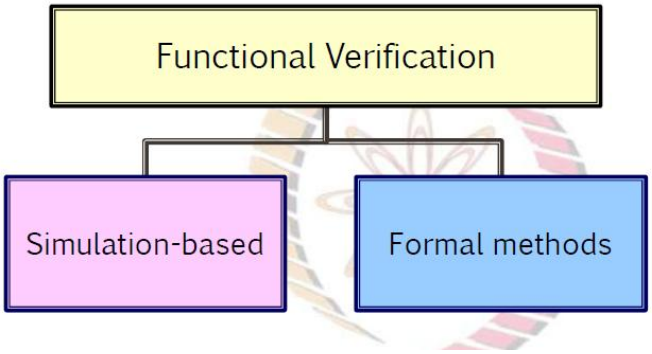
VLSI Design Flow: RTL to GDS

Dr. Sneh Saurabh
Department of Electronics and Communication Engineering
IIIT-Delhi

Lecture 21
Formal Verification-I

Hello everybody, welcome to the course VLSI design flow: RTL to GDS. This is the 17th lecture. In this lecture, we will be looking at formal verification. In the earlier lectures, we have seen that a given design gets transformed multiple times in a design flow. Now, as a design gets transformed, we want that the functionality of a design is retained or preserved. However, a functionality of a design can sometimes be disrupted due to transformation.

These disruptions can come due to human error or miscommunication between various teams which are implementing the given design or wrong uses of an EDA tool or a bug in an EDA tool. Now, whenever such disruption in functionality happens, we want that we detect those disruptions or deviations in the functionality and fix them as soon as possible. So, to do these tasks, we take help of formal verification tools that we will be seeing in today's and subsequent lectures. Additionally, earlier we had seen that to carry out functional verification, we take help of two types of techniques.



The first one is simulation based technique that we had discussed simulation based technique in detail in the earlier lectures and the other is using formal methods. So, in today's lecture and in next three lectures, we will be looking at functional verification using formal methods. Now, before moving to the formal verification, let us look at a quotation from Dijkstra and the statement is '…program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness…' So, though this statement is made in the context of softwares and testing a software for bugs, this is very much relevant for hardware also.

So, when we test our hardware using simulation, it is more or less similar to program testing. We applied test vectors and see the output response and see whether it is matching as per expectation or not. Now, these techniques are very good at finding bugs and we rely on them when we write initial RTL and do a functional verification of our code or of our hardware, but are these techniques establishing a proof that our design is correct or not. So, the answer is no. So, this is one of the biggest problems of simulation that it can show the presence of bugs if it exists, but it does not establish if it is not showing any error from the result of simulation, we cannot say that our hardware is correct with a confidence level of say 100 percent.

So, the only effective way to raise our confidence is to take a hardware model and prove its correctness using some mathematical tools and the mathematical tool that we take help of is the formal verification tool as we will see in today's lecture. Now, before moving to the formal verification tools let us look into the limitations of simulation based verification in a little more detail. Now, let us take an example consider that we need to multiply two 32-bit integers meaning that we have a block which is implementing a multiplier and the inputs to it are two numbers which are of 32 bit size and this is producing a result. Now, to verify it using test vector or using simulation we have to consider 2^{32} combinations of A and similarly 2^{32} combinations of B. So, total we need to check for 2^{64} combinations of input vectors.

Now, if we take a simulator which is very fast and it takes one test vector to simulate in just say 1 microsecond then we can estimate how much time it will take to simulate 2^{64} combinations. So, if we can simply multiply $2^{64} * 1 * 10^{-6}$ and that will give us a number in seconds and if we convert it into number of years it will turn out to be around 0.5 million years. So, evidently this simulation based exhaustive verification is not feasible we cannot apply so many test vectors in the real world designs and establish correctness by simulation. So, what do we do in simulation of course, we cannot test a design exhaustively, we typically apply test vectors.

So, we apply input patterns or test vectors based on our anticipated sources of errors meaning that when we write our design when we code a Verilog model at that time we may be thinking that these areas are some corner cases where my design may fail and may not be producing the correct answer then write test vectors corresponding to that and then do the simulation. But what it turns out that the error typically do not occur in cases where we can easily anticipate errors. Actually errors often get in or a often exist in those

part of the design where a designer has not paid attention. And therefore, there simulation based techniques leaves a lot of room for improvement in terms of the thoroughness of verification. So, the simulation based verification can never prove the correctness of a design and therefore, formal verification methodologies which we will see today provides an feasible alternative.

So, for simulation based methodologies cannot exhaustively test our design. As an alternative we can use formal verification methodologies. Where we can use? How we can use? We will see in subsequent lectures. Now, why does formal verification techniques provide as an alternative and how is it different from simulation in terms of mechanism let us understand that. Now formal verification as an alternative what does it do is that it applies mathematical reasoning to establish proof of correctness. It does not apply test pattern it establishes the correctness of a model by mathematical reasoning, mathematical deductions and other techniques that we will see in today's lecture.

So, once using mathematical reasoning the formal verification tool is able to establish the correctness of a design then by correctness it implies that the system behaves correctly irrespective of the input test vector and implicitly the design will work correctly for any given test vector. So, if using mathematical tool we are able to establish the correctness of our design it implies that if we give any input to the tool based on the mathematical model that was assumed by the tool for establishing the proof, if the input remains in that domain then for any given input the system will behave correctly. So, without applying test vectors, the formal verification tool implicitly covers all the test vectors in our design. Now let us take an example that what it actually means.

