

C-Based VLSI Design
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

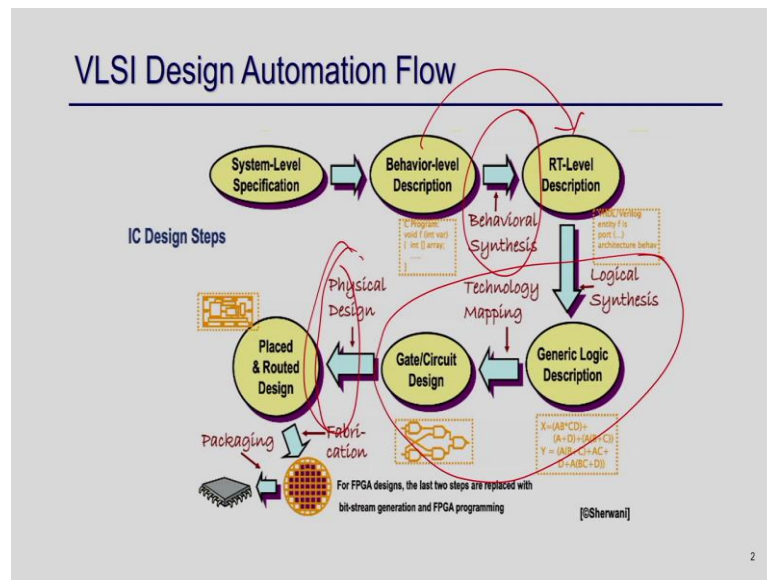
Module - 12
Recent Advances in C-Based VLSI Design
Lecture - 39
Introduction to Logic Synthesis

Welcome everyone to my class. So, last 11 weeks we have learnt about C-based VLSI design. So, specifically, all the details of converting C or C plus plus high-level code into RTL design.

So, in this week the way we plan the week is, something we will see the other levels of this electronic design automation process, which is basically logic synthesis and physical synthesis. So, we will just see a very high-level overview of these two steps and then at the we will conclude this course with some recent advances in high level synthesis with this thing.

So, let us move on, so in today we are going to discuss about Logic Synthesis in very high level.

(Refer Slide Time: 01:31)



So, let us lo back to that VLSI design automation flow that we are already familiar with, because I have shown this particular diagram many times. So, basically, so far, we talked about this high-level synthesis part where basically you convert a behavioral description into RTL. So, we have seen all the detail about the steps.

So, next step is basically this logic synthesis, this is primary logic synthesis and this is basically this physical synthesis. So, what happens in logic synthesis? You have the register transfer level behavior. So, what is register transfer level behavior? Here you express your behavior in terms of transfer of register values.

So, you express your every clock what are the registers get updated by which registers. So, that something is kind of in high level register transferable design. And then what we have to do in finally, we have to go in more detail level. So, all this automation does is basically you bring the things in one level of higher abstraction, but we have to finally, reach to the transistor level design.

So, to do that the next step is the logic synthesis what it does is basically convert this register transfer level design into gate level design. And then we have a physical synthesis where we convert that gate level design into transistor level design with a proper layout. And then finally, you do the fabrication packaging and we IC comes into market, so this is the overall process.

So, let us go into this logic synthesis steps it has basically two sub steps, the first step is that basically you have this register transfer level design you have to represent that in terms of gate level design. So, options are there I will come into that.

And then finally, once that particular generic gate level design is realized then finally, we have to represent them in terms of the technology mapping, in terms of the technology cells. So, if you look into this part this is basically technology dependent at the target dependent.

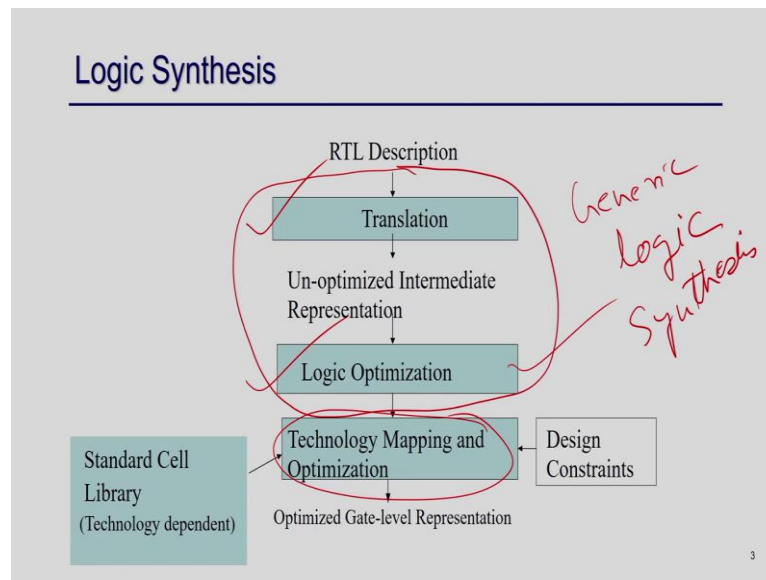
So, if you try to map into ASIC, then what is going to happen? You have a set of cell libraries you have to realize all those gates in terms of only using the cells that is available in that particular cell library. On the other hand, if you try to map that particular gate level design into FPGA board. So, you are mostly aware that the FPGA board has various fixed architecture you have this LUTs you have RAM you have DSPs.

So, effectively you have to map this basically this gate level design into those resource that are available in the particular FPGA board. So, this technology mapping actually is different for ASIC targets and this FPGA target. But this logic synthesis where we actually realize this my circuit in terms of gates is something is very generic approach, which is basically kind of common for both.

So, from this technology mapping things actually bifurcate for ASIC target and FPGA target. Because the physical synthesis is for ASIC is completely different because you have nothing no target architecture you have to realize your whole circuit in a specific area. And on the other hand, in a FPGA, you have a fixed architecture you essentially need to map your logics into those resource available in the architecture.

So, this two needs to bifurcate in this particular I mean when you target ASIC or FPGA. So, let us now move on to this logic synthesis.

(Refer Slide Time: 05:01)



So, as I mentioned this logic synthesis has basically two primary steps the first thing is basically this is the logic synthesis steps. The logic synthesis steps have two sub steps, the first is the translation. So, you have everything in the register transfer level you have to translate them into gate level first.

So, this is kind of logic translation and once you have that logic translated gate level design it is highly unoptimized. So, you need to optimize it design. So, then you have to do logic optimization. So, the generic logic synthesis has two sub steps, logic translation and logic optimization.

This is basically generic logic synthesis, and then will go for the technology mapping, that is a second step of this logic synthesis. So, I am going to do this, I am going to understand about give a very brief high-level overview about this logic translation, logic optimization and technology mapping all these three sub steps I am going to explain in very high level.

(Refer Slide Time: 06:10)

Logic Translation

```
Reg [31:0] a, b, c;  
Always@(posedge clk)  
a = b + c;
```

What kind of adder to choose?

- Full adder
- Carry look ahead adder
- Carry save adder

IIT Guwahati

So, let us move on to the logic translation. So, the objective of this particular class is not to give you all the algorithm behind this logic translations and all these methods rather what is the objective that is going to first here, what is the problem here and how they are actually getting tackled in the automation tool. So, let us first try to take an example.

So, in a register transfer level code we always get some kind of low code like this that you have some registers a b c and you do always at the rate posedge of clock you do a equal to b plus c. Basically you do an addition, so this is basically addition.

So, once you read this particular part of the code, what you have to do? You have to realize this you understand that this particular operation is happening in some every positive as a clock. So, I need to replace represent these particular operations in terms of adder in the RTL. So, then the question that will come, so what are the problem it has to solve during logic translation that I am try to highlight with this example.

The first question is that, first you have to identify this RTL you have to identify what is the things you want to implement. And you when you scan this particular code you understand there is an adder and you have to implement that adder. Now, the question comes into your mind, what kind of adder I am going to implement? Is it a full adder? It is a carry lo ahead adder, you say carry save adder or there are many other adders are available, so many advanced available.

So, this is something is a full adder where the kind of delay is more So, if there is n bit adder, basically each full adder adds one bit of each number a and b and then its carry. So, it is basically the kind of the delay of the circuit is order of n because the delay is getting carry forwarded. So, basically it is not so efficient design.

