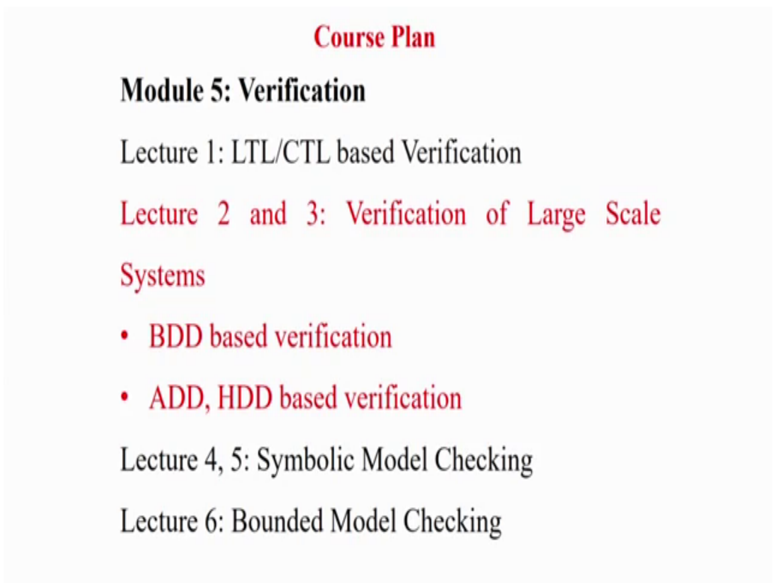
Optimization Techniques for Digital VLSI Design

Dr. Chandan Karfa
Dr. Santosh Biswas
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 21
BDD based verification

Welcome to the next lecture on formal verification which was the part of course; on optimisation techniques of VLSI design.

(Refer Slide Time: 00:34)



Course Plan

Module 5: Verification

Lecture 1: LTL/CTL based Verification

Lecture 2 and 3: Verification of Large Scale Systems

- BDD based verification
- ADD, HDD based verification

Lecture 4, 5: Symbolic Model Checking

Lecture 6: Bounded Model Checking

So, in the last lecture, basically we were trying to see how we can model large scale system binary decision diagram that our motivation came from the fact that verification required explicit modelling of the systems and if you are using a straight forward state based representation due to the exponential nature of the state encoding the systems size blows up. So, we were trying to see basic exciting data structure called binary decision diagrams and we are trying to see, how it can drastically bring down the size of any binary decision tree; yet not losing any information.

So, in the last lecture that is on the binary BDD based verification is a two lecture module or two lecture structure because this quite it has quite a lot of material to be discussed. So, what we have seen that basically that we are if you are given a best span addition tree, then how we can actually compares, it using a BDD without losing any information and it was very interesting to find, I hope that in a any binary decision tree there are lot of redundant nodes which can be easily removed and. So, that the redundancies are gone and you can get a very efficient representation which is nothing, but a BDD, but then where you stop we say that given a binary tree we have to how we can reduce it to a binary decision diagram is what we were discussing that was to show you that what is the composite power of a BDD.

Or in other words, what is the amount of redundancy that is any binary decision tree, but if you tell me that I have to design a BDD in that manner, it is not a very good idea to do that. In fact, it is impossible to do it that way why because if I am giving a such a big tree and then you are compressing the and the size of the tree is the order of 2 to the power n where n is the number of system variables. So, it is impossible to built such a big tree, if the input variables are 20 or more than that a typical number, then it is a it is a foolish idea that first I blow it up and then I compress. So, that is not a very good idea or in feasible idea to do it in that way.

So, today we are going to see is that we are going to get to go for such an approach rather what we will do we will have small small atomic BDD's and then we will merge them. So, that you get an entire BDD which will represent the total function or a circuit which is a more logical way of doing it that is we will try to build up the structure. So, that it does not grow big at any point of time because the other way around, that take a binary decision tree and bring it to be ready is not a possible way of handling the solution.

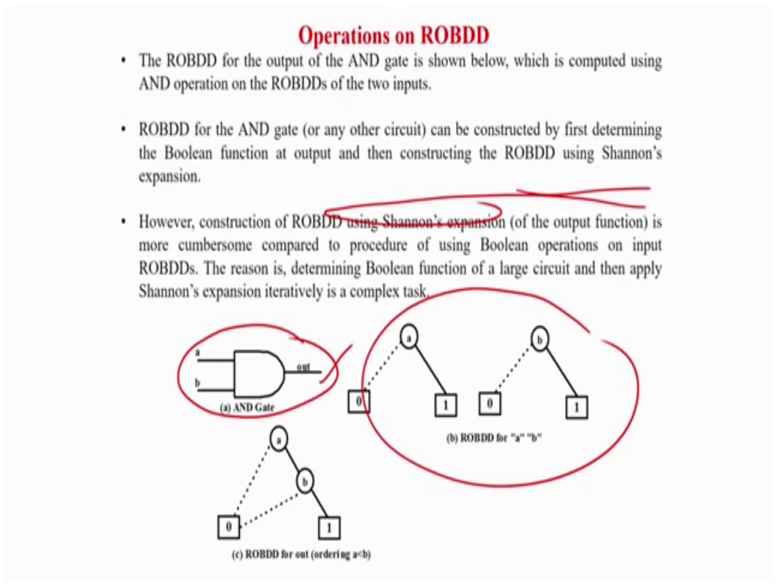
Then we will see that binary decision diagram basically always follows a bit level structure and whenever we have discussed about testing in the last module, we have seen that whatever you do whatever optimisation you do if you are working at bit level you cannot go higher than a certain range.

So, the need of modern day VLSI systems like NOC's and SOC's, you cannot think of anything any representation in terms of bits, you have to go to words or you have to go to integers and which is more abstract view, then the bit level circuits that is we were mainly looking at the RTL level diagrams or RTL level circuits. So, in case of BDD's also, we have to move to higher level of abstraction like arithmetic decision diagram or high level decision diagrams which are actually nowadays used in place of BDD's to represent large scale systems of systems like NOC's and SOC's, the idea is even with BDD's, you cannot handle such large systems.

So, the ADD and HLDD based optimisation and there are so many other decision diagrams like ZDD, etcetera, but we will see the most important 2 of them that ADD and HLDD.

Basically they will show you how you can abstract down and till you can go for a much beyond large much larger system than that can be even concept by a BDD. So, we will now quickly basically go to basically where we had left last time that we have already discussed that how we can go about constructing a BDD from the binary decision diagram and tuning it down, but now we are going in the rather the other way round where we will be doing it from the roots.

(Refer Slide Time: 04:17)



Operations on ROBDD

- The ROBDD for the output of the AND gate is shown below, which is computed using AND operation on the ROBDDs of the two inputs.
- ROBDD for the AND gate (or any other circuit) can be constructed by first determining the Boolean function at output and then constructing the ROBDD using Shannon's expansion.
- However, construction of ROBDD ~~using Shannon's expansion~~ (of the output function) is more cumbersome compared to procedure of using Boolean operations on input ROBDDs. The reason is, determining Boolean function of a large circuit and then apply Shannon's expansion iteratively is a complex task.

So, basically what it says the ROBDD for the output of an AND gate. So, that taking the output of an AND gate. So, which is actually which we can compute by the ending of the 2 BDD's for a and b. So, as I tell you a and b are 2 variables. So, what will be the BDD look like they are simple as something like that a 0 is 0 a 1 is 1 for any single variable, the BDD will be looking like this b equal to 0 means 0 b equal to 1 means 1, this is the simplest way of representing a single variable in terms of BDD.

Now, what you can do. So, this is a AND gate. Now how can we do that. So, the second point says ROBDD for the AND gate can be constructed by first determining the Boolean function at the output that you find the Boolean function then construct the ROBDD using Shannon's expression that in another way, we can say that basically you go for the binary decision tree kind of a diagram and then tune it out, but construction of ROBDD using Shannon's expansion is scrambled some because of the large size. So, this an AND gate.

So, it is trivial, but if you with a very big circuit, you will have a big Boolean formula at the output, you have to do a Shannon's expression and you have to make it that will not be a very good idea to do it that is something like you are having a very big binary decision tree and you are tuning it down to a BDD that is not a very good solution of doing it.

Rather, what you should do rather basically, you should try applying operations on binary decision diagrams like we will have one BDD here, one BDD here, we have to go for and operation we should have the provision of ending the BDD's and finally, you should have a reduced BDD something like this with test that a equal to 0 means 0 a equal to 1 means you have to check the value of v if b is 1, the answer is one else the answer is 0 so.

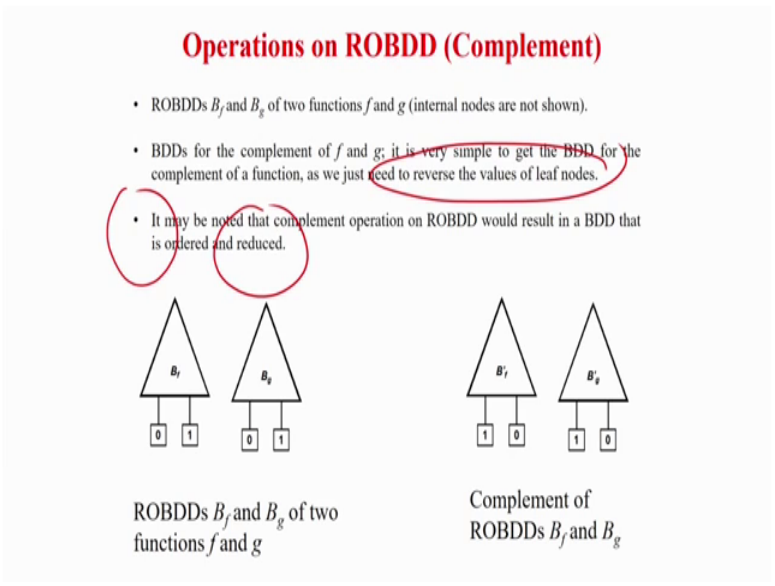
In fact, we have to get this final BDD by operation on these 2 BDD's not by the way that for a given big circuit you basically have the Boolean expression and then go for a Shannon's method of expansion and going for the BDD generation which we have discussed in the last class, you should not follow that, but why we are discussing all this things in the last class to give you an idea what is BDD and what is the power of BDD how much compression it can do.

So, that was the idea we that is why we were showing it in the last class, but practically computing a BDD, you have to grow from the leaf level a very big circuit is there, you take all the input BDD's which are of this form then you go for BDD operations and finally, make the final BDD for the last circuit. So, for that is very very important for us to understand, how BDD's are operated that is BDD a plus BDD b BDD a dot BDD b b d b compliment.

So, all Boolean functions we do like and or xor not bar all things are possible on BDD's. So, that is why it is not only about the compression power of BDD rather there are. So, many interesting features of BDD that has made BDD and. In fact, you should be very thankful to Bryant for finding out such a wonderful data structure. So, basically; obviously, it is compressed.

Secondly, you can do all Boolean operations on BDD's that is another good thing and third is basically canonicity that we if the ordering is same and if the function is same both the BDD will be or the functions are equivalent rather both the BDD's will be identical. So, that was the 3 basic features which make BDD as the most important data structure for VLSI and digital design.

(Refer Slide Time: 07:16)



Operations on ROBDD (Complement)

- ROBDDs B_f and B_g of two functions f and g (internal nodes are not shown).
- BDDs for the complement of f and g; it is very simple to get the BDD for the complement of a function, as we just need to reverse the values of leaf nodes.
- It may be noted that complement operation on ROBDD would result in a BDD that is ordered and reduced.

ROBDDs B_f and B_g of two functions f and g

Complement of ROBDDs B_f and B_g

So, now we will show you very simple way of handling BDD's. So, complement. So, what is the complement very it extremely simple procedure, if say we are considering that they are the 2 BDD's I am not drawing the internal structure this is BDD 1 and BDD 2. So, if I just take this BDD B f and I if I just reverse the values that here the 2 BDD's I mean take the 2 values of 2 BDD's. So, B BDD f and BDD g, now I will do some operations on that, if I want to make because it is BDD B f BDD g are the 2 BDD's, we will be considering in this lecture to show all the operation first one is complement. So, how do a complement extremely simple the most simplest of all operation. So, this is your BDD f and g 0 one and 0 one in compression what in negation what you do you just reverse the terminal nodes or make the values of terminal node flip; obviously, the whole circuit will be or the whole output will be a complemented output entire BDD will remain same the complement operation will just make this node one and this node 0 and so on .

So, just you need to change the leaf node values then the BDD will be complemented. So, it is simple the BDD's for the complement, it is very you just need to reverse the values of the leaf nodes that is what is being sent. So, it may be noted that the complement BDD on ROBDD would result in a BDD that is ordered and reduced one very important point I have to tell you why it has such a meaning one bad thing about the operation is that if you operate to BDD the output of the BDD is correct, but it may not be the reduced one it may have some redundant nodes. So, because I mean there is no direct way people have not been able to find out any direct way of operating the BDD's.

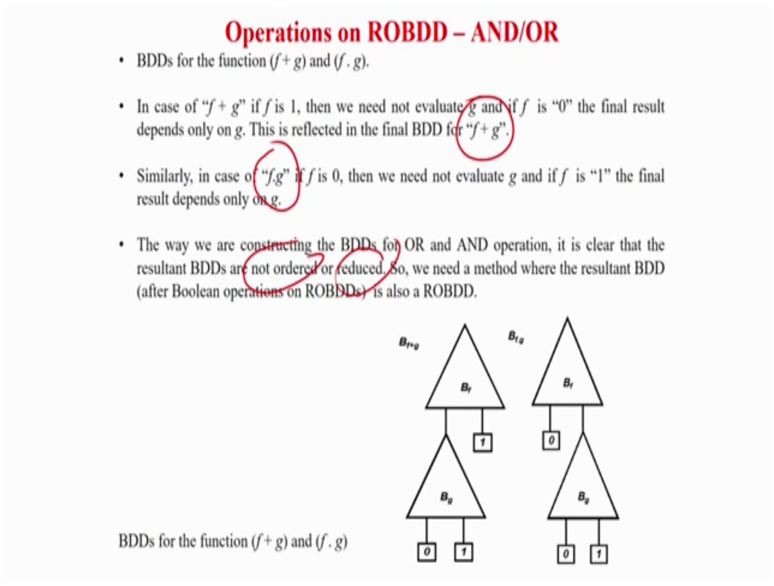
So, that directly you can get a compressed BDD or the most optimised BDD without redundancies. So, what we have to do you do the operations on the BDD and then you have to eliminate the redundancies explicitly, but one thing you have to note that the redundancy will be very slight only a few amount of nodes will be redundant it will be not be like that that you operate 2 BDD's and it will become a binary decision tree that will not happen slight few number of node will become redundant which you have to eliminate out and then it will actually become a BDD.

