

**System Design Through VERILOG**  
**Prof. Shaik Rafi Ahmed**  
**Department of Electrical and Electronics Engineering**  
**Indian Institute of Technology, Guwahati**

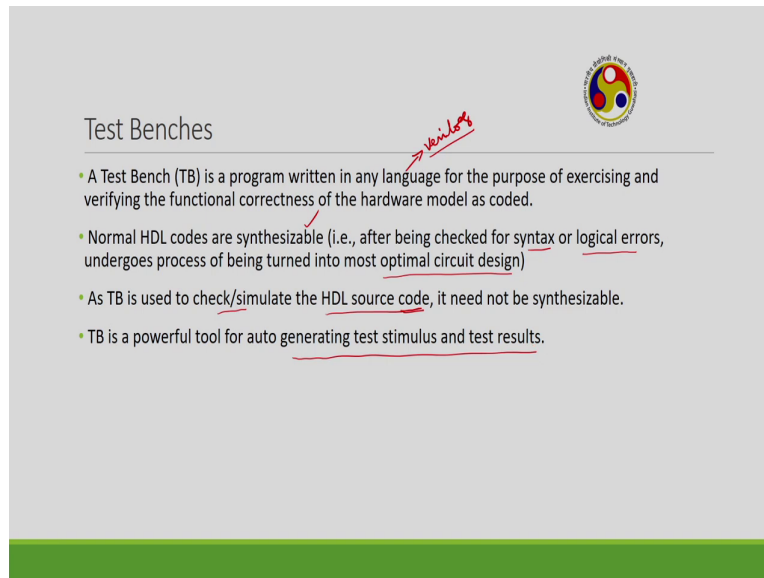
**Test benches**  
**Lecture - 21**  
**Combinational circuit examples**

(Refer Slide Time: 00:32)



In the last lectures, we have discussed about the VERILOG codes for different combinational and sequential circuits. Now, the question is how to check the functionality of this circuit; whether the code that is written is working properly or not. So, for that, we need a module called a test bench. So, today, we will discuss about the test benches in VERILOG.

(Refer Slide Time: 01:03)



The slide is titled "Test Benches" and features a logo in the top right corner. The content is as follows:

- A Test Bench (TB) is a program written in any language for the purpose of exercising and verifying the functional correctness of the hardware model as coded. (Handwritten red arrow points to "Verilog")
- Normal HDL codes are synthesizable (i.e., after being checked for syntax or logical errors, undergoes process of being turned into most optimal circuit design)
- As TB is used to check/simulate the HDL source code, it need not be synthesizable.
- TB is a powerful tool for auto generating test stimulus and test results.

So, what is Test Bench? Test Bench is also another program. This can be written in any language. But normally, we will use VERILOG only to write the test bench also. This is used for purpose of verifying the functional correctness of the hardware model as we have coded. So, we have discussed various codes. To check the functional correctness of all the codes that we have written till now or any other code, we have to use a test bench.

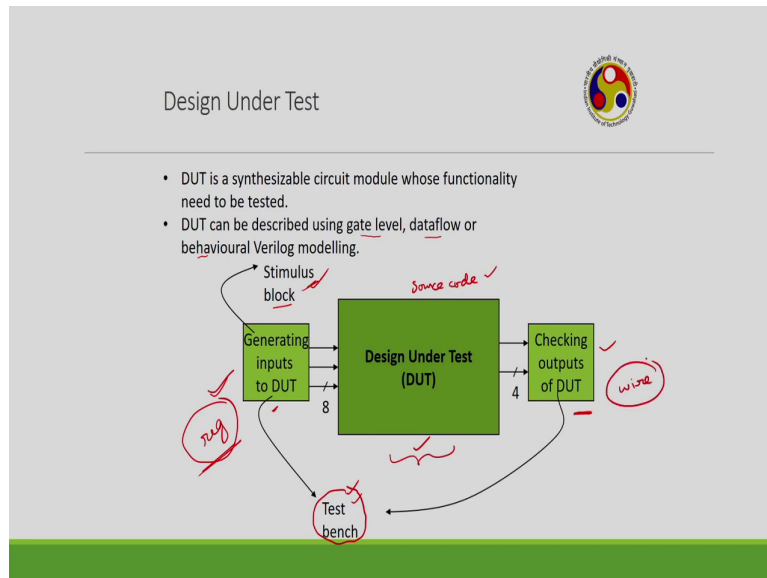
Then, this is also a VERILOG code and we have written the VERILOG code for the different combinational sequential circuits. Then, what is the difference between this test bench VERILOG code and the VERILOG code written for various combinational sequential circuits?

