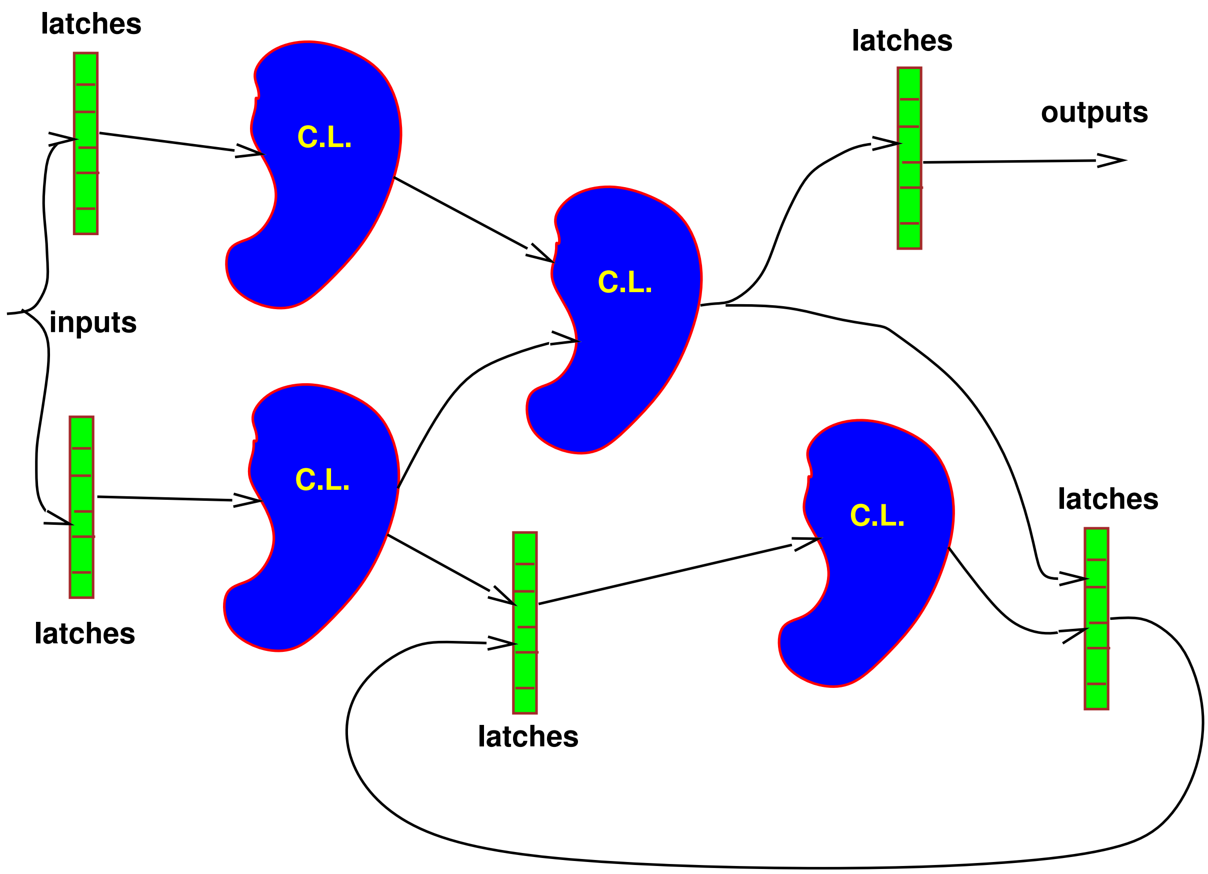
Multi-Level Logic Synthesis - Introduction

Outline

- Representation
  - Networks
  - Nodes
- Technology Independent Optimization
- Technology Dependent Optimization

STRUCTURED SYSTEM

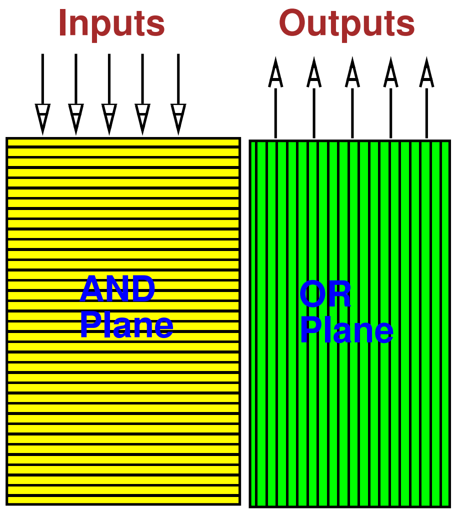
(combinational logic, memory, I/O)