So, again repeating take 2 BDD's operate on them, you will get another BDD, but that BDD will not be ordered, it will not be reduced, but the ordering will be maintained that is very very important that basically ordering means if you have a b c d in the one order and and one more important thing I have to say is that when you are operating the 2 BDD's you have to maintain the order ordering is very important.

So, whenever you think of operation on BDD's a common ordering, you have to maintain across all the functions all your jobs you are doing ordering of the variables has to be seen under that case, if you are doing any operation accepting the complement, we are discussing for all cases it may happen that the BDD some redundant elimination nodes may go into picture, but that will not be very high in number that will become something as use like a binary decision tree that will not happen. So, you have to use like tuning of and few nodes will be eliminated and you will get the finally, reduced BDD, but in case of complement is a very simple operation just you are flipping of the values of the leaf nodes.

So, in that case, the order will be also maintained order is always maintained basically and it will be a reduced BDD, it should you will not have any redundant node a coming into picture once, I will go for some other operation you will find out why some redundant nodes may come up in a other operations, but for the complement just think the idea that you are just reversing the value of the leaf node. So, nothing changes very simple operation ordering of variables remain same in all operations basically not only this, but a special case of complement operation is that the BDD remains reduced.

(Refer Slide Time: 10:41)



Operations on ROBDD – AND/OR

- BDDs for the function $(f+g)$ and $(f.g)$.
- In case of "$f+g$" if f is 1, then we need not evaluate g and if f is "0" the final result depends only on g. This is reflected in the final BDD for "$f+g$".
- Similarly, in case of "$f.g$" if f is 0, then we need not evaluate g and if f is "1" the final result depends only on g.
- The way we are constructing the BDDs for OR and AND operation, it is clear that the resultant BDDs are not ordered or reduced. So, we need a method where the resultant BDD (after Boolean operations on ROBDDs) is also a ROBDD.

BDDs for the function $(f+g)$ and $(f.g)$

Now, things to start becoming more clear, why redundancy comes basically. So, we are not taking about 2 more BDD's that is and or so, 2 BDD's are there, how I will do on and or not, they we can you can read the slides that just about what I am explaining I have written them in text for you to read. So, just see this. So, this BDD what it says that this is B f b or this is the or BDD. So, B f 1 and I am just taking the left node that is a 0 path of B f and I am correcting B g and I am just attaching it directly.

So, what it implies if it implies that if B f is equal to 1, then output is equal to all operation. So, any of the BDD is giving a answer one the answer is one, but if the first BDD is giving a answer 0, then what ten basically you have to check for second BDD if the second BDD is also 0, the answer is 0 else the answer is a 1. So, now, you see it is a very simple way of handling take 2 BDD's or in basically makes such a similar structure that basically it will be connected like this of course, the ordering will be you have to obtain a common order whenever you are taking both the BDD the ordering should be maintained. I will take give some examples when that will become more clear, but here you can see redundancy comes in.

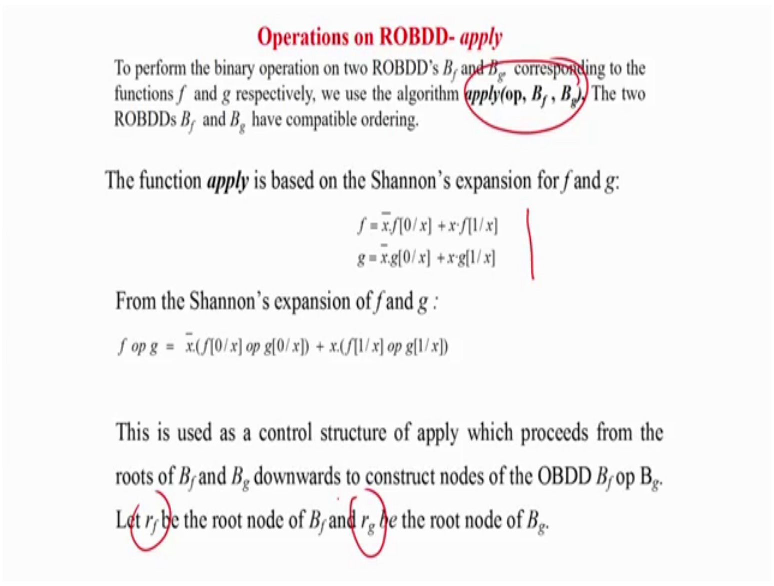
So, how the redundancy will be eliminated you have to have basically you have to remove this node and you have to connect this one to this one to eliminate the nodes. So, not only at the leaf level internal levels also many redundancies will come in which has to be eliminated, but they are limited number of nodes that will become redundant, but anyway just this gives you an idea how it happens and if its and operation, then if any of the gate any of the BDD's are 0 answer is 0, if first BDD's are 1, you have to check for the second BDD if that is also one the final answer is a one else the answers are 0. So, now, you can see 2 redundant leaf nodes internally of some redundant nodes may be there which has to be eliminated and then actually get the reduced order BDD.

So, idea is that you do not go for n number of operations like for example, you may have to do a plus b plus c dot d, you do not the whole computation and start reducing that will actually blow the complexity you do one atomic on one binary operation like 2 BDD's you will get another BDD which may not be reduced reduce it and then go for the second operation and so on. So, at any point of time you are not allowing the BDD to blow up. So, to you do tight redundancy gets in to remove it, then go for the operation with the third operand and keep on doing it and after every operation you reduce you eliminate the redundancy.

So, at no point of time, you will have a very large BDD and your complexity will always remain with bounce. So, what I am saying the whatever I have told you is written in this one in case of f plus g, if f is 1, then we did not evaluate g and if a is 0 the final result depends only on g this is reflected by the BDD for f plus g similarly dual will happen if it is a for g.

So, now you can see the way we are constructing BDD's it is clear that the BDD's are not ordered or not reduced. So, basically the idea is that BDD f may have variables x 1 x 2 and x 3 and BDD g may have variables like a i j k l. So, new variables may come in. So, the ordering may be changed; changed means order may be may not remain same that is what they are trying to say and; obviously, the redundancy is lost redundancy gate seen basically. So, you have to take care of this what do I saying that the ordering is not maintained it says that it may happen that BDD f may have variables a b c and BDD g may have variables a b and f. So, new variables will start coming in. So, ordering may be hampered. So, new variables will be there. So, you have to reduce it and you have to take care of the new ordering which may be a b c and f.

(Refer Slide Time: 14:01)



Operations on ROBDD- *apply*

To perform the binary operation on two ROBDD's B_f and B_g corresponding to the functions f and g respectively, we use the algorithm ***apply*(op, B_f , B_g)**. The two ROBDDs B_f and B_g have compatible ordering.

The function ***apply*** is based on the Shannon's expansion for f and g:

$$f = \bar{x}.f[0/x] + x \cdot f[1/x]$$
$$g = \bar{x}.g[0/x] + x \cdot g[1/x]$$

From the Shannon's expansion of f and g :

$$f \; op \; g \; = \; \bar{x}.(f[0/x] \; op \; g[0/x]) + x.(f[1/x] \; op \; g[1/x])$$

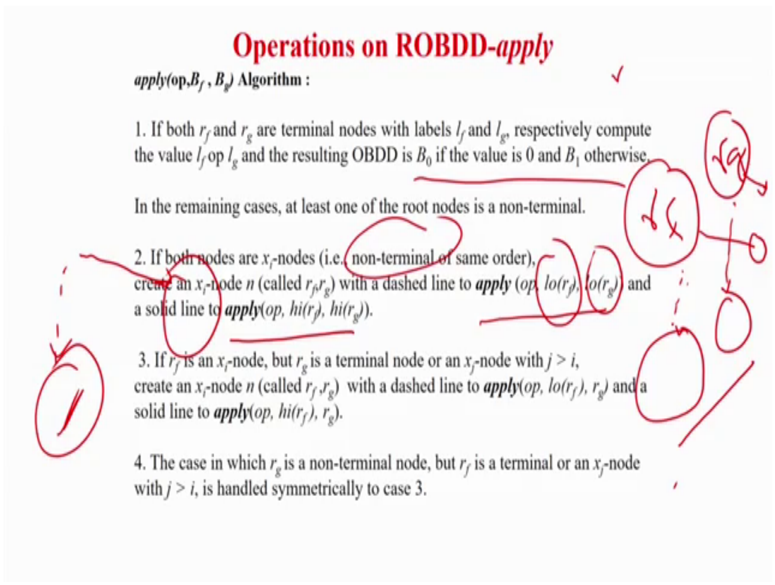
This is used as a control structure of apply which proceeds from the roots of B_f and B_g downwards to construct nodes of the OBDD B_f op B_g.
Let r_f be the root node of B_f and r_g be the root node of B_g.

So, that way, we will take more concrete examples to do it. So, basically this is a mathematical way of using the BDD operation called apply which is the most important one because complement is just a bar apply is you actually take 2 BDD's and do the operations. So, the formal way of writing is apply of B f, B g, B f and B g are 2 BDD's reduced ordered BDD is and operation is add dot xor, etcetera and the Shannon way of representing is you can just see over here . So, f of g is nothing, but equal to x bar the expression f with x 0 plus x with the expression x express as one and similarly, for g and when you are going for the operation this Shannon's expression value is x bar f of g with x x with x made 0 of same operation add star whatever g with same thing all the variables in function g are made 0 and then x with f where all the x are made one operate operation with g where in function g all the x's are made one.

So, basically this Shannon's way of representating and then basically we will now tell you the, this is the mathematical way of representation. Now we will basically tell you the algorithm, how to basically go for operating 2 BDD’s. So, basically they say that this is where the control structure to apply which proceeds from the roots of B and g downwards to construct the nodes of B f operand B g. So, basically we start with the node and then Shannon expansion basically tell that you have to keep on moving from the top node to the root nodes, it has like already we have seen that we first make x equal to 0 and keep the function in the left part and make x equal to 1 do it at the right part, now with the operations the same thing will be done.

We will take give you all the elaboration with examples. So, we are assuming that r f be the root node of B f and r g be the root node of B g. So, we will start with basically the root nodes.

(Refer Slide Time: 15:51)



Operations on ROBDD-*apply*

***apply*(op,B_f , B_g) Algorithm :**

1. If both r_f and r_g are terminal nodes with labels l_f and l_g, respectively compute the value l_f op l_g and the resulting OBDD is B_0 if the value is 0 and B_1 otherwise.

In the remaining cases, at least one of the root nodes is a non-terminal.

2. If both nodes are x_i-nodes (i.e. non-terminal of same order), create an x_i-node n (called r_f,r_g) with a dashed line to ***apply*** (*op, lo*(r_f), *lo*(r_g)) and a solid line to ***apply***(*op, hi*(r_f), *hi*(r_g)).

3. If r_f is an x_i-node, but r_g is a terminal node or an x_j-node with $j > i$, create an x_i-node n (called r_f ,r_g) with a dashed line to ***apply***(*op, lo*(r_f), r_g) and a solid line to ***apply***(*op, hi*(r_f), r_g).

4. The case in which r_g is a non-terminal node, but r_f is a terminal or an x_j-node with $j > i$, is handled symmetrically to case 3.

So, now this algorithm; so, I will just give you the gist of the algorithm, then we will take an example which will make you more clear the algorithm is on the screen please read it because more I means understanding if the algorithm will come when you go for the example and then coming back, but I will give you the gist. So, what it says apply we will go step wise. So, it says that apply B f B g. So, this is the algorithm. So, for the first step first step we have see if both r b and r g are terminal nodes with some labels l f l g that is 0 or 1 respectively compute the value of l f l g that is we are starting with 2 nodes basically say no this is called r f and this is called r g.

So, let let then be 2 terminal nodes may be 0 or 1, then what is may be this is basically this is basically 0 and this is basically one it may happen in that case you do not have do anything just take this 2 nodes and operate if we add answer will be 0 if it is or answer will be one and so forth, if is xor, the answer will be 1 and so forth. So, and the resulting is the and the resulting o BDD is B naught if the value is 0 and other one.

So, in that case you will have a concrete node called 0 or 1, the final answer will be b 1 or b 0 that is the final answer for this operation if both the nodes are basically r b and r g are terminal nodes. So, this will happen when your algorithm will terminate because where we will do the BDD operation is that you will take 2 nodes one node from this tree and one node from this tree and finally, we will stop when both the nodes are basically the your terminal nodes and that time the algorithm will terminate basically one is for termination of the algorithm.

Next the other cases which will maximum happen in the remaining cases at least one of the root node is a non terminal; that means, you have one 2 non terminals or one is a terminal, one is non terminal, there is at least one of the non terminal nodes are there, then they say you just look at this if both the node's are x i nodes that is non terminal if both the nodes are non terminal we are taking a node x i we are calling it as r f ok, if both the node's are x i nodes that is you are taking 2 nodes say basically we are taking may be r f sorry we are taking both are x i nodes.

So, when we are calling both the nodes are x i nodes means both are non terminal nodes of of the same order; that means, it is saying that if that is step 2, it is saying that if both the node's are x i node x i node means non terminal nodes of the same order; that means, say I have a node called r f another node called r g to belonging 2 different say, there may be node x 3, may be x 4 or x 5 that is they are in the same order because when I have taking a operator of 2 BDD's the ordering should be maintained, it is something like that both are of the same order.

In fact, they are the same level nodes. So, basically what you do if they are of the same order, then what you do is that basically you take corresponding to this you have a node and then you will go for this part which one will be equal to r f. So, lo of r f into lo of r g; that means, the idea is something like this, this is for this node is for first BDD and this node is for left b d second BDD and r f and r g assume that they are at the same level; that means, either x 1 x 2 x 3 something like that not at different levels ordering.

Then what you do you have to make another node which will be resulting of this node what will be in that node that node will basically the left I am making the left node first because we are combining the 2 nodes. So, what will be the child left child corresponding to r f and r g, it is something like apply operation it is add or subtract lower path that is lo child lower child of r f and the lower child of r g. So, basically this new node will come up which is actually corresponding to the lower child of this that is left child of this sorry the left child of this and it will also have a left child.

