Digital Circuits and Systems
Prof. Shankar Balachandran
Department of Electrical Engineering
Indian Institute of Technology, Bombay
And
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module - 34
Verilog: Sequential Elements

Welcome all to the last module for this week. So, we are in module 34 now, this last module is going to be about Verilog and modeling sequential elements in Verilog. So, the first part of the video is going to be about styles, last time we discussed different kinds of styles. So, I talked little bit about that and then for the rest I will talk about sequential elements. So, if you go back to the previous week's video on Verilog, you have notice that there are three different ways of doing things.

So, you can either have assign statements or you can have always blocks or you can use perimeters and you can also instantiate these and make different kinds of designs. So, you also put the design files out for you to practice with. So, one thing that I did not talk about last week is that, I did not say when to you use what. So, there are three different styles, but as a designer when do you use what.

So, let us start with the multiplex for an instant, so the 2 to 1 multiplexer is fairly easy, it is easy to describe in terms of equations. So, you could go and write an assign statement with the equation for a 2 to 1 mux or in general you go and write something, which is using if statements. So, but it is a very small design, so it may not matter, so much. So, the key thing is you will use assign statements only when the expression that you have on the right side is fairly simple and it is known to everyone.

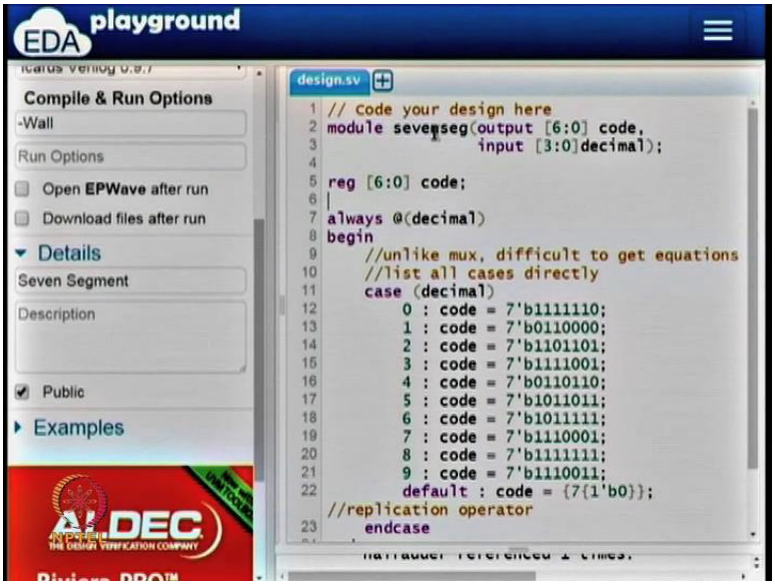
So, if you have case like that, then assigns statements are okay, if you want to do some little bit of logic, just you want to do AND of two wires, OR of two wires or things like that then you use an assign statement. Whereas, when you make more complicated designs, it is not feasible to do a assign statements. So, one example is the 7 segment decoder, so we discussed this one of the classes earlier.

So, if I want to do a 7 segment decoder, I may take inputs which are from 0 to 9 and I have outputs, which are going to be 7 segments, each segment will light a 1 segment of a LED. So, I want to take input from 0 to 9 for which I want to displays digits 0 to 9, for 10 to 15 I want to keep it blank. So, this is a specification that I gave earlier. If you want to do something like that and if you want to do it in Verilog, it does not make sense to go and take each and every combination of input and derive every combination for output.

So, I do not want things like segment 0 is on, when these things are on, when segment 1 is on, when these things are on and so on. So, if you go and think about it as, for every output I will specify the combination of inputs for which it has to be on, it breaks down the way designer's think. So, for us it is easy to think about for k 0, I want all of these on, if it is 1, I want to all of these on. It is easier for us to think that way, rather than if which any of these inputs I want the topmost segment on.

So, if you derive equations you will be forced to do that. For every output, you will derive an expression which is in terms of inputs, you can go and put an assign statement there, but that is not the best way to do it. So, what I want to show is this use of k statement for something like that. So, let us start with this, so I already put in the code here for the, all the view Verilog code that I am going to discuss today, I have already typed it in today.

(Refer Slide Time: 03:50)

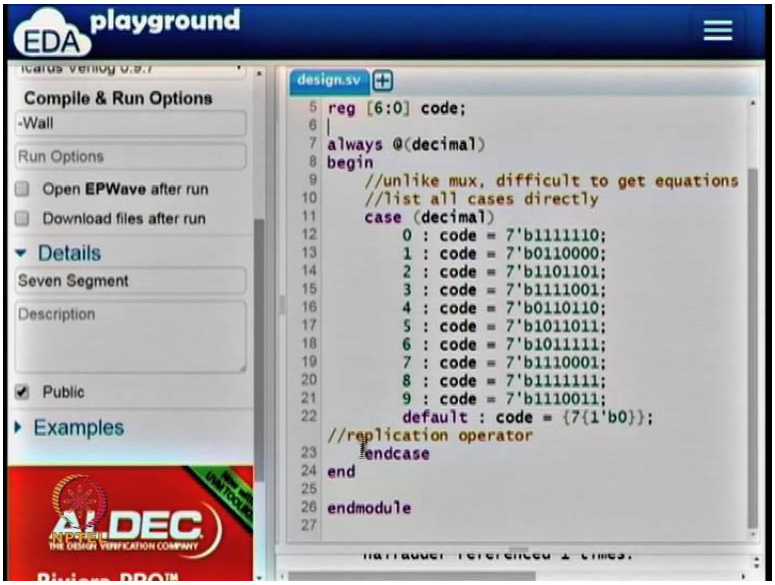


So, let us see this, so I am designing a 7 segment decoder, in the 7 segment decoder, there is an output which is for the 7 segments. And it is supposed to take decimal 3 to 0 and what I have is, I declare code as reg. The reason why we declare reg is, if you have to do any assignment inside always block, then it can be done to regs, it cannot be done to wires. So, we any assignment that you are going to do inside in always block, we will do it only...

