COE 561
Digital System Design & Synthesis
Multiple-Level Logic Synthesis

Dr. Aiman H. El-Maleh

Computer Engineering Department

King Fahd University of Petroleum & Minerals

[Adapted from slides of Prof. G. De Micheli: Synthesis & Optimization of Digital Circuits]

Outline …

- **Representations.**
- **Taxonomy of optimization methods.**
  - Goals: area/delay.
  - Algorithms: Algebraic/Boolean.
  - Rule-based methods.
- **Examples of transformations.**
- **Algebraic model.**
  - Algebraic division.
  - Algebraic substitution.
  - Single-cube extraction.
  - Multiple-cube extraction.
  - Decomposition.
  - Factorization.
  - Fast extraction.

… Outline

- **External and internal *don't care* sets.**
  - Controllability *don't care* sets.
  - Observability *don't care* sets.
- **Boolean simplification and substitution.**
- **Testability properties of multiple-level logic.**
- **Synthesis for testability.**
- **Network delay modeling.**
- **Algorithms for delay minimization.**
- **Transformations for delay reduction.**

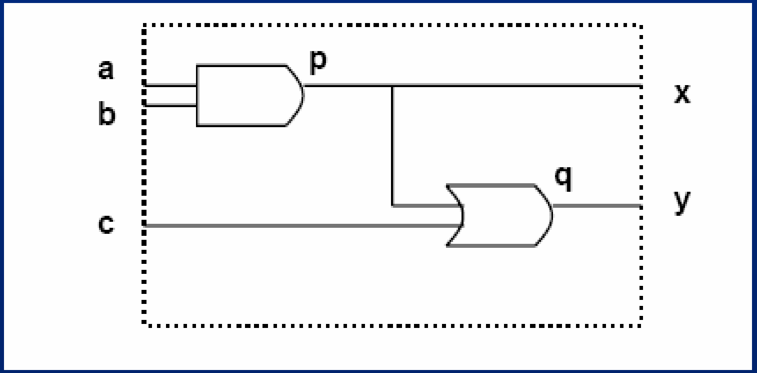
Motivation

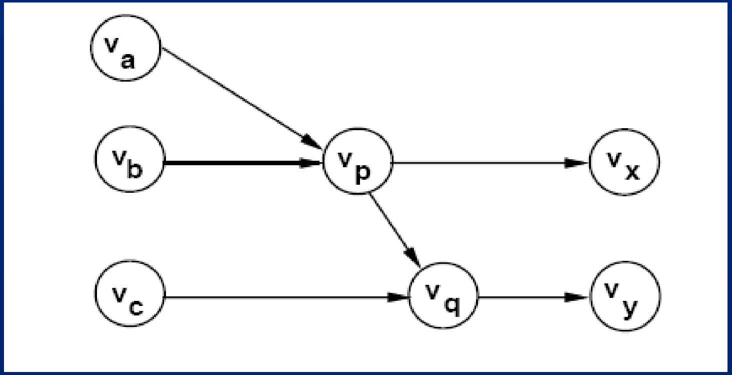
- **Combinational logic circuits very often implemented as multiple-level networks of logic gates.**
- **Provides several degrees of freedom in logic design**
  - Exploited in optimizing area and delay.
  - Different timing requirements on input/output paths.
- **Multiple-level networks viewed as interconnection of single-output gates**
  - Single type of gate (e.g. NANDs or NORs).
  - Instances of a cell library.
  - Macro cells.
- **Multilevel optimization is divided into two tasks**
  - Optimization neglecting implementation constraints assuming loose models of area and delay.
  - Constraints on the usable gates are taken into account during optimization.

Circuit Modeling

- **Logic network**
  - Interconnection of logic functions.
  - Hybrid structural/behavioral model.
- **Bound (mapped) networks**
  - Interconnection of logic gates.
  - Structural model.

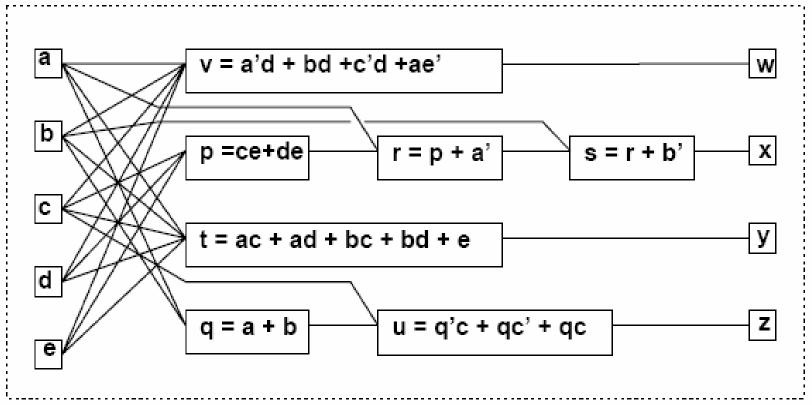
Example of Bound Network



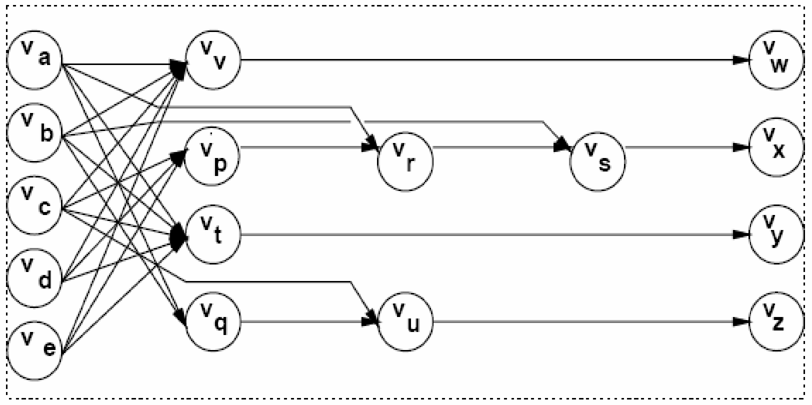


Example of a Logic Network

$$
\begin{aligned}
p &= ce + de \\
q &= a + b \\
r &= p + a' \\
s &= r + b' \\
t &= ac + ad + bc + bd + e \\
u &= q'c + qc' + qc \\
v &= a'd + bd + c'd + ae' \\
w &= v \\
x &= s \\
y &= t \\
z &= u
\end{aligned}
$$



(a)



(b)

Network Optimization

- **Two-level logic**
  - Area and delay proportional to cover size.
  - Achieving minimum (or irredundant) covers corresponds to optimizing area and speed.
  - Achieving irredundant cover corresponds to maximizing testability.
- **Multiple-level logic**
  - Minimal-area implementations do not correspond in general to minimum-delay implementations and vice versa.
  - Minimize area (power) estimate
    - subject to delay constraints.
  - Minimize maximum delay
    - subject to area (power) constraints.
  - Minimize power consumption.
    - subject to delay constraints.
  - Maximize testability.

Estimation

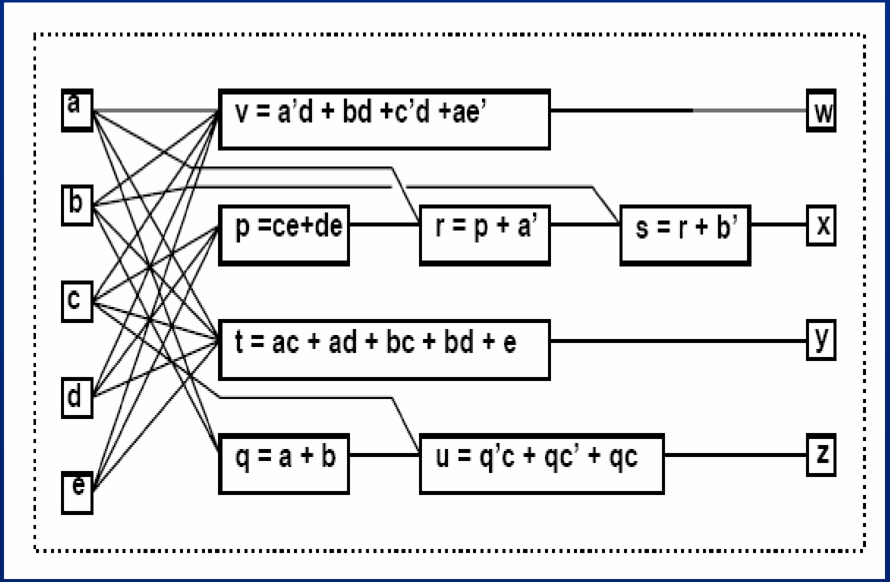
- **Area**
  - Number of literals
    - Corresponds to number of polysilicon strips (transistors)
  - Number of functions/gates.
- **Delay**
  - Number of stages (unit delay per stage).
  - Refined gate delay models (relating delay to function complexity and fanout).
  - Sensitizable paths (detection of false paths).
  - Wiring delays estimated using statistical models.

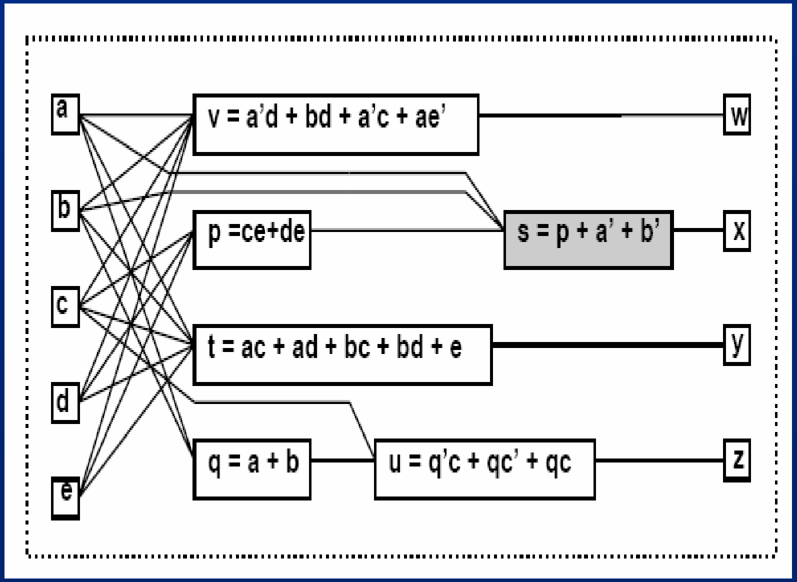
Problem Analysis

- **Multiple-level optimization is hard.**
- **Exact methods**
  - Exponential complexity.
  - Impractical.
- **Approximate methods**
  - Heuristic algorithms.
  - Rule-based methods.
- **Strategies for optimization**
  - Improve circuit step by step based on circuit transformations.
  - Preserve network behavior.
  - Methods differ in
    - Types of transformations.
    - Selection and order of transformations.

Elimination

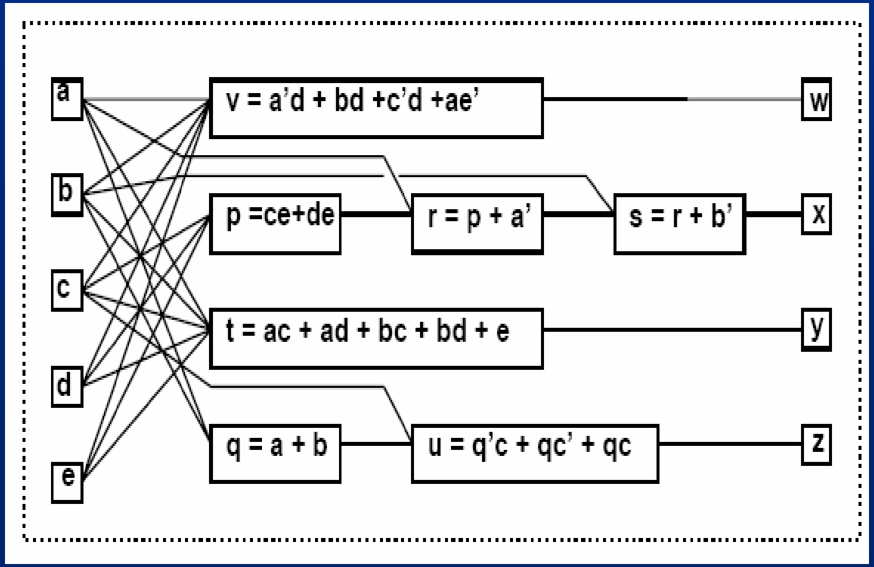
- Eliminate one function from the network.
- Perform variable substitution.
- Example
  - s = r +b’; r = p+a’
  - ⇒ s = p+a’+b’.

