer Slide Time: 25:15)



```
// A ring counter (Modified version 2)
module ring_counter (clk, init, count);
  input clk, init;
  output reg [7:0] count;
  always @ (posedge clk)
  begin
    if (init) count = 8'b10000000;
    else
      count = {count[6:0], count[7]};
  end
endmodule
```

• This is a correct way of modeling using blocking assignment.

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

Now in fact, even without using this non-blocking we can have a correct version like this suppose I write my blocking assignment like this, I write count equal to count 6 to 0 and count 7 concatenated what does this mean count 6 to 0 means you see this part count 6 to 0 bit number; 6 to bit number 0, you take this first these are all 0s; 6 0s concatenate with count 7; count 7 is this it is one you concatenate this with one take this whole thing together. So, one has already come here you assign this to count 0, 0, 0, 0, 0, 0, 1 which is what it should be rotate right rotate left inter fine.

So, this will also work correctly. So, you see that using either blocking or non-blocking assignment, if you know what it means what this statements do and how they execute you will be able to find out that whether or not your model is correct or not because you

may see that you have written something, but well simulation you see that your result of the output is not coming correctly. So, you will have to interpret why it is so, that; what is the problem or what is the error in your code; you will have to understand; what is the meaning of the blocking assignment; how they execute; how they change the values, then only well be able to debug and come up with a correct version of the code. So, with this we come to the end of this lecture.

Now, in the next lecture, we shall be continuing with some other aspects of blocking and non-blocking assignments, till then you can refresh your memory I mean I strongly suggest; whatever I am covering, I am discussing, you try to work them up yourself, you try to run them at least on the simulation platform and get a feel of if you make some changes; what is the impact that will get on the output only, then you will be able to learn the language and will be able to get a confidence on the language constructs that are available.

Thank you.