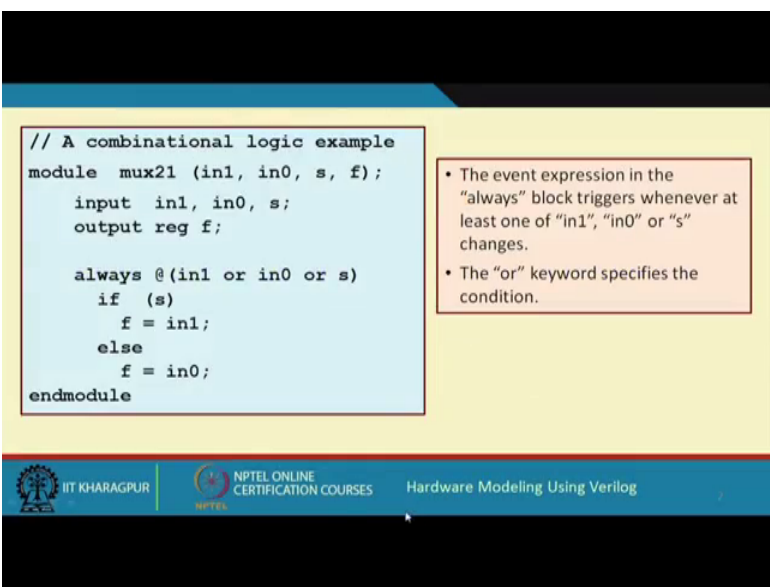
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 15
Procedural Assignment (Examples)

So, over the last couple of lectures we had seen the various kinds of assignments data flow and the procedural kinds of assignments. Now in this lecture we shall be showing you some examples of verilog module descriptions using mainly the procedural kind of assignment statements. So, by doing that we will be understanding some of the design styles and techniques which are good to use and also some of the implications. If you create the design in certain way then you are expected to get something otherwise you will be getting something else. We will be explaining a few such issues ok.

(Refer Slide Time: 00:58)



So, we start with an example of a simple 2 to 1 multiplexer. See we have seen earlier that we had created the multiplexer design using various behavioral and also structural techniques. We use the instantiation of gates to use or create a multiplexer description. We use the assign statements to define the behavior of a multiplexer. So, the I mean also use we use that you just recall that vector with a variable index on the right hand side that creates a multiplexer various ways you have seen. But here we are using a

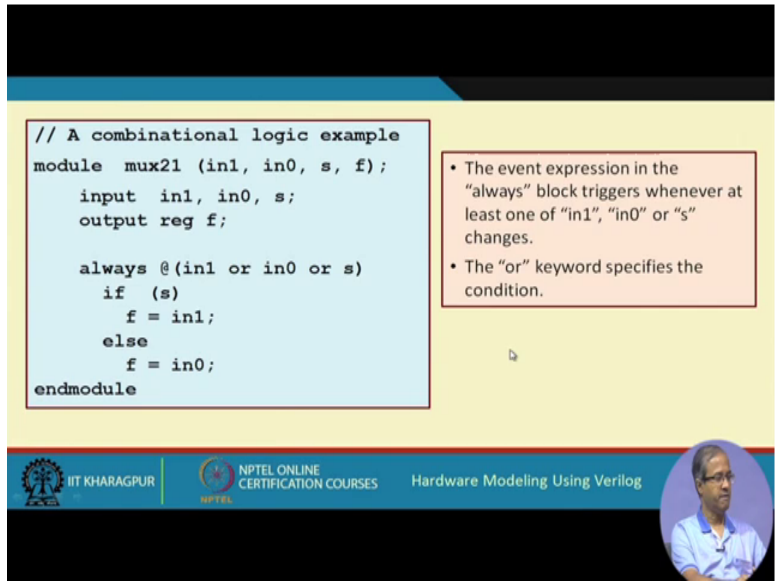
procedural kind of description that also defines the multiplexer using some kind of an if else statement, so let us see this.

So, in this example we are trying to create a simple 2 to 1 multiplexer, where the inputs are in 0 and in 1 and the output is f and the select line is s. So, in one in 0 and s are the inputs. So, f is an output which is also reg see again I am declaring f as reg because inside this always block this f is appearing on the left hand side. Here in the always block I am giving the condition the event expression is whenever any of the input changes in 1 or in 0 or s. So, whenever any of the input changes I update my value of f what I do if s is true; that means, s is 1 then this in one will go to f otherwise if s is 0 in 0 will go to f.

Now, see if whenever you describe something using this or notation like this, normally we are expressing or trying to express a combinational circuit behavior, but of course, there can be exceptions we shall be seen later, that even if we specify the event expression by naming variables using or there are cases where a sequential circuit might be synthesized, but we will try to explain when it happens, but in the example I have given here it is a simple case of a combinational circuit that will be generated. So, here a simple 2 to 1 multiplexer will be generated where this in 0 and in one will be the inputs, this s will be the select line.

Because you are checking on s and f will be the output. This or keywords combines the variables to form the event expression ok.

