

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

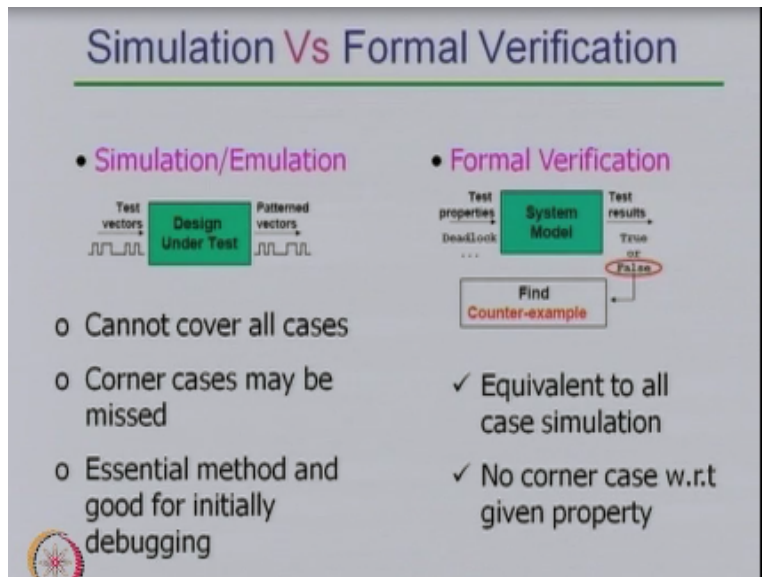
Lecture – 40

VLSI Design Verification: An Introduction

Hello. Welcome to the advance series of VLSI design course. I will take you through the VLSI design verification process in this portion of the lecture series. Last lecture we discussed about various techniques for the design verification. One was the simulation or emulation based technique which is widely used in the industry other technique is the formal verification. And we also discussed the difference between simulation based technique and formal verification based techniques.

As we discussed, in simulation based technique we apply some we applied some test factors to the input of the design.

(Refer Slide Time: 01:09)



So, this design is in the form of VSDL or Verilog code. And we checked the output of the design and we verify with respect to or check with respect to the given specification or golden reference output. If there is a match in that case we say, that most likely design is correct, but because we cannot apply test factors exhaustively and here. We can never say that this design has no bug. So, we can say only the presence of bug, but we can never say the absence of bug.

If we use a simulation or emulation based technique, simulation is pretty slow process. Maybe the speed is something like 1 to 5 hertz, whereas the emulation of the design some reprogrammable logic like a PGA and that is much faster than the simulation. It is roughly 5 to 6 order of magnitude faster. But here the main challenge is how to find out the corner cases which you can exercise and then test.

So, it is like corner cases can be missed and then here you may miss some of the design bugs. Simulation based verification technique is very good for design based debug. If you apply the random vectors, you are likely to capture some of the bugs. Whereas the formal verification technique uses some mathematical techniques to verify the design. So now here what you need to do is you have to supply the model or behavior of the design to the verifier and a set of properties.

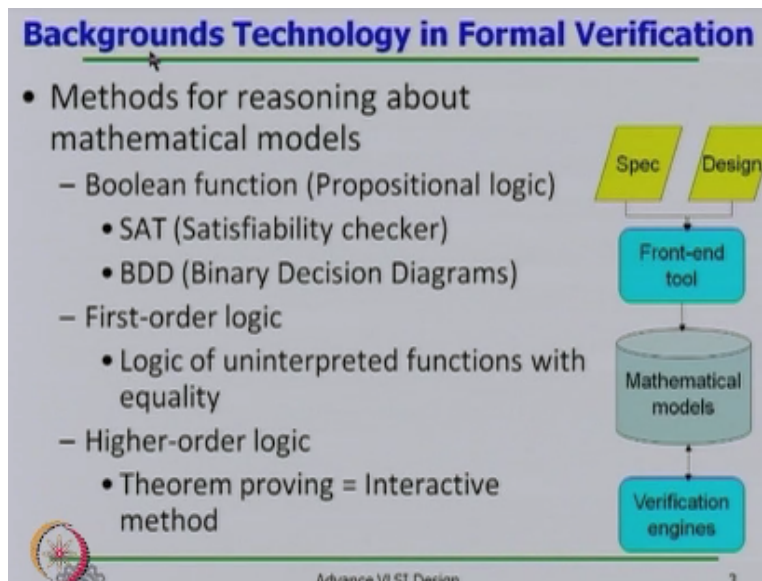
And then it will tell you whether these set of properties are always respected by this design or they fail if they fail. Then it gives you a trace of inputs and proves that it fails and that is known as counter example. So, like here say in order to note the behavior of the design, if it is a sequential circuit then it is a finite discrete machine and what are the properties. Properties are something like if we want to design a traffic light controller.

Then it is supposed to required not giving the green signal to both of the people crossing roads. So that is one of the property that we have to check whether there exists any state where in this can provide the green signal to both of the crossing roads. Other property could be like the signaling should follow some pattern. Like this red, yellow, green something like that. So now here, if the property passes in that case, we can say that design is correct.

And so, this is equivalent to all case simulation. If it is equivalent to all case simulation, in that case there is no concern about corner case which can be missed. say in simulation based verification, now there is no corner case with respect to the given property. Hence, we say that this is always correct for a given property. Now here the challenge is to find out the complete properties of a given design.

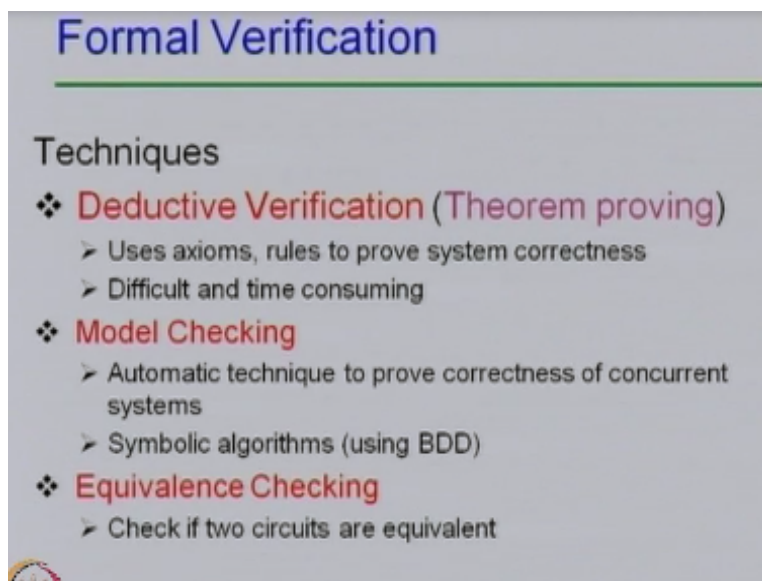
If you can verify for all the properties in that case you can say that, this design is correct in all respect and we can fairly rely on that.

(Refer Slide Time: 05:02)



So now here what are the various challenges we have. So, this technique as I said the formal technique is based on mathematical reasoning and for the mathematical reasoning here, we use the Boolean functions or Boolean Algebra. That is propositional logic and in order to manipulate this we either use satisfiability checker or binary decision diagrams or we also use first order logic or higher order logic for the theorem proving.

(Refer Slide Time: 05:36)



So, as I also mentioned in the last lecture that there are three kind of techniques we use in the formal verification one is deductive verification, deductive verification uses axioms and theorems to prove the correctness of the design, so it is like here, you prove this pythagoras theorem mathematically, you take a scale and measure both of the arms and verify that, so now we need to use a set of axioms, like if you want to verify an adder.

In that case here specification can be defined more abstract way like algebraic form, now add 2 bit 1, now what is the output, and now if you have bit stream what would be the value of interior value of the bit stream, adding 2-bit stream, what would be the outcome of that, so that is higher level of abstraction, you have implementation of your design and you want to verify, it gives you the same mathematical answer or not, so now deductive verification as it uses axioms.

And theorems, and then you have to use that in a certain way, so that you need manual intervention in this. This technique is semi-automatic technique and it is bit difficult and more time consuming. Another technique is called as model checking. In that you model your design like here if it is sequential circuit or a finite state machine and then you identify some of the properties and that you mathematically express and then you have verified those properties on those given model checker.

It is worth to mention that the inventor of model checker got touring award very recently. You can see the effectiveness of the technique. So now This technique is almost fully automatic. That means You just need to supply the model, say a Verilog or VSNL code. Say now your verifier will extract the finite state machine from Verilog or VSDL code and then you express properties in terms of some mathematical logic.

And now here your verifier will check whether this property is respected all the time or not. So now here the challenge is that the number of states are exploding and in order to complete the verification process in reasonable time here we use the symbolic algorithm like binary decision diagram to manipulate these systems. The third technique is equivalence checking. So, equivalence checking is like checking the equivalence of two designs.

