

Electronic Design Automation
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 10
Synthesis: Part 3

I have talked about two-level optimizations. Today we will be extending our discussion. We will be talking about multilevel logic minimization or optimization. We have mentioned that there are many circuits for which multilevel logic optimization proves greatly beneficial as compared to two-level logic. We will start by giving an example of such a circuit for which multilevel minimization helps greatly.

(Refer Slide Time: 01:45)

Multilevel Logic Minimization CET I.I.T. KGP

- In many applications, 2-level logic is unsuitable as compared to random (multilevel) logic.
 - Gates with high fanin are slow, and take more area.
 - It makes sense to transform a 2-level logic realization to multi-level logic.

1 → 2

There are a few things you need to look at that when you talk about multilevel realization. It is not only just the number of levels we talk about; there is also another thing which needs to be looked at. That is the number of inputs to the gates which are required. As I had mentioned that as you have a gate with high number of inputs, gate becomes slow. Just a simple example. Suppose you have a nine input OR gate. Now this nine input OR gate you can implement in two-level realization like this using NOR and NAND. Apparently you can say that here, there is only

one level and here there are two-levels. But in practice this gate is much slower because of this high fan in. Because of the high capacity when high impedance the capacitive load has to face as I had mentioned in the last class. So, particularly for high fan in gates it is beneficial to break that gate up into smaller gates into multilevel logic. So, this is one situation where multilevel logic realization makes sense break up a gate with a large number of inputs into multiple levels of smaller gates the speed will improve. Well another case is just said that for some kind of logic multilevel minimization may prove to be expensive.

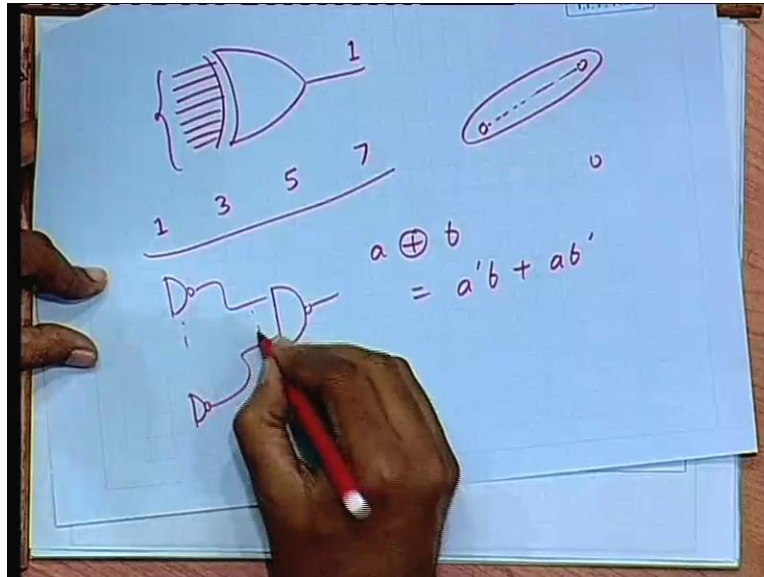
(Refer Slide Time: 03:33)

CET
I.I.T. KGP

- **A classical example :: XOR function**
 - For an 8-input XOR function,
 - For 2-level NAND-NAND realization
 - ${}^8C_1 + {}^8C_3 + {}^8C_5 + {}^8C_7 = 128$ NAND8 gates
 - 1 NAND128 gate
 - For 3-level XOR realization
 - 7 XOR2 gates
 - 28 NAND2 gates
 - Number of levels = 9

So let us take one example of that category. That example is the XOR function. XOR function if you recall it is a circuit where let us just take a eight input function. For an eight input function, say I have an XOR gate; I have eight inputs all right. Eight input functions, I can, suppose I want to realize this in a two-level logic implementation. Now in a two-level logic implementation you think of an XOR gate what is the kind of requirement you have. XOR gate is essentially a parity detector right.

(Refer Slide Time: 04:33)



So just let me tell you I am drawing this big XOR gate. If there are eight inputs the output will be one if odd number of inputs are one odd number of inputs. Means one of the input is 1 3 of the inputs are 1 5 of the inputs are 1 7 of the inputs are 1. Actually how many such combinations are there? The number of combinations will be just out of 8. So how many ways I can have one input to be made 1 3 inputs 5 and 7. This makes 128 and another thing is that one characteristic of this XOR gate is that the true min terms for an XOR gate. The true min terms have a distance of more than one which means they are never connected by a single edge in the cube representation. If they were connected by an edge they could have been combined in a cube. But for example for a 2 input function, say if you have a XOR b, the function says a NOT b OR a b NOT. They cannot be combined.

Similarly for 3 input, 4 input XOR they cannot be combined. So whatever number you have obtained here, that will give the number of NAND gates and since there are 8 inputs each of these NAND gates would be having 8 inputs. So you need this number comes to 128. You need 128 input NAND gates. So in the first level you will be having so many NAND gates and in the second level you will be having one big NAND gate feeding all these 128 input NAND gates. So you try to understand that these eight input XOR I can very easily implement in a two-level

implementation two-level realization. But I am getting, I am requiring 129 gates all of them around very big sizes, say one of them is of extremely big size 128 input.


(Refer Slide Time: 06:32)

© CET
I.I.T. KGP

• **A classical example :: XOR function**

– For an 8-input XOR function,

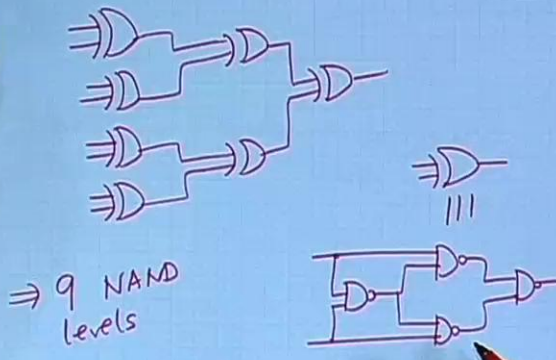
- For 2-level NAND-NAND realization
 - ${}^8C_1 + {}^8C_3 + {}^8C_5 + {}^8C_7 = 128$ NAND8 gates
 - 1 NAND128 gate
- For 3-level XOR realization
 - 7 XOR2 gates
 - 28 NAND2 gates
 - Number of levels = 9



So this is possibly not practical. Now let us try to think of the other way round. So how do we do? It can we break this up into multiple levels and do it.

