Advanced VLSI Design

Prof. Virendra K Singh
Department of Electrical Engineering
Indian Institute of Technology – Bombay

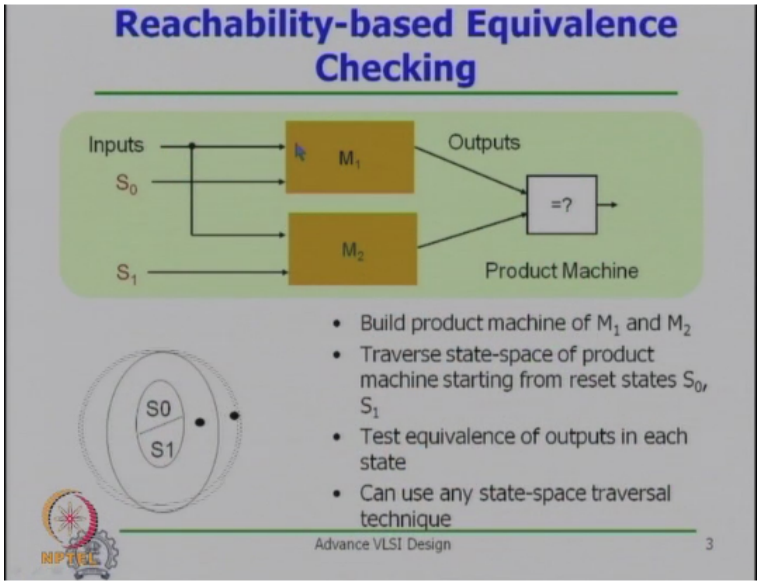
Lecture - 41
VLSI Design Verification: Equivalence/Model Checking

Welcome to the lecture series on advance VLSI design course. In last couple of lectures, we were talking about VLSI design verification. So, now I will take you through remain portion of the VLSI design verification. So, in the last class we discussed about equivalent checking. So, we discussed combinational equivalence checking and we also discussed how we can formulate a sequential equivalence checking problem as a combinational equivalence checking by unrolling sequential circuit.

And we can verify the unrolled combinational circuit vis a vis the other design. So, one of the problems that we discussed was the circuit size increases. The number of input variable increases so, then that is the problem. And other solution that we discussed was we have two finite state machines and then if we can reduce these two finite statement machine and we take the isomorphism of these two reduced finite statement machine.

We can say that both of the machines are equivalent. So, these two techniques we already discussed now I will discuss the third technique which is more common and more popular. That is based on the reachability analysis. So, what do we do is that we have two machines.

(Refer Slide Time: 01:51)



Reachability-based Equivalence Checking

Inputs
S_0
S_1
M_1
M_2
Outputs
=?
Product Machine
S0
S1

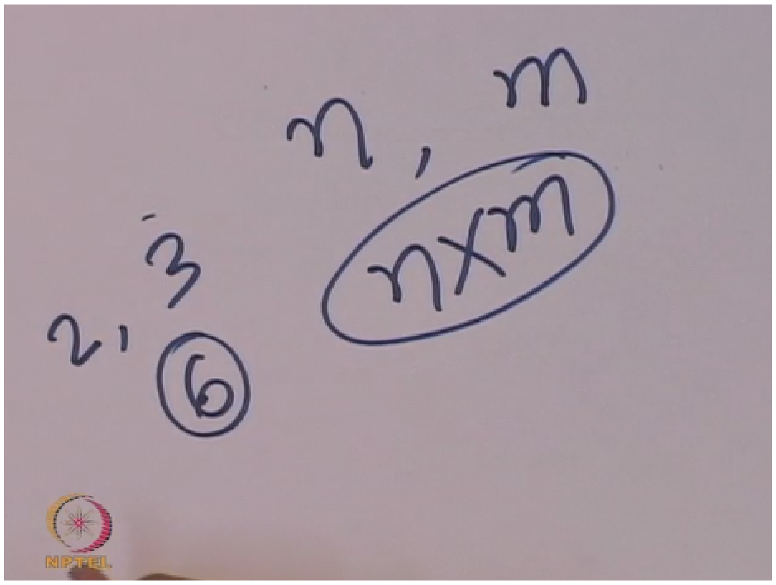
- Build product machine of M_1 and M_2
- Traverse state-space of product machine starting from reset states S_0, S_1
- Test equivalence of outputs in each state
- Can use any state-space traversal technique

Advance VLSI Design 3

Machine M1 and machine M2 and then we generate a product machine out of these two and now for this product machine if we start from a say initial state as 0 of this machine and initial state as when off machine M2 then we check in the product machine whether we can reach some illegal state by traversing this machine or not. If we reach to an illegal state in that case here both of the machines are not equivalent otherwise machines are equivalent to each other.

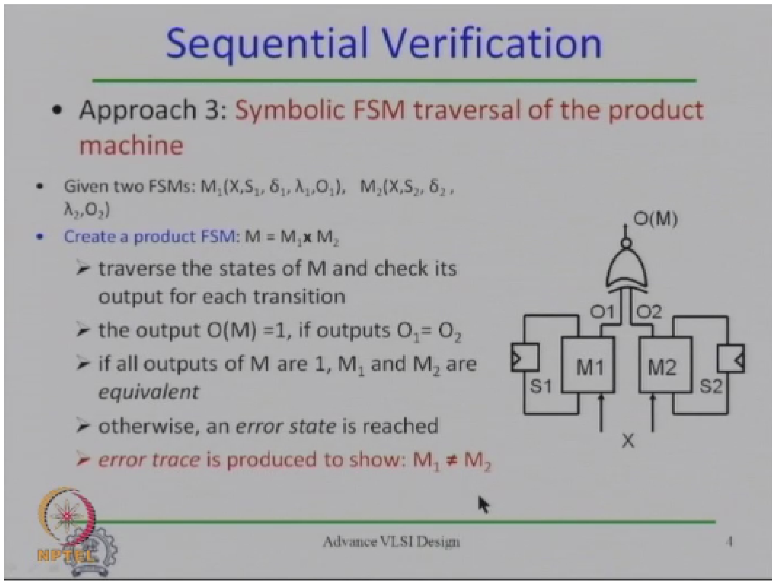
So, as I mention that here we are constructing a product machine so say if one machine has say n number of stairs, another machine has m number of stairs. In that case, here product machine will have total n into m numbers of states. So say one haS2 and another haS3, so that means here we have all possible 6 combinations of these states and based on the state transition some of the combinations are legal.

(Refer Slide Time: 02:48)



Some of the combinations are illegal if we end up to an illegal combination in that case that those two machines are illegal because otherwise these are not supposed to reach to that state.

(Refer Slide Time: 03:12)



Sequential Verification

- Approach 3: Symbolic FSM traversal of the product machine
  - Given two FSMs: $M_1(X,S_1,\delta_1,\lambda_1,O_1)$, $M_2(X,S_2,\delta_2,\lambda_2,O_2)$
  - Create a product FSM: $M = M_1 \times M_2$
    - traverse the states of M and check its output for each transition
    - the output O(M) =1, if outputs $O_1 = O_2$
    - if all outputs of M are 1, M_1 and M_2 are *equivalent*
    - otherwise, an *error state* is reached
    - *error trace* is produced to show: $M_1 \neq M_2$

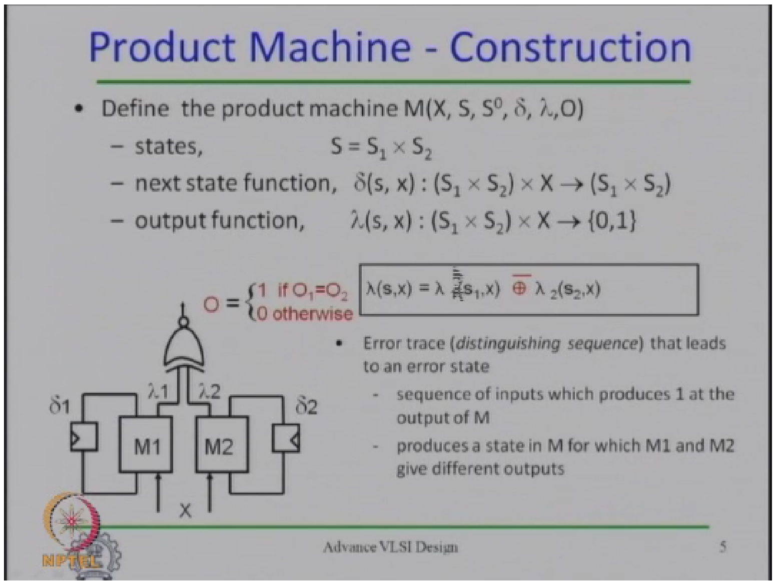
Advance VLSI Design 4

So, as I said that we use the symbolic FSM traversal of the product machine and figure out whether we are able to reach to the illegal state or not. So then here how we formulate that. So, say these are the two machines one is machine M1 that has some combination part and sequential, the free flops. And the another machine is M2 so now what do we do is we construct a product machine and whose output of both of the machines are connected to a XNOR gate.

And this XNOR gate is supposed to produce output one if both of the machines are equivalent otherwise it may produce 0. Now, you have two machines M1 and machine M2 so we create a product machine M that is M1 cross M2 we traverse the states of M that is product machine and check if the output of each of the machine. If the output of machine M iS1 output of machine M iS1 only when the output of both of the machines are one otherwise it is 0.

If all output of M are one in that case here M1 and M2 are equivalent otherwise these are not equivalent and then here we can reach to an error state and then this error trace will produce the counter example. So that means you are looking at the error trace we can find out what is the source of the bunk. So now let us come to how to construct such kind of product machine.