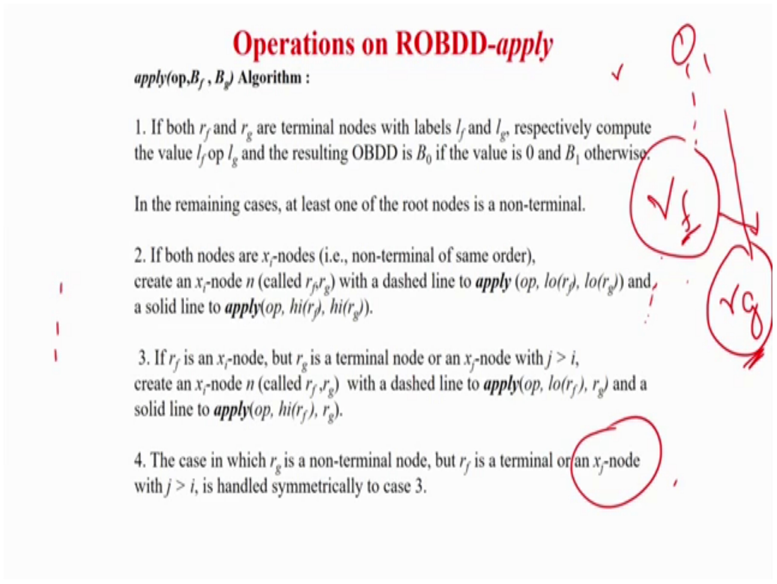
So, both the left child's you take go for the operation and this one will be resulting in the child loop child node corresponding to r f and r g and of course, this one this one will come for the corresponding to the left child similarly we will have a right child corresponding to r f and r g. So, what will be the result that will be the operation on the node which is at the right side of r f operate thing on the right side of r g. So, that one will correspond to the apply op high of r f and high of r g the logic very simple node a of this tree node b of this tree same level ok.

Now what you do corresponding this 2 there will be a left child and a right child how to get a left child you take the 2 parent are there you take the left child of both the node both the parent node operate them, it will become the left child corresponding to that node that is 0 0eth value and then similarly for the right child that is you take this 2 nodes take the right right hand child that is one for the parent nodes do the operation and you will find out the right hand child of the parent node that is the solid line. So, 0 and one you can do it.

That is very simple; that means, it says that if you have 2 nodes at the same level compatible nodes at the same level, this side will be the node these are the 2 nodes the child node of the operate operating final BDD will be the end and or or whatever operation of the 2 left child's and the right hand side of the final BDD will be the operation of the right hand side nodes of the 2 parent; that means, this 2 go to the left operate you are going to get the left child in the resulting tree this 2 parent be ready same level go to the right child of both the both the nodes operate them you are going to get the right child operation, you will get the right child of the in the parent tree.

But the that is what they are saying apply of lo of r f lo of r g and right side means that is the one it will be operation of code of operation high of r f that the high of the parent and high of the second parent prompt be this is simple if the 2 nodes are of the same level problem will happen if basically you have something like say there is a node r f.

(Refer Slide Time: 22:03)



Operations on ROBDD-*apply*

apply(op,B_f , B_g) Algorithm :

1. If both r_f and r_g are terminal nodes with labels l_f and l_g, respectively compute the value l_f op l_g and the resulting OBDD is B_0 if the value is 0 and B_1 otherwise.

In the remaining cases, at least one of the root nodes is a non-terminal.

2. If both nodes are x_i-nodes (i.e., non-terminal of same order), create an x_i-node n (called r_f,r_g) with a dashed line to ***apply*** (*op, lo(*r_f*), lo(*r_g*)*) and, a solid line to ***apply***(*op, hi(*r_f*), hi(*r_g*)*).

3. If r_f is an x_i-node, but r_g is a terminal node or an x_j-node with $j > i$, create an x_i-node n (called r_f ,r_g) with a dashed line to ***apply***(*op, lo(*r_f*),* r_g) and a solid line to ***apply***(*op, hi(*r_f*),* r_g).

4. The case in which r_g is a non-terminal node, but r_f is a terminal or an x_j-node with $j > i$, is handled symmetrically to case 3.

And there is a node called r g, these are the 2 nodes in 2 d to the in the truth trees, but you see j is greater than i; that means, the level of r j is higher than the level of r i, then what is you are going to do because if they are same level it simple you take both the left child operate take both the right child operate and you are done, but here actually you have to do it in a slightly different way as l f r f is slightly above towards the root.

So, what we will do you will operate, but you will not take the child of r j because r g is always in the in the lower side r f is in the upper hand side of the tree. So, you will not operate the right child of r f with the right child of r g you will not do that rather what we will do you will keep the r j constant and try to operate from the right child of r f then again the right child of r f, then again right child of r f with r g you will keep on doing it till the level becomes equal. So, what you do operate lower value of r f that is corresponding to the left and you operate with r g that is basically this r g you are going to operate on and basically.

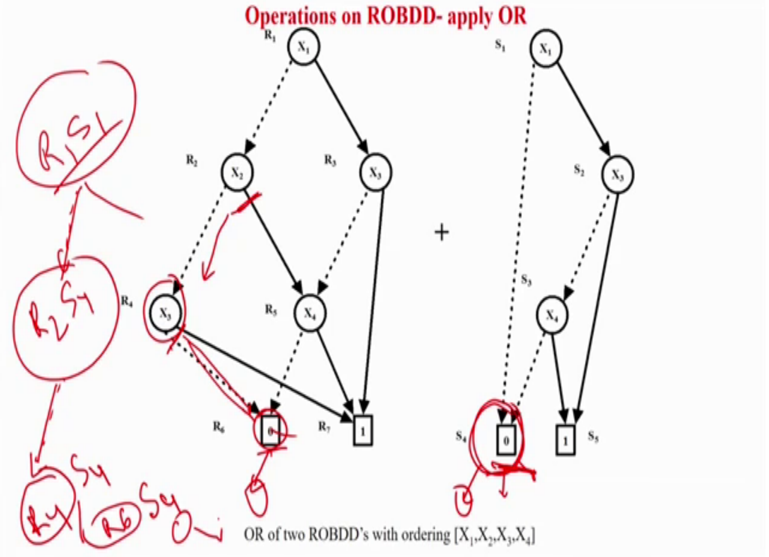
In fact, I was saying that it is basically r g not j r g basically. So, for all such g's and in fact, when you are going to get the right child you will apply optimisation code high of r f basically that is the right side of r f, but you are not going to take any child of r g you will repeat it as long as both the levels are equal. So, what happens basically just look at the mathematics you can read it, I will just give you the gist what is happening r f at the top r g at the bottom.

So, you are going to take the left child of this operate with r g you will not use the child of r g similarly for the right side you will take the child of r g, but operate directly with us you are not going to take the child of r g because r g is quite lower in place corresponding to r f height from the tree from the root. So, the number of depth of r g is much lo this is r this is the root here you have r f and below r g this is something like this. So, r f is at a higher level. So, what happening is basically you have to bring them to the similar ordering level then only you can operate.

Before that you have to just take the one which is more near to the leaf node or which is more further away from the root node that you have to keep constant and you have to take the left child right child left child right child of r f which is more towards the root. So, that they come to a compatible level. So, similarly it can be simultaneously done like what it is saying if r f and r g are vice versa. So, this is actually symmetric. So, in gist what happens?

(Refer Slide Time: 24:41)

Operations on ROBDD- apply OR



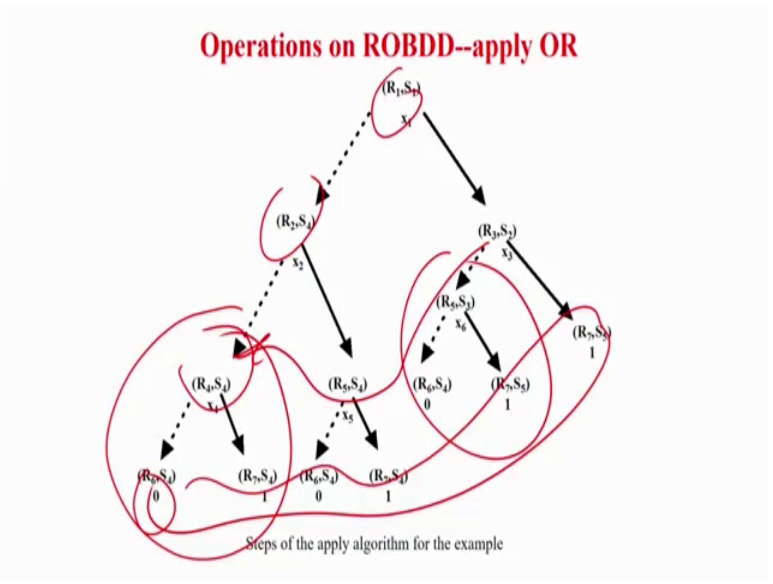
OR of two ROBDD's with ordering [X_1,X_2,X_3,X_4]

So, in gist take 2 root nodes, then if the nodes are at the same level you take the left child of this left child of this final operate get it right child of this right child of this done, you keep on doing it as long as the it will terminate when you operate to terminal, it may always happen that one node is more towards the towards the root node in one tree and the other one will be more towards the leaf node that is levels are not similar, then what you will do you take the child of this node operate on this do not take the child of a lower level tree lower level node, basically, similarly you keep on doing it till they come in the level showing with an example that is the gist you keep on doing it.

Now I will give you with an example which will make the things very clear. So, this the tree I think you can see and you have to do basically and or so, what we will do you will just tell me that x 1, x 1's, these are the 2 nodes name is R 1 and S 1, just I will zoom it for you, see can see they are zoomed versions R 1 and S 1 both are x 1. So, at the same level. So, the operation will be very simple you are going to have a node called R 1 S 1 which is basically nothing, but you have to operate now you are not going to get a that is the or you are not going to get any answer because they are the non terminal nodes.

Next what next I am going to take the 2 nodes basically sorry this is the root node fine, now, what you are going to do now you have to operate on. So, this one is the starting point, now I have to keep on doing it now basically say I have to now, the I will go to show you the structure.

(Refer Slide Time: 26:15)



Steps of the apply algorithm for the example

So, this is the R one and S one. So, this I am already gone done this is root node both are the same level. So, this is done now you just see the next is R 2 and S 4. So, next I have to find out. So, this is the root node corresponding to tree. Now I have to say that I have to find out the next level. So, what is the next level here it is R 2 and here it is S 4, you can see, this is the terminal nodes.

So, it is S 4 till now, it is fine, but now there is a problem, there is an issue, now till now at this point the levels were similar, both were at the same level. Now I am coming to another level where there is a level difference. So, what is the level difference it is saying that it is R 2 and this is S 4. So, the level difference will start coming up from here. So, next is I have I have the next is basically R 2 is the left child of this and S 4 is the right child of this. So, you get a node over here up to it is fine, now if I start going for the higher next level.

So, next level what you are going to see you are going to see that this is x 2 that is nothing, but R 2 and S 4 that is your terminal node now there is a level difference. So, now, you cannot do like previous step previous step what you have done as x 1 and x 1 are at the same level that is R 1 and S 1 of the same level. So, you were taking this and you are going to take this and you are going to find out a node called R 2 S 4, but now you cannot do like that now, what is going to happen, now as I told you, the S 4 is always at a very lower level correspond compare to R 2. So, now, the next one you are going to have these are all dotted lines left ones are dotted just take it.

So, now what is going to happen now you are going to have from R 2 you are going to have R 4 that is the left, but here I will actually not go even if is a leaf node, but what I will do is that I will not take any left or right, rather, I will keep it as S 4 that is here I am taking the left child, but here I am keeping a constant at the constant level because in the second tree I have always further down in the level, but here I have to come down. So, it will be R 4 and R 4 and S 4. So, here I will come over here, but here it will be remaining over here next from here also you are going to take the value called left.

So, it will be R 6m sorry m it will be R 6 and of course, it is going to a left child, but here actually is a leaf node you cannot go at any left child, but assume that if you even if its further upper in the level and it is a non terminal node still as x 3 is at a higher level corresponding, sorry, towards the if your higher level means it is more towards the root and it is more down the leaf. So, you cannot take any left or right child from here till this first tree actually comes to the same level.

So, as of also it is not coming to the same level. So, from R 4 you are going to go to R 6, but S 4 will remain constant because x 3 is more near towards the leaf correspond compare to the S 4. So, now, both of them have come to a leaf level now you can do the operations. So, 0 dot 0 is nothing, but equal to 0.

So, this actually computation, but more importantly to emphasize that it may not be a terminal node it may be some non terminal node also in that case you are going to have R 6 and S 4. Now next step, you may have if you have something like this then these 2 nodes these 2 paths will be operated on and you are going to get a new tree new node again repeating quickly just a gist what we have told you that you have to always maintain the labels.

Operation is very simple, first you will start off with leaf nodes root nodes they are similar level; so, no problem. Secondly, you are going to have a path of combination of this 2 that is the left child of x 1 that is R 2 and the left S 1 that is S 4, you are going to have a node like that after that the interesting things come because S 4 and R 2 are not at the same level. So, what you will do the next node will be R 4 combined with again S 4 because they are not in the same level next, what we will do again you are going to take the left of x 3 and still it will be the S 4 only because now the levels are same now you are going to do an operation you are going to get the value 0, but had repeat not be a terminal node after which in this if I consider this a non terminal node then from here you could have taken the left of this and left of this and find out a . So, you have to wait till the labialisation happens and, but if you read a terminal nodes, you to stop over it, very simple over there.

So in fact, this part will be very similar you will have a single node structure over here both x 3. So, the right hand part will be operation of R 3 and S 2 and x 3, you are going to have a similar level. So, both this left and this left will be there here you are going to have a right node this was the levels are same. So, there is no problem. So, right path there is always maintaining a level same level over here. So, there is no question of any waiting the operation is at the level by level, but in the left part as I have shown you till you maintain till the both the operating nodes come at the same level. So, here the order level means ordering level is 1 2 3 and 4 and then root basically. So, you have to wait till both the levels are come otherwise one guy only will come down and other has to wait till there. So, this is the structure.

