

Logic Synthesis and Verification

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University



Fall 2011

1

Multi-Level Logic Minimization

Reading:

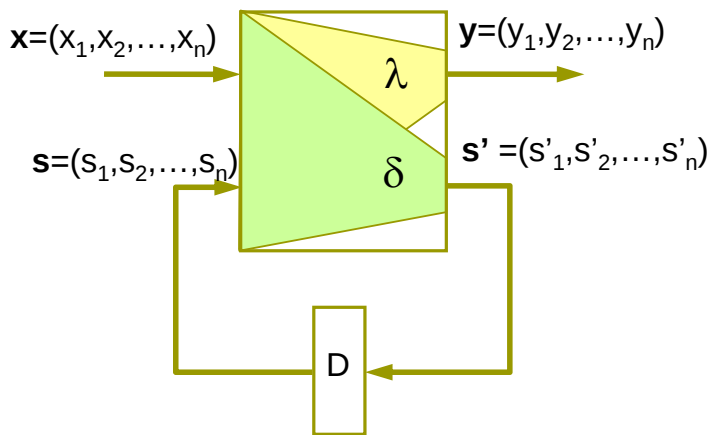
Logic Synthesis in a Nutshell

Section 3 (§3.3)

most of the following slides are by
courtesy of Andreas Kuehlmann

2

Finite State Machine



Finite-State Machine $F(Q, Q_0, X, Y, \delta, \lambda)$
where:

Q : Set of internal states

Q_0 : Set of initial states

X : Input alphabet

Y : Output alphabet

$\delta: X \times Q \rightarrow Q$ (next state function)

$\lambda: X \times Q \rightarrow Y$ (output function)

Delay element:

- Clocked: synchronous circuit
 - single-phase clock, multiple-phase clocks
- Not clocked: asynchronous circuit

3

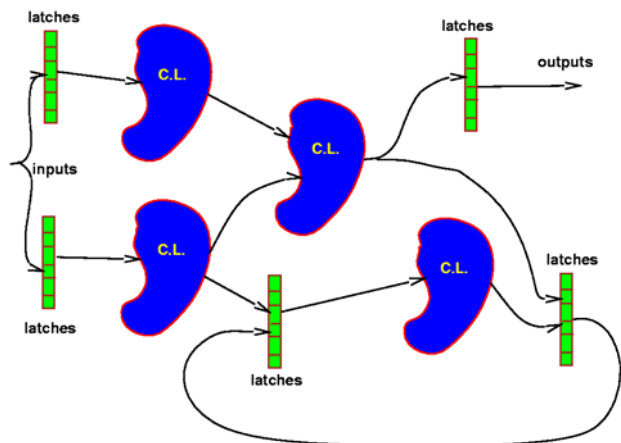
General Logic Structure

Combinational optimization

- keep latches/registers at current positions, keep their function
- optimize combinational logic in between

Sequential optimization

- change latch position/function



4

Optimization Criteria for Synthesis

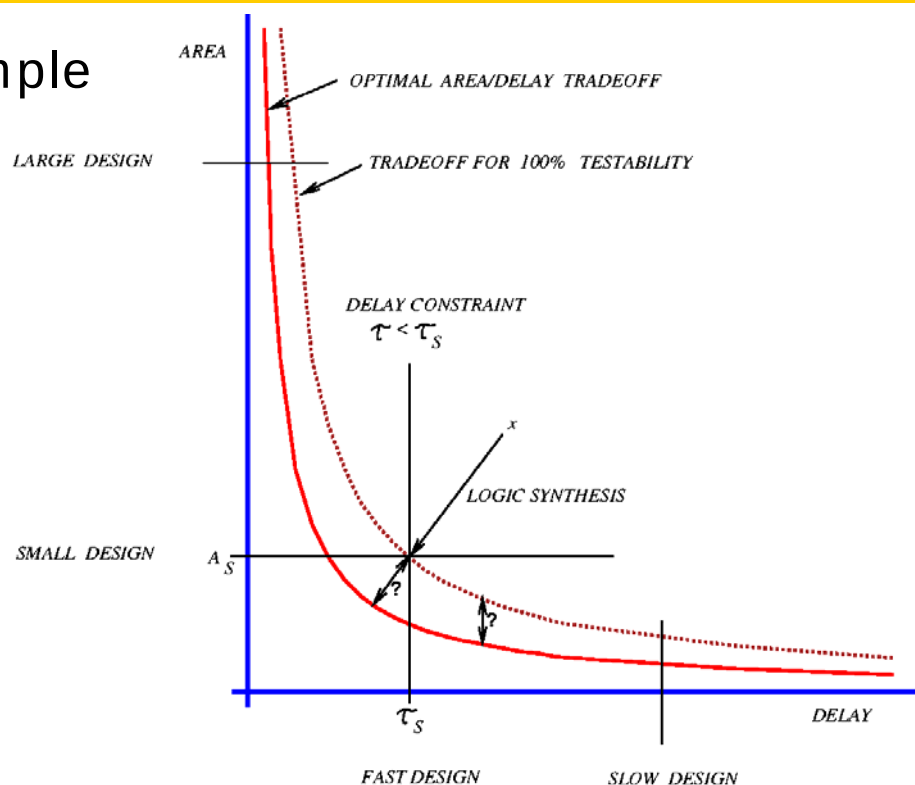
- The optimization criteria for multi-level logic is to *minimize* some function of:
 1. Area occupied by the logic gates and interconnect (approximated by literals = transistors in technology independent optimization)
 2. Critical path delay of the longest path through the logic
 3. Degree of testability of the circuit, measured in terms of the *percentage* of faults covered by a specified set of test vectors for an approximate fault model (e.g. single or multiple stuck-at faults)
 4. Power consumed by the logic gates
 5. Noise immunity
 6. Placeability, routability

while simultaneously satisfying upper or lower bound constraints placed on these physical quantities

5

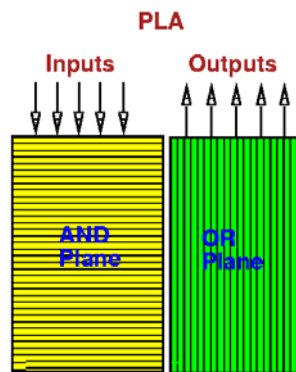
Area-Delay Trade-off

□ Example

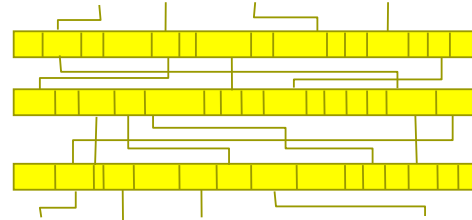


6

Two-Level (PLA) vs. Multi-Level



E.g. Standard Cell Layout



□ PLA

- Control logic
- Constrained layout
- Highly automatic
- Technology independent
- Multi-valued logic
- Input, output, state encoding
- Predictable

□ Multi-level logic

- Control logic, data path
- General layout
- Automatic
- Partially technology independent
- Some ideas of multi-valued logic
- Occasionally involving encoding
- Hard to predict

7

General Approaches to Synthesis

□ PLA synthesis:

- theory well understood
- predictable results in a top-down flow

□ Multi-level synthesis:

- optimization criteria very complex
 - except special cases, no general theory available
- greedy optimization approach
 - incrementally improve along various dimensions of the criteria
- works on common design representation (circuit or network representation)
 - attempt a change, accept if criteria improve, reject otherwise

8

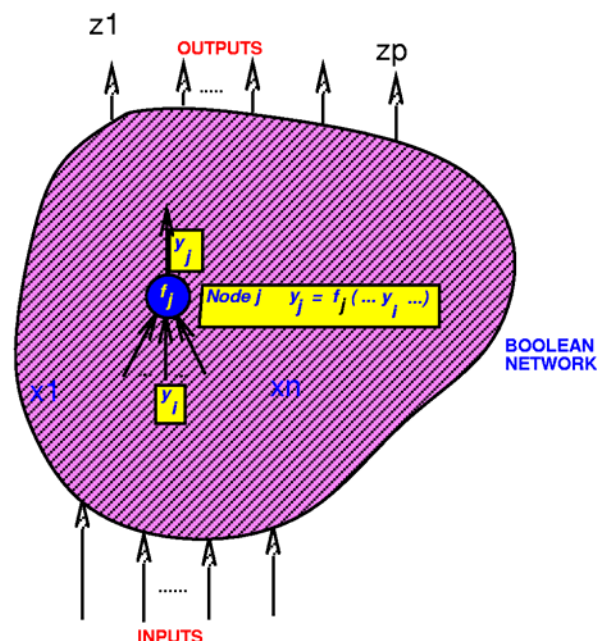
Transformation-based Synthesis

- All modern synthesis systems are transformation based
 - set of transformations that change network representation
 - work on uniform network representation
 - “script” of “scenario” that can orchestrate various transformations
- Transformations differ in:
 - the scope they are applied
 - Local vs. global restructuring
 - the domain they optimize
 - combinational vs. sequential
 - timing vs. area
 - technology independent vs. technology dependent
 - the underlying algorithms they use
 - BDD based, SAT based, structure based

