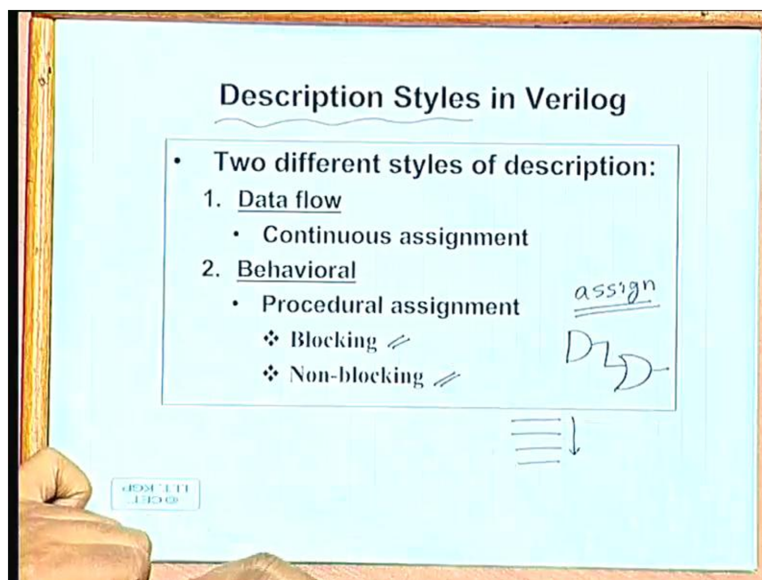
Electronic Design Automation
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No #04
Verilog: Part III

So in this lecture we would be continuing with our discussion on the language Verilog. Now if you recall what we were discussing in the last lecture, we talked about the different logic values that the language Verilog supports. We talked about the primitive gates which are supported as part of the language. We looked at some of the hardware modeling issues and finally we looked at some of the operators that we can use when we are describing a piece of hardware using Verilog. Now continuing with our discussion today we start by stating or explaining some of the description styles which we may use in Verilog.

(Refer Slide Time: 01:58)



See the language Verilog supports some features. There are some instruction some statements and operators you can use them but you should also remember that there are several description styles which are available in the language. The same thing we can

specify in possible different number of ways. Now there are implications. The description style we choose that may be differently interpreted by the simulator or by the synthesizer. So it is necessary to know clearly what the different styles are and what they really mean. So broadly speaking there are 2 styles of description. 1 is the data flow description; other is the so called behavioral level description. Now the main difference between these 2 is that in the data flow description we talk about continuous assignment like you recall the assign statement that we had looked at in the examples earlier.

So assign statement is an example where we are making a continuous assignment on a net type variable and we have mentioned that this kind of style is typically suited for modeling combination logic where the output of a gate may be driving continuously the input of another gate okay. There is nothing like clock or anything else involved. Now in contrast we can have another style of description. Behavioral; this is more like you are expressing something in a high level language. You specify a set of statements which are supposed to be executed 1 by 1 okay. But exactly how they are executed that depends on whether you declare the assignment as blocking or non-blocking. These we would be discussing in some detail. So first let us again look at the continuous assignment data flow style.

(Refer Slide Time: 04:11)

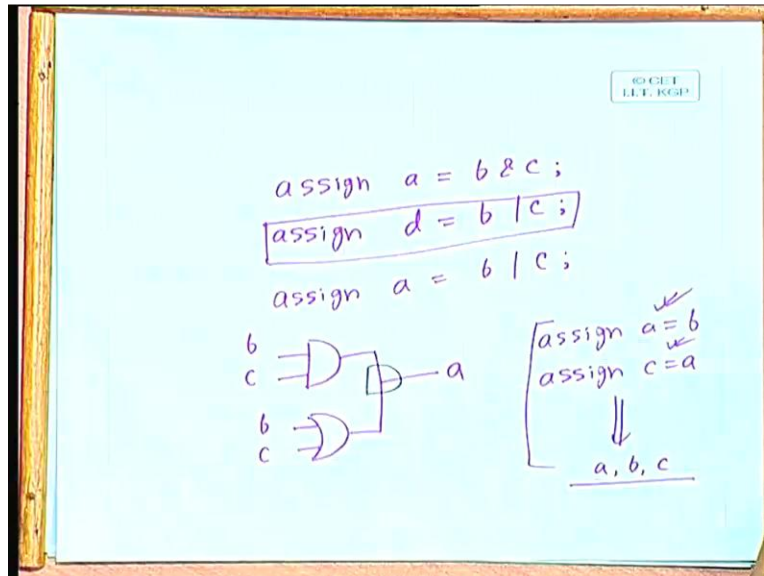
Data-flow Style: Continuous Assignment

- Identified by the keyword "assign".
  - assign a = b & c;
  - assign f[2] = c[0];
- Forms a static binding between
  - The 'net' being assigned on the LHS,
  - The expression on the RHS.
- The assignment is continuously active.
- Almost exclusively used to model combinational logic.

So in the continuous assignment style which you have already seen in the examples I have given earlier this is specified using the keyword assign. These are some typical examples assign a equal to b and c or assign a equal to c. Well here the idea is that whatever you specify on the right hand side for example in this case there is an AND gate which will be synthesized b and c which will be continuously driving a variable a. This, a, is not a register. This, a, has to be a net type variable. So, for a continuous assigning statement it is essentially the assign keyword or the assign statement it creates a static binding between whatever you are evaluating on the right hand side and the variable which you are which you are assigning the value to on the left hand side.

Now the variable you are assigning to in the left hand side. This has to be a net type variable it will be, it cannot be storage because it is storage it cannot be continuously driven. It has to be something like a clock okay. So and 1 characteristic of continuous assignment is that, it is continuously active. Now continuously active means with respect to simulation. The value of the outputs does not change at particular points in time like when there is a clock pulse. So whenever there is a change in the values of b and c then after a delay equal to the delay of this gate this output will immediately start changing. (()) (05:56) Yes. (()) (05:58). Okay, so the question is that in say in a program if there are 2 continuous assign statements.

(Refer Slide Time: 06:11)



Say assign, a equal to b and c. So there is another assign d equal to b OR c. Oh, you are saying that second assign is also? On the same variable. On the same variable, yes this you can have. This I have mentioned earlier that this is a net type variable which is driven by 2 different gates. Now it is your responsibility to ensure that a, is either a wired OR wired AND or a lets call it a wired type. Because essentially this will mean that you are driving this net a, from both these gates. Now unless you specify anything else this will lead to a simulation error or even this will lead to a modeling error. But if this, a, is declared as wired AND, and wired OR then there will be an explicit AND, and OR gate synthesized out here or if these are tri state gates you can have tri state controls. That is also right. Yes.

Student: Assign a equal to c.

