Logic Synthesis and Verification

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Fall 2024

Multi-Level Logic Minimization

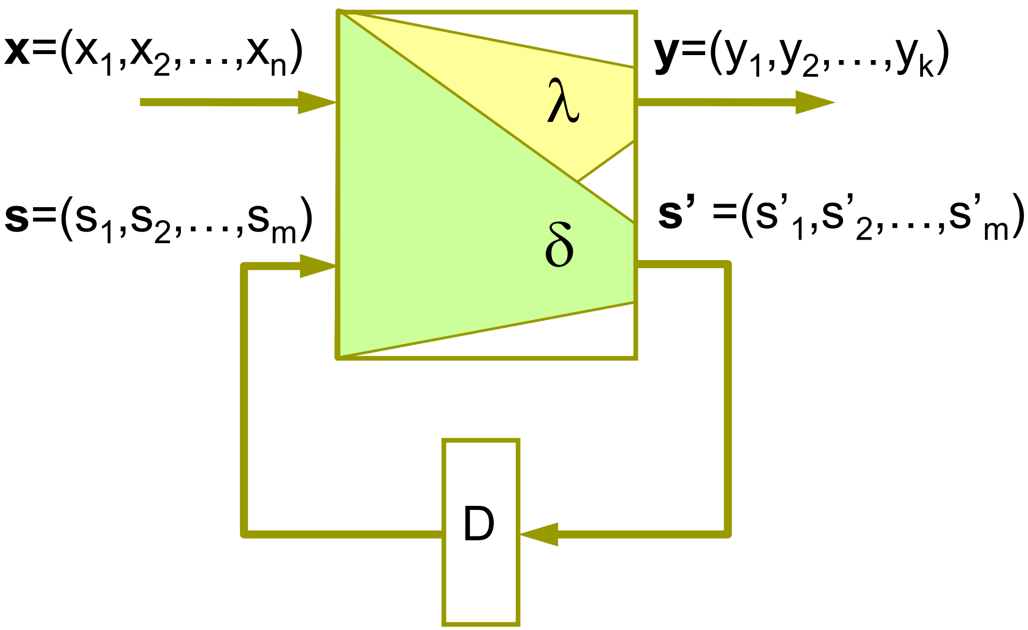
Reading:

Logic Synthesis in a Nutshell

Section 3 (§3.3)

most of the following slides are by courtesy of Andreas Kuehlmann

Finite State Machine



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

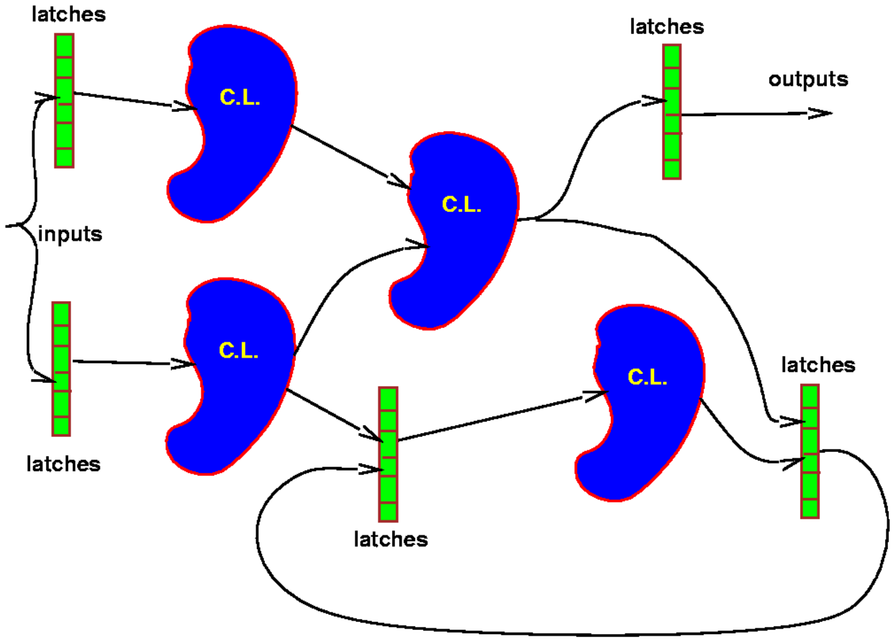
- Q: Set of internal states
- Q_0: Set of initial states
- X: Input alphabet
- Y: Output alphabet
- δ: $X \times Q \rightarrow Q$ (next state *function*)
- λ: $X \times Q \rightarrow Y$ (output *function*)

Delay element:

- Clocked: synchronous circuit
  - single-phase clock, multiple-phase clocks
- Clockless: asynchronous circuit

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



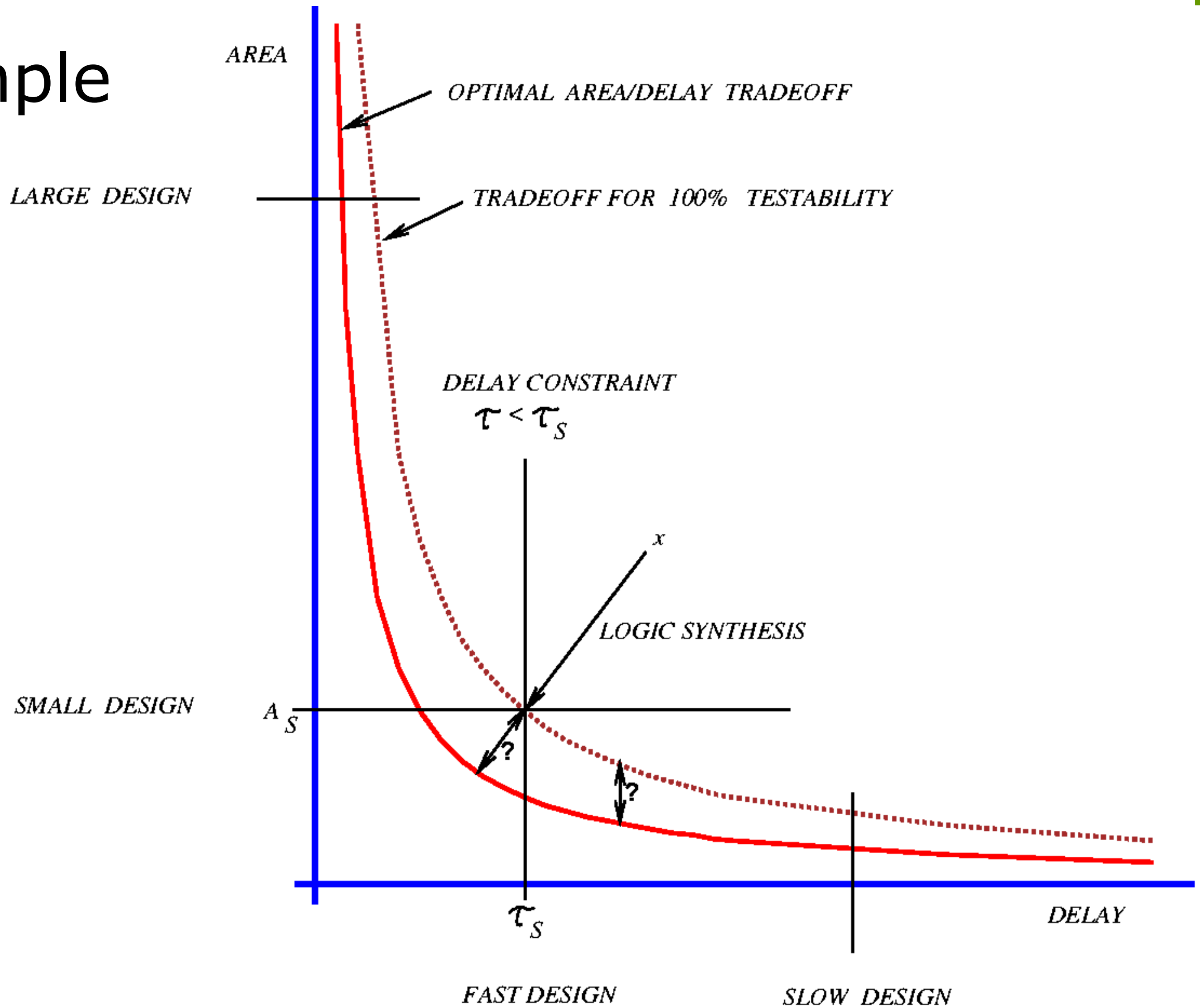
Optimization Criteria for Synthesis

- 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

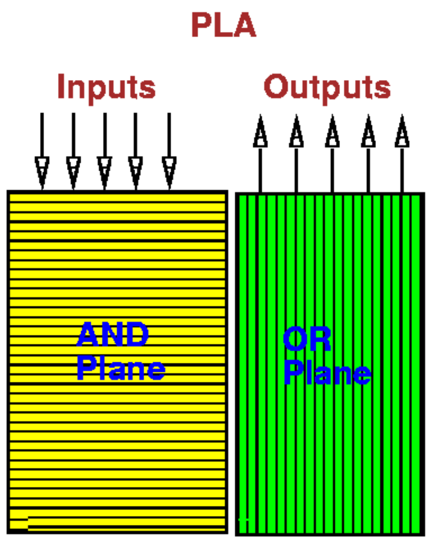
  while simultaneously satisfying upper or lower bound constraints placed on these physical quantities

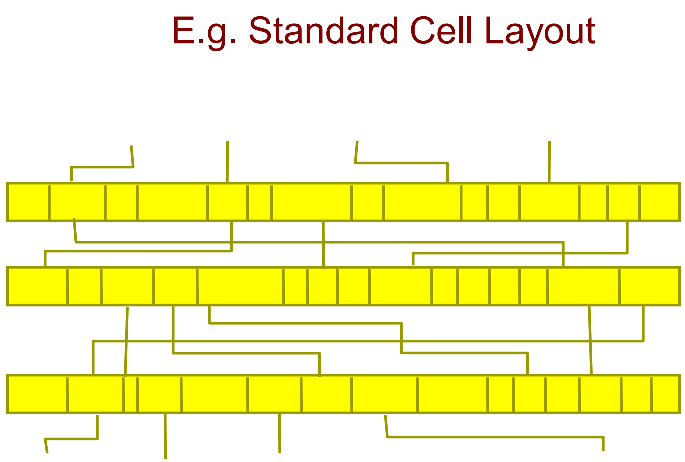
Area-Delay Trade-off

- Example



Two-Level (PLA) vs. Multi-Level





- 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

General Approaches to Synthesis

- PLA synthesis:
  - theory well understood
  - predictable results in a top-down flow