9

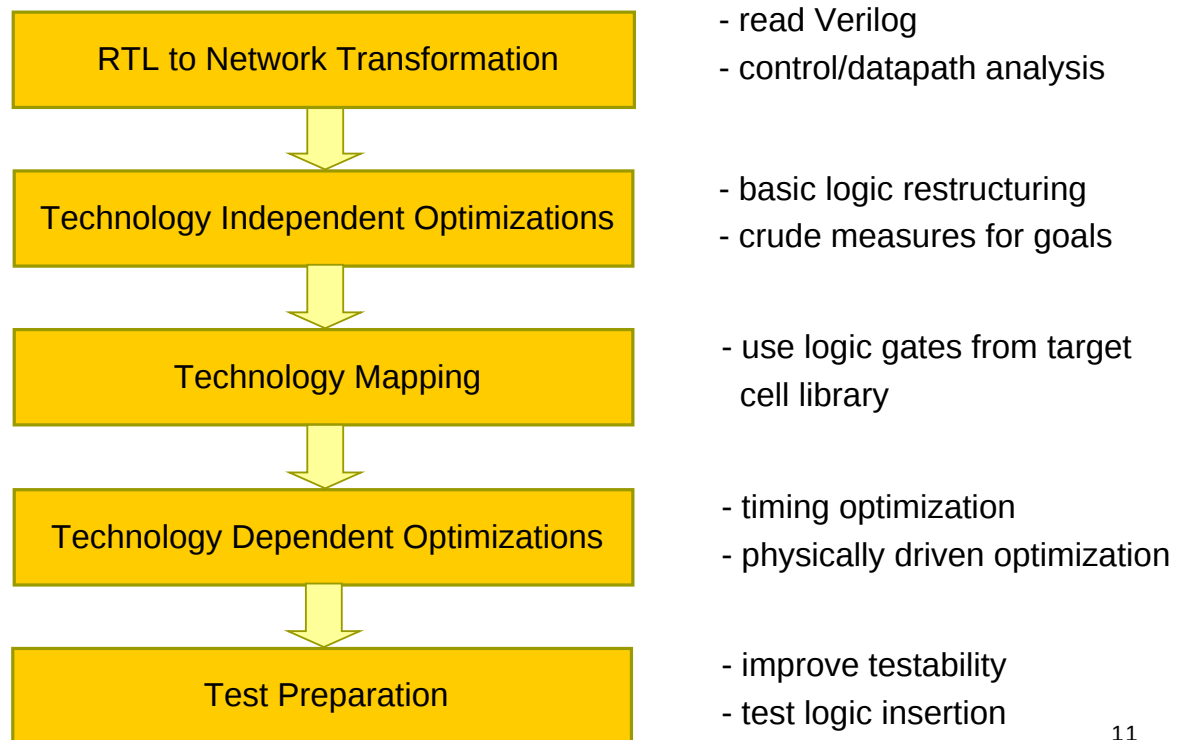
Network Representation

- Boolean network
 - Directed acyclic graph (DAG)
 - Node logic function representation $f_j(x,y)$
 - Node variable y_j : $y_j = f_j(x,y)$
 - Edge (i,j) if f_j depends explicitly on y_i
- Inputs: $x = (x_1, \dots, x_n)$
- Outputs: $z = (z_1, \dots, z_p)$
- External don't cares: $d_1(x), \dots, d_p(x)$ for outputs



10

Typical Synthesis Scenario



11

Local vs. Global Transformation

- Local transformations optimize one node's function in the network
 - smaller area
 - faster performance
 - map to a particular set of cells
- Global transformations restructure the entire network
 - merging nodes
 - splitting nodes
 - removing/changing connections between nodes
- Node representation:
 - keep size bounded to avoid blow-up of local transformations
 - SOP, POS
 - BDD
 - Factored forms

12

Sum-of-Products (SOP)

□ Example

$$abc' + a'bd + b'd' + b'e'f$$

□ Advantages:

- Easy to manipulate and minimize
- many algorithms available (e.g. AND, OR, TAUTOLOGY)
- two-level theory applies

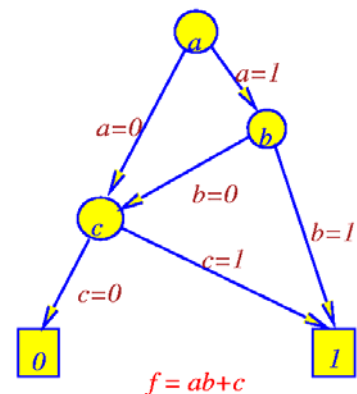
□ Disadvantages:

- Not representative of logic complexity
 - E.g., $f = ad + ae + bd + be + cd + ce$ and $f' = a'b'c' + d'e'$ differ in their implementation by an **inverter**
- Not easy to **estimate** logic; difficult to estimate **progress** during logic manipulation

13

Reduced Ordered BDD

- Represents both function and its **complement**, like **factored forms** to be discussed
- Like network of muxes, but restricted since controlled by **primary input** variables
 - **not really a good estimator** for implementation complexity
- Given an ordering, reduced BDD is **canonical**, hence a good replacement for truth tables
- For a good **ordering**, BDDs remain reasonably small for complicated functions (but not multipliers, for instance)
- **Manipulations** are well defined and efficient
- Only **true** support variables (dependency on primary input variables) are displayed



14

Factor Form

□ Example

$$(ad+b'c)(c+d'(e+ac'))+(d+e)fg$$

□ Advantages

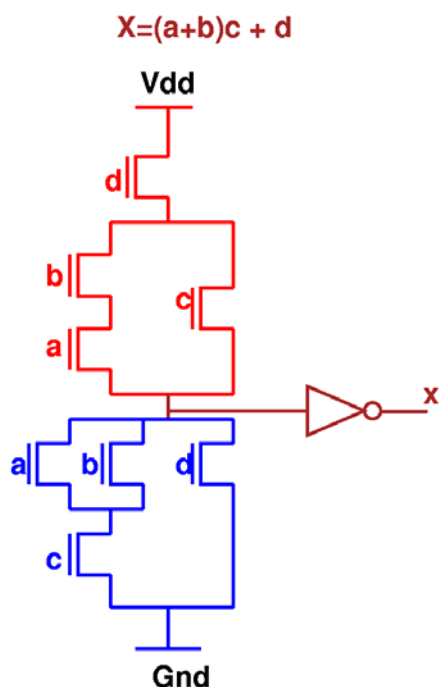
- good representative of logic **complexity**
 $f=ad+ae+bd+be+cd+ce$
 $f'=a'b'c'+d'e' \Rightarrow f=(a+b+c)(d+e)$
- in many designs (e.g. complex gate CMOS) the **implementation** of a function corresponds directly to its factored form
- good **estimator** of logic implementation complexity
- doesn't **blow up** easily

□ Disadvantages

- not as many algorithms available for **manipulation**
- usually **converted** into SOP before manipulation

15

Factor Form



Note:

literal count $\approx$ transistor count $\approx$ area

- however, area also depends on wiring, gate size, etc.
- therefore very crude measure

16

Factored Form

- **Definition:** f is an **algebraic expression** if f is a set of cubes (SOP), such that no single cube contains another (minimal with respect to single cube containment)

- **Example**

$a+ab$ is not an algebraic expression (factoring gives $a(1+b)$)

- **Definition:** The **product** of two expressions f and g is a set defined by $fg = \{cd \mid c \in f \text{ and } d \in g \text{ and } cd \neq 0\}$

- **Example**

$(a+b)(c+d+a') = ac+ad+bc+bd+a'b$

- **Definition:** fg is an **algebraic product** if f and g are algebraic expressions and have **disjoint** support (that is, they have no input variables in common)

- **Example**

$(a+b)(c+d) = ac+ad+bc+bd$ is an algebraic product

17

Factored Form

- **Definition:** A **factored form** can be defined recursively by the following rules. A factored form is either a product or sum where:

- a product is either a single **literal** or a **product** of factored forms

- a sum is either a single **literal** or a **sum** of factored forms

- A **factored form is a parenthesized algebraic expression**

- In effect a factored form is a **product of sums of products** or a **sum of products of sums**

- **Any** logic **function** can be represented by a factored form, and **any** factored form is a representation of some logic function

18

Factored Form

□ Example

- $x, y', abc', a+b'c, ((a'+b)cd+e)(a+b')+e'$ are factored forms
- $(a+b)'c$ is not a factored form since complement is not allowed, except on literals

□ Factored forms are not unique

- Three equivalent factored forms
 $ab+c(a+b), \quad bc+a(b+c), \quad ac+b(a+c)$

19

Factored Form

- Definition: The factorization value of an algebraic factorization $F=G_1G_2+R$ is defined to be

$$\text{fact_val}(F, G_2) = \text{lits}(F) - (\text{lits}(G_1) + \text{lits}(G_2) + \text{lits}(R)) \\ = (|G_1|-1) \text{lits}(G_2) + (|G_2|-1) \text{lits}(G_1)$$

- Assuming G_1, G_2 and R are algebraic expressions, where $|H|$ is the number of cubes in the SOP form of H

■ Example

$$F = ae+af+ag+bce+bcf+bcg+bde+ddf+bdg$$

can be expressed in the form $F = (a+b(c+d))(e+f+g)$, which requires 7 literals, rather than 24

- If $G_1=(a+bc+bd)$ and $G_2=(e+f+g)$, then $R=\emptyset$ and

$$\text{fact_val}(F, G_2) = 2 \times 3 + 2 \times 5 = 16$$

- The above factored form saves 17 literals, not 16. The extra literal comes from recursively applying the formula to the factored form of G_1 .