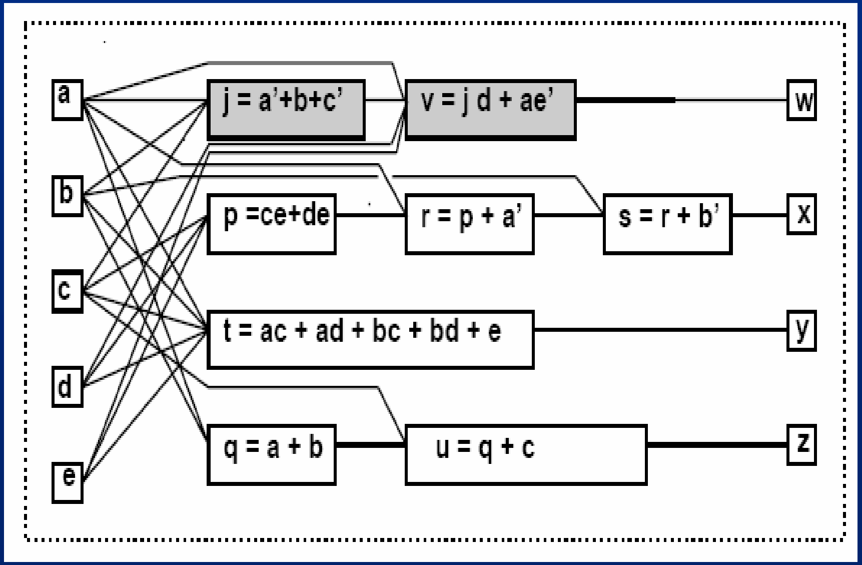




Decomposition

- Break one function into smaller ones.
- Introduce new vertices in the network.
- Example
  - v = a’d+bd+c’d+ae’.
  - ⇒ j = a’+b+c’; v = jd+ae’





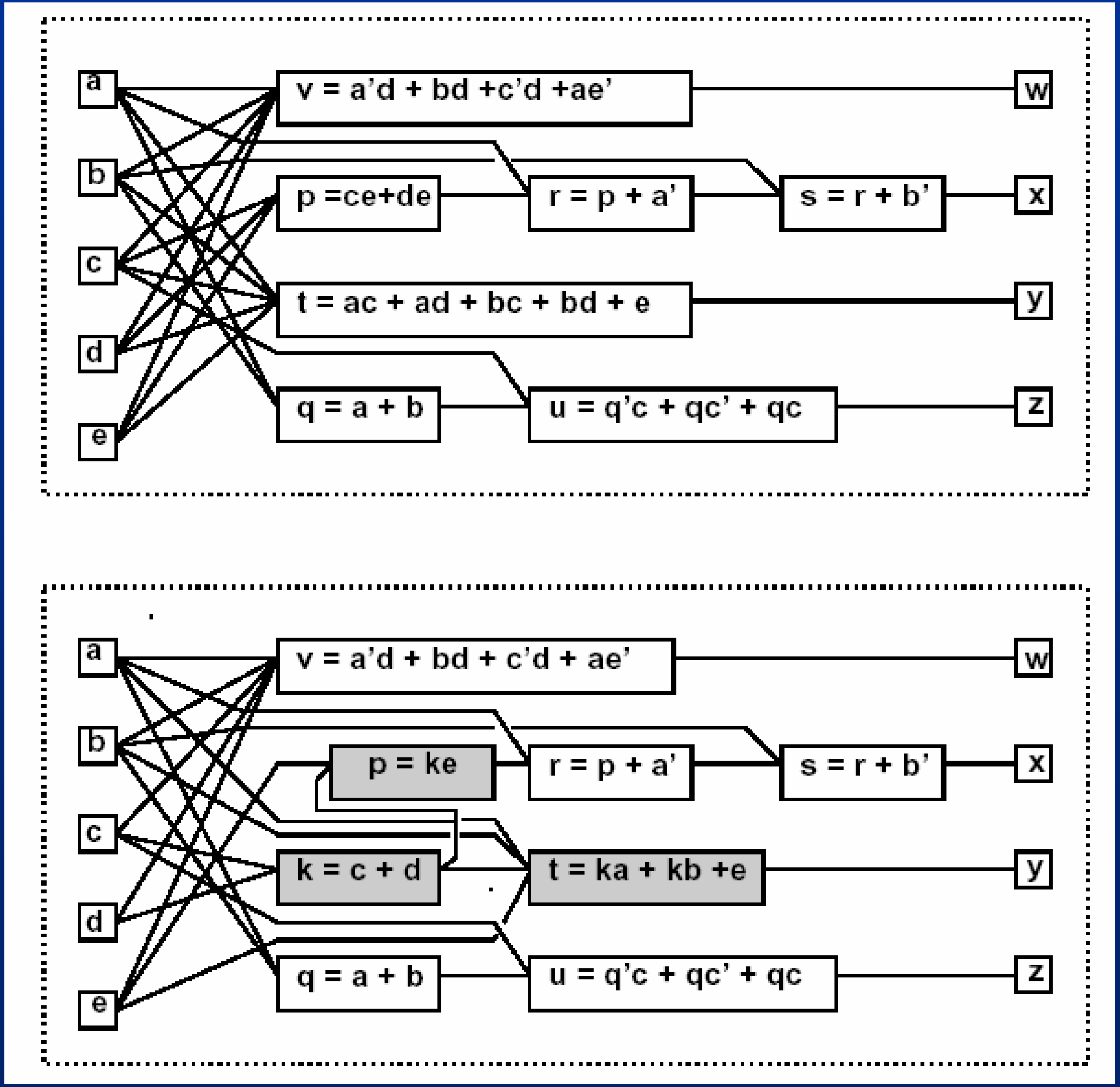
Factoring

- **Factoring is the process of deriving a factored form from a sum-of-products form of a function.**
- **Factoring is like decomposition except that no additional nodes are created.**
- **Example**
  - F = abc+abd+a’b’c+a’b’d+ab’e+ab’f+a’be+a’bf (24 literals)
  - After factorization
    - F=(ab+a’b’)(c+d) + (ab’+a’b)(e+f) (12 literals)

Extraction …

- Find a common sub-expression of two (or more) expressions.
- Extract sub-expression as new function.
- Introduce new vertex in the network.
- Example
  - p = ce+de; t = ac+ad+bc+bd+e; (13 literals)
  - p = (c+d)e; t = (c+d)(a+b)+e; (Factoring:8 literals)
  - ⇒ k = c+d; p = ke; t = ka+ kb +e; (Extraction:9 literals)

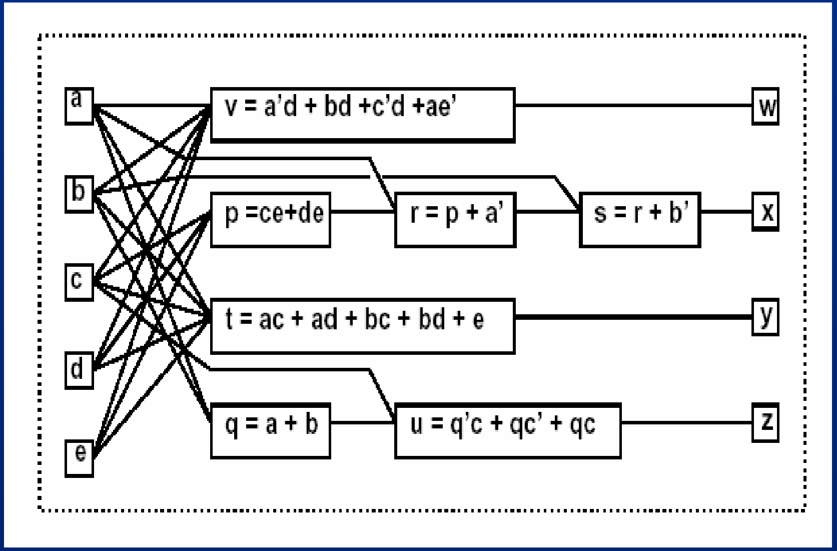
… Extraction

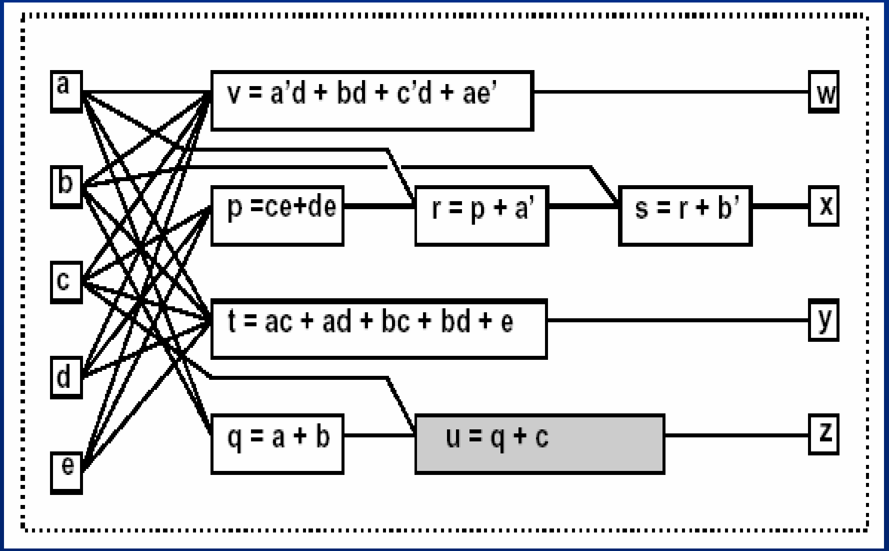


a
b
c
d
e
v = a'd + bd +c'd +ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q'c + qc' + qc
w
x
y
z
a
b
c
d
e
v = a'd + bd + c'd + ae'
p = ke
r = p + a'
s = r + b'
k = c + d
t = ka + kb +e
q = a + b
u = q'c + qc' + qc
w
x
y
z

Simplification

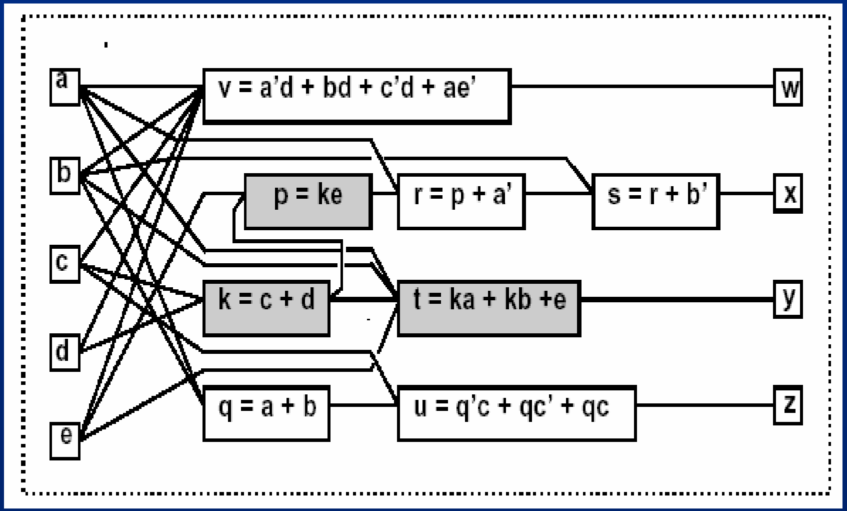
- Simplify a local function (using Espresso).
- Example
  - u = q'c+qc' +qc;
  - $\Rightarrow$ u = q +c;

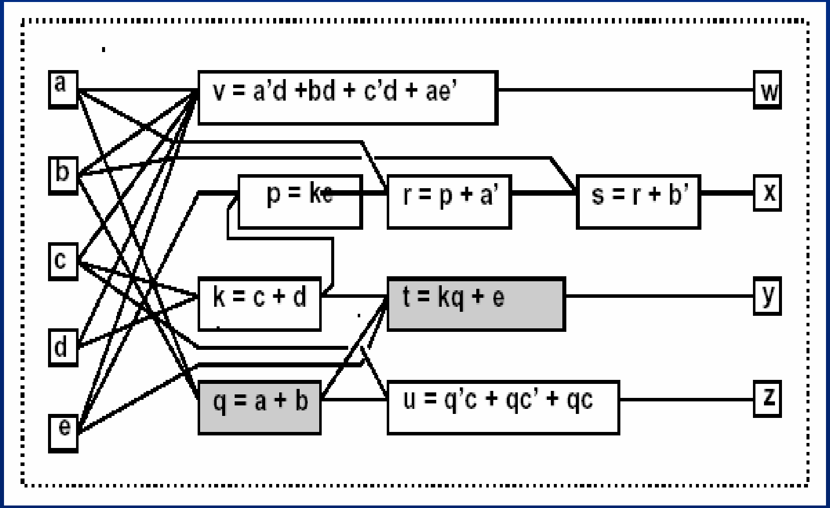




Substitution

- Simplify a local function by using an additional input that was not previously in its support set.
- Example
  - t = ka+kb+e.
  - $\Rightarrow$ t = kq +e; because q = a+b.