So, the main difference is actually normal HDLs are synthesizable. In the sense, so after writing the program we will check for the syntax or logical errors. Once the syntax or logical errors are corrected, then we will transform this into an optimal circuit design. So, this type of code is called as synthesizable code; whereas, the code that is written for this test bench need not be a synthesizable one.

So, because test bench is basically used to check or simulate the HDL source code. So, you saw the original code that we have written. So, it is the reason why the test bench code need

not be synthesizable and the test bench is a powerful tool for auto generation of test stimulus and test results. This will be clear in the next slide.

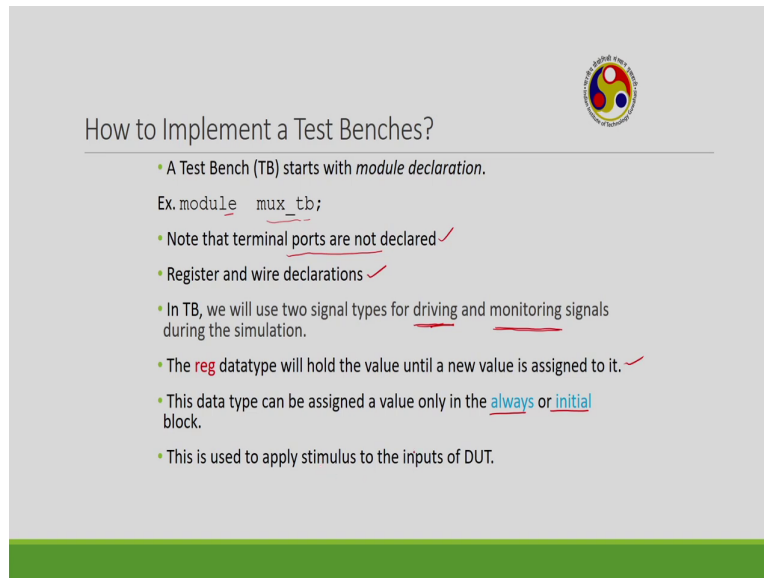
(Refer Slide Time: 02:53)



So, this is the complete setup of the test bench. This is the design under test. So, whatever the code that you have written, so you how to test that design you have to take here. Then, you have to apply some stimulus and then, you have to check the outputs. So, these two stimulus and the checking of the outputs can be performed by using test bench. So, this is called source code and this is synthesizable.

So, this source code can be either in the form of a gate level modeling, data flow modeling or behavioural modeling which we have discussed in the earlier lectures. So, to check this source code, we need to apply some stimulus and then, you have to check some outputs. So, the application of the stimulation checking of this result can be performed by using test bench. Now, what are the various steps involved in the test bench?

(Refer Slide Time: 04:01)



How to Implement a Test Benches?

- A Test Bench (TB) starts with *module declaration*.

```
Ex. module mux_tb;
```

- Note that terminal ports are not declared ✓
- Register and wire declarations ✓
- In TB, we will use two signal types for driving and monitoring signals during the simulation.
- The reg datatype will hold the value until a new value is assigned to it. ✓
- This data type can be assigned a value only in the always or initial block.
- This is used to apply stimulus to the inputs of DUT.

How to implement the test bench? So, the starting point of this test bench is we have to declare the module. So, this is similar to the source code also. In source code also, so initial step is we have to declare the module. But the main difference between the module declaration in the conventional source codes and the VERILOG code of the test bench module is here, we will not declare the ports. This is one of the important point.