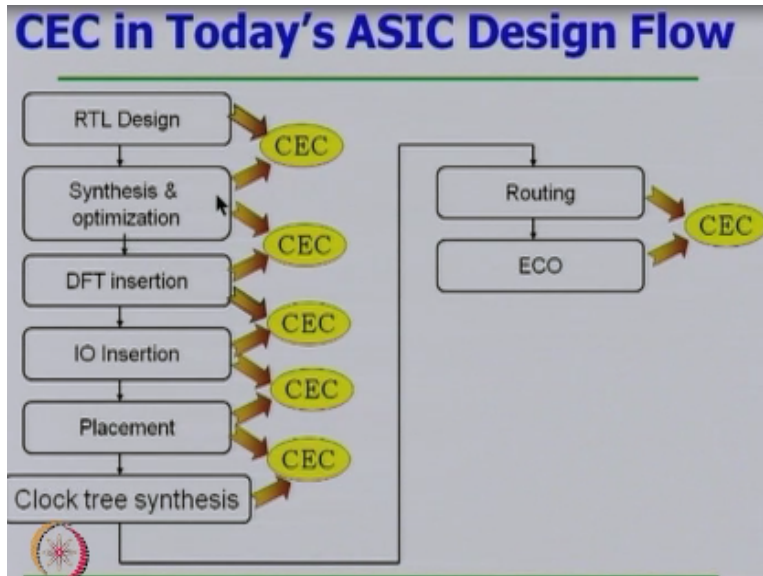
So, if you have two designs, so Both of the designs are supposed to produce the same output if you apply same primary input. So now here like for example, if you design an adder, that may be ripple carry adder it will be cost effective in terms of area. And it will be verified with respect to the given specification, that is mathematical specification you have. Maybe you are using the deductive verification technique.

And after that so now you have optimized that for say timing. So now you have another design say carry look ahead adder, which is more optimized for the timing. So, now you have one ripple carry adder and another say carry look ahead adder. Ripple carry adder is already verified. Now if you prove the equivalence of the carry look ahead adder and the ripple carry adder, in that case you can say that both of the designs are respecting the specification.

So, this equivalence checking again another technique and equivalence checking is purely automatic technique. So, model checking, and equivalence checking are the fairly automatic techniques. The deductive verification is a semi-automatic technique. Industry always prefers to have a technique that is fairly automatic technique, so that human error cannot be done using that. So first let me start from the equivalence checking.

So then again There are two parts. One is the combinational equivalence checking and sequential equivalence checking. So first I will start with the combinational equivalence checking and then we will consider the theorem, model or property checker. So today in the ASIC design flow if you look at the use of the combinational equivalence checking, then you will find almost everywhere that everywhere combinational equivalence checker is being used.

(Refer Slide Time: 10:49)



So, this design flow we discussed earlier. We start from the RTL design, your synthesis that you obtain to get the net level, obtained get net level is equal to the RTL design or not. Then you do the testability analysis inside the Design foot testability. So, after DFT insertion again you have to check if you get net level list is functional equal to the synthesized get level net list or not.

And after that you insert IO Then the placement routing. And then you introduce the clock tree. At every stage you have to check whether your new transform alter the functionality or not. So that means here Before transformation and after transformation, the functionality remains equivalent or not. So now you can see the use of equivalence checker in today's design.

(Refer Slide Time: 11:48)

Combinational Equivalence Checking (CEC)

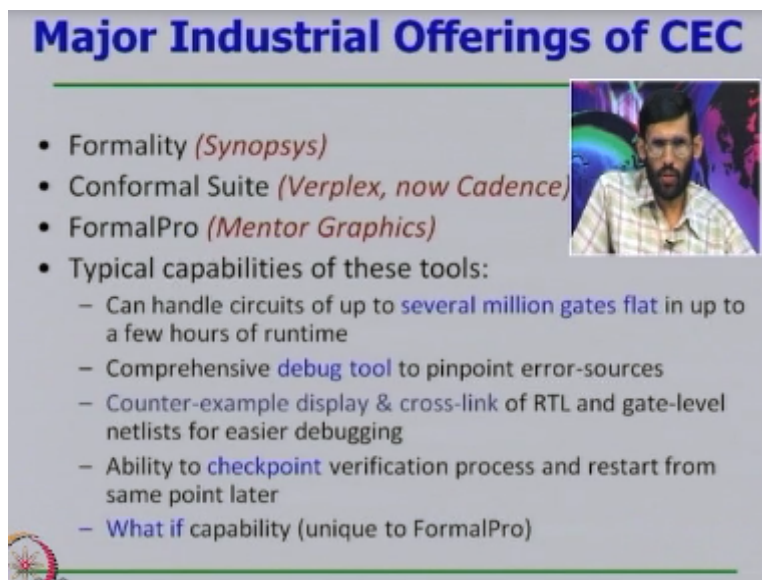
- Currently most practical and pervasive equivalence checking technology
- Nearly **full automation** possible
- Designs of up to several million gates verified in a few hours or minutes
- Hierarchical verification deployed
- Full chip verification possible
- **Key methodology:** Convert sequential equivalence checking to a CEC problem!
 - Match Latches & extract comb. portions for EC

As I said here this the combinational equivalence checker and pervasive technique we have. It can it is almost fully automatic technique. That means it is push button kind of job You say submit design and say now here check for the equivalence, it will tell you whether both of the designs are equivalent or not. If there are not equivalent in that case it will return you the trace or input vector that you can give or produce the different output from two different designs.

Now the combinational equivalence checker can also handle the millions of gates. So that means you can do almost the full chip verification, so that means here if you have millions of gates in one design and millions of gate in another design. And If you want to check whether both of the designs are equivalent or not, you can handle the full designs. Problems source Combinational equivalence checker is still a small portion.


Most of the circuits are sequential circuits and now when you go for the sequential equivalence checker. In that case here, your design space explodes in order to handle that one of the ways most of the people do use. You can formulate or convert the sequential equivalence checker problem as a combinational equivalence checker and you can use the conventional combinational equivalence checker to check the design.

(Refer Slide Time: 13:24)



Major Industrial Offerings of CEC

- Formality (*Synopsys*)
- Conformal Suite (*Verplex, now Cadence*)
- FormalPro (*Mentor Graphics*)
- Typical capabilities of these tools:
 - Can handle circuits of up to **several million gates flat** in up to a few hours of runtime
 - Comprehensive **debug tool** to pinpoint error-sources
 - Counter-example display & cross-link of RTL and gate-level netlists for easier debugging
 - Ability to **checkpoint** verification process and restart from same point later
 - **What if** capability (unique to FormalPro)

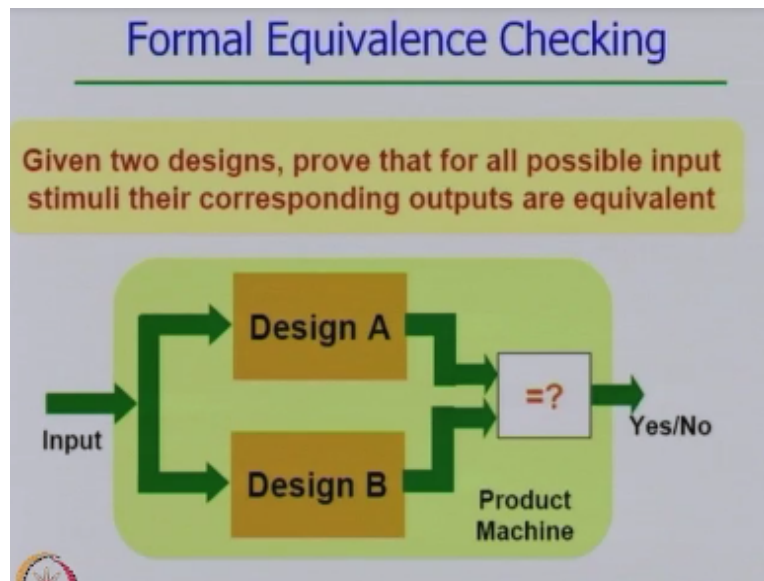


So, if you look at the current industry offering for the combinational equivalence checker, in that case you will see wide variety of tools are available like formality synopsis, conformal suite and

cadence, formal pro from Mentor graphics and these tools have the enormous capability so that means they can handle several million net lists. They have the comprehensive debug capability that means you can localize the bug.

That means here why these two designs are different, where the bug is. Some of the tools have the what if kind of capability as well.

