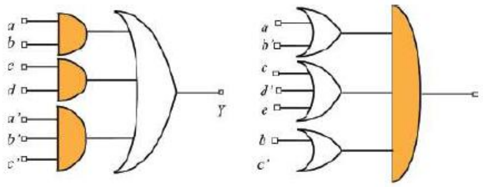
VLSI Design Flow: RTL to GDS
Dr. Sneh Saurabh
Department of Electronics and Communication Engineering
IIIT-Delhi

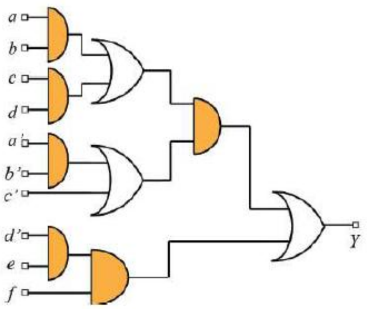
Lecture 19
Logic Optimization: Part II

Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the 15th lecture. In this lecture, we will be continuing with the logic optimization. In the last lecture, we had discussed that there are two types of logic optimization or two types of combinational logic optimization, two-level and multi-level logic optimization. So, in the last lecture, we had looked into two-level logic optimization and in this lecture, we will be looking at multi-level logic optimization. So, let us first understand what is a multi-level logic.

So, a multi-level logic contains more than two levels of logic. So, if we consider this logic circuit, so in this logic circuit, we are taking the sum of product terms and that is why this is a SOP or sum of product representation and this circuit, we are taking the product of various sum terms and that is why it is a product of sum representation or POS representation. So, these two types of representation, sum of product and product of sum, these are two-level logics. Why it is two-level logic? Because there are two levels of gates: AND followed by OR or OR followed by AND.



S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.



Now, in multi-level logic, we have multiple levels of logic meaning more than two levels. So, for example, in this circuit, we have AND followed by OR then again followed by AND and then followed by OR. So, there are four levels of logic in this part. There are four levels of logic, which is more than two and that is why this is a multi-level logic. So, when we code RTL, typically multi-level logic comes up naturally.

Why? Because in RTL, we have seen that there can be procedural blocks. So, within a begin and end statement, we can have procedural statements and we can write statements such that one logic drives the next and the next logic level drives the next and so on. So, there is a sequence of logic levels. So, when we write RTL and in procedural blocks, this kind of multi-level logic appear naturally. Now, how to optimize this kind of multi-level logic? One solution could be that we convert every multi-level logic into two-level logic and then probably apply the techniques that we discussed in the last lecture that is two-level logic minimization or optimization.

That could be one of the ways to handle multi-level logic. But there are some limitations of two-level logic and what are these limitations. So, the first limitation is that for some types of functions, the two-level logic for example, sum of product representation like this one, sum of product representation of some logic functions can become too big. For example, if we take parity functions, adders, multipliers for these types of functions, the two-level logic will be very big in terms of area or in terms of representation and then optimizing them may not be that easy. And the other reason is that the two-level logic is very optimal in terms of timing.

Why it is optimal? Because we have from the input to the output, we have only two-level of logic in all the paths, maximum two-level of logic. And that is why we have a very good timing as such because the number of stages or number of gates from input to the output is restricted to be two. And therefore, the signal can propagate from the input to the output very quickly and from the timing perspective, from timing or delay perspective, two-level logic like these two product of sum or sum of product representation, these are very good. But in terms of area, they are very huge. In terms of area, they are huge and additionally the problem is that we cannot trade off timing to improve the area meaning that in a two-level logic, we cannot increase the delay or trade off delay or trade off performance in that sense and gain area or reduce area of the circuit.

But the multi-level logic allows that kind of freedom. Why? Because in the multi-level logic, we can have say three-level of logic, four-level of logic, five-level of logic. As the number of levels increases, the delay from the input to the output increases and therefore timing becomes worse. But as a result, by restructuring or making the transformation in the logic, we can improve the area of the circuit. So, if we want to evaluate area-timing trade off, then multi-level logic is much richer because there is much greater freedom in multi-level logic than in two-level logic.

And that is why we need to do optimization for multi-level logic also besides optimizing two-level logic. Now for multi-level logic, there are two important representations which are factored form and a Boolean logic network. So, we will be looking into these two types of representation of multi-level logic and then we will go into that in these representations, how optimizations can be done or how we can improve area or some other metrics of our design. So, we can define a factored form in a recursive manner. So, how can we define? We can define a factored form as it consists of literals. A factored form can consist of a literal and it can be sum of other factored form. So, it is a recursive definition. Sum meaning logical OR. Similarly, a factored form consists of product of factored form. So, this is a recursive definition and in a sense this represents that in a factored form there can be an SOP of an SOP of an SOP up to arbitrary depth or we can consider a factored form as a POS of POS of POS of an arbitrary depth.

So, what is a factored form? We will just see an example of factored form then it will be clear that what we mean by SOP of an SOP or the factored form contains sum of factored form. Now, given an SOP expression that is sum of a product representation of a logic function that is two level representation, we can obtain a factored form by factoring by the process which is known as factoring. So, what factoring does it converts a two level logic to multi level representation. So, given an SOP we can do factoring and we can get a factored form representation or multi level logic representation. So, now given an SOP, its factored form is not unique.

So, its factored form is not unique, we can get different factored form for the same SOP. Let us take an example suppose we take an SOP that is the sum of product. So, these are product terms and then we are taking the sum. So, it is a sum of product terms. Now, in a sum of product term if for this expression, Boolean function, we can do a factorization and get a factored form.