(Refer Slide Time: 07:10)

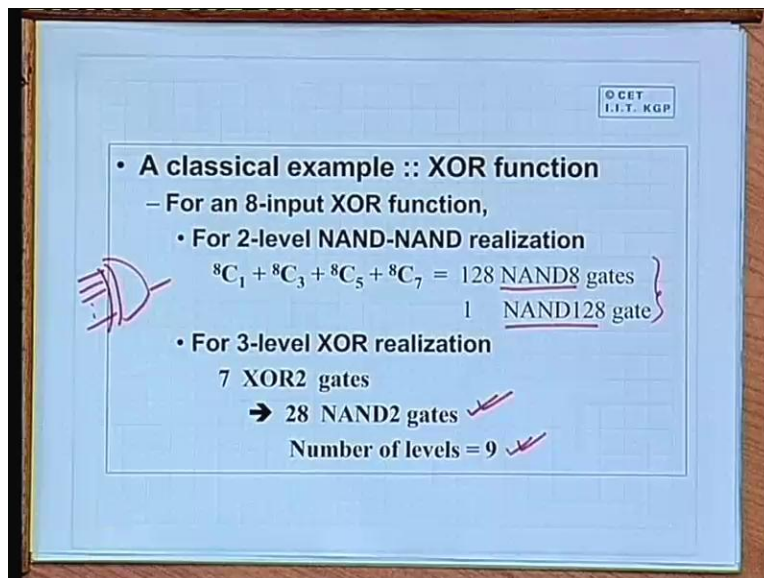
© CET
I.I.T. KGP



⇒ 9 NAND levels

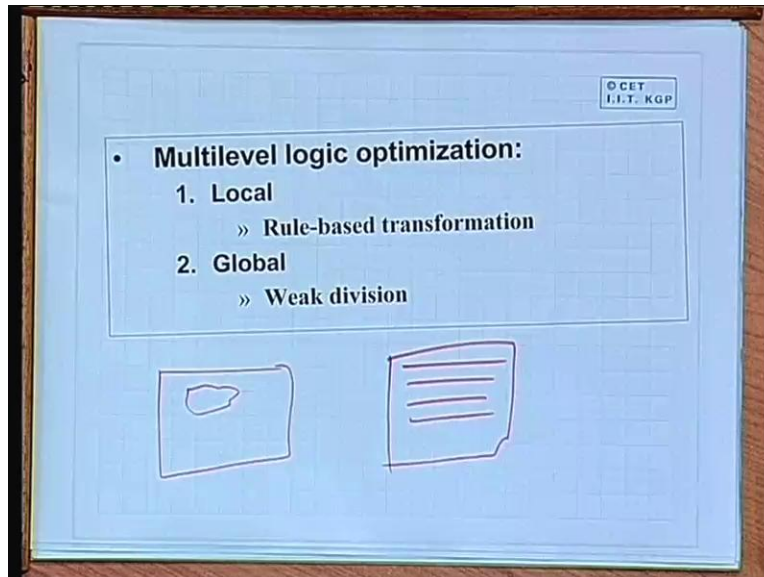
See this 8 input XOR, you can also implement like this by using smaller 2 input XOR gates. Suppose we have something like this. We use cascade of XOR gates like this and you please note all are 2 input XOR gates. So we need seven 2 input XOR gates and each 2 input XOR gate can be implemented like this. So effectively each NAND gate each XOR gate is equivalent to three levels of NAND and there are three levels of XOR. So we have nine NAND levels in total. But the advantage we gain is that we need only 4 into 728, 2 input NAND gates. This is the big advantage we get.

(Refer Slide Time: 08:37)



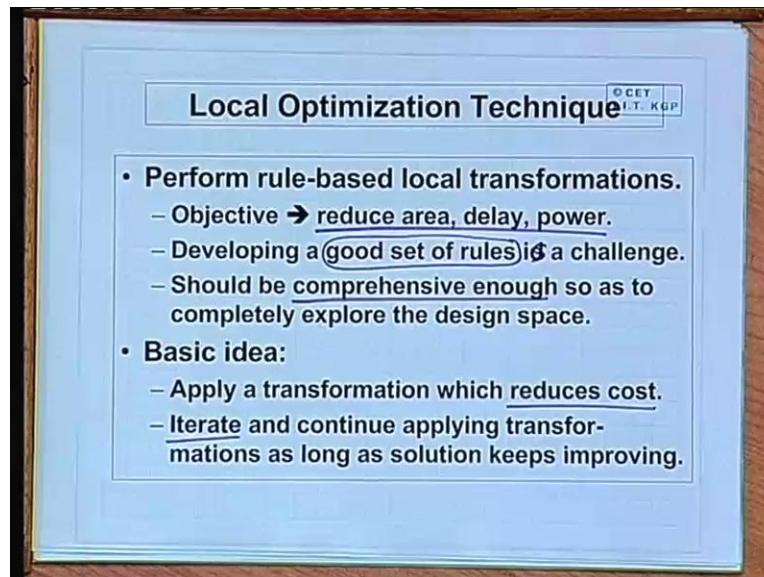
So we require 28 two input NAND gates. Number of levels have increased some what but as I have again mentioned. The delay of this alternate realization will be much less as compared to this simplistic realization. So this particular example tells you or shows you that there are some examples of circuits for which two-level realizations are not feasible. We must go for multilevel realizations and for multilevel optimization there are a number of methods which have been reported.

(Refer Slide Time: 09:17)



Now, broadly these methods can be classified into local and global. Local means that you look at one of the equations or one of the outputs at a time. You try to do some kind of local optimization into a net list you look into a circuit. You look into the small part of it and try to do some optimization there global optimization. Means you look at all the functions at the same time and try to do some transformations on all of them so that you get some optimization. Now since we are looking at the whole problem at the same time, this is called global. In contrast the earlier method is called local. So let me try to give you some idea that how this local and global transformations work. So as you can say it is written here local transformation. Local optimizations are based on some kind of rule based transformations lets say what kind of rules.

(Refer Slide Time: 10:27)



So the objective of this rule based transformations are of course same reduce area, delay or power they are again mutually conflicting. So in this method the main problem is to develop a good set of rules. Because you have a set of rules which you repeatedly, apply to a net list in order to improve upon it. You have a given net list, you have a set of rules you see that whether you can apply a particular rule to the part of the net list if you can you get an optimization out there. So you will have to enumerate as a designer. The list of all such rules you need to apply and check for this kind of local optimization on the circuit and again these rules should be comprehensive enough. Now the basic idea is just what I have just mentioned that you apply one of the transformations to a given net list which reduces the cost of implementation. And you go on iterate this, is a greedy approach. You iterate and continue applying transformations as long as you get better and better solutions. So this is a purely greedy approach it is applied on a given net list based on a set of rules. Now the kind of rules that are user applied are fairly simple.

