

Digital VLSI System Design
Dr.S.Ramachandran
Dept of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 16

Writing a Test Bench

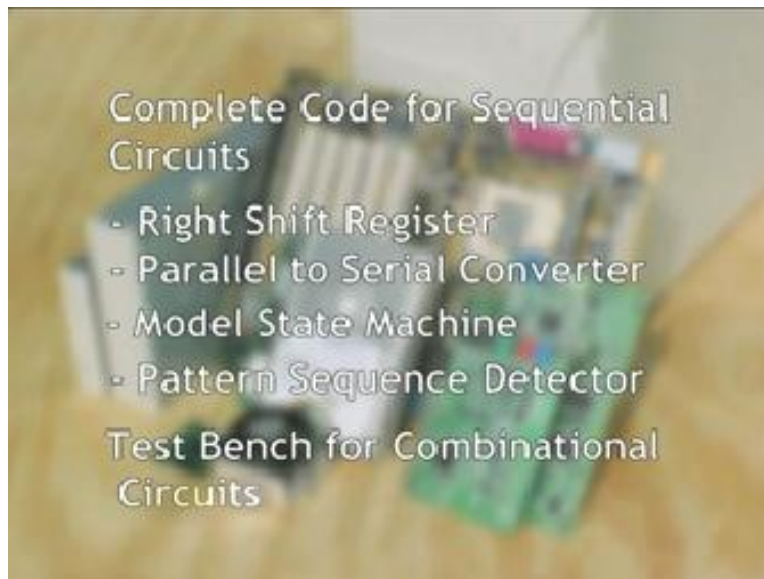
Slide – Summary of contents covered in previous lecture.

(Refer Slide Time: 01:04)



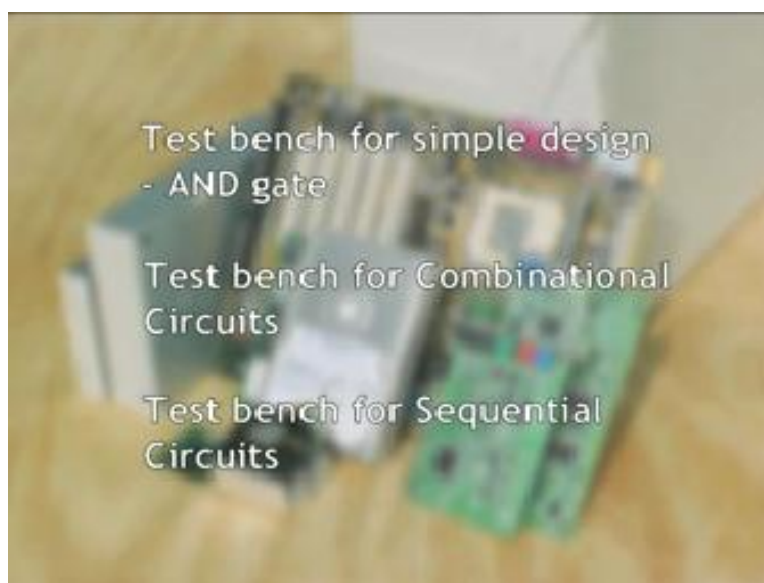
Slide – Summary of contents covered in previous lecture.

(Refer Slide Time: 01:08)



Slide – Summary of contents covered in this lecture.

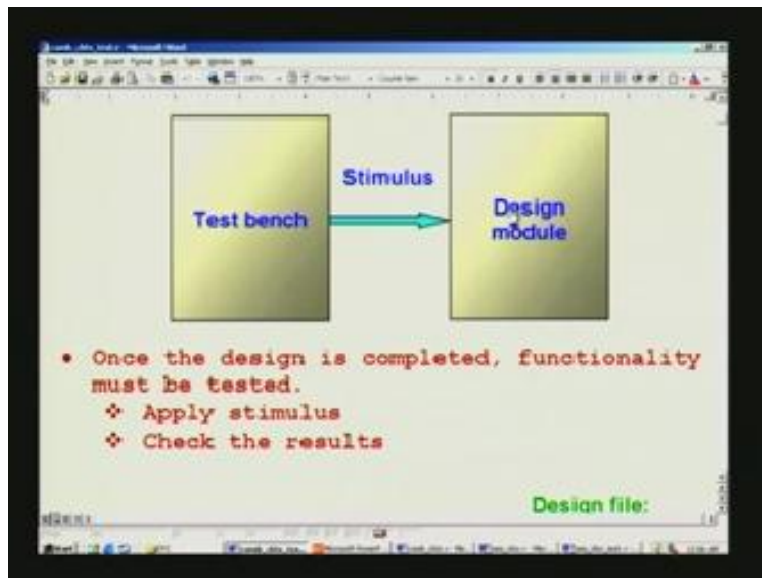
(Refer Slide Time: 01:39)



In the earlier talks, we saw how to write a test bench. We just started writing for a combination circuit. Some of you felt that the signals involved were rather large. I am presenting now to start with, a more simple design. We have seen earlier what a test bench is. It is basically a block; you

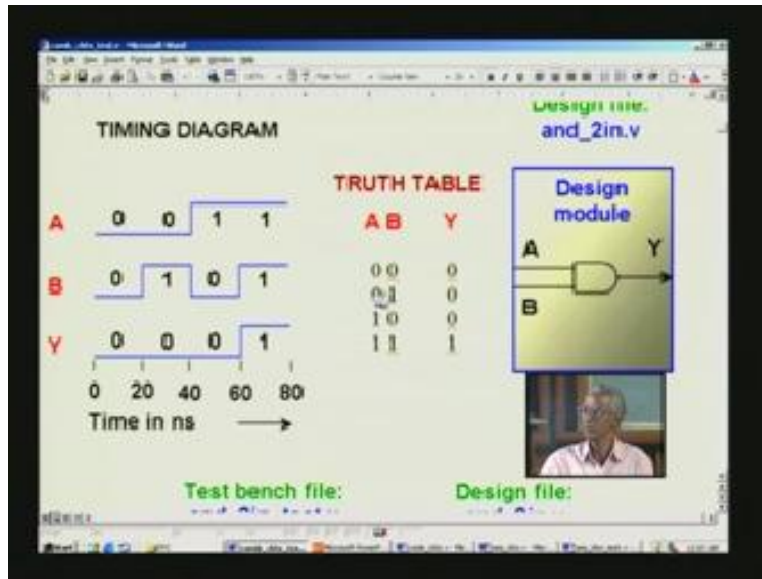
have to code it using the Verilog. You have a design which is required to be tested. This is the design you had done. You need to test this.

(Refer Slide Time: 02:52)



How do you test this? You have to apply what is known as stimulus. This is nothing other than application of inputs in your design module. Finally, you have to see the outputs and we will see in a minute how we do that. Here, we have seen that once the design is completed, the functionality must be tested. Therefore, we apply stimulus and finally we check the results. We have seen this earlier. We will start with simple example. Let us say the simplest possible example can be just an AND gate; your design has nothing more than 2 input AND gate. It has 2 inputs A and B, and its output is Y. All these nomenclatures pertain to the design module alone and not to the test bench. In the test bench, we can rename this as we like and as per our convenience. We need to test this.