So, we can obtain a factored form representation as: we factorize it we get as $(a + b).(c + d) + ce$. So, we can consider this as an SOP it is sum of product in which the product is product terms are simple literals and this is another sum of product. And then we are taking an AND of it and then we got a factored form and then we take an OR of it and get another factored form. So, that is what is represented by this definition sum of factored form. So, we take this is a factored form and we take a sum that is we get another factored form.

And similarly, if we take the product of factored form, we can consider this is a factored form, this is another factored form and then we take a product of it then it is another factored form. So, if we try to do factoring for this Boolean function we can get another factored form also. For example, (a + b).d + (a + b + e).c, this is another factored form for the same function. Now, even SOP is a factored form SOP can also be considered as a factored form similarly, POS can also be considered a factored form. So, now for a given Boolean function we can get multiple factored form now which of the factored form is good for us.

So, we have to measure some characteristics or figures of merit of a Boolean function represented in factored form. So, we can quantify the area taken by a factored form by counting the number of literals in the factored form. Why we can do this because if we think of a CMOS implementation of a logic circuit then the area taken by the CMOS circuit very well correlates with the number of literals in the factored form. And that is why if we just count the number of literals in the factored form we can quantify the area taken by the x Boolean function in the factored form representation. For example, if we consider this factored form.

So, we said that SOP is also a factored form we can count the number of literals 1 2 3 4 5 6 7 8 9 10. So, it has got a number of literals as 10. Now for this factored form representation we have 1 2 3 4 5 6, 6 is the number of literals and for this factored form we have 1 2 3 4 5 6 7. So, there are 7 literals in this factored form. So, we can say that the area will be minimum for this representation and maximum for this representation.

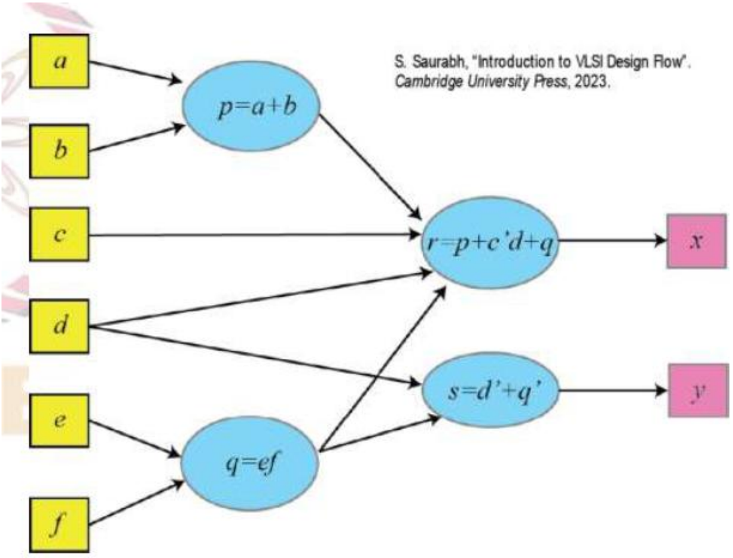
So, why we can say that because see if we try to implement this multi level logic in factored form in CMOS the area will very much correlate with the number of literals. Now another important thing we can see here is with how many number of levels of logic are there in each of these factored form expression. For example, in this SOP expression there are only two levels of logic AND followed by OR for all of them. So, this is a two level logic. Now what about this, here we have one level of logic for both of them when we take an AND.

So, we can draw the circuit and then we can count the number of levels. For example, we can say that we have or of A B and C D and then we have an AND and then we have another OR in which we take an AND of C and E. So, the number of levels in this representation is 1 2 3. So, here we have level as 3 and in this representation how many levels are there one for this, one for this AND and then one for this OR. So, here also we will have 3.

So, we saw that the area is minimum in this and maximum in this and the number of level is minimum in this meaning that the delay will be minimum in this kind of representation in SOP representation and the the delay in the these two representation will be more because they are having three levels of logic. So, what we can say is that if we do factoring and we change a logic level from two level to multi level the area can come down why because the number of literals can be reduced and the delay will increase. So, we have traded off the delay meaning that we have increased the delay we have increased the number of levels of logic and we have decreased the area of the circuit. So, that is what the application of multi level logic is. A multi level logic can actually trade off, can allow us to do trade off delay to improve the area of the circuit as we will see more such examples. Now, another representation that is important is what is known as Boolean logic network.

Now, what is a Boolean logic network? Now, Boolean logic network is a directed acyclic graph. So, in a graph there are vertices and edges. So, in this case what are the vertices? So, in this Boolean logic network is also a graph, but in this the vertices are some local functions and on each vertex we annotate single output local single output local Boolean function. And then we connect one vertex to another vertex to define the dependency among various vertices and these kind of Boolean logic network is very good representation of Boolean logic if we have multiple outputs. There are multiple outputs and they are sharing some logic. So, those kind of things can be very well represented in Boolean logic network as we will see.

So, it can represent conveniently multilevel logic with more than one output. So, if there are more than one output then those output will share some logic elements among themselves and we can represent them conveniently using Boolean logic network. Now, let us look into what is a Boolean logic network? Let us take an example. Suppose we are given a set of equations that is p is equal to a plus b q is equal to e f and so on.



S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

So, these are set of Boolean equations. So, all these are Boolean variables that we have written here and we say that a, b, c, d, e, f are the inputs. Suppose we are saying that a, b, c, d, e, f are independent variables those are inputs of our logic network and we say that there are two outputs x and y. Then how to draw or how to construct the Boolean logic network for this case? So, for first what we do is that for each of the input and the output we create vertices and label them according to their names, the input and the output vertices. Then we create the internal nodes as per these equations that are given.

So, once we do this we get something like this. So, a, b, c, d, e, f these are the input vertices or inputs of the Boolean logic network, we create vertices for them and then we create vertices for x and y. And then we created the internal vertices as shown in this case. For example, we created internal vertices for this equation p, for the next equation q, for r and s. We created vertices for them and in these vertices what notation we do. We have on each vertex, two parts LHS and RHS.

