Hardware Modeling using Verilog

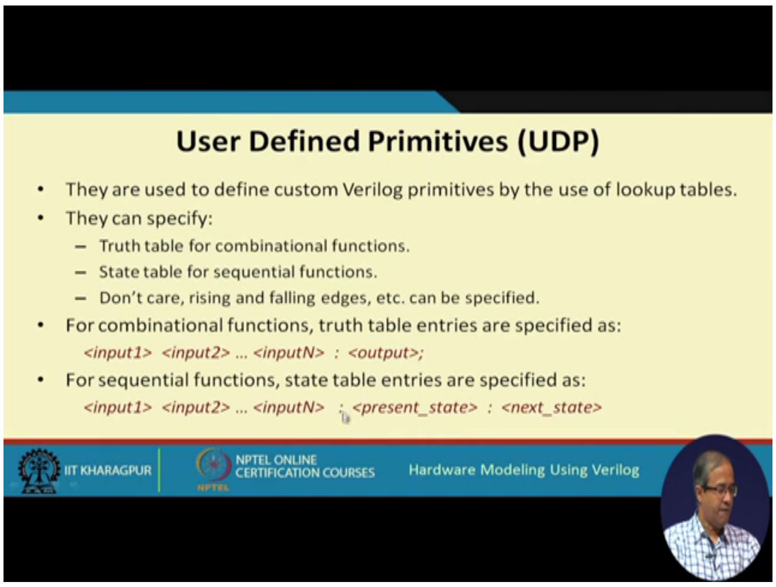
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 20
User - Defined Primitives

So, in this lecture, we shall be discussing a new feature of the Verilog language. It is called user defined primitives. This is the title of our lecture; User - Defined Primitives. So, we have seen so far that how you can write modules in Verilog; we can write modules and when you write modules; the description can be in terms of the behavior or it can be in terms of the structure where we instantiate either some built in primitive gates or some other modules.

Inside this module, we are writing, well, here in this user defined primitive, we are saying; this is also a way of specifying the functionality of a block in a behavioral way, but very specifically, we are specifying the behavior in terms of a truth table for a combinational circuit and a state transition diagram or a state transition table in the case of a sequential circuit. So, many a times, when we are just creating a design where some of the functional blocks; we want to specify in terms of the bits behavior and for some of them the behavior can be very conveniently expressed in terms of the truth table or the state table, there this user driven or user defined primitives can be quite helpful.

(Refer Slide Time: 01:52)



User Defined Primitives (UDP)

- They are used to define custom Verilog primitives by the use of lookup tables.
- They can specify:
  - Truth table for combinational functions.
  - State table for sequential functions.
  - Don't care, rising and falling edges, etc. can be specified.
- For combinational functions, truth table entries are specified as:
  <input1> <input2> ... <inputN> : <output>;
- For sequential functions, state table entries are specified as:
  <input1> <input2> ... <inputN> : <present_state> : <next_state>

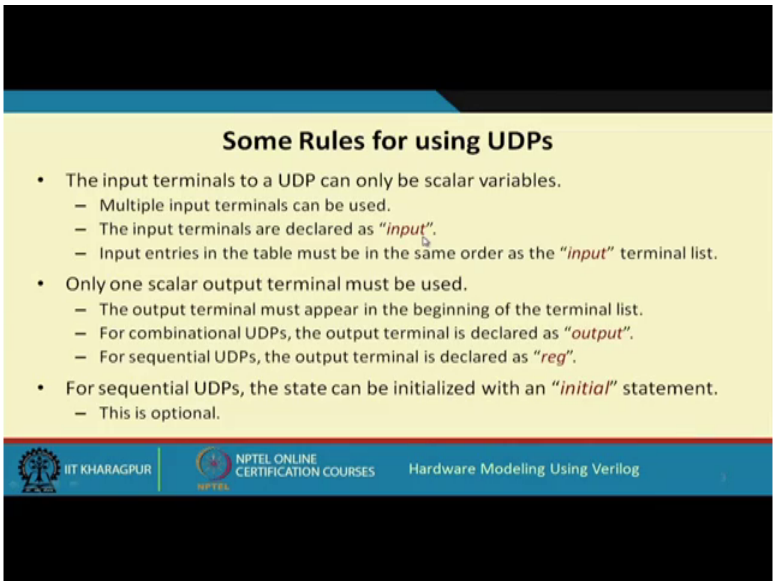
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, let us see what this is. So, this user defined primitives in short UDP; they can be used to define some as it said behavioral specification of some blocks in terms of look up tables. So, we specify the behavior in terms of tables in a particular format; we shall see now using UDP as I had said, we can either specify for combinational functions, its behavior in terms of the truth table or for sequential functions, we can specify the behavior in terms of its state table. Now when we specify these tables, we can also indicate various equal events like some of the inputs can be do not cares some of the inputs can be rising or a falling edge kind of signals. So, these kinds of signals can also be specified; we should see some examples here.

So, for combinational functions; when you specify a particular row of this table; this truth table, the format is like this the value of the inputs. We first give separated by spaces, suppose there are n inputs and then there is a colon then we give the expected output; what is the output of that function when this input is applied. So, if it is a truth table, we shall be giving this kind of a line several times one per row of the truth table, similarly when it is a sequential function, then when we specify the inputs as usual and after colon, we specify what is the present state of the circuit or the machine and after another colon, we specify the next state.

Now this present state and next state can also be in terms of the bid values. So, this is the state table description one row of it.

(Refer Slide Time: 04:00)