Specification: $y = (x-4)^2$
Design: $y = x^2 - 8x + 16$

x	Specification: $y = (x-4)^2$	Design: $y = x^2 - 8x + 16$	Pass/Fail?
0	16	16	Pass
1	9	9	Pass
2	4	4	Pass
-1	25	25	Pass
104	10000	10000	Pass

Formal verification

$$
\begin{aligned}
&(x-4)^2 \\
&= (x-4).(x-4) \\
&= x.(x-4) - 4.(x-4) \\
&= x.x - x.4 - 4.x + 16 \\
&= x^2 - 8x + 16
\end{aligned}
$$

Let us take a simple case in which we are given a specification $y=(x-4)^2$. So, this is the specification. Now in a design what we do is that where we implement this function $y=(x-4)^2$ as $y=x^2-8x+16$. Apparently these two models are equivalent. We can from our elementary algebra knowledge, we can understand that these two models are indeed correct or equivalent. Now how will simulation based technique try to establish the

equivalence of these two models. So, simulation based technique will apply test vectors.

For example, it will say that assume that x can take only integer values then it will apply say an integer value of 0 it will see that the specification produces the answer as 16 and the implementation or design also produces an answer 16 and therefore simulation is possible. Similarly it will apply another test vector say 1 and see that both these two models are producing the same answer. For 2 also it produces same and minus 1 also it produces the same answer and for say 104 or any other number it is producing the same answer and therefore by testing or for a given a few test vectors it is trying to establish the correctness of the model. Since the number of inputs that are possible in this case is infinite because x can take any integer value. Therefore the simulation based technique is a kind of incomplete it is not covering entire test vector.

What if there was a test vector where this model was failing. For these 5 vectors it was passing, but that for a 6th test vector it was failing. We are never sure using simulation we can never be sure. Now how will formal verification technique or formal verification tool attack the same problem of establishing the equivalence between these two models. It will try to establish the equivalence by mathematical analysis or mathematical reasoning and using deductions. For example it will start with $(x-4)^2$ and then using the properties of square or the definition of square it will expand it as (x-4)*(x-4) and then using the distributive property and commutative property it can come up with these or the intermediate representation and finally it will come up with a representation which matches the implementation. Now in this deduction or in this derivation of the implementation, the tool used only the mathematical models that it knows or what are the constraint on the input x and what are the definitions of different operators and so on.

And therefore once it is able to establish that given a specification and given a design or implementation those are matching in terms of representation it can say that these two things are equivalent, these two, specification and design are equivalent. So, note that the the formal verification tool did not use any test vector to establish the equivalence of these two model. It used only mathematical reasoning and deduction to come up with this answer. And implicitly what it means is that if x takes any integer value these two models will always be equivalent. So, this is the basic difference between the approach taken by a simulation based technique and formal verification methods.

Now, let us summarize the main differences between a simulation based technique and formal verification. So, in terms of test vector simulation based verification requires test vectors while formal verification technique do not require test vectors. Then in terms of completeness, simulation based verification is not complete because it cannot cover all possible test vectors for a realistic designs and in formal verification there can be

completeness. The formal verification tool can for a given check, it can say that ok now implicitly I have covered all the test vector and therefore, these two models are equivalent or some property that it is trying to prove gets established. And in terms of mechanism the simulation based technique use test vector and sees the output response and compare it with the expected response that is the mechanism of simulation.

And in formal verification a mathematical proof of correctness is established or if these two models are not equivalent then it will produce a counter example. And in terms of memory and computational resource requirement usually the simulation based verification techniques requires computatively low computer resources while formal verification technique requires more computational resources. Now, we have understood that what is the basic difference between the mechanism of simulation based verification and formal verification. Now, an important thing about formal verification is that it is computationally very much challenged why because establishing a mathematical proof is not an easy problem. And therefore, a lot of research had gone into formal verification in last 50 years or so and in early 1990s some breakthrough came and an efficient formal verification techniques were developed that could be applied for VLSI design and now we routinely use formal verification techniques in our VLSI design flow.

Now, what are these efficient formal verification techniques? So, the breakthrough in formal verification came because of two basic formal verification methods. The first one is a very compact and efficient representation of Boolean function using a model or using a representation which is known as binary decision diagram. And the second technique which led to the breakthrough in or it allowed formal verification tools to scale to realistic designs was the design or was the implementation of very efficient algorithms that could solve the satisfiability problem or in short it is known as SAT problems. So, to understand the formal verification methods used in VLSI design flow we need to understand at this core technologies like what is the mechanism of this formal verification. To understand this mechanism we need to understand a little more deeper into these two methods that is BDDs and SAT solver.

So, in today's lecture we will be looking into the binary decision diagrams and how it helps in formal verification or it helps in efficient formal verification. And in next lecture we will be looking into the SAT solver or the techniques used to solve the SAT problem efficiently. And then in subsequent two lectures we will be using or we will be looking into that how these techniques are actually used inside VLSI design flow that allows efficient formal verification of our RTL model or VLSI models. Now given a Boolean function it can be represented in many different ways. In our earlier lectures, we had looked into some of the representations of Boolean function for example, we looked into truth table we also looked into sum of min terms representation.