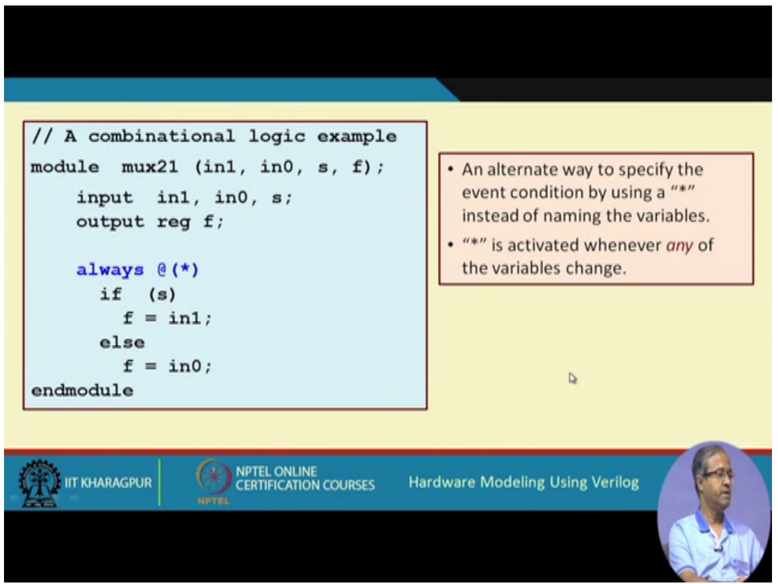
(Refer Slide Time: 03:53)



Now, I have a slightly modified version of the same program where in the earlier one I we give or, so instead of or we can give commas same thing. This is just a alternate way expression which may be a little more compact instead of writing or so many times. You can simply separate the variables by commas. Now this notation is supported in the recent versions of verilog.

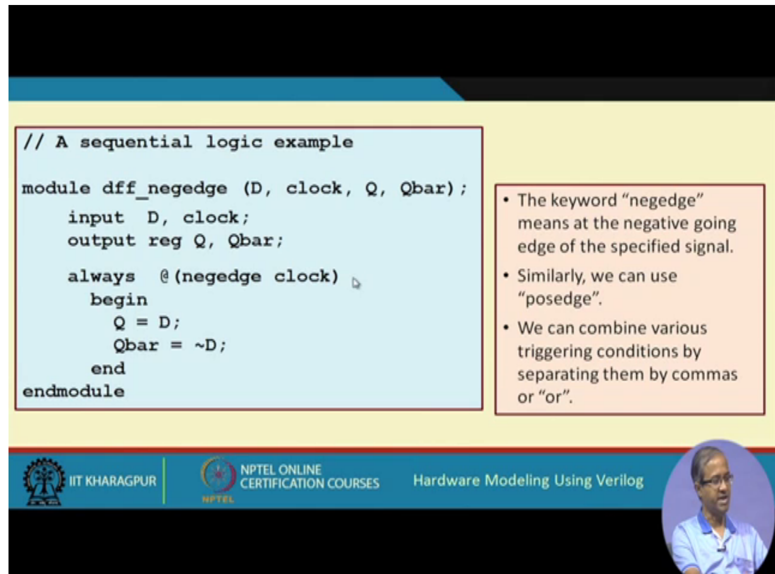
So, you can use this, fine such the third variation.

(Refer Slide Time: 04:25)



So, instead of listing the variables I mentioned we can put a star. What does the star mean? So, whenever any of the input variable changes now here my input variables are in one in 0 and s. So, whenever any of them changes this always block gets activated and the block is executed. So, the body of the block is the same. So, these 3 versions are equivalent. So, you can see this is perhaps the most compact, and in many design you may use this because it will make a code shorter, fine.

(Refer Slide Time: 05:09)

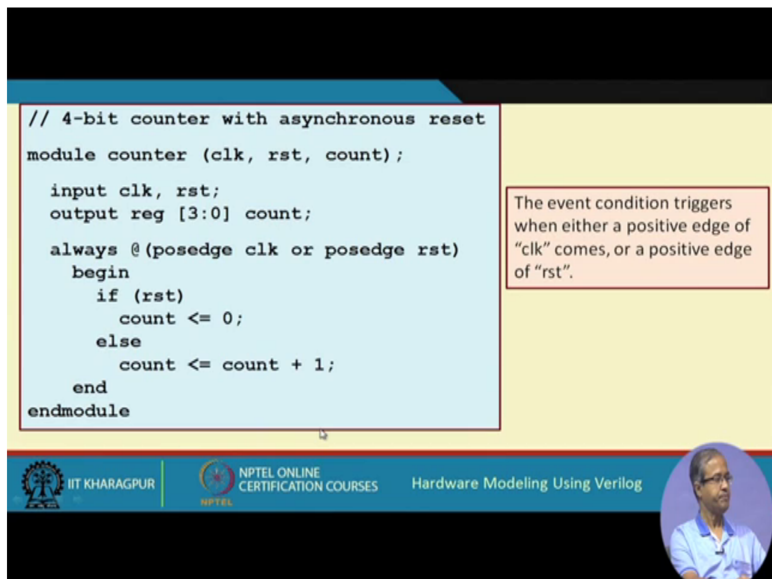


Now, let us come to some sequential logic examples. This is an explicit synchronous D type negative edge triggered flip flop where you are using a clock with a negedge. So, I mean I mean in this description we are trying to define a D flip flop with a D input outputs Q and Q bar with a clock. So, D and clock we are declaring as inputs. And both Q and Q bar we are declaring as output reg. Well of course, we could have done what we saw in the earlier examples where this Q we can declare is reg and Q bar we can generate by using an assign not of Q. But here we have chose to just assign Q and Q bar both inside the always block, this is also a correct description. So, what we are doing? Inside this always block at the rate negedge of the clock.