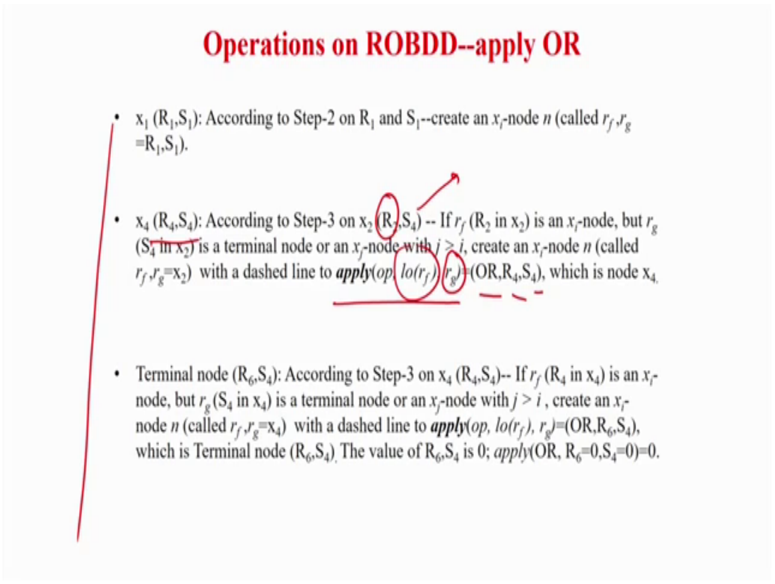
So, R one and S one then next is R 2 and S four. So, this one basically this is the tree R 2 and S 4 then again this one is R 4 and S four. So, as I told you. So, this is R 4 and S 4 and then finally, R 6 and S 4 that is sorry that is R 6 and S 4. Now the labialised is happen by chance this is both are terminal nodes. So, and 0 and 0 the answer is a 0 the right hand side, already I told you it is a very normal way of building the tree here you can see the right side is R 3 S 2 same level.

So, it is R 3 S 2 and this is R 5 and s. So, it is R 5 and S 3. So, similarly this is R sorry this is R 5 and S 3 is this stuff and then again it will be R 6 S 4. So, same level R 6 and S 4. So, both R basically 0. So, 0 or 0 is answer is a 0 so, this is a 0, but if you take the right side. So, that one is going to become a R 7 and S 5. So, R 7 and S 5 that is equal to and of 2 ones. So, the answer is a 1. So, this is how the tree will be built up BDD will be, but up.

Now, here we will be finding out that there will be lot of redundancies like. So, many leaf nodes will be similar ones will have come up. So, that you have to reduce. In fact, you will find out that if you try to calculate many of the basically the internal nodes will also be redundant like as I as you can very easily I can see that may be this part and this part looks somewhat similar type of BDD's. So, all the redundancies you have to eliminate typically the leaf node should have redundancy this part and this part more or less similar. So, I can have a path from here to here. So, all the redundancies steps you have to eliminate now, then you can go for another operation.

So, as I have shown you the way of operation is very simple take 2 compatible nodes do the operation non compatible nodes means non labialised nodes you have to keep on doing left right left right till the operation comes that when both the nodes are at the same level. So, if some is lower in the level to the root only the left will come, then it will have the same level then you can again go on repeating with the same level of operation. So, this keep on going as the as the tree you have done and finally, whenever you will have both the operating nodes are terminal nodes, then you get the values and you stop, but there will lot of redundancy inculcated in this process like many leaf nodes will come up internal node structure as I have shown you over here are redundant. So, you have to eliminate the redundancy.

(Refer Slide Time: 34:04)



Operations on ROBDD--apply OR

- x_1 (R_1,S_1): According to Step-2 on R_1 and S_1--create an x_i-node *n* (called r_f,r_g =R_1,S_1).
- x_4 (R_4,S_4): According to Step-3 on x_2 (R_2,S_4) -- If r_f (R_2 in x_2) is an x_i-node, but r_g (~~S_4 in x_2~~) is a terminal node or an x_j-node with $j > i$, create an x_i-node *n* (called r_f,r_g=x_2) with a dashed line to ***apply***(*op*, *lo*(r_f), r_g)=(OR,R_4,S_4), which is node x_4.
- Terminal node (R_6,S_4): According to Step-3 on x_4 (R_4,S_4)-- If r_f (R_4 in x_4) is an x_i-node, but r_g (S_4 in x_4) is a terminal node or an x_j-node with $j > i$, create an x_i-node *n* (called r_f,r_g=x_4) with a dashed line to ***apply***(*op*, *lo*(r_f), r_g)=(OR,R_6,S_4), which is Terminal node (R_6,S_4). The value of R_6,S_4 is 0; *apply*(OR, R_6=0,S_4=0)=0.

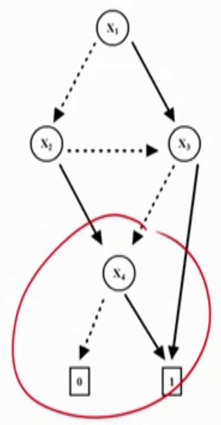
One very important thing I as tell you here you have to reduce all the redundant nodes at operating any 2 BDD’s, before you go for operating the third BDD because if you keep the redundancies things will keep on going up. So, operate reduce operate reduce this way you go. So, I have no stage in time, you are going to have a very big that is what is the idea of BDD’s. So, whatever I told you in English is written in the slides like, say for example, how we have taken R 1 and S 1, then by step 2 of the algorithm in which case both the nodes are labialised. So, you operate R 1 and S 1 and you are going to get the next child that is basically equal to R 4 and S 4.

So, whatever I have told you how the operations are done and if the input is this you start with R one and S one that is the root node; so, you are going to get the root node like this there is basically R 4 and S 4, how do you get R 4 and S 4 you are going to get if you are operating state 3 on R 2 and S 4 because R basically S 4 is the node which is quite down the level compared to sorry S 4 is more down to the leaf node compared to R 4. So, R 4 will basic sorry S R 2 will change to R 4, but S 4 will remain constant. So, which are the step algorithm step I have used that is the step number 3 and so forth.

So, this slide basically tells you whatever I have told you that what are operations being done that it is saying that op of low of r f, but r g is keeping in constant because you are going to take the left right child of r f that is nothing, but R 2, but S 4 is keeping constant because it is down more downwards the leaf node. So, you are going to operate or on R 4 and S 4 R 4 is nothing, but the left child of R 2, but S 4 is the minimal constant. So, whatever I told you the steps how they are some of the steps are being illustrated in this slide you can read it this is just the mathematical representation you can easily correlate.

(Refer Slide Time: 35:56)

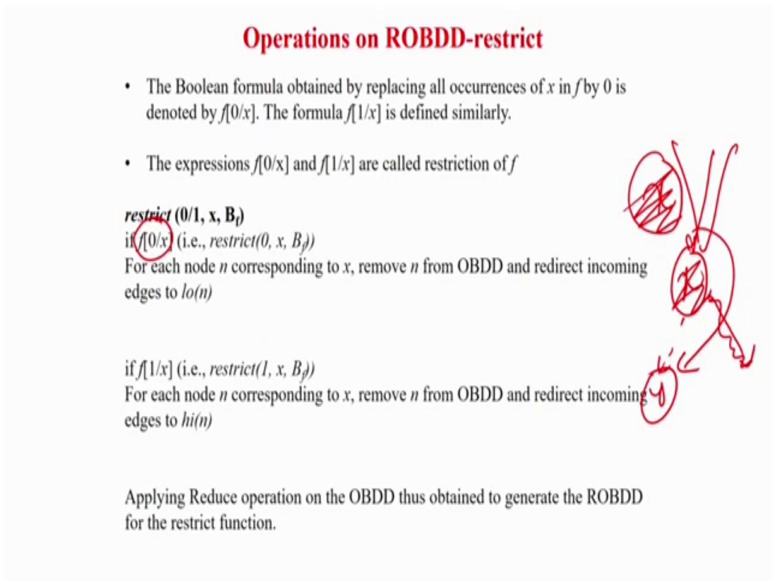
Operations on ROBDD-apply OR



Final ROBDD

So, this is the final o BDD as I told you that there was some similar looking structure in the BDD. So, I have eliminated the 2 similar looking BDD’s that is the sub BDD’s which were similar and I am getting to get a final reduced order BDD now I can do another operation on this that is not a problem it will again may have some more extra redundancy if I do another operation you eliminate and keep on doing.

(Refer Slide Time: 36:17)



Operations on ROBDD-restrict

- The Boolean formula obtained by replacing all occurrences of x in f by 0 is denoted by $f[0/x]$. The formula $f[1/x]$ is defined similarly.
- The expressions $f[0/x]$ and $f[1/x]$ are called restriction of f

***restrict* (0/1, x, B_f)**
if $f[0/x]$ (i.e., *restrict(0, x, B_f)*)
For each node n corresponding to x, remove n from OBDD and redirect incoming edges to *lo(n)*

if $f[1/x]$ (i.e., *restrict(1, x, B_f)*)
For each node n corresponding to x, remove n from OBDD and redirect incoming edges to *hi(n)*

Applying Reduce operation on the OBDD thus obtained to generate the ROBDD for the restrict function.

So, that is what is the idea of BDD construction using operations you have a very big circuit take a take a gate take the inputs constructive BDD for the output of the gate reduce the redundancy there itself then you can take another gate in the forward and find out the BDD for the next level of gate and keep on doing it for every gate computation output of BDD, you have to reduce it at that stage only, you it is not like that you will computer for 2 or 3 gates and then reduce at a time if you do this the intermediate BDD's will become very large in size and your whole idea will be lost.

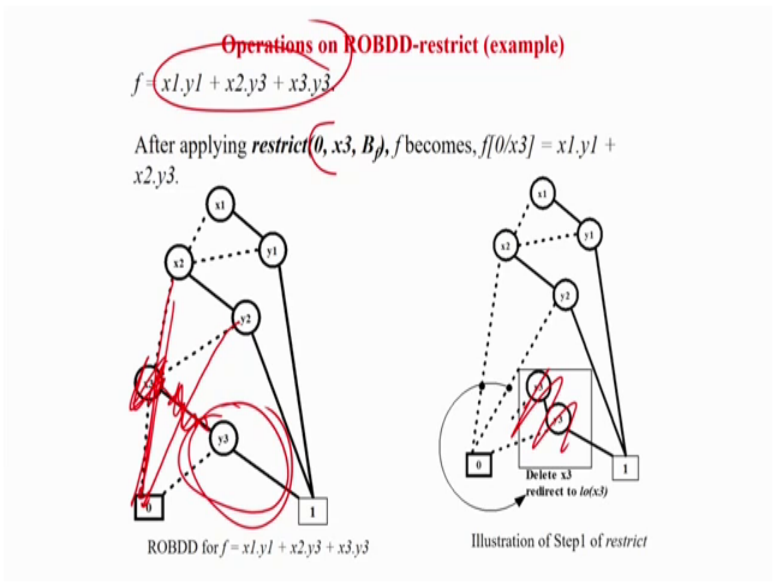
Then one more interesting operation on BDD's called restrict. So, restrict means basically you want to see if I make one variable 0 how the BDD or where are the binary function looks like if I want to make one variable one can the you can ask many questions like if this variable is 0, then what will be the BDD or what will be the binary Boolean function look like can it still be one and if it is still 1, what variable has to be one and. So, forth like if you want to see that this 2 BDD's are not equivalent or this 2 functions are not equivalent, but what if I make y equal to 0 that for example, I assume that for circuit or a system the y is always 0 under that circumstance whether the 2 BDD's become equivalent.

So, that is some very important operation in final verification which is called restrict in restrict means you stop the value of a variable at 0 or 1 and try to study the BDD's itself. So, that is actually called restrict. So, restrict 0 or 1. So, if I want to say that I want to restrict x 2 0. So, what we have to do. So, if there is any variable called x node corresponding to this you eliminate that node and redirect on incoming edges to low off in. So, if there is a variable called x node x. So, this may be the lower path there may be another node called y.

So, all you have to eliminate this and all paths coming to this node has to be redirected to the left child of x; that means, because this one is eliminated because I will not consider in its BDD of sub BDD which corresponds to y equal to x equal to one. So, I will eliminate this and all edges I will redirect over here similarly for the restrict 0 in restrict . So, basically this is sorry this is restrict 0. So, I will do it in this one restrict 0 means you want to study the value of the Boolean function if x is equal to 0 if if you want to restrict it to one; that means, I will make x equal to one then I want to study how the BDD looks like as the example I have given you that 2 circuits whether they will behave the similarly if one input is stuck at one.

Stuck at or in the sense, I restrict it to 1 because I know that that is a reset line and the reset line will always be equal to one for certain purposes. So, under that circumstance whether the 2 circuits are equivalent in that case basically you go for the restrict operation. So, basically in restrict operation we take the node which has to be restricted eliminate it and all the incoming nodes will be direct to the left child or to the right child depending on whether its restrict 0 or whether it is a restrict one example.

(Refer Slide Time: 39:06)



Operations on ROBDD-restrict (example)

$f = x1.y1 + x2.y3 + x3.y3.$

After applying ***restrict*(*0, x3, B$_f$*)**, f becomes, $f[0/x3] = x1.y1 + x2.y3.$

ROBDD for $f = x1.y1 + x2.y3 + x3.y3$

Illustration of Step1 of *restrict*

Like this is a function and I want to basically apply x 3 equal to 0; that means, I want to see if x 3 is equal to 0, how the circuit will be or the function will look like and so forth, there can be different purposes how to do that. So, basically what they will do they will actually eliminate this x 3 and whatever incoming edges from this will be redirect directed to here and of course, this will be no longer there. So, this guy will be hanging you have to eliminate it.

So, that is what has been done you have taken this one from here this one all the incoming edges to this one to the left because you are assuming that x 3 is equal to 0. So, everything will be redirected over here it had one path to the rights which corresponds to x 3 equal to one that you eliminate. So, this is a sub BDD which is hanging which you have to eliminate it and then basically you are going to get a to eliminate it this is what is your BDD will be looking like.

Now, it is I have told you we may have another similar function you want to find the equivalence of this or satisfiability validity whatever by making x equal x 3 equal to ; that means, I want to see the behaviour of some function or circuit if x is restricted to this one. So, these are very important operation. So, like add subtract multiply you can easily do by the off the algorithm I have told you for apply restrict algorithm I have told which is very obvious complement is a very obvious algorithm now there can be lot of other operations of BDD like xor satisfiability, whether they are exist, but everything can be implemented in terms of this basic operation.

So, that is why I am not going into that you have to take some combinations of this and you are going to find out the BDD which will actually take that implementation xor are very straight forward some there they are exist for all. So, all such operations can be done in basic in by the way of this apply restrict and your complement operation. So, basically before going to the high level decision diagram let us come to again a gist what we have understood binary decision tree is very good well accepted standard problem is that exponential complexity.

So, not even circuit with ten gates or ten inputs can be handled what or can be verified that that because a system with ten input variables the order of 2 to the power ten came to a boon is BDD, BDD will actually take a basically you started with explanation side you take a b d binary decision tree is there you keep on eliminating the redundancy you are going to get something called a binary decision diagram compression is as high as ninety nine percent plus for most of the cases. So, BDD become a defect to standard of where you can represent all the systems in a very compressed manner.