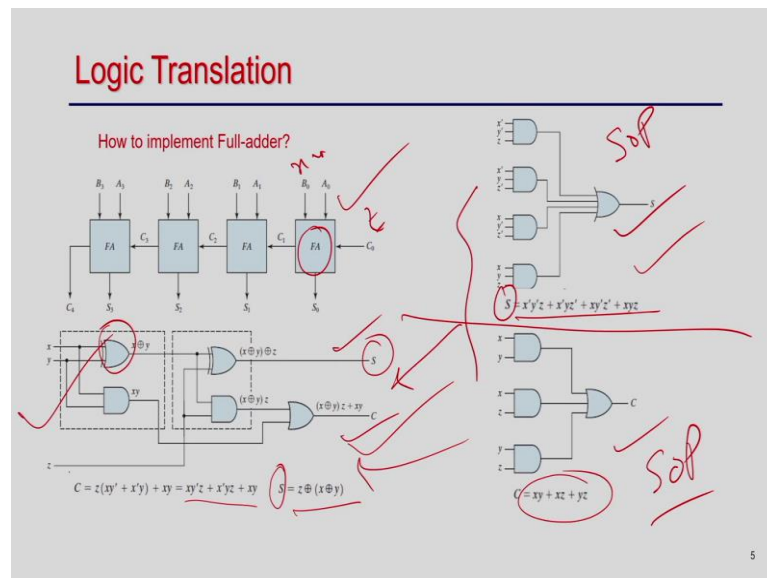
And on the other hand, if you to a carry lo ahead adder you have very fixed number of or constant number of logic delay, which is basically much faster. But it so this something

is there, but and then you have carry save adder and many other. So, based on your design constraint based on your target say your area target or performance target or latency target you have to choose one of them.

So, the tool automatically based on the constraint decides one of this architecture that I am going to add as let us say we decided that I mean even if I implement this adder by full adder, it is fine. It is basically meeting my design constant. So, this is something is the decision, so this logic translation actually has to take.

So, the first step is to identify a particular operation in the RTL and then it has to take a decision the kind of implementation that is going to have in based on the design constraint. So, for sake of simplicity that you decide to go ahead with the full adder. The next question that we will come to the logic translation is that how I am going to implement this full adder.

(Refer Slide Time: 09:26)



So, I have decided to go ahead with the full adder, but the circuit internal circuit of this full adder can be different. So, it is basically doing an addition, so if you just write the truth table of this your S is nothing but is this. So, basically if you think about this is x, y and this is z if you just think about this way.

So, basically this S is nothing but this circuit. And if you just do a kind of two-level representation, sum of product representation, this is the sum circuit. And the carry circuit is nothing but this and this is also again sum of product representation.

So, I can realize my single 1-bit full adder in basically using this, but you all know that this is not the optimized design, the optimized design is basically this. And we can actually do kind of some optimizations or we can do kind of manipulation or this some kind of Boolean expression simplifications or some rules you can apply.

And finally, we can actually identify that S is nothing but XOR of these three gates. So, this is what is your S and then C is nothing but this. And you can actually combine these two circuit into one where you actually share some of the resource.

So, basically you can understand the problem here is that you have two different output sum and carry their inputs are common. They are taking two bits of the number and the carry input and if you give a sum input and carry output. So, since the inputs are common one thing is that you can actually realize them completely independent circuit this is the sum circuit and this is the (Refer Time: 11:00) circuit.

But since they are actually having some common functionality in this particular design this is the efficient implementation of the full adder where we can actually share the resource. For example, you can see here this XOR is getting used for sum as well as to compute the carry.

So, that is the beauty of this, so basically the logic translation has to take decisions. So, how I am going to do that? I am going to do this or that or specifically this will come during the logic optimization that basically represent this adder in some Boolean expression.

So, and then it kind of do some optimization to realize that I can actually do this kind of sharing the resource that is needed for computing the sum and the carry. And finally, I can come up with this kind of optimized gate level representation.

So, you can understand that just realizing adder it has to first identify it is an add operation. Then it has to take a decision based on the constant whether what kind of architecture I am going to do. And then once that architecture is decided it will basically represent that architecture in terms of the Boolean expressions. And then it will do kind of optimization to identify this is the kind of the most efficient implementation of that full adder, this is what this logic translation does.

(Refer Slide Time: 12:14)

Logic Translation

- Identify all RTL blocks/operations and translate them into gate level representation
 - Add, Sub, Mult, Div – all arithmetic operations
 - MUX/DEMUX – all interconnections units
 - Decoder
 - Encoder
 - Comperator
 - ...

6

So, if you just go into this RTL the RTL has many components identifying the adding, subtraction, multiplication, division is kind of easy and you have to implement them. But identifying this MUX, DEMUX, decoder, because in the RTL you usually do not have a block that this is a MUX.

Probably, you have a switch case statement if you have a if else statement and you have to understand this switch case is nothing but a multiplexer, it will be realized as a multiplexer. Probably some from some structure you probably you have to understand this is nothing but a decoder, or something is encoder or it is a comparator.

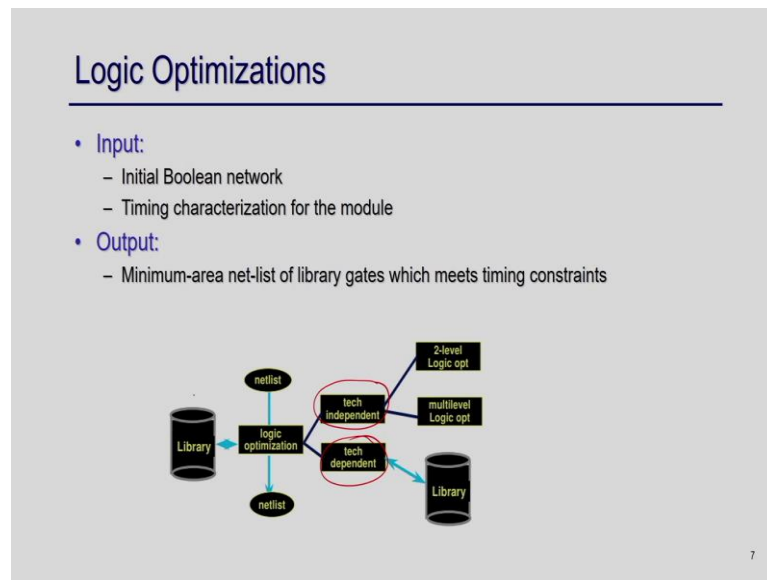
So, basically from the RTL this logic translator has to understand this is basically nothing but this. And then it will identify which target how which architecture it will be considered even for MUX you can have different kind of architecture.

So, and then finally, based on your design constraint it will decide I am going to go by this architecture. And once that particular architecture is also decided then you can actually do kind of optimization to get an optimized gate level representation of this block.

This is how this whole logic translation happens. So, I just try to write it in one sentence that you identify all RTL block of operations and translate them into gate level representation that is what happened in the logic translation. So, we understand that.

So, we this way we can actually represent everything in the gate level circuit. Now, the next step comes logic optimization.

(Refer Slide Time: 13:44)



So, in the logic optimization, why we need to do logic optimization? Because here we do kind of individual block, so we have this we take an individual block and we take a kind of their representation. So, usually they are this all this block we represent in the gate level circuit as a Boolean expression.

So, then there may be many redundant parts in the circuit, there may be many things we can be shared. Specifically, when you have multiple output, which has kind of some same input so, there will be some sub circuit which can be shared among these two cones of outputs, so which is something is the optimization.

So, the next step is basically do the optimizations and this optimization happen in two times one is basically the tech independent that you realize your RTL circuit in terms of gate, you apply the logic optimization you get an optimized gate level design. And then you map this your logic circuit or the gate level circuit into technology cells whether it is a FPGA or ASIC and then again you optimize. So, tech dependent optimization also there so, but usually this logic optimization indicates the tech independent logic optimizations.

So, here what is that logic optimizations input you have kind of a Boolean network which we can which is represented by many forms I will come to that. And you try to map them is basically you try to optimize that design in terms of the number of gates we can assume that.

(Refer Slide Time: 15:24)

Representations of Boolean Functions

- **Forms to represent Boolean Functions**
 - Truth table
 - List of cubes: Sum of Products, Disjunctive Normal Form (DNF)
 - List of conjuncts: Product of Sums, Conjunctive Normal Form (CNF)
 - Binary Decision Tree, Binary Decision Diagram
 - Boolean formula
 - Boolean network
 - Cube form

8

So, the to do that you as I mentioned that you need some kind of representation of the circuit. So, you need to represent your circuit combinational circuit or which is kind of nothing but a Boolean function somehow. So, and we are aware that there are many kinds of representations available one is truth table which we learnt in our undergraduate class.