20

Factored Form

- Factored forms are more **compact** representations of logic functions than the traditional SOP forms

- Example:

$$(a+b)(c+d(e+f(g+h+i+j)))$$

when represented as an SOP form is

$$ac+ade+adfg+adfh+adfi+adfj+bc+bde+bdfg+bdfh+bdfi+bdfj$$

- SOP is a factored form, but it may not be a good factorization

21

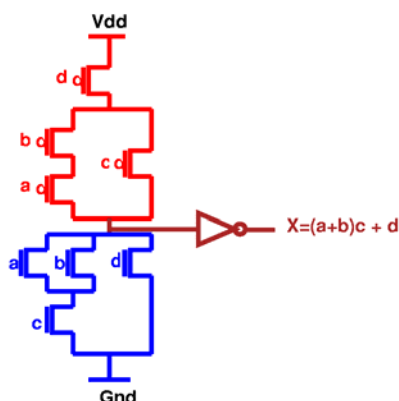
Factored Form

- There are functions whose size is **exponential** in SOP representation, but **polynomial** in factored form

- Example:

Achilles' heel function $\prod_{i=1}^{i=n/2} (x_{2i-1} + x_{2i})$

n literals in factored form and $(n/2) \times 2^{n/2}$ literals in SOP form



Factored forms are useful in **estimating** area and delay in a multi-level synthesis and optimization system. In many design styles (e.g. complex gate CMOS design) the implementation of a function corresponds directly to its factored form.

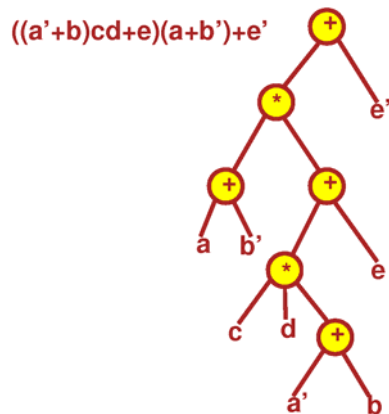
22

Factored Form

- Factored forms can be graphically represented as labeled **trees**, called factoring trees, in which each internal node including the root is labeled with either $+$ or $\times$, and each leaf has a label of either a variable or its complement

- **Example**

factoring tree of $((a'+b)cd+e)(a+b')+e'$



23

Factored Form

- Definition:** The **size** of a factored form F (denoted $\rho(F)$) is the number of literals in the factored form
 - E.g., $\rho((a+b)ca') = 4$, $\rho((a+b+cd)(a'+b')) = 6$
- A factored form of a function is **optimal** if no other factored form has less literals
- A factored form is **positive unate** in x , if x appears in F , but x' does not. A factored form is **negative unate** in x , if x' appears in F , but x does not.
- F is **unate** in x if it is either positive or negative unate in x , otherwise F is **binate** in x
 - E.g., $F = (a+b')c+a'$
positive unate in c ; negative unate in b ; binate in a

24

Factored Form Cofactor

- The cofactor of a factored form F , with respect to a literal x_1 (or x_1'), is the factored form $F_{x_1} = F_{x_1=1}(x)$ (or $F_{x_1'} = F_{x_1=0}(x)$) obtained by
 - replacing all occurrences of x_1 by 1, and x_1' by 0
 - simplifying the factored form using the Boolean algebra identities
$$1y=y \quad 1+y=1 \quad 0y=0 \quad 0+y=y$$
 - after constant propagation (all constants are removed), part of the factored form may appear as $G+G$. In general, G is in a factored form.

25

Factored Form Cofactor

- The cofactor of a factored form F , with respect to a cube c , is a factored form F_c obtained by successively cofactoring F with each literal in c

- Example

$$F = (x+y'+z)(x'u+z'y'(v+u')) \text{ and } c = vz'.$$

Then

$$F_{z'} = (x+y')(x'u+y'(v+u'))$$

$$F_{z'v} = (x+y')(x'u+y')$$

26

Factored Form Optimality

□ Definition

Let f be a completely specified Boolean function, and $\rho(f)$ is the minimum number of literals in any factored form of f

■ Recall $\rho(F)$ is the number of literals of a factored form F

□ Definition

Let $\text{sup}(f)$ be the true support variable of f , i.e. the set of variables that f depends on. Two functions f and g are **orthogonal**, denoted $f \perp g$, if $\text{sup}(f) \cap \text{sup}(g) = \emptyset$

27

Factored Form Optimality

□ Lemma: Let $f = g + h$ such that $g \perp h$, then $\rho(f) = \rho(g) + \rho(h)$

■ Proof:

Let F , G and H be the optimum factored forms of f , g and h . Since $G+H$ is a factored form, $\rho(f) = \rho(F) \leq \rho(G+H) = \rho(g) + \rho(h)$.

Let c be a minterm, on $\text{sup}(g)$, of g' . Since g and h have disjoint support, we have $f_c = (g+h)_c = g_c + h_c = 0 + h_c = h_c = h$. Similarly, if d is a minterm of h' , $f_d = g$. Because $\rho(h) = \rho(f_c) \leq \rho(F_c)$ and $\rho(g) = \rho(f_d) \leq \rho(F_d)$, $\rho(h) + \rho(g) \leq \rho(F_c) + \rho(F_d)$.

Let m (n) be the number of literals in F that are from $\text{SUPPORT}(g)$ ($\text{SUPPORT}(h)$). When computing F_c (F_d), we replace all the literals from $\text{SUPPORT}(g)$ ($\text{SUPPORT}(h)$) by the appropriate values and simplify the factored form by eliminating all the constants and possibly some literals from $\text{sup}(g)$ ($\text{sup}(h)$) by using the Boolean identities. Hence $\rho(F_c) \leq n$ and $\rho(F_d) \leq m$. Since $\rho(F) = m+n$, $\rho(F_c) + \rho(F_d) \leq m+n = \rho(F)$.

We have $\rho(f) \leq \rho(g) + \rho(h) \leq \rho(F_c) + \rho(F_d) \leq \rho(F) \Rightarrow \rho(f) = \rho(g) + \rho(h)$ since $\rho(f) = \rho(F)$.

28

Factored Form

Optimality

- Note, the previous result does not imply that **all** minimum literal factored forms of f are sums of the minimum literal factored forms of g and h

- **Corollary:** Let $f = gh$ such that $g \perp h$, then $\rho(f) = \rho(g) + \rho(h)$

- **Proof:**

Let F' denote the factored form obtained using DeMorgan's law. Then $\rho(F) = \rho(F')$, and therefore $\rho(f) = \rho(f')$. From the above lemma, we have $\rho(f) = \rho(f') = \rho(g' + h') = \rho(g') + \rho(h') = \rho(g) + \rho(h)$.

- **Theorem:** Let $f = \sum_{i=1}^n \prod_{j=1}^m f_{ij}$ such that $f_{ij} \perp f_{kl}, \forall i \neq j \text{ or } k \neq l$, then

$$\rho(f) = \sum_{i=1}^n \sum_{j=1}^m \rho(f_{ij})$$

- **Proof:**

Use induction on m and then n , and the above lemma and corollary.

29

Factored Form

- SOP forms are used as the internal representation of logic functions in most multi-level logic optimization systems
- **Advantages**
 - good algorithms for manipulating them are available
- **Disadvantages**
 - performance is unpredictable - may accidentally generate a function whose SOP form is too large
 - factoring algorithms have to be used constantly to provide an estimate for the size of the Boolean network, and the time spent on factoring may become significant
- **Possible solution**
 - **avoid** SOP representation by using factored forms as the internal representation
 - still not practical unless we know how to perform logic operations **directly** on factored forms without converting to SOP forms
 - the most common logic operations over factored form have been partially provided

30

Boolean Network Manipulation

□ Basic techniques

- Structural operations (change topology)
 - Algebraic
 - Boolean
- Node simplification (change node functions)
 - Node minimization using don't cares

31

Structural Operation

□ Restructuring: Given initial network, find best network

■ Example

$$f_1 = abcd + ab'cd' + acd'e + ab'c'd' + a'c + cdf + abc'd'e' + ab'c'df'$$

$$f_2 = bdg + b'dfg + b'd'g + bd'eg$$

minimizing

$$f_1 = bcd + b'cd' + cd'e + a'c + cdf + abc'd'e' + ab'c'df'$$

$$f_2 = bdg + dfg + b'd'g + d'eg$$

factoring

$$f_1 = c(d(b+f) + d'(b'+e) + a') + ac'(bd'e' + b'df')$$

$$f_2 = g(d(b+f) + d'(b'+e))$$

decompose

$$f_1 = c(x+a') + ac'x'$$

$$f_2 = gx$$

$$x = d(b+f) + d'(b'+e)$$

□ Two problems:

- find good common subfunctions
- effect the division

32

Structural Operation

Basic Operations:

- Decomposition (single function)

$$f = abc + abd + a'c'd' + b'c'd' \Rightarrow$$

$$f = xy + x'y' \quad x = ab \quad y = c + d$$
- Extraction (multiple functions)

$$f = (az + bz')cd + e \quad g = (az + bz')e' \quad h = cde \Rightarrow$$

$$f = xy + e \quad g = xe' \quad h = ye \quad x = az + bz' \quad y = cd$$
- Factoring (series-parallel decomposition)

$$f = ac + ad + bc + bd + e \Rightarrow$$

$$f = (a + b)(c + d) + e$$
- Substitution

$$g = a + b \quad f = a + bc \Rightarrow$$

$$f = g(a + c)$$
- Collapsing (also called elimination)

$$f = ga + g'b \quad g = c + d \Rightarrow$$

$$f = ac + ad + bc'd' \quad g = c + d$$

"Division" plays a key role in all of these operations

33

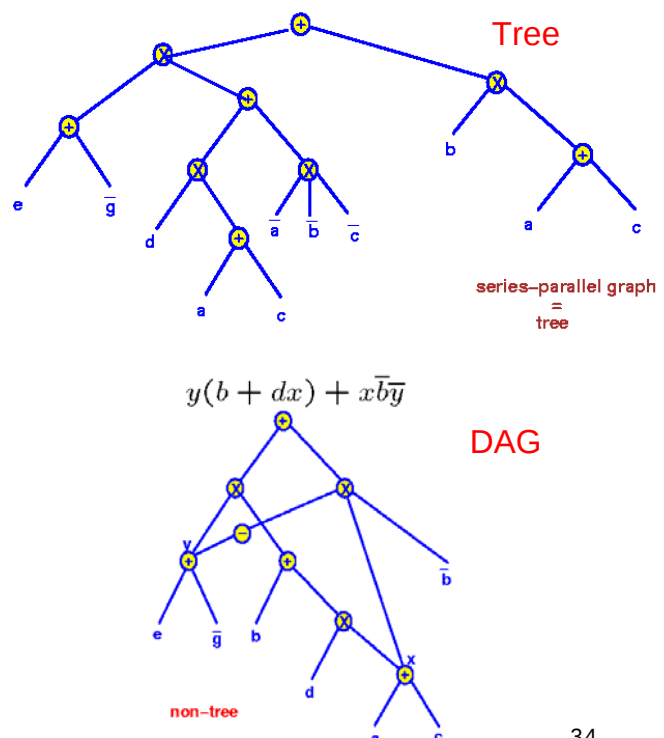
Factoring vs. Decomposition

Factoring:

- $f = (e + g')(d(a + c) + a'b'c') + b(a + c)$

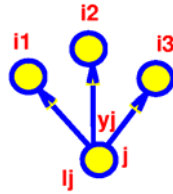
Decomposition:

- $y(b + dx) + xb'y'$
 - Similar to merging common nodes and using negative pointers in BDD. However, **not** canonical, so have no perfect identification of common nodes.



34

Structural Operation Node Elimination



$$value(j) = \left(\sum_{i \in FO(j)} n_i \right) (l_j - 1) - l_j$$

where

n_i = number of times literals y_j and y_j' occur in factored form f_i

■ can treat y_j and y_j' the same since $\rho(F_j) = \rho(F_j')$

l_j = number of literals in factored f_j

with factoring

$$l_j + \sum_{i \in FO(j)} n_i + c$$

without factoring

$$l_j \sum_{i \in FO(j)} n_i + c$$

value = (without factoring) - (with factoring)

35

Structural Operation Node Elimination

Example

■ Literals before

$$5 + 7 + 5 = 17$$

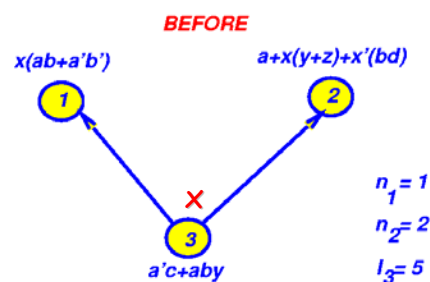
■ Literals after

$$9 + 15 = 24$$

■ Difference:

after - before =

$$value = 7$$



$$value(3) = (1+2-1)(5-1) - 1 = 7$$

AFTER



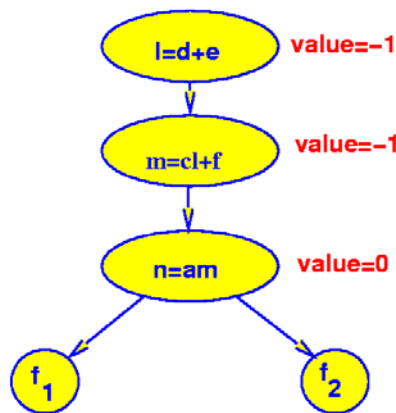
$$value(j) = \left(\sum_{i \in FO(j)} n_i \right) (l_j - 1) - l_j$$

$$= (n_1 + n_2)(l_3 - 1) - l_3$$

$$= (1 + 2)(5 - 1) - 5 = 7$$

36

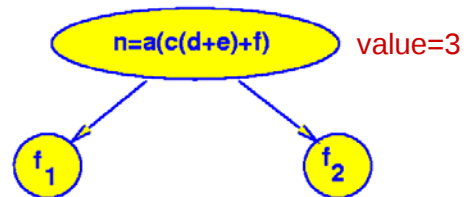
Structural Operation Node Elimination



$$n = a(c(d + e) + f)$$

$$f_1 = b(n + ag) + h$$

$$f_2 = i(n + aj) + k$$



Note: Value of a node can change during elimination

37

Factorization

□ Given a SOP, how do we generate a “good” factored form

□ Division operation:

- is central in many operations
- find a good divisor
- apply division
 - results in quotient and remainder

□ Applications:

- factoring
- decomposition
- substitution
- extraction

38

Division

□ **Definition:** An operation **op** is called **division** if, given two SOP expressions F and G, it generates expressions H and R ($\langle H, R \rangle = \text{op}(F, G)$) such that **$F = GH + R$**

- G is called the divisor
- H is called the quotient
- R is called the remainder

□ **Definition:** If GH is an algebraic product, then **op** is called an algebraic division (denoted **$F // G$**), otherwise GH is a Boolean product and **op** is called a Boolean division (denoted **$F \div G$**)

39

Division

□ **Example:**

$$f = ad + ae + bcd + j$$

$$g_1 = a + bc$$

$$g_2 = a + b$$

■ **Algebraic division:**

□ $f // a = d + e, r = bcd + j$

Also, $f // a = d$ or $f // a = e$, i.e. algebraic division is not unique

□ $f // (bc) = d, r = ad + ae + j$

□ $h_1 = f // g_1 = d, r_1 = ae + j$

■ **Boolean division:**

□ $h_2 = f \div g_2 = (a + c)d, r_2 = ae + j.$
i.e. $f = (a+b)(a+c)d + ae + j$

40

Division

□ Definition:

G is an **algebraic factor** of F if there exists an algebraic expression H such that $F = GH$ (using algebraic multiplication)

□ Definition:

G is an **Boolean factor** of F if there exists an expression H such that $F = GH$ (using Boolean multiplication)

□ Example

■ $f = ac + ad + bc + bd$

□ $(a+b)$ is an algebraic factor of f since $f = (a+b)(c+d)$

■ $f = -ab + ac + bc$

□ $(a+b)$ is a Boolean factor of f since $f = (a+b)(-a+c)$

41

Why Algebraic Methods?

□ Algebraic methods provide fast algorithms for various operations

■ Treat logic functions as polynomials

■ Fast algorithms for polynomials exist

■ Lost of optimality but results are still good

■ Can iterate and interleave with Boolean operations

□ In specific instances, slight extensions are available to include Boolean methods

42

Weak Division

□ **Weak division** is a specific example of algebraic division

□ **Definition:**

Given two algebraic expressions F and G, a division is called a **weak division** if

1. it is **algebraic** and
 2. remainder R has **as few cubes as possible**
- The **quotient** H resulting from weak division is denoted by **F/G**

□ **Theorem:**

Given expressions F and G, H and R generated by weak division are unique

43

Weak Division

```
ALGORITHM WEAK_DIV(F,G) {  
  // G = {g1, g2, ...}, F = {f1, f2, ...} are sets of cubes  
  foreach gi {  
    Vgi = ∅  
    foreach fj {  
      if(fj contains all literals of gi) {  
        vij = fj - literals of gi  
        Vgi = Vgi ∪ vij  
      }  
    }  
  }  
  H = ∩i Vgi  
  R = F - GH  
  return (H,R);  
}
```

44

Weak Division

□ Example

$$F = ace + ade + bc + bd + be + a'b + ab$$

$$G = ae + b$$

$$V^{ae} = c + d$$

$$V^b = c + d + e + a' + a$$

$$H = c + d = F/G$$

$$H = \cap V^{g_i}$$

$$R = be + a'b + ab$$

$$R = F \setminus GH$$

$$F = (ae + b)(c + d) + be + a'b + ab$$

45

Weak Division

□ We use **filters** to prevent trying a division

■ G is not an algebraic divisor of F if

- G contains a literal not in F,
- G has more terms than F,
- For any literal, its count in G exceeds that in F, or
- F is in the transitive fanin of G.