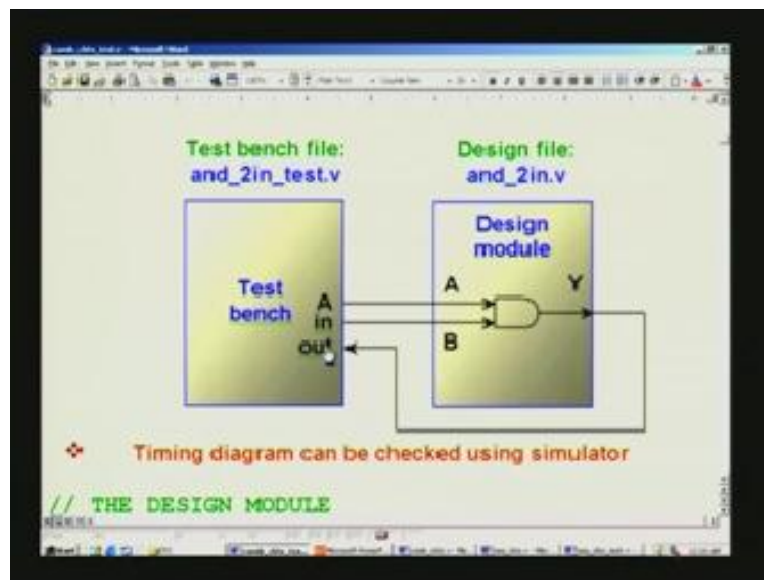
(Refer Slide Time: 04:05)



How to test this functionality? If you had verified that the truth table is working for all its combinations, then it is tested. For example, in this truth table for AND gate is given here for various combinations of A B. You can also depict the same thing in terms of a timing diagram, which is shown on the left here. A, B, and Y are exactly the same sequence, except you need it to read it vertically here, instead of horizontally. These 3 entries triple 0 here. That is precisely appearing here, next entry 0 1 0 is here and 1 0 0 forward by triple 1 here, exactly the same thing. Now, you also see one more time max is here, because once you set timing diagram, you need to refer to a base. Let us say, we start arbitrarily at 1 point, we refer it to as 0 point. Time starts here (Refer Slide Time: 04:55). It always goes forward fundamentally, unless you take the reference somewhere mid-point. To start with, we will simplify that saying that this is going to start at 0. Let us say that time axis is marked in nanosecond. You see that 20 nanosecond, we make a second transition here. So, actually, if you see here, 1 is naturally pointing to the high state here. That is why, we say it as 1. So, 0 1 and 0, it is flat here at the low level. Once again, similar explanation holds good here. At 40 nanoseconds, we have made another change to 1 0 0 and 60 triple 1. This can go on and on depending upon how complex the circuitry is. As far as this is concerned, you can deem it to be a complete test, if you had tested all these 4 combinations here. We will apply the very same block that we had seen earlier. Now, this is the test bench. This test bench as I mentioned, it is also a Verilog file and naturally with a dot v extension. Let us say, we name it as and underscore 2 in underscore test dot v, retaining the same convention that I have

been telling earlier. Since the design is actually this and 2 in which is here and what we have already seen in timing diagram; truth table, etc. pertaining to this and gate.

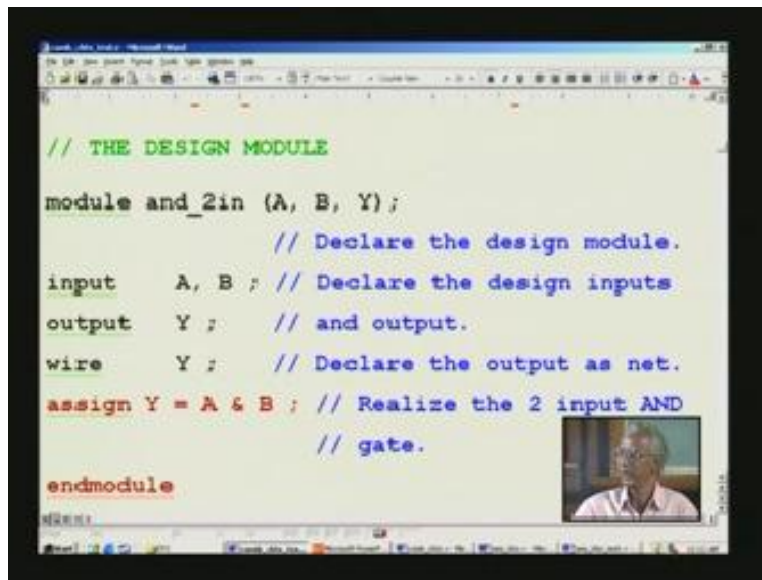
(Refer Slide Time: 07:03)



I have retained once again the very same labels there for IOs. Here, you would notice that I have renamed B input in the test bench as 'in'. Similarly, 'out' also, I have renamed here. In fact, you can rename A also; any meaningful names can be given for these ports. Once I say ports, it refers to either input or output. You can check the timing diagram as we have seen earlier, exactly the same thing you can see in a simulator, which will be covering in the next lecture most probably. That is what is mentioned here and we have a design module to start with. This is the design module and the code verilog code for that is quite simple; you have only to list. First, identify the the design module, which is 'and underscore 2 in'. This is as usual, a module and end module, which defines that there is a design module and you need to list all the IO's here; say A and B are inputs and we declare here. Notice that, now, it has been put in a single line, whereas earlier, we used to put line by line. This is because, it is a very small design; you can as well club all the inputs and outputs. Output happens to be just 1. Note that every statement is terminated by a semicolon. You have also to describe what Y is. Since we are going to use just an assigned statement, which we have already seen for implementing this AND, which is nothing but Y equal to A AND B. This is the symbol for AND, which we have already familiar with. This is declared as a wire, because it is an assigned statement. In design, it will be wire and in test bench, it will

be different thing. We will have a look at the 2 and make a comparison. The comments are all self explanatory.

(Refer Slide Time: 08:51)



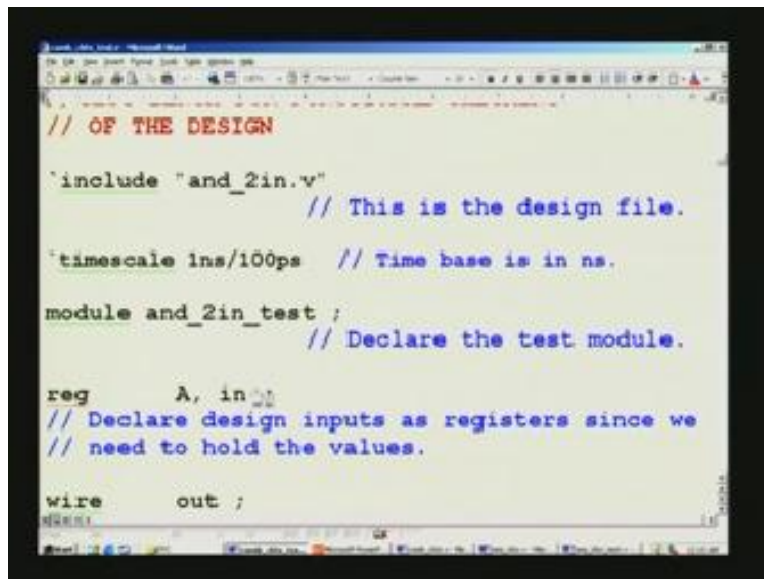
```
// THE DESIGN MODULE
module and_2in (A, B, Y);
    // Declare the design module.
input  A, B ; // Declare the design inputs
output Y ; // and output.
wire  Y ; // Declare the output as net.
assign Y = A & B ; // Realize the 2 input AND
                    // gate.
endmodule
```

This is what we have already seen in the block here. This is the design. Coming to the test bench, this is what we wanted to learn because signals were too many in the previous lecture for combination circuits. We take just the simple thing and write only a full fledged test bench for this particular application. Then we resume where we left the combination circuit. Now, we are going to see the test bench. As I mentioned before, this file is dot v extension and this file is different from this. Although you can put both in a single file and then call stimulus as you have see in general textbooks. I do not follow this, because our ultimate goal is to design VLSI systems. If you see the length of the code, it will be enormous. If you take this approach of stimulus, you will run into difficulties because you cannot make out what you are doing. This is because that you are indulging in so many signals. That is why we will straight away even for a simple example like this we will adopt the right approach, which is applicable for any size file; any big project can also be covered in the same way.

Here, I have mentioned input A is still retained as A and input B in the design is renamed within the test bench as 'in'; similarly, the out connotates this Y of the design here (Refer Slide Time: 10:30). We have seen statement by statement. It is confirming to the design we have already

pictorially depicted that. Now, we will start on the test bench. This is a test bench for functional checking of the design. So, put it on comments and all titles, you can use comment like this. We need to include the design file. That is why we have a reverse tick here and the inclusion of the file name. This is similar to the C structure; remember the codes here. To describe that, this is a design file. Along with the dot v extension, you have to give. If you forget dot v, then it will complain, that is, the compiler will complain. This is the design file. We have also a time base.