We can have this sum of product or product of sum kind of presentation which is basically list of cubes or conjunction. And also, we have different kind of other representation like binary decision diagram; BDD, Boolean formula, Boolean network and say hyper cube form. So, all are basically representing the same thing based on your algorithm you can actually take some form.

(Refer Slide Time: 16:11)

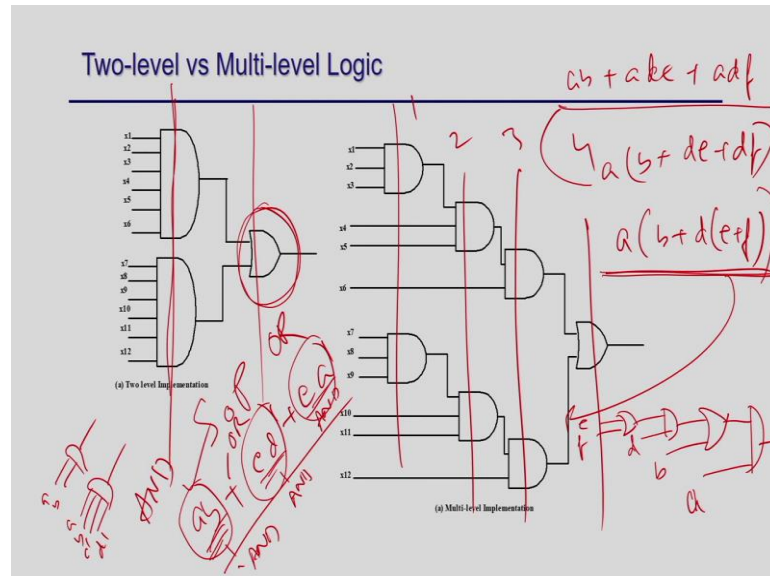
Logic Optimization

- **Two-level Optimization**
 - Karnaugh map
 - Quine-McCluskey Method
 - Heuristic based approach
 - ESPRESSO
- **Multi-level Optimization**
 - Factoring
 - Decomposition
 - Extraction
 - Substitution
 - Elimination

9

And once you take this logic optimization there are two kinds of optimizations involved here, one is two-level optimizations and the other one is basically multi-level optimization. So, what is two-level optimizations and what is multi-level optimization let us try to understand.

(Refer Slide Time: 16:35)



So, in the two-level optimizations we usually represent our Boolean expression in sum of product or product of sum form. So, basically you have two-level of gates. So, if it is a sum of product form this is the AND plane and this is the OR plane to give an example say, $ab + \bar{c}d + e\bar{a}$ and so on. So, this is the product term which can be basically can be represented by AND gates and this is the OR gates. So, this is basically the AND plane where all this product term will be represented AND the sum will be the OR. So, this is what the two-level representation.

So, here the cost is basically the number of product term which is actually called minterm. So, number of product term is there, so those many AND gate is also there and the size of the AND gate, so the AND gate how many pins are there. So, one 2 input AND gate and one 7 input AND gate are not same, the area will be different.

So, the number of product term is also important how many literals are there in each product term that is also important, because that determine the size. For example, if it is $abcd$; that means, I need a AND gate of 4 input. So, if input is 4 then it is a 4 input AND gate say a, b, \bar{c}, d .

So, basically the number of product term is also important and so, each product term is one AND gate. And thus, number of literals in each product term is also important. And then if you see here that number of product term also determines what is the size of the OR gate.

So, if there are 3 product terms; that means, I need a 3 input OR gate if I have say, 10 product terms, I need 10 input OR gate. So, the basically you can understand that for each literal we need kind of some gates. So, if you just reduce the number of literals in this whole expression. So, literally in the sense a , \bar{a} , b , \bar{b} all these are literals.

So, if you just reduce that it will eventually reduce the overall circuit. So, that means, given a Boolean expression in the two-level optimization; what you try to? You try to find out a minimum Boolean expression which has minimal number of literal. So, which is essentially nothing but your optimized design, I will come to that how we will do that.

On the multi-level logic, it is not two-level, so I do not need to represent this as a sum of product or product of some rather it is basically can have a level multiple levels of inputs. So, the same circuit here I just represent in 4 levels it is a two-level it is 4 levels.

So, usually when we write something in a factored form say like a into b plus d into e plus f . So, this when you write an expression like this is not a something sum of product form. So, it is basically mixed expression and if you try to represent this network directly you will basically have multiple levels. So, basically you have a OR gate.

Let me try to understand where I am going to do e and f . Then I have a AND gate where I am going to do with d this, I am going to do a OR gate with b and then I am going to do a AND gate where I am going to give a . So, you can understand that just to when you have this expression in a factored form then when you try to realize we have to realize in a multi-level representation.

So, two-level representation is easy to implement and there are lot of algorithms and is well studied thing. So, sometime we represent the things in as to a sum of product form and we actually apply two-level optimizations because finally, at the end of the day if you just reduce the number of literals that is something will reduce your circuit size.

But sometime you do not convert this into sum of product because any Boolean formula can be represented as a sum of product form, this is a normalized representation. So, even if this I can represent in terms of sum of product form. So, something like this, d plus df plus b this is nothing but this into a .

So, this is I can represent ab plus ade plus adf . So, this is the sum of product representation. So, given a Boolean expression I mean I can always convert into sum of product form or product of sum form. And then I can apply this two-level optimization or I can keep the expression as it is and then I will try to do multi-level logic optimizations, so I will come to into that.

So, in multi-level the primary advantage is that you can actually optimize across functions. So, if you have function 1 which is the expression is this, I have function 2, which are some expressions I can actually identify the common sub-expression here and then I can actually optimize across functions, which is not something happen for two-level.


Because two-level it is basically taken one output bit and it take the expression and try to optimize it. So, this both has applications, but both are getting used in logic optimizations. So, let me just go ahead with this method. So, for two-level optimizations we are all familiar with the what is Karnaugh map what is Quine McCloskey method and there is also Heuristic based approach called ESPRESSO.

So, basically and for multi-level optimizations techniques like factoring, decomposition, extraction, substitution, elimination this kind of techniques getting used. So, let me just move on and just give a very high-level idea of both the technique.

(Refer Slide Time: 22:24)

Two Level Combinational Logic Optimization

- Given
 - A 2-level Boolean function in Sum of Product Form (SoP)
- Objectives:
 - Minimize number of product terms (AND Gates)
 - Minimize number of input to the gates
- Approach
 - Identify essential prime implicants. Select them
 - In addition, select minimum number of prime implicants so that all minterms are covered.



11

So, in two-level optimization as I mentioned that your essential objective is to minimize number of AND gates and or AND gate or AND gate size eventually the number of literals, very high level.

So, what is the approach usually happen? So, here the idea is that you identify the implicants. So, you I mean you probably saw all aware that what is kind of implicant what is prime implicant and what is essential prime implicant. It is basically nothing but you have some of the terms which is 1.

(Refer Slide Time: 22:53)

Essential Prime Implicants

• Example: $F = \Sigma(0, 1, 2, 6, 8, 9, 10, 14)$

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

• Essential implicants :

- Implicant 1 = $\Sigma(0, 1, 8, 9)$
- Implicant 2 = $\Sigma(2, 6, 14, 10)$

• Non-essential implicant: $\Sigma(0, 2, 8, 10)$

12

So, if you just take this function is a sum of product representation and if you just assume the variables are basically ab and cd you can represent this as a product term. So, this is nothing but a bar b bar c bar and d bar this is a bar b bar c bar d and so on.

So, this is how we represent as a product and if you just realize it since how many it terms are here 8 product terms are there. So, and each of them is basically 4 literals. So, I need 8 AND gate each of them basically has 4 inputs. So, basically number of literals is basically 32, so there are 32 literals are there.

And you need basically 8 AND gate and 1 OR gate of size 8, this is how I can realize this circuit without any optimization. Then what usually we do? We try to identify a cover a basically implicant, the basic idea in high level that if you take two consecutive objects, I mean two literals. So, these are all 1 1 minterm.

So, if you take two consecutive minterm I can reduce one of the minterms from there, because they have one literal common. So, this is how I can actually identify implicant and then I can identify the prime implicant basically is the maximum kind of size of this minterm which is basically covering many minterms and which cannot be extended further. So, if you think about the individually these are all 1 1 minterms. So, I expand this to 2.