- combinational optimization

- sequential optimization

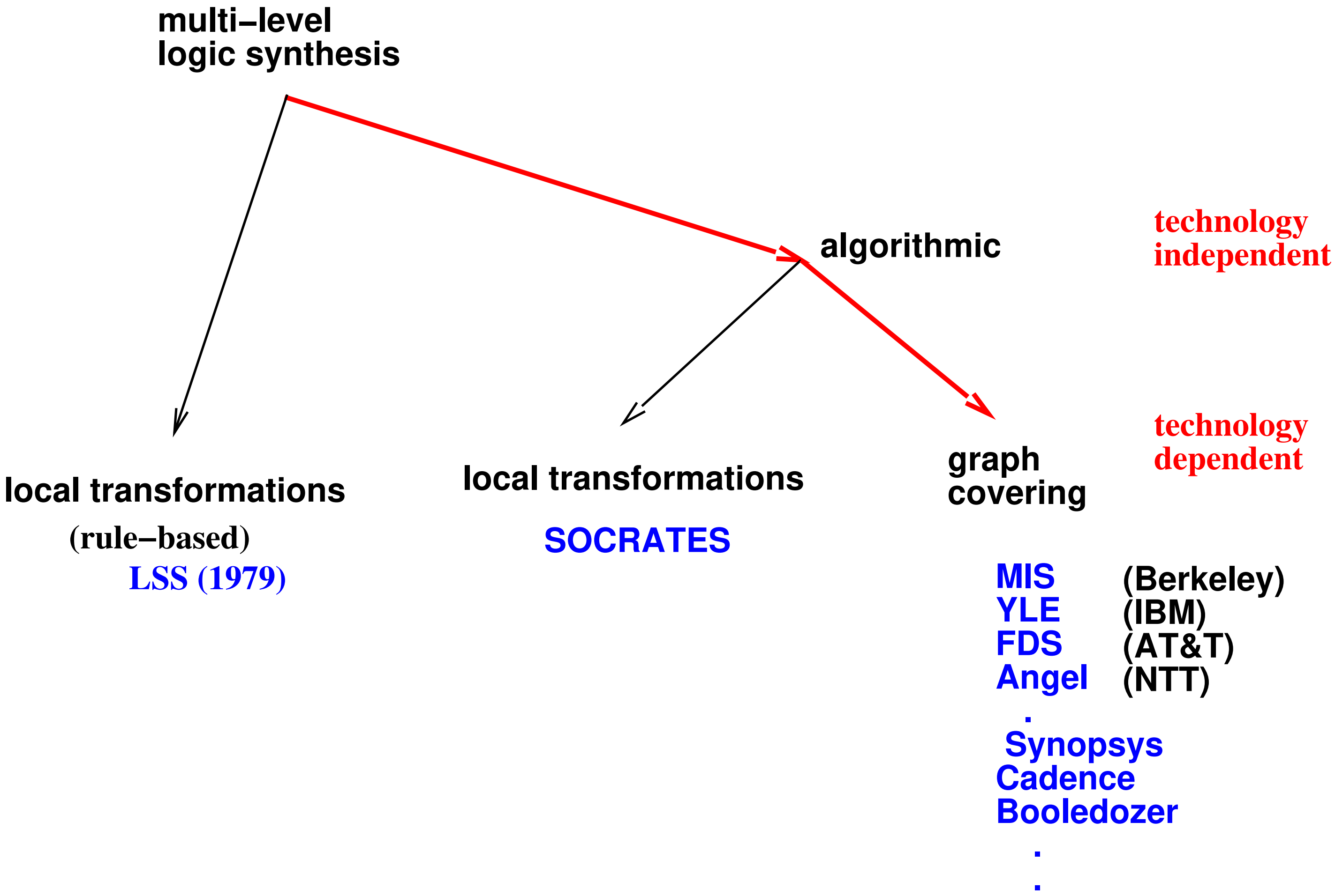
TWO-LEVEL (PLA) vs MULTI-LEVEL



PLA

PLA	multi-level
control logic	all logic
constrained layout	general
highly automatic	automatic
technology independent	partially t.i.
multi-valued logic	some results
slower	can be high speed
input, output, state encoding	some results

APPROACHES TO MULTI-LEVEL



Algorithmic Approach

- continues along lines of ESPRESSO and two-level minimization
- spectrum of speed/quality trade-off algorithms
- encourages development of understanding and theory

OPTIMIZATION COST CRITERIA

The accepted optimization criteria for multi-level logic are to **Minimize** some function of:

1. **area occupied by the logic gates and interconnect** (eg., approximated by **literals = transistors** in technology independent optimization).

2. the **Critical Path Delay** of the longest path through the logic.

3. the **Degree of Testability** of the circuit, measured in terms of the percentage of faults covered by a specified set of test vectors, for an appropriate fault model (eg., single stuck faults, multiple stuck faults, etc.).

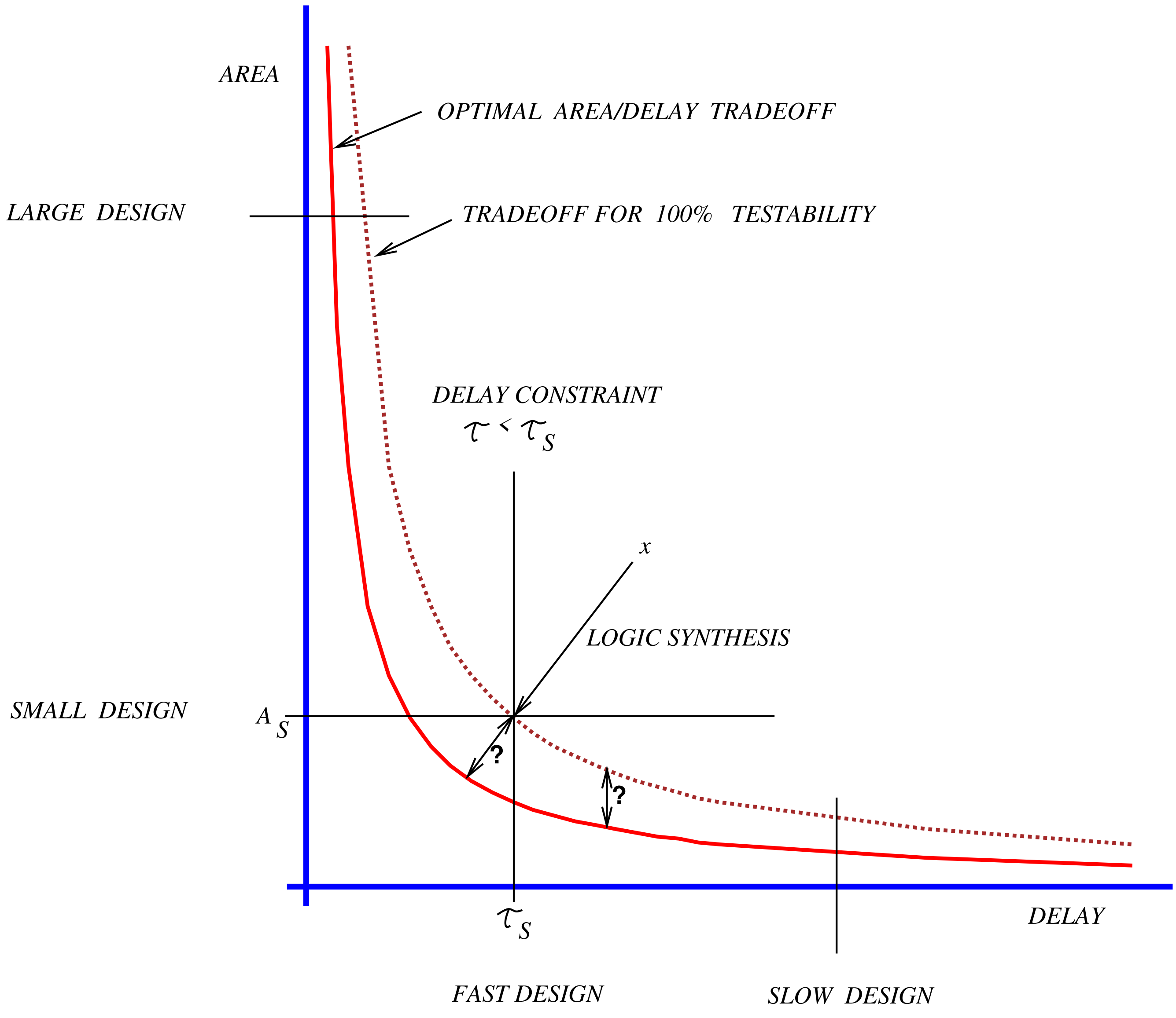
4. **Power** consumed by the logic gates.