(Refer Slide Time: 11:38)

A screenshot of a text editor window showing Verilog code for a test bench. The code is as follows:

```
// OF THE DESIGN

`include "and_2in.v"           // This is the design file.

`timescale 1ns/100ps         // Time base is in ns.

module and_2in_test ;
    // Declare the test module.

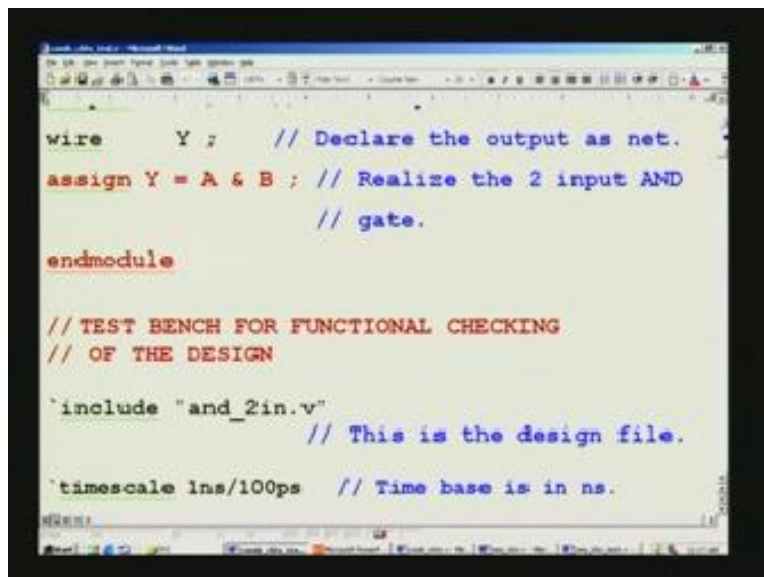
    reg    A, in0;
    // Declare design inputs as registers since we
    // need to hold the values.

    wire   out ;
endmodule
```

We have already seen in the timing diagram that we have marked nanoseconds, and so on. This timing will be covered in this test bench. How is it done? We will have a look at it. For that, we need to declare a time base. That is why this has been done. Again I tick here, then say, time scale without any break here, single word as such. The time base is actually mentioned here. If you want 1 microsecond, then you give 1us. There is no μ in the normal character that you have on the keyboard. So, use 'u' there and for Pico seconds, p. Similarly, m for millisecond, and so on; different timings are there. You can refer to a standard verilog textbook to get little more information on this. How much accuracy resolution you want for this is specified here. This happens to be a point 1nano second. You can easily get just by dividing whatever number you have by 1000 in order to step down to next scale. That is, from Pico to nano, it is a 1400 difference. It is very handy to divide by 1000, so that you can quickly know if you want to convert it to nanosecond. Declare the module for this test bench; we have to declare the module

once again, just as you declared for the design file. This is the place we are doing it. This is named as combination circuits, because I used this very same file. Therefore, it appears to be different here. Module name and this need not be same as mentioned I earlier, but it is good practice to maintain the same name. Otherwise, you cannot keep track of it. Just to clear your doubts, I added this. This appears to be a little extraneous. Anyway, you kindly bear with me for that difference and we need to declare the test module. No dot v extension here, it is only the module name; it is not the file name. We just stick on only to this bare minimum without the extension. Some of you did not understand why reg wire is here.

(Refer Slide Time: 14:02)



```
wire    Y ;    // Declare the output as net.
assign Y = A & B ; // Realize the 2 input AND
                    // gate.

endmodule

// TEST BENCH FOR FUNCTIONAL CHECKING
// OF THE DESIGN

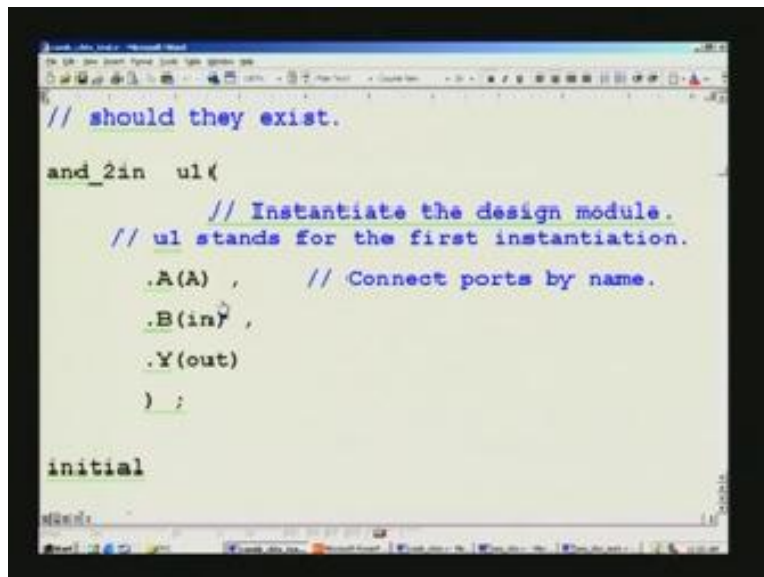
`include "and_2in.v"
                    // This is the design file.

`timescale 1ns/100ps // Time base is in ns.
```

It was already explained here; I will explain it once again. Here, if you see the timing diagram, what we need to do here is, we apply this stimuli; A B are stimuli here. We apply here and hold it till you want to change here at 20 nanoseconds (Refer Slide Time: 14:15). You have to tell the system to hold the value till I veto that. That is the meaning of register in this test bench. That is what we mean, we want to hold A and in. You notice that we are declaring in here not as B, because B pertains to the design, whereas this pertains to the test bench. It is exactly the same signal, but we have renamed it as in here for the sake of test bench. That is how we put reg. In order to hold the values till you change it later on, we need to declare it as a reg. Once again, its expansion is register; it is only to say that it memorizes and wire we have like this. I will come back to this once again, because unless I explain this, I cannot explain this. Now, you have to call

the design module. That is called instantiation in Verilog coding. This is the design module we started with dot v extension, but inside, we had declared the module name as this alone. When you call any of these design modules, you call only by this module name and not with the file extension. I hope you are clear about the file, discriminate the file, and module. As mentioned earlier, we can have like this any number of very same modules called several times. Each time you call, how do you distinguish it? Distinction is done by this u_1 , u_2 , and so on. It is similar to having different packages on circuit board, which we have already seen earlier. Another thing mentioned earlier was also this, what is called connect ports by name. If you want to connect any IO port, say for example, A B Y are all pertained to the actual design there, whereas A in out corresponds to the test bench here. This is the nomenclature we adopt here. The advantage of this one is, you can put A at the last or in between or anywhere. Suppose you forget, it is very easy for us to remember names rather than the order in which we wrote. If it has some 20 signals or 30 signals or even more, you get drowned in which order you had mentioned earlier in the design. Most probably, you will make the mistake.

(Refer Slide Time: 17:12)



```
// should they exist.

and_2in u1(
    // Instantiate the design module.
    // u1 stands for the first instantiation.
    .A(A) , // Connect ports by name.
    .B(in) ,
    .Y(out)
);

initial
```

Changing order: If you change the order, the whole thing will be chaotic. So, we call that by name. That is what it says it here; connect ports by name. You should be clear on this core, because this was not clear earlier since there are too many signals that are involved. In fact, you can have a look at the block diagram is started; then it will become ultra clear. In test bench, we

use A in out for the IO's, whereas we use A, B, and Y in the design module and that is what we have already seen here. In design module, A B Y is used and in test, we need to use A in and out. Notice that there is no such thing such as input or output declaration, because this happens to be the top most level. There is no further higher level than this. Everything is contained in this level. Then what is the need for returning any IO's? IO's imply, it is going to the external world. You remember what all we are doing is, we are making a chip. Although, it resembles a c program, it is not c program. It is a hardware design language. So, what ultimately resides in the chip will be something like this. Even this is not correct; actually, transistors take its place. This is only symbolic again here. For circuit designers, they are familiar with this more than transistors. What I was saying is, this one nanosecond is the actual time base. In the timing diagram, you have seen here, from here to here it is marked (Refer Slide Time: 18:57). Suppose you have 20 there. This time base is in nanosecond, you will have further smaller marking also in that. You will see in the simulator, when you really simulate all this. However, fundamental unit is in nanosecond. That is what it means, but suppose it can show 1 nanosecond here as a finer scale. It is a major graduation. It is something like your ruler that you have a scale. Similar to that, what you will see in simulator is also like that. Although what you are seeing outside is 1 nanosecond, what is the actual resolution accuracy? So, that is what is implied there by that Picosecond, which is the second here. It simply means, it is accurate up to point 1 nanosecond. That is, 1 nanosecond is accurate to point 1 nanosecond. Otherwise, you will get a doubt whether it is 1 nanosecond or 2 nanoseconds. That is the difference.

