Digital Systems Design with PLDs and FPGAs

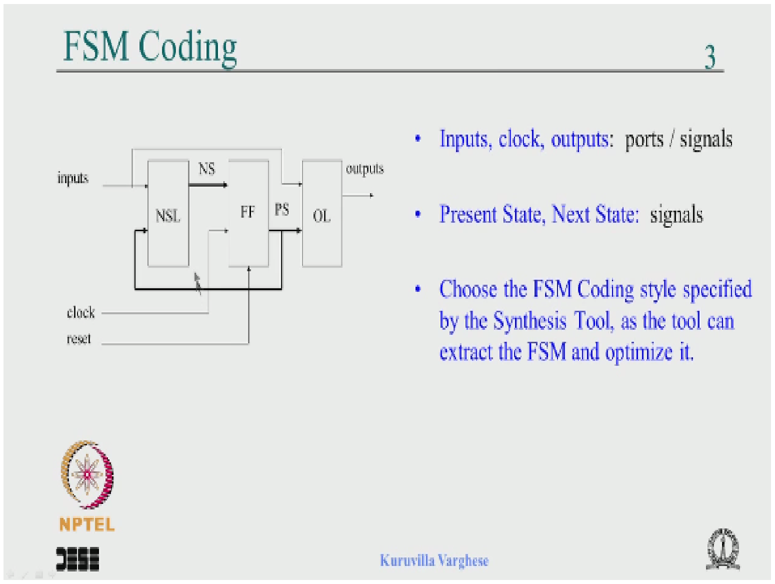
Kuruvilla Varghese
Department of Electronic Systems Engineering
Indian Institute of Science - Bangalore

Lecture-27
FSM issues 4

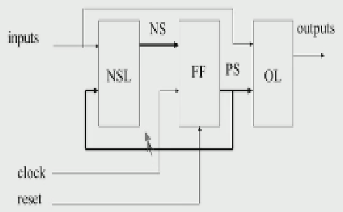
So, welcome to this lecture on advanced digital system design in the course, digital system designed with PLDs and FPGAs. The last lecture we have looked at the VHDL coding of the finite state machine or controller basically we have looked at the various ways you code it code the FSM. And we have taken shown some simple example to illustrate that coding styles.

There are various ways and some schemes are certain requires are trouble it brings synchronise reset we have handle that situation. Today we will continue with issues with the finite statement machine. But before that we will have a quick run through the last lecture slide to be in sink because that may be useful even in this lecture. Let us turned to the slide of last lecture.

(Refer Slide Time: 01:33)



FSM Coding 3



- Inputs, clock, outputs: ports / signals
- Present State, Next State: signals
- Choose the FSM Coding style specified by the Synthesis Tool, as the tool can extract the FSM and optimize it.

Kuruvilla Varghese

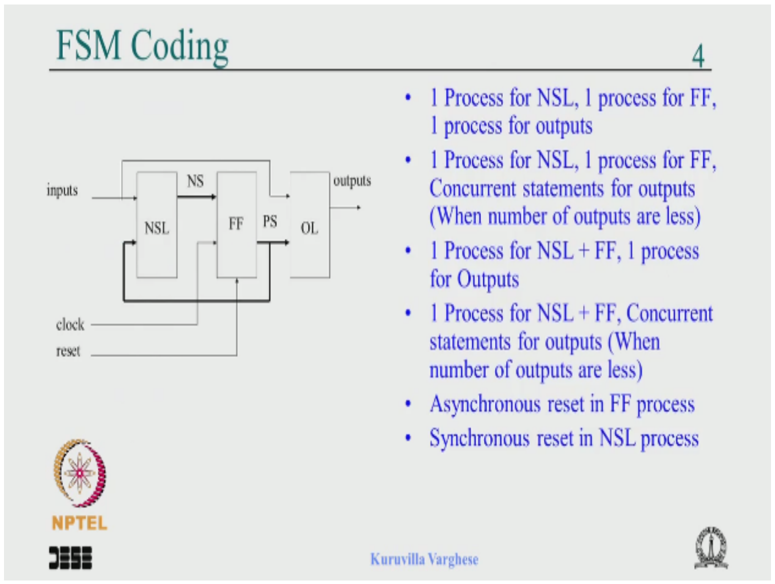
So, here we have looked at the block diagram of the state machine a 3 block diagram, we have first looked at this and we said these inputs clock reset and output definitely clock reset normally comes from outside. So, they can be ports sometime if it is internally it can be signal, but input and output if you are testing it stand alone then it is port, but many a times when you are integrating with the data path.

It could be ports or signal may be some input is coming from outside. So, that could be port and very important thing though we said that you can adopt various ways of coding the finite state machine. You have to choose the style as prescribe by the specified by the synthesis tool. The basic idea is that synthesis tool should be able to infer that this constitute and finite statement machine.

So, that the FSM specific optimisation problem solving can be done on this structure. Otherwise if you just code this as a logic and this is as flip-flop, and this is as logic then the implementation would be correct. it will work perfectly but the synthesis tool will not be able to do any FSM optimisation are any issue it can normally solve it cannot be done example is that.

We have talked about the state diagram optimisation unless the tool infer that it is the finite state machine the state diagram optimisation cannot be done. There will be redundant states unnecessarily wasting area in the next state and output logic and so on okay.

(Refer Slide Time: 03:27)



FSM Coding

4

- 1 Process for NSL, 1 process for FF, 1 process for outputs
- 1 Process for NSL, 1 process for FF, Concurrent statements for outputs (When number of outputs are less)
- 1 Process for NSL + FF, 1 process for Outputs
- 1 Process for NSL + FF, Concurrent statements for outputs (When number of outputs are less)
- Asynchronous reset in FF process
- Synchronous reset in NSL process

inputs
NS
NSL
FF
PS
OL
outputs
clock
reset

NPTEL

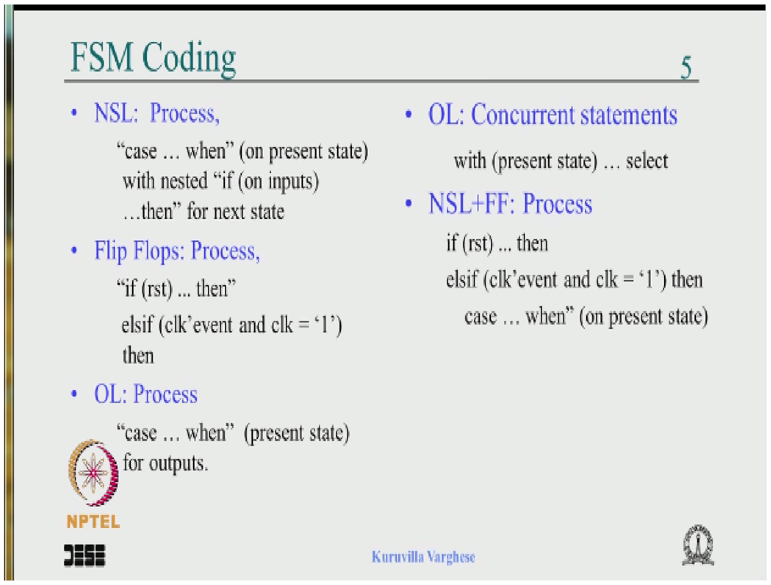
Kuruvilla Varghese

And we have looked at the various ways and here in this 3 block we could have this written in a process, this in a process, this can be written in a process if the number of outputs are greater than the number of states it can be a concurrent statement if number of outputs are much less compare to number of states. Because then there are only 2 outputs you only 2 concurrent statement.

But suppose there are 7 states you may end up writing a case with 7 when, where in only 2 outputs are specified not very kind of concise coding though hardware wise it is makes not different but it is a long code that is why I shall we can write concurrent statement. We also said that you know you can combine these 2 in a single process, because the VHDL. In VHDL you can specify a set of registers with the preceding logic in a single process.

So, that can be done in that case you have a process for this 2 blocks together and with a process or a concurrent statement set of concurrent statements for output logic. And you can include the asynchronous reset here along with the flip-flop. If it is synchronous reset it is included here and when you combine as I said when you combine both together there is an issue. We have discussed that I will just told light on it.

(Refer Slide Time: 05:01)



FSM Coding

5

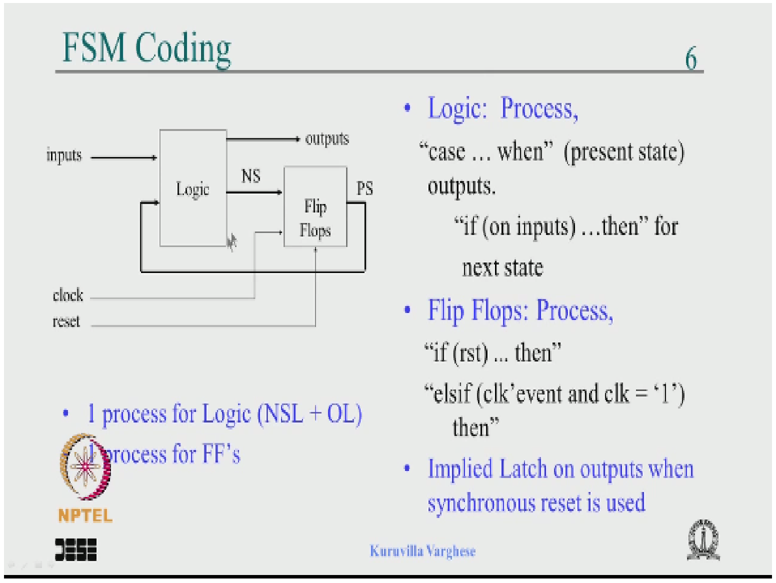
- NSL: Process,
  "case ... when" (on present state) with nested "if (on inputs) ...then" for next state
- Flip Flops: Process,
  "if (rst) ... then"
  elsif (clk'event and clk = '1') then
- OL: Process
  "case ... when" (present state) for outputs.
- OL: Concurrent statements
  with (present state) ... select
- NSL+FF: Process
  if (rst) ... then
  elsif (clk'event and clk = '1') then
  case ... when" (on present state)

NPTEL
Kuruvilla Varghese

And we also said normally when you code the next state logic we write a process with present state as input. And in the sensitivity less and all the inputs, because these 2 are the input with a block and we say case present state is and for each state we specify the transition using if okay. Similarly for Moore kind of output we can say case present state is or with select whichever 1 we use again you can use for the mealy output under the case and so on.

