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

**Lecture No #03**
**Verilog: Part II**

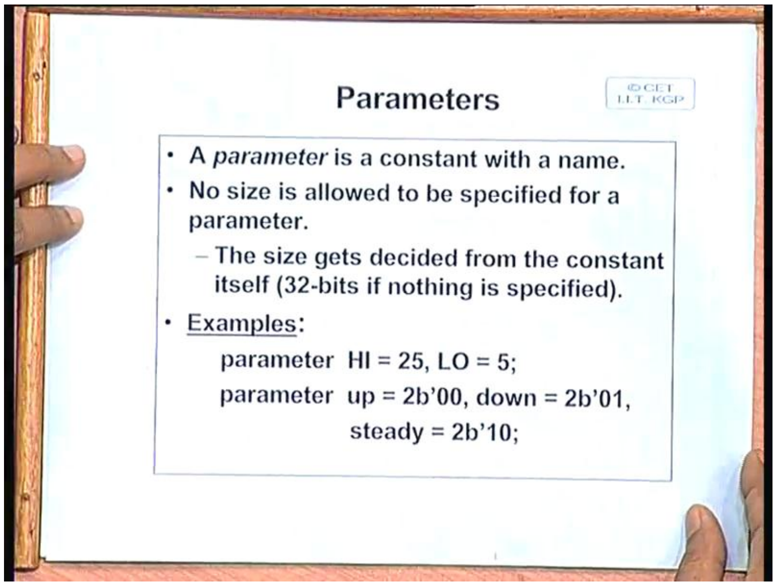
Today we will be continuing our discussion on the language Verilog.

(Refer Slide Time: 01:08)

VERILOG – Part II

So we just recall in our last lecture we had looked at some typical verilog programs codes and we had seen. For instance how we can declare a variable, we had said that we can have a variable of 2 types. Net type and register type and depending on the way we were using it. It can be synthesized into either an interconnecting wire or into a storage element like a register or a latch and we also looked at how we can define a constant value in an expression. So, continuing from that point onward.

(Refer Slide Time: 01:52)



## Parameters

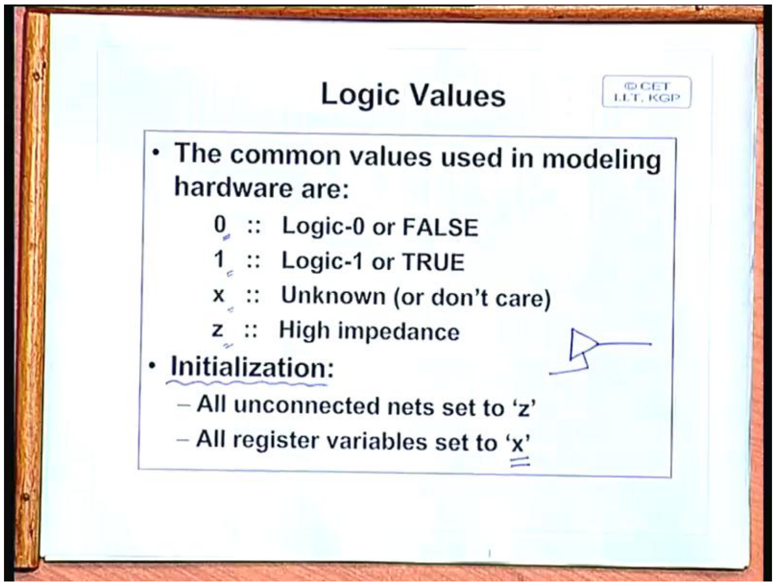
- A *parameter* is a constant with a name.
- No size is allowed to be specified for a parameter.
  - The size gets decided from the constant itself (32-bits if nothing is specified).
- Examples:

```
parameter  HI = 25, LO = 5;
parameter  up = 2b'00, down = 2b'01,
           steady = 2b'10;
```

Today first we shall see that how we can define a named constant; a constant with a name. Now in Verilog it is called a parameter. So a parameter is a constant which can be defined against a name and that name can be used in an expression. For example these are ways in which you can define a parameter. For example HI high equal to 25 LO low equal to 5 or parameter up equal to 0 0. This down equal to 0 1, steady equal to 0 1. You can define like this. In the Verilog code instead of writing 0 0 0 1 and 1 0 you can refer to them by up down or steady okay. Now here 1 characteristic of this parameter type is that you do not specify the size of the parameter when you when you define it in a name. Now this size gets decided from the constant which gets assigned to it okay. For example in this case high equal to 25 or low equal to 5.

Here you can find out easily that high can be represented in 5 bits; low can be represented in 3 bits. So this, the value of the constant will determine the size of this constant. How it will be mapped into the actual hardware and this when it is actually synthesized. But in some cases we can simply define a parameter the value will be coming from outside. It is something similar to a global variable. So there by default the size is taken to be 32 bits because inside the module you really do not know what is the value that will be assigned okay. Fine. Sir in this case the size of high (Student Noise Time: 03:46). Yeah in this case if you specify the value you can you can easily determine how many bits you require to represent that here to be 2. (Student Noise Time: 03:45). If nothing is specified by default it is 32. Yes if it not initialized it is taken to be 32.

(Refer Slide Time: 04:05)



## Logic Values

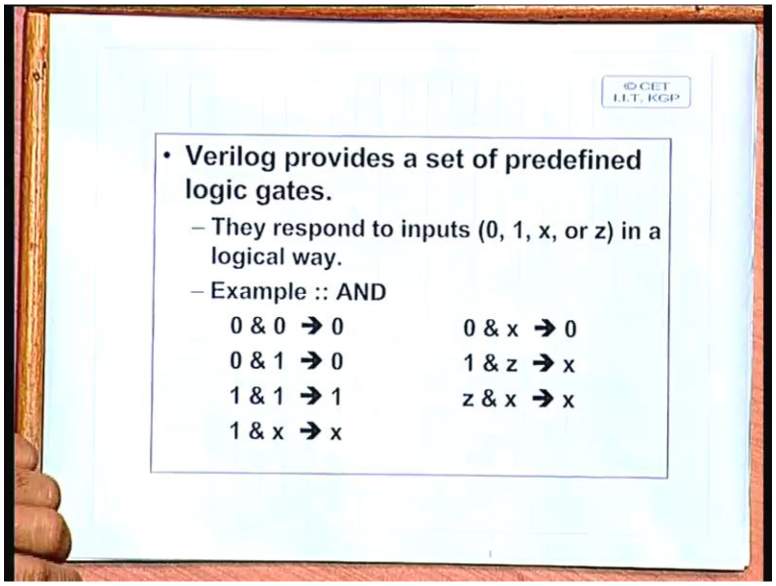
©CET I.I.T. KGP

- The common values used in modeling hardware are:
  - 0 :: Logic-0 or FALSE
  - 1 :: Logic-1 or TRUE
  - x :: Unknown (or don't care)
  - z :: High impedance
- Initialization:
  - All unconnected nets set to 'z'
  - All register variables set to 'x'

Okay next let us come to the different logic values that you can use in specifying for simulation or for synthesis. Now in Verilog these 4 are the logic values which are used. 0 means logic 0. 1 means logic 1. x and z. z means the high impedance state and x is the state which is not known, it is an unknown state okay. So unknown and high impedance are different. So actually when you say that the state of a line is at high impedance state you know for certain that the driver of that line this has a tri state capability and that tri state control is currently disabled. So this line is really at a high impedance state but when you say it is x you really do not know what value the line is at in 0 or 1 okay. So there is a difference and for the purpose of simulation.

This is for simulation only. So when you are simulating a verilog module or verilog code then all the nets which are not connected either physically or logically. Physically means you have defined a net but there is no driver to that net or there is a driver which is currently in the tri state. So they will all be set to the z state and since the register variables have to be initialized before we actually start the process of simulation. So all register variables by default will be set to the undefined state at the beginning of the simulation. So for correct operation of the circuit it will be your responsibility to apply suitable inputs so that the register variables are set to a known set of values okay. Before you can apply meaningful inputs and get the outputs. Okay and corresponding to this 4 logic values Verilog provides a set of redefined logic gates.