So, in the module declaration of a test bench, simply you have written a module mux underscore test bench. So, we have not declared the inputs and outputs. So, what are the inputs of the multiplexer, what are the outputs that we have not declared in the module declaration of the test bench; whereas, in conventional VERILOG codes, so after the module you have to give some name followed by the port declaration.

So, which signal will acts as input, which signal will acts as a output, which signal will act as a in out means dual port and here, we are going to use registers and wires. So, in conventional VERILOG codes also, we will use registers and wires. So, here the purpose of the registers and wires are they are used for the driving and monitoring of the signals.

So, driving is basically to apply the stimulus; monitoring is to apply to the displays. So, as we already discussed that the register data type will hold the value until the new value is assigned to it. This will acts as some sort of a register. This data type can be assigned a value only in

the always and initial block and this data type is used to apply the stimulus. So, here, so this register data type will be used to apply this stimulus.

(Refer Slide Time: 06:16)

- The **wire** datatype is similar to that of a physical connection. It will hold the value that is driven by a port, assign statement, or reg.
- This data type cannot be used in **initial** or **always** blocks.
- This is used to check the output signals from the DUT.

Ex. `reg select,y0,y1;`  
`wire out;`

- Next, DUT instantiation is used
- The purpose of a TB is to verify whether our DUT module is functioning as we wish.
- Hence, we have to instantiate our design module to the test module.
- The format of the instantiation is:

✓ `<dut_module> <instancename>(<dut_signal>(<test_module_signal>),...);`

Ex: `mux m1(select,y0(y0),y1(y1),out(out));`

- We have instantiated the DUT module `mux` to the test module.
- The signals with a dot in front of them are the names for the signals inside the `mux` module.
- While the **wire** or **reg** they connect to the test bench is next to the signal in parenthesis.

On the other hand, wire, the second type of the data type is nothing but a physical connection, which we have discussed several times in the previous lectures. So, it will hold the value that is driven by the port and here, this data type cannot be used with initial and always block.

This is another important difference between the register and wire. So, register will be normally used with always and initial blocks; whereas, wire will not be used with initial and always blocks. And this wire is used to check the output. So, here, register used to apply the stimulus.

So, the test bench basically will perform two basic operations; one is applying the stimulus and checking the outputs. So, to apply the stimulus, normally, we will use register; to check the output, we will use the wire. If I take an example of a 2 by 1 multiplexer and here, I am calling the inputs as `y 0`, `y 1`.

Different notations you can use anything. `y 0`, `y 1` are the inputs and there is a select line and there is a output called `out`. This is 2 by 1 multiplexer. So, here these inputs will be applied

through the stimulus. So, we have to define this  $y_0$ ,  $y_1$  as register and also, select because these are the inputs. Output will be declared as a wire.

So, this is the second step. The first step is we have to declare the module and the second step is you have to define the registers and wires. Normally, all the inputs will be declared as a register, output will be declared as a wire. The third step is DUT instantiation, this is a important step. So, we know that the purpose of test bench is to verify whether the designed DUT is functioning properly or not. So, we have to instantiate our design onto the test module.

So, this will be clear how this instantiation will be done, if I give some example. The general format of the instantiation is we have to write the device under test modules name and then, instance name followed by a dot. Here, there is one dot is there. We have to give some signal and then, within parenthesis, we have to give test module signal. This is device under test signal and this is test module signal. So, these two can be same or you can give different names also. If I take say for example, of the multiplexer.

So, here this multiplexer, I am giving the name as the device under test model name is multiplexer; instance name is m1. These are optionals. Then, we have the select, so the name that is followed by this dot, this select, this represents the names of the signals inside the multiplexer code.

So, this is the name of DUT and their signal. And then, the name within the parenthesis represents that of the test module signal. So, we can give both the names same or different also. So, in order to connect this signal to the output, we will use wire or register, they will connect the test bench to the signal in the parenthesis.

