**Optimization Techniques for Digital VLSI Design**

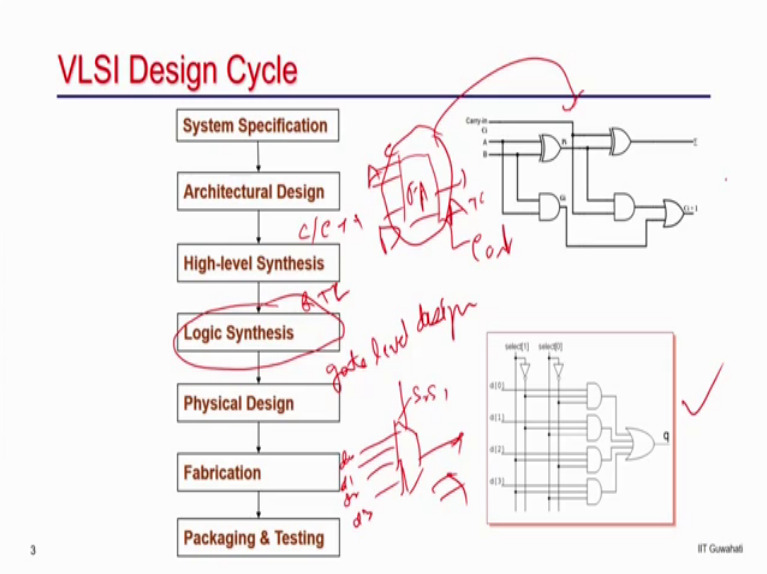
**Dr. Chandan Kafra**
**Dr. Santosh Biswas**
**Department of Computer Science & Engineering**
**Indian Institute of Technology, Guwahati**

**Logic Synthesis: an Overview**
**Lecture – 10**
**High Level Synthesis: Introduction to Logic Synthesis**

Welcome everyone. So, we are going to discuss today, logic synthesis. So, in today's specifically, in today's discussion, we will talk about logic synthesis, in basics and then we will talk about specifically two, kind of optimization, what is called two level optimization and multiple optimizations.

So, if you look into V L S I design cycle, which we have discussed in our module 1 that we start from our initial system specification and then we have some architectural design. And then we have say, this specification of the upper behavior, then we do high level synthesis right. So, here we have some architectural design, which you can represent by C C plus plus and then we convert this through high level synthesis to R T L ok, which is called register transfer level design.

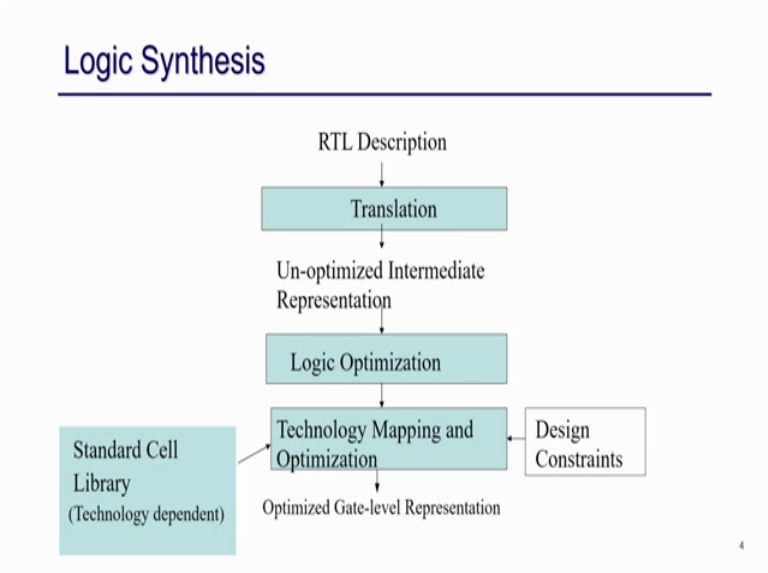
(Refer Slide Time: 01:12)



And then the steps come the logic synthesis. In logic synthesis what it does? It takes this R T L and convert to this gate level design right, gate level. We just convert this R T L into gate level design and then we will do physical synthesis fabrication and packaging.

So, the today's discussion topic is this logic synthesis. So, where it do? What it does is, it just takes some R T L register transfer level design, which may be generated through high level synthesis or some, design I can write the R T L architecture as well right and this logic synthesis tool, which take this R T L and convert them to equivalent gate level design right.

(Refer Slide Time: 02:02)




So, if you look into this translation, this from R T L to gate level, which is basically very obvious right. I mean there is nothing there is no much complexity is there. For example, suppose, in your R T L, there might be some multiplexers right. So, you might have some mass here, which is basically taking say, taking say 4 inputs and then we have say, select line of two bits say 0 and S 1, which select this input i, this d 0 d 1 d 2 and d 3, based on this select value right.

So, this is one mass in your R T L, but you can actually replace, this mass by equivalent gate level design, where this is that equivalent gate level design. So, where we have this, this selection based on this select line and the inputs and then there is a odd right.

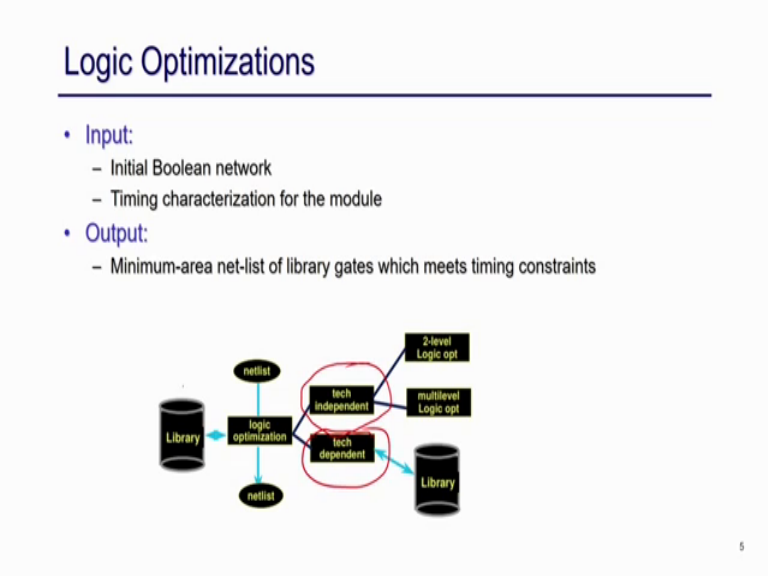
Similarly, suppose you have a full adder right. So, you just do adding 2. There might be some full adder, which just take input A and B and some carry input C. I am doing this A plus B right and some carry out right. So, you can replace this R T L module by equivalent gate level design, this is the full adder circuit, this is for 1 bit and if you have this say 16 bit or 32 bit that will be just chain right. This will be the same block, will be chain, I mean just as a carry input right.

So, is basically the concept here is that in logic synthesis, when you try to convert this R T L to it is equivalent gate level, we can just replace those block by block right, mass by equivalent gate level implementation or say this adder, it is equivalent gate level implementation. There is a multiplier it is equivalent gate level implementation and so on. So, there is nothing much complexities is there right. So, as here description this translation is kind of pretty obvious right.

So, we have this R T L description and we convert it to gate level through this one to one mapping, of this block equivalent gate level implementation, but what is important here is that this optimization. So, once whatever the gate level, the design your going to get it that may not be optimum alright. So, then we have to apply lot of optimization technique, just to reduce the number of gate, because if we just replace all this module by it is equivalent gate level design, those implementation will be very much optimize and there will be lot of dedundance right.

So, the primary objective of this logic synthesis is this logic optimization, when you talk about this logic synthesis, we primary talk about this logic optimizations, because now, we talk about you have this un optimized gate level design. We have and we have to apply such kind of optimization such that you can reduce the overall size of a design; that means the number of gates or may be number, I mean the total time in delay. So, there might different kind of optimization goal, but overall this main primary objective of logic synthesis is to this logic optimization right.

(Refer Slide Time: 04:55)




So, in this logic optimization also have two type of optimization, one is called tech independent, and there is second tech dependent, technological dependent and another one is technology independent. So, we will talk about this technology, independent optimization today. So, in technology independent optimization, what we do is just think about that we have that library of gates are there may be, we have only and gates or gates and we try to replace the whole thing, using those equivalent gate.

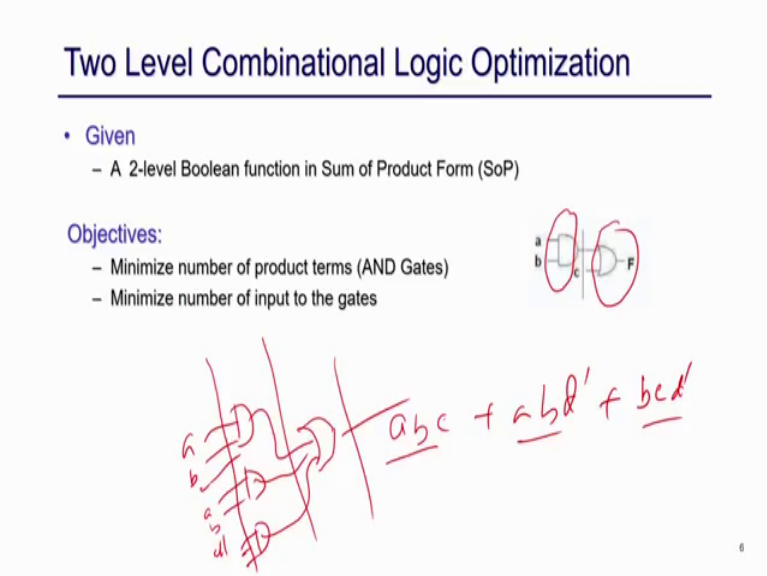
For example, say maybe you have only say, nan gate right or nan or node. So, you try to replace whole your design, using nan and node gate and so on right. So, so that is and then you try to do all optimization without looking into the actual technology dependency tech technology, when your target architecture right. Your target architecture say F P G A or D S Ps also we are not going to consider about those technology dependent thing in the technology independent thing.

But in technology dependence for example, say F P G A, we have say L U Ts, if you have D S Ps, if you have say RAM blocks, block RAM. So, when your are talking about those mapping or optimization, we try to map those and gates or gates and gates to that L U Ts right, because those are the technology unit of that particular F P G A. So, the idea is that we have kind of two step, on first step to optimize without considering the target architecture target target devices. We make it indefinite of the technology target, technology and then we do this technology dependent optimizations ok.

So, today we are going to primary discuss on this independent part and there are primary two kind of optimization is important in this technology dependent, one is called two level logic optimizations and the second one is multi level optimization, multi level logic optimization right.

So, if you just think about this or logic optimization step. So, what is our input, we have a Boolean network so; that means the circuit, where in terms of Boolean gates and we have some timing characteristics of this modules right. So, for each module, what kind of what is the kind of delay and all those informations and then we try to minimize the area right. We try to minimize the area of this overall circuit, the Boolean network which actually meets the timing constant, that is the overall objective here. So, and we will talk about this two level and multi level logic optimization in this context right.

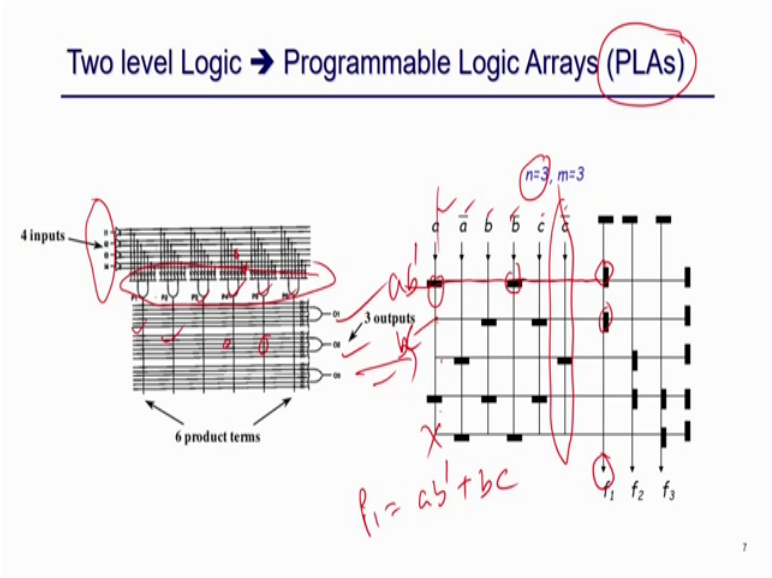
(Refer Slide Time: 07:18)




So, what is two level combination, optimizations, logic optimizations. So, in the two level LO combination logic optimizations, we represent this Boolean formulas as a sum of product form right. So, we have set of product terms and then we have this summation of this. So, this product is basically and gate right. So, we have this; that means, in the logic level, we have two level, I mean two layers. So, in plane, we have the AND gates set of AND gates and the next level, we have the, or gates right. So, this product. So, basically if you just think about say a b c plus a b d dash plus a b c d dash and so on.

So, this are the product terms right and this is sum of product. So, this product is basically nothing, but the, and of this inputs right. So this is, this and plane, we have this a b c d here, a b d dash and so on right, and in the, or plane, we are going to have the, or of all this right. So, this is basically two layer, two level of your representation right, the, I have and level, and then followed by the or level right. So, the idea is that, in two level optimizations, we always represent those Boolean formula.

(Refer Slide Time: 08:39)




In this sum of product from the (Refer Time: 08:29), of the all the products and then we have the summation or the, or gates and, and we try to optimize the circuit in this in this representations, just to, just to explain this two level LO logic optimization more, I mean more clearly. So, we have the set of inputs and they are, I mean they have and their complement and this is the, and plane.

So these are all this kind of product term. So, this is one product, this is one product, this is one product, this is one product and one. Let us say here, some of them may be one right, some of them will be taken care of and then in the odd plane, some of the product will be important for this output right, for this output there may be some product is important right and so on, it is clearly mentioned here.

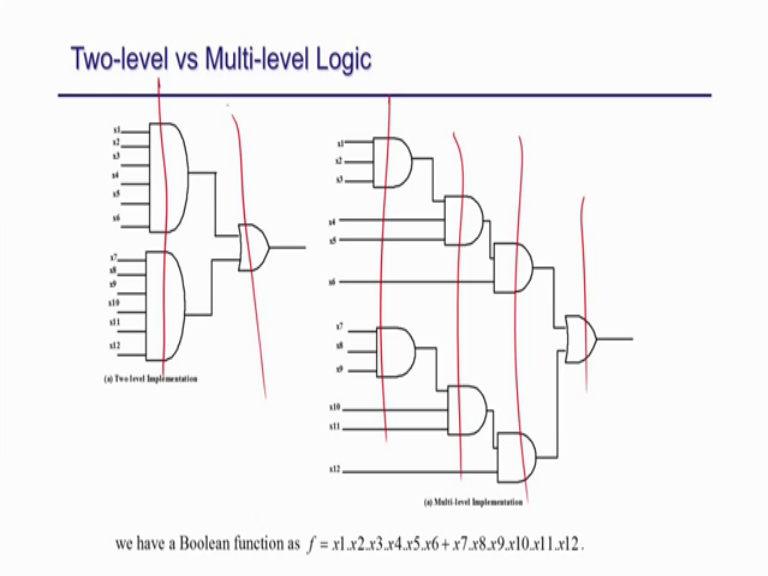
So, for example, here I have 3 input. So, a b 3 inputs. So, their complement, there are total 6 input lines and this each row represent one product term right. So, for example, this product, we have a and we have b bar right. So, this is basically, represent a b bar

right and this is basically b and c right. So, these represent b c and so on. So, each row of that particular plane is representing one product term and now, this particular product output, this is the or gate. So, now, it is, it may not take all the products. So, there are say 6 products here, 1 2 3 4 5 5 product sums are here, and this is taking only these two product term right.

So, this f 1 is effectively means a b bar plus b c right. So, f 2 is, is basically taking this product term, and this product term and so on right. So, this way I can represent my overall circuit in this two level representation. So, it is basically called programmable logic array, when we have this and plane and plane and odd plane and what is the optimization goal here, we try to reduce the number of row right. We try to reduce the number of products; because if you have number of row is less then the size will be less right.