(Refer Slide Time: 06:18)


©CET I.I.T. KGP

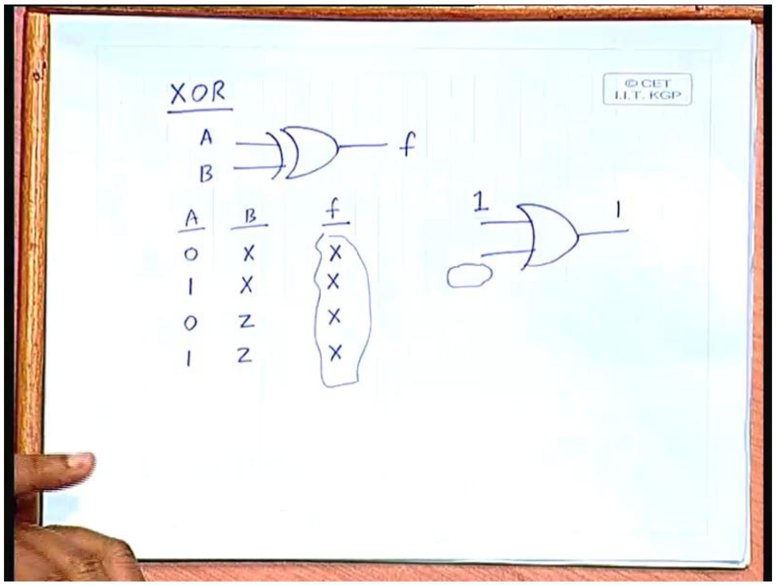
- Verilog provides a set of predefined logic gates.
  - They respond to inputs (0, 1, x, or z) in a logical way.
  - Example :: AND

| | |
|---|---|
| 0 & 0 ➔ 0 | 0 & x ➔ 0 |
| 0 & 1 ➔ 0 | 1 & z ➔ x |
| 1 & 1 ➔ 1 | z & x ➔ x |
| 1 & x ➔ x | |

AND, OR, NOT, XOR, etcetera. So we will see the list of gates but 1 thing you understand. First thing is that verilog as a language it provides this predefined logic gates as the basic primitive. So it is not something which you are including which you are adding to the feature of the language unlike VHDL. VHDL does not have any predefined logic gate by default but Verilog has. Okay. And the way the functionalities of these gates are defined that is quite consistent with these 4 logic values 0, 1, x and z. Just for example if you have an AND gate. Suppose you have a 2 input AND gate, there are 2 inputs and there is 1 output. So the way it is defined is it is very easy to understand.

You see that in the inputs are 0 and 0 obviously output will be 0. 0 and 1 any 1 is 0 output will again be 0 only when both of them are 1 then the output is 1. But if 1 of the inputs are at 1 the other is unknown the output will also be unknown. Because the output will depend on the value here. But if 1 of the input is 0 and the other is unknown then you can definitely say the output will be 0 because this is an AND gate. Similarly if 1 input is 1 and the other is a tri state value output you cannot say what it will be. So it is defined as x. Similarly z and x will remain x. So in this way you can define the values for the other types as well. I am giving another example.

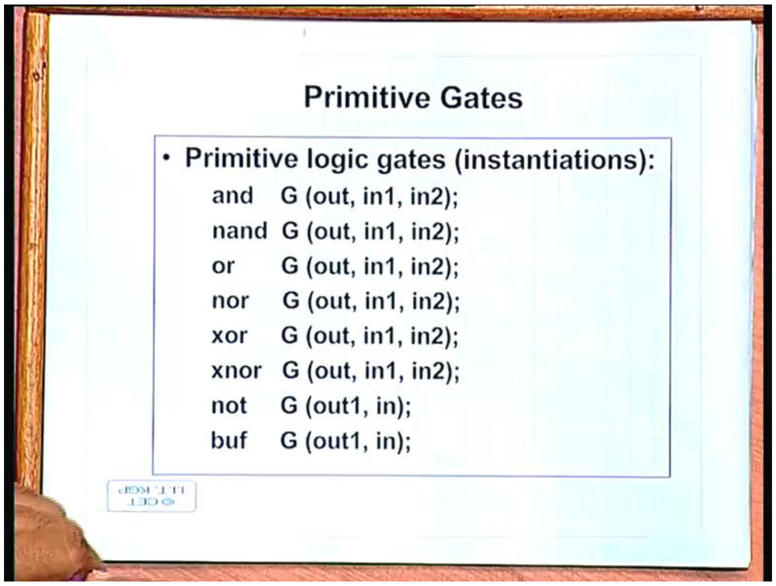
(Refer Slide Time: 08:08)



| A | B | f |
|---|---|---|
| 0 | X | X |
| 1 | X | X |
| 0 | Z | X |
| 1 | Z | X |

Suppose you have an XOR gate. Suppose you have a 2 input XOR. A, B and F. Okay. Well the normal truth table of XOR you already know. I am not showing those things. I am showing the other cases. For example if 1 of the input is zero. B is unknown then output will be unknown. If 1 input is 1 other input is unknown then output will also be unknown. If 1 input is 0 other input is a tri state. This will be unknown. 1 z, this will also be unknown. So this XOR is 1 gate where in where just in addition to the 4 normal completely specified conditions the other combination will all lead to lead to an undefined value in the output. This is unlike AND and OR gate. For example in an OR gate if 1 input is at 1 then whatever you apply in the other input the output will be 1 right. So it really depends on the gate type you are using and you can define the functionality like this. So in the language Verilog this functionality is already defined in this way so that during simulation you can get consistent behavior. Okay and the gate types which are supported are these.

(Refer Slide Time: 09:43)



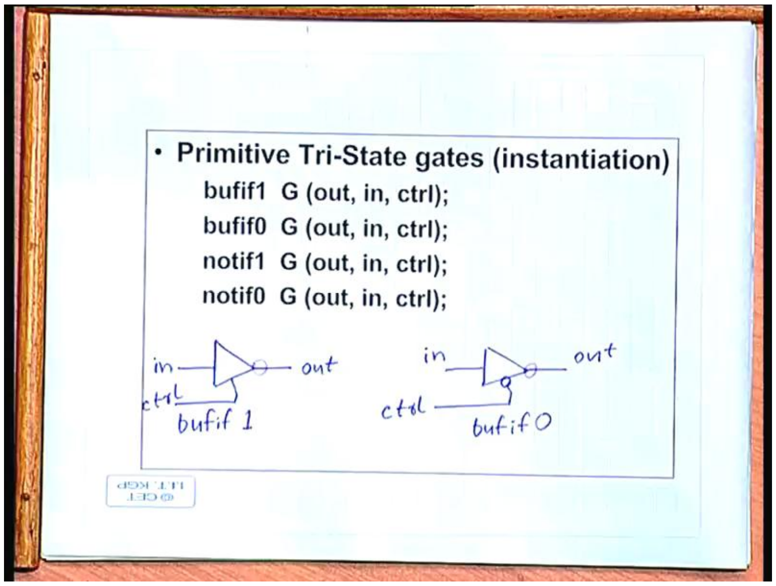
## Primitive Gates

- Primitive logic gates (instantiations):

```
and   G (out, in1, in2);
nand  G (out, in1, in2);
or    G (out, in1, in2);
nor   G (out, in1, in2);
xor   G (out, in1, in2);
xnor  G (out, in1, in2);
not   G (out1, in);
buf   G (out1, in);
```

These are the primitive logic gates which you can include in a Verilog module. So instantiations and NAND, OR, XOR, NOR, XOR, XNOR, NOT buffer. Buffer is just like a driver. It does not invert the logic value and you can see the convention is that the first parameter is the output. The next parameters are the input. For example in the first case the inputs will be in 1 and in two. The output will be out right. So you can have all the basic gate types available as part of the language. These are the normal gates where the outputs are not tri stated and in addition you can have a set of tri state buffers also which you can use in definition.