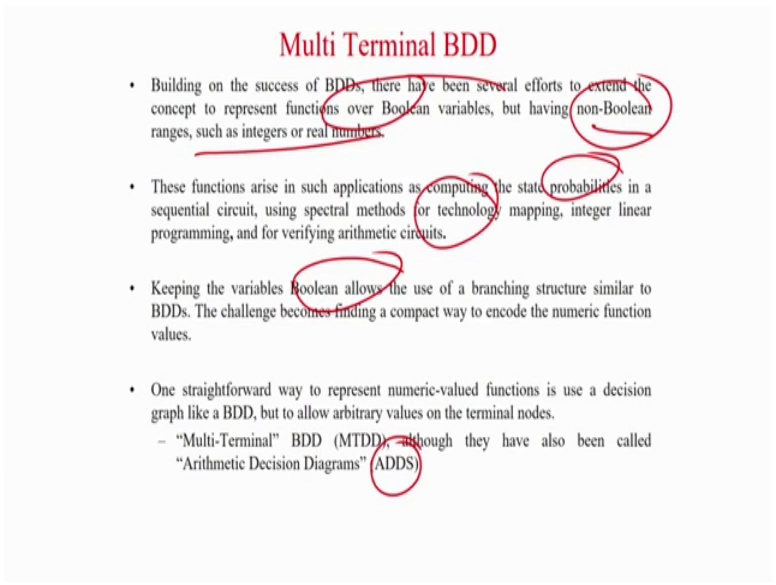
Now, the next the question came is that from where I will construct the BDD from a circuit or from a binary decision tree by using Shannon's expression we want to do it the idea is that how we can build the binary decision tree itself is not possible quite large. So, how we have to do we have to go by the operation method you take any circuit take the input BDD operate and get the final BDD of the output of the gate then eliminate the redundancy there itself because I have shown you operation trade redundancies for operation 2 important things are there, the ordering you have to maintain and after every operation, you have to go and eliminate the redundancies.

Here the redundancies will be slight, it will not be like it will become a binary decision tree slight in slight increase in the number of leaf nodes and intermediate nodes may happen which you have to eliminate, you have to keep on doing it after every operation, if you keep on doing it never at any point of your time BDD will be larger. So, at every operation or after every gate of gate the output BDD will be small inside. So, whenever you get the BDD for the entire second it will be with the manageable .

But still how long if you are going to go for a very big circuit like NOC and SOC even the BDD is going to fail people have found out that typically hundred input circuits if you are directly going to apply BDD there will be lot of issues. So, they break up the circuit into codes of inferences some divide and concurrent policy they do, but anyway they cannot handle a very large circuit like NOC and SOC. So, people said that we have to go away from the bit level to higher level as we have already seen in case of testing that we have to go from stuck at faults or bridging faults to higher models of faults like which are more near to software engineering and you have to go for register transfer level modelling similarly in the case of binary decision diagrams.

So, decision diagrams lots you can find out in the lot of papers, but the godfather is Bryant, he gave the idea of this compression after that people have used it for different kind of binary decision diagram like arithmetic decision diagram high level decision diagram etcetera and there are. So, many, but mainly in this class we will construct around ADD and HDD.

(Refer Slide Time: 43:33)



Multi Terminal BDD

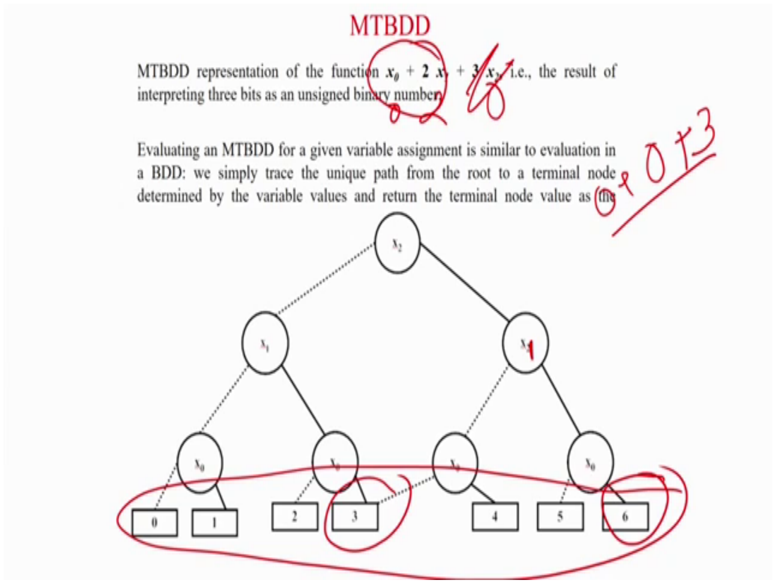
- Building on the success of BDDs, there have been several efforts to extend the concept to represent functions over Boolean variables, but having non-Boolean ranges, such as integers or real numbers.
- These functions arise in such applications as computing the state probabilities in a sequential circuit, using spectral methods for technology mapping, integer linear programming, and for verifying arithmetic circuits.
- Keeping the variables Boolean allows the use of a branching structure similar to BDDs. The challenge becomes finding a compact way to encode the numeric function values.
- One straightforward way to represent numeric-valued functions is use a decision graph like a BDD, but to allow arbitrary values on the terminal nodes.
  - "Multi-Terminal" BDD (MTDD), although they have also been called "Arithmetic Decision Diagrams" (ADDS)

So, one the First one is actually called the multi terminal BDD. So, what is a multi terminal BDD, basically if you have a function in which the outputs are non Boolean generally we may have if you want to do some expression on 3 plus 5 and we want to. So, the answer is in integer if you want to go for BDD you have to make represent 3 as binary 5 as binary and things will start blowing up. So, multi terminal binary the idea is say that it is very similar based on the success of BDD, there has been several efforts to extend the concept to represent over Boolean variables, but having non Boolean ranges the first is something like that BDD is very good x is 0, x is 1, x 2 is 0 1 that is binary enumeration of the variables will be there or the answer may not be represent in binary it can be 2 3 4 5 or integers that is why they call it as a multi terminal BDD.

So, functions of a Boolean variables, but output is non Boolean they may be integers or real's where they are apply many places like probability technology mapping program verification arithmetic verification etcetera. So, many n number of places you may find out because if I represent the numbers in binary size will be larger if I can represent in decimal the smaller. So, therefore, the called lead are the multi terminal BDD or sometimes it may also call it as arithmetic decision diagrams because we have in the terminals you are going to have basically the integers or real numbers.

So, this are the very straight forward approach keeping the variable Boolean. So, if you keep the variable Booleans you can have the same branching structure like a BDD or a binary tree or the or the same philosophy you can apply, but the number of leaf nodes will be more and will be more compress because if you are going for a binary. So, 4 will be a implemented by 1 0 0 5 will be 1 0 1 and things will start blowing up.

(Refer Slide Time: 45:17)



MTBDD

MTBDD representation of the function $x_0 + 2x_1 + 3x_2$, i.e., the result of interpreting three bits as an unsigned binary number.

Evaluating an MTBDD for a given variable assignment is similar to evaluation in a BDD: we simply trace the unique path from the root to a terminal node determined by the variable values and return the terminal node value as the

So, that is what is the idea of multi terminal BDD or arithmetic decision diagram and example is better to illustrate because operating on this BDD or multi terminal BDD also possible compression is also possible reduced ordering is also possible all the stuff we have seen in BDD's are also applicable over here, but I am not going into the details because that is more involved compared to BDD, but the idea is similar take 2 nodes operate take the right child maintain the labialisation the algorithm will remain same, but slight integrities will come in. So, without details we are not discussing in the course, but we are trying to give the philosophy how it evolved. So, there is a function which is actually you can see it is x 0 2 x plus 3 x.

Now, x 0 x 1 and x 2 are binary variable either 0 and 1, but 2 and 3, I am not going to represent as a binary I will keeping at a decimal. So, how the tree look like it is is very interesting to see the tree is something like this 2, 1 and 0, this is the level. So, what I am having x is equal to 0 x x x 2 equal to 0 x 1 equal to 0 x 3 equal to 0 all 0, the answer is a 0 because x 0 x 2 into 0 3 into 0 all 0, but if the value is x 2 equal to x 0 equal to 1 x 0 x 1 equal to 0; that means, 2 into 0 is 0 x 2 is also 0 3 into 0 0 the answer is a 1. So, 0 0 x 0 the answer is a 1, I will take some other path it will be more interesting.

Let us take the case x 2 equal to 0 sorry x 2 equal to 0 x 1 equal to 1, x 2 equal to 0 means this one is equal to 0 x 2 equal to 1 means 2 2 into 1 is 2 and again x 0 is a 0. So, the answer is 0. So, 0 plus 2 plus 0 is equal to 2. So, this is the path and you can see, I have a leaf node which is equal to 2, had it been binary things would have blown up another interesting case x 2 equal to 1 x 2 equal to 1 means 3 into 1 is 3, then x 2 equal to 0; that means, x sorry x 2 equal to 0 means sorry sorry sorry this one is x 1 extremely sorry, this one is x 1 same labialisation.

So, x 2 is equal to 1; that means, the answer is a 3 x 2 a sorry x 1 is equal to 0; that means, the second term is 0. So, the answer is 0 finally, x 0 is also equal to 0. So, 0 plus 0 plus 0 is 3. So, this 3 is a i is a redundant node. So, I could also have a 3 a here, but actually I do not require a 3 over here because already the leaf node is their redundancies eliminated.

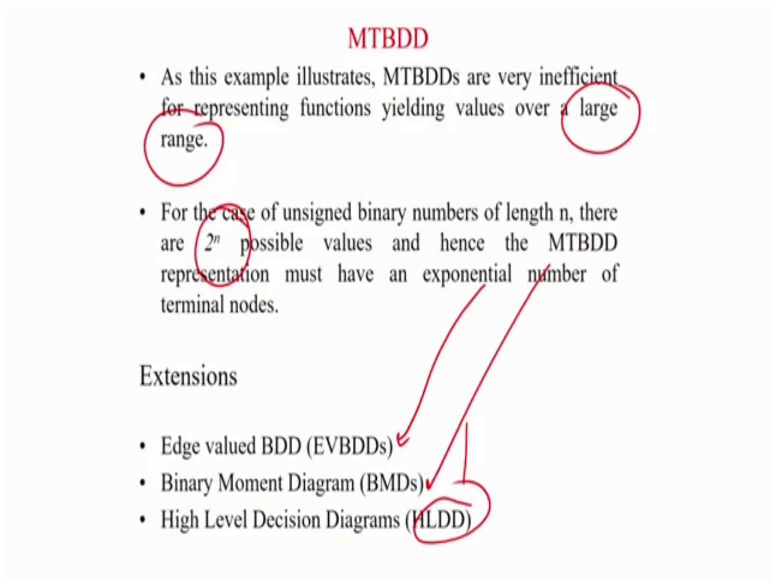
So, because another way I can get tree because x 2 is equal to 0 x 1 equal to 1 and x 0 equal to 1. So, this term goes 2 plus one becomes one. So, that is 3. So, that is another path which is the redundant leaf node. So, I am actually merge them. So, you get the structure very similar to BDD the architecture is very very similar to BDD the logic is very similar to BDD here, I have only shown how the leaf node because the maximum value can be one plus 2 plus 3 that is equal to 6. So, the highest value of the leaf node is 6. So, this is actually a multi terminal BDD you can express any kind of integers there.

So, the leaf nodes are not one, but not two, but rather more compared to each of integer variables, but if you can see the structures somehow looks like a binary decision not a binary decision diagram it looks a binary decision tree, but some philosophy I have applied in the leaf node. So, the answers are eliminated over here. So, people actually started with when people moved away from BDD to abstract level they started something called a multi level BDD just like it philosophy will remain same. They will take a binary decision tree and they will start applying the philosophy of redundancy elimination in slightly roundabout may not in as simple as you can do it in BDD for multi terminal BDD the elimination of redundant nodes etcetera is slightly non straight forward.

So, you can will put up the references in the way. So, which you can which you want to see, but basically the idea is similar you take a BDD you can reduce the if it is a binary decision diagram concept of BDD will reduce redundancy and make it compact same thing, they have tried over here they have started a multi terminal binary decision tree not a diagram then they will start reducing the redundancies and they will get a very compact representation which will be very synonymous to binary decision diagram, but with multiple terminals.

So, that is what people call as the arithmetic decision diagram that is what is the idea have not shown the details steps of how redundancies can be eliminated accepting in the leaf node, but the structure is slightly I mean involved because here is not 0 and one you are having multiple values, but still the same philosophy will be applied and you are going to get a very compressed structure. So, that is one school of thought which actually started moving from BDD's to arithmetic or multi terminal diagrams, that is more abstract or compact representation so.

(Refer Slide Time: 50:00)



MTBDD

- As this example illustrates, MTBDDs are very inefficient for representing functions yielding values over a large range.
- For the case of unsigned binary numbers of length n, there are 2^n possible values and hence the MTBDD representation must have an exponential number of terminal nodes.

Extensions

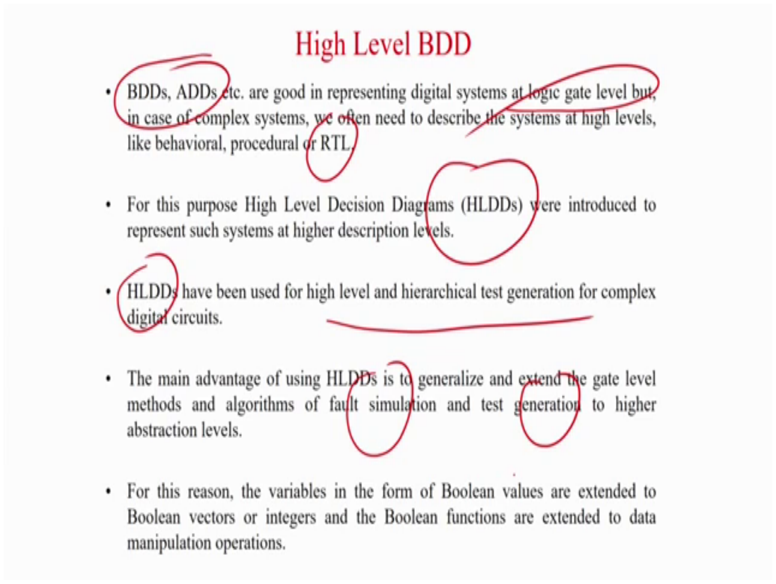
- Edge valued BDD (EVBDDs)
- Binary Moment Diagram (BMDs)
- High Level Decision Diagrams (HLDD)

Ah the problem here is that multi terminal BDD's can implement a large range because 2000, 3000, 4000, this integer can be there, but the problem is that there is 2 two to the power n possibilities because is its looks very similar to a for the philosophy is very similar to a binary decision tree, but idea is that that is what this are the some extensions like edge valued BDD binary moment BDD's they actually take the philosophy of binary decision diagram and applied on multi terminal BDD and they compress it and finally, people found out that such philosophy can be applied, you can get a what you call they call it binary moment diagram not actually binary decision diagram which is very similar to a binary decision diagram well compressed no redundancies, but you can still represent integers, but as I told you these mathematics behind is quite involved.