(Refer Slide Time: 04:48)



Product Machine - Construction

- Define the product machine M(X, S, S^0, δ, λ,O)
  - states, $S = S_1 \times S_2$
  - next state function, $\delta(s, x) : (S_1 \times S_2) \times X \rightarrow (S_1 \times S_2)$
  - output function, $\lambda(s, x) : (S_1 \times S_2) \times X \rightarrow \{0,1\}$

$O = \begin{cases} 1 & \text{if } O_1 = O_2 \\ 0 & \text{otherwise} \end{cases}$

$\boxed{\lambda(s,x) = \lambda_1(s_1,x) \overline{\oplus} \lambda_2(s_2,x)}$

- Error trace (*distinguishing sequence*) that leads to an error state
  - sequence of inputs which produces 1 at the output of M
  - produces a state in M for which M1 and M2 give different outputs

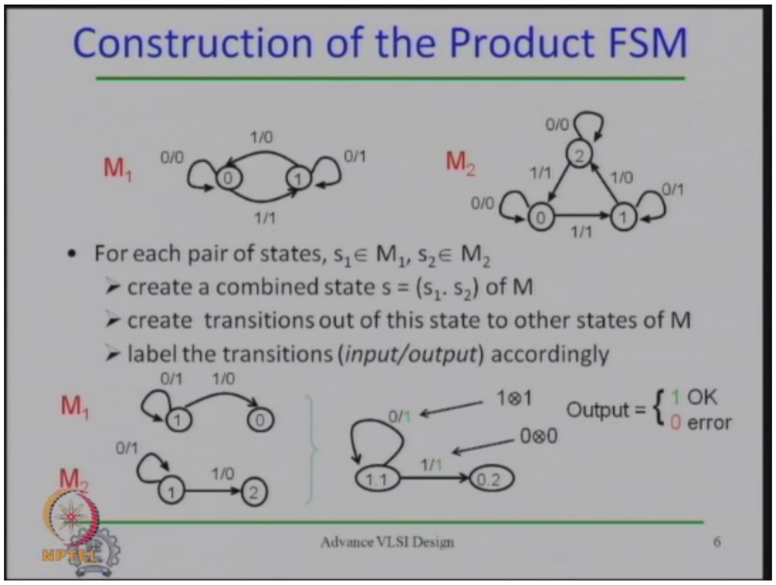
Advance VLSI Design 5

So there are two machines M1 and M2 and say S1 is the set of machine M1 and S2 is the set, set of machine M2. Now here the product machine will have two state set S that is S1 cross S2. So, if S1 is to S2 is three that means here that would be total 6 states in the product machine. And next state function that delta of s into x would be S1 cross S1 into X and this will map to S1 cross S2 and out function lambda of s of x would be S1 cross S2 cross X to either 0 or one.

So, when O1 and O2 are same in that case here output is 1 otherwise it is 0. Construct this and then here this construction of this output is known as (()) (05:47). So, now here the lambda function that is the output function would be, the output function of machine M1 and output X XNOR with output function of machine M2. Sorry, I mentioned earlier X, this is X nor operation.

So, now here we look at the error trace if we are getting output as 0 and that error trace is the distinguishing sequence so that means here sequence that can produce two different output from the two different machines. So, it is sequence of input which produce 1 at the output. Let us look at the how we can distinguish 2 machines.

(Refer Slide Time: 06:23)



Construction of the Product FSM

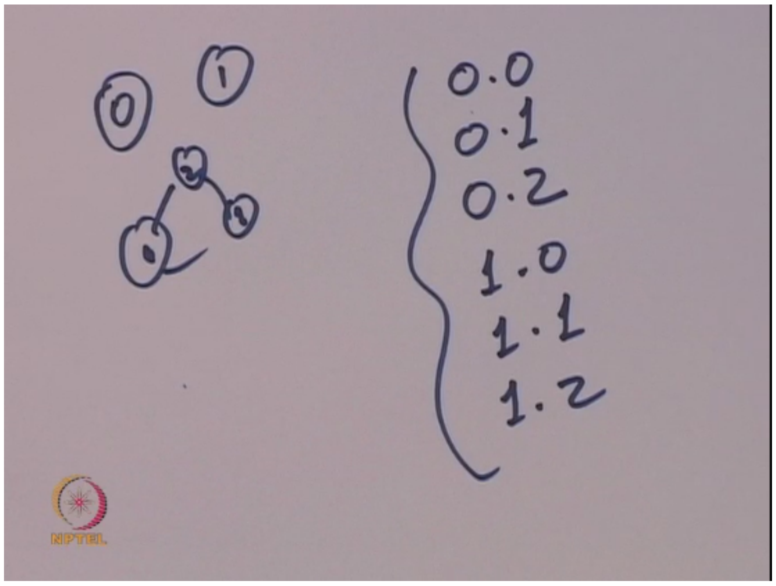
- For each pair of states, $s_1 \in M_1$, $s_2 \in M_2$
  - create a combined state $s = (s_1.\ s_2)$ of M
  - create transitions out of this state to other states of M
  - label the transitions (*input/output*) accordingly

Advance VLSI Design 6

Say machine M1 has 2 states 0 and 1 and machine M2 has 3 states 0, 1 and 2 and these are the state transitions. Now here we would like to see whether both of the machines are equivalent or not. Now what do we do? We need to construct a product machine and product machine will have say here this machine has 2 states, this haS3 state. So, that means here product machine will have 6 states.

Out of 6 states some are illegal some are legal and what we want? We want that here all the legal machines states should be reachable and all the illegal machines should not be reachable. So, now we create a combine state that is S1 into S2 and so say here we have state 0 and 1 here we have 0, 1, 2. So, now the combined state that we can create out of this, this is 0 and 1 and one has 0, 1 and 2.

(Refer Slide Time: 07:34)



Now here the combined state would be 0, 0, 0.1, 0.2, then 1.0, 1.1, and 1.2. So, these are the total 6 states a product machine will have. Some will be legal states; some states are illegal states. Now here we have to look the state transition. So these are the 6 states available and now we have to construct the state transition. So, the state transitions are labeled over these ages so that means here this says that if I apply 0 in that case here this will stay in the same state and output would be 1.

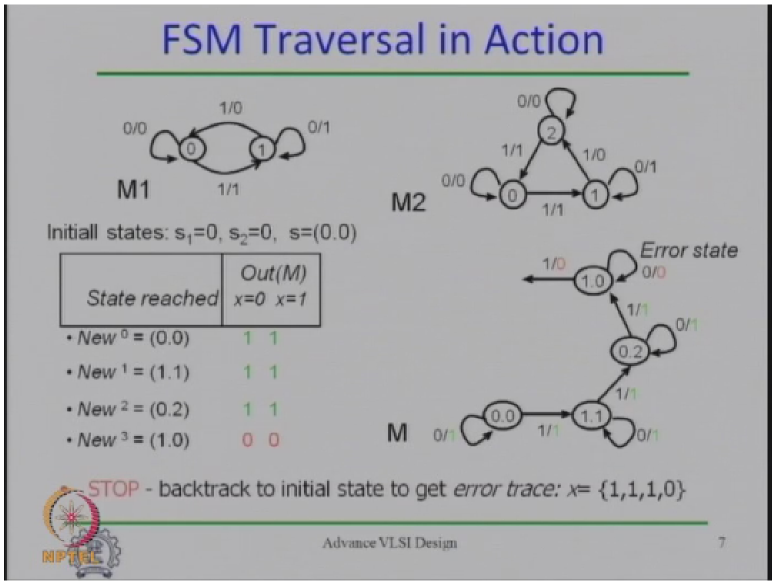
So, let us say that initially this machine is in state 1 and this machine is also in state 1. So, now when it is in the state 1 in that case here on the arrival of 0 it will stay in the state and produce out aS1 and on arrival of 1 here it will go to state 0 and produce output as 0. This is the state transition when it is in state 1. Now here what are the state transitions of this machine M2 when it is in the state.

So when it is in state 1 in that case here on arrival of 0 it will stay in the state 1 and produces output aS1 and on the arrival of 1 here it will go to state 2 and produces output 0. So, now here say in the product machine that may have a state 1, 1 and 0, 2 right. So, now here on arrival of 0, this will stay in state 1, this will stay in state 1 so that means your product machine will have a state 1.1 and on the arrival of 0, it will stay in the same state.

And now here output of this machine iS1 output of this machine iS1 and XNOR of these two outputs would be 1. So, output will 1 now here on arrival of 1 as input in that case here this machine will go 0. This machine will gO2 in that case here the product machine will go to a state that is leveled with 0.2. So, on arrival of 1 here it will go to state 0.2 and because out in machine M1 is 0, output in machine M2 is 0 and XNOR of these two machines would be 1 so in that case here output would be 1.

This way here we construct the state transition in a product machine. So, if output iS1 in that case here these are equivalent it is okay if output is 0 in that case here the product machine enters into erroneous state.

(Refer Slide Time: 10:19)



FSM Traversal in Action

Initiall states: $s_1=0$, $s_2=0$, $s=(0.0)$

State reached	*Out(M)* *x=0*	*x=1*
• *New* ^{0} = (0.0)	1	1
• *New* ^{1} = (1.1)	1	1
• *New* ^{2} = (0.2)	1	1
• *New* ^{3} = (1.0)	0	0

STOP - backtrack to initial state to get *error trace:* $x= \{1,1,1,0\}$

Advance VLSI Design 7

So, let us take an example and see whether these two machines are equivalent or not. So, these are the two machines and I say I start from some state say initial state is this machine in 0 and initial state in this machine is 0. So, now here start to construct a product machine and see how far we can go and how many stairs we can reach. So, now we maintain a repository of states which are already reaches and then the output on arrival of 0 and one.

So, initially both of these machines are in state 0 in that case initial state is 0, 0 that I can write as 0. 0 and now on the arrival of 0 this machine stays in state 0. This machine also stays in state 0. So, that means your product machine will stay in 0.0 state and output of machine M1 is 0, output of machine M2 is 0 and hence the output of the product machine would be 1 because this is

XNOR of outputs of these two machines.

Now, on the arrival of 1 so now here on the arrival of 0 as input, output will be 1. Now, here look at the output on the arrival of 1 so when the arrival of 1 is there in machine M1 it will go to state 1, on the arrival of 1 in this machine M2 it will also go to state 1 so that means here the product machine will go to a state that iS1.1, right. And what would be the output here? This machine produces output 1 this machine produces output 1.

And product of these, no sorry, XNOR of these two would be one hence here so XNOR iS1. So, now here from this initial state I can reach to state 0, 0 or I can reach to state 1, 1. Now here let us look at if I am in state 1, 1 how it progresses? So now if it is in state 1,1 on arrival of 0 it will say in state 1. On arrival of 0 this also stays in state 1 so then here the product machine will also stay in state 1.1.

So now it will stay in state 1.1 when arrival of 0 and in both of the cases both of the machines are producing output 1. Hence XNOE of these two output would be 1 so it will produce out 1. What happens when you get input aS1? On the arrival of input aS1 this machine will go to state 0, right and produces output 0. This machine will go to state 2 and produces output 1 so that means the combine machine or product machine will produce output 1.

So, it will produce out 1 and will go to state 10.2 because this goes to 0. This goes tO2, all right. Now, here you will reach to the new state that is 0.2. So, these are reachable states. Now in 0.2 on the arrival –so you are in 0 here and 2 here. On, arrival of 0 it will stay in 0, on arrival of 0 this will also stay in 0 so that means you are on the arrival of 0. It will stay back in 0.2 state and produces output aS1 because 0 here and 0 here.

So, this is the case when you receive 0 as input. If you receive 1 as input in that case here from 0 it will go to state 1 froM2 it will go to state 2. So, now here it will go to state 1.0, right this goes here. This goes here. So, now here this will go tO2 state and state 1.0 and then here it will produce output 1. Now, when you are in state 0.2 on arrival of 1, sorry on arrival of 0 it is 0.2. On arrival of 0 it stays back in 0. On arrival of 0 here it stays back in 0.