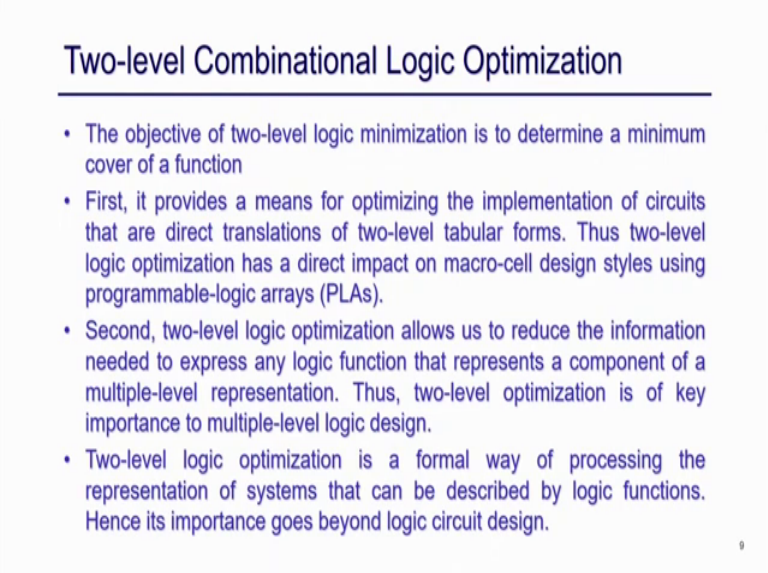
Similarly, I want to reduce the number of column as well; that means, the number of input literals right. So, this a dash, all this kind of call is literals. So, the objective here is that to reduce the number of product terms; that means, if we have number of row is less; that means, the number of and gate will be less is effectively mean the number of and gate will be less. So, it will have a less area and then it is also, if possible then minimize the number of input gates, some time we try to instead of all this, all 6 may be in some scenario. I do not need this particular column right or in some scenario. I do not need this particular row. So, this is how we can actually optimize the two level circuit right.

(Refer Slide Time: 11:04)




So, on the other hand in the multi level when here, you can see there are only two levels right. So, in multi level, we can have multiples level of gates right. It is not, not only two levels.

(Refer Slide Time: 11:22)

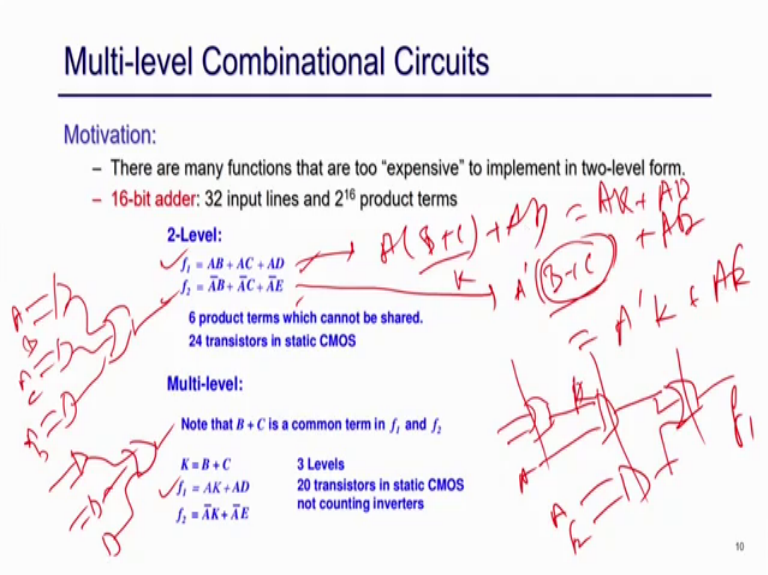



So, and in multi level optimization, we try to do not consider or representation in just two level, we can have multiple levels of gates and we try to optimize the whole circuit in that mode right. Why this two level comb, combinational logic optimization is important, because I mean it. It actually means of optimizing the implementation of

circuit that have direct answers, some of the two level tabular form. So, this is how we actually visualize our circuit right. So, this is and it also have a very one term corresponding to this P L As.

So, for P L A optimizations, this two level optimization is important and although for multi level, even for this two level, optimization is also important, because that this two level, optimization is the key for key important factor, for multi level logic implementation also right. So, basically uh, this when we are taking about this multi level optimizations, we effectively in the background, we componage component is represent as a two level function. So, it is important to understand this two level optimization logic optimization as well ok.

(Refer Slide Time: 12:17)




So, in my, but as I mention this, this is something, is not kind scalable two level optimization right. For example, there are too many too expensive implement in two level form right, in sometime, this function cannot be represented in two level two level form. For example, this you, just think about the example of this 16 bit adder. So, in the 16 bit adder, there are 32 input lines right. So, uh, there are two inputs. So, 16 bit, 1632 bits and then we have put your 16 product terms and then we are going to select some of them right. So, it is basically, you understand just for adder, you have divide 16 number of products term. It is too costly sometime right.

So, you may not have the efficient implementation two level implementation of all function sometime ok. So, that is why I mean, we try to we actually represent this, as Boolean circuits is in multi level. I mean multi level combinational circuit. I mean is in combinational mul, multi level circuit right. A D also sometime, if we represent this multi level, we can have some. We can actually optimize the overall circuit as well just to give an example here. We can see here, there are two function uh, and there are like this right.

So, there are 6 product term and we cannot optimize any of them right. So, I need 6 product terms. So, 6 and gate and 2 or gate to just to represent this, but in multi level what we can see, we can see that there is a common terms here is a right. So, we can represent this as basically A into B plus C right plus A B and f 2 can be represent as this A bar into B plus C plus A E right. So, I can, there is a common term right. So, now, if I represent in three level.

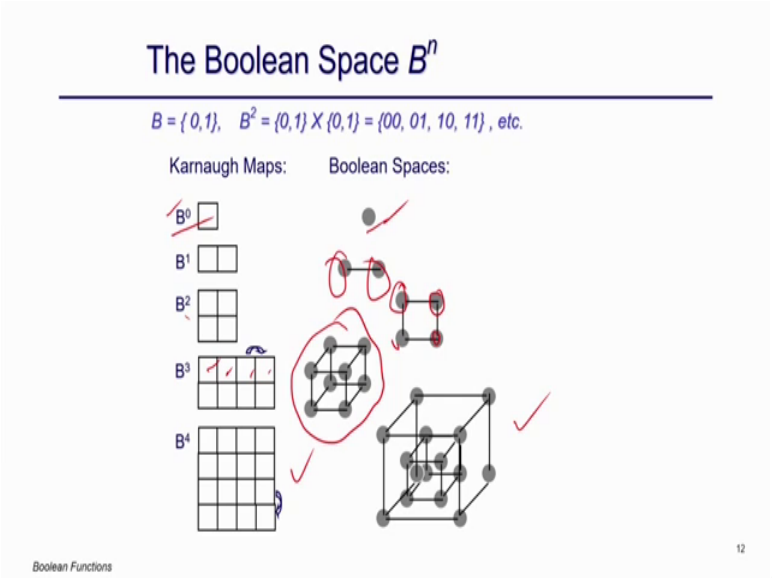
So, I can compute this B, B plus C first right. So, this is my B plus C and then I can do this f 1 is basically nothing, but this is say, this is K. So, A K plus A D right and this is a dash K plus A E right. Now, I can do this. So, instead of just having this, this is basically, if you just think about the actual implementation, I have two level, I have this A B A C A B A C and A D and I have, this is my say f 1 right and similarly, have this three term terms for f 2 right I can have this.

So, on the other here, what I can do? I can just have this, is my D and now, I have this and gate to just to do this say, this is my K right. This is what I just think about, this is K. So, this is my K and now, this is A and then I am just going to do this A E (noise) and this is my (Refer Time: 15:07), this is how I can represent my f 1. So, instead of this two level, I have now, three levels right. Similarly, I can use the same K to represent the f 1. So, when you have this multi-multi level implementation, you can actually have this kind of common sub expressions, which you can actually use for multiple functions and that will give you more optimization right.

So, that is what is the advantage of multi level optimization. So, we are going, you learn both this two level combinational circuit optimizations and multiple level combinational circuit optimization. Today, before going into that we are going to discuss little bit about

the or basic knowledge of Boolean logic, because that will be useful when we are going to discuss about this two level logic optimizations and multi level logic optimizations ok.

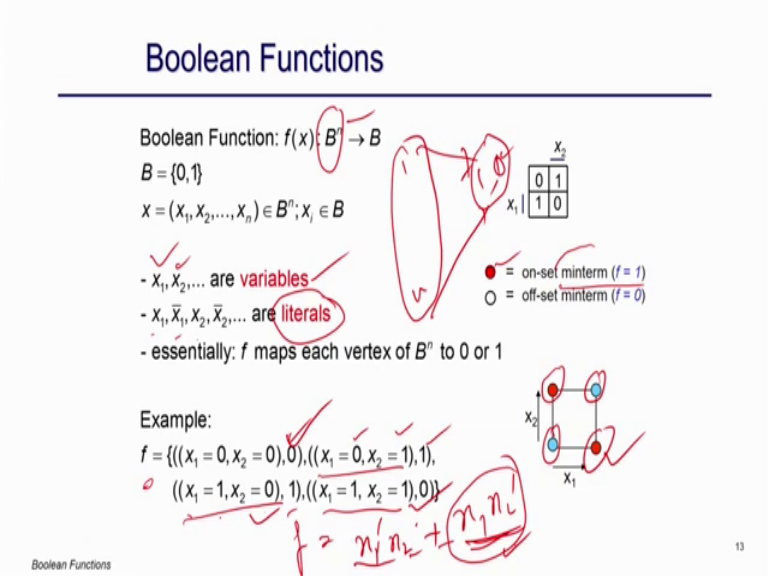
(Refer Slide Time: 15:49)




So, I mean if we just think about the Boolean space. So, if we have say n variable, we have to put your n values right. I mean either this is 0 or 1 right. So, if you just think of the space, if you have only 1 variable, 1 Boolean variable, either this is 0 or 1 right. If you have 2 variables then we have sorry, this is 0 and 1. This is just true right. This is true or false. This is 1 variable, then true or false, if it is say 2 variables either one of them 0. So, there will be 4 values right.

Similarly, there are 3 variables; I have 8 possible values right. So, I can represent this is kind of hyper-hyper graph. Similarly, for 4 variable, I can represent, I have this kind of table tabular form or I can use a hyper graph right. This is called hyper graph. So, this is how you can represent, this Boolean space.

(Refer Slide Time: 16:42)




And then what is Boolean function, it takes some n inputs, some n uh, function of say n Boolean variable 1 to n and that will map to 0 or 1 right. This is 0 or 1. So, this will map to this, this is what is call Boolean function. So, it depends on all this. So, it depends on all this n Boolean variable and that (Refer Time: 17:05) function will map this to HM 0 or 1. So, now, if you just think about the n input variable, we all, we will say this variable X 1 X 2 to them is variables and this X 1 X bar, X 1 bar X 2 X 2 bar as the literals. So, this terminology is important and, and also if.

So, if you say for example, if I just give an example for this f, f is basically what is saying this X 1 bar X 2 bar right, X 2 bar plus. So, basically, what is try to say is this. So, this function is saying that whenever this value is X 1 bar and X 2 bar then output is 0, if X 1 bar is 0 and X 2 is 1 then output is 1 and so on right. So, X 2 1 is 1 and X 2 is 0 then output is 1 and both of them are is 1 then output is 0 right. So, I can represent this f is basically, X 1 bar X 2 and this right. So, this 1 is the on term. So, this is X 1 and X 2 bar right. So, this is, what is the functional.

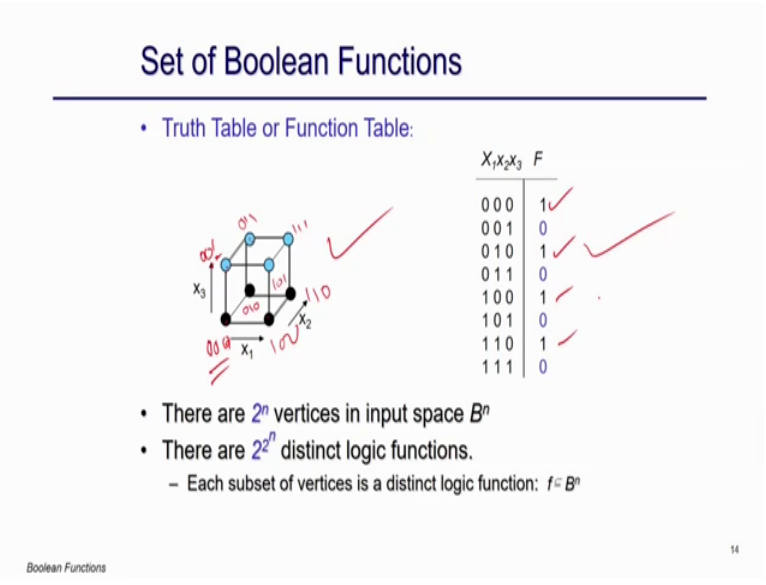
So, we will call this, this all, this term is called mean terms. So, mean terms is that is the product form, where all the variables are present right, either in true form or in bar. I mean either complement or actual form. So, they are present, then we will say that this are mean terms right. So, if there are 2 variables, I have 4 possible values right. So, there are 4 possible values here and on terms is this when represent by red here, where if the

function is 1, the on term is also be 1 right. So, for example, if this function is 1 this or this, if this is 1 sorry, in this, if this mean term is 1 the function will be 1 right, that is what?

This sum of product means, if any, if them is 1, the output will be 1. So, on set is those mean terms, which is basically 1, when the, if that is 1, then the function will be output will be 1 right and offset is what the other one right. So, this is offset, offset, when the value is 0. So, I represent them in the hyper graph like this right and we can also have some donned cap, which we are going to discuss later.

So, the idea is. So, key point here, that the Boolean function, taking in Boolean variables map. It to 0 to 1 and that variables and the literals is that X variable or the complement form and the onset is basically those mean terms, those are, when they are one the function is 1 and the offset is when the function is 1, this, this is 0 definitely, 0 right. So, that is, what is, we represent the Boolean function.

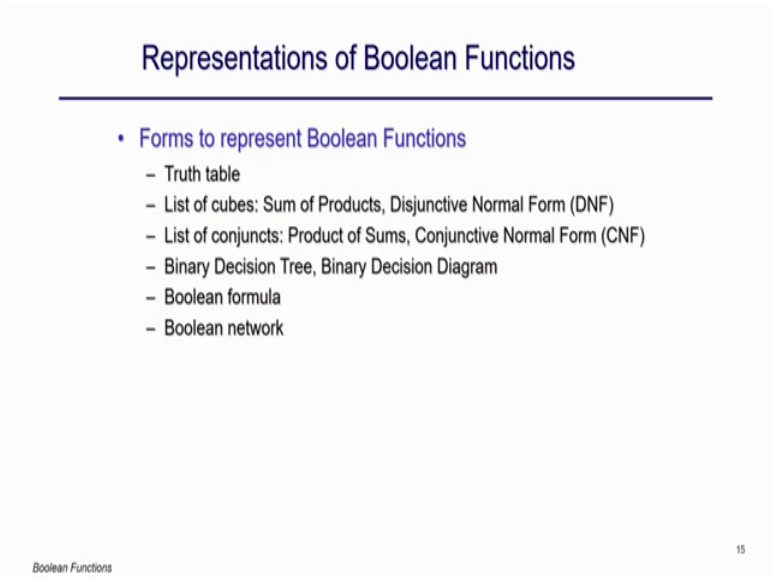
(Refer Slide Time: 19:41)




And how we represent this Boolean function, usually use either graph or say truth table right. So, for example, here you can see here there are three variables and this variables. So, this is, this point represent is 0 0 0; that means, X 1 bar X 2 bar and X 3 bar. This point is representing X 1 bar X 2 and X 3 and so on right.

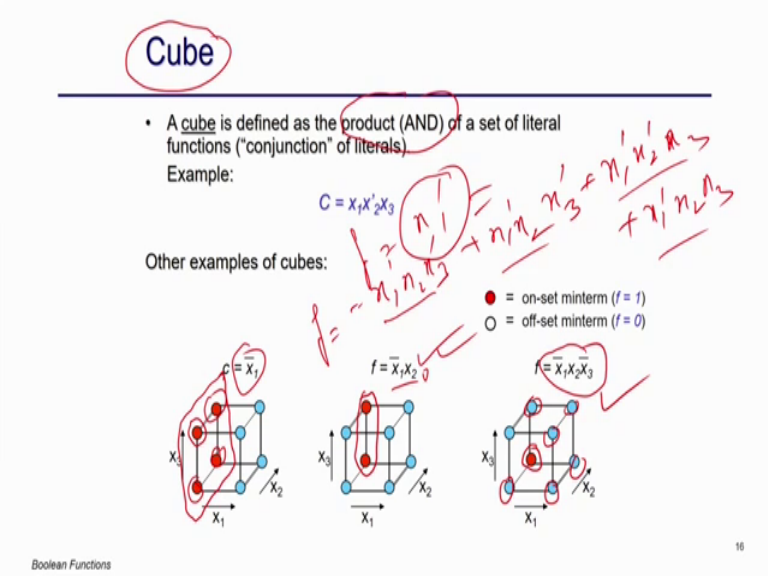
So, this is how the hyper graph is represents and here you can see that between two consecutive 0.2 adjacent points, the only difference of 1 bit right. For example, this point and this point, the only difference is between this, this bit from this point to point. There will be a difference between a, this middle bit right and so on. So, this is how, we can represent this and here this uh, this black dots are the onset and this greens are the offset. For example, this is onset, this is onset, this is onset.

