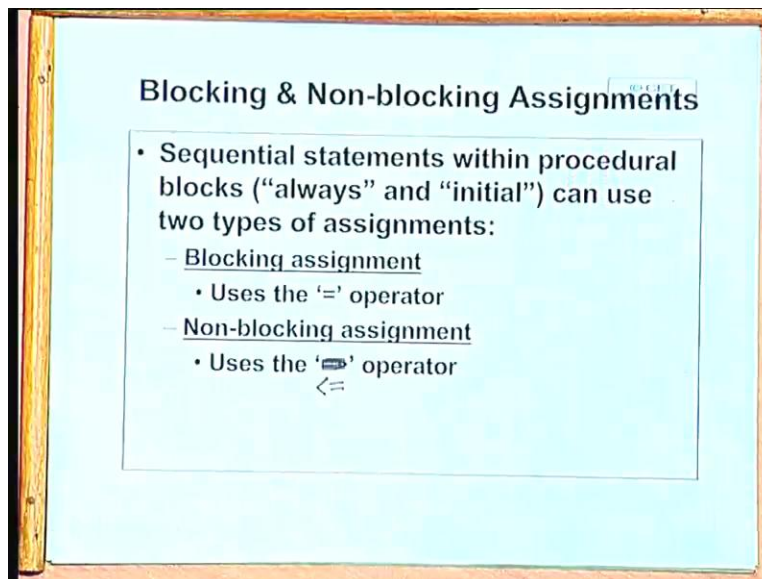


Electronic Design Automation
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No #5
Verilog: Part IV

So in this lecture we would start by talking about the different options we have that whenever you are trying to assign some expression to a value inside an always block, inside a procedural block. Now the, now in the examples we have seen so far we had used the simple operator equal to sign to assign a value to arrange the type variable. But we will see now that in fact there are 2 different ways of doing that. Well we had look only at 1 of the alternative. There is another alternative.

(Refer Slide Time: 01:39)



There are actually 2 different kinds of assignments you can make inside a procedural block. You recall there are 2 different kinds of procedural block always which goes on indefinitely and initial which gets executed only once and used in the test benches. Now whenever you have a procedural block say always, say inside always you have a begin;

you have an end. You have several statements in between. Now exactly how are the statements executing; the semantics of that?

That basically is distinguished by the kind of assignment we are making. There are 2 options: blocking and non-blocking. See the assignment that we have seen or used in the example so far we had used the equal to operator. That is the so called blocking assignment and the other alternative is that instead of equal to you can use this less than equal to. This is called non-blocking. So we will see the difference between these 2 what is the (()) (02:50) Yes. (()) (02:52)

Student: ...series and when are they executed in parallel?

Yeah. Here well your question is that when are the statements executed serially and when are they executed parallel. So actually we are trying to answer that exactly that question. Now means in the blocking assignment statement that 1 particular statement, well there it will block the execution of the future statements. So unless execution of 1 statement is finished you cannot start the next. It is sequential but non-blocking in a sense it is parallel. We will see that how parallel it is. So first let us look at the blocking assignment statement type. Okay.

(Refer Slide Time: 03:35)

Blocking Assignment (using =)

- Most commonly used type.
- The target of assignment gets updated before the next sequential statement in the procedural block is executed.
- A statement using blocking assignment blocks the execution of the statements following it, until it gets completed.
- Recommended style for modeling combinational logic.

$f = a + b;$

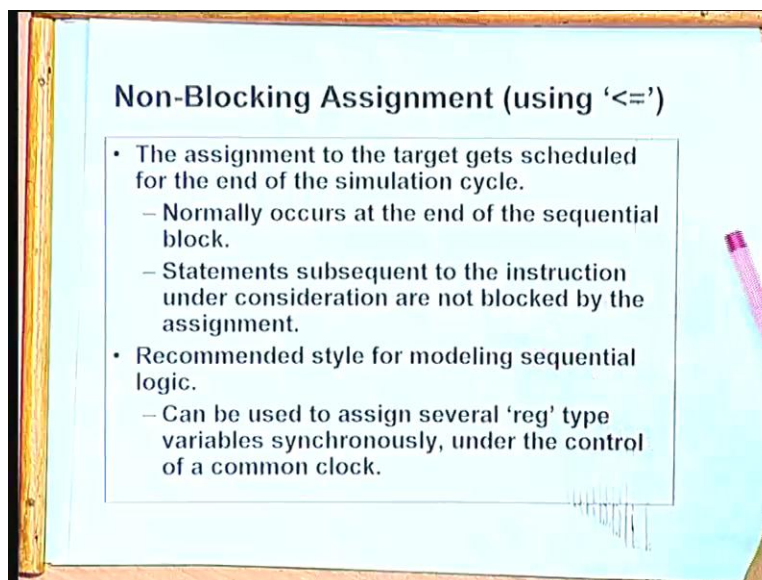
Blocking assignment is of course the most commonly used. See inside the always block whenever we write an expression like say f equal to a plus b . This is an example of a blocking assignment. So the values of a and b are available from somewhere they are they are added and the value is assigned to f . Now this operation can take some time. Now here for the purpose of simulation you can explicitly specify a delay or from the point of synthesis this adder will be having some delay. So this operation will take some time to execute. Now a blocking assignment says that the target of the assignment gets updated.

Means this f will get its new value before you start executing the next statement in sequence before the next sequential statement is executed. So in the blocking assignment there is a strict order of execution which the semantic demands. So it is called blocking because 1 statement is blocking the execution of all statement that follows. A statement using blocking assignment blocks the execution of the statement following it. So the as soon as it gets completed only then the instructions that follow will start, but there is 1 cache here. It is said that this is the recommended style for modeling combination logic and not for sequential logic.

Because in a combinational logic whenever you are specifying some expression some values often there is some data dependency. Like the output of an AND gate feeds the input of an OR gate like that. So most logically you will be specifying the values in that particular order the way it is computed. But if it is a sequential logic just you understand there whenever you are trying to assign a value to it you normally do so in synchronism with a clock. Suppose we give an always block with a posedge, then inside the always block there are 10 statements.