Sorry, it iS1.0. So, in 1.0 here on the arrival of 0 it will stay back in 1 on arrival of 0 this also stays back in 0. So, now here the product machine will stay back in 1.0 state but this produces output 0. This produces output 1 so these are producing out as 0 and 0 output is erroneous. So, now here you are able to reach an erroneous state hence these two designs are not equivalent. So, now here you can further travers like if it is 0 1 in that case here.

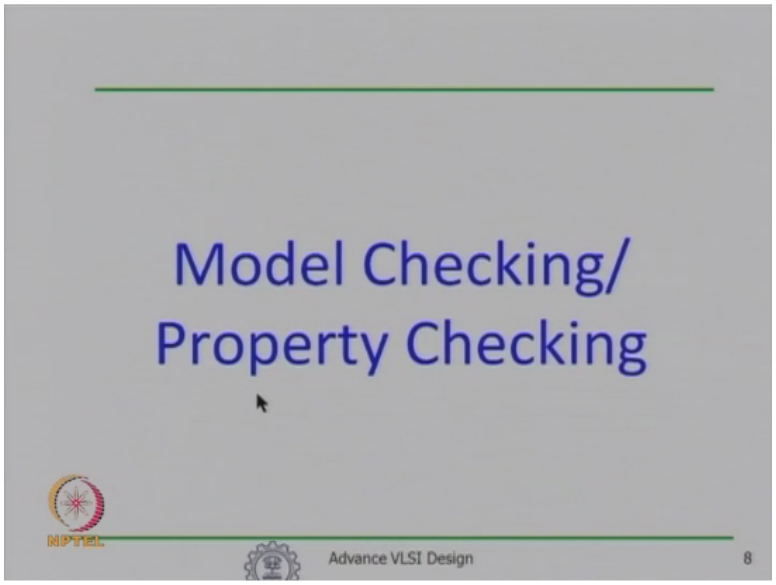
It can go to another state and produces again the erroneous output. So, now here if you want to find out the error trace in that case here your error trace that can result into erroneous state would be 1 1 1 and 0 right. Or 1 1 1 and 1 so these are the error trace so this error trace will tell you that under this condition if these two machines are not equivalent. So, that means under this condition these two machines are producing two different outputs.

Hence these are not equivalent. So, if this process stops either it results into erroneous output or so that means your output is 0 or it is not able to include anymore new states. In that case here you can say that now here all the reachable states are producing output as one and other states which may produce output 0 are not reachable hence your machine is safe under the conditions both of the machines will behave exactly in the same way.

Hence you can verify these two machines. So, this way here we can use the state traversal and we can verify the equivalence of two machines by converting a machine into a product machine. So, far we discussed about the equivalence checking. Three methods of the equivalence checking first method by converting the sequential equivalence checking into a combinational equivalence checking by having the timeframe expansion or by checking the isomorphism between the two state machines or by using the finite statement machine.

So, we have to convert both of the machines into a product machine and we have to traverse that product machine and check whether any of the illegal state is reachable or not. If illegal state is reachable in that case here both of the machines are not equivalent otherwise the machines are equivalent. So, this completes the sequential equivalence checking portion. Now, I switch to the next topic that is model checking or property checking.

(Refer Slide Time: 18:18)



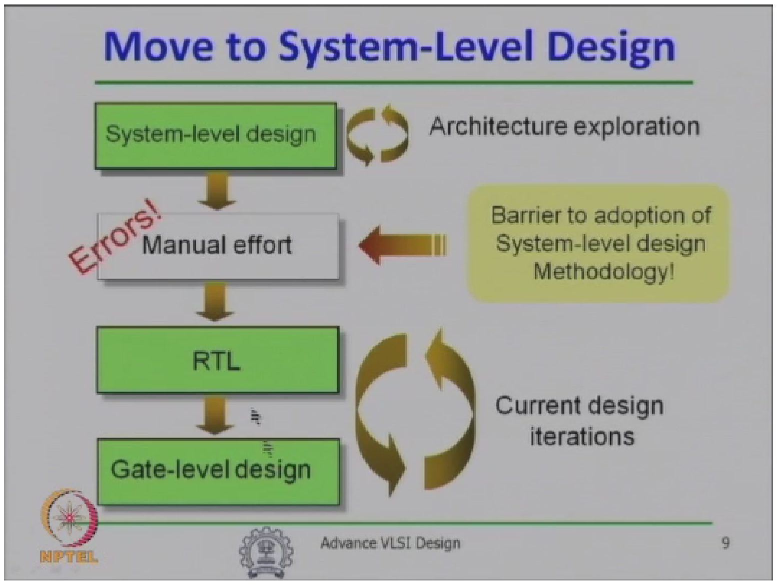
Model Checking/ Property Checking

Advance VLSI Design 8

Where it is more convenient? Where it is more needed like here in the VLSI design flow I have shown you that at various locations we used the equivalence checking so that means here you transform one design from one level of abstraction to another level of abstraction and then you check the equivalence between those two so like here RTL and gate-level net list you apply the equivalence checking and check whether both of the designs are equivalent or not.

Now a day now we are going at higher level of abstraction and that is system level design.

(Refer Slide Time: 18:51)



Move to System-Level Design

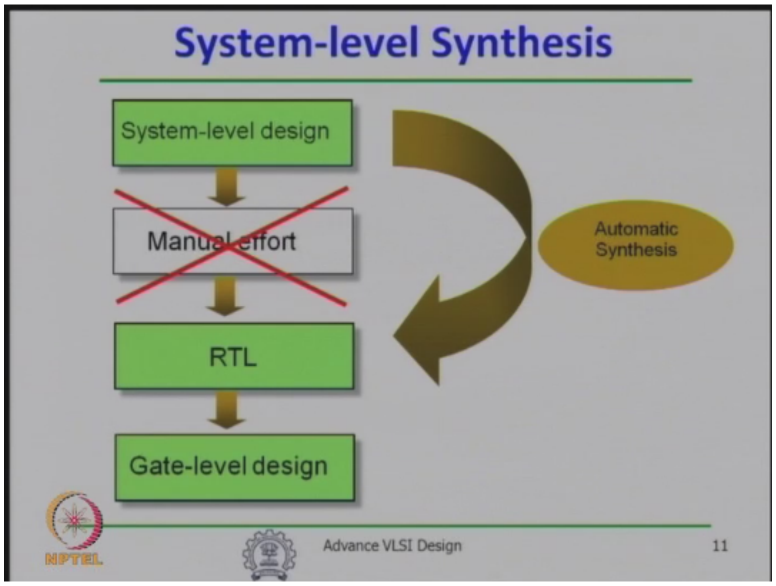
Advance VLSI Design 9

And at system level design we are try to explore the architecture of the system and most of the time the effort from system level design to RTL is manual and if it is manual in that case here we are more error prone. So, this is your specification and now this is the transformation so from this to this if you are going manually in that case here we are likely to introduce some more error.

So, this is the barrier to adopt the system level design methodology because this take is manual somehow if I can do this automatic in that case here I am likely to move myself to a higher level of abstraction where in the complexity is smaller and I can handle the larger design. That is the place where the moral checker or if this is property checkers are more meaningful though these are meaningful at the other level of abstraction as well.

But here your equivalence checking may not help so you have to go for that.

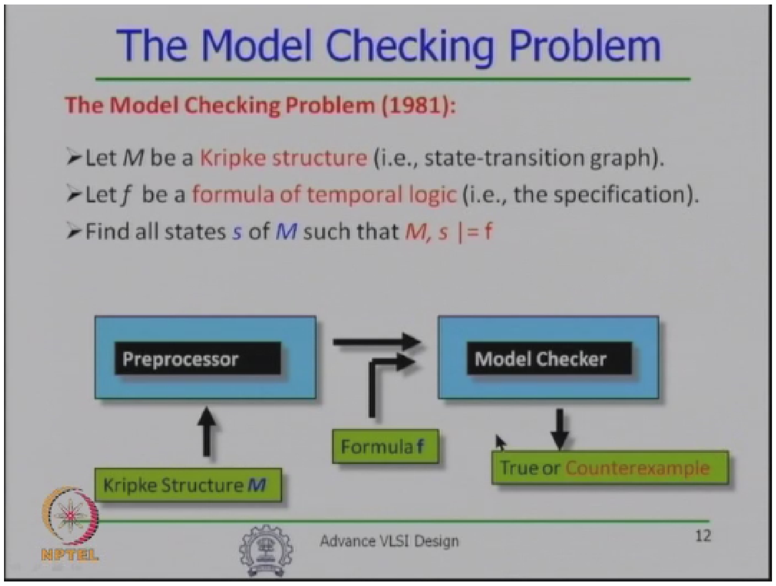
(Refer Slide Time: 20:05)



So now here what we want that whatever properties we have we have to do check the modal for that. So, if you can verify the properties of the system level design and make sure that this design is equivalent after that here you can have equivalence checking between these two and we can upgrade our self to higher level of abstraction. So, that means here we can eliminate after that we can eliminate the manual effort.

So, here what we want because here you have behavioral information about the system and we can have some specification about the system in terms of the properties like if it is an arbiter then we look for the property like deadlock. Like fair chance to each of the master or something like that. If it is traffic light controller in that case, we are interested whether it is giving –it is not proving green signal to cross roads.

(Refer Slide Time: 21:16)



The Model Checking Problem

The Model Checking Problem (1981):

- Let *M* be a Kripke structure (i.e., state-transition graph).
- Let *f* be a formula of temporal logic (i.e., the specification).
- Find all states *s* of *M* such that *M, s* |= f

Preprocessor
Model Checker
Formula f
Kripke Structure M
True or Counterexample
Advance VLSI Design
12

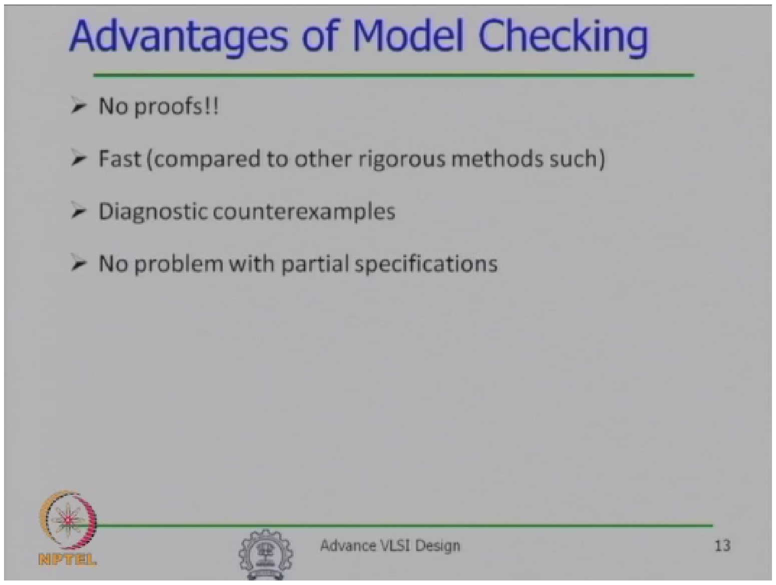
So, this is the place where modal checker can play an important role. Model checker was proposed in 1981 by Emerson and Clarke. So, what it does is it models the implementation in the form of finite state machine or state transition diagram. So, here you have state transition diagram you convert that in little bit different form then state transition diagram that is known as Kripke structure.