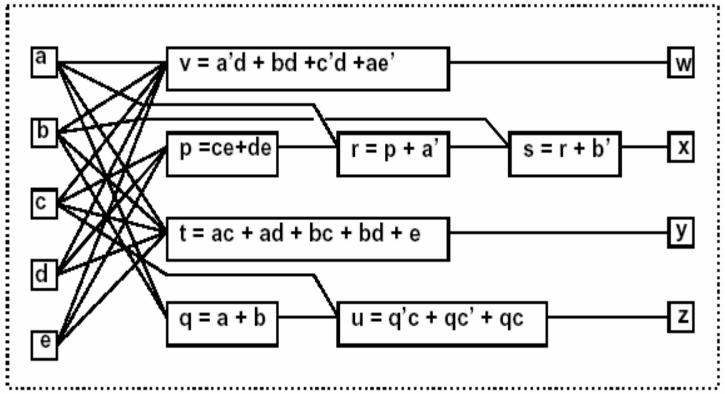




Example: Sequence of Transformations

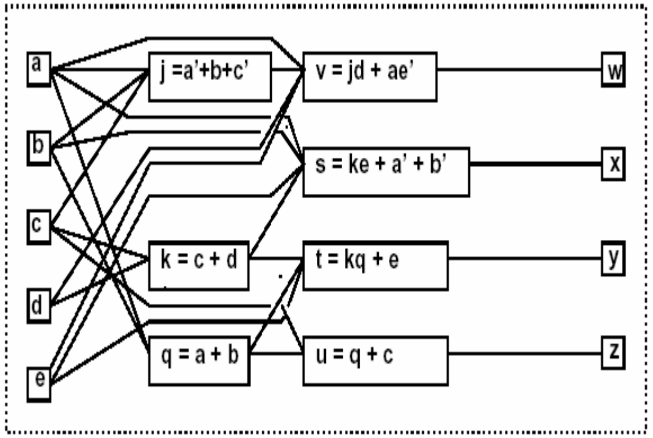
Original Network (33 lit.)

$$
\begin{aligned}
p &= ce + de \\
q &= a + b \\
r &= p + a' \\
s &= r + b' \\
t &= ac + ad + bc + bd + e \\
u &= q'c + qc' + qc \\
v &= a'd + bd + c'd + ae'
\end{aligned}
$$



Transformed Network (20 lit.)

$$
\begin{aligned}
j &= a' + b + c' \\
k &= c + d \\
q &= a + b \\
s &= ke + a' + b' \\
t &= kq + e \\
u &= q + c \\
v &= jd + ae'
\end{aligned}
$$

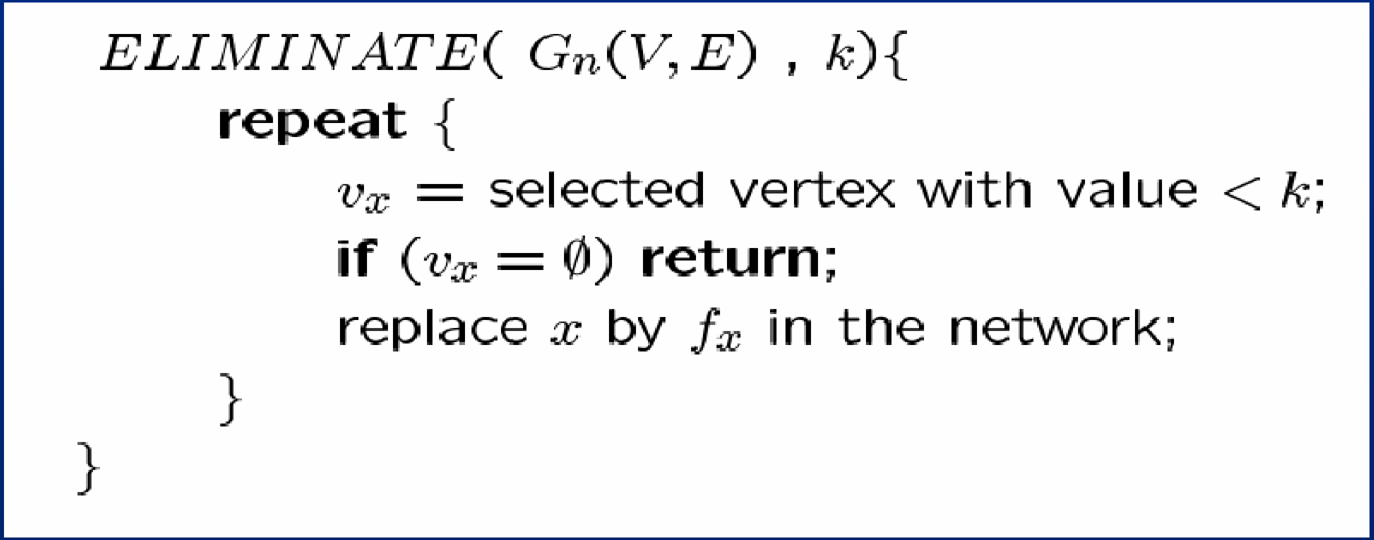


Optimization Approaches

- **Algorithmic approach**
  - Define an algorithm for each transformation type.
  - Algorithm is an operator on the network.
  - Each operator has well-defined properties
    - Heuristic methods still used.
    - Weak optimality properties.
  - Sequence of operators
    - Defined by scripts.
    - Based on experience.
- **Rule-based approach (IBM Logic Synthesis System)**
  - Rule-data base
    - Set of pattern pairs.
  - Pattern replacement driven by rules.

Elimination Algorithm …

- Set a threshold k (usually 0).
- Examine all expressions (vertices) and compute their values.
- Vertex value = n*l – n – l (l is number of literals; n is number of times vertex variable appears in network)
- Eliminate an expression (vertex) if its value (i.e. the increase in literals) does not exceed the threshold.



```
$ELIMINATE(\ G_n(V, E)\ ,\ k)\{$
    repeat {
        $v_x$ = selected vertex with value $< k$;
        if ($v_x = \emptyset$) return;
        replace $x$ by $f_x$ in the network;
    }
}
```

… Elimination Algorithm

- **Example**
  - q = a + b
  - s = ce + de + a’ + b’
  - t = ac + ad + bc + bd + e
  - u = q’c + qc’ + qc
  - v = a’d + bd + c’d + ae’
- **Value of vertex q=**n*l–n–l=3*2-3-2=1
  - It will increase number of literals => not eliminated
- **Assume u is simplified to u=c+q**
  - Value of vertex q=**n*l–n–l=1*2-1-2=-1**
  - It will decrease the number of literals by 1 => eliminated

MIS/SIS Rugged Script

- sweep; eliminate -1
- simplify -m nocomp
- eliminate -1
- sweep; eliminate 5
- simplify -m nocomp
- resub -a
- fx
- resub -a; sweep
- eliminate -1; sweep
- full-simplify -m nocomp

***Sweep* eliminates single-input Vertices and those with a constant function.**

***resub –a* performs algebraic substitution of all vertex pairs**

***fx* extracts double-cube and single-cube expression.**

Boolean and Algebraic Methods …

- **Boolean methods**
  - Exploit Boolean properties of logic functions.
  - Use don't care conditions induced by interconnections.
  - Complex at times.
- **Algebraic methods**
  - View functions as polynomials.
  - Exploit properties of polynomial algebra.
  - Simpler, faster but weaker.

… Boolean and Algebraic Methods

- **Boolean substitution**
  - h = a+bcd+e; q = a+cd
  - $\Rightarrow$ h = a+bq +e
  - Because a+bq+e = a+b(a+cd)+e = a+bcd+e;
    - Relies on Boolean property b+1=1
- **Algebraic substitution**
  - t = ka+kb+e; q=a+b
  - $\Rightarrow$ t = kq +e
  - Because k(a+b) = ka+kb; holds regardless of any assumption of Boolean algebra.

The Algebraic Model …

- **Represents local Boolean functions by algebraic expressions**
  - Multilinear polynomial (i.e. multi-variable with degree 1) over set of variables with unit coefficients.
- **Algebraic transformations neglect specific features of Boolean algebra**
  - Only one distributive law applies
    - $a \cdot (b+c) = ab+ac$
    - $a + (b \cdot c) \neq (a+b).(a+c)$
  - Complements are not defined
    - Cannot apply some properties like absorption, idempotence, involution and Demorgan’s, $a+a'=1$ and $a.a'=0$
  - Symmetric distribution laws.
  - Don't care sets are not used.

… The Algebraic Model

- **Algebraic expressions obtained by**
  - Modeling functions in sum of products form.
  - Make them minimal with respect to single-cube containment.
- **Algebraic operations restricted to expressions with disjoint support**
  - Preserve correspondence of result with sum-of-product forms minimal w.r.t single-cube containment.
- **Example**
  - (a+b)(c+d)=ac+ad+bc+bd; minimal w.r.t SCC.
  - (a+b)(a+c)= aa+ac+ab+bc; non-minimal.
  - (a+b)(a’+c)=aa’+ac+a’b+bc; non-minimal.

Algebraic Division …

- **Given two algebraic expressions $f_{dividend}$ and $f_{divisor}$, we say that $f_{divisor}$ is an Algebraic Divisor of $f_{dividend}$,** $f_{quotient} = f_{dividend}/f_{divisor}$ when
  - $f_{dividend} = f_{divisor} \cdot f_{quotient} + f_{remainder}$
  - $f_{divisor} \cdot f_{quotient} \neq 0$
  - and the support of $f_{divisor}$ and $f_{quotient}$ is disjoint.
- **Example**
  - Let $f_{dividend} = ac+ad+bc+bd+e$ and $f_{divisor} = a+b$
    - Then $f_{quotient} = c+d$ $f_{remainder} = e$
    - Because $(a+b)(c+d)+e = f_{dividend}$
    - and $\{a,b\} \cap \{c,d\} = \varnothing$
  - Non-algebraic division
    - Let $f_i = a+bc$ and $f_j = a+b$.
    - Let $f_k = a+c$. Then, $f_i = f_j \cdot f_k = (a+b)(a+c) = f_i$
    - but $\{a,b\} \cap \{a,c\} \neq \varnothing$

… Algebraic Division

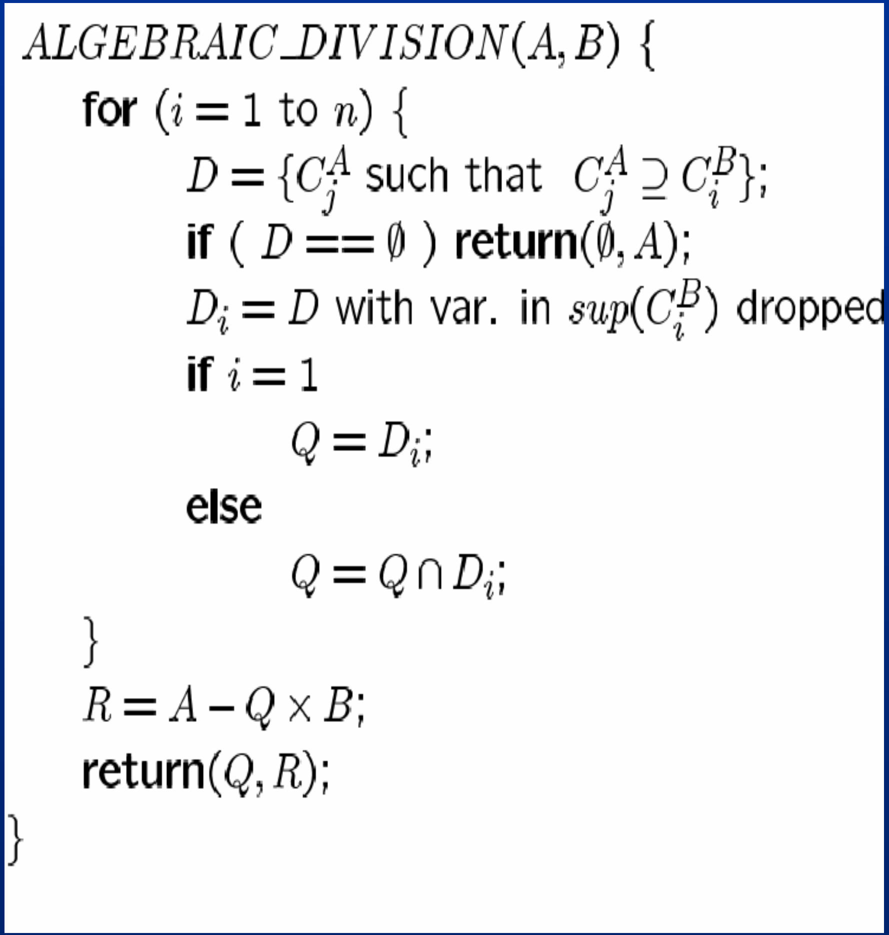
- An algebraic divisor is called a factor when the remainder is void.
  - a+b is a factor of ac+ad+bc+bd
- An expression is said to be cube free when it cannot be factored by a cube.
  - a+b is cube free
  - ac+ad+bc+bd is cube free
  - ac+ad is non-cube free
  - abc is non-cube free

Algebraic Division Algorithm …

$A = \{C_j^A,\ j = 1,2,\ldots l\}$ *set of cubes (monomials) of the dividend*

$B = \{C_j^B,\ j = 1,2,\ldots n\}$ *set of cubes (monomials) of the divisor*

- **Quotient Q and remainder R are sum of cubes (monomials).**
- **Intersection is largest subset of common monomials.**



```
$ALGEBRAIC\_DIVISION(A, B)$ {
    for ($i = 1$ to $n$) {
        $D = \{C_j^A$ such that $C_j^A \supseteq C_i^B\}$;
        if ( $D == \emptyset$ ) return($\emptyset, A$);
        $D_i = D$ with var. in $sup(C_i^B)$ dropped
        if $i = 1$
            $Q = D_i$;
        else
            $Q = Q \cap D_i$;
    }
    $R = A - Q \times B$;
    return($Q, R$);
}
```

… Algebraic Division Algorithm …

- **Example**
  - **$f_{dividend}$ = ac+ad+bc+bd+e;**
  - **$f_{divisor}$ = a+b;**
  - **A = {ac, ad, bc, bd, e} and B = {a, b}.**
  - **i = 1**
    - C^B_1 = a, D = {ac, ad} and D_1 = {c, d}.
    - Q = {c, d}.
  - **i = 2 = n**
    - C^B_2 = b, D = {bc, bd} and D_2 = {c, d}.
    - Then Q = {c, d} $\cap$ {c, d} = {c, d}.
  - **Result**
    - Q = {c, d} and R = {e}.
    - $f_{quotient}$ = c+d and $f_{remainder}$ = e.