Okay

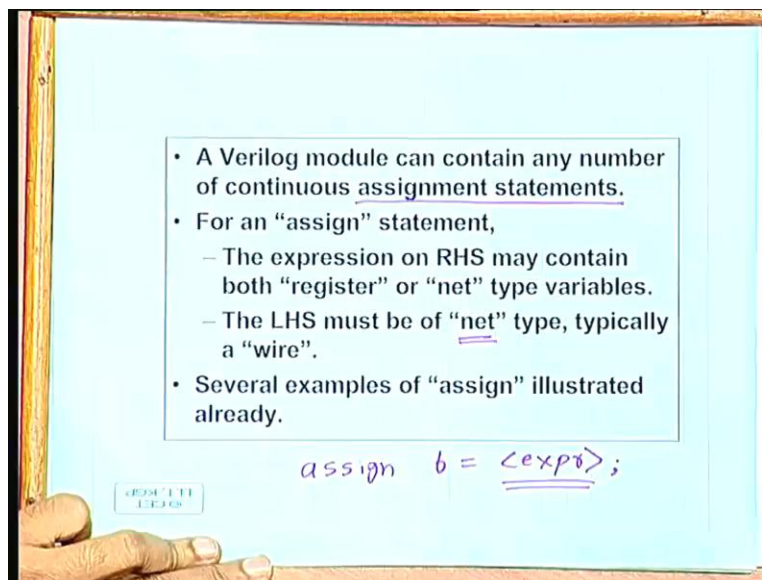
Student: And then assign b equal to a.

Okay.

Student: So all the 3 a, b, c will be wired.

You are saying if there is a statement assigned to a equal to b; a equal to b and assign b equal to a something like this or b equal to c; c equal to a. Well now in this example see this different assign statement means they are all executed concurrently okay. So a, and b have a correspondence; c and a, are correspondence. So this a b c will be referring to the same net. So logically same. (()) (07:58). Yes, yes. So it is not that first this statement will be executed and then then this statement will be executed. They are all executed together. So essentially this synthesizer will be synthesizing as a single net which may be referred as a may be referred as b, may be referred as c same okay. Okay.

(Refer Slide Time: 08:25)



So a Verilog module whenever you are defining it can contain any number of continuous assignment statements right. So as I had given the examples and as I said that the different assignment statements which are there using the assign keywords they are all logically being executed in parallel concurrent okay. So there is no ordering of execution

as such. (()) (08:47) Yes, whatever the new value comes in it takes the net variable on the left hand side takes the value immediately. Now, (()) (08:59) after a delay of the gate delay. Of course yes and that net if it appears on the right hand side of any other assignment statement that will trigger the execution or computation of that immediately. So this continuously goes on. So as I said this is a natural way of modeling or representing typically combinational circuits.

But we will see later that even using assign statements we can we can model a sequential circuit also right okay. Now there are a few other things. This I had mentioned that are assign statements. So I have assign b equal to something some expression. The first thing is that the expression this can consist of any number of variables which can be of any type. They may be register data type they may be net data type. So the sources of this assignment or this expression can be coming from any kind of variable but when you are assigning it this must be of net type. This is a restriction for continuous assignment statement. The left hand side has to be a net type variable right. Next some of the examples we have already seen earlier. So now let us look at a few more examples that how these kinds of assigned statements can be used to synthesize some particular kind of you can say hardware blocks like this is a very simple example.

(Refer Slide Time: 10:30)

```
module  generate_mux (data, select, out);
   input  [0:7]  data;
   input  [0:2]  select;
   output  out;

   assign  out  =  data [ select];
endmodule
```

out = data [2] ;

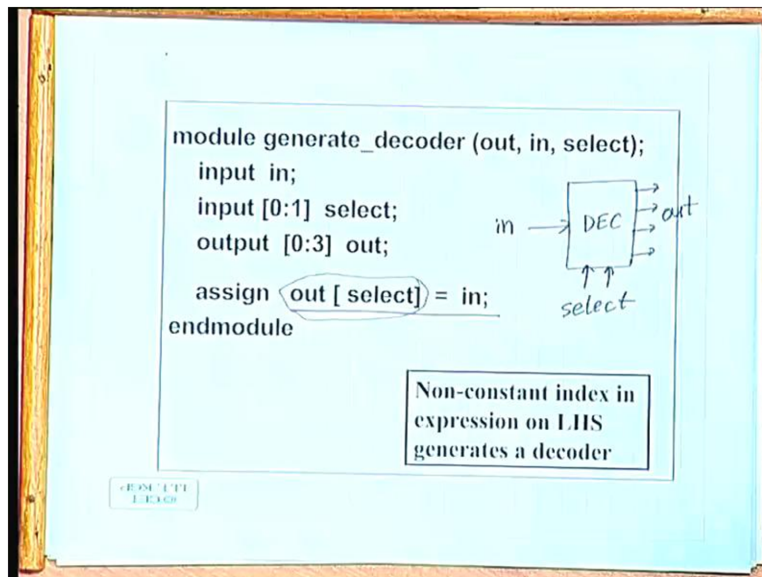
Non-constant index in expression on RHS generates a MUX

This is a module which takes 3 parameters data, select and out. Data is an 8 bit vector, select is a 3 bit vector, out is single bit and here we give an assign statement out equal to data select. Now you understand one thing. Right hand side contains a variable which is defined as a vector type and the index we are we are accessing it to the index and the index is again a variable. This is not a constant. So if the index were a constant like if it was something like this out equal to data 2 then simply the bit number 2 of this data would have been greatly connected to out. But since this is driven by a variable this will be synthesized as a multiplexer. So if you give this give this for synthesis. So this is data, the 8 bits of data. So these 3 bits of select will come here. So this is select and this will be out. So this synthesizer is intelligent enough that whenever it encounters a, you can say vector type register more than more than 1 bit with a variable as the index.

Then automatically a multiplexer is realized provided this this appears on the right hand side of the expression. This is appearing on the right hand side of the assign expression. So as a result a multiplexer is getting synthesized. But if this appears on the left hand side. Yes. (()) (12:28). Now out this is an assign statement, this is taken as default as well because I have told you if you do not define anything depending on the context, you are

using some default is getting assigned. So in this case since you are using as the left hand side of an assign it will be taken as a wire. But it is always good to explicitly declare all the variables, all the parameters you have defined, input output port. That is a good practice well this kind of a variable with a variable index if it appears on the left hand side like this example shows.

(Refer Slide Time: 13:12)



In this example the right hand side does not contain an expression as such a single input data bit in but the left hand side contains a variable with a variable index. Now this will get synthesized into a decoder. There will be a single input in there will be 4 outputs these are the out and there will be 2 select lines depending on what the value of select is this in will be going to 1 of these 4. So this is something you can you can say easily understand and easily correlate that if such a thing appears on the left hand side then in the synthesized hardware you can expect to say a decoder. If it appears on the right hand side you can say a multiplexer okay. Now there are ways to generate you can say set of multiplexers or a vector of multiplexers like I am giving another example.