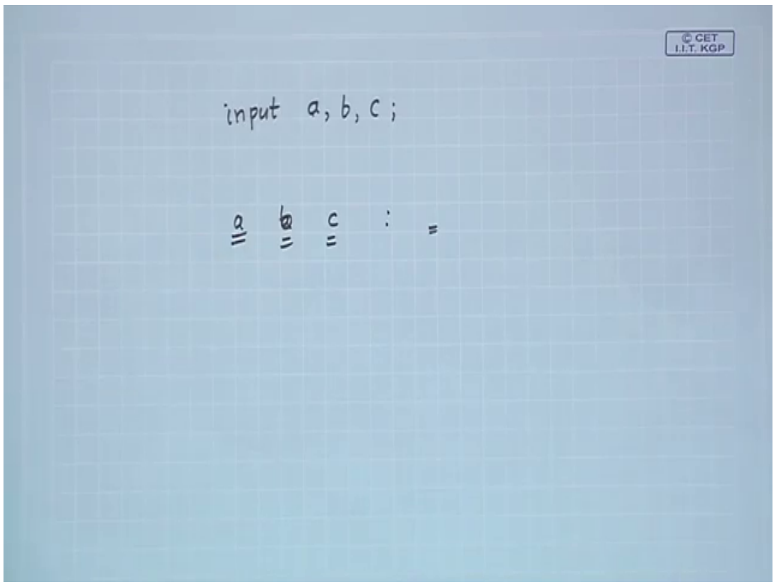
Some Rules for using UDPs

- The input terminals to a UDP can only be scalar variables.
  - Multiple input terminals can be used.
  - The input terminals are declared as "*input*".
  - Input entries in the table must be in the same order as the "*input*" terminal list.
- Only one scalar output terminal must be used.
  - The output terminal must appear in the beginning of the terminal list.
  - For combinational UDPs, the output terminal is declared as "*output*".
  - For sequential UDPs, the output terminal is declared as "*reg*".
- For sequential UDPs, the state can be initialized with an "*initial*" statement.
  - This is optional.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, there are some rules that we need to follow when you are using this kind of user defined primitives the rule are as follows that the input terminal the inputs that we are giving, they cannot be vectors, they can only be individual scalar variables, but of course, we can use multiple such variables multiple input terminals can be used and they have to be declared as input using the keyword input and the point to note is that in the input description the order in which we apply the input for example.

(Refer Slide Time: 04:41)

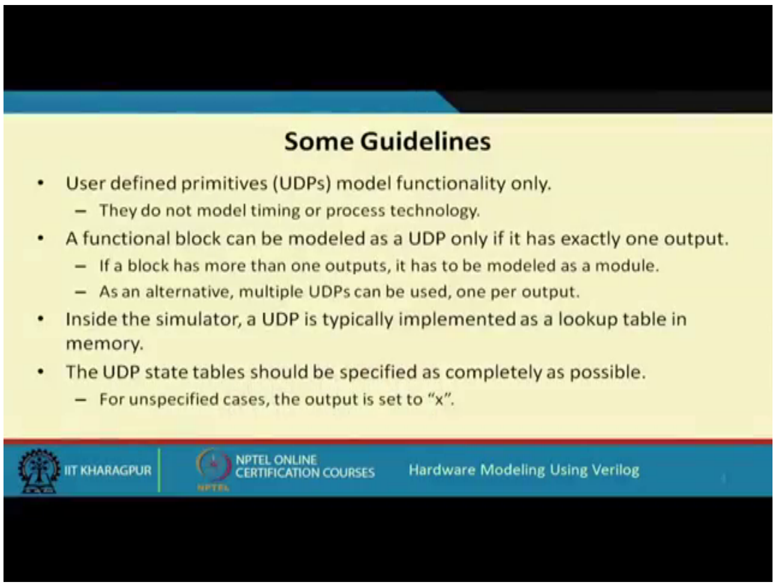


In the input description we write; let us say input a, b and c. So, when we give the table, we have to first give the value of a, then give the value of b, then give the value of c and then the expected output. So, the order in which you specify the variables in the input declaration the input variables should appear in the table in exactly the same order. This is something you have to remember.

So, the input entries in the table must be in the same order as they are specified in the input declaration and another point to notice that in one UDP, you can only define a single output function. So, only one scalar output terminal can be used. It cannot be a vector a single bit and in the terminal list when you declare the terminals, I shall give an example later, the output terminal must be the first one appearing followed by the input terminals and just like in a module the output terminal has to be distinguished using the keyword output, but for sequential circuits the output terminal should be declared as reg, right.

Particularly for sequential UDPs where we are talking about the present state and the next state, sometimes we may have to initialize the state of the machine; suppose unless we specify the initial state; whenever I apply an input, we really do not know what the next state will be because they do not know the present state. So, for sequential UDP and additional facilities; there you can include an initial statement one using which you can initialize the present state of the machine.

(Refer Slide Time: 06:40)



Some Guidelines

- User defined primitives (UDPs) model functionality only.
  - They do not model timing or process technology.
- A functional block can be modeled as a UDP only if it has exactly one output.
  - If a block has more than one outputs, it has to be modeled as a module.
  - As an alternative, multiple UDPs can be used, one per output.
- Inside the simulator, a UDP is typically implemented as a lookup table in memory.
- The UDP state tables should be specified as completely as possible.
  - For unspecified cases, the output is set to "x".

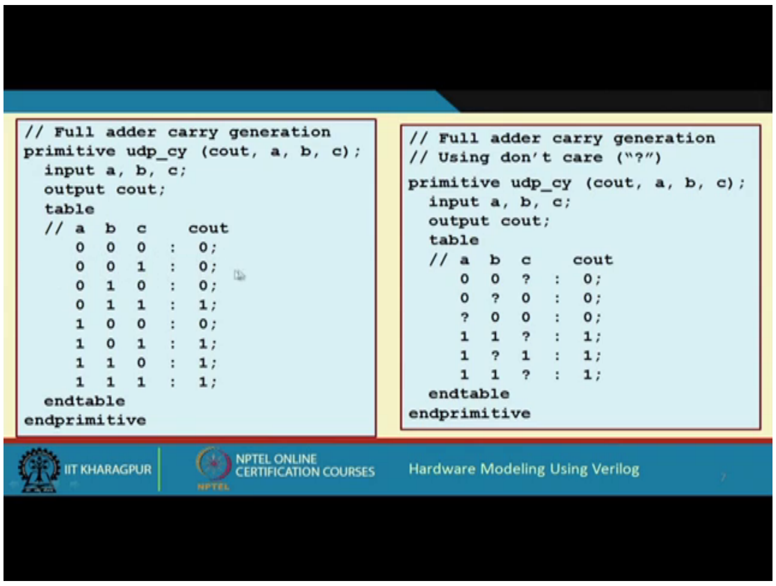
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, this state can be initialized with an initial state; however, this is optional; you may or may not give this some guidelines are as follows well means here as it said, when you declare or define UDPs, you define them in the form of a table or a truth table or a state table. So, it models only the functionality just the input output behavior. So, you do not specify any kind of delay or process technology, see we are not talking about anything related to the hardware that you are going to design, we are simply defining our functionality in terms of black box. This will be my input, this will be my output; that is it, we are not talking about any kind of gates any kind of delays any kind of process technology, what kind of delays are the things will be encountered and so on.