5. **noise**

6. **wireability**

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

OPTIMIZATION COST CRITERIA



Multi-Level is Natural for High-Level Synthesis

Example:
$w = ab + \overline{a}\overline{b}$
If w, then $z = cd + \overline{a}\overline{d}$; $u = cd + \overline{a}\overline{d} + e(f+b)$
else $z = e(f+b)$; $u = (cd + \overline{a}\overline{d})e(f+b)$

A:

$$
\begin{aligned}
w &= ab + \overline{a}\overline{b} \\
z &= w(cd + \overline{a}\overline{d}) + \overline{w}e(f+b) \\
u &= w(cd + \overline{a}\overline{d} + e(f+b)) + \overline{w}(cd + \overline{a}\overline{b})e(f+b)
\end{aligned}
$$

B:

$$
\begin{aligned}
w &= ab + \overline{a}\overline{b} \\
t &= cd + \overline{a}\overline{d} \\
s &= e(f+b) \\
z &= w\overline{t} + \overline{w}s \\
u &= w(t+s) + \overline{w}ts
\end{aligned}
$$

.

NETWORK REPRESENTATION

In implementing multi-level logic our first aim is to:
Establish a structure on which to develop

1. theory and
2. algorithms

independent of technology on which manipulations can be made and optimization **progress can be well estimated**.

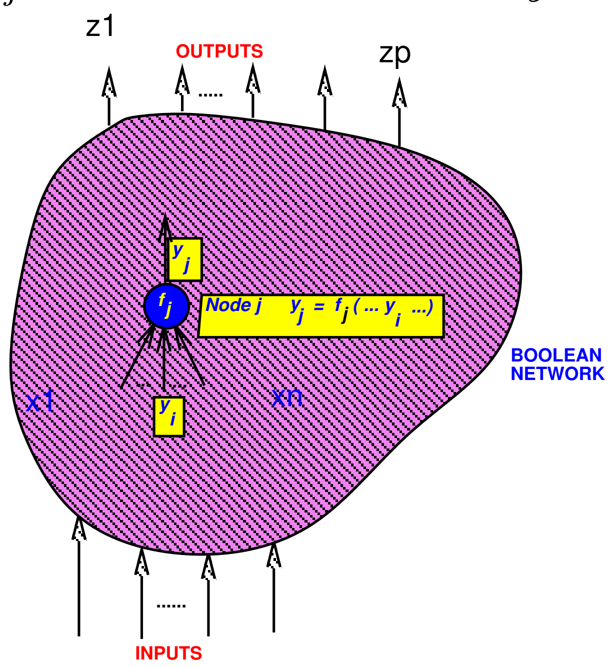
This leads to two abstractions:

- Boolean Network
- Factored Forms

NETWORK REPRESENTATION

Boolean Network:

- DAG
- node logic function representation $f_j(x, y)$
- node variable y_j: $y_j = f_j(...y_i...)$
- edge (i, j) if f_j depends explicitly on y_i



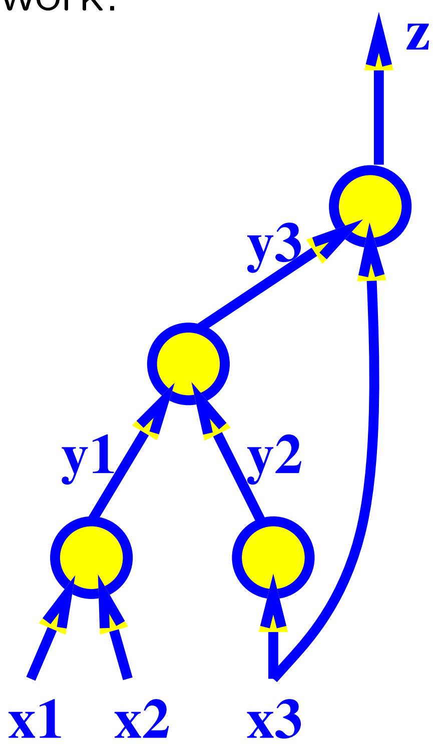
Inputs: $x = (x_1, ..., x_n)$

Outputs: $z = (z_1, ..., z_p)$

External Don't Cares: $d_1(x), ..., d_p(x)$

NETWORK REPRESENTATION

Boolean Network:



y1 = x1 x2
y2 = x3'
y3 = y1 y2
z = y3 + x3

NODE REPRESENTATION

Some Choices:

- Merged view (technology and network merged)
- Separated view
  - technology dependent
  - technology independent
    - node representation
      General node
      Generic node
      Discrete node

NODE REPRESENTATION

Merged view: (more accurate estimators, slower, more constrained)

- Each node is a valid "gate" chosen from a library of gates to be used in the final implementation.
- Representation and implementation are one.

Separated view: (less accurate estimators, faster, more general - lends itself to more theory

- Two representations allowed.
- Technology independent, i.e. does not have any connection with the building blocks of final implementation.
- Technology dependent which uses only "valid" gates (i.e. found in a cell library or meeting some criterion).
- In the separated, technology independent view, there are several choices.

NODE REPRESENTATION

Separated, technology independent view, general node:

- Each node can be a representation of an **arbitrary** logic function.
- A theory is easier to develop since there are no arbitrary restrictions dependent on the technology.
- **THE CHOICE OF SIS** (includes all others as special cases, except for multiple-output nodes).

NODE REPRESENTATION

Separated, technology independent view, **generic node:**

- Every node in the network is a simple generic node, like a 2-input NAND gate.
- Some manipulations are much faster using this structure.
- But the network is finely decomposed in a particular way, and some natural structures may be lost.

