

Embedded Systems Design
Prof. N. Vidya
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 11
Tutorial – III

Hello everyone. In the last class we studied about Verilog HDL.

(Refer Slide Time: 00:24)

Number representation

- Representation: `number` ::= `number`
 - size :: number of bits (regardless of base used)
 - base :: base the given number is specified in
 - number :: the actual value in the given base
- Can use different radix(base)
 - d or D :: Decimal :: default if no base specified
 - h or H :: Hexadecimal Hex
 - o or O :: Octal
 - b or B :: Binary
- Size defaults to at least 32.
 - You should specify the size explicitly!
 - Why create 32-bit register if you only need 5 bits?
 - May cause compilation errors on some compilers
- To be absolutely clear in your intent it's usually best to explicitly specify the width and radix.

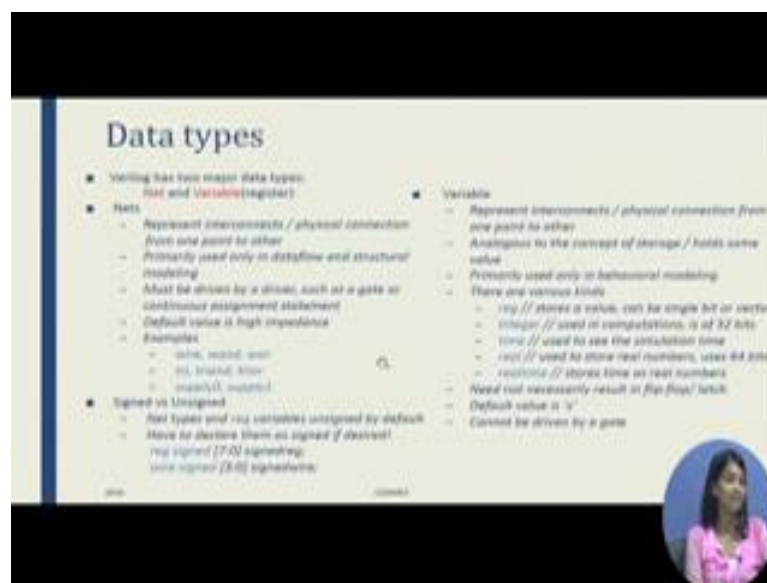
Number	Decimal Equivalent	Actual Binary
4'd3	3	0011
8'hA	10	00001010
16'o26	22	00010110
3'b111	7	00111
8'b101_1101	93	01011101
8'h1101	-	xxxx1101
-8'b6	-6	11111010

How we write a Verilog code, it is started with modules, ports, comments, it is convention, semantics, and we studied how to instantiate another module inside a module, and how 4 valued logic is used in Verilog code? So, in this tutorial we will try to learn more about Verilog how to write a Verilog code and at the end we will try to test whatever we have written in the Verilog code using a test bench. So, we will continue with number representation in a Verilog code, as you can see in the slide the number is representation starts with size, take base and a number for example, if we are defining a decimal number we use `4'd3` and if it is a 4 bit number we use the `4'b1101` as a number representation in Verilog; and as we are designing a hardware, it actually boils down to 0 0 1 1 in binary.

So, in the hardware there is no distinguishing between decimal and hexadecimal different formats it is just for the understanding of the Verilog code writer that we have 4 different types of number representation which is basically decimal, hexadecimal, octal and binary

these are the basis you using which we can represent a number; also we do have unsigned and signed number representation in Verilog. So, as you see these are all the examples of number representation, by default the numbers take 32 bit value if it is not mentioned and it is usually preferred that we always define the width and radix of the number otherwise it goes to default and if you want suppose 5 digit number and it defaults to 32 bit number then it takes more hardware. So, we do not want unnecessary resource utilization. So, we generally make it practice to declare the width and radix of the number we are using.

(Refer Slide Time: 02:40)



So, this is the basic number representation; we will move on to the data types that I use in Verilog, till now I discussed about wires and log. So, wire is a type of data type called net, and there is another data type called variable. So, nets and variables are generally physical connection or inter connection in a Verilog module, where we interconnect 1 module inside a top level module of Verilog using nets or variables.

Variables are used to store some values and nets are generally used for connecting one module to another module is a single bit wire or a vector. So, next they generally do not have a memory and variables hold memories. So, there are different kinds of nets and variables we have buyer w and w or all this kind of nets and in variables we have registers reg integer time these are all keywords for net and variable types and in general we use wire and reg in mainly RTL codes we use by run reg generally. So, that since we

learned about 4 value large logic, variable state by default x as their value if it is not initialized and for reg also it is it is high impedance step which is z.

So, net side by default z; net shall assign z by default if it is not in surely as and variables take the value x. So, as I discussed earlier number can a value can be signed or unsigned in data in Verilog code and unless and until we have specified what it is a sign number by default Verilog takes it as a unsigned number, and this is a syntax for declaring a signed value or a net auto variable. So, you can see how reg signed register is declared is a 8 bit register value which is a sign value; generally a two's complement format is used in Verilog in hardware to represent a sign value.