I will tell you what that Kripke structure is then because here if it is a sequential circuit in that case it has some temporal behavior. So, that means here we have to specify, the specification in terms of the temporal logic. So, temporal logic is or specification is specified in terms of temporal formula and then we have to check whether in all these states of modal M it satisfies your modal that your finite state machine or Kripke Structure satisfies this formula.

If it satisfies in that case here this is equivalent to all case stimulation for that kind of property or formula. So, the structure is as follows you have a Kripke structure that is nothing but a variant of finite state machine or state transition diagram. You pass it through a preprocessor and then you write your specification in terms of formula F. You supply this to the modal checker, modal checker will tell you whether this formula is respected by this modal or design all the time or not.

If it is not in that case it produces counter example. I will show you one example how this can verify one particular design.

(Refer Slide Time: 23:12)



Advantages of Model Checking

- No proofs!!
- Fast (compared to other rigorous methods such)
- Diagnostic counterexamples
- No problem with partial specifications

Advance VLSI Design 13

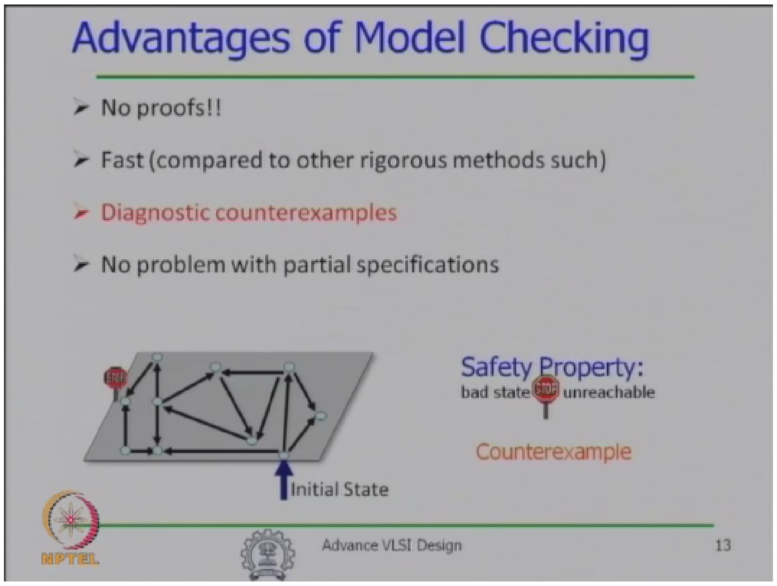
So the advantages of modal checker are as follows. Like in theorem proving, we need to prove a couple of theorems and then the proven theorem we can use as (()) (23:27) then through the remaining theorem. So, in this case and that is as I mentioned earlier that theorem proving is a semiautomatic process. Hence here you need to have manual intervention and industry generally does not like that and again here the theorem provers have scalability problem.

So, here you do not need any proof. This is faster as compared to the other methodologies. It has diagnostic capability based on the counter example. So, counter example can tell you what could have gone wrong that is why your design is not respecting the specification and so other very important thing is that you need to worry about full specification even if you have partial specification you can check for those specifications.

And you can say that these partial specifications or properties are always respected by the design. So, it is not necessary that you need to supply all the specifications. One of the example I can tell you that for a traffic light controller you need not to define all the properties or all the specifications may be you are mostly interested in checking your property something like that it should not give green signal to cross roads.

That is most important, that is defined as safety property. So, other things may still be okay but that is never admissible so now if you say that please check for this property in that case here it will check for that property and tell you that whether that property violated at any point in time or not. So, now here for partial specifications are okay. Many times we do not have access to the complete specifications, as these specifications are evolving.

(Refer Slide Time: 25:36)



Advantages of Model Checking

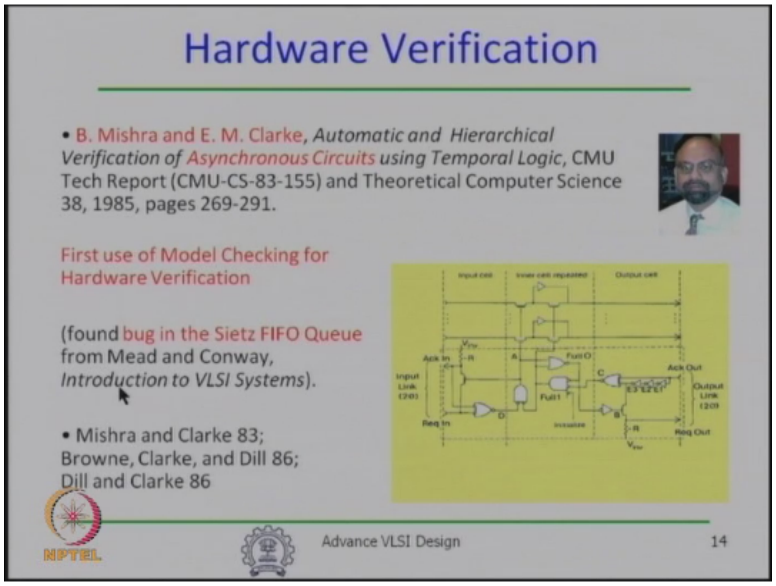
- No proofs!!
- Fast (compared to other rigorous methods such)
- Diagnostic counterexamples
- No problem with partial specifications

Safety Property:
bad state unreachable
Counterexample
Initial State
NPTEL
Advance VLSI Design
13

So now here as I said that here you say this is my said space and then here these are the state points. So, I start from some initial state and then here if I hit here some illegal state in that case here I can identify that here the property in not respected. So now here in this case if you start from here if you hit to some stop state or bad state in that case here it should stop and then you can traverse back and come back to the initial state.

And that back traversal will tell you the counter example and that tells you that if you pass through this particular rout or states you will encounter a bad state and hence your design is not respecting this property. These things the hardware verification was first exercised by Mishra and Clark from CMU in 1985.

(Refer Slide Time: 26:36)



Hardware Verification

- B. Mishra and E. M. Clarke, *Automatic and Hierarchical Verification of Asynchronous Circuits using Temporal Logic*, CMU Tech Report (CMU-CS-83-155) and Theoretical Computer Science 38, 1985, pages 269-291.

First use of Model Checking for Hardware Verification

(found bug in the Sietz FIFO Queue from Mead and Conway, *Introduction to VLSI Systems*).

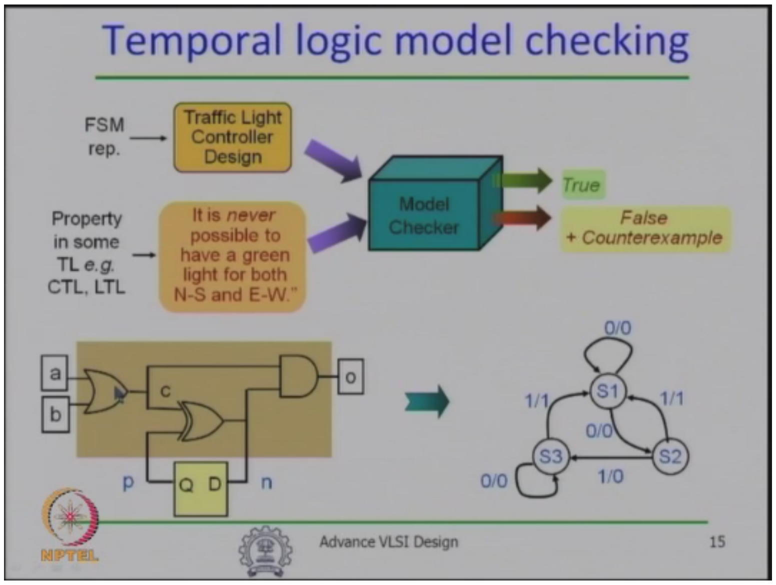
- Mishra and Clarke 83; Browne, Clarke, and Dill 86; Dill and Clarke 86

Advance VLSI Design 14

So, this was the modal checker exercise for hardware verification. Though here the modal checker was developed before that in 1981 but then here modal checker was explored for the software verification not for the hardware verification and what Mishra found out was that they were verifying the FIFO implementation from Mead and Conway book, I guess all of you must have gone through this book.

This is one of the standard texts in VLSI design. Now, they find out a bug that was the first bug in standard FIFO implementation was explored. So, this was the first instance of hardware verification using modal checker.

(Refer Slide Time: 27:33)



Temporal logic model checking

FSM rep.
Traffic Light Controller Design
Property in some TL e.g. CTL, LTL
It is *never* possible to have a green light for both N-S and E-W.”
Model Checker
True
False + Counterexample
a
b
c
o
p
Q D
n
0/0
S1
1/1
1/1
0/0
S3
S2
0/0
1/0

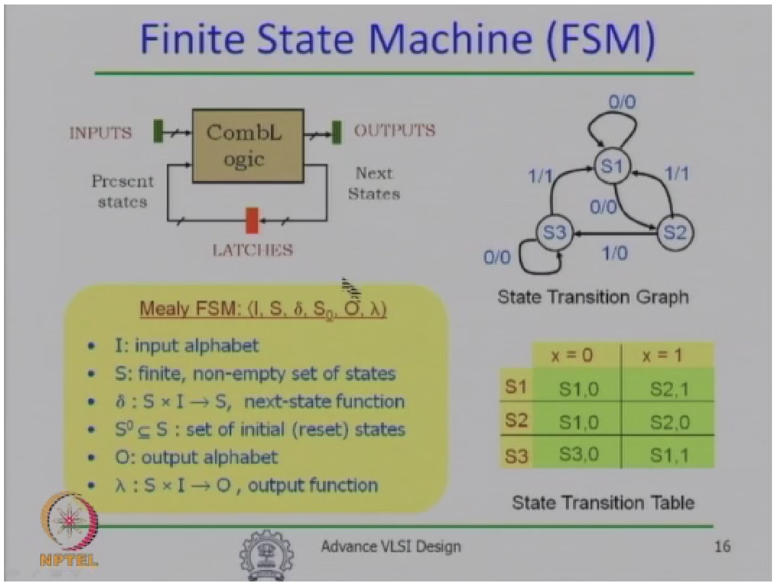
Advance VLSI Design 15

So now here how modal checker process flow goes and what are the various steps. How we can do that? So, as I mentioned that we need to supply two information to modal checker one is your design. Like here for example your design may be at traffic light controller. So, now your design can be supplied as a finite state machine of traffic light controller. Then you have to specify some interesting properties that you would like to verify.