So, the left side is for code, so it has to be declared as a reg, so do not worry about why it is so and so on. So, this is we will leave it to the language designers, the language designers of decided, it will be like this, so let us tick with that. So, given that we have declared reg 6 to 0 of code this means, there is code of 6, code of 5, code of 4, code of 3, all the way up to code of 0 and this is a 7 bit code.

So, now, if I am consulting a device which has 4 inputs, 4 bits, so that is called decimal, always at decimal I have written a statement here and always block. So, this always block begins here and ends somewhere here.

(Refer Slide Time: 05:07)



So, I have scrolled it up, so begin and end for the always block is here and within that I have written a case statement. So, let us see why, what the case statement does first, so it takes, it does case on decimal. So, since there are 4 bits, there will be 16 possible choices. On the left side you see the 10 choices that matter to us, k 0 to case 9 and for each of the case, we assign code equals something.

So, for instance this one says code, so 6, 5, 4, 3, 2, 1 all of them have to be on and 0 which stands for the middle segment is off. So, the meaning of this one is code 6 is on the top, then you have code 5, code 4, code 3, code 2, code 1 which goes in cyclic order and the code 0 is supposed to be for the middle segment. So, this one is very clear, it is easy to write. If I want code 0, I can then go and quickly write, I want all the things on the periphery to be 1 and this to be 0.

So, for if I want 1 to be displayed, the top segment must be off, then the right most segments are both on, all the other segments are off. So, this is how we think and we imagine the design, so why bother writing equations for each and every bit of code. So, what you see here is something very drastically different, you are not seeing code of 6 equals in equation, code of 5 equals in equation and so on.

Instead, we are assigning code of 6, 5, 4, 3, 2, 1, 0 all of them to a vector and the reason why we do that is, it is much more natural for us to think for each case, what should be the different values that should be assigned to code as supposed to, what each bit of code must be. So, let us go and look at for instance 6, so it has a vector which says 1 segment the top right segment is off. So, you can see that here that is 0, so this is simple and nice and if it is not any of the 10 cases from 0 to 9, but default code is assigned to all 0's. So, this is an operator which is slightly more interesting.

So, what it says is one tick b 0 means, 1 bit of 0 and this 7 and within calibrations if we put 1 tick b 0 says, I wants 7 copies of bit 0. So, what this code is getting is 7 copies of bit 0, which means 0 0 0 0 0 0 0 which means, the display will be off. So, this is how you can model a 7 segment decoder. So, the nice thing about this is for whatever reason later I decide that, I want for 10 and 11 also I want a and b to be displayed, I can go and write case statements here.

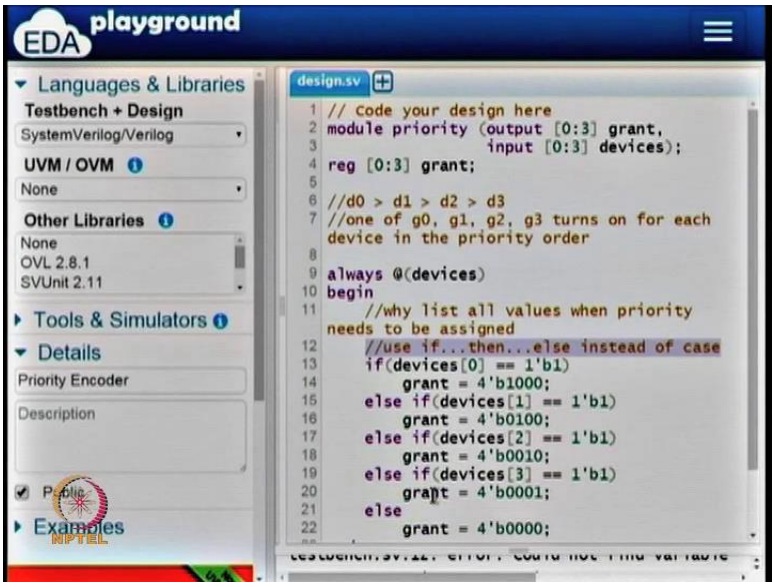
If you derived equations, you have to go and plug in more complicated things, especially if we have simplified it, it becomes worst. So, instead this gives you the flexibility, you are thinking about the decoder. So, you want 4 to 7 decoder, then this is a very direct way to write it, so it gives you something very nicely. So, whenever you know equations you use the assign block, if you do not know the equations, but you have a lot of choices and for each choice, you want to specify what a certain output bit must be, then you use a case statement.

So, in this case there was 16 choices, for the 10 choices these values where all different, so I want to assign for each of the 10 choices. For choices from number 10 to number 15, the last 6 choices I want all of them to be 0. So, if is not any of these they get 0, so the key thing is for all the 16 possible combinations at explicitly listed them. In fact, if you want to display a hexadecimal digit let us say a, b, c, d, e, f also you want to display, then you will put 6 more case statements below these statement, from line number 22 you will put 6 more case statements.