On the LHS we have the name of the node for example, the name of this node is p, the name of this node is q. So, on the LHS we have the name of the node or the label of the node and on the RHS we have the local function. So, this p is equal to a plus b that is what the local function is. So, this vertex represent a local function a plus b and its output is p that is the label of the vertex. And how do we create edges in this graph we create edges based on the dependency.

So, we create the incoming edges to a vertex, denote the variables on which the local function at that vertex depends. For example, in this case the local function is a plus b and it depends on a and b that is why there are two incoming edges. Similarly, if we look into this node r it depends on p. Now the name of the vertex this was p. So, there we create a edge from p to this r and then we have c dash. So, we create an edge from c to r and also from d to r and also it depends on q. So, we create an edge from q to r. Similarly, we do for q and s and so on. So, vertices denotes the dependency among the vertices and the label in the vertex denotes the name of the vertex and the local function. Now why do we want to use Boolean logic network? The reason is that it is very flexible in terms of representation.

Logic circuit is also kind of a Boolean logic network, but in logic circuit we have individual gates that can be the local functions. For example, AND gate, OR gate whatever we have chosen. However, in Boolean logic network we do not have that kind of constraint. The local function can be anything, any complicated Boolean function can also be a local function for a vertex. So, that is why it is more flexible than the logic circuit.

So, local functions in a Boolean logic network can be arbitrarily complicated and both its underlying graph and the local functions can be manipulated. When we say that we can manipulate the underlying graph we can say that we can manipulate the incoming edges, outgoing edges and the dependencies among the vertices, those can also be manipulated in a Boolean logic network and of course, the local function that we have at each vertex that local function can also be manipulated as we will see. So, optimization can explore both the behavioral and the structural features of the implementation. So, because of flexibility of the Boolean logic network we can explore both the behavioral aspect of the Boolean function and the structural aspect.

Now, given a Boolean logic network how do we estimate the area and delay? So, whenever we do a transformation or optimization of a Boolean logic network, we need to measure the area and the delay of the representation. Whether we are improving, we are degrading or whatever we are doing we have to measure it. Now, to allow easy measurement of the area and delay of a Boolean logic network what we do is that we apply some constraint. We say that for the local function we restrict it to be an SOP expression, sum of product only.

So, we say that we can have an arbitrary complicated expression or arbitrary complicated Boolean function as a local function, but the representation of this local function will always be SOP. And another constraint we apply or we say that the constraint is that this SOP representation is minimal with respect to single implicant containment. What we are saying is that if a there is an expression say y is equal to one product term, another product term and so on. So, these are product terms and then we are taking a sum.

So, this is a SOP representation and this is how we represent the local function. Now, we say that this SOP representation will be such that it will be minimal with respect to single implicant containment. Meaning we can consider this product terms as implicant. We had discussed the definition of implicant in the last lecture. Now, if we consider this as implicant then any of this product term or any of these implicant should not be contained by any other implicant of this SOP representation. So if we have an expression like ab+abc’ for example.

Now, this implicant (abc’) is fully contained by this implicant (ab). So, we will not allow this to exist we will say that we will drop this (abc’) in the local function. Why we are doing this? Then if we the local function to be in SOP form and minimal with respect to single implicant containment then we can easily measure the area and delay and how we can do it? We can estimate area just by counting the number of literals in the Boolean logic network. We just count the number of literals and that will give an estimate of the area. Total number of literal count will give an estimate of the area of the Boolean logic network.

And similarly if you want to estimate the delay we can just count the number of stages from the input to the output node and that will be a good measure or good estimate of the delay in a Boolean logic network. Now, having looked into the Boolean logic network representation. Now, let us look into how the optimization is done. Now, as we had discussed in the last lecture, the optimization of multilevel logic network is a difficult task and we try to optimize a Boolean logic network using some heuristic and some transformations as we have done for say two level logic in heuristic two level logic minimizer that we saw in the last lecture. So, we follow the similar kind of strategy in multilevel logic also.

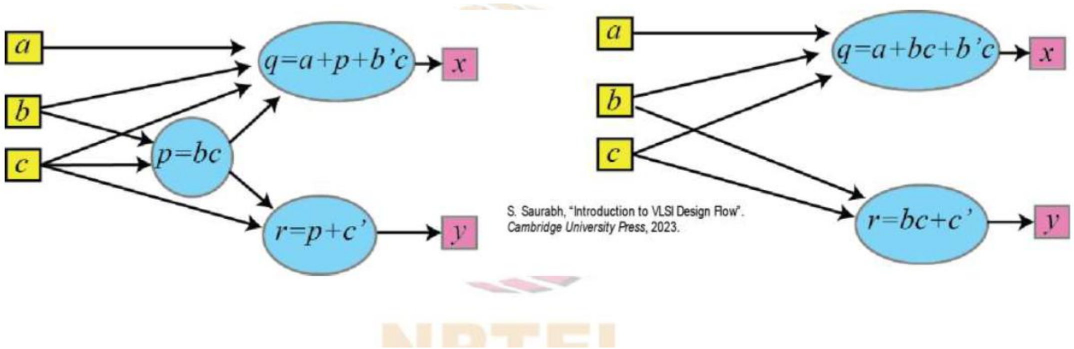
So, what we do is that for a multilevel logic optimization we apply some transformations, a sequence of transformations. So, these transformations can be viewed as operators. So, what are these operators, we will just look. So, these transformations can be viewed as operators for the Boolean logic network. So, given a Boolean logic network, we apply operators one by one in sequence and we go on doing it.