(Refer Slide Time: 14:11)



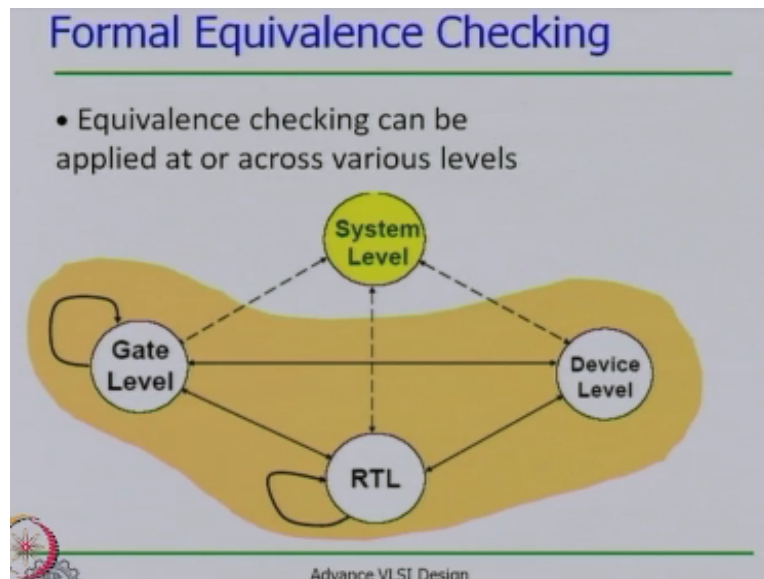
Now as we closely look at the combinational equivalence checker problem. What we want and how this being done. So, in this combinational equivalence checker you have two designs, design A and design B and when I apply some input to design A and design B here, these two designs are supposed to produce the same output. If they are producing different output in that case here, both of the designs are not equivalent.

Now the question is may be for some of the inputs two designs can be equivalent but for some other inputs two designs are not equivalent. So Now here, one of the ways is that need to find out, we need to check whether for all the possible inputs, both of the designs are producing the same output or not. Like here As I discussed in the last lecture, If I have OR gate and XOR gate, say if I want to implement an XOR gate and by mistake I have implemented OR gate.

Now here if I want to verify. In that case, 3 inputs out of 44 inputs for the 2 in out XOR gate or OR gate, both of the designs are producing the same output or not. And if I am checking for the three inputs, both the designs are equivalent. So, like 000110, OR and X-OR will produce the same output. So now the complexity is we need to check for all the input and as I mentioned that here we cannot exhaustively check that.

As this is the application of input and getting the output and checking whether the outputs are same or not. That is called as stimulation based verification. So now here, what we want is, we want to verify these two designs. That should give me the capability that both of the designs, it is like here for all case stimulations.

(Refer Slide Time: 16:25)



So now one of the good thing with equivalence checker is, that we have seen earlier as well that this can be applied at one level of abstraction or across the level of abstraction. So that means here you can check the gate level net list with respect to RTL. RTL with respect to gate level net list or RTL with respect to RTL or gate level net list with the same gate level net list, gate level net list with vc or transistor implement, so and so forth.

So That we have seen in every stage in the design transformation we use in use equivalence checker.

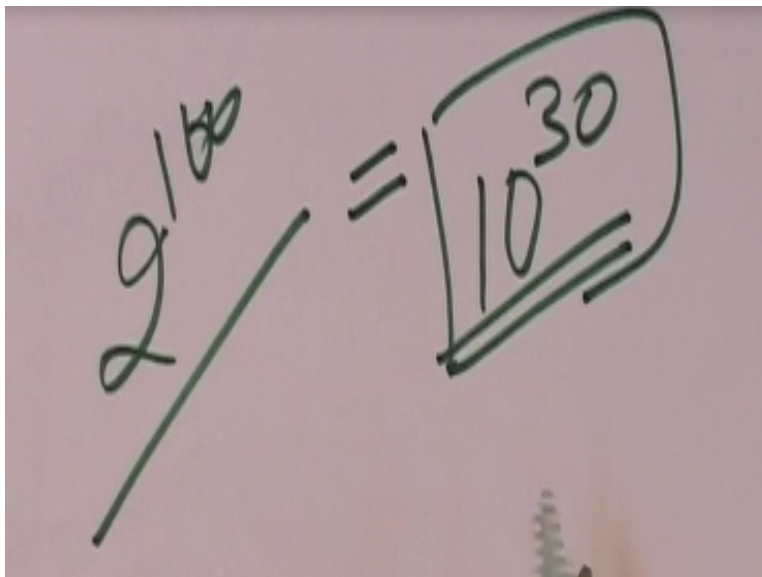
(Refer Slide Time: 17:00)



So, let us go a little bit more inside the theory of equivalence checker. So, if you have these two designs, design A and design B and I want to check, because these two designs look different. If you look at the structure, but now here the question is, if you want to check the correctness of these two designs or equivalence of these two designs. In that case If you have to say, for all possible inputs here output of these two designs, should be same in the same.

In that case we can say both the outputs have to be same. In that case, we can say both of the designs are same. So now one of the ways is that you generate a truth table and check the entry through the truth table. Say Assume I have 100 inputs to a design.

(Refer Slide Time: 17:54)



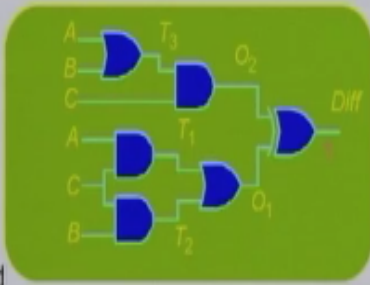
And then you need to have 2 raised to the power 100 entries in the truth table. So, first thing is you have to store that. And 2 raised to the power of 100 is roughly, 10 raised to the power of 30. That means here 10 raised to the power of 30 storage location at least you need, so that, you cannot fit your total truth table in your storage space and hence that is impractical. So now here you have to have some way that can represent, that can have unique representation of this circuit.

So that means, if both of the circuits are equivalent in that case, the unique representation must be same. And now I can check the unique representation. So, the other thing like here the truth table is unique but this is not manageable, but this is not compact. So, I want to have representation which is compact and easy to manipulate. So, these are the two requirements. And we will see that.

(Refer Slide Time: 18:58)

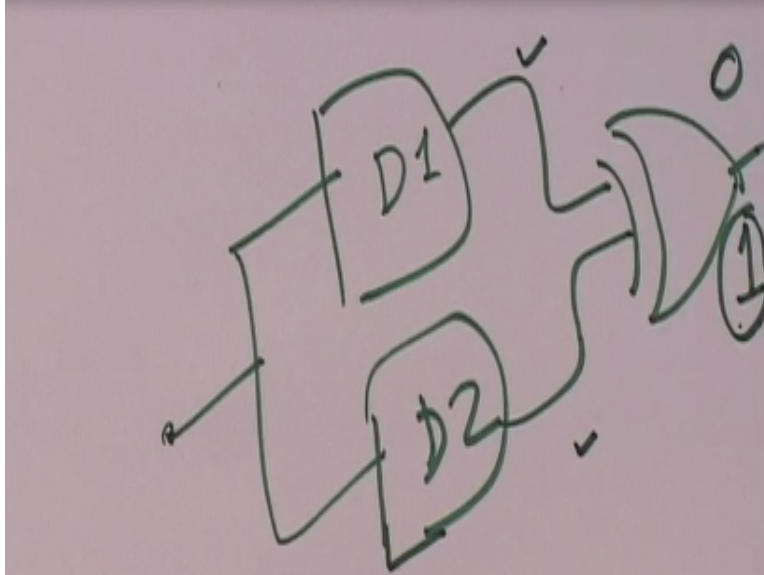
SAT/ATPG Based Equivalence Checking

- Satisfiability Formulation
 - Search for input assignment giving different outputs
- Branch & Bound
 - Assign input(s)
 - Propagate forced values
 - Backtrack when cannot succeed



So, binary representation diagram is one of them. So now here then there are two ways to verify the equivalence of these two designs. So, say This is one design and this is another design.

(Refer Slide Time: 19:18)



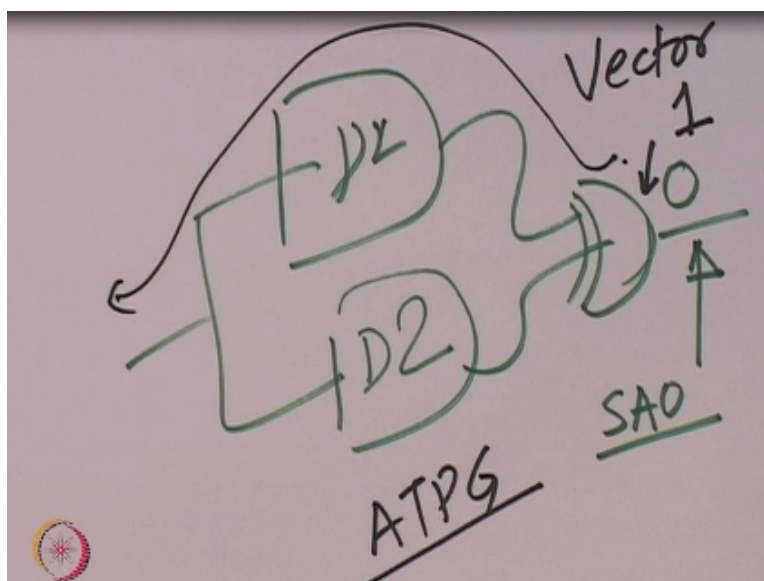
One of the way is you have design A, you have design 2 and then you need to check both of the designs. So now here If I say I XOR the output of these two output, all possible outputs should always remain 0. So, for all possible outputs it should always remain 0 here, if I XOR these two Now, if I find out some way, some set of input for which the output of XOR becomes 1, I can say that both of the designs are not equivalent.

