# Logic Synthesis and Verification

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Fall 2010

---

# Multi-Level Logic Minimization

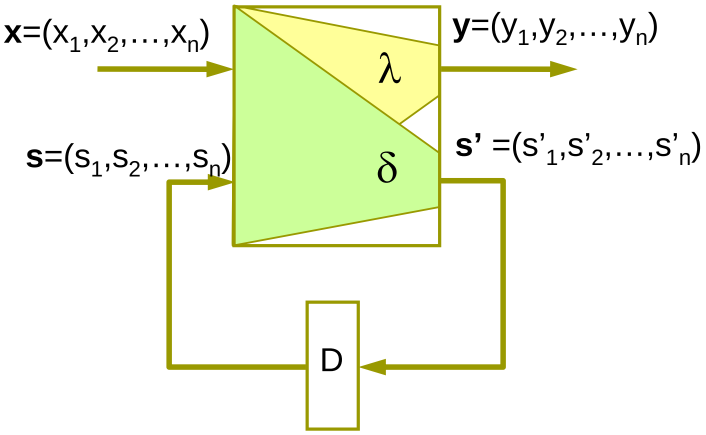
Reading:

*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
- $\delta$: $X \times Q \rightarrow Q$ (next state *function*)
- $\lambda$: $X \times Q \rightarrow Y$ (output *function*)

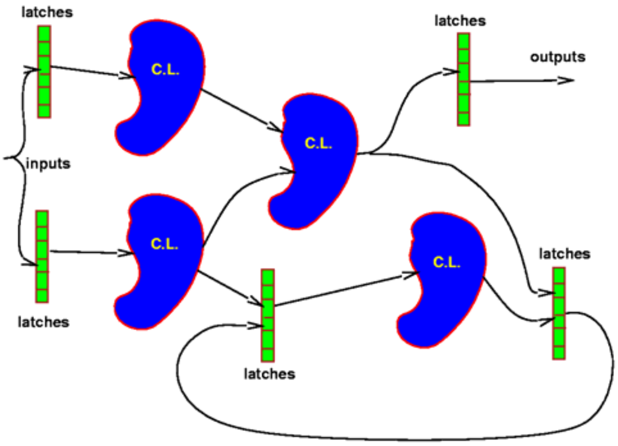
Delay element:

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

---

# 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:

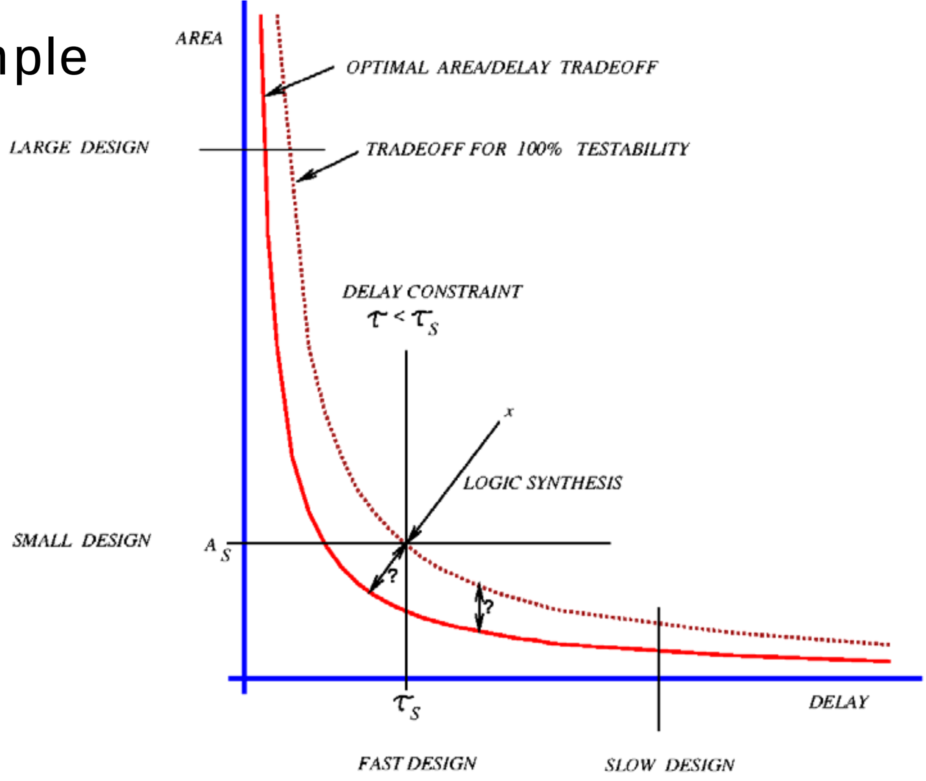
1. Area occupied by the logic gates and interconnect (approximated by literals = transistors in technology independent optimization)
2. Critical path delay of the longest path through the logic
3. Degree of testability of the circuit, measured in terms of the percentage of faults covered by a specified 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