So, whenever there is a negative edge of the clock we are executing 2 statements, what we are doing? Whatever is on D that goes to Q and whatever is on D bar that goes to Q bar? This is the logic that is that defines a D type flip flop, which is negative edge triggered. So, just again to repeat we mentioned this in detail earlier that what is meant of meaning of negedge. Negedge means whenever this signal is having a negative going edge. Now negative going edge the meaning I told either 0 to 1 or 0 to x or z or x or z to 0. Similarly if you want to have it triggered on the positive edge you can use posedge instead of negative edge.

And this condition several conditions you can separate by commas or ok.

(Refer Slide Time: 07:11)



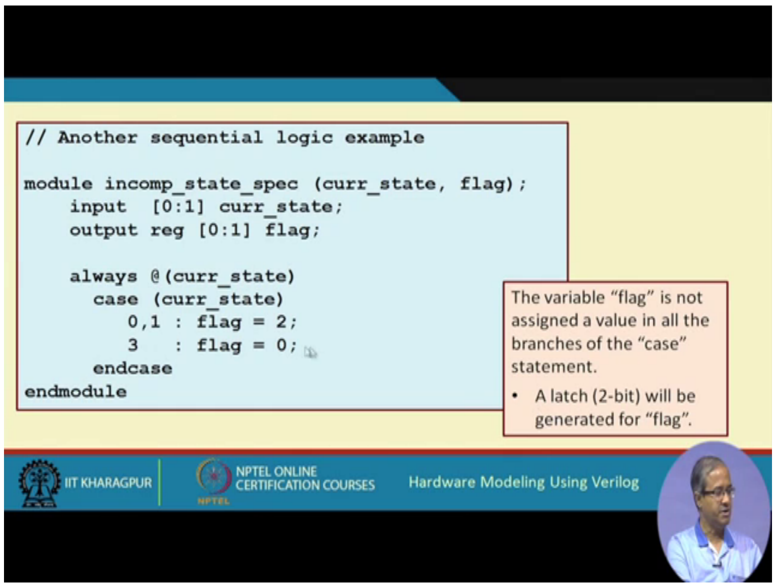
Let us take another example this is a simple 4 bit counters. So, you know what a 4 bit counter is? So, a counter is a digital circuit normally if we do not specify anything else we mean it is a binary counter; that means it counts in binary. Suppose it is a 4 bit counter, it can store a value which is 4 bits long 0 0 0 up to 1 1 1 1. That means 0 to 15. So, whenever a clock comes the counter will be counting up to 0 1 2 3 4 and whenever it reaches 15 at the next clock again it will become 0, that is our counter works.

So, here we are trying to design a counter where we can also reset it from outside by activating a signal rst. Reset means we can reset it to all 0. And it is asynchronous; that means, it need not be synchronized with the clock, whenever the reset is active immediately I can initialize the counter to 0. So, let us see how the description will be. So, here clock and reset are the inputs, and count here is the example of 4 bit counter. So, the count value is of 4 bits, which is output which is also reg again because count is appearing on the left hand side of the assignments.

So, here our event condition says either positive edge of the clock or positive edge of reset. So, if any one of them is true this event condition is considered to be true. So, you go inside and check first if reset was active. Positive edge means there are other positive edge and reset has become 1. So, you check whether reset is 1. Then you make the count value 0 else reset was not active just a clock came. So, you will have to increment the count by 1. This is this is a simple description of a counter with asynchronous reset, but

if you do not need asynchronous reset if you need a synchronous reset then you drop. The second part just only write positive clock, then everything will take place in synchronism with the clock ok.

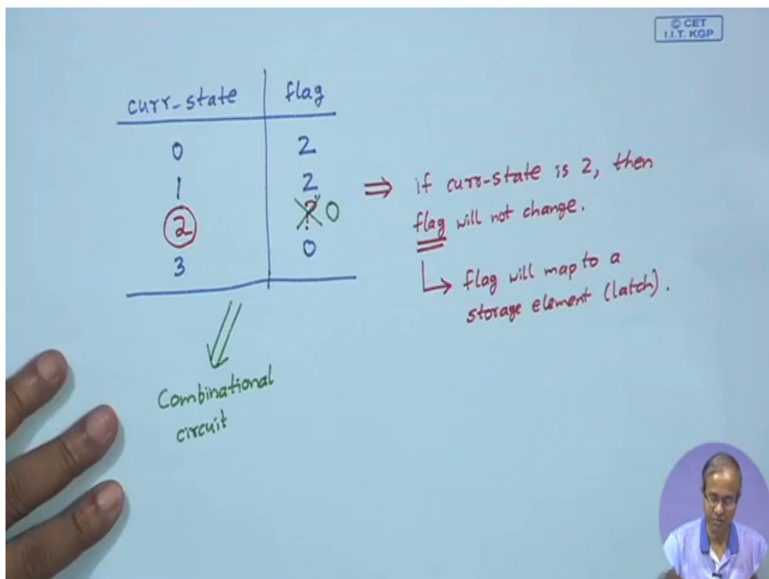
(Refer Slide Time: 09:36)



Here this is a little surprising example. See first thing is that here we have mentioned in the comment that another sequential logic example. But you look at the description does it look like a sequential circuit or sequential circuit description. Let us look at it in some detail. This is a module. This is the name of the module, just forget the name for the timing there are 2 parameters current state and flag. Current state is a 2 bit vector 0 1, flag is also a 2 bit quantity which is output also reg because flag is appearing on the left hand side. Now let us look at our procedural block. Well, always whenever the current state changes you do this. So, what is my logic? My logic is case there is a case statement.