And that input is known as counter example, for that input both of the designs are producing different output. Now how I can do that. So One of the technique is these are the SAT based technique or ATPG based technique in SAT based technique we search an input assignment that can give different output for these two different designs.

(Refer Slide Time: 20:26)

Now SAT source says 3 set problem, will be a complete problem, now you need to use juristic in order to. You need to solve this in reasonable time. You can use branch and bound. In that you can assign some input, you propagate those input and check whether you are able to get the output or not. If there is a conflict in that case, you have to back track until you succeed. The same way here you can use ATPG.

(Refer Slide Time: 20:58)



And what ATPG will do is, like here this is design 1 and design 2 and now here what do you want. For a given input, output is always 0. So, this corresponds to output is always struck to 0. So now here I need to test and generate a vector that can produce output 1 here. If I can produce

output 1 here, in that case, I can distinguish the behavior of two designs. So now I can use the automatic test pattern generation techniques.

There are various techniques like D algorithm, podem algorithm, fine algorithm, in this series of lecture, I cover ATPG algorithm in VLSI test lecture series . So, you can refer to that. I do not want to repeat that. So now you want to find out a set of input that can produce, output 1 here. If you can obtain that set of vector in that case here, you can say that the both of the designs that means one set of vectors that can produce that can produce different output from two different designs.

Hence these two designs are not equivalent, so whether you formulate this as an SAT problem or whether as an ATPG problem. In both the cases, the challenge is finding out the assignment is going to be complete problems. Hence here you need to exercise lot of inputs before saying. See, generation of test vector is easier. But here we are saying that this test factor does not exist. And this is the most difficult problem and now, here so in this case, when both of the designs.

Design A and design B are equivalent in that case here, there should not be any test vector, that produces output 1 here. So now here in order to prove that here typically, you need to explore the significant portion of input space. So that means here the timing complexity is exponential. So now one of the way is as I mentioned earlier, we can use either SAT vector or ATPG based technique.

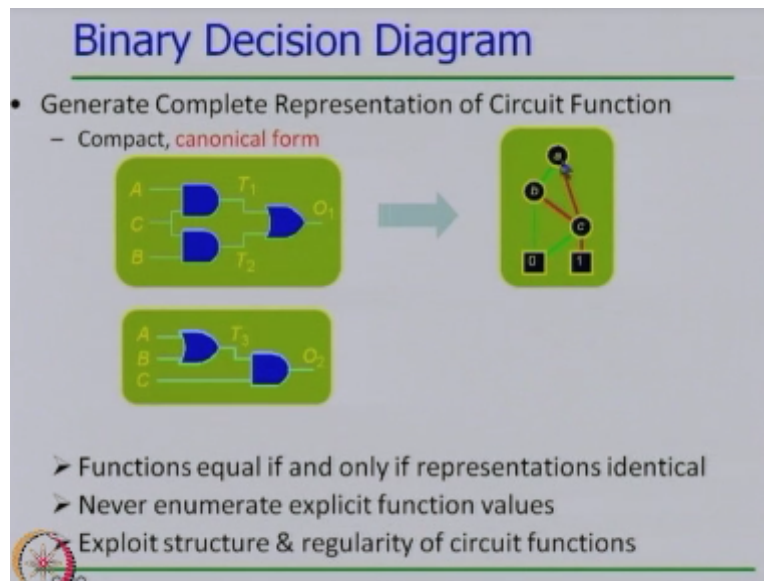
In that you want to find out a set of input that can produce two different outputs from two different designs. If there exists such vector, then we can say both of the designs are not equivalent otherwise we say that both of the designs are absolutely equivalent so that is the challenge. Other approach is the functional approach. As I explained earlier as well, that if you can represent the functionality of your design by unique representation.

So, as I said truth table is one of the unique representation but it is not economical in terms of storage space. So now here, we need to have a unique representation that is compact. And now here, if both of the designs are functionally equivalent I can have the unique representation of

both designs and then I can compare. So, if there is match in that case, we can say both of the designs are equivalent or both of the designs are not equivalent.

So now here functional approach, I will discuss little bit more in detail. So now we say these are the two designs design T1 and design T2. If these two designs are absolutely equivalent and in that case, they are unique representation or canonical form of representation should be identical.

(Refer Slide Time: 25:04)



And that should be compact and binary decision diagram is one of the compact way of representing. That is the graphical representation of a truth table. So now this is the binary decision diagram, where in you have the inputs. And now here based on say, A is 1, B is 1, in that case here output will be 0. And in case if A is 1 and B is 0, C is 0, then output would be 1. So, now in this way I can read, from here I can generate the binary decision diagram for this circuit.

I generate the binary decision diagram for this circuit than here, I can match these two graphs. So, this is graphical representation, I have to match these two graphs, how I match these two graphs, I have to match node by node and edge by edge. Now here it has five nodes and five edges and here it has five nodes and five edges. If I match these nodes and edges, they are equivalent in that case I can say that both of the designs are equivalent.

So, if we can say functions equal if and only if representations are identical. In this we are never enumerating the explicit function values. We are thus generating a compact representation and then we are comparing that. We also explore the structure and regularity in the structure.

(Refer Slide Time: 26:29)

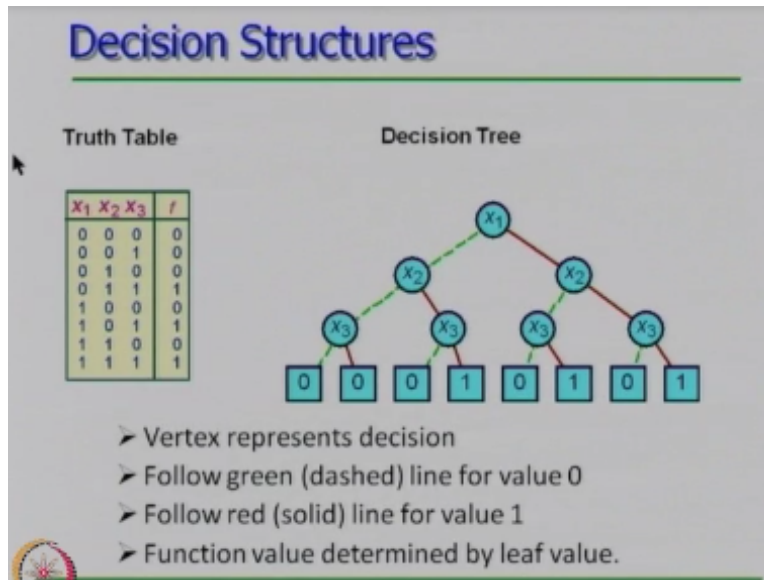
$$\begin{aligned}
 & f(x_1, x_2, \dots, x_i, \dots, x_n) \\
 &= x_i \cdot f(x_1, \dots, x_i=1, \dots, x_n) \\
 &+ \bar{x}_i \cdot f(x_1, \dots, x_i=0, \dots, x_n) \\
 &f = x_i (f_{x_i}) + \bar{x}_i (f_{\bar{x}_i})
 \end{aligned}$$

So here now let us come to this point, how we obtain the binary decision diagram. So, As I said binary decision diagram is a graphical representation of a circuit or Boolean function. So Now here say my Boolean function is function some input of my f of x_1, x_2, x_i and x_n , now this function I can use Shannon composition theorem, I can this function is evaluated to 0 or 1 based on the value on these variables x_1 to x_n .

So say I decomposed this function with respect to some variable. So, let us say x_i . So Now here I can write this using the Shannon expansion theorem. x_i into f of all the variable, with x_i equal to 1 and x_n . So, this is the evaluation of function when x_i is equal to 1 + x_i bar, because x_i can also take the value 0, f of $x_1, x_i = 0, x_n$. So now I can represent, this is the value of function when x_i is equal to 1.

So, I can represent this function as $x_i f$ of x_i + x_i bar f of x_i bar +. So, f of x_i is the value of function when x_i is equal to 1 and f of x_i bar is the value of function when x_i is equal to 0.