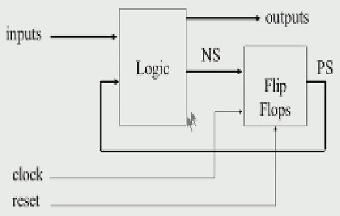
And this is a process where use if statement with the reset and clock event or rising edge clock for the flip-flops okay.

(Refer Slide Time: 05:56)



FSM Coding

6



- Logic: Process,
  "case … when" (present state) outputs.
  "if (on inputs) …then" for next state
- Flip Flops: Process,
  "if (rst) ... then"
  "elsif (clk'event and clk = '1') then"
- Implied Latch on outputs when synchronous reset is used

- 1 process for Logic (NSL + OL)
- 1 process for FF's

Kuruvilla Varghese

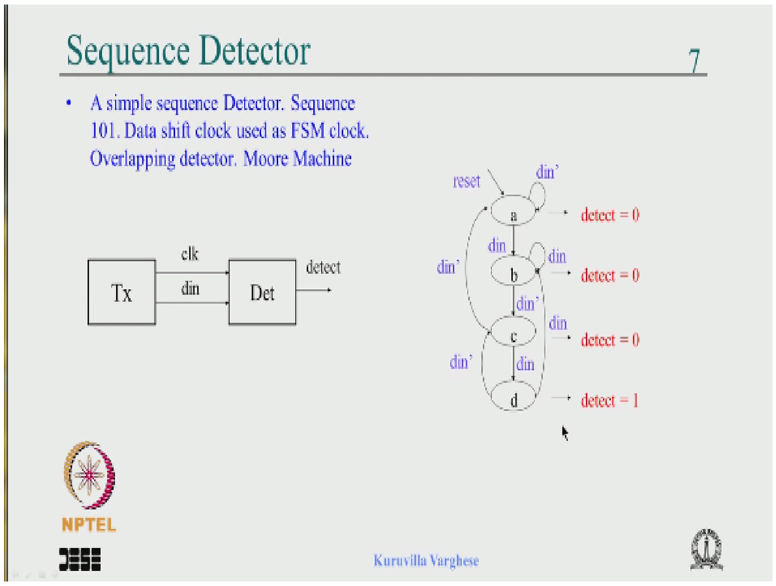
Next possibilities that we come to a 2 block view where the next state logic and output logic is combined into a single block where the outputs are next state logic and the outputs. So, now you can write this in a single process and flip-flop in a process and flip-flop is as usual. But here again you do a case present state for each choice you specify the output if it is Moore outputs straight away.

And mealy then you can write if the n kind of structure and for the transition under each state for each when statement of the case you can specify the transition various transition using the if. So, one advantage of writing this way is that when you have state diagram like quickly looking at the state diagram you can complete the whole logic in 1 shot okay. So, that is one advantage of writing it together.

But if you write as in early then when you write the next state logic coding you have to keep the state diagram run from top to bottom. Then again come back you know write the output logic and top to bottom you write and when you debug something goes wrong depending on where the trouble is you have to go through both code okay. I mean these things are come with experience and for some it occurs a naturally those who were very systematic.

Sometime by their habit, they stumble upon these kind of advantages. But then it all shows that you need some kind of planning experience. Because even writing the code though they are equivalent some style of coding is helpful in a quick kind of design, debug and things like that, that you should keep in mind.

(Refer Slide Time: 08:07)



Sequence Detector

7

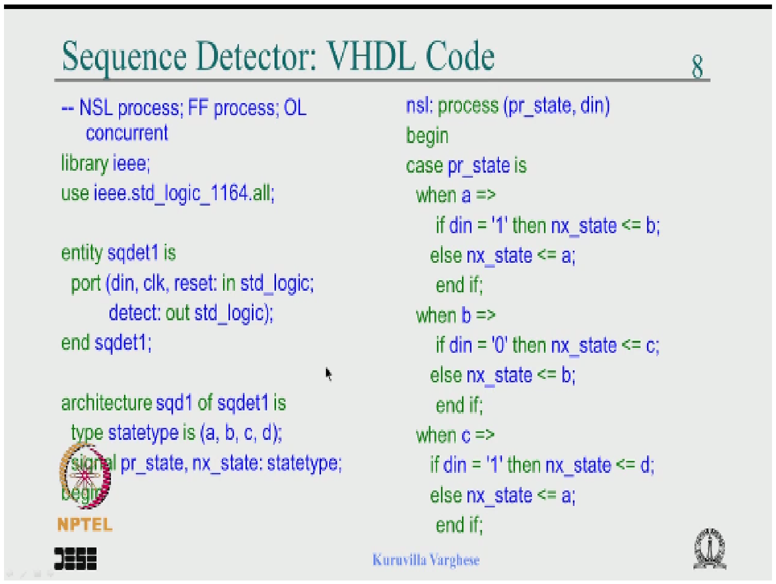
- A simple sequence Detector. Sequence 101. Data shift clock used as FSM clock. Overlapping detector. Moore Machine

NPTEL

Kuruvilla Varghese

And we have looked at the problem of a sequence detector mainly as an exercise not as a serious kind of design I do not think you need fuse an FSM base sequence detection in communication okay you can better than that. You can do much simpler schemes then doing this. And we said it is an overlapping detector, that means like in the case 101 you know you have a 01 here then the 3, 4, 5 also form a detector, sequence detection and we have seen the kind of the state diagram.

(Refer Slide Time: 08:47)



Sequence Detector: VHDL Code

8

```
-- NSL process; FF process; OL
   concurrent
library ieee;
use ieee.std_logic_1164.all;

entity sqdet1 is
  port (din, clk, reset: in std_logic;
        detect: out std_logic);
end sqdet1;

architecture sqd1 of sqdet1 is
  type statetype is (a, b, c, d);
  signal pr_state, nx_state: statetype;
begin
```

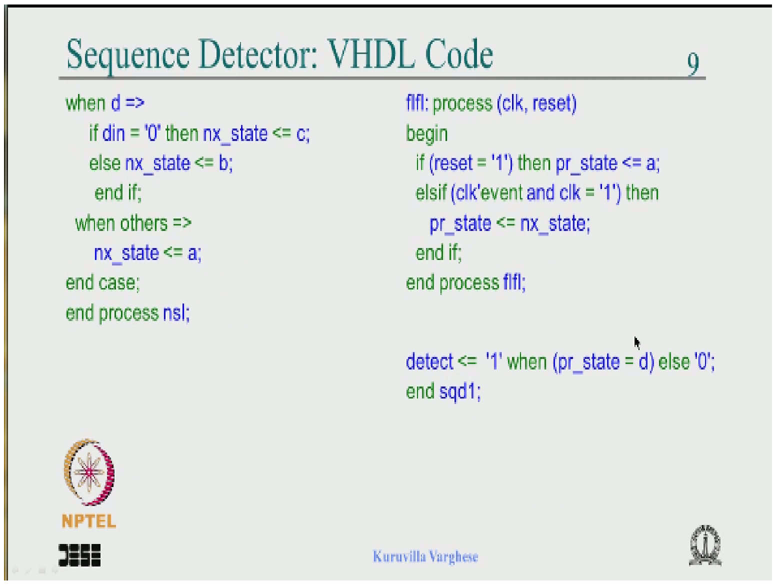
```
nsl: process (pr_state, din)
begin
case pr_state is
  when a =>
    if din = '1' then nx_state <= b;
   else nx_state <= a;
    end if;
  when b =>
    if din = '0' then nx_state <= c;
   else nx_state <= b;
    end if;
  when c =>
   if din = '1' then nx_state <= d;
   else nx_state <= a;
    end if;
```

Kuruvilla Varghese

And we have seen the coding the main thing is that library, entity and the most tool allows you to define an enumerated kind of data type for this and define the present state and next of that type. Then you write a process for next state logic with the present state and inputs in this case only one input is there inputs in the sensitivity less then you go state by state, case present state when a.

Then you use if on that to specify the transition important thing is to do all the transition, if else everything do not think that because you are in state a you do not need to else next state day. Else that will cause implied latch, so you continue through.

(Refer Slide Time: 09:41)



Sequence Detector: VHDL Code

9

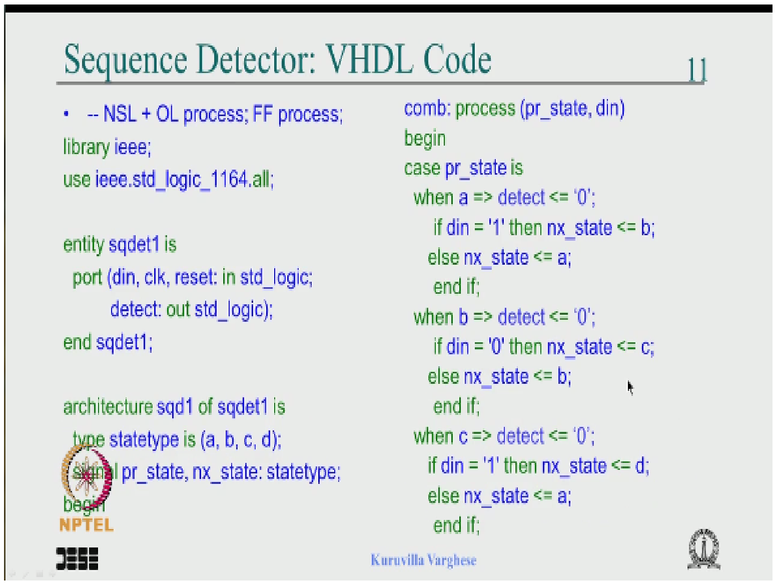
```
when d =>
   if din = '0' then nx_state <= c;
   else nx_state <= b;
    end if;
 when others =>
    nx_state <= a;
end case;
end process nsl;
```

```
flfl: process (clk, reset)
begin
 if (reset = '1') then pr_state <= a;
 elsif (clk'event and clk = '1') then
    pr_state <= nx_state;
 end if;
end process flfl;


detect <= '1' when (pr_state = d) else '0';
end sqd1;
```

Kuruvilla Varghese

And the flip-flop is know like a asynchronous reset you know how to write that if reset then present state is starting state or upon the clock present state get next state and when you write output as a concurrent statement you can use when else or which select. This which select you say with present state select or with when else, you can write like that.