(Refer Slide Time: 10:42)



- Primitive Tri-State gates (instantiation)

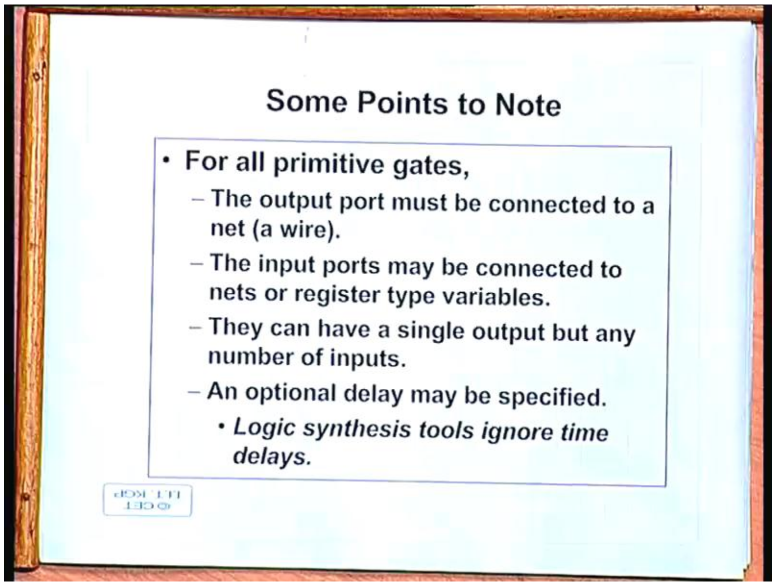
```
bufif1  G (out, in, ctrl);
bufif0  G (out, in, ctrl);
notif1  G (out, in, ctrl);
notif0  G (out, in, ctrl);
```

in
out
ctrl
bufif 1
in
out
ctrl
bufif 0

There are 4 types. Buffer if 1 buffer if 0 not if 1 not if zero. The meaning is like this. In the first 1 this is buf if one, this says that the input of this buffer is in. This is a non-inverting buffer. So there is no inversion. The output is out and it will be enabled if the control is at one. So it will be like this control. But in the second case buf if 0, this equation will be like this in out. In control you will have an inversion output. This is buf if 0. The last 2 cases are similar. Just instead of a buffer there is an inversion there. These are actually inverters. Right?

So these are the primitive gates which are available to you. You can use them in your design. You can instantiate them. You can create any netlist of these. Okay. (Student Noise Time: 11:57) Control by definition must not be high. If it is in the highness state the output will be x. (Student Noise Time: 12:05). Okay if it is in the other state also we have to try state it. Just to enable it you will have to make it 1 or 0 and if it is in the other state output will be z. So these 4 gate types, these all have output tri state control. Okay fine. Now let us note a few points.

(Refer Slide Time: 12:38)



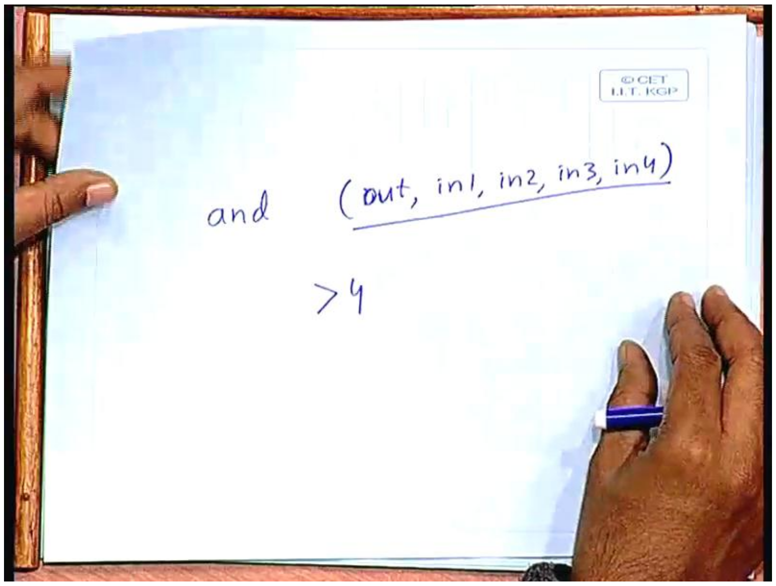
## Some Points to Note

- For all primitive gates,
  - The output port must be connected to a net (a wire).
  - The input ports may be connected to nets or register type variables.
  - They can have a single output but any number of inputs.
  - An optional delay may be specified.
    - *Logic synthesis tools ignore time delays.*

See here for all the primitive gates that we have just now seen the output of the gate that must be connected to a net. That is how we had given a few examples earlier that whenever you have a gate that gate must drive some net which may be an output or which may drive another gate. Now the input ports for the for each gate the input port the inputs which are coming that can again be nets which are coming from the outputs of some other gates or they can be register type variables which are possibly coming from some register or flip flop or latches. So the input ports to a gate can be of net type or it can also be of register type but 1 thing you remember.

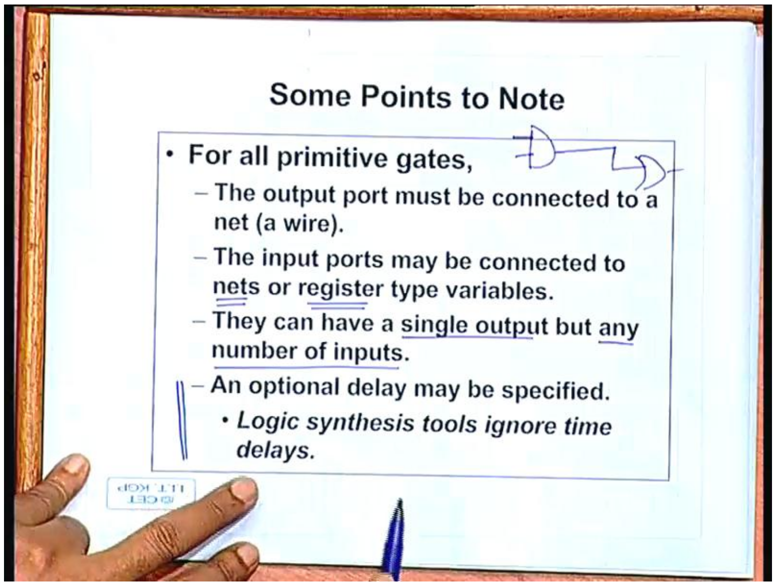
A register type variable does not always mean that it will be synthesized into a register. Well it is most likely to be synthesized in register but in some cases as we shall see that the translator or the synthesizer can do some optimization for the register can be done away with it. This is not required. Okay fine. So this primitive gate as I have said they have a single output by any number of input. See in the examples I have we have seen like here. The gates all of them have 2 inputs.

(Refer Slide Time: 14:17)



But you can also have a gate declaration say and which will have sorry. But an output and for example there are 4 inputs. So you can use any number of parameters involved. Verilog does not put any restriction to that but to be realistic you should limit the number of inputs to a gate. Typically we do not use gate fan in greater than 4. Because if you try to use gate fan in more than 4; the delay of the gate goes on increasing. So that leads to a problem.

(Refer Slide Time: 14:57)



## Some Points to Note

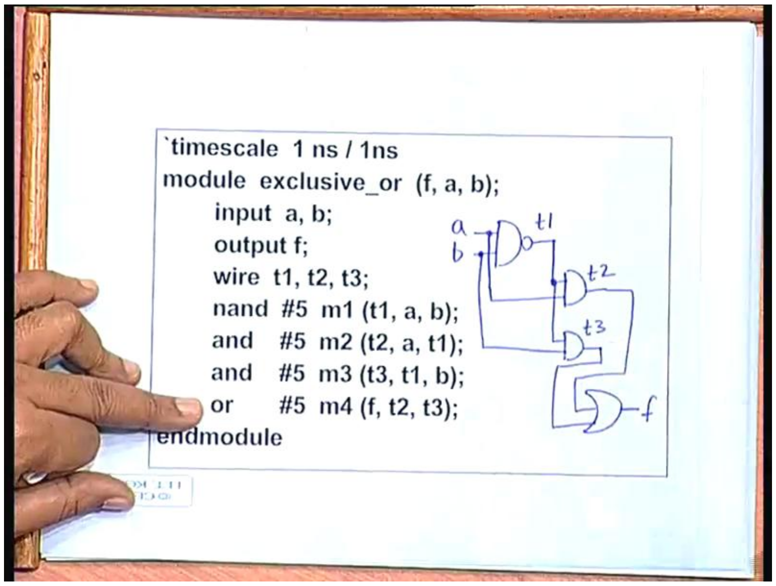
- For all primitive gates,
  - The output port must be connected to a net (a wire).
  - The input ports may be connected to nets or register type variables.
  - They can have a single output but any number of inputs.
  - An optional delay may be specified.
    - *Logic synthesis tools ignore time delays.*

So the primitive gates as we use them in our design they will obviously have a single output. But they can have any number of inputs and as we shall see this is again used solely for the purpose of simulation not for synthesis that you can specify a delay with each gate or with each component to use in your design. See once you have this delay specified then you can interpret your result of simulation that after this much time the output should change state. You can see the timing diagram for example and see that it is really changing after that time right. So this delay specification is used only for simulation and when you are doing or carrying out a synthesis. Synthesis simply ignores those delay specification commands because synthesis is carried out by you can say a software tool which will be targeting some target libraries which is already there.