So, these operators are applied iteratively until no more improvement in some QoR measure is possible. So, we go on doing it until we are not getting an improvement. This is a heuristic that is typically used in transformations of the Boolean logic network to improve some QoR measure. So, the important thing here is the final QoR depends on the order of operations of these operators. There will be multiple operators. In which order we apply these operators, the final QoR will depend on it. And what will be a good ordering of these operators for a given Boolean logic network that is also difficult to predict.

So, we cannot predict that we apply these transformations in this particular order will get the best QoR. It is difficult to get that particular order or good order. So, what tools typically do is that they do experiments with different kinds of orders and different kinds of transformations on Boolean logic network. And then see using experiments that which order worked for majority of the designs and that is kept as a default order in the tool. The way of applying these operators on the sequence of operators and which operators to apply, those are experimentally derived based on some statistical analysis that which kind of operators and in which order gives best result for which kind of designs. And that kind of order is put as a default order in logic synthesis tool.

But understand that for some logic network or for a design if you change the order of the transformations then probably you might end up improving or degrading the area or other QoR measure. Now what are these operators on the Boolean logic network? The first one is eliminate. So, what eliminate does it removes a vertex from a graph and replaces all its occurrences in the network with the corresponding local function. So, let us take an example of this. So, in this network there is a vertex p=bc.

Now if we try to eliminate this vertex, what we can do is that wherever p is appearing, in the vertices of the outgoing edge from this vertex. So, it will be here and here. So, in here we have p and here we have p.

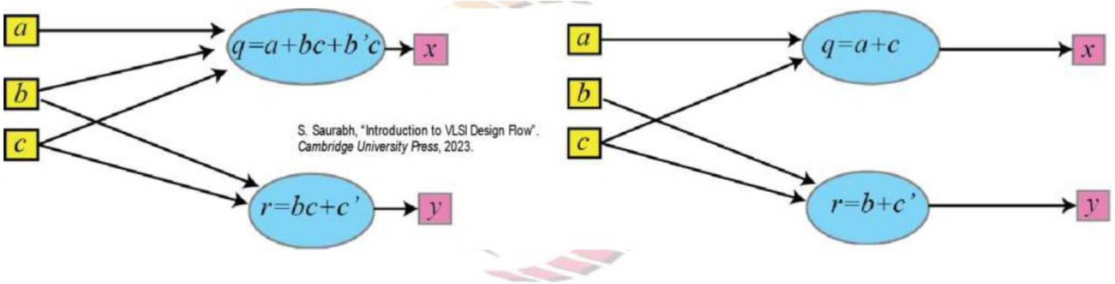


S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

So, instead of p we can replace it with bc. So, the transform network will be like this. We eliminated this node p=bc we reduce one vertex and the functionality of that vertex is embedded or put in the outgoing edge of that vertex. So, in this case it is q and at r we are putting the function p=bc directly. So, this will not actually lead to much benefit in terms of number of literals if we count in terms of area. So, initially we had how many literals? Initially we had 1, 2, 3, 4, 5, 6, 7, 8, 8 literals and later how many literals we had 1, 2, 3, 4, 5, 6, 7, 8, 8.

So, we have 8 literals there is no change in area. But why we are doing this elimination? We are doing this in the hope that in later transformation we can improve the area. So, we carry out eliminate in the hope that the subsequent transformations can reduce the cost (area) and what can that be the later transformation? That later transformation can be the simplify. So, simplify basically simplifies the associated local sum of products expression. So, it works locally on each of the vertex and tries to optimize it or reduce the number of literals in the vertices in the local functions. And since it is in SOP form we can simply use two level logic optimizer for this.

For example, ESPRESSO that we discussed in the last class we can directly use in the function in the transformation simplify. So, if we take the earlier network or the network that we saw in the previous slide. And if we apply two level logic optimization on these two vertices separately then we will land up or we will end up getting this network. What has happened is that this expression or this Boolean function or this SOP representation can be simplified to this. It is very easy to see because we have b and b’ and we can take that as common and so it will become a and a + c. And here also there is c +c’ and therefore, we can eliminate one c applying the property of Boolean logic.

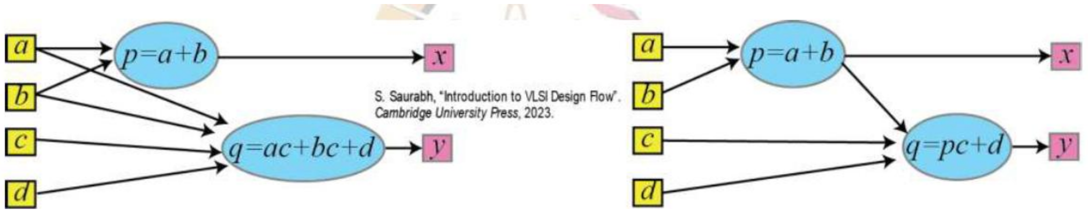


S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

So, the local function was optimized in this key in the function. As a result what benefit we get. Initially we had 8 literals we had seen earlier. Now, how many literals we have one two three four. So, we have four literals instead of eight literals.

So, the number of literal count has reduced from eight to four. So, the eliminate plus simplify have given us a reduction in the literal count. Now, let us look at another transformation which is known as substitute. Now, substitute what does it do? It replaces the local function with a simpler SOP by creating new dependencies and possibly removing other dependencies. So, what it does is that if there is a local function then that local function is replaced by another vertex and a dependency is created for that and the older dependencies are given.

So, it creates dependencies by searching for the appropriate match. Now if you want to replace an SOP in another SOP then we have to first see that whether this SOP is contained in that SOP or not then only we can do a substitution. So, it creates dependency by searching for appropriate match whether a given local function is contained in some other local function. So, those kind of dependency has to be explored. Now it adds more structural information to the network and therefore it is a more difficult task than eliminate and simplify. Because those were easier to do and it was not adding the structural information, it was eliminating the some structural information or working on a local function.