So, I check the current state. So, if the current state is either 0 or 1. So, in case if there are multiple cases which can result in the same expression you can separated them by commas like in the example shows. If it is 0 or 1, you set flag to 2. If it is 3 you set flag to 0. Now the question is why am I calling it a sequential logic. So, it is clearly like a combinational logic.

(Refer Slide Time: 11:14)



Now, just you try to construct a truth table kind of a thing. Suppose I am listing my current state and the expected value of flag which I am setting. So, according to my description if my current state is 0 I set flag to 2, if my current state is 1, I set flag to 2 again. If my current state is 3, I set flag to 0.

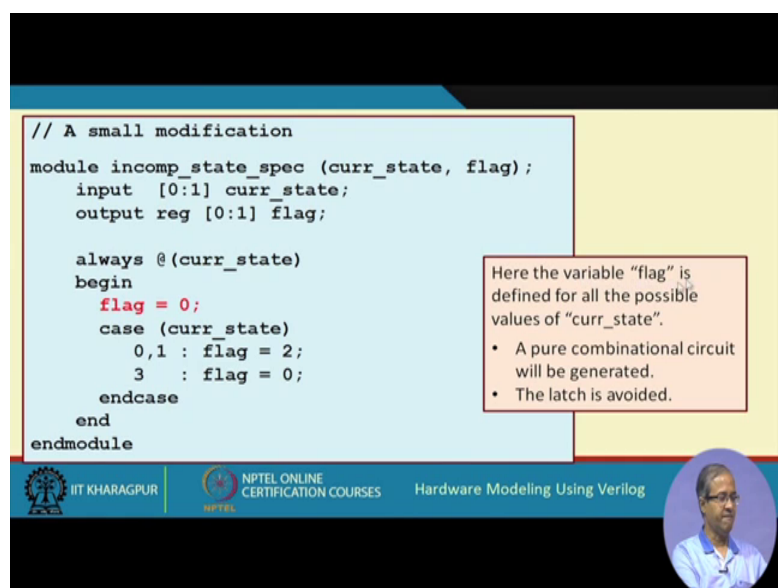
So, it is just like a combinational circuit. But there is a catch here. Here we have not specified what will happen if the current state is 2. So, we have not specified the value of the flag. So, what is the meaning? Now the verilog simulator or the synthesizer will assume the meaning as follows. If some of the input descriptions are missing in the case statement, like here the value 2 was missing. The interpretation will be if current state is 2 then flag will not change. Because we have not specified the value of flag the interpretation will be the value of flag will remain what it was in the previous state which implies me, which implies that flag will map to a storage element in latch.

So, you will have to store it just for this purpose. So, whenever there is a incomplete specification in order to satisfy the interpretation that the that the verilog simulator or the synthesizer assumes. That if it is one of those unspecified condition the output value will not change, to implement that the output value has to be stored somewhere in a latch. Which implies this becomes a sequential circuit, not a combinational circuit anymore. So, for this example the variable flag is not assigned a value in all the cases of the all the

conditions of the case statement, 0 1 3 we have specified, but for 2 we have not specified because flag is a 2 bit register.

So, a 2 bit latch will be generated for the flag, this is how it will work right. Now you see means as a designer I know that if I specify a circuit like this, by not specifying all the conditions case conditions, then the synthesizer will be generating a latch. Well although, I do not want this to be a sequential circuit. So, how can I modify it? I can make a small change in my description. Like I can just add one line here remaining, remaining part is identical just one line here before the case.

(Refer Slide Time: 14:49)

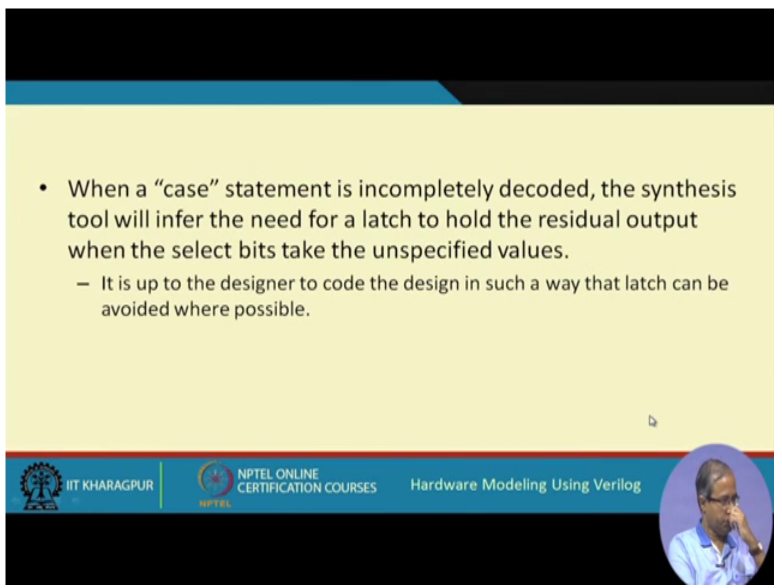


Before the case I add one line. Now there are 2 statements that is why I have given a begin end flag 0.