(Refer Slide Time: 20:05)

```
// OF THE DESIGN

`include "and_2in.v"
// This is the design file.

`timescale 1ns/100ps // Time base is in ns.

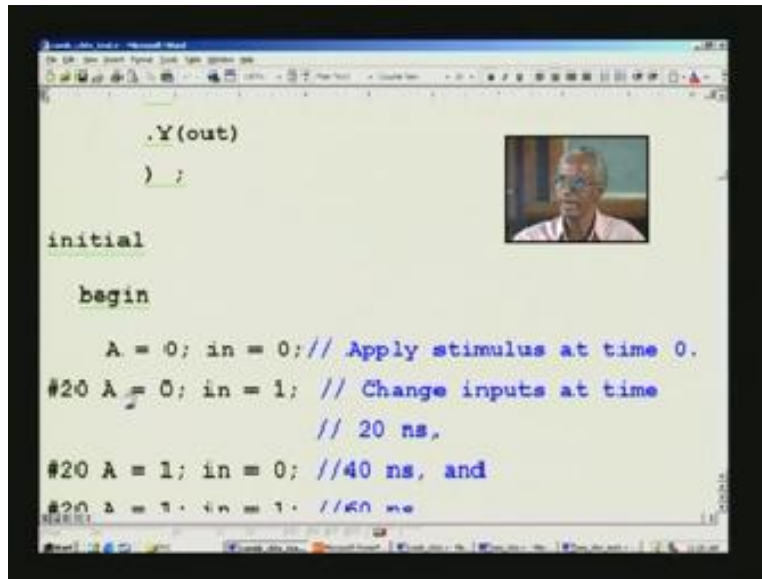
module and_2in_test ;
// Declare the test module.

reg    A, in ;
// Declare design inputs as registers since we
// need to hold the values.

wire   out ;
```

Now, we start on with a new thing what is called initial. So, these are basically behavioral statement. When we say initial, what we mean is, we want to apply those stimulus. Here, this is the core for the test bench. In the timing diagram, what we wanted is, here we wanted to apply in the first instance at 0 nanosecond A and B to be 0 0 each. After 20 nanoseconds, 0 1; so you just remember every 20 nanosecond here we wanted to apply. That is what we are going to do now. How to write that? It is quite simple. We are in 0 nanoseconds if it is not mentioned. There is no harm in mentioning 0 nanoseconds. How to mention? We will see it shortly. We wanted A to be 0 and in to be 0. Remember that we are in the test bench, so use in, not B. Otherwise, it will report error. Apply 0 0. That is what you want for the first combination and it should be at time 0. Next thing, you want to apply 0 1. At what time? After 20 nanoseconds. Mention that this is the special marking for indicating that it is time. This is how verilog recognizes that you are mentioning after a delay of 20 nanoseconds, then only I need to apply this; after 0 nanosecond, you apply this; it interprets like that. When it comes to this statement, it says wait for 20 nanoseconds. After doing this one, immediately it comes to this statement, but it cannot proceed; because you are saying wait.

(Refer Slide Time: 22:00)



```
.Y(out)
);

initial

begin

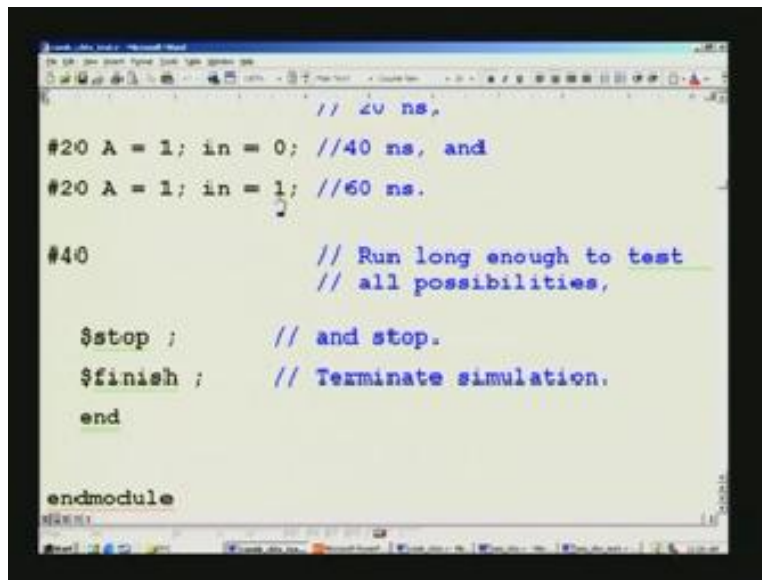
    A = 0; in = 0; // Apply stimulus at time 0.
#20 A = 0; in = 1; // Change inputs at time
                // 20 ns,
#20 A = 1; in = 0; //40 ns, and
#20 A = 1; in = 1; //60 ns
```

That is what you mean by this. Wait for 20 nanoseconds and then apply the fresh input. By doing so, you are getting precisely the same waveform that you have already seen earlier. So, this is the case for other combinations 1 0, then 1 1. One more thing I need to mention is, we started with 0 nanoseconds and then we went onto 20 nanoseconds. What will be the time here? It says 20, but it is not 20. It is cumulative. 20 plus 20 we have to take and that is what you understand here. You can see the sequential nature here, if you use this timing element. On the other hand, if you do not put any timing, all this will be processed simultaneously, because we are dealing with hardware.

In c, what will happen? It goes statement by statement and there will be delay from one statement; no matter how many giga hertz you have put. It will take finite number of cycles and go to the next, only after it has processed one statement. It is the sequential statement that all the microprocessor DSPs and in contrast to that, FPGA and ASICs are all concurrent. No matter how many thousands of statements you may have here; all of them will be processed simultaneously. This is because, all these are translated right into gates and that is the reason, it is precisely same as you had earlier with circuit boards putting so many digitalized packages. Although we write like a program, end result is actually the real hardware. That makes the real difference between c and this. **Conversation with student (23:57)**. There is no provision here; but, you can artificially do it. Suppose, you allocate a variable - a counter; we have already seen how to put a counter.

What prevents you from putting a counter and keeping track of the actual time element? You can see the particular counter; you can do that only if you Run. Not while looking at the code here, there is no way looking at the code. When you execute, you can have it; but even the counter is a superfluous thing. This is because the time axis will itself reveal where you are.

(Refer Slide Time: 25:01)

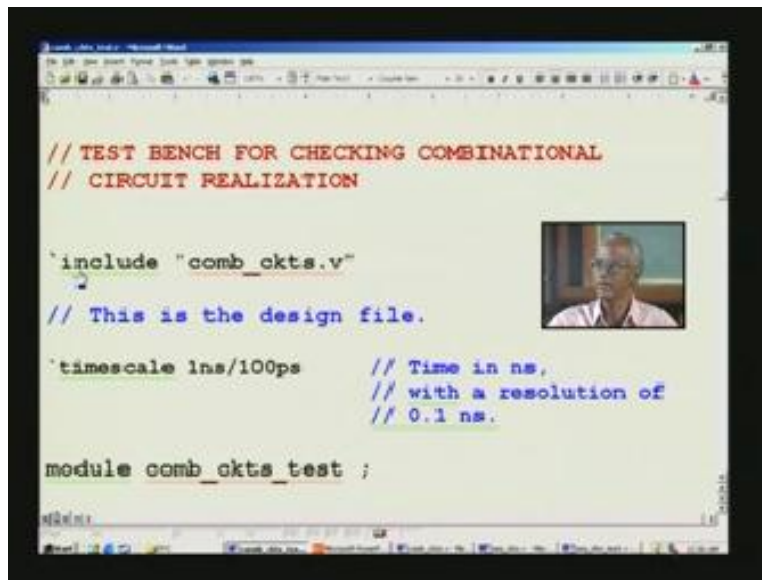
A screenshot of a Verilog code editor window. The code is as follows:

```
        // 20 ns,  
#20 A = 1; in = 0; //40 ns, and  
#20 A = 1; in = 1; //60 ns.  
  
#40          // Run long enough to test  
            // all possibilities,  
  
$stop ;     // and stop.  
$finish ;   // Terminate simulation.  
end  
  
endmodule
```