And those properties could be like here you can say that it is never possible to have a green light for both north, south and east, west roads. It is safety violation because otherwise there could be an accident. So you have to avoid that so now here it will either say that your properties always respected or it is true. Or if property fails in that case here it gives you the counter example that under this condition this property violates.

So, in general in our hardware design we have this king of sequential circuit where in you have couple of flip flops and input output that I can form this design I can extract finite state machine so this finite statement machine and that is referred as finite state model for the given design. So, this process is extraction of finite statement machine or modal from the design itself.

(Refer Slide Time: 29:10)



Finite State Machine (FSM)

Mealy FSM: $(I, S, \delta, S_0, O, \lambda)$

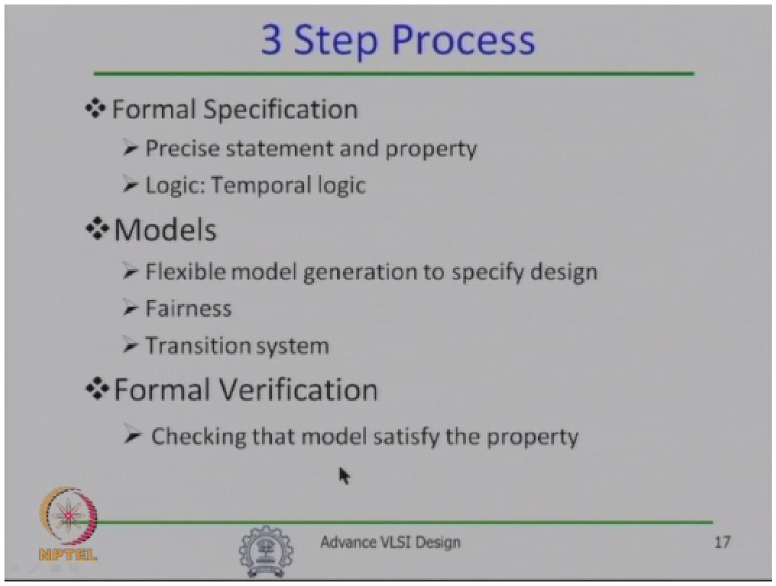
- I: input alphabet
- S: finite, non-empty set of states
- $\delta : S \times I \rightarrow S$, next-state function
- $S^0 \subseteq S$: set of initial (reset) states
- O: output alphabet
- $\lambda : S \times I \rightarrow O$, output function

	x = 0	x = 1
S1	S1,0	S2,1
S2	S1,0	S2,0
S3	S3,0	S1,1

State Transition Table

So, finite statement machine all of you know that finite statement machine may be Moore machine or Mealy machine and that can be represented by (()) (29:24) that is I is input, or set of states, set transition function, initial states out outputs and then the output functions. So, this can be defined formally like this.

(Refer Slide Time: 29:40)



3 Step Process

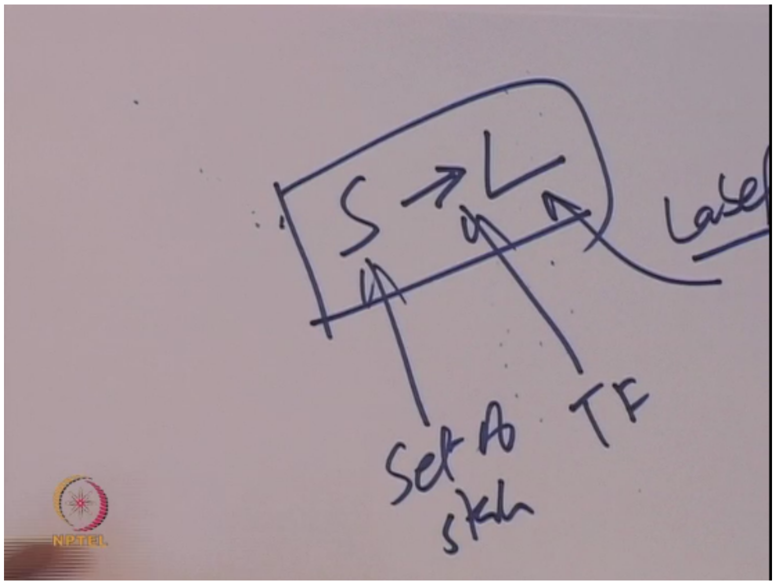
- ❖ Formal Specification
  - ➢ Precise statement and property
  - ➢ Logic: Temporal logic
- ❖ Models
  - ➢ Flexible model generation to specify design
  - ➢ Fairness
  - ➢ Transition system
- ❖ Formal Verification
  - ➢ Checking that model satisfy the property

Advance VLSI Design 17

As I mentioned earlier your modal checker process iS3 step process in that you need to formally specify or I can say mathematically specify the behavior or the specification of the system. So, this is the precise statement and properties and generally because here sequential circuits are temporal in nature in that case here we have to specify this in temporal logic. And the behavior of design or implementation is referred as models.

And that is defined as flexible modal for a given specified design. So, if it is model in that case here it will have transition systems. So transition system will have (()) (30:37) So, a state transition and label so now here state is set of states.

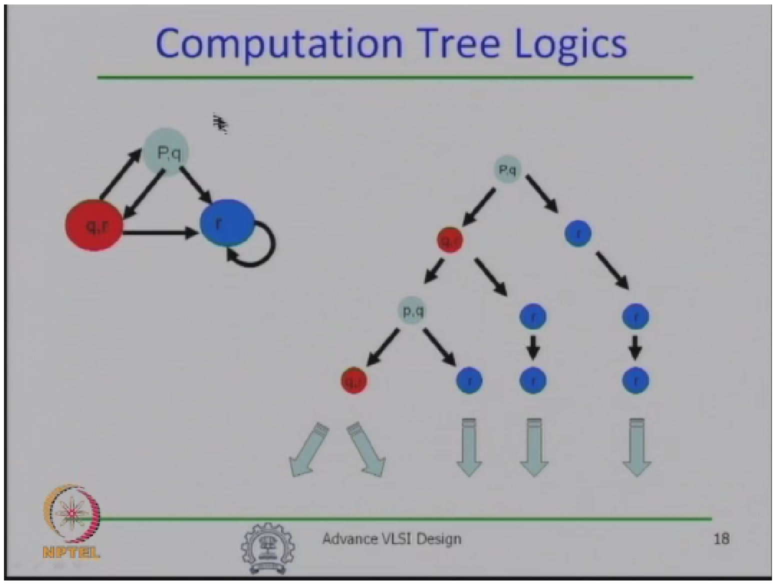
(Refer Slide Time: 30:47)



This is transition function and these are the labels. So, the labels are the variables which hold good or which are true in that particular state. So this way you define the model. Here you define modal in terms of finite statement machine here you define formal specification in temporal logic that can be computational trilogy or that can be linear temporal logic. Then here you have to submit this to the formal verification.

Formal verification tool will check whether your model satisfies the property all the time or not.

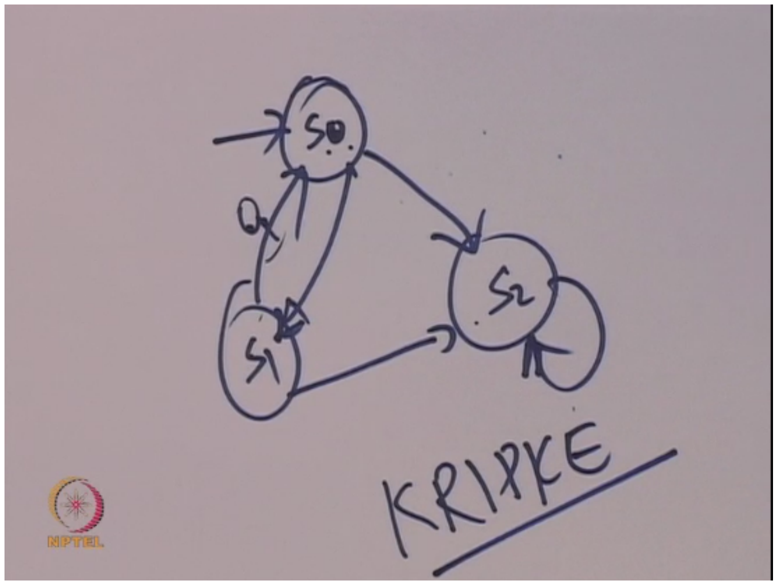
(Refer Slide Time: 31:29)



Computation Tree Logics

So, now if you have say finite state machine say this is your fine state machine. And in this finite state machine I just do little bit changes in that. Finite statement machine if you look at then machine is something like this.

(Refer Slide Time: 31:45)



This may be the machine and now here if it is in the state this say S0, S1, S2. This can go from this state to this state or this can go from S1 to S2 on arrival of some input. Now, here when it goes from this state to this state may be some variable may become valid or high. So, say here whatever variable which are true in that particular state we label in that state. And this kind of convergence.

So, see in this what we are interested in? We are interested in whether all the states are respecting the properties specified in the specification and or in terms of temporal logic. So, now we are interested that whether these properties are respected in all the states so now we are not interested what output it is producing or what input it is getting to go from state S0 to S1. We know that this can go from state S1 to S0.

So, now here we want to eliminate I o from the edges just here we want to move this input, output information from edges to the states. And this conversion is known as conversion to Kripka structure. So, now here say in this if say variable P and Q are true, variable Q and R, are true here and variable R is true in this. So, now here we have to argue or reason about this design based on these two variables.

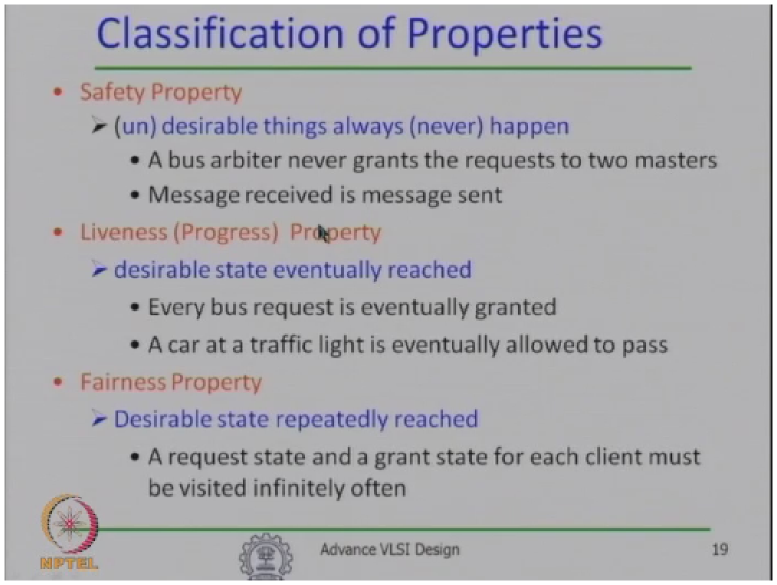
So, now here how the computation behavior progresses so if say this is the initial state where in P and Q are true in that case here always it will start from this state. Now here based on the value of input either it can go to state where in the q and r, are true or it can go to a state where r is true. So, now it can go here or here. In the next time, from this Q or state, it can either go to PQ state or it can go to a state where r is true.