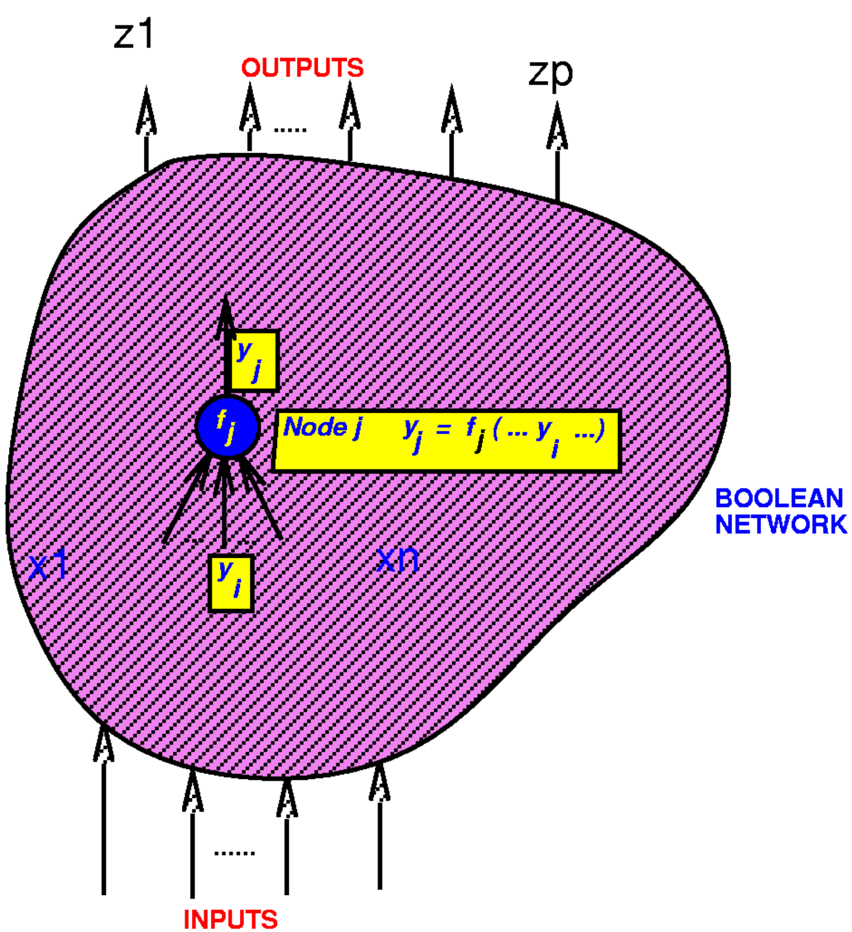
- Multi-level synthesis:
  - optimization criteria very complex
    - except special cases, no general theory available
  - greedy optimization approach
    - incrementally improve along various dimensions of the criteria
  - works on common design representation (circuit or network representation)
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

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

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: $x = (x_1, \ldots, x_n)$
- Outputs: $z = (z_1, \ldots, z_p)$
- External don't cares: $d_1(x), \ldots, d_p(x)$ for outputs



Typical Synthesis Scenario

RTL to Network Transformation

- read Verilog
- control/datapath analysis

Technology Independent Optimizations

- basic logic restructuring
- crude measures for goals

Technology Mapping

- use logic gates from target cell library

Technology Dependent Optimizations

- timing optimization
- physically driven optimization

Test Preparation

- improve testability
- test logic insertion

Local vs. Global Transformation

- Local transformations optimize one node's function in the network
  - smaller area considered
  - 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
    - AIG + cut computation (modern logic synthesis method)

Sum-of-Products (SOP)

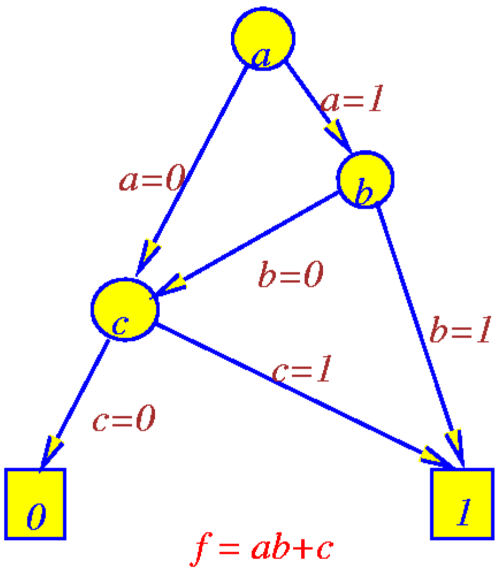
- Example
  $abc' + a'bd + b'd' + b'e'f$

- 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

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



Factor Form

- Example

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

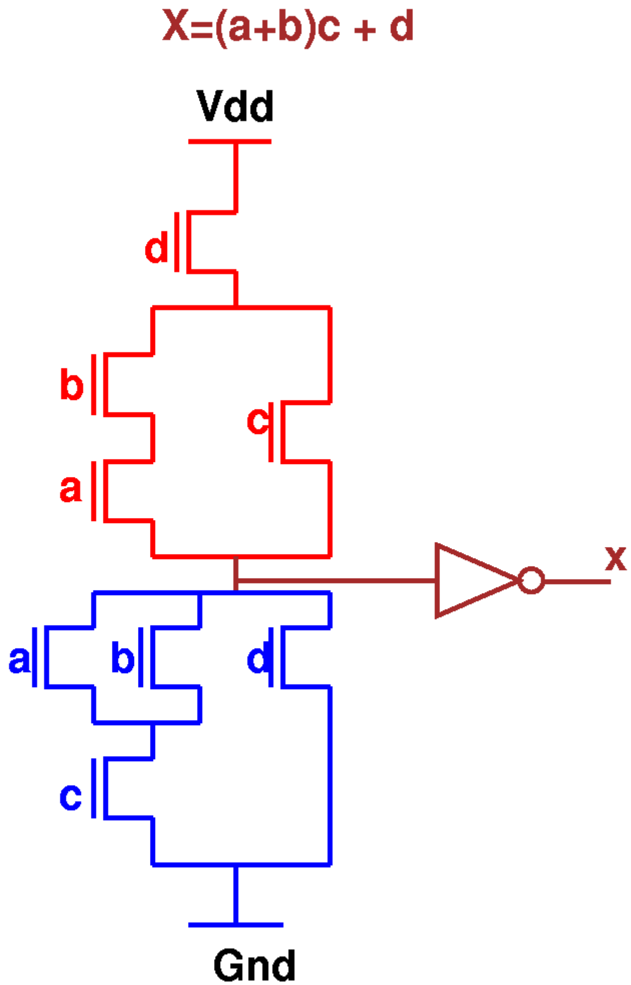
- Advantages
  - good representative of logic complexity

    $f=ad+ae+bd+be+cd+ce$

    $f'=a'b'c'+d'e' \quad \Rightarrow \quad f=(a+b+c)(d+e)$
  - 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

Factor Form



Note:

literal count ≈ transistor count ≈ area

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

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

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

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

    $ab+c(a+b)$, $bc+a(b+c)$, $ac+b(a+c)$

Factored Form

- Definition: The factorization value of an algebraic factorization $F=G_1G_2+R$ is defined to be
  $fact\_val(F,G_2) = lits(F) - (lits(G_1) + lits(G_2) + lits(R))$
  $= (|G_1|-1)\ lits(G_2) + (|G_2|-1)\ lits(G_1)$
  - Assuming G_1, G_2 and R are algebraic expressions, where $|H|$ is the number of cubes in the SOP form of H
  - Example
    $F = ae+af+ag+bce+bcf+bcg+bde+bdf+bdg$
    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 $R=\varnothing$ and
    $fact\_val(F,G_2) = 2\times 3+2\times 5=16$
    - The above factored form saves 17 literals, not 16. The extra literal saving comes from recursively applying the formula to the factored form of G_1.

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

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

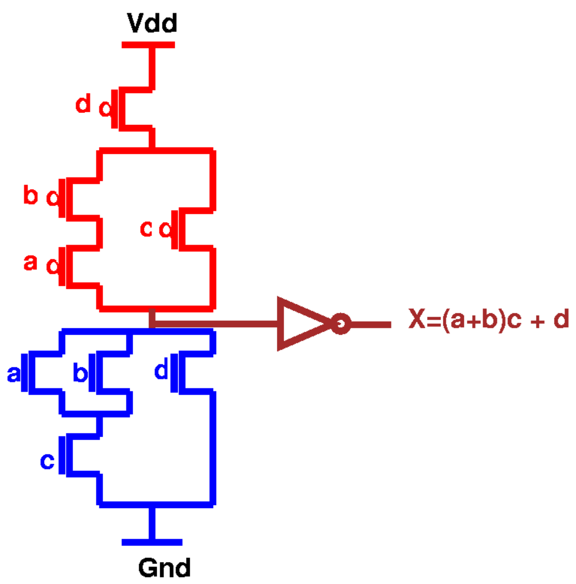
- SOP is a factored form, but it may not be a good factorization

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)×2^{n/2}* literals in SOP form

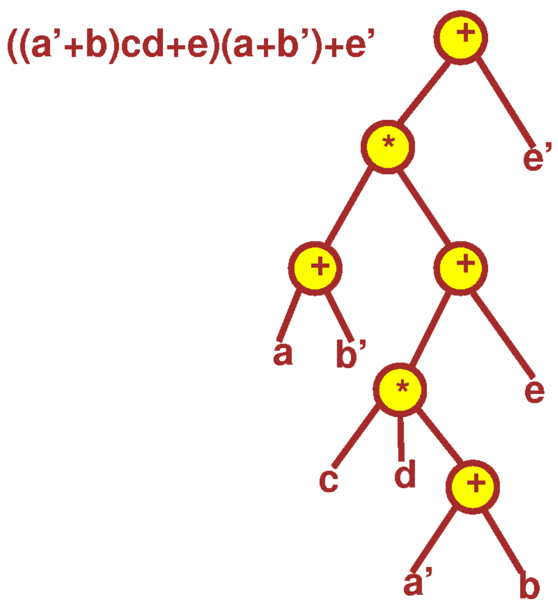


Factored forms are useful in estimating area and delay in a multi-level synthesis and optimization system. In many design styles (e.g. complex gate CMOS design) the implementation of a function corresponds directly to its factored form.