(Refer Slide Time: 20:32)



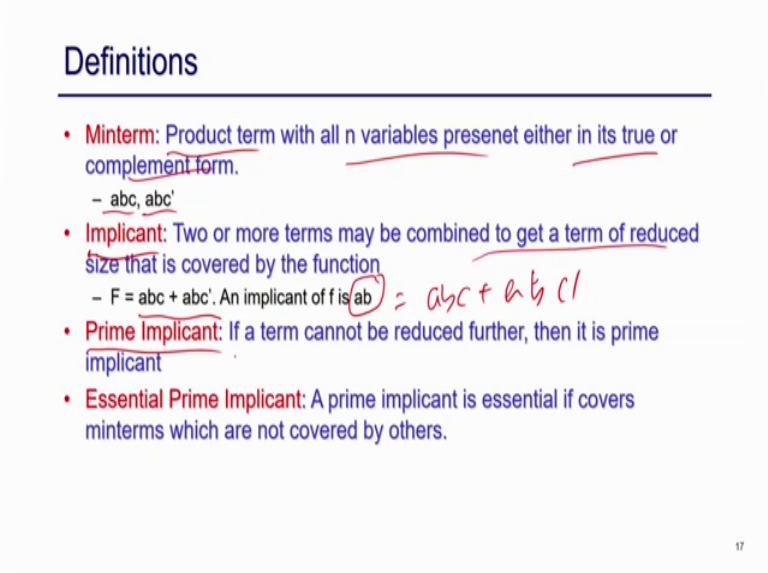

And this is onset right. This is how, we can represent the thing. So, there may be various other options this is one is truth table one is sum of product or product of sum or binary decision tree B D D binary decision diagram, Boolean formula or Boolean network right. So, you can represent this functions in various form usually, we I represent by this hyper graph or say truth table or say sum of product form or say binary decision diagram and all there may be various way to represent this Boolean functions right.

(Refer Slide Time: 20:58)




Now, you come to discuss what is cube right, if you just my Boolean function as a this hyper, hyper graph then each minterm right.

(Refer Slide Time: 21:11)




So, minterm is something, what I just define. Here is basically is a product term, where all n variables present, either in true or their complement form. For example, (Refer Time: 21:18) there are three input variables say A B C then A B C is a product term A B C dash is product term right. So, those are min terms right. So, in the cube, this is also a

product term right, but that may not contain all the variables, in the true or complement form. For example, if you just think about this X 1 bar X 2 and X 3. So, this is that note.

So, this is a product term and that is also a cube right and now, if you just think about, I have only 2 variable, this will represent 1, this tool thing right. So, this is what is X 1 bar and X 2. So, because this X 1 this is X 1 bar and X 2 right. This is 2. Similarly, if you just think about my cube is only X 1 bar then it will represent the whole plane right. This whole is representing by this right, this is basically cube is also a kind of product term, but it may not contain all the literals right so, and if it has less number of literals it actually covering more than one minterm.

So, in the hyper graph each monode is kind of a minterms right. So, all because in this point, all the variable has either, they have in from true form or their complement form and if we have cube, which is less number of variable, then it is actually covering more than one term. For example, if you have 1 less here, it is covering 2, there are two less here two less variables are there its covering a one space itself right. So, this is how, we are going to represent this right and now, this is this is important, because the concept of implicant what is minterms, when two or more terms may be combined to get a term of reduced size that is covering that a function right.

So, for example, suppose I have a function A B C and A B C bar. So, an implicant if this is A B, because A B is basically, A B C plus A B C bar. So, it is covering both of them right. So, this cube is kind of a, an implicant. So, this cube either can be a minterm or it can be implicant. For example, there here, this, this is an implicant, because now, it is covering two minterms. This is also an implicant, because it is covering four minterms right. So, it is basically, we try to we have the minterms, because minterms are the product terms and that has actually, if you represent your function in some of product associative form then this minterms is the onset right.

Those minterms in the sum of product terms are there, those are the onset of the particular variable and we try to find implicant, which actually try to cover more than one minterms right, that is what is something is called implicant and then what is important, there is prime implicant. So, we try to grow right. For example, if you have say, this 4 variables are there, initially you might have minterms all 4 right. So, individually they are all minterms right, then you try to grow it for uh, you cover a

minterms or say implicant, which cover only this 2, then you go got an another implicant, which covering this 2, then you can extant further explain, further you got this minterm. X 1 bar, which is covering all this 4 right.

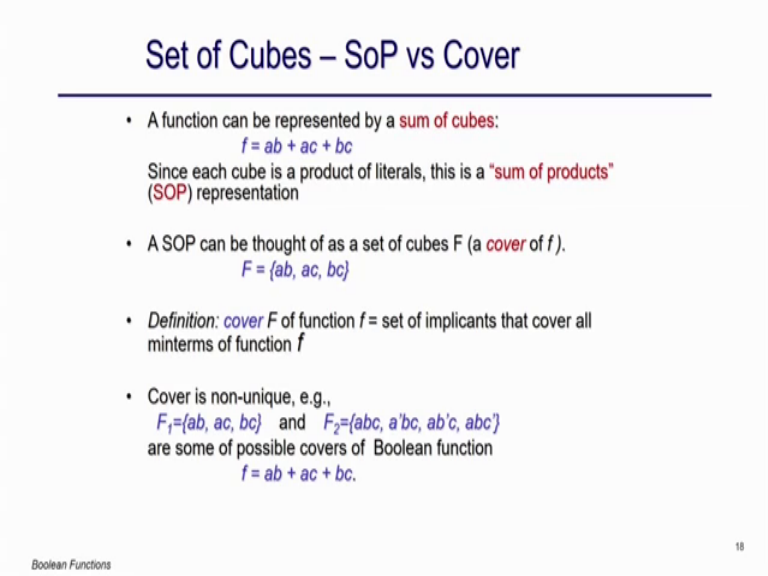
Then this is called prime implicant, because now, I am not able to go further, because this is even if I grow, it will not, because if we assume there are only 4 onset here, even if I grow, then I am going to cover some offset right, that is not imp that is (Refer Time; 24:33), that is not correct.

Because all objective is this onset are important, because if those any of them is one, the function will become one. So, we will only bother about, this onset. We never try to cover any offset, because that will actually change the function rate of your of the function right. So, the important point, here is that. We have the minterms, which is the sum of this product form, then we try to find the implicant, implicant is basically an product term, which try to cover more than one minterms and then it go, for the and then it got a saturation point, which is called prime implicant, because I cannot try to reduce further number of variable there.

So, that I can cover any other new on term right, if we try to grow further, it will cover some offterm, which is not correct. So, that is called primary implicant as I explained in this diagram right and then there is term called essential primary implicant; that means, if there is a prime implicant is essential, if it cover some minterm, which are not covered by others right. You might have two prime implicant, which have some subset right. For example, suppose, you have say one implicant is like this. So, let me just draw a share.

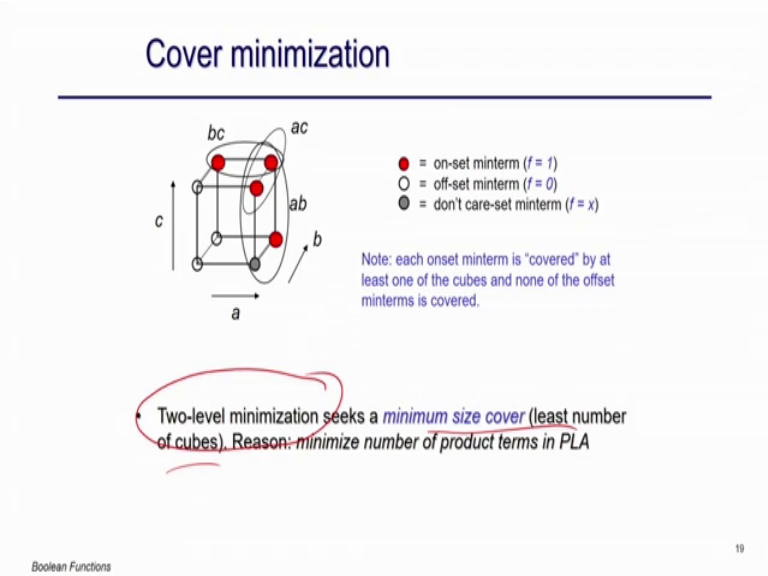
Suppose, this is my onset here, I have this, this is my one implicant right, because it is covering this 2 and then I have another implicant like this right. Now, this is, these two are basically essential implicant, because they are covering some minterm, which is not covered by any other minterm. So, they are called essential prime implicant right, because. So, they are called essential and now, since this not prime, because I can have a bigger of implement there. But that is essential in the sense that you have some implicant, which is covering some minterm, which is not covered by somebody else right, that is what is called essential implicant.

(Refer Slide Time: 26:17)




So, what is the objective as I mention here, I mean earlier also that we have those minterms right. We try to find out those implicants, which actually cover more than one minterms and we try to grow it and try to find the prime implicant right, because those prime implicant is the biggest cover right, because that is, that is covering maximum number of impli minterms and I cannot grow it further right. So, our objective is to try to find those minimum set of such cover that is covering all the minterms and so, that I and I can represent that particular function, using those prime implicant only right.

(Refer Slide Time: 26:55)



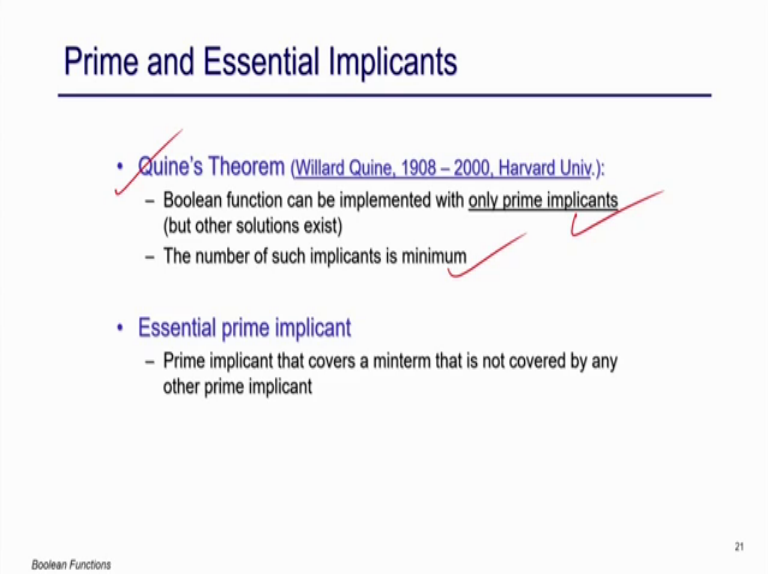
And that is what is the objective of this two level minimization that you try to find out the minimum cover, which is the least number of cube right. So, that is the number cube, which is covering all the number of product term right, fa that is what is called ca cover minimization and what is why this is important you understand? Suppose, I have this 4 value right, whatever I just told you. So, this 4 value is nothing, but if you just write my a f is this right.

So, this is basically X 1 bar X 2 bar X 3 bar. This is basically X 1 bar X 1 bar this is X 2, because that is 1 X 2 1 and X 3 bar right X 3 bar and this is, this is basically X 1 bar X 2 bar X 3 bar. This is X 3 sorry and this is basically X 1 bar X 2 and X 3 right. This is your X 3. So, this 4 impli, I mean this 4 minterms are this, this function. Now, if you find a prime implicant, which is covering all of them right.

So, whatever I found it then, this is just my f is become X 1 bar right, that is what I got it, this is my X 1 bar. So, you understand this prime implicant. Now, instead of. So, you can, if you have a circuit, which is covering. So, I have and gate for this I have and gate for this, I have and gate for this and I have and gate for this and there is a or gate. So, there are 4 and gate plus or gate, but this represent by just X 1 bar, which is nothing, but this single line right.

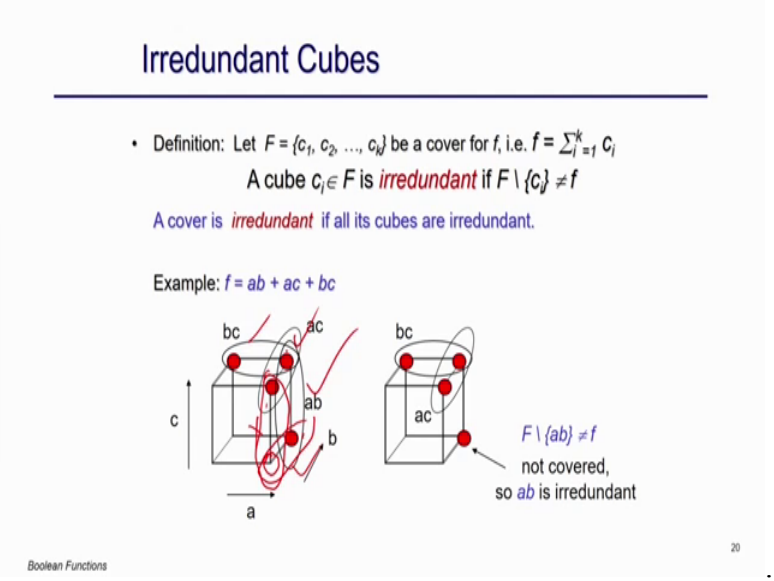
So that is what the advantage right. So, if we just try to try to find out the prime implicant that is the kind of mini maximum possible size which is required minimum number of gate to implement right and then we try to have cover all the minterm using those prime implicant and that will actually have the has a kind of minimum possible way of representing that particular function or in the other word this is the optimal representation of that particular function in sum of product form right that is how we you should do here right. So, that is what the objective of this two level implementation right. So, this is what is talked about, here that this prime implicant.

(Refer Slide Time: 29:21)




So, this is given by this (Refer Time: 29:23) theorem that Boolean function can be implemented only using the prime implicant right, that is what I just talked about and the number of such implicant is minimum. So, you can always have a minimum number of prime implicant that can be represented in Boolean function right, that is what is going to, we are going to it.

(Refer Slide Time: 29:43)



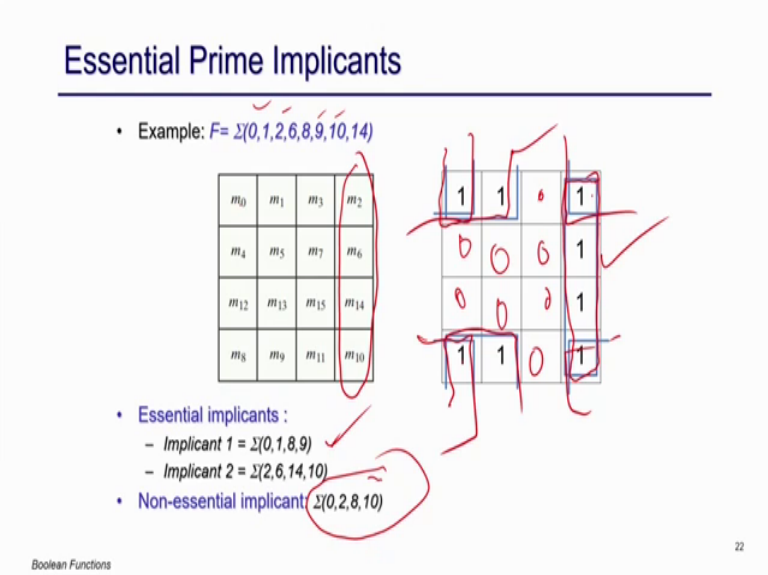

So, what is irredundant? Irredundant cube is something, which basically, if I remove this particular cube, some of the minterm will be uncovered. For example, here I have say 3

minterm, where this is b c, this is a c and this is a b and this 4 are the on terms and if I just remove this particular a b then what will happen? Then what will happen? This, this leaven uncovered right, because, now this minterm is not covered, this particular a b is a irredundant cover, because I cannot remove this particular cube, because if I remove it, some of the minterm will be uncovered.