So now what is the difference? Now the synthesizer or the simulator if it does a dataflow analysis, it will find that well for 0 and 1 flag is 2 for 3 flag is 0, but for the condition 2 also flag will be 0 because we have already initialized flag to 0 and I have entered; so if I do not specify a flag will remain at 0. So now, if we look at this truth table we had constructed earlier, now for this case instead of a question mark here this value will also become 0. So, this will be a pure combinational circuit. And there will be no latches that will be generated.

So, as a designer it will be your responsibility to write the specifications in such a way that latch is not generated unnecessarily unless we explicitly wants it to be generated fine. So, here as I said by doing this the variable flag is getting defined for all possible values of current state, which implies that a pure combinational circuit will be generated by the synthesis tool and the latch as in the previous example will be avoided ok.

(Refer Slide Time: 16:35)



So, to summarize our I means observation whatever you saw, that when you use a case statement in a verilog description.

And the case statement is incompletely specified; that means, all conditions are not given. Then the synthesis tool will generate a latch to hold the output values because whenever one of those means unspecified values are coming. So, the synthesis tool will try to keep the output in the previous state, and for that it needs it to store it in a latch. Now as I said it will be up to the designer finally, to code it in such a way that latch can be avoided. Because if you specify your design in such a way that it is incompletely specified, the synthesis tool will assume that as a designer you are wanting to create a latch and a latch will be created.

(Refer Slide Time: 17:47)

```
module xyz (input a,b,c, output reg f)
  always @(*)
    if (a==1) f = b&c;
endmodule
```

For a=0, value of f is unspecified.

```
module xyz (input a,b,c, output reg f)
  always @(*)
    begin
      f = c;  <= when a=0
      if (a==1) f = b&c;
    end
endmodule
```

Well, let us take a very simple example to just to basically consulate this observation once more. Let us take a very simple example. Suppose we are creating a module description let us say the name of the module is x y z. Well, another thing I am specifying here see the input and output description you can specify here also the modern versions of verilog supports this. So, you can specify it like this, module x y z is means earlier you had mentioned a b c f and this input and output would mention later, but you can you can specify in the same line also. When you are declaring you can say input a b c output reg f this is also allowed.

Now, my module description goes like this. Always whenever any variable changes start. So, inside the always block there is a single statement it says if a is equal to 1, f equal to b and c, this is bitwise and, end module. Simple, this is a simple specification. So, here we are saying whenever some input is changing you check if a is 1. So, if a is one you assign b and c to f. Now you see here I have not used a case statement, but there is a conditional if else kind of thing. So, the situation is very similar. Here also we are not specifying the value to be assigned to f for all possible values of a we say that if a equal to 1 you do this, but what will happen if a equal to 0? I have not specified. So, if a equal to 0 the interpretation will be the value of f will not change; that means, you will need a latch. So, the inference is that for the condition a equal to 0, value of f is unspecified. So, what will happen for that? So, if you give this description to a synthesis tool. So, possibly your synthesis tool will be generating a circuit like this. It will be generating an

end gate, whose inputs will be b and c, then it will be generating a latch where this will go to the D input. And the input a will go to the enable input, and the output Q will be your f. You see this exactly models this behavior what you have given, if a equal to 1.

So, whenever a equal to 1 this enable will be active and b c, b c will be stored in the latch. And that will be available on f, but if a equal to 0 which means you are not enabling the latch. So, your previous value stored in the latch will remain in f, this is what you wanted. Now let us look at a slightly modified version of the description. So, we are using the same example same parameters, output reg f. Now here there were 2 statements that is why I am giving a sorry, always at the rate star now here there was one statement no begin end was required, but here I am giving a begin end.

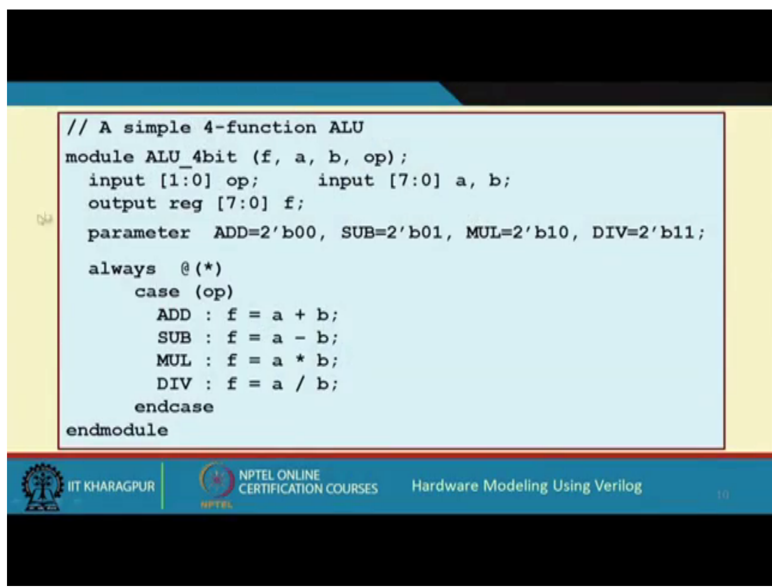
So, what I do? I add a statement let us say f equal to say suppose my descriptions like this f equal to c, and then something like this. If a equal to equal to 1 f equal to b and c, end and end module. So, if my description is like this then what is the interpretation? Then you see my description says if a is one then f will get the value of b and c. And for a equal to 0 I have already initialized f to some value. So, this will be the value when f is 0. So, I have defined now f for both conditions sorry, for a equal to 0, for a equal to 0 this will be condition, for a equal to 1 this will be the condition. So now, I have specified the value of the output f for all values of a.