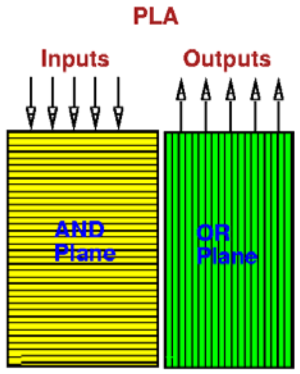
---

# Area-Delay Trade-off

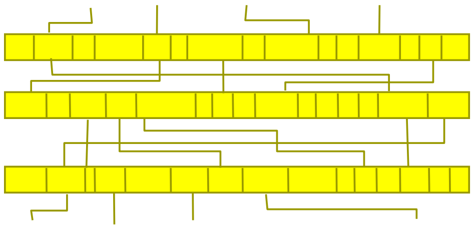
- Example



# Two-Level (PLA) vs. Multi-Level



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

---

# 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

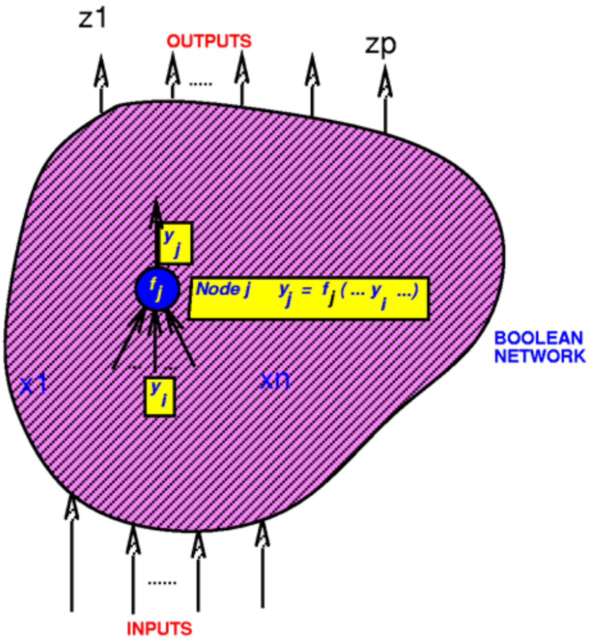
# Transformation-based Synthesis

- 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<br>- control/datapath analysis |
| Technology Independent Optimizations | - basic logic restructuring<br>- crude measures for goals |
| Technology Mapping | - use logic gates from target cell library |
| Technology Dependent Optimizations | - timing optimization<br>- physically driven optimization |
| Test Preparation | - improve testability<br>- test logic insertion |

---

# 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
  - spitting nodes
  - removing/changing connections between nodes

- Node representation:
  - keep size bounded to avoid blow-up of local transformations
    - SOP, POS
    - BDD
    - Factored forms

