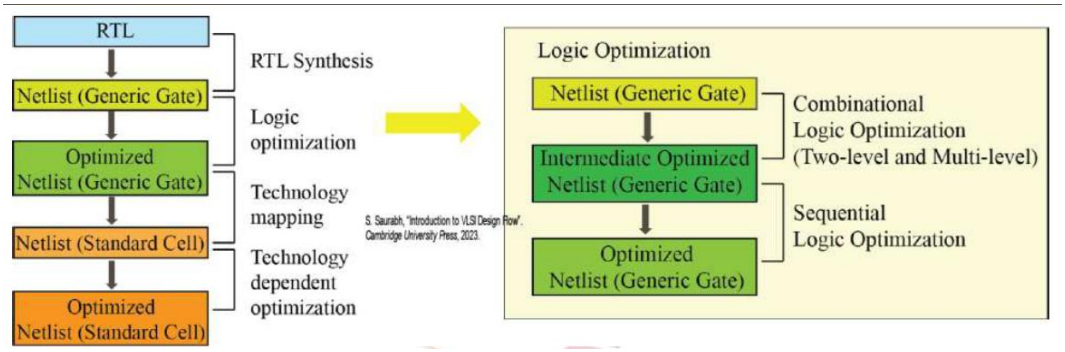
VLSI Design Flow: RTL to GDS

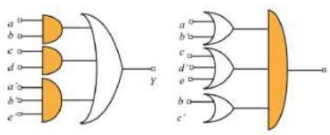
Dr. Sneh Saurabh
Department of Electronics and Communication Engineering
IIIT-Delhi

Lecture 17
Logic Optimization: Part I

Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the 14th lecture. In this lecture, we will be looking at logic optimization. In the earlier lectures, we had seen that the logic synthesis task can be broken down into various simpler tasks as shown in this figure. In the last two lectures, we had seen RTL synthesis and in this lecture, we will be looking into logic optimization. Logic optimization basically transforms a given netlist which is in the form of generic logic gates into an optimized netlist such that the functionality is preserved while its figures of merit primarily the area improves.

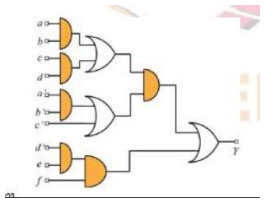


Now, if we look into the logic optimization task, it is of two types. The first type of logic optimization is combinational logic optimization which targets the combinational logic codes in a given circuit. And the second part of logic optimization is related to sequential logic optimization or the finite state machine (FSM) representation of a given netlist. Now, the combinational logic optimization are of two types, two-level logic optimization and multi-level logic optimization.



Now, these two types of optimization differ in the representation of a Boolean function. Now, two-level logic optimization works on the representation of a Boolean function in which the Boolean function is represented in two levels of logic as shown in this figure. It can be either sum of product form. So we have all the products and then we are taking the sum of it. So it is a sum of product representation or it can be product of sum.

So it is the sum and we are taking the product or it is a product of sum form. So in these type or in these representations there are two levels of logic AND followed by OR or OR followed by AND. Only two-levels of logic are involved and the logic optimization which works on this form of representation is known as two-level logic optimization. And the other form of logic function or another form of representation of the logic function is multi-level form as it is shown here. So this is a multi-level logic representation in which from the input to the output there are multiple levels of logic.



In this case there are 1, 2, 3 and 4 levels of logic input. So logic optimization which works on multi-levels of logic those are known as multi-level logic optimization. So in this lecture we will be looking into two-level combination logic optimization and in subsequent lectures we will be looking into the multi-level logic optimization and also the sequential logic optimization.

Now before going deeper into the logic optimization let us look into a statement from R.W. Emerson. The statement is, “We ascribe beauty to which is simple; which has no superfluous parts; which exactly answers its end…” Now this statement was made in a different context but it is very much relevant to logic optimization because logic optimization basically tries to achieve this. So what does logic optimization do? It tries to get rid of the superfluous parts of logic. As a result of that our logic becomes simple and also it becomes beautiful in terms of figures of merit. So now let us look into that how logic optimization is able to achieve this kind of simplification and improve the figures of merit of our design.

Before looking into two-level logic optimization we should be familiar with some important definitions related to Boolean functions and logic optimization. Now what is a Boolean variable? A variable that can take one of the two values 0 or 1 those are known as Boolean variables and we represent Boolean variable by symbols like say x1, x2, a, b

and so on. Now what is a Boolean function? A function that takes Boolean variables as arguments and evaluates to 0 or 1 those are known as Boolean functions and we represent Boolean functions as y=f(x1, x2, x3,…, xn) where x1, x2, x3, xn are Boolean variables. So we call the y as the output of this Boolean function and the Boolean variables on which this output depends x1, x2, x3, … xn these are the input to the Boolean function. Now what is a literal? Literal is a Boolean variable or its complement.

So it includes both the Boolean variable and its complement. For example, we say that if x1 is a Boolean variable then x1 is also a literal, x2 is also a literal. Additionally, if we take the complement of x1, i.e., x1'. So when we say that x1 is a variable and we write x1' then x1' represents the complement of x1. So x1' is also a literal, x2' is also the literal meaning that any variable or its complement both of them are considered as literal.

Now what is a cube? Cube is a product of literals. Product meaning the AND operation. If we do AND operation of various literals we get a cube. For example, x1x2x3 is a cube. So x1x2x3 is basically x1.x2.x3 where dot represent the AND operation and in short form we write as x1x2x3 and it is a cube. x1x2x3 is a cube.

Similarly, x1x2' is a cube. Also the literals are also considered as a cube. For example, x2' is also a cube consisting of only one literal. Now let us consider Boolean function of n variables where n is a number. So if for a Boolean function of n variables, we can define a minterm.

Now what is a minterm? Minterm is the cube of n literals in which each variable or its complement appears exactly once. So if a function is of n variables then all the n variables should appear either as a variable or its complemented form. All of them should appear and in minterm we have a cube meaning that it is a product, it is a product of the literals. Now similarly instead of the product or instead of AND operation if we take the sum of n literals for a Boolean function of n variables then we get a maxterm. So again in maxterm all the n variables should appear either as a variable or its complemented form.