46

Weak Division

- Weak_Div provides a method to divide an expression for a given divisor
- How do we find a “good” divisor?
 - Restrict to algebraic divisors
 - Generalize to Boolean divisors
- Problem:
Given a set of functions $\{ F_i \}$, find common weak (algebraic) divisors

47

Divisor Identification Primary Divisor

- Definition:
An expression is **cube-free** if no cube divides the expression evenly (i.e., there is no literal that is common to all the cubes)
 - “ab+c” is cube-free
 - “ab+ac” and “abc” are not cube-free
 - Note: A cube-free expression **must** have more than one cube
- Definition:
The **primary divisors** of an expression F are the set of expressions
$$D(F) = \{ F/c \mid c \text{ is a cube} \}$$
Note that F/c is the **quotient of a weak division**

48

Divisor Identification

Kernel and Co-Kernel

□ Definition:

The **kernels** of an expression F are the set of expressions

$$K(F) = \{G \mid G \in D(F) \text{ and } G \text{ is cube-free}\}$$

- In other words, the kernels of an expression F are the **cube-free primary divisors** of F

□ Definition:

A cube c used to obtain the kernel $K = F/c$ is called a **co-kernel** of K

- $C(F)$ is used to denote the **set of co-kernels** of F

49

Divisor Identification

Kernel and Co-Kernel

□ Example

$$\begin{aligned} x &= adf + aef + bdf + bef + cdf + cef + g \\ &= (a + b + c)(d + e)f + g \end{aligned}$$

kernels

$a+b+c$

$d+e$

$(a+b+c)(d+e)f+g$

co-kernels

df, ef

af, bf, cf

1

50

Divisor Identification

Kernel and Kernel Intersection

□ Fundamental Theorem

If two expressions F and G have the property that

$\forall k_F \in K(F), \forall k_G \in K(G) \rightarrow |k_G \cap k_F| \leq 1$
(k_G and k_F have at most one term in common),
then F and G have **no common** algebraic divisors
with **more than one cube**

■ Important:

If we “kernel” all functions and there are no nontrivial intersections, then the only common algebraic divisors left are **single cube divisors**

51

Divisor Identification

Kernel Level

□ Definition:

A kernel is of **level 0** (K^0) if it contains no kernels except itself

A kernel is of **level n** or less (K^n) if it contains at least one kernel of level (n-1) or less, but no kernels (except itself) of level n or greater

- $K^n(F)$ is the set of kernels of level n or less
- $K^0(F) \subset K^1(F) \subset K^2(F) \subset \dots \subset K^n(F) \subset K(F)$
- level-n kernels = $K^n(F) \setminus K^{n-1}(F)$

□ Example:

$$F = (a + b(c + d))(e + g)$$

$$k_1 = a + b(c + d) \in K^1 \\ \notin K^0 \Rightarrow \text{level-1}$$

$$k_2 = c + d \in K^0$$

$$k_3 = e + g \in K^0$$

52

Divisor Identification Kerneling Algorithm

```
Algorithm KERNEL(j, G) {  
  R =  $\emptyset$   
  if(CUBE_FREE(G)) R = {G}  
  for(i=j+1, ..., n) {  
    if( $l_i$  appears only in one term)          continue  
    if( $\exists k \leq i, l_k \in$  all cubes of  $G/l_i$ ) continue  
    R = R  $\cup$  KERNEL(i, MAKE_CUBE_FREE( $G/l_i$ ))  
  }  
  return R  
}  
MAKE_CUBE_FREE(F) removes algebraic cube factor from F
```

53

Divisor Identification Kerneling Algorithm

□ **KERNEL**(0, F) returns all the kernels of F

□ Note:

- The test “($\exists k \leq i, l_k \in$ all cubes of G/l_i)” in the kerneling algorithm is a **major** efficiency factor. It also guarantees that no co-kernel is tried more than once.
- Can be used to generate all co-kernels

54

Factor

```
Algorithm FACTOR(F) {  
  if(F has no factor) return F  
  // e.g. if |F|=1, or F is an OR of single literals  
  // or of no literal appears more than once  
  D      = CHOOSE_DIVISOR(F)  
  (Q,R) = DIVIDE(F,D)  
  return FACTOR(Q)×FACTOR(D) + FACTOR(R) //recur  
}
```

- different heuristics can be applied for **CHOOSE_DIVISOR**
- different **DIVIDE** routines may be applied (algebraic division, Boolean division)

57

Factor

□ Example:

$F = abc + abd + ae + af + g$

$D = c + d$

$Q = ab$

$P = ab(c + d) + ae + af + g$

$O = ab(c + d) + a(e + f) + g$

Notation:

F = original function

D = divisor

Q = quotient

P = partial factored form

O = final factored form by
FACTOR restricting to
algebraic operations only

■ Problem 1:

O is not optimal since not maximally factored and can be further factored to “ $a(b(c + d) + e + f) + g$ ”

- It occurs when quotient Q is a single cube, and some of the literals of Q also appear in the remainder R

58

Factor

□ To solve Problem 1

■ Check if the quotient Q is not a single cube, then done

■ Else, pick a literal l_1 in Q which occurs most frequently in cubes of F . Divide F by l_1 to obtain a new divisor D_1 .

Now, F has a new partial factored form

$$(l_1)(D_1) + (R_1)$$

and literal l_1 does not appear in R_1 .

■ **Note:** The new divisor D_1 contains the original D as a divisor because l_1 is a literal of Q . When recursively factoring D_1 , D can be discovered again.

59

Factor

□ Example:

$$F = ace + ade + bce + bde + cf + df$$

$$D = a + b$$

$$Q = ce + de$$

$$P = (ce + de)(a + b) + (c + d)f$$

$$O = e(c + d)(a + b) + (c + d)f$$

Notation:

F = original function

D = divisor

Q = quotient

P = partial factored form

O = final factored form by

FACTOR restricting to

algebraic operations only

■ Problem 2:

O is not maximally factored because “ $(c + d)$ ” is common to both products “ $e(c + d)(a + b)$ ” and “ $(c + d)f$ ”

■ The final factored form should have been “ $(c+d)(e(a + b) + f)$ ”

60

Factor

□ To solve Problem 2

■ Essentially, we reverse D and Q!!

- Make Q **cube-free** to get Q_1
- Obtain a new divisor D_1 by dividing F by Q_1
- If D_1 is cube-free, the partial factored form is $F = (Q_1)(D_1) + R_1$, and can recursively factor Q_1 , D_1 , and R_1
- If D_1 is not cube-free, let $D_1 = cD_2$ and $D_3 = Q_1D_2$. We have the partial factoring $F = cD_3 + R_1$. Now recursively factor D_3 and R_1 .

61

Factor

```
Algorithm GFACTOR(F, DIVISOR, DIVIDE) { // good factor
  D = DIVISOR(F)
  if(D = 0) return F
  Q = DIVIDE(F,D)
  if (|Q| = 1) return LF(F, Q, DIVISOR, DIVIDE)
  Q = MAKE_CUBE_FREE(Q)
  (D, R) = DIVIDE(F,Q)
  if (CUBE_FREE(D)) {
    Q = GFACTOR(Q, DIVISOR, DIVIDE)
    D = GFACTOR(D, DIVISOR, DIVIDE)
    R = GFACTOR(R, DIVISOR, DIVIDE)
    return Q × D + R
  }
  else {
    C = COMMON_CUBE(D) // common cube factor
    return LF(F, C, DIVISOR, DIVIDE)
  }
}
```

62

Factor

```
Algorithm LF(F, C, DIVISOR, DIVIDE) { // literal
  factor
  L = BEST_LITERAL(F, C) //L ∈ C most frequent in F
  (Q, R) = DIVIDE(F, L)
  C = COMMON_CUBE(Q) // largest one
  Q = CUBE_FREE(Q)
  Q = GFACTOR(Q, DIVISOR, DIVIDE)
  R = GFACTOR(R, DIVISOR, DIVIDE)
  return L × C × Q + R
}
```

63

Factor

- ❑ Various kinds of factoring can be obtained by choosing different forms of **DIVISOR** and **DIVIDE**
- ❑ **CHOOSE_DIVISOR**:
 - LITERAL** - chooses most frequent literal
 - QUICK_DIVISOR** - chooses the first level-0 kernel
 - BEST_DIVISOR** - chooses the best kernel
- ❑ **DIVIDE**:
 - Algebraic Division
 - Boolean Division