(Refer Slide Time: 14:26)

```
module  level_sensitive_latch  (D, Q, En);
   input  D, En;
   output  Q;
   assign Q = en  ?  D  :  Q;
endmodule
```

Using "assign" to describe sequential logic

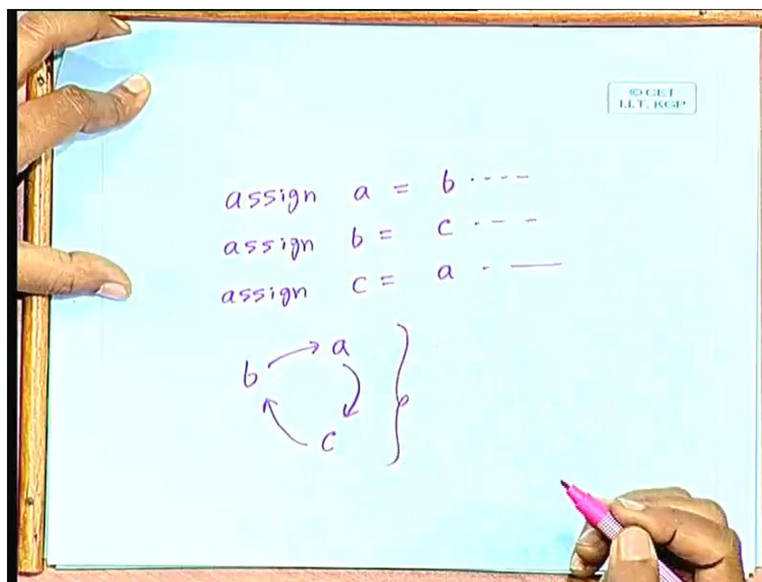
See this, a, b, f and select. These are the 4 parameters where a, and b are vectors of size 4. This f is also the same and we are giving an assign using an implicit if statement. If select, then a b. So effectively what we are saying is that since this a b f these are all vectors of size 4. So in this case what will happen? This synthesizer will be synthesizing 4 multiplexers not 1. So this multiplexers will all be 2 line 2 1 multiplexers because this, if is selecting 2 things and out of these 2 things you are selecting 1 of them. So each of them will be having 2 inputs. So basically this will be a 0 and b 0, a 1, b 1, and so on, a 3 and b 3. Similarly output will be f 0, f 1, f 2 and f 3 and all of them will be driven by the same select line sel all the 4.

So this single statement will represent a set of multiplexers, the number of which will depend on the size of the vectors. These are implicitly synthesized. So now we shall show an example where even using assign statement you can synthesize a sequential logic because, this assign statement does not necessarily always imply that the logic you are generating will be combination in nature. See the idea is that you will have to understand or the synthesizer will have to understand if there are any conditions where in which you

will have you may have to remember the last value of some variable. In that case you will have to synthesize a latch or a flip flop.

Just look at this simple example. This example model is a level sensitive latch so what we are actually trying to model is that latch with a input called D and output called Q and enable en. So every time En is high this D will be latched. If it is low it will remember the last value. So inputs D and E are in and output is Q. You just look at this assign, Q will be equal to D if en equal to high. Q will be equal to Q if en equal to low. So this same variable Q is appearing on the left as well as on the right hand side of an expression. Now if this kind of a cycle exists in the assignment this means that you are trying to remember something from the previous step. In a similar way you can have not necessarily a single statement.

(Refer Slide Time: 17:50)

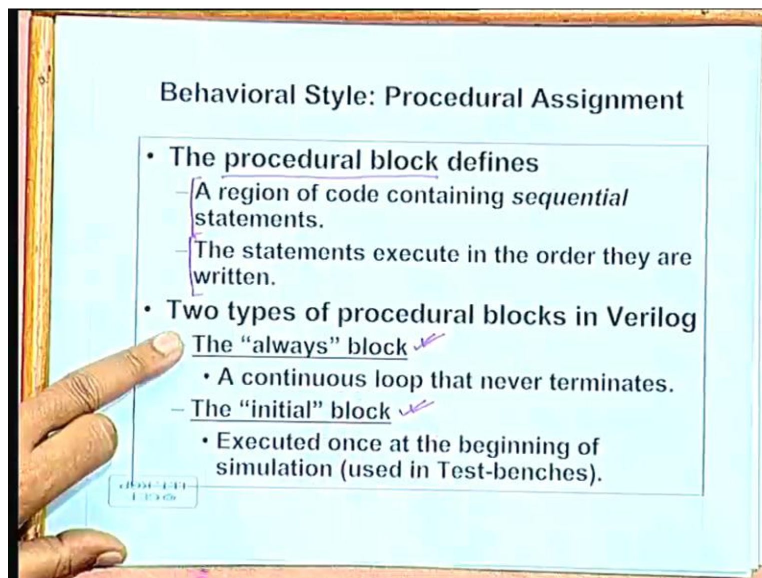


Suppose you have a scenario where you are assigning a equal to something which depends on b there is something else. There is another assign b equal to say c and something else. There is another assign c equal to a, and something else. So this means that you have a dependency in the first statement. The value of b is used to compute a in

the second statement the value of c is used to compute b and in third statement the value of a, is used to compute c. So here also there is a cycle. Cyclic dependency among these 3 variables. So the wherever you have this kind of cyclic dependency you must have a latch otherwise your logic evaluation will go wrong.

Right? So essentially this case as well as the simpler example that I have shown. This synthesizer will essentially check whether there is a cyclic dependency with respect to some variable on the assigned statements. So if such a cycle is detected this will automatically be translated into a latch in the synthesized hardware. Fine. So these are the continuous assignment statements using assign okay. So now let us look at the other style the so called behavioral style.

(Refer Slide Time: 19:24)

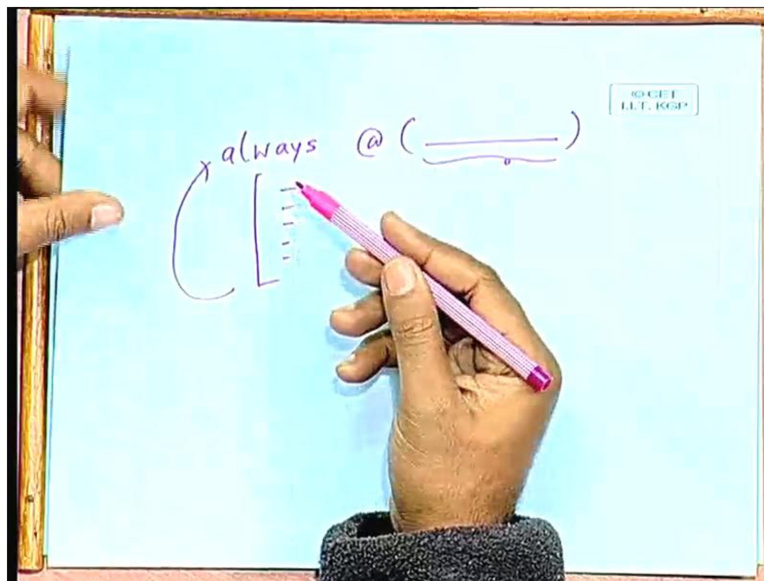


Now the behavioral style which is sometimes also called procedural assignment statements or procedural assignment block such a block defines 2 things. First just like any high level language. You write a program in any high level language. You can define a block of statements say in C you define a block of statements using the curly braces. So similarly here also you define a region of code which will contain statements which are

supposed to be executed one by one right? So just like a procedural language high level language like C you have a region of code which contains sequential statements and your understanding is when you are writing the code there the statements will be executed in the order they are written.

