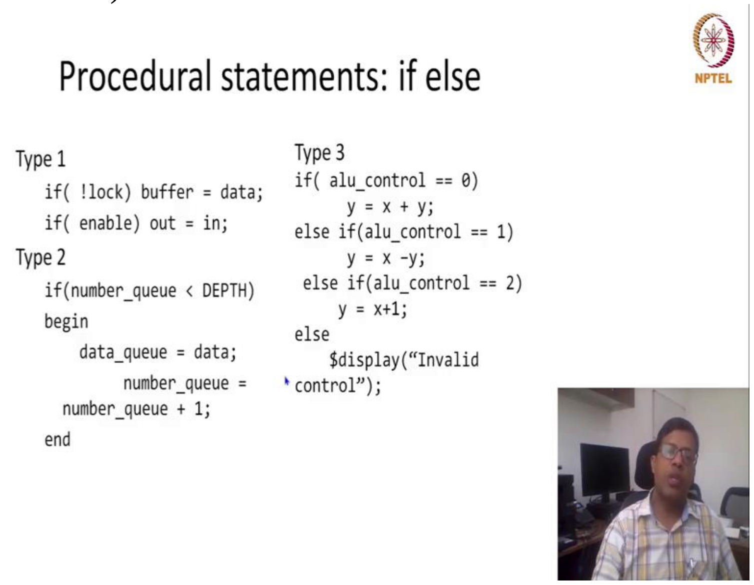
# Digital System Design
**Professor Neeraj Goel**
**Department of Computer Science Engineering**
**Indian Institute of Technology, Ropar**
**Lecture 46**
**Verilog-Behavior Model-2**

(Refer Slide Time: 00:15)



## Procedural statements: if else

```
Type 1
   if( !lock) buffer = data;
   if( enable) out = in;
Type 2
   if(number_queue < DEPTH)
   begin
       data_queue = data;
            number_queue =
     number_queue + 1;
   end
```

```
Type 3
if( alu_control == 0)
      y = x + y;
else if(alu_control == 1)
      y = x -y;
 else if(alu_control == 2)
     y = x+1;
else
    $display("Invalid
control");
```

So procedural statements are very similar to the statements we have in high level language like C, C plus plus. So, for example, we have this if and else statements. In case of if and else, we have, for example, we can write if and after that we have can a logical expression. So, based on the result of the logical expression, we can assign the output or we can have any kind of a statement here. So, this statement could be a logic expression or it could be a, it could be any behavioral expression.

But the important point here is that in the parenthesis there would be a logical expression. So, logical expression means it would be either conditional statement, less than, equal to or more than or it would be a, it would be logical an, logical or, those kinds of statements.

So, for example, this statement means if enable, if enable is 1, then out would be assigned as in. So, because, we are using if statement and because we are using this equal to sign, this will become a blocking statement. So, this is a blocking statement and after that if we are using next case, then this will also be a blocking statement. So, in this type of, even though there is no else, then also it will work.

Now, the other possible example is let us say, we have more than one statement which we would like to write in this if block. So, in that case, for example here, we are saying a number queue is less than DEPTH. So, this is a logic expression. if this expression is true, then for binding multiple statements or multiple expressions, multiple statements, then we have to use begin and else, being and end.