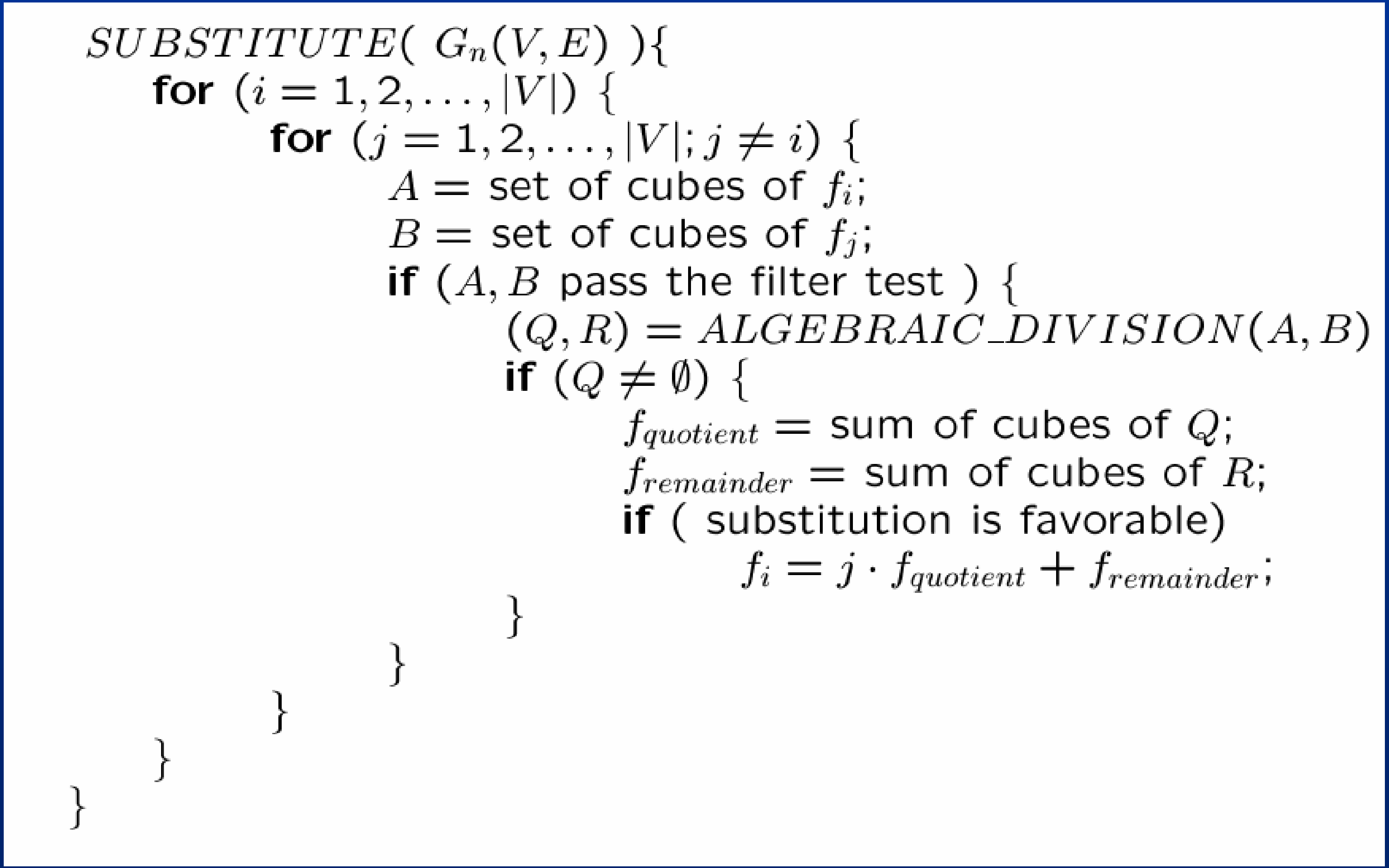
… Algebraic Division Algorithm

- **Example**
  - Let $f_{dividend}$ = axc+axd+bc+bxd+e; $f_{divisor}$ = ax+b
  - i=1, $C^B{}_1$ = ax, D = {axc, axd} and D_1 = {c, d}; Q={c, d}
  - i = 2 = n; $C^B{}_2$ = b, D = {bc, bxd} and D_2 = {c, xd}.
  - Then Q = {c, d} $\cap$ {c, xd} = {c}.
  - $f_{quotient}$ = c and $f_{remainder}$ = axd+bxd+e.
- Theorem: Given algebraic expressions f_i and f_j, then f_i/f_j is empty when
  - f_j contains a variable not in f_i.
  - f_j contains a cube whose support is not contained in that of any cube of f_i.
  - f_j contains more cubes than f_i.
  - The count of any variable in f_j larger than in f_i.

Substitution

- **Substitution replaces a subexpression by a variable associated with a vertex of the logic network.**
- **Consider expression pairs.**
- **Apply division (in any order).**
- **If quotient is not void**
  - Evaluate area/delay gain
  - Substitute $f_{dividend}$ by $j.f_{quotient} + f_{remainder}$ where $j = f_{divisor}$
- **Use filters to reduce divisions.**
- **Theorem**
  - Given two algebraic expressions f_i and f_j, $f_i/f_j=\emptyset$ if there is a path from v_i to v_j in the logic network.

Substitution algorithm



```
$SUBSTITUTE(\ G_n(V, E)\ )\{$
    for $(i = 1, 2, \ldots, |V|)$ {
        for $(j = 1, 2, \ldots, |V|; j \neq i)$ {
            $A =$ set of cubes of $f_i$;
            $B =$ set of cubes of $f_j$;
            if $(A, B$ pass the filter test $)$ {
                $(Q, R) = ALGEBRAIC\_DIVISION(A, B)$
                if $(Q \neq \emptyset)$ {
                    $f_{quotient} =$ sum of cubes of $Q$;
                    $f_{remainder} =$ sum of cubes of $R$;
                    if ( substitution is favorable)
                        $f_i = j \cdot f_{quotient} + f_{remainder}$;
                }
            }
        }
    }
}
```

Extraction

- **Search for common sub-expressions**
  - Single-cube extraction: monomial.
  - Multiple-cube (kernel) extraction: polynomial
- **Search for appropriate divisors.**
- **Cube-free expression**
  - Cannot be factored by a cube.
- **Kernel of an expression**
  - Cube-free quotient of the expression divided by a cube (called co-kernel).
- **Kernel set K(f) of an expression**
  - Set of kernels.

Kernel Example

- $f_x = ace+bce+de+g$
- Divide f_x by a. Get ce. Not cube free.
- Divide f_x by b. Get ce. Not cube free.
- Divide f_x by c. Get ae+be. Not cube free.
- Divide f_x by ce. Get a+b. Cube free. Kernel!
- Divide f_x by d. Get e. Not cube free.
- Divide f_x by e. Get ac+bc+d. Cube free. Kernel!
- Divide f_x by g. Get 1. Not cube free.
- Expression f_x is a kernel of itself because cube free.
- $K(f_x) = \{(a+b); (ac+bc+d); (ace+bce+de+g)\}$.

Theorem (Brayton and McMullen)

- **Two expressions f_a and f_b have a common multiple-cube divisor f_d if and only if**
  - there exist kernels $k_a \in K(f_a)$ and $k_b \in K(f_b)$ s.t. f_d is the sum of 2 (or more) cubes in $k_a \cap k_b$ (intersection is largest subset of common monomials)
- **Consequence**
  - If kernel intersection is void, then the search for common sub-expression can be dropped.
- **Example**

$f_x = ace+bce+de+g$;	$K(f_x) = \{(a+b); (ac+bc+d); (ace+bce+de+g)\}$
$f_y = ad+bd+cde+ge$;	$K(f_y) = \{(a+b+ce); (cd+g); (ad+bd+cde+ge)\}$
$f_z = abc$;	The kernel set of f_z is empty.

Select intersection (a+b)

$f_w = a+b$	$f_x = wce+de+g$
$f_y = wd+cde+ge$	$f_z = abc$

Kernel Set Computation …

- **Naive method**
  - Divide function by elements in power set of its support set.
  - Weed out non cube-free quotients.
- **Smart way**
  - Use recursion
    - Kernels of kernels are kernels of original expression.
  - Exploit commutativity of multiplication.
    - Kernels with co-kernels *ab* and *ba* are the same
- **A kernel has level 0 if it has no kernel except itself.**
- **A kernel is of level n if it has**
  - at least one kernel of level n-1
  - no kernels of level n or greater except itself

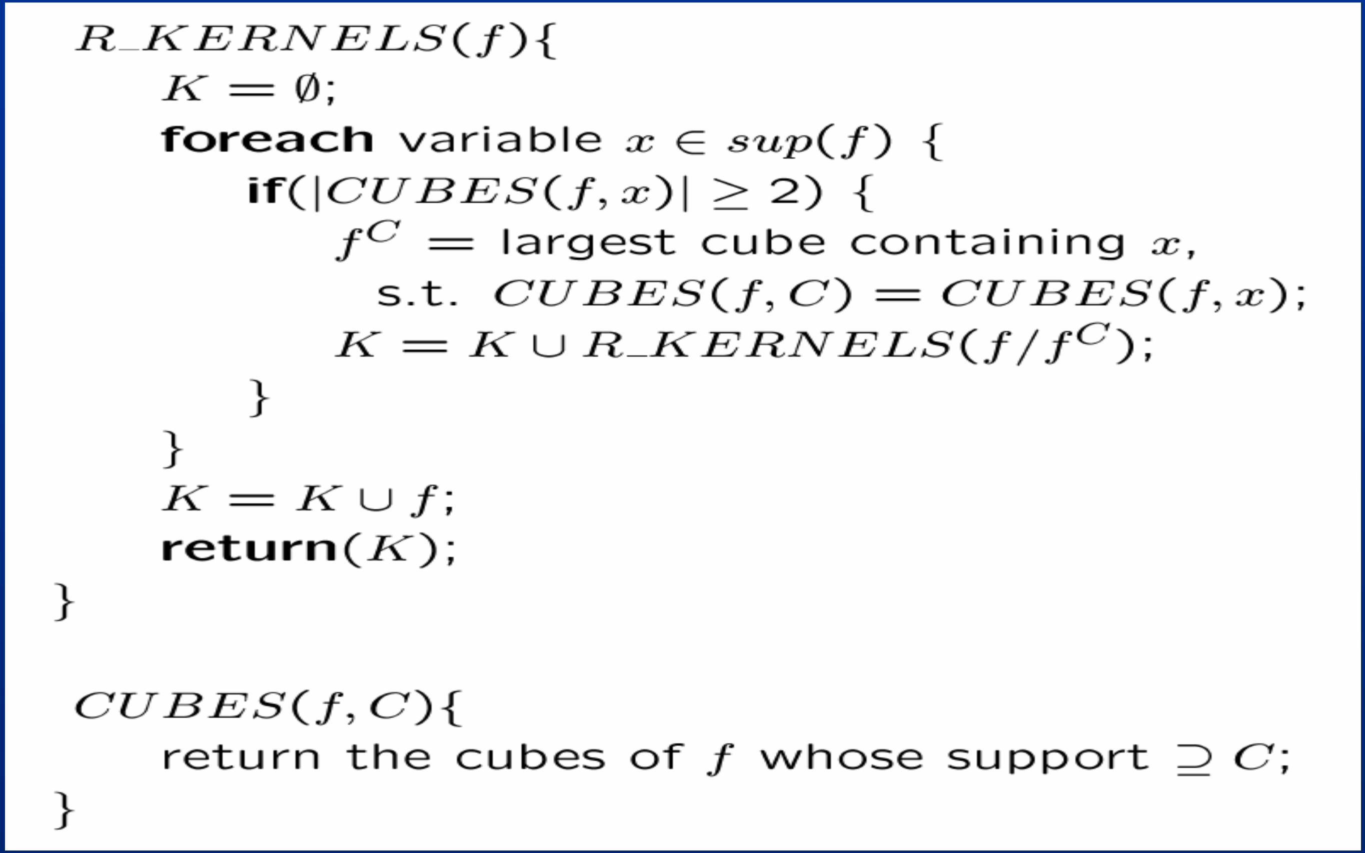
…Kernel Set Computation

- Y= adf + aef + bdf + bef + cdf + cef + g

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

Kernels	Co-Kernels	Level
(a+b+c)	df, ef	0
(d+e)	af, bf, cf	0
(a+b+c)(d+e)	f	1
(a+b+c)(d+e)f+g	1	2

Recursive Kernel Computation: Simple Algorithm



```
$R\_KERNELS(f)\{$
    $K = \emptyset;$
    foreach variable $x \in sup(f)$ {
        if($|CUBES(f,x)| \geq 2$) {
            $f^C$ = largest cube containing $x$,
              s.t. $CUBES(f,C) = CUBES(f,x)$;
            $K = K \cup R\_KERNELS(f/f^C)$;
        }
    }
    $K = K \cup f$;
    return($K$);
}

$CUBES(f,C)\{$
    return the cubes of $f$ whose support $\supseteq C$;
}
```