But in terms of the hardware realization you can understand that you cannot always necessarily guarantee this that they will be executed strictly in the order they are written that gives some priorlism also. Now in fact there are 2 different ways you will see that we can execute this statements. This we will come a little later okay. But before that let us tell do one thing that the procedural block whatever you are saying in Verilog there are 2 types of such procedural block you can use okay. One we have already seen in a couple of examples earlier. You have a, always block. This always block is a loop which never terminates.

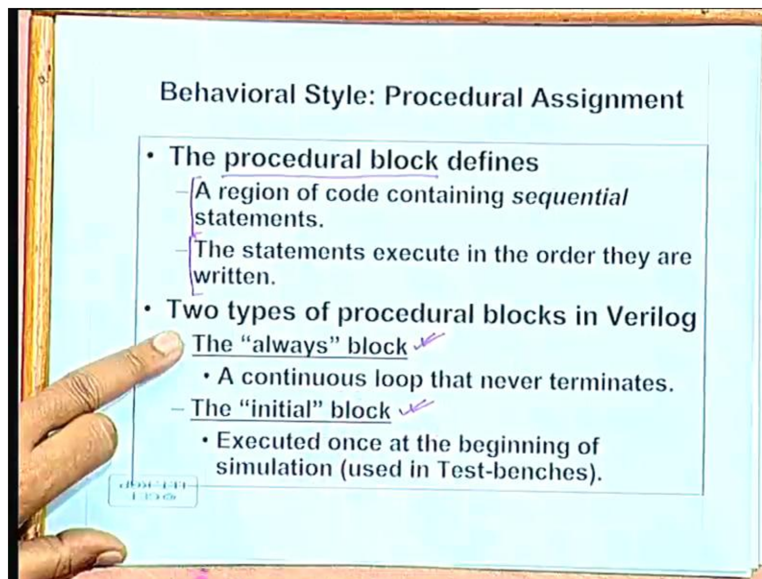
(Refer Slide Time: 21:10)



So the idea is that whenever we have an always block always you recall there is something like this. Always at within parenthesis we have some triggering condition posedge clock or some variables. Then you have a block of statements in between. Now

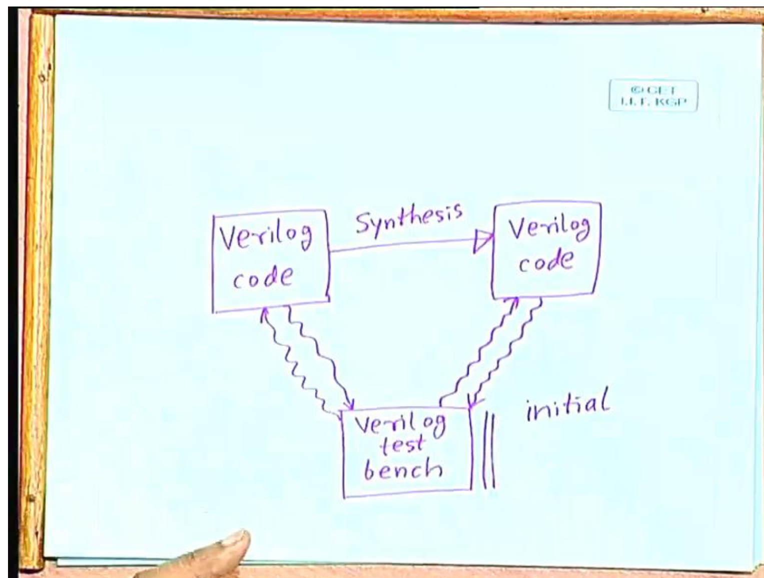
this always block is like an infinite loop. It continuously goes on executing and every time this event expression evaluates to true this block gets executed once. This block never terminates. This goes on executing indefinitely. (()) (21:43) Statements it depends again. I have told that there are a couple of alternatives. We will see this very shortly that exactly how these techniques are executed. But whatever they are sequentially or parallel whatever they are executed every time this triggering expression holds true and this is executed continuously in loop. This never stops.

(Refer Slide Time: 22:07)



But in contrast there is another kind of a procedural block called initial block. This is executed only once. Executed once at the beginning of simulation. See there is something called a test bench. This initial block is used in a test bench. Let us try to see what is a test bench? (()) (22:29) Initial block is not a synthesizer.

(Refer Slide Time: 22:32)



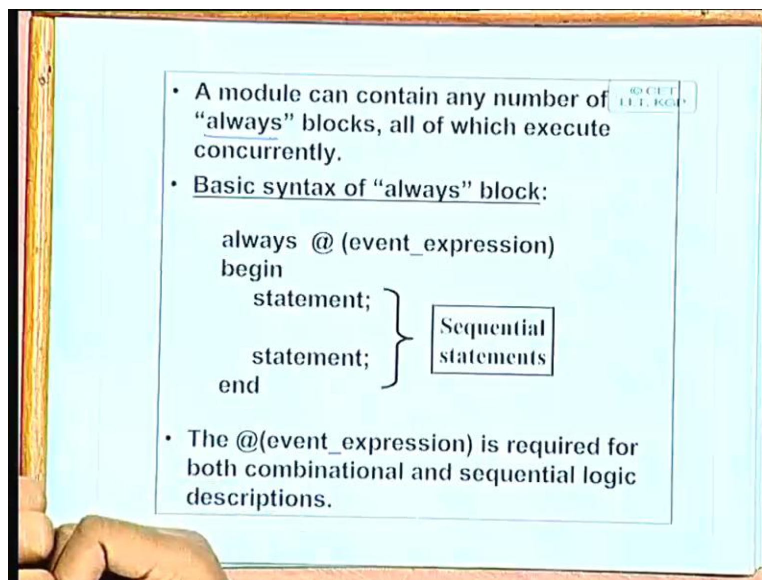
The idea is that suppose you have written a Verilog code. This Verilog code can be written at the behavioral level also at a high level. Then possibly you are using the synthesizer, you are carrying out synthesis to get a low level description, may be a gate level netlist. This can also be a Verilog code. After synthesis the synthesizer can also generate the synthesized netlist in terms of Verilog. Now the problem is that after every translation it is always better to check whether this specification and whatever you have got they are matching. This is done with respect to something which is called Verilog test bench. This is something which is again written in Verilog.

A Verilog test bench what it does is that, this is basically a Verilog program which is calling this Verilog module and it is explicitly applying some input values to it and it is checking what is the output. It is exactly during simulation whatever you are expected to do. That is done by the Verilog test code. So the same Verilog test bench may be exciting this code and evaluating the response. Also it can be exciting this code and evaluating the response. The same Verilog test bench program can be used once to check this and another time to check this or to compare whether the outputs of the 2 are the same or not.