So, in that sense, there may be some, some cube, which is can be irredundant. For example, suppose, you have say if I just take another example; suppose, this is my. So, this is my one of the cover as I given here and you can have also another. So, this is say sub, say this is also. So, if you just think about this. So, there is another one node here and this is another cover and this is another cover right. Now, you can see. So, this is kind of irredundant cover, because this is not required right. Even if you do not have this particular cover, I am actually covering all the nodes. So, this is covering this 2, this 2 is covering by this 2, this 2 is covering by this 2 and this node, the minterm is covering this 2.

If I just add this, you just ir, irredundant one. So, I can remove this one. So, this is called redundant cover, but other three are irredundant, because they are necessary, if you just remove them your uh, all the minterms will not be covered right. This is what is all about right.

(Refer Slide Time: 31:20)

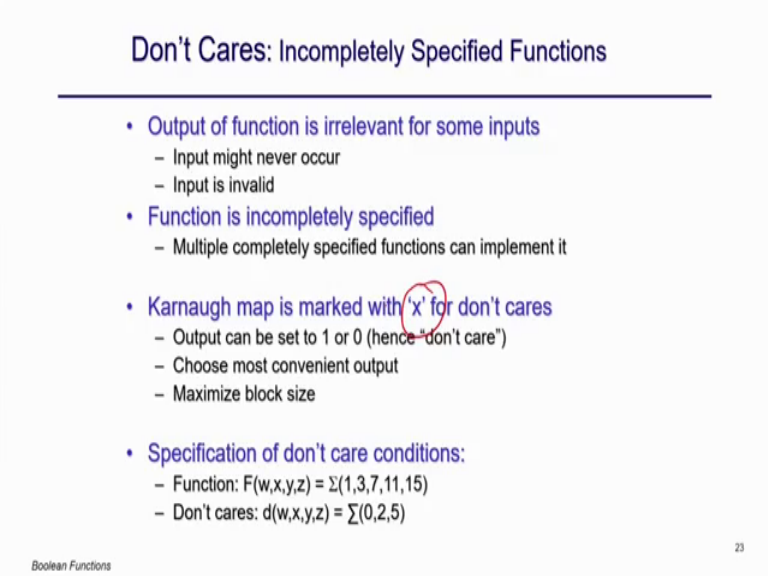


So, what is essential prime implicant, I already mention that this is not cover by any other prime implicant. So, if you just think about this two level logic optimization. We can have different different kind of methodology, one is kind of this karnaugh map right. We represent the variables. So, for example, here the onset consist only this, this are the onset right. So, the onset, the other are 0s right, they do not occur here.

Now, our objective is to do this prime implicant of finding. So, in hyper graph, we have done that in karnaugh map. What we just do? We try to find out the prime implicant is the same, we try to find out the bigger big, such cube, which represent this right. So, this is one and this is one right. This, I what is the implicant right. So, I have if you just think about this implicant, I have this implicant 1, which is this and, and this implicant 2 is covering 2 6 4 18. This is 2 6 4, 10. So, this is my implicant 2 and this is my implicant 1 and if you just think about another implicant.

For example say; this I just take these four corner 1. So, this will become another implicant right and this, this is covering this 0. This is 0 2 10 and 8. So, this is kind of no, non essential implicant, but these two are essential implicant right. So, this is in Karnaugh map also. We try to find out, this implicant effectively, this prime implicant, which is basically covering all the, on terms. This is how, this happens.

(Refer Slide Time: 32:49)

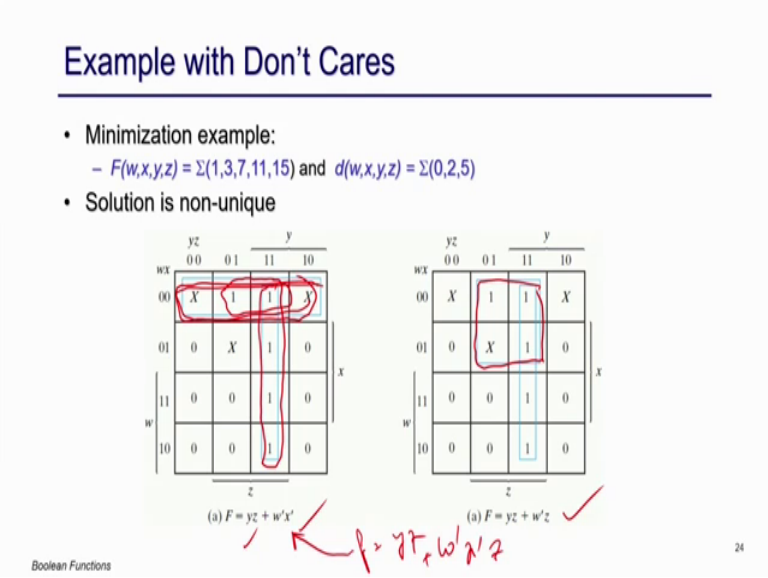



So, we do not talk about do not care yet. So, do not care is something, which we, that particular input combination never occurs right. We assume that, that particular scenario

will never occur of input right. So; that means, that those input is invalid. So, that is why, they are called do not care. So, what we can do? Since, we know that particular, particular this input combination will never occur, we may take those advantage, we take those particular do not care term, which say them they do not care, because they do not care about this input and we can use those do not care to optimize further your logics circuit right, because the assumption is that, that particular combination never happen.

So, even if you take them, there is no I mean, I mean optimize our circuit, using those do not cares. It does not matter right HM, because. So, this is how we just use the do not do not care is represented by X right.

(Refer Slide Time: 33:42)



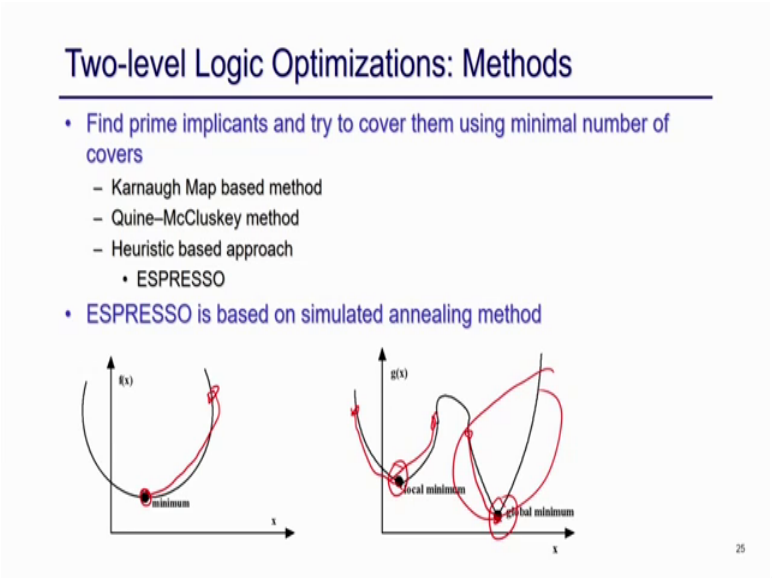

For example here, I just give an example. So, this is my onset 1 and this are the onset and if I just represent this function and this say, this are the, x are the do not care right. So, what we can do? So, this is my one prime implicant, that is covering 4 and if I do not compare this do not care, I have only this, this is another cover right, if I just represent this, this is basically y z, if I just do not take this. So, my F will be y z and this is basically w bar x bar and z right.

So, this is my representation of this function, but if I take this to do not care, I can reduce it further right. So, my representation will be now, this y z and this is w bar x bar. So, I have reduce it further, because one term one term literal is removed from here right. So, this is how we try to utilize the do not care. So, the advantage of this do not care is that,

we may took the stunts to go, find a bigger cover or we may not take this, but onset is the mandatory thing, we have to find a cover, which will cover all the onset right.

Do not care based on the recommend, we may take it or may not take it and we, because of do not care, we might have a different solution also right. For example, if I consider, because I have to cover this 2 1 right. So, I can cover this way or I can cover this way also right, because that is also same then I have a different solution, this total different, but we can have both the solution or kind of optimum right. So, that is how we actually use the do not care in our Boolean function right.

(Refer Slide Time: 35:12)




Now, I will move on to thus discussion of this two level logic optimization. So, what we have understand that is basically two level logic optimization is nothing, but try to find the prime implicant of an Boolean function and then we try to cover those prime implicant. I mean basically, try to cover all the minterms using minimum number of such covers right. So, basically this prime implicant is covering some of them. We try to find out the minimum number of such prime implicant, which cover all the minterms right and we have lot of I mean various method, what is very well known is Karnaugh map based method. We have this Quine Mccluskey based method.

This Karnaugh map is kind of mapping a table and we try to expand, try to find out the prime implicant on the table right and on the, this this, Quine Mccluskey is also kind of a tabular method each (Refer Time: 36:02). We try to find out the implicant. We try to find

out the common part of two product term and we try to find the implicant and move on right. So, we know those method I am not going to detail discussing on those, what I am going to talk about? This, in this lecture is something that Heuristic base approach right, what is called expression?

Because the problem is this Karnaugh map based method, Quine Mccluskey, they are not scalable there. If the number of unit is very small say, 2 4 10 till 10 that is fine, because if you just think about the table size, where you have 10 input is to divide by n number of entries right and now, handling that is very difficult and it is very difficult to visualize as well. So, on the other hand in ESPRESSO, I mean, Heuristic based approach, it does not always give you the exacts solution which is the fact for this Karnaugh map based method or Quine Mccluskey based method, where we always get the, whatever the solution we get that is the optimum one, but ESPRESSO will always give you near optimal solution, but they are very first, they are scalable and that is also any time.

In any time, if you stop, we have a solution right that may, may be, may not be optimum one, but we have a solution to that problem. So, that is why this ESPRESSO is kind of the most use widely. Used I mean algorithm for this two level logic minimization. So, instead of going into this Karnaugh map based or Quine Mccluskey method, we just try to discuss espresso in this, in this discussion ok.

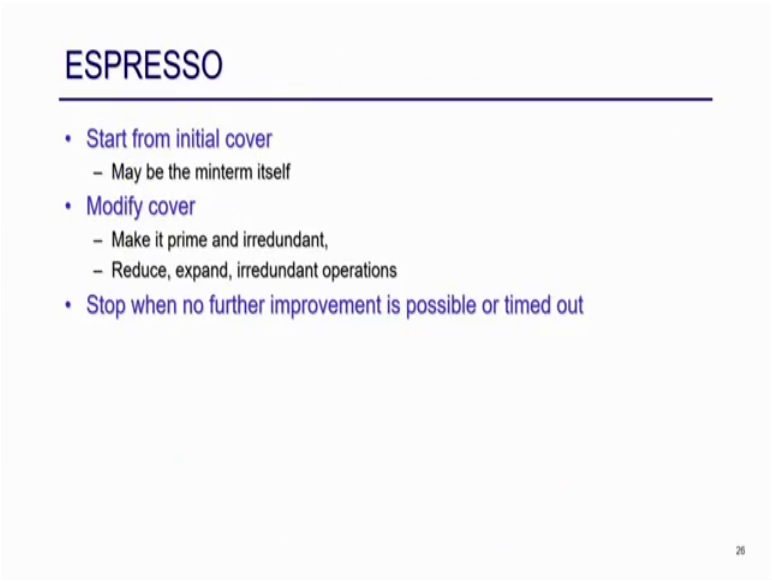
So, ESPRESSO is based on stimulated handling base method. What is stimulated handling based method? It is basically instead of finding local minimum, we try to find out the global minimum here. So, for example, if you just look into this, if we just take start from some point right and you have some optimization function, we will move on and we will find out this minimum right, because this is a convex. We cannot go out of it right, because this is, will stop here, but if we just start, but may be this is the local minimum.

Actual global minimum is somewhere else, but if I start from some node, I will always stop here in this point, here I will stop here, but if I start from some other node, I might go this right. So, simulated annealing method is actually doing this. So, it is basically, will start from some point. We try to find out some local minimum, then we arbitrarily move on to some other starting point and then try to get the optimization function right

and this way there is a high chance that we always, I mean always going into that global optimum one right.

So, we may not go into it, because my, my choice of input is not really random, if you do not find any point in this part then I will not able to get this minimum, but if you just take really random I mean starting points, we might most of the time get this global minimum right. That is what is the simulated annealing method all about.

(Refer Slide Time: 38:41)




So, in ESPRESSO what we start? We start from some initial cover, because you, the objective is again, I mean same right. We have set of minterms, we try to find out the cover, that will cover. I mean basically, the prime implicant, which will cover all the, all the minterms right. So, we will take any starting point. So, may be all the minterms right.

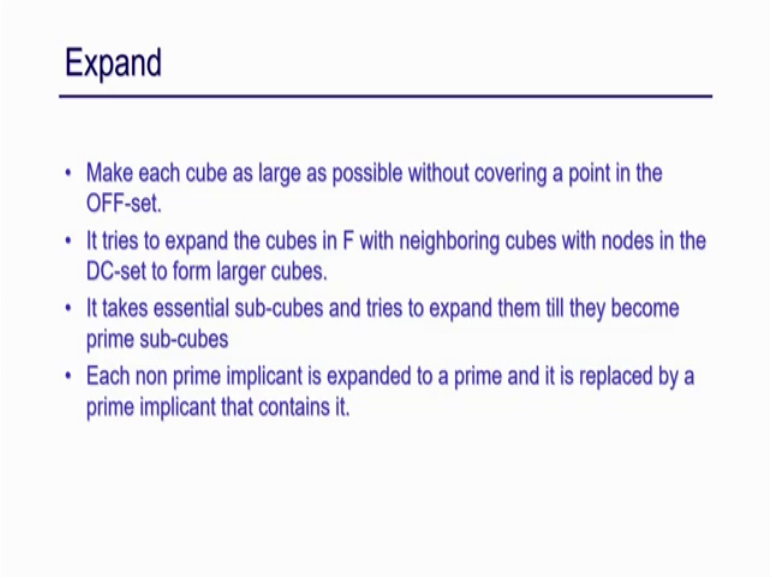
So in the initial starting point, may be all the minterms itself is the cover right, because we just consider all the onset that is the cover of that. Then we try to update that cover in each iteration right. So, what we do? We have some operation called reduce expand and irredundant. We will just apply those operation in a iterative way and we try to update this cover right and then we try to get some cover right. So, up when we do not have any other option, we do not have any improvement in this, then what we will do?.

We will do some arbitrary change in this cover and start try to find out some another initial cover from where again, we will do the same thing right, that is the simulated

annealing per right per tub per. So, what I am going to do is, I am starting for the basic ideas that I am taking from the initial solution. We apply certain kind of operations iteratively, unless I reach a saturation point, saturation point in the sense, I do not, I, I cannot have a further improvement over the previous solutions, then I will stop there.

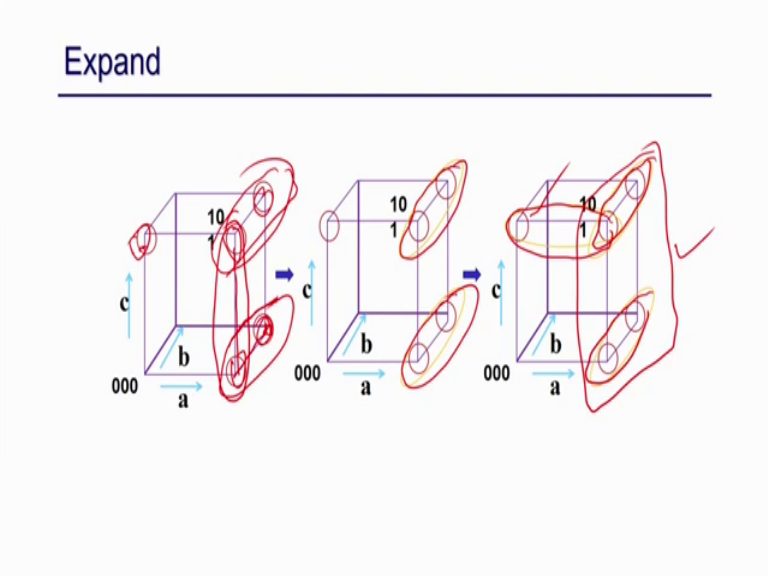
Because that is the best solution for now, and then I will randomly choose another starting point and then I will keep doing this either with timed out or I am happy with my solution right, that is how we, this whole thing goes on right.