- *f* is assumed to be cube-free
    - If not divide it by its largest cube factor

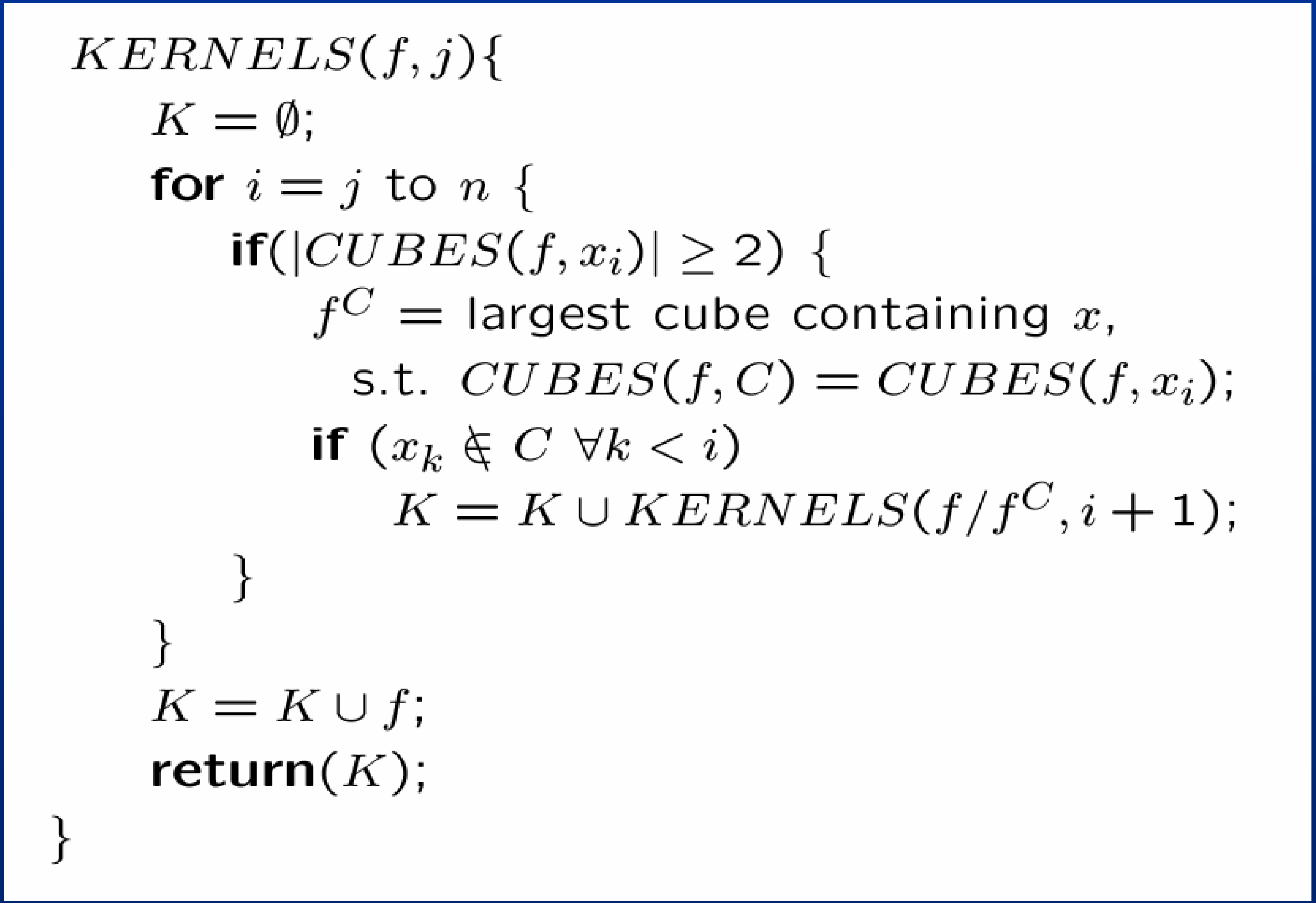
Recursive Kernel Computation Example

- f = ace+bce+de+g
- **Literals a or b. No action required.**
- **Literal c. Select cube ce:**
  - Recursive call with argument (ace+bce+de+g)/ce =a+b;
  - No additional kernels.
  - Adds a+b to the kernel set at the last step.
- **Literal d. No action required.**
- **Literal e. Select cube e:**
  - Recursive call with argument ac+bc+d
  - Kernel a+b is rediscovered and added.
  - Adds ac + bc + d to the kernel set at the last step.
- **Literal g. No action required.**
- **Adds ace+bce+de+g to the kernel set.**
- **K = {(ace+bce+de+g); (a+b); (ac+bc+d); (a+b)}.**

Analysis

- **Some computation may be redundant**
  - Example
    - Divide by a and then by b.
    - Divide by b and then by a.
  - Obtain duplicate kernels.
- **Improvement**
  - Keep a pointer to literals used so far denoted by *j*.
  - *J* initially set to 1.
  - Avoids generation of co-kernels already calculated
  - *Sup(f)=*$\{x_1, x_2, \ldots x_n\}$ *(arranged in lexicographic order)*
  - *f* is assumed to be cube-free
    - If not divide it by its largest cube factor
  - Faster algorithm

Recursive Kernel Computation



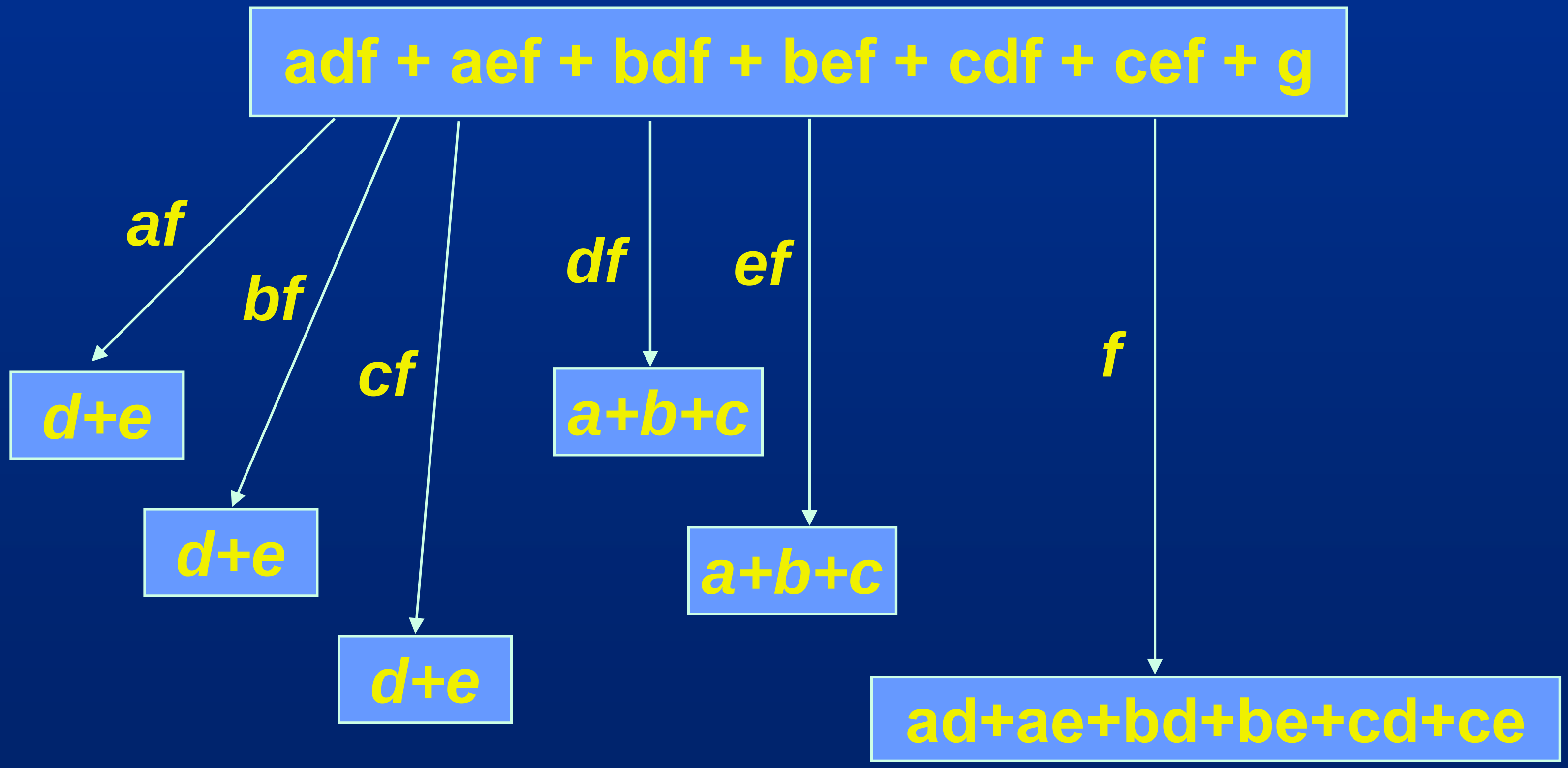
```
$KERNELS(f, j)\{$
    $K = \emptyset$;
    for $i = j$ to $n$ $\{$
        if$(|CUBES(f, x_i)| \geq 2)$ $\{$
            $f^C$ $=$ largest cube containing $x$,
              s.t. $CUBES(f, C) = CUBES(f, x_i)$;
            if $(x_k \notin C \ \forall k < i)$
                $K = K \cup KERNELS(f/f^C, i + 1)$;
        $\}$
    $\}$
    $K = K \cup f$;
    return$(K)$;
$\}$
```

Recursive Kernel Computation Examples…

- **f = ace+bce+de+g; sup(f)={a, b, c, d, e, g}**
- **Literals a or b. No action required.**
- **Literal c. Select cube ce:**
  - Recursive call with arguments: (ace+bce+de+g)/ce =a+b; pointer j = 3+1=4.
  - Call considers variables {d, e, g}. No kernel.
  - Adds a+b to the kernel set at the last step.
- **Literal d. No action required.**
- **Literal e. Select cube e:**
  - Recursive call with arguments: ac+bc+d and pointer j = 5+1=6.
  - Call considers variable {g}. No kernel.
  - Adds ac+bc+d to the kernel set at the last step.
- **Literal g. No action required.**
- **Adds ace+bce+de+g to the kernel set.**
- **K = {(ace+bce+de+g); (ac+bc+d); (a+b)}.**

…Recursive Kernel Computation Examples

- **Y= adf + aef + bdf + bef + cdf + cef + g=(d+e)(a+b+c)f+g**
  - Lexicographic order {a, b, c, d, e, f, g}

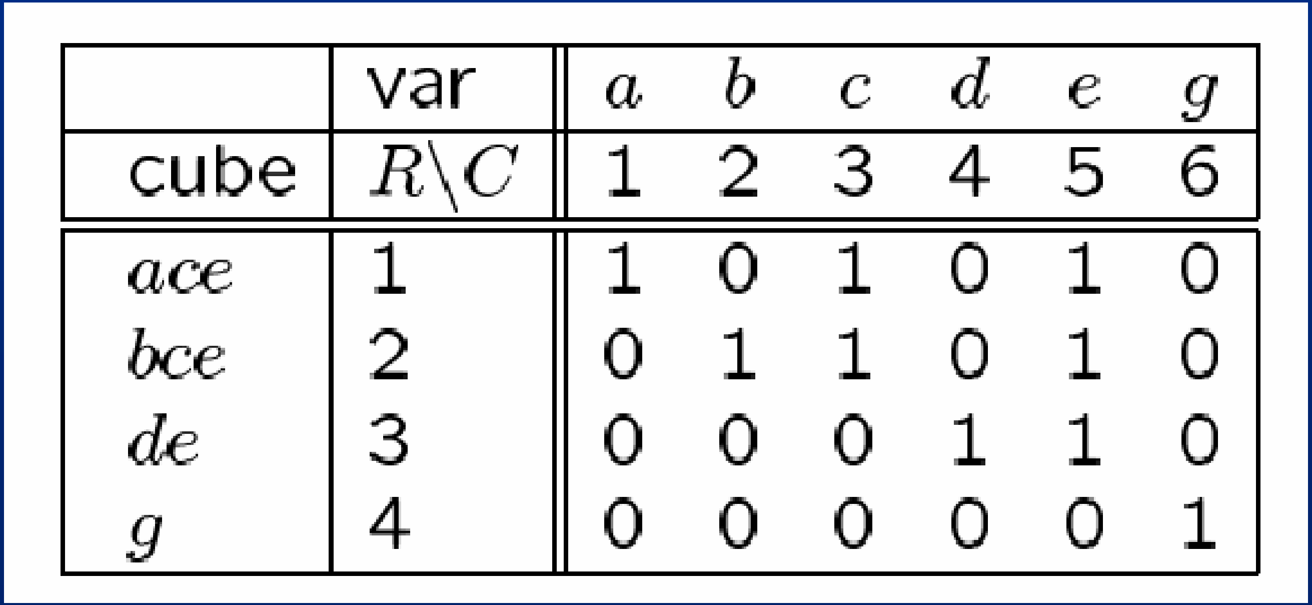


Matrix Representation of Kernels …

- **Boolean matrix**
  - Rows: cubes. Columns: variables (in both true and complement form as needed).
- **Rectangle (R, C)**
  - Subset of rows and columns with all entries equal to 1.
- **Prime rectangle**
  - Rectangle not inside any other rectangle.
- **Co-rectangle (R, C’) of a rectangle (R, C)**
  - C’ are the columns not in C.
- **A co-kernel corresponds to a prime rectangle with at least two rows.**

… Matrix Representation of Kernels …

- f_x = ace+bce+de+g
- Rectangle (prime): ({1, 2}, {3, 5})
  - Co-kernel ce.
- Co-rectangle: ({1, 2}, {1, 2, 4, 6}).
  - Kernel a+b.



	var	a	b	c	d	e	g
cube	$R\backslash C$	1	2	3	4	5	6
ace	1	1	0	1	0	1	0
bce	2	0	1	1	0	1	0
de	3	0	0	0	1	1	0
g	4	0	0	0	0	0	1

… Matrix Representation of Kernels …

- Theorem: K is a kernel of *f* iff it is an expression corresponding to the co-rectangle of a prime rectangle of f.
- The set of all kernels of a logic expression are in 1-1 correspondence with the set of all co-rectangles of prime rectangles of the corresponding Boolean matrix.
- A level-0 kernel is the co-rectangle of a prime rectangle of maximal width.
- A prime rectangle of maximum height corresponds to a kernel of maximal level.