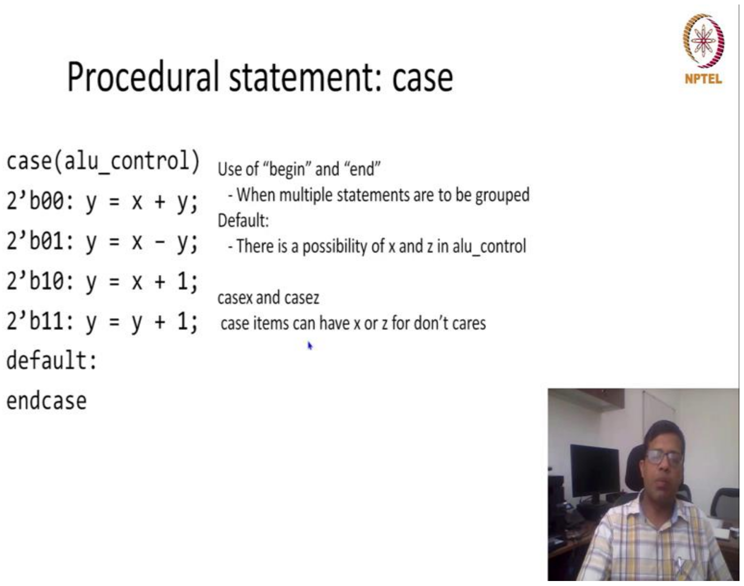
So, in C plus plus, we use curly braces. Start curly brace and end curly brace. Here, we are using being and end. Nothing else is different. So otherwise, the syntax is very similar to the C syntax. The other thing which you can also see, for example here, we are seeing number queue is equal to number queue plus 1. So, that means, we do not have that incremental operation, operant here. That urinary operator is not there so that is why we have to write exquisitely that number queue is equal to number queue plus 1.

Now, let us say if we would like to have if as well as else, so either it could be nested or it could be a non nested one. So, in case it is nested, then we can write if, then the expression, then statement corresponding to that, then else, after that we again have to write if. You can see, this particular syntax is also quite similar to the syntax of C or C plus plus. The only difference is that if we have multiple statement in this if block, then we will use begin and end otherwise there is not much other difference. In the end of this logical if else ladder, so we can end the ladder by else.

So, another comment here that this if else ladder or if else case, you can also see that it is also very similar to priority, if you are designing something like priority encoder because this if is going to have highest priority. if this condition is not true, then only you are going to the else condition and then you are checking for the next logic comparison.

So, using this if else, if, if, else, if statements then we can also create something like, if we have to design something of a priority conditions, then that can be designed using if else statements. So, this is about if else conditions and sometimes we would also like to have multiple wave branches.

(Refer Slide Time: 04:42)



## Procedural statement: case

```
case(alu_control)
2'b00: y = x + y;
2'b01: y = x - y;
2'b10: y = x + 1;
2'b11: y = y + 1;
default:
endcase
```

Use of "begin" and "end"
- When multiple statements are to be grouped

Default:
- There is a possibility of x and z in alu_control

casex and casez
case items can have x or z for don't cares

So, in multiple wave branches, we can use case statements. Case statements are also similar to our C statements. So, we have to write case and then the variable which it is controlling. So, whatever is the variable name we are writing, then there would be case items. So, these case items would compare the value of control or value of this variable and compare with all the values and then we will keep on, then we will write the statements corresponding to that particular comparison.

So, for example, here, we are seeing case alu control. Alu control is a variable which is deciding our switch case. Now, if the value of Alu control, which is a two-bit signal here, so if the value is 0 0, then we are seeing y equal to x plus y. If the value of this control signal is 0 1, then y equal to x minus y. And then 1 0, y equal to x plus 1. If it is 1 1, then we are seeing y equal y plus 1.

So, these statements could be anything and again, if we would like to have multiple statements then we have to use begin and end. So now, you can also ask this thing that if it is a two-bit control, if alu control is only a two-bit variable, then if we have covered all the cases, 0 0, 0 1, 1 0, 1 1, then should we write a default here or not? So, default means when any of these cases is not matched, then what should be the, where should we go or what should we do?