Let us take an example to understand this. So consider a function of 3 variables x1, x2, x3 these are 3 variables. Then what could be the minterms? A minterm, an example of minterm is x1x2x3. Why it is a minterm? Because all 3 variables are appearing exactly once. Similarly x1'x2x3 is again a minterm because x1 is appearing as a complement form but it is still appearing.

And x2 and x3 are also appearing. Similarly x1'x2'x3 is also a minterm. But is x1x2 a minterm? No, it is not a minterm. Why it is not a minterm? Because if you are considering 3 variables x1, x2, x3; x3 is not appearing in it. Therefore x1x2 is not a

minterm.

Is x1x2x3x3' is it a minterm? No, it is not a minterm because x3 is appearing twice. Each variable should be appearing as a literal only once or exactly once. It cannot appear less than or more than once. So therefore this is also not a minterm.

Similarly max term are (x1 + x2 + x3) because all the variables are appearing exactly once. (x1 + x2' + x3) is another max term. Now let us look at a few representations of a Boolean function. Why we should be worried about representation of a Boolean function? Because many logic optimization tasks and manipulation of Boolean function is strongly dependent on the representation of the Boolean function. For some Boolean function representation, the task of manipulating them may be very difficult.

For some logic optimization task or doing some transformation of a Boolean function may be much easier in a given Boolean representation and may be very difficult in other kind of representation. And therefore while doing logic optimization the representation of a Boolean function is very important. Now let us look at a Boolean function representation which is known as truth table. You might be aware of this from your course on digital circuits. So in a truth table we have various rows.

$$y = x_1x_2 + x_2x_3 + x_3x_1$$

x_1	x_2	x_3	y	Minterm
0	0	0	0	$x_1'x_2'x_3'$
0	0	1	0	$x_1'x_2'x_3$
0	1	0	0	$x_1'x_2x_3'$
0	1	1	1	$x_1'x_2x_3$
1	0	0	0	$x_1x_2'x_3'$
1	0	1	1	$x_1x_2'x_3$
1	1	0	1	$x_1x_2x_3'$
0	1	1	1	$x_1x_2x_3$

$$y = x_1'x_2x_3 + x_1x_2'x_3 + x_1x_2x_3' + x_1x_2x_3$$

And in each row what it represent? It represents the value either 0 or 1 assigned to each input variable x1, x2, xn in a Boolean function and the corresponding output value. And since if there are n Boolean variable in a function and each Boolean variable can take a value 0 or 1. So the number of possible combination of input variables is 2^n and therefore the number of rows in a truth table will always be 2^n. Now let us take an example of a Boolean function y = x1x2+x2x3+x3x1 and draw its truth table. So we have since there

are 3 variables here x1, x2 and x3 and we will have 8 rows $2^3 = 8$ and therefore we have 8 rows in the truth table.

Now each row represent assignment of the variables or unique way of assigning variables. So if we take the first row in this row we have x1 = 0, x2 = 0, x3 = 0. Now if we put x1, x2, x3 = 0, 0, 0 then y will evaluate to 0 because all these terms will be 0, and therefore y = 0. Now let us take another row. In this row x1 = 0 and x2 = 1 and x3 = 1.

Now if we see y: x1x2 will be 0, because x1 is 0, x2x3 will be 1 because x2 and x3 both are 1 and x3x1 will again be 0 and if we take the sum of them we will get 1. Similarly all the rows of the truth table are constructed. Now let us look at another representation of a Boolean function which is known as minterm representation. Now corresponding to each row in the truth table we can associate a minterm with it. For example, let us see how do we construct the minterm for a given row of truth table.

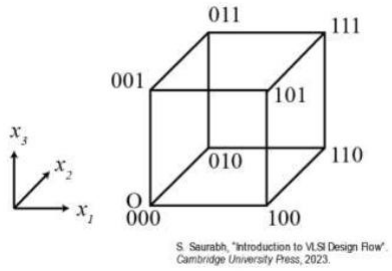
For example let us consider this row. So for this row, for any row what we do is that we construct the minterm by taking an uncomplemented value of the variable if it takes a value 1 and its complemented value if it is taking 0. So x1 is taking a value of 1 so we take it as not complemented x1 simply x1. Now x2 is taking a value of 0 and therefore we make it as x2 bar, means we take a complement of it or x2', and x3 is taking a value of 1 so we are not taking the complement, so the minterm is x2x2'x3. Similarly we can associate minterm with each row of the truth table.

Now what will be the property of these minterms? So the minterm will be evaluating to 1 only for the assignment of variables corresponding to that row in the truth table. For example if we take this particular minterm it will evaluate to 1 if x1=1 into x2=0, and if we take complement of x2' it will become again 1 and x3=1 so product of 1.1.1 = 1. So the minterm will evaluate to 1 for the corresponding rows if the values of the variables are corresponding to that row, and for all other assignments of variable the minterm will evaluate to 0. For example if we evaluate x1x2'x3 for any other row we will see that one of these three terms or three literals will turn out to be 0 in the minterm and therefore this will turn out to be 0. So what it means is that the minterm will evaluate to 1 for only one row in the truth table or that row corresponding to the truth table for which that minterm was constructed.

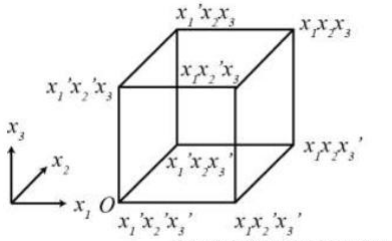
For all other rows it will evaluate to 0. Now using this observation we can represent a function as sum of minterms. What we do is that we take the sum of only those minterms for which the function evaluates to 1. Now in this case the function evaluates to 1 for this case, this case, this case and this case. So corresponding to these minterms or we take the minterms corresponding to these rows and form a sum.