So, that means here it can go to p, q or r and from r it can stay with r only. So, now here this way the computation progresses, computation of this state machine and so now here it is constructing a tree and that is known as computation tree. So, now here we have to specify the specification based on this computation tree which is specified here so that logic is known as computation tree logic.

Now, here the behavior or the timing behavior here would be either you can believe that time in every cycle it is going from this state to, this state, to this state, to this state. And then this here keep on going. Now, here if we look at one path in this computation tree in that case here in very clock ticket advances to the next state to the next state, to the next so in that case here the behavior is linear.

Or if I look at the sum in the computation tree in that case if I look at some state in that case here it can branch out to either this way or this way when it goes here in that case again it can branch out this way or this way. So that means here behavior depends on the branching node. So, that is known as computation tree logic or branching tree logic.

(Refer Slide Time: 35:34)



Classification of Properties

- Safety Property
  - (un) desirable things always (never) happen
    - A bus arbiter never grants the requests to two masters
    - Message received is message sent
- Liveness (Progress) Property
  - desirable state eventually reached
    - Every bus request is eventually granted
    - A car at a traffic light is eventually allowed to pass
- Fairness Property
  - Desirable state repeatedly reached
    - A request state and a grant state for each client must be visited infinitely often

Advance VLSI Design 19

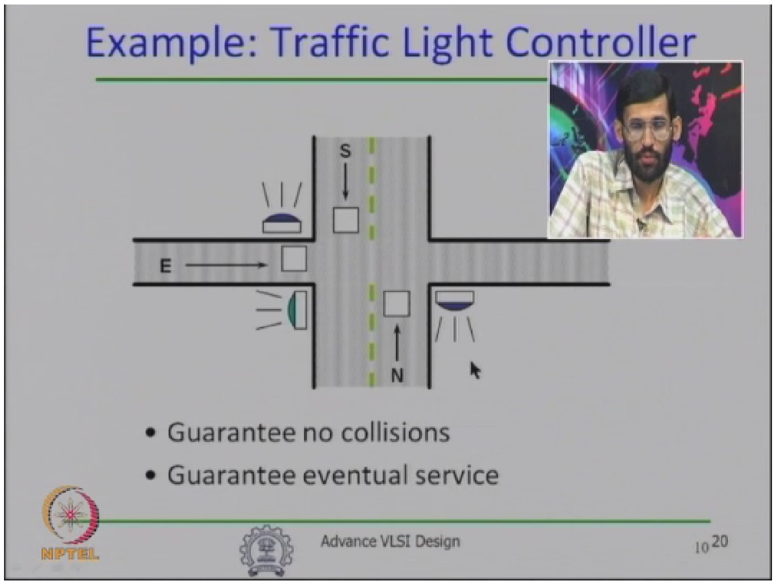
Now the question is so this way we can construct a computation tree. Now the question is what kind of means how we can specify property or what kind of property we need to specify. So, there are couples are types of the properties we can specify say one is the safety property and safety property says that desirable things should always happen and undesirable things should never happen.

So, that means like here for example if I have a bus arbiter in that case here bus arbiter should never grant request to two masters otherwise both will start to use the common shared resource that is illegal. Or if I send a message from sender in that case here the same message must be received at receiver. So message received must be send by some body. These are the safety property.

There are other properties like here liveness property that tells you about the progress of the system. So, what it says is that desirable state should eventually be reached. So that means like here if I desire to get to some state in that case here add some point in time that state must be reached. So for example a bus arbitrator request is eventually granted. A car which arrives at a traffic light that should eventually be allowed to pass through.

So, these are the kind of liveness properties we use. There are say fairness properties. What fairness properties say? That desirable state should repeatedly reach. So, that means here if I desire some state in that case here at some point in time that should be reached and then here again and again there should be at least one of the pass that can take you to the that state again and again. So, a request state and a grant state for each client must be visited infinite stimuli often.

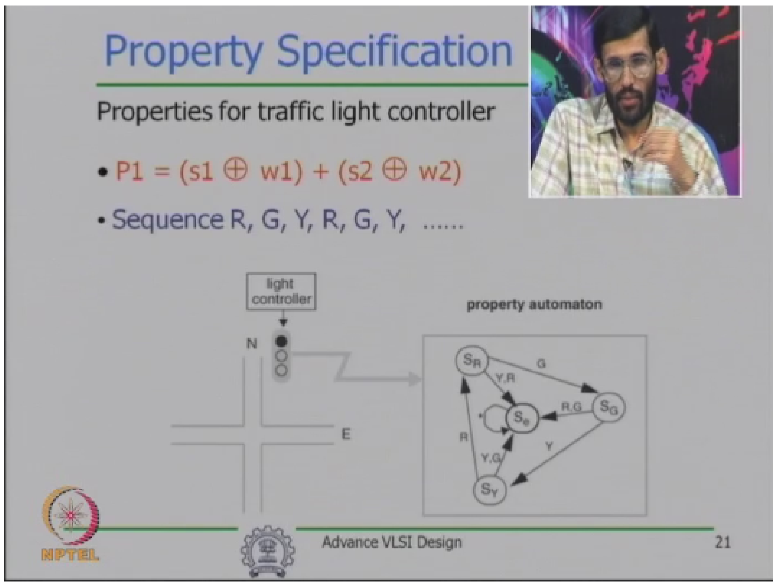
(Refer Slide Time: 37:39)


Example: Traffic Light Controller

- Guarantee no collisions
- Guarantee eventual service

So, like here for example in this traffic light controller I am sensing these vehicles by some of the detectors and now here what we can do is we have to make sure that here there should not be any collision and when there can be a collision if you provide green signal to both of the cross roads in that case there may be a collision. And we have to guarantee the service eventually. So that means here if say some car arrives here it should be eventually be passed from this line. Okay.

(Refer Slide Time: 38:18)


Property Specification

Properties for traffic light controller

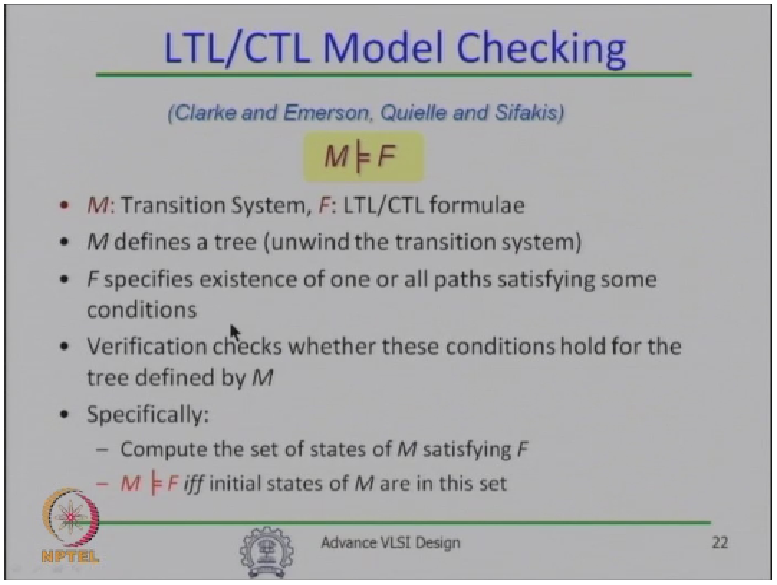
- P1 = (s1 ⊕ w1) + (s2 ⊕ w2)
- Sequence R, G, Y, R, G, Y, ...

Advance VLSI Design 21

So now here some of the properties we can specify like this. Some properties may be very static in nature. Let us say this is a finite statement machine for traffic light controller wherein I have these three states red, green, yellow and then here if there is an error in that case here it may go to the erroneous state. It is shown here. So, now here if I say there are four states and if I represent 2 bits per state.

In that case here I can say statically that in any of the state these 2 bits which are representing these states should never be same. So, that means that can make sure that here at the same time I cannot green and green signals.

(Refer Slide Time: 39:20)


LTL/CTL Model Checking

(Clarke and Emerson, Quielle and Sifakis)

$M \models F$

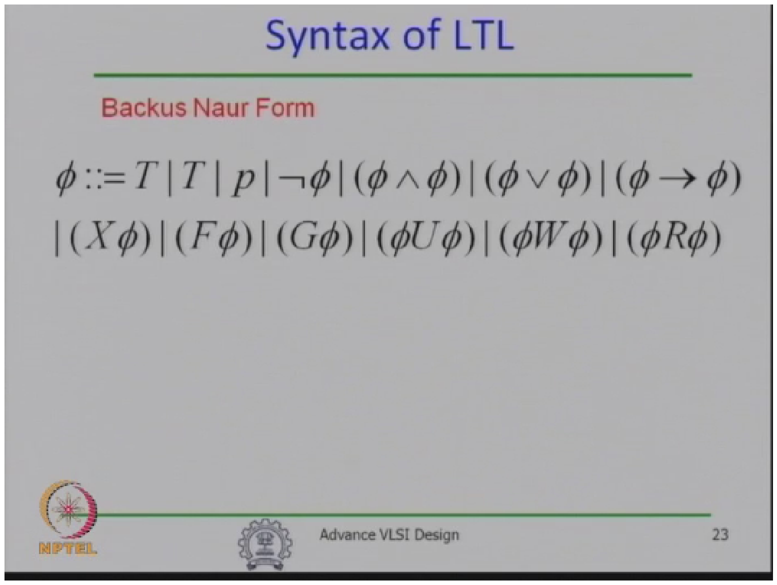
- *M*: Transition System, *F*: LTL/CTL formulae
- *M* defines a tree (unwind the transition system)
- *F* specifies existence of one or all paths satisfying some conditions
- Verification checks whether these conditions hold for the tree defined by *M*
- Specifically:
  - Compute the set of states of *M* satisfying *F*
  - $M \models F$ *iff* initial states of *M* are in this set

Advance VLSI Design 22

This is the repetition of the same slide that here you can represent a specification using formula F and that is in terms of temper logic and you can represent your design using state machine so in this your M is the transition system. As I said that transition system will have a set of states, transition function, and label of each and every state. So, now here M is defined as computation tree.

So, it is a tree and specifies the formula this verification checks whether these conditions hold for all the tree defined by this machine M. So, it computes the set of state of M satisfying formula F and M satisfies F if and only if initial state of M are in the same state.