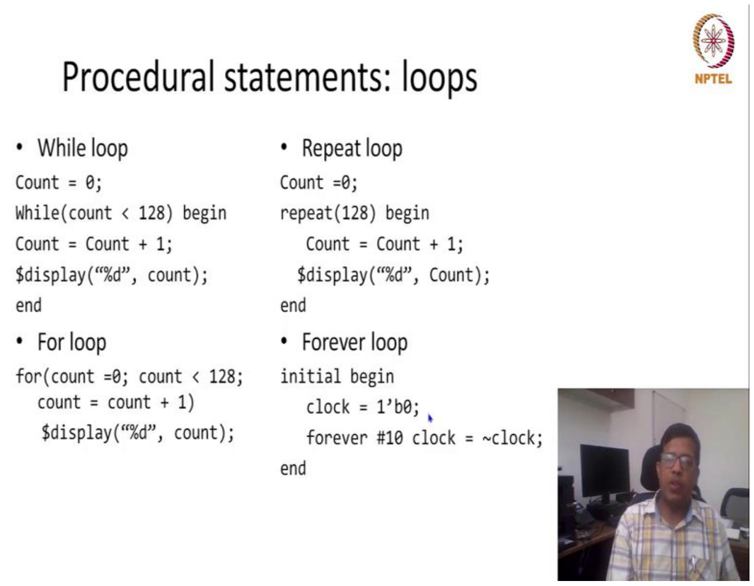
So, in that case, it looks like here there is no default statement required but default statement is required or essential because of the type of variables we have. So, you see, in case of Verilog, any particular bit or any particular wire can take four values, 0, 1, x and z. So, if the value of alu control is let us say x x or basically unknown, then what should be the output? It should depend on the default case. And then all the case items are done, then we have to write endcase, so that we can switch or basically close this case block.

So, the case block can be utilized to design multiplexes. So, for example, this is a kind of a multiplex where we are deciding, based on the control, we are deciding what would be the output. So, this we have already discussed that in the multiple statement, we have to use begin and end for that particular item and default also, we have understood, we have seen that because there is a possibility of x and z.

Now, one more thing here, that, so there is some more advanced case statements. It is called case x and case z. So, in case of case x, what sometimes, for example, we would like to have a do not care condition. So, let us consider this case itself. So, we would like to have a three-bit control. In the third bit, we are seeing that either it is 0 0 0, or 0 0 1, then we have to perform this operation. Then instead of writing both the case items, we can write 0 0 x and then y equal to x plus y.

So, which means that, and now, the third bit is taken as a do not care. So, in case of case z, wherever we would like to take do not care, we can write z there and wherever we like to have a do not care, then we can write x in the case of case x. So, this case x and case z could also be used if we have a larger case, cases and some of the case items can be combined because some particular bits are do not care there. So, this case is also a good mechanism to essentially a multiplexes or de multiplexes.

(Refer Slide Time: 09:07)



## Procedural statements: loops

- While loop

```
Count = 0;
While(count < 128) begin
Count = Count + 1;
$display("%d", count);
end
```

- For loop

```
for(count =0; count < 128;
   count = count + 1)
    $display("%d", count);
```

- Repeat loop

```
Count =0;
repeat(128) begin
   Count = Count + 1;
  $display("%d", Count);
end
```

- Forever loop

```
initial begin
   clock = 1'b0;
   forever #10 clock = ~clock;
end
```

Now, the other set of statements in procedural, procedures are Loops. So, there could be multiple types of Loops. While Loop. In case of While loop, we write a condition, we initialize and after that we say that this loop will keep on running until this condition is true. So, for example here we are saying While count is less than 128, then, we will keep on running this procedure.

If count is 0, so that means we have to initialize it to 0 and after that we can have this While and then expression, logical expression. Until this logical expression is 1, you will keep on rotating in this loop. Again, if there is only single statement, begin and end, this begin and this end is not required but if the number of statements is more than one, then we have to use begin and end. So, here we are saying count equal to count plus 1 and then we are displaying.

So, similar to While loop, there can also be a For loop. The For-loop syntax is exactly the same as the syntax in C. So, you can write For and then initialization part and after that, your logical expression and then you can have the increment part also and decrement part also. The only thing is that the urinary operator is not there in values. You have to write count equal to count plus 1, or something similar. Because there are only single statement, I am not using begin and end, I can write the singe statement as the way it is.