So, earlier there are 8, so each minterm has 4 literals. So, there are 8 literals when I combine them, they will turn into 3. So, from 8 it will become 3 literals. And then if I extend to another level, so this is now 4 you can assume that this is a folded structure. So, this is how the Karnaugh map realized, so then it will become 2.


So, from 8 literals it become 2 literal and 1 prime implicant. So, the idea is that you try to cover this your 1 term and sometimes there are the do not care terms also with

implicants, so that you can actually cover all this 1 with minimum number of such implicant.

(Refer Slide Time: 25:18)

Two Level Combinational Logic Optimization

- Given
 - A 2-level Boolean function in Sum of Product Form (SoP)
- Objectives:
 - Minimize number of product terms (AND Gates)
 - Minimize number of input to the gates
- Approach
 - Identify essential prime implicants. Select them
 - In addition, select minimum number of prime implicants so that all minterms are covered.



11

And the approach is basically I mean is very well known that you identify the essential prime implicant. That means, some prime implicant they are actually covering some of the 1, which is not covered by somebody else you need you must need this particular prime implicant.

So, you identify all the essential prime implicant and you have to select them because they are covering something which is not covered by anybody else. And in addition, you identify other prime implicants to cover all the 1s and do not cares of your circuits.

So, this is must and then you basically select minimum number of other implicant or prime implicant, so that all of your minterms are actually covered. And this is the overall approach and if you go into the Karnaugh map the example I was taking here. So, you basically find this is one essential prime implicant which is basically nothing but cd .

So, let me just try to understand, this is $\bar{a}\bar{b}$ and this is 00 , this is 01 this is 10 is 11 . So, basically here you can understand that, this is 10 this is 11 . So, this is \bar{b} and then this is 0001 and \bar{c} , so basically, we can understand that.

So, basically this is one essential prime implicant and this is one essential prime implicant. So, I can understand that there are 32 literals earlier and now you have only cd and $\bar{b}\bar{c}$, so this basically has 4 literals.

So, this is the optimization this Karnaugh map does. And if you go into this Quine McCloskey method tabular method it also does the same thing it identify all the essential prime implicant and then try to identify the other prime implicant to cover all the terms. This is the approach it does, but you can understand this is only four variables

(Refer Slide Time: 27:18)

Cover minimization

• Two-level minimization seeks a *minimum size cover* (least number of cubes). Reason: *minimize number of product terms in PLA*

Boolean Functions

13

But if you have say more variables this tabular method or Karnaugh map method is basically not so efficient.

(Refer Slide Time: 27:21)

The Boolean Space B^n

$B = \{0,1\}$, $B^2 = \{0,1\} \times \{0,1\} = \{00, 01, 10, 11\}$, etc.

Karnaugh Maps: Boolean Spaces:

Boolean Functions

14

So, for that and this method is not so scalable also you can understand that.

(Refer Slide Time: 27:34)

Two-level Logic Optimizations: Methods

- Find prime implicants and try to cover them using minimal number of covers
 - Karnaugh Map based method
 - Quine–McCluskey method
 - Heuristic based approach ✓
 - ESPRESSO
- ESPRESSO is based on simulated annealing method

15

So, for that usually the thing is happening is using a heuristic approach is called ESPRESSO. So, for that you can actually realize your n Boolean variable function as a n dimensional Boolean space. So, basically if you have a one variable it is nothing but this, if you have two variable your cube is this, if you have three variable it is a cube and if it is a four variable it looks like this and you can actually realize it this way.

So, the idea here is that once you have this kind of circuit, so each of this circle is representing one minterm. And in a given function only some of the minterms will be 1 not all then it the function become 1. So, and your idea is to cover all these 1 terms with minimum number of prime implicant, that is the idea.

So, for example, here you have this is the 3-dimensional space these red circles are the one terms and this is one down curved term. So, basically your idea is to identify this is the one essential prime implicant and this is one of the essential prime implicant and that will cover all the terms here. So, basically you need two kind of product term.

So, this is the overall idea, so somehow you have to cover all these on terms and do not care with minimum number of such prime implicants. And as I mentioned this Karnaugh map method Quine McCluskey method is not so scalable, because, they do not work well for large number of variables.

So, usually you go for go by heuristic approach which is ESPRESSO and this is nothing but a simulated annealing base method. Because and if you understand the simulated annealing method is like basically if you start from some solution space and if you try to optimize you try to reach to a global minimum.

If it is a convex space, you have always unique such thing you can always reach to that, but usually the solution space may be like this. And if you starting from this you try to optimize you might reach this point, but this is kind of local minimum. So, you may not

reach to the global minimum, because from this starting point if you try to optimize you try to always reduce the cost you may not never get this.

So, the what simulated (Refer Time: 29:44) annealing method does? Basically, it starts from any some random point and try to reach the minimum and this is the current base solution. Then again it starts from any other random points say starting from this random point.

Again, if you start from these random points it will reach to this point again, it will go to the same solution. But say it start from this point now eventually it will go to this global minimum. So, the simulated annealing method is like you start from one random point you try to reach the local minimum.

And then again you randomly start from other starting point and then try to get the minimum points. And if you keep doing this eventually you can hit up on this global minimum. And this algorithm is also kind of any time algorithm because every time you have a solution, if you just select a random point this is also solution.

And based on your time availability you can actually run the heuristic according to your requirement. And if you run say, for 5 second you might get this is a solution, if you run for say 1 minute probably you will get this solution, at that point that is the base solution. It will always keep track of what is the base current solution.

If you run for say for 10 minutes probably you can reach this global solution as well this is any time algorithm. And, but it is all basically a heuristic based approach and this ESPRESSO is very popular and specially widely used in the EDA tools for logic optimizations. So, I will give you the very brief idea about this ESPRESSO.

(Refer Slide Time: 31:15)

The slide is titled "ESPRESSO" in a large, bold, blue font. Below the title is a horizontal line. The main content is a bulleted list of three items, each with a blue dot and blue text. The first item is "Start from initial cover", with "initial cover" circled in red. Below it is a sub-bullet: "- May be the minterm itself". The second item is "Modify cover", with "Modify cover" circled in red. Below it are two sub-bullets: "- Make it prime and irredundant," and "- Reduce, expand, irredundant operations", with "Reduce, expand, irredundant operations" circled in red. The third item is "Stop when no further improvement is possible or timed out", with "no further improvement is possible or timed out" underlined in red. In the bottom right corner of the slide, there is a small number "16".

ESPRESSO

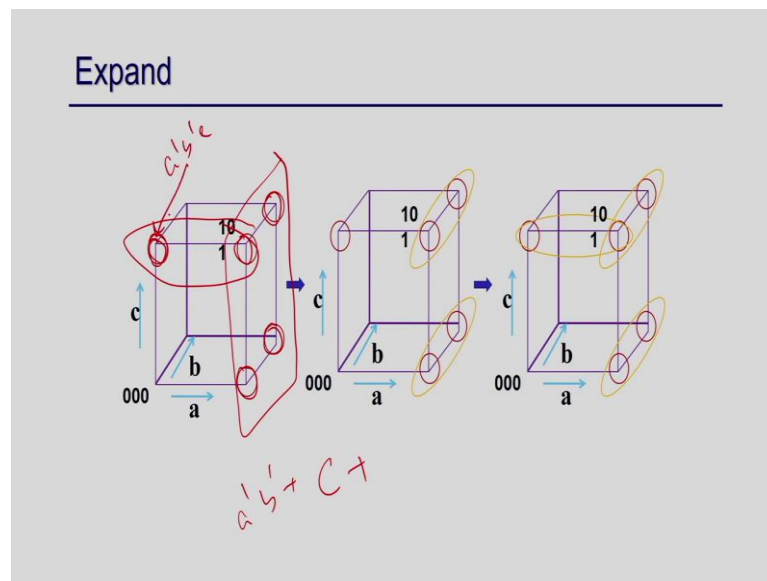
- Start from initial cover
 - May be the minterm itself
- Modify cover
 - Make it prime and irredundant,
 - Reduce, expand, irredundant operations
- Stop when no further improvement is possible or timed out

16

So, basically as I mentioned it will start with some initial cover of the terms, because what is the problem? You have set of 1 terms and some do not care and some cover, which is covers all the points.