(Refer Slide Time: 28:11)



So this way if I can represent my function here I can generate a graph, where x_i is in one node and so x_i is my node and now here this is my function f of x_i . This will evaluate to this one, this is my function f of x_i . When I get 1 it will go to evaluate this one. And 1 it will go to evaluate this function f of x_i , now here again this function will have whole variable x_1, x_2, x_n except x_i . In the same way, it will have all variable x_1 to x_n except x_i .

Now again, I can decompose that with respect to another variable say x_j . So now here again I can say this is x_j , and I will find out a function that is $x_i x_j$, and then here would be a function of x_i and x_j . And this way if I keep on expanding this and I will get the binary decision tree. So, this would be binary decision tree, from every node there can be two branches. So how this binary decision tree is generated.

Let us say this is my function, that is represented by this truth table. Now it has three variable x_1, x_2, x_3, x_n , so I start from a variable x_i and I assume that I evaluate my function in given order of variable and that order of variable say I decided x_1, x_2, x_3 . So now here if I have x_1 in that case here, either x_1 can be 1 or 0. If x_1 is 1 in that case here, this would be the function and if x_1 is 0 this would be the function.

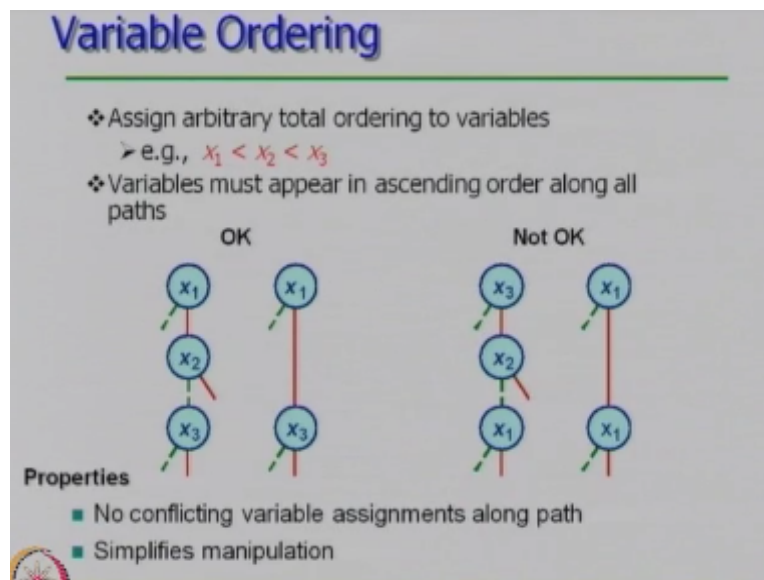
Now here, if you evaluate with respect to x_2 , then here you will have x_2 is equal to 1, this would be the function when x_1, x_2 is equal to 0 then this would be the function. Now here again you

have only one variable and evaluate with respect to x_3 . So now when you have x_1 equal to 1, x_2 is equal to 1, x_3 is equal to 1, then the output is zero it is corresponding to this entry. So now here, every path in this binary decision tree represents one entry in the truth table.

Now you can say that what we achieve out of this, here we have number of paths equal to the same number of entry in the truth table. And for the large number of inputs, it is not possible to store these things. Yes, that is true. But there are some ways to minimize this, Now I will discuss those how we can minimize. So, in this truth table, you have only 8 entries. Whereas if you see here you have $8 + 4 + 2 + 1$, total 15 nodes.

Right, and 15 nodes and 15 edges. So now here your binary decision tree is much more bigger than your truth table. Now here the question is how you can minimize that.

(Refer Slide Time: 31:14)

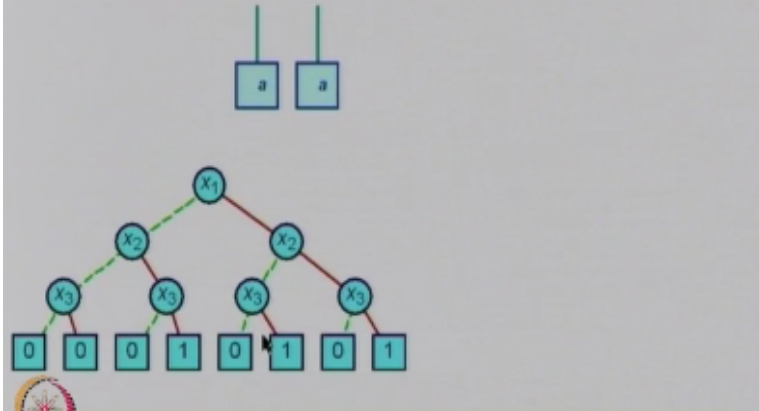


So, there are a couple of properties for binary decision tree, but here in every path, the variable should be followed in some particular order. There cannot be, somewhere you have $x_1 x_2 x_3$, somewhere you have $x_1 x_3 x_2$. So, they should follow some order. So, this order x_1, x_2, x_3 is okay, x_1, x_3 is okay, but these cannot be $x_1 x_3 x_2$. So, this order is not correct. No conflicting variable assignment can be there. And now here Binary decision diagrams are easy to manipulate.

(Refer Slide Time: 31:58)

Reduction Rule #1

Merge equivalent leaves

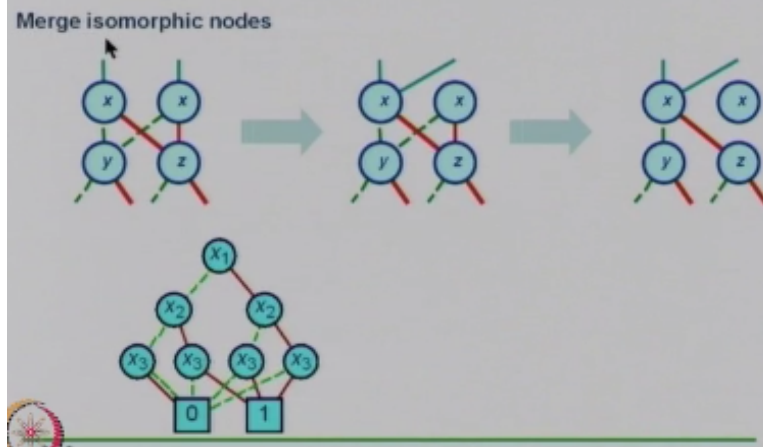


So now the question is how to minimize that. You know that there is a lot of redundancy in this binary decision tree. There are a couple of nodes, which are restoring the same value. Like if you look at the leaves node, they are storing either 0 or 1, but there are totally 8 number of nodes and they are storing only 2 values. So, if there are two nodes, in that case, there are no point to store these 2 values again and again in redundant fashion. So, you can reduce that.

So now one of the way is that you merge the equivalent leaves nodes. Leaves nodes are nodes which are always restoring 0 or 1 value. Right So now if you merge these two in that case, there is a significant reduction in the binary decision tree. Now this becomes a binary decision diagram or binary decision graph. So here it has 15 nodes and you reduce that you 9 nodes.

(Refer Slide Time: 32:56)

Reduction Rule #2



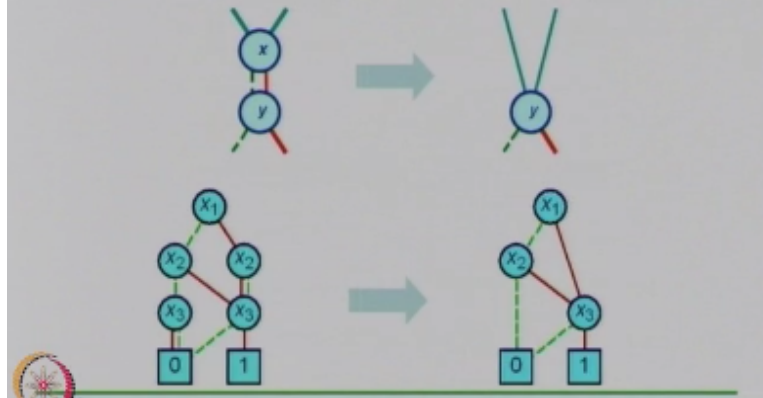
Again, there are a couple of nodes, which have the same binary decision tree below that node. So, like here if you look at this node x and this node x , here both have the similar kind of binary decision trees in both of the branches. So now here there is no point to store these two nodes differently. So, what if we can merge these two nodes. So, what you need to do is, you need to bring in, this branch here.

So, when you merge these two here, so this node has no role to do and hence I can remove this node. So, in this diagram if you look at these are the two nodes, say x_1 and this is x_2 . This x_3 . This x_3 and this x_3 These three nodes are evaluating in the same phase. These nodes are called as isomorphic nodes. And you merge isomorphic nodes, so here once you merge isomorphic nodes you can again reduce that.

