Digital VLSI System Design

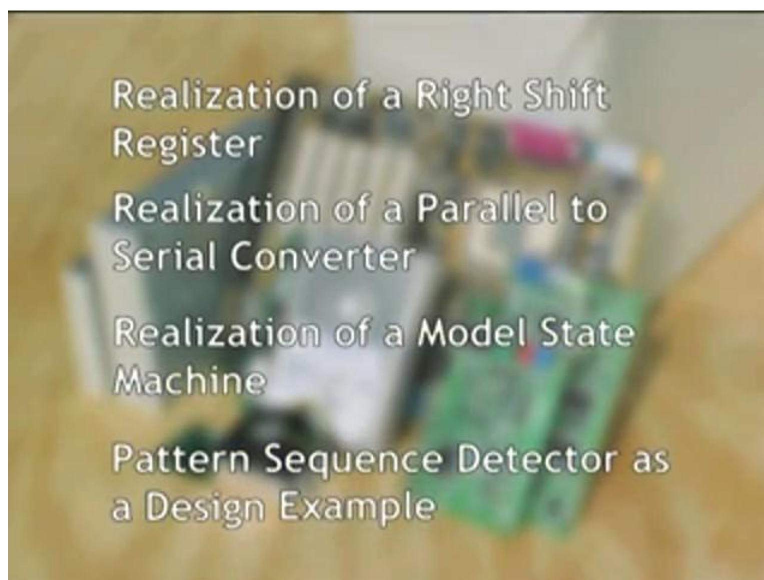
Prof. Dr. S. Ramachandran
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 13

RTL Coding Guidelines

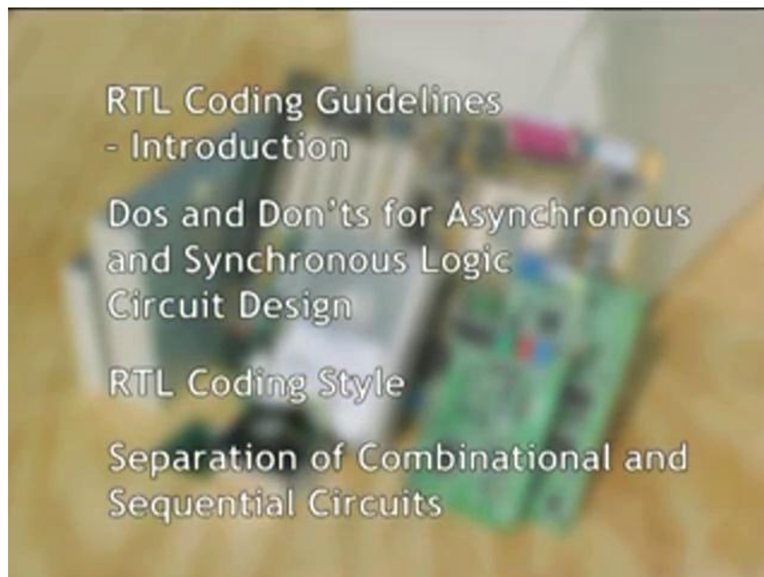
Slide – Summary of contents covered in previous lecture.

(Refer Slide Time: 01:09)



Slide – Summary of contents covered in this lecture.

(Refer Slide Time: 01:27)



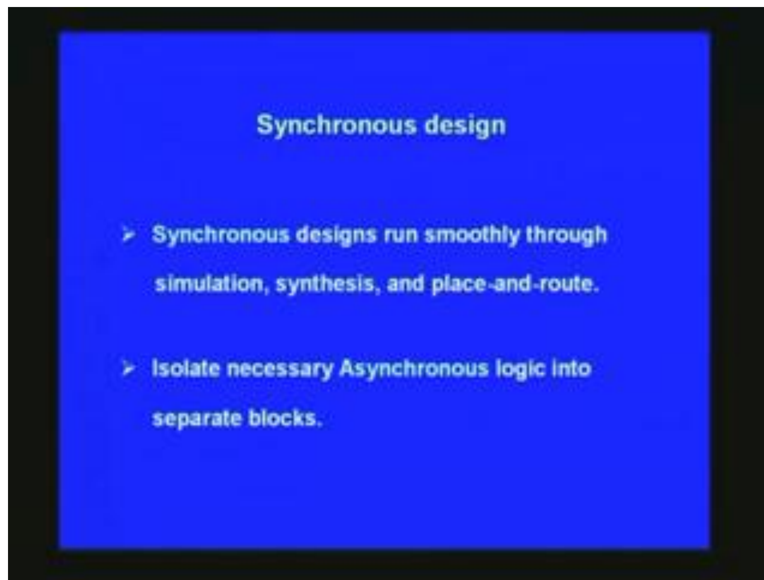
Slide – Summary of contents covered in this lecture.

(Refer Slide Time: 01:49)

We have so far seen how to model in Verilog the combinational and sequential circuits which are vital ingredients in any digital VLSI system design. The ultimate goal of the designer is to finally map it on to a device such as an FPGA or in ASIC. This is possible only if you follow certain guidelines. The popular guideline is to follow the RTL coding; RTL stands for Register Transfer

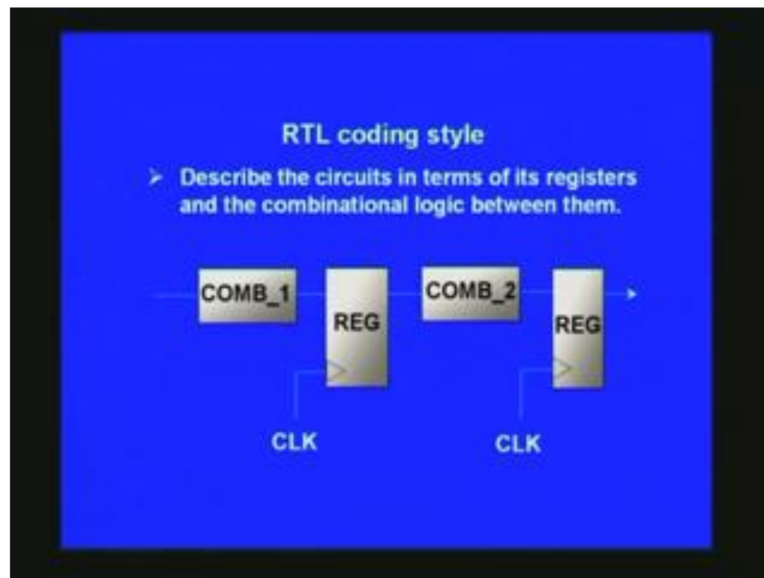
Logic level. It is basically a synchronous circuit design which we already used. It signifies the data flow and how you process the data.

(Refer Slide Time: 03:02)



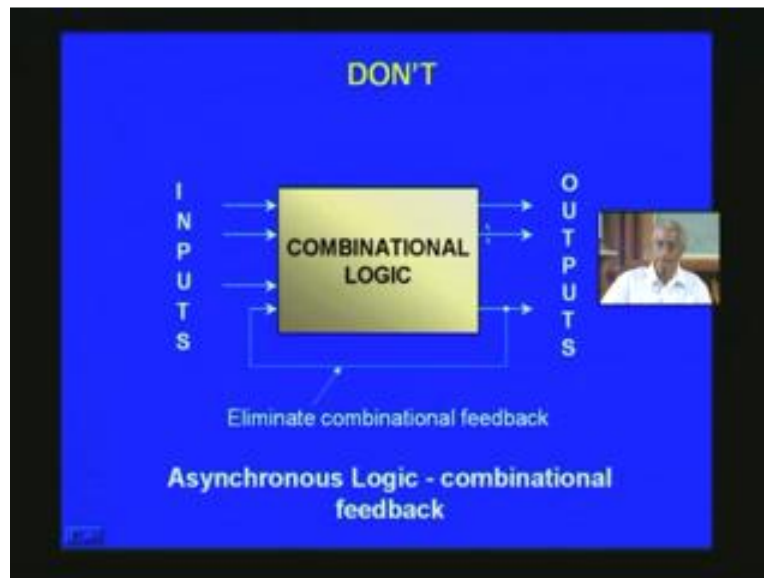
Basically, it is a synchronous design which should naturally run smoothly through simulation; then synthesis and then finally on place and route. In order to do this, you have to isolate the combinational and sequential circuit; asynchronous fall under that combinational which we have already seen in our earlier lectures and it has got to be separated out as we have already done in the previous class.