(Refer Slide Time: 12:06)

• **AND/OR transformations** // CET
I.I.T. KGP

- Reduce the size of the circuit, critical path.
- Typical transformations:
 - $a \cdot 1 = a$
 - $a + 1 = 1$
 - $a + a' = 1$
 - $a \cdot a' = 0$
 - $(a')' = a$
 - $a + a' \cdot b = a + b$
 - $\text{xor}(\text{xor}(a_1, a_2, \dots, a_n), b) = \text{xor}(a_1, a_2, \dots, a_n, b)$

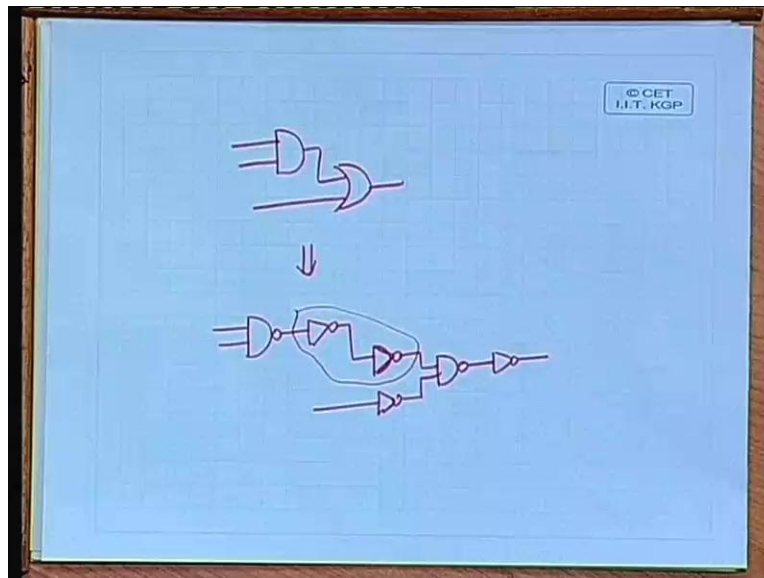
• **Transform the AND/OR form to NAND form (or NOR form).**

So the first sets of rules are like this. These are called simple AND or transformations. I have shown some of these rules. Many of them are based on simple. You can say simple identities of Boolean algebra; a AND one equal to a, a OR 1 equal to one these kind of things. So if in a circuit you encounter, for example this, if you encounter two NOT gates in cascade obviously you will replace everything by a single wire. So XOR of some input set of inputs AND b is the same as the single XOR gate with all of them as inputs. So this means if we have XOR of some inputs and another XOR of some other input then you replace it by a single XOR gate with all the inputs. So this kind of rules can be there. Many of them these are very simple rules fairly simple rules.

You just apply these rules one by one. Now another thing is that most of the practical implementations or realizations are based on NAND-NAND or NOR-NOR logic. Now this requirement comes out from the ultimate because of the target you are trying to implement on typically CMOS and you know CMOS can realize only negative gates NAND, NOR this kind of things. So in general after these transformations you will be getting a net list which will contain AND and OR. So after the total we are doing after this AND OR transformations. We will be translating this AND and OR gates to NAND or NOR form. Now you know that how a NAND

gate can be, how a AND gate can be converted to a NAND gate NAND followed by a NOT. Similarly OR to NAND which you know.

(Refer Slide Time: 14:19)



So after doing this transformation, we will be getting say I am giving an example. Suppose you had a circuit like this and, AND followed by an OR. Now, after doing this transformation what you get is this. AND will be replaced by NAND followed by NOT and OR will be replaced by NOT; NOT this kind of thing. So you see that after you do this kind of translation, these kinds of chains of NOT gates are very common. You can combine this to a NOT gate and you can make it a single one you can replace them by a single wire. So after this transformations number of additional optimizations will come up which you can apply. But here the advantage is that you need to concentrate only on NAND or NOT.

(Refer Slide Time: 15:27)

© CET
I.I.T. KGP

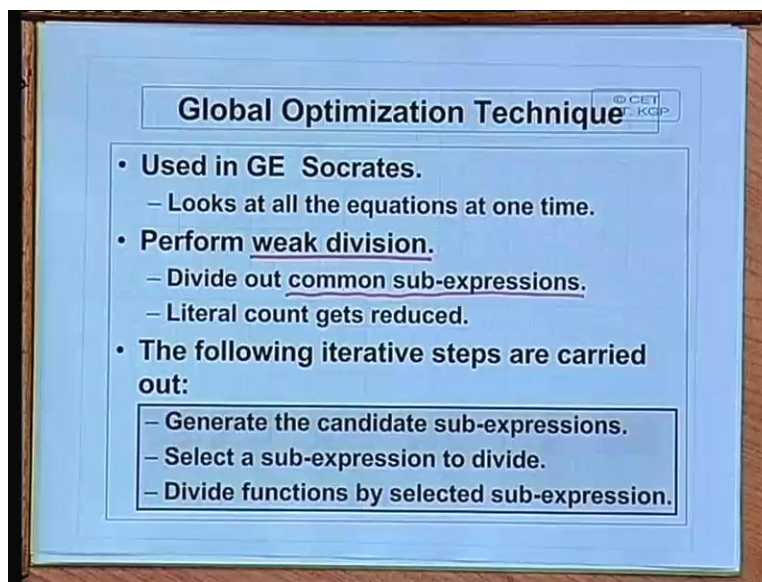
- **NAND (NOR) transformations**
 - Some synthesis systems assume that all gates are of the same type (NAND or NOR).
 - Does not require technology mapping.
 - Rules framed that transform a NAND (NOR) network to another.
 - Examples:
 - $\text{NAND}(\text{NOT}(\text{NAND}(a,b)), c) = \text{NAND}(a,b,c)$
 - $\text{NAND}(\text{NAND}(a,b,c), \text{NAND}(a,b,c')) = \text{NAND}(a,b)$

So there are a second set of transformations you need to apply here. These are called NAND, OR, NOR transformations. Now here after this is done you can very well assume that all gates are of the same type. Typically we use NAND more often than we use NOT. So since all gates are of type NAND and which is one of the basic gates, you do not require any specific technology mapping. Because NAND is always a part of any library and now we have a set of rules which are framed based on NAND or NOR networks like we have an example out here. This example means if you just what it means NAND of a and b which means we have NAND of a and b. NOT of that NOT of that NAND of that and c, there is another NAND c. See what this rule says that, if you see that you have a net list like this, a sub graph like this entire thing is equivalent to a three input NAND. So you can replace it by a 3 input NAND. Similarly if you have NAND of a b c NAND of a b c bar, NAND of that c goes you simply have a two input NAND.