(Refer Slide Time: 10:09)



Sequence Detector: VHDL Code

11

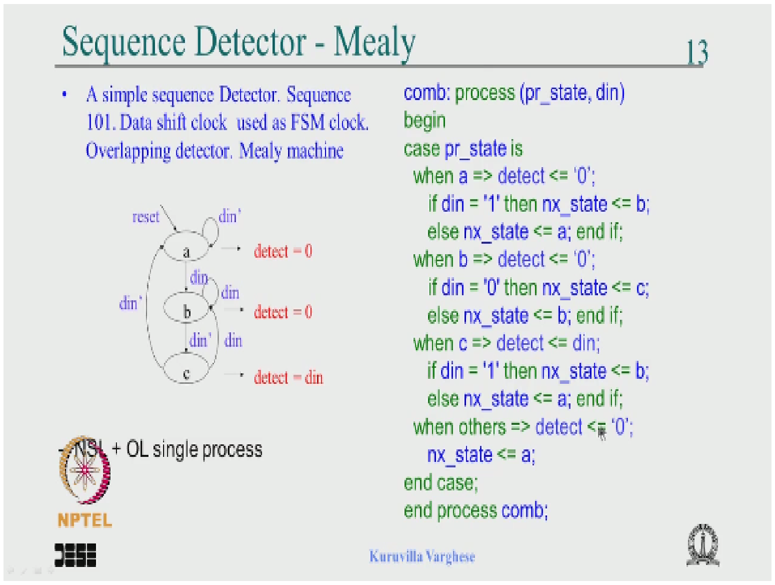
```
• -- NSL + OL process; FF process;
library ieee;
use ieee.std_logic_1164.all;

entity sqdet1 is
  port (din, clk, reset: in std_logic;
        detect: out std_logic);
end sqdet1;

architecture sqd1 of sqdet1 is
  type statetype is (a, b, c, d);
  signal pr_state, nx_state: statetype;
begin
```

```
comb: process (pr_state, din)
begin
case pr_state is
  when a => detect <= '0';
     if din = '1' then nx_state <= b;
    else nx_state <= a;
     end if;
  when b => detect <= '0';
     if din = '0' then nx_state <= c;
    else nx_state <= b;
     end if;
  when c => detect <= '0';
    if din = '1' then nx_state <= d;
    else nx_state <= a;
     end if;
```

Kuruvilla Varghese

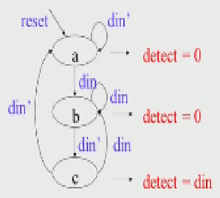
And we have seen the next state logic and output logic together. Only difference is that in the case of each choice of this present state. You specify the transition as well as the output okay. And when is Moore there is no condition on it, when it is mealy you can write a simple logical condition here if it is simple case or you can use if or case or whatever is a appropriate thing to use at that point in time

(Refer Slide Time: 10:41)



Sequence Detector - Mealy

- A simple sequence Detector. Sequence 101. Data shift clock used as FSM clock. Overlapping detector. Mealy machine



NSL + OL single process

```
comb: process (pr_state, din)
begin
case pr_state is
 when a => detect <= '0';
   if din = '1' then nx_state <= b;
   else nx_state <= a; end if;
 when b => detect <= '0';
   if din = '0' then nx_state <= c;
   else nx_state <= b; end if;
 when c => detect <= din;
   if din = '1' then nx_state <= b;
   else nx_state <= a; end if;
 when others => detect <= '0';
   nx_state <= a;
end case;
end process comb;
```

Kuruvilla Varghese

The flip-flop is change then we have seen a mealy kind of machine for the same problem and we have combined the next state logic and output logic, you can see that the detect here it is written as a mealy type detect gets din that means if din is 1, detect is 1 else detect is 0. And many a times when you can write it in a very straight forward kind of logic is better to write like that sometimes what happens is that.

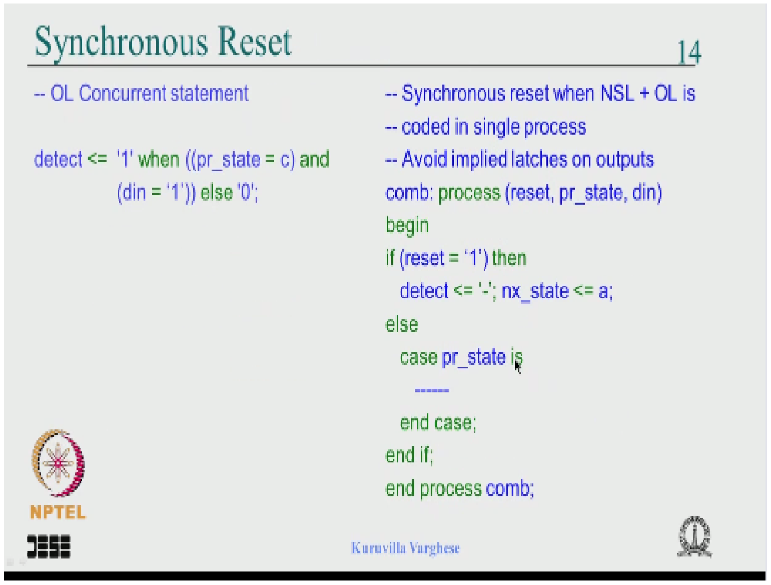
People get use to this syntax and even for simple things like this 1 to 10 write the if saying that, if din is 1 then detect is 1 else detect is 0 and all that. And forget all about what could be the hardware, you know it is nothing you know it detect here. You say decode of c which is in our case maybe q1 bar q0 so, in this case the mealy machine. We are combining the next state logic and output logic in a process.

And the code is similar only thing is that we have this detect which is written as a part of I mean detect as a function of din. So, when you decode it is like you know the state c and din is detect so, it is q1 bar q0 and din okay to and it better you to remember these basics always what happens is that when you one danger with more and more sophisticated tools and these kind of languages.

That you 10 to forget or10 not associate the hardware with it and people like I have seen students writing this simple thing as if din is 1 the detect is 1 else is detect and so on. And forget what is the hardware it is not a good thing I mean I was the some students what is the hardware what is a synthesis tool generate if you write a statement like that it entire sometime at last even with some very trivial silly staff.

So, whenever you can bring clarity by making this things simple you should make it simple, then use all the if case and you know next state and things like that. You know though we are learning about it wherever things can be made simple elegant you should do it.

(Refer Slide Time: 13:21)



Synchronous Reset

14

```
-- OL Concurrent statement

detect <=  '1' when ((pr_state = c) and
           (din = '1')) else '0';
```

```
-- Synchronous reset when NSL + OL is
-- coded in single process
-- Avoid implied latches on outputs
comb: process (reset, pr_state, din)
begin
if (reset = '1') then
  detect <= '-'; nx_state <= a;
else
  case pr_state is
    ------
  end case;
end if;
end process comb;
```

Kuruvilla Varghese

And if the same thing is written as a concurrent statement instead of this if you are writing it outside here we are combining both next state logic together. But if it is outside then you can write like this detect is 1, when present state equal to c and din equal to 1, else 0 okay. So, you can write like that and here to write with select is difficult, because you have 2 inputs**.**

If you are kind of adamant then you can do that what we can is do that you have to kind of group present state and the din together into a temporary signal. Then you say with that signal select and put this value and so on okay but with again it is tough with an enumerated data type. Because here we are using some kind of you know symbolic assignment and here it is a numerical value.

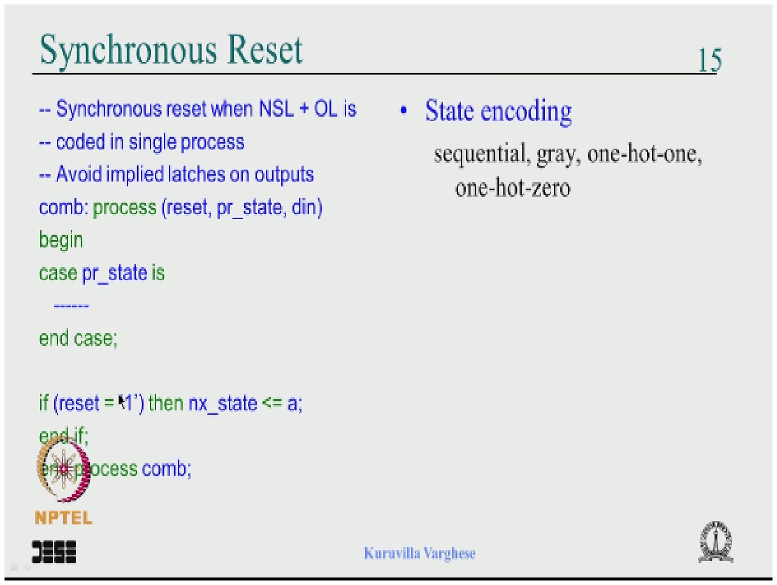
And though ultimately it will all be converted to standard logic vector and so on **.** It is difficult at least to do with select so, in know because you are specifying both mutually exclusive condition. So, there will not be any priority and this will result in a simple logic. Then comes the synchronous reset along with when you write the next state logic and the output logic together.

If it is alone next state logic alone there would be problem. When you combine you can make a mistake in the sense that you say if reset is 1 then you say next state is gets a. Because that is what you want and you forget all about the outputs. You say else and case present state is and you say when s0 you write the outputs. There could be 10 outputs and all the transition and so on okay.

But mind you there is as for as if is concern, there is 1 choice here and kind of mutually exclusive choice here underneath the case come there the output is specify. And here it is not specified so, you get a implied latch for the reset is equal to 1 it is dangerous. So, we have to say output is some value so, do not care is one kind of because this say when the reset is getting kind of asserted for that duration.

So, you can even make it inactive you do not have to say do not care you can make it the inactive value to be very sure that during when reset is 1. You do not get very nasty surprises but it is improbable so that is one way of shorting this out another way of shorting it out is that write the if just for this and you say end if okay.