(Refer Slide Time: 03:33)



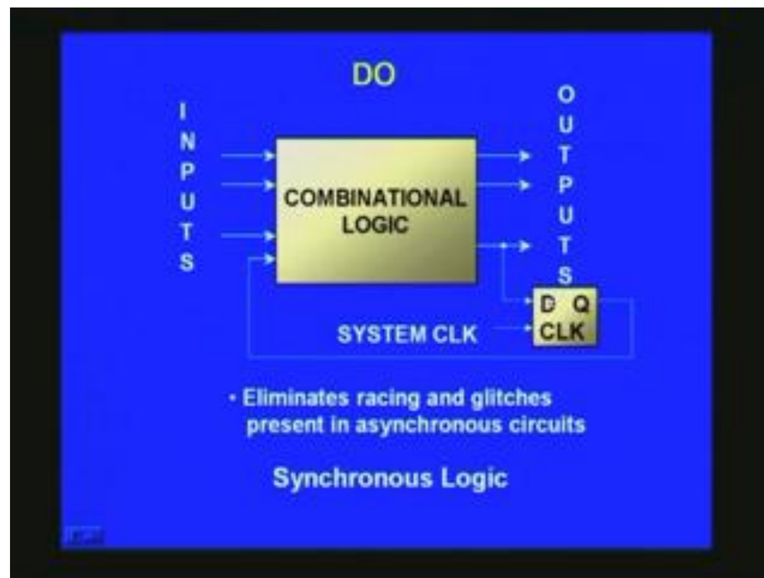
Basically RTL coding style is once again same as synchronous design. You have a combinational circuit and it can be any complex combinational circuit then follow this with registering this combinational output. This may be an intermediate output then follows the same pattern successively. This keeps going in this fashion and data flows in this fashion naturally. It is normally referred to as a pipeline which we will cover in depth later on then we deal with arithmetic circuits. For the time being this is enough if you know that RTL coding style is basically based on separating out combinational as well as sequential circuits.

(Refer Slide Time: 04:36)



It basically consists of dos and don'ts, because this is supposed to be a guideline for the designers. The common mistake is to take a combinational output and feed it back; this is detrimental in making a chip. For example, if you apply a 0 at the input here and the combinational produce 1 here (Refer Slide Time: 05:10) and then 1 being fed back it keeps on rapidly switching from one state to another. This will be uncontrollable and thereby it will not work in the ultimate chip that you have designed. Furthermore this may be rejected at the synthesis level itself when you use the synthesis tool, this is not permitted. It will give warning or error message; perhaps you can veto that and go along but ultimately you will end up messing the whole design. For simple reason that your design will not work for ASIC or FPGA that you have ultimately done. That is the reason why we should stick on to this RTL coding guidelines without which you will not get a working chip. Remedy for this is to break the feedback. Fortunately we have synchronous approach for this that is depicted in the next slide.

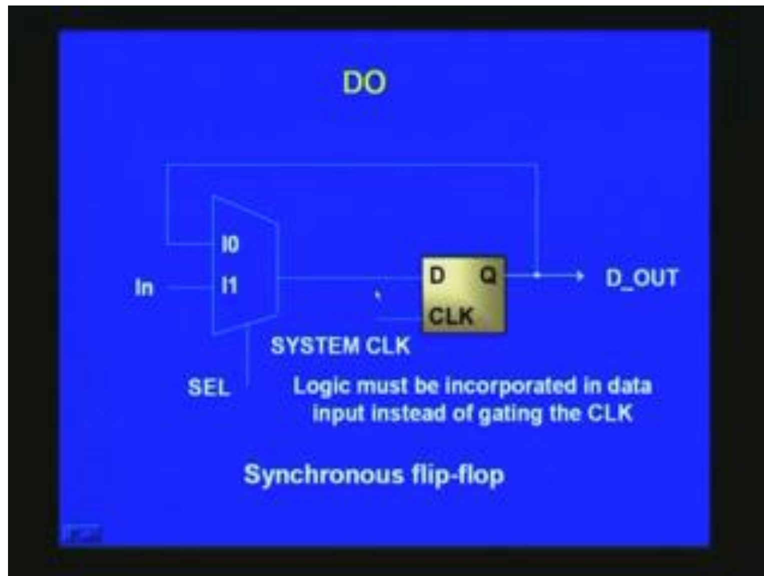
(Refer Slide Time: 06:14)



These you have a flip-flop here, D input flip-flop. You break this earlier feedback here pass it through a flip-flop. Feed it to the D flip-flop and there will be a system clock which will be separately given to a clock input. Q Output is actually fed back. Whatever is the combination logic that you have inside will produce this output. This will be processed only when the system clock arrives that is at the rising edge if it is programmed for rising edge direction. Q output is naturally delayed by 1 clock pulse because this will take effect for the next iteration or whatever you call next processing only at the next clock pulse. This is the price that you have to pay for breaking that combinational logic but this will result in system which will work on the IC; whereas the other previous one will not work on the IC. Instantly, it eliminates racing as well as glitches which are normally present in asynchronous circuits which we will see later. This is the basic building block for asynchronous logic.

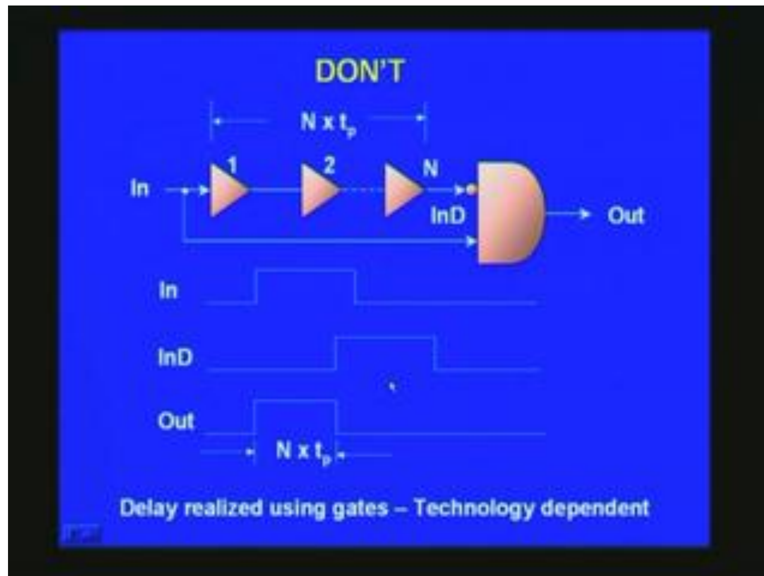
Another mistake designer does is gate the clock. This is very detrimental because it introduces skew in the clock. If you have so many registers in your system and the clock may arrive at one point of time here and the same clock may arrive at different points of time; owing to which we will be violating set up and hold times which we have seen earlier what they are. The solution is once again breaking this gating of the clock and use in this fashion.

(Refer Slide Time: 08:20)



After all what you want is to process the data for different conditions which you can do by using a MUX like this. Your input data can be fed to the one of the inputs and you can interpose once again a register here, D flip-flops here. A MUX output being input to the D here and once again system clock separately routed to the clock input of flip-flop and Q fed back to one of the inputs (Refer Slide Time: 08:49). This means D out is preserved when I0 is selected. I0 can be selected if select is 0. If you want to select the actual input which is the combinational or a synchronous input select will add 1 here, this is clear from this picture. This will be 1 and then this input will go through this path and land up into this register which will be registered at the positive edge of the clock that will be stored here. What is stored here will also be fed. This is a combinational path here. Once the select goes to 0 it will register the same input which was the previous output here. I think this is clear to you. That is what it says the logic must be incorporated in data input instead of gating the clock.