So, this is if for every input combination you know exactly what the output is, in some sense we are actually put the truth table for each of the code bits, this is if you do that,

but even this is not good for a different kind of a problem. So, let us switch to the next problem in this... So, in the next problem....

(Refer Slide Time: 09:41)



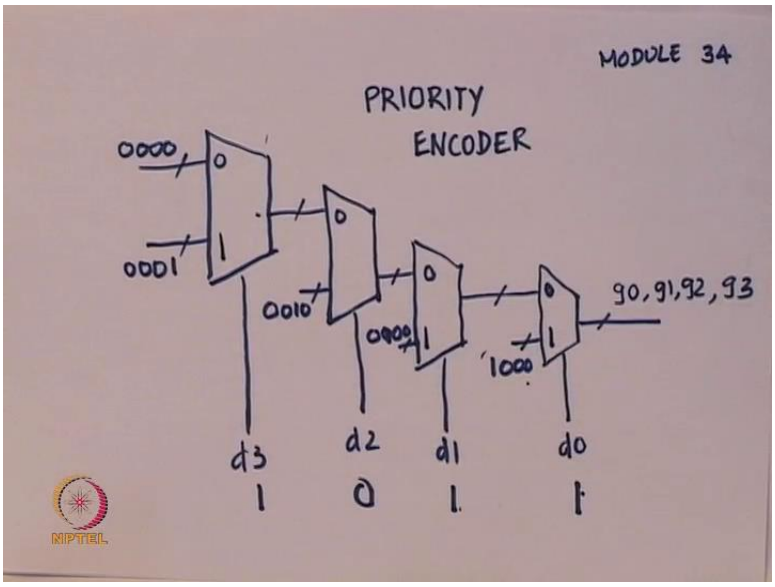
So, we discuss the next another problem called priority, so we forget the code for a while now, we did this little thing. So, I said there is a boss, there is a manager and assistant manager and a clerk these four have request and based on the four request, there should be 4 grams. So, we design a state machine for that, but let us say, now I want to design a combination circuit to do that. Based on the inputs, there are 4 lines that are going to come in, they are going to be called devices 0 to 3.

So, device 0 is for the boss, devices of 1 is for the manager, devices of 2 is for the assistant manager and devices of 3 is for the clerk and you want to see, who gets an grand, whether the boss gets the grand or whatever. So, we know that you want this priority d naught is greater than d 1 greater than d 2 greater than d 3. So, that is the order and we want exactly one of g naught to g 3 to be turned on, so this is the problem specification.

So, for this problem specification you can go and write all the 16 combinations and we can derive equations and we can write grand of 0 equals 1 under these conditions, grand

of 1 equal to these equation, grand of 2 equal to these equation and so on you can do that. That is not natural, you can actually try and do a k statement, but then I have 4 devices you will list 2 power 4 combinations. To more if I give you 10 devices for which I want priority, then it should be riding 2 power 10 lines, that is not feasibly either. What you are looking for is, in some sense you want priority of one line over the other and so on.

(Refer Slide Time: 11:27)

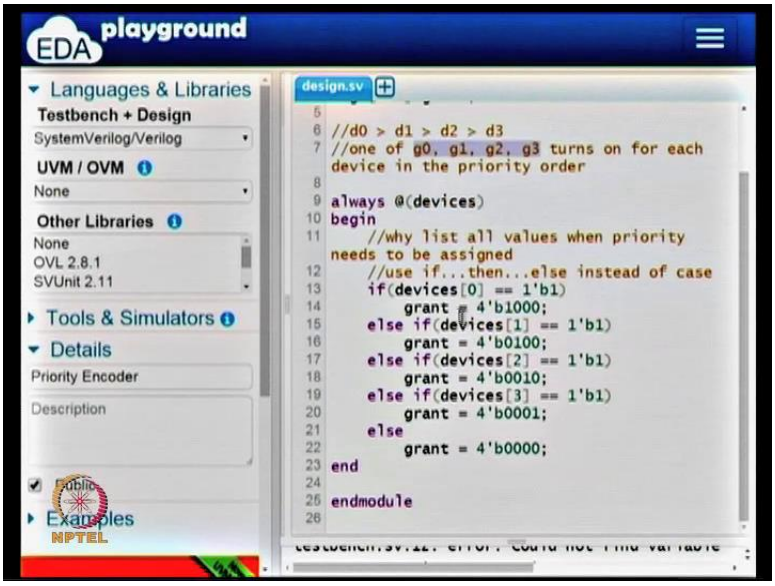


So, let us look at the picture here, I have already drawn the hardware diagram that I want. So, this structure is called a priority encoder. What we have is, we have devices of 0, devices of 1, devices of 2 and devices of 3, these are 4 single bit values, I give them as a select to a mux. So, these mux are 2 to 1 mux, as in it takes 2 inputs and puts one of these things from the output, but each of these lines is a 4 bit line, so this is a 4 bit line and this is a 4 bit line.

So, let us see how this whole things will look at priority, let us take a small example. Let us say d 3 was 1, d 1 was 1 and d naught was 1, let us assume that d 2 was 0. If this is the case, then what would happen it means that, there is a manager who is requesting, assistant manager is not requesting, clerk is requesting and the boss is requesting, so d naught is for the boss. Now, how would this thing work? If d 3 is 1, it places 0 0 0 1 on this line, so this 0 0 0 1 will come on to this line.