So, as I said earlier also that in an UDP, you can have exactly one output terminal. So, if you need to define a block which has more than one outputs, then it has to be declared as a module and not as a new db, but; however, you can use multiple UDP is one per output that is also permitted. Now when you simulate a design that contains UDP typically inside this simulator, this UDP is implemented as a table. So, if it is a truth table, it maintains that table in memory.

So, whenever some input is applied, there is a table lookup and the output command determined. Similarly, the state table of a sequential circuit that is also maintained like a table in memory; so, it maintains the input output behavior in terms of a lookup table in simulator memory and when you are declaring sequential circuit behavior sequential functions, the state table should be specified as completely as possible, well, if you have some combinations where it is not the inputs are not specified for those cases, the outputs will be automatically set to undefined or x values.

(Refer Slide Time: 08:59)



```
// Full adder carry generation
primitive udp_cy (cout, a, b, c);
  input a, b, c;
  output cout;
  table
  // a  b  c     cout
     0  0  0  :  0;
     0  0  1  :  0;
     0  1  0  :  0;
     0  1  1  :  1;
     1  0  0  :  0;
     1  0  1  :  1;
     1  1  0  :  1;
     1  1  1  :  1;
  endtable
endprimitive
```

```
// Full adder carry generation
// Using don't care ("?")
primitive udp_cy (cout, a, b, c);
  input a, b, c;
  output cout;
  table
  // a  b  c     cout
     0  0  ?  :  0;
     0  ?  0  :  0;
     ?  0  0  :  0;
     1  1  ?  :  1;
     1  ?  1  :  1;
     1  1  ?  :  1;
  endtable
endprimitive
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

Let us now look at some examples; first modeling of combinational circuits; let us start with something we already know about the sum output of a full adder. You see this is the declaration of a user defined primitive of this sum output. So, it looks exactly like a module in terms of declaration, but instead of the module keyword, we use a primitive keyword here and a end primitive keyword here. So, the name of the block in this case they will user defined primitive. So, UDP sum is the name we have given and these are the arguments. So, as I said the output must be the first one appearing in this list sum is the output and a, b, c are the inputs.

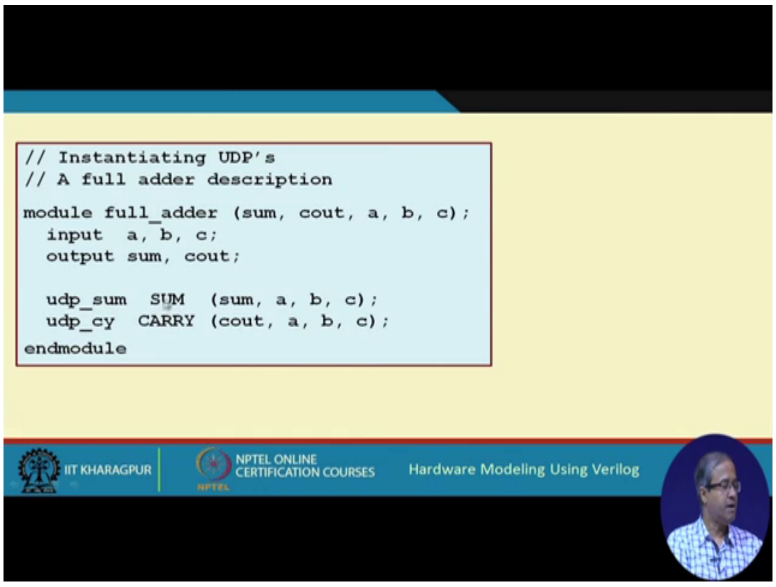
So, inputs are declared by the keyword input, the output is declared by the keyword output, then the truth table is specified using the keywords table and end table just for convenience, I have included a comment line here just to tell you that this is a, this is b, this is c and this is sum. So, the truth table there with 3 input, there will be 8 rows. So, all the 8 combinations are specified if the input is 0, 0, 0, sum will be 0, if it is 0, 0, 1, sum will be 1 and so on, if it is 1, 1, 1, sum will be 1. This is how you can specify the truth table of a sum output.

So, in this example, we have specified the output value for all possible input combinations, but there can be functions where some of the inputs you can specify as don't cares, also now here the don't care values are indicated by a question mark. So, we shall be illustrating with another example; just the carry output of the full adder. This is just the other example, well, this is instead of sum this is the carry.

So, I have shown 2 alternate descriptions, both are equivalent, first one is the one where we have expressly specified all the 8 combinations in the truth table. So, it is exactly like this. Some example we took earlier. So, here the output variable is c out; carry out and in the comment, I have just mentioned just for convenience these are a, b, c; this is c out. So, for each input value; what is the expected value of carry out? We have just listed them in this table.

Now the description on the right is exactly the same thing, but here we have used don't cares. Well, how we have used on kids, you see we are talking about generation of carry. So, if any 2 of the inputs are 0, there cannot be any carry. So, if a and b are 0, but c is an don't care, there cannot be occurring. Similarly, if a and c are 0 and b is a don't care, then also no carry and if b and c are 0, then also no carry. Well, the condition for carry generation is at least 2 of the inputs must be 1. So, either a, b is one, c is don't care or a and c is 1; b is don't care or a and b is or this should be actually I think, there is a small typo; this should be a; should be don't care and this should be 1, fine.

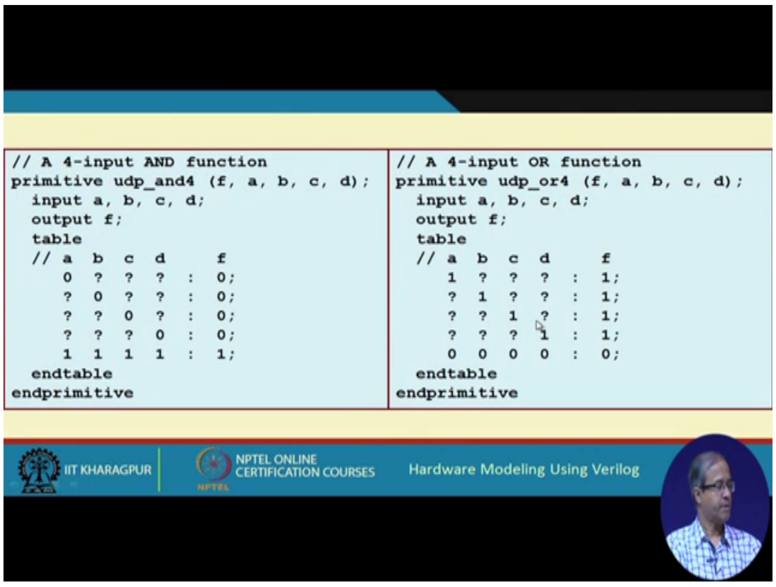
(Refer Slide Time: 12:43)