Now as you can see a Verilog test bench is something which is used only for the purpose of verification through simulation.

This has to be executed once when you are actually simulating. So it is not that you are you are synthesizing a piece of hardware which has to be executed continuously. And in the Verilog test bench you have that initial block which is the analog of the always block which is executed only once right. And as we have correctly said test bench is a program which is not synthesizable. You will see that you have statements which can read some data from a file which can display some values on the screen. Those kinds of statements can be included in this okay. Fine, right. So the initial block will come a little later when we talk about test benches. So for the time being we start by looking at the always block only.

(Refer Slide Time: 25:35)



Now a module can contain any number of always blocks. There can be 4 always blocks and the semantic says that all the 4 always block are executing concurrently. They will correspond to 4 different pieces of hardware which are executing without any interdependence. This is the basic syntax, this always block we have already seen. This

always at the rate, there is something called an event expression. Now every time this event expression evaluates to true the block starts executing. The block is bounded by the begin and end keywords. The begin and end keywords will identify the boundary of the block and there can be several statements in between.

Now this event expression as we had seen in the example earlier this we give both for modeling combinational as well as sequential logic. For sequential logic if there is an edge of the clock which is specified, posedge clock or negedge clock that that implicitly means that we have time synthesized a sequential logic or for combinational logic you can even say some combination of variables always at a or b or c. If either a or b or c changes it value you want to evaluate this block but whether it will be synthesized into combinational or sequential, that depends on a number of other factors. We will see shortly, okay, fine. So this is the syntax of the always statements and there are a few things.

Now in the always statements you see this always statement is triggered by some event. Just try to understand, this is triggered by some event. So you really do not know when the event will take place and in the statements this will typically be some assignment statements. Some expression is being evaluated to assign to some variable. Suppose some value was assigned to a variable. Now you really do not know when the next event expression will take place. So till then you will have to remember or memorize the value. So this implicitly means that the left hand side of those expressions or the statements they cannot be a simple net type variable. They will have to be a register type variable reg type variable. (()) (27:58) Yes.

Student: On the left hand side…

Yeah, always block on the left hand side has to be reg type variables because you really do not know when the condition will be evaluating to true. So till then the value have to be retained to its old value okay. So, (()) (289:18) Yes

Student: Whenever 1 value changes the other values after gate delay so it is …

See there are a number of statements which are which are running concurrently. But their sequential whatever you are saying is that there is a logical dependence between say the output of a AND gate is driving the input of an OR gate. But in terms of the hardware there is no one which is telling the OR gate that you please do not evaluate until the AND gate output becomes true. Now just if you just look at the simulation results, you will see that there will be a period of time when the output of the OR gate will be indeterminate. Finally it will be settling to a stable value. (()) (29:02) Well it is sequential at a low level of granularity where the sequential nature is driven by the delays of the gates not by any clock or any external event. (()) (29:17)

Student: …delay 0 will there be some…

Well you can specify the delays but means I also mentioned whether the delays are meaningful only for the purpose of simulation; not for synthesis. When you are doing a simulation if you specify some well defined delays it will be easier for you to interpret the results. You can say that after 10 units of time this is supposed to change state. You look at the timing diagram and see that well yes this is after 10 unit time some value it is changing. (()) (29:48)

Student: driving an OR gate.

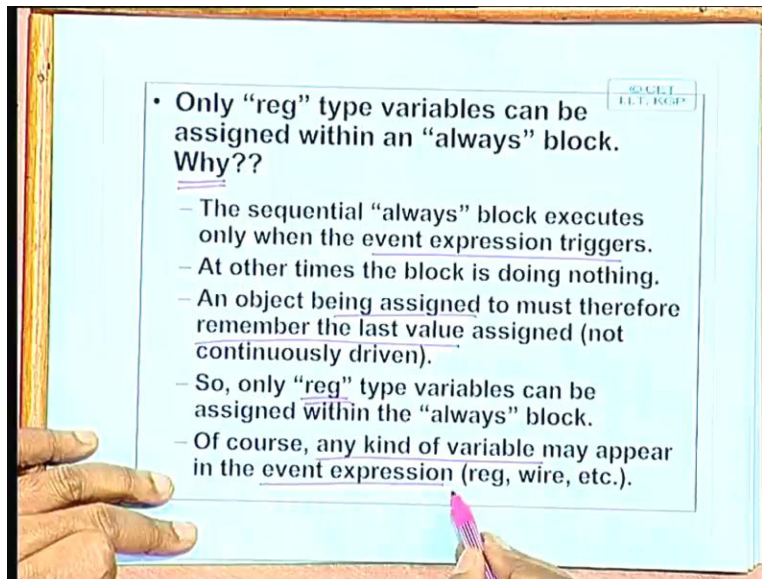
And it is driving OR gate.

Student: The OR gate will help change the value or…

See if you make the delay value as 0, then the simulation result can give some inconsistent values, depending on the order these statements are appearing. They will evaluate in that order. (()) (30:10) That depends on the Verilog simulator you are using not really the computer Verilog simulator you are using. This can vary from one

simulator to another okay. Fine. Then the always statement as I had mentioned. So let me just again repeat this step.

(Refer Slide Time: 30:31)

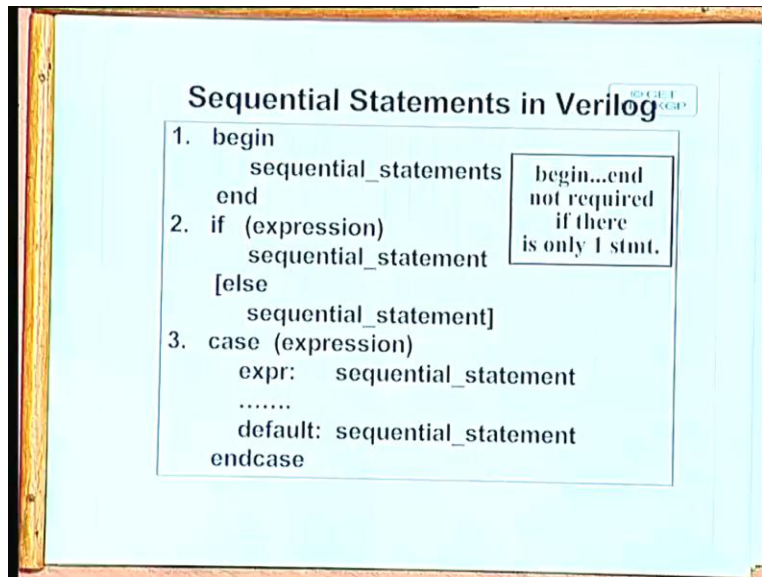


That only reg type variables can be assigned within an always block. This I have already given the justification. Because this will execute only when the event expression triggers and all other times the block is doing nothing. It is lying in an idle state. So the variable you are assigning to must remember the last value. This is just what I had just mentioned. So in always block the left hand side has to be a reg type variable. But however on the right hand in the expression you can use any kind of variable may appear in the event expression or also the event expression. So let us again go back to this one. So in this statements the right hand side of the statements as well as the event expression you can have any combination of any kind of variables.