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 ×, and each leaf has a label of either a variable or its complement
  - Example

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



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

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.

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

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

    Then

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

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

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 $sup(f)$ 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 $sup(f) \cap sup(g) = \emptyset$

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 $sup(g)$, 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 $SUPPORT(g)$ ($SUPPORT(h)$). When computing F_c (F_d), we replace all the literals from $SUPPORT(g)$ ($SUPPORT(h)$) by the appropriate values and simplify the factored form by eliminating all the constants and possibly some literals from $sup(g)$ ($sup(h)$) 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 $\rho(f)=\rho(F)$.

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 $f_{ij} \perp f_{kl}$, $\forall i \neq k$ or $j \neq l$, 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.

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

Boolean Network Manipulation

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

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

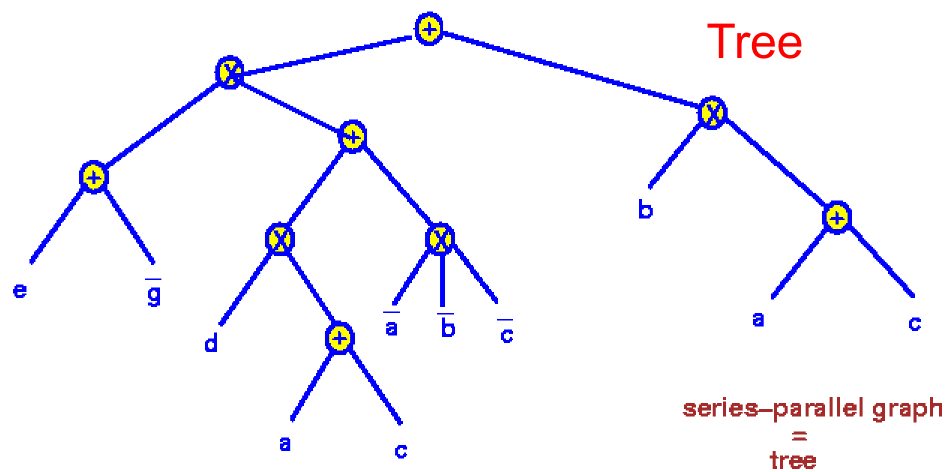
Structural Operation

- Basic Operations:
  - Decomposition (single function)
    $f = abc+abd+a'c'd'+b'c'd' \quad \Rightarrow$
    $f = xy+x'y' \quad x = ab \quad y = c+d$
  - Extraction (multiple functions)
    $f = (az+bz')cd+e \quad g = (az+bz')e' \quad h = cde \quad \Rightarrow$
    $f = xy+e \quad g = xe' \quad h = ye \quad x = az+bz' \quad y = cd$
  - Factoring (series-parallel decomposition)
    $f = ac+ad+bc+bd+e \quad \Rightarrow$
    $f = (a+b)(c+d)+e$
  - Substitution
    $g = a+b \quad f = a+bc \quad \Rightarrow$
    $f = g(a+c)$
  - Collapsing (also called elimination)
    $f = ga+g'b \quad g = c+d \quad \Rightarrow$
    $f = ac+ad+bc'd' \quad g = c+d$

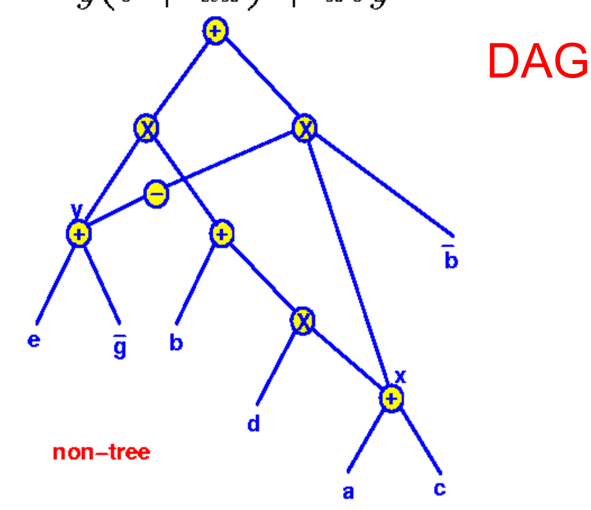
  "Division" plays a key role in all of these operations

Factoring vs. Decomposition

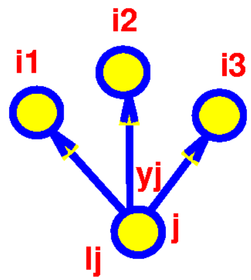
- Factoring:
  - $f=(e+g')(d(a+c)+a'b'c') + b(a+c)$
- Decomposition:
  - $y(b+dx)+xb'y'$
    - Similar to merging common nodes and using negative pointers in BDD. However, not canonical, so have no perfect identification of common nodes.



$y(b+dx)+x\overline{b}\overline{y}$



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

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

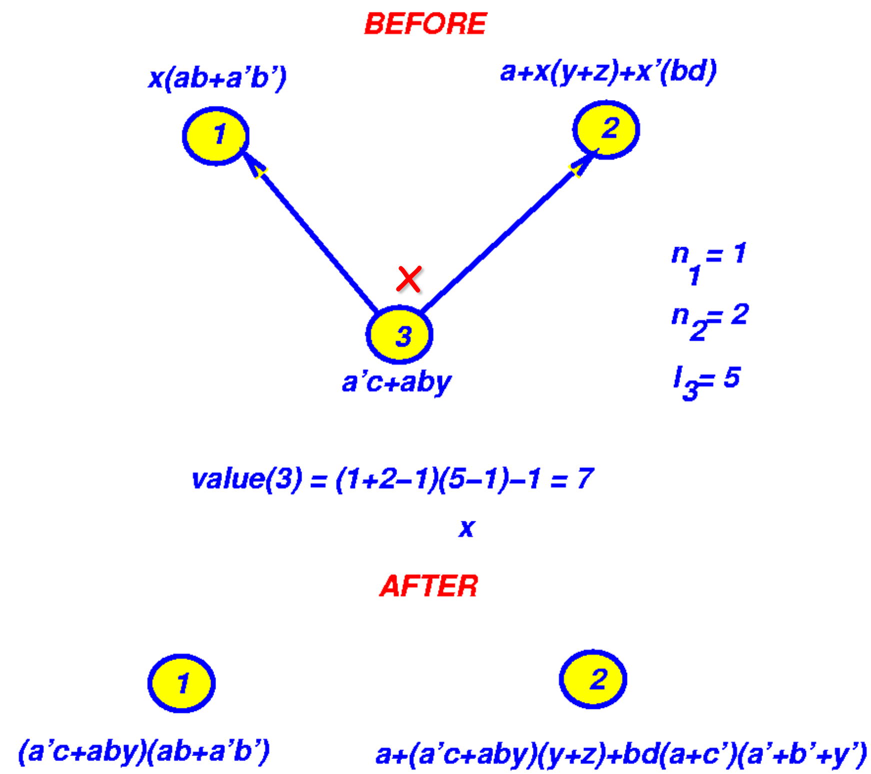
without factoring

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

value = (without factoring) - (with factoring)

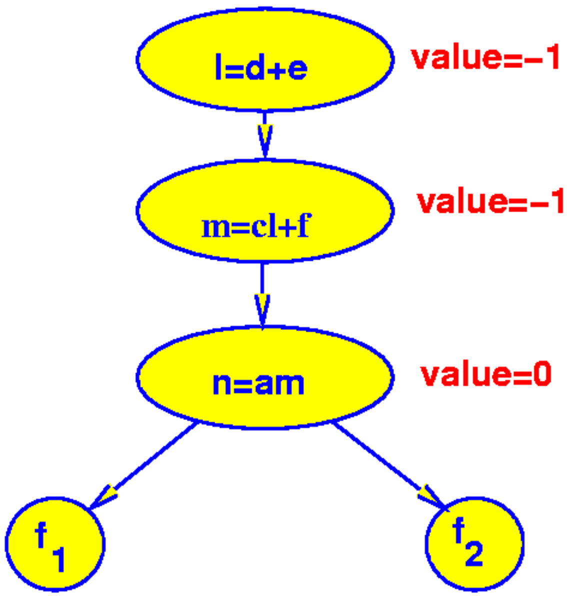
Structural Operation
Node Elimination

- Example
  - Literals before
    5+7+5 = 17
  - Literals after
    9+15 = 24
  - Difference:
    after - before =
    value = 7

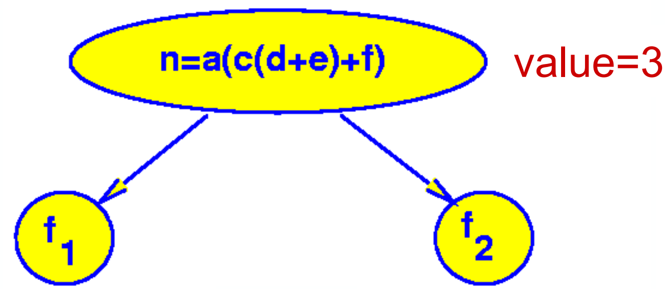


$$value(j) = \left(\sum_{i \in FO(j)} n_i \right) \left(l_j - 1 \right) - l_j$$
$$= (n_1 + n_2)(l_3 - 1) - l_3$$
$$= (1+2)(5-1) - 5 = 7$$