… Matrix Representation of Kernels

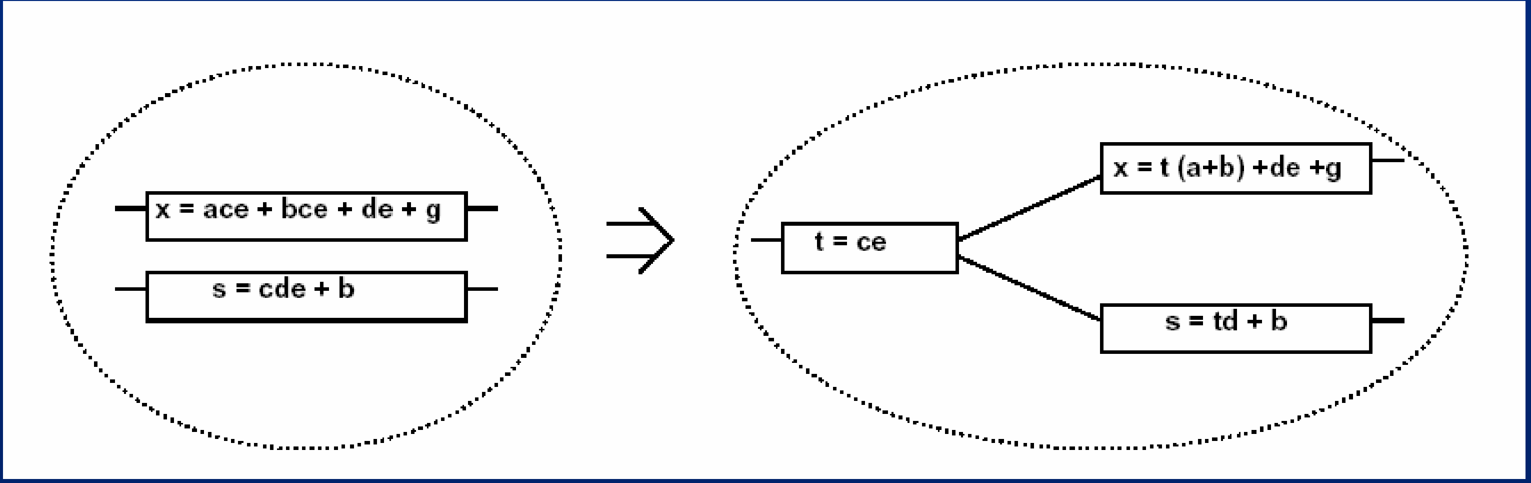
- **Example**
  - F = abc + abd + ae

		1	2	3	4	5
	Cube	a	b	c	d	e
1	abc	1	1	1		
2	abd	1	1		1	
3	ae	1				1

  - Prime Rectangles & Co-Rectangles
    - PR:{(1,2),(1,2)}: corresponding to co-kernel ab
    - CR:{(1,2),(3,4,5)}: corresponding to kernel (c+d)
    - PR:{(1,2,3),(1)}: corresponding to co-kernel a
    - CR:{(1,2,3),(2,3,4,5)}: corresponding to kernel (bc+bd+e)

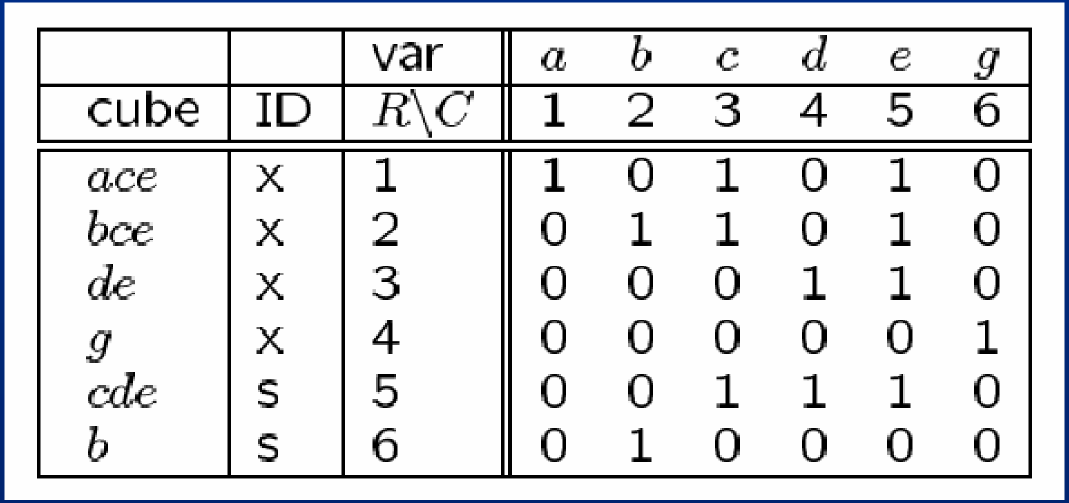
Single-Cube Extraction …

- **Form auxiliary function**
  - Sum of all product terms of all functions.
- **Form matrix representation**
  - A rectangle with at least two rows represents a common cube.
  - Rectangles with at least two columns may result in savings.
  - Best choice is a prime rectangle.
- **Use function ID for cubes**
  - Cube intersection from different functions.



… Single-Cube Extraction

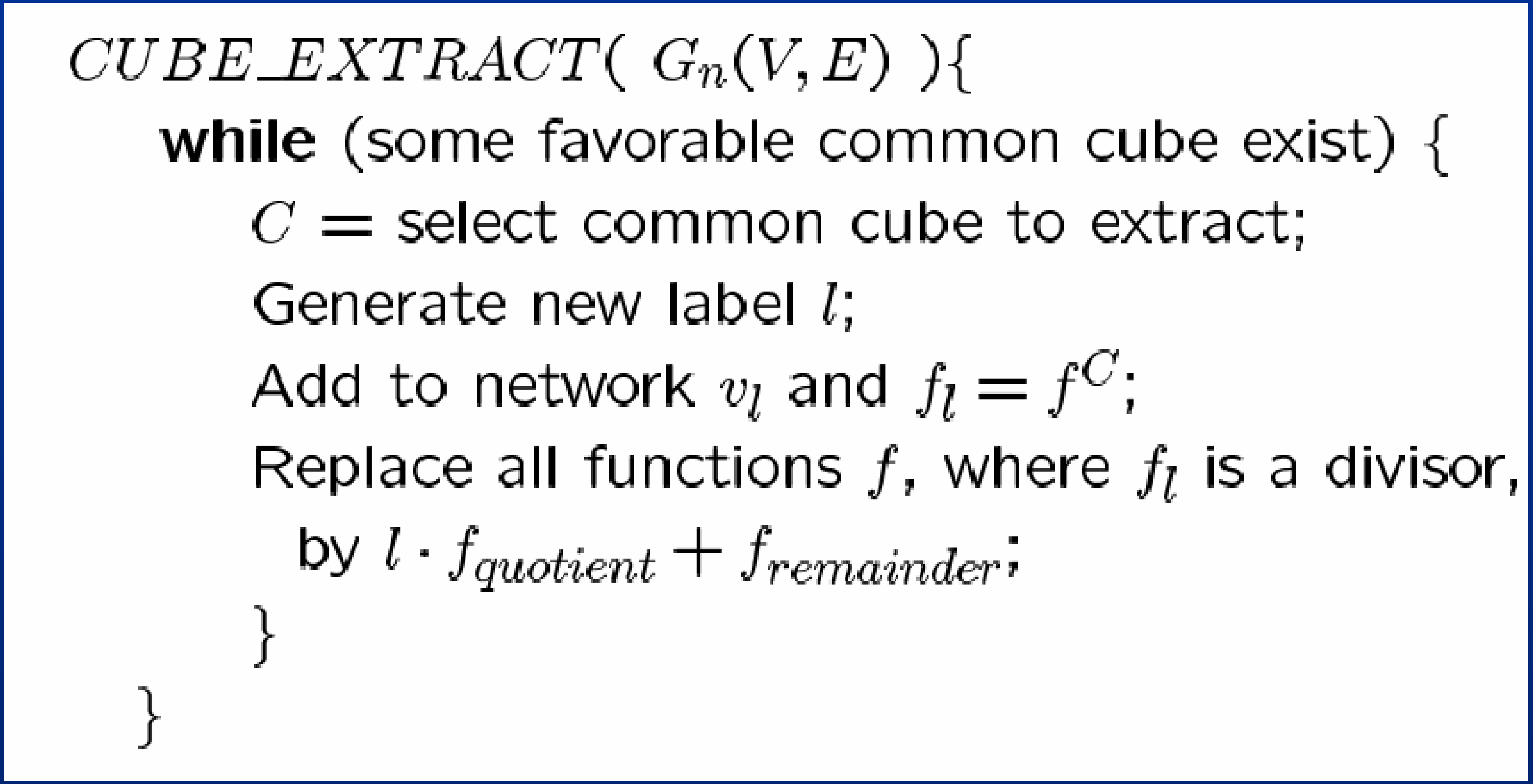
- **Expressions**
  - f_x = ace+bce+de+g
  - f_s = cde+b
- **Auxiliary function**
  - f_{aux} = ace+bce+de+g + cde+b
- **Matrix:**



		var	a	b	c	d	e	g
cube	ID	$R\backslash C$	1	2	3	4	5	6
ace	x	1	1	0	1	0	1	0
bce	x	2	0	1	1	0	1	0
de	x	3	0	0	0	1	1	0
g	x	4	0	0	0	0	0	1
cde	s	5	0	0	1	1	1	0
b	s	6	0	1	0	0	0	0

- Prime rectangle: ({1, 2, 5}, {3, 5})
- Extract cube ce.

Single-Cube Extraction Algorithm

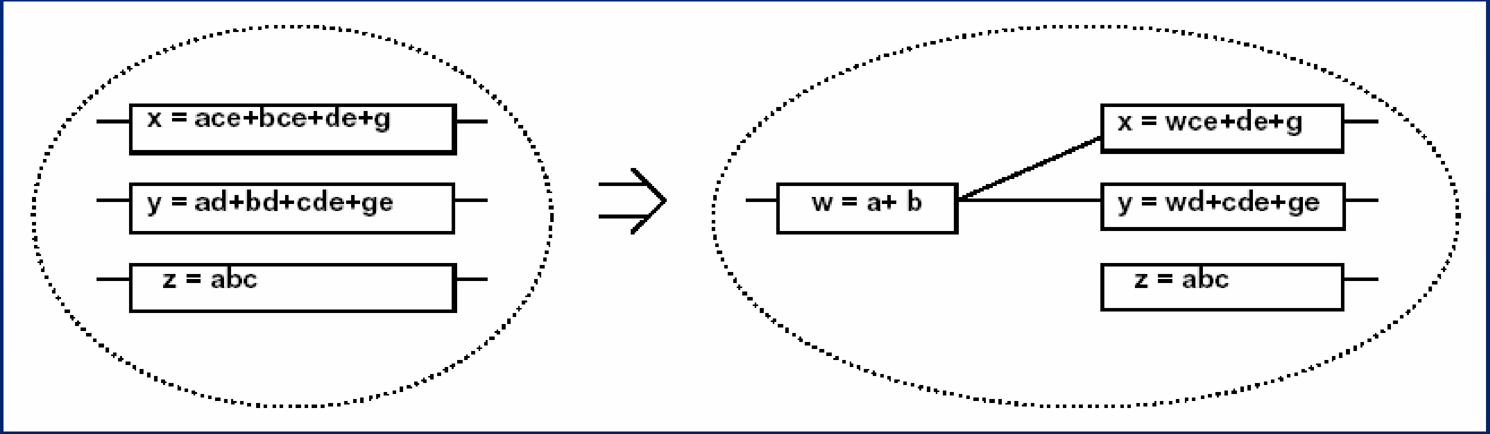


```
$CUBE\_EXTRACT(\ G_n(V,E)\ )\{$
    while (some favorable common cube exist) {
        $C$ = select common cube to extract;
        Generate new label $l$;
        Add to network $v_l$ and $f_l = f^C$;
        Replace all functions $f$, where $f_l$ is a divisor,
          by $l \cdot f_{quotient} + f_{remainder}$;
        }
    }
```

Extraction of an l-variable cube with multiplicity n saves (n l – n – l) literals

Multiple-Cube Extraction …

- We need a kernel/cube matrix.
- Relabeling
  - Cubes by new variables.
  - Kernels by cubes.
- Form auxiliary function
  - Sum of all kernels.
- Extend cube intersection algorithm.