Now, d 2 is 0 which means this line would appear here, so that will appear as 0 0 0 1. Now, if d 1 is 1, it takes this line which has 0 1 0 0 and places it here and this one over write this combination and instant places this combination on to the output. So, essentially d naught being 1 will place 1 0 0 0 on the output. So, what we are given is, even through there are different priorities coming up to this level, at this level d naught being 1 or 0 can decide which one should go. So, d naught has higher priority over other d 1, d 2, d 3, so this is the hardware structure that I want. So, if you want to do something like this, so let us see the code again. If you want to do something like this, it is natural for us to, if you have to write it in language, it is natural for us to go and think in, what order the input should be check.

(Refer Slide Time: 13:35)



So, let us see the code here I have written in a always statement, it takes all the devices, at devices means all the bits of devices, devices of 0 to 1, 2, 3 all 4 of these bits. If any of those 4 bit change, you have to reevaluate your grand. So, who is going to get that access to the device you have to be revaluated. So, the way it is written, if you want priority encoder is, if devices of 0 is 1, then grand goes to the boss. So, the code is grand of 0 becomes 1, grand of 1, 2, 3 are all 0, so from left side to right side it is 0 to 3, so grand of 0 becomes 1.

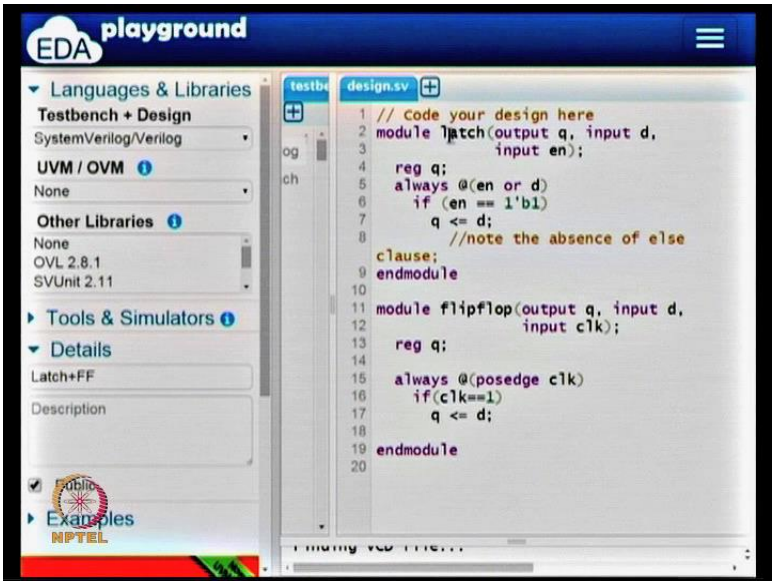
Else if, devices of 1 is 1, so if this will happen only if the boss has not made a request. So, if the boss has not made a request, only then you go and check if the manager has made a request. If the manager has not made a request only then you go and check if the assistant manager has made a request and so on. So, that is what we have here, so there are 4 cases, there is a first if case, then else if, else if and else if, if none of them made a request, then there is no access that needs to be given, so you have grand equals this.

So, this is the much more easier way to express, what we have in mind and in terms hardware also, this is what you are looking for, we are looking for a mux kind of thing. Now, in this set up, if I say that by the way it is, was not 4 people, there were 10 people and I have the hierarchic of the 10 people and can you modify the design. Then all you have to do is, you go and change these, you go from 0 to 9 and in here, you have to go and add 6 more lines for each one of the new people that you have added.

As suppose to, if you did this using case statement, you would have gone from 2 power 4 lines to 2 power 10 lines for which you have to explicitly give, that does not make sense of all. So, this is much more cleaner, this is using an always block, the previous one also used in always block. However, even within the always block, which kind of statement you pick. So, there is beauty in the code here, this is much more natural and in terms of hardware, this is much more easier to translate and this is less error grow.

Because, this is explicitly says, these are the bits I am looking at and that is all. If you have to type, even let us say 2 power 6 lines somewhere you may make a mistake, so that will not happen here. So, this is a very nice style for writing this priority encoder. So, now I am going to switch, given some talk about sequential elements. So, there are two sequential elements that I want to talk about, one is the latch, the other is the flip flop, so let us look at latch.

(Refer Slide Time: 16:25)



So, I have written down the modules already, let me explain what, how these things are designed. So, a single bit latch has a single output q and input d and an input enable and these are the three inputs. So, I assume that q bar is not needed, so I have only modeling for q and I have used always statements. So, in fact if I want to model sequential elements, you cannot do assign, you cannot instantiate basic logic gates, there is no basic logic primitive for sequential elements.

So, latch is a sequential element, so we have to write it using always block. So, let us see the always block here, this always... So, the latch has two signals, there is an enable signal and there is a d signal and both the signals are actually, what are called triggering signals. Because, change in any one of those can result as a change in the output, so both enable and d are triggering signals. So, we put both in the sensitivity list, so both the inputs enable and d come in the sensitivity list.

And what you want now, if enable is 1, then q should follow d, so we have that here. If enable is 1, q equals d and if enable is 0, what should you do. If enable is 0, you have supposed to remember the old value of q, you are not supposed to change q at all. In fact, if you do not change the value of q, you do not have to write it as else q equals q, you do not have to do that. Automatically, if enable is 0 whatever the value of q is, it will be

automatically retains.

So, implicitly if you see these statement, q retaining it is old value is implicit, if enable is 1, q equals d else what, else retaining the old value of q. So, that is implicit and that is what we have here in the top half, so what we have for a module latches, if enable is 1, q is d and that is it. So, there are few interesting things here we are put both the elements on the sensitive list, because a change in enable or a change in d can change the value of the latches output and between have an else class.