Now the libraries contain some gates or other components which whose delays are already known pre specified. So user cannot give the delay. Suppose you are trying to target a design set of CMOS technology. Now in CMOS technology in the library already the gates will be available as complete layouts. So once the gates are available in that form the exact physical and electrical characteristic of those gates are already known. So the delays are already known. You cannot specify anything. So as soon as you pick up that gate from that library the delay gets specified automatically. Okay. Yes. (Student Noise Time: 16:48) Yes. (Student Noise Time: 16:52) No, no output ports of a gate I mean output port mean the output of a gate. (Student Noise Time: 16:59)

No, no. As I said that by default is the register but the translator can sometimes replace a register by a wire if the register is really not needed. So unless you have some clocking there is no need to hold the value in a register and then forward it okay. So unless you have explicit clocking this will be translated into a net but otherwise it will be a register. That is what I am saying. I have not talked anything about clock as yet. That is why I am making the statement. Okay now let us give an example where you can specify these delays.

(Refer Slide Time: 17:45)



```
`timescale 1 ns / 1ns
module exclusive_or (f, a, b);
     input a, b;
     output f;
     wire t1, t2, t3;
     nand #5 m1 (t1, a, b);
     and  #5 m2 (t2, a, t1);
     and  #5 m3 (t3, t1, b);
     or   #5 m4 (f, t2, t3);
endmodule
```

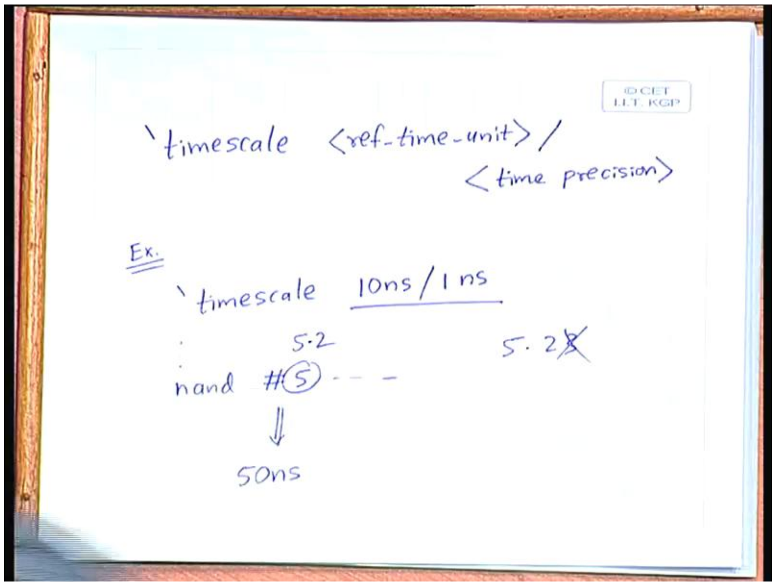
Okay. This is a very simple exam of a Verilog module. Just ignore the first line for the time being. I will come to this first line. This is a module which realizes the exclusive or function as connection of NAND, AND and OR gates. So if you look at this netlist there is a NAND gate which takes a, and b as the inputs, the output is t 1. There is another and gate which takes a, and t 1 okay. So there is and gate. 1 input is t 1, other input is a. This gives t 2. There is another AND gate which takes t 1 and b; t 1 and b. this is t 3 and there is an OR gate which takes this t 3 and t

2 and generates f. Now means if you evaluate you will find f is nothing but the exclusive or function of a, and b.

Just you have implemented in this way. Okay. Now otherwise the specification we find f a b are the parameters, a b are the input, f is the output. T 1, t 2, t 3 at the internal signals which have been declared as wires. But you observe here that we have specified some number just alongside these modules were instantiating hash followed by a constant value. Now this constant value indicates the delay. Delay in some time units. So this hash 5 actually means 5 time units of delay. Okay. So well you can specify the same time unit or you can specify you can specify any variable time unit for the different modules you are instantiating for the different basic components you are instantiating. You can have 1, 2, 3; anything you can give. Fine. Now this numbers you specify are just numbers and that number will get multiplied by some basic time scale you can say in order to get the actual time.

Now that actual time you specify by the first line. Now again this first line is used only for simulation purpose for synthesis this does not make any meaning; does not make any sense. See reverse code timescale 1 nanosecond slash 1 nanosecond. This is a typical statement. Now let us see what this really means. (Student Noise Time: 20:56). If is a register type you really do not know because this module f might be going to another gate in another module. In that case you really do not need a storage element here. We have not explicitly specified, it depends on the context in which you are using that whether you have mapped. It can also be register it can also be a net. It depends on the context you are using. Yes true. (Student Noise Time: 21:27) Timescale is a Verilog keyword. Yes now let us explain what this. (Student Noise Time: 21:33) Reverse slope, it has to be used with the reverse slope or it has to be used with the reverse slope. Yes. So the all these are basically simulated directive. The simulated directives all begin with a reverse code. Okay. So let us see that what this means, the reverse slash timescale.

(Refer Slide Time: 17:45)



`timescale <ref_time-unit> / <time precision>

Ex.

`timescale 10ns / 1 ns

5.2 5. 2~~3~~

nand #(5) - - -

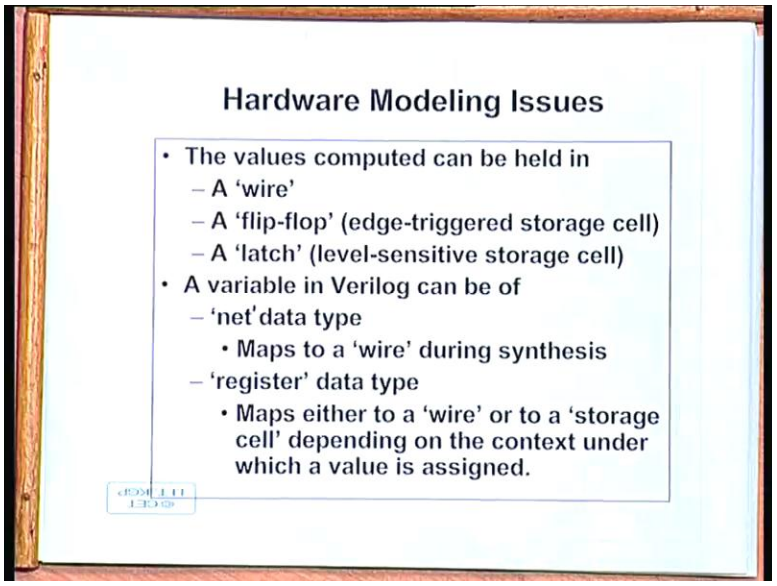
⇓

50ns

The general syntax is reverse slash timescale then you specify a reference time unit, then a slash. Then you specify something called time precision. Just an example; suppose I specify reverse slash timescale. Say 10 nanoseconds slash 1 nanosecond. This will actually mean that say here you again look at this. You are specifying delays like this. Suppose in a specification somewhere I give NAND hash 5 then something. Now this hash 5 will actually mean. 5 multiplied by 10 nanoseconds. This will actually mean 50 nanoseconds. But the resolution with which I can specify the delay is this. I can specify 5.2 also. If I specify 5.23, then this 3 will be ignored. This, the first number represents the reference time unit with which the specified numbers will get multiplied.

The second 1 represents up to how many decimal places or means what is the precision with which you can specify the time. Okay. Now in this example we have given both of this to be same 1 nanoseconds and 1 nanoseconds which means that we will be specifying only integer values. Okay. So the purpose of the timescale command is this. (Student Noise Time: 23:53) See the ratio ten divide by one. It is 10. That means we can represent up to 1 decimal place. But if it is 100 picoseconds for example, 10 nanoseconds slash 100 picoseconds then you can represent up to 2 decimal places. This 10 picosecond you can represent up to 3 decimal places. Well now let us come to some of the hardware modeling issues.