(Refer Slide Time: 40:16)




So, the, this primary operation is operation is expand. So, expand is as mentioned that we have some cube, which is covering all the minterms and we try to expand this cubes, sub cubes as large as possible without covering a point in the offset, because offset is those term, which should not cover right so; that means, it taking advantage of the other sub cubes; that means, the other onset or some do not care set right, it try to starting from some starting some initial sub cubes. It try to expand it as much as possible, just to cover in the neighboring cubes or minterms or some do not care set right and this is how the expand happen right.

(Refer Slide Time: 40:59)

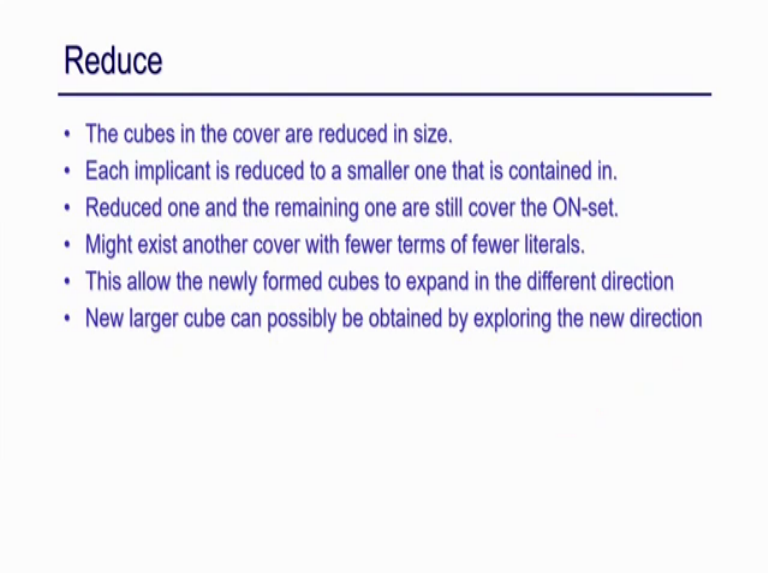



There is an example here. Suppose, initially my cover is all this 5 right. This 5 are the onset starting from this. So, my initial cover is 5 nods right and then I try to expand it, by expand I just try to cover this 2 right. Now, I have only this one cover and then I try to expand from this node, I cover this way, I will get right. I got this two right. Similarly, I starting from this node, I try to expand. This way I got this.

So, I have 3 expanded cover. Now, again I can expand further and I get this is as a cover right. So, now, I have only 2 cover, this is a prime implicant. This is another cover, which is covering all the minterms. You can understand here the directions may be different right. So, from here I may go this way to cover this or may I go my go this way and I can take this kind of cover, just to cover this 2 right. So, direction can be different and the choice is not unique right.

So the, that is the idea. So, if you just start from here, if you just expand one direction, you will get some solution that may not be the best solution right. So, then what you should do you try to expand it in other direction as well. So, for that we have another option is called reduce.

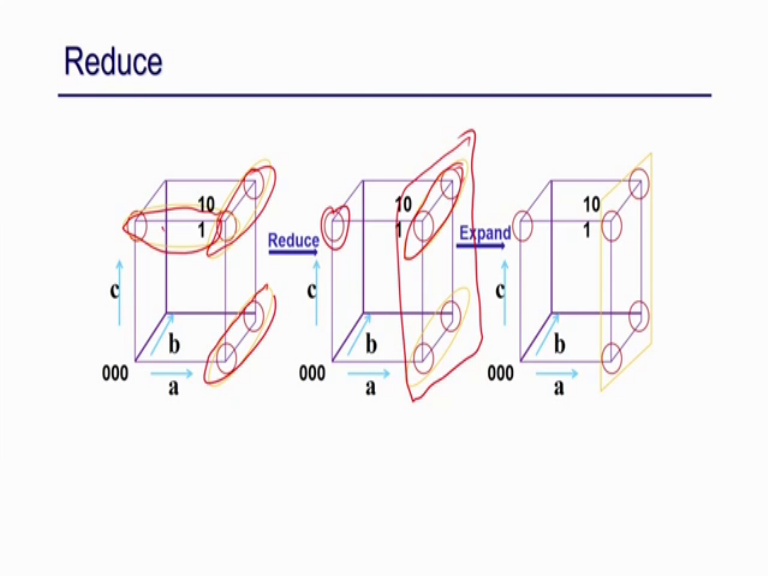
(Refer Slide Time: 42:05)




So, what is that, we have some set of covers? So, set of cubes that is covering all the minterms, its just the reverse of the expand operation right. So, instead of expanding now I am reducing to the smaller sub cubes. The advantage here is that now if you have smaller cube, if I just again apply this expand now go in the other direction whatever just I explained here right that.

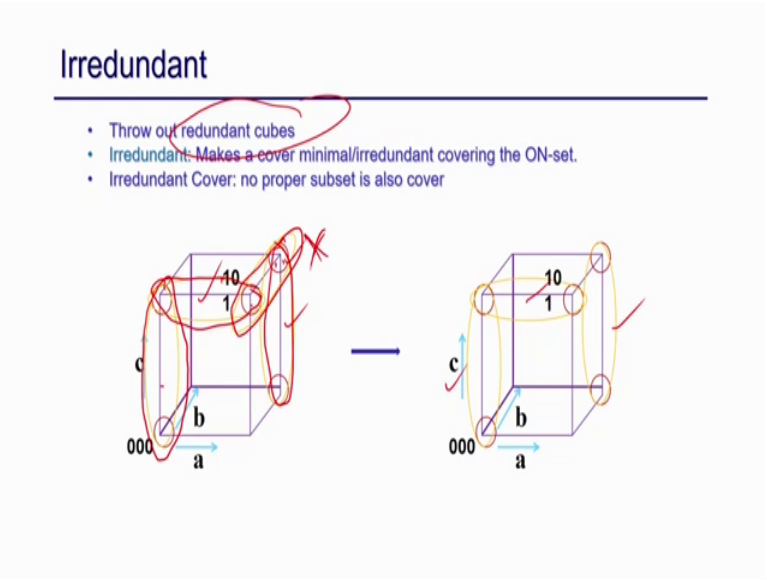
From here I did expanded this way and I stop here. May be this is not the ideal example, but in some scenario may be this is not give you the good solution, but if you expand this way you will probably get a better solution right. So, that is something that reduce talks about. So, basically reduce what is that, it start from some cover, it reduce the size, so that there may be this allow the newly formed cubes by expand in the other direction right. So, that will give you the advantage.

(Refer Slide Time: 42:58)




So, it is example, so initially suppose I have this three cube which is covering all the minterms then I reduce it. So, I reduce it I just got this one right and this. Then I reduce this, then I can apply the expand again, so that I am covering this two right. So, this is just to say that you can actually have a reduction followed by expansion, which can give some different cover which is a better cover of the minterms right; that is what is called reduce.

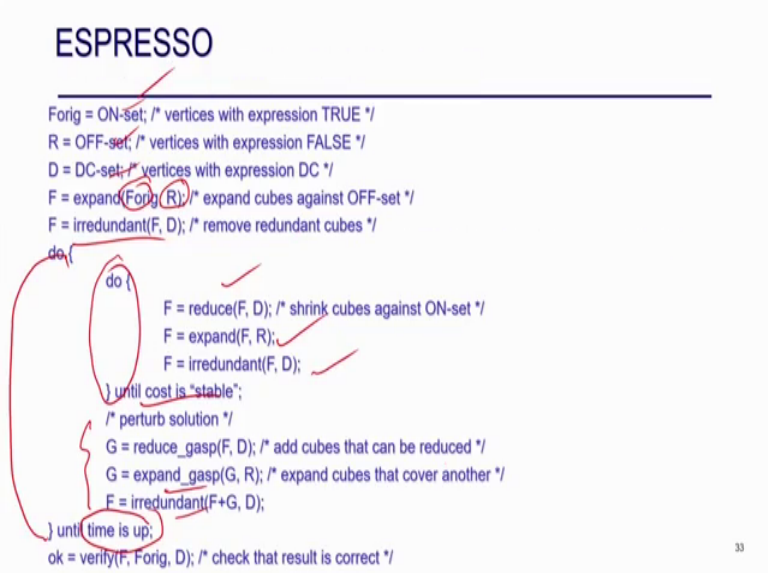
(Refer Slide Time: 43:24)



We have another operation is called irredundant. As I mentioned earlier that there are some imp implicant, prime implicant is essential; that means, that is covering some ca cube or say some minterm which is not covered by somebody else right. So, there may be some implicant or say prime implicant which is covering some, some node which is already covered by somebody else, so that is the kind of irredundant, redundant one. So, its basically throughout the redundant cubes.

for example, here I have this say 4 cube covering all this 5 minterms. You can see here this, this is already covering this two right. This is also covering these two right, this is covering this two. So, in that sense this particular cover is redundant right, this is not redundant, because this is covering this two minterm which is already covered by somebody else, this is covered by this, this is covered by this. So, I should reduce remove this. So, this is what is the irredundant term right. So, from this four cover I will. So, there are 1 2 3 4, I end of this three right, this 3 which is covering all the minterms. So, this is what is called irredundant.

(Refer Slide Time: 44:39)




So, what it does this espresso, its basically taking advantage of this three particular operations iteratively right. So, it start from this ON-set. ON-set is that minterm; that is it I mean that actually whenever there is one, I mean that they are one, then the function will be one. OFF-set mean they are always zero, when the function is one they are zero

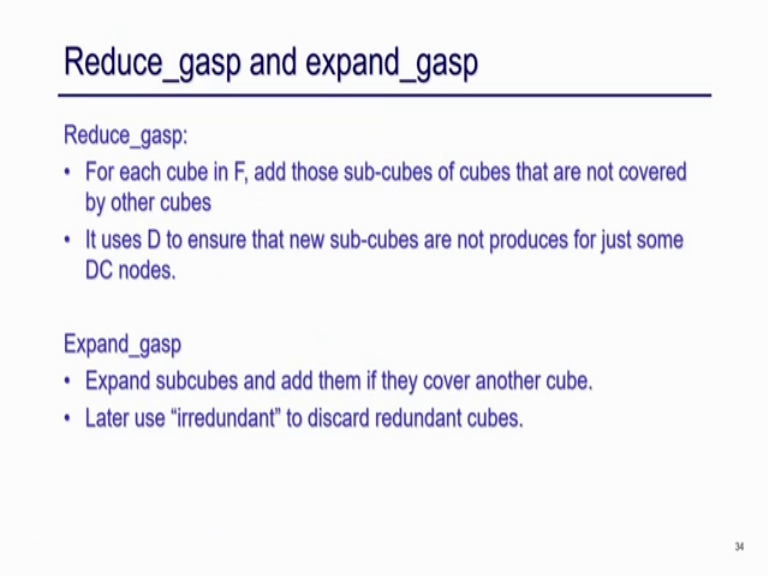
and do not care I can mention, they may not they will never offer, but I can take advantage of this do not cares right.

So then what it does? It try to expand, because initially it cover is all the ON-set as I mentioned, all the minterms are my cover, initial cover right, then I try to expand it, expanding through this ON-set and the do not care set as much as possible so, that I will get a bigger cover or kind of prime implicant right. Then I just do this irredundant to throughout the redundant cubes right, then I get some kind of, I mean essential cube which is covering or my minterms right. Then within the loop what I am doing in this loop I am going to do this, this operation reduce, I will just try to reduce it, just to go in the other direction, then I will expand again, then I will throughout the redundant, this I am going to do it iteratively unless my cost is stable.

Stable in the sense between two successive iteration, I do not have any further improvement. So, I if I found any scenario and there are between two successive iteration of this loop, I do not have further improvement; that means, I got a saturation point then I will stop. So, whatever the solution I got is kind of best solution for the direction I have expanded right. So, the direction I have covering that things right. Then this is this come the simulation annealing part. Now I am going to do this reduce gaps which is basically try to find out some other starting point, I will I will talk about that and then I will effectively expand and then irredundant.

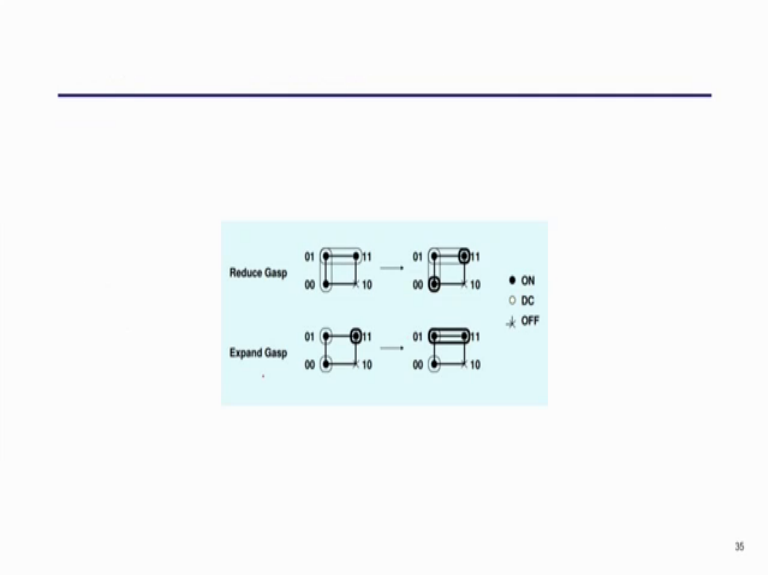
So, I will try to create some another cover which is completely orthogonal to the current cover. Then again I will go back to this and I will do the same way right, again I am going to expand, reduce expand redundant, so I am going to do this things right. I am going to stop it when I my time is up, or I am satisfied some result right; that is what is talk about.

(Refer Slide Time: 46:44)




So, what is this reduce gasp, its talking about that for each cube in F add those sub cubes, sub cubes that are not covered by somebody else right some other cubes.

(Refer Slide Time: 46:54)




For example here, I have this two cube right. So, now, I will find this minterm which is not covered by anybody, only this and for this, this no minterm is covered by only this nobody else right. So I am going to add this two minterms as the cover right. I am going to out this as the as the sub cubes in the in my cover right. Instead of putting this two I

am now putting this two right, because from there I can actually do this expansion again right. So, I just try to expand from here I will get this right and so on.

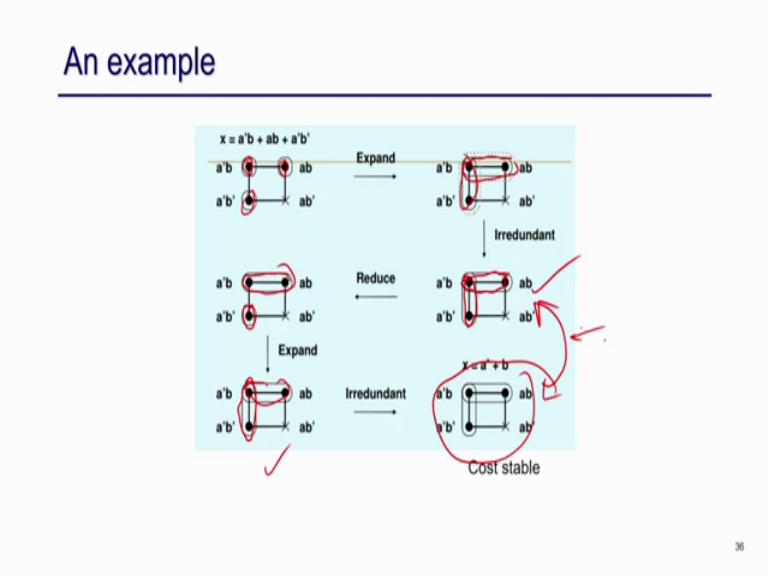
So, this is what is talk, it uses D to ensure that new sub cubes are not produced just some for some just D C nodes. So, basically it will start from any other minterms and try to grow it in other way right. So, this is what is that reduce gasp is all about, and then you just add sub cubes and add them, I mean if they cover other cube right, this is just normal expand operation and then I can use this irredundant to remove the things. So, this is all about this espresso.

Its specifically (Refer Time: 48:01) we try to explore all directions, I mean all directions, I mean one direction it get a best solution then we try to reduce it, try to explore the reduct other directions and that is this way this particular loop will the inner loop will give you after some iteration that the best solution in this from this initial starting point, then we just part of the solution; that means, we just try to, try to find out some other initial cover which will cover all the minterms and from there again I am try to do this expand reduce those steps right.