64

Factor

□ Example

$$x = ac + ad + ae + ag + bc + bd + be + bf + ce + cf + df + dg$$

LITERAL_FACTOR:

$$x = a(c + d + e + g) + b(c + d + e + f) + c(e + f) + d(f + g)$$

QUICK_FACTOR:

$$x = g(a + d) + (a + b)(c + d + e) + c(e + f) + f(b + d)$$

GOOD_FACTOR:

$$(c + d + e)(a + b) + f(b + c + d) + g(a + d) + ce$$

65

Factor

- QUICK_FACTOR uses GFACTOR, first level-0 kernel DIVISOR, and WEAK_DIV

□ Example

$$x = ae + afg + afh + bce + bcfg + bcfh + bde + bdfg + bcfh$$

$$D = c + d \quad \text{---- level-0 kernel (first found)}$$

$$Q = x/D = b(e + f(g + h)) \quad \text{---- weak division}$$

$$Q = e + f(g + h) \quad \text{---- make cube-free}$$

$$(D, R) = \text{WEAK_DIV}(x, Q) \quad \text{---- second division}$$

$$D = a + b(c + d)$$

$$x = QD + R, \quad R = 0$$

$$x = (e + f(g + h)) (a + b(c + d))$$

66

Decomposition

- Decomposition is the same as factoring **except**:
 - divisors are added as **new** nodes in the network
 - the new nodes may **fan out** elsewhere in the network in both **positive** and **negative** phases

```
Algorithm DECOMP( $f_i$ ) {  
   $k = \mathbf{CHOOSE\_KERNEL}(f_i)$   
  if ( $k == 0$ ) return  
   $f_{m+j} = k$  // create new node  $m + j$   
   $f_i = (f_i/k)y_{m+j} + (f_i/k')y'_{m+j} + r$  // change node  $i$  using  
  // new node for kernel  
  DECOMP( $f_i$ )  
  DECOMP( $f_{m+j}$ )  
}
```

Similar to factoring, we can define

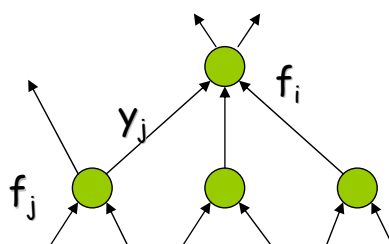
QUICK_DECOMP: pick a level 0 kernel and improve it

GOOD_DECOMP: pick the best kernel

67

Substitution

- **Idea**: An existing node in a network may be a useful divisor in another node. If so, no loss in using it (unless delay is a factor).
- Algebraic substitution consists of the process of algebraically dividing the function f_i at node i in the network by the function f_j (or by f'_j) at node j . During substitution, if f_j is an algebraic divisor of f_i , then f_i is transformed into
$$f_i = qy_j + r \quad (\text{or } f_i = q_1y_j + q_0y'_j + r)$$
- In practice, this is tried for each node pair of the network. n nodes in the network $\Rightarrow O(n^2)$ divisions.



68

Extraction

- **Recall:** Extraction operation identifies **common** sub-expressions and restructures a Boolean network
 - Combine **decomposition** and **substitution** to provide an effective extraction algorithm

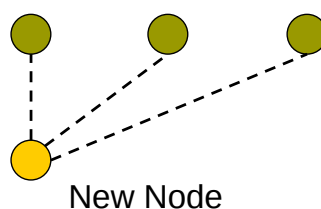
Algorithm **EXTRACT**

```
foreach node n {  
    DECOMP(n) // decompose all network nodes  
}  
foreach node n {  
    RESUB(n) // resubstitute using existing nodes  
}  
ELIMINATE_NODES_WITH_SMALL_VALUE  
}
```

69

Extraction

- **Kernel Extraction:**
 1. Find **all** kernels of all functions
 2. **Choose** kernel intersection with best “value”
 3. **Create** new node with this as function
 4. Algebraically **substitute** new node everywhere
 5. Repeat 1,2,3,4 until best value $\leq$ threshold



70

Extraction

□ Example

$$f_1 = ab(c(d + e) + f + g) + h$$

$$f_2 = ai(c(d + e) + f + j) + k$$

(only level-0 kernels used in this example)

1. Extraction:

$$K^0(f_1) = K^0(f_2) = \{d + e\}$$

$$K^0(f_1) \cap K^0(f_2) = \{d + e\}$$

$$l = d + e$$

$$f_1 = ab(cl + f + g) + h$$

$$f_2 = ai(cl + f + j) + k$$

2. Extraction:

$$K^0(f_1) = \{cl + f + g\}; K^0(f_2) = \{cl + f + j\}$$

$$K^0(f_1) \cap K^0(f_2) = cl + f$$

$$m = cl + f$$

$$f_1 = ab(m + g) + h$$

$$f_2 = ai(m + j) + k$$

No kernel intersections anymore!!

3. Cube extraction:

$$n = am$$

$$f_1 = b(n + ag) + h$$

$$f_2 = i(n + aj) + k$$

71

Extraction Rectangle Covering

□ Alternative method for extraction

□ Build co-kernel cube matrix $M = R^T C$

- rows correspond to co-kernels of individual functions
- columns correspond to individual cubes of kernel
- m_{ij} = cubes of functions
- $m_{ij} = 0$ if cube not there

□ Rectangle covering:

- identify sub-matrix $M^* = R^{*T} C^*$, where $R^* \subseteq R$, $C^* \subseteq C$, and $m^*_{ij} \neq 0$
- construct divisor d corresponding to M^* as new node
- extract d from all functions

72

Extraction Rectangle Covering

Example

$$F = af + bf + ag + cg + ade + bde + cde$$

$$G = af + bf + ace + bce$$

$$H = ade + cde$$

Kernels/Co-kernels:

$$F: (de+f+g)/a$$

$$(de + f)/b$$

$$(a+b+c)/de$$

$$(a + b)/f$$

$$(de+g)/c$$

$$(a+c)/g$$

$$G: (ce+f)/\{a,b\}$$

$$(a+b)/\{f,ce\}$$

$$H: (a+c)/de$$

		<i>a</i>	<i>b</i>	<i>c</i>	<i>ce</i>	<i>de</i>	<i>f</i>	<i>g</i>
<i>F</i>	<i>a</i>					<i>ade</i>	<i>af</i>	<i>ag</i>
<i>F</i>	<i>b</i>					<i>bde</i>	<i>bf</i>	
<i>F</i>	<i>de</i>	<i>ade</i>	<i>bde</i>	<i>cde</i>				
<i>F</i>	<i>f</i>	<i>af</i>	<i>bf</i>					
<i>M = F</i>	<i>c</i>					<i>cde</i>		<i>cg</i>
<i>F</i>	<i>g</i>	<i>ag</i>		<i>cg</i>				
<i>G</i>	<i>a</i>				<i>ace</i>		<i>af</i>	
<i>G</i>	<i>b</i>				<i>bce</i>		<i>bf</i>	
<i>G</i>	<i>ce</i>	<i>ace</i>	<i>bce</i>					
<i>G</i>	<i>f</i>	<i>af</i>	<i>bf</i>					
<i>H</i>	<i>de</i>	<i>ade</i>		<i>cde</i>				

73

Extraction Rectangle Covering

Example (cont'd)

$$F = af + bf + ag + cg + ade + bde + cde$$

$$G = af + bf + ace + bce$$

$$H = ade + cde$$

■ Pick sub-matrix *M'*

■ Extract new expression *X*

$$F = fx + ag + cg + dex + cde$$

$$G = fx + cex$$

$$H = ade + cde$$

$$X = a + b$$

■ Update *M*

		<i>a</i>	<i>b</i>	<i>c</i>	<i>ce</i>	<i>de</i>	<i>f</i>	<i>g</i>
<i>F</i>	<i>a</i>					<i>ade</i>	<i>af</i>	<i>ag</i>
<i>F</i>	<i>b</i>					<i>bde</i>	<i>bf</i>	
<i>F</i>	<i>de</i>	<i>ade</i>	<i>bde</i>	<i>cde</i>				
<i>F</i>	<i>f</i>	<i>af</i>	<i>bf</i>					
<i>M = F</i>	<i>c</i>					<i>cde</i>		<i>cg</i>
<i>F</i>	<i>g</i>	<i>ag</i>		<i>cg</i>				
<i>G</i>	<i>a</i>				<i>ace</i>		<i>af</i>	
<i>G</i>	<i>b</i>				<i>bce</i>		<i>bf</i>	
<i>G</i>	<i>ce</i>	<i>ace</i>	<i>bce</i>					
<i>G</i>	<i>f</i>	<i>af</i>	<i>bf</i>					
<i>H</i>	<i>de</i>	<i>ade</i>		<i>cde</i>				

74

Extraction Rectangle Covering

- Number literals before - Number of literals after

$$V(R', C') = \sum_{i \in R, j \in C} v_{ij} - \sum_{i \in R'} w_i^r - \sum_{j \in C'} w_j^c$$

v_{ij} : Number of literals of cube m_{ij}

w_i^r : (Number of literals of the cube associated with row i) + 1

w_j^c : Number of literals of the cube associated with column j

- For prior example

- $V = 20 - 10 - 2 = 8$

		a	b	c	ce	de	f	g
F	a					ade	af	ag
F	b					bde	bf	
F	de	ade	bde	cde				
F	f	af	bf					
M = F	c					cde		cg
F	g	ag		cg				
G	a				ace		af	
G	b				bce		bf	
G	ce	ace	bce					
G	f	af	bf					
H	de	ade		cde				

75

Extraction Rectangle Covering

- Pseudo Boolean Division

- Idea: consider entries in covering matrix that are don't cares

- overlap of rectangles ($a+a = a$)

- product that cancel each other out ($a \cdot a' = 0$)

- Example:

$$F = ab' + ac' + a'b + a'c + bc' + b'$$

Result:

$$X = a' + b' + c'$$

$$F = ax + bx + cx$$

		a	b	c	a'	b'	c'
F	a				*	ab'	ac'
F	b				a'b	*	bc'
M = F	c				a'c	b'c	*
F	a'	*	a'b	a'c			
F	b'	ab'	*	b'c			
F	c'	ac'	bc'	*			

76

Fast Kernel Computation

- Non-robustness of kernel extraction
 - Recomputation of kernels after every substitution: expensive
 - Some functions may have many kernels (e.g. symmetric functions)
- Cannot measure if kernel can be used as complemented node
- Solution: compute only subset of kernels:
 - Two-cube “kernel” extraction [Rajski et al '90]
 - Objects:
 - 2-cube divisors
 - 2-literal cube divisors
 - **Example:** $f = abd + a'b'd + a'cd$
 - $ab + a'b'$, $b' + c$ and $ab + a'c$ are 2-cube divisors.
 - $a'd$ is a 2-literal cube divisor.