(Refer Slide Time: 16:28)



Synchronous Reset

15

```
-- Synchronous reset when NSL + OL is
-- coded in single process
-- Avoid implied latches on outputs
comb: process (reset, pr_state, din)
begin
case pr_state is
  ------
end case;

if (reset = '1') then nx_state <= a;
end if;
end process comb;
```

- State encoding

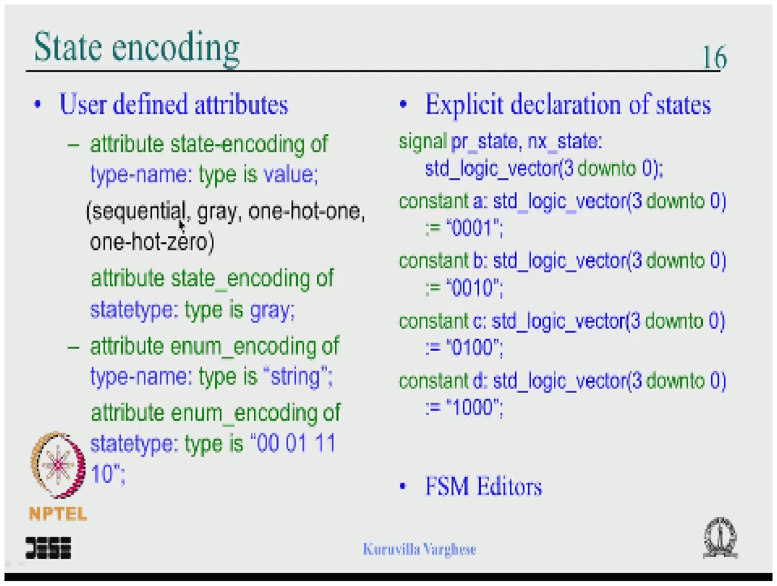
  sequential, gray, one-hot-one,
  one-hot-zero

Kuruvilla Varghese

And write the case separate okay separate it out so, if reset is 1 then next state is a end if. Because we are not combining there is no else here. So, like that you do I am do the case present state is as I said I do not prefer this. Because there is this looks as if there is no else kind of thing. So, it could be problematic so, do not write like that it better to write like this and we also said normally if you do not specify anything.

Like you just leave the code like this and then the tools will assign a simple kind of encoding mostly sequential encoding like this should be 0, 1, 2 and 3 okay. And if you want to kind of whatever reason and we are going to see what reason you could there could be to change this assignment. But then if you want it you can use the kind of predefine attributes of VHDL to control that **that** is called state encoding sequential grey one hot one, one hot zero.

(Refer Slide Time: 17:40)



State encoding

16

- User defined attributes
  - attribute state-encoding of type-name: type is value;

    (sequential, gray, one-hot-one, one-hot-zero)

    attribute state_encoding of statetype: type is gray;
  - attribute enum_encoding of type-name: type is "string";

    attribute enum_encoding of statetype: type is "00 01 11 10";
- Explicit declaration of states

```
signal pr_state, nx_state:
   std_logic_vector(3 downto 0);
constant a: std_logic_vector(3 downto 0)
   := "0001";
constant b: std_logic_vector(3 downto 0)
   := "0010";
constant c: std_logic_vector(3 downto 0)
   := "0100";
constant d: std_logic_vector(3 downto 0)
   := "1000";
```

- FSM Editors

NPTEL

Kuruvilla Varghese

And this I am showing what was supported in integrated you know system environment of the Xilinx ISE or IS okay but I have seen that attribute in the recent tool like the Vivodo of the Xilinx this attribute is change. So, please check which tool you are using and refer to vendor manual for the appropriate attribute. But more or less the may be some name changes.

But more or less the syntax is the same so, here the syntax is the same attributes state encoding of whatever state you have given the name for that numerated data type. So, type is grey are sequential one hot one, one hot zero, and we will say what is one hot one yesterday said what is it and then you get encoding define by this. But if you want a very specific encoding you can use attribute enum encoding of state type.

Then you literally say what is encoding okay and if that is not possible you could instead of you know you can go the kind of numerated data type. And you can kind of hot code the present sate and next state into vectors standard logic vectors. And you can define constant so, that you can you do not need to change, like if there is a change in the state diagram, then if you write numerical values in VHDL code.

You have to go and change everywhere it is error prone so, you can declare define constant for each state with the proper state assignment whatever is the assignment, then you can implement that code then if there is a change need to be done it need to be changed only in one place. You can add more state you can change the state assignment you can remove the state and so on in one place okay, definitely the code also if you add.

And delete you need to change in VHDL code but minor changes in state assignment you do not need change anything and I also said about a FSM editors. There are graphical tools which allows you to draw the state diagram, and which will create the equivalent code VHDL or very log there is no magic and a because it is 1 to 1.

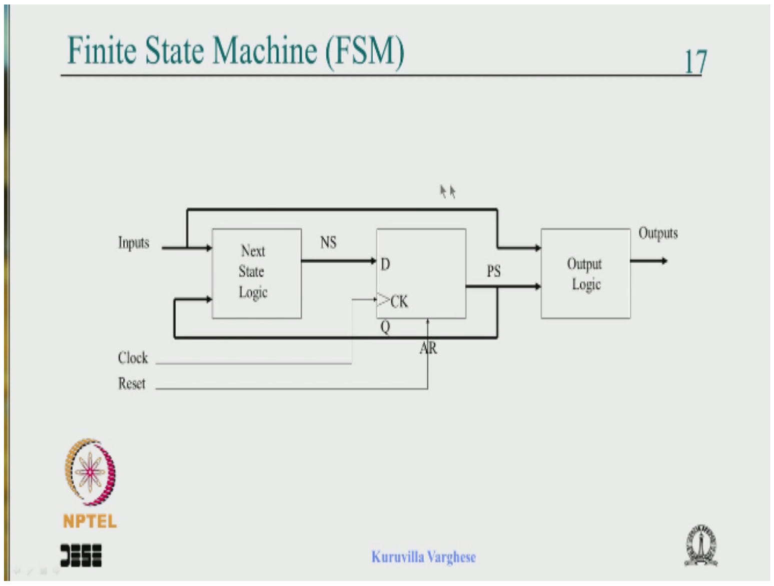
And it as you draw as you put various elements it know the moment you put a bubble it know this as a state and arrow is a transition. So, that is kept right and ultimately in some data structure and ultimately some template codes are generated. It is not very difficult as I said if you are I mean it can be done very easily only you need to be good in programming. So, that is what we have you know discuss in the last lecture.

I have gone through this, so that again that gets into your mind properly. Now we will continue with we have actually in the earlier lectures we have looked at certain issues with a finite state machine. If you remember we have looked at how to do the state diagram optimisation okay, what is the basic idea what are the equivalent state and which are redundant, how to detect that from the next state logic and output logic and so on.

And we have also looked at the output races when there is a kind of multiple flip-flops state change state. At the same time owing to difference in propagation delay, there could be transitory state and if these states produce output that could create problem, and the circuit. So, how to handle that and we have looked at the gray coding output registering and we mentioned about VHDL coding for that.

Then we looked the transition table, if we choose the various type of flip-flop. Basically D type, T type and JK type of flip-flops and we have kind of discussed what are maybe the advantage, disadvantage and so on. So, let us continue with the state assignment okay. So, let us take that part today.

(Refer Slide Time: 22:20)


Finite State Machine (FSM)

So, remember that you have this state machine where the next state logic output logic is there. And we have the state sorry about this, this should come here this q. So, we say the state machine is going through different states and when we design we say state a, state b, state c, state d. But before proceeding with the next state table and output table we have assign some numerical value to it okay.

Now we are going to see whether that affect the question is that what is the effect of state assignment on the next state logic okay, say or output logic say the question is that if you do the assignment in a particular way can the area required for implementation of this can be minimal okay. Suppose there are 20 ways of doing the state assignment in at least in one of them or multiple of them.

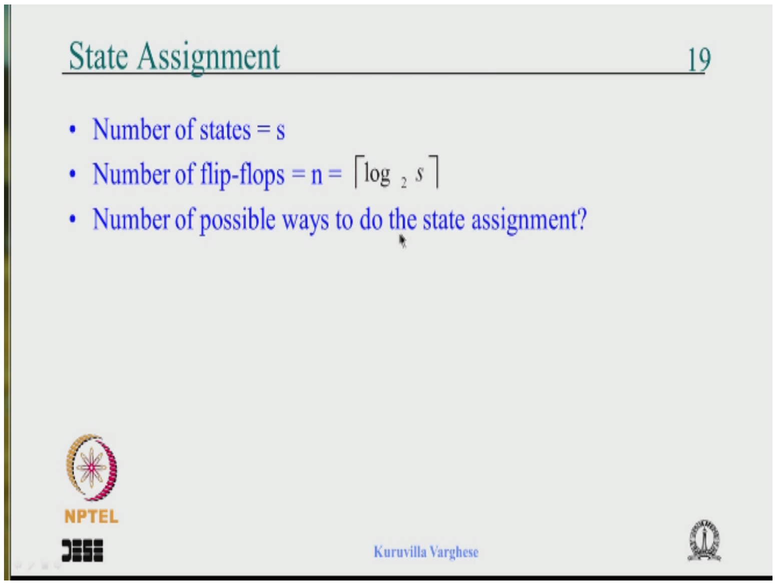
The area is half of the maximum okay, then it is better to choose that particular state assignment okay. And that is natural because at least for the next state logic, the next state is a function of the present state and input. And this is a combinational logic and we so, you imagine the next state table you have inputs, you have the present state, you have the next state and we are trying to form equations of various d2, d1, d0 in terms of the Q2, Q1, Q0 and inputs.

So, you and you can imagine if you are familiar with the (()) (24:14) you try to group them in terms. So, our question is that if you do the assignment in a particular way you can group it such that there are redundancy can be removed and this become minimal. Similarly here, because output is the decode of the present state and the input and if you do the assignment in such a way

Then whether this logic can be minimise or there is an assignment such that both can be minimise, there is a very great saving in area. So, if the area is reduce many a times interconnecting wire length can be reduce, the delay will be less, the power dissipation will be less. So, in the digital VLSI in FPGA and all that it is very wise to kind of optimise area, minimise area.

That will result in lot of you know advantage with respect to the power with respect to the timing and so on. So, that is what we are going to look, so at this point it looks like say you work out all possible assignment and find out the area which is minimal okay. That is what we are going to look it whether that is a possible thing.

(Refer Slide Time: 25:30)



State Assignment

19

- Number of states = s
- Number of flip-flops = n = $\lceil \log_2 s \rceil$
- Number of possible ways to do the state assignment?

Kuruvilla Varghese

So, that is the idea of the state assignment. So, take this case the number of states are S okay. Suppose we have 5 states, then the number of flip-flops for the binary encoding is the logarithm of S to the base 2 and normally you take the sealing. Because this could be a fraction say in the case of 5 if you say log 2 it will be some 2 point something some decimal number. So, we need to use take the sealing which is 3.