Net type reg type, but only in the left hand side of these statements this must be reg type. This is the restriction. Okay so before talking about concurrency within an always block there are 2 alternatives out here. We will come to that. Before that let us very quickly look at the different sequential statements that are there in Verilog. I have mentioned that

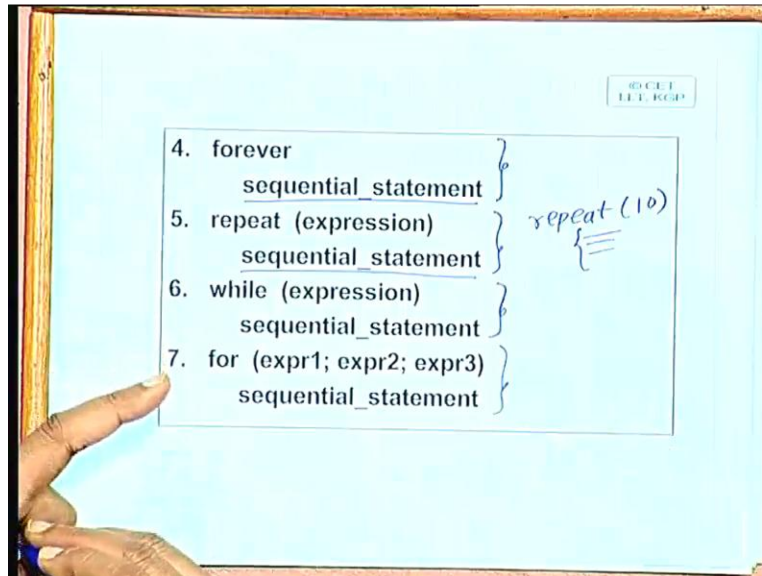
within a, always block you can have several statements which are sequential 1 by 1. Now the sequential statements that are supported in Verilog are like this.

(Refer Slide Time: 32:00)



First of course is the conventional begin end. Just within begin end you can have any number of statements and just like C or any other language mostly used that if there is a single statement in the block begin end is optional and you can have an if else kind of a statement. If expression, then a block, else a block. Now inside this blocks you can again have begin end if there are multiple statements, this else is optional. Then you can have case exactly like C. The syntax is very similar to C case within bracket the case expression. These are the different values. This expression will be evaluated, if it matches this then execute this. There can be number of such and if does not match with anything then a default, just like C.

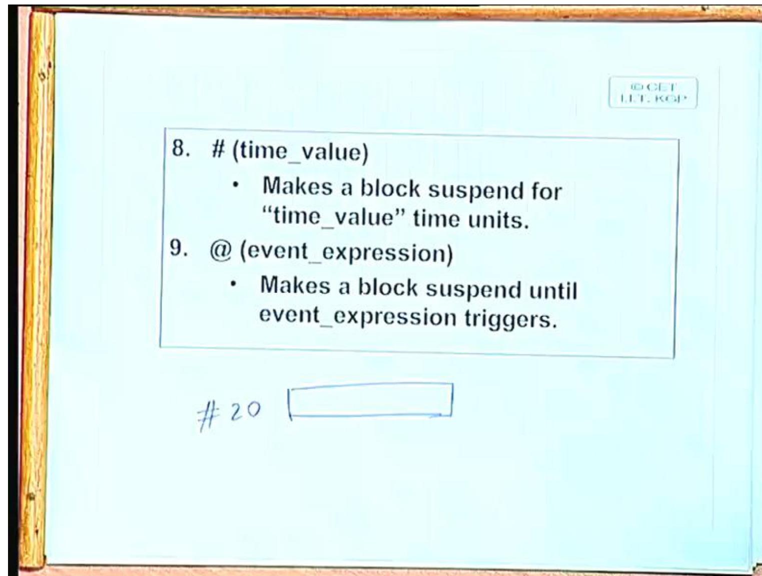
(Refer Slide Time: 32:55)



There are a couple of other kinds of statements which are not available in a language like C. For example this represents an infinite loop, forever a block. These set of statements are supposed to be executed indefinitely. This you can have and in this version you can say that I want to execute it so many times. I can say repeat 10, then some statements. This set of statements will be executed 10 times. So if you want to execute them a fixed number of times you can use the repeat statement. This while statement is exactly similar to what you have in C.

There is an expression which is evaluated and if it is true you execute this otherwise next statement. Yes. So after repeat the block of statements you want to repeat, see all this statements I am saying sequential statement. This can be macro statement a begin end block. This can be a begin end block. If it is a single statement it can just lie as it is. But if it is a more than 1 statement this will lie in beginning. (()) (34:09) And this again this for expression is exactly the same as what you have in C. So these are the different options you have from me and a couple of other things. These also we have seen in examples.

(Refer Slide Time: 34:32)



That you can specify some time value using the hash, say hash 20. Then you can give some assignment statement. This actually specifies a number which is a multiple of a basic time unit. This is I am repeating. This is meaningful only to the simulator and this specifies the block of statement that follows the block that follows. This can be a single statement, this can be a begin end again. This block has to be suspended for so many time units. So after so many time units you execute this. So in a sense you are specifying the delay of this block that means after how much time the output of this block will get evaluated okay and at event expression. This I have already mentioned, this makes a block suspend until event expression becomes true or it triggers. Okay, so now let us look at a few examples for you will see that how we can use this always block for instance to model different kinds of logic.

(Refer Slide Time: 35:48)

```
// A combinational logic example

module mux21 (in1, in0, s, f);
   input in1, in0, s;
   output f;
   reg f;

   always @ (in1 or in0 or s)
     if (s)
       f = in1;
     else
       f = in0;
endmodule
```

First let us take an example of a simple 2 line to 1 line multiplexer. There are 4 parameters. So actually what we are trying to model is that there will be 2 inputs, in 0 and in 1. There will be 1 select line. (()) (36:11) Okay, so he is asking that how do I know that that whether this will be synthesized into a combination logic or a sequential logic. Well here we are trying to address exactly that question. So let us look at the couple of examples. We will just understand how this decision or means or actually how does the how the synthesizer know that that exactly what to synthesize.

Just you look at this example first. This says that you have in 1 in 0 s f. Well since we are using inside the always block this f I am declaring as a reg type variable, this always block number 1 thing is that it does not contain any clock okay. So this does not directly imply that we will have to use a sequential logic. But you will have to see the internal of that block in order to finally conclude. Now let us see. This always block will trigger if any of the inputs change in 1 or in 0 or s. This says the computation says that if s is 0 then then we assign in 0 to f. If s is 1 we assign in 1 to f. See you are taking a decision based on the value of s and along all the parts of this decision process you are assigning something to f.