Then we also looked into hyper cube representation of a Boolean function. And we also looked into factored form representation for multilevel logic and we also looked into we Boolean logic network and Boolean formulas. Now for these kind of representation, a very important attribute for these representation, is the compactness of the representation. So, what do we mean by compactness of a representation? So, compactness of a representation basically quantifies the growth in the size of the representation with the increase in the number of Boolean variables. So, if we increase the number of Boolean variables say from 5 to 10 then how does the size of that representation grow? Does it grow exponentially, linearly or so on? That is what the compactness of a representation denote.

Let us take an example for this. Suppose we represent a Boolean function in terms of truth table. Now we know that given N variables in a Boolean function, it will take 2^N number of rows in the representation of that Boolean function in a truth table. Now as we increase N, the size will grow and this size will actually grow exponentially for a truth table why because N is in exponential, 2^N. So, truth table size increases exponentially with the number of Boolean variables and therefore, we can say that the truth table is not a compact representation its size is simply growing exponentially as the number of variables is increasing. What it essentially means that we cannot represent a a big function in terms of number of variables for example, there are 100 variables we cannot think of representing it in terms of truth table.

And, why because it is not a compact representation. On the other hand for example, say logic formula for example, we say y=ab+acd+b'd+bc' this can be very compact representation. If we allow this logic formula representation to be flexible then this logic formula representation can grow not exponentially, but at a slower rate than truth table and it can represent bigger functions. Now what do we desire in terms of compactness of a Boolean function representation? We want that a Boolean function representation should be as small or as compact as possible. Why do we want such a behavior because if a Boolean function is very compact then we can represent in computer as data structure and the size of that data structure will be smaller and therefore, it will consume less memory and also it will be probably easier to manipulate those Boolean functions. So, ideally we want that our Boolean function representation should be as compact as possible.

Now there is another attribute of a representation of a Boolean function and that is known as canonicity. Now what is meant by canonicity? Canonicity: if a function is canonical or if the representation of a Boolean function is canonical then two equivalent functions are represented identically. So, if there are two equivalent functions suppose

there is a function f1 and there is an f2 and we say that these two functions are equivalent then the representation of these two functions will be identical that is what we mean by canonicity. Now conversely if a representation is canonical and if two functions have the same representation suppose f1 and f2 had the same representation then what it means that these two function will be essentially be equivalent. So, if in a representation these two properties are satisfied then we say that this representation is a canonical representation.

For example, a truth table is a canonical representation of a Boolean function why because if we exhaustively list rows for a truth table and maintain a variable order a given function can be represented in truth table in only one particular way there is no other way in which a Boolean function can be represented and therefore, a truth table is a canonical representation. And whether a logic formula is a canonical representation? No the logic formula is not a canonical representation. For example, if we take a function y=ab+ac+bc this is one representation. Now this same function can be represented in terms of logic formula as y=a(b+c)+bc that is we have taken a as common and then we have factored. And similarly we can write as ab+c(a+b) this is another representation then we can add a redundant product term and get another representation aa' is essentially 0. So, we can add it anywhere and the function will be still the same.

Similarly we can multiply any function with (a+a') which is essentially 1 and we get the same representation or same function and the representation is different. So, a given function can be represented in many forms as a logic formula and therefore, a logic formula is not a canonical representation. And in general we want a representation to be canonical why we want it to be canonical because manipulating a Boolean function becomes easier if it is canonical. For example, if you want to test the equivalence of two functions we can just check their representation maybe in terms of just pointer value and see whether those representations are same. If those representations are same we know for sure that these two functions are equal.

So, thus manipulating Boolean functions becomes easier in canonical representation. So, ideally what do we want? Ideally we want that the representation should be compact meaning the size should not be growing exponentially, but growing at a slower rate and it also should be canonical. Ideally these are some of the attributes that we want for a Boolean function representation for formal verification and also in logic optimization and logic synthesis. So, these compact canonical representation of Boolean functions are very very important.

But what we typically see is that, for example truth table. So, if we say there are two attributes of a representation one is canonicity and the other is say compactness. Then for say truth table, we can say that it is not compact, it is exponential therefore, this is not a

good attribute, but it is canonical. And if we look for say logic formula, then for a logic formula, it is a compact representation, more compact than a truth table, but it is not canonical. So, typically we do not have both these attributes together meaning canonicity and the compactness, but thankfully we have a representation which is known as binary decision diagram which has these characteristics, canonicity and compactness for many kinds of functions or many kinds of useful functions which are useful from the perspective of VLSI. Now, let us look what is a binary decision diagram or BDD. BDD is a data structure to represent Boolean functions canonically.

So, BDD is a canonical representation of a Boolean function and BDD is compact for many practically relevant functions. So, it is compact for many of the functions which are of practical relevance in VLSI, not for all. So, we will see for which cases it is not compact, but BDD is a canonical representation and it is compact for many useful functions. Now, additionally there are several Boolean operations which are relevant to formal verification that can be carried out very efficiently using BDD and therefore, BDD finds a lot of application in formal verification. Now, how do we build a BDD or BDD representation for a given Boolean function? We take help of Shannon expansion and what is Shannon expansion? Shannon expansion basically splits a given Boolean function into two sub functions.