So, in order to connect this signal in the parentheses, so the parenthesis signal is test module signal; whereas, this is actual DUT signal. So, this can be connected through the wires or registers. So, this select and this select will be connected through this register select declaration and this  $y_0$ ,  $y_0$  will be connected through this again register declaration;  $y_1$   $y_1$  is connected through this register declaration and this out will be connected through this  $y$  declaration.

So, basically we have one is design under test; another is test module design under test is having 3 signals. So, this is our design under test; 3 input signals y 0, y 1, y 2; y 0 y 1 select and then, one output y. So, overall, if I want to connect this to the test bench, test bench will be having some 3 signals, inputs, one output. So, the name of this one also same as this you can give y 0, y 1 select or you can give different name also. In that case, we have to change this.

So, now, how to make this connection between these by using this register connection? So, if I give the register, if I declare this as a register, this will connect to this. If I connect this as a defined as a register, this will connect to y 1. This will connect to select if I give a register.

Similarly, the final output of this test module we are calling as out only. You can give different name also. So, these two can be connected through a wire. So, this is how this instantiation of the DUT module will be done. This is one of the important step in writing the test bench.

(Refer Slide Time: 12:33)

```
module test_module_name;
    // Declare local reg and wire identifiers.
    // Instantiate the design module under test.
    // Specify a stopwatch, using $finish to terminate the simulation.
    // Generate stimulus, using initial and always statements.
    // Display the output response (text or graphics (or both)).
endmodule
```

So, if I take the general the summary of the test bench module, so this is the summary. So, the module, we have to give some name. There is no port declaration and then, end module. In between, what are the various steps as I have defined? The first step is port declaration, second step is we have to define registers and wires then, we have to instantiate the design

module under test. Then, we have to specify the timings. This, I will explain while explaining you some examples.

Then, you have to generate this stimulus using the initial and always statements. By default, these waveforms will appear on the screen. If we want the numerical values, we can use some display commands.

So, display the output, we can display either in terms of the waveforms or in terms of the values. If you want the values then we have to use some commands, by default, you will get the waveforms. So, you see the step by step procedure, we have to use to write a test bench. So, with this background, I will go to some examples of the test benches.

(Refer Slide Time: 13:53)

```
AND gate example

module AND_2_behavioral (output reg Y, input A, B); always @(A
or B) begin
if (A == 1'b1 & B == 1'b1) begin
Y = 1'b1;
end
else
Y = 1'b0;
end
endmodule

#include "AND_2_behavioral.v"
module AND_2_behavioral_tb;
reg A, B;
wire Y;
AND_2_behavioral Instance0 (.Y(Y), A(A), B(B));
initial begin
A = 0; B = 0;
#1 A = 0; B = 1;
#1 A = 1; B = 0;
#1 A = 1; B = 1;
end
endmodule
```

Source HDL code

Task bench Verilog module

#

So, I will start with a simple AND gate. So, this AND gate is having two inputs; A, B and 1 output Y. This one is the source HDL code and this one is test bench; both we have written using Verilog only. This also we have written using Verilog, this also we have written using Verilog. I have already described the differences between these two Verilog codes.

So, this is behavioral model of 2 input AND gate. So, we have defined the output register as Y input is A AND B, then always at A OR B. So, whenever A changes, our B changes. So, we have to begin. So, what is the operation if A is equal to 1? This is binary value of 1 and B



is equal to binary value of 1. Then, output is equal to 1; else, output is equal to 0. This is a simple description of this AND gate, then end module.

Then, what is the corresponding test bench corresponding to this AND gate. So, as you have told the first step is the declaration of the module. So, this is the model, I have given the name as AND 2 behavioral tb.

But there is no port declaration. So, there is difference between this and this is here, we have port declaration; here, we do not have port declaration. Then, we have to define registers and wires. As I have told inputs will be defined by the registers, output by wire.