(Refer Slide Time: 10:00)

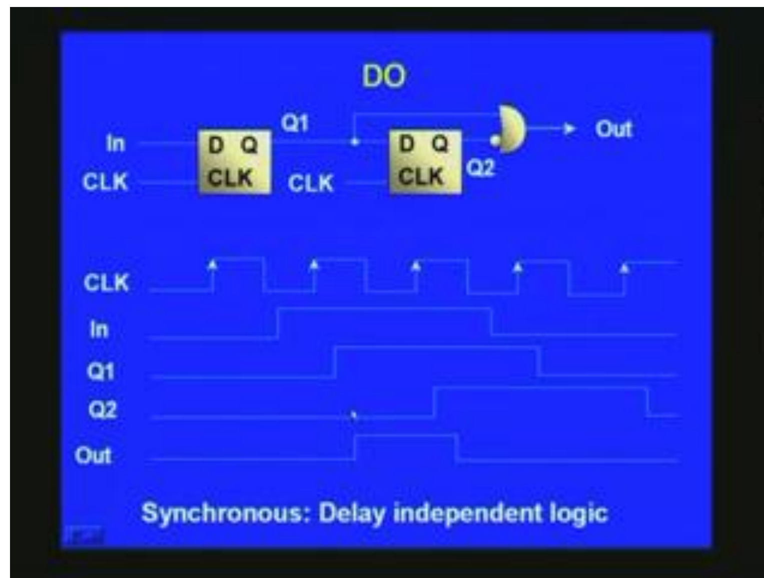


Another point is normally we would like to create pulses. Let us say we wish to create a pulse for a typical application can be a timer say, photography timer you want to produce single push button activation and you want to stimulate that so you need a pulse of that kind. It can be either single pulse or it can be long duration pulse. In the traditional way designers normally use an erroneous design practice. We will need to put different buffers in order to achieve this n times propagation delay. For example, t_p is the propagation delay for each of this buffer. You have a total delay of n into t_p at this point. This is inverted here and fed here in the gate and also input is fed to this gate with a view to get this final output, positive pulse from the 2 inputs that we have here. This input is basically like, as I said a push button switch, of course you cannot straight away have proper de-bouncing and that is implied here.

This output here is nothing but the same input after a delay. This is the delay that you have. This delay is turned by the total numbers of buffers that you have, the total delay appears at the output also If you combine these two in this fashion with a note the bubble here then out comes the desired single pulse the duration of which is given by this n times t_p. This is the traditional way but this unfortunately depends upon the technology. For example, if you are in point 65 micron technology, you might have produced let us say 100 units of time, say 100 milliseconds or something. The technology has changed say currently it is on point 09 micron technology. Naturally several-fold has galloped may be 8 to 10 times. In which case, if you had not changed

the design and but you have changed the device, then what will happen? The time delay that you have put here will not be achieved in the current technology so it will be 10 fold less it may not be enough to trigger some other mechanism that you might have any point of design that it is a technology dependent.

(Refer Slide Time: 12:59)

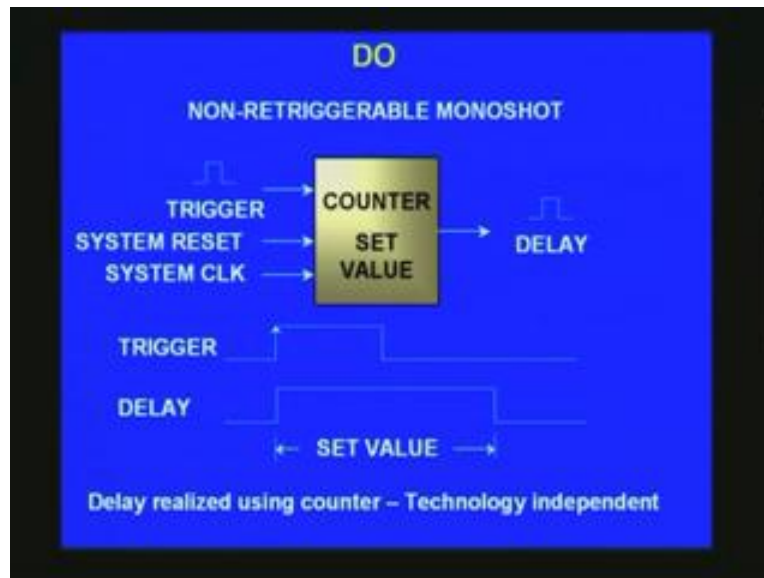


So the way out here is if you are interested in just creating a single pulse like this so what you can do is you can have two flip-flops and input is applied here and that is here . It may appear at any point of time. Let us say it appears here. There is a clock here and we are going to reckon the whole thing on positive edge basis. First flip-flop output is Q1, naturally when rising edge clock is encountered here at this point of time In is 0; naturally the output is 0 here. When it is subsequently it is 1 here but, this will be registered only at the positive edge of the clock.

It should have happened here but going to delay here between the clock and the Q output. This is naturally staggered a bit here that is a delayed bit and otherwise this is nothing other than the input itself. This in turn is applied to another flip-flop and that output also can just have a look. That is also a delayed output of this for Q1. Once again we get it here in this fashion. Ultimately we get a pulse like this (Refer Slide Time: 14:20). Notice that actual width of this pulse is actually the width of this clock period. Except that there is a delay here owing to the delay between clock and Q because internally there are circuits and any gate means delay that is what

is getting reflected finally. This is how we solve for a single pulse if you desire just a single pulse to be created. On the other hand, if you want high timings say of the order of milliseconds or even seconds or even beyond; we have already seen one application called non retriggerable mono that is what is put here.

(Refer Slide Time: 15:07)

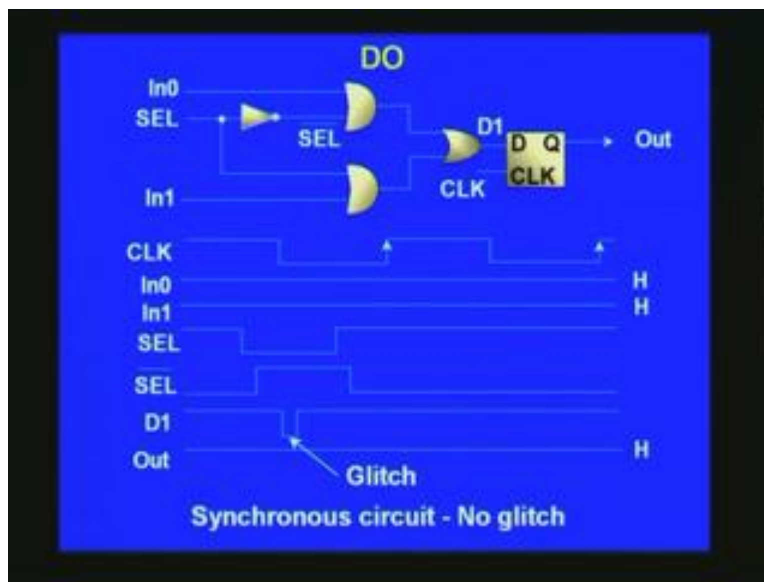


Just to recollect what we have done earlier it is based on a counter. It was an 8 bit wide. You had a setting also built there. You can trigger this basically a monoshot or you can view it as a timer it can be triggered - that trigger is a starting of the delay. The desired delay is the same as in the previous case except that this delay is now controlled by the set value that you have already programmed it in. It has a system reset and a system clock. The simple wave form for this is when trigger is encountered at the rising edge of this; delay also starts here. Whatever is the set value here you will get exactly in number of clock cycles. Had you set 255 inside you will get exactly the same 255 clock cycles.

Now you would notice as in the previous case, where a single pulse was produced as well as in this particular example it is technology independent. It is dependent only on the system clocks you can see in the previous thing (Refer Slide Time: 16:30). This final delay is based on the clock and not the technology. Even if technology changes, any future technology that would come your design investment that you had made and adopted this 1 of the 2 schemes is still

preserved. That is how we say it is a technology independent design. A designer will have to keep all this dos and don’ts in mind prior to actually designing any circuit.

