Electronic Design Automation

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

Lecture No #2
Verilog: Part I

So in this lecture we would be talking about the Verilog hardware description language.

(Refer Slide Time: 01:13)

© CET
I.I.T. KGP

VERILOG
Hardware Description Language

1

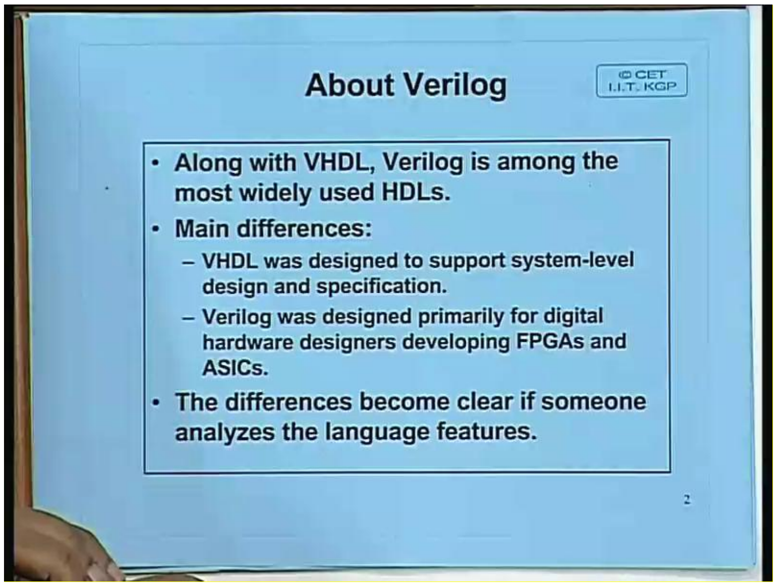
Now before I go into the syntax and the semantics because there are some features of the language which requires clear understanding of the semantics. What it really means from the point of view of the hardware design and implementation. So I would like to mention one thing. Verilog may be used by a designer in two different contexts. Number one you specify a design in this language you simulate and see that whether your specification is correct. So here you are not thinking about synthesis. Just you are specifying something in a hardware description language. You are simulating, you are looking at the timing diagram.

You are happy that the cycle is working properly but you will have to understand or remember one thing that when you are using this language in this particular mode that your objective is to simulate and verify whether simulation is correct. There you are using some features of the Verilog language which is perhaps not possible to synthesize. So what I mean to say is that the Verilog language or the instruction set or the features there are two different parts to it. Their one portion of it is said to be synthesizable. Other portion is said to be non-synthesizable. So you should clearly know that which are the features? Which can be synthesized? And which are the features which cannot be synthesized?

So if your target is to synthesize a design then you must restrict yourself to the synthesizable sub set. Okay. But otherwise you can use others other features. (Student Talking: 3:09: Why does the other part exist?) Other parts existing because in many cases you may not want to go down the synthesis. Say for example in some course work you want to just learn the language Verilog. Just you want to write some coding Verilog and simulate and see that if it is correct. See the CAD tool for synthesis is very expensive but the CAD tool for simulation is very cheap and available even there are a number of versions available on the net for free. So that in many institutions what in order to impart training in Verilog what people do they teach the language Verilog and ask the students to simulate and see that whether whatever they have written is correct or not.

Just I am giving you giving you some examples. (Student Noise Time: 04:00). The portion that cannot be synthesized can be simulated like I can give you some examples. There are some instructions to read some data from a file to print something. To specify delays that after this statement give a particular delay then do this something like an infinite loop, something like in-completive statement. So there are certain thing which in terms of semantic is fine. But when you think of the hardware implementation it becomes very difficult for the synthesizer. That is why the synthesizer does not consider those features at all. Okay. So we will see these things. Fine.

(Refer Slide Time: 04:42)



About Verilog

© CET I.I.T. KGP

- Along with VHDL, Verilog is among the most widely used HDLs.
- Main differences:
  - VHDL was designed to support system-level design and specification.
  - Verilog was designed primarily for digital hardware designers developing FPGAs and ASICs.
- The differences become clear if someone analyzes the language features.

2

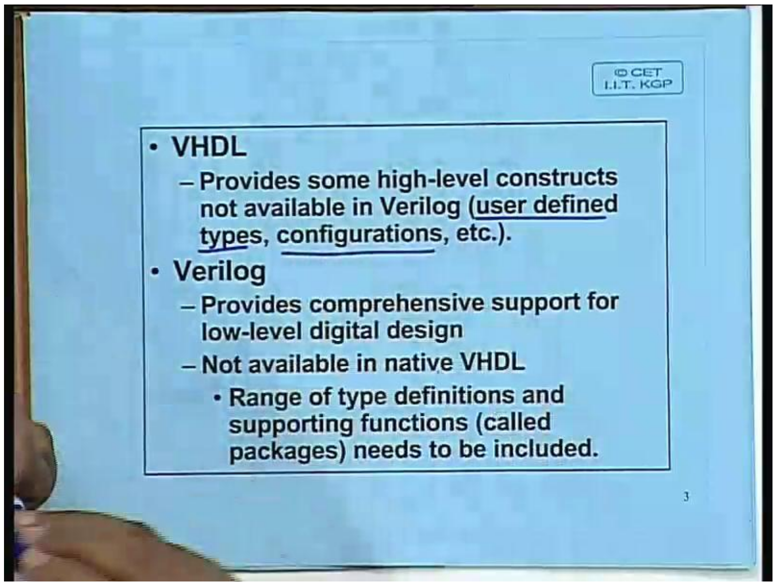
So now we would be (Student Noise Time: 04:45) given a behavior description a synthesizer will automatically synthesize it down to the gate level netlist. Yes. (Student Noise Time: 04:58) Does not worry about synthesis. There are a number of you can say flags which the user can specify whether you want to optimize or not. There are a number of features whether you want to incorporate testability or not. There are certain options which the user must specify. But otherwise up to the logic level it is fairly automatic. There is no user interference. It is fully optimized but the problem is that once you look at the optimized layout you will not understand or you mean you will not recognize your design.

Suppose your design had four modules and after optimizing all four modules are mixed together. So you cannot really identify those four modules. So a good practice is to do a partitioning at the high level itself and to synthesize the four modules separately, around the floor on the silicon you place them manually. That is a good trade off. (Student Noise Time: 05:58) Internal design is optimized but even at the level of layout you would like to see that the basic blocks remain unidentifiable. This is your CPU. This is your controlling unit but if the CPU and controlling unit are mixed together you may be little worried. If there is a (Student Noise Time: 06:14) what to do? Fine. So as I mentioned in the last lecture there are two popular hardware description languages which people very widely use nowadays.

One is of course Verilog other is VHDL. Well we also mentioned that in this course we would be considering mainly Verilog. There is some reason of course. But first let us try to understand some of the differences between VHDL and Verilog. See the language VHDL is evolved initially to support higher level specifications to support system level design and specifications. It did not support low level design like for example a gate level netlist. VHDL does not support users to specify a gate level netlist. For example here you can support system level design where the blocks are specified to specify the blocks their behavior and how they are interconnected.

But Verilog was designed from the other way round. This was designed by the designers who were developing FPGAs and ASICs who are more familiar with the low level description languages like logic equations, like circuit diagrams at the RTL level gate level. So they design Verilog from that point of view. So you will see that in Verilog it is very easy to specify designs from that level. We have already seen some examples. So the language features are different as I mentioned. So let us try to see some of the differences.