There is no part in this example at least where you are not assigning anything to f. So there whatever is the value of s something will get assigned to f. This implies you need not have to remember anything from the past. The current execution block. (()) (38:11) Yes, yes, yes. If there is no else part then you will have to remember the last value of f. So if you have specified all possible parts and the corresponding values that need to be assigned to this f you need not synthesize into a sequential logic. So, (())(38:33) if as a register

Student: …input values to a register… you need combinational logic.

Yes, yes. (())(38:44) Now 1 thing see this f is a register does not mean that the logic you are realizing to assign to f has to be sequential. See here we are synthesizing the logic that is used to generate the value of f.

Student: But sir it will synthesize some value, some hardware that can store a data.

See, reg whenever you declare then the synthesizer has an option. You can synthesize it as a register if required or if you see that it does not require storage really, you can also map it into a wire

Student: So optimization.

Some kind of optimization. So reg does not always imply that you are generating a register. Okay fine. So these are example where you are synthesizing combination logic. Now for sequential logic we look at a couple of examples. First you see that we have an example out here where we have a sequential logic where which is say just let me repeat once.

(Refer Slide Time: 40:05)

```
// A sequential logic example

module dff_negedge (D, clock, Q, Qbar);
   input D, clock;
   output Q, Qbar;
   reg Q, Qbar;

   always @ (negedge clock)
     begin
        Q = D;
        Qbar = ~D;
     end
endmodule
```

Say here we have an example out here. So we are trying to synthesize a D flip flop in this case. The input will be D. You have a clock and you have both Q and Q bar. This is what we are trying to synthesize. Now in this case we are explicitly saying that always at the negative edge of the clock. So this will be a negative edge trigger. So always at the negative edge of the clock we assign D to Q and D bar to Q bar. So this is something which is triggered by the clock. So implicitly this has to be a sequential logic because this is clock trigger.

See everything if in the always block there is a phase where the negedge or the posedge of clock is used this implies that you must have it realized or implemented as a sequential logic. Because whatever assignments you are making that will be done only when the clock edge comes. That can be done only if you have a sequential logic out there. Fine, now let us look at an interesting example

.

(Refer Slide Time: 41:28)

```
// Another sequential logic example

module  incomp_state_spec  (curr_state, flag);
  input    [0:1]  curr_state;
  output   [0:1]  flag;
  reg      [0:1]  flag;

  always  @ (curr_state)
    case  (curr_state)
      0, 1  :  flag = 2;
      3     :  flag = 0;
    endcase
endmodule
```

The variable 'flag' is not assigned a value in all the branches of case.
→ Latch is *inferred*

Say we have an example like this. This is some specification where you have 2 parameters to this module, current state and flag. Current state is a 2 bit vector, flag is also a 2 bit vector which is declared to be reg. Because we will be using the flag on the left hand side inside always. Now you have always block out here. Just have a look at this always block. This always block checks the current state. Now if the current state is 0 or 1 you assign some value 2 to flag. If it is 3 you assign 0 to flag. But in this case statement you have a case. So this case statement will can potentially have 4 arms because the current state can hold 4 values 0, 1, 2 and 3.

But you are assigning the values only if it is 0 if it is 1 and if it is 3. But when the value is 2, this case statement does not take care of that condition. So as per the semantic goes the meaning of this means that if the value is 2 then the last value of flag should be retained it should not be true. So this implies that flag has to be stored in a flip flop okay. Flip flop or latch whatever. It has to be stored in a storage element because there are certain conditions under which you will have to remember or retain the old value of the flag. So this is what the synthesizer looks at whether inside the always block the variable you are assigning. The value 2 they get assigned a value for all possible branches.

(Refer Slide Time: 43:33)

```
// Another sequential logic example

module incomp_state_spec (curr_state, flag);
  input  [0:1] curr_state;
  output [0:1] flag;
  reg    [0:1] flag;

  always @ (curr_state)
    case (curr_state)
      0, 1  :  flag = 2;
      3     :  flag = 0;
    endcase
endmodule
```

The variable 'flag' is not assigned a value in all the branches of case.
→ Latch is *inferred*

What I meaning to say is that suppose you have an always block. You have always block you have begin, end. There are several statements. There can be several if statements case statements whatever. So in this block if you do a code analysis you consider there are a number of possible parts of execution depending on the if and else. So what the synthesizer will check that it will check that whether along all the possible parts the values are getting assigned to the variables which are appearing on the left hand side out here.

Say if f is the variable which is appearing on the left hand side. There has to be an some f equal to assignment along all this possible paths. If such a thing is there, then only you can synthesize this as a combination logic because you know that whatever conditions on the inputs are coming. You know based on that input you can have some value of, f but there is no path where the value of f is unspecified. If it is unspecified then we will have to map it into a sequential logic right. So just you look at the previous example once more. The value of 2 was not specified.

Student: Sir, flag was used to be as a larger. In earlier case, since you have specified all the values it can be a wire also

Yes, if I specified all the values it can be a wire also. But since here I have not specified all the values this flag can be a latch. But suppose as a designer well I understand that what I really mean is that if flag if the current state is 2. That means I do not want to change the flag or I want to set the flag to some value say 0, then actually how exactly what do I do that I can make a small change to the previous code.

(Refer Slide Time: 45:37)

```
// A small change made

module incomp_state_spec (curr_state, flag);
  input  [0:1] curr_state;
  output [0:1] flag;
  reg    [0:1] flag;

  always @ (curr_state)
    flag = 0;
    case (curr_state)
      0, 1 : flag = 2;
      3    : flag = 0;
    endcase
endmodule
```

'flag' defined for all values of curr_state.
→ Latch is *avoided*

This is 1 way I can handle. There are other ways also. I just add this 1 statement out here. Well here I think I have missed 1 thing. There will be a begin end. I missed the begin end statement. So we are beginning statement here. There will be an end statement here. So here what it says that it always statement this this always block consider 2 statements. One is flag equal to 0 and the other is the case. Now you see you are starting with by setting flag equal to 0. Then in the case statement you have a path where the current state is 0 where you are setting flag equal to 2. There is another path where state is 1 where again you are setting flag equal to 2.

There is another path where the state is 3 where you are setting flag equal to 0, but what about the other path. Other path also gets a value assigned to f because at the beginning irrespective of the current statement you are assigning some value to f. So the only thing we need to check is that whatever path is followed some value should be assigned to the variable along the path. Now since we have already assigned something at the beginning. So the value by default is getting assigned for all possible conditions. So this specification will be generating a combination logic.

So by this we are avoiding the latch. So here we can either do this or you can add another case with 2. Both are okay. If the flag was outside the always block then it will not be any case because the always block is something which is going in an infinite loop. So the every time the always block starts we will have to check that edge clock, edge triggered something then. Okay what I am what I have said earlier is that if in the always event expression, there is an edge of a clock then whatever is inside irrespective of that it will be a sequential logic.

But if this is an expression based on only some values there are no clocks involved. Then we will have to check inside to decide whether this can be mapped into combination or sequential. (()) (48:07) For the simulator it does not matter but for the synthesizer it matters. See this you can map it on a synthesizer as a flip flop or not as a flip flop not flip flop latch. Latch will be enabled continuously. But if you simulate, if you look at the value the value will be the same in both the cases. May be if you have a latch the delay will be slightly more. That is all.