In fact, if you put a else class and if you put q equal to a let say, let say there are 3 inputs enable or d or a and if you do if enable is 1 q equal to d, else q equal to a that would mean you are actually inspiring a multiplexer. Because, if enable is 1 then that is the select line q is select d; otherwise, q is select a, in this case we are not inspiring multiplexer, we want a latch there is no else class.

So, if you want a latch you do not put the else class and you leave it as it is. In fact, when you are designing a mux if you leave a else class that is a problem. Because, it may infer it as a latch, we will discuss that in a later class, but as of now we can see that always block as a fairly simple statement, always at enable are d you do this. Now, let us take a little while the think about what a flip flops.

So, if you think about a flip flop, a flip flop is little peculiar device, so it has a clock which is an input, which is the triggering signal and d is suppose to be a sample signal. So, change in d will not reflect as change in output, unless there is a clock that came to sample it. So, this is the deal with flip flop, so let see the module design for a flip flop, flip flop also takes d and clock as inputs and q as an output.

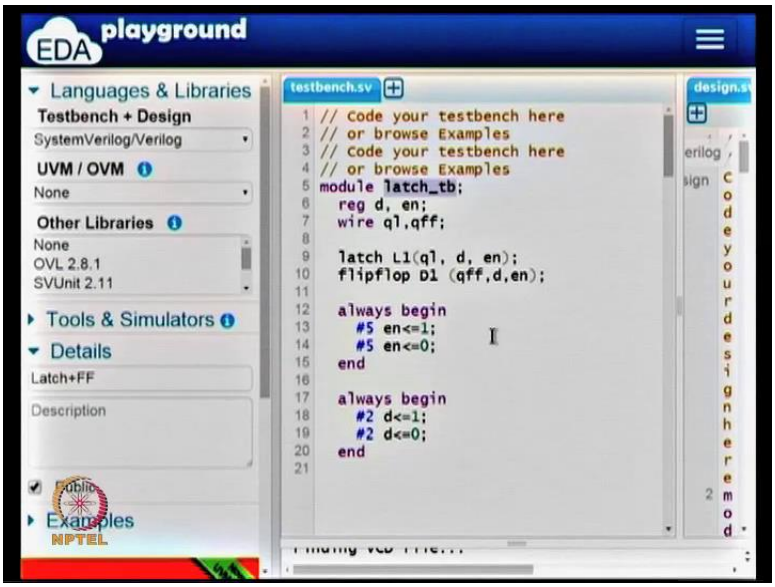
So, we have reg q as before, so anything that assigned inside in always statement, you are going to declare a reg for that. And let us look at the sensitivity list first, the sensitivity list it does not have always had clock, it has some new thing called posedge of clock. So, always at clock would mean this statement should get executed, when clock changes from 0 to 1, as well as when clock changes from 1 to 0 that is not what you want for a flip flop if it is a rising edge flip flop, then we want it only due or positive edge

regard flip flop, you want it to sample the value d only when clock goes to 1 from 0.

So, this always at posedge of clock is posedge as a keyword, always at posedge will look for whether the clock is going from 0 to 1. So, this statement this always block will get trigger only when the rising edge condition is satisfied. If the rising edge condition is satisfied, then q equals d again implicitly else do nothing, do nothing means keep the old value which means during the level 1 or when the flip flop goes from 1 to 0 or when the clock is at 0 in all these cases, you are going to retain the old value.

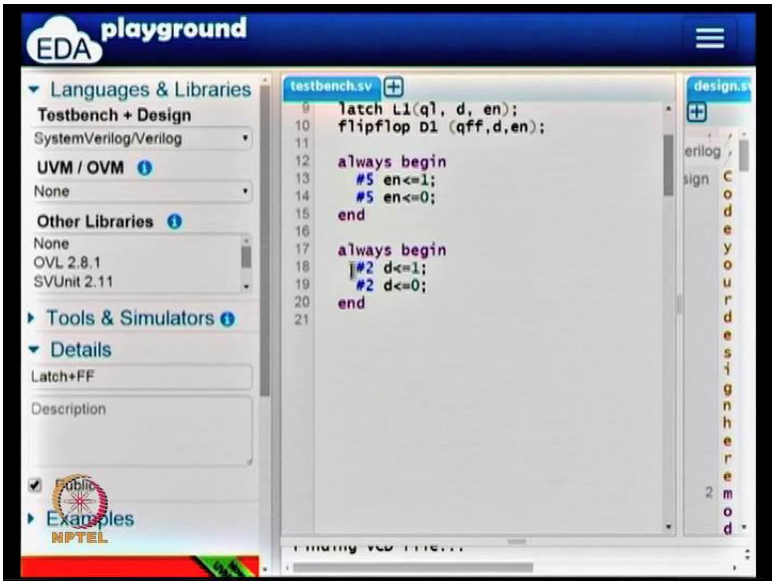
So, the only case where d should be sampled is when there is a rising edge of the clock and that is what you have here, always at posedge of clock, if clock equals 1 which means you are regent to 1, then q equals d; otherwise, implicitly it is not going to change anything.