(Refer Slide Time: 08:11)


© CET
I.I.T. KGP

- VHDL
  - Provides some high-level constructs not available in Verilog (user defined types, configurations, etc.).
- Verilog
  - Provides comprehensive support for low-level digital design
  - Not available in native VHDL
    - Range of type definitions and supporting functions (called packages) needs to be included.

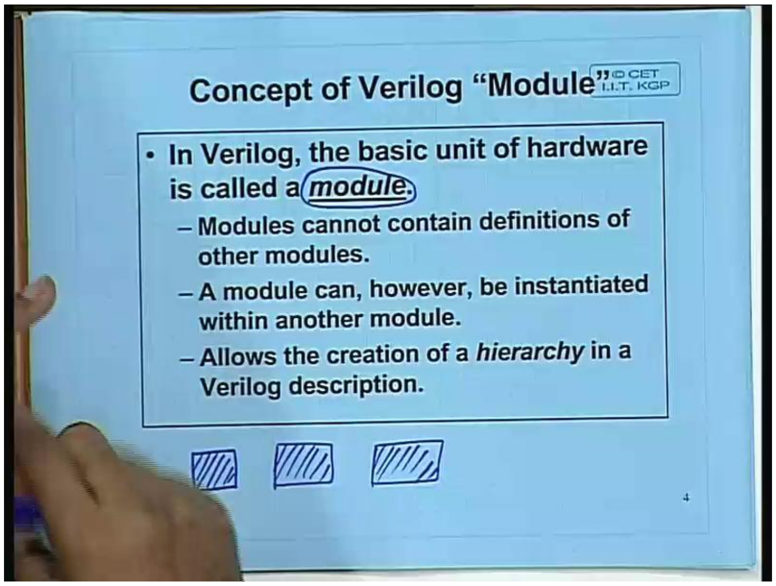
3

VHDL provides some high level constructs because I had mentioned this was primarily designed for system level designs and the high level you can have so many different kinds of requirements to specify. So there are something called user defined types user defined configurations that are possible in VHDL but in Verilog these are not there. VHDL there are no predefined types. All types have to be defined by the user like integer Boolean etcetera. These types are not that free existing but in Verilog since the primary emphasis was for low level designs all the basic data types and constructs and the basic gates they are available as primitives. So you can straightaway start using Verilog to design your circuits even from a low level.

Now in VHDL if you want to do this you cannot do it straightaway. You will have to import or include something called packages. Packages are something which someone has already written and designed for you. Packages are basically a set of modules where the behaviors of all the low level functionalities have been specified to the thread by detail. So unless you do this you cannot use VHDL, but in Verilog as I told these things are available as language primitive. They are part of the language but in VHDL the data type the basic gates the modules we use they are not part of the language. They have to be imported or included as packages okay.

So if you have the packages of course you can use VHDL then. But without packages the language VHDL is moduled. But if you have packages you can use of course. But again I am telling you Verilog is more natural for persons for logic designers who are more familiar with specifying in the terms of logic in terms of interconnection of gates etcetera. Verilog is a more natural way of expressive as compared to VHDL. That is why in this course we have chosen to select Verilog. Of course you can also use VHDL. There is nothing wrong with it. So not let us try to look at some of the language features of Verilog.

(Refer Slide Time: 10:49)



Concept of Verilog "Module"

© CET I.I.T. KGP

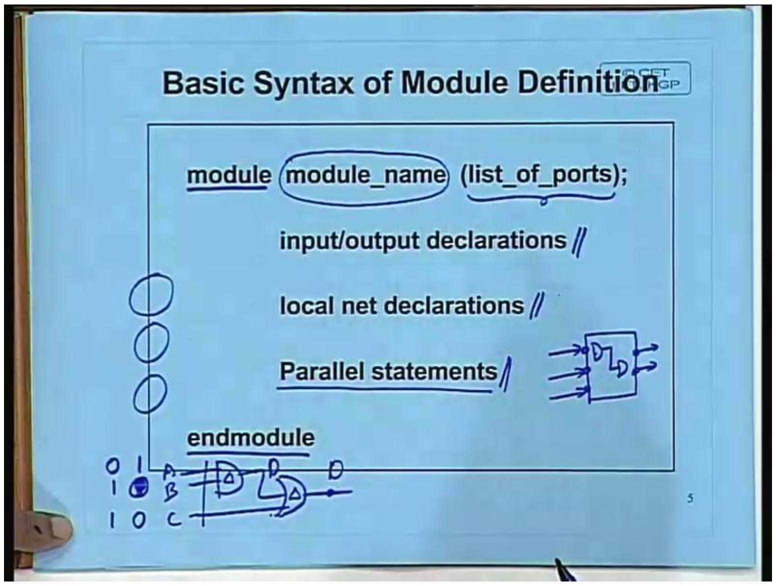
- In Verilog, the basic unit of hardware is called a *module*.
  - Modules cannot contain definitions of other modules.
  - A module can, however, be instantiated within another module.
  - Allows the creation of a *hierarchy* in a Verilog description.

4

Fine in Verilog the basic building block using which we design everything is called a module. So a module is the basic unit of hardware in terms of which we can specify the behavior or the structure whatever. So whatever we specify in Verilog that is in the form of a module. Now in a design there can be several modules. There are some restrictions within a module definition you cannot include another module definition. So a module definition within another module definition is not allowed. But what is allowed is like the example that we had seen in the last class we had seen we can instantiate. We can define a module externally.

We can include it as part of a high level module. That is what is possible and since this is possible we can have a nice hierarchy of designs. You can design it using a bottom up fashion. To define the low level modules use them to design slightly. High level modules use them to design high level modules. So using this we can have a hierarchy of design you can either proceed top down or bottom up as you wish. The language Verilog supports this. Okay. Now we have already seen how a typical module looks like but in terms of the syntax. The basic constituents of a module are these.

(Refer Slide Time: 12:37)



Basic Syntax of Module Definition

```
module module_name (list_of_ports);
        input/output declarations
        local net declarations
        Parallel statements
endmodule
```

So a module definition we start with the keyword module and end with a keyword end module. This is the module name. You have to give a name to each module and here you will have to list all parameters. See if you treat a module as a black box there will be some inputs to this module. There will be some outputs from this module. These are sometimes called ports. So you will have to list all the input and output ports out here.

Next you will have to specify that which of these ports are input; which of these ports are output. Well you can also specify something called bi directional in input and output port. So there are some input output declaration lines out here. So you will have to specify which of these are input parameters; which of these are output parameters and what are their data types.

They are single bit or vector or what and as I mentioned also in the last class there can be some interconnection which is internal. Say inside a module you can have a gate here you can have a gate here. There can be an interconnecting line in between. But this line is neither present in the input port list or the output port list. But when you specify this circuit in a structural form you will have to give some name here. That will come here. Local net declarations nets means interconnecting wires. So this interconnecting wires which are internal to these module this will come after this. Then the actual specification of your block.