Student: No. What I am saying like you said if it all the other rates of clock they are sequential or it is current state it is combinational…

For the simulator it does not make difference.

Student: If I name my clock as some variable which I want to like name my clock as clock.

Name anything, but if I give the posedge or negedge okay. Posedge negedge means I am triggering something at the edge of some variable that has to be en edge triggered something okay.

Student: Sir.

Yes. If statement well here I think I have not expressed the syntax in a proper way. Case statement can have a break after this flag 2 break. Yeah, you are right. There will be a break here just like C.

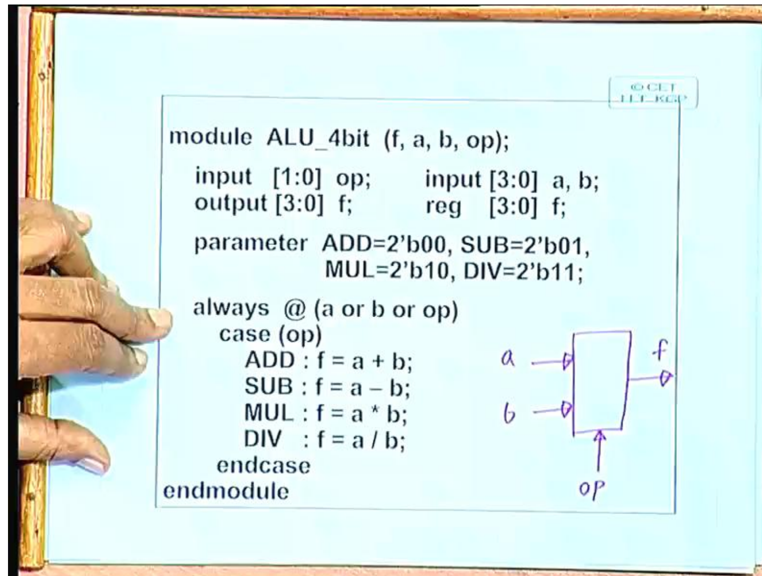
Student: ...in a way destroying the vale of latch where…

We are destroying the value of latch if it is okay with my specification but…

Student: Is it a good practice?

No, no, see if my requirement the design I am trying to do. If it is such that under the condition this state equal to 2, the value of flag I assigned is a do not care thing then I can afford to do this. But otherwise I should not do this. You are right. This is not a good practice. But if under certain condition it is really a do not care thing then we can do this. Yes, so let us just look at a couple of other examples.

(Refer Slide Time: 50:17)

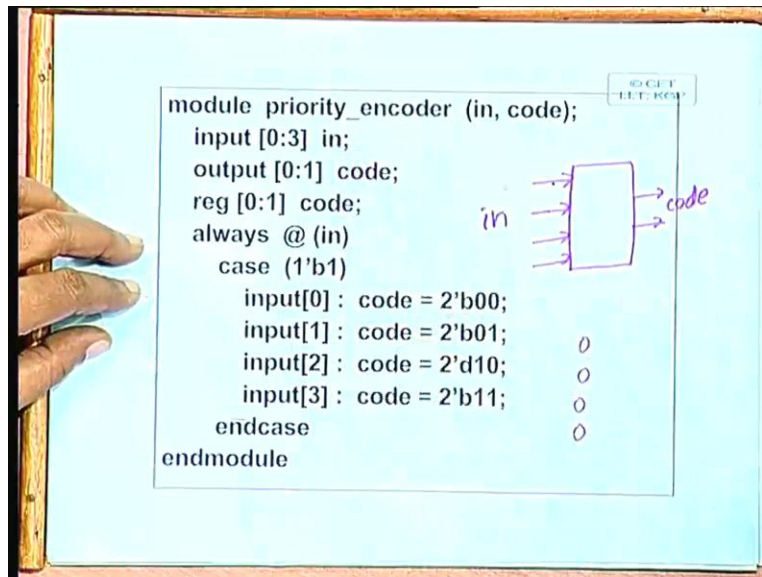


This of course does not specify anything new. Here you have 4 bit ALU specification. So the ALU is such that you have 2 input operands a and b. These are vector 4 bit vectors a and b. You have you have a function which is f. This f is also defined as a 4 bit not f sorry f is the output. F is the output and op is the function op is a 2 bit quantity. Now here well you recall I have said a parameter is a named constant. For readability of the code it is always good to define the constants with some names. Like for example I have defined the parameter 0 zero bit combination means add 0, 1 means sub, 1 0 MUL, 1 1 DIV and I can write the always block like this. Whenever any of these 3 changes, case op if op is ADD do this. If it is SUB, do this. MUL, do this.

Now you see here this is a correct specification. But this is not a structural specification. This is a specification which I am doing at the behavioral level because I am simply this adding subtracting multiplying dividing 2 vectors. But if I want, I can specify it at a slightly lower level. I can say that well the addition has to be done using say a carry loop adder or the multiplication has to be done using algorithm say. So if I want to specify those just instead of simply writing f equal to a star b I will have to write a module for multiplication and I will have to instantiate a callee here. These are combination. Yeah,

these are combination because op all possible combinations we have specified. So let us show another combination logic example.

(Refer Slide Time: 52:23)



This is an example of a priority encoder. Priority encoder here we have 4 inputs. This is in we have 2 outputs. This is your code and just you recall what a priority encoder is. The priority encoder says there are 4 inputs, normally the inputs are 0. Say the inputs are normally 0, 0, 0, 0. So which input is 1, I will be encoding that into a 2 bit binary number. Now you look at this code this always in whenever in changes state I will execute this block. Here I am giving a case statement case. Well the expression is a constant value 1. So I am comparing this with 1 and the order in which I am applying this is the order in which the check will be made. First you check if input 0 is 1. Then we check input 1 is 1, 2 is 1 or 3 is 1. So which 1 you give encounter first that code you generate. So the priority goes from this way. Right, so just using a case statement like this you can synthesize a priority encoder, Fine. So in this lecture in this lecture we had looked at. Yeah. (()) (54:00)

Okay, in this example let us again come back to the example once. This input 0, 1, 2, 3, we are comparing to 1. See case statement what it does. Here you specify an expression, expression gets evaluated and the value of that expression is being compared with these values which I have specified. So I am specifying whether I am checking whether this is equal to 1 then this is equal to 1. This equal to 1 or this equal to 1. This is the order I am checking. There is a priority. That is the semantic of the case statement that it has to be checked from top to bottom. There will be a break. There will be a break. I have not shown the break because it checks the semantic of case says that it has to be checked from top to bottom. So the synthesizer will have to take care of that. That is how it will be synthesizing it as a priority encoder and not a conventional normal encoder. Okay. So, so in this lecture we have looked at some of the modeling styles how we can model both combination and sequential logic using the always block. Now the 1 thing we have not seen as yet that what are the different kinds of assignments we can make inside an always block. This is something we will be looking at in the next class.