(Refer Slide Time: 24:25)



## Hardware Modeling Issues

- The values computed can be held in
  - A 'wire'
  - A 'flip-flop' (edge-triggered storage cell)
  - A 'latch' (level-sensitive storage cell)
- A variable in Verilog can be of
  - 'net' data type
    - Maps to a 'wire' during synthesis
  - 'register' data type
    - Maps either to a 'wire' or to a 'storage cell' depending on the context under which a value is assigned.

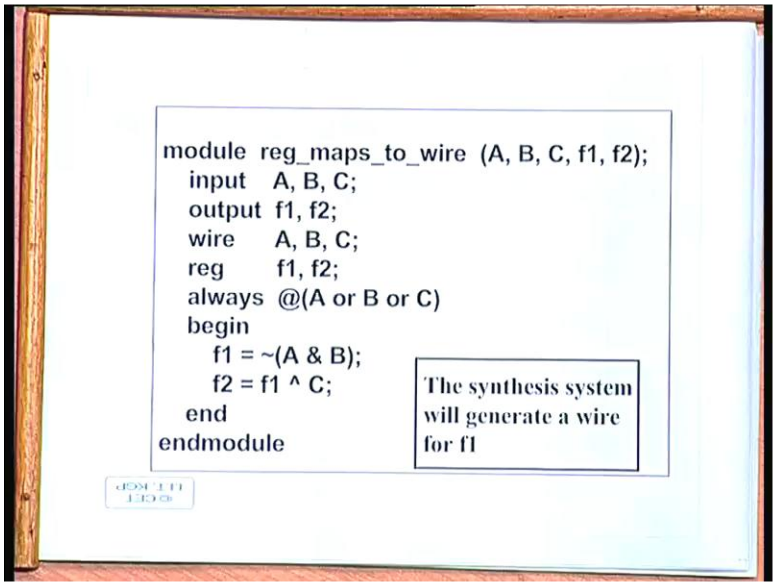
Now here we have seen some examples. Examples of some modules. We have seen that we can calculate values in some statements that can be an assigned statement or in 1 example we have seen that we can use something like an always block. Inside the always block you can also do some computation we had seen that count equal to count plus 1 something. So we are computing some value. Now the after computation the value must be held somewhere okay. Now in Verilog depending on the types of the variables and the way you are using them the values computed can be stored temporarily I mean either in a wire or in a flip flop or in a latch. This wire is the typical scenario when you are modeling pure combination logic. There is no storage element. The value is temporarily stored in our, well see here we say it is temporarily stored solely from the purpose of simulation.

But in the actual hardware implementation you are really not storing anything in the wire. It is where the signal is propagating from 1 gate to the other. But when you are simulating your time will be advancing by units of time which may be equal to the gate delays. So after this much gate delay the output of the gate will get a value. That value will be retained logically till the next time unit comes. So in that sense we say that the value is stored or retained okay. But actually it is not stored in any register or any storage and if you want to really store the value somewhere you are not need it now. You will need it sometime later. You need to store the value; you need to have either a flip flop or a latch.

Well the difference between flip flop and latch you know. The flip flop is triggered by the edge of a clock and latch is typically enabled by the level of the signal. If it is low or high the latch is enabled otherwise it is not enabled. Now you recall that a variable declaration in Verilog whenever you define a variable it can be either of the net data type or the register data type. Now looking from the point of view of the hardware implementation of these, here, whenever you define a net or a variable of type net. This will invariably map into a wire. This wire is not the wire data type, I mean a physical wire. They will be mapped into an interconnection; simple interconnection. Interconnecting wire, there will be no storage but as I mentioned register data type whenever you are using it the after synthesis.

It can map either into a storage cell or it can also map into an interconnecting wire. This depends on the context you are using, we shall be looking at some examples later we will see the (Student Noise Time: 27:39) Yeah. Some kind of optimization is done. The translator can easily find out that you really do not need to store this value somewhere. (Student Noise Time: 27:49) It will use a while in that case. But if it cannot if it finds out that the value has to be retained for certain input combination that means it will have to store it somewhere. So in that case it will map a synthesizer to a storage cell okay. So it really depends on the context how you are using the values. Let us take some examples. Let us see some examples.

(Refer Slide Time: 28:22)



```
module  reg_maps_to_wire  (A, B, C, f1, f2);
   input    A, B, C;
   output  f1, f2;
   wire      A, B, C;
   reg        f1, f2;
   always  @(A or B or C)
   begin
      f1 = ~(A & B);
      f2 = f1 ^ C;
   end
endmodule
```

The synthesis system will generate a wire for f1

This is a simple example which shows that how a register type declaration can map to a wire. See the first thing is that that any signal you can define as a register type variable if you want and here one thing you must be thinking that sometimes we using assign sometimes we are using this always. Well we shall be clarifying these differences later but (Student Noise Time: 28:52) equal to is assignment. But there are some differences. We shall be looking into the of this a little later but 1 thing you just remember. In the always block the variables you are assigning to they must be register type variables. But in contrast in an assigned statement the variables you are assigning values to they are net type variables. This is 1 difference. Okay.

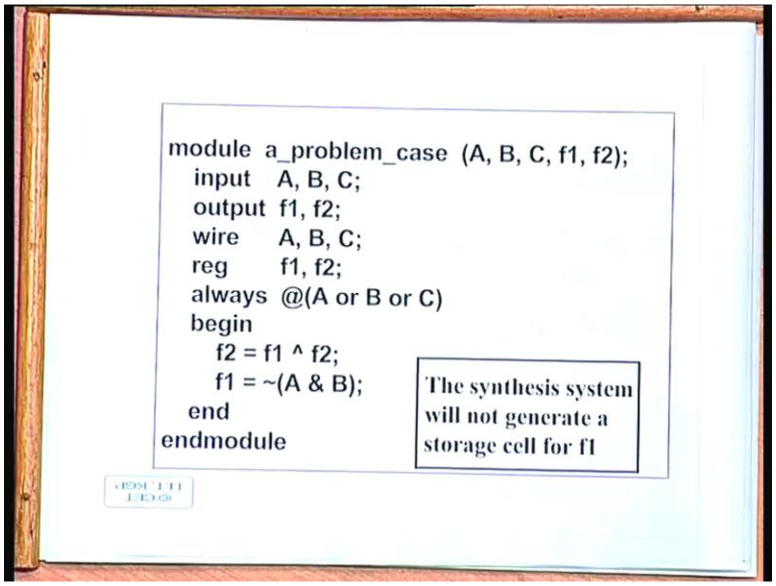
So you look at this description. This is just a dummy example if it does not do anything meaningful. A, B, C, f1, f2 are the parameters. A, B, C at the inputs. F1, f2 are the outputs and we have defined A, B, C as wires explicitly and f1, f2 as register explicitly. This statement always at the rate at A or B or C. This statement actually means is that the block that is inside always that will be evaluated if this expression changes value, not true. It changes value. That means either A changes or B changes or C changes that means there is an event. This is some kind of an expression I am specifying. This is sometimes called an event expression. So therefore whenever there is an event out here, the block gets evaluated. You recall in the example we gave earlier. Here in the event expression we had written at pos edge clock. Pos edge clock is the event. There is a positive edge in the clock then we evaluate it. But here there is no clock.

So here we are perhaps talking only about combination logic but we are saying that if A, or B or C change only then we evaluate. But 1 thing you again understand. This kind of statement I am making this can possibly make sense only to the simulator. But when you have hardware implementation, for example you have this is AND and NOT. It is a NAND gate. So you have a NAND gate with the inputs A and B. This is XOR. This is f1, this gets into an XOR gate. The other input is C and you get the output f2. But once you have hardware, implementation you really cannot tell the hardware. That means only when A, B or C changes you evaluate. The evaluation will be continuous.

There is no particular time or particular point in which you say well you can argue that the internal values cannot change unless 1 of the input changes. That is true. But it is a continuously driven thing. Just you cannot say that you start the evaluation. There will be a continuous evaluation which is going on if it is a combination circuit. Now the synthesizer can look at it and find out that well number 1 there is no clock involved. There is no triggering by an edge. It says whenever the value changes you evaluate and inside it says that well there is a data dependency between these 2 statements. So it will possibly synthesize it like this and it will also find out that you really do not need to store or hold f 1 in a latch or a flip flop.

This can be implemented as a simple interconnecting wire right. So this is 1 simple example where this reg type variable can also map to a wire. See in fact here you are trying to model combination circuit only and the synthesizer is intelligent enough to determine this and map it into a non-storage solution okay. ((Student Noise Time: 33:04) Sir only when there are clock) not necessarily we will see some examples here. Clocks are 1 thing. If there is a clock it has to take place at the edge of the clock, there has to be a register. But there are other cases also.