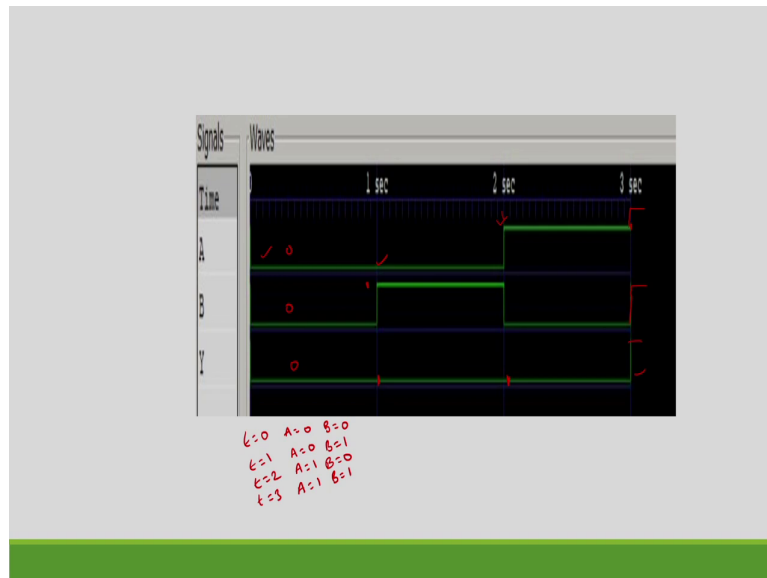
So, the inputs A B are registers, output is wire, is Y. Then, the next step is you have to instantiate it. So, we are instantiating a 2 input and gate this one. So, we are assuming that this is the input of the test and even test bench also we having the same the inputs, even the test bench also having the same inputs as capital A, capital B and Y.

So, in that case, dot Y we have to apply to the Y, A to A, B to B. Suppose, instead of this if I use some say P, Q, R; say instead of using A, B, Y if I use P, Q, R as I have told here. So, this first name is DUT and second name is test bench within the parenthesis.

So, in that case, what we have to do is we have to change this to the test bench values Y is R, this A is P, B is Q or we can use the same symbols also, then initial begin. So, initial always will be executed at the time t is equal to 0; the time scale.

So, at t is equal to 0, A is equal to 0, B is equal to 0. So, this represents the delay as we have discussed in the earlier lectures also. So, a delay of 1 second; after a delay of 1 second, A is equal to 0, B is equal to 1; after another 1 second A is equal to 1, B is equal to 0; after another 1 second A is equal to 1, B is equal to 1; then, end and then end module. This is the test bench correspond to a simple 2 input AND gate.

(Refer Slide Time: 17:48)



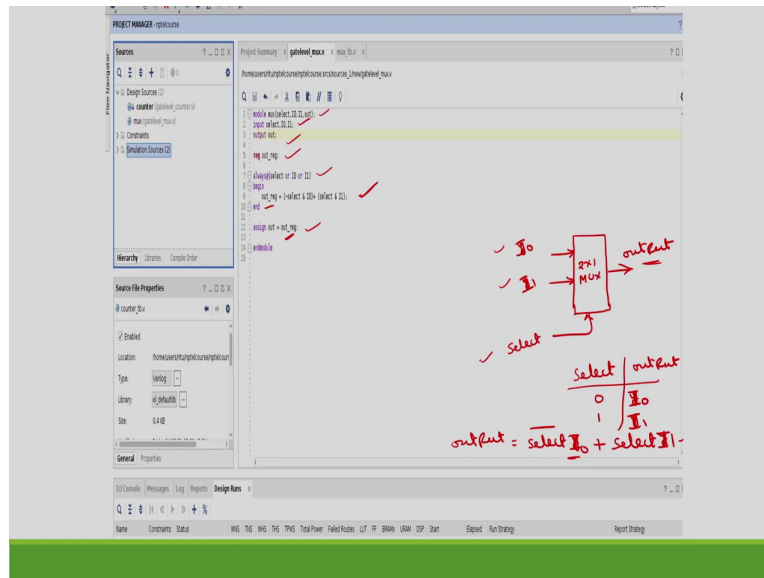
So, if we take the test bench waveforms. By default, the waveforms will be displayed on the screen. So, as you have written this test bench such that initially A is equal to B is equal to 0. Initially, at t is equal to 0, A is 0, B is 0; obviously, output is 0 and after 1 second, 0 1 1 0 1 1 pattern after every 1 second.