Separated, technology independent view, **discrete node:**

- A node can be one of a small set of logic functions, such as AND, OR, NOT, DECODE, ADD.
- Multiple output nodes are also allowed.
- Used mostly in rule-based systems.
- Complex blocks of logic can be manipulated as a single unit.
- A theory for such networks is more difficult.

NODE REPRESENTATION

Separated view, general node, choices of function representation:

- Sum-of-products form
- Factored form
- Binary decision diagram (BDD)

SUM-OF-PRODUCTS

Example: $ab\overline{c} + \overline{a}bd + \overline{bd} + \overline{b}\overline{e}f$ (sum of cubes)

Advantages:

- easy to manipulate and minimize
- many algorithms available (AND, OR, TAUTOLOGY, ...)
- Two-level theory applies

Disadvantages:

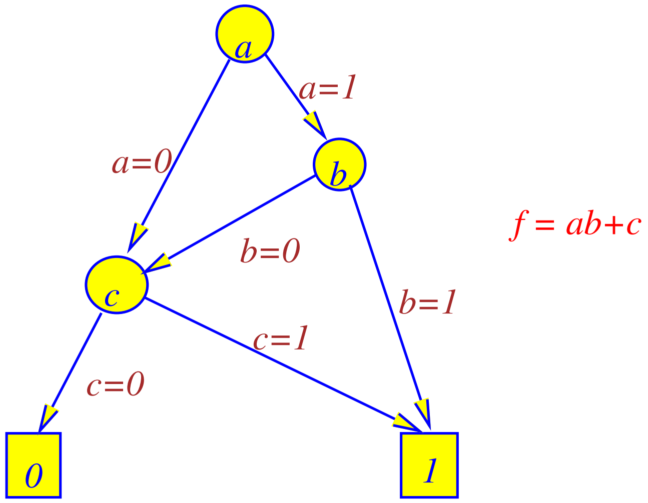
- not representative of logic complexity
  example:

$$\begin{aligned} f &= ad + ae + bd + be + cd + ce \\ \overline{f} &= \overline{a}\overline{b}\overline{c} + \overline{d}\overline{e} \end{aligned}$$

  These differ in their implementation only by one invertor.

- hence not easy to estimate if logic is becoming simpler - i.e. not a good estimator of progress during logic manipulation.

REDUCED ORDERED BDD's



- like factored form, represents both function and complement

- like network of muxes, but restricted since controlled by primary input variables. Not really a good estimator for complexity of implementation.

- given ordering, **reduced** BDD is canonical, hence a good replacement for truth tables.

- for a good ordering, ROBDD's remain reasonably small for complicated functions (except multipliers)

- manipulations are well defined, efficient. True support (dependency) is displayed.

FACTORED FORMS

Example: $(ab + \overline{b}c)(c + \overline{d}(e + a\overline{c})) + (d + e)(fg)$
Advantages:

- representative of logic complexity. Example:

$$\begin{aligned} f &= ad + ae + bd + be + cd + ce \\ \overline{f} &= \overline{a}\overline{b}\overline{c} + \overline{d}\overline{e} \\ f &= (a + b + c)(d + e) \end{aligned}$$

- In many design styles (e.g. complex-gate CMOS) the implementation of a function corresponds directly to its factored form.

- good estimator of complexity of logic implementation.

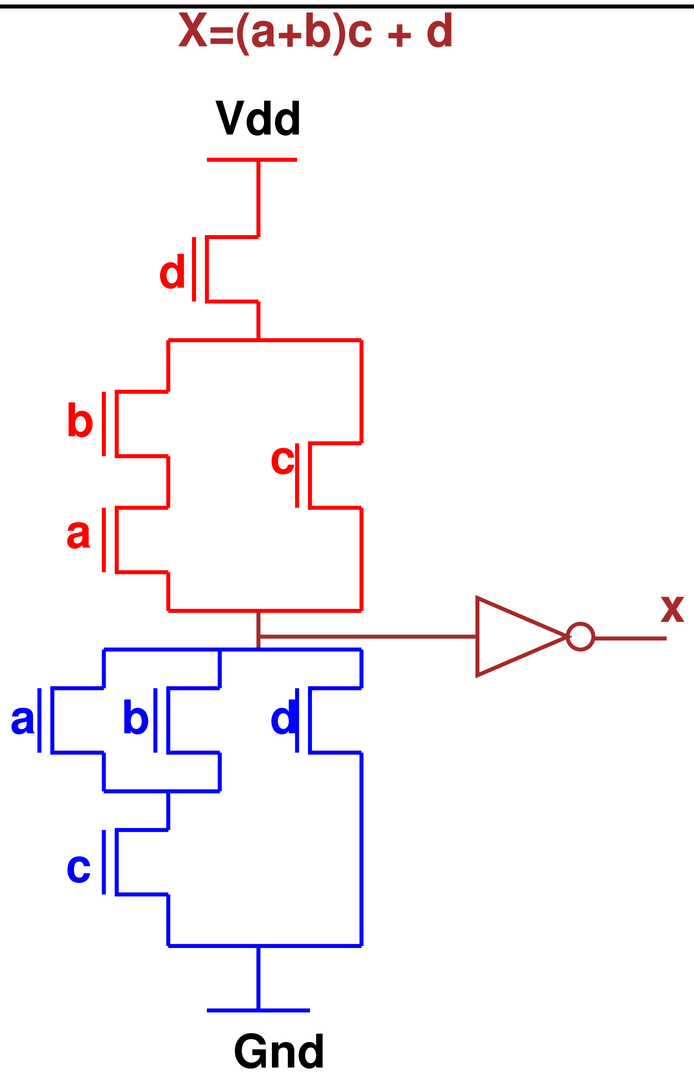
- doesn't blow up so easily

Disadvantages:

- not as many algorithms available for manipulation

- hence usually must convert into SOP before manipulation

FACTORED FORMS



Note, factored form literal count $\propto$ transistor count $\propto$ area. (However, area also depends on wiring).

FACTORED FORMS

DEFINITION 1 *f is an* **algebraic expression** *if f is a set of cubes (SOP) such that no one cube contains another (minimal wrt single cube containment).*

Example: $a + ab$ is not an algebraic expression.
(This is ruled out because in factoring it we would get $a(1+b)$.)