Now if we write y = sum of these minterms then this y will evaluate to 1, one of these terms will evaluate to 1 for each of these four rows. And for all other rows all these minterms will evaluate to 0 and therefore y will evaluate to 0. And for these four rows one of the minterms out of these four minterms will evaluate to 1 and therefore y will take the value of 1. So now we can represent a Boolean function in sum of minterm form. Now let us look into another form of representation of a Boolean function which is known as hypercube representation. Why we want to represent Boolean function in a form of hypercube? Because many of the logic optimization task that we will be looking later in this lecture those can be visualized very well on a hypercube.

And therefore let us represent Boolean function in terms of hypercube. Now Boolean functions of n variables y = f(x1, x2, x3, …, xn) spans n dimensional Boolean space. And an n dimensional Boolean space can be represented and visualized using n dimensional Boolean hypercube. So let us see what is an n dimensional Boolean hypercube. Let us take n = 3 because for 3 dimensional hypercube we can easily visualize.



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



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

Now this is an example of, this figure is an example of 3 dimensional Boolean hypercube. And we can associate each variable with one dimension of the hypercube. So if we consider a Boolean function of 3 variables y = f(x1, x2, x3) these are 3 input variables x1, x2, x3 then we can associate each variable with one dimension of the hypercube. So we say that in this direction is x1, in this direction it is x2 and in this direction we have x3. So we can associate each direction or we can associate a Boolean variable with each direction in a Boolean hypercube.

And then the corners of the hypercube represent binary valued n dimensional vectors. Now in the truth table we saw that each row represented the combination of values for the input variables. Now in the case of Boolean hypercube each corner represents the value assigned to each of the Boolean variable. So for example, let us take this corner 110. So the ith entry in the vector corresponds to the value of the variable xi.

So if we consider the the vector as 110 for this corner then we can associate this first 1 with x1, second 1 with x2 and third 1 with x3. So this point corresponds to the assignment of variable x1, x2, x3 as 110 where x1 is assigned to first 1, x2 to the second and x3 to the third 1. And we can see that if we take this variable then at this point on the x1 side it is 1. On the x2 side it is 1.

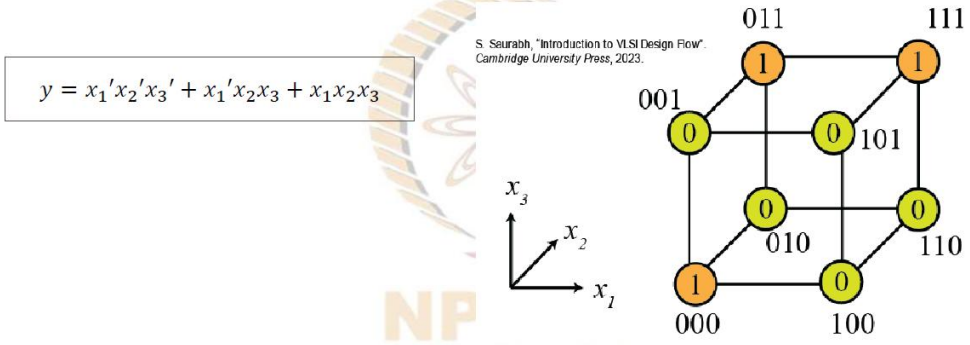
And on the x3 side it is 0. And that is why we can understand that the coordinate of this point, of this corner is 110 and we can understand that at the point 110 the value of x1 is 1, x2 = 1 and x3 = 0. Now a Boolean hypercube has 2^n corner. Now if there are n variables and each variable can take a value of 0 or 1, it will have 2^n corners. In this case there are 3 variables and therefore we have 8 corners. So it represent 2^n input combinations similar to a truth table.

It is a similar representation as a truth table. Only difference is that rather than writing it in a table form we are drawing it in a kind of a cube form. Why we are drawing it in a form of cube? Because it will help us visualize the operations, logic minimization task that we will be looking at subsequently in this lecture. So we can also associate each corner with a minterm similar to what we had done for the true table. For example, at this corner x1 = 0, x2 = 0 and x3 = 0.

And since all of them are 0 we can say x1'x2'x3' is the minterm associated with this particular corner. Similarly if we look into this corner, so x1 is taking a value of 0. So we can take x1', x2 and x3 are 1 so we just write x2x3 and this is the minterm associated with this corner or in this case this corner, x1'x2x3.

So we can associate each corner with one minterm. Now how do we represent a Boolean function in a hypercube? So we mark the corners in the Boolean hypercube with the value of the function for the associated input combination. So in a true table we marked the value in a separate column, the output column. In this case what we do is that we mark the value directly at the corner. For example if we consider this function y = x1'x2'x3'+x1'x2x3+x1x2x3. So there are three minterms, and if we draw the representation of this Boolean function in the hypercube notation then we will just say that for this minterm which correspond to this, we write 1 for this minterm, this

corresponds to the corner x1 = 0, x2 = 1 and x3 = 1.



So we say that 011 that is for this corner we write a value of 1. Similarly x1x2x3 correspond to 111 and we have 1. For other corners we write a value of 0. So this way we can represent any given Boolean function in a hypercube notation. Now in some Boolean functions there can be don't care conditions and what are don't care conditions?

For some input combinationsof the function y = f(x1, x2, x3, …, xn) may not be specified. We do not know the values for those cases or we are not given the values for those cases. So these input combinations are known as don't care meaning that in that case the value of y is not specified and it can be taking any value 0 or 1. We do not care for those input combinations.

So we denote these DC, don't care conditions as X. As we say that for 0 we write 0 for 1 we write 1. Similarly for don't care we say X. Now don't care conditions are related to the input combinations that can never occur in a circuit. Now how can it be? For example suppose we are designing a combinational logic cone in which the inputs are coming in a form of binary coded decimal. Now in binary coded decimal digits, decimal representation is in the 4 binary digits.