So here I have shown only a couple of rules. But there will be a big set of such rules which you can apply to this NAND level net list and you can carry out optimization. But you try you see that this now becomes a search problem. The given net list which you have that is like a big graph and the rules that you can potentially apply on them to carry out optimization. They are

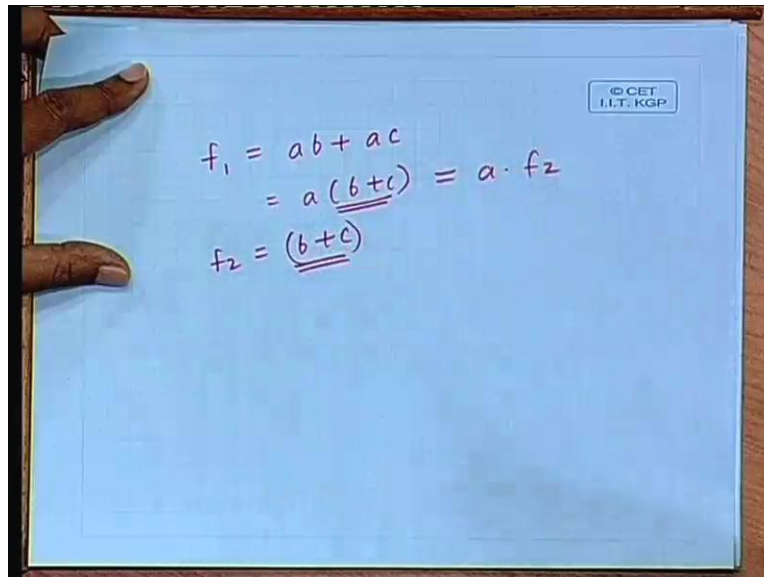
like small sub graphs. Now you will have to search the net list for presence of those sub graphs. And if you see that one of the sub graphs are present. You can apply that rule to carry out the optimization. And more you do the optimization your circuit will become smaller and faster. So and I said that this is a greedy approach. Here you look at some rules and you look at some parts of circuit where that rule can be applied and you straight away apply that. But here you are not looking at the total picture at a time; looking at say all the functions, looking at how to make transformations to the circuit. So that you can say globally you can have some optimization. We are not looking at that we are looking at some local benefits by making some transformations here and there. So talking about global optimization.

(Refer Slide Time: 18:34)



The technique we mentioned was used in a package called Socrates. This uses a method called weak division. Say idea is very simple. This is a very well known method which has been used for a long time by the compiler writers. Basically you try to identify the common sub expressions and you try to divide them out.

(Refer Slide Time: 19:05)


$$\begin{aligned}f_1 &= ab + ac \\ &= a\underline{(b+c)} = a \cdot \underline{f_2} \\ f_2 &= \underline{(b+c)}\end{aligned}$$

See common sub expression means if you have, say if you have a function a b plus a c well, which you can write as a into b or b plus c. You have another function say only b OR c, then you can see that this is a common sub expression and instead of evaluating it twice why do not you evaluate to only once? This f 1 instead of writing a, into b plus c, I can simply write a, into f 2. So I can save one OR gates. So the idea is this that if you can identify the common sub expression you evaluate to only once using gates in that you use it in the other place. So you can save on the hardware. Now in this approach you use the following iterative steps. Idea is again simple.

You find out all possible sub expressions which are common to two or more Boolean equations. Now after that you try to evaluate. Suppose you see that there are four possible sub expressions. Now you try to find out which one of them is better there must be a way to evaluate. Now the one which is best you find you use that for factoring the equations and you go on repeating this. So select a sub expression to divide based on the goodness measure; we will show how we do that. Then we actually divide the functions now see dividing up into sub expressions. And using this is something like division. That is why the term weak division is used. Now I am explaining these steps with an example.

(Refer Slide Time: 21:08)

$u = ab$

Example

© CET
I.I.T. KGP

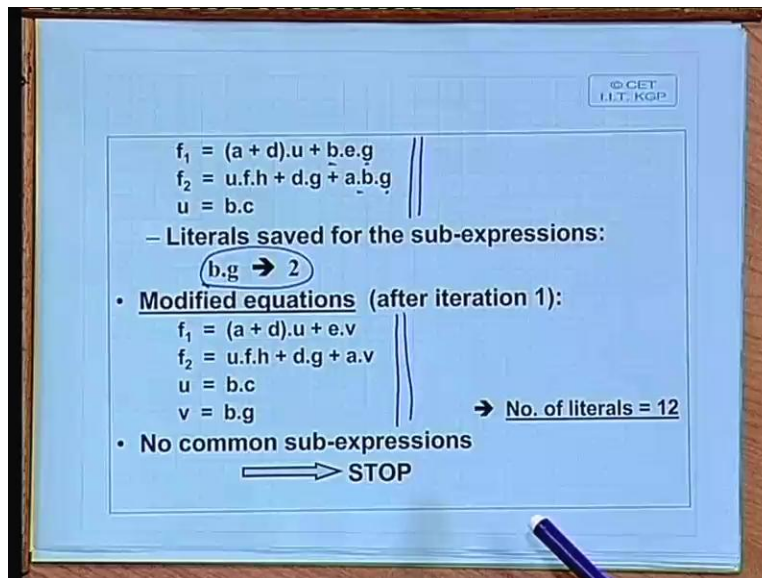
- **Original equations:**
 $f_1 = a.b.c + b.c.d + b.e.g$ $f_2 = b.c.f.h + d.g + a.b.g$
→ No. of literals = 18
- We find literals saved for sub-expressions:
 $b.c \rightarrow 4$ $a.b \rightarrow 2$
 $a + d \rightarrow 2$ $b.g \rightarrow 2$
- Select the sub-expression bc.
- **Modified equations (after iteration 1):**
 $f_1 = (a + d).u + b.e.g$
 $f_2 = u.f.h + d.g + a.b.g$
 $u = b.c$ → No. of literals = 14

Suppose I have a problem like this. There are two equations. Two expressions to be solved; actually this corresponds to a circuit comprising of two outputs f_1 and f_2 . This, a, b, c, d, e, f, g are the inputs gate. Now our measure of goodness is simply to count the number of literals. This is a fairly you can say accurate approach and many methods simply count the number of literals in order to estimate the cost of implementation. Now in the original one how many literals are there? 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18. So in the initial problem number of literals was 18. But if you look at these expressions now and you try to find out what are the common sub expressions which exist in any one or more than one. First you see b, c, b, c is present here, here, and here a, b ; a, b is present here and here a, OR, b . Well a, OR, d you can have from here by factoring by taking b, c common a, OR, d that is all b, g is here and b, d is here.