(Refer Slide Time: 34:03)

Reduction Rule #3

Eliminate Redundant Tests

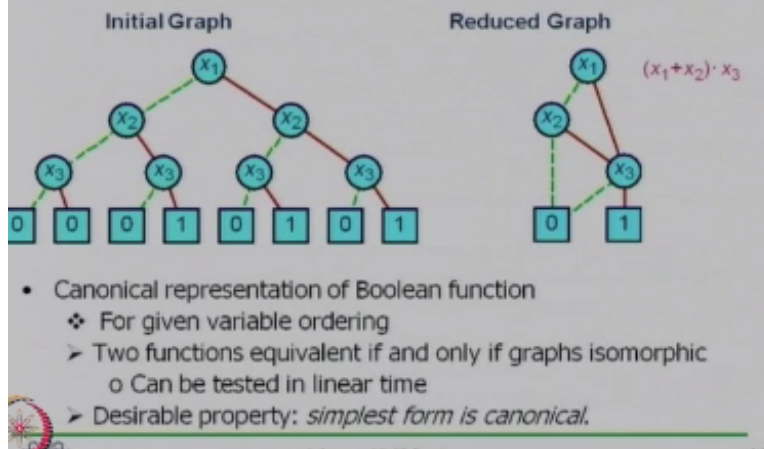


Now again look at whether there is a possibility to reduce it further or not. So now if you look at there is a still possibility to reduce this further, now if you look at these two nodes, when both of the edges are going to the same node, what does that mean? That means that irrespective of what is the value of x , your function will evaluate in the same way. So that means x does not have any role in decision making process.

Hence why I need to x in the diagram itself. I can remove that x . So now you remove that x and bring in these in edges to x to y . So now here in this diagram if you look at that now this x_3 to 0, both the edges going to that and from here x_2 to 0, here you have both of the edges going through that. So now here, you can see that, you can minimize, this binary decision diagram Binary decision tree which initially had 15 nodes and 15 edges to like here 5 nodes and 5 edges.

(Refer Slide Time: 35:16)

Example OBDD



So, there is a significant reduction from this binary decision diagram to this binary decision tree. So, this is known as reduced because here we applied all the three reduction techniques. So, this is reduced diagram. And we also decided the order of variables. This is known as reduced order binary decision diagram. And Reduced order binary reduced diagram always represents a canonical form of a Boolean one. That is the simplest form.

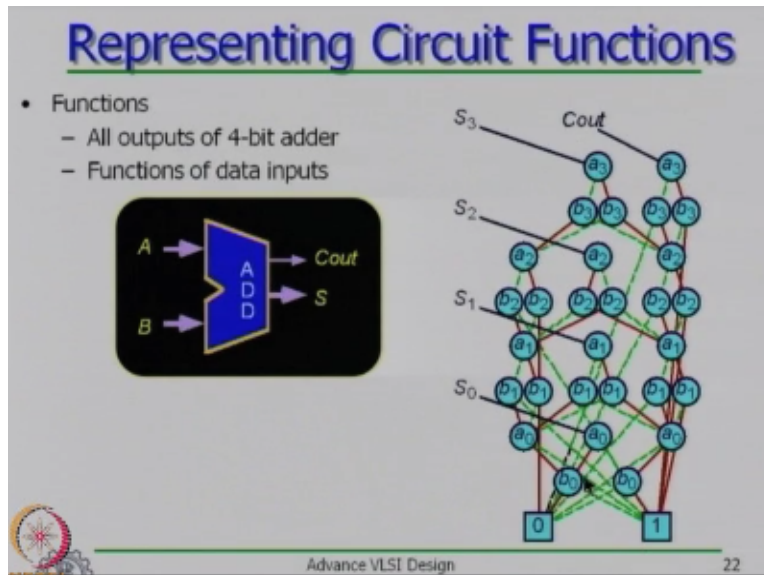
So, another thing is, because here if you change the order of variable, in that case here the shape and size of the binary decision diagram will change, but still this will give you the same unique representation for a given function or given circuit that will have the same shape and size for a given order of variable.

(Refer Slide Time: 36:20)



If you look at some function, like here it has constant and only one node. If it is variable in that case, you will have this kind of node. If it is typical function, then you will have this kind say some generate circuit. If it something like a symmetrical function, in that case your binary decision diagram will also be a symmetrical one. So, these are couple of examples.

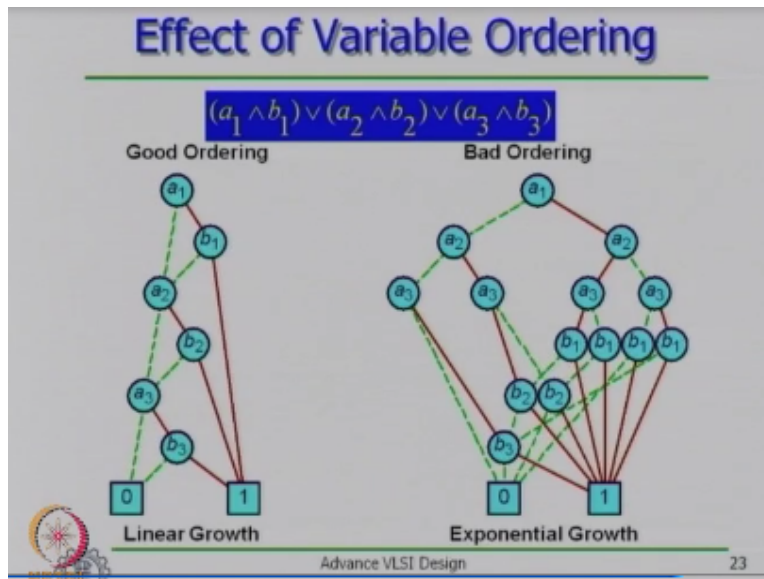
(Refer Slide Time: 36:45)



If you have say a circuit like here, 4 input adder, then in that case, 4 and 4, 8 input and output will be 4 bits and some output, and 1 bit carry 4 or five outputs. So, if you generate a binary decision diagram in that case, your binary decision diagram will look like this one. Say 4 input, it has 31 nodes. And If you go for 64 inputs, the number of nodes you will have is 571.

So now with respect to the input the graph is a linear one.

(Refer Slide Time: 37:20)



As I said that here shape and size depends on the order of variables. So now here say for this function, here if I take the order of variable as $a_1 b_1 a_2 b_2 a_3 b_3$, in that case here this will be the reduced order binary decision diagram. If you use the $a_1 a_2 a_3 b_1 b_2 b_3$ as the order in that case you can see, the reduced order binary decision diagram would be much fatter, and it will have more number of nodes and edges.

Now here if you see, what you want, when you want to generate OBDD, that represents canonical form and after that if you need to generate for one design and another design and you want to compare node by node age by age, so now what you want, is that the number of nodes and number of ages should be as small as possible. 0 nodes e by one will take ages. So that it will take less time in comparison and it will take less space in order to store these nodes.

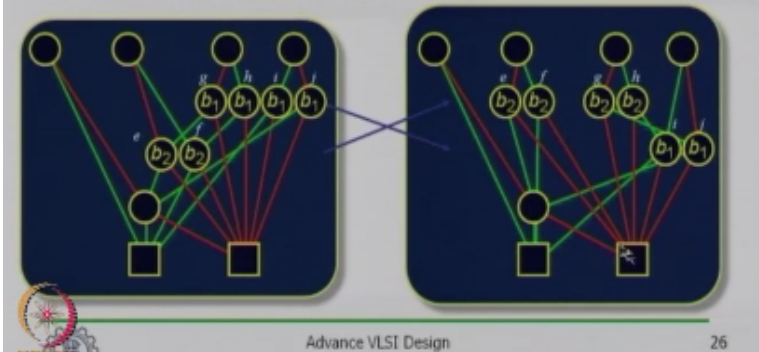
So now it is very important to find out a good order of variable and finding out the good order of variable is an interactive problem. And hence here we need to use couple of heuristics to find out the good order of variable. There are various good heuristics. Those are like here, static heuristics that you can apply using the fan in of the gate hysterically or can apply the dynamic ordering.

(Refer Slide Time: 38:10)

Swapping Adjacent Variables

❖ Localized Effect

- Add / delete / alter only nodes labeled by swapping variables
- Do not change any incoming pointers



So, I will discuss here Dynamic ordering which is being often used in the practical circuit. So, one of the way here is that dynamically when you are generating the ROBDD, what you need to do is, you have to check the swap the adjacent variables and check whether the ROBDD is getting reduced or not. So that it can come out from the localized effect. So, you add , delete or alter, only the nodes which are labeled to swapping.