The sub function that is obtained by assigning 1 to a variable or 0 to a variable. Now, if we create two sub functions from a given Boolean function by assigning value to a given Boolean variable first 0 and then 1. So, once we assign a value to a variable as 0, the sub function that we get that is known as negative cofactor and once the sub function that is obtained by assigning 1 to a given variable that is known as positive cofactor. Now, let us look into that how we can break or decompose a given Boolean function into negative cofactor and positive cofactor. Let us take an example, suppose the given function was ab+acd+b'd+bc'.

Now, assume that the given variables on the basis of which we want to decompose it is 'a'. Suppose we want to decompose it with respect to variable a. Then what we do is that first assign the value of a as 0. Now, if we assign value to a as 0 this term will become 0, this term will also become 0 and will be left with these two terms. So, this function y0 that we obtain after assigning a=0 this is known as negative cofactor. And then what we do is that we assign a=1.

Now, if we assign a=1, then what happens in this, this a term goes away and in this a term goes away. So, we are left with y1=b+cd+ the other terms and this sub function that we obtain after assigning a=1 is known as positive cofactor. And we can write this function y in the expanded form as y=a'* the negative cofactor, this part. We have

factorized or we have expanded with respect to the variable a and for the negative cofactor a' comes in the function. And for the positive cofactor we take a without taking its complement.

So, the same function y can be represented in this way a'*negative cofactor+a* positive cofactor. So, intuitively we can understand why this function and this function are equivalent or are the same. The reason is that if a can take only two of the possible values if a takes a value 0, then this term goes away and we are left with only the negative cofactor term. Similarly, if a takes a value of of 1 then this term goes away and this term takes a value of 1 and we are left with a function which is nothing but the positive cofactor. And that is why this representation of the function y in terms of positive and negative cofactor is correct.

Now Boolean functions can be represented as binary decision tree by applying Shannon expansion theorem recursively. We apply the Shannon expansion theorem first with respect to one variable, we get two functions, then to each of these sub functions we do Shannon expansion with respect to another variable. And recursively go on doing it until we reach the leaf level meaning the function takes a value as 0 or 1. So, let us take an example how we can build a binary decision tree using Shannon expansion theorem. So, let us take an example of a function y=x1x2+x2'x3+x1'x3'.

Boolean functions can be represented as **Binary Decision Tree** by applying **Shannon Expansion Theorem** recursively

$y = x_1x_2 + x_2'x_3 + x_1'x_3'$

Level 1: (Expand w.r.t x_1)

$y_{x_1=0} = x_2'x_3 + x_3'$

$y_{x_1=1} = x_2 + x_2'x_3$

Level 2: (Expand w.r.t x_2)

$y_{x_1=0,x_2=0} = x_3 + x_3'$

$y_{x_1=0,x_2=1} = x_3'$

$y_{x_1=1,x_2=0} = x_3$

$y_{x_1=1,x_2=1} = 1$

Level 3: (Expand w.r.t x_3)

$y_{x_1=0,x_2=0,x_3=0} = 1$

$y_{x_1=0,x_2=0,x_3=1} = 1$

$y_{x_1=0,x_2=1,x_3=0} = 1$

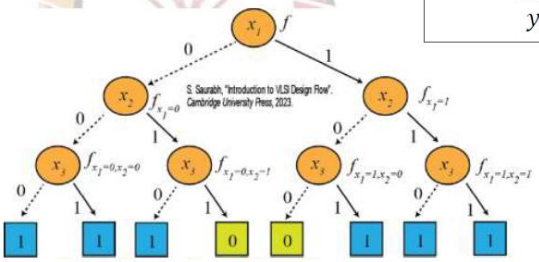
$y_{x_1=0,x_2=1,x_3=1} = 0$

$y_{x_1=1,x_2=0,x_3=0} = 0$

$y_{x_1=1,x_2=0,x_3=1} = 1$

$y_{x_1=1,x_2=1,x_3=0} = 1$

$y_{x_1=1,x_2=1,x_3=1} = 1$



To make it canonical and compact, we need to add constraint on variable ordering and remove redundancies

So, now let us first expand with respect to x1. So, if we expand with respect to x1 then we first take x1=0, if we take x1=0 then this term goes away and we are left with these two terms. Similarly, if we take x1=1 then this term becomes this term becomes x2 and this term x1', this term goes away and we are left with these two terms. So, first we

expanded with respect to the variable x1. Next we have got two functions which are functions of x2 and x3.