And then you try to apply three operations which is called reduce, expand and irredundant. There are three operations just keep applying until it reaches some global point unless you find a minimum solution. Then it basically for further improvement it will basically start from another random point and I will do that.

(Refer Slide Time: 31:53)



So, let me try to explain this expand, reduce and irredundant operation. So, as I mention that it starts with some initial starting point and what can be the starting point. So, your cover may have all the individual minterms. So, suppose these are the points these are the minterms are there in your function you try to find out a cover which will cover all this minterm with minimum literal.

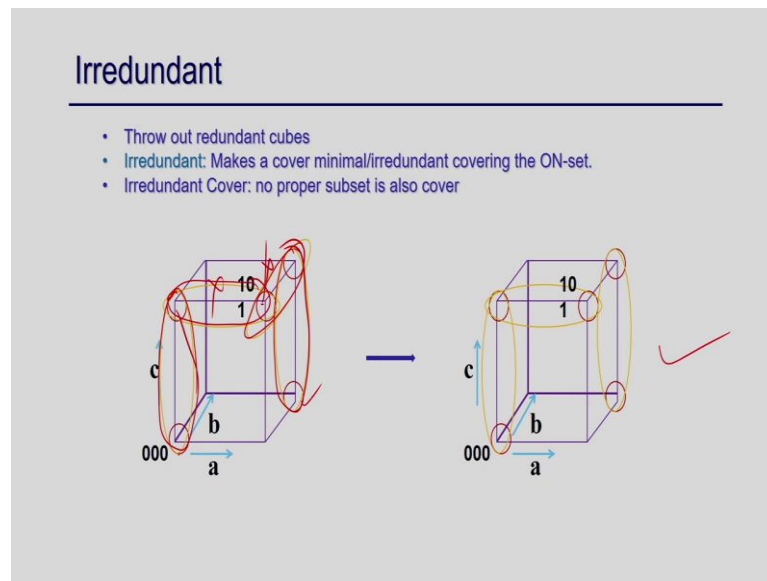
So, the solution here is basically is this is 1 and this is 1. So, you can understand this is basically c and this is nothing but a bar b bar, so this is basically the solution for this. And this individual literally basically represents nothing but a bar b bar and c this is in this term, and so on you can understand that.

So initially you can assume that my all minterms are the cover. So, initially I have 5 minterms and there are 15 literals because each minterms has 3 literals. Then what expand does? It starts from any available minterm and try to expand in any direction. So, if I try to expand this way because I will get this. So, this is the minterm where I have 2 literals (Refer Time: 33:08) now it would not try to go this way because this is not a 1 term, it will just expand this way.

So, the idea of expand is that randomly if you pick a solution it tries to expand in some direction. So, it will basically try to expand this way, then it will try to expand this way, it will take start from this point and again try to expand this way. So, now, I have this many minterms.

So, many minterms will be created which is basically one level smaller. Earlier, each minterm has 3 literals now I have every minterm has 2 literals, this is for the expand operation. And it will between done randomly it will pick any random minterms and try to expand, if expanding is possible, it will create the new terms that is all.

(Refer Slide Time: 33:50)

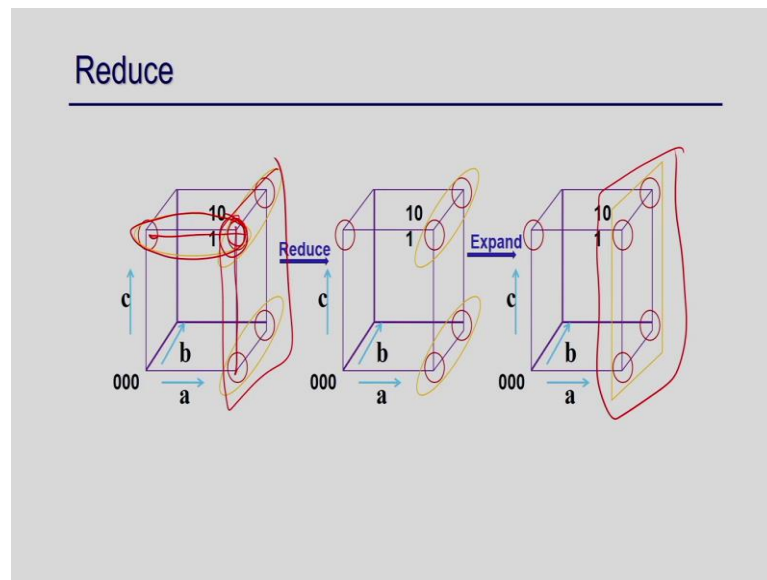


Then once you create that you can understand that you might create many redundant minterms. Because you might have some terms which is covered by more than one minterms and as a result you can actually remove some of them.

So, for example here, say suppose some expand happen and these are the kind of minterms you obtain, which is marked as the yellow line. And you can clearly understand this is one essential because it's covering this one, this is essential because this is covering this one which is not covered by anybody. This is also essential because this is covering this one which is not covered by anybody, but this is basically not essential.

So, I can remove this one or other way if you pick this one is as essential because this will pick this then I can remove this one which is the solution given here. So, after applying this expand you just remove some of the redundant cubes . So, that is what the idea of this irredundant.

(Refer Slide Time: 34:45)



And the reduce is the reverse operation of the expand because you expand some way and in that direction you may not be able to give a good solution. So, you reduce randomly and then you try to go in other direction.

So, for example, suppose if you started from this you reach this way you get this one. But if you probably come back and if you start from this point and if you try to go this way and probably you can actually cover the whole thing, that is what is the best solution.

So, the idea here is that at some point you basically reduce some of the minterm. So, if it is a n variable minterm it will become $n + 1$ now and then you try to expand in other directions. And it will actually try to give a better other solution and then you can actually probably get a better solution.

(Refer Slide Time: 35:37)

```
ESPRESSO

Forig = ON-set; /* vertices with expression TRUE */
R = OFF-set; /* vertices with expression FALSE */
D = DC-set; /* vertices with expression DC */
F = expand(Forig, R); /* expand cubes against OFF-set */
F = irredundant(F, D); /* remove redundant cubes */
do {
  do {
    F = reduce(F, D); /* shrink cubes against ON-set */
    F = expand(F, R);
    F = irredundant(F, D);
  } until cost is "stable";
  /* perturb solution */
  G = reduce_gasp(F, D); /* add cubes that can be reduced */
  G = expand_gasp(G, R); /* expand cubes that cover another */
  F = irredundant(F+G, D);
} until time is up;
ok = verify(F, Forig, D); /* check that result is correct */
```

So, this is the three operation it does if you just lo into the ESPRESSO this is the inner loop. So, it will find a local minimum you apply reduce you expand and you do irredundant, you keep doing till until your cost is stable that we know further improvement is possible. So, this will give you a local minimum, then what you do?

You pick a random point, so what you basically you are doing here? You just add some random cubes you expand some of the cubes randomly and then you make sure that you have an irredundant cover and then you go back and try to do the same process again. So, this will actually give you the global optimize points.

So, in general this ESPRESSO is something actually getting implemented no Karnaugh map no Quine McCluskey for two-level optimization usually in tool ESPRESSO gets implemented. So, I just give you the overall idea of this expression how it does the whole process.

So, that is all about these two-level expressions now let us move to the multi-level. So, why this two-level is not always sufficient? Because the first thing is that it has too restrictive as only two-levels and sometime and that does not give you the best representation. Sometime this SoP representation is actually impractical because it is become exponential.

(Refer Slide Time: 36:50)

Need for Multi-level Logic Optimization

- In some cases SoP representation is impractical
 - Example: Parity bit generator, multiplication

4-bit received message				Parity error check C_p
A	B	C	D	C_p
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

CP	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

Exponential number of implicants in SoP form for parity bit generator

16
8
32

The classical example is the parity bit, so if you suppose you have an odd parity generator. So, in odd parity it makes sure the number of 1 in your circuit is always say odd number if it is not then parity will become 1. So, for example, say if this is 0 0 0 0 and the parity is 0; that means, this is for even parity.

So, it is correct because this parity is even, but here you can see here this is 0 0 0 1 and this is 1 then the parity bit is 1. That means, this is parity check is wrong it is not basically even number of 1 is there. So, if this is something the parity bit generator it will just say whether your number of 1 in a particular number is basically even then it will say 0, if it is odd it will say 1.

And if you try to realize this parity bit generator into Boolean circuit, because you can see here there are four variables. And the bits where it is 1 if you just put it here, they are actually these circles you can see here no Karnaugh map can reduce anything. So basically, if there are 4-bit parity, 4 bit received message.