Then we will end up using 3 flip-flops okay. So, now the question come is that we are trying to kind of do all possible assignment that is what we have decided okay. Let us do all possible assignment. So, what are the number of possible ways to do the assignment say we have S states, we have n flip-flops okay, let us assign a variable to it. Instead of putting this in the expression log is to the base 2 is n, the sealing of it, that is n.

So, how many possible ways you can do the state assignment in terms of n and S . First thing to kind of decide is that whether in all these kind of counting problems you should be very clear whether it is a permutation or a combination**,** so every time if you have confusion like for a like for those are familiar it is 1 shot okay, if is it struggle with it the best thing to take an examples.

Take an example of suppose you have say for states, possible states like 0, 1, 2, 3 you have 2 sorry 4 possible states and like 2 flip-flops for possible state and 2 real states s0 and s1. So, then you know that first state s0 can get any of the 4. So, there are 4 possibilities okay and if you assign for possibility one of the 4 possibilities. Then when it comes to the next one you have only 3 choices okay.

So, it is a 4 into 3, so it is a permutation. Because each assignment is unique it is not a combination like s0 getting 0 and s1 getting 1 is different from s0 getting 1 and s0 getting 0. So, that alone should clarify the position, so basically if you look at the number of possible ways to do the state assignment is permutation of 2 rise to n, because for n flip-flop there are 2 rise to n kind of possible states.

And S is the real number of state, so p 2 raise to n , s okay and let us take an example there is a state machine with 17 states not a I am kicks the real life case. There could be state machine with 17 states. So, the n is 5 okay, you will accuse me of choosing something close to the 16. But in real life it can be strain and let us find out the possible ways of doing the state assignment.

The permutation of 2 rise to n, so that is 2 rise to n is 32, 2 rise to 5 is 32 state is 17. So, that is 32 factorial divided by 32-17 factorial so, you have some like p n or then it is n factorial by n-r factorial. So, 32 factorial by 32-17, so it is essentially 32 into 31 into all the way into 16 okay.

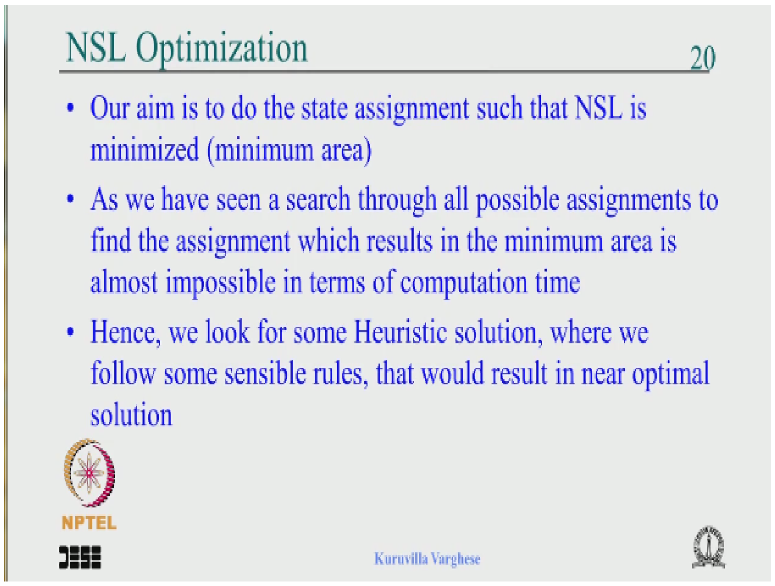
And if you look at this number it is some 2.5, some decimal number into 10 rise to 44, so huge possibilities you know.

This should that is why I put this you know. So, if you try to do all kinds of all possible assignment and try to minimise area. It is an intractable problem it is not going to be solved in a kind of (()) (30:08) time period it is a for advantage we get amount of computation. We have to go through is tremendous this is not possible like this is mind boggling so, this huge number.

So, this should tell you that you should not try to attempt such a thing okay of course in a simple case may be there are 4 possible state and 3 assignment. You have to make it is possible, but then when the numbers are even a it looks very small you end up with this. Because of this kind of you know the exponential way it grows so, the question is that we cannot very systematically go and solve it.

But our aim is that somehow minimise the next state by doing a state assignment okay or by doing a state assignment minimise the output logic.

(Refer Slide Time: 31:11)



NSL Optimization 20

- Our aim is to do the state assignment such that NSL is minimized (minimum area)
- As we have seen a search through all possible assignments to find the assignment which results in the minimum area is almost impossible in terms of computation time
- Hence, we look for some Heuristic solution, where we follow some sensible rules, that would result in near optimal solution

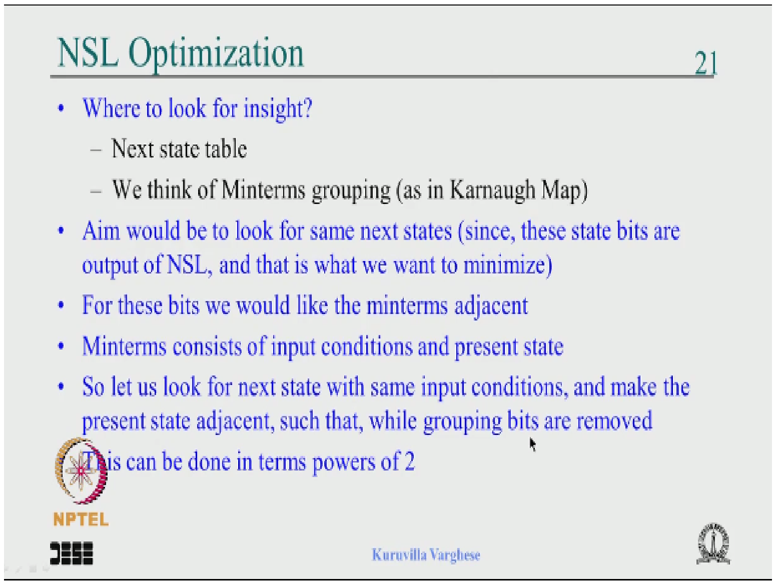
NPTEL

Kuruvilla Varghese

So, trying out possible kind of assignment and finding out the minimal is not computationally viable solution. So, then in such cases, when things you cannot go very kind of go through all the cases and find you know optimal solution. We try some kind of sensible kind of rules which is called heuristic that we know that if you this rule make a some sense that like if you from there is a basic principle which tells you that.

If you do this probably there is could be a reduction and in area and we do that. We may not get the absolute minimum but we can be close to the minimum or we can do better than a random kind of assignment or a sequential assignment. So, we call that heuristic so, since we have this systematic method does not give any solution let us try some heuristic. So, that for we are going to do it and we will concentrate on the next state logic minimisation.

(Refer Slide Time: 32:25)



NSL Optimization

21

- Where to look for insight?
  - Next state table
  - We think of Minterms grouping (as in Karnaugh Map)
- Aim would be to look for same next states (since, these state bits are output of NSL, and that is what we want to minimize)
- For these bits we would like the minterms adjacent
- Minterms consists of input conditions and present state
- So let us look for next state with same input conditions, and make the present state adjacent, such that, while grouping bits are removed
- This can be done in terms powers of 2

NPTEL

Kuruvilla Varghese

So, the question is that where to look for insight okay, we want to come out with the some sensible rule to apply for the next state logic optimisation where do we look for insight okay naturally the next state logic is specified in the next state table. So, that is where we should look okay, we should look and the next state table and now the next state table you know it is composed of input present state sorry present state and next sate.

And next sate is are for a d flip-flop some d2, d1, d0 and if you remember you are Karnaugh map for d2 we form midterms in terms of the input and the present state. And then you put it in the Karnaugh map and try to group them okay, you can group them only their adjacent okay so, that is what that is that tells some rule saying that if you play with the logical adjacency of the states that might give you and advantage in while grouping.

You know while grouping and applying basically applying the ad logical adjacency theorem if you remember the adjacency theorem say that ab bar or ab is nothing but a into b or bar which is nothing but a okay. That is we do in logical adjacency and that is I am talking about one variable that can be applied to any power of 2 like you can apply to 2 variables. Then the 4 states are there it can be possible assignment.

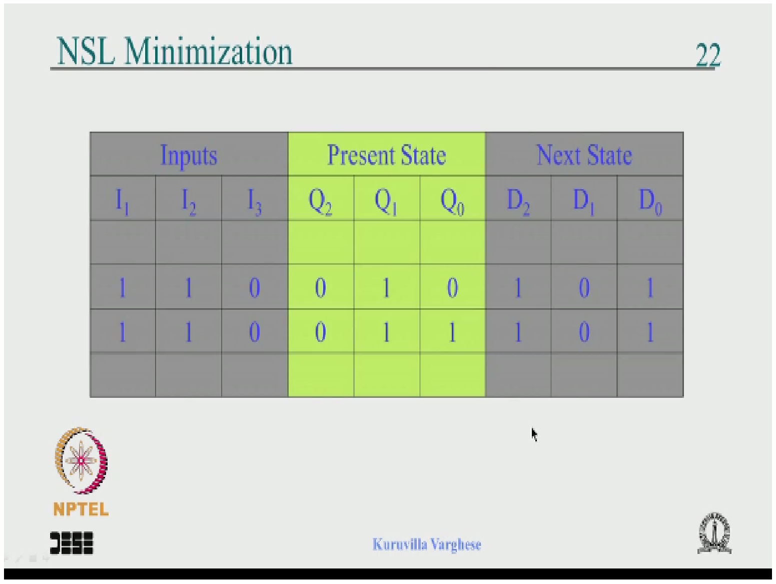
Then it can be combine to remove the tool literals in a midterms so, that is I mean so, aim would be to look for same next states since, these state bits are output of NSL, and that is what we want to minimize. So, we would like you look for the same next state you know we are in a next state table. We have input present sate and the next state, and we are trying to optimise the implementation of the next state. So, look for the same next state bits.

Suppose you are looking at d2 look d2 where ever d2 is 1 you know are the d2, d1 d sorry q2, q1, q0 where ever there is next state you know kind of identical then you can make things adjacent and for this we would like the midterms adjacent, midterms consisting of input condition and the present state okay. So, the best bet will be that we are playing we have only choice in the state assignment, present state.