So, you can look at it, but in this course we are just going to give a given the idea how it happens by actually eliminating the redundancy at the leaf node. So, that it can motivate you that the same philosophy can be applied rather what we will go to is something the most advanced version one which is actually called the high level decision diagram that. In fact, rather than trying with this, binary variables itself because in this case the x 0 x 1 x 2 are binary variables.

There are then go in binary and taking the output as decimals or real the whole thing can be itself.

(Refer Slide Time: 51:22)



High Level BDD

- BDDs, ADDs etc. are good in representing digital systems at logic gate level but, in case of complex systems, we often need to describe the systems at high levels, like behavioral, procedural or RTL.
- For this purpose High Level Decision Diagrams (HLDDs) were introduced to represent such systems at higher description levels.
- HLDDs have been used for high level and hierarchical test generation for complex digital circuits.
- The main advantage of using HLDDs is to generalize and extend the gate level methods and algorithms of fault simulation and test generation to higher abstraction levels.
- For this reason, the variables in the form of Boolean values are extended to Boolean vectors or integers and the Boolean functions are extended to data manipulation operations.

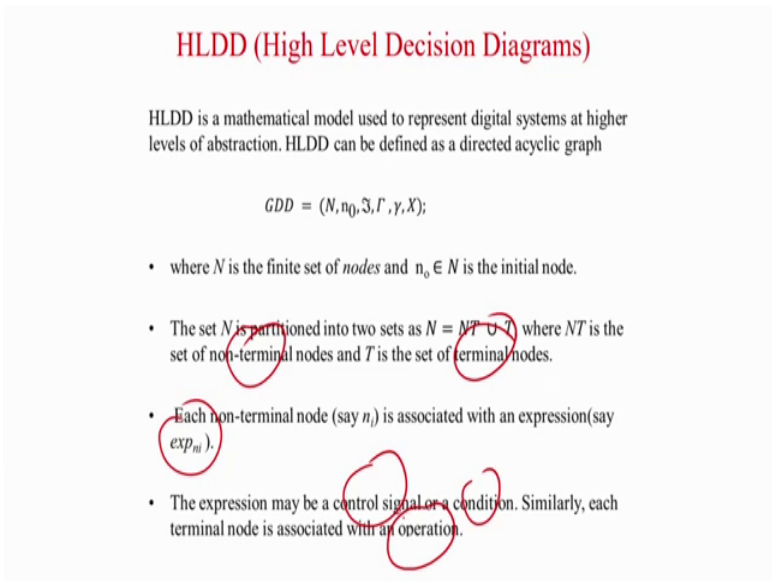
Converted into a high level abstraction that is the most advanced version of so called most advanced means most published high level binary decision diagrams or high level decision diagrams is what is the most advanced or it can handle the most complicated kind of circuits by the philosophy of high level decision diagram that not even the input variables will be binary it is that everything will be at the RTL level at the c functional level or the very log level or the hardware definition level. So, that is why you are going to look more into HLDD. So, as I told you BD is an HLDD are good for digital systems are logic gate level, but wherever you go for abstract system of systems like NOC and SOC no way you can look at binary you have to go at RTL.

So, the HLDD's or high level decision diagram sometimes we call HDD, sometimes we call at HLDD were introduced 2 knows that gap that if you have a HDL representation or RTL representation, how do you go about it. So, HDL is are very very popular for high level and hierarchical test generation, it can be very easily you can represent or model circuits at the RTL level. So, it is can be used for fault simulation test generation a normal means verification of circuits at the RTL level.

So, when you are verifying circuitry RTL level we assume that when the RTL level will be converted to gate level the equivalents will not be lost because nowadays from RTL to gate level conversion is more or less automated by some of the algorithms which Professor Chandan sir has already taught you that express O 2 level minimisation and so forth. So, given the RTL the gates can be always obtained by some state forward algorithms which are known to be correct by construction.

So, we are not much bothered at the equivalence level at the gate and. In fact, you cannot do that it is infeasible taking consideration of the size of NOC's and SOC's. So, what best we can have we can have the equivalents tests at the RTL and below that we assume that as they are more or less mechanised. So, equivalence will be there. So, therefore, basically the Boolean values are now extended to basically high level high level means representation in terms of RTL. So, that is why variables in form of Boolean values are now extended to vectors Boolean vectors or integers that is and Boolean functions are extended to data manipulation operations; that means, we are all moving it from binary or Boolean level to functional level.

(Refer Slide Time: 53:43)



HLDD (High Level Decision Diagrams)

HLDD is a mathematical model used to represent digital systems at higher levels of abstraction. HLDD can be defined as a directed acyclic graph

$$GDD = (N, n_0, \Im, \Gamma, \gamma, X);$$

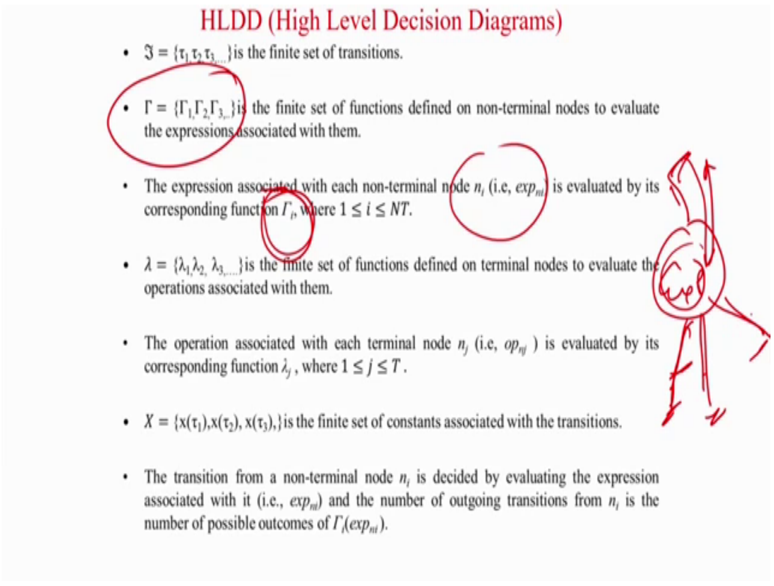
- where N is the finite set of *nodes* and $n_0 \in N$ is the initial node.
- The set N is partitioned into two sets as $N = NT \cup T$ where NT is the set of non-terminal nodes and T is the set of terminal nodes.
- Each non-terminal node (say n_i) is associated with an expression(say exp_{ni}).
- The expression may be a control signal or a condition. Similarly, each terminal node is associated with an operation.

Rather, I will just very quickly I means go into the mathematical definition which I will I will try to deal with bit quickly because this can you read it offline and understand what it means rather I will focus more on examples. So, what is a HLDD? So, basically it represents it is representate by this formula GDD, it has some nodes initial nodes some transitions some functions within the nodes some other variables like output function and all. So, it is a 1 2 3 4 6; 6 tuple.

So, now what is n? N is the set of nodes n and n is the initial node that is always there then n is partitioned in 2 sets here actually one important thing is there here the nodes will have terminal and non terminal nodes is non terminal node has an expression called exp n i. So, do not think about the mathematics that you can read offline just you think that it is a graph and there are some nodes some nodes are terminal and some nodes are non terminal that is leaf nodes and non leaf nodes and each node non terminal node, we will have an expression and each terminal and each expression may be a control signal or a condition.

And terminal nodes basically have a operation and each terminal node is associated with an operation like add or subtract and each non terminal node basically is a control signal or or a condition like signal input signal a equal to one input signal a equal to 0 input signal a is or a is greater than b, some conditions checks based on the input signals or the register values will be the non terminal nodes terminal nodes will be some operation a or b, b or c, c or d which will update the register it is something like depending on some of the some of the variable values like a is greater than b, a is less than b; that is more or less the register outputs that will be represented by the none terminal nodes and based on the conditions some operations happen which update the value of the registers. So, that will be represented by the terminal nodes just take it that way.

(Refer Slide Time: 55:38)



HLDD (High Level Decision Diagrams)

- $\Im = \{\tau_1, \tau_2, \tau_3, ...\}$ is the finite set of transitions.
- $\Gamma = \{\Gamma_1, \Gamma_2, \Gamma_3, ...\}$ is the finite set of functions defined on non-terminal nodes to evaluate the expressions associated with them.
- The expression associated with each non-terminal node n_i (i.e, exp_{ni}) is evaluated by its corresponding function Γ_i, where $1 \leq i \leq NT$.
- $\lambda = \{\lambda_1, \lambda_2, \lambda_3, ...\}$ is the finite set of functions defined on terminal nodes to evaluate the operations associated with them.
- The operation associated with each terminal node n_j (i.e, op_{nj}) is evaluated by its corresponding function λ_j , where $1 \leq j \leq T$.
- $X = \{x(\tau_1), x(\tau_2), x(\tau_3), \}$ is the finite set of constants associated with the transitions.
- The transition from a non-terminal node n_i is decided by evaluating the expression associated with it (i.e., exp_{ni}) and the number of outgoing transitions from n_i is the number of possible outcomes of $\Gamma_i(exp_{ni})$.

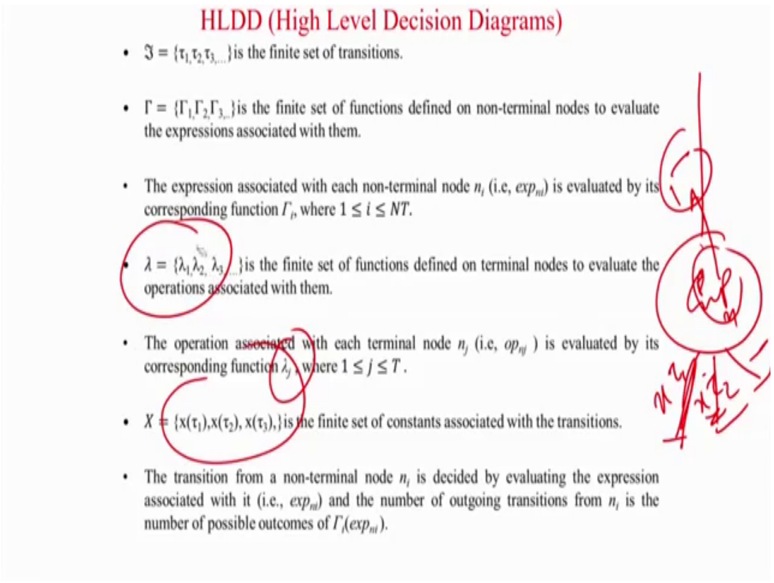
Then there are of course, transitions there are of course, transition they represented by the transitions and the another important thing which are which are additional basically compared to any normal finite state machine or be if any structure there are some finite functions dependent on the non terminal nodes to evaluate the expression associated with them that is the non terminal nodes will have some functions like which I have told over here which will see while other they will be used then they are the expression associated with each non terminal node n I is abbreviated by tau i.

That means, if you have a non terminal node, then you will have some expression internally that is expression gap and there is a function this one which will evaluate this expression and based on the evaluation basically you are going to take the outgoing transition if this function on this expression is true for this path it will take this or it will take this or transition will have some fixed marker condition whose truth will tell whether it will go or not; that means, these non terminal and expression is fixed the function which is which each non finite non terminal node.

It will operate on that and it will find out some answer if that matches with this transition it will fire otherwise the other one and so forth and output of each node as I told you, if you look at this these are the 2 these are 2 things I have discussed they say that the third point says that these are some of the finite set of functions defined on the terminal nodes that is as I told you each of the terminal node will do some operations. So, this will be depending on that one the operation associated which is terminal node is evaluated by the function.

Now, one more important thing that is each of this state will have a finite set of constant associated with transitions these are some of the constants; that means, as I told you there is a state.

(Refer Slide Time: 57:30)

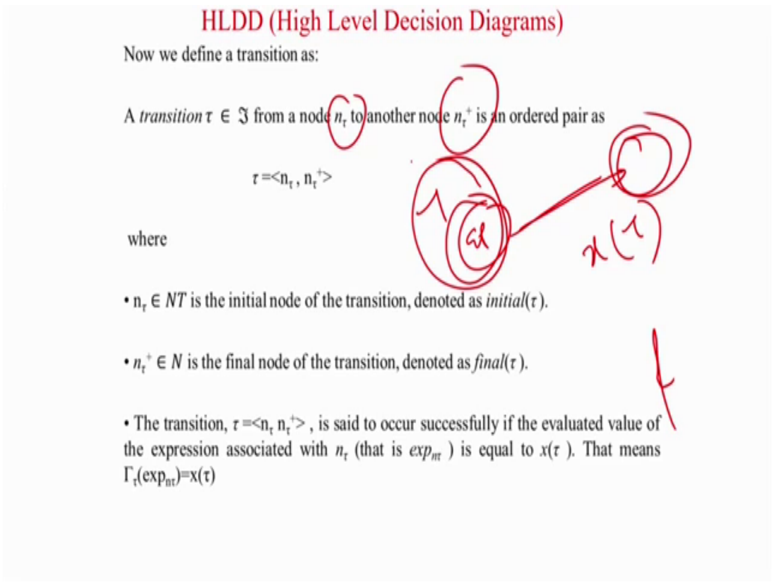


HLDD (High Level Decision Diagrams)

- $\Im = \{\tau_1, \tau_2, \tau_3, \ldots\}$ is the finite set of transitions.
- $\Gamma = \{\Gamma_1, \Gamma_2, \Gamma_3, \ldots\}$ is the finite set of functions defined on non-terminal nodes to evaluate the expressions associated with them.
- The expression associated with each non-terminal node n_i (i.e, exp_{ni}) is evaluated by its corresponding function Γ_i, where $1 \leq i \leq NT$.
- $\lambda = \{\lambda_1, \lambda_2, \lambda_3, \ldots\}$ is the finite set of functions defined on terminal nodes to evaluate the operations associated with them.
- The operation associated with each terminal node n_j (i.e, op_{nj}) is evaluated by its corresponding function λ_j, where $1 \leq j \leq T$.
- $X = \{x(\tau_1), x(\tau_2), x(\tau_3), \}$ is the finite set of constants associated with the transitions.
- The transition from a non-terminal node n_i is decided by evaluating the expression associated with it (i.e., exp_{ni}) and the number of outgoing transitions from n_i is the number of possible outcomes of $\Gamma_i(exp_{ni})$.