(Refer Slide Time: 05:03)

Operators

Operator Type	Operator
Bitwise	~, &, , ^, ~^, ~
Logical	!, &&,
Unary/Reduction	&, ~&, ^, ~^, ^~, ~^~
Relational	<, <=, >, >=
[In]Equality	==, !=, ===, !==
Conditional	?:
Arithmetic	*, /, ** (power), %, +, -, ++, --, ~, ~^
Concatenation	{, }

a	b	a & b	a b	a ^ b	a ~^ b
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

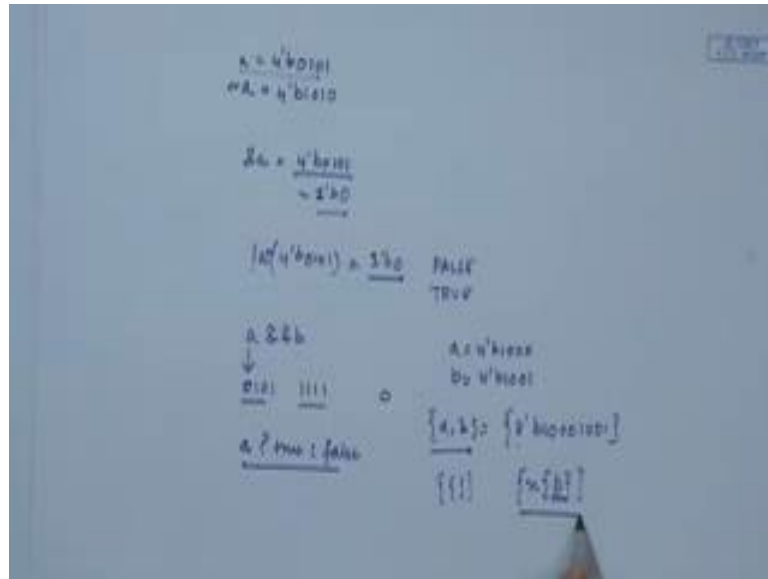
a	~a	&a	~&a	^a	~^a
0	1	0	1	0	1
1	0	1	0	1	0

a	b	a && b	a b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Now, we move on to the operators that are used in Verilog, we have 8 types of operators in Verilog these are bitwise, logical, reduction, relational, inequality or equality conditional, arithmetic and concatenation operators. So, bitwise reduction and logical operators are the Boolean operators that we use in Verilog and these are most commonly used. This arithmetic and conditional operator are not a Boolean operation.

So, I will just give one or two examples of each of them for you to understand and if you want to look it up more you can look it up into in website called as sequel dot com. So, since I have limited time, I just describe 1 or two of their examples. So, first we start with bitwise operation; bitwise operations are operated on single bit of a number.

(Refer Slide Time: 06:06)



So, suppose a and b are 4 bits and if I take a as a 4 bit 0 1 0 1 value. So, if we have to apply an operator call not on a we may get a value of 1 0 1 0. So, in a bitwise operation not is applied to each bit of the variable a. So, we get not of 5 as a similarly a and b and or XOR and XNOR operations are applied on different operands in Verilog.

We also have reduction operators which are also a Boolean operation, wherein we get we get output as only 1 bit value as opposed to bitwise operator where we get if you give an input of 4 bit variable we get an output of 4 bit variable, in reduction operation we get 1 bit value. So, if you are applying suppose a. So, if we are doing and of a and we have 4 bit 0 1 0 1, we apply and to each of the bits. So, we and 0 and 1 and then 0 and 0 and then 0 and once we will be get answer as 1 bit 0. So, 4 bit number is reduced to a 1 bit number, when we apply a reduction operation. So, that is why it is called a unary operator also because we get an output of 1 bit.

So, reduction operators are again and n and or nor and XOR XNOR. In logical operation we get an output of true or false like if the operand satisfy the condition we get true or false as output, they are generally used in if and k statements or to check some conditionality between two variables. So, if we have a not of a in a logical statement, suppose 4 take 0 1 0 1 if this is equal to this then we get 1 tick b 0, but this is different from what we have defined in a bitwise operation, here we have either false or a true