So, that I can actually go for some other global minimum right; that is what is the espresso all about. And I mention here that your Sa Boolean function is represent by hyper graph. Here I have shown this an hyper graph where the three variable. Now the question is how we represent a hyper graph with four variable right. So, that is nothing, but we take two such hyper graph of three and then I connect the nodes which are have a difference of one bit right, then I just connect them accordingly right. So, then it will give a 4 I mean a hyper graph of size 4.

Similarly, if I take two hyper graph of size 4, I can connect the node which I have only one bit difference, then it will give a hyper graph of 5 and so on. So, this way I can represent my fa function in the hyper graph and this reduce expansion and then irredundant and this expand will occur as I discuss here. So, this is how this espresso works right. So this already discussed.

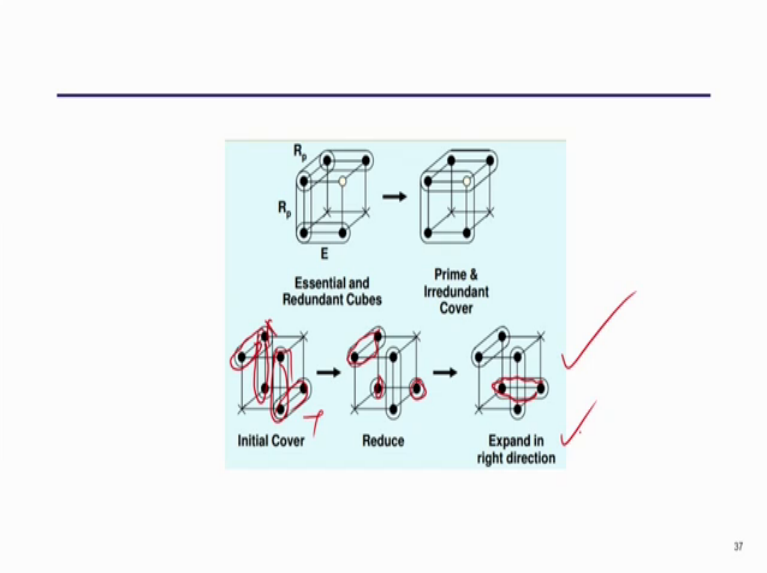
(Refer Slide Time: 49:35)




So, where is this saturation point I have an example, suppose I have initially have this three minterms, I expand this way, so I got this is a minterm, this is a minterm. After that I do the irredundant, there is no irredundant here though, all are important essential prime implicants. So, this is my correct one, then I reduce it again right so I reduce. So, I say this will become this, I remove one of them this is a, again I am expanding, so I am getting this.

So now you see this and this is the same solution right, if I just do this irredundant I got this which is same as this. So, up between two successive iteration I found there is no improvement right. So, then I will stop here. This is how the inner loop works right, this inner loop and then I will probably start from some other starting point as I mention. I probably start from this starting point and I will move again, so that is that second part right.

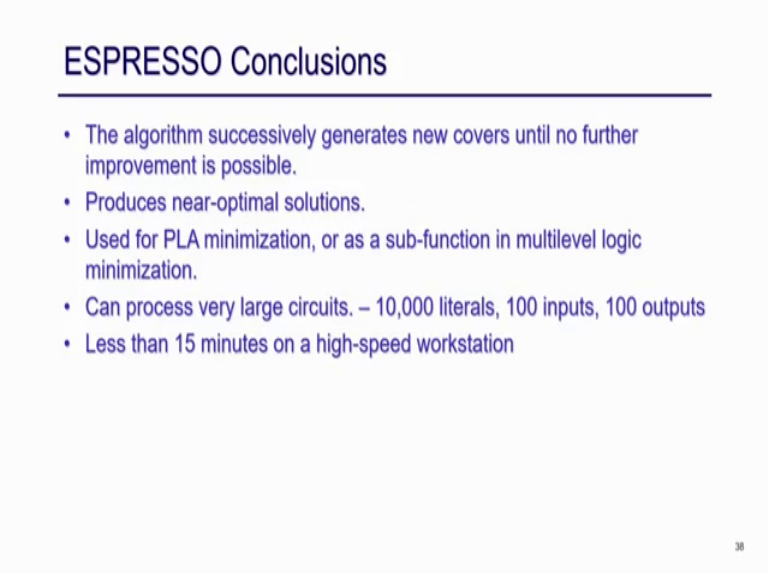
(Refer Slide Time: 50:23)




How this will give you improvement here, there is an example. Suppose my one solution give you this right, so this is my initial cover right, this four, and then just I reduce it, so I just remove some of them, so end up in I have this, I just reduce this one and this one. So, my deduction is this now right, this I covered only this two and this two nodes. Then again I expand this way, I got a expand in right direction, because just to cover this four terms I this is the optimal irredundant set of covers right, so I have three now.

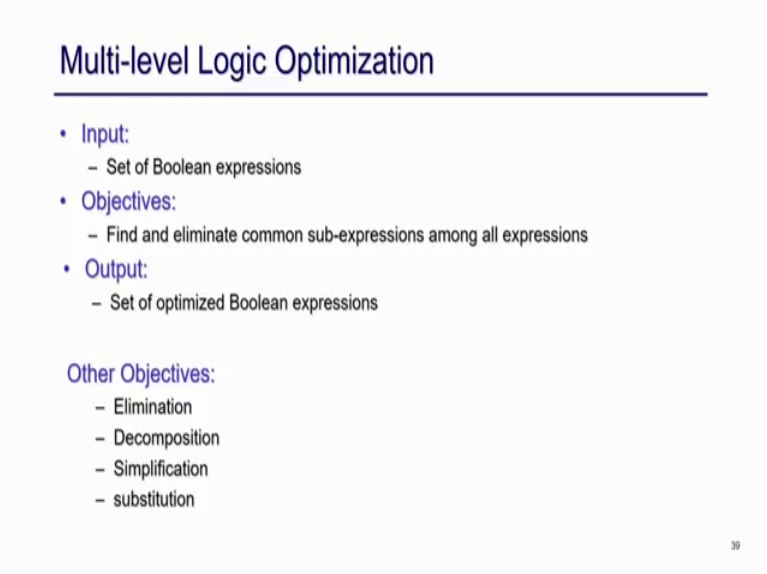
So, initially I have 4 4 cube which is covering all the minterms. Now I have this three which is the optimal one, so this is how this will work so if you have units in cover which is not optimal, but if you reduce and expand you, I mean some iteration you might expand in right direction and that will give you the optimal solution.

(Refer Slide Time: 51:20)




Just to conclude on espresso is basically is successfully generates new covers until no further improvement is possible, and it is kind of give you near optimal solution right. It always try to give you near optimal solution and this is used for primarily for P L A or minimization, and map the biggest advantage of this ESPRESSO is that very much scalable and it works for very large circuit, which is contrary to the other conventional exact approach like Karnaugh map or say Quine Mccluskey, those are not kind of scalable right. So, this espresso actually works for bigger circuits and practical circuits ok.

(Refer Slide Time: 51:59)



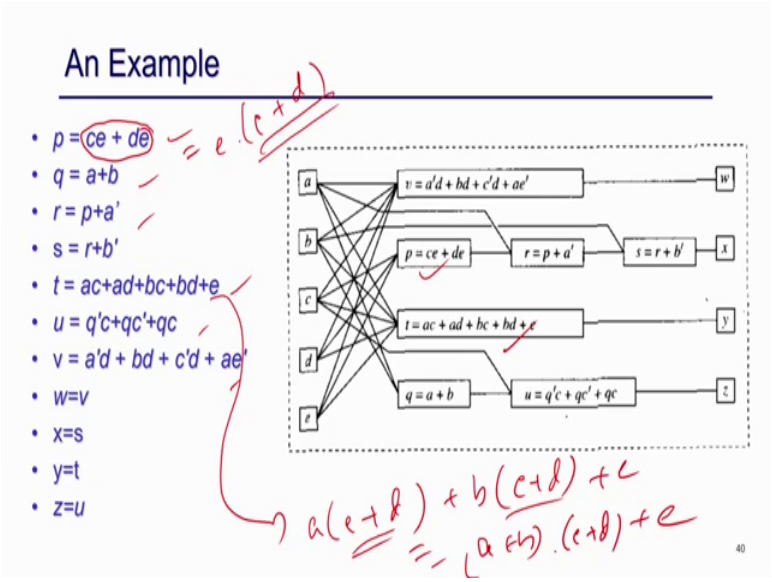
So, I will now move on to this multi level logic optimizations. As I mentioned this two level logic optimization is not sometime is correct way of representing your circuit, because sometime what happen that this is too big right, its not representable as I mentioned already. So, in prime in practical we, our Boolean circuits are represented in a multi level implementation, we have multi level implementation of the logic circuits. So, now, how we optimize those multi level circuit right; that is what is we are going to discuss now.

So, multi level logic optimization in input is set of Boolean expression, you understand you have set of Boolean expressions and up to up to output will be a set of optimized Boolean expressions right. And our objective is try to here the, objective is try to do some network of transformation right, some transformation on the circuit which will result in some optimized Boolean expression right. And of the most common approach is that eliminate common sub expression,

Because the common sub expression is something in sub part of the circuit which is reuse in both two component of the function right. If you find out those common sub expression and a reuse it, then it will reduce the overall size right. We can have other objectives optimization; like elimination, decomposition, simplification and substitution, we can always have some rules for that, we can always try to find out some, I mean some way to just reduce this simply some circuit some, with substitute, some complex part way simpler one and so on.

So, but the primarily we, when we talk about this multi level logic optimization we primarily talk about identifying the common sub expression between two expressions and then eliminate them right, that will give you a big boast. I mean optimize your circuit by large right. For example here you take this example, so I have lot of variables right.

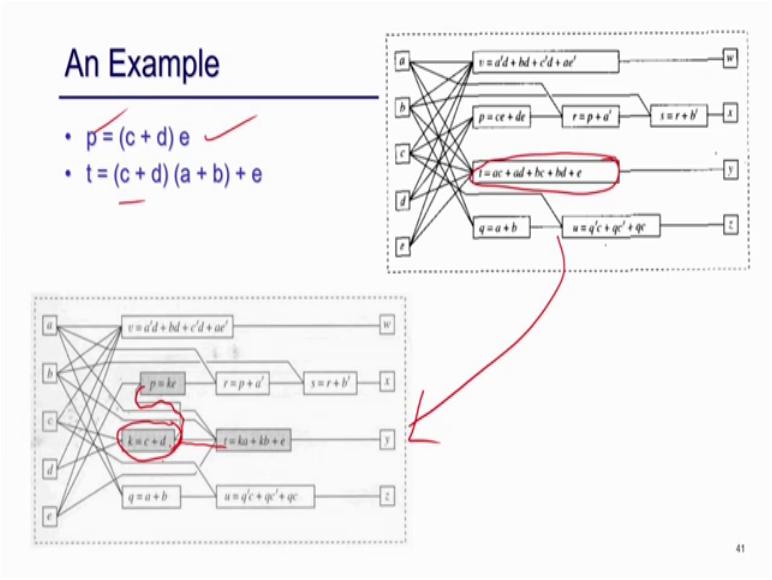
(Refer Slide Time: 53:39)




So, these are various variable which is updating here, and this is a kind of network representation for that right, so we have this inputs this is the kind of dependence we have right. This is actually representing this circuit. Now if you look into this that common sub expression; for example, this p right. So, here if it is p I have this is I can represent e into c plus d right and this t if I represent here, I have this you can see here a into c plus d, and then b into c plus d right plus e.

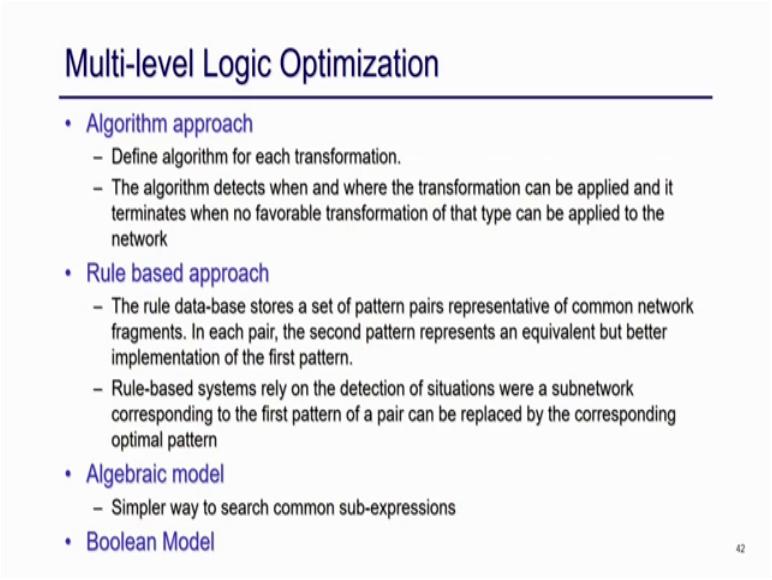
So, I can represent this equivalent this a plus b into c plus d plus e right. This is how we represent this t. So, you can see here this is the common sub expression right. So, this common sub expression I can compute in my, so I am computing this again here and here, so this is actually giving some extra I mean unnecessarily the double computation of the same thing right.

(Refer Slide Time: 54:44)




So what I can do here, from this circuit I can make it this, so that I can compute this c plus d as I mention here once. And then I can use this c plus to compute this p, as I shown here, so same thing is written here. I am using this to compute this p, I am using this to compute t as well right, so this is what I have done here. So, now, you can understand that this complexity of this node versus this node is simpler right, so this is how I actually can optimize your circuit. So, in multi level optimization, we primarily try to find out this kind of common sub expression between multiple expressions and we try to remove them, I am going to use one expression I am going to reuse it right; that is the kind of optimization we try to do it here.

(Refer Slide Time: 55:25)




So, the question is here is that how you can do that right, how you can do this kind of things right some. And I mean in generic you can have other transformation as well. So, there are two kind of approach; one is kind of say algorithmic approach, algorithmic approach what it does it try to find some algorithm, has a algorithmic for each transformation right; one for common sub expression, one for say simplification, one for decomposition and all.

And then we try to find out those detect, those transformations can be applied where it can be applied, and it keep doing that right and when there is no way it will stop, when there is no way to apply those transformation right. And in the rule based approach is basically we have set of rules, so this is the input pattern and corresponding this is is the optimize pattern, and.

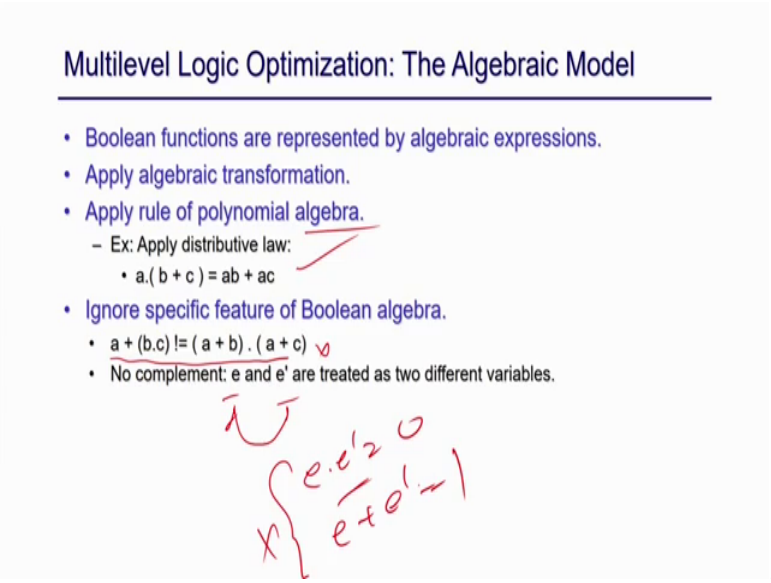
We try to find out some pattern matching thing. We try to find out those patterns in your design and it will replace by the optimize version right. So, this is how we can do this do this rule best approach, and we can have this pattern, this optimize pattern for each each kind of transformation as well right. This is the way how usually do, this algorithm approach, and the rule based approach.

And on the other hand we usually think about our circuit represent as a Boolean circuit, as I mention for espresso also right that everything is Boolean right. I have a variable

complement of a variable, and we do all this rules De Morgans Law, all the rule of Boolean arithmetic we applied right.