(Refer Slide Time: 17:06)



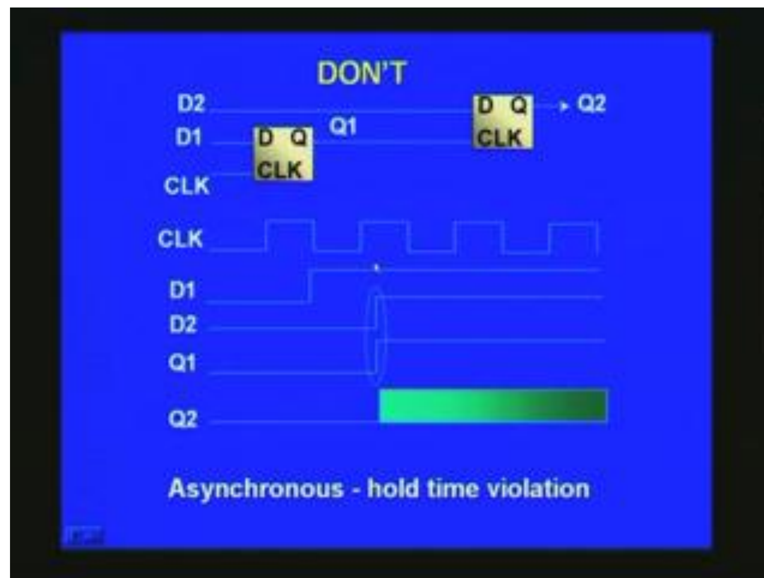
Another thing we talked of earlier is the glitch which is an unwanted uninvited guest that you would like to eliminate. For example, this circuit gives a glitch as far as this combinational circuit is considered and a narrow pulse like this is known as the glitch (Refer Slide Time: 17:30). How is it produced? We will just have a look. These are all plain AND gates here with an inverter here. Finally, these 2 OR are here then applied to a D flip-flop. Let us consider a case where in In0 and In1 both inputs are high. In such a case what will happen to this circuit let us have a look. Let us say this select goes from high to low as in this case. Naturally, after inverter this select bar will be the inverse of that so here and of course that is the delay here this gate delay is because of this inverter here. If you see the output here at D1 what happens? It is nothing but ORing of this if since In0 and In1 are 1 each.

Whatever you apply here is select bar here and select here; this will appear at the respective ends here; this OR you get basically this precisely this; this is caused because you can just see here this is 1 and this is 0. Being an OR 1 plus 0 is or the other way it is going to be 1 just as here also 0 1 here. All through it will be 1 except for a small interval here, at this portion it is 0 here and it is also 0. This 0 will produce 0 here right at the D1 that is how you get the glitch here. This is

contributed by the inverter. The propagation delay of the inverter is responsible for creating this. If you had processed this, say, elsewhere, without the flip-flop what will happen? It may happen when your circuit is sampling, it may sample at this point, that is not the desired point. You wish to know the sample only when actual D1 is high, but you might sample here without your knowledge. That will play havoc had you given this as a clock input for some other circuit. It is very essential that we get it of this glitch at all costs. The way to do is put the flip-flop here. What happens if there is a flip-flop here? Naturally can be assured that there are other flip-flops from which other signals are coming it may even be asynchronous in which you cannot directly put here you need another flip-flop and then only process other asynchronous input which we will see in the later slides.

This is happening, all this glitch will happen before this rising edge of the pulse; total delay of this one you should keep in mind and then only select your frequency. The frequency operation is given mainly by the total delay you encounter in a combinational circuit. If you had taken this into account then you can always condition your whole circuit in such a fashion that whatever glitch appears will be ignored and steady state value only will be registered. In this particular case it will be registered only here. We make sure that all these unwanted signals such as this glitch occur prior to this positive edge rising edge. Then you will be safe and not register the glitch but, it will only register the steady state output here. Prior to this you would have registered this here provided that is the logic that you have. This is possible only if you have a synchronous circuit.

(Refer Slide Time: 21:37)



Another thing normally designer does is, as in a ripple counter you give the Q output of one flip-flop into another and to clock of next. This is another practice which is not encouraged. In fact, it will create problems. For example, you may violate whole time if you do not take one particular state. This is the clock and D1 D2 are coming from external source and suppose D1 changes somewhere here arbitrarily which is here; D2 changes here, simultaneously let us say Q1 also changes. This is possible because D2 is not in our control. It may appear at any point of time so what prevents you from getting that D2 change when there is a rising edge for this clocks you cannot. In such an event what will happen? You are actually violating the whole time. So, actually these are all the steady state of the previous clock.

Normally this will not host any problem that means it already satisfies the setup time. What is going to be violated is the actual whole time. When clock rises, data is not stable. As per the whole time the data will have to be stable for t_H time beyond that. Naturally this is violated and you have no control over that. It has to be eliminated from it. The way out is, after all you may decide to have a counter basically. We have already seen how to implement a counter and that implementation actually confirms to the RTL coding. In fact all that we have considered right from beginning, it confirms to the RTL coding style. Now what we are formularizing is the need for following that style, some of that we have seen already.

(Refer Slide Time: 24:01)

Separate Combinational and Sequential Circuits

- Easy to read and self-documenting

```
module rtl_model (d1, d2, clk, Q, Q_n)
        input d1, d2, clk ;
        output Q, Q_n ;
        reg d, Q, Q_n ;
always @ (d1 or d2)
        begin: COMBINATIONAL CIRCUIT
                d = d1^d2 ;
        end
always @ (posedge clk )
        begin: SEQUENTIAL CIRCUIT
                Q    <= d ;
                Q_n <= !d ;
        end
endmodule
```

Earlier, we have seen the basic RTL coding style we will follow the pattern as depicted in this figure (Refer Slide Time: 24:18) first combinational; then register that and then follow it by another combinational register and so on; this is how a data flows. It flows from one point here; then it flows to this; then sets down here and keeps on going stage after stage. It is the same pattern we have followed there in that particular code which we have. We mentioned earlier that we need to separate out combinational as well as sequential circuits. Here it gives one particular module. A module is, you can regard it as a single file in which you need to follow certain patterns while coding. What we have seen earlier was only piece-meal coding and now it gives little more information here. For example, it says that the code that you are going to put is some module. A module is something like a black box; you have a block diagram which you are already familiar. It will have an IOs that is precisely what you have here. That block can be given a particular name and the block itself we can call it as a module.

Once you have a module you should signal to the tool that is going to process further by a corresponding signal called endmodule. Note that there is no gap after end and module; it is a single word. Module and endmodule; this is a vital thing that you have to keep in mind. Once you have identified it as a module then you have to give a module name. For example, this is a model for RTL, so I will name it as RTL underscore model (Refer Slide Time: 26:15). Like this you may have several such modules but it is illegal to nest modules within modules. The nesting

itself is putting a module within module. For example, I cannot say somewhere here before the end module, call, another module, sequential module. I cannot put another module that is an illegal thing in Verilog coding.

Once you identify the actual module, this is the module name then list the IOs here. Put it any form you want; IOs can be in any order that you like. These are all separated by commas and this is nothing but a D flip-flop which we have already seen. Only thing is we point out here the need to separate out the combinational as well as sequential circuits. That is what we are doing here. List all the inputs which happen to be D1 D2 and clock here. You can put it like this or separate it out in different statements. So is the case with output; you just list them. We will see more details when we actually put all the codes that we have learned earlier in one file as a preparation for simulation later on. Once you have just 2 blocks, 'always' block will come to this statement little later. This is an 'always' block which you are once again familiar with. This is just a combination realization whenever D1 or D2 changes. For example, if D1 was 0 earlier; now it changes to 1 or let us say D2 was 1 earlier, now it changes to 0.