Structural Operation
Node Elimination



$$
\begin{aligned}
n &= a(c(d+e)+f) \\
f_1 &= b(n+ag)+h \\
f_2 &= i(n+aj)+k
\end{aligned}
$$



Note: Value of a node can change during elimination

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

Division

- **Definition:** An operation **op** is called division if, given two SOP expressions F and G, it generates expressions H and R (<H,R> = **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 $F \div G$)

Division

- Example:

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

  $g_1 = a + bc$

  $g_2 = a + b$

  - Algebraic division:
    - $f \mathbin{/\!/} a = d + e$, $r = bcd + j$

      Also, $f \mathbin{/\!/} a = d$ or $f \mathbin{/\!/} a = e$, i.e. algebraic division is not unique
    - $f \mathbin{/\!/} (bc) = d$, $r = ad + ae + j$
    - $h_1 = f \mathbin{/\!/} g_1 = d$, $r_1 = ae + j$
  - Boolean division:
    - $h_2 = f \div g_2 = (a + c)d$, $r_2 = ae + j$.

      i.e. $f = (a+b)(a+c)d + ae + j$

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 a 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 $f = (a+b)(c+d)$
  - $f = \neg ab + ac + bc$
    - $(a+b)$ is a Boolean factor of f since $f = (a+b)(\neg a+c)$

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

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

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);
}
```

Weak Division

- Example

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

  $G = ae + b$

  $V^{ae} = c + d$

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

  $H = c + d = F/G$ $\qquad$ $H = \cap V^{g_i}$

  $R = be + a'b + ab$ $\qquad$ $R = F \setminus GH$

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

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.

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

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 | c is a cube}

  Note that F/c is the quotient of a weak division

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

Divisor Identification
Kernel and Co-Kernel

- Example

  $x = adf + aef + bdf + bef + cdf + cef + g$
  $= (a + b + c)(d + e)f + g$

kernels	co-kernels
$a+b+c$	df, ef
$d+e$	af, bf, cf
$(a+b+c)(d+e)$	f
$(a+b+c)(d+e)f+g$	1

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

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
  - $K^0(F) \subset K^1(F) \subset K^2(F) \subset \ldots \subset K^n(F) \subset K(F)$
  - level-n kernels = $K^n(F) \setminus K^{n-1}(F)$

- Example:
  $F = (a + b(c + d))(e + g)$
  $k_1 = a + b(c + d) \quad \in K^1$
  $\notin K^0$ ==> level-1
  $k_2 = c + d \quad \in K^0$
  $k_3 = e + g \quad \in K^0$

Divisor Identification
Kerneling Algorithm

```
Algorithm KERNEL(j, G) {
  R = ∅
  if(CUBE_FREE(G)) R = {G}
  for(i=j+1,...,n) {
    if(l_i appears only in one term)            continue
    if(∃k ≤ i, l_k ∈ all cubes of G/l_i)        continue
    R = R ∪  KERNEL(i, MAKE_CUBE_FREE(G/l_i))
  }
  return R
}
MAKE_CUBE_FREE(F) removes algebraic cube factor from F
```

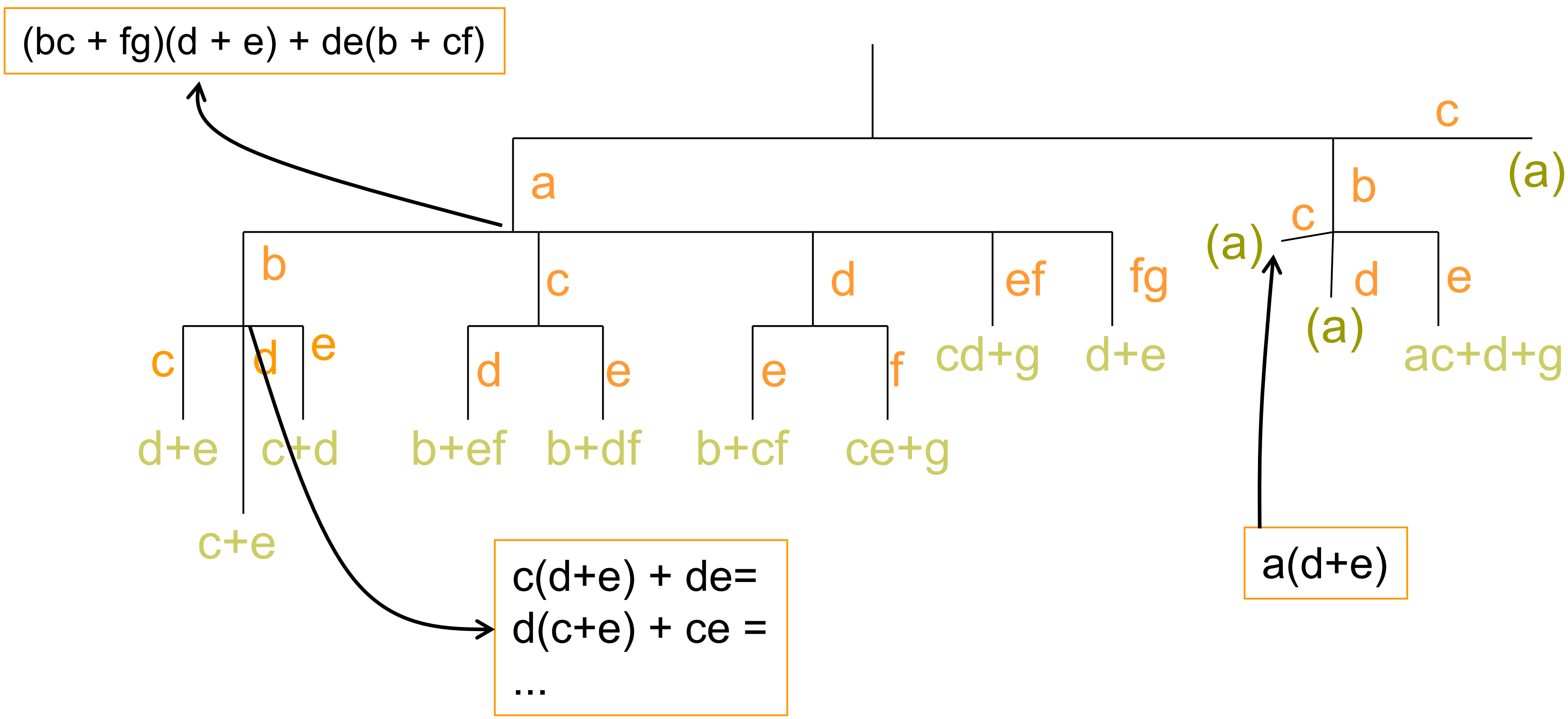
Divisor Identification
Kerneling Algorithm

- KERNEL(0, F) returns all the kernels of F

- Note:
  - The test "($\exists k \le i$, $l_k \in$ 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

Divisor Identification
Kerneling Algorithm

- Example
  F = abcd + abce + adfg + aefg + adbe + acdef + beg
  (Let a, b, c, d, e, f, g be l_1, l_2, l_3, l_4, l_5, l_6,l_7, respectively.)



Divisor Identification
Kerneling Algorithm

- Example

co-kernels	kernels
1	a((bc + fg)(d + e) + de(b + cf))) + beg
a	(bc + fg)(d + e) + de(b + cf)
ab	c(d+e) + de
abc	d + e
abd	c + e
abe	c + d
ac	b(d + e) + def
acd	b + ef

Note: F/bc = ad + ae = a(d + e)

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)

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

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.

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)”

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.

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)
  }
}
```

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
}
```

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

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

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 ---- level-0 kernel (first found)
  Q = x/D = b(e + f(g + h)) ---- weak division
  Q = e + f(g + h) ---- make cube-free
  (D, R) = WEAK_DIV(x, Q) ---- second division
  D = a + b(c + d)
  x = QD + R, R = 0

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

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 = 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

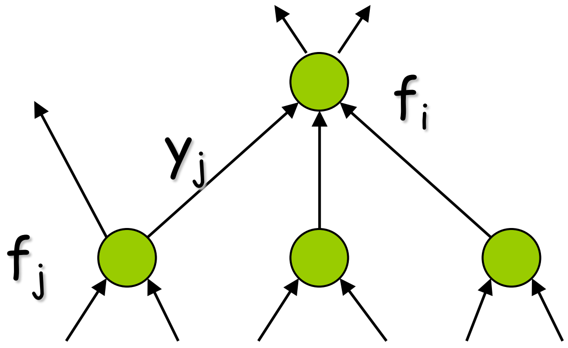
GOOD_DECOMP: pick the best kernel

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
  $f_i = qy_j + r$ (or $f_i = q_1 y_j + q_0 y'_j + r$)

- In practice, this is tried for each node pair of the network. n nodes in the network $\Rightarrow O(n^2)$ divisions.



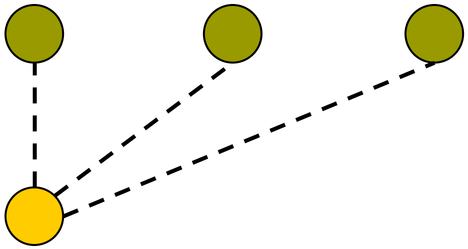
Extraction

- 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
}
```

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



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:

     $n = am$
     $f_1 = b(n + ag) + h$
     $f_2 = i(n + aj) + k$

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

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

			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				

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

$M =$

		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					
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				

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: 1+Number of literals of the cube associated with row i
  - w_j^c: Number of literals of the cube associated with column j
  - For prior example
    - V = 20 - 10 - 2 = 8

$$M = \begin{array}{ll|ccccccc}
 & & a & b & c & ce & de & f & g \\
\hline
F & a & & & & & ade & af & ag \\
F & b & & & & & bde & bf & \\
F & de & ade & bde & cde & & & & \\
F & f & af & bf & & & & & \\
F & c & & & & & cde & & cg \\
F & g & ag & & cg & & & & \\
G & a & & & & ace & & af & \\
G & b & & & & bce & & bf & \\
G & ce & ace & bce & & & & & \\
G & f & af & bf & & & & & \\
H & de & ade & & cde & & & &
\end{array}$$

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·a’ = 0)
  - Example:

    F = ab’ + ac’ + a’b + a’c + bc’ + b’

Result:

X = a’ + b’ + c’

F = ax + bx + cx

$$
M = \begin{array}{cc|cccccc}
 & & a & b & c & a' & b' & c' \\
\hline
F & a & & & & * & ab' & ac' \\
F & b & & & & a'b & * & bc' \\
F & c & & & & a'c & b'c & * \\
F & a' & * & a'b & a'c & & & \\
F & b' & ab' & * & b'c & & & \\
F & c' & ac' & bc' & * & & &
\end{array}
$$

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.