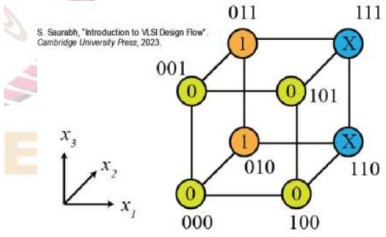
So it is represented in 4 bits. And the values that this binary coded decimal can take is between 0 and 9 and it is coded between 0000 to 1001. So the other possible values meaning after 1001 like 1010, 1011, 1100, 1101, 1110, 1111 those are not possible in binary coded decimal representation of a number. And therefore if we are designing, a combinational logic block which is receiving numbers in binary coded decimal format then these numbers can never appear and therefore these combination of values can be considered as don't care in the combinational logic block that is receiving binary coded decimal digits. Sometimes there can be cases for the input combinations for which output is never observed. Suppose there is a input combination which makes the output or the

block which is observing the output go in a sleep state.

Now if the block is in the sleep state whatever output we produce, it will not be able to read it or it does not care to read it. And therefore a function producing an output for a block that is in the sleep state it can consider those inputs, which is taking the block at the output in the sleep state, those as don't care conditions. Now there can be a Boolean function which contains don't care or Boolean function with don't care condition are known as incompletely specified Boolean function. Now how can we represent incompletely specified Boolean function? We can represent incompletely specified Boolean function using three sets. The first set is the onset, it contains input combinations for which the output is 1.

Example:

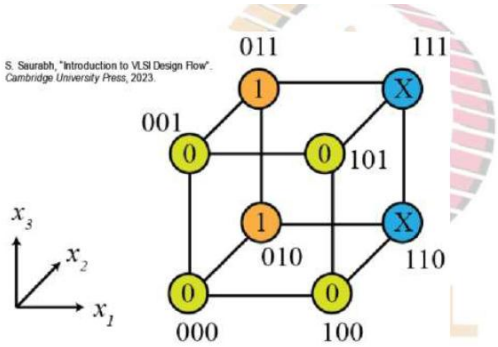
- A function of three variables x_1,x_2, and x_3.
  ON-set={010, 011},
  OFF-set={000, 001, 100, 101}, and
  DC-set={110, 111}.



The second set is the offset meaning the input combinations for which the output is 0. And the third combination, third set is the don't care set or DC set with the input combinations, for which the output is don't care or X. Let us take an example suppose there is a function of three variables x1, x2, x3 and the value of this function takes 1 for these input combinations. So we can say onset = the set of input combination that produce 1 then offset is the set of input combinations which produces an output of 0 and don't care set are the set of input combination for which the function is a don't care.

So we can represent this function also in hypercube. So we say that for 010 and 011 we write the output as 1 for 000, 001, 100 and 101 we write the output as 0 and for 110 and 111 we write as X. So we can represent an incompletely specified Boolean function also in a hypercube representation. Now let us understand a term which is known as implicant. What is an implicant of a Boolean function? Implicant of a Boolean function is a cube whose corners are all in the onset or don't care set. It can be either in the onset or in the don't care set but it cannot be in the offset.

So what is an implicant? It is a cube and what is a cube? Cube is a product of literals. So implicant is a product of literals whose corners are only on the onset or don't care set. It should not go into the offset. So let us take an example. So suppose the Boolean function was this or this is an incompletely specified Boolean function which was given.



Implicants:

- $x_1'x_2x_3'$
- x_1x_2
- x_2x_3'
- x_2

Not an implicant:

- $x_1'x_3'$
- x_1'

Now in this case what can be the implicant? So we can say that x1'x2x3 meaning that the combination is, we can write the combination as 0 and 1 and 0. For this 010 we see that 010 is this corner, and the value is 1. So it is in the onset and that is why this is an implicant. What about x1x2? Now x1x2 means that x1 can take a value of 1, x2 can take a value of 1 and x3 can take any value. Now what does it represent? It represent this corner 110 and 111 this corner, and both are don't care, and therefore it is an implicant.

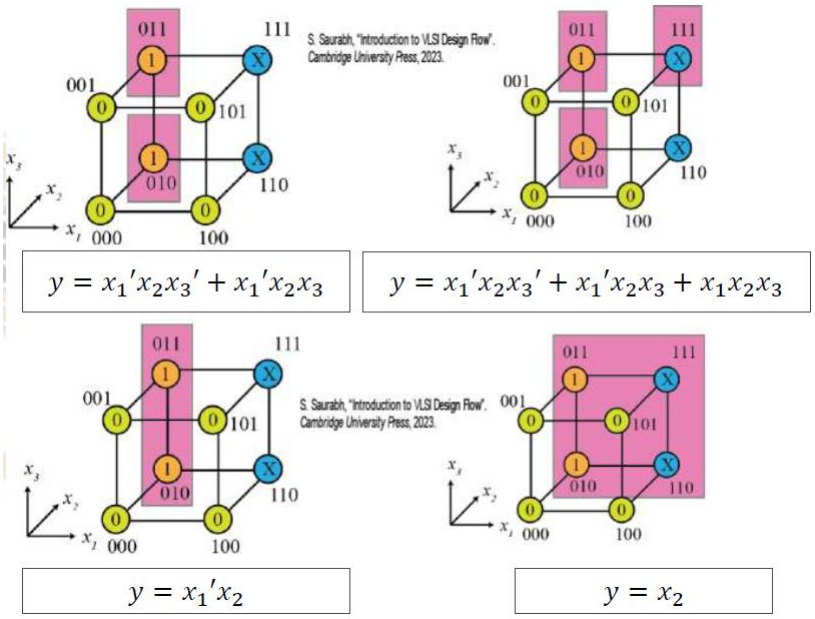
Now let us take the third example. The third example is x2x3'. So in this case x1 can take any value, I am specifying as dash (_) and then x2 can take a value of 1 and x3', x3 can take a value of 0. So 010 this one and 110 this. So 010 is in the onset. This one is in the onset and this one 110 is in the don't care set. So these two guys are there in x2x3' and they are either in the onset or in the don't care set and therefore x x3' is also an implicant.

What about x2? If x2 is there meaning that x1 can take any value x2 can take 1 and x3 can again take any value. So in that case these are the 4 corners, these are the 4 corners corresponding to x2. This one, this one, this one and this one. In these 4 corners x2 is taking a value of 1 and x1 and x3 can take any value. Now all these are either in the onset or in the don't care set and therefore these are implicants.