# Sum-of-Products (SOP)

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

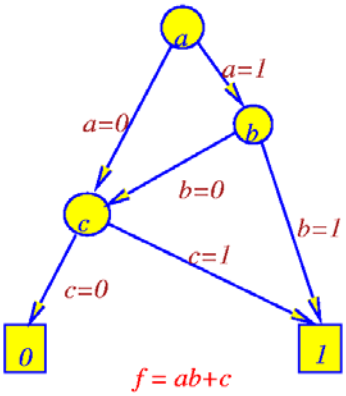
- Advantages:
  - Easy to manipulate and minimize
  - many algorithms available (e.g. AND, OR, TAUTOLOGY)
  - two-level theory applies

- 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

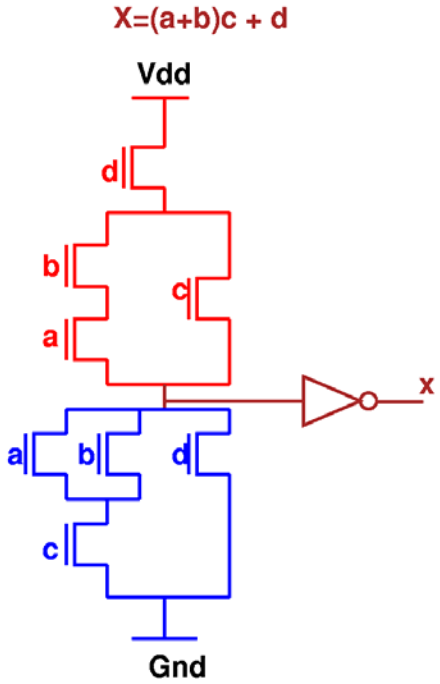
  $(ad+b'c)(c+d'(e+ac'))+(d+e)fg$

- Advantages
  - good representative of logic complexity
  - $f=ad+ae+bd+be+cd+ce$
  - $f'=a'b'c'+d'e' \Rightarrow f=(a+b+c)(d+e)$
  - 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

---

# Factor Form



Note:

literal count $\approx$ transistor count $\approx$ 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 $fact\_val(F,G_2) = 2\times3+2\times5=16$
    - The above factored form saves 17 literals, not 16. The extra literal comes from recursively applying the formula to the factored form of $G_1$.

# 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

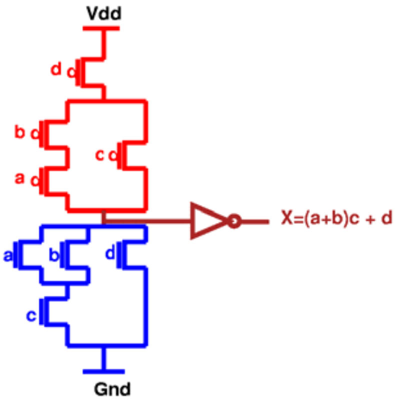
---

# Factored Form

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

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

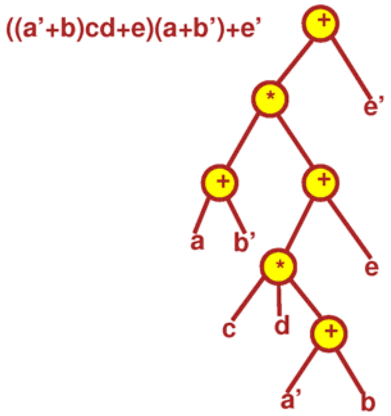
      $n$ literals in factored form and $(n/2)\times 2^{n/2}$ literals in SOP form



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.

# 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 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, $f \perp g$, if $sup(f) \cap sup(g) = \varnothing$

---

# 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 $f_{ij} \perp f_{kl}$, $\forall i \neq j$ 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.