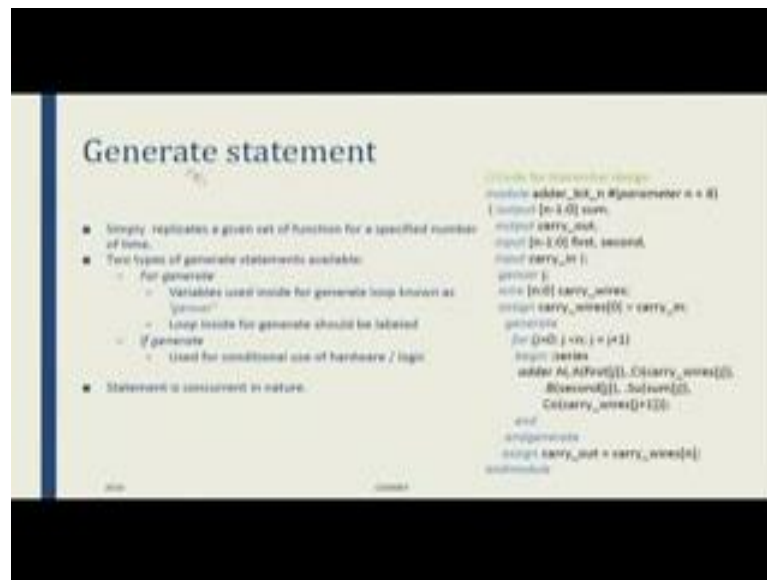
output hope you understand what is the difference between these two and if you take a and example and b example. So, you a can be any like if it is 0 1 0 1 and 1 1 1 1.

So, this and this will come to 0; because we have 0 as. So, we get false or two as output, these are the logical operators and in logical operators we also consider inequality operators there are two kinds of inequality operators which as you can see double equality and triple equality; triple equality operators are used for case x statements like if we have x in x in the inputs at that time we use equality operators triple equality operators; also the conditional operator that we use is same as what we use in c, I think in c also we have a conditional operator. So, if suppose some condition is true then this value the first value is executed otherwise the second value is executed.

So, this if you simply write this corresponds to a marks in a hardware language. So, this mostly use to infer a marks in hardware, and a normal arithmetic operation addition, subtraction multiplication, and power of a variable, division model is generally division and mod lope operators are not synthesized into a hardware in a Verilog code.

Next operation is concatenation; here concatenation means that we add two variables and for example, if we take a as 4 tick this and b as then a concatenation b will become as 8. So, this variable will become 8 bit a will be appended by b at the end. So, this is what kind concatenation operator does and the double concatenation operator is a replication operator which replicates n number of times the value inside n. So, b will be replicated n number of times, this what replication operator means. So, we have now somewhat covered the operators and these are generally used in a behavioral type of model that we will discuss in some time.

(Refer Slide Time: 11:49)



Generate statement

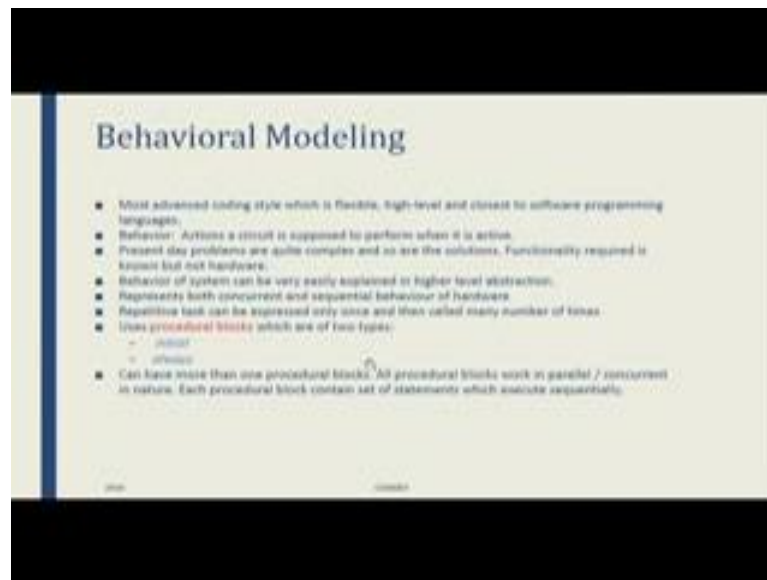
- Simply replicates a given set of function for a specified number of times.
- Two types of generate statements available:
 - For generate
 - Variables used inside for generate loop known as 'genvar'.
 - Loop inside for generate should be labeled.
 - If generate
 - Used for conditional use of hardware / logic.
- Statement is concurrent in nature.

```
//Example for instantiation design
module adder_8bit_n #(parameter n = 8)
  (input [n-1:0] sum,
   output carry_out,
   input [n-1:0] first, second,
   output carry_in);
  genvar i;
  wire [n-1:0] carry_wires;
  assign carry_wires[0] = carry_in;
  generate
    for (i=0; i < n; i = i+1)
      begin : iter
        adder A1_A[0][i].C(carry_wires[i],
          A[second][i].A(sum[i],
            C(carry_wires[i+1]));
      end
  endgenerate
  assign carry_out = carry_wires[n];
endmodule
```

So, I want to discuss a statement called generate statement, which we generally use in Verilog when we want to instantiate the same module several number of time. For example, if you see in this code, there is a 1 bit adder which I want to use to make an 8 bit adder. So, I used generate blocks to instantiate more than one instances of the same adder. Generates it statements are generally used to replicate the functionality of a particular module or function inside a Verilog code. So, we have two types of generate statements for generate and if generate and the variables inside a general statement is called is declared as gen wire j like how it is there in the slide.