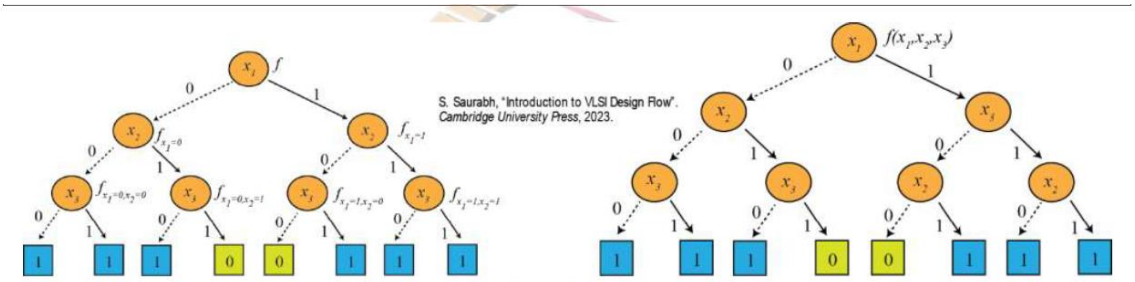
Now let us expand it with respect to x2. If we expand with respect to x2 the first function we uptake x2=0 if we take x2=0 this term will become 1 and we will be left with x3 plus x3'. Similarly, now we take x2 as x2 as 1. Now if we take x2 as 1 this x2' becomes 0 and this term goes away and we are left with x3. Similarly, we carry out expansion for this sub function and we get two more sub function with a by Shannon expansion. Now initially we started with one function, we got two sub function, now we got four sub function.

Now we again expanded with respect to x3. Now for this sub function, we take first x3=0, if we take x3=0, then this term will become 1 or this sub function will become 1. Next we take x3=1, then this term goes away and this becomes 1 and therefore, this function sub function becomes 1. Similarly, we expand each of this sub function with respect to x3 and we will get eight such sub functions and since we have exhausted all the variables now we will get the final value or the function will take a value of 0 or 1. Now we can represent these functions in the form of a tree we will take these elements, the final value in terms of 1 and 0 as the leaf of the tree and build functions. So let us see. So, this is the function f now we say that x1 takes a value 0.

So, this whole part represents the sub function or positive or negative cofactor with respect to x1 and this part represent the sub function which we get after making x1 as 1 and therefore, this represents a positive cofactor. Similarly, we get negative cofactor for each of the child and finally, we get the leaf level values which is nothing but the leaf level functions that we obtained after applying recursive Shannon expansion to the functions. Now is this representation compact of course, not why because at the leaf we will have 2^N elements if there are N variables in the Boolean function. So, this kind of representation is similar to a true table because we will have the number of leaf nodes will still be exponential.

So, it is not a compact representation. Now is it canonical, it may not be canonical if we apply, if we choose different vertices while expanding. For example, if we leave the freedom to choose any variable at any time while doing Shannon expansion then it may not be canonical also. For example, in this path if we say choose x2 as the variable on which we are expanding and in other we take x3 as a variable on which we are expanding. If we do that or we allow the freedom to do that then this Boolean decision tree representation is not even canonical. But the good thing is that we can make this representation compact as well as canonical as we will just see. Now at an intermediate

node we can choose any variable for expansion in binary decision tree that is what we can choose either x2 or x3 at different levels.



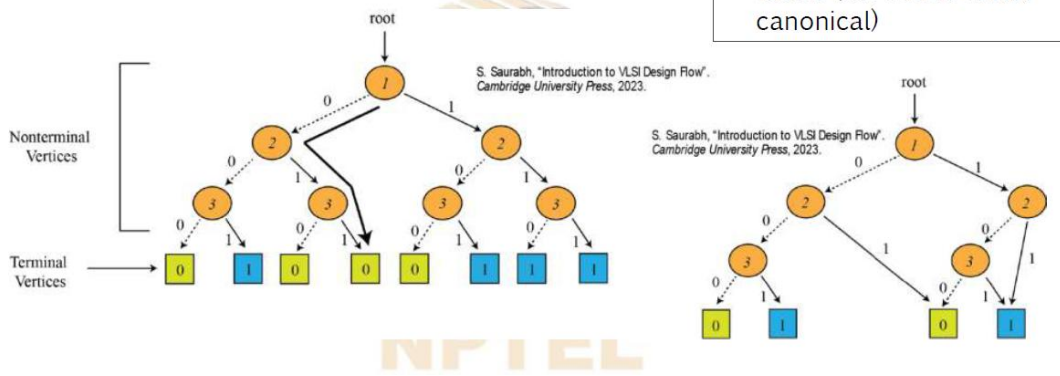
So, in general different variable orders can give different binary decision tree. Now for example, this is one representation and this is another representation. Here we have chosen x2 and here we have chosen x3 for expansion, here we have chosen x2 and x2. So, in this path, the variables chosen are x1, x2, x3 in that order and in this branch we have x1, x3, then x2 and so on. So, the variable order is not fixed. Now to enforce canonicity we need to add constraint on the binary decision tree and so once we apply this constraint then what we get is an ordered binary decision diagram or OBDD.

What is an OBDD? OBDD is a binary decision diagram or binary decision tree becomes an OBDD if the decision variables follow the same order in all the paths from the root to the leaf. Meaning that if we apply the constraint that the path from the root, this is the root node for this tree and these are the leaf nodes. If we enforce the constraint that the order of variables from root to leaf is always the same mind that some variables may be missing also, but if the constraint is such that the order from the root to the leaf, the variable separating in a path is always the same, then we say that it is an ordered binary decision diagram. Now let us look into ordered binary decision diagram more formally. So, let us take a Boolean function, Boolean function f(x1, x2, x3, …, xn) which is consisting of n Boolean variables.