So, there are 2 to the power 4 there are 16 minterms and we have always half of them is basically even, half of them is odd. So, there will be 8 minterms is always present and each of them is 4 bits. So, always it will be 32-bit literals expression. So, basically you need always exponential number of implicants in SoP form.

So, in such cases sum of product is not something or a two-level representation is not something sufficient.

(Refer Slide Time: 38:40)

Need for Multi-level Logic Optimization

- **Advantage:**
 - Multi-level logic circuits require less area and delay compared to the corresponding two-level realizations and hence are more practical.
 - More than one functions can be optimized together.
- **Disadvantage:**
 - It is difficult to obtain provably optimal multi-level realizations because of the much larger design space available for exploration.

Phases Of Multi-level Logic Synthesis

- Technology Independent Synthesis
- Technology Dependent Synthesis

$$\begin{aligned} &= ab + ac + bc + dc \\ &= a(bc) + d(bc) \\ &= (bc)(a+d) \end{aligned}$$

So, on the other hand specifically for multiple function of multiple functions you cannot find out at the common sub-expression or the common part of two expression in two-level. So, that is why you usually multiple level optimizing is also very important. So, the best advantage is that it actually tries to give you a circuit which is need less area and delay compared to the two-level optimization. Because it can actually identify common sub-expression in a big expression, it just not a does not reduce the literals, but also can identify common sub-expression.

So, for example, say you have this a into b plus c and then say you have c into b plus c. So, if I just write this in SoP form. So, this is basically a b plus a e plus d b plus d c.

So, it is basically you can represent this way and then it is basically b plus c into a plus d. So, you can understand this is much simpler expressions in compared to this sum of product representation. So, usually if you just consider this multi-level representation you can actually identify this is the common sub-expressions this is the common sub-expressions.

I mean more realistically and also you can actually identify this as such kind of common sub-expression across function also. So, that is another big advantage, but this is something is a more difficult problem it is not so easy to solve. So, again you can do this multi-level synthesis for both technology dependent and independent, but I am just talking about for independent synthesis.

(Refer Slide Time: 40:28)

Objectives

- Reduces the number of literal in expression
 - No of transistors is twice of no of literals
- Identifying common sub-expression within an Boolean Expression (for single output) or between multiple Boolean expressions (for multiple outputs)

$$\begin{aligned}
 PEC &= \bar{A} \bar{B} (\bar{C} D + C \bar{D}) + \bar{A} B (\bar{C} \bar{D} + C D) + A B (\bar{C} D + C \bar{D}) + A \bar{B} (\bar{C} \bar{D} + C D) \\
 &= \bar{A} \bar{B} (C \oplus D) + \bar{A} B (\bar{C} \oplus \bar{D}) + A B (C \oplus D) + A \bar{B} (\bar{C} \oplus \bar{D}) \\
 &= (\bar{A} \bar{B} + A B) (C \oplus D) + (\bar{A} B + A \bar{B}) (\bar{C} \oplus \bar{D}) \\
 &= (A \oplus B) \oplus (C \oplus D)
 \end{aligned}$$

AB \ CP	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

So, now, take that parity bit example again and how does it help? so if I just write these expressions that all 1 this is my expression. So, these are all 1 if you just write this is Boolean expression and then you can actually find out the common sub-expression the example I just give in the previous slide.

So, this is a common sub-expression, this is a common sub-expression. So, I can combine them this is a common sub-expression and I can combine them and this is nothing but XOR and this is XNOR. So, I can actually rewrite this whole thing in terms of this I just identify these common expressions and I realize this is nothing but XOR I just replace this as XOR and this is by XNOR.

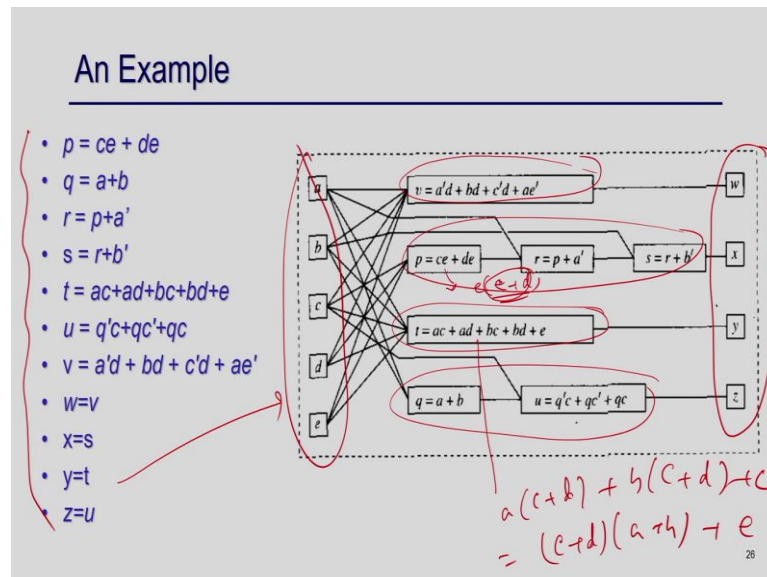
And then I identify this is actually common. So, I just combine this term with this term and I will get this and I can combine this term and this term I will get this. And again, I identify this is nothing but a XOR representation. Because you can see a bar b and this is a b bar. So, this is nothing but this.

So, I can actually I can identify common sub-expressions and I can realize these 32 literal things using just 4 literal and the circuit is this. So, this basically there are 3 XOR circuit. So, this is the optimal representation of this parity bit generator and this is possible when you actually take these whole expressions and you try to handle this multi-level logic optimizations which is basically the idea is that to identify the common sub-expression.

This is the most important optimizations you identify within the functions and then you can actually realize this kind of optimize representation which is not possible if you go for two-level optimizations. So, this is very good example to highlight this particular thing.

And then also you can take multiple expressions and again you can identify such common sub-expressions among multiple Boolean expressions and you can actually share that particular resource. So, here is a good example.

(Refer Slide Time: 42:35)



So, say suppose you have this circuit where these are the inputs and this is the output. And usually for this kind of multiple logic optimizations you actually draw a Boolean network. So, basically you can just think of it is a gate level representation sometime one node can represent a sub-expression as well.

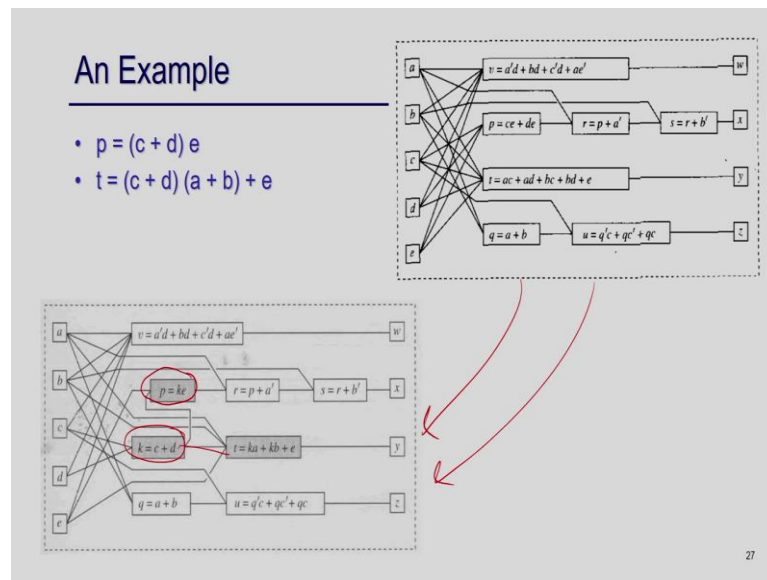
So, either it can be an individual gate or sometime you can combine some individual gate and I can find out some expressions and each node can represent a sub-expression as well. So, if you take this expression, it is equivalent representation is this, I am not going into detail how you get it, but you can always look into that.

But what is my objective there are 4 output I try to identify the common commonality and I try to share them here you can understand they are all individual. So, w is computed by this expression, x is computed by this expression, y is computing by these expressions you can understand there is a Boolean gate corresponding to that and this is this. So, my point here I try to understand is there any commonality between these two behaviors.

So, I can see here that here this is nothing but e into c plus d. So, this c plus d is something a sub-expression here and if you look into this, I can also see that basically a into c plus d plus b into c plus d plus e. So, it is nothing but c plus d into a plus b plus e.