(Refer Slide Time: 40:24)



Syntax of LTL

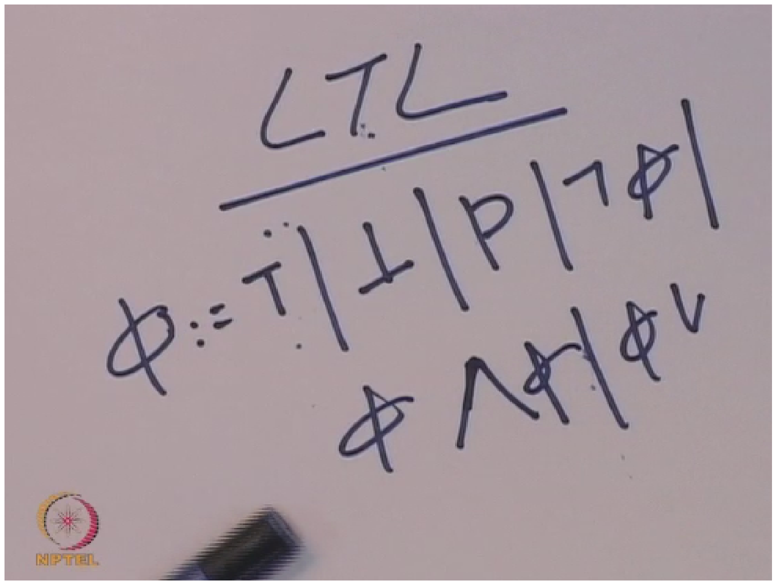
Backus Naur Form

$$\phi ::= T \mid T \mid p \mid \neg\phi \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi)$$
$$\mid (X\phi) \mid (F\phi) \mid (G\phi) \mid (\phi U \phi) \mid (\phi W \phi) \mid (\phi R \phi)$$

Advance VLSI Design 23

So now in order to formally specify as a language here temporal logic that uses the linear semantics is defined as linear time temporal logic LTL

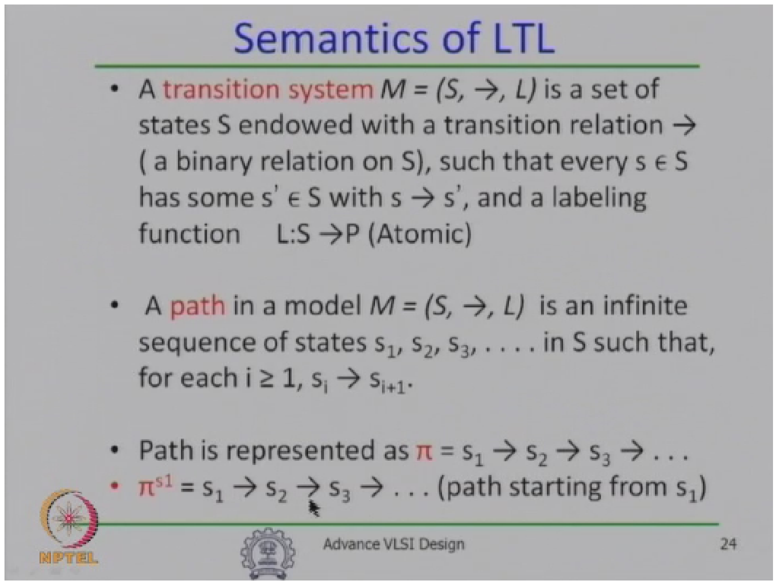
(Refer Slide Time: 40:39)


LTL
$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee$

An LTL is formally defined using Backus–Naur form that is being used in computer science. So a phi is a formula that phi can be given as universal truth that is tautology. Then it can be given as negation of the universal truth that is inverse of tautology and then this can be a formula P. This can be negation of formula, negation of phi. This can be an formula intersection formula conjunction with another formula disjunction with another formula.

This formula implies another formula and then some temporal operators those are X so this operator this conjunction, disjunction, and implications are the static formula and then there are temporal formula that is X F G until weak until and R. So, now here I will explain you in detail what are these formulas.

(Refer Slide Time: 42:01)



Semantics of LTL

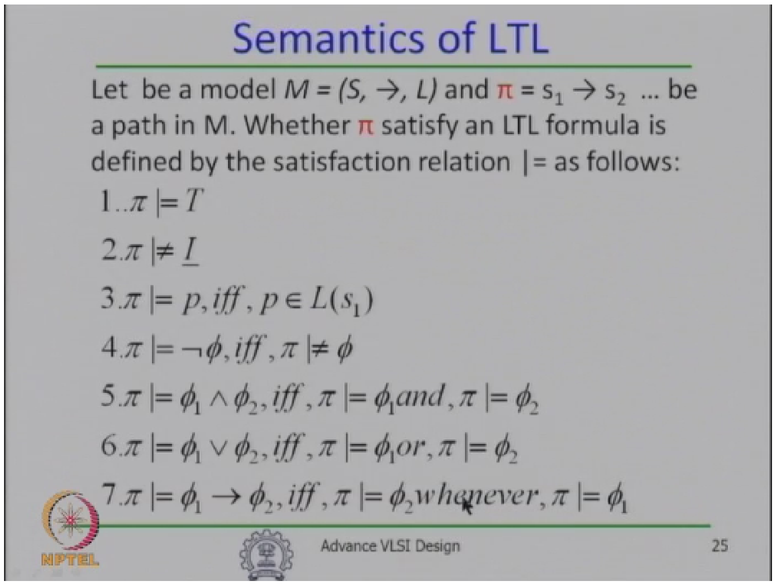
- A transition system $M = (S, \rightarrow, L)$ is a set of states S endowed with a transition relation $\rightarrow$ (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$, and a labeling function L:S $\rightarrow$P (Atomic)
- A path in a model $M = (S, \rightarrow, L)$ is an infinite sequence of states $s_1, s_2, s_3, \ldots$ in S such that, for each $i \geq 1$, $s_i \rightarrow s_{i+1}$.
- Path is represented as $\pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$
- $\pi^{s1} = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ (path starting from s_1)

Advance VLSI Design 24

So, now here we specify a system as a transition system S is the set of states. This is transition function and then here L are the label. So this is a set of states as endowed with a transition relation and in such way that every state S has some state S dash. This function is specified as a transition system M that is S transition function and L and this is specified as a set of states S endowed with a transition relationship.

On such that every state small s is set of member of s has some state, s dash this which belongs also to the set s where in you have transition from s to s dash and that is labelled by label S. In that here we define a path that is in a modal M which is an infinite sequence of state S1, S2, S3, s n. And this path is represented as a pi and pi starting from a path S1 is defined as S1 to S2, S3, and S4 and Sn.

(Refer Slide Time: 43:16)



Semantics of LTL

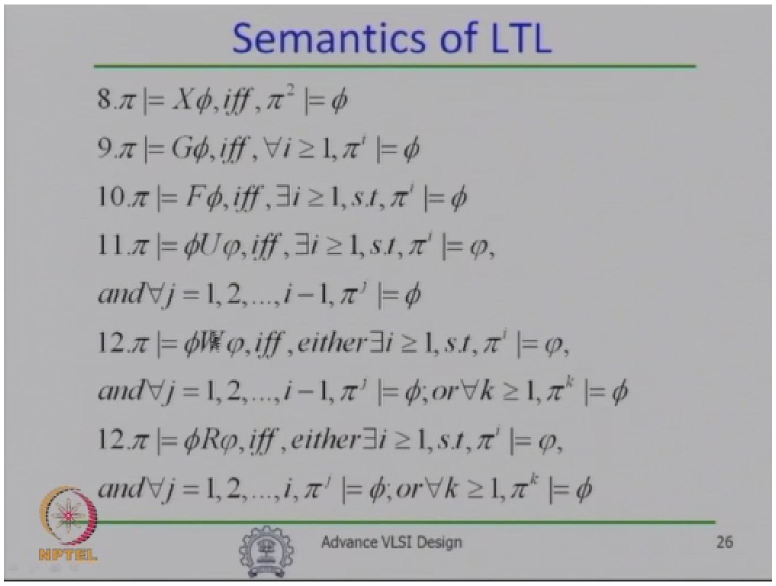
Let be a model $M = (S, \rightarrow, L)$ and $\pi = s_1 \rightarrow s_2$ ... be a path in M. Whether π satisfy an LTL formula is defined by the satisfaction relation |= as follows:

1. $\pi \models T$
2. $\pi \not\models \underline{I}$
3. $\pi \models p, iff, p \in L(s_1)$
4. $\pi \models \neg\phi, iff, \pi \not\models \phi$
5. $\pi \models \phi_1 \wedge \phi_2, iff, \pi \models \phi_1 and, \pi \models \phi_2$
6. $\pi \models \phi_1 \vee \phi_2, iff, \pi \models \phi_1 or, \pi \models \phi_2$
7. $\pi \models \phi_1 \rightarrow \phi_2, iff, \pi \models \phi_2 whenever, \pi \models \phi_1$

Advance VLSI Design 25

So this formula as we discussed earlier this formula those are called as LTL formula those are defined here. That this pi is can be a temporal formula T that can be a temporal formula of inverse of tautology that can be a formula that is valid LTL formula this can be negation, this can be conjunction, disjunction and implication. So, then there are temporal operators X, G, F until week until so.

(Refer Slide Time: 45:53)



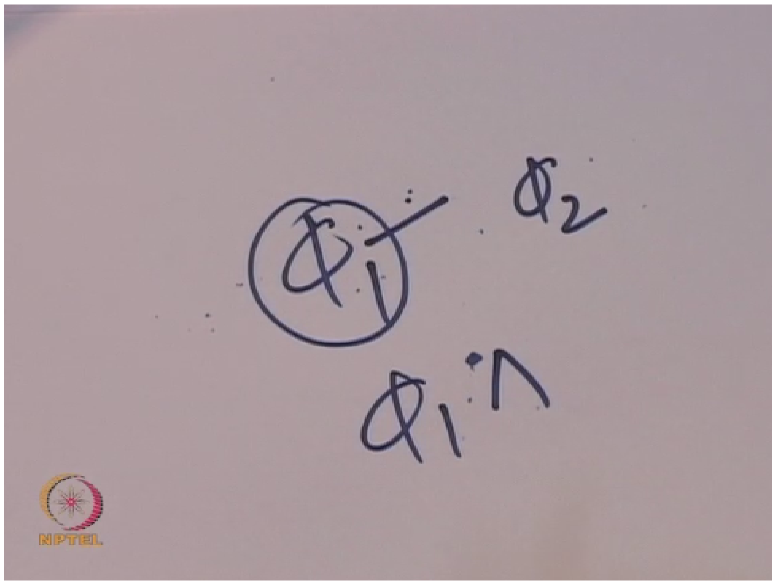
Semantics of LTL

8. $\pi \models X\phi, iff, \pi^2 \models \phi$
9. $\pi \models G\phi, iff, \forall i \geq 1, \pi^i \models \phi$
10. $\pi \models F\phi, iff, \exists i \geq 1, s.t, \pi^i \models \phi$
11. $\pi \models \phi U \varphi, iff, \exists i \geq 1, s.t, \pi^i \models \varphi,$ $and \forall j = 1, 2, ..., i-1, \pi^j \models \phi$
12. $\pi \models \phi W \varphi, iff, either \exists i \geq 1, s.t, \pi^i \models \varphi,$ $and \forall j = 1, 2, ..., i-1, \pi^j \models \phi; or \forall k \geq 1, \pi^k \models \phi$
12. $\pi \models \phi R \varphi, iff, either \exists i \geq 1, s.t, \pi^i \models \varphi,$ $and \forall j = 1, 2, ..., i, \pi^j \models \phi; or \forall k \geq 1, \pi^k \models \phi$