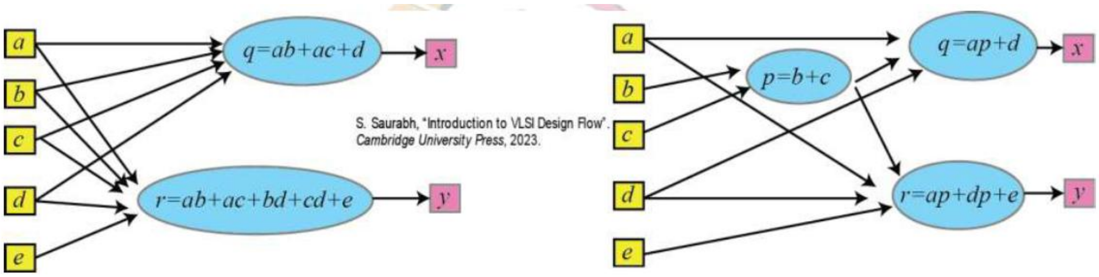
S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

Now let us take an example of substitute. Now consider this Boolean logic network. This node is p=a+b. So, this vertex p=a+b. Now if you look into this q vertex it also contains a+b if we take c as common then we have (a+b).c. So, we can replace this a+b in this SOP by p and we will get this kind of this kind of logic network. So, we need to somehow figure out that p is contained in this local function q and if that is found then we replace the corresponding local function with p and create a new dependency.

So, once we create the new dependency now this function q does not depend on a and b which was earlier and therefore those dependencies will be removed. So, that is what we meant that possibly removing other dependencies and create new dependencies. So, what benefit we got from this transformation? It reduce the literal count. Earlier how many literals were there 1, 2, 3, 4, 5, 6, 7. So, there were 7 literals here and now how many literals are there 1, 2, 3, 4, 5.

So, 5 literals are there. So, we got a reduction in literal count by 2. So, substitute operator needs to find whether local function f_i divides another local function f_j. In this case we said that a+b is a local function, it divides this q and if this kind of division is possible then we can represent this q as the quotient plus remainder and we take a product of this ($f_j = Q. f_i + R$). So, in this case the quotient is c and the remainder is d. So, now to implement a substitute operation or transformation what do we need? We need to perform the division operation efficiently why because we have to find whether one local function divides other local function then only we can do a substitution otherwise we cannot do if there is a division possibility then this kind of transformation is possible and therefore, we need to perform division operation efficiently in Boolean logic network transformation. The other type of transformation is extract. So, what do we do in extract? Extract finds a common sub expression for functions associated with two or more vertices.

So, there are multiple vertices and if there is some function which is common in this and in the other vertex also then we extract out that create a new node, create a new vertex for that and then replace the occurrence of this common sub expression with this new vertex. Then replace the common sub expressions in the original functions with the variable of the new vertex. So let us take an example so we have a vertex q which is ab+ac+d and we have another vertex r=ab+ac+bd+cd+e. So, we can see that b+c is a common sub expression which is present here also and here also and therefore, we can extract out b+c and create a new node and in the q function we can replace b+c occurrence with the new node that is p and in the vertex r we do the same thing we replace b+c with p and we get this kind of network. So, now again here we create new dependencies these are new dependencies between p and this. The earlier dependency of q was on b and c also, so, that dependency gets eliminated for q similarly for r.



S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

So, new dependency will be created with respect to p and other dependency can be removed. Now as a result of this what benefit we got? Earlier how many literals were there 1 2 3 4 5 6 7 8 9 10 11 12 13 14 earlier we had 14 and now how many we have 1 2 3 4 5 6 7 8 9 10. So, we have 10. So from 14 the number of literal decreased to 10 and therefore, we expect to save in area in the CMOS implementation of this circuit. Now extract operator needs to find divisors for the local functions and then search vertices with matching divisors. So, in this case the divisor was b+c first we have to find that what are divisors for q and whether this divisor b+c is present in r also.

So, we have to find divisors and then search vertices matching divisors. So, implementing extract operator is computationally challenging. So, this kind of task finding divisors and then finding the occurrence of that divisor in other vertices is difficult task. So when we are talking about Boolean logic network optimization the number of nodes or number of vertices in the network can be in thousands or so. So, if we have a big network in that network how efficiently we can find divisors and matching divisors with other vertices those are very critical. Now what are the challenges of multilevel logic optimization? There is a huge search space.

So, this multilevel logic will be very big and in that how to do this optimization there are lots of solutions in sense that we can do transformation in many different ways. Out of that which one to pick which is actually improving our QoR in terms of area and timing and other measures it is difficult to explore and come up with a one solution. So, because that is the search space or the optimization space is huge and to accomplish this task what do we rely on? We rely on division operation we need to have division operation which does it very efficiently why because during multilevel logic optimization we need to carry out divisions too many times. For example, for the extract operation or also for substitute operation or transformation we need to extract divisors out of it and therefore, in multilevel logic optimization we need to carry out division operation many a times and the efficiency of multilevel logic optimization very much depends on how efficiently we can perform division. Practical circuits have thousands of vertices in Boolean logic network and might need to divide each vertex with rest of it to find that whether we can substitute it or whether there are matching divisors and so on.

So, we might need to compare one vertex with all others and if we do that then it is a kind of O(n^2) kind of comparison meaning compare with 1 to n and there are n such cases so n*n is equal to n^2. So, n^2 kind of division operation might be required to divide each vertex with the rest and therefore, the efficiency of division operation is very important for multilevel logic optimization. The other thing is that what are good divisors for a Boolean logic network. So, for each vertex we need to find good divisors meaning that if we take that out in extraction or substitution then it will lead to some improvement in the QoR measure. So, we need to extract good divisors from the vertices and this is a challenging task.