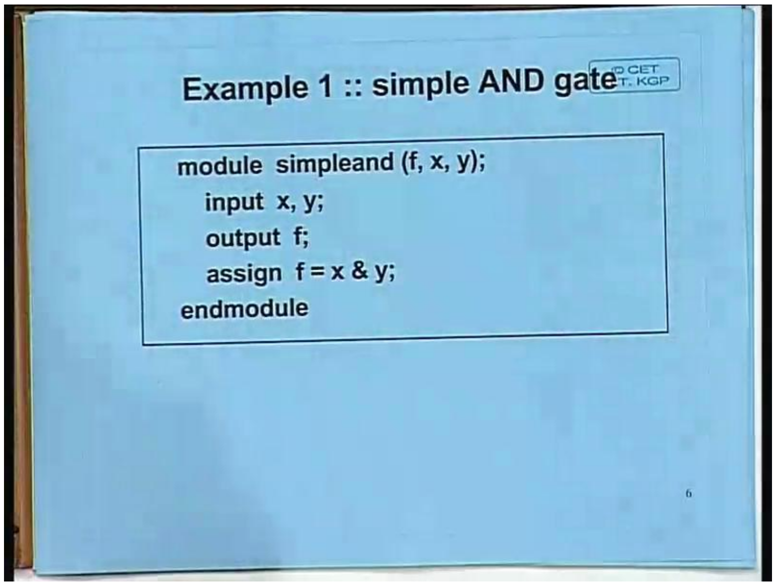
Well here I will say that these are parallel statement because here you can specify a number of statements and you understand in Verilog we are trying to specify hardware and each of these statements will be getting mapped into some hardware blocks. Now if you have several hardware blocks computation will obviously go on in parallel. Well unless they are interconnected in cascade or there is some clock or something which is driving them. So if there is no such thing all these computations will go on in parallel. If there is an AND gate if there is an OR gate they will evaluate together. Okay. If they are in cascade for example if there is an AND gate if there is an OR gate they will evaluate in parallel.

But the output of the OR gate will get the valid value after one unit of time. Suppose these inputs change state. These are the inputs A, B and C. These inputs are changing. So the output of the AND gate will change first and after a delay the output of the OR gate will change. So the gate delays will take care of that. Verilog take care of the delay. You can explicitly specify delays or when you are picking up some modules from the library or predefined or otherwise there are some pre assigned delays to the gates. If you are having a behavioral description, (Student Noise Time: 15:52) Well if you are having a behavioral description of the gate you can explicitly specify the delay in terms of a number.

You can specify this block as a delay of 10. This block as delay of 5. There are ways of doing that. We will see it later. (Student Noise Time: 15:52) A, B, C you are saying suppose A B C the values were says 0, 1 and 1. Now they change to say 1, 0 and no 1, 1 and 0. Now what I am saying that, these values are changing together. Now just assume each gate has a delay of delta. So after a time delta the output of the AND gate will change to one. It was 0 earlier. AND gate was 0 earlier. Now if you change to one. Now this value was one already so this output of OR gate is still one. Sorry this is not a good example of taking 0 1 0 right. Making it 1 0. Fine. So if 1 0 then both will be 1 because this will be 0. Okay but earlier this was one. So this OR gate was evaluated at this time with the older value so the output will be 1.

So only after time twice delta this new value will be used to simulate. Now it will take 0 0 the output will become 0. What I am saying is that (Student Noise Time: 17:47) Delay can be different because when the simulation is actually carried out it is an event driven simulation depend on the delays of the gates. There will be an event queue which is maintained. The gate which is changing state earliest will be at the head of the key. That will be a priority key that can be list. We will be talking about this later. Simulation is an important part. Yes. Well at the logic level at the pre layout simulation phase interconnect delays are assumed to be 0. They are all ideally connected. Only at the level of post layout simulation you can take care of that. That we will ignore. Fine. So now let us again start by looking at very simple examples.

(Refer Slide Time: 18:45)

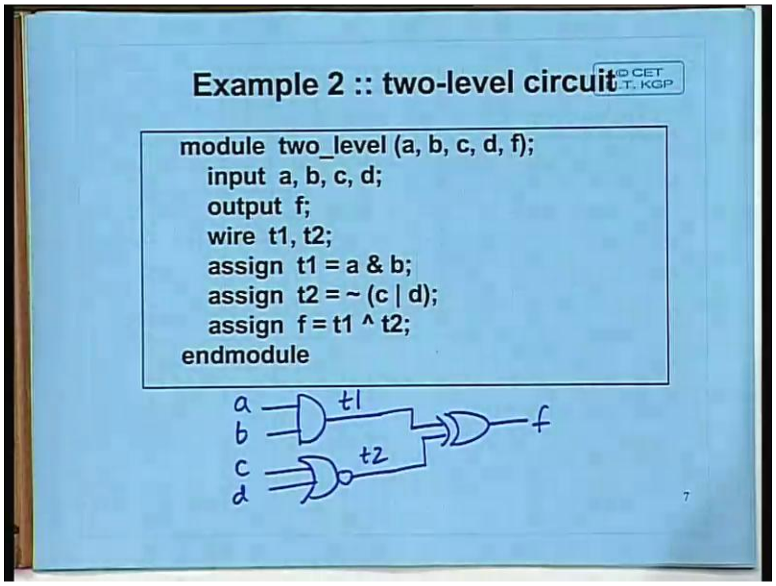


Example 1 :: simple AND gate

```
module  simpleand (f, x, y);
   input  x, y;
   output  f;
   assign  f = x & y;
endmodule
```

Start with a simple AND gate which we are specifying in terms of the behavior. So just to correlate with the previous example module. In module, this is the name of the module. These are the parameters. Two of these I am specifying as input. One I am specifying as output and assign is a keyword. This assign says that we evaluate the right hand side and assign it to the variable f. So this will typically be synthesized by an AND gate x y the output will be driving f. Okay simple. Now assign we will have to write something otherwise it will give an error syntax error. There are different ways of doing it. I will. This assign is one way of making assignment. There are other ways also. We will see it later.

(Refer Slide Time: 19:45)



Example 2 :: two-level circuit

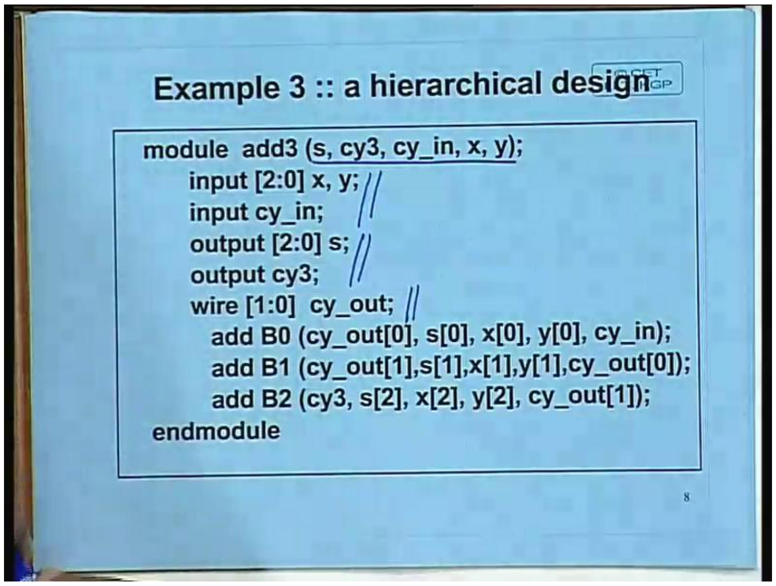
© CET I.I.T. KGP

```
module two_level (a, b, c, d, f);
  input a, b, c, d;
  output f;
  wire t1, t2;
  assign t1 = a & b;
  assign t2 = ~ (c | d);
  assign f = t1 ^ t2;
endmodule
```

a
b
c
d
t1
t2
f
7

And as I mentioned that if we have some intermediate wires for example as this example shows you can get a two level circuit. Here this ampersand means AND, bar means OR, tilder means NOT and hack means exclusive OR. So actually what we are getting is we are having a AND gate with a b as the input. Output is t 1. This is a NOR gate okay. OR and NOR c and d the output is t 2. Then we have an exclusive OR of t 1 and t 2. This is f. So depending on the intermediate wires we use here we are specifying a two level logic. Right. Well there is some problem here that can occur during some simulation. That we will see later. We will compute a little later. So that if you are not very careful about it then our simulation results and the actual synthesis may differ. Even in case of simple case like this. Yes (Student Noise Time: 21:13) That is the problem that can occur. I will come to the example and I will just come back to your answer a little later that means what if this your write before t 1 and t 2. Yes. There is a problem that may arise there. I will come to that shortly.