So, we are playing with symbolic state and we are assigning it input conditions we have no control okay. So, it will be good to assume that if for the same next state if there are same input condition. You make the present state logically adjacent, then when you group it some literals will be thrown of from midterms. So, we let us look at the same next state with same input condition and make the present state adjacent said that.

While grouping bits are removed okay that is the basic idea and thi scan be done in a powers of 2 okay.

(Refer Slide Time: 35:57)



NSL Minimization

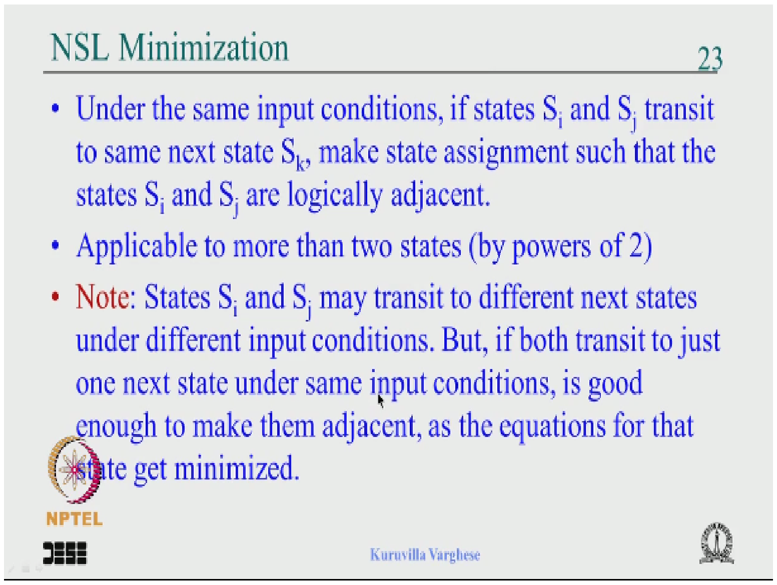
Inputs			Present State			Next State		
I_1	I_2	I_3	Q_2	Q_1	Q_0	D_2	D_1	D_0
1	1	0	0	1	0	1	0	1
1	1	0	0	1	1	1	0	1

Kuruvilla Varghese

So, that is what I am going to show suppose you have 2 state and I have already run the assignment but say this is Si, this is sj and it is transiting to the same Sk 101. So, the rule is that for some 2 symbolic state for the same input condition, the transit to the same next state what you do is that you make these 2 states Si and Sj logically adjacent okay. So, this is 010011 it is made adjacent then when you like it group together in.

When you apply the adjacency theorem you see that everything else is same. But this is different so, this can be knocked off okay. So, this can be extended to powers of 2. You have a 101 here, another 101 here all the input conditions are same, and hopefully this is like you know 0 all 00001 then you can combine all the 4 and q1 and q0 is removed from the equation of d2 and d, you know d0 and all that. You know that is the idea of heuristic.

(Refer Slide Time: 37:14)



NSL Minimization

23

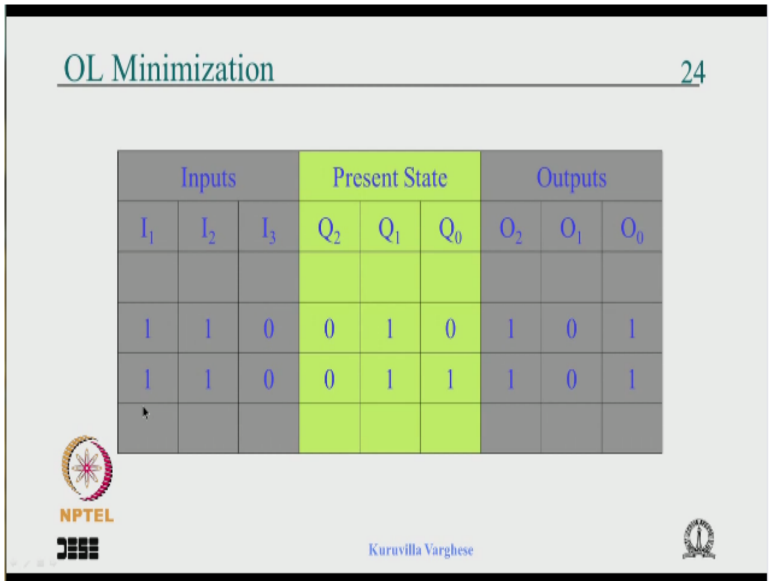
- Under the same input conditions, if states S_i and S_j transit to same next state S_k, make state assignment such that the states S_i and S_j are logically adjacent.
- Applicable to more than two states (by powers of 2)
- Note: States S_i and S_j may transit to different next states under different input conditions. But, if both transit to just one next state under same input conditions, is good enough to make them adjacent, as the equations for that state get minimized.

Kuruvilla Varghese

We can apply to the next state logic minimization so, under the same input condition if the state Si and Sj transit to same next state Sk make state assignment such that the states Si and sj are logically adjacent. And this is applicable to more than two states okay, and you can say that you know it i seven if you can catch on 1 bit it is like you say here you look that even if in 1 bit suppose this is you cannot get a solution across the bit even for this d2 alone If you make it adjacent **.**

That is useful because at least for that there will be a reduction in the next state logic for the d2 okay. That is what I have written here that states Si and Sj may transit to different next state at different input condition. But if both transit to just one next state it is good enough to make them you know adjacent okay. So, like if you can Si and Sj may transit to different kind of next state at different condition. But Sk, sl and all that but even for1 Sk if we can make it adjacent it is useful.

(Refer Slide Time: 38:25)



OL Minimization

24

Inputs			Present State			Outputs		
I_1	I_2	I_3	Q_2	Q_1	Q_0	O_2	O_1	O_0
1	1	0	0	1	0	1	0	1
1	1	0	0	1	1	1	0	1

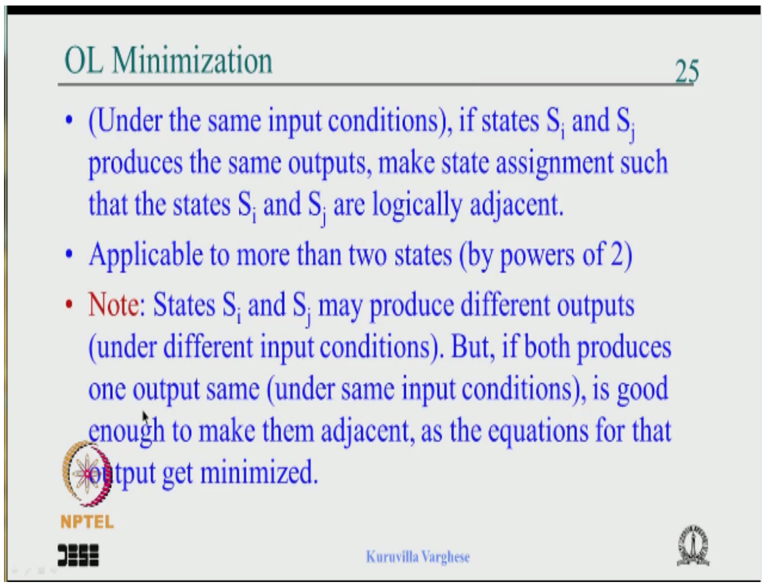
NPTEL

Kuruvilla Varghese

So, now you get like the same heuristic can be applied to the output logic. So, we are almost kind of you can straight away the rule the heuristic that you have 2 present state say let us take the Moore kind of output. Then the outputs are same, then you just make them adjacent like you have Si and Sj produces the same output make them logically adjacent. So, that when we group one thing kept knocked off.

And similarly you have 4 states adjacent then you make all the 4 logically adjacent saying that 111, 1011, 00, 01 so that 2 literals get knocked off the midterm. And if it is a mealy kind of output then you have to combine the inputs you have to make sure that are logically adjacent for the same kind of input condition that you have to keep in mind.

(Refer Slide Time: 39:29)



OL Minimization

25

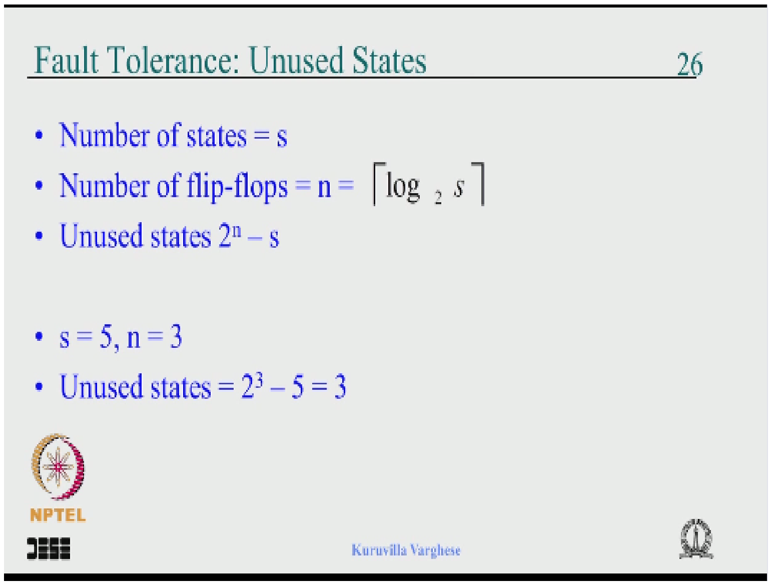
- (Under the same input conditions), if states S_i and S_j produces the same outputs, make state assignment such that the states S_i and S_j are logically adjacent.
- Applicable to more than two states (by powers of 2)
- Note: States S_i and S_j may produce different outputs (under different input conditions). But, if both produces one output same (under same input conditions), is good enough to make them adjacent, as the equations for that output get minimized.

So, that is what is stated here you know under the same input condition that is for mealy type. If it is not mealy you can remove this, if state Si and Sj produces same output make the state assignment such that the state Si and Sj are logically adjacent okay. And it is applicable to more than 2 states by powers of 2, there is no point in making I mean you cannot make 3 logically adjacent you have to make say after 2**.**

The thing we have to try is the 4, 8 and so on. And state Si and Sj may produce different output and different input condition. But if both produces even 1 output saying is good enough to make them adjacent as equation for that output get minimise, what I am saying is that you may not get a chance. Suppose you do not get a chance to make kind of all like such a pattern for 2 states you can look at the subset of these outputs to do that.