So, finding a good set of divisors for a given Boolean expression is non trivial. Now, how do we accomplish this complicated task of carrying out divisions of Boolean functions efficiently and also finding common divisors between various local Boolean functions efficiently. So, we accomplish this task by using a model which is known as algebraic model. So, in algebraic model what we do is that we neglect some Boolean properties of the local functions. So, we neglect Boolean properties and what we do is that we treat the local Boolean functions simply as polynomials and employ the rules of polynomial algebra rather than Boolean algebra. What we mean is that suppose there is a function y=ab+a'c+d then in this case for this function what we will do is that we will treat a variable and its complement as separate variables meaning that a and a bar or a' these two entities will be treated as separate entities in algebraic model, but in Boolean model we know of course, that a bar is basically the complement of a or a' is the complement of a.

So, a' and a are related in Boolean model, but in algebraic model we treat each variable and its complement as separate variables and we apply the rules of polynomial algebra. We do not apply the properties of Boolean algebra, but of polynomial algebra in the algebraic model and as a result what benefit we get is that we can formulate efficient algorithms to carry out division in the algebraic model rather than in the Boolean model by treating variable and its complement as separate variables and applying the rules of polynomial algebra we can formulate very efficient division algorithms. So, that is the first application of the algebraic model. The second application is that good divisors or divisors which can reduce the area of a given circuit or a given Boolean logic network or common sub expressions can be extracted from a local function much more efficiently in an algebraic model rather than in the Boolean model. So, again when there are multiple nodes or multiple vertices in our Boolean logic network then checking and extracting which divisors are common among various vertices and so on those can be done very efficiently in algebraic model rather than in Boolean model why because there are some properties of the algebraic model that allows us to prune the search space we can just by looking at a few expressions or so we can say that whether there is a commonality between two expression or there can exist a common sub expression between two vertices or not that test can be performed very efficiently in this algebraic model rather than in Boolean model.

So in this course, we will not go in depth into the algebraic model and how those kind of division and common sub expression extraction are found or determined in Boolean logic network. But just to summarize that using the algebraic model and the associated mathematics of algebraic model we can formulate very efficient algorithms to do this substitution and extract operation in a Boolean logic network and this forms actually the basis of fast multi level optimization in the contemporary logic optimization tools. So most of these tools these advanced logic synthesis tool internally use this multi level logic optimization and why this multi level logic optimization techniques are efficient, the root is algebraic model and the associated mathematics of. So if you want to go into depth of this please look into the references that I will be listing out at the end of this lecture. Now when instead of a Boolean model if we use algebraic model are we losing some optimization possibility? the answer is yes.

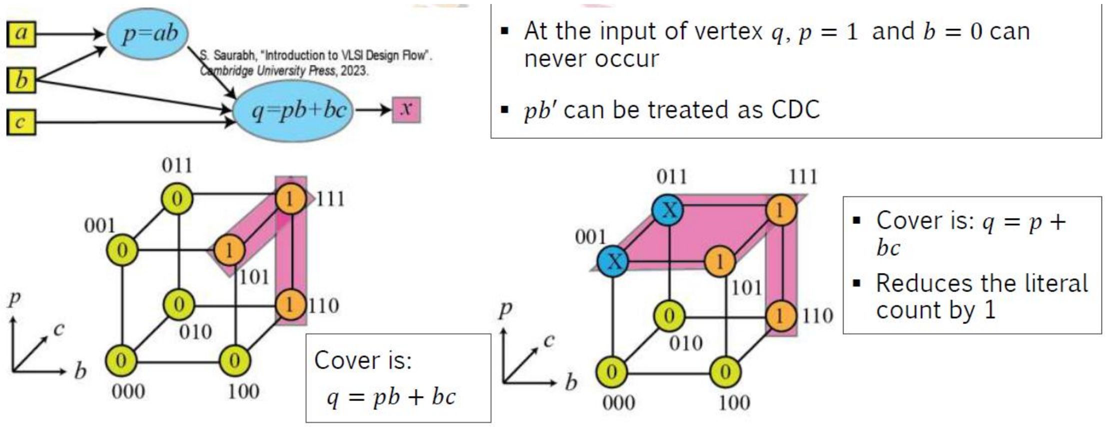
For example, if there is an expression a+a' in a Boolean model we can simply write it as 1 while in algebraic model we cannot do those kind of simplification. Why because this variable and its complement both are treated as separate variables we cannot do this kind of optimization in the algebraic model. So an algebraic model cannot fully optimize a Boolean logic network. So what do we do? So what we do is that post algebraic model optimization we apply transformations that utilize the power of Boolean model also. So given a multi level logic network or Boolean logic network we first do the optimization based on algebraic model after that we apply more optimization techniques based on the Boolean model and further reduce the area or other figures of merit of our set. So let us look into what is this Boolean model or what optimization can be done in Boolean model which perhaps cannot be done in the algebraic model.

So one of the important aspect of logic network is that there can be don’t care conditions. So you might be aware of the don’t care condition in your earlier courses of digital circuit also. But in the earlier courses of digital circuit you might have seen that don’t care conditions or DC conditions are given to us. Given a logic network, we are given a function and we are also given that these are the don’t care conditions. Now do the optimization and reduce it to a smaller function F or a minimal function f. This is what typically we do in digital circuits that the don’t care conditions are given to us.

But in a Boolean logic network what happens is that don’t care conditions arise naturally. How it arises naturally we will see. Intuitively this don’t care conditions arise because of the graph structure of the Boolean logic network and the dependencies among the local functions. What are those dependencies we will see just now. If we have don’t care conditions for a local Boolean function then we can apply optimization based on this don’t care condition and the local Boolean functions can be further optimized and our Boolean logic network can be made more compact or the area can be reduced. So don’t care conditions are rich source of optimization in multilevel logic synthesis.