And if generate segment is generally a conditional use of hardware logic inside the Verilog code. So, the statements inside a generate statement are executed concurrently. So, if you synthesize this module it will make an 8 bit adder, instantiating 8 1 bit adder instances inside it. So, this was about the generate statement.

(Refer Slide Time: 13:15)



Next we will study the behavioral modeling; behavior modeling as we have discussed earlier it is the only model which describes the design in terms of its functionality, the structural modeling that we learnt earlier had the deferred advantage of not allowing complex design to be designed using that kind of modeling. Behavioral modeling helps us to design a design of circuit in a behavioral way means we can use a high level language like English or C like language to describe the functionality of a circuit. So, also the data flow modeling that we learnt earlier had the disadvantage that memory elements were not we were not able to design a memory element. So, we overcome that in behavioral modeling and we use concurrent and sequential elements inside behavioral modeling to design a hardware.

So, in behavioral modeling we have procedural blocks, which are mainly of two types initial and always blocks and we can have more than one procedural box inside a behavioral modeling code. So, all procedural blocks that are there in a Verilog module works in parallel or concurrent way and each of these in procedural blocks have a set of statements. So, these set of statements are called procedural assignment statements, since they are used in a procedural block.

(Refer Slide Time: 14:57)

Ports and Datatypes

We can add the `reg` keyword to output or inout ports (we wouldn't be assigning values to input ports), or we can declare nets using `net` instead of `wire`.

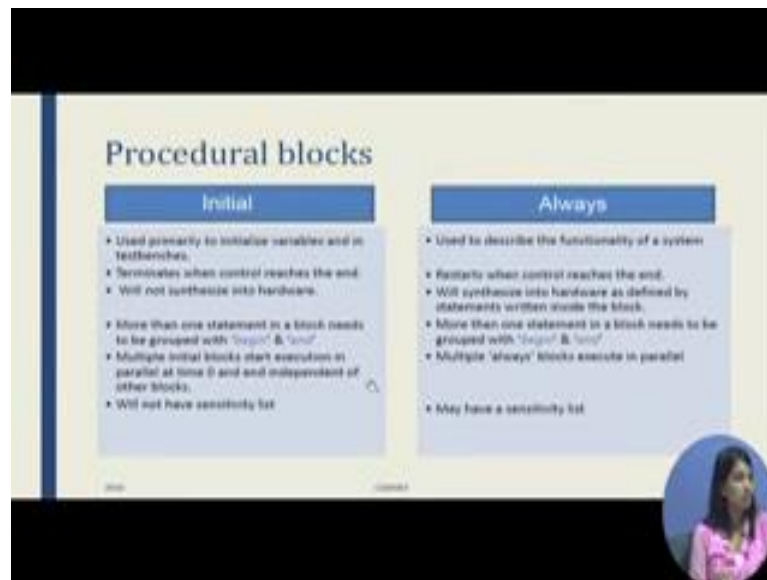
We've been using `wire` declarations when naming nets (nets are declared as `wires` by default). However, nets appearing on the LHS of assignment statements inside of procedural blocks must be declared as type `reg`.

The diagram shows a module with an `Input Port` connected to a `net`, which is then connected to an `Output Port` (labeled `assignable`) that outputs to a `net`. A `Heat Port` is also shown connected to a `net`.

So, before going more into the procedural statements and all I would like to discuss about ports and data types with we define ports and data types separately, now we will see how it is related in a Verilog module. So, if you see it generally we use the approach that the input ports are declared as nets, and the output ports are declared as either nets or register and the in out ports are generally declared as net inside a module and when we instantiate this module into another module the input port can be connected with a net or register type variable, and output port can be connected to a net variable, similarly in out port can be connected to a net variable.

Verilog has a strange rule that they will a left hand side of an assignment statement in a procedural block has to be a register variable, whether it is a wire or a net or a very or a register it does not matter, but it has to be a register variable if it is in the inside a procedural block like in inside, initial or always block. So, not an initial block only in always block, it has to be always declared as a reg type variable.

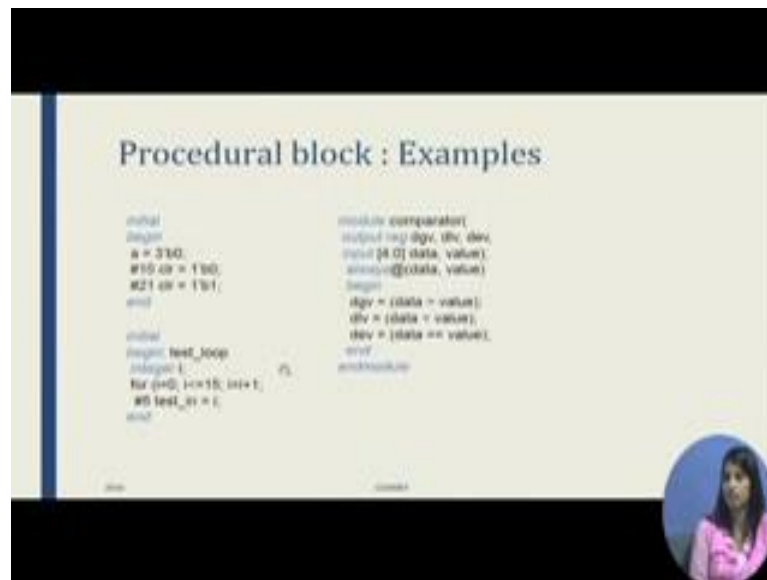
(Refer Slide Time: 16:20)