77

Fast Kernel Computation

- Properties of fast divisor (kernel) extraction:
 - $O(n^2)$ number of 2-cube divisors in an n -cube Boolean expression
 - Concurrent extraction of 2-cube divisors and 2-literal cube divisors
 - Handle divisor and complemented divisor simultaneously
- **Example:**
 - $f = abd + a'b'd + a'cd$
 - $k = ab + a'b'$, $k' = ab' + a'b$ (both 2-cube divisors)
 - $j = ab + a'c$, $j' = ab' + a'c'$ (both 2-cube divisors)
 - $c = ab$ (2-literal cube), $c' = a' + b'$ (2-cube divisor)

78

Fast Kernel Computation

□ Generating all two cube divisors

$$F = \{c_i\}$$

$$D(F) = \{d \mid d = \text{make_cube_free}(c_i + c_j)\}$$

- c_i, c_j are any pair of cubes in F
 - I.e., take all pairs of cubes in F and makes them cube-free
- Divisor generation is $O(n^2)$, where n = number of cubes in F

□ Example:

$$F = ax^3 + ag^3 + bcx^3 + bcg^3$$

$$\text{make_cube_free}(c_i + c_j) = \{x^3 + g^3, a^3 + bc^3, ax^3 + bcg^3, ag^3 + bcx^3\}$$

- **Note:** Function F is made into an algebraic expression before generating double-cube divisors
- Not all 2-cube divisors are kernels (why?)

79

Fast Kernel Computation

□ Key results of 2-cube divisors

Theorem: Expressions F and G have a common multiple-cube divisors **if and only if** $D(F) \cap D(G) \neq \emptyset$

Proof:

If:

If $D(F) \cap D(G) \neq \emptyset$ then $\exists d \in D(F) \cap D(G)$ which is a double-cube divisor of F and G . d is a multiple-cube divisor of F and of G .

Only if:

Suppose $C = \{c_1, c_2, \dots, c_m\}$ is a multiple-cube divisor of F and of G . Take any $e = (c_i + c_j)$. If e is cube-free, then $e \in D(F) \cap D(G)$. If e is not cube-free, then let $d = \text{make_cube_free}(c_i + c_j)$. d has 2 cubes since F and G are algebraic expressions. Hence $d \in D(F) \cap D(G)$.

80

Fast Kernel Computation

□ Example:

Suppose that $C = ab + ac + f$ is a multiple divisor of F and G

If $e = ac + f$, e is cube-free and $e \in D(F) \cap D(G)$

If $e = ab + ac$, $d = \{b + c\} \in D(F) \cap D(G)$

As a result of the Theorem, all multiple-cube divisors can be “discovered” by using just double-cube divisors

81

Fast Kernel Computation

□ Algorithm:

- Generate and store all 2-cube kernels (2-literal cubes) and recognize complement divisors
- Find the best 2-cube kernel or 2-literal cube divisor at each stage and extract it
- Update 2-cube divisor (2-literal cubes) set after extraction
- Iterate extraction of divisors until no more improvement

□ Results:

- Much faster
- Quality as good as that of kernel extraction

82

Boolean Division

□ What's wrong with algebraic division?

- Divisor and quotient are orthogonal!

- Better factored form might be:

$$(g_1 + g_2 + \dots + g_n) (d_1 + d_2 + \dots + d_m)$$

- g_i and d_j may share same literals

- redundant product literals

- Example

$$abe + ace + abd + cd / (ae + d) = \emptyset$$

But: $aabe + ace + abd + cd / (ae + d) = (ab + c)$

- g_i and d_j may share opposite literals

- product terms are non-existing

- Example

$$a'b + ac + bc / (a' + c) = \emptyset$$

But: $a'a + a'b + ac + bc / (a' + c) = (a + b)$

83

Boolean Division

□ Definition:

g is a **Boolean divisor** of f if h and r exist such that $f = gh + r$, $gh \neq 0$

g is said to be a **factor** of f if, in addition, $r = 0$, i.e., $f = gh$

- h is called the **quotient**

- r is called the **remainder**

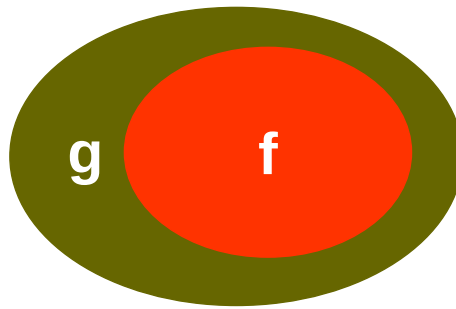
- h and r may **not** be unique

84

Boolean Division

□ Theorem:

A logic function g is a **Boolean factor** of a logic function f if and only if $f \subseteq g$ (i.e. $fg' = 0$, i.e. $g' \subseteq f'$)



85

Boolean Division

Proof:

($\Rightarrow$) g is a Boolean factor of f . Then $\exists h$ such that $f = gh$;
Hence, $f \subseteq g$ (as well as h).

($\Leftarrow$) $f \subseteq g \Rightarrow f = gf = g(f + r) = gh$. (Here r is any function $r \subseteq g'$.)

□ Note:

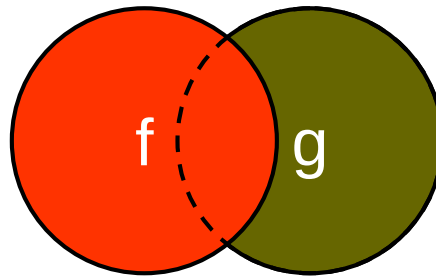
- $h = f$ works fine for the proof
- Given f and g , h is not unique
- To get a small h is the same as to get a small $f + r$. Since $rg = 0$, this is the same as minimizing (simplifying) f with $DC = g'$.

86

Boolean Division

□ Theorem:

g is a Boolean divisor of f if and only if $fg \neq 0$



87

Boolean Division

Proof:

$(\Rightarrow)$ $f = gh + r$, $gh \neq 0 \Rightarrow fg = gh + gr$. Since $gh \neq 0$, $fg \neq 0$.

$(\Leftarrow)$ Assume that $fg \neq 0$. $f = fg + fg' = g(f + k) + fg'$. (Here $k \subseteq g'$.)

Then $f = gh + r$, with $h = f + k$, $r = fg'$. Since $gh = fg \neq 0$, then $gh \neq 0$.

□ Note:

- f has many divisors. We are looking for some g such that $f = gh + r$, where g, h, r are simple functions. (simplify f with DC = g')

88

Boolean Division

Incomplete Specified Function

□ $F = (f, d, r)$

□ Definition:

A completely specified logic function g is a **Boolean divisor of F** if there exist h, e (completely specified) such that

$$f \subseteq gh + e \subseteq f + d$$

and $gh \not\subseteq d$.

□ Definition:

g is a **Boolean factor** of F if there exists h such that

$$f \subseteq gh \subseteq f + d$$

89

Boolean Division

Incomplete Specified Function

□ Lemma:

$f \subseteq g$ if and only if g is a Boolean factor of F .

Proof:

($\Rightarrow$) Assume that $f \subseteq g$. Let $h = f + k$ where $kg \subseteq d$.

Then $hg = (f + k)g \subseteq (f + d)$.

Since $f \subseteq g$, $fg = f$ and thus $f \subseteq (f + k)g = gh$.

Thus

$$f \subseteq (f + k)g \subseteq f + d$$

($\Leftarrow$) Assume that $f = gh$.

Suppose $\exists$ minterm m such that $f(m) = 1$ but $g(m) = 0$.

Then $f(m) = 1$ but $g(m)h(m) = 0$ implying that $f \not\subseteq gh$.

Thus $f(m) = 1$ implies $g(m) = 1$, i.e. $f \subseteq g$

□ Note:

- Since $kg \subseteq d$, $k \subseteq (d + g')$. Hence obtain $h = f + k$ by simplifying f with DC = $(d + g')$.