So similar to While loop and For loop, there could be a Repeat loop here. So, in Repeat loop, we know the constant number of times when we are going to repeat a particular loop. So, we can, now in, the same thing, the same example we are writing in terms of repeat loop. So here, we are specifying like a constant that how many times we would like to repeat this loop. We can say Repeat and that number, how many times we would like to have this particular block repeated. Then we can have begin and end and then the list of statements which we would like to get repeated.

Similar to Repeat loop, there is another one which is called Forever loop. So, in Forever loop, we can, there is, this value of repeat is essentially infinite. It will keep on running. For generating signals like clock, this could be used. For example, we can say, it could be written in initial clock. We are initializing some value and then we can say Forever. Every time, or this will keep on executing that after 10 units of time clock is equal to not clock.

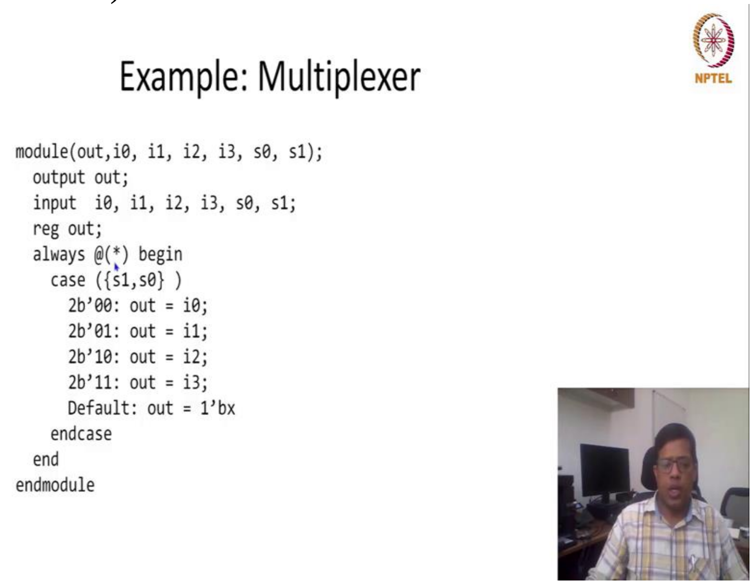
Now, again, you will say that does this Forever mean that this will keep on executing? Actually, yes. But what will stop? So, let us say, if you are writing anywhere in your test bench or anywhere in your design that what is the condition you would like to do dollar or finish. So, if you specify that after so many numbers of cycles, I would like to do dollar finish, then the execution or simulation can finish.

The other thing is, you can also have some sort of a logical expression and say that when this particular logical expression will happen, then you can assign a dollar finish. That means your condition, for example, you can have some application, let us say you are doing sorting. So, whenever this sort will finish, then you can generate a signal and you say that when this signal is 1, then I will do a dollar finish or I will finish my simulation.

So here, for simplicity, because we do not know how many times or what should be the number of cycles that it will take, so we writing, for simplicity Forever. But in some other block, we are writing a dollar finish according to when the simulation should finish. So, these were the major or important procedural statements.

There is also a possibility to create functions in Verilog but for sake of simplicity, I am not covering that and the kind of assignments or kind of Verilog programs we are going to write in this course, they would not be required. These statements would be more than sufficient to design most of the logic. So, if you want to explore more, you can go into tutorials and read books and see that what could be the other statements which you can use. But more or less, whatever statements we have covered, they should be sufficient in most of the scenarios.

(Refer Slide Time: 14:20)



## Example: Multiplexer

```
module(out,i0, i1, i2, i3, s0, s1);
  output out;
  input  i0, i1, i2, i3, s0, s1;
  reg out;
  always @(*) begin
    case ({s1,s0} )
      2b’00: out = i0;
      2b’01: out = i1;
      2b’10: out = i2;
      2b’11: out = i3;
      Default: out = 1’bx
    endcase
  end
endmodule
```

So now, let us take couple of examples at how we can use behavioral modeling. So, we have already seen multiplexer to certain extent but let us do it again. So now, I would like to model a multiplexer using behavior model. In case, I would wanted to model of multiplexer in structural model, then I will start will And and Or and then I will list all the logic where it has And Or. Or in case of data flow modeling also, I will find out all the conditions, then my out would be 1.