```
// Instantiating UDP's
// A full adder description

module full_adder (sum, cout, a, b, c);
  input  a, b, c;
  output sum, cout;

  udp_sum  SUM  (sum, a, b, c);
  udp_cy  CARRY (cout, a, b, c);
endmodule
```
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, this is how you can use don't care in the truth table description. You see instead of 8 here, we need only 6 rows to specify it. There are some examples of combinational functions. Now once you declared these kind of sum here and carry here these, you can instantiate in a module just like module description which you saw earlier suppose we have declared these 2, this UDP is 1 is called UDP. Some other is called UDP carry, we instantiate them just like their modules.

So, for instantiation it really does not make any difference whether it is another module or it is a UDP the way for instantiation or the rule for instantiation is the same just you specify the name of that module; give it a instantiated name and the parameter list. So, the way to include or instantiate a UDP is the same as that for a module. So, for a complete full adder description, you can have a description like this; you instantiate the UDP some instantiate the UDP caddy and have the full adder description.

(Refer Slide Time: 13:59)



```
// A 4-input AND function
primitive udp_and4 (f, a, b, c, d);
  input a, b, c, d;
  output f;
  table
  // a  b  c  d     f
     0  ?  ?  ?  :  0;
     ?  0  ?  ?  :  0;
     ?  ?  0  ?  :  0;
     ?  ?  ?  0  :  0;
     1  1  1  1  :  1;
  endtable
endprimitive
```

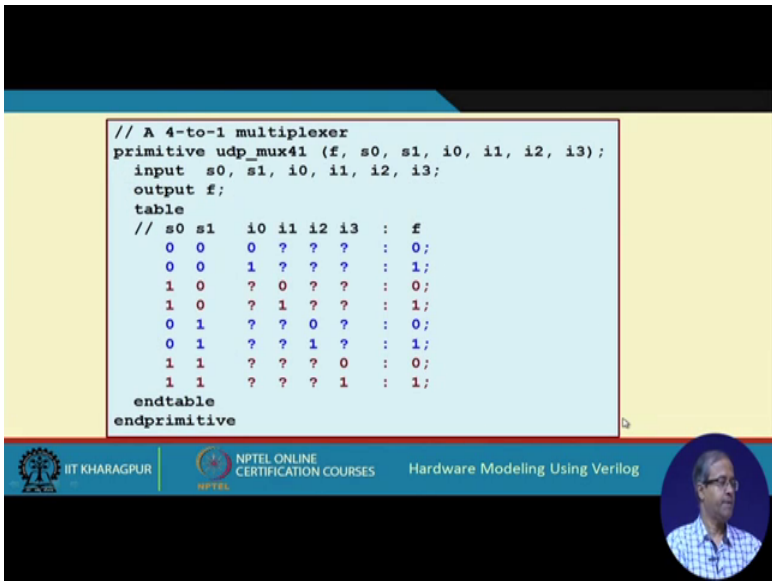
```
// A 4-input OR function
primitive udp_or4 (f, a, b, c, d);
  input a, b, c, d;
  output f;
  table
  // a  b  c  d     f
     1  ?  ?  ?  :  1;
     ?  1  ?  ?  :  1;
     ?  ?  1  ?  :  1;
     ?  ?  ?  1  :  1;
     0  0  0  0  :  0;
  endtable
endprimitive
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, as I said, they can be instantiated just like any other Verilog module. Let us take some other very simple functions where you will see that we can use a lot of don't cares. Just consider we want to implement or we want to specify a 4 input end and a 4 input or function. So, for end function, again the output will be the first one appearing then the 4 inputs a, b, c, d; they are declared. So, again; they just in a comment, I have shown the inputs and the outputs.

Now for end function, the output will be one if all the inputs are 1. So, 1, 1, 1, 1; that is the only combination when the output will be 1 and the output will be 0; if at least 1 of the inputs is 0. So, if a is 0, b, c, d are don't cares; then 0 or if b is 0 other said don't cares; c is 0 or d is 0, f will be 0. So, c instead of 16 rows in the truth table, here we are able to specify it only in 5 rows. Similarly for OR, it is just the reverse for an OR gate; the output will be 0, if all the inputs are 0s. This is the last row is specified and if any one of the input is 1; the others can be don't cares; the output will be 1. So, as you can see here also, we need 5 rows to specify.