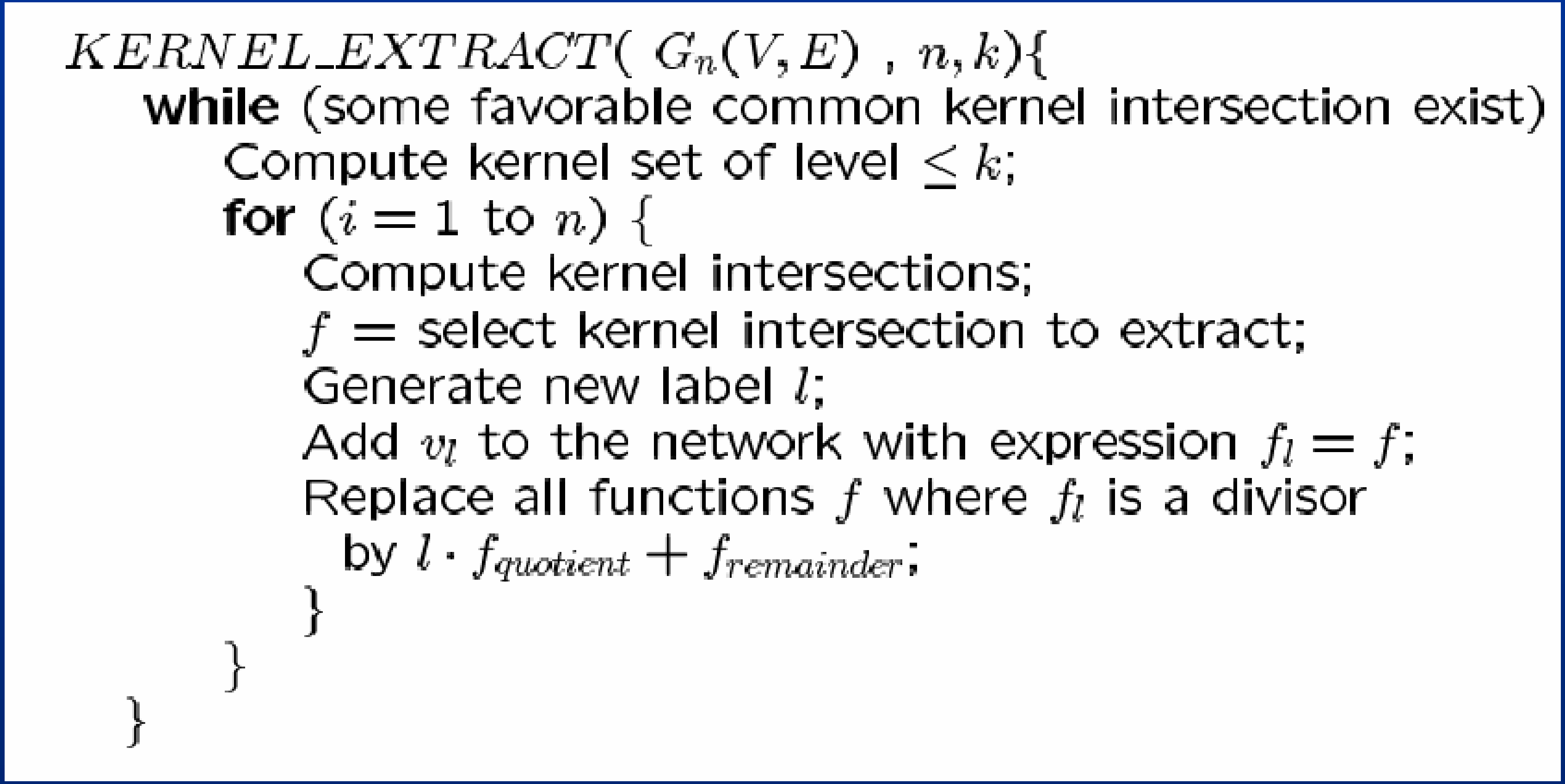


… Multiple-Cube Extraction

- **f_p = ace+bce.**
  - $K(f_p) = \{(a+b)\}$.
- **f_q = ae+be+d.**
  - $K(f_q) = \{(a+b), (ae+be+d)\}$.
- **f_r = ae+be+de.**
  - $K(f_r) = \{(a+b+d)\}$.
- **Relabeling**
  - $x_a = a$; $x_b = b$; $x_{ae} = ae$; $x_{be} = be$; $x_d = d$;
  - $K(f_p) = \{(x_a, x_b)\}$
  - $K(f_q) = \{(x_a, x_b); (x_{ae}, x_{be}, x_d)\}$.
  - $K(f_r) = \{(x_a, x_b, x_d)\}$.
- $f_{aux} = x_a x_b + x_a x_b + x_{ae} x_{be} x_d + x_a x_b x_d$.
- Common cube: $x_a x_b$.
  - $x_a x_b$ corresponds to kernel intersection a+b.
  - Extract a+b from f_p, f_q and f_r.

Cube	x_a	x_b	x_{ae}	x_{be}	x_d
$x_a x_b$	1	1			
$x_a x_b$	1	1			
$x_{ae} x_{be} x_d$			1	1	1
$x_a x_b x_d$	1	1			1

Kernel Extraction Algorithm …



```
$KERNEL\_EXTRACT(\ G_n(V,E)\ ,\ n,k)\{$
    while (some favorable common kernel intersection exist)
        Compute kernel set of level $\leq k$;
        for ($i = 1$ to $n$) {
            Compute kernel intersections;
            $f$ = select kernel intersection to extract;
            Generate new label $l$;
            Add $v_l$ to the network with expression $f_l = f$;
            Replace all functions $f$ where $f_l$ is a divisor
              by $l \cdot f_{quotient} + f_{remainder}$;
            }
        }
    }
```

N indicates the rate at which kernels are recomputed
K indicates the maximum level of the kernel computed

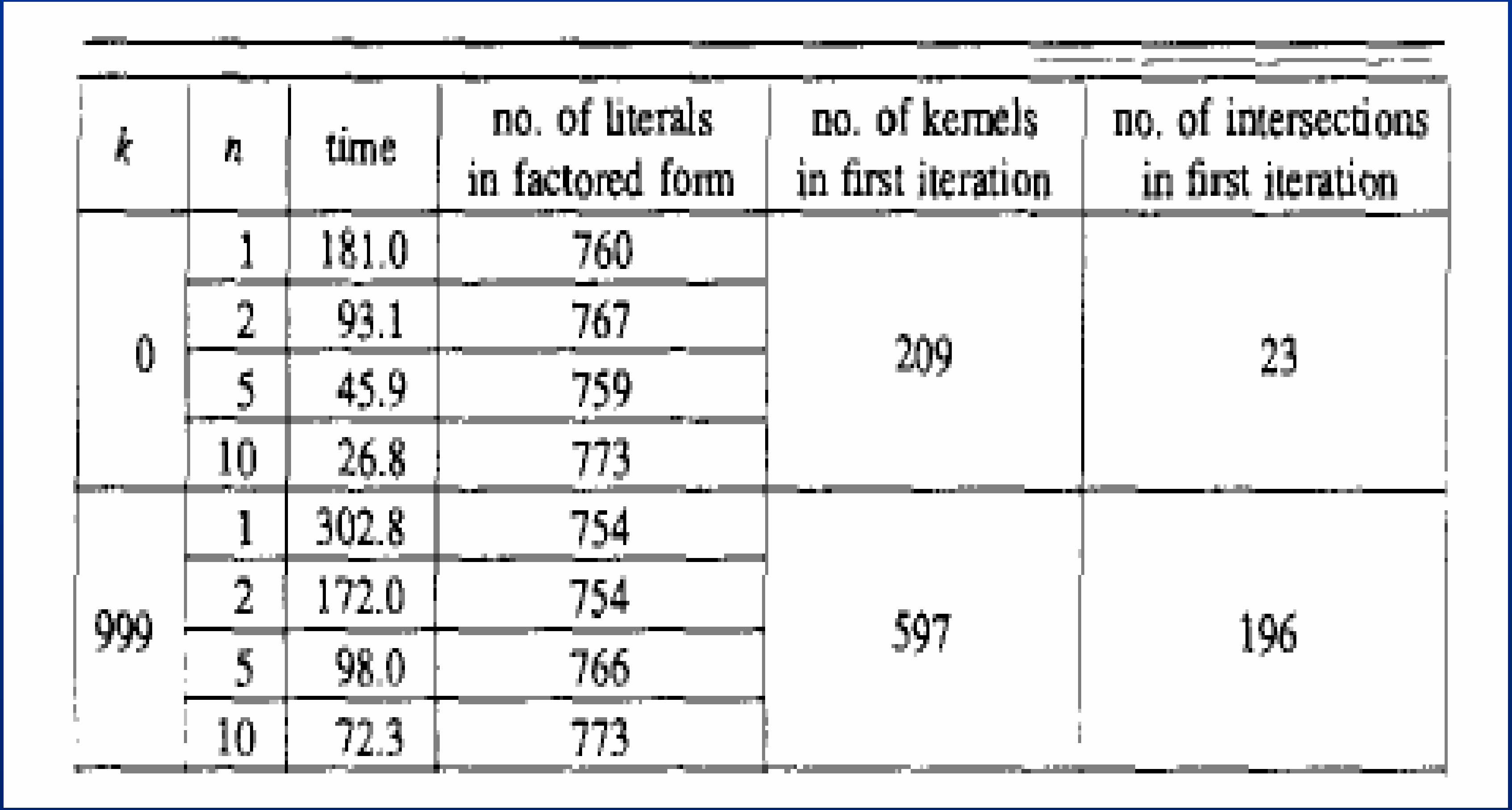
… Kernel Extraction Algorithm

- **Example**
  - F1= ac+bc; Kernels: {(a+b)}
  - F2= ad+bd+cd; Kernels: {(a+b+c)}
  - F3= ab+ac; Kernels: {(b+c)}

Cube	x_a	x_b	x_c
$x_a x_b$	1	1	
$x_a x_b x_c$	1	1	1
$x_b x_c$		1	1

After extracting kernel (a+b), kernel (b+c) is no longer a common kernel. This is why kernel intersections need to be recomputed.

Tradeoffs in Kernel Extraction



k	n	time	no. of literals in factored form	no. of kernels in first iteration	no. of intersections in first iteration
0	1	181.0	760	209	23
	2	93.1	767		
	5	45.9	759		
	10	26.8	773		
999	1	302.8	754	597	196
	2	172.0	754		
	5	98.0	766		
	10	72.3	773		

Area Value of a Kernel …

- Let n be the number of times a kernel is used
- Let l be the number of literals in a kernel and c be the number of cubes in a kernel
- Let CK_i be the co-kernel for kernel i
- Initial cost = $\sum_{i=1 \text{ to } n} (|CK_i|*c+l)=nl + c *\sum_{i=1 \text{ to } n} |CK_i|$
- Resulting cost = $l+\sum_{i=1 \text{ to } n} (|CK_i|+1) = n+l+ \sum_{i=1 \text{ to } n} |CK_i|$
- Value of a kernel = initial cost – resulting cost

  $= \{nl + c *\sum_{i=1 \text{ to } n} |CK_i|\} – \{n+l+ \sum_{i=1 \text{ to } n} |CK_i|\}$

  $= nl – n –l + (c-1) * \sum_{i=1 \text{ to } n} |CK_i|$

… Area Value of a Kernel

- **Example:**
  - X = acd + bcd = (a+b)cd (6 literals)
  - Y = adef + bdef = (a+b)def (8 lietrals)
  - Initial cost = 14 literals
- **After Kernel extraction:**
  - Z=a+b (2 literals)
  - X=Zcd (3 literals)
  - Y=Zdef (4 lietrals)
  - Resulting cost = 9 literals
  - Savings = 14 – 9 = 5 literals
- **Value of kernel = nl – n –l + (c-1) * $\sum_{i=1 \text{ to } n} |CK_i|$**
  - =2*2-2-2+(2-1)*(2+3)=5 literals