So that what does it mean? That, that every time there is an edge of the clock this 10 statements will be executed 1 after the other, not really, this semantic says that at the edge of the clock all the statements are to be executed together right. So when you are modeling a sequential logic there has to be something else. In the blocking assignment you cannot model that. That mean all the assignments are taking place together but whenever you have you are modeling sequential logic you have a clock. You want that all the assignments you are trying to make they are being taking place together simultaneously and for that purpose you need the non-blocking assignment. Okay.

(Refer Slide Time: 06:51)

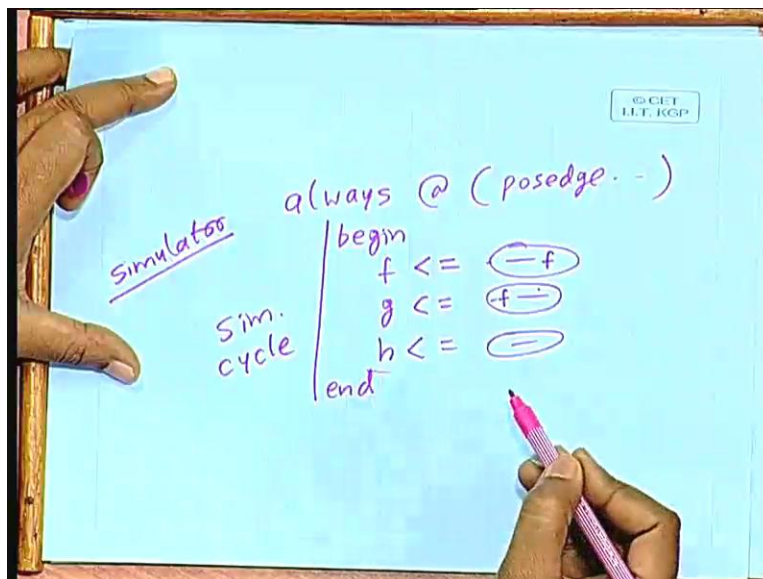


Non-Blocking Assignment (using '<=')

- The assignment to the target gets scheduled for the end of the simulation cycle.
 - Normally occurs at the end of the sequential block.
 - Statements subsequent to the instruction under consideration are not blocked by the assignment.
- Recommended style for modeling sequential logic.
 - Can be used to assign several 'reg' type variables synchronously, under the control of a common clock.

Non-blocking assignment the only difference is that the operator instead of equal to you use less than equal to this symbol. Now first let us look at the semantic because from the point of view of hardware I can understand. Well this is the edge of the clock, whenever the edge comes this assignments will take place. But with respect to simulation what is this semantic? Because we often check or we often validate a design by simulating. Now let us see what it means with respect to simulation. Okay. So the non-blocking statement here the assignment will get scheduled for the end of the simulation cycle. So end of the simulation cycle means.

(Refer Slide Time: 07:42)



See here what I mean to say is that suppose you have an always block. You have an always block which is triggered by some posedge or negedge of some variable which is a clock. Now inside the begin end you have a several such statements f assigned something, g assigned something okay. Now this always block whenever it is executing with let us say a simulator. You think from the point of view of the simulator, this simulator will schedule this block for execution whenever there is a posedge on this specified variable. So whenever there is a posedge on the variable we start our so called

simulation cycle. So in 1 simulation cycle we are supposed to evaluate the value of f assigned to it, value of g assigned to it.

Now what the semantic of this non blocking assignment says that the assignment is scheduled normally at the end of the sequential block, which means again let me let me come to it that when the simulator tries to handle this block it evaluates the right hand side of all the statement there, there can be others also h assigned something. It evaluates the right hand side of all of them but when it reaches the last statement in the always block only then it assigns the values right. So it is quite possible that you have say f appearing in the right hand side of some expression. No problem there, because you are taking the old value for calculation. The value will get assigned only at the end. After all the statements all the expressions on the right hand side have been evaluated. (()) (09:45) Actually modeling sequential logic yes.

So this normally occurs at the end of the sequential block and very naturally this statement subsequent to the instruction which is currently being evaluated is not blocked by the assignment. Because if there are 4 assignments the 4 assignments are they are essentially going on concurrently. There is no dependency between the assignments. All the expressions are evaluated and at the end of the block they are all assigned together. They are going on in parallel okay and you can understand that very naturally this is a recommended style for modeling sequential logic. Yes. (()) Yes we will come to that. This is the recommended style for modeling sequential logic and well.

(Refer Slide Time: 09:50)

Non-Blocking Assignment (using '<=')

- The assignment to the target gets scheduled for the end of the simulation cycle.
 - Normally occurs at the end of the sequential block.
 - Statements subsequent to the instruction under consideration are not blocked by the assignment.
- Recommended style for modeling sequential logic.
 - Can be used to assign several 'reg' type variables synchronously, under the control of a common clock.

So here as I have said that we are essentially using several reg type variables and you are assigning some values to them synchronously under the control of a common clock. This is typical usage of this okay.

(Refer Slide Time: 11:02)

Some Rules to be Followed

- Verilog synthesizer ignores the delays specified in a procedural assignment statement.
 - May lead to functional mismatch between the design model and the synthesized netlist.
- A variable cannot appear as the target of *both* a blocking and a non-blocking assignment.
 - Following is not permissible:

```
value = value + 1;
value<=> init;
```

#10
#20

So some rules you need to remember and you need to follow. See I have repeatedly told this that user can specify some delays along with each of the blocks. Now whenever you are carrying out synthesis the delays are all ignored. Delay makes sense only from the point of view of simulation okay. But there is 1 problem here. Suppose in a code you explicitly specify some delays. Hash 10, hash 20 something. Then you are possibly running into a problem if your final target is to do synthesis because after synthesis delay will be as far the components you are using in the library the technology library and in simulation delays are as per the user specified values. So when you see and compare the simulator simulation outputs there may be mismatches the values may not be changing at the same times. So you have to remember this when you are doing the simulation, so this is what is mentioned here.

This may lead to functional mismatch between the model; model means the specified code and the synthesized netlist. Design model we are, we are checking through simulation and synthesized netlist you are obtaining through synthesis and then again you are simulating. (()) (12:42) We can use delays in the always block but they will be used only if only for simulation purpose. (()) (12:51) See the as soon as we include delays. So unless the synthesizer ignores the delays they will not be synthesizable. The always block will not be synthesizable but see synthesizable you have a subset of Verilog features which are synthesizable. We will be discussing this issue later that actually what is synthesizable and what is not.