90

Boolean Division

Incomplete Specified Function

□ Lemma:

$fg \neq 0$ if and only if g is a Boolean divisor of F .

Proof:

($\Rightarrow$) Assume $fg \neq 0$.

Let $fg \subseteq h \subseteq (f + d + g')$ and $fg' \subseteq e \subseteq (f + d)$.

Then $f = fg + fg' \subseteq gh + e \subseteq g(f + d + g') + f + d = f + d$

Also, $0 \neq fg \subseteq gh \rightarrow ghf \neq 0$.

Now $gh \not\subseteq d$, since otherwise $ghf = 0$ (since $fd = 0$),
verifying the conditions of Boolean division.

($\Leftarrow$) Assume that g is a Boolean divisor.

Then $\exists h$ such that $gh \not\subseteq d$ and

$f \subseteq gh + e \subseteq f + d$

Since $gh = (ghf + gh'd) \not\subseteq d$, then $fgh \neq 0$ implying that $fg \neq 0$.

91

Boolean Division

Incomplete Specified Function

□ Recipe for Boolean division:

($f \subseteq gh + e \subseteq f + d$)

■ Choose g such that $fg \neq 0$

■ Simplify fg with DC = $(d + g')$ to get h

■ Simplify fg' with DC = $(d + fg)$ to get e (could
use DC = $d + gh$)

□ $fg \subseteq h \subseteq f + d + g'$

$fg' \subseteq e \subseteq fg' + d + fg = f + d$

92

Boolean Division

- Given $F = (f, d, r)$, write a cover for F in the form $gh + e$ where h and e are minimal in some sense

Algorithm:

1. Create a new variable x to “represent” g
2. Form the don't care set ($\tilde{d} = xg' + x'g$)
(Since $x = g$ we don't care if $x \neq g$)
3. Minimize $(f\tilde{d}', d + \tilde{d}, r\tilde{d}')$ to get $\tilde{f}$
4. Return $(h = \tilde{f}/x, e)$ where e is the remainder of $\tilde{f}$
(These are simply the terms not containing x)
5. f/x denote weak algebraic division

93

Boolean Division

- Note that $(f\tilde{d}', d + \tilde{d}, r\tilde{d}')$ is a partition. We can use ESPRESSO to minimize it, but the objective there is to minimize the number of cubes - not completely appropriate.

Example:

$$f = a + bc$$

$$g = a + b$$

$$\tilde{d} = xa'b' + x'(a+b) \text{ where } x = g = (a+b)$$

- Minimize $(a + bc)\tilde{d}' = (a + bc)(x'a'b' + x(a+b)) = xa + xbc$
with $DC = xa'b' + x'(a+b)$
- A minimum cover is $a + bc$ but it does not use x or x' !
- Force x in the cover. This yields $f = a + xc = a + (a+b)c$.

Heuristic:

Find answer with x in it and which also uses the least variables (or literals)

94

Boolean Division

Assume F is a cover for $\mathfrak{F} = (f,d,r)$ and D is a cover for d .

First Algorithm:

```
Algorithm Boolean_Divide1( $F, D, G$ ) {
   $D_1 = D + xG' + x'G$  // (don't care)
   $F_1 = FD_1'$  // (care on-set)
   $R_1 = (F_1 + D_1)' = F_1'D_1' = F'D_1'$  // (care off-set)
   $F_2 = \text{remove } x' \text{ from } F_1$  // positive substitution only
   $F_3 = \text{MIN\_LITERAL}(F_2, R_1, x)$  // Filter for Espresso
  // (minimum literal support including  $x$ )
   $F_4 = \text{ESPRESSO}(F_3, D_1, R_1)$ 
   $H = F_4/x$  // (quotient)
   $E = F_4 - \{xH\}$  // (remainder)
  return ( $HG+E$ )
}
```

95

Boolean Division

Assume F is a cover for $\mathfrak{F} = (f,d,r)$ and D is a cover for d .

Second Algorithm:

```
Algorithm Boolean_Divide2( $F, D, G$ ) {
   $D_1 = D + xG' + x'G$  // (don't care)
   $F_1 = FD_1'$  // (on-set)
   $R_1 = (F_1 + D_1)' = F_1'D_1' = F'D_1'$  // (off-set)
  //  $F_2 = \text{remove } x' \text{ from } F_1$  (difference to first alg.)
   $F_3 = \text{MIN\_LITERAL}(F_2, R_1, x, x')$  // Filter for Espresso
  // (minimum literal support including  $x$ )
   $F_4 = \text{ESPRESSO}(F_3, D_1, R_1)$ 
   $H_1 = F_4/x$  // (first quotient)
   $H_0 = F_4/x'$  // (first quotient)
   $E = F_4 - (\{xH_1\} + \{x'H_0\})$  // (remainder)
  return ( $GH_1 + G'H_0 + E$ )
}
```

96

Boolean Division

Minimal Literal Support

Support minimization (MINVAR)

Given:

$$\mathfrak{F} = (f, d, r)$$

$$F = \{c^1, c^2, \dots, c^k\} \quad (\text{a cover of } \mathfrak{F})$$

$$R = \{r^1, r^2, \dots, r^m\} \quad (\text{a cover of } r)$$

1. Construct blocking matrix B^i for each c^i

2. Form "super" blocking matrix B

3. Find a minimum cover S of B ,

$$S = \{j_1, j_2, \dots, j_v\}.$$

4. Modify $\tilde{F} \leftarrow \{\tilde{c}^1, \tilde{c}^2, \dots, \tilde{c}^k\}$ where

$$(\tilde{c}^i)_j = \begin{cases} (c^i)_j & \text{if } j \in S \\ \{0, 1\} = 2 & \text{otherwise} \end{cases}$$

$$B = \begin{bmatrix} B^1 \\ B^2 \\ \vdots \\ B^k \end{bmatrix}$$

97

Boolean Division

Minimal Literal Support

Given:

$$\mathfrak{F} = (f, d, r)$$

$$F = \{c^1, c^2, \dots, c^k\} \quad (\text{a cover of } \mathfrak{F})$$

$$R = \{r^1, r^2, \dots, r^m\} \quad (\text{a cover of } r)$$

n : number of variables

Literal Blocking Matrix:

$$(\hat{B}^i)_{q,j} = \begin{cases} 1 & \text{if } v_j \in c^i \text{ and } v'_j \in r^q \\ 0 & \text{otherwise} \end{cases}$$

$$(\hat{B}^i)_{q,j+n} = \begin{cases} 1 & \text{if } v'_j \in c^i \text{ and } v_j \in r^q \\ 0 & \text{otherwise} \end{cases}$$

Example:

$$c^i = ad'e', \quad r^q = a'ce$$

$$\hat{B}_q^i = \begin{matrix} abcde a'b'c'd'e' \\ 1000000001 \end{matrix}$$

98

Boolean Division

Minimal Literal Support

□ Example (literal blocking matrix)

on-set cube: $c^1 = ab'd$

off-set: $r = a'b'd' + abd' + acd' + bcd + c'd'$

	a	b	c	d	a'	b'	c'	d'
a'b'd'	1	0	0	1	0	0	0	0
abd'	0	0	0	1	0	1	0	0
acd'	0	0	0	1	0	0	0	0
bcd	0	0	0	0	0	1	0	0
c'd'	0	0	0	1	0	0	0	0

■ Minimum column cover {d,b'}

■ Thus b'd is the maximum prime covering ab'd

■ Note:

For one cube, minimum literal support is the same as minimum variable support

99

Boolean Division

□ Example

$F = a + bc$

Algebraic division: $F/(a + b) = 0$

Boolean division: $F \div (a + b) = a + c$

1. Let $x = a + b$

2. Generate don't care set: $D_1 = x'(a + b) + xa'b'$.

3. Generate care on-set:

□ $F_1 = F \cap D_1' = (a + bc)(xa + xb + x'a'b') = ax + bcx$.

□ Let $C = \{c^1 = ax, c^2 = bcx\}$

4. Generate care off-set:

□ $R_1 = F'D_1' = (a'b' + a'c')(xa + xb + x'a'b') = a'bc'x + a'b'x'$.

□ Let $R = \{r^1 = a'bc'x, r^2 = a'b'x'\}$.

5. Form super-variable blocking matrix using column order (a, b, c, x), with a', b', c', x' omitted.

$$B = \begin{bmatrix} B^1 \\ B^2 \end{bmatrix} = \begin{bmatrix} 1000 \\ 1001 \\ 0010 \\ 0101 \end{bmatrix}$$

100

Boolean Division

□ Example (cont'd)

6. Find minimum column cover = {a, c, x}
7. Eliminate in F_1 all variables associated with b
So $F_1 = ax + bcx = ax + cx = x(a + c)$
8. Simplifying (applying expand, irredundant on F_1), we get $F_1 = a + xc$
9. Thus quotient = $F_1/x = c$, remainder = a
10. $F = a + bc = a + cx = a + c(a + b)$

It is important that x is forced in the cover!

$$B = \begin{bmatrix} B^1 \\ B^2 \end{bmatrix} = \begin{matrix} & a & b & c & x \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \end{matrix}$$