Now let us take a few examples of a cube which are not implicant. For example let us look into the cube x1'x3', x1' meaning that x1 takes a value 0, x2 is not appearing so it can take any value and x3', x3 is taking a value of 0. Now so this corner and this corner. So this corner is okay, it is still in the onset but this one is a 0 set. It is a 0 set and therefore it is intersecting with the offset and therefore it is not an implicant. Now what

about x1'? So it means that x1 can take a value of 0, x2 can take any value, x3 can take any value.

So it represent these corners. So in these all these corners x1 is 0 and x2 and x3 are valid. Now again in this case these are in the offsets, these zeros are in offset and therefore this is not an implicant. Now let us look into what is a cover of a Boolean function. Now cover of a Boolean function is the set of implicants that includes all its minterms.



For example, consider a function here. So these two are the minterms, which corresponds to value 1 or the input combination or the corners at which the Boolean function takes a value of 1, those are basically the minterms of the function. So in this case these are two minterms. Now what is a cover? Cover is the set of implicants that includes all its minterms meaning that this can be included, this can be included. Now this is a set of implicants. So by definition as we discussed in the last slide implicants will cover only the onset or don't care set, it cannot cover a an offset.

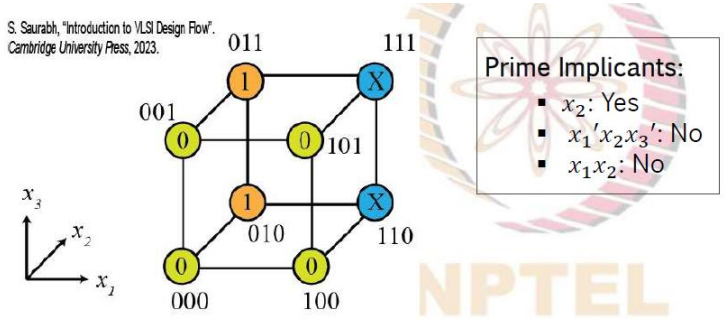
So cover will only be covering the corners corresponding to don't care set and the onset, it cannot cover any corner corresponding to the offset, because cover is a set of implicants. So what is a cover? Cover is a set of implicants that includes all its minterms. So if we take a set as set of these two minterms or these two implicants x1'x2x3' and x1'x2x3 these two elements in the set, then it is covering both ones and therefore this is a cover. Now let us take another example. Now in if we consider this, this, this as three implicants, then these three set of implicants cover both the ones.

Of course one is covering don't care, but that is okay. So this is again a cover, the set of these three implicants is a cover. Let us take another example. Now we take an implicant as x1'x2. x1'x2 means that x1 can take a value of 0, x2 can take a value of 1 and x3 can take any value.

So it correspond to this corner and this corner. Now this covers both the ones or both the minterms that were in the function and therefore x1'x2 is a cover of this function. Let us take another example, another implicant. So another set of implicants in this set there is only one element x2, only an element x2. It means that x1 can take any value, x2 will take a value of 1 and x3 can take again any value. So this corresponds to these four vertices in the Boolean hyperspace and since it covers these two ones and therefore this implicant is also a cover.

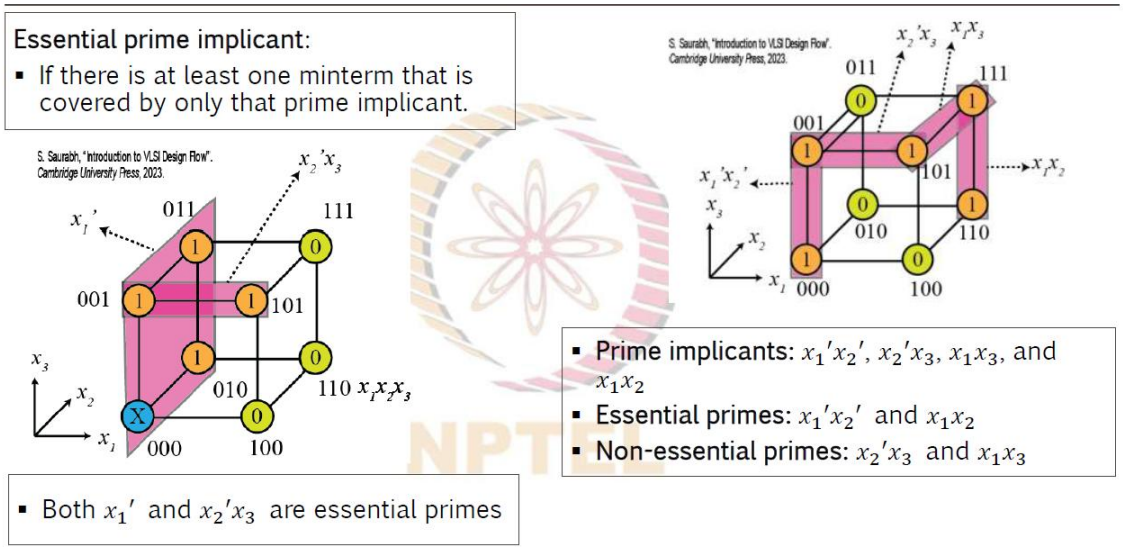
So the number of implicants in a cover is known as the size of the cover. So the size of this cover is 2, because there are two elements in this cover. In this the size is 3. In this the size is 1 because there is only one element in the cover and in this the size is 1. Now the cover with the minimum size is known as the minimum cover.

So in this case there are two covers with the same size. So this and this both are the minimum cover, these two are the minimum cover. Now let us look into another term which is known as prime implicant. Now what is a prime implicant? Prime implicant is an implicant of a function that is not covered by any other implicant of that function.



In short we also call them as prime. So now let us take an example. Suppose this is the function, this is the function. Now in this function if we look into x2, the implicant x2. Now x2 represents all these corners and this implicant is not covered by any other implicant of this function and therefore this is a prime implicant. Now if we look into say another implicant x1' meaning x1 = 0, x2 = 1 and x3 = 0, 010, this corner. So is it a prime

implicant? No, it is not a prime implicant because this implicant covers, this implicant covers the other implicant also, and therefore this is not a prime implicant.