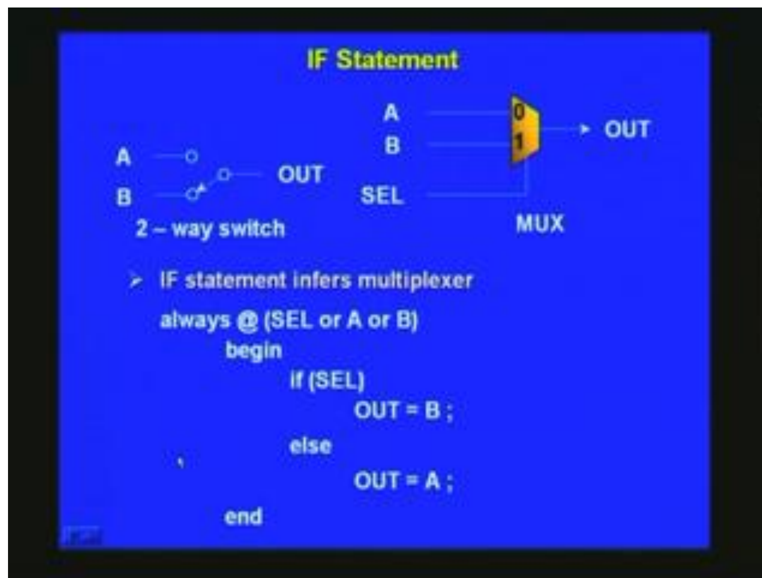
Only in the event of change occurring this will be processed. You notice that we have given a name here after begin you are already familiar with 'begin' and 'end'; what has come as extra here is just a colon here and combinational circuit. That means to say, whatever block that you have here you can give a specific name and perhaps call it later on when the need arises or we can just for improving the readability you can have this. It is a good practice to have this and this does nothing more than take an exclusive OR of 2 inputs and assign it to D that is what this logic is doing. This is just a D input for a flip-flop and the flip-flop is written in segregated block here which is also an 'always' block; action being taken at positive edge of clock. For simplicity, I have omitted that negative edge of reset which we had considered earlier and once again in 'begin' and 'end'; note this once again. This was identified this block as a combinational circuit and this block as a sequential circuit. Here what you need is whatever is the data input we nearly had two registers when the positive clock occurs edge of the clock occurs. This also we have seen earlier. This is nothing but an inversion of D and if you look at this synthesis tool later on you will see that you will be surprised (29:49) (Conversation between professor and student) (30:04).

Here we are inverting this D; you will be surprised in the synthesis tool to find 2 flip-flops there not just 1 flip-flop. Our intention was just to make 1 flip-flop and the other one is merely an inversion of this. Unfortunately, tool has not that much intelligence and as a designer you have to know the limitations of the tool; then find out ways and means to circumvent those limitations. One way to circumvent is, do not put this statement here; use an assign statement outside here and then assign Q underscore n equal to this exclamation mark d or you can even put tilda here - sign wave like pattern. Coming back to this, we have not set what register is. You note notice that the outputs in this 'always' block are d, Q and Q underscore n. These are all registered values that mean it is basically a flip-flop.

It is not really a flip-flop here in the usual sense of a sequential circuit but, still it memorizes that event because whenever there is a change only then we take action. Some sort of storing is taking place there. Whatever output is defined in a particular block - always block; those things you declare as a reg; as a general rule. Here it was an intermediate output this d which is nothing but exclusive of 2 inputs and you declare this d as a reg here and that is what we have done; naturally Q and Q underscore n are reg being the D flip-flops outputs.

Another thing is if you code it like this, segregating combinational or a synchronous circuit and to make it different from sequential circuits, then it will be very easy to read your code and you can minimize your comment writing also. Naturally, that itself is a self documenting feature here and that is the advantage in following this RTL coding style; this is a typical model of a RTL coding (Refer Slide Time: 32:38).

(Refer Slide Time: 32:40)

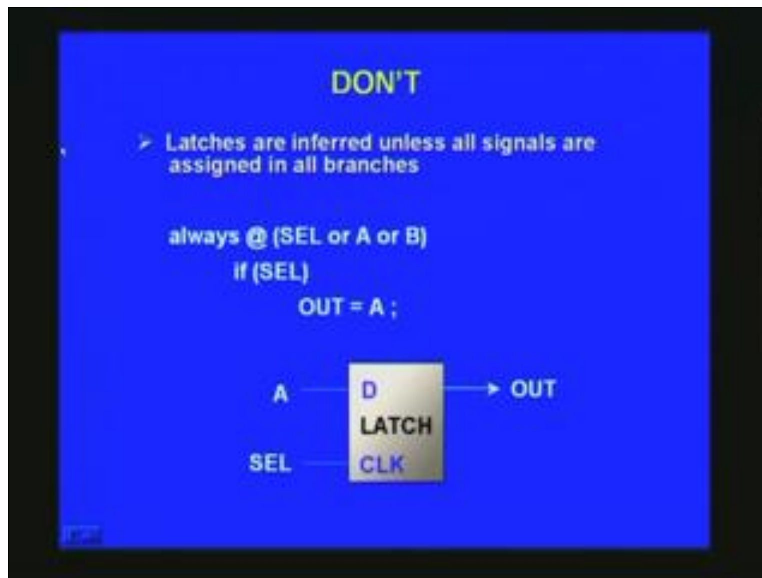


We have also considered 'if' statement; we revisit that one in order to see what exactly the synthesis tool maps. We are all familiar with the 2 way switch you would have used a band switch in radio just for a short wave and medium wave basically; a SPDT shown here is Single Pole Double Throw. It rests in either this or that position in a mechanical switch. You can stimulate this one and we can get out either signal A or B in this case depending upon this rocker. This digital analogy for this is some MUX which you are already familiar with A B are precisely the same here, so also the output here. Select is not apparent here but it is the mechanical control that you have here and that is stimulated by 0 or 1.

This you know anyway; 0 selects A and outputs here otherwise B is output there and 'if' statement therefore infers a multiplexer which we have already seen earlier; we will see it once again here. It is 'always' block; it is a combinational circuit and that selects inputs A and B. Suppose if u want output here it should be B so what should we have select must be 1 here; then only B is selected that precisely this statement means. If select is if you do not put equal to 1 which you can also do it implies that it is 1. That is a short notation; but I personally discourage this you put double equal to then 1 stroke D1 that will convey how many bits and all at one stroke. As we said before it be should as much self documenting as possible; so as to minimize your comments which you need to be the bare minimum; explaining the modules rather than every statement that you have written. If select is 0 it will not process the statement, this 'else'

will take into effect and out will be assigned A now. Notice that we have fully covered all this possibilities here; the only possibility is here select. So you have only 2 states; we have covered completely here and that is how it is basically a MUX which is precisely same that is what we said it happens; if statement normally infers a multiplexer.

(Refer Slide Time: 35:30)



Another thing that you should not do is omit that ‘else’. In the previous example we had an ‘else’here if you omit that ‘else’what happens, the tool will infer it as a latch. A latch is very detrimental. What is a latch? Basically you have an A input to D input, A B applied to the D input as long as A is varying the output also will just follow suit is nearly transparent so long as the clock is 0. That is, when clock is not applied whatever is applied to A without any hindrance that appears at the output, What will happen here? It keeps on changing here. If you had mixed it with a synchronous circuit it may latch register at the undecided point. You should avoid the latches here and the reason for creating the latches you have omitted ‘else’.