This another way of doing it is like algebraic model. We might assume this Boolean formula is algebraic algebraic formula right. So, what I am doing here instead of representing this Boolean function by algebraic expression, all the variable become algebraic variable. Now I can apply the rules of algebra right, algebraic transformation you can apply on that to do this, and that is what is called algebraic model, and that is is most I mean one convenient way to do this multi level logic optimizations right. I am going to talk about that right, just to give an elaborate this for example, in this algebraic model.

(Refer Slide Time: 57:23)

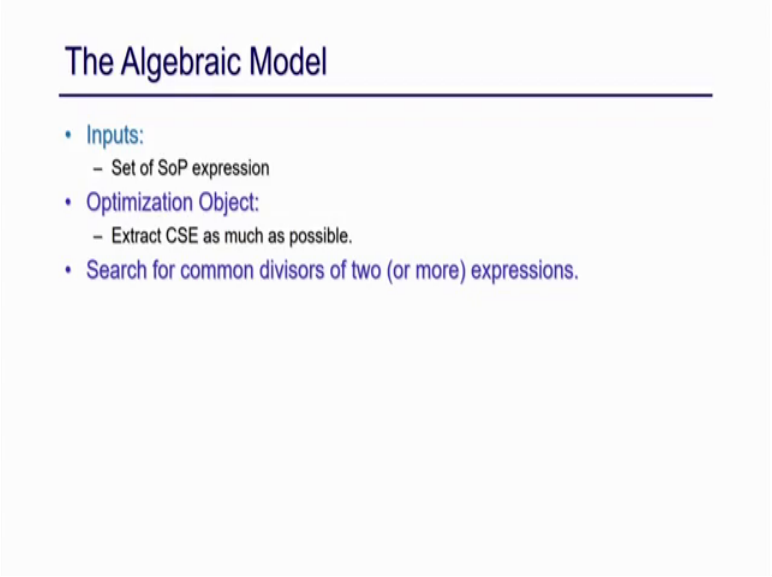



I apply the distributive law of this algebra right, but I cannot use this De Morgans Law, this is what is equal in Boolean, but this is not equal in De Morgan, I mean in for algebra, when this a b c or all sub, all are basically algebraic variable not the Boolean variable, then this is not true.

For example this e and e dash which is basically complement right, and you can have some property right e into e dash is always zero e plus e dash is always one. Now those are not applicable for algebraic model right, because here this e and e dash at two different variable, they are not consider to be a complement to each other. So, these are the kind of restriction are there, but if we just assume my all this expression are some

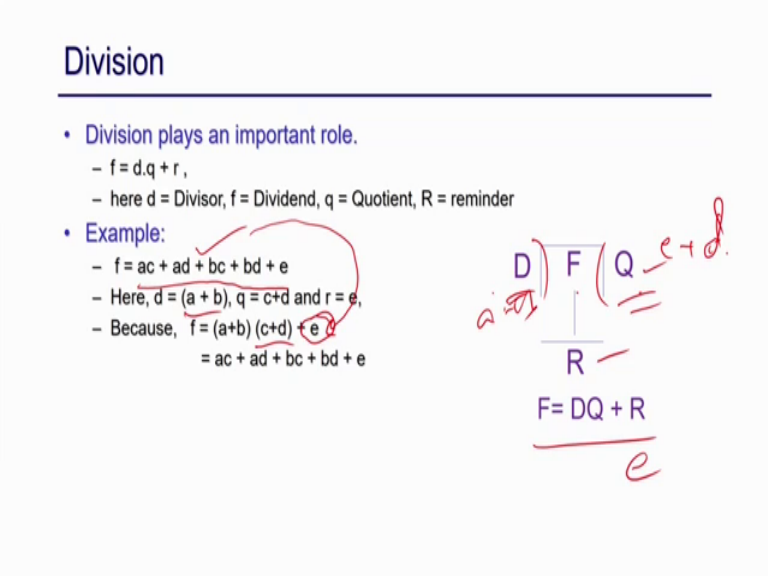
algebraic expression, then we can actually use the optimization or the properties of the algebraic expressions and we can actually do some simplification.

(Refer Slide Time: 58:19)




So, one of the so basically idea is that we have set of some set of this some product expressions, and then we want to find sub common expressions as possible and then this we try to find out this common this common sub expression between this expressions right, using algebraic model. And now the question is where to search this common sub expression right. This is in using the division algorithm right. So, division in the common division of two expression just to illustrate this SoP.

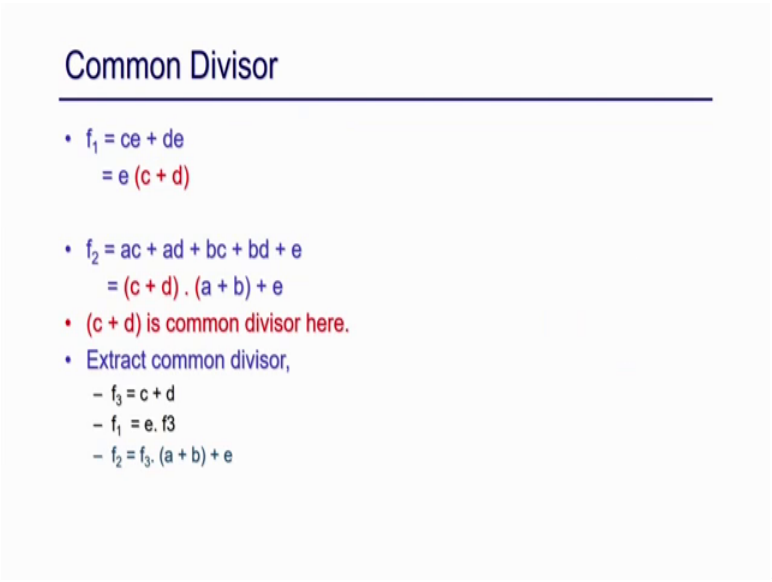
(Refer Slide Time: 58:49)




Suppose I have this division, this is basically suppose I have dividing this F which is basically my dividend this is my divisor and this is the result I mean quotient and this is my remainder, so my F is basically D into Q into R right. Now, if just think about my f is this right a b plus a d plus b c d b d plus e, I can represent this d say is my a plus b right, and then I can represent this equivalent to this right.

So, now you can see here this my quotient will be c plus d right, this is my c plus d, this is a plus b and the remainder will be this e right. So, now, I can I just apply this division operation on this Boolean expression right, my now this is my quotient, this is my divisor and this is my remainder right. Now the question here is that.

(Refer Slide Time: 59:43)



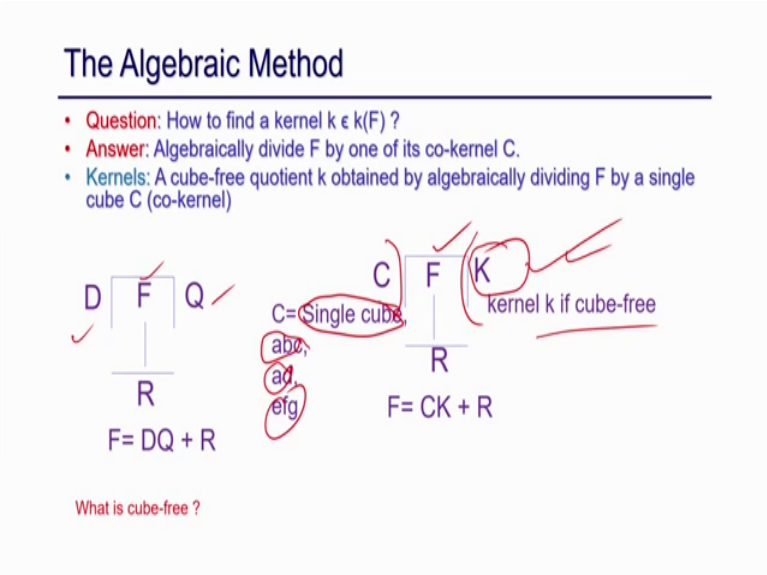

How to do apply this division algorithm just to find out the common sub expression right so, the question is that the common divisor, so if this two particular expression has some common divisor, then that is the common sub expression right. So, for example, here suppose my f one is this c plus d which in nothing, but e into c plus d and f 2 is the same earlier one, so which is re present by this. So, you can see this c plus d is the common divisor right, this is this is the common divisor.

So, this is something is my common sub expression. If I can found such common divisor, then I can rewrite my expression like this f 3 is c plus d and I can rewrite f one is e into f 3, and f 2 is f 3 into a plus b in to e right, so this is what is all about. So, this is the same as whatever the example I have shown here right, so this is what the same thing is happening here.

So the idea is that I am going to use this divisor property, this division algorithm on this Boolean expression which I consider the algebraic expression now and we try to find out this common sub expression (Refer Time: 60:44) I mean common sub expression as the common divisor here; that is the overall approach here, I mean this is how I am going to do it. Just to go into more detail so the our objective is to look for common divisor between two expression right. Now question is that w here to look for this divisor right, because we have two formula now right, so we have just two formulas.

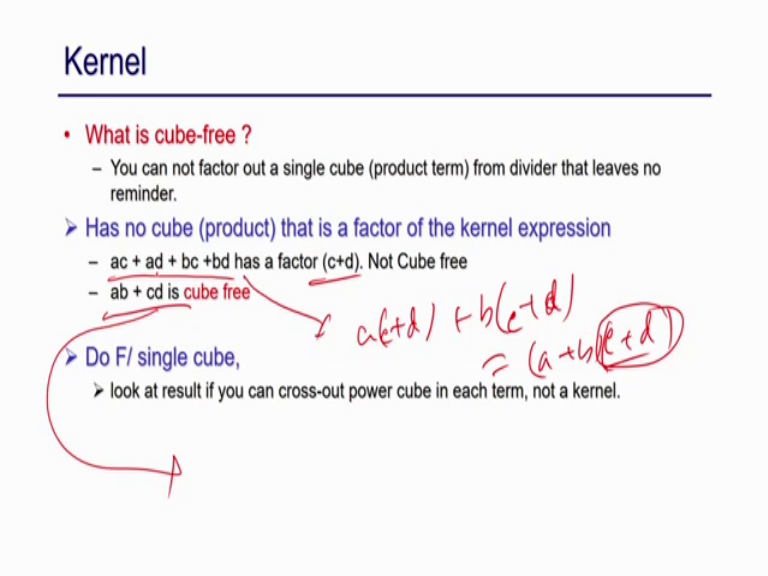
Now, where to look for this common divisor, where to look for this divisor for a function right the quest answer is the in the kernel of a function right. Now the question is what is kernel? So kernel is defined as a cube pre quotient of k obtain by objectively dividing a by a single cube right, which is called co kernel. Not clear, so we will just talk about this in detail. So, as I mention here that I can actually do this operation.

(Refer Slide Time: 61:35)




So, this is my divisor sorry this is my divisor, this is my yeah, this is divisor, this is dividend, this is quotient and this is remainder right. So, now my, if this c is a single cube right, so which is like a b c a d e f g its not some expression right, its a single cube. So,, if I divide this by this single cube and if I get some the result the quotient, if the quotient is cube free then this is a kernel, so this k is a kernel if this is a cube free. Now the question is what is cube free, that you should understand right.

(Refer Slide Time: 62:06)

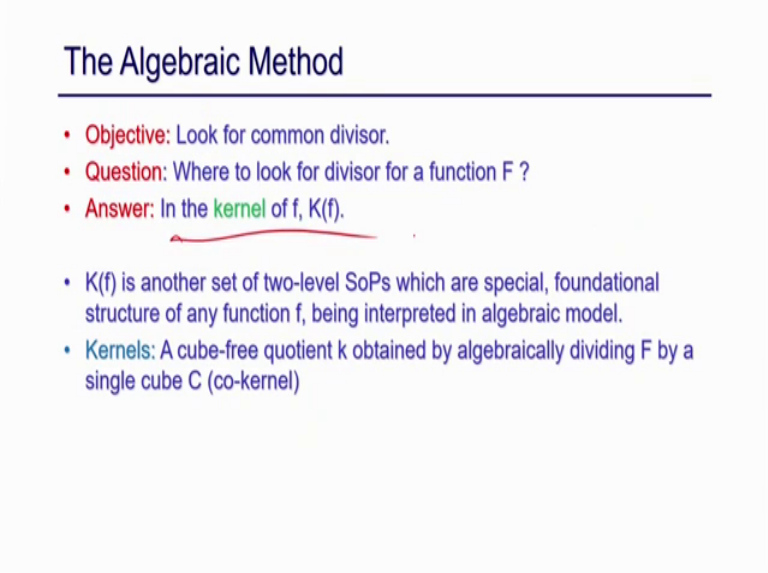



So, the idea is that something is cube free which does not have any co factor, you cannot factor it further right. For example, suppose we have this, this is not a cube this is not a cube free, because this I can write like this right, so I can write this as a into c plus d c plus d plus b into c plus d which is equivalent to a plus b into c plus d. So, I have a co factors c plus d of this right, this is not a cube free, but this is cube free, because I cannot represent, I cannot find a co factor of this, I cannot rewrite this expression in such form that I can found a such co factor then this is cube free right. So, if you come coming back into this.

So; that means, if you have a expression, this is my expression, F is my expression right, this is my expression, I divide this by some single cube, not by multiple cubes, cube means basically product of those literals right that you already discuss. So, a single cube means you have only one product term here. So, if I divide this particular function by some single cube by a product term, whatever the quotient I am going to get it, if that if that particular quotient does not have any other factor, we cannot factor it some sub expression from that, then that particular quotient is kernel right, so that is what is called kernel.

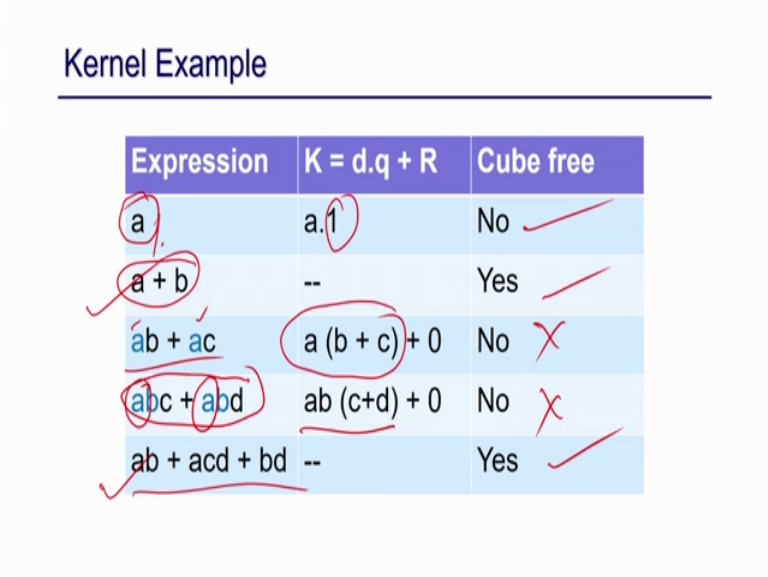
So, the answer I just talked about this is that, algebraically divide the function by co kernel this single Q is the co kernel and if this k is a kernel then that is particular the place where should we look into this right, that is what is I just talked about here right.

(Refer Slide Time: 63:50)




That the algebraic divide answer is in the kernel of k, but how it important I will just talk about this, before that I will explain example this kernel further right.

(Refer Slide Time: 63:54)

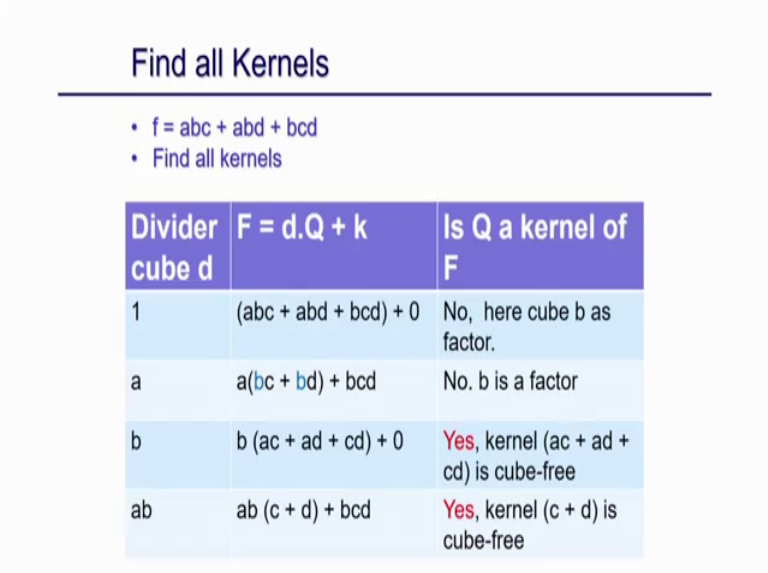



So, suppose a is my expression, is it kernel, no because it has all a common divisor co factor one, so this is not a cube free. Is this cube free? This is cube free, because there is no other factor here right this is cube free, this is cube free this expression, no because I have a common co factor here a, so I can represent this by this, so this is not cube free expression. Is this cube free, you can understand this is no, there is a common factor here

right, so I can represent this by this so this is not cube free, but this is cube free, because there is no way I can represent this like this right, so this is cube free.