And if you just include delays somewhere there are some synthesizers which will work very well. They will ignore the delays and go through. But there are some other synthesizer which will give you some error messages while synthesizing. It will say that the always block. Block is not synthesizing synthesizable. Those error messages are generated from the you can say from the prediction that if you are using such delays in the always block later on your simulation results will not match. So for that purpose it will give you those messages at the (()) (13:58). So delays are allowed by some synthesizers but some do not allow. But you will have to just really understand exactly what the synthesizers do with the delays exactly how they deal with it.

But from the users point of view you just remember 1 thing. Delays are ignored by the synthesizer. They are they are used or utilized by the simulator. That is all. (()) (14:30) There is no standardization of synthesizers unfortunately. The same code which for example you will run correctly on synopsis you will get a lot of error on hidden stones. There is no standardization as such (()) (14:48). Yes so that every particular implementation of the synthesizer they will have a booklet they will give to you. That well this subset is synthesizable. This is not synthesizable. You follow a restrict to this set but that set is not universally accepted. Some synthesizers accept something outside which others do not accept. (()) (15:14)

Student: Filp flops.

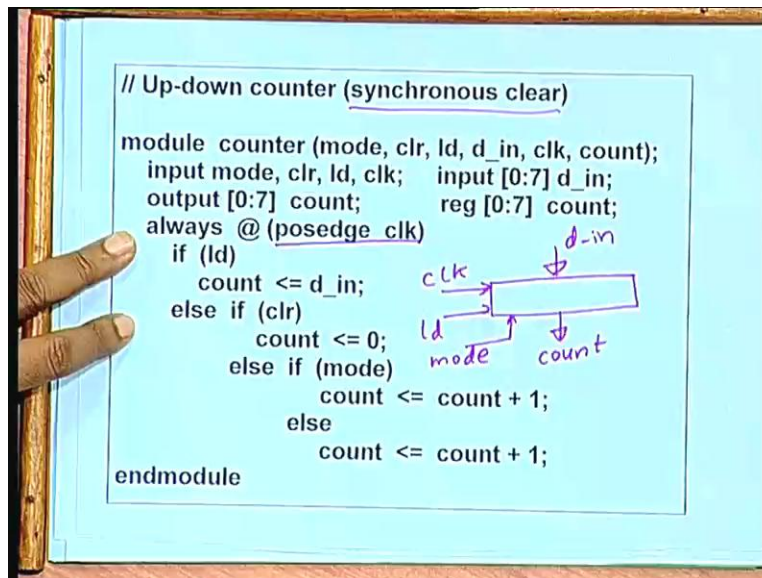
It is up to put exactly the flip flop is or you do not use the delays. You just use the syntax or the semantics of the language take care of it. So if it is for example, non-blocking assignments you know that all of them will be assigned at the end of the clock. You need not give any delay and if it is a continuous assignment statement you give 0 delay. That is a 0 delay simulation we assume. (()) (15:45) Yeah, yeah. That delay you can give globally outside the always block. But that inside the always block you need not have to give the delay that with each statement you need not have to give the delay okay fine.

Now here 1 of you was asking earlier that if we can mix blocking and non blocking assignment in the same always block. Yes we can but the only restriction is that we cannot use the same variable as the target for both but we can mix them up. The semantic will be the same wherever we are using non blocking assignments they will not get blocked by the blocking assignments. They will all be evaluated concurrently and waiting for the clock edge to come at the next simulation cycle and wherever you have the blocking statements, they will be executed sequentially. But it is not a good practice to do so for obvious reasons. You either use only blocking or only non-blocking in an always block. (()) (16:56)

Student: Non-blocking assignment in always block. So there we given period does not make any sense.

Non-blocking assignment given delay does not make any sense because it is always triggered by the clock. Yes true. So now let us look at some of the examples.

(Refer Slide Time: 17:18)



This is an example of an up down counter with synchronous clear. Well here what we are modeling, we are modeling a counter. It is an 8 bit counter with data in which is given by d in the output value we are calling count. Now in addition we have several other controls, we have a clock. We have a load control and we have a control mode. This is a counter where all the signals are handled in a synchronous way. That means clear load everything they will be are count. All will be done in synchronism with the clock. That is how in this Verilog module we have written the code. So you look at it. Input and output these are 8 bit quantities. Output is of reg type.

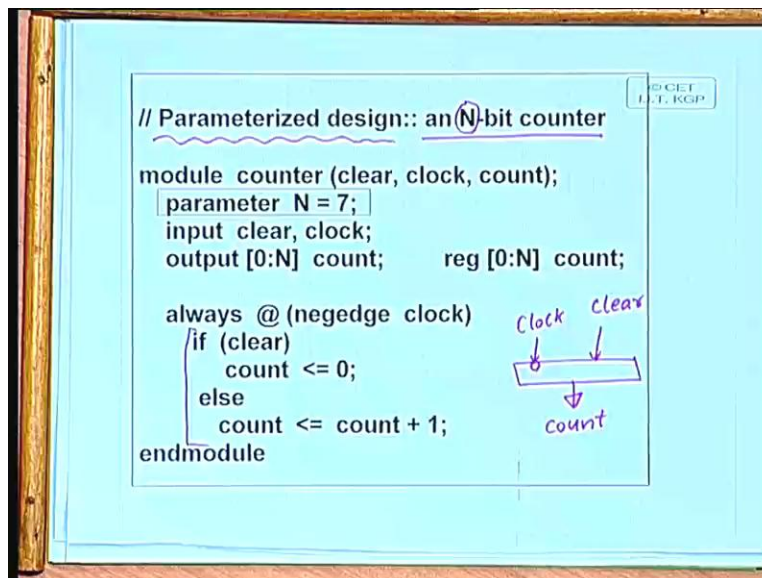
Here in the always block we are triggering by the clock, posedge of the clock, clock is a positive edge clock. In whatever we are doing inside the block that will take place

whenever the clock edge occurs. Now what you are doing? You are checking if load is one. If load is 1 this input value is getting loaded in the counter parallel, else if clear is 1 counter is cleared. Else if mode, if mode is 1 then it is up count if mode is 0. Sorry this will be down count. Count minus 1 right and see here all the assignments are non-blocking assignments. They are all assigning to the same variable. But depending on the values of load clear and mode only 1 of them will be executed. Right. (()) (19:26)

Student: actually does it make a difference?