DEFINITION 2 *The* **product** *of two expressions f and g is a set defined by*

$$fg = \{cd \mid c \in f \ and \ d \in g \ and \ cd \neq \emptyset\}.$$

Example:
$(a+b)(c+d+\overline{a}) = ac + ad + bc + bd + \overline{a}b$

DEFINITION 3 *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 1 $(a+b)(c+d) = ac + ad + bc + bd$
is an algebraic product

FACTORED FORMS

A factored form is a parenthesized algebraic expression.

DEFINITION 4 *A* **factored form** *can be defined recursively by the following rules:*
A factored form is either a product or a 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.*

In effect, a factored form is a product of sums of products of ... or a sum of products of sums of ....

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

FACTORED FORMS

- Examples of factored forms:

$$\begin{aligned}&x,\\&y',\\&abc',\\&a+b'c,\\&((a'+b)cd+e)(a+b')+e'.\end{aligned}$$

- The following is not a factored form: (complementation is not allowed - complements only allowed on literals)

$$(a+b)'c.$$

- Example of three equivalent factored forms. (Factored Forms are not unique):

$$\begin{aligned}&ab+c(a+b),\\&bc+a(b+c),\\&ac+b(a+c).\end{aligned}$$

FACTORED FORMS

DEFINITION 5 *The* **factorization value** *of an algebraic factorization* $F = G_1 G_2 + R$ *is defined to be*

$$\begin{aligned} \textit{fact\_val}(F, G_2) &= \textit{lits}(F) - (\textit{lits}(G_1) + \textit{lits}(G_2) + \textit{lits}(R)) \\ &= (\mid G_1 \mid -1)\textit{lits}(G_2) + (\mid G_2 \mid -1)\textit{lits}(G_1), \end{aligned}$$

assuming G_1, G_2, *and* R *are algebraic expressions. Here, for any algebraic expression* P, *lits*(P) = *Number of literals in SOP form of* P.

FACTORED FORMS

The algebraic expression

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

can be expressed in the factored form

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

Note that only 7, rather than 24 literals are required.

If $G_1 = (a + bc + bd)$, and $G_2 = (e + f + g)$, then $R = \emptyset$, and fact_val$(F, G_2) = (2)(3) + (2)(5) = 16$. Note the given factored form saved 17, rather than 16 literals. The extra literal saved comes from recursively applying the formula to the factored form of G_1.

FACTORED FORMS

- Factored forms are more compact representations of logic functions than the traditional sum-of-products forms. For example, the factored form

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

  when represented as a sum-of-products form is

  $$\begin{array}{l} ac + ade + adfg + adfh + adfi + adfj \\ +bc + bde + bdfg + bdfh + bdfi + bdfj. \end{array}$$

- Of course every sum-of-products form can be viewed as a factored form.

FACTORED FORMS

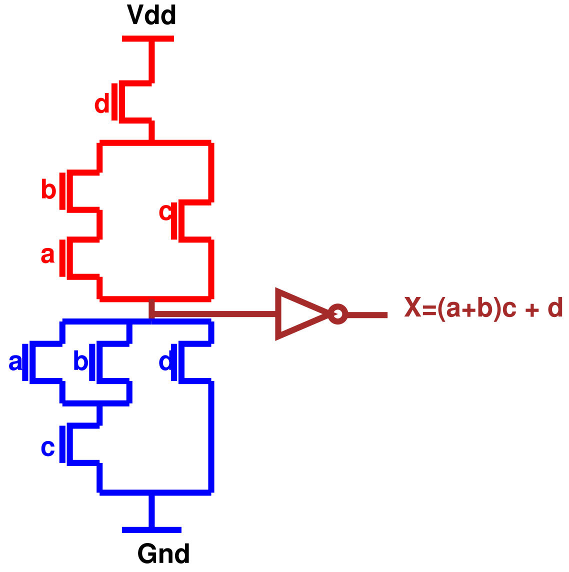
When measured in terms of number of inputs, there are functions whose size is exponential in sum-of-products forms, but polynomial in factored forms.

Example: Achilles' heel function

$$\prod_{i=1}^{i=n/2} (x_{2i-1} + x_{2i}).$$

There are n literals in the factored form and $(n/2) \times 2^{n/2}$ literals in the sum-of-products form.

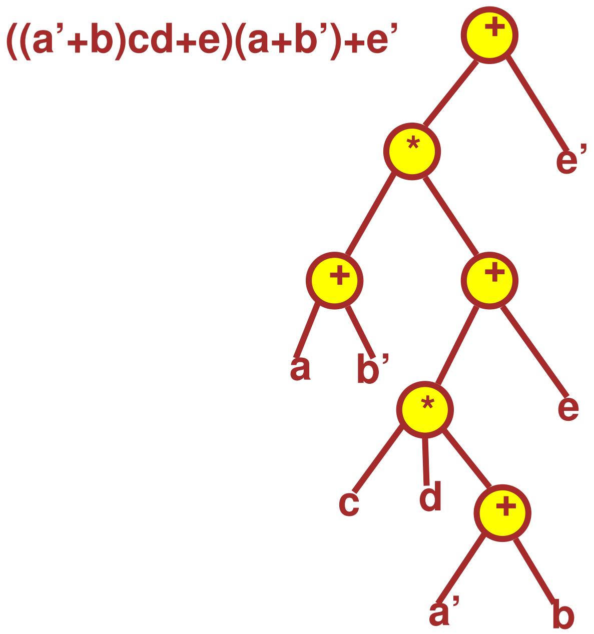
Factored forms are useful in estimating area and delay in a multi-level logic synthesis and optimization system. In most design styles (for example, complex-gate CMOS design) the implementation of a function corresponds directly to its factored form.



FACTORED FORMS

Factored forms can also be graphically represented as labeled trees, called factoring trees, in which each internal node including the root has a label of either "+" or "*", and each leaf has a label of either a variable or its complement.

Factoring tree of $((a' + b)cd + e)(a + b') + e'$:



FACTORED FORMS

The **size** of a factored form F (denoted $\rho(F)$) is the number of literals in the factored form.

Example:

$\rho((a+b)ca') = 4$ and $\rho((a+b+cd)(a'+b')) = 6$.

A factored form is **optimum** if no other equivalent factored form has less literals.

A factored form is **positive unate** in x if x appears in F but x' does not.
negative unate in x if x' appears in F and x does not.
F is **unate** in x if it is either positive unate or negative unate in x.
F is **binate** in x if it is not unate in x.

Example:

$(a+b')c+a'$ is positive unate in c, negative unate in b and binate in a.