But the problem here is that the logic synthesis tool need to discover those don’t care conditions. Though don't care conditions is not given, the tool needs to find out those don't care conditions by network analysis and then use it subsequently for the optimization of the local logic function. So we will see how it can be discovered. So there are two types of don't care conditions that are useful in simplifying logic functions in Boolean logic network and these two types of don't care conditions are controllability don't cares and observability don't cares. So we will look into both of them one by one. So what is a controllability don't care or CDC? So the combination of input variables that can never occur at a given vertex in a Boolean logic network produces CDC.

So we have a vertex and the inputs are coming at it. Now there can be some combination of this input which will never appear at this node and if we can identify those conditions then perhaps we can optimize the local function of a given node. That is what the controllability don't care is and how they can be used in optimization. So local functions can be simplified by accounting for CDCs and using two level logic minimizers. For example, consider this logic network. Now in this logic network if we consider the node or vertex q the hypercube representation is shown here and the function q=pb+bc and we cannot further optimize it just by looking at this Boolean logic function.


a
b
c
p=ab
q=pb+bc
x
S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.
011
001
111
101
110
010
000
100
p
c
b
0
1
X

- At the input of vertex q, $p = 1$ and $b = 0$ can never occur
- pb' can be treated as CDC

Cover is:
$q = pb + bc$

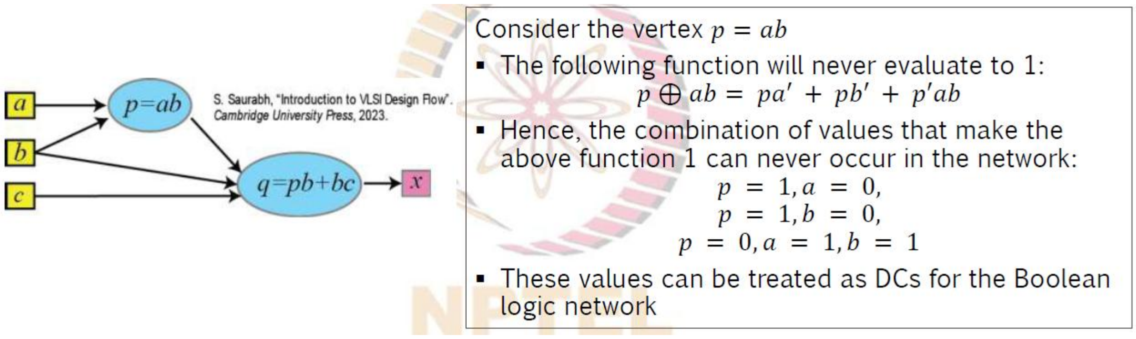
- Cover is: $q = p + bc$
- Reduces the literal count by 1

There is no scope of further optimization as it is apparent from this hypercube representation. Now at the input of vertex q, now at the input at this point p=1 and b=0 can never occur. If b=0 then p will always be 0 by the definition or the constraint that p=ab, if b=0 then p has to be 0 it cannot be anything else and therefore p=1 and b=0 can never occur at this node. So, we can treat pb’ that is we made an implicant out of this condition that p=1 and b=0.

So, pb’ can be treated as a controllability don't care for the vertex q. Why we can treat as a don't care because this combination p=1 and b=0 can never appear at this node and this information can be utilized in optimizing the node q. So, for the implicant pb’ we made the don't care condition. So, pb’ corresponds to this hypercube on this axis; we have b on this axis we have c and on this axis we have p. So, if we say pb’ then we are saying that b should take a value of 0 and p should take a value of 1 and c can take any value and that is why this corresponds to 001 and 011, these two nodes.

So, we have represented pb’ in this hyper cube as x. Now given this representation of the function which is now in completely specified function because it has got don't care also we can optimize it and find the cover as shown here. These two are the two implicants in the cover and q can be written as q=p+bc, rather than writing q=pb+bc, we write as q=p+bc. Now if we do that then what effect we get? Initially we had 1 2 3, 4 literals and now we have 1 2 3, 3 literals. So, it reduces the literal count by 1. So, we have reduced the literal count in this network by employing the controllability don't care for this vertex.

Now in this case it was easy to guess that what is the controllability don't care condition. But in a large network how the tool will get to know the controllability don't care. So, in a large network what the tool does is that it employs satisfiability don't cares to derive controllability don't care. Now what is satisfiability don't cares? So, CDCs can be computed using efficient algorithms by exploiting satisfiability don't cares or SDCs and SDCs get enforced by local functions associated with the vertex and its output. So, let us take an example suppose this was the network the same network that we saw in the last slide.