Advance VLSI Design 26

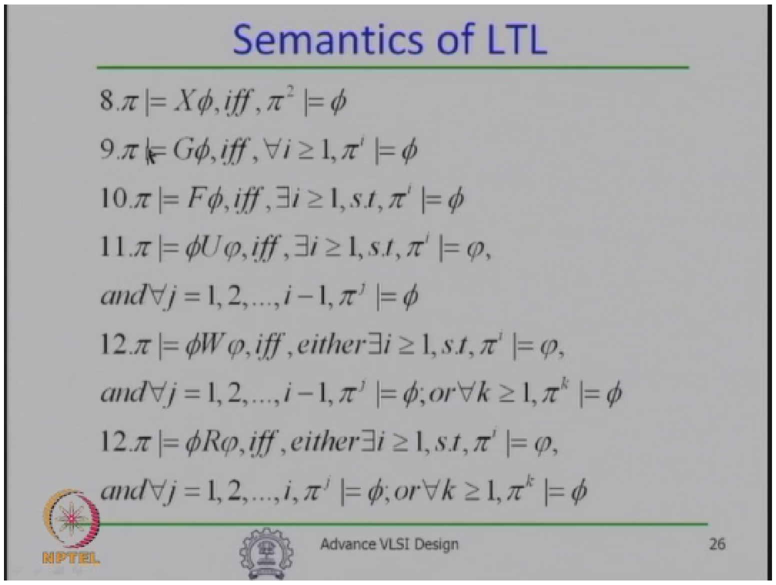
So, now in this if we define this formula say this phi, phi 1 is one formula and phi 2 is another formula.

(Refer Slide Time: 44:04)



So, now here phi 1 and say phi 2. So, this formula holds good if phi 1 holds good and phi 2 holds good. So, phi 1 and distance phi 2 holds good if phi 1 holds good or phi 2 holds good and the implication holds good if phi 1 satisfies so that means phi 2 holds good if and only if phi 1 holds good.

(Refer Slide Time: 44:45)



Semantics of LTL

8. $\pi \models X\phi, iff, \pi^2 \models \phi$

9. $\pi \models G\phi, iff, \forall i \geq 1, \pi^i \models \phi$

10. $\pi \models F\phi, iff, \exists i \geq 1, s.t, \pi^i \models \phi$

11. $\pi \models \phi U \varphi, iff, \exists i \geq 1, s.t, \pi^i \models \varphi,$
$and \forall j = 1, 2, \ldots, i-1, \pi^j \models \phi$

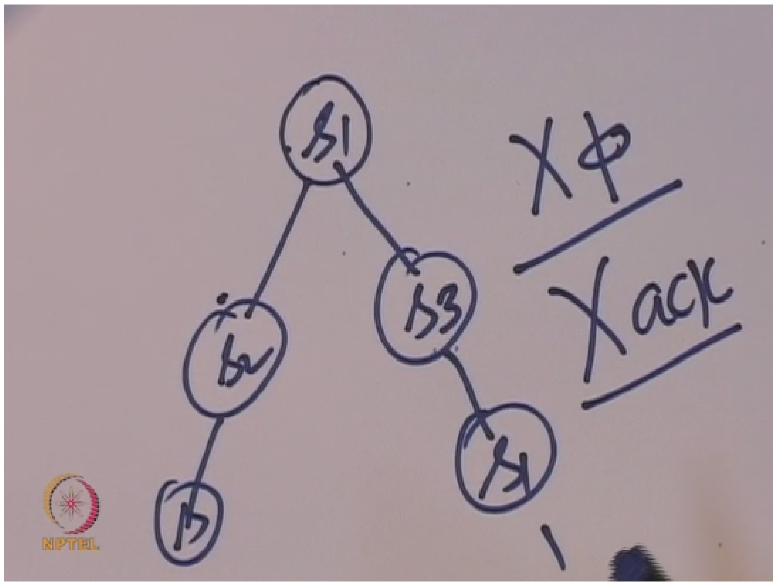
12. $\pi \models \phi W \varphi, iff, either \exists i \geq 1, s.t, \pi^i \models \varphi,$
$and \forall j = 1, 2, \ldots, i-1, \pi^j \models \phi; or \forall k \geq 1, \pi^k \models \phi$

12. $\pi \models \phi R \varphi, iff, either \exists i \geq 1, s.t, \pi^i \models \varphi,$
$and \forall j = 1, 2, \ldots, i, \pi^j \models \phi; or \forall k \geq 1, \pi^k \models \phi$

Advance VLSI Design 26

Now here say in a computation tree, if I go from state S1 to state S2 to state S3 from S1 I can go to say state S3 to state again S1 and so on and so forth.

(Refer Slide Time: 44:47)

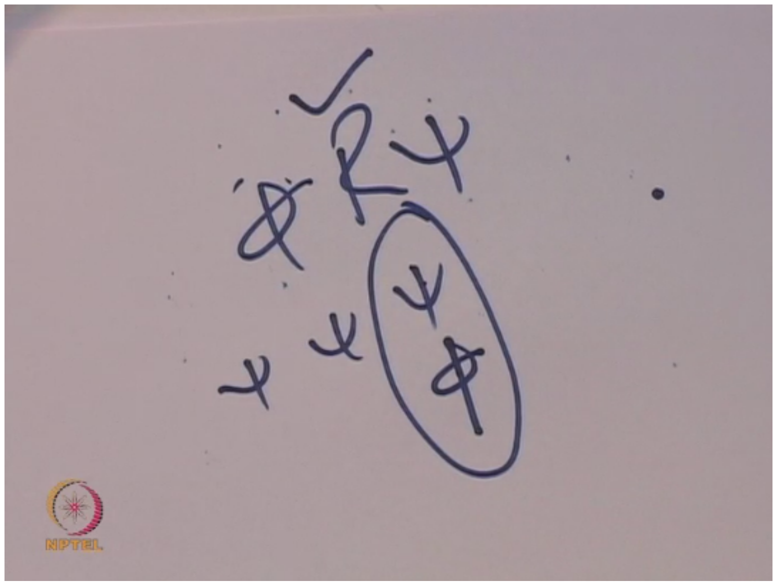


So, now if some formula holds good in exactly in the next state then I define that as X of Phi this is next of Phi that means here like here for example if I get a request for a common resource and I acknowledge that in very next state in that case here acknowledgment holds good in very next state hence here I can then specify that X of acknowledgment.

So, that means here acknowledgment holds good in very next state. Sometime if some master raises a request and that request may be entertained after a while or granted after a while so in that case you have to wait until it is granted so that means here we are not very sure whether it would granted in the next cycle or next to next cycle or next to next cycle. So in that case here those properties can be specified as using a future operator.

So that means here the formula will hold good sometimes in the future not necessarily exactly in the next state or next to next state or next to next to next state but somewhere in the future. Then we have operator as until operator. Until operator says that say phi formula holds good until psi hold good.

(Refer Slide Time: 46:28)



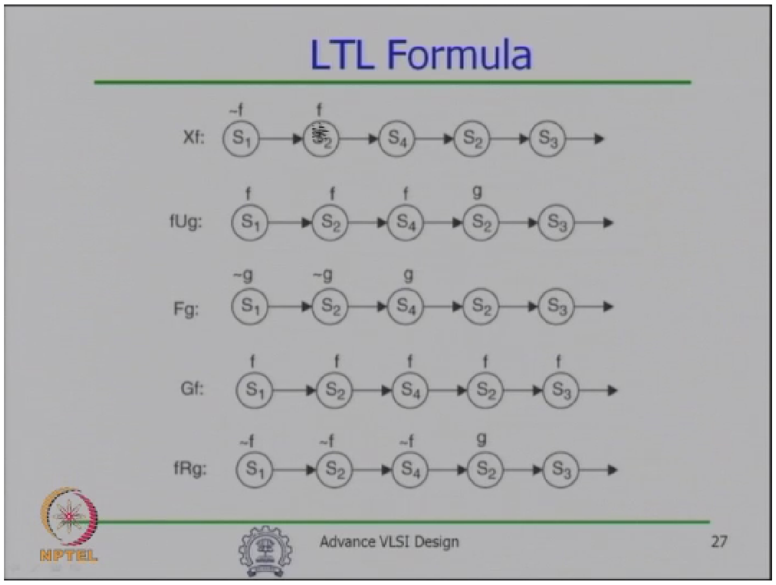
So what it says that if this is the state transition in that case here my formula phi should remain hold good until I reach to a formula psi. So psi should hold good. So that means here in this always here there must be some at least in one of the states psi should hold good. This is little bit strong and there is weaker version of that. That is weak version phi weak until psi what it says is that phi should hold good until psi arrives.

So, that means here this phi should hold good. Phi should hold good so one of the example could be your acknowledge sorry the request should remain active until it is granted. So, that means here at least at some point in time you will have grant that we can represent using the until. So that means here request hold good until it is granted. I can write that. But in some of the cases it need weaker version.

So that means here like either this phi should hold good until psi arrives or phi continue to hold good. Phi, phi, phi, phi infinite so that means here phi, psi never arrives in that case also this formula holds good this is weaker version of until and then there is another formula that is known as phi releases psi. So phi releases psi, this says that here when phi arrives, in that case here psi would be released.

So that means here you have psi psi psi and there would be at least one state in which here this phi releases psi This is the difference between psi and the release and until operator that here at least there should be one state in which both of the formula should hold good. After that we cannot argue about that. So now if now here if you look at the progression of this. This shows you that in computation path your x operator will say that here x of f means here in the very next state f should hold good.

(Refer Slide Time: 48:56)



LTL Formula

Xf: ~f S1 → f S2 → S4 → S2 → S3 →
fUg: f S1 → f S2 → f S4 → g S2 → S3 →
Fg: ~g S1 → ~g S2 → g S4 → S2 → S3 →
Gf: f S1 → f S2 → f S4 → f S2 → f S3 →
fRg: ~f S1 → ~f S2 → ~f S4 → g S2 → S3 →

Advance VLSI Design 27

And in this state f should not hold good f until g that means here f should keep hold true until g arrives. F of g will tell you that at some time in the future g will hold good. G is another operator that is global operator that means it says that in every state in this path your app should hold good and release operator will say you that here g releases path f. So with the definition of the syntax of LTL formula I complete my lecture here.

And I will continue with this in the next class wherein we will see how we can specify a transition system using this formula and then here how we can verify that using model checker. Thank you very much. Good day.