Now for each of these literals that you estimate that if I use this factorization that how many literals I am saving. So I am not showing the detailed step. Actually what we will do is that, suppose you select for example a, b a, b is the sub expression you select. So you actually generate a, b from some other a, b is here. Suppose you write $u = a, b$ and u use here as well as here and you count how many literals you are saving in the process. So this process will repeat for all the possible sub expressions you find that for this particular example b, c gives you the best

benefit. So you select the sub expression b c. So after the first iteration you modify the equations by replacing b c with some variable u and where ever b c is there, you replace it by u. Here also I am factoring at the same time for the next step. So this is the new equation. Now here you find that the number of literals is 14. So we get a saving of 4. So you go on repeating this. So you have this with you you have this.

(Refer Slide Time: 24:21)



This is now the result at the end of iteration one. Now you check the sub expressions now b g gives you the best benefit b g. So the next iteration you select b g is here as well as here. So define another variable called v. So now the literals become 12, after this you find that there is no other common sub expression that you can find. So we will stop out here. So after doing this factoring and this division you get you get a modified specification. You can say where the common sub expressions are not there and where ever the common sub expressions were there you had utilized them by division and you have reduced the number of literals out there.

So you had obtained some implementation which will be more efficient in the process. Now the problem with this kind of an approach is that every time you are trying to go from a given solution to a better solution. So, essentially this is a greedy algorithm every time you are

selecting the best approach. So I am again repeating any algorithm which always tries to go for the best alternative in the present scenario has a great chance of getting stuck into a local minima, it may miss out the globally best solution. It can get stuck into a local solution which may be far of from the best solution.

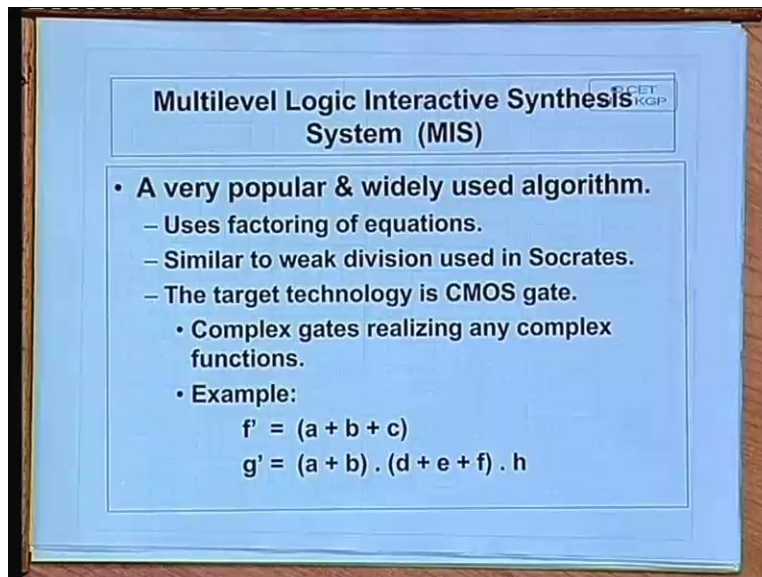
(Refer Slide Time: 26:18)

About the algorithm © CET I.I.T. KGP

- **Basically a greedy algorithm**
 - Can get stuck in local minima.
 - Give a “jerk” to come out of local minima.
 - Like the “gasp” function in Espresso.
- Generation of all candidate expressions is expensive.
 - Some heuristic used.

So this is a greedy algorithm, it can get stuck into a local minima, and where ever, which ever method has a chance of getting stuck in a local minima, some how we give a jerk or some kind of randomness to come out of the local minima. So in terms of the contour of the solution space like this, if it is greedy it will always go down. You will go stuck out here. But from here, if you give a jerk like this, somehow may be, may be, from here you can start exploring another part of the solution space and you can get to another solution which is better lower cost. And another thing is that generation of all candidate expressions is expensive. I have taken a very small example. But imagine if we have a very large number of equations, with very large number of variables, generating all possible sub expressions which are common, that it itself is an expensive process. So some kind of heuristic must be used. Now in fact there are many multilevel optimization packages that are being reported and are used and one of the methods. I mentioned very briefly about is one of the more successful and the most popularly used method.

(Refer Slide Time: 27:43)

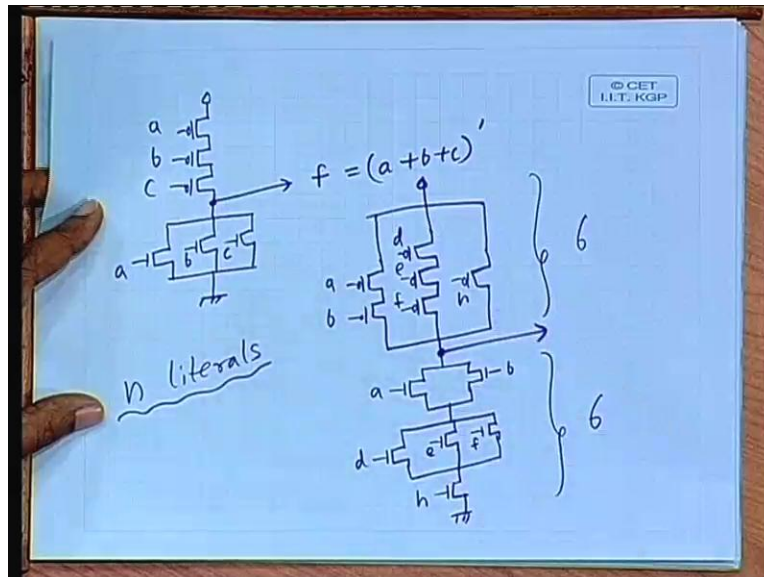


Multilevel Logic Interactive Synthesis System (MIS)

- A very popular & widely used algorithm.
 - Uses factoring of equations.
 - Similar to weak division used in Socrates.
 - The target technology is CMOS gate.
 - Complex gates realizing any complex functions.
 - Example:
$$f' = (a + b + c)$$
$$g' = (a + b) \cdot (d + e + f) \cdot h$$

This is called MIS multilevel logic interactive synthesis. Now this MIS also uses factoring of equations just like the weak division we just mentioned. And one thing out here is that this MIS specifically targets CMOS gate as the target technology. Target technology CMOS means that what ever equation I write, it is a negative equation because I am showing you why. Let us take this example. This is a three input NOR gate. Now in CMOS a 3 input NOR gate will look like this.