COFACTOR of FACTORED FORMS

The **cofactor** of a factored form F with respect to a literal x_1 (or $\bar{x}_1$) is the factored form $F_{x_1} = F|_{x_1=1}(x)$ (or $F_{\overline{x}_1} = F|_{\overline{x}_1=1}(x)$) obtained by

1. replacing all occurrences of x_1 with 1 and all occurrences of $\overline{x}_1$ with 0 and

2. simplifying the factored form using the following identities of the Boolean algebra:

$$\begin{aligned} 1x &= x \\ 1 + x &= 1 \\ 0x &= 0 \\ 0 + x &= x \end{aligned}$$

   After constant propagation (all constants are removed), part of the factored form may appear as $G + G$. In general, G is another factored form. In fact, the two G's may have different factored forms.

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

Example 2 *Let* $F = (x + \overline{y} + z)(\overline{x}u + \overline{z}\overline{y}(v + \overline{u}))$ *and* $c = v\overline{z}$. *Then*

$$\begin{aligned} &F_{\overline{z}} = (x + \overline{y})(\overline{x}u + \overline{y}(v + \overline{u})) \\ &F_c = (x + \overline{y})(\overline{x}u + \overline{y}) \end{aligned}$$

FACTORED FORMS

Sum-of-products forms are used as the internal representation of logic functions in most multi-level logic optimization systems.

1. good algorithms for manipulating them are available.

Cons:

1. performance is unpredictable. They may accidentally generate a function whose sum-of-products form is too large.

2. Factoring algorithms have to be used constantly to provide an estimate for the size of the Boolean network. Time spent on the factoring may become significant.

FACTORED FORMS

Possible Solution:

- avoid sum-of-products forms by using factored forms as internal representation.
- But not practical unless we know how to perform logic operations on the factored forms directly without converting to sum-of-products forms.
- Extensions to factored forms of the most common logic operations have been partially provided. More research needed.

OPTIMUM FACTORED FORMS

DEFINITION 6 *Let f be a completely specified Boolean function, $\rho(f)$ be the minimum number of literals in any factored form of f.*

Recall that $\rho(F)$, when F is a factored form, is the number of literals in F.

DEFINITION 7 *Let $sup(f)$ be the true variable support of f, i.e. set of variables f depends on. Two functions f and g are said to be* **orthogonal**, *denoted by $f \perp g$, if*

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

.

LEMMA 8 *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 respectively. Since $G + H$ is a factored form of f,

$$\rho(F) \leq \rho(G) + \rho(H)$$

Let c be a minterm of $\overline{g}$. Since g and h have disjoint support, we have

$$f_c = (g+h)_c = g_c + h_c = \overline{\overline{g}_c} + h_c = \overline{1} + h_c = 0 + h_c = h_c = h$$

Similarly if d is a minterm of $\overline{h}$, $f_d = g$. Because $\rho(h) = \rho(f_c) \leq \rho(F_c)$ and $\rho(g) = \rho(f_d) \leq \rho(F_d)$,

$$\rho(g) + \rho(h) \leq \rho(F_c) + \rho(F_d).$$

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

$$\rho(F_c) + \rho(F_d) \leq n + m = \rho(F)$$

Putting everything together, we have

$$\rho(f) \leq \rho(g) + \rho(h) \leq \rho(F_c) + \rho(F_d) \leq \rho(F) = \rho(f).$$

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

COROLLARY 9 *Let $f = gh$ such that $g \perp h$, then $\rho(f) = \rho(g) + \rho(h)$.*

Proof. Let $\overline{F}$ denotes the factored form obtained using DeMorgan's law. Then, we have $\rho(F) = \rho(\overline{F})$ and therefore $\rho(f) = \rho(\overline{f})$. Plus the above theorem we have

$$\rho(f) = \rho(\overline{f}) = \rho(\overline{g} + \overline{h}) = \rho(\overline{g}) + \rho(\overline{h}) = \rho(g) + \rho(h)$$

$\blacksquare$

THEOREM 10 *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 previous Lemma and Corollary. $\blacksquare$

OPTIMUM FACTORED FORMS

DEFINITION 11 *For an incompletely specified function* (f, d), *if no optimum factored form of* (f, d) *can contain variable* x, *then* x *is said to be redundant.*

LEMMA 12 *If* F *covers* (f, d), *then* F_c *covers* (f_c, d_c) *for any cube* c.

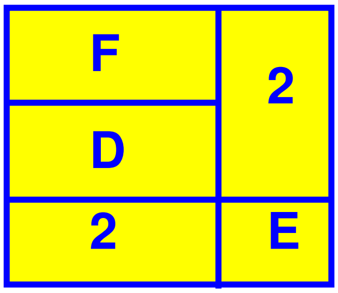
Proof. Easy induction on number of variables in c. ∎

THEOREM 13 *Let* $(f, d+e)$ *be an incompletely specified function such that* $sup(f + d) \cap sup(e) = 0$. *Then, variables in* $sup(e)$ *are redundant, which implies* $\rho(f, d + e) = \rho(f, d)$.

Proof. Let F be any optimum factored form of $(f, d + e)$. Assume $e \neq 1$, and let c be a minterm of $\overline{e}$. By previous Lemma, F_c covers $(f_c, d_c + e_c)$. But, $f_c = f$, $d_c = d$, and $e_c = 0$. So, F_c also covers (f, d). Since F is optimum, and $\rho(F_c) \leq \rho(F)$, we must have $\rho(f_c) = \rho(f)$, which implies that $sup(F) \cap sup(c) = sup(F) \cap sup(e) = 0$. Thus sup($e$) variables are redundant. ∎

OPTIMUM FACTORED FORMS

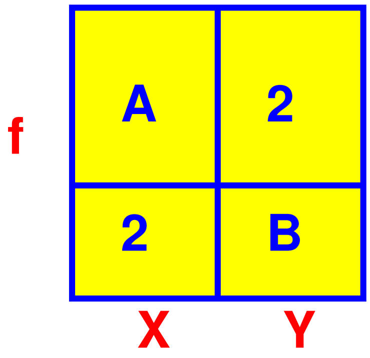
The above Theorem can be interpreted by the figure. If F, D, and E are represented as sum-of-product expressions in a positional cube notation and the matrix has the structure indicated in the figure, then the set of cubes in E can be removed from the don't-care set without affecting the final result.