Does not make any difference. Does not make any difference. So here all means depending on the if-statement this will be evaluated whatever value is being evaluated that will go into the count only at the end of the simulation cycle. This is 1 example and I am giving another example where we are trying to design a general n bit counter because sometimes in a design instead of designing a hardware block of a specified size it is good to have a general design.

(Refer Slide Time: 20:07)



The slide displays Verilog code for a parameterized N-bit counter. The code is as follows:

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

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

Handwritten annotations include a box around 'parameter N = 7;', a box around 'clear', and a box around 'count <= 0;'. To the right of the code is a small schematic diagram of a counter block. It is a rectangle with two input arrows at the top labeled 'clock' and 'clear', and one output arrow at the bottom labeled 'count'.

In this example we are showing an n bit counter and since it is specified by a variable it is sometimes called a parameterized design and having a parameterized design is a good idea wherever possible. Here there are 3 input outputs. Clear, clock and count simple. So there is no parallel load facility. So you have only the count value available count. We have a clock and we have clear. So this n is a parameter. A constant with a name, so in this example we have taken to be 7. You can change this 1 line. This will change. So always at the negedge of the clock clock is negative edge triggered. If clear count is 0, else count is count plus 1 simple. In this way you can create a parameterized design.

So the purpose of this example is to show how you can make or create a parameterized trigger. So it is a good idea to have a parameterized design but 1 thing. If you have a parameterized design then in some cases of course in a counter it is simple. In some cases it may be difficult to specify in the structural level. You may have to give the specification slightly at the behavioral level. Okay. (()) (21:32) Else is not concurrent. Either this or this will be executed. One of them will be executed. There is only 1 block. In fact in this always there is a single statement but there can be multiple statements also.

(Refer Slide Time: 21:52)

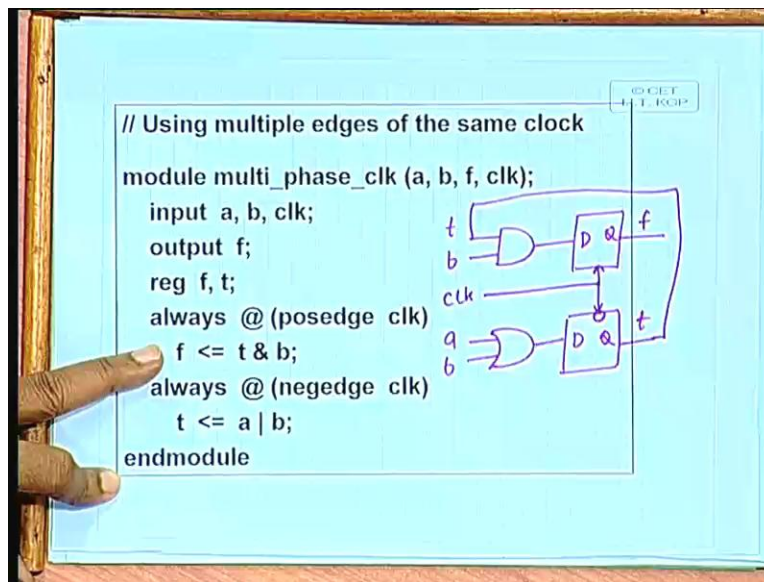
```

// Using more than one clocks in a module
module multiple_clk (clk1, clk2, a, b, c, f1, f2);
input clk1, clk2, a, b, c;
output f1, f2;
reg f1, f2;
always @(posedge clk1)
    f1 <= a & b;
always @(negedge clk2)
    f2 <= b ^ c;
endmodule

```

Now a few other points that inside a module you can have more than 1 clocks. This is perfectly permissible. Well of course this example is nothing meaningful but this shows that you have 2 different always blocks. One is triggered by clock 1. Other is triggered by clock 2. So actually when this is synthesized we will be getting a hardware like this. This will be an XOR gate, this will be flip flops. The first flip flop will be clocked using CLK 1. Second 1 negative edge will be clocked using CLK 2. This will be d input. This will be q output d q. So the synthesizer will be synthesizing this into some hardware like this. So you can have multiple clocks in the same module. No problem. Not only that you can have multiple edges of the same clock.

(Refer Slide Time: 23:13)



Say this is a similar example that here it will be synthesized like this. You have an AND where the inputs are t and b, this t and this is b. You have this is an OR, this is a, and b, this will be 1 flip flop. This will be another flip flop and first 1 will be a generating a value called f, second 1 will be generating a value called t. So actually this t will be fed back here right and this f will be going. And regarding clock this same clock will be triggering this as well as this but here the edges are different. So it is a very good practice in high speed design but you try to do some computation at both edges of the clock. Just

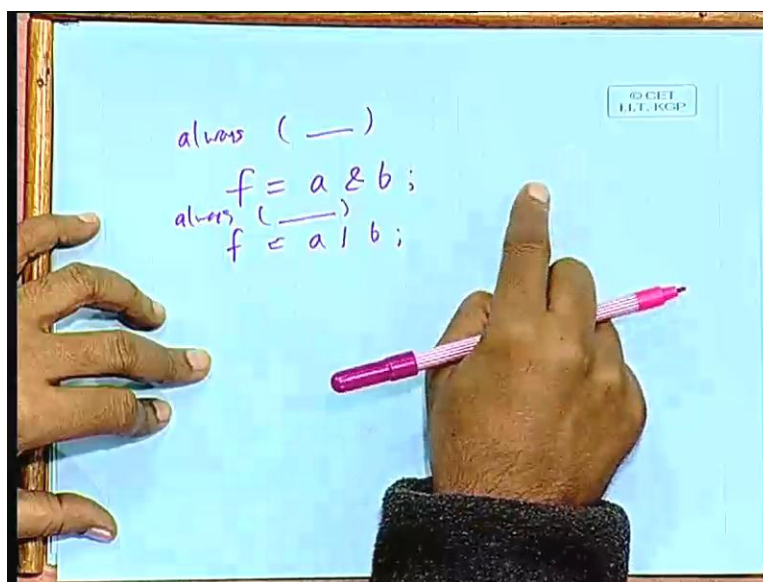
instead of waiting for the next clock edge you can do something in the leading edge and something else in the falling edge. (()) (24:25) No problem there. (()) (24:32)

See there can be 2 always blocks which can be triggered by same event. This can be posedge clock. This can also be posedge clock. But 1 thing you should ensure that the left hand side of this blocks will be different. Yes, (()) (24:47) Same values cannot be assigned for 2 different blocks. That will be a conflict. So the synthesizer will be giving you an error. There will be a conflict. (()) (24:59) Assign will be same thing yes. You are assigning this to f, also this you are assigning to f at the same (()) (25:05) repeating the statement. Well if the synthesizer is intelligent enough it can try to optimize that but otherwise it will give an error. Normally it will give an error. (()) (25:22) If you assign to this same thing?