(Refer Slide Time: 21:42)



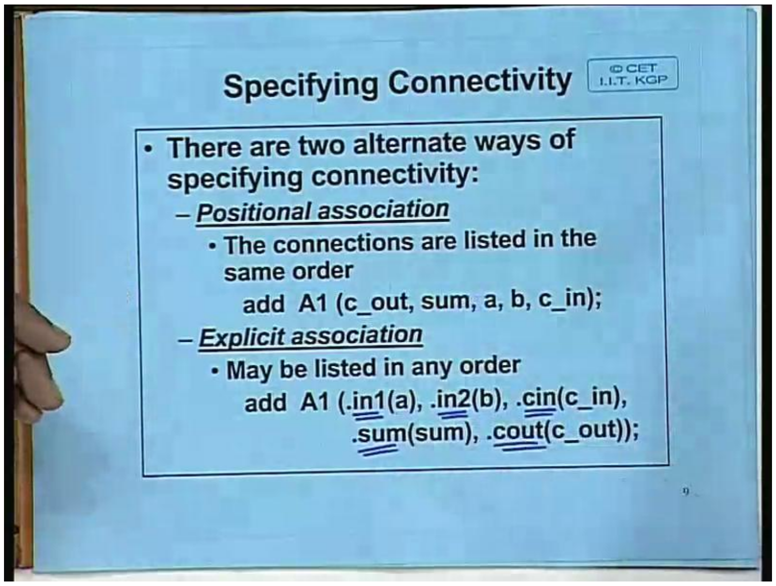
Example 3 :: a hierarchical design

```
module  add3 (s, cy3, cy_in, x, y);
      input [2:0] x, y;
      input cy_in;
      output [2:0] s;
      output cy3;
      wire [1:0]  cy_out;
         add B0 (cy_out[0], s[0], x[0], y[0], cy_in);
         add B1 (cy_out[1],s[1],x[1],y[1],cy_out[0]);
         add B2 (cy3, s[2], x[2], y[2], cy_out[1]);
 endmodule
```

8

And this kind of example we have already seen earlier. I am just showing this there. You can specify a hierarchical design by instantiating modules. Well here again we have the ports. You specify some of them as input some of them as outputs. These are the local interconnecting nets and these are the modules we are putting. So here why I have put this is that you can either specify a design in terms of its behavior or you can specify in terms of its structure. This is a structural design. Basically I have picked up three modules I have put them together and I have interconnected them okay. Fine. Now as you see from this example that the way we specify the interconnection is by explicit naming. This cy out 0 and cy out 0 means that the output of this module B 0 will go into the input of this module B 1.

(Refer Slide Time: 22:41)


Specifying Connectivity

© CET I.I.T. KGP

- There are two alternate ways of specifying connectivity:
  - *Positional association*
    - The connections are listed in the same order

```
add  A1 (c_out, sum, a, b, c_in);
```

  - *Explicit association*
    - May be listed in any order

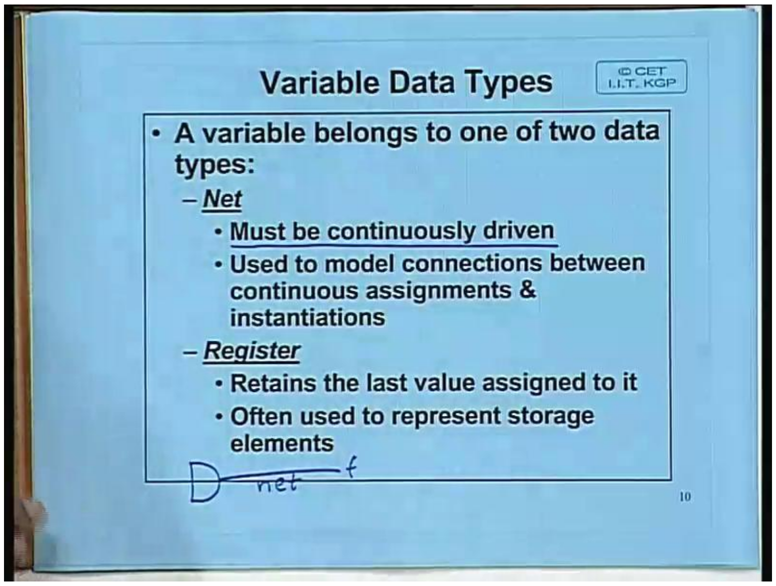
```
add  A1 (.in1(a), .in2(b), .cin(c_in),
         .sum(sum), .cout(c_out));
```

9

Now there are two ways of specifying such connectivity. One is called positional which was illustrated in the previous example. Here the relative positions of the different parameters are maintained. I will assume that the first parameter is the output carry out. Second parameter is the sum out. Next two parameters are the two inputs. The last parameter is the carry in. This is the assumption we will use and we will maintain. This is called positional association. The connections are listed in the same order. So when we instantiate a module we list the names of the nets in the same order. The first one is carry. Second one is sum. Two inputs and then carry in. But you can also choose to jumble up this order. This is called explicit association. There we can list them in any order. But obviously if you do that we will have to explicitly say which one is worth. So you will have to specify it in a way something like this.

You instantiate the module and this in one into c in and sum c out these are the names which were defined in the parent module description. And a b c in sum and c out are the variables in the present module. So you say that a is actually in 1 b is actually in 2 c is actually c in and this you can specify in any order. So we can use either of them. Some people say that explicit association is better because your code becomes selfdocumenting. Otherwise in a just complex layout you may have to go back and forth consider which one is you may lose track of the parameters if there are many. Okay. Now let us come to the variables. Now as the example showed that we have used some variables in our description. Well apart from the input and output we had used only one other kind of a variable that was called wire. Now let us look at it a little more formally.