(Refer Slide Time: 28:36)

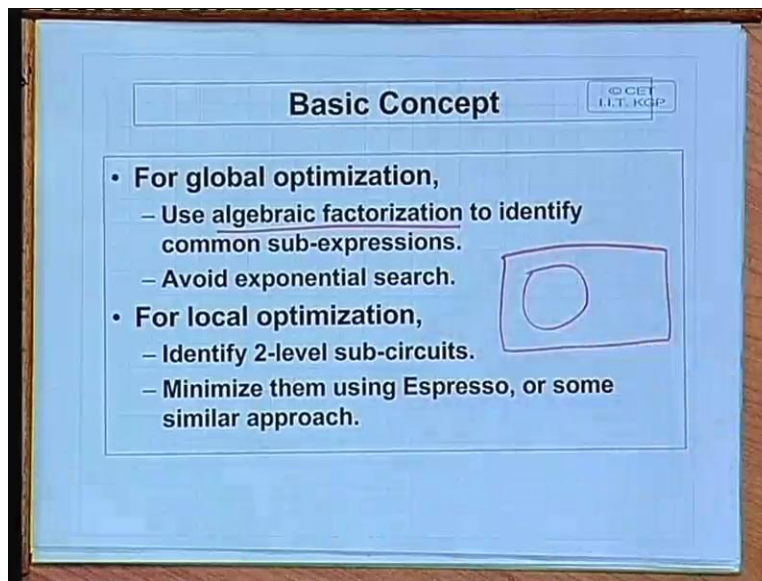


These are the three PMOS transistors. And the three NMOS transistors are in parallel. So we have a out here, b out here, c out here, a b c. And this is the output and one characteristic of the CMOS gate implementation is that it always implements a negative logic. That means f equal to a OR, b OR, c NOT, or f NOT equal to a OR b OR c whatever it is. Similarly if we have something like this a OR b and d OR e OR f h. This also you can implement very easily. So you will be having a pull up network and pull down network. Say I am showing this a OR b. So I am first drawing the pull down network there are three parts which are in series first a OR b a b and d OR e OR f d e f h. This is the pull down network.

Similarly pull up network will be the complement of this. So it will be parallel combination of a b. This will be PMOS a and b. Then there will be another d f another part, three transistors in series. This will be d e and f another part will be h. Now this will be connected to the pull up network. This also will implement a negative phase. So when I say the target technology CMOS, it means we are trying to go for a solution like this. Now you see in this one the number of literals were 1, 2, 3, 4, 5 and 6. In terms of the CMOS realization there are six transistors in the pull down, six transistors in the pull up.

So if there are n literals in terms of CMOS implementation we can very easily compute the cost. There will be two n transistors. This is why number of literals used sometimes used to estimate the cost of a solution. So MIS specifically targets CMOS gates and for that purpose it uses the number of literals as a measure of computing the cost.

(Refer Slide Time: 31:52)



The slide is titled "Basic Concept" and includes a copyright notice for "© CET I.I.T. KGP". It contains two main bullet points:

- **For global optimization,**
 - Use algebraic factorization to identify common sub-expressions.
 - Avoid exponential search.
- **For local optimization,**
 - Identify 2-level sub-circuits.
 - Minimize them using Espresso, or some similar approach.

To the right of the text, there is a hand-drawn diagram of a rectangular box containing a smaller circle, representing a sub-circuit.

So basic concept is well at this level it is similar to Socrates it uses algebraic factorization identifies common sub expression. But it tries to avoid exponential search using some heuristics and for local optimization it identifies two-level sub circuits and minimizes them using espresso. So from the bigger circuit it also identifies smaller sub circles in between and it tries to optimize them using espresso. So espresso is used as a function which is called from MIS. This is I am not going in detail. But this is roughly the approach this is used for local optimization.

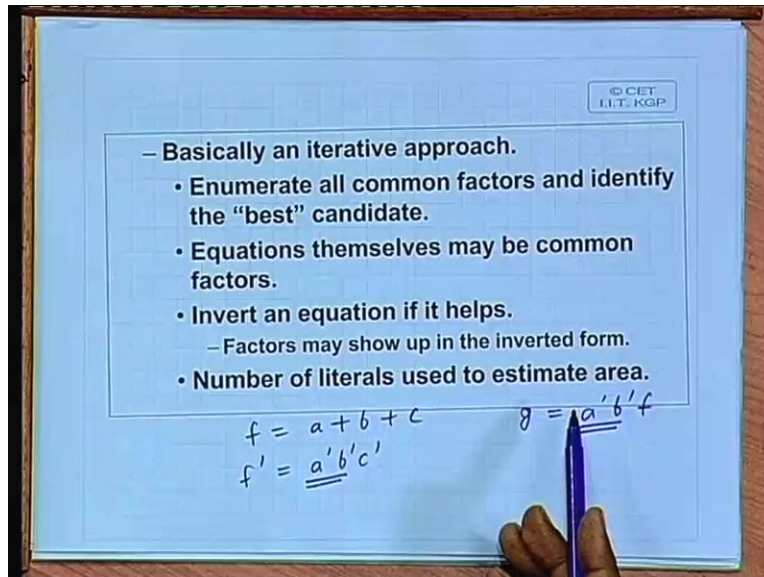
(Refer Slide Time: 32:42)

Global Optimization Approach

- **Given a netlist of gates**
 - Scan the network
 - Apply simple heuristics to “clean up” the netlist.
 - Constant propagation
 - Double inverter elimination
 - Espresso minimization of each equation.
 - Then proceed for global optimization with a view to minimize area.

But for global optimization it uses a few techniques. It scans the whole network it uses simple heuristics to clean up the net list like the two NOT gates in series, gets eliminated. This is called cleaning up. There is something called constant propagation. Constant propagation means if we have a gate with a and a constant one in the output feeding another gate, say an OR gate b then it replaces this entire thing by a. So this is equivalent to the simple OR gate a and b this is called constant propagation. This constant propagates the value of a here directly. So this gets eliminated double invert as I have already mentioned and each equation which you get for the sub circuits you use espresso for further optimization because espresso is known to work well there. So after doing this, you proceed to global optimization by using factoring and other thing which is similar to Socrates.

(Refer Slide Time: 33:59)



So this is also an iterative approach. So you just eliminate the common factors identify the basic candidates etcetera. So equations themselves may be common factors some times it helps in inverting a equation. Say you have an equation like f equal to a OR b OR c and somehow you are unable to find out any sub expressions. Suppose there is another function g which is $\overline{a b} f$, but apparently there are no sub expressions. But suppose invert this equation f' , then this becomes $\overline{a b c}$ and immediately this sub expression gets exposed. So sometimes inverting an equation may expose further factors of sub expressions and again. Since we are using CMOS as the target number of literals is used to estimate the total area, right?