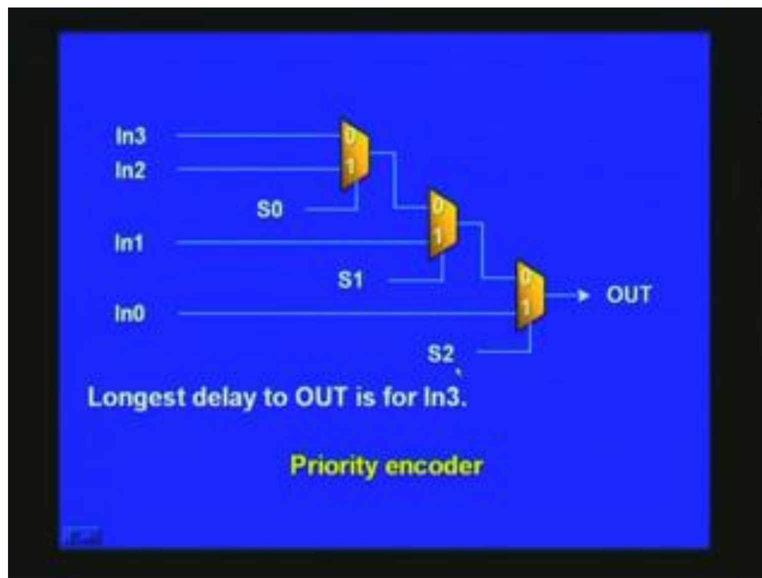
(Refer Slide Time: 36:37)

- Priority encoders are inferred by IF-ELSE-IF statements

```
always @ (S2 or S1 or S0 or In0 or In1 or In2 or In3)
  begin
        if (S 2 == 1)
        OUT = In0;
        else if (S 1 == 1)
        OUT = In1;
        else if (S 0 == 1)
        OUT = In2;
        else
        OUT = In3;
  end
```

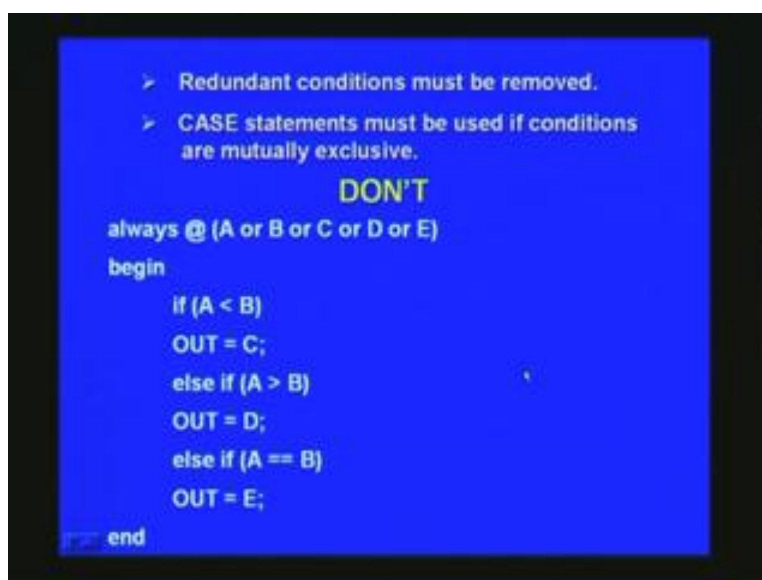
Next thing we are going to see is once again already covered it is priority encoder. This is the ‘always’ block which brings out the priority encoder. This is given the top most priority. It can regard it as a single 3 bit or as a 3 different inputs base S stands for select. We have three selects here and we have 4 inputs, this is also a MUX basically. You need to get either In0 or In1 and so on depending upon S2 S1 or S0 in this fashion. For example, if S2 is asserted then what you need is at the output of the circuit that is here In0. If this is not satisfied only then it will go to this so not until then. It implies that S2 is 0 if it had come anywhere here it implies that S2 is 0. If 1 it would have passes this and got out of this loop; otherwise if S1 is 1 you assign In1 here this is the case for In2 and otherwise here what I want to point out is this you are already familiar. What is the point in repeating here is how the synthesis tool uses this and how many MUX it places. That is the point I would like to show you.

(Refer Slide Time: 38:11)



The very first 'if' statement was for S2. This is the top priority as I mentioned, that means, if S2 is 1 this input is straight away output. Gate delay here as far as the input is concerned it is only from here to here. On the other hand, if you go to in selection of the input 1 then it has to go through 2 MUX not occurred at the output. So the case with this and one of this will be the longest path encountered. That is how you get a priority. The price you are paying is 3 MUX delays.

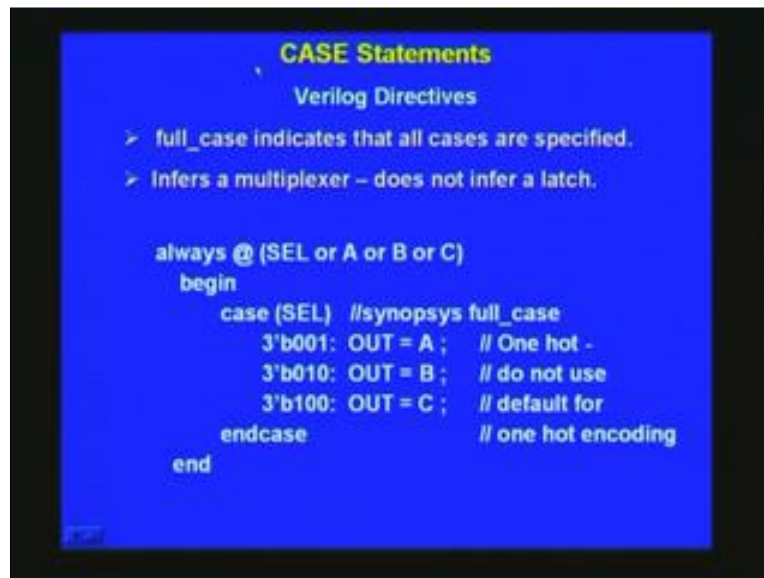
(Refer Slide Time: 39:02)



This is another case where redundant conditions appear and you will have to remove that. You have to identify where the redundant case is and remove that what is redundant. We will have a look at this and do not follow this type of coding, if you do you will create additional MUX and hence additional delay and that in turn will slow down your frequency of operation. How it does let us see. This also we are familiar with comparator we have done earlier. If 3 numbers are there here and basically two numbers are being compared and C D E are outputs here and if A is less than B, set C as the output. If it is greater than B, set D as the output. If none of these that is that implies A equal to B, then do this. What is redundant in this statement can you spot it out? I will tell you, A less B implies if this is not satisfies it goes over here. It implies that it is either equal to or greater; this statement takes care of the next possibility.

What is left is only equal to here, what is known, why do you want to put another condition here? If you put this condition what happens it creates one more additional MUX as we have already seen; one MUX for this if statement then else if one more MUX and one more MUX here as we have seen earlier; precisely the same. Is there any other possibility? The solution is simple and you can dispense with one of the MUXes; so this is precisely same and after A is equal to B, D is assigned 2 Mux already pressed in service and now when we come to this step it is already A equal to B. Instead of putting 'else if', just put 'else'and then E can be straight away assigned here and by doing so notice that 'else if' combine with 'else'both means single MUX only as we have seen earlier also; that is how we say one MUX and thereby improve the speed of operation.

(Refer Slide Time: 41:35)

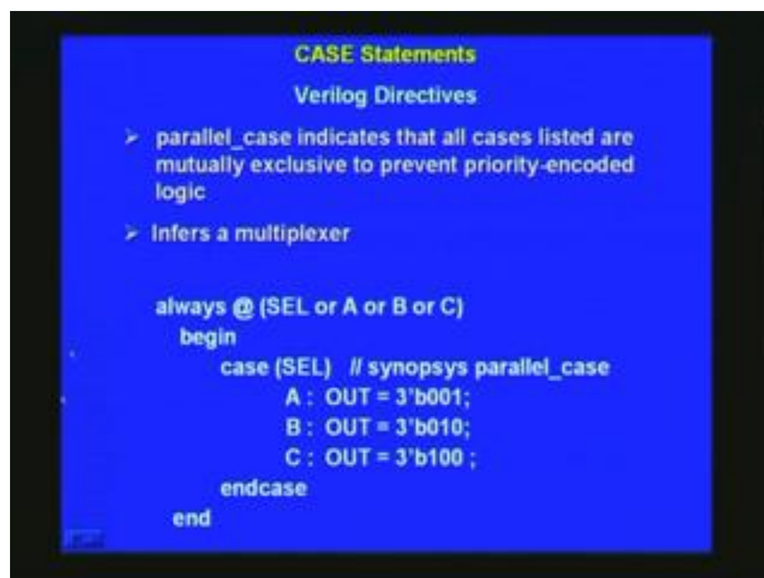


We have also considered case statements and we will now see what Verilog directives are. There are basically 2 directives called full case and parallel case; this indicates when you have a case statement here. Let us consider 3 bits of information and if all the conditions are specified here obviously it is not specified here. For 3 bits you have 8 possible states and three such states are only are considered here. What happens to the rest? When such a condition is encountered will it be encountered? That is the first question we will have to put our self and if it is encountered we have to see remedial action for that. For the time being we will see how this does. There is a directive called synopsis full case full under case this looks like a comment but it is a directive. It is more than a comment. Take care that you do not put this as any of your comments; if you do you will get into trouble. What all it signifies for the synthesis tool or even a compiler is that this we have given only partial list here we have not given the entire thing but you regard it as a full case; that is what the implication here is so just by giving this here you do not have to list all the conditions.