(Refer Slide Time: 25:04)



Variable Data Types

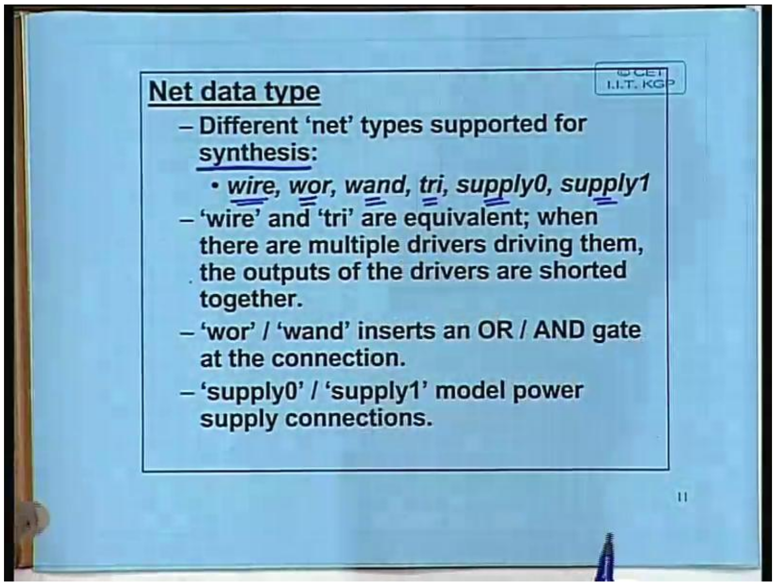
© CET I.I.T. KGP

- A variable belongs to one of two data types:
  - *Net*
    - Must be continuously driven
    - Used to model connections between continuous assignments & instantiations
  - *Register*
    - Retains the last value assigned to it
    - Often used to represent storage elements

The variable data types can broadly fall under two categories. One is called net and as the implies a net is primarily used to interconnect two things and register. Register again as the name implies it is used to store something. Some kind of storage to register net and register the main difference is net must be continuously driven. Continuously driven means suppose I have an AND gate. The output of that AND gate this can be a net say f. F is a net. Continuously driven means if you have an AND gate you always have some value at the output. There is no time for you to do not have a value. Continuously driven means net will assume its value instantaneously as soon as the gate changes state. It will not have to wait till the next clock right. That is how we say it is continuously driven. It is not synchronized with any other event.

As soon as the gate output changes the net will assume its new value and this as I mentioned this is typically used to model interconnection between models. Continuous assignments and instantiations we have used these things. So the example I have shown we have used these things. Register as the name implies it is supposed to retain the last value assigned to it. But you will see later it is not necessary that always it will have to retain because the synthesizer at the time of optimizing it may see that well a register variable is means a register is really not required. We can use it as a net variable also. These kinds of optimizations can be done. We will see it later. But by definition a register variable is one which is supposed to retain the last value assigned to it. That means there has to be some storage facility. So this is used to represent storage elements typically. So first let us look at the net data types in some depth. What are the different kinds of net data types available?

(Refer Slide Time: 27:33)

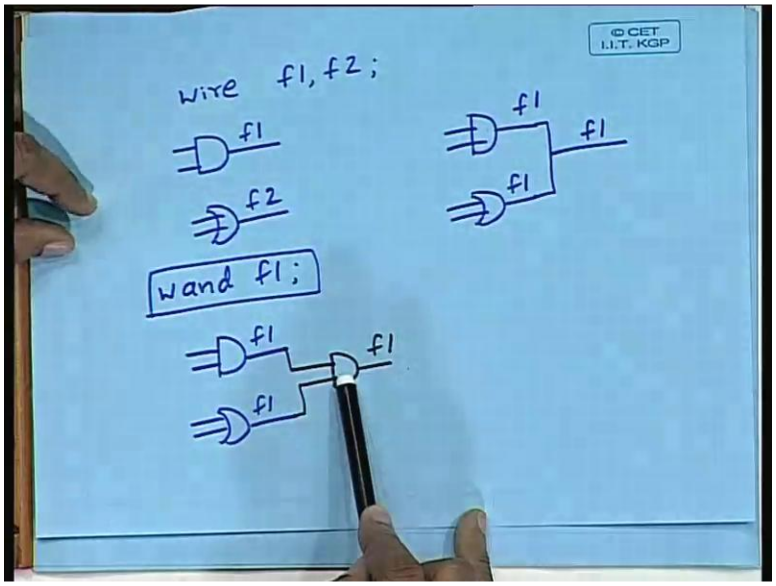


Net data type

- Different 'net' types supported for synthesis:
  - *wire, wor, wand, tri, supply0, supply1*
- 'wire' and 'tri' are equivalent; when there are multiple drivers driving them, the outputs of the drivers are shorted together.
- 'wor' / 'wand' inserts an OR / AND gate at the connection.
- 'supply0' / 'supply1' model power supply connections.

See there are a few other data types we are not going into that we are only considerating our attention to those which are synthesizable. There are net data types called wire. Wire we have already used in our examples. There is wire, wor, wand and tri state, supply 0, supply 1. This wire and tri state they are the same. There is no difference between these two descriptions. This wired or and wired and they are slightly different. First let us see wire and tri. Wire and tri they are equivalent. The idea is like this. If you are using wire or tri state, say it is something like this.

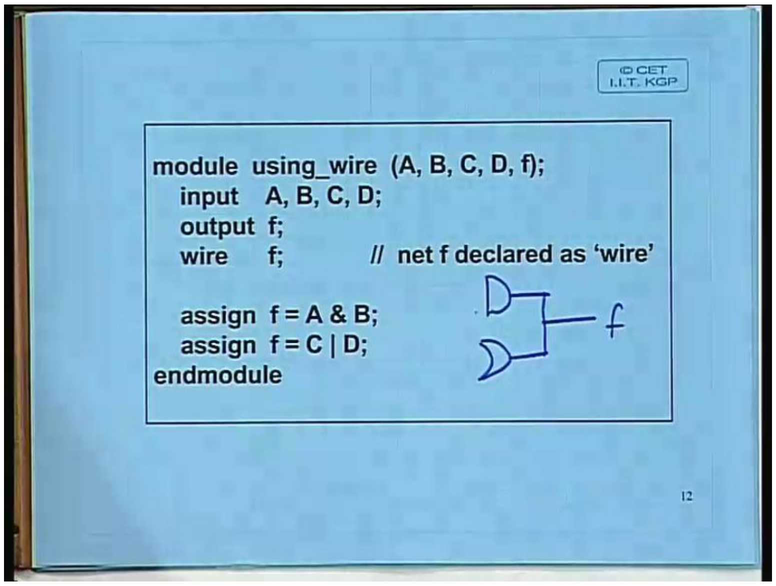
(Refer Slide Time: 28:23)



Suppose you use a wire description lets call it f 1 and f2. Now suppose you have well I am just showing the diagram, I have not shown the Verilog code. We have a gate which is driving f 1. You have another gate which is driving f 2. This kind of an example we have already seen but you can have another scenario where this gate is driving f 1. The other gate is also driving f 1. So the synthesizer will not given an error. What it will do? It will short these two outputs and will give the output as f 1. It will not give an error. So the synthesizer will assume that you have taken care that the designer has taken care that these two gates will be having tri state control. But if not then it is the design error you will be detecting during simulation. So the output logic value will be showing as in determinant. But if you are using wired or wired and suppose I had description as wired and f one and if I had something like this.

This is an AND gate which is driving f 1 and an OR gate which is driving f 1. Suppose I had a situation like this. Then what the synthesizer will do. Synthesizer will explicitly add another AND gate. We will connect these two to this and will call the output as f 1. Similarly if it is wired or it will add a OR gate okay. (Student Noise Time: 30:17) If it is only one then the AND will not be there. If it is more than one then the AND will be there. (Student Noise Time: 30:25) OR also same thing. So this wired or and wired and are similar. They insert an AND and OR gate and this supply 0 and supply one are used because in some cases you may need to supply a constant zero logic or one logic. So you can use this keyword supply zero and supply one to specify that. So let us look at some more examples very quickly.