(Refer Slide Time: 15:26)



```
// A 4-to-1 multiplexer
primitive udp_mux41 (f, s0, s1, i0, i1, i2, i3);
  input  s0, s1, i0, i1, i2, i3;
  output f;
  table
  // s0 s1   i0 i1 i2 i3   :   f
     0  0    0  ?  ?  ?    :   0;
     0  0    1  ?  ?  ?    :   1;
     1  0    ?  0  ?  ?    :   0;
     1  0    ?  1  ?  ?    :   1;
     0  1    ?  ?  0  ?    :   0;
     0  1    ?  ?  1  ?    :   1;
     1  1    ?  ?  ?  0    :   0;
     1  1    ?  ?  ?  1    :   1;
  endtable
endprimitive
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

Let us look at a slightly more complex example of 4 to 1 multiplexer; you see functional blocks like multiplexers or decoders; they are pretty convenient to specify in terms of the input output behavior or truth table. So, multiplexer is one classical example where this UDP description can be pretty convenient. Let us see how we can do it. So, we are declaring a 4 to 1 multiplexer. So, we had said, we cannot define vectors the inputs have to be all scalars.

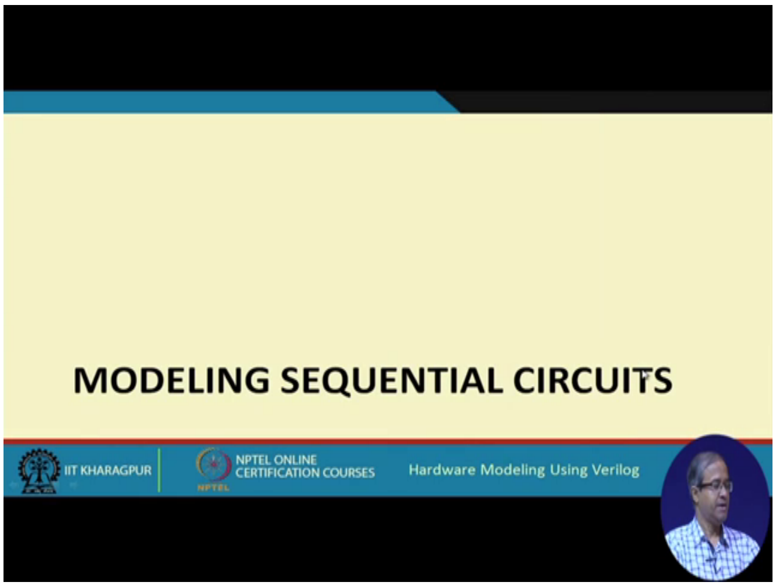
First one is the output, then next 2 are the select inputs, next 4 are the primary inputs to the OR, the inputs to the multiplexer the 4 inputs. So, this 6 are the inputs s 0, s 1 and i 0 to i 3 and f is the output. Now in the description, we have again given a comment; the first 2 are the select lines; this is the least significant next one is the most significant and then the 4 inputs i 0, i 1, i 2, i 3, then colon f see this line; we give normally in order to understand that which bit is what just to have a feel. Now the idea is that the inputs there will be exactly coming in the same order; you have specified here s 0, s 1, i 0, i 1, i 2, this should be same order the last one will be the output.

So, select line; there are 2, there can be 4 combinations, I have shown them by colors; there can be 0 0, they can be 0 1, they can be 1 0 or they can be 1 1. So, if it is 0 0, then i 0 is supposed to be selected and it should be going to f. So, we have written 2 rows, if i 0 is 0 f will be 0, if i 0 is 1 f will be 1; the other set don't cares. Similarly, when the inputs are 1 and 0, s 1 is 1; is 0 is 0. So, 0 is 1 s 1 is 0 then i 1 is supposed to be selected. So, if i

1 is 0 output is 0 i 1 is one output is one other said don’t cares. Similarly, for 1 0 combination, i 2 is selected and for 1 1 combination; i 3 is selected.

So, you see; there are 6 inputs. So, the complete truth table is supposed to contain 64 rows 2 to the power 6, but in this compact district description in terms of the functionality of the multiplexer we need only 8 rows to specify. So, and these 8 rows are sufficient to specify the functions of the multiplexer, exactly, what you are trying to do? We want to use this s 0 and s 1 to select one of the inputs. So, whether they are getting selected or not that is what we have specified here.

(Refer Slide Time: 18:32)

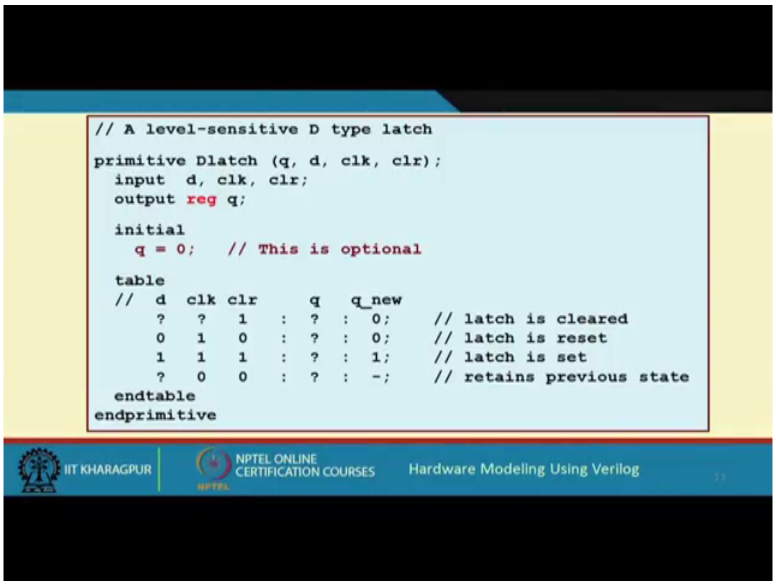


MODELING SEQUENTIAL CIRCUITS

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

Just 8 rows; next let us see; how we can model sequential circuits. Now in a sequential circuit, you remember you recall. So, it is unlike a combinational circuit where we apply input you get an output well in a sequential circuit there is something called a state, it remembers some previous history. So, whenever we apply an input the output will depend not only on the input or applying but also in terms of this state where the machine is in. So, in the UDP of a sequential circuit; you can specify the input you can specify the present state, you can specify the next state now the restriction of a UDP is your next state that is the output has to be a single bit single scalar variable, it cannot be multiple.

(Refer Slide Time: 19:34)



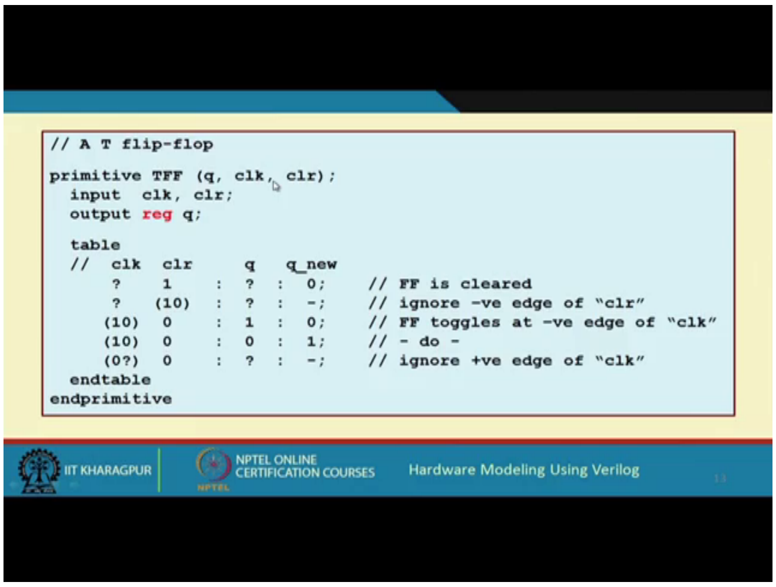
```
// A level-sensitive D type latch

primitive Dlatch (q, d, clk, clr);
  input  d, clk, clr;
  output reg q;