(Refer Slide Time: 35:09)

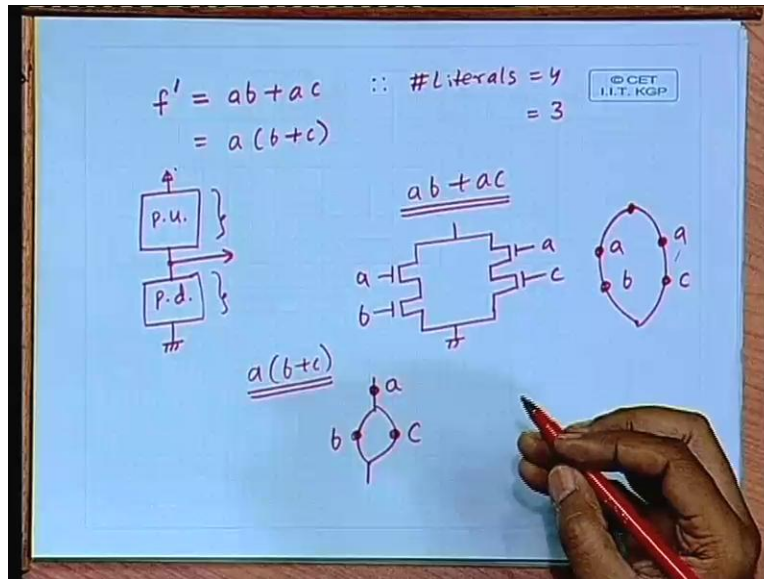
Some Illustrative Examples © CET I.I.T. KGP

- **Factoring can reduce area.**
 - An equation in simple sum-of-products form can have many literals.
 - Many transistors for CMOS realization.
 - Factoring the equation reduces the number of literals.
 - Reduces number of transistors in CMOS realization.

Handwritten diagram below the text:
A central point branches into two points: "common subexpr" and "factoring". Both are underlined.

So let us look at some of examples, it will be clear that how it works. What we are trying to say is that in the global optimization step we are using factoring. Factoring means that weak division that we are trying to find out the common sub expression and factoring. Suppose you have almost you are also going to do factoring sub expression together because there are two things. One is common sub expressions other is factoring both will give benefit as we will see through examples. So our ultimate target is to do some kind of global optimization technique which can lead to a reduction in area. Now an equation may be given in simple sum of products form. It will have literals and we have said from the literals. We can immediately count the number of transistors for CMOS realization. And our idea is that by identifying either common sub expressions or through factoring we want to reduce the number of literals. And if we can reduce the number of literals we can immediately reduce the number of transistors in the CMOS realization also. Now let us try to look at some examples and see that how it can be done or how this is achieved lets take a simple example.

(Refer Slide Time: 36:57)

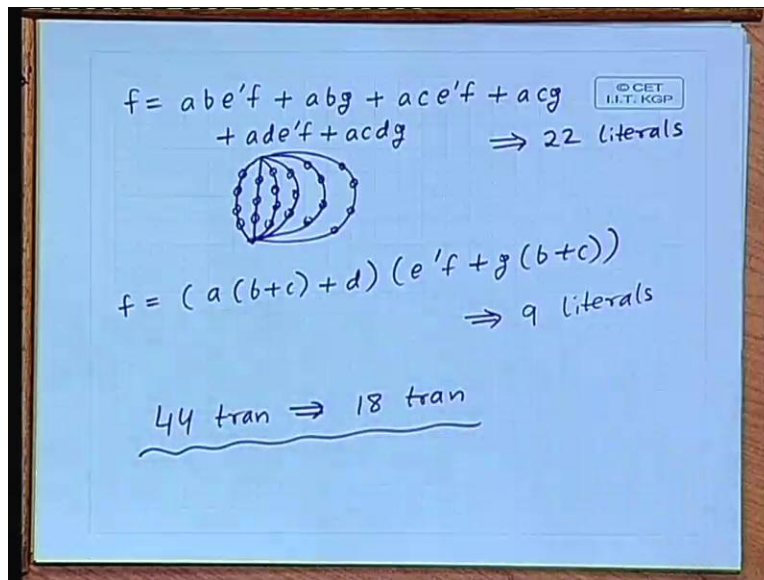


Well since it is CMOS realization only negative gates, I assume that it is complement. I want to implement this function $a b$ or $a c$. Now naturally you can do factorization. And you can see that through factorization in the first case the numbers of literals were 4, 1, 2, 3, 4 and here this has been reduced to 3. So apparently we can expect less number of you can say transistors in CMOS realization. Now in a CMOS network there is a pull up network comprising of p type transistors. There is a pull down network comprising of n type transistors. This is connected to v d d. This is connected to ground pull up and pull down are connected and this is the output. Now pull up and pull down network are complementary of each other. So if we show one of them, other will be simply complement of that and number of transistors out here is equal to the number of transistors out here. So for this function, for example if I had this, $a b$ or $a c$. Then the pull down network would look like this $a b a c$, four transistors.

This sometimes draws pictorially like this. Just for conveying instead of the transistors just to show like this $a b a c$ these dots represents transistors. Instead of drawing, this we will be drawing it like this and if you now do the factoring this $a b$ OR $a c$. You can immediately see that our drawing this picture now becomes like this is $a b c$. So instead of four there were three transistors in the pull down. So in the pull up also there will be three transistors instead of having

two. So number one thing is that if we are able to do factoring, then invariably you are reducing the number of literals and hence the number of transistors. So this is one thing you have to keep in mind. Let us take another example.

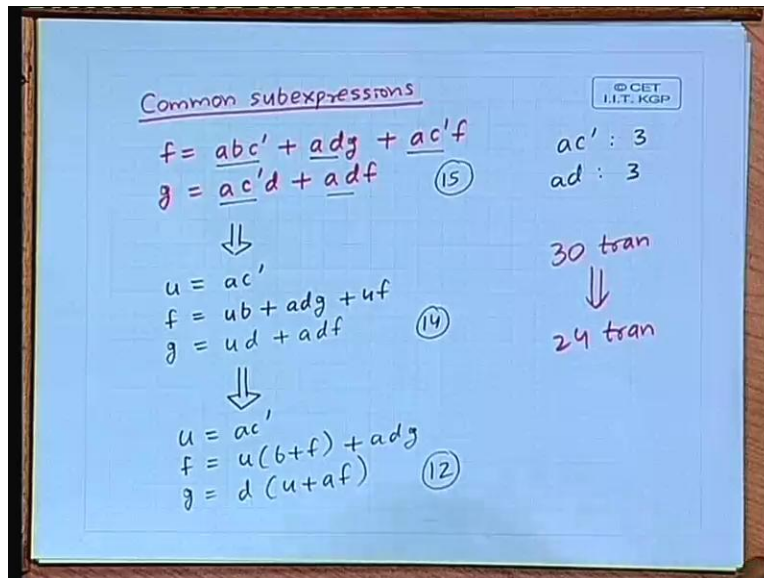
(Refer Slide Time: 39:54)