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)

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 = axe + ag + bcxe + bcg

  make_cube_free(c_i + c_j) = {xe + g, a + bc, axe + bcg, ag + bcxe}
  - Note: Function F is made into an algebraic expression before generating double-cube divisors
  - Not all 2-cube divisors are kernels (why?)

Fast Kernel Computation

- Key results of 2-cube divisors

  Theorem: Expressions F and G have a common multiple-cube divisors if and only if $D(F) \cap D(G) \neq 0$

  Proof:

  If:

  If $D(F) \cap D(G) \neq 0$ 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 $C = \{c_1, c_2, ..., c_m\}$ 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 $d \in D(F) \cap D(G)$.

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

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

Boolean Division

- What's wrong with algebraic division?
  - Divisor and quotient are orthogonal!
  - Better factored form might be:
    $(g_1 + g_2 + \ldots + g_n)\ (d_1 + d_2 + \ldots + d_m)$
    - g_i and d_j may share same literals
      - redundant product literals
      - Example
        abe+ace+abd+cd / (ae+d) = ∅
        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) = ∅
        But: a'a+a'b+ac+bc / (a'+c) = (a+b)

Boolean Division

- Definition:

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

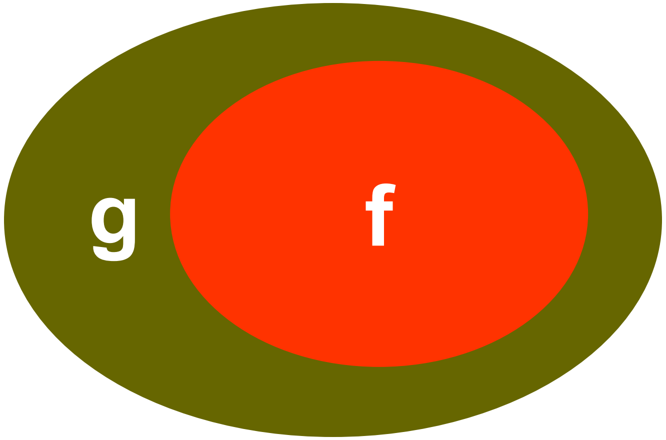
  g is said to be a factor of f if, in addition, $r = 0$, i.e., $f = gh$

  - h is called the quotient
  - r is called the remainder
  - h and r may not be unique

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. $g' \subseteq f'$)



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).

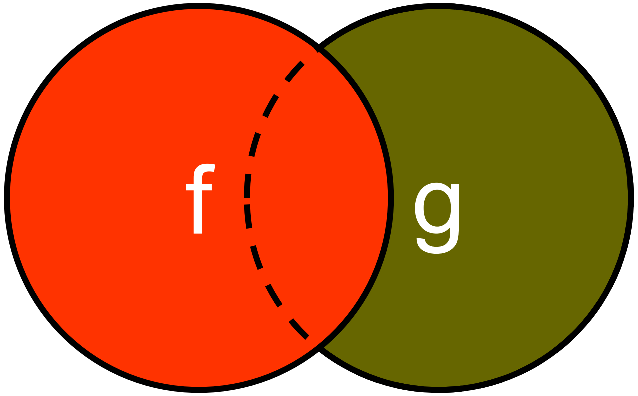
($\Leftarrow$) f $\subseteq$ g $\Rightarrow$ f = gf = g(f + r) = gh. (Here r is any function r $\subseteq$ g'.)

- 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'.

Boolean Division

- Theorem:

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



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')

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 $gh \not\subset d$.

- Definition:

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

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

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 $f \not\subset gh$.
  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 = $(d + g')$.

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 $gh \not\subset d$, 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 $gh \not\subset d$ and
  $f \subseteq gh + e \subseteq f + d$
  Since $gh = (ghf + ghd) \not\subset d$, then $fgh \neq 0$ implying that $fg \neq 0$.

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)

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

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

SAT & Logic Synthesis

Functional Dependency as Boolean Division

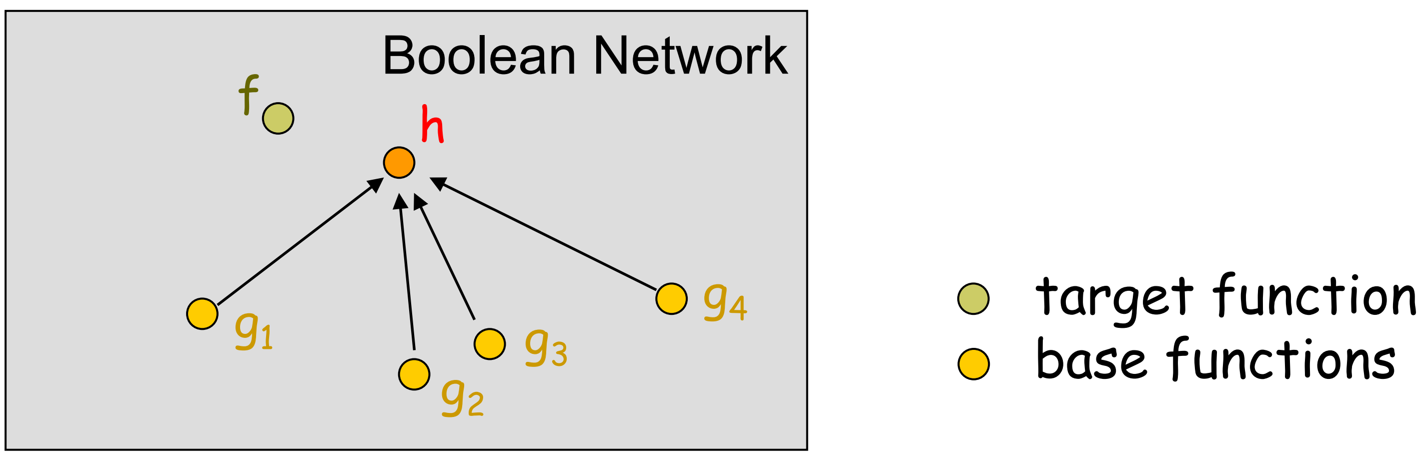
Functional Dependency

- $f(x)$ **functionally depends** on $g_1(x)$, $g_2(x)$, …, $g_m(x)$ if $f(x) = h(g_1(x), g_2(x), …, g_m(x))$, denoted $h(G(x))$
  - Under what condition can function f be expressed as some function h over a set $G=\{g_1,…,g_m\}$ of functions ?
  - h exists $\Leftrightarrow \nexists a,b$ such that $f(a)\neq f(b)$ and $G(a)=G(b)$

i.e., G is more distinguishing than f

Motivation

- Applications of functional dependency
  - Resynthesis/rewiring
  - Redundant register removal
  - BDD minimization
  - Verification reduction
  - …

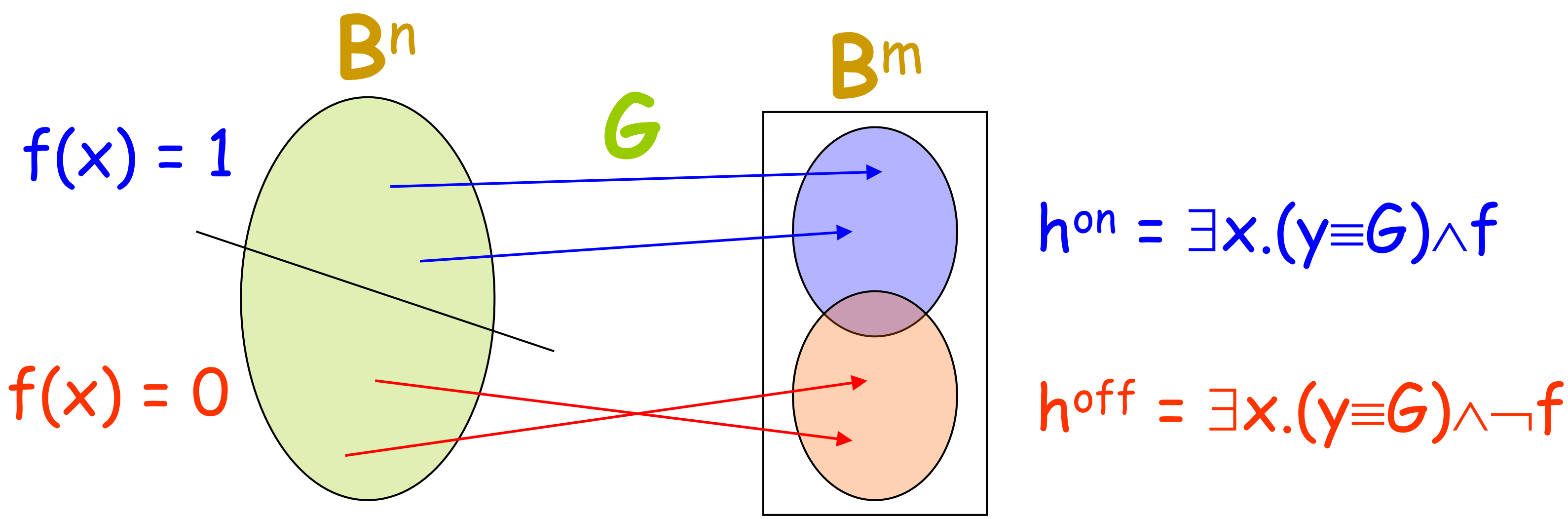


BDD-Based Computation

- BDD-based computation of h

  $h^{on} = \{y \in \mathbf{B}^m : y = G(x) \text{ and } f(x) = 1, x \in \mathbf{B}^n\}$