  initial
    q = 0;   // This is optional

  table
  //   d   clk clr     q    q_new
       ?    ?   1   :  ?  :  0;    // latch is cleared
       0    1   0   :  ?  :  0;    // latch is reset
       1    1   1   :  ?  :  1;    // latch is set
       ?    0   0   :  ?  :  -;    // retains previous state
  endtable
endprimitive
```

So, if there are multiple variables; you want to generate or declare; you have to declare it in terms of a module not a UDP. Let us see this is a very simple description of a D type latch; let us see just like module we have declared primitive end primitive name is D latch the inputs are d clock and this clock is like an enable because if the level sensor is not real clock it is like enable whenever clock is high d will go to q and clear to clear the output.

So, d clock and clear at the inputs and this d for a sequential circuit, you have to declare the output of the reg I told you. So, I am declaring output at reg q. So, in terms of the sequence circuit description, I told you; have to first specify the inputs in this case with specifying in this order d clock clear, then the present state, then the next state; we are just referring to as q and q new just look at the combinations first row says that I have applied clear equal to one d and clock are don’t care. So, if clear equal to 1 irrespective of d and clock irrespective of the present state, my next state will be 0. It is cleared, right and if I apply a equal to 0 and make clock equal to high and not clear, if clear is 1, it look cleared, if clear is 0, then the output will be 0, here also I made a typo, they should be 0, fine.

So, if I apply d equal to 0 and clock high the next state will be 0 and if I apply d equal to 1 and clock high; this one will go, next state will be 1 and this will be irrespective of what my present state is and of course, my clear should be 0 not clear and last row specifies that if my clock is not active my enable is not active my clear is also not active, then you see in the next state, I put a dash; dash means no change retains previous state. So, for a detail latch the next state is not directly dependent on the previous state, but on what you are applying the only combination that depends on the professor is the last one dash it will remember whatever it was previously, right.

(Refer Slide Time: 22:14)

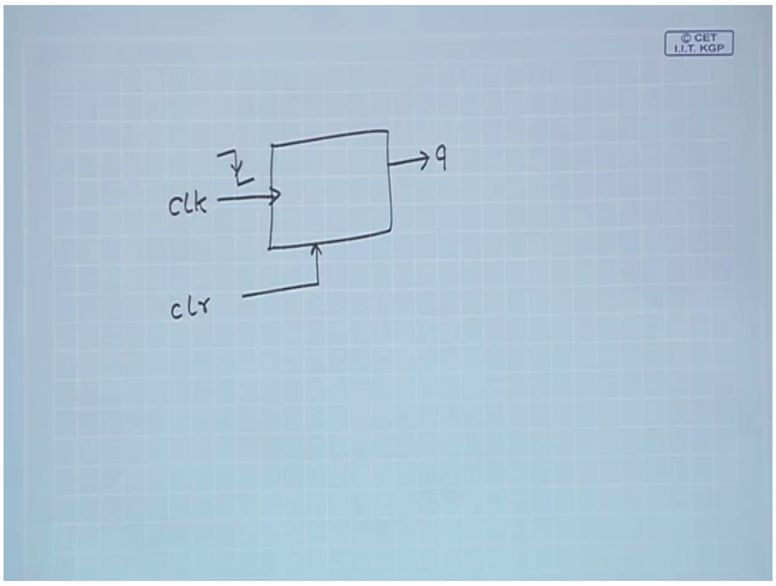


```
// A T flip-flop

primitive TFF (q, clk, clr);
  input  clk, clr;
  output reg q;

  table
  //   clk  clr      q   q_new
        ?    1    :  ?  :  0;    // FF is cleared
        ?   (10)  :  ?  :  -;    // ignore -ve edge of "clr"
      (10)   0    :  1  :  0;    // FF toggles at -ve edge of "clk"
      (10)   0    :  0  :  1;    // - do -
      (0?)   0    :  ?  :  -;    // ignore +ve edge of "clk"
  endtable
endprimitive
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

Now, let us look at a flip-flop where there is a clock signal proper clock signal. So, here we are defining a toggle flip-flop or a T flip-flop where I mean our descriptions like the q clock and clear. So, our description of this flip-flop is that there is a clock input there is a q output and there is a clear input.

(Refer Slide Time: 22:31)

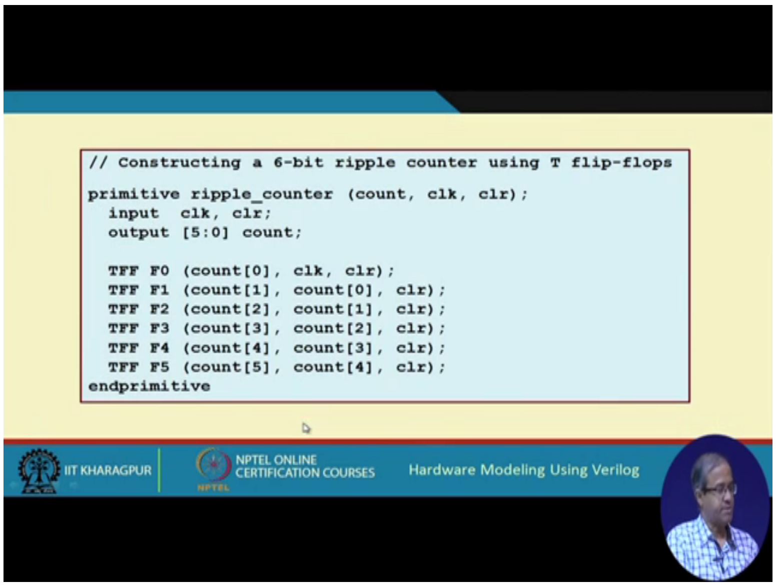


So, whenever there is a clock signal, I am assuming it is following, it is triggered 1 to 0; the output will be toggled if it was 1, it will become 0 if it was 0 it will become 1. So, whenever there is a clock; this is a simple description of a sequential circuit; I am assuming there is a clock input; a clear input and q whenever there is a clock the output will toggle if it is a clear; then output will be 0 let us see.

So, again clock and clear the inputs and q is the output declare a step reg. So, we are specifying. In this order, first row specifies if clear is high, then irrespective of whether clock is not; whatever was the previous state the output is becoming 0 next state is 0 q q new next was specifies my clear is becoming 1 to 0, I have withdrawn the clear, it was clear; I withdrawn it. So, the output value will remain in the previous state. So, if there is a negative agent clear we are ignoring it, we are retaining the previous state. Now there is a clock you say within bracket, if we write one 0 it means a falling edge see in clock within bracket we are writing ones or this means one to 0 transition there is a transition from one to 0; that means, a falling edge.