So, t is equal to 0, A is equal to 0, B is equal to 0. At t is equal to 1, A is equal to 0, B is equal to 1; t is equal to 2, A is equal to 1, B is equal to 0; t is equal to 3, A is equal to 1, B is equal to 1. That is we can see from the waveforms.

At t is equal to 1, A is 0, still B is 1, so output will be 0 only. At t is equal to 2, this is second instant 2 seconds A is equal to 1, A is changed to 1 and B is changed to 0, still output will be 0.

At the third second at t is equal to 3 seconds, so this will also 1, this will also 1, so the output is also going 1. You can see here, there is a transition. If we want this time scale as nano seconds or pico seconds, initially you can define that. So, we see about the complete three different modules. So, the source code, test bench and then, simulation waveforms. So, this will complete the testing of a given design under test.

(Refer Slide Time: 19:46)



I will consider some more examples. So, this is another example of 2 by 1 multiplexer. As I already explained in the earlier slides, this is 2 by 1 multiplexer, inputs have labeled as y 0, y 1, this is up to you, And select line by using select only; output is y.

So, this is actually Verilog module. So, this module multiplexer, select I 0 I 1, then out. We have declared the ports; whereas, in test bench, we are going to see in the next coming slides. So, we do not have this port declaration. Then, the inputs outputs we have defined. Then, output we have defined as register also. So, always at select or I 0 or I 1.

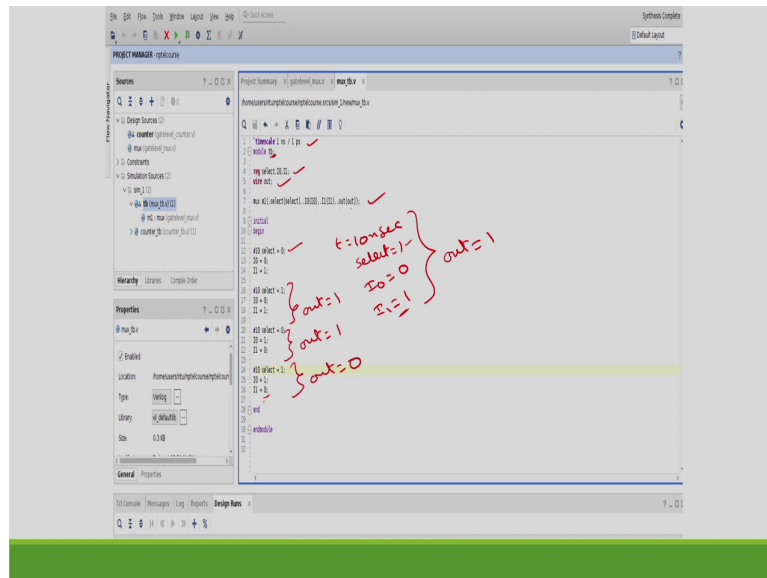
So, whenever anyone of this three selects, then the output will change. Output is equal to select and with I 0 select and with I 1 because we know that this expression already I have given in the previous lectures. So, what is the operation of this? This select is going to decide the output y. If select is 0, sorry this is output. Output is y 0, select is 1, output is y 1.

So, what is the Boolean expression for this output? Is select bar into y 0 plus select into y 1. So, this is what we have rearranged here. So, this select bar into I 0 y 0 plus select into y 1. Here of course, instead of y 0, y 1, I think here the inputs are I 0, I 1.