Now what is a essential prime implicant? Now if there is at least one minterm that is covered by only that prime implicant then that prime implicant is known as essential prime implicant. For example, consider this function which we had seen earlier. Now in this case if we consider x1' then these three corners are covered or these, this is not the minterm, these two corners are covered only by x1' and therefore it is a essential prime implicant. And if we consider x2'x3, x2'x3 covers these two corners, and out of these two corners this corner is covered only by x2'x3 and therefore x2'x3 is also an essential prime.

Now let us take another example. In this case there are four primes x1'x2', this one the another prime is this one, another is this one and third one, fourth one is this. There are four primes in this case. Now which of them are essential in this case? We can see that out of these four prime implicants x1'x2', this prime implicant is essential because this corner is covered only by this prime implicant. And similarly this corner is covered only by the prime implicant x1x2 and therefore x1'x2' and x1x2 are essential prime implicants.

And the prime implicants x2'x3 that is this one is not a essential prime implicant. Why because this corner is covered by x1'x2' and this corner is also covered by x1x3, and therefore x2x3' that is this implicant is not a essential prime implicant. Similarly x1x3 is not a essential prime implicant because each of these corners are covered by other prime implicants also. Now let us look into having looked into the definitions of various terminologies. Now let us look into that what is logic or exact two-level logic minimization.

So the aim of exact two-level logic minimization is to find the minimum cover. So we saw that what is minimum cover. So for a given Boolean function we saw that there can be multiple covers, as in saw in the previous examples there can be multiple covers and different covers will have different sizes and out of those covers the cover which has got the minimum size that is known as the minimum cover and the goal of exact two-level logic minimization is to find that cover with the minimum size. Now why do we want to find a cover of minimum size because it correlates well with the circuit area. If we can minimize the size of a cover we will typically be able to reduce the hardware cost or reduce the area of the circuit. Now for simpler problems you might have in your digital circuit course you might have looked into various method of minimizing a logic function.

For example you might have done minimization using Boolean algebra methods for example, you apply De Morgan's law and commutative, associativity and other properties of Boolean operators and based on that you could have done the minimization or you might have used the Karnaugh maps to minimize the logic. Now these techniques work when the number of variables are small, but when the number of variables increases typically say more than 20 variables then conventional techniques are not of much use and in that case we need to use some systematic way to do logic minimization that will be looking at. Now to find the minimum cover we reduce the search space. Now out of all possible covers there are many covers possible for complicated Boolean function maybe say if the Boolean function is having more than 20 variables or so.

So in that case we need to reduce the search space where to look at for the minimum cover. So we reduce the space of exploration of minimum cover with the help of a theorem which is known as Quine's theorem. So let us understand what the Quine's theorem is. So to understand Quine's theorem first let us define a term which is known as prime cover. So what is a prime cover? Prime cover is a cover which is consisting only of prime implicants.

It will contain not any other implicants, but only the prime implicates. A cover which consists only of prime implicates that is known as the prime cover. Now what is Quine's theorem? Quine's theorem states that there exists a minimum cover which we are trying to find, consisting only of prime implicants, or there exists a minimum cover which is a prime cover. Now why it is true? What is the basis of Quine's theorem? We can make the following observation. Now consider a minimum cover assume that it is not a prime cover. Assume first we are saying that there is a minimum cover which is not a prime cover, against the Quine's theorem.

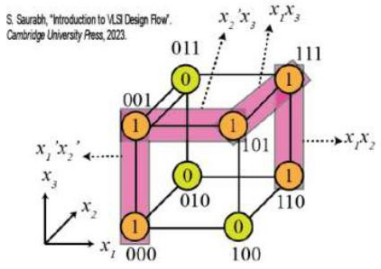
Then it implies that it contains some implicant which is not a prime implicant. Since a

prime cover consists only of prime implicants. Now we are saying that there is a minimum cover which is not a prime cover meaning that this minimum cover will have some implicant which is not a prime implicant. Now if that is the case then we can replace each non prime implicant with a prime implicant.

We can always do that. Why? Because prime implicant covers other implicants. So if there was a non prime implicant then that non prime implicant can also be covered by some prime implicant otherwise that would have been a prime implicant itself. So we can replace each non prime implicant in the minimum cover with a prime implicant that contains it. Now we got a new cover that consists only of primes.

So why it happened? Because we have replaced all the non prime implicants with the prime implicants. Now this minimum cover contains only the prime implicants. And what will be the size of this cover? The size of this will be the same as the original size. The size was now since we have replaced each of the non prime implicants with a prime implicants, so the size is not changing. But what we have done is that we made a non prime cover to a prime cover. Hence there exists a minimum cover that is a prime cover. So what it means is that now we derived another cover out of it that is now a prime cover. So given a minimum cover we can always come with a prime cover which is of the same size or it will be better or smaller size than other covers which contains non prime implicants. So what is the application of this Quine's theorem? The application is that we can focus only on the prime cover or we can only consider those covers which are consisting of prime implicants. So we can focus only on those covers that consists of prime implicants and therefore, our search space for finding the minimum cover that has reduced substantially.

We need not bother about the covers which are there and consist of non prime implicants. Now how to find minimum cover among the prime covers? Now we can do it by finding first the set of prime implicants for a given function and then building a prime implicant table. Now what is a prime implicant table? So we can build a prime implicant table by arranging the prime implicants in separate columns. So we have separate columns for different prime implicant 1, prime implicant 2 and prime implicant 3 and so on. And the minterms are in the individual rows. So we arrange minterm 1, minterm 2 and so on in row and we make an entry and we make an entry in the table as 1 if the given prime implicant covers the given minterm otherwise make it as 0. So let us take an example and understand. Suppose this was the given function and in which we have 4 prime implicants. These are the 4 prime implicants first one, second, third and fourth. Four prime implicants are there.



Minterms	Prime Implicants			
	$x_1'x_2'$	$x_2'x_3$	x_1x_3	x_1x_2
$x_1'x_2'x_3'(000)$	1	0	0	0
$x_1'x_2'x_3(001)$	1	1	0	0
$x_1x_2'x_3(101)$	0	1	1	0
$x_1x_2x_3'(110)$	0	0	0	1
$x_1x_2x_3(111)$	0	0	1	1

How to find minimum cover using prime implicant table?