(Refer Slide Time: 30:59)

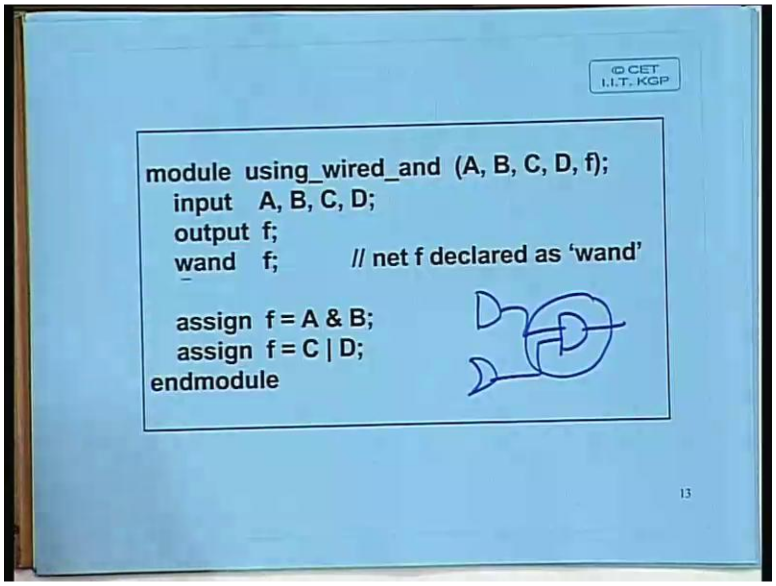


```
module  using_wire  (A, B, C, D, f);
   input   A, B, C, D;
   output  f;
   wire    f;          //  net f declared as 'wire'

   assign  f = A & B;
   assign  f = C | D;
endmodule
```

So this is just like the example that I had mentioned. There is one AND gate driving f. There is another OR gate driving f. Since it is a wire type they will be shorted together right? So the way you specify it this is a wrong specification. It will give it will give a simulation error at the output.

(Refer Slide Time: 31:25)

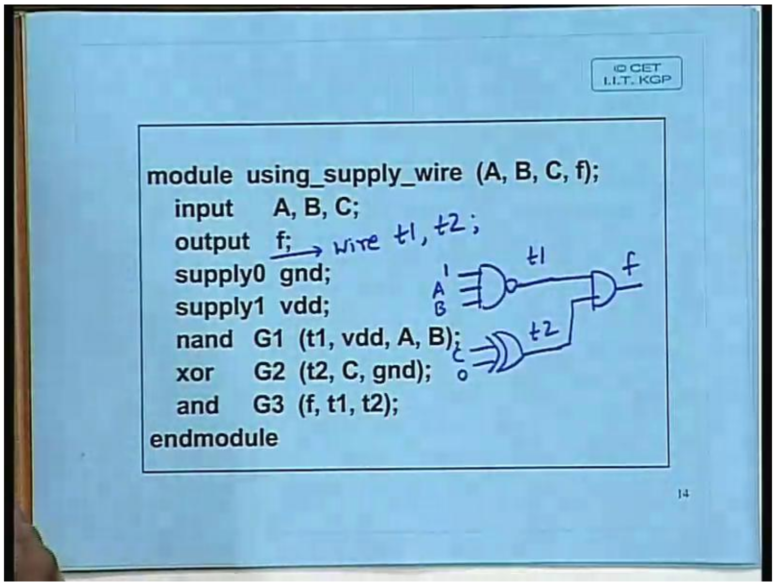


```
module  using_wired_and  (A, B, C, D, f);
   input   A, B, C, D;
   output  f;
   wand    f;          // net f declared as 'wand'

   assign  f = A & B;
   assign  f = C | D;
endmodule
```

And this example again this example I have taken if we use a wired AND this AND gate and this OR gate we will have an explicit AND gate added to it seems to be similar.

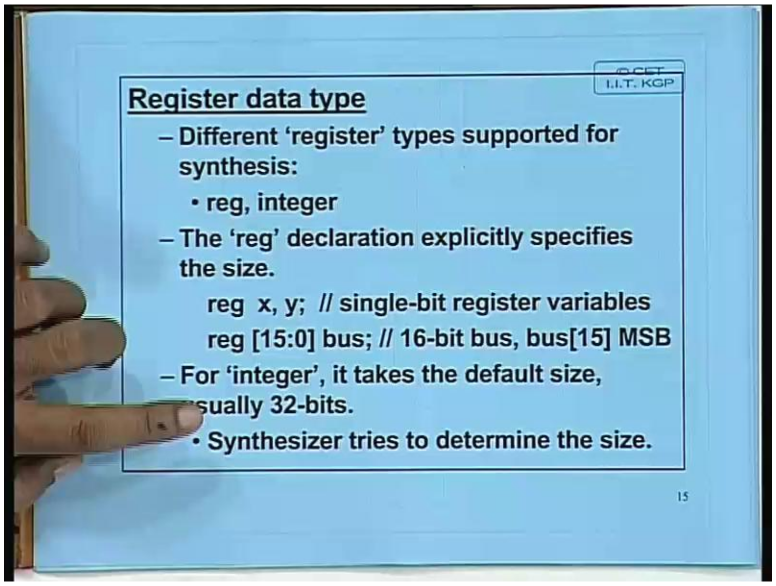
(Refer Slide Time: 31:45)



```
module  using_supply_wire  (A, B, C, f);
   input     A, B, C;
   output    f;  → wire t1, t2;
   supply0  gnd;
   supply1  vdd;
   nand   G1  (t1, vdd, A, B);
   xor    G2  (t2, C, gnd);
   and    G3  (f, t1, t2);
endmodule
```

And this example shows that how we can use the supply 0 and supply 1. Well supply 0 is a data type. Supply 0 this is the variable gnd supply 1 vdd. These are the variable names and the way I have specified here I think we have missed something here. There should be another line here wire t 1, t 2. So this will be there will be an AND gate. There are three inputs vdd A and B. It is a three input AND gate 1 A and B. The output is t 1. There will be an XOR gate. This will be a 2 inputs C and ground. This is C and this is ground zero. Output is t 2. These will be ANDed. Output is f. So here you can specify zero and one constant values also like this. Okay. So these are regarding the net data types wire, wired OR, wired AND, supply 0, supply 1. These are the five values we typically use.

(Refer Slide Time: 33:14)



Register data type

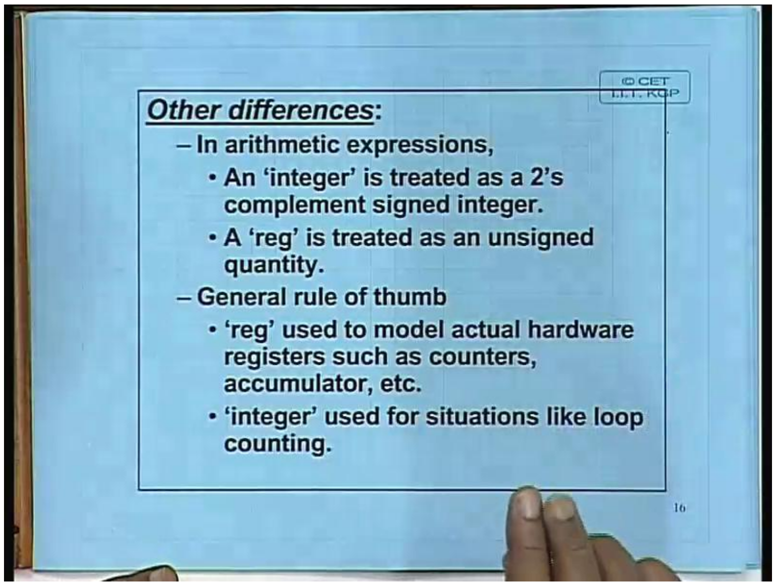
- Different 'register' types supported for synthesis:
  - reg, integer
- The 'reg' declaration explicitly specifies the size.

```
reg  x, y;  // single-bit register variables
reg [15:0] bus; // 16-bit bus, bus[15] MSB
```

- For 'integer', it takes the default size, usually 32-bits.
  - Synthesizer tries to determine the size.