So, in the previous example, I have written as y 0 y 1, but here in the program this is written as I 0 I 1. So, this is I 0, I 1. Then, so, we have to assign the output as output register, whatever this we have defined that is output, this itself is the output. So, this is a simple

Verilog code for a 2 by 1 multiplexer. Then, the next step is what is the corresponding test bench.

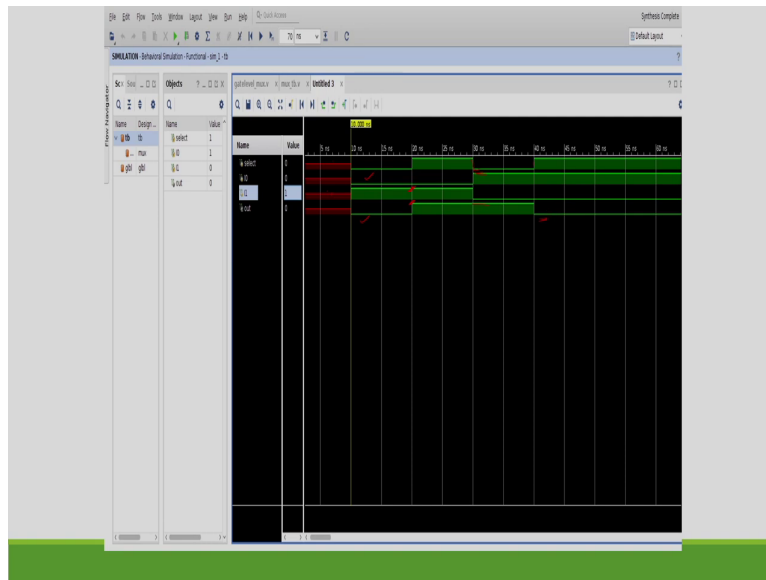
(Refer Slide Time: 22:44)



So, as you have told, here I have defined the time scale as 1 nano second or 1 pico second, then module tb, I have given just name as test bench; I have not given any multiplexer and this is up to you. So, you can give any name. So, then, the inputs will be registers I 0, I 1 select; output will be wire out. Then, you have to instantiate it. So, I am using the same names. So, select select and then I 0 I 0, I 1 I 1, out out.

So, then, initially when select is equal to 0 at t is equal to 10 nanoseconds. I am making select is equal to 1 and I 0 is equal to 0, I 1 is equal to 1. So, that means, what will be the output? Output will be select is equal to 1, I 1 is selected 1. Similarly, here, what will be the out? Select is equal to 1 means I 1. What is the I 1 value? 1. Here select is equal to 0 means I 0. So, out is 1. Here, select is equal to 1, so I 1 will be selected, I 1 is 0; so, output is 0. So, for every 10 nanoseconds, we are going to change the select values and I 0, I 1; correspondingly, the output value will be displayed.

(Refer Slide Time: 24:21)



So, the third step is if we check this test bench waveforms. We can easily see that the select is initially high and this is I 0, this is I 0, this is I 1, this is out. So, when high I 1 is selected, I 1 is high, so this one is also high. After 10 nanoseconds, select is 0. Select 0 means I 0 will be selected, I 0 is also going to go 0, this I 0 is 0, so output is also 0. At 20 nanoseconds, select is 1, so I 1 is selected.

So, I 1 is high, so this is also high. At 30 nanoseconds, select becoming 0; 0 means I 0 is selected, I 0 is 1, so this one will continue. So, at 40 nanoseconds, select is high, I 1 is selected, I 1 is 0, so this is going to 0. This is how we can test the functionality of the circuit.