(Refer Slide Time: 33:23)



```
module  a_problem_case  (A, B, C, f1, f2);
   input   A, B, C;
   output  f1, f2;
   wire    A, B, C;
   reg     f1, f2;
   always  @(A or B or C)
   begin
     f2 = f1 ^ f2;
     f1 = ~(A & B);
   end
endmodule
```

The synthesis system will not generate a storage cell for f1

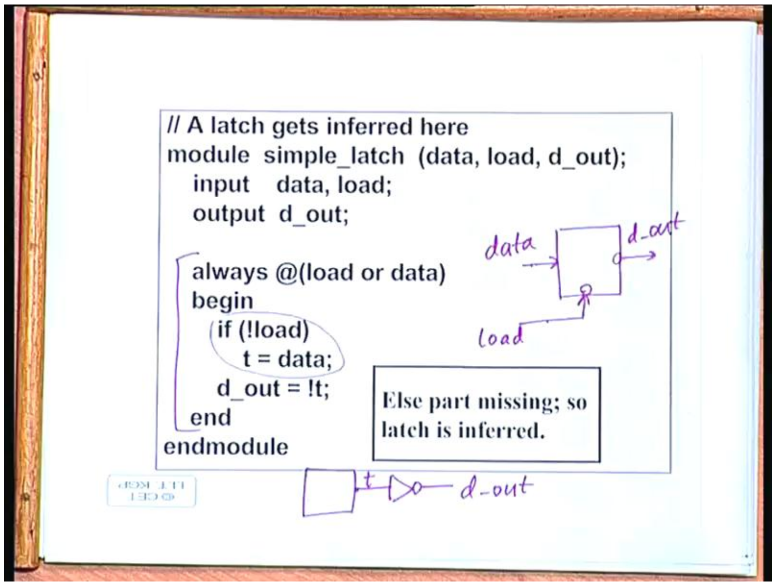
You look at these examples. This is apparently a problem case. Well this problem case why I am calling it a problem case. See the first part is the same as in the previous case. The only thing we have done we just look at the previous example. The only thing we have done we have interchanged f 1 and f 2. These 2 lines see 1 thing you understand. (Student Noise Time: 33:49) Yeah. Right. This should be sorry this should be C. This should be C sorry. Fine. So 1 thing you understand. When we are writing statements inside this block these statements are supposed to be executing confidently. This is the scenario. But when the user is writing the statements well perhaps it is playing on the back of my mind that the first statement when I am writing f 2 equal to f 1 XOR C.

This f 1 is the last value of f 1. Then I am calculating a new value of f 1. So when the user writes the code the user can perhaps make that mistake that well I am actually wanting the f 1 of the previous step and then I am calculating a new f 1. But Verilog will not do that. Verilog will simply carry out data dependency analysis assuming these statements are independent. And it will synthesize the same circuit as in the previous example. This will synthesize the same circuit as in the previous example with a simple interconnecting wire. But what the user was thinking if you had to implement that then you needed a latch or a register. Because in 1 step of the iteration you are calculating the value of f 1. In the next step of the iteration when 1 of the input changes, you really do not know when the input is changing.

The input may change 1 hour later. But you will have to hold the old value of f 1 till that time occurs. So actually what the user apparently thinks and what the verilog translator does there can be a conflict if you are not very careful about it. But of course if you know how the translator works and how the translator will deal with this situation then it is fine. But there is chance of making mistake here. Well (Student Noise Time: 36:06) Old value f 1 we have not used. Yes you see. This block will be executed whenever 1 of the inputs change. So what I am saying whenever 1 of the input changes it is not the old value of f 1 which is used to calculate f 2. It is this value of f 1 which is used always okay. (Student Noise Time: 36:36) F2 will always use the new value of f 1. Because here the semantic says that inside a beginning block all the statements are executing concurrently. No concurrently means again I am saying again I am saying that this is getting synthesized into a hardware circuit like this. So concurrently means you can say that the, that f 1 is getting a new value here f 2 is getting a new value here at some point in time. But after a time equal to the gate delays the value of f 2 will again change because this value has changed.

Yes so it is not only depending on A or B or C. Depending on the gate delays there will be a chain of changes which goes on. Yes, yes, yes. If you require that we use the old value you would have explicitly specified in a different way. That we will see how we will specify that. (Student Noise Time: 37:57) Because the chain of the value of f 1 will depend… Chain of value of, f 2 will depend on delays of the gates as it goes on. So that you simulate this you will see in the timing diagram that after some delay the values you are getting changed or computed. Yes. Well now let us look an example where a latch or a flip flop gets explicitly inferred synthesized.

(Refer Slide Time: 38:19)



```
// A latch gets inferred here
module simple_latch (data, load, d_out);
   input   data, load;
   output  d_out;

   always @(load or data)
   begin
     if (!load)
        t = data;
     d_out = !t;
   end
endmodule
```

Else part missing; so latch is inferred.

data
d_out
load
t
d-out

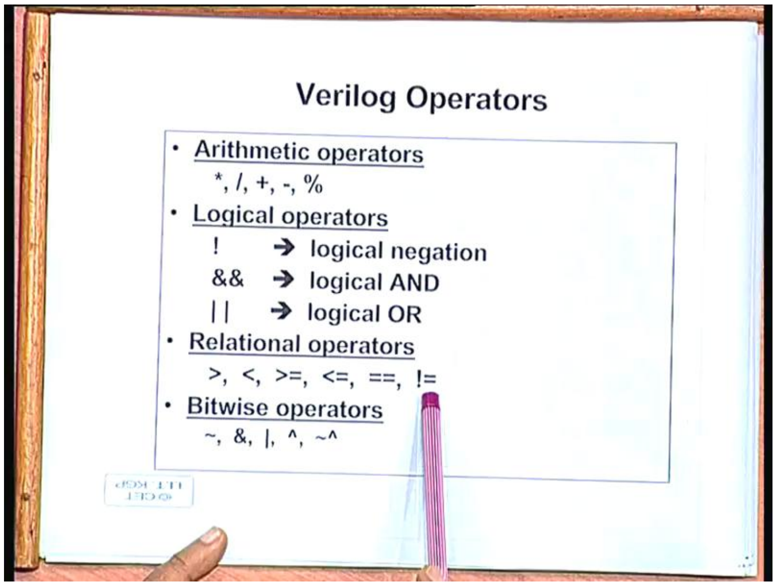
Double slash means commenting Verilog. This you can give anywhere. So this is the specification of a simple latch where you have a data input. You have a data output and you have a load signal. Data and load are inputs. D out is an output. Now if you look at the specification this also uses an always. It says always load or data. So unless data or load changes there is no need to make any computation. So this computation will take place whenever either data changes or load changes. Now you look here. It says if not load. This is this is active load here this is active load. If load is 0 then data gets into t and d out equal to not well. Here there is an inversion out here.

And the output also gets inverted. It says if the load is 0 then the data gets stored into a t. So this actually if you draw the diagram in a different way the latch output is d and after t there is an inverter. That is your d out. So if load is 0 the data gets latched into t. After that not of t gets assigned to d out. But if load is not 0 then whatever the value t this it will not be executed. Then whatever the value of t was there not of t will go out. Right? So if the load is not active just the data changes there will be no change in the output. Now you look at this code from a different angle. See there is an, if statement but we have not specified the actions for all the branches of the, if statement.

We have said what to do if the if condition is satisfied is true. But we have not specified what is to be done if the if condition is false. That means the else part of the, if statement is not specified. Well it can mean. (Student Noise Time: 41:07). Yeah but 1 thing you just try to understand. This can be an, if statement. This can be a case statement whatever. This is a multi-way branch kind of a statement where all the conditions are not specified but here you are calculating some value t which is get assigned in some other statement outside this if. So if all the parts of the, if statement were specified, then this could have been mapped into a combination circuit.