I do not know whether the tools do it sometime in a simple state machine may not be worth to try that. I am not sure whether which synthesis tool try maybe at least the maybe some advance synthesis tool like simplify pro may try it I do not know about this reason the Xilinx excess t synthesis tool are the synthesis tool of the vivado, I am not tried it**.** If I tried it I will maybe in the later lectures I will tell you.

(Refer Slide Time: 41:09)



Fault Tolerance: Unused States

- Number of states = s
- Number of flip-flops = n = $\lceil \log_2 s \rceil$
- Unused states $2^n - s$

- s = 5, n = 3
- Unused states = $2^3 - 5 = 3$

Kuruvilla Varghese

Now let us come to this kind of issue so, what we have discuss is the state assignment to reduce the area and we are found that for in are to evaluate all possible scenarios state assignment. And come out with the minimal area is an intractable problem we have seen the huge computation type required. So, then we have come out with some heuristic which is based on some sensible kind of basics you know basics of minimisation.

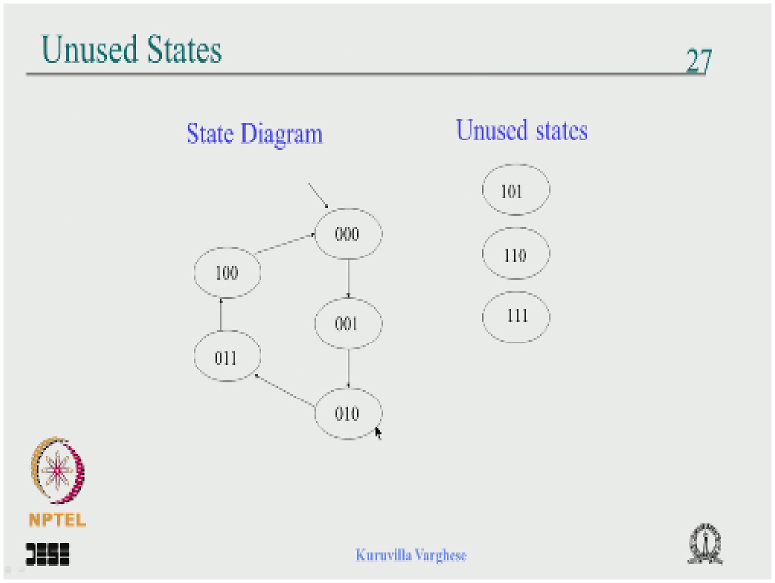
So, it should work okay only thing is that to try and do the next state logic minimisation and output logic minimisation together maybe contracting like that. You try out the heuristic for next state logic minimisation and you come back and write to do again for output logic both may not work together. You know that is 1 problem with that heuristic if you are luck it works together so, at least you can try one thing.

If you fine the next state area is much you know larger than the output logic area, then you can try the heuristic for the next state logic than the output logic. But we are not sure whether you can do it together it could be contracting the heuristic could be contracting it may not reap the benefits as you think so, let us come to next issue which is something to do with the fault tolerance of the state machine.

So, this is about the unused state suppose we have the number of states as s okay. Then we know the number of flip-flops is n which is log s to the base 2 the sealing of that okay so, there could be a unused state. Because this the fact that you end up with the fraction and take sealing says that there could be a unused state okay like here. So, the unused state the all possible assignments the states are 2 raise to n, because n flip-flops are there.

So, -s is the unused states if you have states number of states are 8 then n is a not a fraction if it takes just the log. Then n is 3 so, in that case 2 raise of 3 –a 8 is 0. But i fit is sis 5. So, that is the case here n is 3, then the unused states are2 raise to 3 – 5 equal to 3 okay. So, the question is that what to do with that okay.

(Refer Slide Time: 44:02)



See here I am showing an example of 5 state this 5 states so, we have 3 unused states and assume that we have done a state assignment like a sequential state assignment. So, very simple state diagram and mind you this could be like in most cases there could be simple state diagram like that only thing is that. There could be kind of self looks here there could be the only a difference from a practical state diagram.

So, let us I am not showing all the input transition but assume something like this and you have stage going through 0, 1, 2, 3 and 4. But 5 I am not use say 5, 6, 7 is not use okay and assume that we have not taken we have not specified this okay. We have kind of told the tool to 3 kind of do not care or something like that and what happens the state machine is after all some 3 flip-flops okay.

And suppose the circuit FPGA or the chip we are designing is in an noisy environment okay. It can be some kind of radiated EMI electromagnetic interferons or conducted EMI you know that the noise conduct through the supply lines through the ground and there was some noisy equipment connected to the same power supply and some noise gets picked up either radiated or conducted.

And this suppose this flip-flop was in say assume that it is in 001 okay. So, Q2 is 0, Q1 is 0, Q0 is 1. Suppose the Q2 has flip from 0 to 1 oaky it was in this state and suddenly it is flipped this Q2 as flipped from 0 to 1. Then the state machine gets into this state 101 oaky. Now that then what happens the question is what happens okay. Suppose we have not taken care, we have not explicitly taken care of that in the design what can happen okay.

So, that what are all happen I mean that is what the first thing let us forget about all the technical reasoning what are all can happen use you just as a kind of a creative or what are possibilities are there like you can say S gets in there oaky 101. And in the next clock it comes back here fine then it is very good it is gone there and it is come back and it continues okay or you can say instead of coming back here, it comes back here**.**

That could be dangerous depending on the application, because you are doing something very sequentially here. Suddenly it is keep this 2 steps and go there it can be very dangerous depending on the application know. You can imagine this is controlling suppose the state machine is controlling radiation of a patient and this is you know happens something initialisation happening, this is the radiation state.

And with improper initialisation goes there what was 5 minutes is becoming 50 minutes then it is too dangerous you know. So, depends on the application maybe in a simple thing like a vending machine you ask for the for a soft drink and a potato chips come out it may not matter good for customer little large for the vendor. But may not nothing like threatening happens.

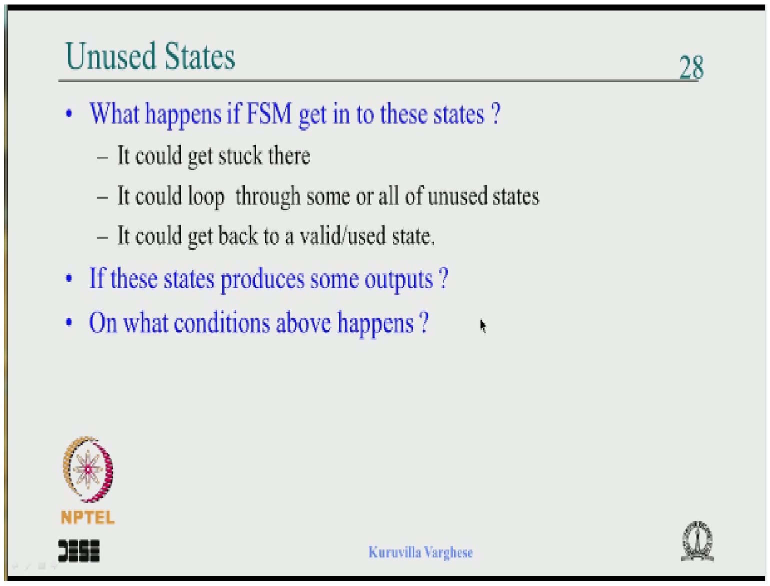
But then it depends on application, so one possibility is that it can go there and gets come back somewhere here. And at least for us a probably it looks safe than you know getting it stuck okay. That is another possibility you know you get it you by now is a it is goes there and it stays there like again the clock comes. But somehow it stays there you know. We will see what are the chances of it staying there.

But suppose it stays there, then it is bad okay. So, if nothing happens it is okay. But if something happens with the data path then it is dangerous okay. Another possibility is that it may loop through some or all of the unused state okay, maybe like after 10 it gets inot 101. It comes 111, comes to 101 and loop are loop through this you know whatever way okay. It cycle through some or all of the unused state.

Now assume that this unused state produce some output okay. Some random output we had some random output. We had some say 5 output orbitally produces some output. Then it can be very extremely dangerous okay. So, that is the issue we are looking at it, the fault tolerance, if there are new state in the state diagram of state machine and if you not design properly and by noise or by some chance it gets into the unused state.

It can come back to obituary state in state diagram. It can gets stuck there, it will loop through and if that states produce random output if you are not taken care, then it can be extremely dangerous.

(Refer Slide Time: 49:57)



Unused States

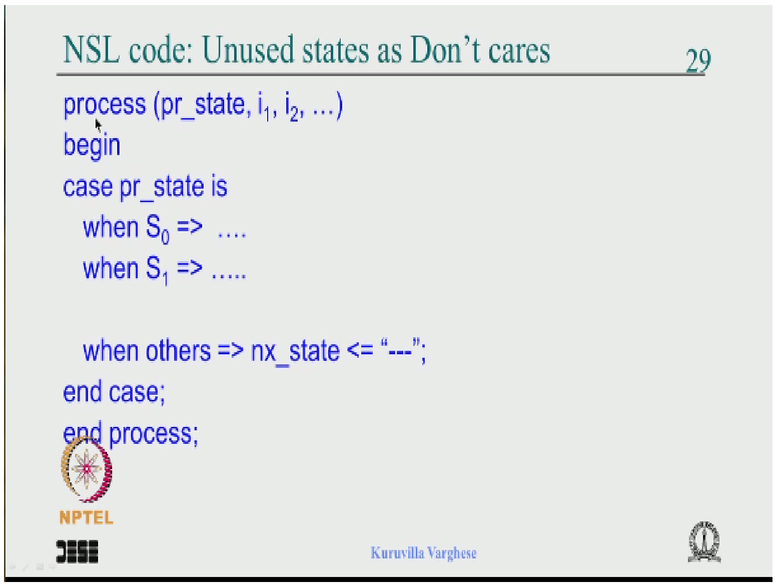
- What happens if FSM get in to these states ?
  - It could get stuck there
  - It could loop through some or all of unused states
  - It could get back to a valid/used state.
- If these states produces some outputs ?
- On what conditions above happens ?

Kuruvilla Varghese

So, that is what is stated here what happens if FSM get in to these state get stuck there, it could loop through some or all of unused state. It could get back to a valid or used state okay, it get back to some of the state in the state diagram. And if that state produce some random outputs, then dangerous. So, on what conditions above can happen okay. So, we would like to look at in which scenario kind of it get stuck there.