But here, we can write it using a case statement. My modules definition would be similar to any data flow model or a structural model. I will specify that what is my output, what are my inputs, then I have to specify my ports that output port and i0, i1, i2, i3, s0 and s1 are the select inputs, so overall, these are the input.

Now, because we are planning to use behavior model and we know we are going to write on to out, so we have to specifically specify that this out is actually a register or reg type. Although we have defined it as output but in addition to that, we have also specified that this out is of type reg. So, if we do not specify then it would be assumed as a wire which it is not because we are going to write it in a procedural statement. So, we have defined out as a reg type and then we have to define an always block.

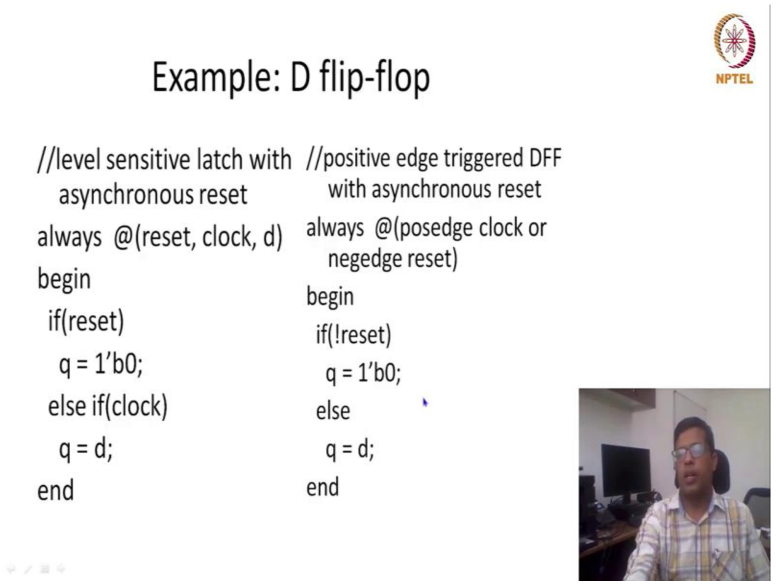
And in the always block, we can write a case statement. So, in the case statement, we are catenating s0 and s1. So, we are using this parenthesis operator which is a catenation operator. Now, s0 and s1, both, together is 1 variable against which we are checking all the values. It is a two-bit variable, it become a two-bit variable.

So, if that two-bit value is 0 0, then we are saying out equal to i0, if two-bit value is 0 1, we are out equal to i1, similarly, when two-bit value is 1 0, then we are saying out equal to i2 and when two-bit value is 1 1, we are saying out equal to i3. And when it is none of them, when we have to say that because we are not, so that means, what does it mean? It means either of s1 or s0 is x or z because my control input to my, my select input to my multiplexer is not known so that is why it is good to say that out is equal to unknown. So, out is also unknown if we, our select is unknown.

Now, the question, when should it get triggered? We would like to get this particular block triggered, whenever there is a change in s0, whenever there is a change in s1 or whenever there is a change in i0, i1, i2 and i3. So, for example, s0 and s1 is not changing. But i0 gets changed. So, there is a possibility that s1 and s0 value are 0 0 and then whatever is the value of i0, that should be given to output.

So, that is why my sensitivity list or the list of variables when it will get triggered should be all, s1, s0, i0, i1, i2, i3. So, these are all right-hand side variables and in short, we can also write at the rate, parenthesis and asterisk. So, this asterisk or wildcard means that all the variables which we are going to read would be in this sensitivity list. So, when any of these variables will get changed, then we are again triggering this particular always block. So, this is how we can model multiplexer.