That is also a redundant thing only. So, there is no provision. Then, we have almost finished the test bench. It is so easy. Now, what we say is, we have already accounted for all the combinations, we give little more extra time. We run for little longer time so as not to miss any information. For example, it will take another 20 nanoseconds to complete that 1 1. It would be sufficient, if we put 20 here. You can put any number as there is no hard and fast rule for that. Only thing is make sure in sequential circuit later on, when we go we may not be in a position to assess this correctly. Play safe by putting a large figure there. In simulator, the moment you load and run your port, pat comes the reply. No matter how big these values are, so long as it is below 10,000 or so. Then, you have to signal that you have finished the whole thing. Then there is a special command using a dollar here and followed by stop. It stops all the testing. Finally, this is only temporary stop and you can resume if you wish. I forget the command for resuming. Suppose you want to finish the entire thing, that is, terminate the simulation, you can give this. If you do not give this also, it does not really matter. Finally, we started with initial and then begin, a matching end is this. We also started a module, so there must be an end module. This

completes the test bench for the most primitive design just 2 input AND gate. With this background, we can go to the next. This is what we have started earlier I will go a little fast, because you are already familiar with most of the things and we have already explained that. Now, it should be just a cake walk for you.

(Refer Slide Time: 27:00)

A screenshot of a code editor window displaying Verilog code. The code is a testbench for a combinational circuit. It includes a comment in red: "// TEST BENCH FOR CHECKING COMBINATIONAL // CIRCUIT REALIZATION". The code includes a file named "comb_ckt.v" and sets a timescale of 1ns/100ps with a resolution of 0.1 ns. The module is named "comb_ckt_test". A small video inset of a man speaking is visible on the right side of the code editor.

```
// TEST BENCH FOR CHECKING COMBINATIONAL
// CIRCUIT REALIZATION

`include "comb_ckt.v"
// This is the design file.

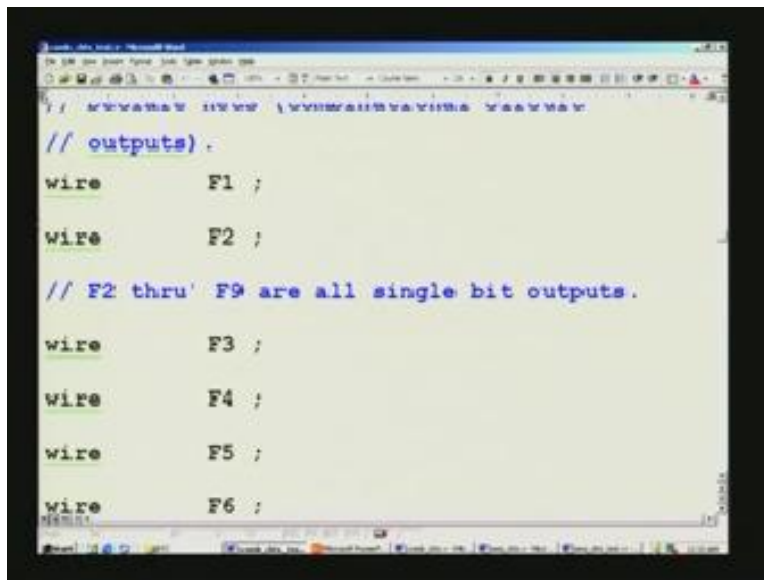
`timescale 1ns/100ps // Time in ns,
                    // with a resolution of
                    // 0.1 ns.

module comb_ckt_test ;
```

We have seen that include will include the design here. In this case, combination circuits and time scale is also exactly the same. That is what we put here, with a resolution of point 1 nanoseconds, which was not earlier. Now, we start the module. This is the test bench. So, we give the combination circuits underscore test. This is a meaningful name. As usual, we declare all the inputs and inputs are all marked here: A, B, C. You remember that, we had done so many logical gates and here is the one which we want to recollect. All this we have already covered and that is precisely what we have here. Here, in test bench, we have to declare all inputs as reg in order to hold the value. We saw 8 inputs for 8 input MUX. We had a magnitude comparator which compared 2 numbers and they are all inputs. What we are going to speak up to this point are all inputs. Here, size is also taken into account and I had forgotten that wire earlier. I will explain it here, because it is fundamentally same. Here, we have already seen how to invoke a design module. You may have several design modules and you may invoke that. In the course of invoking, you may like to have one or more of the signals from one module to be used in some other modules. How do you declare those signals? You declare them as wire, because it is a

physical wire from one IC to another. Its counterpart is that. That is, it is analogous to that. So, that is the reason why, we declare it as a wire.

(Refer Slide Time: 29:11)

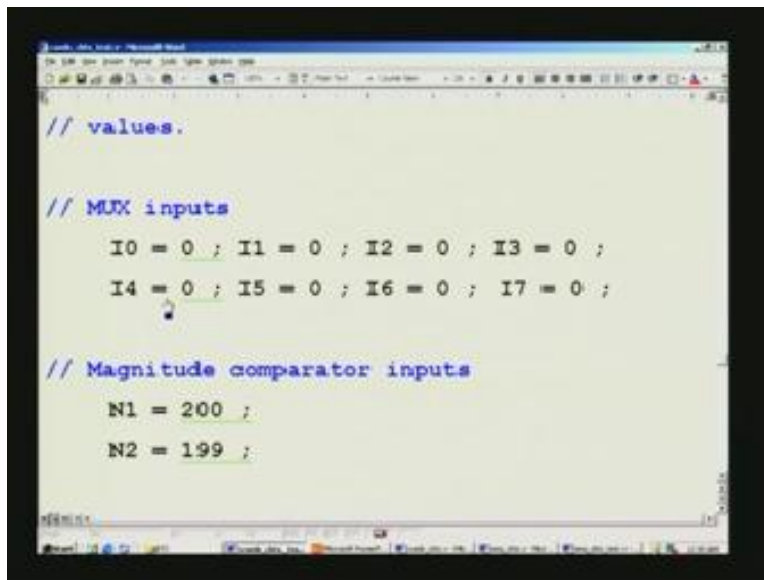


```
// outputs).  
wire      F1 ;  
wire      F2 ;  
// F2 thru' F9 are all single bit outputs.  
wire      F3 ;  
wire      F4 ;  
wire      F5 ;  
wire      F6 ;
```

This is applicable only in the case of test bench. Do not confuse it with the design. The declarations are entirely different there. Based on the environment you are placed in, you have to see. The logical reason is clear here. We want to physically connect. That is why, it has got to be the net. The keyword for net here in Verilog is wire. That is why, we do this and these are all single bit outputs. These dimensions we have seen here. These all are listing for various output combinations. We have done behavioral full adder, then data flow, and gate level; all these are listed here. This is for the magnitude comparator outputs, all of them are clear. Then, we had one more design example, wherein 2 numbers are sum. We have already seen n_1 , n_2 earlier and corresponding outputs are this. Once again, all outputs are declared as wire. Now, we call this combination circuits design; we instantiate here. Like this, we could have instantiated different modules. If there is more than one, we could have done and any signal that you use may have to be connected to another. That is why we had to declare that particular signals as wire. That thing is very clear now and connects ports by name. This also we have seen and I will rush it through, because it is precisely what we had already seen. IO's are here. Last one should not have a comma separation. That is what it is here and you complete the brackets here. Once again, here also you have an initial. We are going to apply the similar thing (Refer Slide Time: 31:20). This

is little longer, because you have so many signals. First, at 0 time, this is selector input here and we apply 0 0 0 here. We also need for MUX. I just initialized all of them to 0 here.