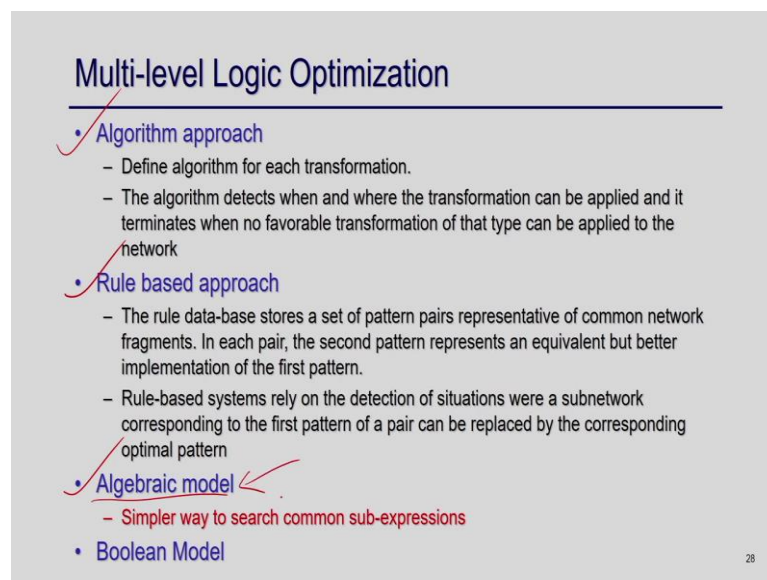
So, I can see here that this c plus d can be shared for x and y and I just identify this common sub-expression among these two and I can actually simplify this Boolean network.

(Refer Slide Time: 44:20)



This is what the example is given here that you identify this. So, this is now c plus d and I can utilize it to calculate p and I can utilize this. So, this is what is actually getting shared between these two and that simplifies the network. This is what to be done in the multi-level optimizations.

(Refer Slide Time: 44:40)



So, how it can be done? so basic idea is that you have big expression you identify the common sub-expression within the expression and also you identify the common sub-expression among two multiple expressions. The question is how we can do that?

So, there are three or say four approach algorithmic base you can actually have defines an algorithm where you can actually define certain transformation you can just try to see

that whether that kind of transformation is possible in the network. Or you can actually go for rule base you can define certain rules which is basically again kind of a you have a database of the rules and you just try to apply the rules in your circuit.

But the most way the common way to has been done is basically algebraic model. So, you can understand there is a common sub-expression. So, how we can identify the common sub-expression? First within the function and then across the function that something can be done very efficiently using algebraic model.

So, what I am going to do is I am just going to give very high-level idea of this algebraic model.

(Refer Slide Time: 45:43)

The Algebraic Model

- **Inputs:**
 - Set of SoP expression
- **Optimization Object:**
 - Extract CSE as much as possible.
- **Search for common divisors of two (or more) expressions.**

The diagram illustrates the algebraic model of division. It shows three examples of dividing a sum of products (SoP) expression by a common divisor (kernel).
 1. $(ab+ac) \div a = (b+c)$ with a remainder of 0.
 2. $(ab+ac+e) \div a = (b+c)$ with a remainder of e . The remainder e is labeled as the "Kernel".
 3. $(ab+ac) \div a = (b+c)$ with a remainder of 0.

So, in the algebraic model the idea here is that I am going to represent that expression in terms of Boolean expression, but I have considered that particular expression as an algebraic formula. So, it is algebraic formula and then the basic idea here is that you basically identify the common sub-expressions. So, I am going to apply the division operation, so a very nice idea.

So, for example, say you have this say the same example let me take again ac plus say e. So, if you this is the expression if I just say divide it by a, what I am going to get? I am going to get here b plus c I am going to get e here. So, this is my quotient, this is the dividend and this is the actual function, and this is my remainder.

So, the idea is that I am going to apply divide this expression using some minterms. So, this is the minterms here I am going to always take the minterms. So, minterms is basically a (Refer Time: 46:43) product term it is not any OR operator say a or abc say abc, bc these are all minterms.

So, I am going to always divide my actual Boolean expression as an algebraic formula with some minterm and I am going to get some quotient and some remainder. And this quotient is something I will talk about this is basically kernel if this is the minimum there is no sub-expression again.

So, if you get something say, you get ab plus ac you can get this also. So, ac , so this is not a kernel because it can be factored again. So, you can represent this as b plus c , but this is basically a kernel because in this particular expression no further division is possible this is the minimum expression then I am going to take about the kernel.

So, this is a common sub-expression. So, the basic idea is that you take a formula you keep dividing with some minterm you get some quotient. You check that quotient can be further simplified or further factored if it is not then this is a kernel. And these kernels are nothing but the common sub-expression.

And in the within the formula if you identify such common sub-expression multiple times you can combine them across the formula. If you get such common sub-expression, you can actually extract and you can share them that is the basic idea.

(Refer Slide Time: 48:12)

The Algebraic Method

- **Question:** How to find a kernel $k \in k(F)$?
- **Answer:** Algebraically divide F by one of its co-kernel C .
- **Kernels:** A cube-free quotient k obtained by algebraically dividing F by a single cube C (co-kernel)

$C = \text{Single cube, } abc, ad, efg$
 $F = CK + R$
 K kernel k if cube-free

What is cube-free ?

So, this is what I just explained there that you actually divide this with some single cube or the minterms and then you basically get a quotient. This quotient is basically kernel if it is a cube free; that means, further factorization is not possible.

And whenever you get the kernel, this is nothing but a common sub-expression. And you can actually identify such common sub-expressions and you can eliminate them.

(Refer Slide Time: 48:34)

Find all Kernels

- $f = abc + abd + bcd$
- Find all kernels

Divider cube d	$F = d.Q + k$	Is Q a kernel of F
1	$(abc + abd + bcd) + 0$	No, here cube b as factor.
a	$a(bc + bd) + bcd$	No. b is a factor
b	$b(ac + ad + cd) + 0$	Yes, kernel $(ac + ad + cd)$ is cube-free
ab	$ab(c + d) + bcd$	Yes, kernel $(c + d)$ is cube-free

So, again example say suppose I have a function given I say first I divide it by 1 and I will get this. So, this is not a cube free because it has a common term b then I divide this term with a, I will get this as a quotient. But this is not a kernel because you can simplify further with b. I can write this as b into c plus d, so this is not a kernel.

But if I divide this f by b and then I will get this. And this is kernel because this cannot be simplified further. Or if I divide this by ab I will get, so this is my ab, so this is my quotient and this is my remainder, this is a kernel because this cannot be simplified further.

So, for this formula there are two sub-expression one is this and one is this. So, this I identify for this formula and now I can identify the same thing for the other formula also.

(Refer Slide Time: 49:34)

Brayton and McMullen Theorem

- **Kernels: Why are they important ?**
 - If they are important, how do we actually compute them?
- **Brayton and McMullen Theorem:**
 - Expression 'F' and G have common divisor d if and only if,
 1. There are kernels $K_1 \in K(F)$, $K_2 \in K(G)$ such that $d = k_1 \cap K_2$ (i.e. SOP form with common cubes exist)
 2. d is an expression with at least 2 cube in it.

Interpretation:

- To find common divisor of two expressions, the only place to look for is in the intersection of kernels.
- This intersection of kernels is the divisor, there is no other.

So, I will identify those formula and then how this can be done is basically given by this Brayton and McMullen Theorem. It is basically saying that you identify all such common sub-expression one formula you identify all the common sub-expression other formula. And the common sub-expression for the for both the formula it will be in the intersection of these two kernels.

So, the idea is very simple you take a formula you identify all the kernels; that means, all the common sub-expressions and then you take intersection of these two. And whatever the intersection has as common sub-expression that is something is the common sub-expression among multiple expression.

(Refer Slide Time: 50:10)

The Algebraic Method

1. Find kernels of F and G
2. Find kernels in intersections of $K(F)$ and $K(G)$
3. Extract multi-cube common divisor D
4. Rewrite F and G using D

The diagram illustrates the algebraic method for finding a common divisor. It shows two input boxes, F and G, each containing a set of kernels. For F, the kernels are K_1, K_2, K_3 . For G, the kernels are K_4, K_5, K_6, K_7 . These two sets of kernels are intersected, with the condition that the intersection must contain at least two cubes. The intersection results in a set of kernels K_1, K_6 , which is labeled 'yes'. From this intersection, a multi-cube common divisor D is extracted, represented by the expression $d = ab + c + \dots$. Finally, the original expressions F and G are rewritten using this divisor D to produce F_{new} and G_{new} .