- Find the minimum set of columns that covers all the rows in the prime implicant table.
- Solve set covering problem (can grow exponentially with the number of variables)

Simplification:

- Identify essential primes: ($x_1'x_2'$ and x_1x_2)
- Work on remaining minterms [$x_1x_2'x_3(101)$]: cover either using $x_2'x_3$ or x_1x_3]
- Efficient reduction and covering algorithms

And how many minterms are there? 1, 2, 3, 4 and 5 minterms are there. So we will build a table. First we arrange all the minterms in various rows. So we have 5 rows for 5 minterms. And then we have the prime implicants arranged along the columns.

So there are 4 prime implicants. So we have 4 columns. Now we make the entries of this table as 1 or 0. Now where do we make the entry as 1? For example, x1'x2' is this prime implicant and it covers the corner 001 and 000 and therefore, 000 and 001 these 2 minterms are covered and we make the entry as 1. For rest we make it as 0. Similarly x2'x3 is this implicant and it covers the corner 001 and 101 and therefore, we make these 2 entries as 1 and rest as 0. And similarly we draw make the entries of other entries in the table.

Now how to find minimum cover using prime implicant table? Now we have built the prime implicant table. Now how to minimize it? Now how to get a minimum cover out of it? So now finding the minimum set of columns that covers all the rows in the prime implicant table that is the job that needs to be done. Now we want to cover all the minterm then only we get a valid cover. So we have to select minimum number of columns such that all the minterms get covered.

So this is the problem. Now this problem is same as set covering problem. Set covering problem which is studied extensively in graph theory and this is the same problem that we encounter here. So this problem is known to be an NP complete problem or a difficult problem. And the complexity of this problem increases with the number of variables. Why because as we increase the number of variables, then the prime implicant can increase greatly or exponentially and the number of minterms can also grow

exponentially.

So the table size will become huge. So if the table size becomes huge then finding a solution for set covering problem also becomes difficult. And therefore this set covering problem can be solved for simpler Boolean optimization problems. So now this is a difficult problem and we can make some simplification or simplify the task for set covering problem. What we can do is that first notice that if there are some essential primes those will always be there in the cover.

Why it has to be there in the cover? Because there are some minterms which are covered by only this essential primes. So in this case we had seen earlier that this prime and this prime these two were essential prime. Why because this corner was covered only by x1'x2' and this corner was covered only by x1x2. And therefore, these two primes will always be there in the cover. So we identify these essential primes and then get rid of the the rows which are already covered by this, by these two essential primes. So in this case this is already covered, this is also covered and if we select this prime also then this is covered and this is also covered.

Now we need to work on the remaining minterms. So in this case the remaining minterms is only 101. So for 101 we can take either this or this, any of that will do. So if we take the cover as this set, this and this or we can take this, this and this. There are two possible minimum cover in this case. So there I just gave one example of the reduction technique that is based on essential primes. There are other many efficient techniques which try to reduce the number of variables or reduce number of vertices that needs to be handled by the set covering problem.

So with the help of those those reduction techniques and some heuristics in the covering algorithm this problem of exact logic minimization can be tackled efficiently. However, when the problem size becomes very large, in that case suppose say number of variables is more than 100 or the number of minterms is say more than 1000 or may a number of cubes that needs to be handled by or number of primes that needs to be handled by the minimizer is say more than 1000. In that case we do not do exact logic minimization or we do not try to find the minimum cover. What we try to do is what we try to do is take help of heuristic minimizer and what is heuristic minimizer? So for large problems we prefer heuristic minimizer over exact minimization and heuristic minimizers are faster for large problem sizes and we often do not need the exact minimum cover. So many a times what happens is that say suppose there is a cover which the exact theoretical minimum cover size is say 980 and if a heuristic minimizer which tries to find a minimal cover rather than a minimum cover and the number of elements or number of cubes in the cover or number of implicants in the cover is, instead of 980, it is say 988.

It is not exactly same as the minimum cover but slightly more than that. It is still okay. So there will be slightly a small area penalty or there will be some scope of logic minimization that is still left but still it is okay. So we often do not need exact minimum cover. Any solution that can be found quickly and is near optimal is acceptable. So in this case rather than spending an hour if we can find a approximate minimal minimum cover so say instead of 980 implicant we have say 988 implicant.

Then also for most application it is acceptable. And that is why we go for heuristic minimizer which are very fast compared to exact logic minimization and they are able to do minimization fairly well not exact logic minimization but close to theoretical logic minimization. Now the heuristic minimizer tries to find a minimal cover rather than the minimum cover. And what is a minimal cover? A minimal cover satisfies certain local minimum cover property rather than the global minimum property. And an example of this local minimum cover property is that a cover in which no implicant is contained in any other implicant of the cover. So we find a cover in which there does not exist any implicant which is covered by any other implicant.

If there is such an implicant we get rid of that implicant safely from the cover and then the cover that we get it is known as a cover which is minimal with respect to single implicant containment. So the heuristic minimizer tries to find a minimal cover which is minimal with respect to single implicant containment meaning that no implicant is there in the cover which is covered by any other implicant of the cover. So the size of the minimal cover can be more than the size of the minimum cover but we are fine with it for because if the minimal cover is still close to the minimum cover we are fine with it. And for many applications or many practical cases this heuristic minimizer is able to find a minimal cover which is close to the minimum cover. Now what is the approach of the heuristic minimizer? It basically starts with an initial cover it need not be a prime cover it can be initial cover and it iteratively improves the solution by applying some operators on it.

So we start with some cover which may not be a prime cover and then we successively apply various operators on it and we apply it till we are not able to make any improvement. So the iteration terminates when the algorithm can no longer improve the solution. And what are these operators? These operators are for example, the expand operator.