So now if you give this description to the synthesis tool, the synthesis tool will be possibly generating a circuit like this, to generate b c there will be an end gate. Then it will be synthesizing a multiplexer, a 2 to 1 multiplexer. Where the other input of the multiplexer will be c, this will be your input 0 input 1. And your select line will be a, the output will be f. So, see what is happening here? You are saying when a equal to 0, c should go to f. So, if a equal to 0 c will be selected. And if a is one this will be selected b c will be selected, this will go to f. So, this is a purely combinational circuit you see. So, this is something which is up to the designer.

So, you will have to design it in a proper way. So, that the synthesis tool should not get confused and generate a latch, where I mean what you actually wanted is a combinational circuit right. So, it is greatly up to the designer to specify how you are wanting to guide the synthesis tool. So, you should not mislead the synthesis tool in

believing that what you are specifying is actually a sequential circuit when it was not fine. So, let us take some more examples.

(Refer Slide Time: 24:50)

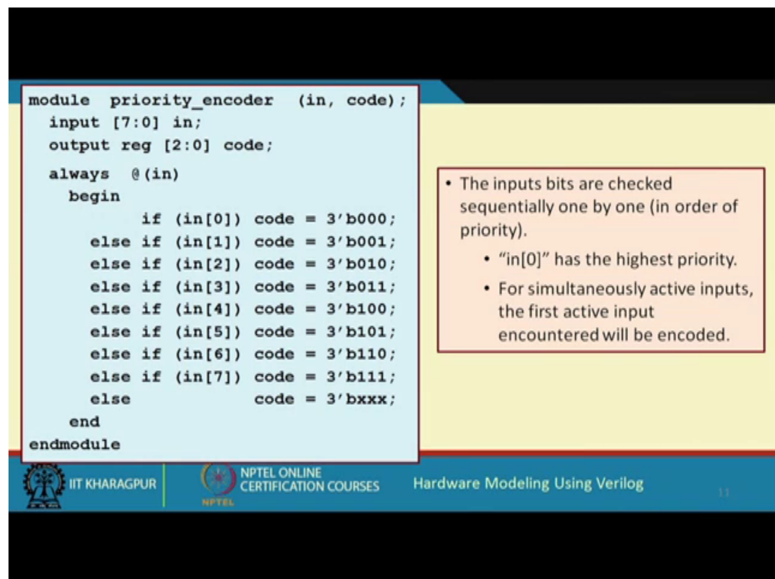


So, here we have a very simple 4 bit arithmetic logic unit, where the parameters are f a b and op. But op specifies these are 2 bit it specifies one of 4 operations, and a b are 8 bit inputs. This f is also 8 bit output which is of type reg, because f is on the left hand side. Well, using parameter command you can specify some constants I give some example earlier, like here suppose we are supporting 4 operations addition subtraction multiplication and division.

Now to make our program more readable we have specified them as add sub mul and div. Here add means this op is 0, 0 sub means 0, 1 mul is 1 0 div is 1 1. And here we have an always block where you are saying whenever it is star; that means, any of the input changes depending on the value of op, but instead of writing 0 0 0 1.

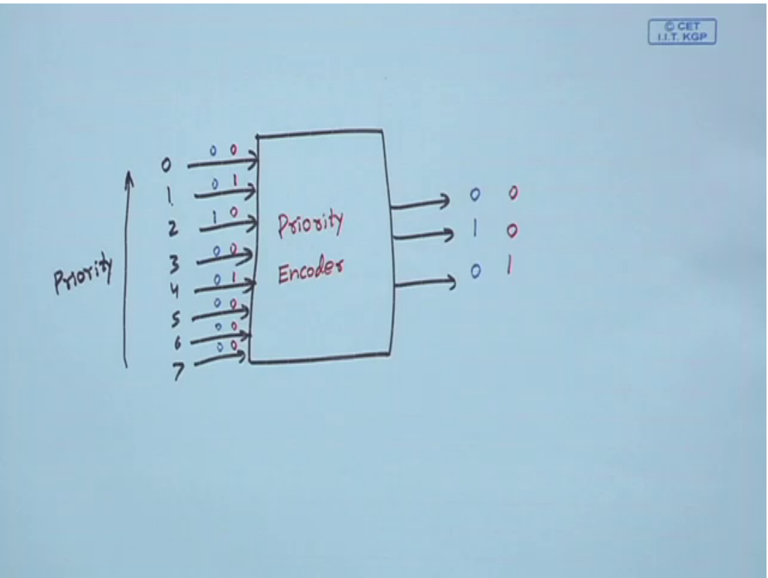
And 1 0 1 1 I straight way I am writing add sub mul div. So, it becomes much easier to understand the code. So, the operation is given directly. So, this is a very simple way to specify an alu in a behavioral fashion right.

(Refer Slide Time: 26:19)



Fine, this is another example of a priority encoder. This priority encoder has 8 inputs and it has 3 3 outputs.