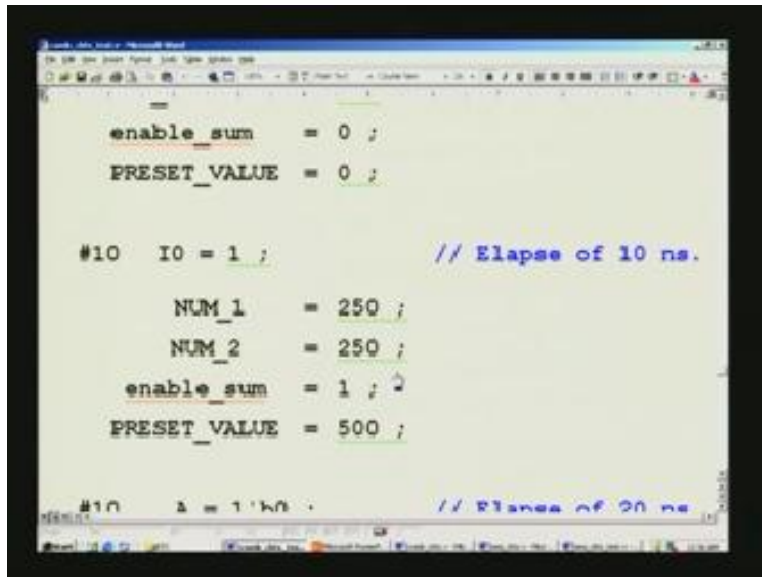
(Refer Slide Time: 31:46)



```
// values.  
  
// MUX inputs  
I0 = 0 ; I1 = 0 ; I2 = 0 ; I3 = 0 ;  
I4 = 0 ; I5 = 0 ; I6 = 0 ; I7 = 0 ;  
  
// Magnitude comparator inputs  
N1 = 200 ;  
N2 = 199 ;
```

We have a magnitude comparator. So, we use just 2 numbers; remember 200 and 199. Obviously, 200 is greater than 199, more output will be set and so on. You can keep on changing your input data; for various combinations, you keep on changing. At 0 time, I apply all these inputs and keep on changing at different points of time. At 10 nanoseconds, I am changing one of the MUX; I am applying just 1. So, I will be applying systematically one after another, so that it will be easy for you to look at the simulator result, one thing at a time. Otherwise, you will get bugged down in details. Now, I change once again after 10 nanoseconds. I am changing either the same inputs or some other inputs or fresh inputs may come into being here.

(Refer Slide Time: 32:45)



```
enable_sum = 0 ;
PRESET_VALUE = 0 ;

#10 I0 = 1 ; // Elapse of 10 ns.

    NUM_1 = 250 ;
    NUM_2 = 250 ;
    enable_sum = 1 ;
    PRESET_VALUE = 500 ;

#10 I1 = 1 ; // Elapse of 20 ns
```

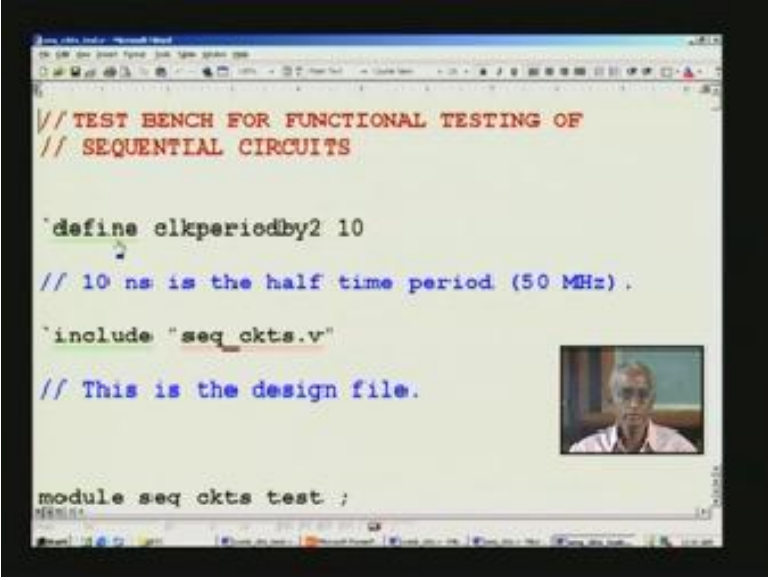
Some of them you can see here. We will not worry about these things now. When we go into simulation, we will refer to this and go in more detail. It is sufficient that the whole thing runs in the same sequence, a ten nanosecond I am just advancing it by one, the selector. We can have I1 process. Once again, here, I1 is still made 0. I just want to delay that, because I wanted to put some more numbers for other modules. Once again, this is cumulative, I think we are in 20 nanoseconds. You have to unfortunately keep track, but you can always keep track by keying it as a comment here. That is why I had put in many places. Once again, all the inputs; you remember this stimulus is only for inputs, you need only to apply to input.

Simulator will give you everything in one shot as a waveform; both inputs as well as outputs. You can see that comment here. I have kept track at 40 nanoseconds; again, I change different inputs. In this way, it keeps going. I do not have to go repeat the same thing and we will finally go to the end. That is 140 nanoseconds. It has taken so long, then we give some more cushion there, so that this being a combination circuit, we do not have to really give much cushion, may be 20, 30 nanoseconds will suffice here. Still, you can play safe, but it may be required in sequential circuits as we will see right now. That is what I have commented here; run long enough to test all possibilities. The possibilities may spill over beyond this time. That is why, we need a questioning. As usual, we have a stop. I have deliberately omitted finish there, because it is not really mandatory. Finally, we have an 'end', because we had initial 'begin'. We had put

everything in a module. Corresponding end module must be there. So, this completes the test bench for the combination circuits. These figures alone are put as far as combination circuits are concerned. This is what, we have already seen just now for the test bench.

Now, we will see the sequential circuits test bench. We started with 2 registers; you can use it for delaying or pipelining. Then, we have a counter realization, a monoshot realization, shift register and varieties of shift register; although pictorially it is showing only 1 here. Then, we saw parallel to serial converter and a state machine model also we had seen. Finally, we have seen a pattern sequence detector. This is what we are going to go for test bench. Once again, here, you give a meaningful name as to what you are doing and then define the new thing which we are introducing here. Here, we are dealing with a sequential circuit. So, we need to create a free running clock. Without a clock, there is no life in sequential circuit. For that matter, any digital system needs a clock and clock in actual practice, when you finally make a circuit, I mean a chip, and a clock generator will be there. That will feed into the chip which you finally design.

(Refer Slide Time: 37:00)

A screenshot of a text editor window displaying Verilog code for a test bench. The code is as follows:

```
// TEST BENCH FOR FUNCTIONAL TESTING OF
// SEQUENTIAL CIRCUITS

`define clkperiodby2 10
// 10 ns is the half time period (50 MHz) .

`include "seq_ckts.v"
// This is the design file.

module seq_ckts test ;
```

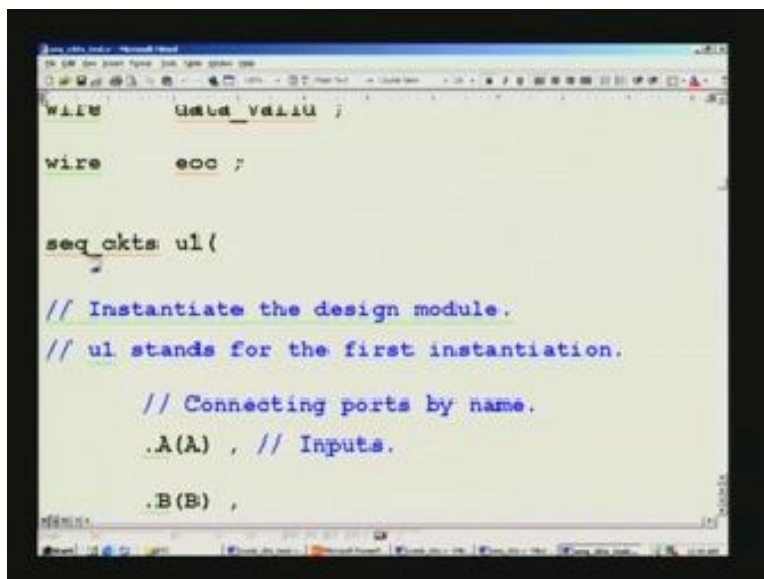
The code is color-coded: comments are in red, preprocessor directives are in blue, and the module declaration is in black. A small video inset of a person is visible in the bottom right corner of the code editor window.

What do we do in the simulation environment? You have to create it once again by the verilog coding itself. In order to facilitate this one, we first define what is called a clock period by 2. This is the name that I have given. If you feel it is too long, you can give any other precise name. All these are like a c program which you can give, where you can give meaningful names. What

we say is, this is unit of time. This implies that there must be a time scale definition also. Compiler may give a warning, but you can overrule that warning. You do not have to put time scale every time. You have already taken into account the test bench. So, you need the time scale only for the simulation, remember that these all the times we have spoken hash, and so on, all these will be filtered out during the synthesis process. This is because we said that, the verilog designing fundamentally using RTL coding guidelines to get rid of the technology dependence.