And then for this function, we build an OBDD and what is an OBDD? OBDD is a rooted directed acyclic graph consisting of only two types of vertices. So, what is an OBDD? It is a rooted directed acyclic graph and what does a OBDD contain? It contains only two types of vertices one is the terminal vertex and other is non terminal vertex. And what are the characteristics or attributes of these terminal and non terminal vertices? So, terminal vertices will have outdegree=0 meaning that no edge will go out from a terminal vertices for it those are leaves of the graph of a tree. But note that OBDD may not be a tree. It is a graph structure, there can be edges going from one to the other and so on. So, we will see an example of OBDD in subsequent slides.

$y = x_1x_2 + x_1x_3 + x_2'x_3$

- A Boolean function is not uniquely defined by an OBDD (i.e. OBDD is not canonical)



So, there will be some terminal vertices in this OBDD. So, in terminal vertices there will be no outgoing edges, there will be no edge going out, only incoming edges. And these terminal edges will have two types of labels either it will have a label 0 or it will have a label 1. So, if it has got a label 0 it is known as 0 node and it indicates that the function assumes a value of 0. And if it is a 1 node then the label will be 1 and it indicates that the function takes a value of 1. And what are the non terminal vertices? So, each non terminal vertex v has the following: one is that it will have two children, one is low and the other is high.

For a given vertex v, there will be a low(v) which indicates its low children and the other is high(v) which is the high children. And then the vertex v will contain an index and index will be an integer. It will be from or a function of n variables. The index can be between 1 to n and it refers to the corresponding variable in the set x1, x2, x3, …, xn. So, there we have a correspondence between this index and the variable, for example, 1 will correspond to x1, 2 will correspond to x2, 3 will correspond to x3 and n will correspond to xn. So, there are vertices in the OBDD and each vertex has a label 1, 2, 3, 4 and these labels correspond to the variables for example, label 1 correspond to variable x1, 2 correspond to x2, and n correspond to variable xn. Now, each edge between vertex v and low(v) represents the case when the corresponding variables assumes $x_{index(v)}=0$.

It means that suppose there was a vertex 8, vertex had the label or index as 8. So, of course, this corresponds to the variable which is x_8. Now it will have 2 child. So, 2 outgoing edge: 1 will be the left outgoing edge, which is the edge between a vertex v and low(v) represents the case when the corresponding variable takes a value of $x_{index(v)}=0$ meaning if x_8 takes a value 0 then whatever is represented in the fan out of this that represents the value of the function when x_8 takes a value of 0 and this node is known as

low of 8, low(8). Similarly there will be another edge outgoing edge from this where x_8 will take a value 1 and it will correspond to high of 8, high(8), meaning that the sub function which will be represented by low(8) correspond to the sub function when x_8 takes a value of 0.

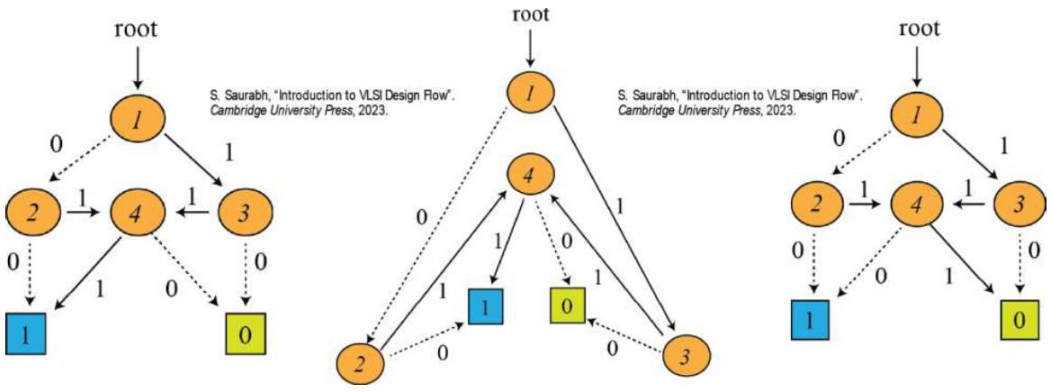
Similarly high(8) is a sub function which represents the sub function obtained when x_8 takes a value of 1. Now, to enforce ordering of non terminal vertices we enforce this constraint. Now we say that from the root to the leaf, the ordering of variables will be in certain way and to enforce that we say that the index(v) always be less than low(index(v)) and index(v) should be less than high(index(v)) meaning that if there is a low(v) of 8 then the value of or the index of this node can be 9, 10, 11 or greater than 8. Similarly the vertex which represent high(v), it will have index which will have value more than 8, it can have 9, 10, 11 and so on. So what this constraint means is that if we traverse from the root node to the leaf node with an index will only go on increasing it cannot decrease and that is why this is an ordered binary decision. Let's take diagram an example of an OBDD.

Suppose we are given a function y=x1x2+x1x3+x2'x3. Let us draw the OBDD for this. So since there are three variables we have these vertices and with each vertex we have associated variables. For example, this vertex is labeled 1 and with this the variable x1 is associated, with this the variable x2 is associated and so on. So the vertices which are shown in shown in orange color those these are non terminal vertices and the one which are at the leaf at the last these are the terminal vertices. Now at the root we have a label of 1 meaning that we are taking sub functions with respect to the variable x1.