(Refer Slide Time: 25:26)

Full adder *gate level modeling*

```
module fa(a, b, c, sum, carry);
input a;
input b;
input c;
output sum;
output carry;
wire d,e,f;
xor(sum,a,b,c);
and(d,a,b);
and(e,b,c);
and(f,a,c);
or(carry,d,e,f);
endmodule
```

*(TB)*

```
module fulladdt_b; //testbench
reg a;
reg b;
reg c;
wire sum;
wire carry;
fa ut(.a(a),.b(b),.c(c),.sum(sum),.carry(carry));
initial begin
#10 a=1'b0;b=1'b0;c=1'b0;
#10 a=1'b0;b=1'b0;c=1'b1;
#10 a=1'b0;b=1'b1;c=1'b0;
#10 a=1'b0;b=1'b1;c=1'b1;
#10 a=1'b1;b=1'b0;c=1'b0;
#10 a=1'b1;b=1'b0;c=1'b1;
#10 a=1'b1;b=1'b1;c=1'b0;
#10 a=1'b1;b=1'b1;c=1'b1;
#10 $stop;
end
endmodule
```

*Sum*  
*Carry*

So, if you consider another combinational circuit which is a full adder. So, we know that full adder will be having 3 inputs and 2 outputs; a, b, c, sum and carry. So, here, I have written this as gate level. As you have told we can write this code using any of the three levels either behavioral or gate level or data flow modeling. So, you can write this code using any of this modeling either behavioral modeling, data flow modeling or gate level modeling. Here, I have used gate level modeling.

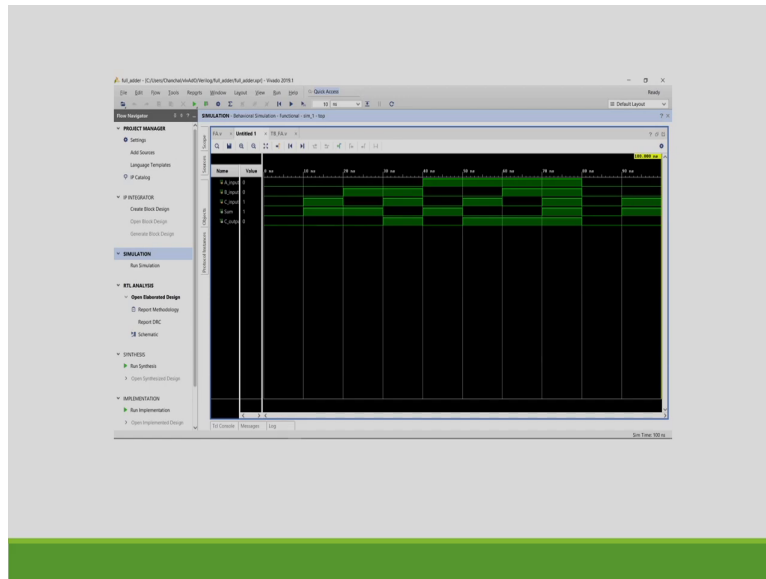
So, module name, full adder; then a, b, c, sum out; a we have defined as input port at output port. This is as usual and the wires, we have defined as d, e, f. So, why the wires are required? Because we are going to use this gate level, I should have this circuit diagram of this full adder.

So, this is a, b, c; sum is exclusive or between a, b and c and carry is a b, b c, c a and then finally, you have to or, this is carry. So, here we require 3 wires; d, e, f. So, we have defined the wires as d, e, f. Exclusive or will give directly sum; d is nothing but a b output; a b output is d; b c output is e; c a output is f and finally, we have to take the OR operation between this it will give carry and then end module.

Next coming for the test bench, this is test bench corresponding to this Verilog module. So, as you have told the inputs will be declared by registers. So, a, b, c are the inputs; sum and carry

are the output. So, wires and we are instantiating. So, we are giving the same name. We can give different names also as we have discussed in the earlier example. Then, initial begin. So, initially, at 10 nanoseconds or 10 seconds, we have given a as this is binary representation is 0 0 0. So, we have given all the 8 combinations. After 10 seconds, we are going to stop, end module.

(Refer Slide Time: 28:30)



So, if you see the simulation waveforms, we can verify that. So, here, depending upon the a, b, c values you can find out the sum and carries. You can easily verify the functionality of this full adder. So, these are the few examples of the test benches.

In today's lecture, we have discussed about only simple test bench circuits. So, in the coming lectures, we will discuss some more complicated including the test benches for the sequential circuit and all.

Thank you.