Student: Two different expressions to the same thing a and b a and b blocking then also there will be error.

I did not get your question.

(Refer Slide Time: 25:40)



So you have f equal to a and b and you have okay. There is 1 always block here. There will be another always block here, f equal to a or b concurrent yes. See depending on the event list well if the synthesizer can find out that event list are not mutually exclusive then it will give an error. That is conflict. The synthesizer should be able to find out that this event list is mutually exclusive. Then only it is allowed otherwise it is not allowed. Let us look at another interesting example. (()) (26:31) Yes. In the earlier case this one, (()) (26:39) Okay now blocking means this statement well if there are more than 1 statement in the same block then only it is blocked. There will only be 1 statement. So it does not matter. See the blocking makes sense if there is more than 1 statement. Unless the first statement finishes the second cannot start but here since there is only 1 statement that that phenomena does not occur. But here this a b will be evaluated and will be evaluated and assigned to f . (()) (27:20)

Student: All data has to be read for a and b , isn't it?

No, no that depends on the event place. If the event is triggered so whatever is the value of a b it will take. Okay?

Student: a b in the same block so then it first evaluates f equal to a and b then...

(()) (27:43)

Student: I need always f equal to a and b .

Then that is enough, yes.

Student: Always send the same statement like 2 blocks always the same event then effectively is this the same?

This is the same block but if your synthesizer does the optimization it will find that they are identical they are redundant they will remove it. (()) (28:02) They would not get any conflict. (()) (28:07) It will, it will give an error that there are 2 blocks which are not mutually exclusive with the event list they are tried to assign with the same variable. (()) (28:19)

Student: ...always block. In hardware how it would be? How could if AND gate or OR gate at the same are hardly about?

See the optimizer again will be they will be removing the AND gate. There will be a simple OR gate. They will the last 1 will remain. (()) (28:40) That depends on the synthesizer. We do not know how the synthesizer will do. If does not optimize it will remove the, it will remove the AND gate and will not consider the delay at all. It will take only the OR gate. Okay.

(Refer Slide Time: 29:10)

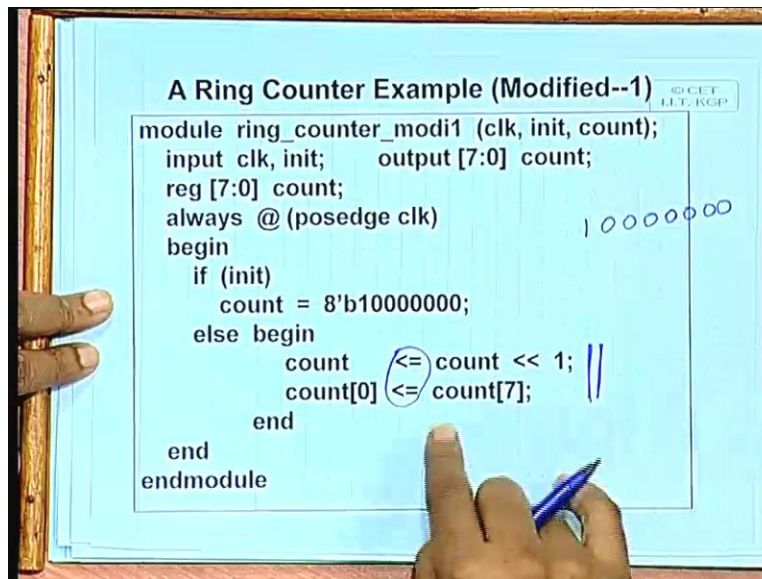
```
module ring_counter (clk, init, count);
input clk, init;    output [7:0] count;
reg [7:0] count;
always @(posedge clk)
begin
if (init)
count = 8'b10000000;
else begin
count = count << 1;
count[0] = count[7];
end
end
endmodule
```

So here we are trying to quote a ring counter module. Ring counter you remember what it is? There is an 8 bit ring counter. There is an 8 bit register where there will be a single 1

and the others are 0. They will be, this bits will be rotating. So this single 1 will be moving around. It is a left rotate ring counter. This is the 1 way in which I have coded this. Now let us inspect and find out if this program is correct or wrong. Clock init count, this output is 8 bit count, count 8 bit. This is I declare as 8 because it appears on the left hand side and we are activating this block always at the positive edge. In fact this consists of a single statement. It says that if init is active at the clock then you initialize the register with this initial value. Else we left shift count by 1. See count the way we have defined the right most bit is 0. Left most bit is 7. Then we assign count 7 to count 0.

Now tell me is this a correct of specifying? This 1, both are blocking statements you recall, first this will be executing then this. (()) (30:49) Yeah, so if this was the initial value for example. The first line will do a simple left shift on this and the value will become all 0. That 1 is lost. So here whatever you are you are assigning to count 0 is the current value of count 7. That is 0 now. So this is a wrong way of specifying this. You cannot do this. (()) (31:15) Yes, either move the second statement on top that also you cannot do. That also you cannot do because non-blocking or in some way you have to specify that these 2 are concurrent. Okay let us have a non-blocking solution and let us discuss the solution.

(Refer Slide Time: 31:43)



This is the same thing. We have changed the blocking assignments to non-blocking. The rest are all same. Now here do you force any problem. Suppose you have this initial value, you are doing these 2 things. You are evaluating count shifted by 1. (()) (32:07) You are using 7 bit of the previous count, this count 0. This is all right. But you can take the solution with a pinch of salt. You are not sure what the synthesizer will really do. Now in the first statement you are assigning something to an 8 bit register.