I mean if you can understand that maybe it will at least it is better than you know thinking like you have such a problem and somehow bring it back to some kind of reasonable state. But how like in what conditions goes there and get stuck on what condition it get loop through some of the unused state. Let us at least for an academic curiosity let us look at it from the basic principle okay.

(Refer Slide Time: 51:00)



NSL code: Unused states as Don't cares

29

```
process (pr_state, i1, i2, ...)
begin
case pr_state is
  when S0 =>  ....
  when S1 => .....

  when others => nx_state <= "---";
end case;
end process;
```

NPTEL

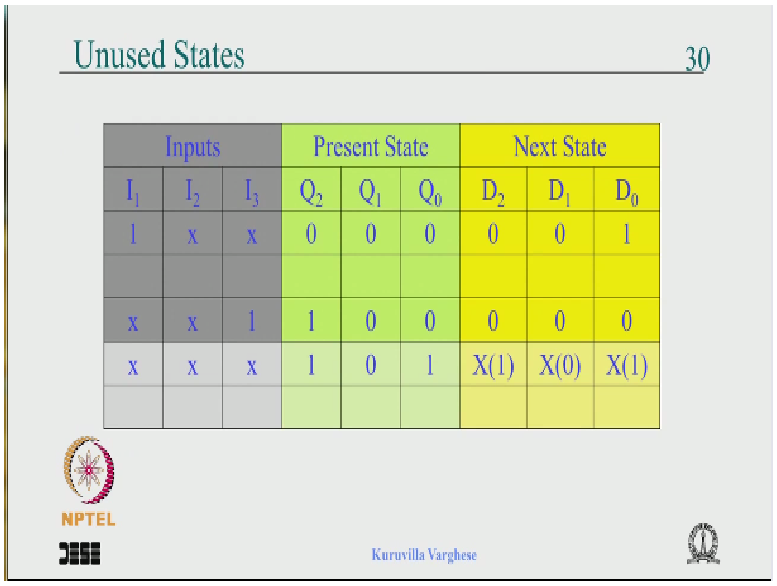
Kuruvilla Varghese

Now for that happen assume that we have done the next state logic coding here like this you have process, present state and various input i1, i2, like that and you say begin and you say case present state is and you say when S0, S1 we have 5 states. So, you go all the way to when S4. But there is a S5 and S6 and S7. And we do not specify that is a, we say when others okay combining all S5, S6, S7 and all possible kind of states for the simulation you now, using the standard logic other cases.

Then you say next state is do not care, we are hoping that if you say do not care the next state area is minimise you know because we are not explicitly specifying. That means for these S4, S5, S6 you can treat the next state is 1 or 0. And we are hoping that some minimisation happen with the proper assignment and the area get minimise. That is why we say next state is do not care.

And that also shows the use of that particular do not care. Because when we discuss standard u logic and standard logic which are there is something called do not care. And we have not seen it being use. But this is way it is used and we are hoping that it gets reduce area.

(Refer Slide Time: 52:29)



Unused States

30

Inputs			Present State			Next State		
I_1	I_2	I_3	Q_2	Q_1	Q_0	D_2	D_1	D_0
1	x	x	0	0	0	0	0	1
x	x	1	1	0	0	0	0	0
x	x	x	1	0	1	X(1)	X(0)	X(1)

NPTEL

Kuruvilla Varghese

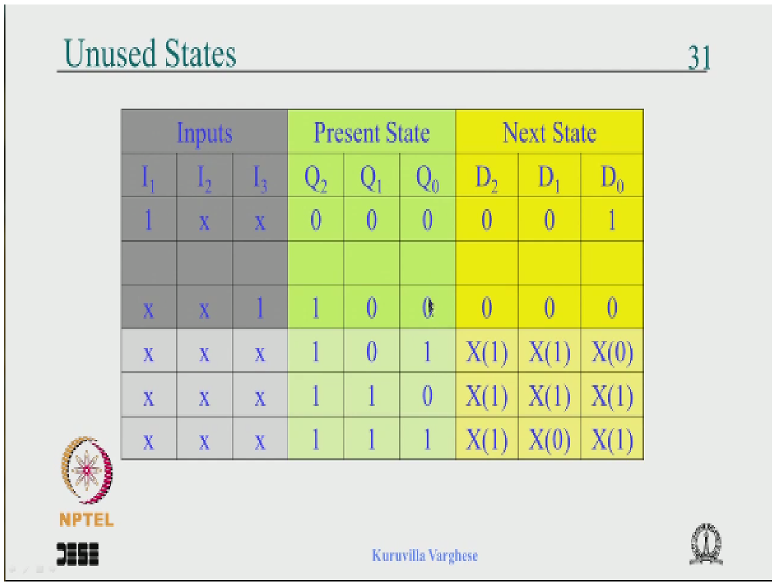
So, let us look at 1 condition okay, 1 possible assignment the tool makes okay. So, there you see this is a valid state 000 then 001 and there are lot of entries up to this ellipsis continuity like 100 okay which is kind of S0 to S4 five valid states. And suppose this is an kind of unused state which is 101 I am showing with the different colour and we say the input condition is do not care okay we do not care.

And we said the next state is also the do not care okay. Assume the synthesis tool for optimisation purposes treated this as 1, this as 0, this as 1 okay. That means if such a thing happen by we have written the code like that and this is the state this is the kind of assignment the tool does for optimisation. Then you see what happen by mistake the present state become 101.

And the next state decoded is 101 itself from the current present state the respective of the input, the next state decode is 101 itself. So, the moment if gets in there the clock transit to the same 101. Then it loops through that and we are not even talked about the output. We do not know, because if you in the output table if you write 101 and say do not care, then the same thing will happen the tool will pick assign some 1, some 0s and all that.

And then it will produce some random output which can be extremely dangerous to the application okay. So, that is how this getting stuck happens. So, now you can imagine how the loop through can happen oaky. That is what is shown in the next slide.

(Refer Slide Time: 54:39)



Unused States

31

Inputs			Present State			Next State		
I_1	I_2	I_3	Q_2	Q_1	Q_0	D_2	D_1	D_0
1	x	x	0	0	0	0	0	1
x	x	1	1	0	0	0	0	0
x	x	x	1	0	1	X(1)	X(1)	X(0)
x	x	x	1	1	0	X(1)	X(1)	X(1)
x	x	x	1	1	1	X(1)	X(0)	X(1)

NPTEL

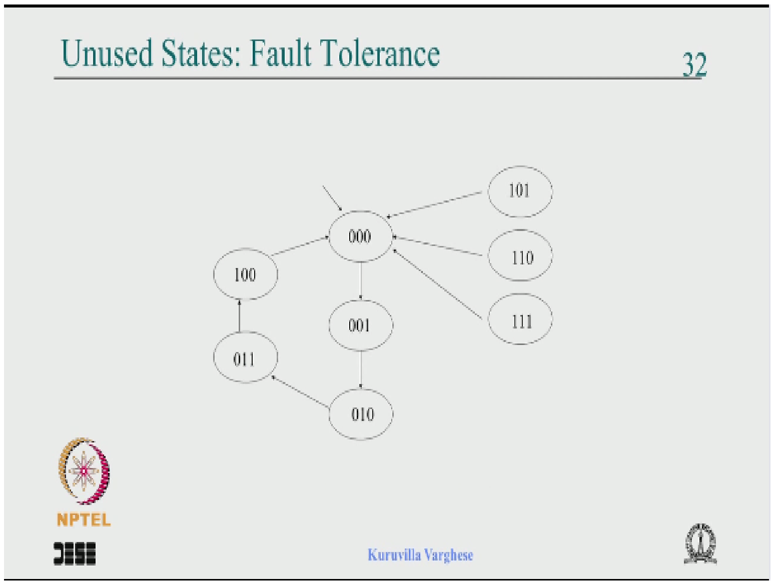
Kuruvilla Varghese

So, we have the state 0 to 4 S0 to S4 and we write all the valid kind of state diagram. So, everything happens. But here for S5, S6 and S7 we said it is all do not care oaky, that is what we have said. Now if the tool do this kind of assignment for whatever reason. For optimisation say 110, 111 and 101 then what happens you see that if it gets into 101 the respective of the input condition, the next state is 110.

If it get into 110, then the next state is 111 and if it is 111 it is 101. So, that is back here, so it loops 101, 110, 111, 101 okay. Now in this case all the 3 it can loop through the part state suppose here it was 110 and in the 110 comes here it is back to kind of like 101, become 110 and for 110 it is 101 then it loops through the 2 **2** states okay. So, this quite dangerous.

And as I said we are not sure what outputs are produce by this then again the do not cares and all apply in the output logic table.

(Refer Slide Time: 55:59)



So, it is wise as for as the fault tolerance is concern to bring it back okay. Now when you say bring it back we have to bring it back to some state some valid state okay, definitely valid state. But in this picture I am showing it to bring it back to the initial state oaky. But we can state that bring it back to a safe state okay. So, that is something which we have to keep in mind.

We have to bring it back to some safe state. And in most cases maybe the starting state is good enough. Suppose it take an example of a multiplication, then suppose something goes wrong half a through the multiplication. Then it is gone to this unused state**.** Nothing happens is the best thing is to bring to the ready for the next multiplication maybe this result is not corrupted.

But at least the next one happens properly. But that need not be true for all kind of cases like you have a **a** state machine you know kind controlling something complex and which takes long time and it is gone through half a and you just bring it to the original state may not be a good idea. So, maybe in the next lecture we will see how to code for this and discuss kind of the scenario little more bring little more priority to it.

So, the last thing we have discussed is how to handle the unused state. Because if the state machine gets to unused state depending on how we have coded we can get stuck there or loop through some unused state or it can come back to obituary state which could be dangerous. So, you should bring it back to some kind of safe state at least for the time being let us kind of decide on that and then.

We will see may be more little more detail about that with regard some applications and we will continue in the next lecture with the other issues of the state machine. We will try to complete that so, please go back revise try to grasp these concept from the basics like I have shown some minimisation if you are not kind of forgot the minimisation please go back and look at it.

So, the though the basics are simple some time knowing the simple thing connecting it can kind of you will get lot of benefit out of that, than getting stuck with some kind of high level language and try to write some if then and case with rhyme with no particular grasp and understanding. So, that is what I want to say so, I wish you all the best and thank you.