---

# 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 - they 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

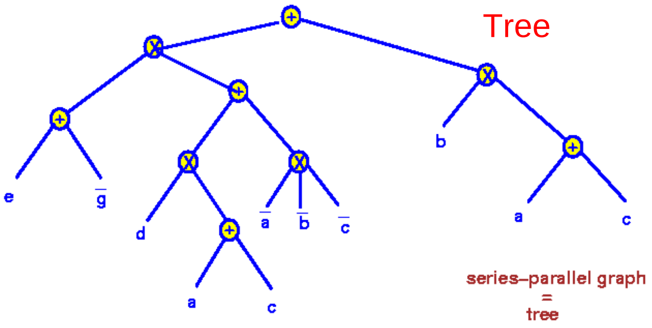
- Basic Operations:
  - Decomposition (single function)
    *f = abc+abd+a'c'd'+b'c'd'* ⇒
    *f = xy+x'y'   x = ab   y = c+d*
  - Extraction (multiple functions)
    *f = (az+bz')cd+e   g = (az+bz')e'   h = cde* ⇒
    *f = xy+e   g = xe'   h = ye   x = az+bz'   y = cd*
  - Factoring (series-parallel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

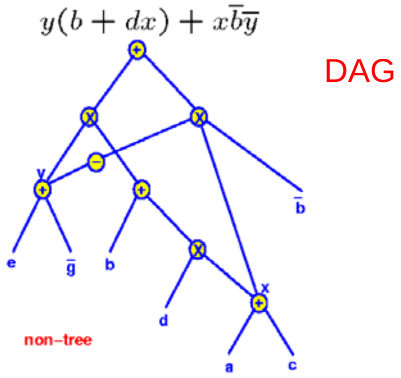
"Division" plays a key role in all of these operations

---

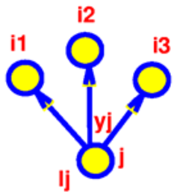
# Factoring vs. Decomposition

- 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.





# Structural Operation
# Node Elimination



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

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

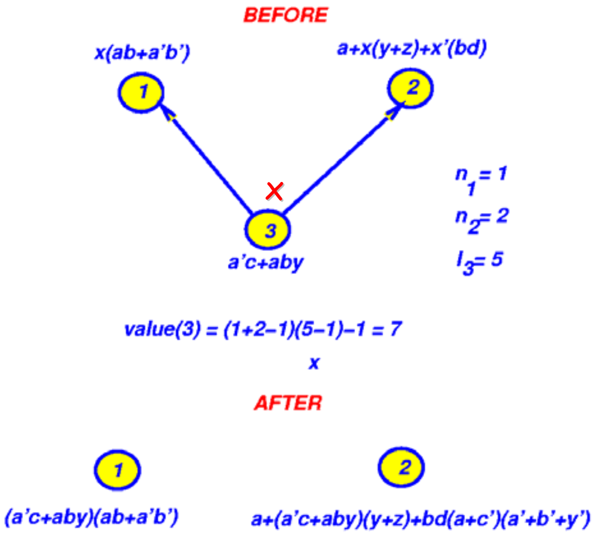
$$l_j \sum_{i \in FO(j)} n_i + c$$

*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

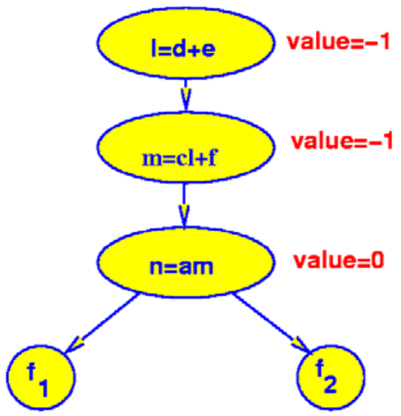


$$value(j) = \left( \sum_{i \in FO(j)} n_i \right) \left( l_j - 1 \right) - l_j$$

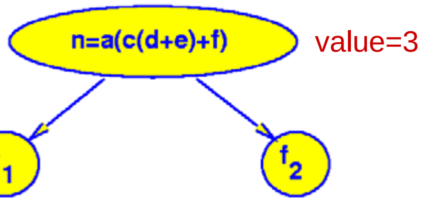
$$= (n_1 + n_2)(l_3 - 1) - l_3$$

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

# 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 ÷ G)

---

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