So, this is the overall algebraic method that you take a formula you identify all the kernels which is basically the common sub-expressions using the division method. You take another formula you identify all the kernels then you do an intersection you identify the intersection whatever the formula is there you identify the multi cube common divisor.

Or, basically the common sub-expression here and that is the common sub-expressions and you can rewrite the formula again with the common sub-expression. Because that is the part I am going to share, this is the idea of doing the multi-level logic optimization using algebraic method.

(Refer Slide Time: 50:44)

Example

$F = ae + be + cde + ab \approx e \cdot F' + cde + ab$
 $G = ad + ae + be + bc \approx (a+b)$

K(F)	Co-kernal	K(G)	Co-kernal
$a + b + cd$	e	$a + b$	e
$b + c$	a	$d + e$	a
$a + e$	b	$d + e + c$	b
$ac + be + cde + ab$	1	$ab + ac + be + bc$	1

$(a + b + cd) \cap (a + b) = (a + b)$
 So, this is workable multi-cube divisor of F and G.

So, again I just if I take a two example say F these are the kernels and for G these are the kernels you can verify cross check and for that the divisor was this the co kernels. And so, among these two I can only see that if I take intersection only this and this has some common term the intersection has a plus b.

So, this a plus b is the common among these two. So, I can identify for these two formulae there is a common sub-expression which is nothing but a plus b. And I can rewrite this formula where a plus b is something a common term you can actually cross check.

So, basically this is nothing but a plus b see if I just find out this a plus b is the common sub-expression you can rewrite this as this that and say e into that sub-expression F dash that was the sub-expression which is a plus b this is F dash. And then you have c d e plus a b and here also you can see here this is nothing but a d plus a e into a plus b plus bc, so this is the common sub-expression.

(Refer Slide Time: 51:54)

Example

$\bullet F = ae + be + cde + ab \approx e \cdot F' \rightarrow eae + ab$
 $\bullet G = ad + ae + be + bc \approx a \cdot a^{-1} \otimes (a^{-1} + bc)$

K(F)	Co-kernal	K(G)	Co-kernal
$a + b + cd$	e	$a + b$	e
$b + c$	a	$d + e$	a
$a + e$	b	$d + e + c$	b
$ac + be + cde + ab$	1	$ab + ac + be + bc$	1

$(a + b + cd) \cap (a + b) = (a + b) = P'$
 So, this is workable **multi-cube** divisor of F and G.

(Refer Slide Time: 52:19)

Example

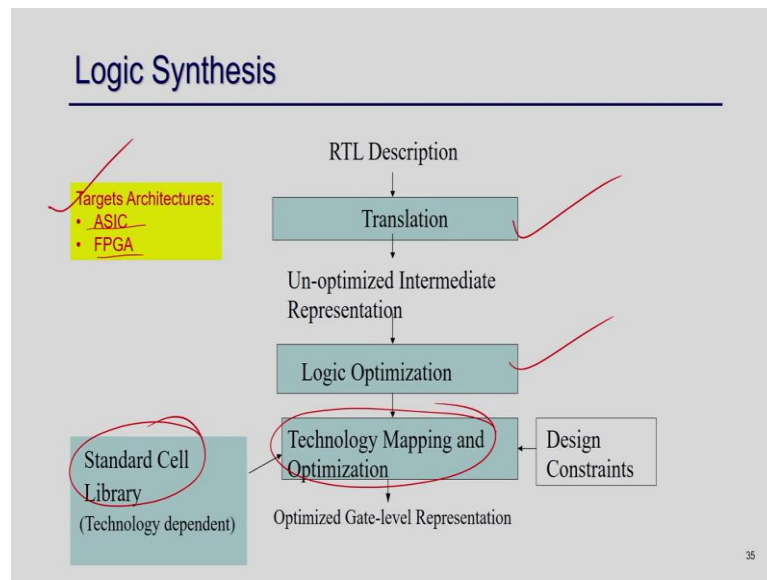
$\bullet F = ae + be + cde + ab \approx e \cdot F' \rightarrow eae + ab$
 $\bullet G = ad + ae + be + bc \approx a \cdot a^{-1} \otimes (a^{-1} + bc)$

K(F)	Co-kernal	K(G)	Co-kernal
$a + b + cd$	e	$a + b$	e
$b + c$	a	$d + e$	a
$a + e$	b	$d + e + c$	b
$ac + be + cde + ab$	1	$ab + ac + be + bc$	1

$(a + b + cd) \cap (a + b) = (a + b) = P'$
 So, this is workable **multi-cube** divisor of F and G.

So, I can replace this by basically e into F bar. So, this is the I can rewrite, so that is the overall idea of this algebraic method.

(Refer Slide Time: 52:26)



So, this is what it is done and it is something very important because certain cases two-level optimization is impractical. And it has many advantages this multi-level optimization because you can actually identify common sub-expression among multiple expression as well.

So, this is all about this logic translation and then logic optimization. The last phase as I mentioned, so this I have already discussed, this I have also discussed the next phase is basically technology mapping. And as I also may already mention that for technology mapping target architecture is important.

So, we have different two type of targets architecture one is ASIC one is FPGA. So, ASIC I just mention this application, so it basically does not have any target architecture. So, you have what is you have you have a trans standard cell library. So, the classic example is basically you have minimum number of cells here, which is a real complete set which can represent any gates.

(Refer Slide Time: 53:27)

Technology Mapping - ASIC

- ASIC
 - Standard cell library (NAND, NOR, etc)
 - Replace each gate with equivalent representation using cells from cell library

Truth Table

Input A	Input B	Output Q
0	0	1
0	1	1
1	0	1
1	1	0

Desired AND Gate
 $Q = A \text{ AND } B$

NAND Construction
 $Q = A \text{ AND } B = (A \text{ NAND } B) \text{ NAND } (A \text{ NAND } B)$

Desired NOT Gate
 $Q = \text{NOT } A$

NAND Construction
 $Q = \text{NOT } A = A \text{ NAND } A$

Desired OR Gate
 $Q = A \text{ OR } B = (A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)$

NAND Construction
 $Q = A \text{ OR } B = [(A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)] \text{ NAND } [(A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)]$

Finally optimize redundant the circuits again

36

So, for ASIC design the idea is like this you have a standard cell library usually it is only a NAND gate or say only NOR gate or you can have AND, OR and NOR gate which is the complete set basically. And you have in your circuit in the gate level circuit you have all possible gate XOR gate XNOR gate AND NAND NOR everything is there.

What you have to do? You have to basically represent the other gates in terms of the gates or the cell that is available in the cell library. So, for example, suppose your objective is to do the AND gate. So, this is the AND gate, but say suppose you are in the target line you have only the NAND gates.

So, I can represent AND by this you can cross check I am not going to detail. So, I can rewrite my AND gate using this my NOT gate using the NAND gate like this my OR gate using the NAND gate is like this. Say, NOR gate I can rewrite using only NAND gate like this and so on, so there are many other also.

So, this how I can actually rewrite my logic circuit only using the technology cell libraries. You can understand that if you just do that you have lot redundancy into your circuit again you have to do the optimizations, again you can apply this multi-level optimizations to make this circuit minimal.

(Refer Slide Time: 54:55)

FPGA Technology Mapping

- Technology Mapping:
- FPGA – fixed architecture
 - LUT
 - DSP
 - RAM/ROM
- In FPGA:
 - Gate level design → LUTs
 - Multiplier/MAC → DSP
 - Memory/Large Registers → RAM/ROM

37

So, if you consider the FPGA mapping as I already mention that it has fixed architecture it has specific resource like LUTs, DSPs and RAM, ROM. So, your objective is to map this gate level design into LUTs, if there is a multiplier or multiplier and accumulate kind of circuit it has to be map into DSPs, if you have a large memory you try to map into ROM.

So, again this is something different from ASIC target. And I am going to take one particular session where I am going to talk about how efficiently or how what are the steps what are the kind of algorithm involved how to map this LUT mapping, DSP mapping and RAM ROM mapping in FPGA technology.

So, this is how this whole logic synthesis works. And I just try to explain that the whole process in very high level and try to give the complexities and the kind of algorithm which is basically very popular or commonly getting used in the whole process without going into much detail. So, with this I conclude today's class.

Thank you.