(Refer Slide Time: 26:34)



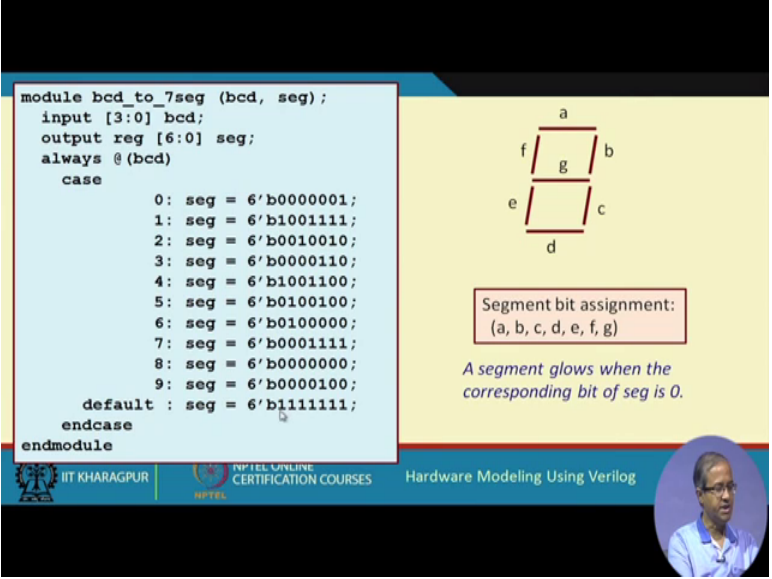
So, what is a priority encoder? So, a priority encoder the example that have set there are 8 inputs, and there are 3 outputs. And inputs are given some index value 0 1 2 3 4 5 6 7. Now I am assuming that the priority increases in this fashion, my input 0 has a higher priority than input 7.

Now, the idea is that suppose I set the input number 2 to 1, and the others are all 0s. Then the binary equivalent of 2 which is 0 1 0 should appear in the output. Now let us take an example where any 2 of them are one let us say this input one is also 1, input 4 is also 1, the others are 0s. So, if more than one of the inputs are at then according to the priority you choose the one which has highest priority that is 1. So, binary equivalent of 1 0 0 1 this should be in the output. So, this is how a priority encoder works. So, let us see the description. So, the input is an 8 bit vector, this is the output. Code is a 3 bit vector of reg because it appears on the left hand side.

So, again in the always block whenever there is a change in there is a begin with a nested if. If in 0 means in 0 is true, which means in 0 is equal to 1. If in 0 code equal to 0 0 0, else if in one code is 0 0 1 simple. If in 7 1 1 1 and if none of them are active, else I am initializing the code then to x x x. You see the order in which I have checked the bits that will define the priority. So, if both your in 1 and in 4 are active, but I will get a match in one first. So, code will become 0 0 1. This is how priority is determined, the order in which I check the bits right.

So, as I said the input bits are checked sequentially. So, in 0 will be having the higher highest priority. And if more than one inputs are active the first input that is checked will be encoded ok.

(Refer Slide Time: 29:26)

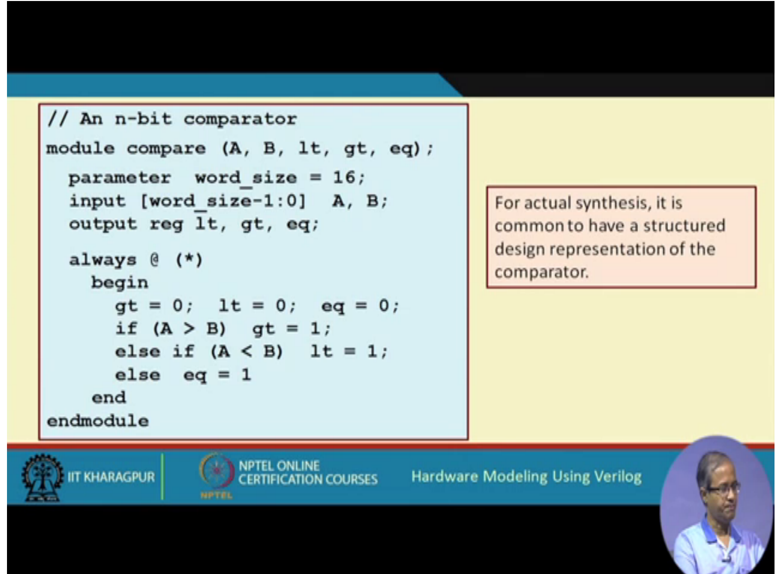


Let us take another example which is also quite common bcd to 7 segment decoder. Now bcd you know bcd is a way of encoding decimal numbers, decimal digits are 0 to 9. They can be coded in 4 bits 0 0 0 0 up to 1 0 0 1 that is 9. Now 7 segment display looks like this, where there means either LED's or LCD's whatever, there are 7 segments a b c d e f g in that order.

And depending on what you want to display you can glow some of the segments and you can switch off some of the segments; so in this example. So, we are representing the 7 bit as a 7 bit vector in this order. So, a is the most significant bit, and g is the least significant bit. And another assumption we are making that a segment glows whenever the corresponding bit is 0. And when the corresponding bit is 1 this segment will be off. So, the description is very simple. So, input to this circuit will be a 4 bit bcd number a digit 0 2 3 4 bit. And output will be this 7 bit vector, 0 to 6 you call it seg. So, always whenever this bcd changes I have a case statement. If it is 0 0 means I have to glow all except g. So, g should be one all other should be 0 see g is one all others are 0.

Similarly for one b and c will glow. So, you see only b and c are 0, for 2 2 will be like this. So, c and f will be switched off, c and f are switched off like this you can check. So, all these digits are glowing. And the default is you are switching off all these segments if it is input is something other than 0 to 9, because in 4 bits you can specify any number from 0 to 15. So, if either 10 11 12 13 14 15 comes you switch off all the segments right.