OPTIMUM FACTORED FORMS

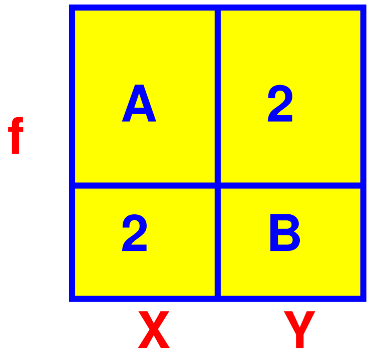
A natural question to ask at this point is: is E unique? If yes, how to find it? The following Theorem answers the questions.

THEOREM 14 *Let f be a Boolean function, if there is a sum-of-product representation whose cube matrix has the structure as indicated in the figure, where the support of A and B are X and Y respectively, then no prime p contains variables from both X and Y.*



Thus we just need to take DC and get a prime representation for it. If it has the above structure, then D and E are displayed.

OPTIMUM FACTORED FORMS



Proof. Let p be a prime of $f = A(x) + B(y)$. Suppose p contains variables from both X and Y. Let $p = p_x p_y$ where p_x and p_y are parts of p containing only variables from X and Y respectively. Since $p \subseteq f$, then

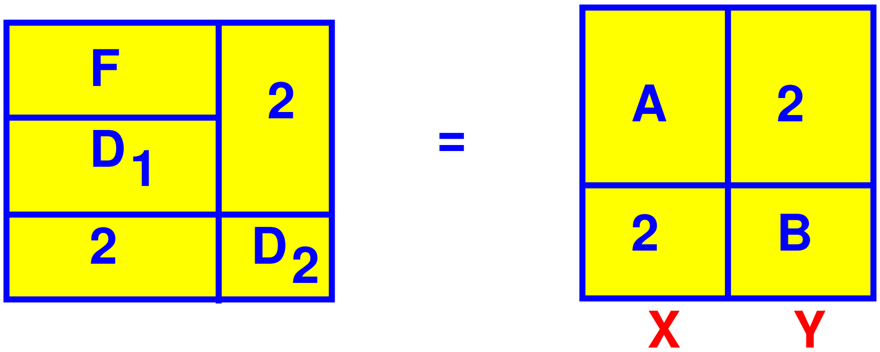
$$A_{p_x}(x) + B_{p_y}(y) \equiv 1$$

If $A_{p_x}(x) \equiv 1$ then $p_x \subseteq A \subseteq f$ contradicting that p is prime. Hence $B_{p_y}(y) \equiv 1$. But this implies $p_y \subseteq B \subseteq f$ contradicting that p is prime. Thus, p must contain variables either entirely from X or entirely from Y. ∎

OPTIMUM FACTORED FORMS

Not only does the Theorem show the uniqueness of the partition (if we make all cubes prime), it also indicates a procedure for obtaining it.
Given an incompletely specified function (F, D):

- expand all cubes of $F + D$ to primes,
- build a cube matrix M,
- partition M into blocks of disjoint supports,
- remove from M each block of the partition that is entirely contained in D.



MANIPULATION OF BOOLEAN NETWORKS

Basic Techniques:

- Structural operations (change topology)
  - Algebraic
  - Boolean
- Node simplification (change node functions)
  - Don't Cares
  - Node Minimization
- Phase assignment

STRUCTURAL OPERATIONS

Restructuring Problem:
Given initial network, find **best** network.

Example:

$$
\begin{aligned}
f_1 &= abcd + abce + a\overline{b}c\overline{d} + a\overline{b}\overline{c}\overline{d} + a\overline{b}\overline{c}\overline{d} + \overline{a}c + cdf \\
&\quad + ab\overline{c}\overline{d}\overline{e} + a\overline{b}\overline{c}d\overline{f} \\
f_2 &= bdg + \overline{b}dfg + \overline{bd}g + b\overline{d}eg
\end{aligned}
$$

minimize:

$$
\begin{aligned}
f_1 &= bcd + bce + \overline{bd} + \overline{b}f + \overline{a}c + ab\overline{c}\overline{d}\overline{e} + a\overline{b}\overline{c}d\overline{f} \\
f_2 &= bdg + dfg + \overline{bd}g + \overline{d}eg
\end{aligned}
$$

factor:

$$
\begin{aligned}
f_1 &= c(b(d+e) + \overline{b}(\overline{d} + f) + \overline{a}) + a\overline{c}(b\overline{d}\overline{e} + \overline{b}d\overline{f}) \\
f_2 &= g(d(b+f) + \overline{d}(\overline{b} + e))
\end{aligned}
$$

decompose:

$$
\begin{aligned}
f_1 &= c(\overline{a} + x) + a\overline{cx} \\
f_2 &= gx \\
x &= d(b+f) + \overline{d}(\overline{b} + e)
\end{aligned}
$$

Two problems:

Find good common subfunctions

Effect the division

STRUCTURAL OPERATIONS

Basic Operations:

1. decomposition (single function)

$$
\begin{aligned}
f &= abc + abd + \overline{a}\overline{c}\overline{d} + \overline{b}\overline{c}\overline{d} \\
&\Downarrow \\
f &= xy + \overline{x}\overline{y} \\
x &= ab \\
y &= c + d
\end{aligned}
$$

2. extraction (multiple functions)

$$
\begin{aligned}
f &= (az + b\overline{z})cd + e \\
g &= (az + b\overline{z})\overline{e} \\
h &= cde \\
&\Downarrow \\
f &= xy + e \\
g &= x\overline{e} \\
h &= ye \\
x &= az + b\overline{z} \\
y &= cd
\end{aligned}
$$

3. factoring (series-parallel decomposition)

$$\begin{aligned} f &= ac + ad + bc + bd + e \\ &\Downarrow \\ f &= (a+b)(c+d) + e \end{aligned}$$

4. substitution

$$\begin{aligned} g &= a + b \\ f &= a + bc \\ &\Downarrow \\ f &= g(a+c) \end{aligned}$$

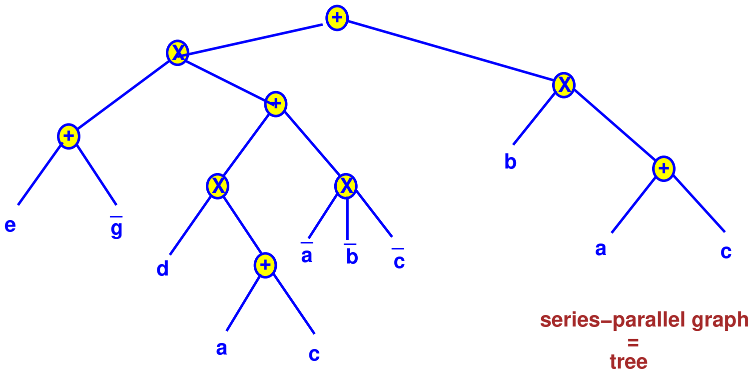
5. collapsing (also called elimination)

$$\begin{aligned} f &= ga + \overline{g}b \\ g &= c + d \\ &\Downarrow \\ f &= ac + ad + b\overline{c}\overline{d} \\ g &= c + d \end{aligned}$$

NOTE: "Division" plays a key role in all these.