So, I spoke about initial and always block. So, we will just quickly see what are the differences in initial and always block; initial and always blocks are used in a behavioral type of modeling, they have begin and end if there are more than one statement inside them and initial work is generally used to initialize variables in a design and are they are used generally in test benches, always blocks are use to describe the functionality of the system. Initial box terminates when the control reaches an end while the always block continuously runs according to the inputs and the sensitivity list. So, initial blocks are not synthesized into a hardware and in hardware always block always block the synthesized into a hardware, according to the statements that are written inside it.

Also when there are multiple always blocks or initial blocks they always execute in parallel and in if there are multiple initial blocks they start at the same time, but the ending depends upon the statements that are inside the in each initial blocks. So, always block have a sensitivity list and initial book does not have an insensitivity list.

(Refer Slide Time: 17:44)



So, if I take example of always an initial block. So, this initial block can be are generally used in test benches as I have said those that is why I am not mentioned we have not read about test benches yet. So, I have not mentioned that code. So, this is generally the design of an initial block they have begin and end, and we have statements assignment statements inside them and this is a module which uses always block for describing a functionality.

So, in this module we have declared a comparator which compares the values input values and gives an output value. So, if you can see output is declared as reg, input are generally default wire and all the left hand side variables are declared as reg in the always block. So, I spoke about sensitivity list; sensitivity list is are the values that are there inside this bracket in the always block. So, always at data and value, data and value comprises of the sensitivity list of always block. So, whenever there is a change in data and value these statements will be executed, that is what it means.

(Refer Slide Time: 19:14)

Conditional statement

- Statements written inside procedural blocks are known as **procedural assignment statements**.
- There are two sub-classes known as conditional assignment statements:
 - if-else, case
 - loop statements
 - for, while, forever, repeat
- Some of these statements synthesize while others do not.
- Conditional statements are behavioral equivalent of the conditional operator.
 - These can be nested as many times as needed.
 - When written properly will infer a multiplexer or mux like structure.

```
// 4 to 1 multiplexer
module mux4_1
input a,b,c,d, input [1:0] sel, output
reg z, zbar;
always @(*)
begin
    if (sel == 2'b00) z = a;
    else if (sel == 2'b01) z = b;
    else if (sel == 2'b10) z = c;
    else if (sel == 2'b11) z = d;
    else z = zbar;
end
// when sel is 0 or 1, statement inside
// module body always leads to the
// following assignment because "always"
// the if statement has been executed!!
zbar = z;
endmodule
```


So, this was an example of a procedural block that we studied. So, I spoke about procedural assignment statements that are there in procedural block. So, they are of two kinds: conditional assignment statements and loop statements. In conditional statements we have if else and case statement and in loop statements we have for while forever and repeat; not all of these synthesize into a particular hardware, generally we use if else in case and sometimes we use for with a predefined value, but while forever and repeat are not synthesizable in hardware.

So, as you can see in this code this is a code for 4 to 1 multiplexer, which we studied earlier in structural modeling this has been redesigned using conditional statement if and else. So, if else statement as you can see can be nested and it can be nested as many times as possible and if you design it properly it will infer a multiplexer. So, generally we have to cover all the cases that input can take inside any final statement for example, in here I have covered all the states that selection to bid select line can take, and at the end I have if the values goes to x or z suppose the input values go to z, the l statement specified if the values goes other than what is mentioned in here the output takes this value.

So, then we have a z bar which is a not of z. So, what is written in this comment is if we do not declare z bar after the if and else statement, the value of z bar will not get updated if we have an x or z in the selection line. So, if we have an x or z in the selection line the

output value gets updated only at the end. So, it is necessary that any value that depends on this output has to be assigned after the if else statement. So, that is what is there in this comment.

(Refer Slide Time: 21:25)



Case statement

- Cases of if then else statements aren't the best way to indicate the intent to provide an alternative action for every possible control value.
- `case` looks for an exact bit-to-bit match of the value of the case expression (e.g., `sel`) against each case item, working through the items in the specified order. `case/casez` statements treat X/Z values in the selectors as don't cares when doing the matching that determines which clause will be executed.