  $h^{off} = \{y \in \mathbf{B}^m : y = G(x) \text{ and } f(x) = 0, x \in \mathbf{B}^n\}$



BDD-Based Computation

- Pros
  - Exact computation of h^{on} and h^{off}
  - Better support for don't care minimization
- Cons
  - 2 image computations for every choice of G
  - Inefficient when $|G|$ is large or when there are many choices of G

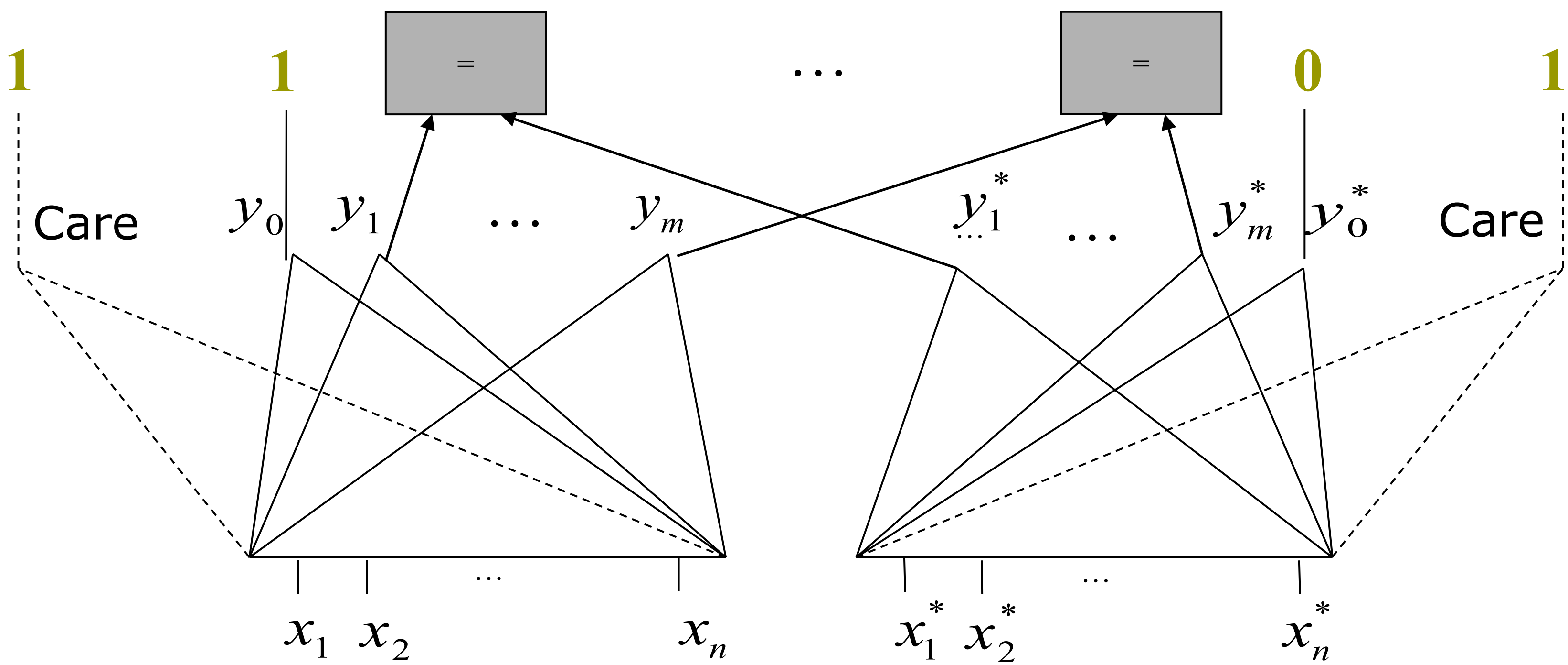
SAT-Based Computation

- h exists $\Leftrightarrow$
  $\nexists a,b$ such that $f(a)\neq f(b)$ and $G(a)=G(b)$,
  i.e., $(f(x)\not\equiv f(x^*))\wedge(G(x)\equiv G(x^*))$ is UNSAT

- How to derive h? How to select G?

SAT-Based Computation

- $(f(x) \not\equiv f(x^*)) \wedge (G(x) \equiv G(x^*))$ is UNSAT

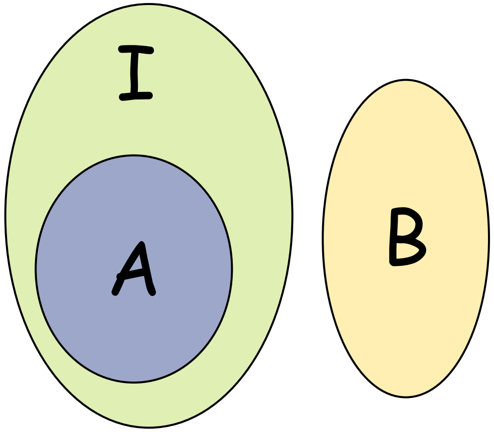


y_0 is the output variable of f; y_i is the output variable of g_i, $i > 0$

Craig Interpolation

- [Craig Interpolation Thm, 1957]
  If $A \wedge B$ is UNSAT for formulae A and B, there exists an interpolant I of A such that

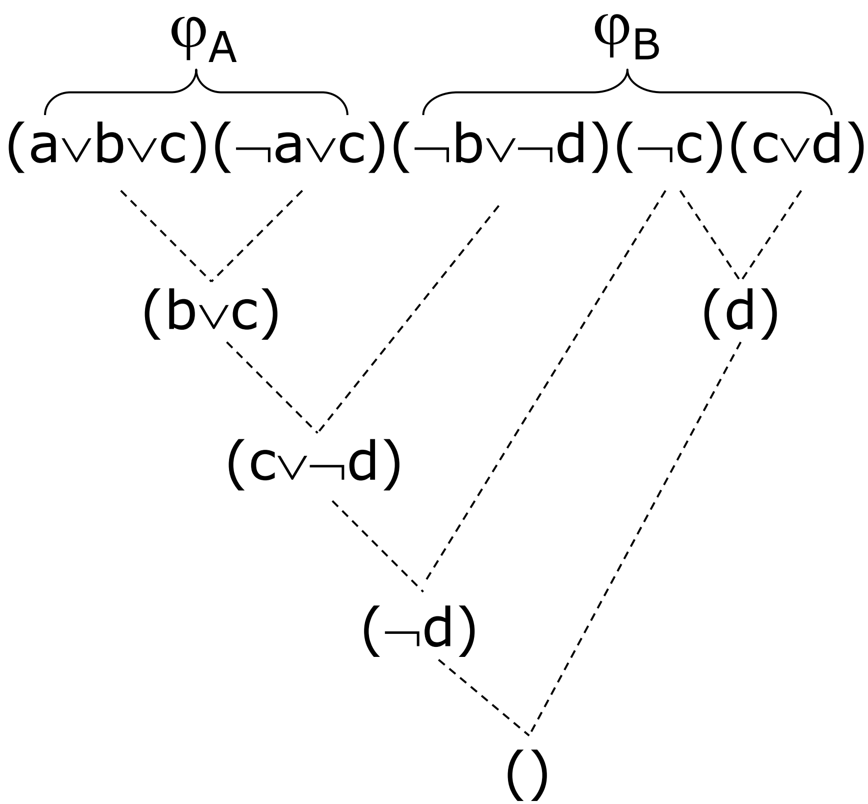
  1. $A \Rightarrow I$
  2. $I \wedge B$ is UNSAT
  3. I refers only to the common variables of A and B

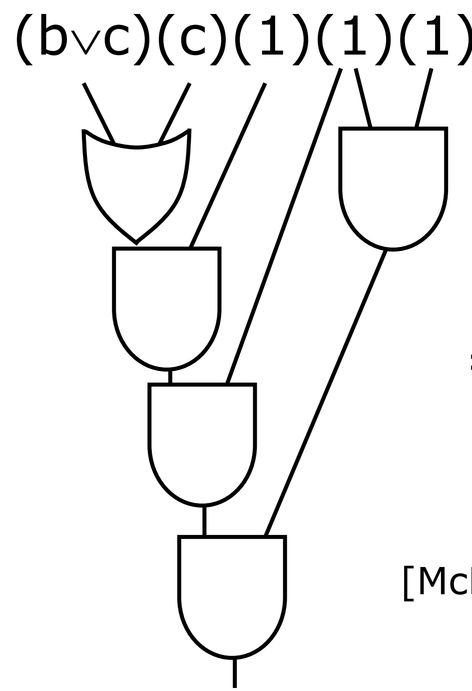


I is an abstraction of A

Interpolant and Resolution Proof

- SAT solver may produce the resolution proof of an UNSAT CNF φ
- For $\varphi = \varphi_A \wedge \varphi_B$ specified, the corresponding interpolant can be obtained in time linear in the resolution proof