And there is some expression this expression will be evaluated by this and there will be some outward transitions and the constants basically like x of tau one x of tau 2 this state is x tau 2 some conditions will be or some constants will be assigned to each of the transitions. So, if this expression on this function corresponds to this condition if this one will fire otherwise this will be fire; so forth. So, this is way the things will go on this is the mathematical representation I mean just as we all defined such complex mathematics before we go to a real example.

(Refer Slide Time: 58:02)



HLDD (High Level Decision Diagrams)

Now we define a transition as:

A *transition* $\tau \in \Im$ from a node n_τ to another node n_τ^+ is an ordered pair as

$$\tau = \langle n_\tau, n_\tau^+ \rangle$$

where

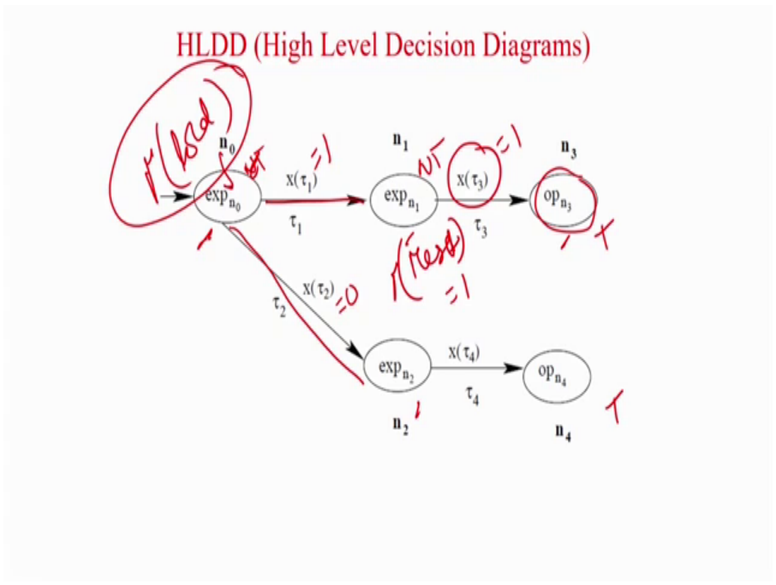
- $n_\tau \in NT$ is the initial node of the transition, denoted as *initial*(τ).
- $n_\tau^+ \in N$ is the final node of the transition, denoted as *final*(τ).
- The transition, $\tau = \langle n_\tau, n_\tau^+ \rangle$, is said to occur successfully if the evaluated value of the expression associated with n_τ (that is $exp_{n\tau}$) is equal to $x(\tau)$. That means $\Gamma_\tau(exp_{n\tau}) = x(\tau)$

So, this just the mathematical if do not go means go means just think that it is very complicated it is just a mathematical way of representing an HLDD.

So, when I will just tell you the example I will show you an example things will become very very clear, but when you are talking some formalism of decision diagram anyway you have to tell the formal definitions. So, a transition basically from one node n tau to n tau plus basically is generating is always something happens that it will always from a non terminal to a terminal or a non terminal to a non terminal. So, this actually the initial tau this, the final tau and each transition basically we have some expression over here this is a function and there is a constant x of tau over here that is what is being told over here.

So, this one will evaluate this and if this one will match this constant then the transition is going to fire that is what is being told by this formal definition. So, the transition tau is said to occur successfully if the evaluated value of the association expression this one on tau is matching to x of tau this the capital tau, then basically that is what is being told over here that if expression of tau evaluates to equal to x of tau; that means, this is x. So, this is x of tau, then this condition will take place example.

(Refer Slide Time: 59:21)

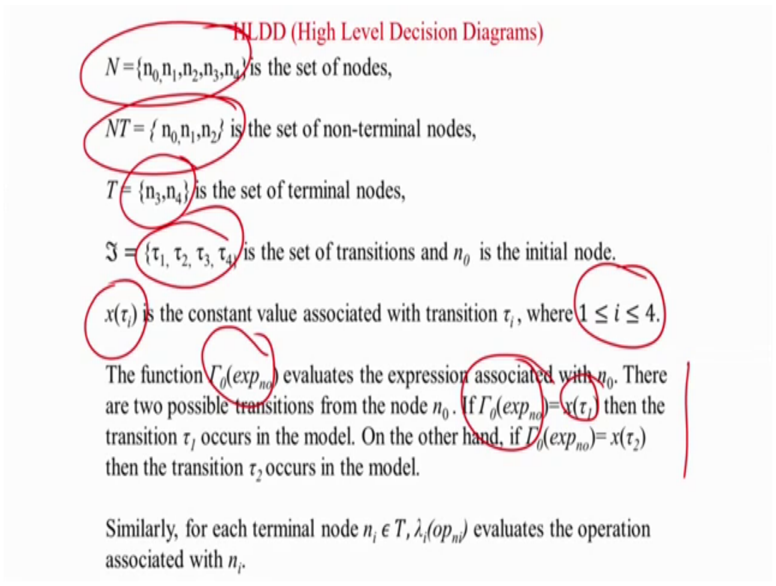


So, this is basically your graph. So, as you as I told you these are the or the these are the non terminal nodes non terminal nodes these are the terminal nodes these are the terminal nodes. So, now, as you can see each of the non terminal nodes have some expressions and sorry this is terminal node terminal node has a operation operation n operation n 4, there can be a one plus a 2 a 2 plus same thing like that non terminal means some expression.

So, expression may be a one greater than a 2 or it can be just one signal itself like reset. So, it will evaluate whether reset is true or not and this x of tau 3 is a constant for example, is I say that expression 3 is reset and this one x 3 is equal to one so; that means, what this is going to fire if and only if reset is going to be a true; that means, tau of this one is going to be a; that means, tau of capital tau of n one node will check whether reset is one or not if it is one it is going to fire similarly here I can say that there is a signal called hold.

So, capital tau of this of hold will be evaluating let this be equal to one x of tau one is equal to one x of tau 2 this is the constant value is equal to 0. So, this transition is going to fire if the hold signal is one and this one is going to fire if the hold is equal to 0 now who is going to verify this tau function which is expression with this one here expression n 0 is nothing, but here hold . So, which very very simple on that the few slides, I had shown before is the mathematical way of representation. So, whenever you are going for verification when you are going for binary decision diagram representation you have to go for the automata theory way of representing.

(Refer Slide Time: 60:49).



HLDD (High Level Decision Diagrams)

$N = \{n_0, n_1, n_2, n_3, n_4\}$ is the set of nodes,

$NT = \{n_0, n_1, n_2\}$ is the set of non-terminal nodes,

$T = \{n_3, n_4\}$ is the set of terminal nodes,

$\mathfrak{I} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ is the set of transitions and n_0 is the initial node.

$x(\tau_i)$ is the constant value associated with transition τ_i, where $1 \leq i \leq 4$.

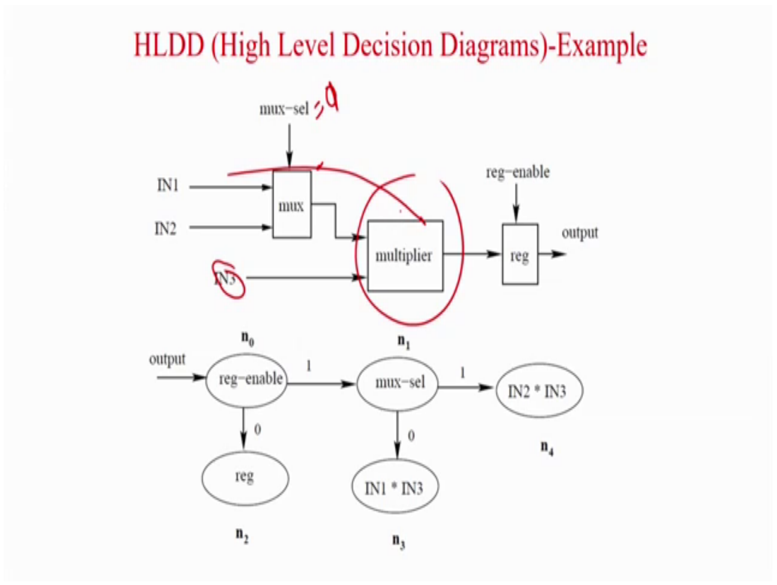
The function $\Gamma_0(exp_{no})$ evaluates the expression associated with n_0. There are two possible transitions from the node n_0. If $\Gamma_0(exp_{no}) = x(\tau_1)$ then the transition τ_1 occurs in the model. On the other hand, if $\Gamma_0(exp_{no}) = x(\tau_2)$ then the transition τ_2 occurs in the model.

Similarly, for each terminal node $n_i \in T$, $\lambda_i(op_{ni})$ evaluates the operation associated with n_i.

So, this is just the example I have told you. So, it says that what are the set of nodes these are the set of non terminal nodes n 3 and n 4 are the set of terminal nodes what are the transitions what are the constant values the x i is constant value then basically how this one tells that how the function is evaluated like if tau expression evaluates with n if the value of tau naught 0 expression is equal to tau of x 1 then expression one will take place otherwise the expression 0 will take place that is what I have expect that is written in this slide in text.

So, that you can read in when you are doing the lecture because whenever we are trilling the example it is simple to understand, but when we are going try to correlate mathematics with it then it requires some slow phase of understanding. So, you have to just read through the slide and try to correlate anyway the nodes will also be there.

(Refer Slide Time: 61:36)



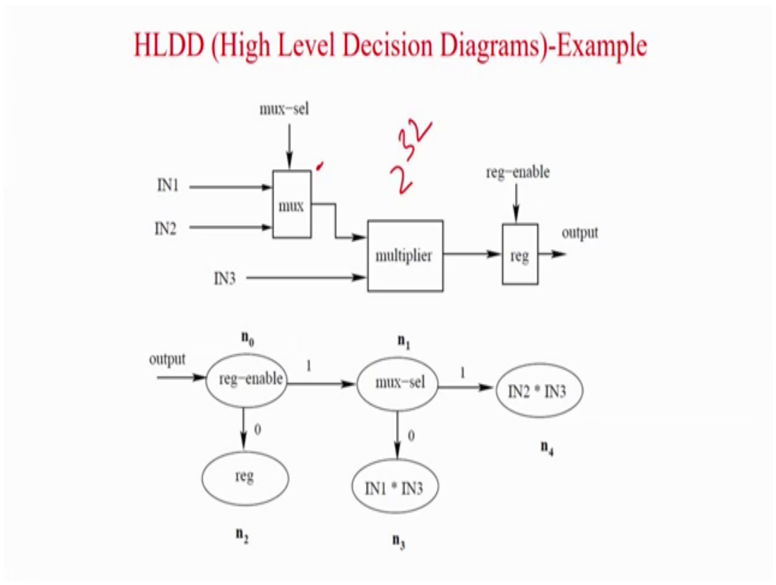
HLDD (High Level Decision Diagrams)-Example

Just finally, before we close down with the decision diagram see example this is a simple multiplexer circuit there is a multiplier over here. So, what it will multiply it will either multiply in 3 and it depends whether in one or in 2 will go because it is a multiplexer is over here this is a multiplexer. So, the multiplexer will actually tell you whether in one will go or in 2 will go and the multiplexer will this multiplexer will be either multiplying in one its selects in equal to 0 then in one is going to come it will be multiplying with in 3 and if this is equal to 1 and in 2 will come and you are going to get the multiplication and you are going to basically get the answer.

So, that is what is the circuit and there is of course, a register which will latch the value if the register enables signal is equal to one.

Now, internally you can see multiplier can be very very be it can be a 32 bit multiplier.

(Refer Slide Time: 62:69)


HLDD (High Level Decision Diagrams)-Example

So, it can have multiplication values from 32 bits means 2 to the power 32 can be the range order of that. So, you can understand if it is a multi terminal BDD how many nodes it will have infeasible numbers. So, how it, but here it is only a simple structure because is an RTL level. So, the whole data path has been actually compressed all this already we have basically discussed when we are discussing RTL level testing that if in a data path if you are going to explicitly enumerate the numbers. It just going to blow up even if it is a BDD, it is a multi terminal BDD forget BDD multi terminal BDD.

BDD means 2 to the power 32, even BDD fails for such high input size and BDD never works for multiplier is already been seen that compression is always very good, but for multiplier circuit BDD generally does not work that well does not work means compression is not that efficient. So, in that case basically anyway those part aside. So, here the multiplier is given as we have seen, but when you are taking an RTL, it is as simple as what I have shown over you just 5 nodes because it is at a RTL level means it is quite high abstraction and. In fact, it is the only optimisation this is the only way of optimising circuits that is going from circuits bits to RTL level is only way if you want to model large system like NOC's and SOC's.

So, what you see these are the 2 terminal nodes this corresponds to the multiplication answer evaluation. So, n 2 into n 2 into n 3 and this one is n 1 into n 3 because as I told you there only 2 ways of doing it n 3 is constant, it will be multiplied either with n 1 or n

2, then next what. So, these are the non terminal nodes. So, register enable is the expression behind this and if I consider the function tau over here, it just checks whether the value is 0 or 1 just reads and find out what we enable is one or not.

If the enable value is 0 then register is just going to hold the old value. So, therefore, this terminal node the expression is reg, this actually evaluation a kind of dummy evaluation that it just retains the old value if register enable is equal to 1, then only it is going to latch. So, then I am going in this direction if the multiplier select is equal to one; that means, this terminal is going to come in you are going to get n 2 into n 3.

So, this is what is the expression if mux is equal to 0 n o is going to come and the output is equal to n one into n as simple as that. So, even RTL circuit just you have to model it in terms of this high level decision diagrams and lot of, but one thing I should tell you it is not as convenient as BDD because till now there is no very widely accepted result that it is canonical or whether it how it can be reduced or how equivalence can be studied.

So, they are people are all trying to work out the good things about BDD's canonically canonicity or if you take 2 orders if there is equivalent the system will look similar redundancy if you eliminate then you are going to get a minimum structure for that function and so forth. So, all thing operations on BDD all the Boolean operations can be done. So, now, people are trying to work find out algorithms that how those philosophies can be moved to high level decision diagrams.