Now let us look at the register data type. The register data as I have mentioned it is something which is supposed to be synthesized into a storage element. A register data type can be of two types reg and integer. The main difference is that when you define it as reg you must explicitly specify the size of the storage. Like if you simply specify reg x comma y this will mean that x and y are single bit variables single bit storage or you can specify in terms of vector like this. Reg 15 colon 0. This will be a 16 bit register and the convention is that 15 will be the MSB, zero will be the LSB and for integer normally the synthesizer or the simulator will be taking the default size which is 32 bits. But if you are trying to optimize then it can do a data flow analysis and it can try to deduce that what should be the minimum possible value of that integer and it can just use that minimum value. I will give an example. I will give an example that I will show you and just other than this. This is one main difference in register you have to explicitly specify the size and integer you do not specify the size. This is the default value is taken normally.

(Refer Slide Time: 34:54)



***Other differences*:**

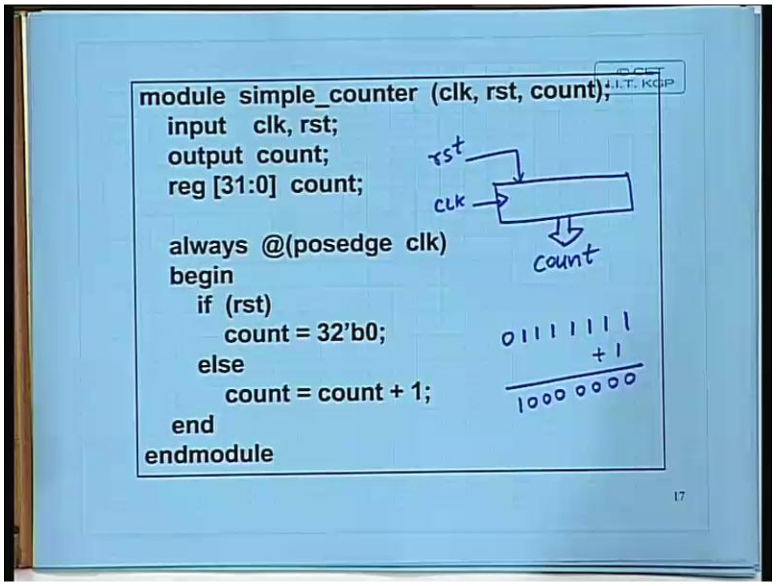
- **In arithmetic expressions,**
  - **An 'integer' is treated as a 2's complement signed integer.**
  - **A 'reg' is treated as an unsigned quantity.**
- **General rule of thumb**
  - **'reg' used to model actual hardware registers such as counters, accumulator, etc.**
  - **'integer' used for situations like loop counting.**

© CET
I.I.T. KGP
16

And other difference is that you can use the register variables in any arithmetic expression and if you are using an integer it is taken as a 2s complement signed integer. But a register value register value is taken as an as an unsigned quantity. There is no concept of a negative value there. This is another difference and just to give you a broad guideline you will be using a reg data type for all cases where you are actually trying to synthesize a register or a storage element. But sometimes when you are specifying in the behavioral level you may have to use a for loop kind of a thing. There the loop index variable. For example that you can use as an integer. That is just a rule of the thumb. The ones where you know exactly what is happening you know the size you specify the register. The others you can leave it. You can also specify some register no problem. But you can also specify them as integer. (Student Noise Time: 36:01) Input output connections they are neither reg nor there.

They are just some signals coming in. They will their type will be depending on the driver module the module which is driving those signals. So, whether they are driving from a reg type or a. No, no. I will show that example. There actually we will be specifying the size. I am showing with example. The type this by default it is taken to be register. But default it is taken this. is taken to be register. Okay. Where you can also use a vector because in that example of that adder we had used a vector here where we want a 0. They are synthesizable if the number of loops that is a constant. There is some kind of infinite loop or a variable loop. Then the some of the synthesizer allow it some synthesizer do not allow it. Now let us look at examples. Well there is some features they have not yet mentioned. But you will understand what this means.

(Refer Slide Time: 37:32)



```
module  simple_counter  (clk, rst, count);
   input   clk, rst;
   output  count;
   reg [31:0]  count;

   always  @(posedge  clk)
   begin
      if  (rst)
         count = 32'b0;
      else
         count = count + 1;
   end
endmodule
```

©CET I.I.T. KGP
rst
clk
count
0 1 1 1 1 1 1 1
+ 1
1 0 0 0 0 0 0 0
17

This is the Verilog description of a simple binary counter with synchronous research. This counter if you choose it as a black box, this will be having one clock input. This will be having one reset input and of course the count value. Clock and reset are inputs.

Output is the count and here you can explicitly specify count to be a register. This is another way of doing it. You can simply either specify it here itself. Or after declaring you can explicitly say count that whatever is coming this is a reg. (Student Noise Time: 38:28) Yeah if we want you can specify it. (Student Noise Time: 38:31) No yeah. No, no, no. Yeah, if you just specify only count will take it to be just one bit.

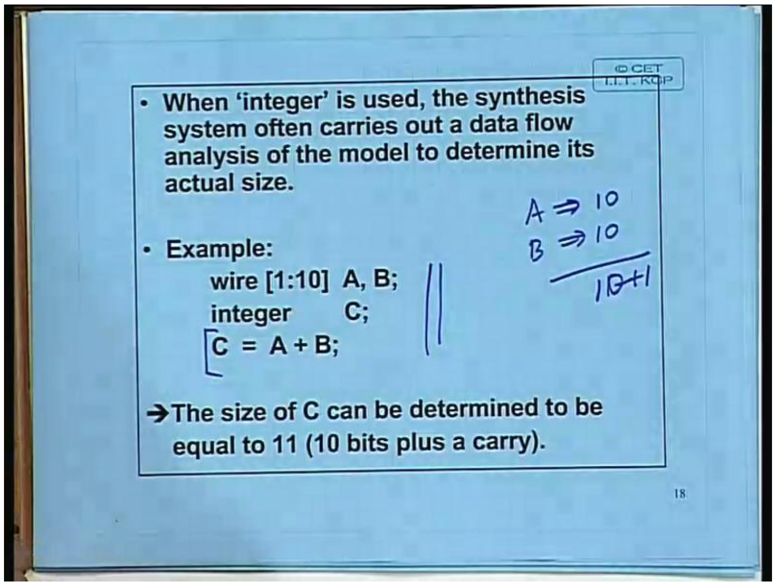
If you specify by vector you are specifying the exact size. (Student Noise Time: 38:42) Output 31 to 0 count you can also specify like that. Now in terms of normally in the body this always is a very important step you can understand what this means. This we will just see later more options. It says begin end this is a block. It says that you execute this block every time there is a positive edge on the clock. Always at posedge clock. Similarly you can specify negedge. That means there is something like a clock. Every time there is a leading edge on the clock you execute this. This says that if reset is also one then count this is one way of specifying a 32 bit 0 quantity, 32 is the size, apostrophe b means it Is in binary 0. So 32 times 0.