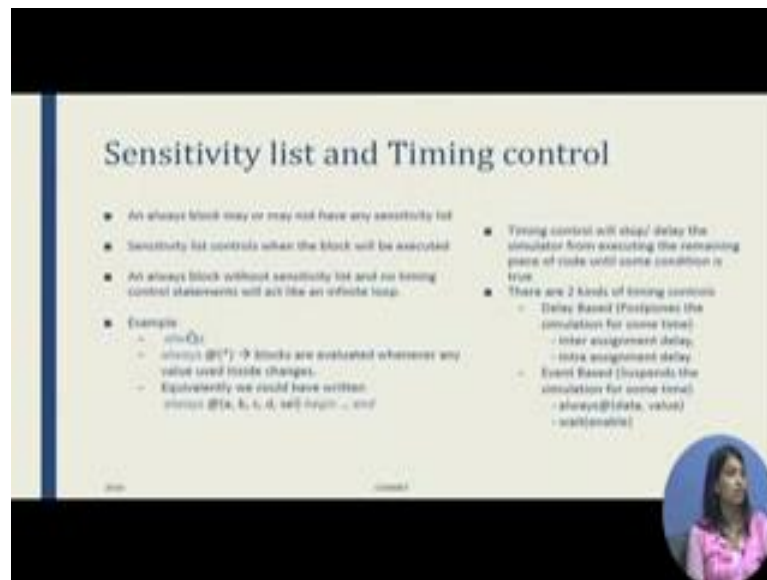
```
// 2-to-1 multiplexer
module mux2_1
input a,b,c,d, sel; // 00 sel, output reg
output #0;
begin
case (sel)
3'b00: z <= a;
3'b01: z <= b;
3'b10: z <= c;
3'b11: z <= d;
default: z <= 1'b0; // no case sel is 0 or 1
endcase
end
endmodule
```

So, next we will see case statement; as you saw if else statement are true like works only if there are very less loop, if there are many if else statements nested inside we tend to skip some of the else part of the design and which tends to unnecessary hardware infringe. So, we generally use case statement instead where we have a default value mentioned which corresponds to any of the value which is not mentioned inside the other cases it takes a default value if it is not mentioned.

So, case statement has a sensitivity list of select and the variables that select can take is matched bit by bit in a case statement, first it and there are priorities assigned to the case statement. So, if the select takes 0 0 then the first line will get executed, similarly all other values of select corresponds to the particular statement in the case.

So, if you can see there are 3 types of case statements that we have in Verilog k here normal case statement there is case x and case z. So, if we have a x or z in select line inside the normal case statement, it will take the default value as the output, but if you use case x or case z it will try to match the exact value of x by considering x or z as a do not care condition and it will take the first priority select condition that is there in the inside the case statement. So, that is how case x and case z are used in a Verilog module.

(Refer Slide Time: 23:21)



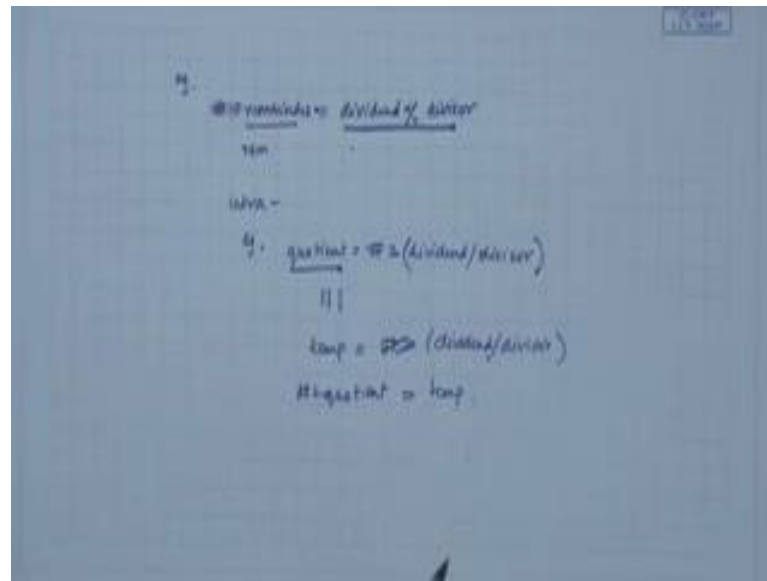
Sensitivity list and Timing control

- An always block may or may not have any sensitivity list
- Sensitivity list controls when the block will be executed
- An always block without sensitivity list and no timing control statements will act like an infinite loop.
- Example:
 - `always @(*)` → blocks are evaluated whenever any value used inside changes.
 - Equivalently we could have written:
`always @(A, B, C, D, sel, input ..., and`
- Timing control will stop/ delay the simulator from executing the remaining piece of code until some condition is true.
- There are 2 kinds of timing controls:
 - Delay Based (Postpones the simulation for some time):
 - inter assignment delay,
 - intra assignment delay.
 - Event Based (Suspends the simulation for some time):
 - `always @(data, value)`
 - `wait(time)`

So, we will quickly see what is a sensitivity list and timing control in Verilog. So, as discussed sensitivity list is the expression that comes inside an always block statement, when we have an always at star mentioned in the design the block is always executed, when there is any change in the any input inside the always block statement and always at some in some parameters if we give register or net variable then any change in the variables that is mentioned in the expression will then only the always block will execute.