Now if I want to swap the variable b_1 and b_2 , in that case here all the nodes which are labeled by b_1 and b_2 , here there can be some new nodes can be introduced or some nodes can be deleted. And now, if there is a reduction in the size, here we keep that order otherwise here we keep the previous order. So now by swapping here These two here you can see the difference in the ROBDD. If there is reduction we keep the newer order, otherwise we go back to the previous order.

(Refer Slide Time: 40:15)

Dynamic Variable Reordering

Richard Rudell, Synopsys

- Periodically Attempt to Improve Ordering for All BDDs

- ❖ Part of garbage collection
- ❖ Move each variable through ordering to find its best location

- Has Proved Very Successful

- ❖ Time consuming but effective
- ❖ Especially for sequential circuit analysis

Other techniques were like here shifting. Which was Proposed by Richard Rudell from the Synopsys. So, what it does is that, it periodically attempts to improve ordering of the variables, so now here this move each variable from BDD the given location to all other locations and find out the best place of that variable. Best place is the place where it results to the reduced ROBDD. Though this is very time consuming, but it is a very effective technique.

So how it works, say now this is my ROBDD. Now I want to find out the best place for variable B1, so what I will do is I will start shifting this from b1 to b3 to this side, b1 and a1 to this side and I will find out the place where it results into the smallest BDD and I will fix it there. So, this is greedy approach. So, I will fix it there and then I will work with the other variables and I would not change the order of these variables.

So, if you move to the next location, shift to the next location, in that case here, this would be the BDD. Then if you shift to the third position B3, in that case here this would be the BDD, then you go up you move to a3, then a2, then a1. So now here when you go to A2 in that case here, mean this would be the BDD. And then if you go to A1 then this would be the BDD, so now this two BDD s are the smallest ones.

So now you will fix the location or order of A2 at second look position or at the first position and once it is fixed, it is frozen, you are not allowed to change that. So now here this way you can

find out the order of variable. So now you are finding out optimal order of variable is very difficult that can be complete problem.

(Refer Slide Time: 42:20)

ROBDD sizes & variable ordering

- Bad News ☐
 - Finding optimal variable ordering NP-Hard
 - Some functions have exponential BDD size for all orders e.g. multiplier
- Good News ☐
 - Many functions/tasks have reasonable size ROBDDs
 - Algorithms remain practical up to 500,000 node OBDDs
 - Heuristic ordering methods generally satisfactory

But now here, most of the function, you can find out ROBDD in reasonable time. And algorithm remains practical up to say 1 million nodes in ordered binary decision diagram.

(Refer Slide Time: 42:30)

Sample Function Classes

Function Class	Best	Worst	Ordering Sensitivity
ALU (Add/Sub)	linear	exponential	High
Symmetric	linear	quadratic	None
Multiplication	exponential	exponential	Low

❖ General Experience

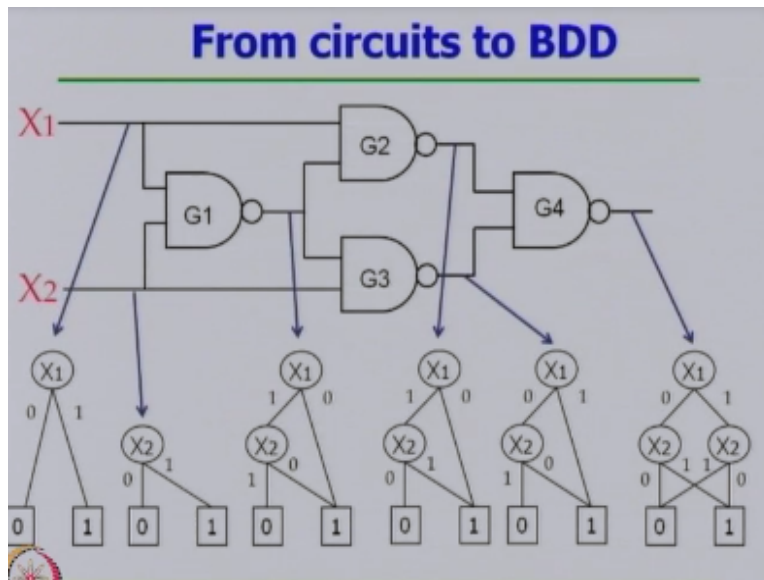
- Many tasks have reasonable OBDD representations
- Algorithms remain practical for up to 500,000 node OBDDs
- Heuristic ordering methods generally satisfactory

If you look at how the functions are changing or sensitive to order of variable. Like here ALU it has very high sensitivity in the best case you will have with respect increase in the variable, there can be linear increase or there can be an exponential increase in the shape and size of ROBDD.

Whereas symmetrical functions are not sensitive at all means it is a matter of what variable you are choosing.

For multiplication though here, the sensitivity is low, but the growth is always exponential, so with respect to increase in the number of inputs.

(Refer Slide Time: 43:15)



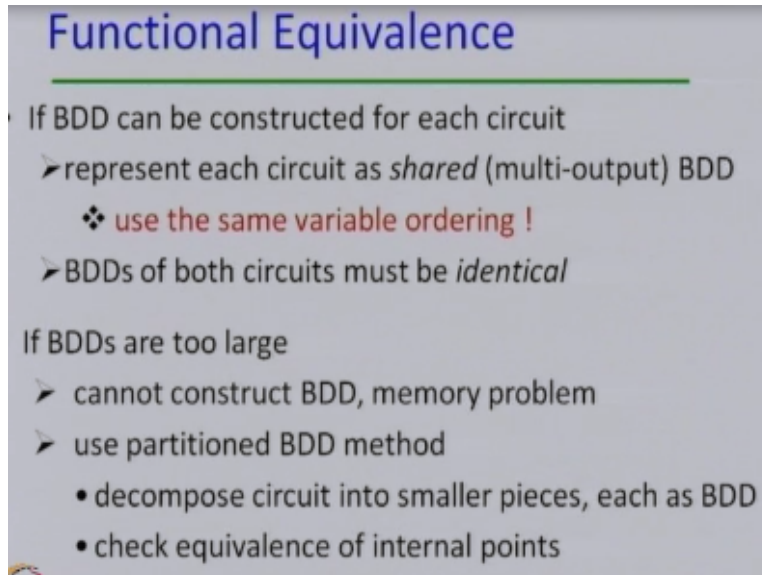
This is the story I told you how to construct ROBDD. so from a given truth table no from the truth table I told you that how you can construct the binary decision tree and minimize that and get the ROBDD, but the size of binary decision tree must be bigger than the truth table. That is impractical. So, this was just to illustrate you how to do that. In practical you do not do like that.

In practice say this is circuit, this is NAND implementation of XOR gate, in practice here we apply function on ROBDD and we generate the BDD by graphical manipulation or mathematical manipulation on these BDD s. So now we say here the BDD of X1 is one variable, we know the variable of x2, we know the BDD of this. Another variable we know the BDD of this, now if you want to have BDD at the output.

In that case you apply NAND function on that and generate the BDD. So now here just by graphical manipulation we obtain this one. And then again you apply the graphical manipulation

and finally you will get this BDD, and now here dynamically we here keep on applying the dynamic variable ordering restrict the explosion of BDD to certain extent.

(Refer Slide Time: 44:45)



Functional Equivalence

- If BDD can be constructed for each circuit
 - represent each circuit as *shared* (multi-output) BDD
 - ❖ use the same variable ordering !
 - BDDs of both circuits must be *identical*
- If BDDs are too large
 - cannot construct BDD, memory problem
 - use partitioned BDD method
 - decompose circuit into smaller pieces, each as BDD
 - check equivalence of internal points

So Now if you have BDD, mean say, if you have function or design to generate BDD, then you can compare these two BDD s. If they are equivalent in that case, you can say that both of the circuits are. We cannot handle BDD s more than a million nodes or so. So now here if this circuit is much bigger, what to do with that. We cannot even store the BDD. So now what You need to do is you need to partition BDD.

And so that means for that you need to decompose your function into smaller pieces and generate the BDD for that and compare these BDD at some internal points.

(Refer Slide Time: 45:36)

CEC in Practice

Key observation: The circuit being verified usually have a number of internal equivalent functions

To prove $f(i_1, i_2, \dots, i_n) = f'(i_1, i_2, \dots, i_n)$

Check

$$x_1(i_1, i_2, \dots, i_n) = x_1'(i_1, i_2, \dots, i_n)$$

$$x_2(i_1, i_2, \dots, i_n) = x_2'(i_1, i_2, \dots, i_n)$$

$$\vdots$$

$$x_k(i_1, i_2, \dots, i_n) = x_k'(i_1, i_2, \dots, i_n)$$

$$f(x_1, x_2, \dots, x_k) = f'(x_1', x_2', \dots, x_k')$$

So now if this is your design f and this is another design f' , you find these internal equivalent points and then you have to generate BDD for all these internal points and compare that.