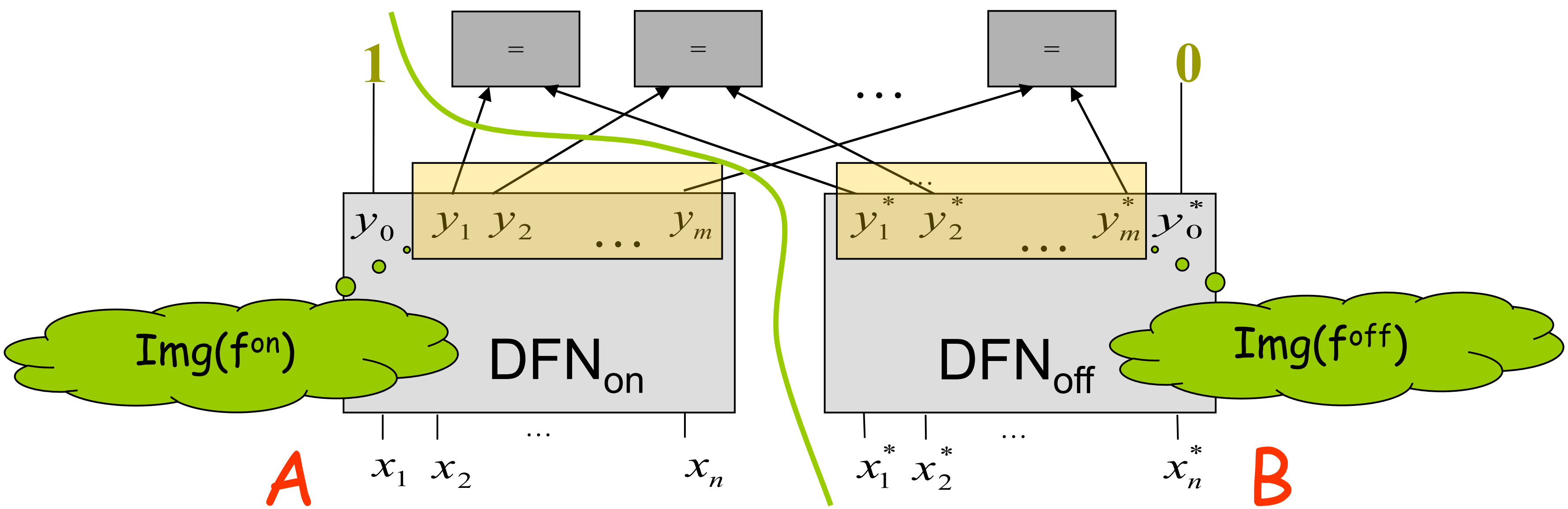


$= (b \vee c)$

[McMillan, 2003]

SAT-Based Computation

- Clause set A: C_{DFNon}, y_0
- Clause set B: C_{DFNoff}, $\neg y_0^*$, $(y_i \equiv y_i^*)$ for i =1,…,m
- I is an overapproximation of Img(f^{on}) and is disjoint from Img(f^{off})
- I only refers to $y_1, \ldots, y_m$
- Therefore, I corresponds to a feasible implementation of h



Incremental SAT Solving

- Controlled equality constraints
  $(y_i \equiv y_i^*) \rightarrow (\neg y_i \vee y_i^* \vee \alpha_i)(y_i \vee \neg y_i^* \vee \alpha_i)$
  with auxiliary variables α_i

  α_i = true $\Rightarrow$ i^{th} equality constraint is disabled

  - Fast switch between target and base functions by unit assumptions over control variables
  - Fast enumeration of different base functions
  - Share learned clauses

SAT vs. BDD

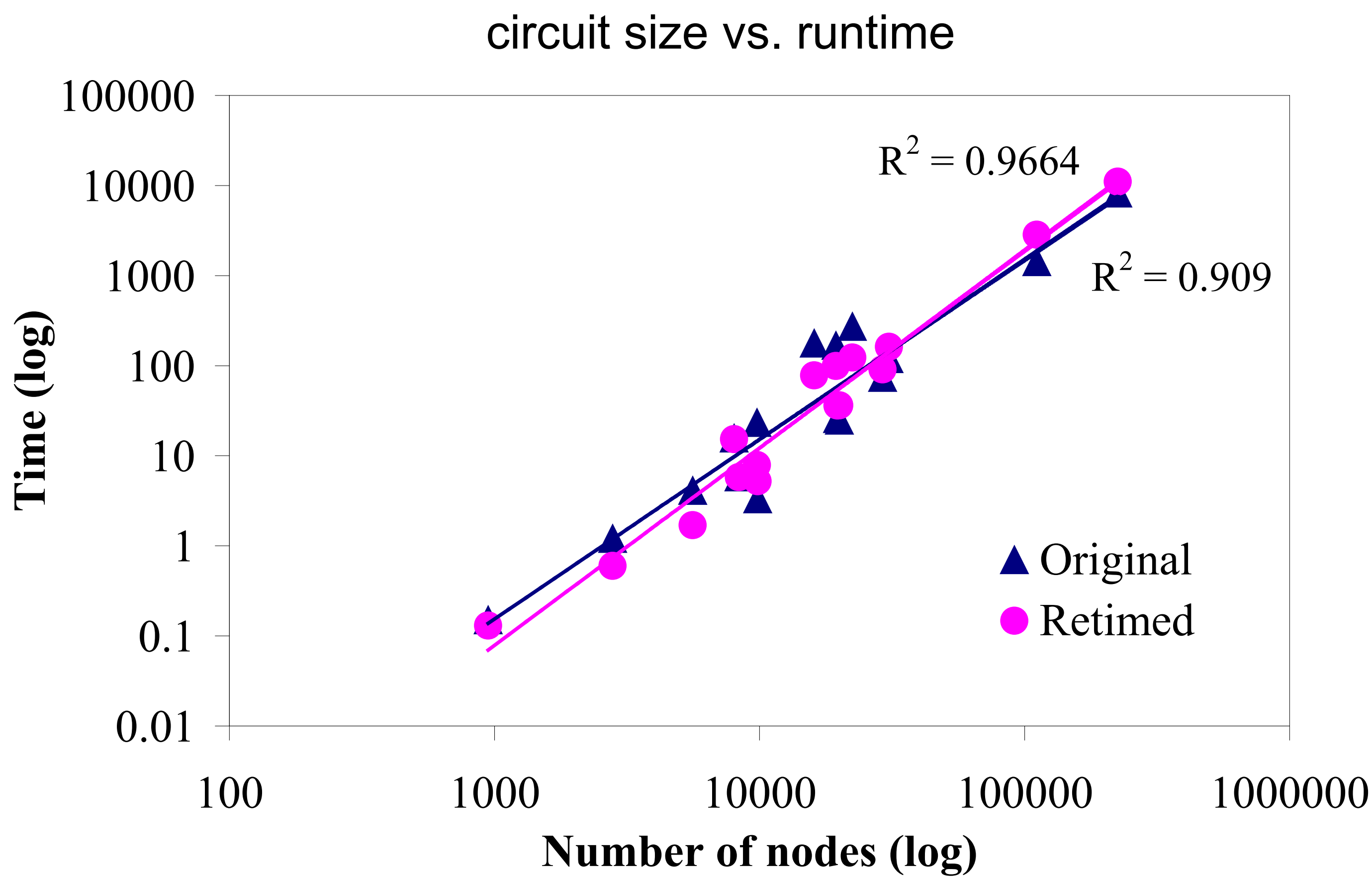
- SAT
  - Pros
    - Detect multiple choices of G automatically
    - Scalable to large $|G|$
    - Fast enumeration of different target functions f
    - Fast enumeration of different base functions G
  - Cons
    - Single feasible implementation of h
- BDD
  - Cons
    - Detect one choice of G at a time
    - Limited to small $|G|$
    - Slow enumeration of different target functions f
    - Slow enumeration of different base functions G
  - Pros
    - All possible implementations of h