Before we consider that one, let us see what this does. This infers a latch for the output since all possible cases are not specified. Here, as we have seen earlier, once again 8 such possibilities are there of which we have considered only 1 here 1 here 1. We have not considered 1 1 and1 0 1 and above. We have not considered 5 different cases here and we have not even told what to do about it (Refer Slide Time: 43:40). We have not put here either full case that directive. What will

happen in this case? Because this is not specified and there is not even a directive, simply this tool will infer that this is a latch. That means, let us say it has passed through for a particular condition. Let us say it has passed through this and it has previously stored d let us say because this is A. Although this is a combinational circuit this may be ultimately used in a sequential circuit wherein you may register. Still here also it does matter and what will happen so the previous value was this. Now the present case let us say, one illegal thing as come so it is not in any of this. It looks here it finds no match. What can it do? What can it infer? It will be confused. It does not know what to do. What it will do is merely latch whatever was the previous signal. In this case it came from B, it continues to be in that state and this may be detrimental for your circuit design.

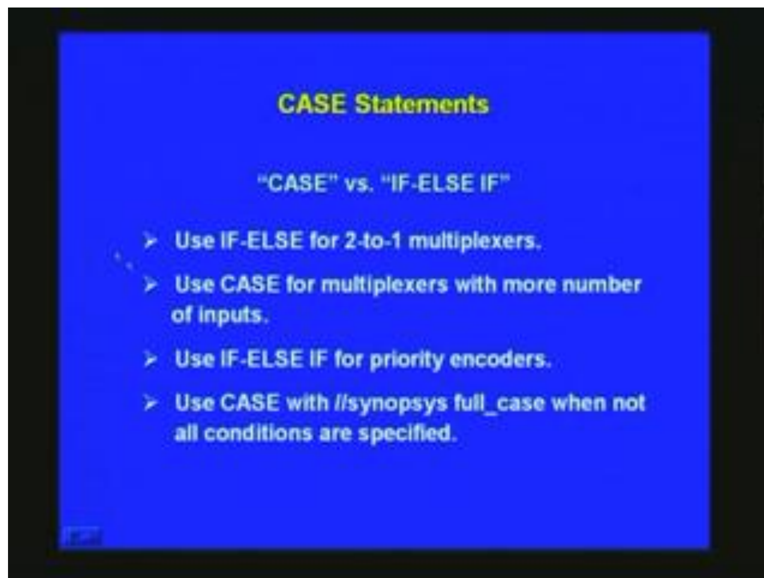
(Refer Slide Time: 44:51)



Another Verilog directive is a parallel case which indicates that all cases listed are mutually exclusive to prevent priority encoded logic. We have seen 'if' 'else' 'if' combination to be priority encoded and case to be not priority. It appears straight away whatever be the condition here it appears straight away at the output either here or here all equally at the same time. There is no nesting as such as in the case of 'if' 'else' 'if' wherein you had normally nesting takes place between MUX. It straightaway appears at different statements; here the directive for such a case what mutually exclusive is, if you observe this one this is completely different from this. It looks like a one hot machine; so you can see 1 here, 1 here and then 1. All this cases are mutually

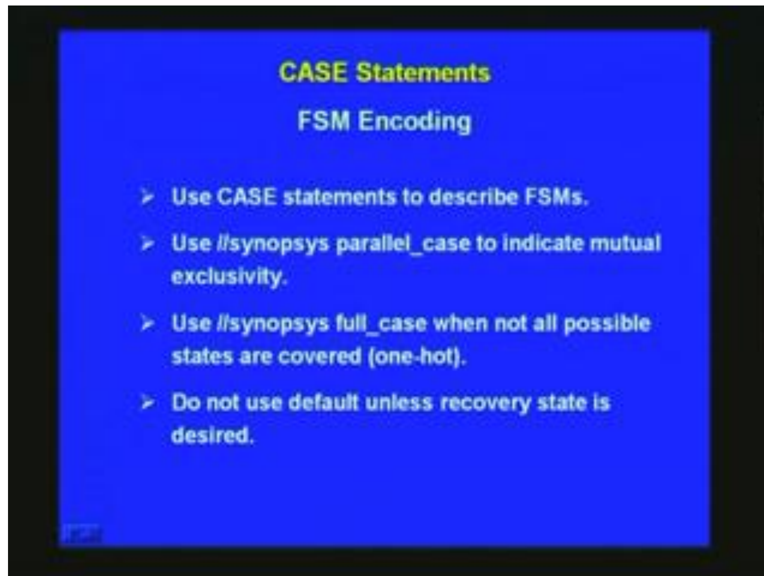
exclusive and based upon the actual input you can output either this pattern or this pattern depending upon the actual input itself themselves. That is the style adapted for synopsis parallel case so you have to tell once again that only these mutually exclusive cases are being considered here. If you do not put this once again it will infer latch like the previous case. Notice that this full case and parallel case it is not a must that you should adapt this. If you are not happy with it you can refrain from using them and use the conversion method which had been already covered in the first, second lecture or third lecture that also we can see here. Another point about this full case and parallel case is, this particular thing may not these statements directives may not work in some of these systems. They may be simplicity, may not encounter one of this I am not sure we will have to experiment with it and find out. It is most likely that it will work in synopsis platform which is basically a tool for basic design. So as far as possible this full case parallel case also you try to avoid and in fact you can write code although it may be little longer code but, it will be safer code if you go the usual way.

(Refer Slide Time: 47:25)



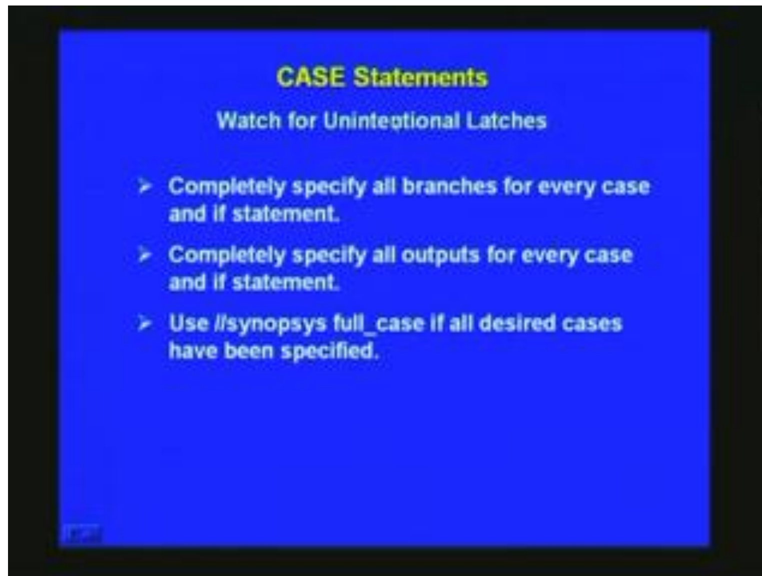
To summarize this and make a comparison between case and ‘if’ ‘else’ ‘if’, it is listed here. We have already seen if your intention is to make 2 to 1 MUX use always ‘if’ ‘else’ statement. If you want more than 2 inputs you use case statement. If you want priority encoders we have already seen ‘if’ ‘else’ ‘if’ and you want to have a synopsis fill case or parallel case use case here and full case when not all conditions are really specified. That is what we have already covered.