(Refer Slide Time: 21:51)



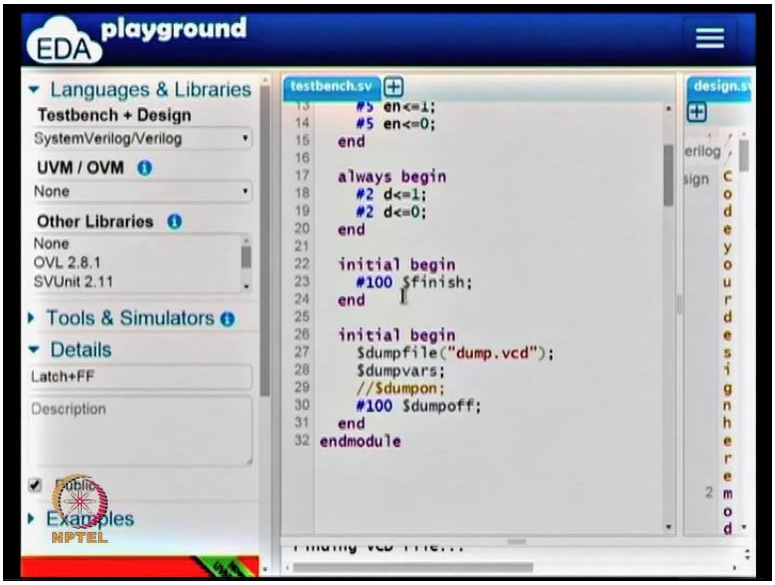
So, I also written in test bench for this, what I have done is I have return a test bench in which there is a... So, this is actually both a latch and flip flop test bench, I instantiated a latch here and I will call the output of the latches q l, I instantiated a flip flop here and I am going to call these flip flop output q f f.

(Refer Slide Time: 22:12)



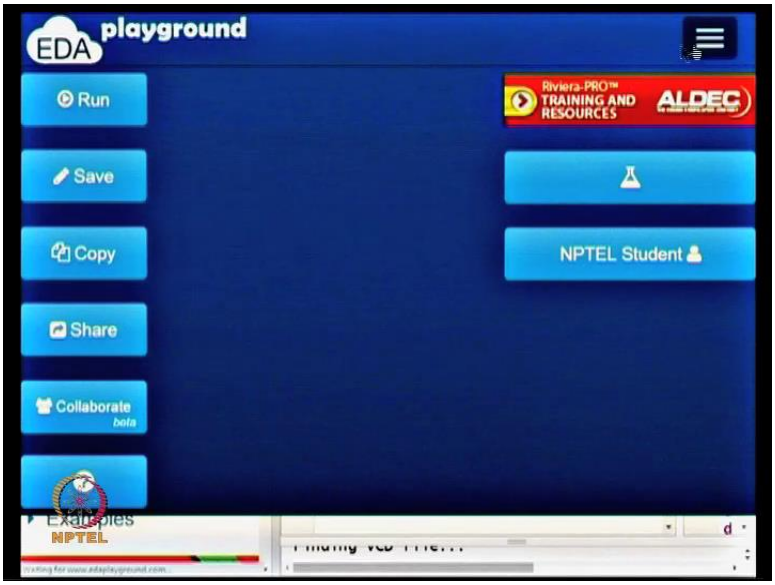
And I have a clock here which is toggling every 10 time units and I have a data which is actually changing every 4 time units.

(Refer Slide Time: 22:24)



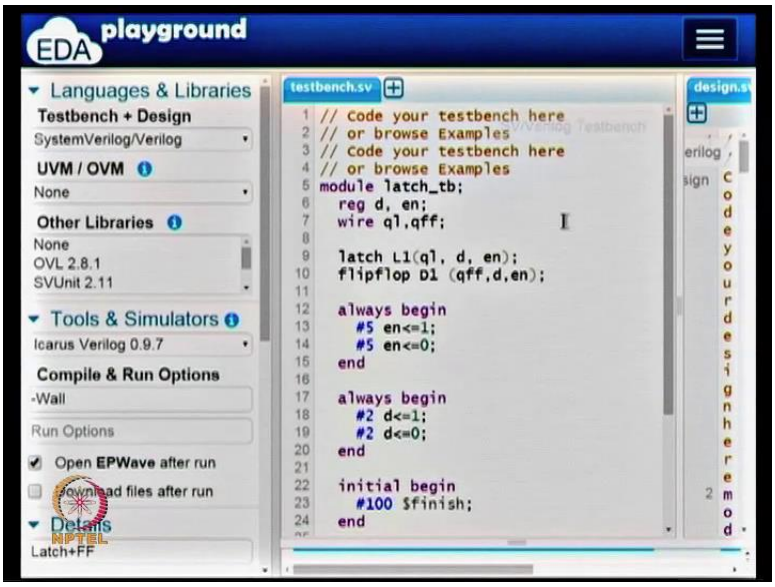
So, and I have put various other things, so that the simulation will end and so on.