In the second statement you are assigning something to the list significant bit of the same register so that how can you do the 2 things together. Got the idea? Possibly this is also not the correct solution. Some simulator or synthesizer may give the correct results. Some may give wrong result because in fact this is 1 example which gives different results for different synthesizers for some results. Now a correct solution will be to do the 2 things concurrently. So the last solution we are showing.

(Refer Slide Time: 33:21)

A Ring Counter Example (Modified - 2)

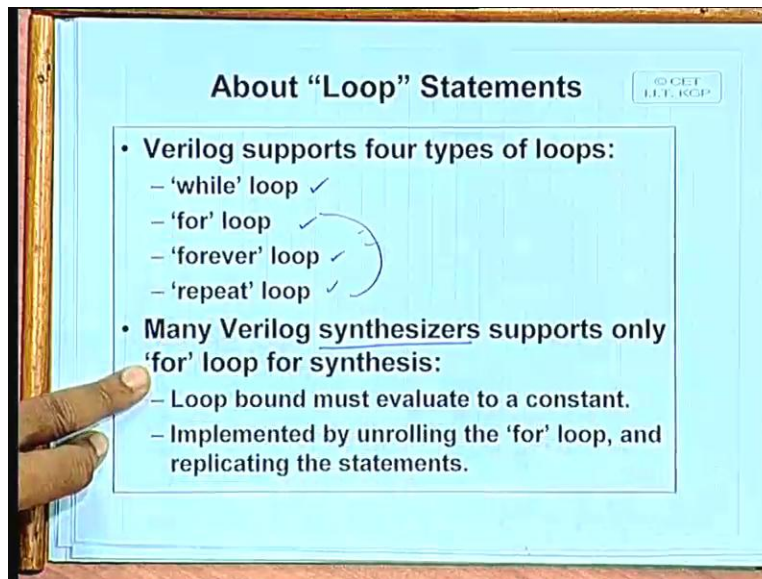
```
module ring_counter_modi2 (clk, init, count);
input clk, init;    output [7:0] count;
reg [7:0] count;
always @ (posedge clk)
begin
if (init)
count = 8'b10000000;
else
count = {count[6:0], count[7]};
endmodule
```

Diagram: An 8-bit register labeled 'count' with bit 7 on the left and bit 0 on the right. A blue arrow points from bit 6 to bit 0, indicating a shift operation. Below the diagram are two blue arrows pointing left, representing a left shift.

Here using concatenation operation we are doing this concurrently. See what we have done. This is your count. This count is say an 8 bit register, this is your bit 0, this is your bit 7, you are taking count 6 to 0, you are taking only this much. You are concatenating with this on the right. The whole thing you are assigning it to count. That is only what you want. This is a current description but it does not contain the shift operator. So your synthesizer must be intelligent enough to analyze the statement and find out this is actually nothing but a shift place. (()) (34:18)

You have given the right trigger. But in terms of hardware a shift register has a known stuff. It is already there in the library perhaps. So if the synthesizer can find out that it synthesizes to something it is already there in the library it can simply pick it up from there. Otherwise it may have to redesign the will. Use flip flops to implement this. From there it may start. So the area will be more delay will be more because the design features, there in the library they are highly optimized. That is the advantage okay. Fine, now inside the always block you can use look statements.

(Refer Slide Time: 35:07)



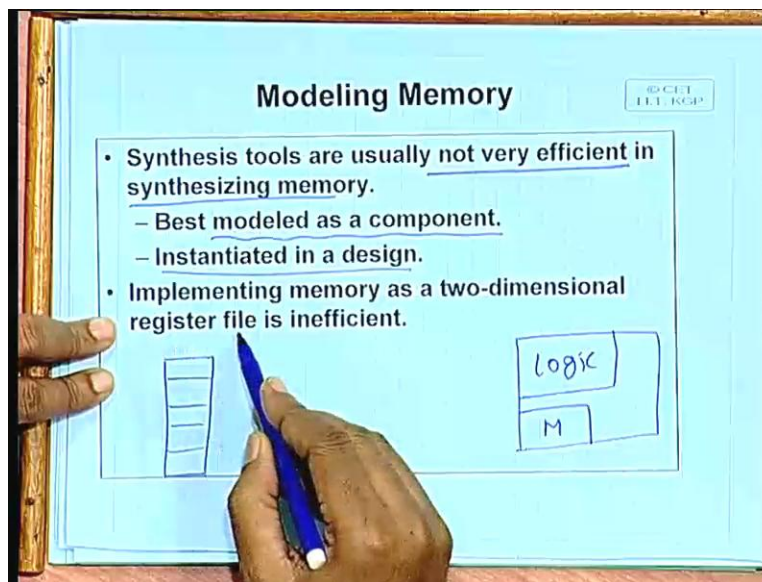
Now in Verilog I have mentioned that you can have 4 different kinds of loops. While just like C based on some condition you repeat for given some condition exactly we will see. Forever is an infinite loop and repeat is a loop which is executed specified number of times. (()) (35:32) Repeat is more readable, that is why. See for, for means you always write for C program. Someone should not like that I have to write so many things just you specify it there to repeat it 10 times, repeat 10 is much easier to specify just like that. Repeat can be repeated can be synthesized with for easily. But these are the 4 statements which are available to the user.

Well now well this issue we shall again deal with later when you talk about the synthesizable and non-synthesizable subsets. Well it depends on the synthesizers, some synthesizer supports only for some synthesizers support for and repeat. But whatever is the case they will support only loops with constant indices. Just if you write a sorting program with a parameter n and give it to the synthesizer and then if you ask to synthesize the hardware it will give an error. You will have to specify how many times the loop is executed exactly. Because the way the synthesizer will work is that it will typically unroll the for-loop. If the loop is executing 10 times it will create 10 copies of

the body, then it will try to optimize and then synthesize right. Because implementing the loop as such may not be every efficient.