So this sub function basically represents the sub function when x1 takes a value of 0 and this sub function represents the function or sub function when x1 takes a value of 1 and this root basically represent the given function y=x1x2+x1x3+x2'x3. Now given a Boolean function, the OBDD is still not canonical. though it is ordered meaning that if we go from say the root to the leaf everywhere the path is from 1, 2, 3 and so on. It is never 1, 3, 2 from the root to the vertex, to the leaf. So therefore, it is an ordered binary decision diagram, but an ordered binary decision diagram is still not canonical. For example, we can represent the same function in another way.

So we say that this part is nothing but 0 and therefore, we take an edge from here to here. And similarly this part is nothing but 1 and therefore, we take an edge directly from 2 to leaf. Now this is also an OBDD. Note that there can be path from 1 to 2 to leaf directly 3 is not appearing, that is ok, but if 3 appears then it should be only after 3 and that is why it is ordered.

So this is also an OBDD, this is also an OBDD, but the representations are different. Now OBDD can be made canonical by removing redundancies. So once we remove those redundancies then what we get is a reduced OBDD or ROBDD. Now what is an ROBDD? To understand that we have to understand the concept of isomorphism. So let us understand what is isomorphism. So 2 OBDDs f1 and f2 are isomorphic if there exists a one-to-one mapping between their set of vertices such that adjacency is preserved.



So the correspondence of the value at the terminal vertices and the index at the non terminal vertices must also exist. So let us take an example of isomorphic OBDDs and then this concept will be more clear. Suppose this is an OBDD and let us understand whether this another OBDD is this isomorphic to it or not? Now you have a root, here also we have root and then we have 1, so 1 is here. Now the outgoing edge of 1, the 0 outgoing edge of 1 goes to 2, from here also 0 goes to 2, from 1 again 1 goes to 3, 1 goes to 3 so these 2 edges are fine.

Now let us look into 2, that for the 2, this is the corresponding vertex. Now for these 2 vertices, the 0 edge goes to 1, and 1 edge goes to 4. Similarly these 2 edges are also fine. Now let us look into the vertex 3, now for the vertex 3, we see that for the 1 edge goes to 4, and 0 edge goes to 0. Therefore, the correspondence of edge and the adjacency list is established for 1, 2, 3 vertices. What about 4? Now for the 4, the 1 outgoing edge is 1 and for the 0 edge is going to 0. So, this is also established and therefore, these 2 graphs are isomorphic or these 2 BDDs are isomorphic.

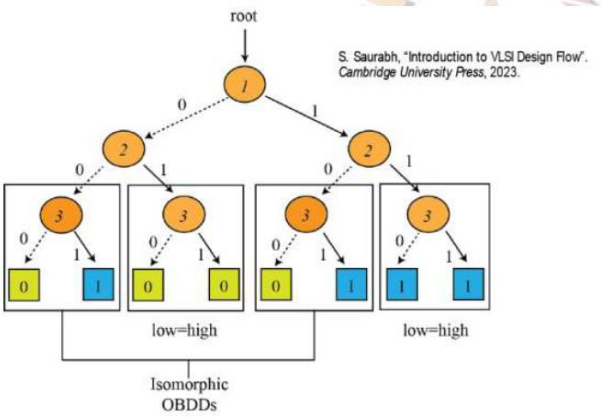
Now let us take another OBDD. Now let us see whether this OBDD and this OBDD these are isomorphic or not. Now we see that from the shape it looks like these 2 are isomorphic, but here the 0 edge of 4 goes to 1 in this case and in this, 0 edge goes to 0 and the 1 edge goes to 0 in this case and 1 edge goes to 1 here. So, the leaf vertices do not correspond and therefore, these 2 OBDDs are not isomorphic. So, now we understood that what is an isomorphic OBDD and what is an non isomorphic OBDD.

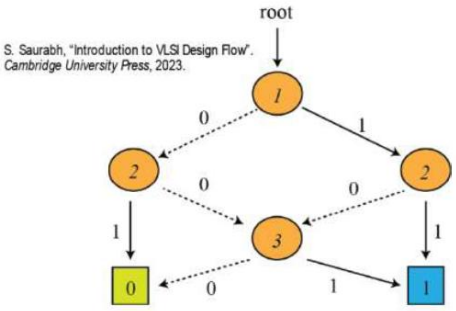
ROBDD is an OBDD with the following constraints:

- No vertex v has $low(v) = high(v)$
- No pair of vertices $\{u, v\}$ exists in the OBDD such that subgraph rooted at u and v are isomorphic.

Can be obtained by systematically removing vertices from the OBDD:

- Any vertex with identical children is removed and replaced with any of its children.
- Two vertices with identical OBDDs are merged into one.





Now let us understand what is a reduced ordered binary decision diagram or ROBDD. ROBDD is an OBDD with the following constraint. What constraints are there in ROBDD? The first is that there should not be any vertex v which has got low(v)=high(v) that cannot be. If low and high(v) are same then we need to do some transformation to make, an OBDD, ROBDD. And there should not be no pair of vertices (u, v) exist in the OBDD such that the sub graph rooted at u and v are isomorphic, this condition should also be applied meaning that in the OBDD there should not be 2 vertices which have isomorphic OBDDs or sub graph rooted at that point. So, we can obtain an ROBDD by systematically removing vertices from OBDD. What can we do is that any vertex with identical children we can remove it. If there is a vertex, 0 edge and 1 edge goes to an identical vertex then we can replace that node with any of its children.