This is a slightly more complex example. Suppose you have a function like this $f = abe'f + abg + ace'f + acg + ade'f + acdg$. Suppose you have a function like this, you count the number of literals. You see that they are 22, 7, 11, 14, 18, 22, so there are 22 literals. Now if we want to have a straight away implementation of this function, then with respect to the pull down network, there will be one arm with four transistors 1, 2, 3, 4. There will be another with 3, 1, 2, 3, there will be another with 4, 1, 2, 3, 4, another with 3, 1, 2, 3, and another two with 4 each 1, 2, 3, 4, 1, 2, 3, 4. So as you can see only 22 transistors in the pull up pull down as well as in the pull up. Now if you just I will deliberately manipulate, this equation you can see that it can be factorized this equation you can factorize and final form. You can get in factorized into this. This is the final form and here if you count the number of literals you have 1, 2, 3, 4, 5, 6, 7, 8, only nine literals in terms of the implementation. You have gone down from 44 transistors to 18 transistors. This is a significant saving. So simply factoring the

equations alone will lead to a great saving in terms of the number of transistors. Well not only that you can also use common sub expressions.

(Refer Slide Time: 42:42)

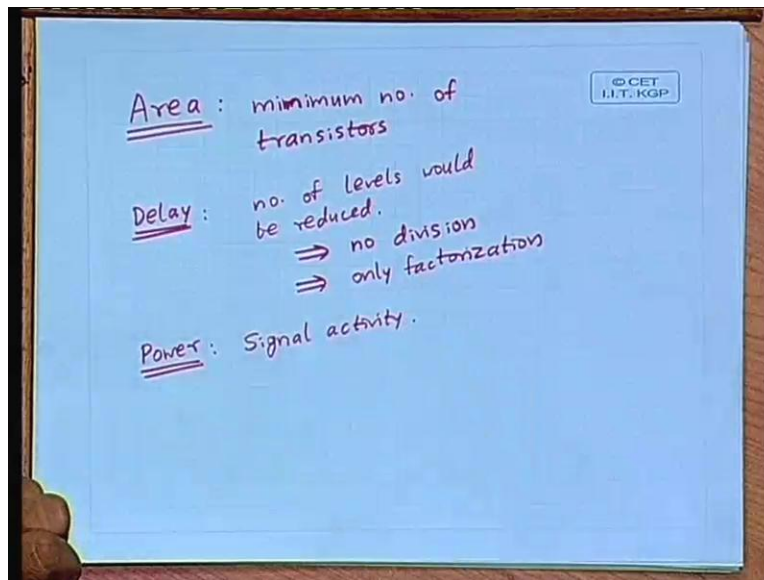


Like let us take another example. This example will highlight the use of common sub expressions. Suppose I have a function f equal to $a b c \bar{d} + a d g + a c \bar{d} f$. There is another function $g = a c \bar{d} + a d f$. Here again if you follow the method which has been used by Socrates something similar to that. We will see that there are common sub expressions. One such common sub expression you can identify is $a c \bar{d}$ $a c \bar{d}$ occurs here, here and here. It occurs in three places there is another sub expression $a d$ $a d$ also occurs in three places. So you can use any one of them. Suppose we decide to use $a c \bar{d}$.

So our modified equation will be u equal to $a c \bar{d}$ f will become $u b$ OR $a d g$ plus $u f$ g will become $u d$ plus $a d f$. If you count the number of literals now 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14. Here 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15. So you get 1 reduction 15, it has gone down to 14. So you therefore common sub expression, the gain may not be as significant as factoring. So factoring is something which you should always try to do when you target a CMOS realization.

So even after doing this, you will be doing a further transformation using factoring. So u will be a common factor for f . You see that you have a common sub-expression u . So you take u common b plus f OR a d g . Similarly here you have d common in g you take d out u OR a f . So here the number of literals has come down to 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, so from 15 you could bring it down to 12. So in terms of the implementation, again in terms of the implementation, again from here from 30 transistors you have gone down to 24 transistors. So essentially whatever this does, is based on transformations like this. Now in the method of espresso there also the numbers of literals were used to estimate the cost of the implementation. But in espresso the target technology was not necessarily CMOS. That is why there is some inaccuracy in the estimate of the hardware. Because they were concentrating only on the gate level net list. I do a minimization optimizations, I get a gate level net list. But here since our target is well defined we are looking for CMOS implementations. So we have a very fair estimate of the number of transistors number of literals twice of that is equal to number of transistors. So that is one.

(Refer Slide Time: 47:13)



Now in the optimization, what ever we have talked about so far, we had primarily looked at the problem of area. Delay is related area we are minimizing number of transistors. Minimum number of transistors. Now if our objective were to minimize delay, then we would have

followed a different approach the number of levels would be reduced. See this is, this implies that we will not do any division. No division because when ever you do division you are actually adding one level of logic, right? If you look at the previous example when ever you did this division, you have to compute this u and output of u has to go to here. So there is another level of logic you are adding. So if delay is more importance, then you should not use division. You use only factorization. But when you greater consist power of course power optimization cannot be accessed so easily. So there somehow we will have to measure or estimate the amount of signal activity in the circuit.

The idea is that in a CMOS gate if the inputs do not change state. The output obviously is also not changing state and the quiescent power consumption of the gate is almost close to 0. But whenever the circuits switches means the output goes from 0 to 1, momentarily the pull up and pull down networks both are turned on. So there is a quiescent current flowing during switching. So this switching current of CMOS can become quite significant, particularly when it is run at a high frequency. So when you are targeting power optimization we have to ensure that the amount of switching in the circuits is reduced as far as possible. There are ways to achieve this. But they are slightly beyond the scope of this class. There are ways to estimate the signal transition activity in a circuit. Ways to control them by using control inputs by adding additional gates. You can reduce the amount of signal transition activities in a circuit.

So today we had looked at some of the basic problems of multilevel logic optimization and we had tried to get a feel that how this multilevel optimization tools work. There are some other approaches which is also used for practice for the purpose of representation as well as manipulation and synthesis of Boolean functions. In the next class we will be looking at one such method, it is based on something called BDD's binary decision diagrams. That is a very efficient data structure to represent a function and we will show that using BDD, we can even carry out synthesis. We can synthesize a circuit. We can obtain some kind of net list starting from a given Boolean specification. And subsequent to that would be talking looking at the problem of high level synthesis were from a given higher level description. We can arrive at a register transfer level net list from that discussion. We shall start for our next lecture. Thank you.