Practical Evaluation

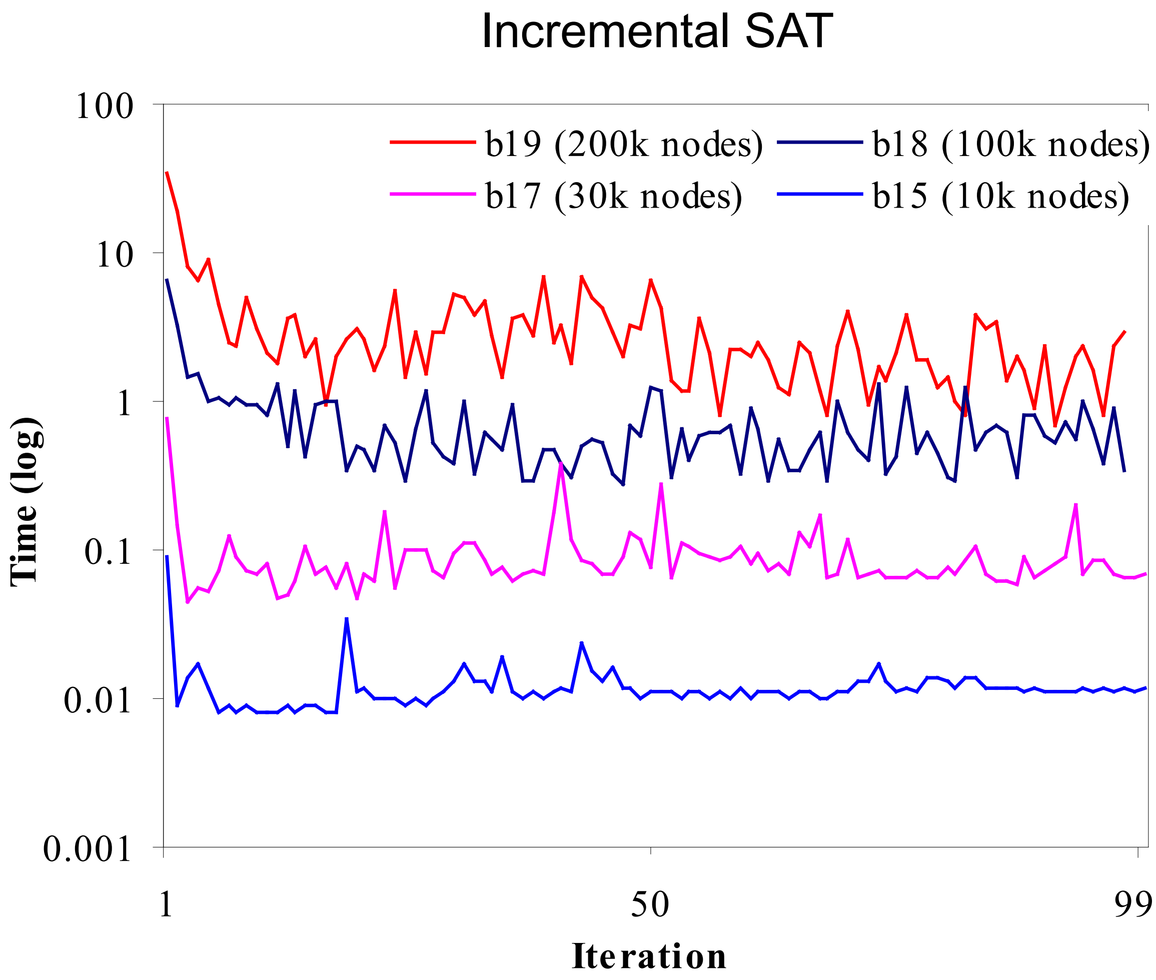
SAT vs. BDD

		Original			Retimed			SAT (original)		BDD (original)		SAT (retimed)		BDD (retimed)	
Circuit	#Nodes	#FF.	#Dep-S	#Dep-B	#FF.	#Dep-S	#Dep-B	Time	Mem	Time	Mem	Time	Mem	Time	Mem
s5378	2794	179	52	25	398	283	173	1.2	18	1.6	20	0.6	18	7	51
s9234.1	5597	211	46	x	459	301	201	4.1	19	x	x	1.7	19	194.6	149
s13207.1	8022	638	190	136	1930	802	x	15.6	22	31.4	78	15.3	22	x	x
s15850.1	9785	534	18	9	907	402	x	23.3	22	82.6	94	7.9	22	x	x
s35932	16065	1728	0	--	2026	1170	--	176.7	27	1117	164	78.1	27	--	--
s38417	22397	1636	95	--	5016	243	--	270.3	30	--	--	123.1	32	--	--
s38584	19407	1452	24	--	4350	2569	--	166.5	21	--	--	99.4	30	1117	164
b12	946	121	4	2	170	66	33	0.15	17	12.8	38	0.13	17	2.5	42
b14	9847	245	2	--	245	2	--	3.3	22	--	--	5.2	22	--	--
b15	8367	449	0	--	1134	793	--	5.8	22	--	--	5.8	22	--	--
b17	30777	1415	0	--	3967	2350	--	119.1	28	--	--	161.7	42	--	--
b18	111241	3320	5	--	9254	5723	--	1414	100	--	--	2842.6	100	--	--
b19	224624	6642	0	--	7164	337	--	8184.8	217	--	--	11040.6	234	--	--
b20	19682	490	4	--	1604	1167	--	25.7	28	--	--	36	30	--	--
b21	20027	490	4	--	1950	1434	--	24.6	29	--	--	36.3	31	--	--
b22	29162	735	6	--	3013	2217	--	73.4	36	--	--	90.6	37	--	--

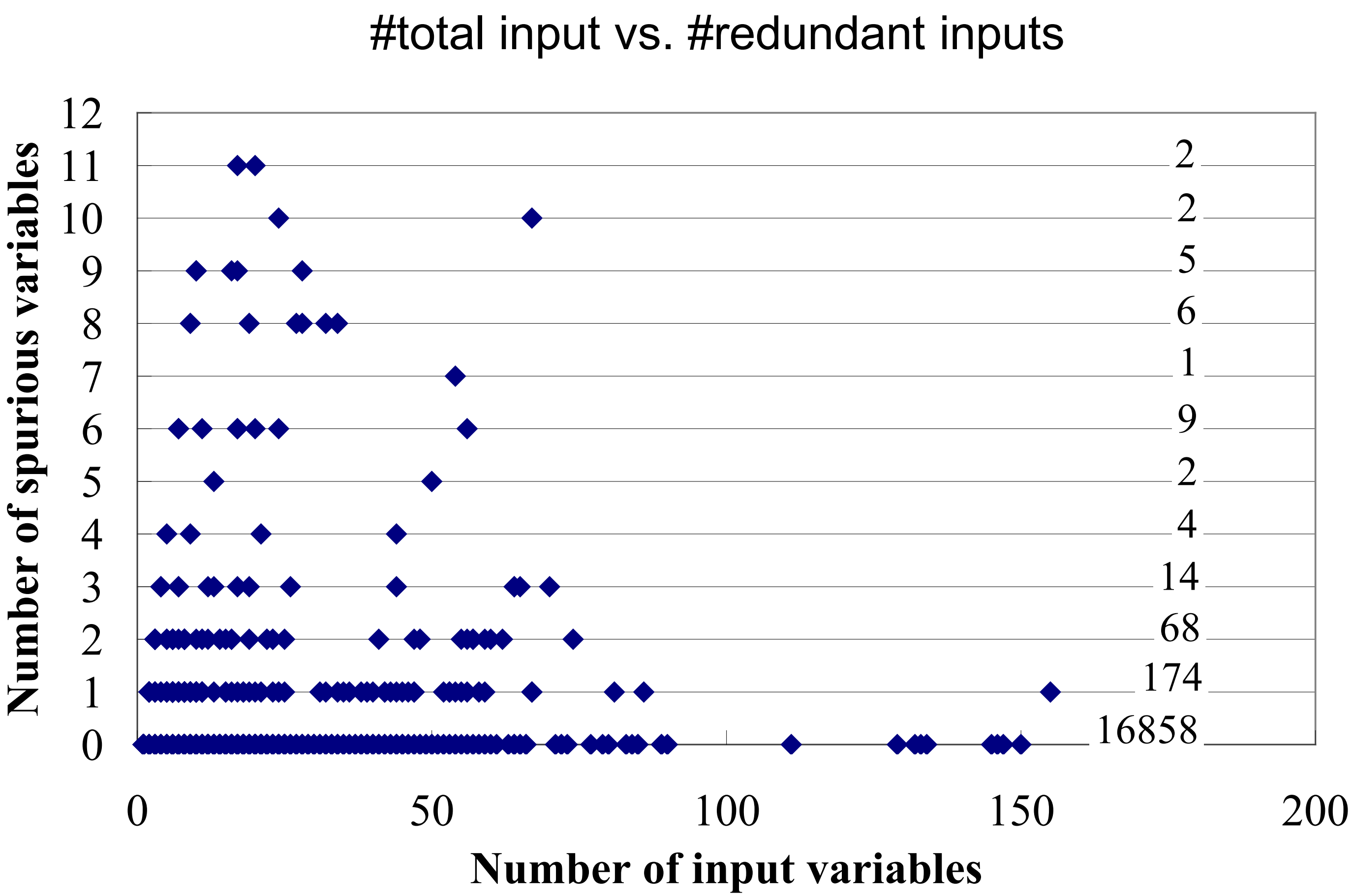
Practical Evaluation



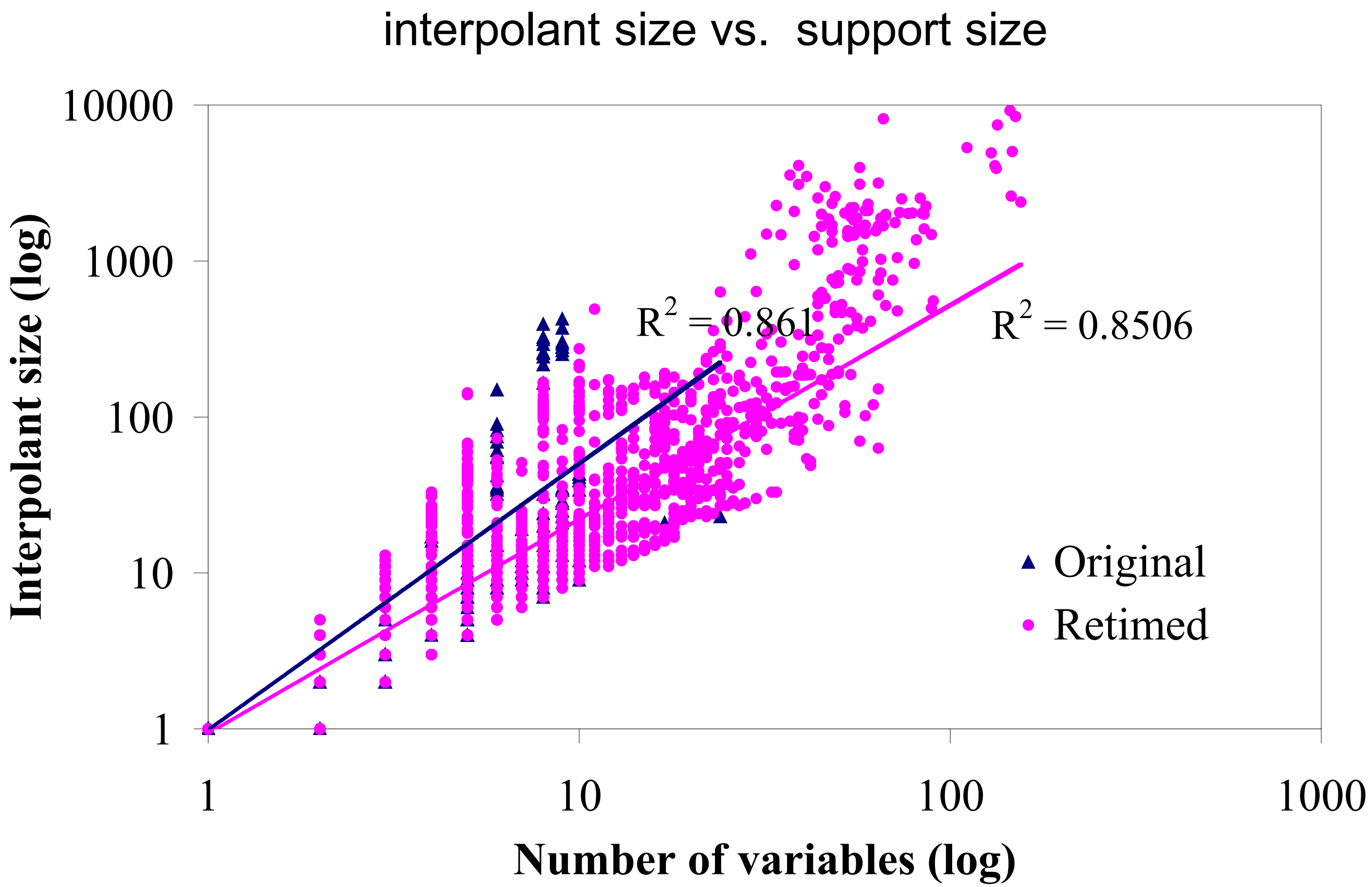
Practical Evaluation



Practical Evaluation



Practical Evaluation



Summary

- Functional dependency is computable with pure SAT solving (with the help of Craig interpolation)
- Compared to BDD-based computation, it is much scalable to large designs