Well you can do it. In some cases really need to do that. For example I am giving you 1 example. Suppose you are designing a control unit of a computer CPU. There are certain regular operations you need to execute in a loop repeatedly. There at the structural level you can explicitly specify that these are the blocks that need to be repeated but not using a for-loop. You have to explicitly specify the hardware for that a counter checking the comparator etcetera. If you simply specify for loop, it has to be a constant in terms of (()) (37:45) They do not support there. These are fine for simulation but when you are interested to synthesize into hardware the synthesizer can run into problem if you use while, okay. A few other things, well in many designs you may require some high level blocks like memory.

(Refer Slide Time: 38:17)



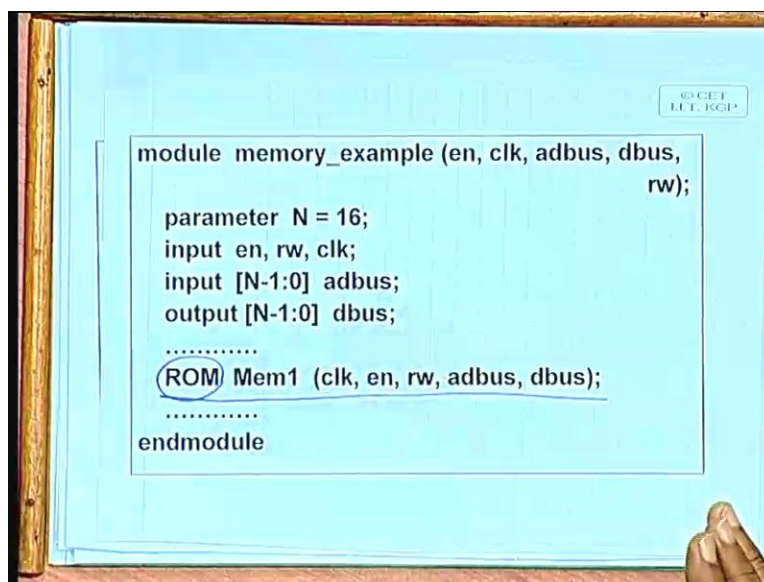
Like you may have a design which contains some logic. It can contain some memory also. Now here there are several options you may explore. Normally the synthesis tools which are existing which are there commercially, they do not synthesize memory. Well

there can be some there can be some memory modules already there in the library like I can have a 4k, 8k, 100k, 1meg memory module already there in the library. It can simply pick it up from there but in my Verilog code I specify a memory of any arbitrary size. It will automatically generate that memory. That is normally not the case. The synthesis tool they are not well you should say not at all efficient in synthesizing memory. Use memory in your design is to have them available as components in the library and in your design you can instantiate them as a component. This is the...

Student: If it is not there...

If it is not there you will have to make with either what is available or you can specify it as an array of registers. But that will be highly inefficient in terms of hardware. So you can implement memory as 2 dimensional register file. It will work correctly, but in terms of performance and power consumption it will be very bad right. So just apart from the case where you need small memory banks means small register banks. There possibly you can just use them as register file but in other case you typically use them as components and I am giving a typical example how you may instantiate that you can have something like this.

(Refer Slide Time: 40:24)



See well this is the high level block which can use module other than memory there can be some logic also but here we are instantiating this ROM component in the design. There can be other statements also but this is how you will do it. I am not going into detail of this and well when you are designing say processor or some kind of high level data paths then in order to reduce the number of driver circuitry drivers or MUX, DEMUX we often need to model tri state buffers or tri state gates. There are some explicit ways of modeling tri state gates. I am giving an example.

(Refer Slide Time: 41:20)

©CET
IIT KGP

Modeling Tri-state Gates

```

module bus_driver (in, out, enable);
input enable;    input [0:7] in;
output [0:7] out;  reg [0:7] out;

always @(enable or in)
if (enable)
out = in;
else
out = 8'bz;
endmodule;

```

The diagram illustrates an 8-bit bus driver. It consists of eight tri-state buffers connected in parallel to an 8-bit bus. The bus is labeled 'in' at the input and 'out' at the output. Each buffer is controlled by a common 'enable' signal. The outputs of the buffers are indexed from 0 to 7. A hand is shown pointing to the code and diagram with a pink marker.

This is a simple example which models a bus driver n out enable. In fact this models an array of 8 bus drivers. There are inputs and outputs. These are indexed by the numbers 0, 1, 2 up to 7. These are in 0. These are out 0 out 7 and you have a control unit called enable. This enable goes into the control of all these okay. If you specify like this, that if some of the input changes state either enable or 1 of the inputs then well if enable is high means the buffers are enabled then in goes to out; else we explicitly specify z here; tri state. This high impedance state goes to out. So if there is explicit assignment like this. Then this will be modeled as a tri state gate but in continuous assignment if you drive 2

computations into the same net. By default it will not synthesize as tri state. Explicitly you will have to specify in this way that you require a tri state gate out here. Then only then it will be synthesized as a tri state. Right. So this is actually 1 way we can do that. Okay so (()) (43:17) Yes. (()) (43:25)

Student: It becomes indeterminate.

It will become indeterminate, it becomes indeterminate. Simulator will give some x value in the result. Indeterminate value and possibly during synthesis you will be getting a conflict error.

Student: If we do not synthesize it will use the register for out a latch for ...

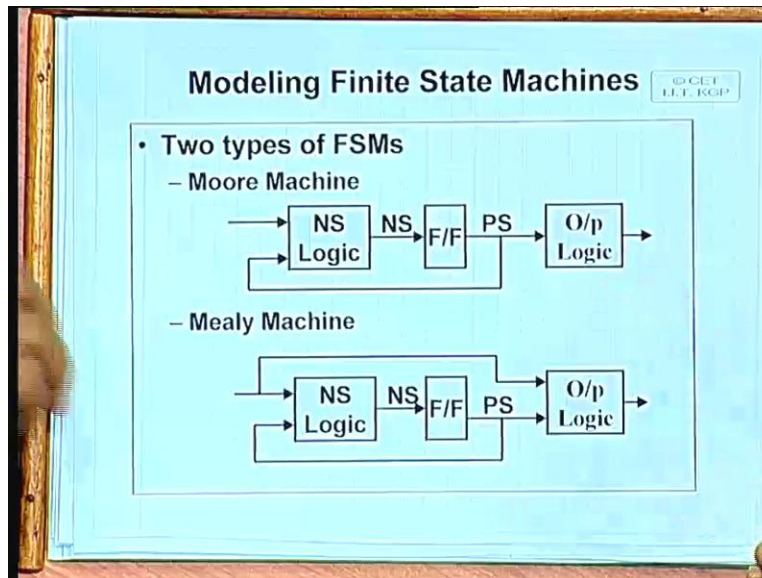
If you do not specify this, if you do not specify the else part then this will be a simple latch, yes, then this will be an 8 bit latch right. (()) (44:07) Previous one because enable you do not know when enable will be active. (()) (44:16)

Student: In that you do not restrict.