We should be independent of the technology and that is possible only if, there is no time element. However, we have introduced time element just to facilitate simulation. Therefore, it appears only in the test bench. During synthesis, it will reject the test bench. That is the reason why, in the design, we include all the down files sub modules of the design inside the design and include the top design file only in the test bench. This implies you can treat it as a variable here. So, this is again a behavioral thing just like initial and they are not RTL compliant. We have seen earlier, what are RTL compliance? What is not? You should discriminate the 2. As usual, this is equivalent to a variable called clock period by 2 is equal to 10. We include the design here and we declare this test bench module. Again, a meaningful name and we declare all the inputs as reg and outputs as wire. That is what you see here. All these are wires (Refer Slide Time: 39:29). These are all outputs here and multi bit precision. So, we have seen a state machine. Those outputs are also listed here. This out corresponds to; we have a pattern sequence director.

(Refer Slide Time: 40:28)



```

wire      data_val10 ;

wire      eoc ;

seq_okts u1(

// Instantiate the design module.
// u1 stands for the first instantiation.

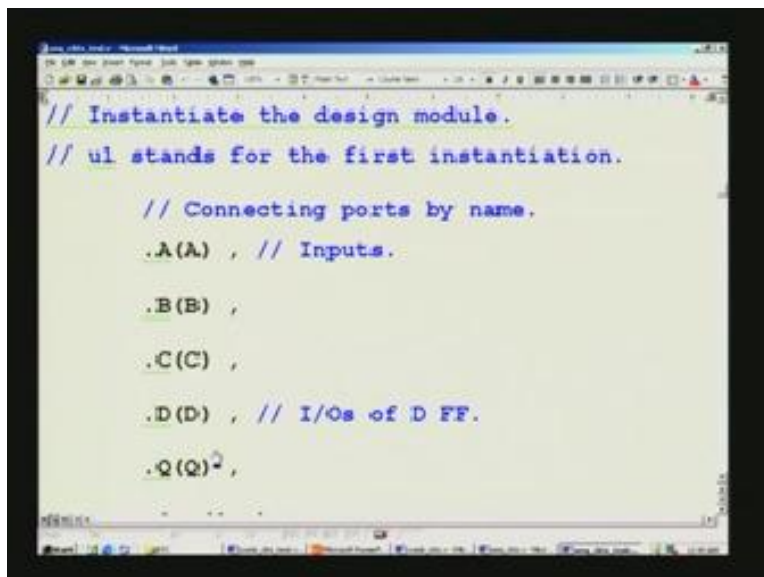
// Connecting ports by name.
.A(A) , // Inputs.

.B(B) ,

```

In and out single bit are this. So, this happens to be the output further. Hence, we declare it as wire. We had some counters and so on. I do not remember for what module exactly. You can always make it out. At the time of simulation, you will see in detail. Some end of conversion is there. I think that was a parallel to serial conversion we had done. Pertaining to that, these signals are present and then, we instantiate the design module. This is the design module which we call instantiation here. As usual, you can give the chip name, instance name here. List all the inputs of this. You remember that we are doing this for the design. Adopt that calling connect ports by name we have changed.

(Refer Slide Time: 40:58)



```
// Instantiate the design module.
// ul stands for the first instantiation.

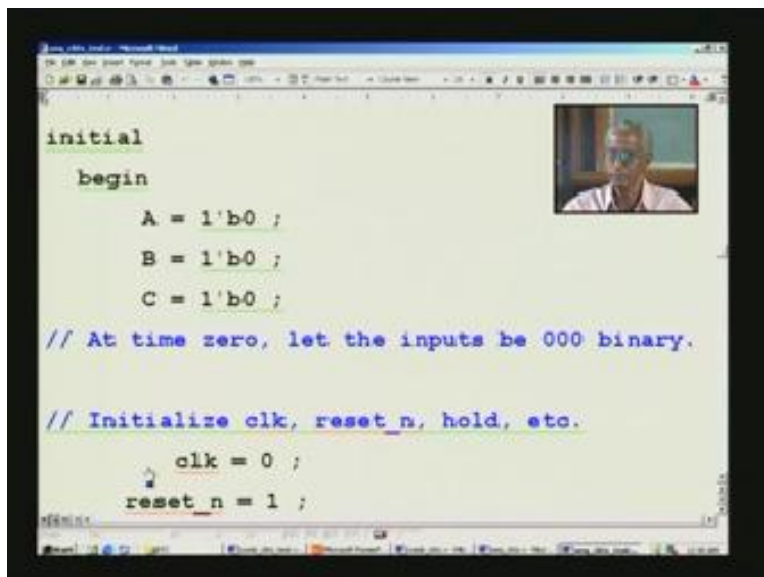
// Connecting ports by name.
.A(A) , // Inputs.
.B(B) ,
.C(C) ,
.D(D) , // I/Os of D FF.
.Q(Q) ,
```

That is the nomenclature we give. Here, you would appreciate that need for this, because suppose if I had forgotten and put instead of B here, it will be chaotic as I mentioned earlier. In order to do that, we have called by name. This lists precisely all the IO's here and what it is, it is mentioned here. Inputs of state machine here; this in1 and in2, we use in the state machine. Trigger is the monoshot input here and then all these are pertaining to the monoshot here. Then, outputs of parallel to serial converter, we have already seen earlier corresponding inputs. These are all the outputs here and output of this state machine is here. Note that all of them are same. One thing you can mention, that was the doubt raised in the earlier lecture also. You can have different name here, because this pertains to this test bench, whereas this pertains to the design. In design, you may call it by one name and here, you may call it by different name. For example,

for out here, you may actually be using a counter; not the out. Of course, if it is a single bit, it has no meaning a counter. If it is a multibit illustration, I am showing. You can redefine in this particular test bench, you may be using another counter which you want to connect here. What you do is, you can call it as a counter 1 or something like that. The only thing that should match is, dimension of this variable, this signal, and this signal. Is that clear?

We have completed listing of IO's here in the instantiation. That is why we have the 'end' here. Once again, we have initial and I do not have to go step by step in this. It is sufficient to say that all the inputs that we need for testing this whole sequential circuit will appear one at a time. When the time is right for that, it will strike. At 0 time you initiate all these. These are basically some of the inputs used here. A, B, C are the inputs used for setting a reset of a register and so on. We not covered clock. We said that this is what we are going to generate. We saw that earlier having defined clock divided by 2 or something like that and equating it to 10. It means the clock half time, it is not the time period, but half the time period is 10 nanoseconds. If you had declared time scale as time base 1 nanosecond, then that is what is implied there. Now, clock is actually a variable here. It is a purely behavioral statement and not an RTL coding confirmation.

(Refer Slide Time: 44:33)

A screenshot of a code editor window displaying Verilog code. The code is enclosed in an 'initial' block and a 'begin' block. It initializes variables A, B, and C to 1'b0. It includes two comments: '// At time zero, let the inputs be 000 binary.' and '// Initialize clk, reset_n, hold, etc.'. Below the comments, it sets 'clk = 0;' and 'reset_n = 1;'. A small video inset in the top right corner shows a person speaking. The code is as follows:

```
initial
begin
    A = 1'b0 ;
    B = 1'b0 ;
    C = 1'b0 ;

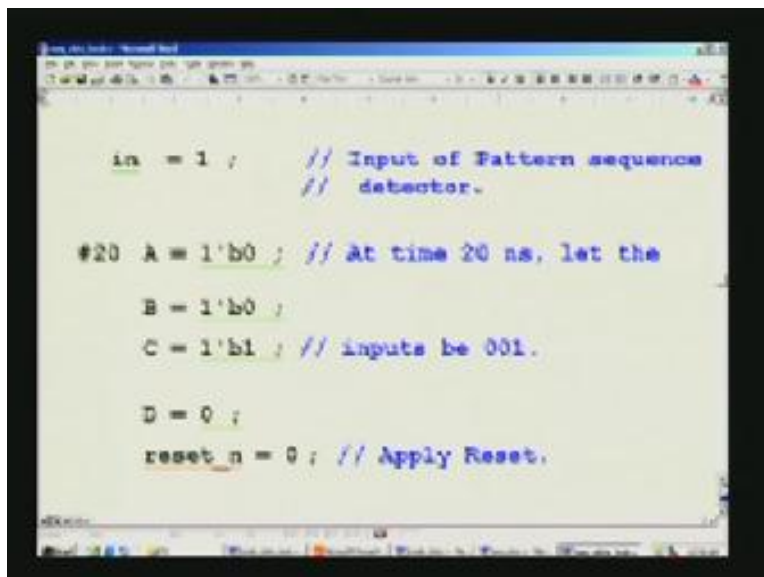
    // At time zero, let the inputs be 000 binary.

    // Initialize clk, reset_n, hold, etc.
    clk = 0 ;
    reset_n = 1 ;
```