(Refer Slide Time: 48:02)



While writing an FSM state machine you can use case statement which we have already seen couple of examples before and use case statements to describe basic state machines, finite state machine this also we have already seen Synopsis parallel case to indicate mutual exclusivity and full case when not all possible states are covered as for instance one hot machine you can use here. Do not use defaults unless you want a recovery state. We have one of the examples we considered earlier. We have used that recovery state as a default. We took it to a safe initial state S0 condition in while looking at an FSM earlier.

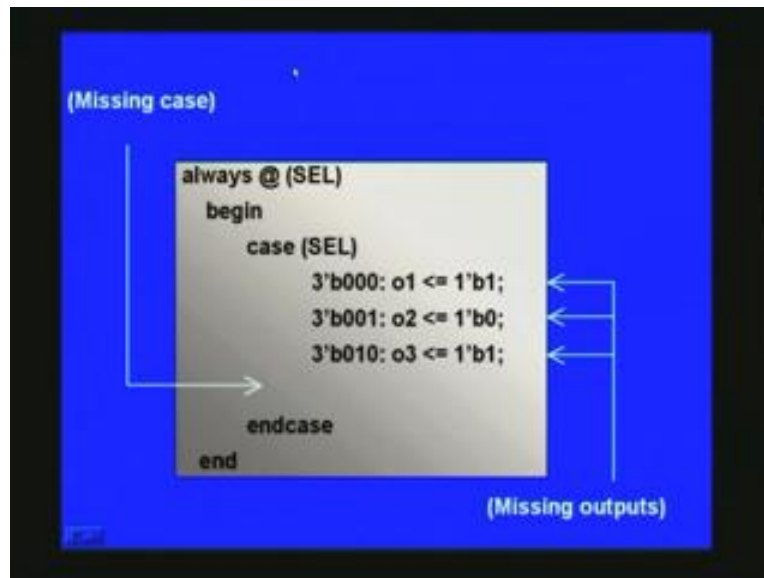
(Refer Slide Time: 48:55)



You have to be very watchful otherwise you will be creating unintentional latches which we have already considered. In order to do this completely specify all braches for every case at every case statement for every possibility you have give consider it wholly so also for if else if statements you have considered all the possibilities. Do not leave anything to chance. Otherwise you will get into trouble later on with synthesis tool itself it will not permit to go beyond and even if you wait through all the tools finally when you make the chip you will see that will not work when you took up the circuit or it may still work and years later on customer end has been functioning and we have changed the technology mean while then it may malfunction.

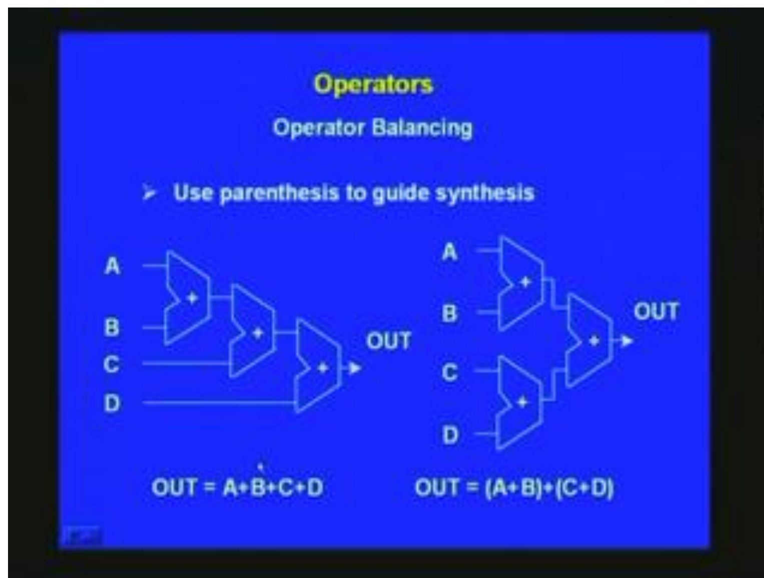
For example, the time dependant delays which we have covered earlier will be detrimental here. Similarly, completely specify all outputs for every case and 'if' statement we have seen this also in always block earlier using case and we had some eight outputs such as D0 through D7 and we took care at every case statement that we had reset all other unwanted outputs other than the one which we wanted to set high that is what it implied here use full case if all desired cases have been specified.

(Refer Slide Time: 50:31)



Before we wind up let us see what is wrong with this particular example. We have an 'always' block and this will clear any doubts what you had earlier. For example, you have not covered all the 5 cases here what is missing here. This will create a latch as we have seen and outputs are not specifically mentioned. For example, o2 o3 what will happen to this? It is not mentioned here. You have taken care for 1 output but whatever was the earlier output it will continue to latch. Here also you will have a problem. You should take care that you reset all the outputs. Either you can do it here or you can have a statement earlier which will do reset all the outputs right on the top. It will be better although code is lengthier may be better if you stick on here because, verifying your code will be easier for you.

(Refer Slide Time: 51:35)

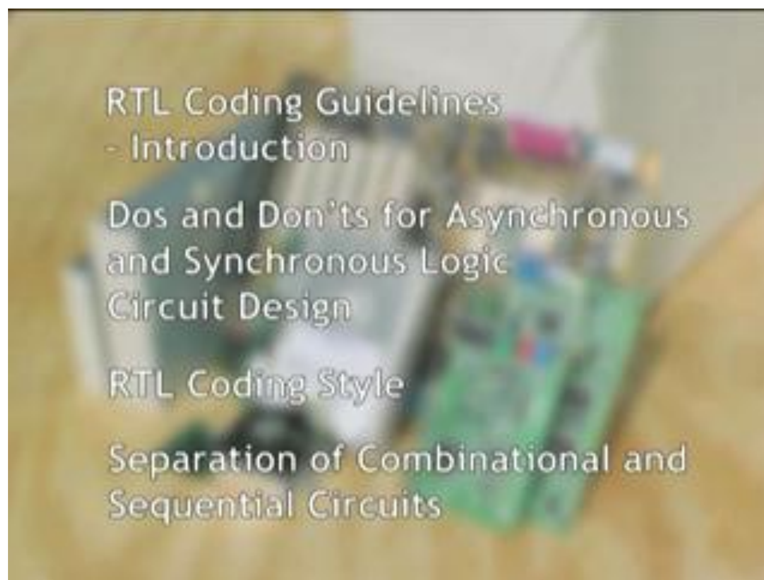


We have one more thing before we wind up. Let us say we want to have arithmetic operation done. Let us say A plus B plus C plus D is what we want to compute. You will have while the tool will put, if you put this condition here, the synthesis tool will map it like this it will put 2, 3 adders A plus B result will be available here. This will be added by C with C and that will appear here and get added the final last added and finally the output will be this. You can improve this speed of processing what will happen you have 3 adders cascaded. So frequency operation gets hit here. A better way to do is guide this synthesis tool by putting a parenthesis here for example if you put parenthesis here A plus B and C plus D here will be computed in parallel as in these two blocks. These 2 results you can once again add up here to produce the desired output which is same as this.

All that we had done is just put parenthesis just to guide the synthesis tool. With this I will be covering RTL coding guidelines. From time to time, if you get any other new points we will cover it as we take up. Next what we will do is, having learnt combinational, sequential and artial guidelines; we will put the combinational sequential codes that we have put earlier as in a piecemeal manner. We will put it together as one file which will be working on this tool which we will be taking up on later sessions. Thank You.

(Refer Slide Time: 53:25)

(Refer Slide Time: 53:28)



(Refer Slide Time: 53:52)

Next Lecture:

Coding Organization – Complete Realization

(Refer Slide Time: 54:18)