(Refer Slide Time: 18:46)



## Example: D flip-flop

```
//level sensitive latch with
   asynchronous reset
always  @(reset, clock, d)
begin
 if(reset)
   q = 1'b0;
 else if(clock)
   q = d;
end
```

```
//positive edge triggered DFF
   with asynchronous reset
always  @(posedge clock or
   negedge reset)
begin
 if(!reset)
   q = 1'b0;
 else
   q = d;
end
```

The next example could be the D flip-flop. So it's an example of a sequential logic. So, before going to D flip-flop, let us start with the D Latch. Latch means it is a level sensitive and in addition we are also saying it is asynchronous. So, my gated value here is clock and reset is my asynchronous. So, what I can do is, I can write… always at the rate reset clock or d. So, whenever there is a change in reset, whenever there is a change in block and whenever there is a change in d, this block should get triggered.

So, what does asynchronous reset means? That whenever, so this has the highest priority, so whenever a reset is 1, we have to say that q, my output is going to be 0. So, we also have to remember that because we are writing q, this q should be of r e g type, reg type. So, whenever reset is 1, then output q would be equal to 0.

So, remember that, if there is no change in the reset, then only we will go to the next block, means else of this statement but if there is a change in reset and reset is equal to 1, when we are not even going to the next, any of the next statement. So, this is how priority will get decided and that is how we have to write and that is how reset has got higher priority than d or clock. So, if reset is 0, so that means, then now, we have to see whether clock is 1. Clock is a gated signal here, if clock is 1, then q equal to d. if clock is 0, or my gated signal is 0, that means my q is going to remain as a precious value.

So, in summary, we are able to write a behavior of my D latch, we can summarize our D latch behavior that if reset is 1, then we are saying my output is 0. In case reset is 0, then we are checking what clock, if clock is 1, then q is equal to d, if clock is 0, then q is actually the previous one which we are not writing here but it is implicit. So now, let us say we would like to have an edge triggered flip-flop. So, the major would be there in the edge triggered flip-flop is in the list of the events we have to write positive edge or negative edge of the clock.

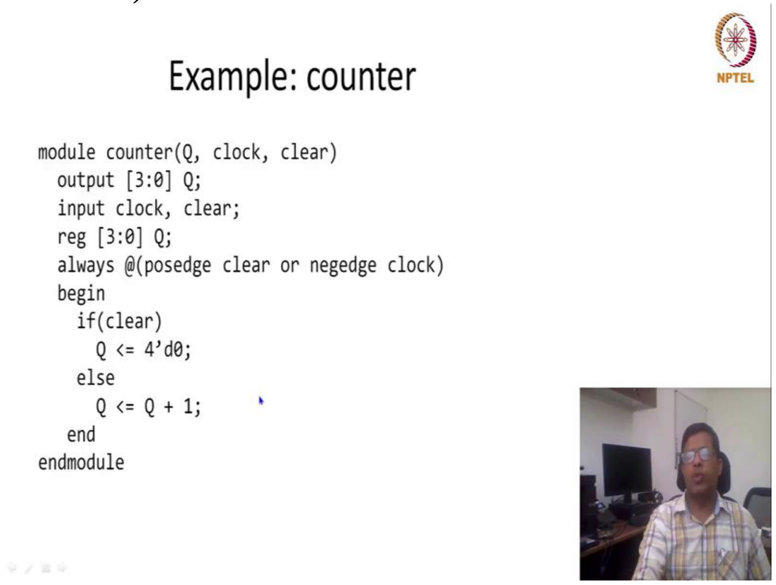
So, let us say we are writing positive edge of the clock. Because we have asynchronous reset, so let us say the reset is level low, is active low. So, if it is active low, then we have to write negative edge of the reset. Why we are writing negative edge of the reset? Because whenever negative edge is there, then only the possibility of reset being 0 would be there.