It violates the RTL coding, because it is only to create the clock for the sake of simulator. It is not the actual chip level clock, which needs to be generated separately using another IC or

whatever. You initiate reset, then trigger is for mono shot, then if you want to hold all that. You can make it either 0, that is low, that is inactive state, most of them and active state is 1. We can also use this one notation, meaning that 1 stands for the number of bits here. So, it happens to be only 1 and what it is? It is 0 level, low. In binary, you have only 2 states 0 and 1. So, b for binary, this tick will also attribute. Here, we had shift register. So, we need to set data; that is 16 bits. I am setting it to AAAA, because this generates alternate 1s and 0s. You can also write it in a binary if you wish. That is what you have here; exactly same thing and this seems to be slightly different.

(Refer Slide Time: 44:33)



```
in = 1 ; // Input of Pattern sequence
// detector.

#20 A = 1'b0 ; // At time 20 ns, let the
B = 1'b0 ;
C = 1'b1 ; // inputs be 001.

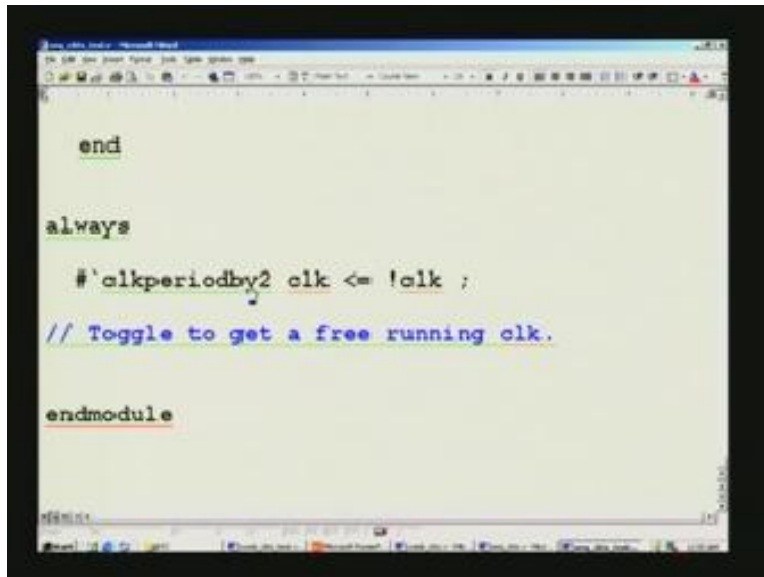
D = 0 ;
reset_n = 0 ; // Apply Reset.
```

Data output tends to parallel to serial converter; so this is the example which is correct. Then, we apply different inputs for this; when it should be shifted, then right or left right. So, for state machine, we need in1, in2, and so on. Now, we change the stimulus. After 20 nanoseconds, it has been changed from 0 0 0 to 1 here. Now only, we are applying the reset, this is the power on reset or system reset. Earlier had you noticed that one? It would have been 1, it is 1 here. To start with, it was 0 nanoseconds. Let us not reset immediately, we will give some time lag and then reset the system. 1 will not reset, because this is negative here. So, it is active low. That is what we have already seen; asynchronous active low. After that, we bring it back to 1 from 0 after 20 nanoseconds, so that normal working of the sequential circuit can commence. Only subsequent to this, positive edge clock will arrive. From then onwards, its system starts working. The whole

affair goes within 20 to 30 nanoseconds. At 40 nanoseconds, you can see subsequent to this, clock must arrive. It may arrive somewhere in between here around 50 nanoseconds. When we see the simulation, we will see that exact time. That you can find out, because that clock variable we have given, we have made it to 0. We will come to that later on. When it will change, we will see. That determines the rising edge of the clock. It was 0, when you make it 1, and then naturally, rising edge is encountered. Similarly, you keep on changing for various inputs. That is what you see here. This is for the sequential machine basically in1 and in2. **Question by the student.**

This is the pattern sequence detector. For that, you notice that all along, I have been using 10, 20, and so on. I have made this deliberately 5. If I had put this 20, this pattern sequence detector will not work. This is because for simple reason which I explained earlier also while describing this. You have to apply positive edge of the clock, only after subsequently applying the data and the data must be stable. By doing so, I am applying 5 nanoseconds earlier. I am applying the data, so that it will be valid when actually the clock strikes. That is why this is done. Again, all the inputs change and finally, we will go to the 'end'. These are all nothing but the same pattern; for every 20 nanoseconds, I am changing the input pattern as per the pattern that we have listed earlier. Once again, we want this being a sequential circuit, you give more cushion. Arbitrarily, I gave this one. After running the simulation, you will know that it may not be required. You can always come back and change to what you want. There by quicken the simulation process, but it is hardly noticeable for these values. You do not really have to worry about that. Finally, the test bench is complete with a stop and a dollar finish, which I have not put here. If you wish, you can put that. Finally, an 'end' is there. Remember that we have not ended the module so far.

(Refer Slide Time: 50:30)



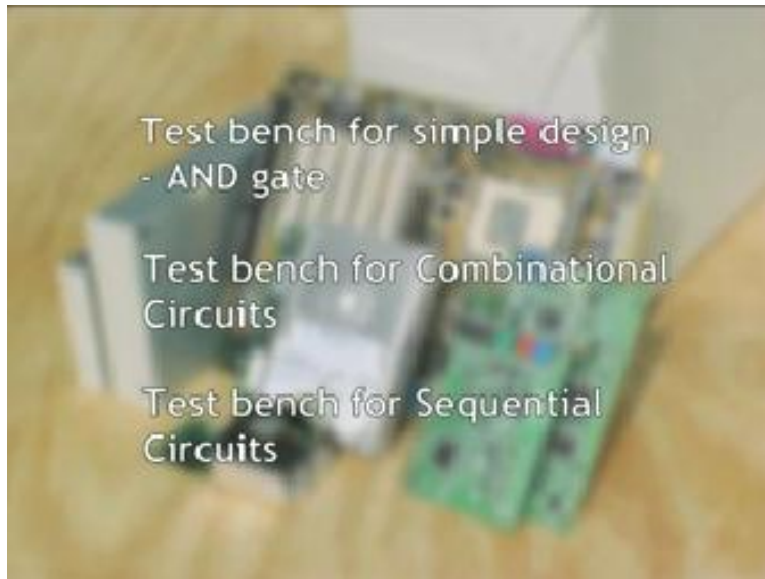
```
end

always
  #'clkperiodby2 clk <= !clk ;
// Toggle to get a free running clk.

endmodule
```

We said after clock period by 2, this was equated to 10 there, which implies 10 nanoseconds. That means you take this decision only every 10 nanoseconds afterwards. This applies for every 10 nanoseconds. This means I said that all these statements are concurrent as well. So, this is 'always', a separate block. There is no beginning and no 'end', because it has single statement here. This clock if you notice, it is inverted. This symbol is used for equal to as I said before and 'always', block must have this symbol, not the 'equal to'. In assign and initial blocks, and so on, you can give that equal to. That is how, it will toggle. Finally, for our module, you need a matching end module. With this, I complete the test bench. **Conversation with the student (51:30)**. We started with clock equal to 0 and what is 0 inverse? It is 1. So this will happen every nano second because it is 10. This means, after 10 nano seconds it will do this: so 0 will be 1, 1 will be 0, it will go on for eternity till it fails. Next time, we will see how to use the simulator. Thank you.

(Refer Slide Time: 52:33)



(Refer Slide Time: 53:01)