So, in the next 2 rows, we are saying that there is a falling edge on the clock not clear and my previous state was 1 and 0. So, now, if my previous state was 1, it will be toggled, my current state will be 0 if my previous state was 0 my current state will be one and if there is a positive edge of the clock it was 0 to question mark was anything, then there will be no change. So, that is indicated by a dash this is a very concise description of a T flip-flop in terms of the state table let us take some more examples.

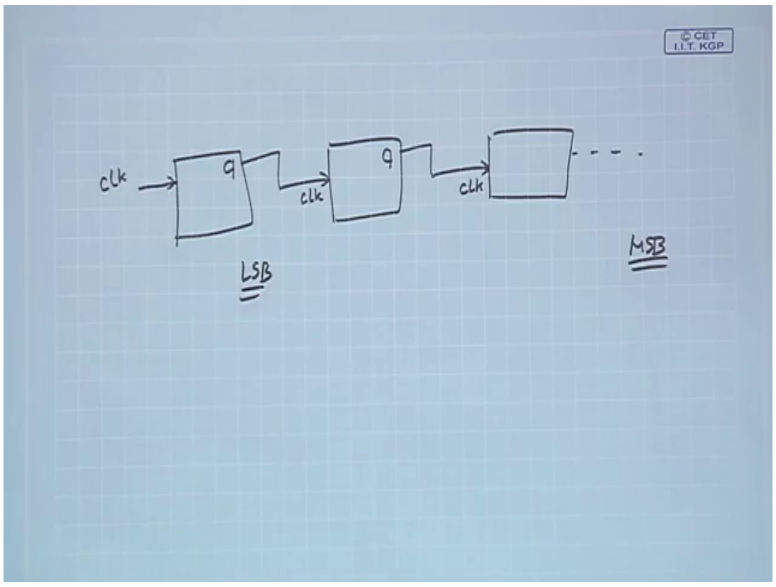
(Refer Slide Time: 24:52)



```
// Constructing a 6-bit ripple counter using T flip-flops
primitive ripple_counter (count, clk, clr);
  input  clk, clr;
  output [5:0] count;

  TFF F0 (count[0], clk, clr);
  TFF F1 (count[1], count[0], clr);
  TFF F2 (count[2], count[1], clr);
  TFF F3 (count[3], count[2], clr);
  TFF F4 (count[4], count[3], clr);
  TFF F5 (count[5], count[4], clr);
endprimitive
```

This T flip-flop that I have just now defined well we can connect a number of such T flip-flops to create a ripple counter. So, how do create a ripple counter a number of T flip-flops connected in cascade.

(Refer Slide Time: 25:11)

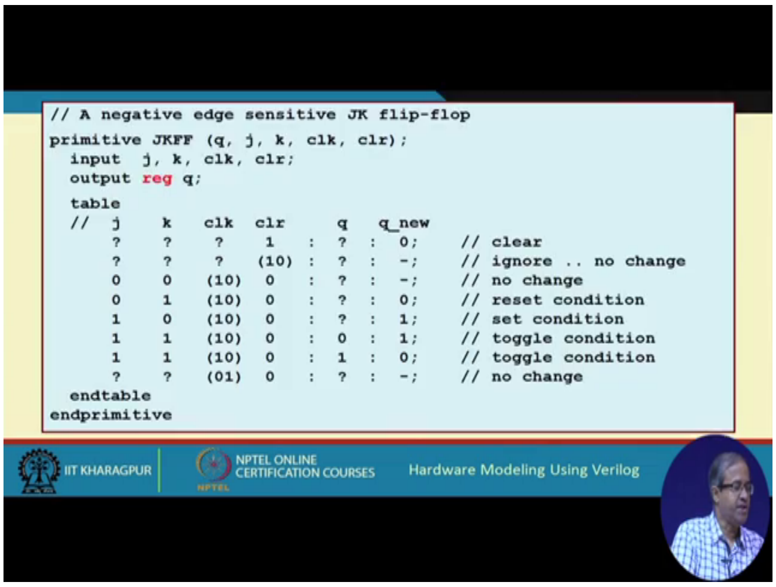


So, how do we do it, suppose I have several such flip-flops? So, the output of one flip-flop; I am connecting into the clock input of the next flip-flop, the output of the next I am connecting to the clock input of the next flip and so on.

And this is my final clock. My initial profit is applying. So, it will be counting this and the q outputs which I will be coming this will be your LSB and on the right side, this will be a most significant bit this will be your counter. So, here I am just showing you how this flip-flop can be instantiated to create a 6 bit ripple counter. So, what we have done? We have taken this same flip-flop; you see first one is q, second one is clock, third one is clear. So, we have taken the same one we have instantiated it 6 times 6 copies of the flip-flop for the flip-flop on the left hand side the first one.

So, the output you are calling us count 0 clock, we are directly applying like you see the clock you are applying from outside we are just applying clock from outside, that is why the input clock is connected, here clock and clear is connected to all of them second flip-flop the output, I am calling it as count one instead of clock I am connecting the output of the first flip-flop, here you see what we did here again the output of the first flip-flop is connected to the clock of the next similarly here. So, we have done the same thing, this output of this flip-flop is connected as a clock here output of this is connected as clock here like this. So, in this way, we have created a 6 bit ripple counter.

(Refer Slide Time: 27:07)



```
// A negative edge sensitive JK flip-flop
primitive JKFF (q, j, k, clk, clr);
  input  j, k, clk, clr;
  output reg q;
  table
  //  j    k   clk  clr      q    q_new
      ?    ?    ?    1    :  ?  :   0;    // clear
      ?    ?    ?   (10)  :  ?  :   -;    // ignore .. no change
      0    0   (10)  0    :  ?  :   -;    // no change
      0    1   (10)  0    :  ?  :   0;    // reset condition
      1    0   (10)  0    :  ?  :   1;    // set condition
      1    1   (10)  0    :  0  :   1;    // toggle condition
      1    1   (10)  0    :  1  :   0;    // toggle condition
      ?    ?   (01)  0    :  ?  :   -;    // no change
  endtable
endprimitive
```