So in expand operator what is done is that it expands a non-prime implicant. So we started with the cover which was not a prime cover. So there will be some implicant which is not a prime implicant. What does expand do? It expands a non-prime implicant

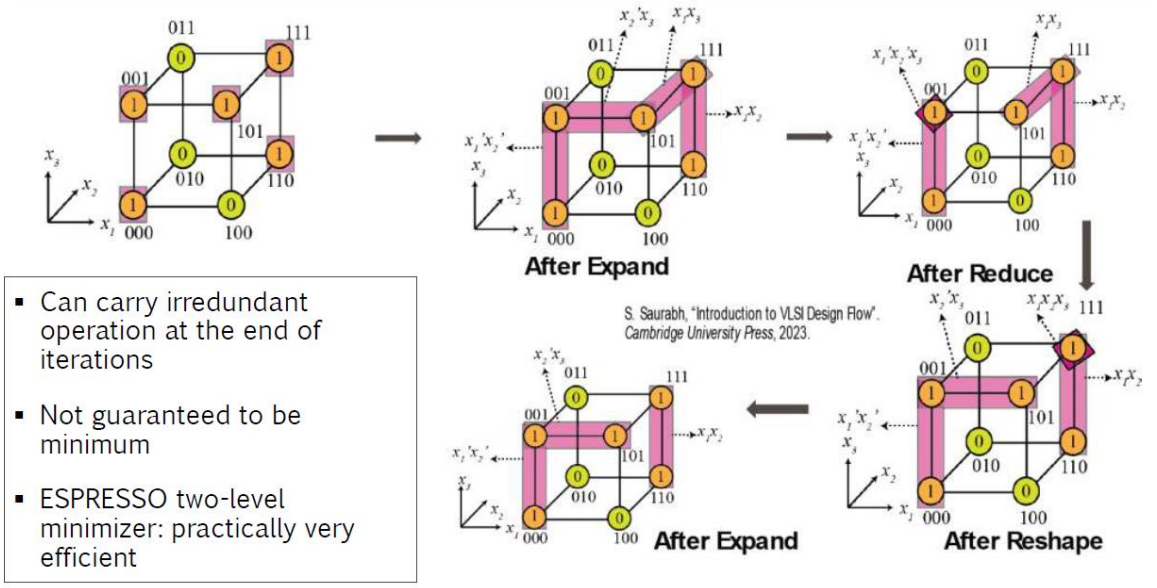
to make it prime and as a result of it, it removes implicants covered by the expanded implicant. So the expand operator will make a non-prime implicant as a prime and it will remove the implicants which are already covered by other implicants or the expanded implicant.

Then the other operator is the reduce operator. So what it does? The reduce operator replaces an implicant with a reduced implicant. So it will cover fewer minterms. So it will cover fewer minterms. So as the cover size will not change, and the function will still be covered. Why? Because it replaces an implicant with a reduced implicant such that the function is still covered.

So it is not removing any or it is not reducing an implicant such that it makes a corner which was covered by it, uncovered. So it does not reduce the size of the implicant such that it removes or it uncovers a vertex which was solely covered by it. So it means that if it is reducing, it is reducing from a vertex which was already covered by some other implicant also. So we carry out this reduce operator so that in successive operations a better solution can be achieved.

Then there is an operator which is reshape. It operates on a pair of implicants. It expands one implicant and reduces the other such that the function is still covered. Note that the size of the cover remains the same. And next is the irredundant operation. It makes the cover irredundant.

And what is an irredundant cover? It is a cover in which if we remove any implicant then it will be no more a cover. Meaning that if there is an implicant which covers the corners which is covered by a set of other implicants of the cover then we say that that particular implicant is a redundant implicant and we can remove it. So the irredundant cover removes a redundant implicant. And after it has removed all the redundant implicants, the cover that we obtain is known as irredundant cover. And in an irredundant cover if we remove any of the implicant out of it then some minterm will be left uncovered then it will no more be a cover.



- Can carry irredundant operation at the end of iterations
- Not guaranteed to be minimum
- ESPRESSO two-level minimizer: practically very efficient

Now let us take an example like how the heuristic minimizer works. Now let us take this function in which we have say 1, 2, 3, 4, 5, 5 minterms. And we start with a solution which is consisting of only the minterm, sum of minterms form. So there are 1, 2, 3, 4, 5, 5 implicants are there.

Now we apply the expand operator. So this expanded in this, this expanded this, this expanded to this, this expanded to. So the after the expand operation and removing the implicant which gets covered by expansion we get a cover. Now earlier we had 1, 2, 3, 4, 5 implicants. Now we have 1, 2, 3, 4 implicants. Now from 5 we move to 4 after expansion.

Now after we reduce this particular implicant. So it now covers only this minterm and others are unaffected. So the size of the cover is still 4. Now after reduce we do a reshape. In reshape what we do is that we expand this minterm or this implicant and reduce this implicant we reduce this this implicant. So we still get after reshape the number of implicants that is in the cover, the cover size is not changing it is still 4.

So we have one implicant and this one got reduced to an implicant which covers only 1 minterm and we have the fourth implicant. And then again we apply an expand operator and once we apply the expand operator then we get rid of this implicant and finally we have 3 implicants in the cover or the cover size is 3. So we started with 5 then we got to 4 then got to 4, and finally we reach to a cover size 3. Now in this case there is no redundant implicant if we remove any of these implicant it will no more be a cover. But typically what is done that after we have carried out an iteration of expand, reduce,

reshape, expand, reduce, reshape, multiple times at the end we again carry out an irredundant operation so that if possible some implicant can be reduced.

So in this case it is not required to carry out an irredundant operation or it will not yield any improvement. Now in this case it turns out that we are getting a minimum cover also in this heuristic minimizer. But it is not that we will always be getting a minimum cover using this heuristic minimizer. Now if you want to go deeper into these topics, this is a very good reference and you can also look into this reference. Now to summarize in this lecture what we have done is that we have looked into the logic optimization task and in particular we looked into logic optimization or logic minimization of two-level combinational logic.

So in two-level combination logic we looked into two types of minimization one was exact minimization and the other was heuristic minimization. So in the next lecture we will be looking into another logic optimization technique that is multi-level logic optimization. Thank you very much.