So, this and this is a cube free expression here others are not right. So, when the idea is here, when you divide this function by a single cube, and then whatever the quotient you get, you try to find out whether this is kernel or not right. If it is not a kernel then we there is no common division, but if there is a kernel. Kernel means there is no common divisor there, so that is a kind of a unique expression, and then we should look for the common sub expression in that kernel itself right.

(Refer Slide Time: 65:11)

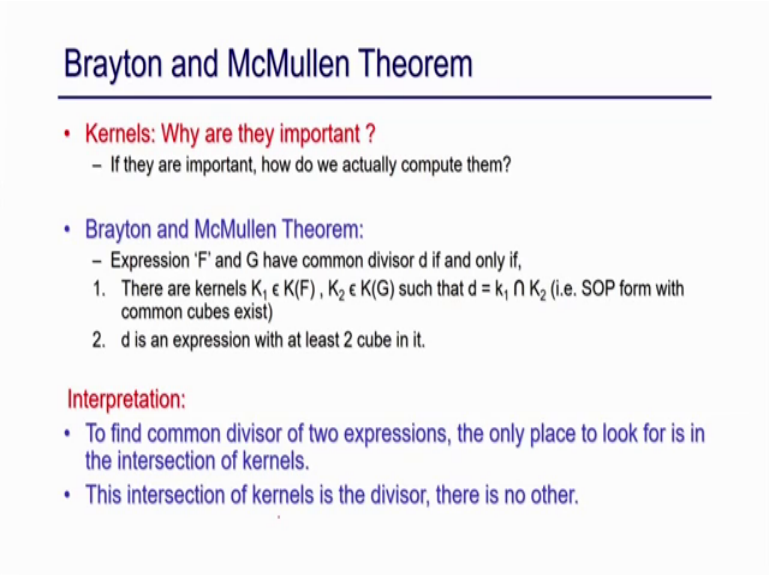



Just to go further here the idea is that you might have a multiple kernels right. So, the idea is here that you take that function; you try to find out all the kernels right. For example, here ah; that means, you try to divide that particular function by all product terms right, all single cube all the product terms of the input variable. So, I divide this by one, so this is not ah; obviously, not kernel, because this have b has, because here b is all common, so this is b has co factor

So, then if I divide this by a this is my quotient right, but again this is not kernel as I discussed here right, so because it has this a as the co factor, so this is not a kernel right. So, similarly here, so this is b is the factor here. So, now, if I just divide it by b I got this is as the quotient, and this is a kernel, because this does not have any, this is cube free. So, this is a cube free; that means, a kernel.

Similarly if I divide this by a b I will get c d as the quotient and this is; obviously, a cube free. So, this way if I just apply this divide this particular function by multiple such single cube, then I will get some quotient, and I will just check whether they are cube free or not, whether they are kernel or not, if they are kernel, so I will find out all this kernels right. Once I have all this kernels right, then what we can do

(Refer Slide Time: 66:28)



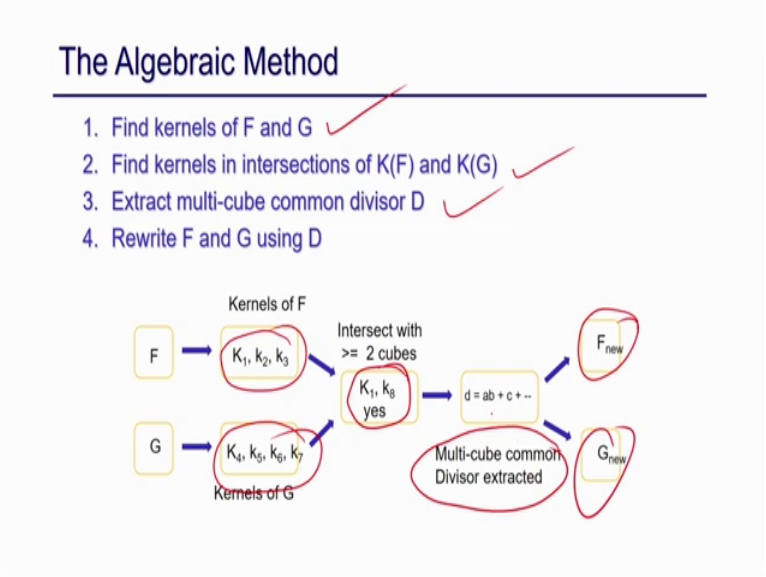

Because I now this is only for one function right. I have now multiple such functions and what I can do that, is given by this theorem Brayton hernal and McMullen theorem right that is important, because now when you try to find out this common sub expression between two function the. This theorem says that you will always look into the intersection of this kernels right, what it says that for expression F N G have common divisor D, if and only if they are kernels this, there are kernels this and this in their kernel, this k is the kernel of F, all the kernels of F and K G is the all the kernels of G. So, then your D must be in the intersection of this k 1 and k 2 right, and D is an expression of at least two cube right (Refer Time: 67:18) at least on that

So, what is the importance of this theorem is that interpretation or the importance of that theorem is that that, whenever you try to find out the common sub expression between this function, you find out the all the kernels of this two function and then you take the intersection right. If there is, if there are some inter, I mean common sub expression exist that must exist in that interaction only right. If there is no, there is no common sub

expression between this two function, there is no, then if it is not there in the ker, their intersection of kernels, there is, there is no other right; that is the important right.

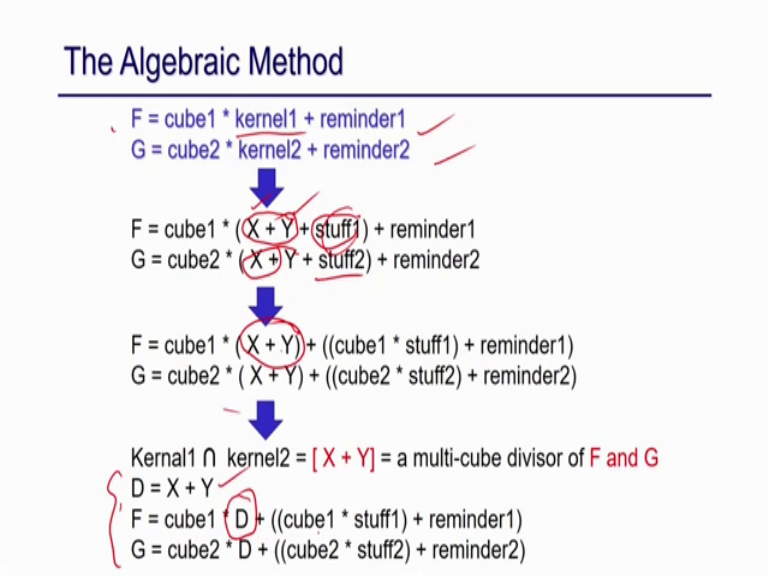
So, the idea is that to find common sub divisor of two such expression, the only place to look into is that, this intersection of kernel right. If that particular intersection does not have any common divisor, then there is no other right; that is actually give you some algorithmic approach now so you can understand that. Now I got a algorithm approach to find out the common divisor using the division algorithm, find out the common sub expression, if this two function right. We just talked about this right, so you find. So, you have two function say F and G, the steps is like that.

(Refer Slide Time: 68:29)




So, you find all the kernels of F and G. So, suppose for F 1 you got this k 1 k 2 k 3, all the kernels here right and then for G you get say this 4 kernels right. Then you take this intersection of this two kernel, where at least you should have 2 cubes, because there is another expression that it must be of 2 cubes right, so you find. Suppose for example, this k 1 and k 8 has the only kernel, where this has more than 2 cubes, and then you find out the multiple multi cube common divisor right from this k 1 and k 8 right; that is you extract the multi cube common divisor of D from this intersection right, and then you can rewrite this two using this common divisor, and that is D is your common divisor now just to elaborate it further.

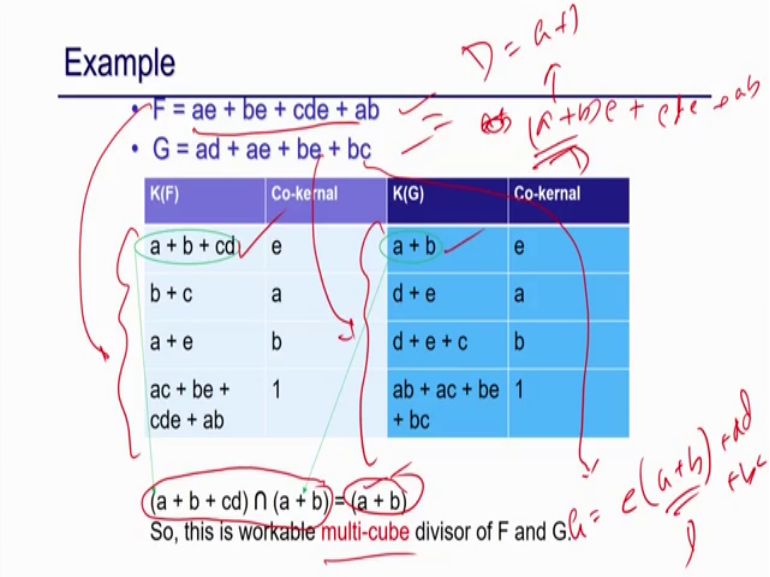
(Refer Slide Time: 69:10)




Suppose, you have this F is basically this. So, for example, this kernel 1 you can represent, because you are dividing this F by cube 1. So, you can rewrite F by this G by this, and then this kernel 1 has some common sub expression right; that is what is the point right. So, this is the, this is the kernel 1, and kernel 2 is that that intersection we just talked about right. So, this is about this k 1 and k 2 and then I can repre rewrite this by X, because that is the common sub expression part in the kernel. So, I can rewrite this kernel 1 by this, some other stuff is there, some other sub expressions and this kernel 2. This is the common part and this is the other part right.

Now, what I can do? I can just find out this is this cube 1 into this plus cube 1 into the other stuff, and I can rewrite this by this right, then what is happening this X plus Y is the common divisor part right, then I can have, this is my common divisor which is part of the kernel and which is have the at least 2 cube. So, this is the kernel, I found it, I can rewrite my function in this form right. So, D is X plus Y F 1 is cube 1 into D plus the other things, this cube 2 into D into this right.

(Refer Slide Time: 70:18)



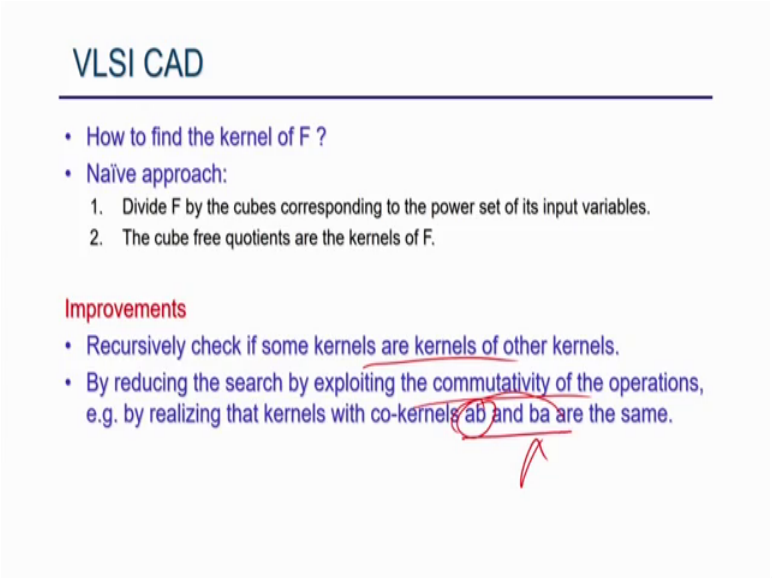

So, just to give an example. Suppose my F and G is this. So, I found all the kernels. So, this is say, this are the kernels of F right and this are the kernels of G. So, I can apply the other, the other algorithm just to find out the kernels. Now we try to find out the kernels in the intersection right. So, intersection I have this right. So, if I just do this, if you take this and this intersection of this two will give you this a plus b, which is basically have more than two terms, at least two terms so this is a multi cube. So, this a plus b is a common divisor of this right.

So, I can rewrite this expression using as this already I have written, so a plus b right. So, this a plus b into 71 e plus this is the other stuff and this is the reminder c d e plus a b right. So, this is my common divisor and this I can rewrite like this G equal to I can do this a, this e into a plus b plus a d plus b c. So, this is my common term right. So, I can replace this a and this by some expression say d equal to a plus b, and I can just do a d here, d here right, this is what I am doing.

So, this is how the whole thing works right so just to summarize, you just try to find out the kernels; that means, you divide those two function using single cube, single product terms to find out the dividend and you see whether that is kernel cube free or not; that means, then that is a kernel, you find out all the kernels of two function, and when you take the intersection of the kernel and you find out those intersection, some sub expression which is basically are more than two cubes right, and that is your common

sub expression right. This is how you can do this whole algorithm. Now the question is here, how to find the kernels of an F right; that is something what we have not discussed.

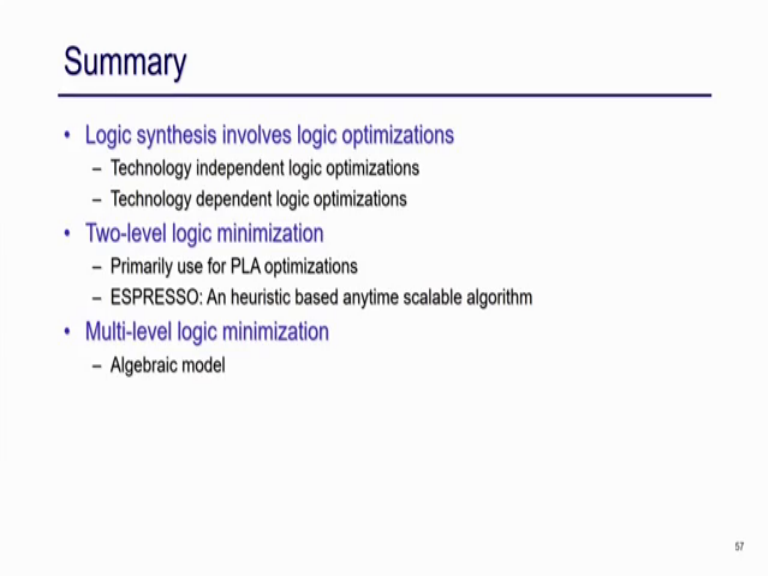
(Refer Slide Time: 72:21)




The naive approach is something you divide this D by all possible product term right, because that is the part set of input right. Because this is single cube, I am going to talk about only that single product right. I mean this its not the multiple product right and then I can, whenever I found some kernel which is cube free, I will just take quotient which is cube free, I can take that right on the improvement side.

I can actually recursively check if some kernels are the kernel of other right sub set of other, or I can just see like commutativity operation, whether if this a b and b a is in the same kernel. If I found a b I am not going to explore b a and so on right. This is something you can do. So, this is how I can actually find the kernels right. This is the overall summary that this is how I, actually I can use this division operation, just to find out the common sub expression among multiple expressions right.

(Refer Slide Time: 73:17)




So, just to summarize in today's discussion. So, we have talked about this logic synthesis and what we have found that, logic synthesis is straight forward, but only complicated part is that logic optimization part, which can be technology dependent and technology independent. So, today we have discussed about this independent part. So, we have discussed two strategies; like it can be two level optimizations or say multi level optimization. And in two level we can have exact solution like Karnugh map based solution or say this Quine McCluskey solution and you also have heuristic solution like espresso.

So, we have discussed ESPRESSO in detail and also for multi level optimizations. We have can have such several set of rules and we can apply those rule algebraically or say rule, based way just to find out some optimum circuit and we can have two different model; Boolean model or algebraic model. So, we have discussed today how we can use this algebraic model just to find out some common sub expression using the division of algorithm of between two functions right, and this is how we conclude this discussion. In next discussion we are going to talk about this technology dependent logic optimization specifically for F P G as ok.

Thank you.