(Refer Slide Time: 22:33)



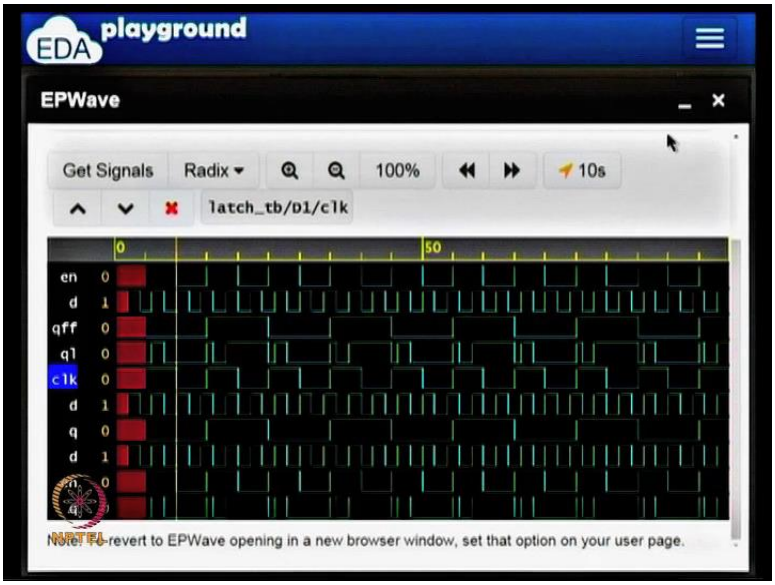
So, let me save it and run it.

(Refer Slide Time: 22:36)



So, I actually want the way form to be shown also, so let me run it once more.

(Refer Slide Time: 22:58)



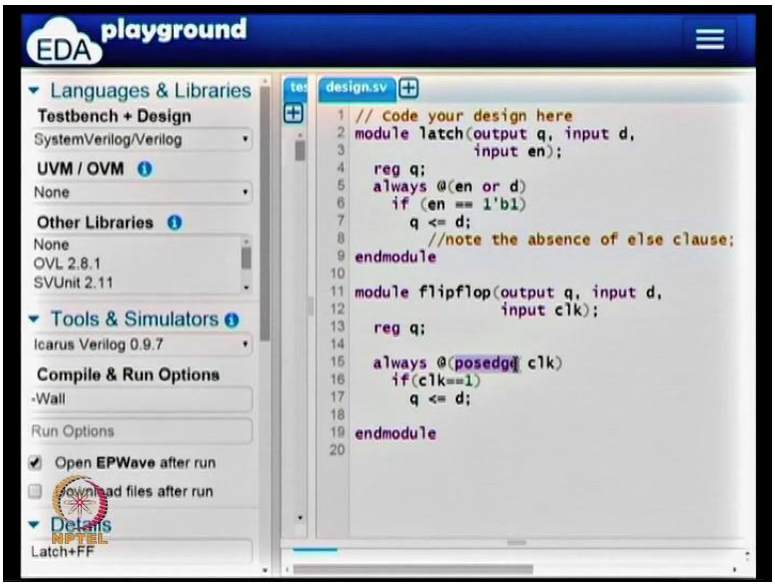
So, let us see these here, so it actually opened up all the things that are in my way form, but what we care about is the enable line which is the clock that I have used, the q f f and q l. So, let me move this one up, so I want the, this is my clock signal this e n is called my clock signal, d is my data input, q f f is my flip flops output and q l is my latches output. So, forget all the lines below that, so it is showing everything inside the design also.

So, we have instantiated something it is also showing all the signals has seen from the inside the instance, let us look at the top four things. So, in the top four things the clock is a positive edge trigger clock. So, whenever it is going positive d get sample, so for the q flip flop you can see that when it when positive we copied with sample 0, then it when positive at that point sample 1, again it went positive here it sample 0, it sample 1 here and so on.

So, you can see that q flip flop is behaving the way it should be q l of the other hand it is not sampling only on the edge. So, at this point level became 1 and the input was 0, so q l became 0, but this part where d is 1 is also track then d became 0. So, that became 0 here, so let me pick a line here at this point d is to 0, then if I pick between these two places enable is 0 which means any change in input will not reflect the changes output

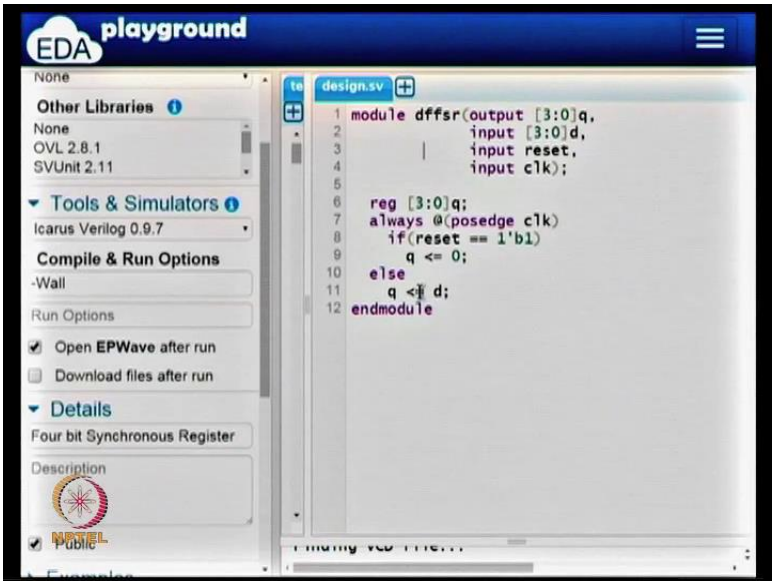
for the latch also. So, you can see that only when enable is on, d is copied to q l, when enable is off d is not copied to q l. So, this is the design that you want it for a flip flop and a latch.

(Refer Slide Time: 24:59)



So, we have that here, so I suggest that you go and change this posedge, you remove the posedge and just keep it as clock and see what happens and try an reason what would happen if I just had clock here, instead of posedge of clock.