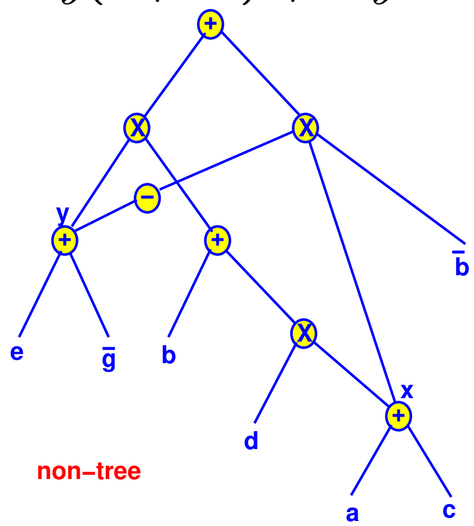
Factoring vs. Decomposition

Factoring $f = (e+\overline{g})(d(a+c)+\overline{a}\overline{b}\overline{c})+b(a+c)$



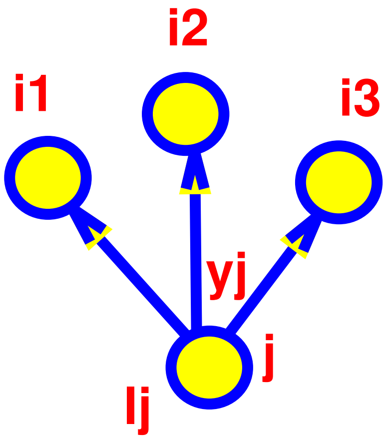
Decomposition:

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



Note: this is similar to BDD collapsing of common nodes and using negative pointers. But not canonical, so don't have perfect identification of common nodes.

Value of a Node and Elimination



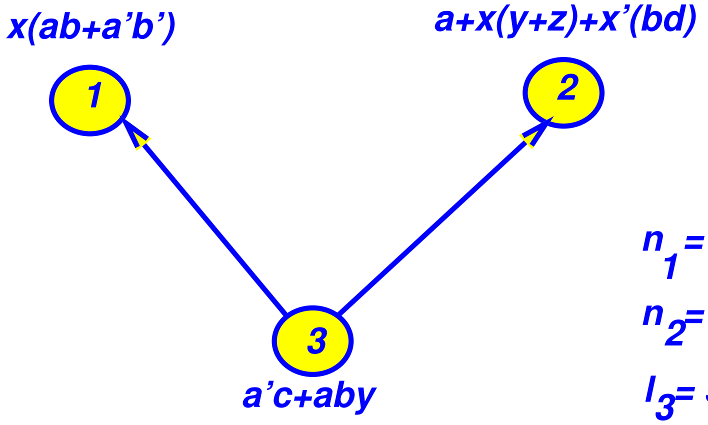
$$value(j) = ((\sum_{i \in FANOUT(j)} n_i) - 1)(l_j - 1) - 1$$

where
$n_i = \#$ times literal y_j and/or $\overline{y}_j$ in factored f_i
$l_j =$ number of literals in factored f_j

with factoring: $l_j + \sum_{i \in FANOUT(j)} n_i + c$
w/o factoring: $l_j \sum_{i \in FANOUT(j)} n_i + c$
value $=$ w/o factoring $-$ with factoring

Can treat y_j and $\overline{y}_j$ the same since $\rho(F_j) = \rho(\overline{F}_j)$.

BEFORE



$n_1 = 1$

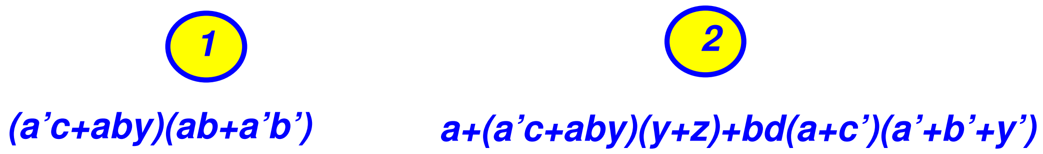
$n_2 = 2$

$l_3 = 5$

value(3) = (1+2–1)(5–1)–1 = 7

x

AFTER

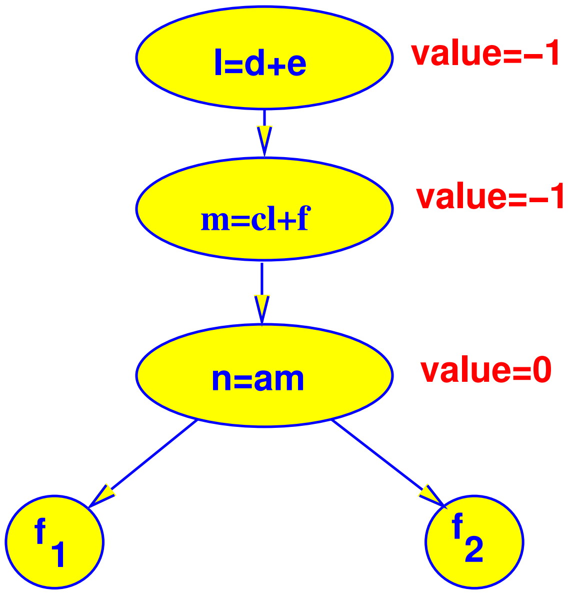


Literals before = 5+7+5 = 17
Literals after = 9+15 = 24
––
7

Difference after – before = value = 7

But we may not have the same value if we were to eliminate, simplify, and then refactor.

Value of a Node and Elimination



eliminate -1

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