(Refer Slide Time: 45:50)

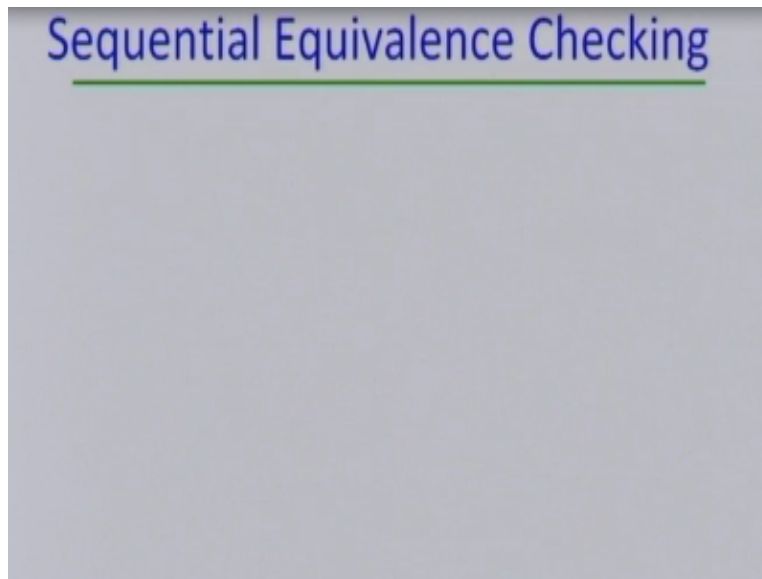
Functional Decomposition

- Decompose each function into *functional blocks*
 - represent each block as a BDD (*partitioned BDD method*)
 - define *cut-points* (z)
 - verify equivalence of blocks at cut-points
 - starting at primary inputs

Like these are the two functions one is f and g . Now you are decomposing the sub functions f_1f_2 and g_1g_2 . and now here you have to compare the intermediate Z and then you compare f and g , now you have to define the cut points, these are the structurally equivalent cut points and then you have to verify that this z and z are equal. So that means here this F_1 and c_1 are equal. And then here in terms of you generate BDD of two in terms of z and y .

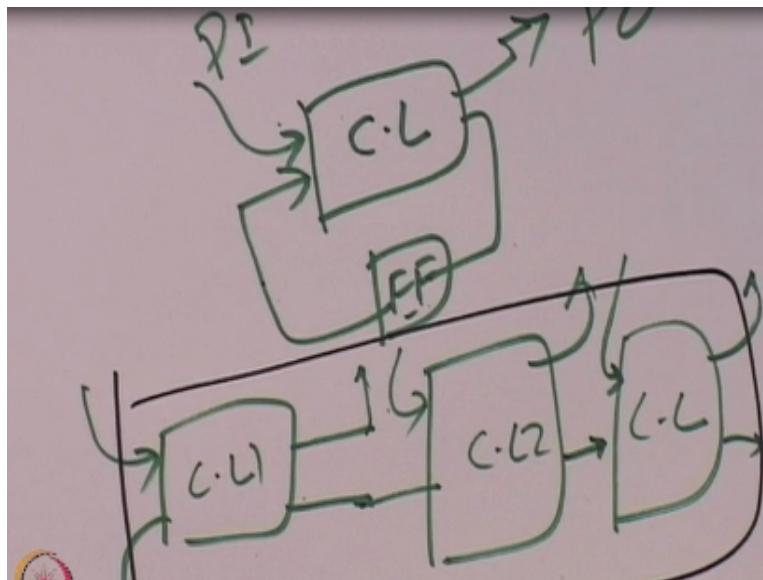
And here you also generate the BDD for g_2 in terms of z and y and then compare that.

(Refer Slide Time: 46:37)



So, this way you can verify the combinational logic. Now generally in reality we do not have combinational circuit we have sequential circuits. And now how to verify the sequential circuit That is the biggest challenge. And one of the way is you have to formulate that sequential equivalent checking problem into combinational equivalent checking.

(Refer Slide Time: 47:04)



How I can do that, say this is my sequential circuit that has some combinational part and some flip flop to store the state variable and there are some input and some primary output. This is primary output, and this is primary input, How I can convert this circuit into equivalent and

combinational logic. This circuit will behave differently in different cycles. So now if I can convert the entire circuit its behavior as a combinational logic.

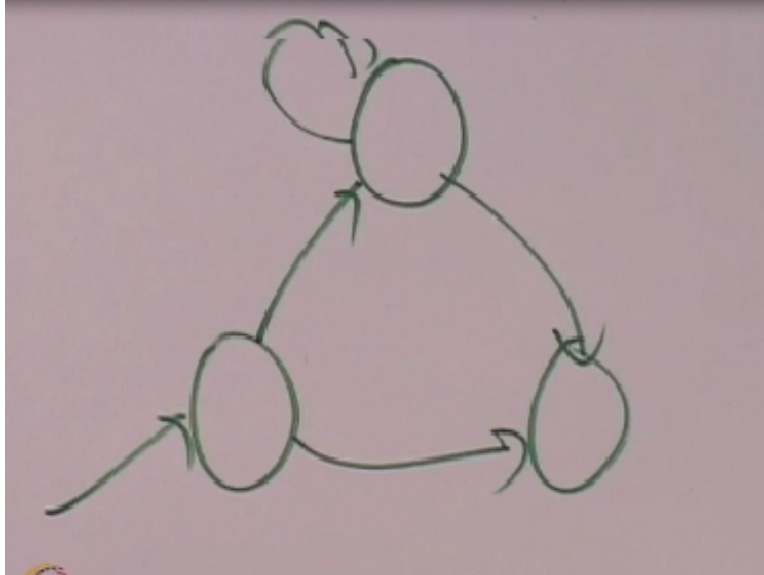
So initially say the flip flop may have some value. Then This is I am getting some value from the flip flop, primary input and this will behave, and this will produce some output which will go to the flip flop. Now next time this output generated by this one here it will go to the same combinational logic as it is. This would be the output and you will get new output. And then again, the output produced by this, again it goes to the combinational logic.

Right, and it will produce some output and will get some input from the primary input. Now if I expand this more couple of time, you can say that this is equivalent to the combinational circuit that is represented here for given number of time frames. So Now here this is equivalent functional equivalent of sequential circuit for given number of time frames. Now If you expand this using the time frame expansion into combinational logic.

Then your combinational logic is replicated n times if you have expanded this n times. And now the complexity of this combinational logic will be n times and number of inputs will also be increased by n times. So now your combinational logic would be much bigger, but still you can use the combinational equivalence checking for that. The only problem is that the circuit is too large.

And now this can give you guarantee up to certain number of time frames, it cannot give you guarantee infinitely long sequences. Other way to check the sequential logic is you have say;

(Refer Slide Time: 49:28)



Generally, we express sequential circuit or we can model sequential circuit using state diagrams. So, say this might be the state diagram of a combinational or a sequential circuit. So, if you have state diagrams of two machines, now you want to these two Machines are equivalent in that case, they must have isomorphic state transition graph. state transition graph is just simply finite state machine.

So now what you need to do is, you have to check whether the two machines are isomorphic or not.

(Refer Slide Time: 50:10)

Sequential Verification

- Approach 2: Based on isomorphism of state transition graphs
 - two machines M1, M2 are *equivalent* if their state transition graphs (STGs) are *isomorphic*
 - perform state minimization of each machine
 - check if STG(M1) and STG(M2) are isomorphic

M1

State min.

M1_{min}

M2

Advance VLSI Design
16

Say you have one machine as this one, another machine as this one. Say M1 and M2. These are not isomorphic. So, means by looking at these you can say that these are not isomorphic and been you can say that these machines are not equivalent: so that is not the case. So, what you can do is, there are state minimization technique. You must have studied that. So now you can apply the state minimization technique.

You can reduce this state diagram into this machine and now if you look at the reduced machine M1, minimized machine M1 and machine M2, in that case now these two machines are isomorphic. You can compare node by node. Edge by edge. So now the machines are isomorphic if you can generate another machine by simply relabeling the states. So, if I can relabel these state 1,2, because this is obtained by merger of these two states.

As one in the case both of them are absolutely equivalent and hence you can say these two machines are equivalent. This is another way to prove that. So, what you need to do is, you have to reduce both of the state machines and then you check the isomorphism of the reduced state machines of two designs. So today we discussed about combinational equivalence checking and how we can use combinational equivalence checking for sequential circuits as well.

And a couple of techniques we discussed like here time frame expansion model as well as the isomorphism of two state machines. And we have also seen that how we can use binary decision diagrams to check the equivalence. So, with this I complete today's lecture here and will discuss the remaining portion of equivalence checking and property checking or model checking in the next lecture. Thank you very much. Good day.