So, if there is a negative edge of the reset or there is a positive edge of the clock then this Always block will get triggered. So again, because reset has, is asynchronous and that means it has higher priority so if reset is 0, then we are saying q equal to 0 or output equal to 0, else we are saying q equal to d.

So, you see, we are not writing if clock because positive edge of the clock will make sure that my clock is at this positive edge transition and it would be 1, when this d value would be taken before the positive edge and it would be assigned to q and q will remain there for one clock period.

So, this is how we can design a D flip-flop or model a d flip-flop and because we are writing this always block at the rate posedge clock and negative reset et cetera, so synthesis tool has, is able to, will be able to identify this kind of a syntax and it will be able to act accordingly, whenever this kind of a syntax is there so, sort of pattern matching would happen and this would be synthesized as a flip-flop in case of a hardware synthesis also.

(Refer Slide Time: 23:44)



## Example: counter

```
module counter(Q, clock, clear)
  output [3:0] Q;
  input clock, clear;
  reg [3:0] Q;
  always @(posedge clear or negedge clock)
  begin
    if(clear)
      Q <= 4'd0;
    else
      Q <= Q + 1;
   end
endmodule
```

So, let us say, we would like to design a, design a counter. So, this is another example. So now, again I am writing complete modules so, module, the counter module will have three inputs, two inputs and one output. Output is q, let us say it is a four-bit counter so, and there would be another input clock and another input is clear.

So, the functionality, let us say, is that whenever clear is there, then counter at start at 0 and otherwise at every clock, it will keep on incrementing. What would happen if four-bit value of q is 1 1 1 1, then the next value will become 0 0 0 0. So that is the overall behavior of counter.

So first, we have to define the module and then we have to say the output is equal to, the output is q and we are defining input also. And because, here, during the definition of the module, we are not specifying how many bits but then we are specifying output port or input port, then we explicitly say how many bits are there in the output or input.

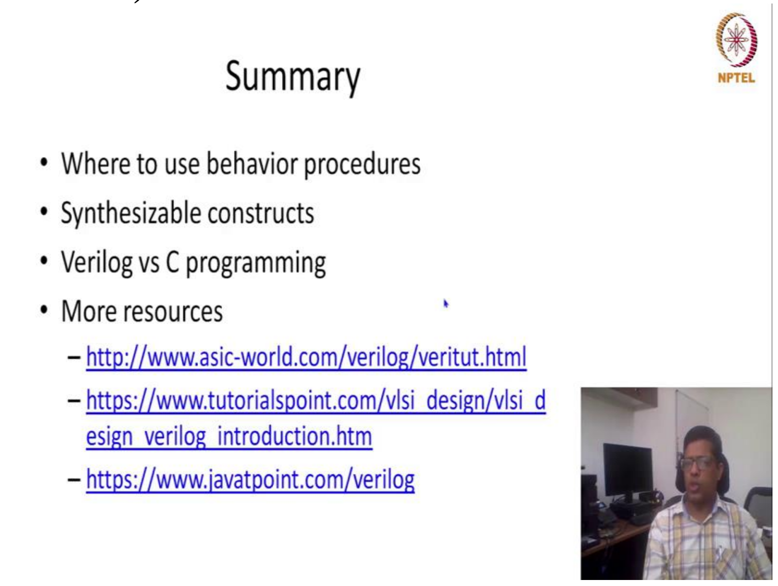
Now, since q is the output and because it is being also written in an always block we have to define it as a reg type. So again, let us say that this particular block is, this particular counter will get triggered, whenever there is a negative edge of the clock or, and my clear is actually active high.

So, because my clear is active high, so I can say always at the rate posedge because after posedge, there would be, clear would be 1. So, when we are saying so, then we can say that because if clear is 1, then I need not to think about clock so I can say if clear is there and clear can be there only if there is a posedge. So, only at that particular point of time, I will trigger this clear thing and if clear is there after positive edge then I can assign my output q to 0.