S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

Consider the vertex $p = ab$

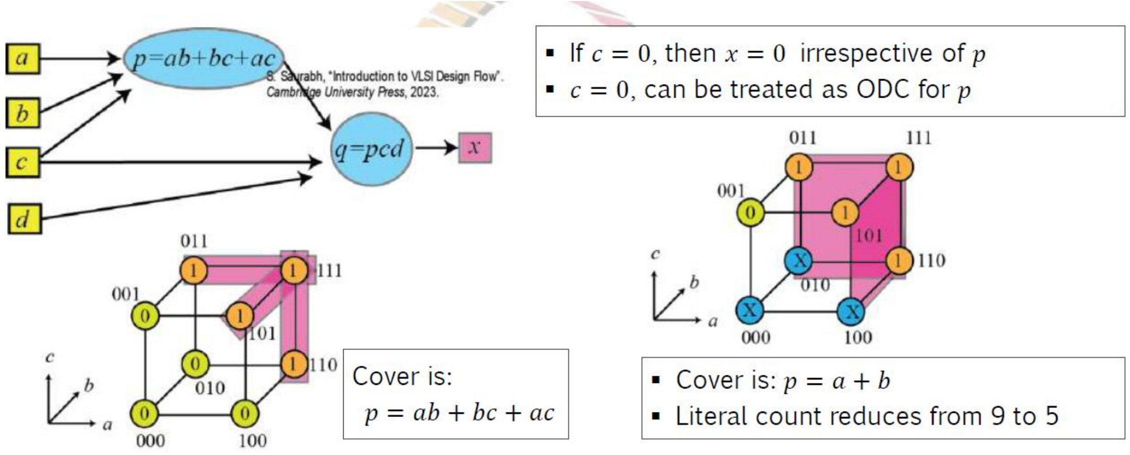
- The following function will never evaluate to 1:
$$p \oplus ab = pa' + pb' + p'ab$$
- Hence, the combination of values that make the above function 1 can never occur in the network:
$$p = 1, a = 0,$$
$$p = 1, b = 0,$$
$$p = 0, a = 1, b = 1$$
- These values can be treated as DCs for the Boolean logic network

Now consider the vertex p= ab we are considering this vertex. Now for this vertex the function p XOR ab will never be 1 why because p=ab it will always be there. So, if p is 1, ab will be 1 if p is 0, ab is equal to 0. So, p XORs ab can never be equal to 1. The function p XOR ab if we expand it. We can expand p XOR ab as p(ab)'+p'(ab) and this becomes pa'+pb'+p'ab we have changed (ab)' as a'+b'. So, that is what it is written. Now if this function can never take a value of 1 meaning that all of them, all these three implicant or these terms will always be 0.

So, hence the combination of values that make the above function 1 can never occur in the network and therefore, this has to be 0. So, p=1, a=0 can never occur p=1 and b=0. So, these are the combinations which makes this term as 1 and they can never happen they can never occur. So, p=1, a=0 can never occur p=1, b=0 can never occur and p=0 and a=1 and b=1 that corresponding to this will never occur. And these values can be treated as don't cares for the Boolean logic network and in this case what we have done is that we have treated pb' or we have treated only this as the don't care condition why because the dependency is only between b there is no dependency of a and q there is a dependency between b and q and that is why we took this don't care condition as our controllability don't care in optimizing this node q.

So, logic synthesis tools typically derive CDC algorithm algorithmically using SDCs and subsequently CDCs get utilized in simplifying local functions as we have seen. The other type of don't care is observability don't care. So, how do we derive observability don't care we derive it by looking at the fan out of a vertex rather than looking at the fan in we look into the fan out of the vertex and the observability don't cares are the input variable combinations that obstruct the vertex output from being observed at the network output. So, what it means is that suppose there is a vertex and it is producing either 0 or 1, but under certain input conditions the effect of this vertex that is whether it is producing 0 or 1 is not being seen at the output of the network. So, at output of the network, there is no impact for certain input conditions then that case, that input condition is known as observability don't care.

So, let us take an example of this. Now let us consider this vertex p now if we try to optimize it there is not much possibility of optimization by two level logic optimizer. So, if we see the function p=ab+bc+ac it and draw its hypercube representation then we can see that there are three implicants in the cover and we cannot further reduce it. But if we utilize observability don't care then we can perhaps optimize it let us see how. Now at the output of this vertex p there is the node q and if there is some condition at the node q which will obstruct p from being observed at x.



S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

- If $c = 0$, then $x = 0$ irrespective of p
- $c = 0$, can be treated as ODC for p

Cover is:
$p = ab + bc + ac$

- Cover is: $p = a + b$
- Literal count reduces from 9 to 5

So, this is the case. So, we see that yes there is the case for example, if c takes a value 0, then output becomes 0 irrespective of what p produces in that case, p produces 1 or 0 it does not matter. So, whatever the p produces does not matter if c is equal to 0 the output will always be 0 and therefore, we can treat c is equal to 0 as an observability don't care for the vertex p. Now with this information that c=0 is a don't care condition we can draw the hypercube representation. So, for the case c=0, c is in this direction.

So, c=0 is this case, we can say that these all are x or don't cares. Now if these nodes are taken as don't care we can optimize the Boolean function as a+b we can simply take two implicant, the b implicant and the a implicant and cover this function. So, cover is now a+b as a result of that what improvement we got? The improvement is the number of literals. Earlier how many literals were there? Earlier we had 1 2 3 4 5 6 7 8 9 and now after we replace this with a+b how many literals, we will have 1 2 3 4 5. So, number of literal reduced from 9 to 5. So, note that the function p is now different this function ab+bc+ac and a+b are different.

So, the output here will be different for the unoptimized network and the optimized network. But the thing is that even if this produces a different value for the cases when it produces different values the output will not determined by this p but by some other input in this case c and that is why we can make this kind of transformations. So, these transformations can be made using Boolean optimization techniques in a Boolean model. So, we have seen a very simple case of controllability don't cares and observability don't cares in which the nodes are very close to the input or the output and the number of vertices in the network are very limited. Once we have our thousands of vertices in our network then we have to do a kind of traversal of the graph and then discover the don't care condition and then utilize them in our for Boolean optimization of the local functions and the network. So, those are very sophisticated techniques and if you want to look into those techniques these are this book Micheli is a very good reference.

So, this brings us to the end of this lecture. So, let me summarize what we have done. So, in this lecture we have looked into the multilevel logic optimization and we saw that there are two types of models which are used in multilevel logic optimization: the algebraic model and the Boolean model and in the Boolean model we looked into that how don't care conditions can be extracted or discovered from the network and then the functions can be optimized. So, in this lecture and in the last lecture we covered the combinational logic optimization technique. In the next lecture we will be looking at the sequential optimization technique. Thank you.