And if 2 vertices with identical OBDDs are there then we merge them into 1. So, let us take an example and see that how we can obtain an ROBDD using OBDD. Suppose this is an OBDD which is given to us now how can we get a reduced ordered binary decision diagram or ROBDD. Now in this OBDD we see that for this vertex 0 and 1 edge both leads to 0 and for this both leads to 1. So, what we can do is that we can remove this 3 vertex and directly connect this to 1. Similarly, we can directly connect it to 0 and we can remove all these node.

This is the first reduction, we remove any vertex with identical children. Now, the other thing we can notice that if we consider this vertex 3 and this vertex 3 we have an identical isomorphic OBDD rooted at 3, 0 goes to 0, 1 goes to 1; 3, 0 goes to 0, 1 goes to 1. So, these 2 are identical what we can do is that we can remove one of them and point this 0

edge to this and we will obtain the reduced ordered BDD. So, this is a well drawn diagram. So, what we have done is that we have simply removed these 2 vertices which leads to identical sub graphs.

So, we have removed this and corresponding to 1 we have now 0 directly. So, corresponding to this we have only this edge and corresponding to this 1, we have removed this and we have got 1 vertex. So, this edge. And now 3, these 2 vertices were identical, these 2 vertices had isomorphic OBDDs. So, we removed one of them. So, we have this vertex 3, whose 0 edge goes to 0, 1 edge goes to 1 and both this and this points to the same vertex.

So, this is how we can create an ROBDD from a given OBDD. Now, what we have done is that now given a binary decision diagram, we enforce the ordering of variable, then we got ordered binary decision diagram and on ordered binary decision diagram, we perform some reduction operation and remove the redundancies. Once we have removed the redundancies from OBDD we get what is known as ROBDD and Bryant who was the inventor of this BDD proved that an ROBDD is a canonical representation of a Boolean function. Meaning that if there are 2 functions which are equivalent then their representation will be identical in ROBDD. Meaning we cannot represent a given function in 2 different ways. There has to be only one representation for 2 equivalent functions. Similarly, if there are 2 functions and their representations are same then we can derive or deduce from that, that these 2 functions are equivalent, it has to be equal.

So, Bryant proved mathematically that ROBDD is a canonical representation of a Boolean function. And in typical literature and when we talk of BDD we actually refer to ROBDD, reduced order binary decision diagram. Because the ROBDD are really useful because they are canonical and many a times for many useful Boolean functions they are compact. Now what are the applications of BDDs? So, the application is that the testing of equivalence of 2 function becomes very very easy why because if we have represented functions in a BDD in popular BDD packages then their representation in terms of pointers will be exactly same. And by just comparing the pointer we can say if 2 functions are equivalent or not. And testing satisfiability and satisfiability meaning that given a function, can the variable assignment be made such that the function takes a value as 1.

Now just looking at the BDD and comparing whether there is a path from that node to a 1 node we can say that it is satisfiable or not satisfiable or it is a tautology meaning that whether a function is always taking a value 1. If that is the case then there will be for a Boolean function representation the ROBDD will be just 1 node that is the leaf node having a value of 1. So, set testing satisfiability and tautology and testing for equivalence

of Boolean functions they become very easy using BDDs. And what about the compactness of a BDD or ROBDD? So, we have seen that ROBDD are canonical that is a very powerful tool for us for formal verification. But what about the size of BDD? So, the size of BDD or ROBDD for many Boolean functions grows as polynomial with the number of variables and that is the beauty of ROBDD.

Meaning that it is canonical and for many useful functions it is the representation grows linearly or grows at a slower rate with the number of variables. However, note that the size of ROBDD is dependent on the variable order meaning that if we choose right variable order then probably it will give a very compact representation. If we choose wrong or not optimal variable order then the size of of ROBDD can grow exponentially. For example, if we look into say adder circuit. If we represent adder function in an ROBDD then depending on the variable order the size of ROBDD can be linear or it could be exponential. So, the order of variables has to be chosen carefully while building an ROBDD. Now, but the problem is that finding a good variable order is difficult and tools, BDD packages use various heuristics to come up with a good variable order so that the size of the Boolean representation remains compact. For some functions such as multiplier, the size of ROBDD is always exponential. So, we should not think that ROBDD is always compact and we can always make it compact by choosing a good variable order. The first problem is that choosing a good variable order is not an easy problem and the second problem is that there are some functions for example, multiplier function for which the BDD representation will always be exponential. So, BDD representation is very useful, but in some cases it may not help and therefore, we look at other techniques also for formal verification. So, if you want to look further into the topics that we discussed in this lecture you can refer to these materials.

Now, let us summarize what we have done in this lecture. So, in this lecture we have looked into formal verification and how it differs with simulation based techniques and we also looked into two methods or techniques that made formal verification popular for VLSI design flow and these two techniques were the compact and canonical representation of binary decision diagram or reduced order binary decision diagram and the second technique is the efficient SAT solvers or Boolean satisfiability solvers. So, in this lecture we had looked in detail binary decision diagrams and in the next lecture will be looking into what is the satisfiability problem and what does an efficiently satisfiability solvers do. Thank you very much. .