That will not restrict. You will have to explicitly specify that that under certain condition the value will become z. Then only you will be having a tri-state value. So we have seen so far that using Verilog language we can model combination logic. We can model some sequential logic also. Now we will see specifically how we can model FSM - finite state machines. See using if then else you can do fine. But normally finite state machines they are specified either using a state diagram or a state table. From there they should be a very structure and well defined way of specifying a finite state machine and if you follow those well defined styles, then it will also be easier for the synthesizer to identify that well the user is actually trying to model an FSM and let me use the best known method to synthesize it. So there are some design styles which are there. If you follow the design style it becomes easier for the synthesizer and it is always good to use those design styles. So you can always give a big array of if then else statements, but that can be a little

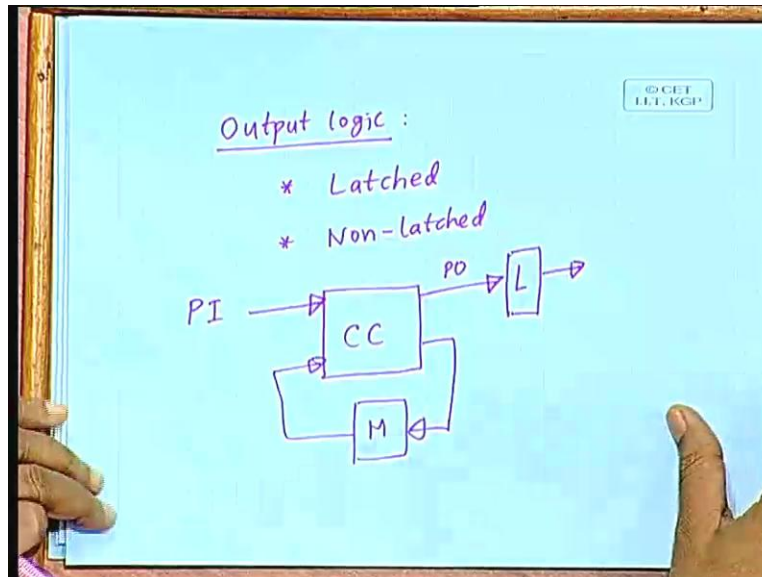
problematic for the synthesizer to get an optimum realization. So these are the things we will be discussing little in the next class.

(Refer Slide Time: 45:55)



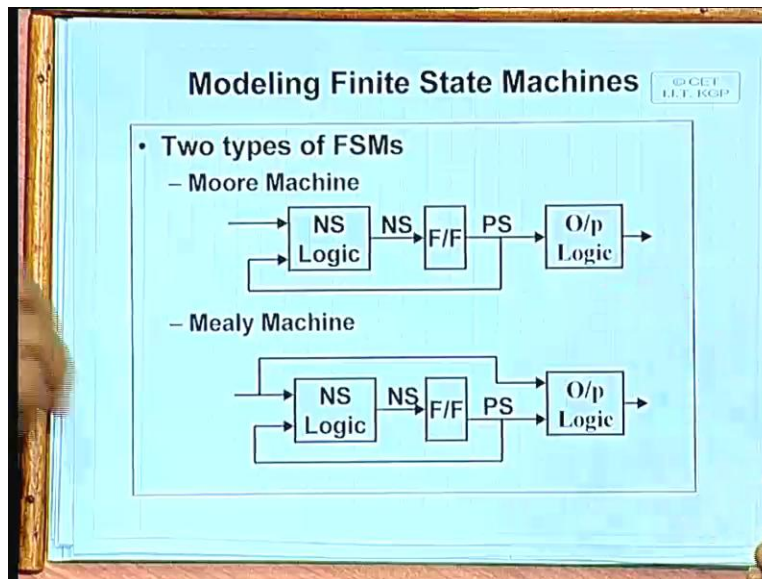
But just let me tell you now that we would be broadly looking at 2 different kinds of FSM as you know. Moore and mealy, both the examples we would be looking at. Now if you recall Moore machine, Moore machine the next state logic generating the next state this depends on the present input and the present state. Whereas the output logic depends only on the present state that means a typical example is a counter. Counter the output value does not depend on the input. Whatever is its present state that is the output? Well in contrast in a mealy machine the output at the present point in time may also depend on the input you are applying. So present state in combination with the input will give you the output. Now here again we would looking at the different design styles that we can use. We can follow like say for instance you have the output logic in both cases.

(Refer Slide Time: 47:12)



Now it depends on the application, the output logic can be latched. It can be non-latched like say for example you have a circuitry. You have a, some combination logic. Here you have the primary inputs which are coming and the primary outputs which are coming out. There can be some latches out there because there can be some scenarios where the output values need to be kept stable as the inputs change. So there you possibly need a latch from where the outputs are to be driven. So there are different design styles. You will be having the memory elements out here; they will be driving the memory elements in that period. So the different design styles we would be looking at.

(Refer Slide Time: 48:18)



So that the output logic either in case of the Moore machine or in case of the mealy machines they remains consistent and they will be latched or non-latched. It depends on the situation. This means all these things we would be discussing in the next lecture with the help of some example. We will possibly take an example of a simple; simple FSM like a sequence detector and in the example you can see that how you can model it using Moore machine or mealy machine you can see that how you can synthesize it. After that after looking at the synthesis of FSM we would talk about some good design styles or what to do and what not to do. Synthesizable and non-synthesizable, when your target is for synthesis what are the language features you should limit yourselves to and what are the things you should not use. There are some rules and guidelines and in fact as I told you this rules and guidelines there are separate such rules and guidelines from each and every vendor. There is nothing consistent. But broadly some of some of the things are common but there are a few things which may vary across different synthesis tools. So here, all these things we will discuss in next class. Thank you.