So, you remember this syntax that we are saying that my q is four bit but here, the specification here, or basically representation here is decimal type. So, we are not writing in binary, we are not writing in hexadecimal, we are writing in decimal. Now, in the else case, because there is no posedge of clear, the only other condition left is the negative edge of the clock so at every negative edge of the clock I am just going to say q equal to q plus 1.

Now, you can see here that I have used a non blocking statements here, non blocking assignments. So usually non blocking assignments are used, are usually preferred in case of, in case of flip-flop or flip-flop like structures and they are also preferred whenever we are trying to model a data flow or a combinational logic. So, this whole procedure, we will be able to have, we will be able to count whenever there is a negative edge of the clock. So, we can end the module by end module. So, this is how, we can use this behavior modeling and we can summarize our lecture.

(Refer Slide Time: 27:32)



## Summary

- Where to use behavior procedures
- Synthesizable constructs
- Verilog vs C programming
- More resources
  - http://www.asic-world.com/verilog/veritut.html
  - https://www.tutorialspoint.com/vlsi_design/vlsi_design_verilog_introduction.htm
  - https://www.javatpoint.com/verilog

So, we have seen behavior modeling in this lecture and we have also seen more powerful. Maybe, one question will be arising in your mind that why, why we were not taught about this, behavior modeling when we were talking, initially about Verilog? So, and why it was not recommended or when it was recommended?

So, essentially, this behavior model can be used for data flow modeling. Sorry, this behavioral model can be used for combinational logic, for sequential logics as well as for test benches. But we have not preferred, or we have not toughed the behavior model till this particular assignment because, it would be more efficient, it would be more appropriate to use behavior model whenever we are modeling sequential circuits.

When we are using sequential circuits, we do not have alternatives, we have to use behavior model because to implement D flip-flop or any kind of a memory structure, behavior models are the most appropriate ones because you have this reg type which is internally defined. This reg type essentially can hole two different values. One is the current value, another could be the future value, while wire cannot have two different values. So, the previous value will get erased.

So, that is why it is usually good to have behavior, whenever we are using sequential. Whenever we are modeling sequential circuits, it is good to use behavior models. For combinational logic also, we can use behavior model but during synthesis we are not sure what we are doing. And another biggest catch is that whenever we are using behavior model, we tend to write, like a C program. However, if we are modeling a hardware, we have to visualize, we have to write on paper what my hardware module is going to be. What kinds of operations will be there, how many cycles it will take, et cetera.

So, all of those things can be done if we are using either combinational, either if we are using structural model or data flow model. When we are using behavioral model, we tend to skip those details and we tend to write in a, like a C programming way. The other disadvantage of using behavioral model without thinking much, is sometimes, some of the construct is not synthesizable.

So, finally, when we are going to design a circuit or when we are going to synthesize it on that pg et cetera, so in those cases, it will not give the desirable result. We may be expecting, for example, a multiplexer but it may be designing something else. So, the cost may be different. Because we are leaving it to our synthesis tool that what to infer.

So, I would suggest to use judicially that when should we use behavioral statements and when do we use data flow and structural statements. The ideal would be to use a mix of them. So, sometimes we can use behavioral statements and, so for example, designing flip-flop, we do not have options. You have to write behavioral statements. But we can design some basic modules and when we connect those modules again using structural model and their, all the assignments require to be continues when we use data flow assignments.

So, this is about different kind of statements, different kind of modeling styles. So, if you require any more help in your assignments while doing and maybe, some more constructs or maybe confusion in some constructs, so any of these links could be a very helpful means. Asic World has a very good, nice explaining method and they also give a lot of examples. And Tutorial Point, as well as Java Point also have list of topics.

Mostly, what I have observed during my experience is that books may not be a very good idea but these online methods because the language is like we use. They are quite extensive and there could be quite small scenarios or corner cases which may not be covered in the books but these tutorials can help you quite a lot, whenever you are dealing with any particular point. So, with this, I would close this, today’s lecture. Thank you very much.