And if you do not mention any sensitivity list inside the always block, the always block operates as a loop, it always execute. So, this is what an i. So, sensitivity list has a very important role in declaring in using an always block. Now we come to timing control which is used generally in test benches and in simulator, which will stop or delay the execution of a piece of code or a statement. So, there are two kinds of timing control that are available in Verilog, delay based timing control and event based timing control. So, if you can see in delay based timing control we have two kinds of delays inter assignment delay and intra assignment delay. So, I will give an example of how inter and intra assignment delays are used. So, delay based timing control generally postpones the simulation time of a particular statement.

(Refer Slide Time: 25:15)

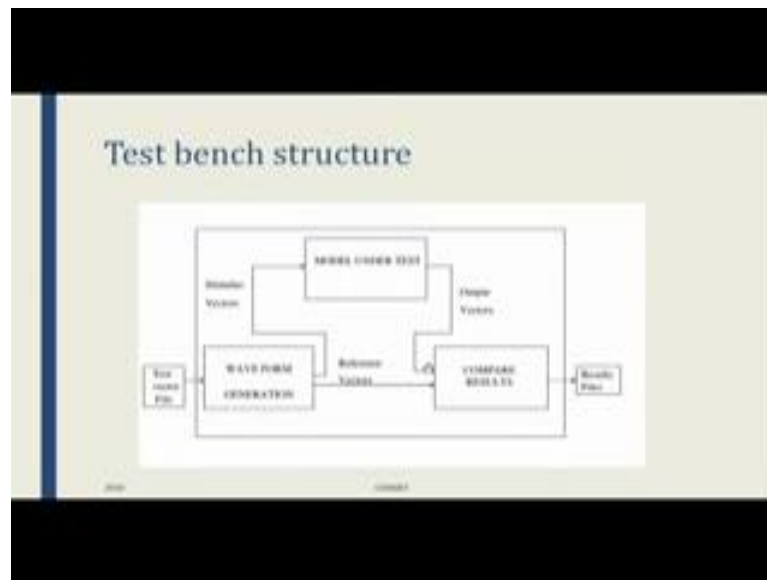


So, if I have for example, hash 10 if I have an example of this assignment statement, in this statement the simulator executes the value of remainder after 10 units of time. So, that that is what inter assignment delay means. So, whenever there is a change in dividend and divisor, the simulator computes right hand side of the value and it assigns the value to the remainder, and the remainder value takes on this value after 10 units of time.

So, this is the inter assignment delay and if you take, if I give you an example of intra assignment delay that will be. So, in intra assignment delay what happens is we compute this expression and the value of this expression is assigned after two units of time to this variable. So, this is the main difference between inter and intra assignment statement. So, this actually boils down to temporary variable equal to hash divisor and then this temporary variable is assigned to the question after 2 units of time something like this sorry this is wrong.

So, first this is calculated and then after two units of time we get the value. The second type of timing control that we have is event based timing control, which suspends the simulation time for some time for example, if you take an always block inside a simulator, it will suspend the value until we until the data or value is changed or until we get an enable. So, wait is a kind of event based statement.

(Refer Slide Time: 27:57)



We will now look at the basic structure of a test bench that is used in Verilog. So, if you can see there is a model under test, we have a test vector file and we compare the results of the output according to the input. So, a text vector file generates input that is required for a model and the model runs its functionality according to the inputs and gives us an output, we compare the output with the expected result and we get the result files. So, this is the basic flow of a test bench that is there in Verilog.

(Refer Slide Time: 28:40)

Components of test bench

- The simplest test benches are those that apply some sequence of inputs to the circuit being tested so that its operation can be observed in simulation.
- Waveforms are typically used to represent the values of signals in the design at various points in time.
- A test bench must consist of a component declaration corresponding to the unit under test(UUT), and a description of the input stimulus being applied to the UUT.
- A test bench is a top level module without inputs and outputs
 - Data type declaration:
 - Declare storage elements to store the test patterns
 - Module instantiation
 - Instantiate pre-defined modules in current scope
 - Applying stimulus
 - Describe stimulus by behavior modeling
 - Stimuli are given by variables of data type register
 - Display results
 - Response is captured by variables of data type real
 - By text output, graphics output, or waveform display tools
- Three kinds of testbenches are in use
 - One which only supplies the stimuli
 - One which not only supplies stimuli but also captures the result
 - One which supplies stimuli, captures result and also carries out timing check

I will just quickly describe what are the components of test bench and test bench is basically a top module without ports input output ports; it has modules instantiated in the main module that is called design under test or the module under test, whose functionality we have to verify and we have input stimulus and we display the output either to wave form or through some specified file.

So, these are the basic components of test benches; there are three different kinds of test benches first is that we only apply the stimulus and see the output in waveforms; second is that we apply the stimulus and we get the results in some files output results and other is that we give the input we get the output and we compare the input output if it is correct or not and display the results. So, these are the different kinds of test benches that we have.