(Refer Slide Time: 25:16)



And finally, I want to talk about the fourth one where it is a register. So, in this case it is a 4 input, 4 output register. So, it is a register which as a parallel input and a parallel output. So, in the parallel input is d it as 4 bits d 3, d 2, d 1, d naught and it has 4 bits as output q 3, q 2, q 1, q naught and it has a reset and a clock. So, this dffsr is d flip flop based synchronous register, so that is the name that I have used d flip flop based synchronous register.

So, what is the synchronous register it is, so synchronous reset. So, a flip flop is synchronous reset is 1 in which the clock comes in, once the clock comes in it is samples both reset and d, if reset is 1 the register is set to 0, if the reset is 0, then d get sample. So, this is the meaning that you want and you can see that, that meaning that we need is actually automatically reflected in the code here.

So, we have always at posedge of clock, so this is looking similar to the flip flop itself, the basic flip flop that I wrote, it as only posedge of clock. If posedge of clock comes in, you then go and check if reset is 1. So, what it what this is doing is, when the rising edge of clock comes in you are sampling the value of reset, if reset is 1, then q is 0 else q equals d. So, this looks like for all the combinations of reset, you are actually assigning the value of q.

But, this is not a combinational circuit, this is still a sequential circuit, because the way we are modeled it... So, if I put d reset clk all of them in the sensitivity list, this is supposed to be modeling a combinational circuit. In the sequential circuit only the sampling signals must appear in the sensitivity list and the sampled signals should be used inside directly. In this case both reset and d are sampled signals, we are sampling the value of reset, we are based on the condition of reset, we will sample the value of d.

So, that is return in this form, if reset is 1 q is 0 else q is d, so this is the specification for a d flip flop based synchronously resettable register. So, there are two things that I have done, so in this case I have used input and output as 3 to 0 ((Refer Time: 27:52)) and in the devices add declare it as 0 to 3. So, unlike programming languages like C or C plus plus Verilog will let you pick the left most index.

So, if you say 0 to 3 the left most index is 0 and the right most index is 3, if you declare something as 3 to 0, the left most index is 3 and the right most index is 0. So, depending on how you declare you have to interpret the bits appropriately, so for the devices and grand I set the left most should be called 0 and the right most should be called 3. Whereas, in the d flip flop I said the left most bit of q should is q of 3 and the right most bit is q of 0.

So, this is just convenience, because designers think about set of connections in different ways. So, Verilog let us you actually pick indices the way you want, so in fact you can even say q of 5 to 2. So, if you do not want index always starting from 0, you can say 5 to 2 may be for some reason, you are picking you have a burst of bit 8, there are 8 parallel lines going, you want inspect only the bit position 5 to 0, you can actually slice only that in use it also.

So, Verilog gives you a lot of flexibility for these things, because that is how hardware designers do it. So, it is not correct to go back and ask what is this software feature that I have, I have this software feature in C and C plus plus how to do that in Verilog that is the wrong way to approach hardware design. So, the way to look at it is in hardware I want this how do I model this in Verilog.

So, Verilog should not be use thought about as a substitute for a sequential language, like C or C plus plus. So, very log is primarily for hardware modeling, so the way should think about Verilog itself is I want to do this in hardware, how do I model this in Verilog or given something in Verilog what would the hardware implication of that be, what would it be inferred in hardware . Because, there are tools which can take Verilog code and generate gates for you.

So, for those tools it is necessary that you write Verilog in a certain way, so that the tools can interpreted as a template for this is a mux, this is a decoder, this is a flip flop, this is latch and so on and generate appropriate gates for you. So, there are 100s of thing you can do inside Verilog I am teaching a very small subset of Verilog that is needed for hardware design, there are lot more things like for loops and so on which are in Verilog they do not have semantics for hardware always.

So, that is the reason why I am doing is not actually teach those things, once you get compatible with how do you use very log well, then in a later course or some time when you pick up designs skills, I suggest that you go and explore the other options. Till then you use the fundamental things by looking at the code you should be able to say that this is the hardware you are trying to infer and many tools in fact keep it simple they cannot handle, so many complicated cases.

So, they expect your code style to be fairly simple and straight forward from which hardware can be infer. So, this brings me to the end of this week's lecture, in this week again we saw quite of few things, we looked at state machines and different aspects of state machines, particularly we looked at state assignment and we looked at state minimization, you also add in earlier video on delay analysis of a sequential circuit.

So, I suggest that you go and review the material, because these are the advance topics in a digital design course. So, other things you might have seen in text books, but delay analysis is not something that you see in text books usually. But this is a useful thing to learn, because this is useful for when you take up a job or even when you go on interview these things might be useful. Of course, Verilog as a language that I am teaching is also useful, if you want to get into the hardware design industry.

So, I suggest that you observe the materials slowly, but studiedly and again if there are questions please come back on the forum, as see that the questions of the forums are usually about when is the quizzes can I extend the deadline and so on. I do not see technical discussions on the forum, other then what is related to the quiz, nobody is asking questions about that course itself or the material in the course and so on.

So, please participate, because if you have a doubt you post it probably others also have the doubt they will come back and discussion and so on. So, that is a healthy way to run this course, so otherwise this is just like being a new spectator to the hole RBL and so for me I do not get the cu. So, I cannot see how you are reacting and how you are learning. So, it is better that you discuss things on the forum and only then I will know how you are reacting to the material in the course.

So, this brings me to the end of week 6 and week 7 will have more exciting and new topics both related to Verilog and digital design. So, I will see you this week we also have Verilog, assignments programming assignments. So, this video is necessary for you to go and do the assignments, go and review all the Verilog videos before you attend the assignments.

So, thank you and see you next week.