(Refer Slide Time: 31:52)



So, there is another simple example of an n bit comparator. This is a pure behavioral description where A and B are the numbers, and the outputs are less than greater than or equal to. Here again I am using a parameter, where I am defining variable word size to be 16. So, A and B I am defining to be 0 up to word size minus 1. So, for 16 it will be 15, 2, 0. And lt gt eq are actually this will be output reg let me just correct. It this will be output reg fine. So, here again in the always I put a star if any of the input changes.

So, I start by initializing gt lt and eq to 0s. It is very simple. I am making a comparison like this A and B are numbers say for 16 these are 16 bit numbers. I am making a comparison if A is greater than B, I said gt to 1. Else if A is less than B I said less than lt to 1. Else they are equal eq equal to 1, So it is simple. Now in actual synthesis actually we normally do not design a comparator in this way we shall see later. We typically use a structured design using by instantiating some smaller comparator modules.

(Refer Slide Time: 33:27)

```
// A 2-bit comparator
module compare (A1, A0, B1, B0, lt, gt, eq);
  input  A1, A0, B1, B0;
  output reg  lt, gt, eq;

  always @ (A1, A0, B1, B0)
    begin
      lt = ({A1,A0} < {B1,B0});
      gt = ({A1,A0} > {B1,B0});
      eq = ({A1,A0} == {B1,B0});
    end
endmodule
```

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

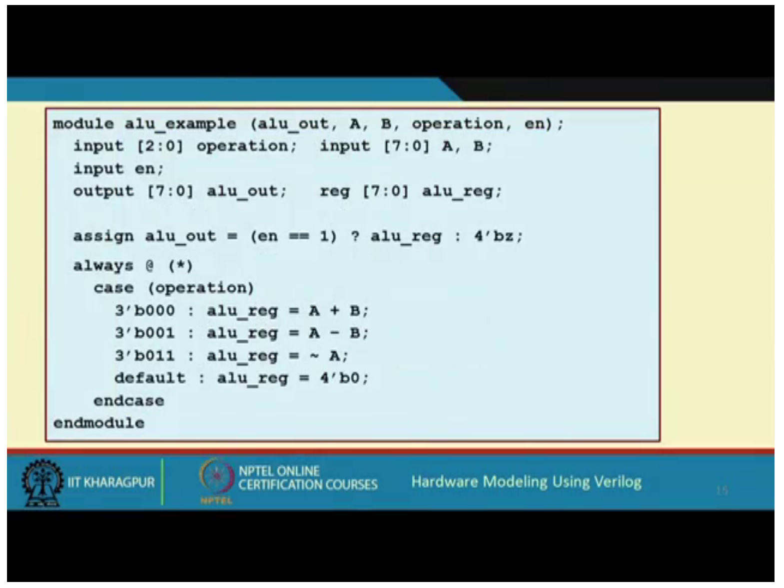
Now, another means example I am showing here just to show another style of specifying this is a simpler 2 bit comparator.

So, there 2 numbers one is A0 A1 other is B0 B1, and the outputs as usual are less than greater than equal to. So, A0 A1 B1 B0 these are all inputs and these 3 are the outputs. Now we check always whenever one of the input changes. So, I am writing like this some assignments, this lt equal to you see A1 and A0 this 2 bits make the first number. So, I use the concatenation operation to define the first number. So, there will be a curly

bracket here let me correct this, there will be a curly bracket yes; so A1 A0 concatenation and B1 B0 concatenation if A1 A0 is less than this.

So, can I think I have missed some brackets let me just good, I have just changed the brackets. If A1 A0 is less than B1 B0 if this condition is true which means true means 1. So, that one will be assigned to lt. If this is greater than B1 B0 if this condition is true means one that one will be assigned to gt and this equality again if it is true that one will be assigned to eq. See here were exploiting the fact that for logical comparisons relational operators true is represented by 1 and false is represented by 0. So, we are directly storing the result of the comparison in the target, variable indicating less than greater than or equal to fine.

(Refer Slide Time: 35:23)



So, as the last example we take a slightly more complex alu example, you see here we are concerned an alu with 2 inputs a b, which are 7 bits. There is an operation which is 3 bits and enable which is one bit. And in the alu out there is the output of the alu 8 bits and the output will be stored in another register we call it alu reg. Now this output will be generated in alu reg and it will be stored in here. So, you see this alu out we are just I said doing a continuous assignment, if enable equal to 1 then alu reg will be transferred to alu out otherwise it will be tri state z 4 z. And here description is very simple whenever I mean any of the input changes on case operation let us say 0 0 0 means add, then alu reg equal to A plus B.

Suppose 0 0 1 means subtract A minus B. Suppose 0 1 1 means logical naught alu reg equal to bit by naught, and if it is any one of the others. So, in case I can mention default, then I initialize alu reg to 0. So, this example illustrates the combination of assign and case in always block and so on, this also generates a new combinational circuit. So, with this we come to the end of this lecture. In this lecture we have looked at a number of different examples various ways of using the procedural style of coding. The always block the different ways of using always block and so on.

And we shall be continuing these discussions later and look at more complex and more elaborate structures using which we can model particularly the sequential circuits and finite state machines. Till then we will have to wait for the next lectures.

Thank you.