(Refer Slide Time: 29:38)



This is I have just given an example of test bench. So, this is a full adder circuit, which uses 4 bit full adder which uses a 1 bit full adder, and we have a test bench structure we have instantiated full adder 4 bit as a module inside the test bench, we do not declare any ports in the test bench and we declare the ports of the instantiated design under test inside the test bench as registers and wires as you can see.

So, these as we discussed earlier ports and data types have a particular way that they are declared, we have to follow that rule inside the test bench also. So, here we want to test whether the full adder functionality is proper or not whether it is giving the addition

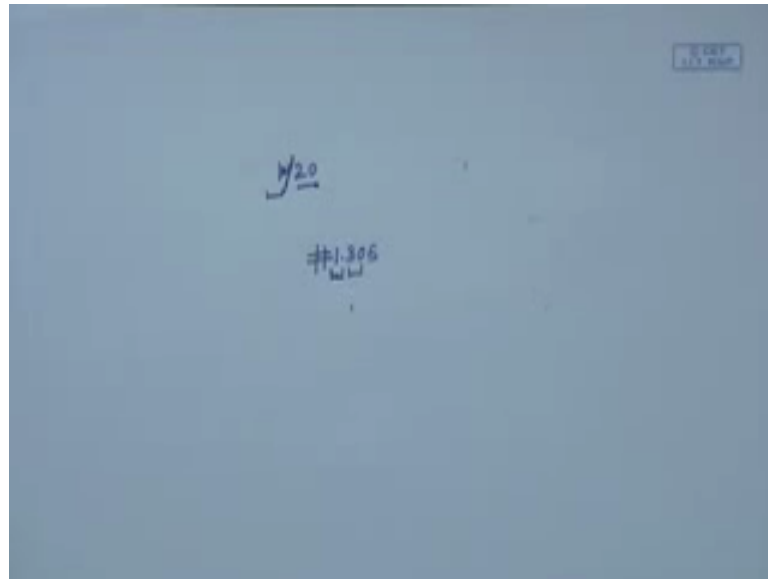
result properly. So, we give different kinds of inputs at different times and verify whether the outputs output is proper or not. So, we use initial statement which is a procedural block and we give different inputs at different times.

(Refer Slide Time: 31:01)



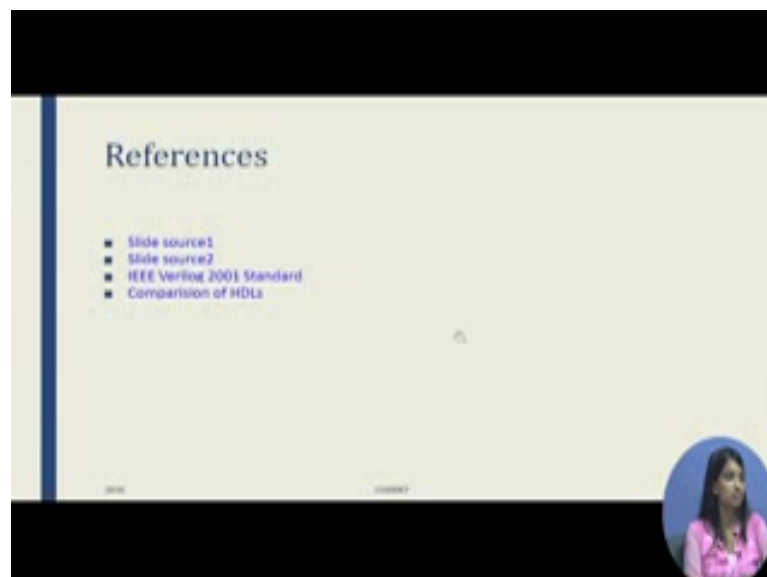
So, now we know how to write a Verilog code Verilog module and test its functionality using a test bench. There are some compiler directives that I use inside Verilog, which can be used across Verilog codes. So, there are few like macro definition, conditional compilation. So, all comparing directive starts with a tick macro accent and they are generally used to def like define statement is generally used to declare a variable which is used throughout the Verilog code. Similarly if def and ticking tick include is used to include some of the header files that we have declared in Verilog, tick time scale is used to mention the simulation time of a particular code.

(Refer Slide Time: 31:47)



So, it is generally mentioned as one slash 20 or something at the beginning of the code which means that the simulation time is 1 nanosecond and 20 is a resolution of the time like 20 picosecond is a time. So, if you declare some delay as 1.305. So, the simulation time will be 1 nanosecond and 30 picosecond something like that. So, we declare it 1 nanosecond slash 20.

(Refer Slide Time: 32:31)



So, this example of tick time scale, so if we want the Verilog code to not take a default net type, we can use default net type none compiler directive where none specifies that

we do not specify any data type for the default value. If variable is not declared properly then it takes no value; they are more compiler directives, but these are the basic ones that we use, these are the reference files that I have used you can refer to them also the sites that I have mentioned in the class. So, that is all today.