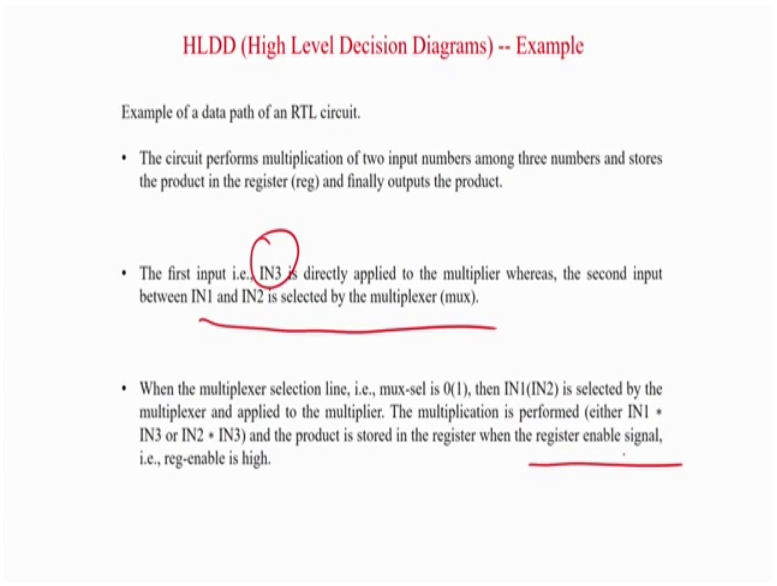
So, as I told you that these are all not much these are not much I mean there is not much published work or well accepted work that how this can be directly mapped into the philosophy of BDD they are all open problems. So, one of the idea of doing this course is that to lead you into good research problems.

So, one of the research problem is that how you can verify large circuit at RTL level because at the gate level BDD is there lot of tools are available over there which Professor Chandan has already discussed with you, but when you go for the abstract level you will have a binary decision diagram or a high level decision diagram arithmetic decision diagram, but we are do not we are I mean such type of well established algorithms which will prove the canonical that the or the algorithms which you are based on the properties of canonicity or if the ordering is same you will same BDD.

So, all those philosophies actually are not that way widely accepted or not yet published. So, the idea is set that you have to go at the RTL level, you have to go at the high level decision diagrams, but what will be the exact structure of the high level decision something like this or slightly something has to be modified. So, that the structure you are going to get is canonical and if you and if you take a same variable ordering you are and if the 2 functions are similar, you are going to get a similar HLDD which are the most important parameters required for verification hold or not.

So, the research you have as a students or I mean industry people or researchers you have to try think again that direction that how I will go for high level decision diagrams and still go still have the philosophy of BDD built in it ok. So, this slide is nothing, but whatever I have told you.

(Refer Slide Time: 66:48)



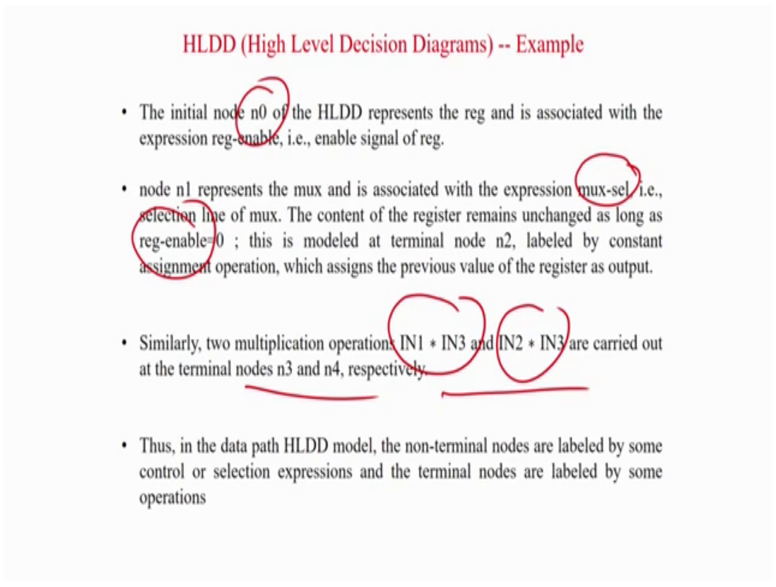
HLDD (High Level Decision Diagrams) -- Example

Example of a data path of an RTL circuit.

- The circuit performs multiplication of two input numbers among three numbers and stores the product in the register (reg) and finally outputs the product.
- The first input i.e., IN3 is directly applied to the multiplier whereas, the second input between IN1 and IN2 is selected by the multiplexer (mux).
- When the multiplexer selection line, i.e., mux-sel is 0(1), then IN1(IN2) is selected by the multiplexer and applied to the multiplier. The multiplication is performed (either IN1 * IN3 or IN2 * IN3) and the product is stored in the register when the register enable signal, i.e., reg-enable is high.

In English that basically that how this BDD has been constructed is written over here that is basically that is it has first input is a n, it is always there is a multiplexing in between this then how basically how the register enables signal, etcetera.

(Refer Slide Time: 67:07)

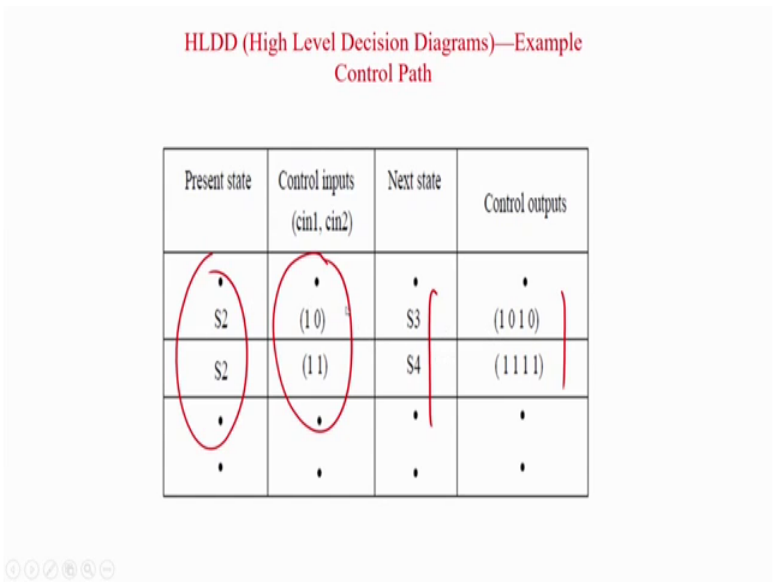


HLDD (High Level Decision Diagrams) -- Example

- The initial node n0 of the HLDD represents the reg and is associated with the expression reg-enable, i.e., enable signal of reg.
- node n1 represents the mux and is associated with the expression mux-sel, i.e., selection line of mux. The content of the register remains unchanged as long as reg-enable=0 ; this is modeled at terminal node n2, labeled by constant assignment operation, which assigns the previous value of the register as output.
- Similarly, two multiplication operations IN1 * IN3 and IN2 * IN3 are carried out at the terminal nodes n3 and n4, respectively.
- Thus, in the data path HLDD model, the non-terminal nodes are labeled by some control or selection expressions and the terminal nodes are labeled by some operations

And how the BDD is basically constructed that is explained in these 2 slides, it tells that n node is the initial number. So, the 2 multiplication operations this and this are done by the terminal nodes and all the non terminal nodes basically have these signals called multiplexer select and basically the register enable. So, that is the signals which corresponding to the non terminal node these are the operations which corresponds to the terminal nodes.

(Refer Slide Time: 67:33)



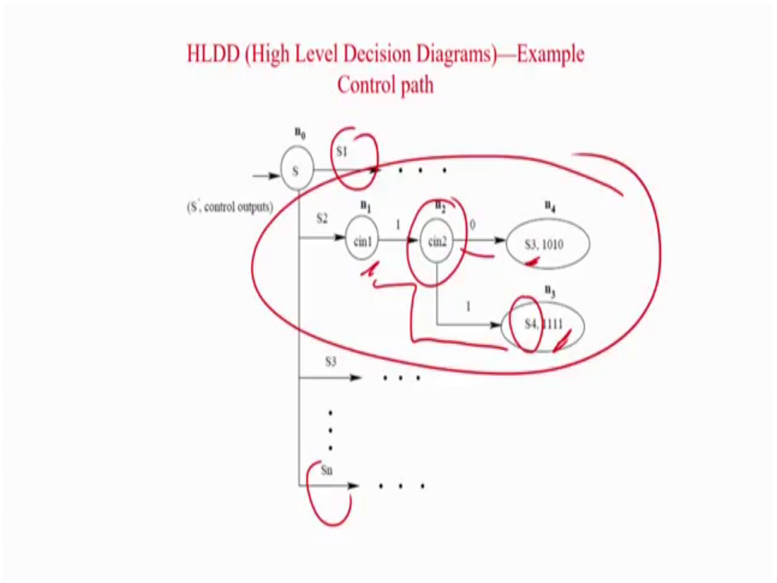
HLDD (High Level Decision Diagrams)—Example
Control Path

Present state	Control inputs (cin1, cin2)	Next state	Control outputs
•	•	•	•
S2	(1 0)	S3	(1 0 1 0)
S2	(1 1)	S4	(1 1 1 1)
•	•	•	•
•	•	•	•

So, whatever I have told you in the example is written in the slide. So, that you can read offline and come out with it one just before I close down basically this was about the data path because already when we have talked about testing we know data path is very very difficult to do because now 32 bit, 64 is starts going up, but similarly there is also a control path the control path can also be very easily modelled using BDD or HLDD because if you have a circuit one part you are modelling as BDD other part you are modelling an HLDD does not look good.

So, some people have also tried to taught that thing how we can represent control paths in terms of HLDD be, but control circuits are generally smaller BDD's will suffice, but just they have given the example just to complete for the sake of completeness. So, say that there is some state which are present state some control inputs are there these are next state and these are the output how do you represent very simple basically obvious it will be just like a case structure.

(Refer Slide Time: 68:19)



HLDD (High Level Decision Diagrams)—Example

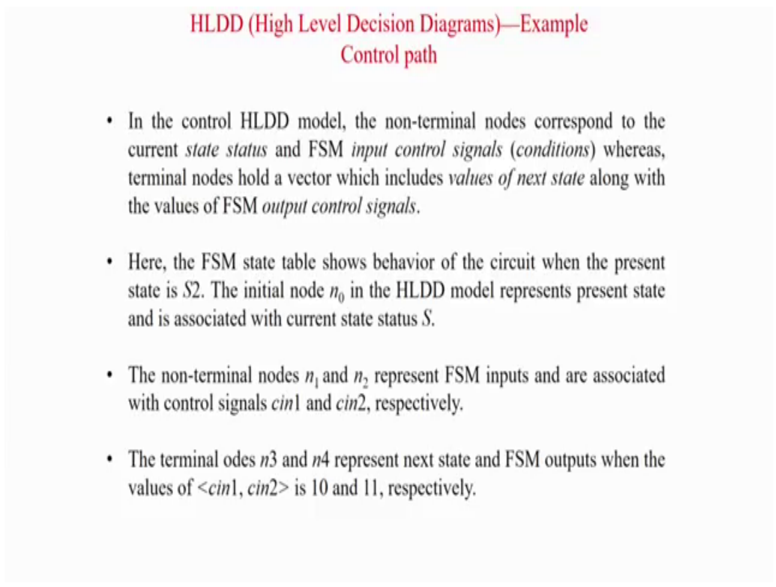
Control path

So, the initial node and may be this may be the S 2 is the current state we are talking about because for that example now are present state is S 2 and we are going to see what is the next state depending on the control variable. So, this is initial state these are for state its one up to state S n we are just going to look at for state S 2. So, S 2 these are non terminal node which corresponds to signal c 1 will be another non terminal corresponding to signal S 2 what are this terminal nodes here doing the terminal nodes here will be the outputs and what are the next state. So, ones next state is S 3 and the other next state is S 4.

So, in case of data path the non terminal nodes are expression and the terminal nodes are some operation data operation a plus b, a minus b, etcetera, but here they are basically which next step to who and what are the control outputs. So, for example, if c n equal to 1 and c, n equal to ,0 you are going to get S 3 in the next state and the output is 1 0 1 0 basically that is what is 1 0 1 0 and next state is S 3 similarly if i c 1 is 1 and c in 2 is 0 you are going to next state is S 4 and the control signals are 1 1 1 1.

So, basically that is what. So, this is just a I should call it as a modelling, it is just a translation from the table to a high level decision diagram, but in fact, basically those things can also be done using a binary decision diagram also. You need not go to high level decision diagrams because generally controls circuits are generally smaller the problem only happens with the data path already we have elaborated this states in the test module.

(Refer Slide Time: 69:32)



HLDD (High Level Decision Diagrams)—Example
Control path

- In the control HLDD model, the non-terminal nodes correspond to the current *state status* and FSM *input control signals* (*conditions*) whereas, terminal nodes hold a vector which includes *values of next state* along with the values of FSM *output control signals*.
- Here, the FSM state table shows behavior of the circuit when the present state is *S*2. The initial node n_0 in the HLDD model represents present state and is associated with current state status *S*.
- The non-terminal nodes n_1 and n_2 represent FSM inputs and are associated with control signals *cin*1 and *cin*2, respectively.
- The terminal odes *n*3 and *n*4 represent next state and FSM outputs when the values of <*cin*1, *cin*2> is 10 and 11, respectively.

So, basically again the whatever I have told you control path has been designed the high level decision control path has been designed is this line you can read it. So, anyway this 2 lecture series actually brings us to the end of decision diagram for verification we started with binary decision tree then we have seen for BDD's then we have seen that what are the gains we have got and. In fact, we should always be thankful to professor

Brandt who invented such a nice data structure and in fact, all, but as circuit started becoming larger the bit level did not work then people started moving into arithmetic decision diagram high level decision diagram etcetera, but the main philosophy still remains the same as BDD's redundant reduce redundancies first thing and the structure should be canonical what do you mean by canonical the canonicity means basically if the ordering of the variables remain same and the 2 functions are similar you are going to get the same BDD that is what is a very very important property of BDD that is very much required for equivalence checking.

So, therefore, HLDD is a long way we have come from BDD, but still some main philosophies of BDD's still needs to be proved on this. So, as researchers I want you to please think and work on those directions that at higher level, how we can have a canonical high level diagram high level diagram how I can do verification using those structures.

Thank you and in the next classes onwards, we will try to see how this BDD and HLDD's, etcetera will be used for exactly modelling sequential circuits or state machines because this, we have seen that verification main problem is we are getting stuck with the circuit, I mean model size state model size the BDD's and ADD's whatever we have seen are basically showing how to represent the functions in a compressed manner. Now we will use that philosophy to model the states that is basically a called symbolic model checking bounded model checking were we will see how the finite state machines can be modelled using binary decision diagrams. So, that problem of explosion of the state space modelling of the model for verification is solved.

Thank you.