Since you have not specified what will be the value of t if load is not zero. This means you will have to remember the old value of t because the old value of t will get assigned then. So a rough rule of thumb is if you have a incompletely specified if statement or a case statement this will imply a latch. Okay you need to store the values somewhere. But if the if statement was complete even in the else part you had written some t equal to something say then you explicitly know that you check this if it is true t is this then t is this then you go here. So it will be pure combination logic. There is no concept of remembering the old value. So if you have an incompletely specified if statement it can be a case statement also. You will see these examples later in more detail. Then we will infer a latch, t should be a wire. I forgot to do this; t should be a wire. It should be a reg t not a wire. Reg t because it is inside the always block. Right? So we shall be looking at more of these examples later so that we will see this in more detail.

(Refer Slide Time: 43:14)

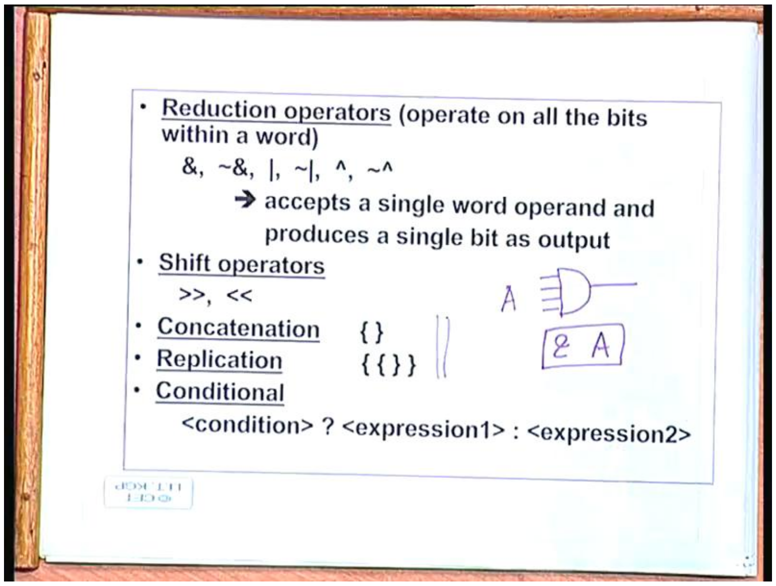


## Verilog Operators

- Arithmetic operators
  *, /, +, -, %
- Logical operators
  ! → logical negation
  && → logical AND
  || → logical OR
- Relational operators
  >, <, >=, <=, ==, !=
- Bitwise operators
  ~, &, |, ^, ~^

Now let us very quickly browse through the different operators which are there in the language. This you can use in your coding. The arithmetic operators are very much like C. Multiplication, division, addition, subtraction and this module modulus. Logical operators are also same. Negation logic AND, logical OR.  Relational operators are also same. Nothing to say here bitwise operator. Bitwise means NOT, AND, OR, XOR, XNOT. Bitwise means if you say A AND B then there will be bitwise ANDing. Say A is a 4 bit number. B is also a 4 bit number. There will be a bit by bit ANDing 0 0 1 0. This will be the result right and logical operators are used only in case of relational expressions in an if-statement. If this AND this okay. Bitwise operators are there. These are also there in C. It is similar but what is new out here is something called reduction operators.

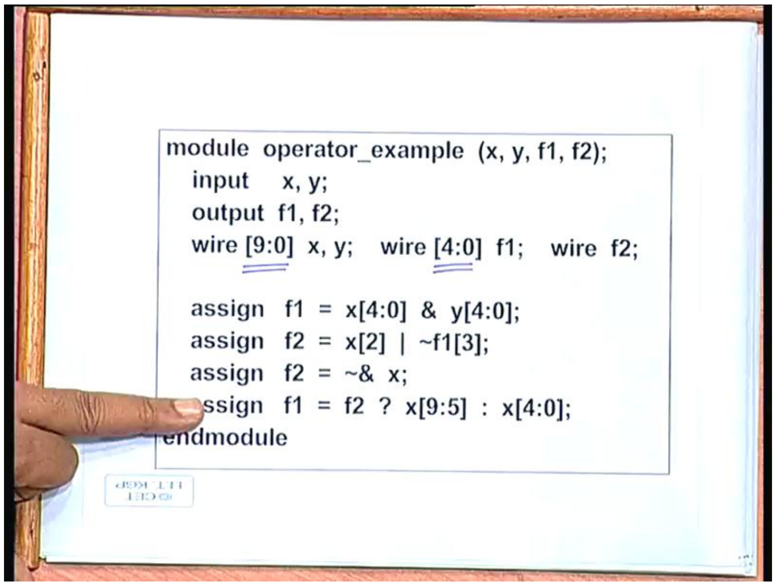
(Refer Slide Time: 44:33)



- Reduction operators (operate on all the bits within a word)
  &, ~&, |, ~|, ^, ~^
  ➔ accepts a single word operand and produces a single bit as output
- Shift operators
  >>, <<
- Concatenation { }
- Replication { { } }
- Conditional
  <condition> ? <expression1> : <expression2>

Reduction operator is something like this. Suppose you have a number A which is a 4 bit number and I want to do a ANDing of all this 4 bits to generate a single bit. This is called reduction. This operates on all the bits of the word and generates a single bit result. So it can be AND, NAND, OR, NOR, XOR, XNOT anything. So you can have this reduction operator also you can use this ampersand followed by A. This means reduction. This is how you use this. Shift operator is very similar to what you have in C. Concatenation replication we will show with examples. Concatenation means you can concatenate several different things to create a single bit stream. A single number replication some of the bit stream you can replicate. We will see with examples these 2 things. These 2 are new conditional again is available in C. Okay. Condition if true then this, false, then this. Now let us take some examples in which you can illustrate how to use these things.

(Refer Slide Time: 46:01)

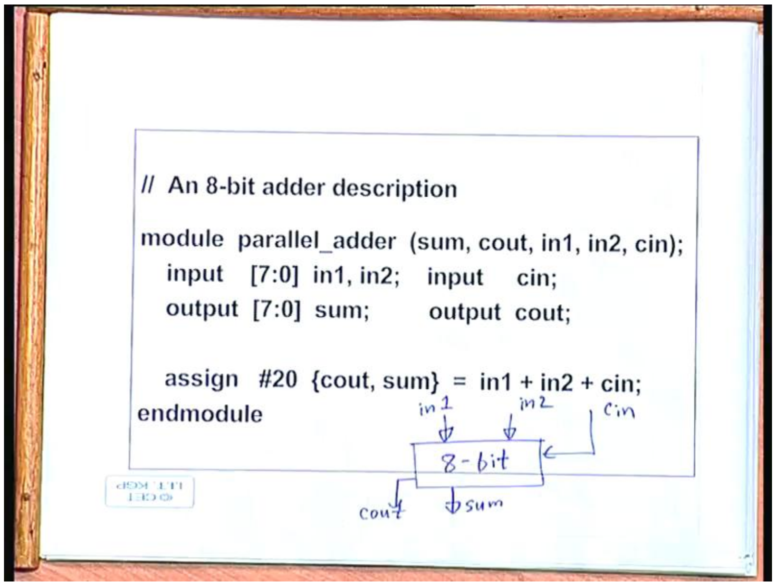


```
module operator_example (x, y, f1, f2);
  input   x, y;
  output f1, f2;
  wire [9:0] x, y;   wire [4:0] f1;   wire f2;

  assign  f1 = x[4:0] & y[4:0];
  assign  f2 = x[2] | ~f1[3];
  assign  f2 = ~& x;
  assign  f1 = f2 ? x[9:5] : x[4:0];
endmodule
```

This is 1 simple example which illustrates some of the operators and also some other concepts. You see this is an example which takes 2 inputs x and y and 2 outputs f 1 and f 2 x and y are both ten bit quantities. These are of type wire f1 is 5 bits and f2 is a single bit right. There are 4 assign statements. The first assign statement says that you extract a subset of a number x. See x is a vector whose indices range from 0 to 9. So here when I say x4 equivalent 0, I am extracting the least significant bits of that. Similarly here I am extracting the least significant 5 bits of y. I am doing a bit by bit handing of these two that I am assigning to f 1. These kinds of things I can do. Even if I have a big sized vector I can extract any subset of bits from there and I can use it in an expression.

Similarly I can use this thing. F 2 equal to x 2 a particular bit of this x OR NOT f 1 3. A particular bit of this, so I invert it and I take a OR. This is a reduction operation. X is a 10 bit vector. I take the NAND of that bit by bit NAND. All bits I take a NAND operation. Okay. These are if and else if f 2 is true then you take these 5 bits else you take these 5 bits. This we assign to f 1. Okay. So these are the things you can do in Verilog which well if you recall in C you cannot do these things. You cannot extract any arbitrary subset. Of course there are other languages which are predecessor which are predecessors of C like your there is a language called P L 1. There it was possible to do all these things. But in C you cannot do.