It is a 32 bit number. This count is initialized to zero or else count is incremented by 1. so this is just a counter With every clock it will either increase or it will be reset to zero if the reset is also high at that time. so this is an example where you have specified reg and of course in synthesis the counter will be synthesized as a register with storage element. (Student Noise Time: 40:18) 1 bit, 1 bit you are overriding the definition in the next line. This definition you can override. Just in the input output parameters you can simply specify. This is input, this is output. The sizes you can override in the next statements that option you have. (Student Noise Time: 40:43) Yeah output you will register you can also specify here output 31 colon 0. That also you can specify. That is one way. 31 not. I am just written this.

Now to show you that it is possible write like this. Okay now earlier I mentioned that that instead of register if we use integer then the exact size of the variables is not known but the synthesizer tries to deduce the size of the value. So I have an example to show you that. Yes. (Student Noise Time: 41:23) 2s complement in arithmetic. Whenever you are doing arithmetic it will be 2s component arithmetic. Well now here it will not matter. It will be the same because the whenever when (Student Noise Time: 41:45). Yes. (Student Noise Time: 41:50). You see if the count was initialized as integer, well just an example lets take that it is an 8 bit okay. 32 bit, 2 bit. Suppose the count value was this.

Maximum possible number in one. Now after that whenever you add one your count will be this. But if you treat it as a 2s component number this will mean minus 128. But if it is a simple counter you do not care what it means in 2 complement. More important to you is that the count value is this which to you this is equal to plus 128. So 1s complement, 2 complement is a 2 complements is a way of looking at it. (Student Noise Time: 42:42). No, no you can override this specification. You can override everything here but by default they have taken as registers. Okay.

(Refer Slide Time: 37:32)



- When 'integer' is used, the synthesis system often carries out a data flow analysis of the model to determine its actual size.
- Example:

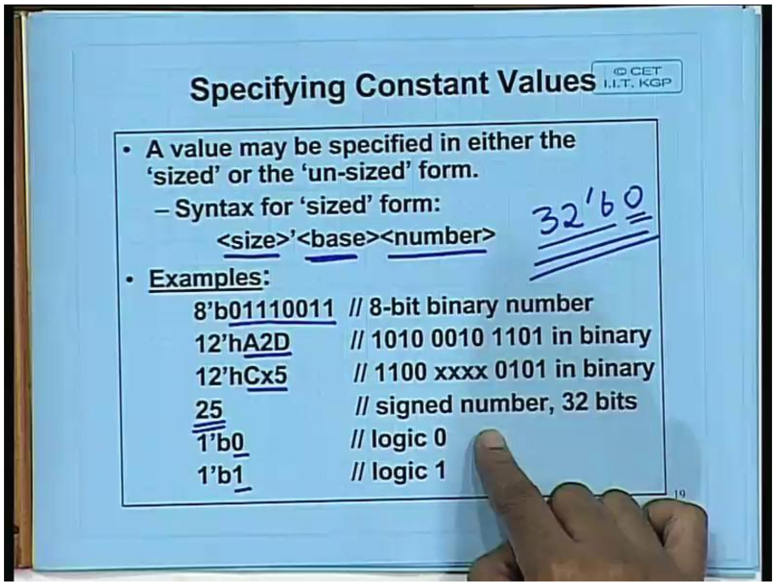
```
wire [1:10]  A, B;
integer      C;
C = A + B;
```

→The size of C can be determined to be equal to 11 (10 bits plus a carry).

Let us share a small example to illustrate that when you are using integers the synthesis system can deduce the exact size of that value. For example if we have a description like this. This is a Verilog port segment where one colon ten. You can also specify like this. This means 10 will be your LSB 1 will be your MSB. Numbering you can specify in whatever way we want. Integer C and in the computation part you write C equal to A plus B. Now by using a data flow analysis this synthesizer can deduce that A is a ten bit quantity. B is a 10 bit quantity and we are adding them up. So the sum can be 10 plus 1 11. So there will be 10 bits for this sum and one carry.

So C will be synthesized as a 11 bit. 10 bit raised to 1. So often the synthesizer tries to do this kind of data flow analysis and try to deduce the value of see instead of instead of by default taking 2 by 32 raised always it can try to find out what can be the maximum value possible and take it. (Student Noise Time: 44:31) No. As of now I have not written just as an example I have just written these three statements. This assign will also be there. Assign will be there assign will be there. So this is the (Student Noise Time: 44:47) I have shown.

(Refer Slide Time: 44:50)



Specifying Constant Values

© CET I.I.T. KGP

- A value may be specified in either the 'sized' or the 'un-sized' form.
  - Syntax for 'sized' form:
    <size>'<base><number>
- Examples:

```
8'b01110011  // 8-bit binary number
12'hA2D      // 1010 0010 1101 in binary
12'hCx5      // 1100 xxxx 0101 in binary
25           // signed number, 32 bits
1'b0         // logic 0
1'b1         // logic 1
```

This example I have already shown this example in one of the example. If for the counter when you initialize it to 0. So how do we specify constant value. Well a constant value you can specify like this also by simply specifying the value. 25 for example or you can specify the size in terms of the number of bits. Then apostrophe. Base. Base can be binary, hexadecimal and the actual number. There are different ways of doing that. See 8 base b. This means this is an 8 bit binary number and this is the value. So when I specify binary I have to give this value binary. 12 hex this is the hex value. 12 bit hex.

This is the hex value. One bit binary this is the binary value. One bit binary value. Now in the early example we have written something like this. 30 bit binary 0. This means this is a 30 bit binary number with a value 0 to the value 0 means 32 0s. So all 32 0s will be assigned. Sir how do you (Student Noise Time: 46:18) unsigned form it is up to you whether it is signed or unsigned it is up to your interpretation because these are constant values are assigned. No unsized. Unsized. Well unsized if you do not specify anything simply 25. Default is 32 bits like this just simply specify whether can you take it in decimals. You can take it in decimals. You can take in decimal by default. (Student Noise Time: 46:52). You assign it to a wire.

You cannot you cannot assign it to wire. You can assign it to a resistor because a constant cannot drive a wire. Only another module can drive a wire. So in the next class we will be continuing from this point onwards. We would be looking at some other features of the Verilog language. How to specify constants in terms of something called parameters. In this all these blocks we will be looking in some more detail and (Student Noise Time: 47:27) in fact here for the time being we have looked at only at ways of specifying combination logic and only one counter we have seen.

That is we will see that depending on the design style it will depend exactly what hardware will be synthesized. Okay. So if we have something on the back of the mind. For example we want to synthesize a decoder we can write our code in that way so that a decoder will be synthesized. So that we can force the synthesizer into synthesizing it in a particular way. That is possible that we will slowly do in the next week classes. (Student Noise Time: 48:07). Wire has to be continuously driven. (Student Noise Time: 48:10). Output of an AND gate is a high impedance state.

It is continuously driving the wire that means the wire is having a value x z. z is high impedance. So z is also a valid value. Zero one undefined do not care. So actually when you are saying it is continuously driven it means that it will have a value at all times. But when it is clocked only when clock comes the value will change. But if the output of the AND gate changes from one to a high impedance state the wire will also change to a high impedance state. Okay. It is not clocked. So let us stop for today. We will be continuing with our next lecture. Thank you.