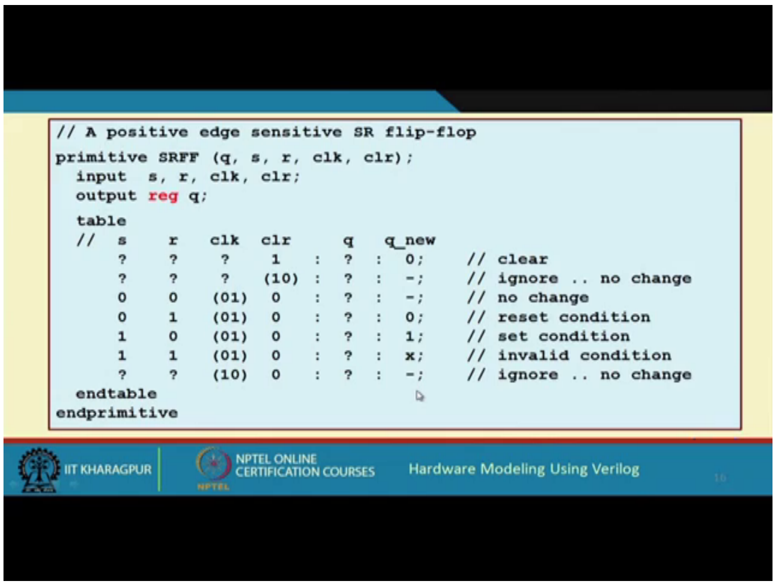
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL
Hardware Modeling Using Verilog

Now, just one thing here; we have called it primitive, but if you want you can also call it a module because it is as good as a module description. So, I have given primitive and end primitive, but you can as well express it as a module this can be a module where the 6 flip-flops have been instantiated, right.

Now, let us look at slightly more complex kind of flip-flops here is a negative edge sensitive j k flip-flop. So, in a negative edge sensitive j k flip-flop there is an output, q j k are the inputs, there is a clock and a clear j k clock, clear the inputs, output q is the output. So, we have specified them in this order and these are the output. So, the first row says again if clear is 1, irrespective of anything else; output will be 0 and if the clear changes from one to 0 there will be no change next row says if j k is 0, 0, you just recall the definition of a j k flip-flop if j k is 0, 0, the output is supposed to remain same no change and there is a clock; clock goes from 1 to 0, it is not clear. So, output will not change, but if its 0 1, then the output will be 0, if there is a clock the irrespective of the previous output one 0 the output will be 1, but if it is 1, 1, the output will be toggled.

So, here we specify the previous if it is 1, 1, if the previously output was 0, not only 1 if previously it was 1, now to be 0 and if the clock is rising 0 to one not the active edge then again no change.

(Refer Slide Time: 29:10)



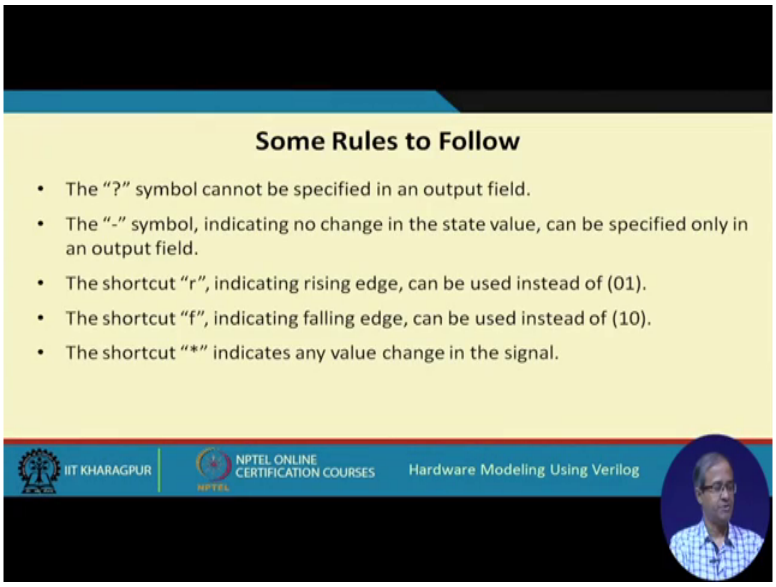
```
// A positive edge sensitive SR flip-flop
primitive SRFF (q, s, r, clk, clr);
  input  s, r, clk, clr;
  output reg q;
  table
  //  s    r   clk  clr     q    q_new
      ?    ?    ?    1    :  ?  :  0;    // clear
      ?    ?    ?   (10)  :  ?  :  -;    // ignore .. no change
      0    0   (01)  0    :  ?  :  -;    // no change
      0    1   (01)  0    :  ?  :  0;    // reset condition
      1    0   (01)  0    :  ?  :  1;    // set condition
      1    1   (01)  0    :  ?  :  x;    // invalid condition
      ?    ?   (10)  0    :  ?  :  -;    // ignore .. no change
  endtable
endprimitive
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, this is a very concise description of a j k flip-flop. Similarly, you can have an s r flip-flop, let us give the name properly s r flip-flop, fine. So, this is positive edge sensitive; s r flip-flop; same way you see the first few rows are identical to j k first 2 if 0, 0, no change; 0, 1; it will be 0, 1, 0 to be 1, but 1, 1 is an is an undefined condition for a s r flip-flop. So, when the inputs are 1 and 1 and if there is a clock, suppose I am saying; it is a positive edge sensitive 0 to 1, then the output; I am marking as x undefined and if the clock is on the other edge no change, right.

So, in this way, we can actually specify means any kind of simple combinational or sequential circuit blocks provided there is a single output. So, we have taken examples of flip-flops where there is one output, we have taken examples of gates and multiplexers where there is one output only such functional descriptions can be specified using such UDPs.

(Refer Slide Time: 30:26)



Some Rules to Follow

- The "?" symbol cannot be specified in an output field.
- The "-" symbol, indicating no change in the state value, can be specified only in an output field.
- The shortcut "r", indicating rising edge, can be used instead of (01).
- The shortcut "f", indicating falling edge, can be used instead of (10).
- The shortcut "*" indicates any value change in the signal.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, there are a few rules that you need to follow that the question mark symbol is the don't care symbol. This you can use only for the inputs you cannot use in the output field and dash indicates no change. This again you cannot use in the input you can use only in the output field and 0, 1, as it said indicates it changes from 0 to 1. So, instead of 0, 1, you can write a shortcut r; also r is equivalent to rising. Similarly 1, 0, you can write f falling and star in place of an input signal means any value change any change in that value is indicated by star. So, you can also include this in your table. So, with this way come to the end of this lecture.

Now, in this lecture, as I said, we had looked at a new feature of the language Verilog namely the user defined primitives. So, if you want; you can include such UDP descriptions for some blocks in a design bigger design you can instantiate the UDPs in your main design modules.

Thank you.