(Refer Slide Time: 48:20)



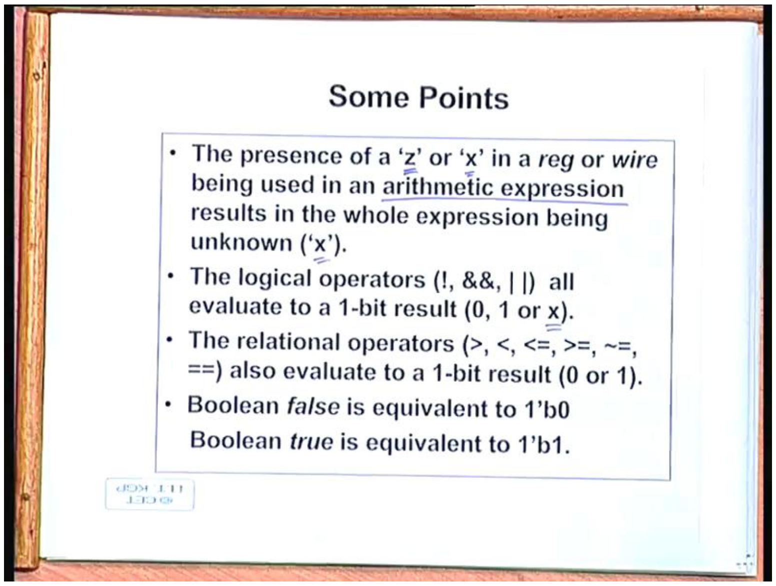
```
// An 8-bit adder description

module parallel_adder (sum, cout, in1, in2, cin);
   input  [7:0] in1, in2;   input   cin;
   output [7:0] sum;        output  cout;

   assign #20 {cout, sum} = in1 + in2 + cin;
endmodule
```

in1
in2
cin
8-bit
cout
sum

And you can also have some kind of a vector arithmetic operation as this example illustrates. See here we are trying to model a parallel adder parallel adder says that we have 1 input vector. This is a vector in 1. There is another input vector in 2. There is a carry in and in the output you have sum. This is also a vector and we have a carry out. This is an adder we want to model and we are wanting or requiring an 8 bit adder. So in 1 and in 2 we have declared as 8 bit quantities. Sum is also 8 bits. C in and c 2 are single bit quantities. Now you look at this line in 1 plus in 2 plus carry in. This will be the result.

Result will go where result will go into the concatenation of c out and sum. So even while assigning I can specify concatenation. This will be an 8 plus 1, 9 bit quantity. So the result will be computed in nine bits and that will be assigned to the combination of cout and sum. This nine bit combination okay. So this an example of using concatenation in the assignment in the left hand side and this is again for the purpose of simulation. We assumed that this addition will take twenty units of time. Right?

(Refer Slide Time: 50:12)



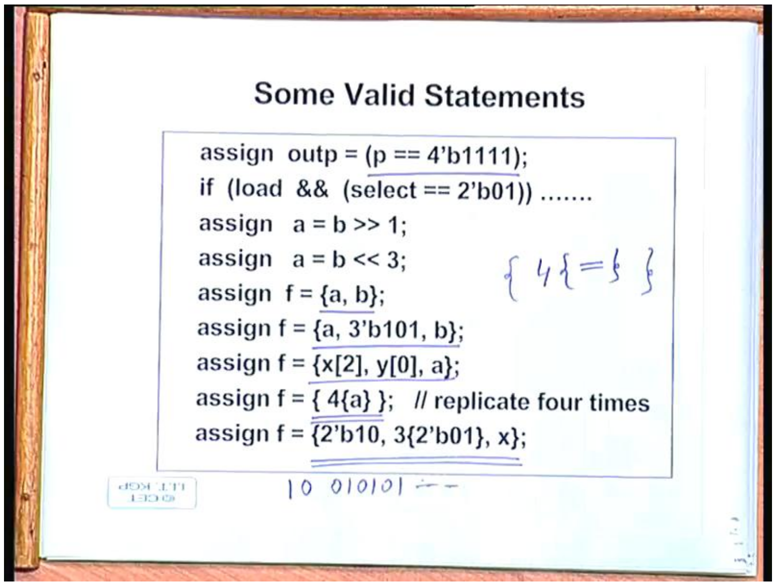
## Some Points

- The presence of a 'z' or 'x' in a *reg* or *wire* being used in an arithmetic expression results in the whole expression being unknown ('x').
- The logical operators (!, &&, | |) all evaluate to a 1-bit result (0, 1 or x).
- The relational operators (>, <, <=, >=, ~=, ==) also evaluate to a 1-bit result (0 or 1).
- Boolean *false* is equivalent to 1'b0 Boolean *true* is equivalent to 1'b1.

Now there are a few other things you should keep in mind that if in an expression you have z or x. I mean it is not a logical expression. It is an arithmetic expression. You are doing some addition subtraction multiplication division. If any 1 bit of any 1 of the operands is either at x or at z then the entire result will be x. For in case of logical expressions you have seen that we can use some logic to generate the output values but for arithmetic expression we cannot say anything. We can straightaway say it is x not known and another thing is that the logic operator or the relational operator they all evaluate to binary values 0 and 1.

Of course sometimes logical operators if the operands are x or z they can evaluate to undefined but relational operator less than greater than defined to yes or no. Boolean false is represented as 0; boolean true is represented as 1. Just like C you can use them as numbers and use in an expression you can compare if you want. (Student Noise Time: 51:27) Logical operator cannot have tri state. It cannot evaluate a tri state. Logical operator means this condition is true and this condition is true. It is not an AND gate you are modeling.

(Refer Slide Time: 51:50)



## Some Valid Statements

```
assign  outp = (p == 4'b1111);
if (load  &&  (select == 2'b01)) ......
assign   a = b >> 1;
assign   a = b << 3;
assign  f = {a, b};
assign f = {a, 3'b101, b};
assign f = {x[2], y[0], a};
assign f = { 4{a} };   // replicate four times
assign f = {2'b10, 3{2'b01}, x};
```

So just 1 last slide to show some more examples of assignments. See here you can assign do an assignment like this. This is very much like what you can do in C. Right hand side compares if p is equal to 1 1 1 1. 4 bit combination. This is a relation expression. This can evaluate to either 0 or 1. This will be assigned to output and similarly if you if you can write if statement like this. If load AND some condition, select equal to 0 1. You can make an assignment. This is right shift left shift, right shift by 1 place, left shift by 3 places, assign f equal to a b. This is concatenation. This, a b you concatenate. Then you assign the total thing here. Similarly this is another concatenation. a followed by this 3 three bit number 1 0 1 then b whatever it is that you assign to f, this is also concatenation. x2 This 1 bit, y 0 another bit and a this you just assign. This is an example of replication.

See within curly brackets if you give a number again a curly bracket within curly bracket you specify something then that something will be replicated so many times. So in this example whatever is the value of a, that will be replicated 4 times and then that will be assigned to f. This replication and concatenation you can mix up binary 1 0 followed by 3 times 0 1. So 1 0 followed by 0 1 0 1 0 1 followed by whatever is the value of x. That this thing is concatenated and the value is assigned to f. Yes. No, no. This is also a value statement see something like of course see you cannot write this. This is a relational expression. Double equal to means if p is equal to this 1 or 0. This will evaluate to a single bit value. That will be assigned to output, f has to be appropriate size. Appropriate size, yes just for example purpose I have given. F has to be an appropriate size.

It must be of the same size as the right hand side and hash 20 that is just for the sake of specifying the delay of the statement. (Student Noise Time: 54:50) This is a concatenation. I am assigning to this cout and sum. That is the cout is 1 bit and sum is 8 bit. Total is 9 bit and the result will come as 9 bits. That will get assigned to these 9 bits. So the MSB will go to cout. Now so in our next class we shall be looking at some of the Verilog description styles; slightly high level description styles which you will see interestingly. They will be mapping in terms of synthesis into very well-known hardware modules. So as designer you need to know that if I write my code like this. This will get mapped into a multiplexer. If I do it like this this will get mapped into a decoder or something like that. So we shall be looking at some of those design styles in the next class. Thank you.