Issues in Common Cube and Multiple-Cube Extraction

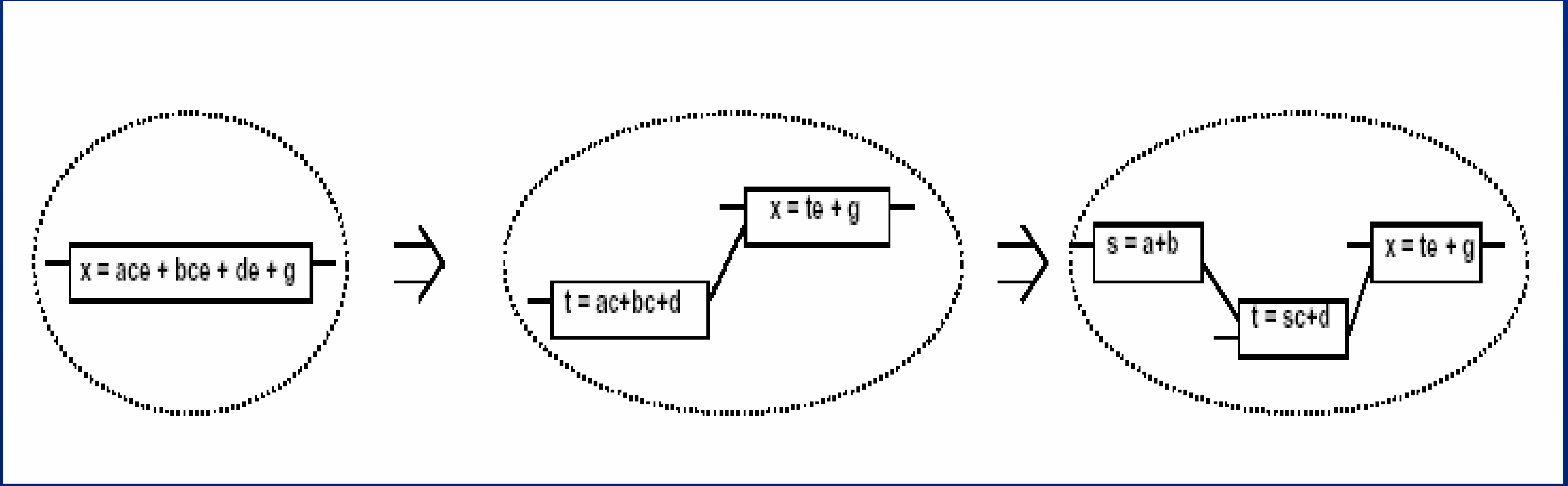
- **Greedy approach can be applied in common cube and multiple-cube extraction**
  - Rectangle selection
  - Matrix update
- **Greedy approach may be myopic**
  - Local gain of one extraction considered at a time
- **Non-prime rectangles can contribute to lower cost covers than prime rectangles**
  - Quine’s theorem cannot be applied to rectangles

Decomposition …

- **Goals of decomposition**
  - Reduce the size of expressions to that typical of library cells.
  - Small-sized expressions more likely to be divisors of other expressions.
- **Different decomposition techniques exist.**
- **Algebraic-division-based decomposition**
  - Give an expression f with $f_{divisor}$ as one of its divisors.
  - Associate a new variable, say t, with the divisor.
  - Reduce original expression to $f = t \cdot f_{quotient} + f_{remainder}$ and $t = f_{divisor}$.
  - Apply decomposition recursively to the divisor, quotient and remainder.
- **Important issue is choice of divisor**
  - A kernel.
  - A level-0 kernel.
  - Evaluate all kernels and select most promising one.

… Decomposition

- f_x = ace+bce+de+g
- Select kernel ac+bc+d.
- Decompose: f_x = te+g; f_t = ac+bc+d;
- Recur on the divisor f_t
  - Select kernel a+b
  - Decompose: f_t = sc+d; f_s = a+b;



Decomposition Algorithm

```
$DECOMPOSE(\ G_n(V, E)\ ,\ k)\{$
    repeat {
        $v_x$ = selected vertex with expression
          whose size is above $k$;
        if ($v_x = \emptyset$) return;
        decompose expression $f_x$;
    }
}
```

K is a threshold that determines the size of nodes to be decomposed.

Factorization Algorithm

- **FACTOR(f)**

```
If (the number of literals in f is one) return f
K =choose_Divisor(f)
(h, r) = Divide(f, k)
Return (FACTOR(k) FACTOR(h) + FACTOR(r))
```

- **Quick factoring: divisor restricted to first level-0 kernel found**
  - Fast and effective
  - Used for area and delay estimation
- **Good factoring: best kernel divisor is chosen**
- **Example: f = ab + ac + bd + ce + cg**
  - Quick factoring: f = a (b+c) + c (e+g) + bd (8 literals)
  - Good factoring: f = c (a+e+g) + b(a+d) (7 literals)

One-Level-0-Kernel

```
One-Level-0-Kernel(f)
    If (|f| ≤1) return 0
    If (L = Literal_Count(f) ≤ 1) return f
    For (i=1; i ≤n; i++){
        If (L(i) > 1){
            C= largest cube containing i s.t. CUBES(f,C)=CUBES(f,i)
            return One-Level-0-Kernel(f/f^C)
        }
    }
```

- **Literal_Count returns a vector of literal counts for each literal.**
  - If all counts are ≤1 then f is a level-0 kernel
- **The first literal with a count greater than one is chosen.**

Fast Extraction (FX) …

- **Very efficient extraction method**
  - Based on extraction of double-cube divisors along with their complements and,
  - Single-cube divisors with two literals.
  - Number of divisors in polynomial domain.
  - Preserves single stuck-at fault testability.
  - [Rajski and Vasudevamurthy 1992].
- **Double-cube divisors are cube-free multiple-cube divisors having exactly two cubes.**
- **The set of double-cube divisors of a function f, denoted $D(f) = \{d \mid d = \{c_i \setminus (c_i \cap c_j), c_j \setminus (c_i \cap c_j)\}\}$ for $i,j=1,..n$, $i \neq j$**
  - n is number of cubes in f.
  - $(c_i \cap c_j)$ is called the base of a double-cube divisor.
  - Empty base is allowed.

… Fast Extraction (FX) …

- Example: f = ade + ag + bcde +bcg.
- Double-cube divisors and their bases:

Double-cube divisors	Base
de+g	a, bc
a+bc	g, de
ade+bcg	{}
ag+bcde	{}

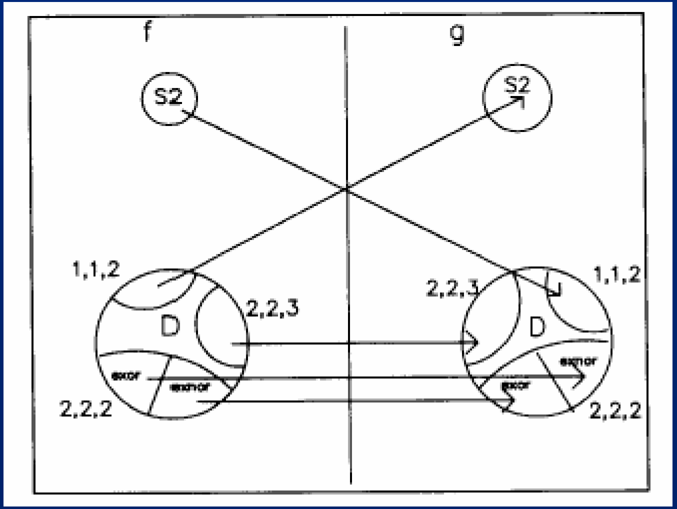
- A subset of double-cube divisors is represented by $D_{x,y,s}$
  - x is number of literals in first cube
  - y is number of literals in second cube
  - s is number of variables in support of D
- A subset of single-cube divisors is denoted by S_k where k is number of literals in single-cube divisor.

Properties of Double-Cube and Single-Cube Divisors

- **Example:**
  - $xy+y'zp \in D_{2,3,4}$
  - $ab \in S_2$
- **$D_{1,1,1}$ and $D_{1,2,2}$ are null set.**
- **For any $d \in D_{1,1,2}$, $d' \in S_2$.**
- **For any $d \in D_{1,2,3}$, $d' \notin D$.**
- **For any $d \in D_{2,2,2}$, d is either XOR or XNOR and $d' \in D_{2,2,2}$.**
- **For any $d \in D_{2,2,3}$, $d' \in D_{2,2,3}$.**
- **For any $d \in D_{2,2,4}$, $d' \notin D$.**

Extraction of Double-cube Divisor along with its Complement

- **Theorem: Let f and g be two expressions. Then, f has neither a complement double-cube divisor nor a complement single-cube divisor in g if**
  - $d_i \neq s_j'$ for every $d_i \in D_{1,1,2}(f)$, $s_j \in S_2(g)$
  - $d_i \neq s_j'$ for every $d_i \in D_{1,1,2}(g)$, $s_j \in S_2(f)$
  - $d_i \neq d_j'$ for every $d_i \in D_{xor}(f)$, $d_j \in D_{xnor}(g)$
  - $d_i \neq d_j'$ for every $d_i \in D_{xnor}(f)$, $d_j \in D_{xor}(g)$
  - $d_i \neq d_j'$ for every $d_i \in D_{2,2,3}(f)$, $d_j \in D_{2,2,3}(g)$



Weights of Double-cube Divisors and Single-Cube Divisors

- **Divisor weight represents literal savings.**
- **Weight of a double-cube divisor d $\in D_{x,y,s}$ is**

  $w(d) = (p-1)(x+y) - p + \sum_{i=1 \text{ to } p} |b_i| + C$
  - p is the number of times double-cube divisor is used
    - Includes complements that are also double-cube divisors
  - $|b_i|$ is the number of literals in base of double-cube divisor
  - C is the number of cubes containing both *a* and *b* in case cube *ab* is a complement of d $\in D_{1,1,2}$
  - (p-1)(x+y) accounts for the number of literals saved by implementing d of size (x+y) once
  - -p accounts for number of literals needed to connect d in its p occurrences
- **Weight of a single-cube divisor c $\in S_2$ is k – 2**
  - K is the number of cubes containing c.

Fast Extraction Algorithm

```
Generate double-cube divisors with weights
Repeat
    Select a double-cube divisor d that has a maximum weight $W_{dmax}$
    Select a single-cube divisor s having a maximum weight $W_{smax}$
    If $W_{dmax} > W_{smax}$ select d else select s
    $W = \max(W_{dmax}, W_{smax})$
    If $W > 0$ then substitute selected divisor
    Recompute weights of affected double-cube divisors
Until ($W<=0$)
```

Fast Extraction Example

- F = abc + a’b’c + ab’d + a’bd + acd + a’b’d’ (18 literals)

d	Base	Weight
ab+a’b’	c	4
bc+b’d	a	0
ac+a’d	b	0
b+d	ac	2
abc+a’b’d’	{}	-1
a’c+ad	b’	0
b’c+bd	a’	0
a’b’+ad	c	0
c+d’	a’b’	1
ab’+a’b	d	4
b’+c	ad	1
ad+a’d’	b’	0
a’b+ac	d	0
bd+b’d’	a’	0
acd+a’b’d’	{}	-1

Single-cube divisors with W_{smax} are either ac or a’b’ or ad with weight of 0

Double-cube divisor=ab + a’b’ is selected

[1]=ab + a’b’
F= [1]c + [1]’d + acd + a’b’d’

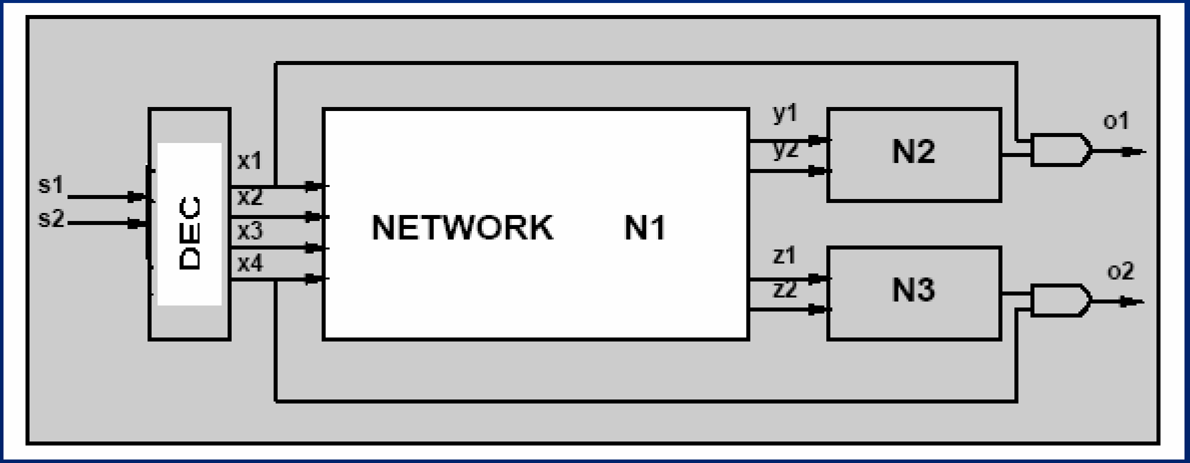
(14 literals)

Boolean Methods

- **Exploit Boolean properties.**
  - Don't care conditions.
- **Minimization of the local functions.**
- **Slower algorithms, better quality results.**
- **Don’t care conditions related to embedding of a function in an environment**
  - Called *external don’t care* conditions
- ***External don’t care* conditions**
  - Controllability
  - Observability

External *Don't Care* Conditions …

- ***Controllability don't care set*** CDC_{in}
  - Input patterns never produced by the environment at the network's input.
- ***Observability don't care set*** ODC_{out}
  - Input patterns representing conditions when an output is not observed by the environment.
  - Relative to each output.
  - Vector notation used: $\mathbf{ODC}_{\mathbf{out}}$.



… External *Don't Care* Conditions

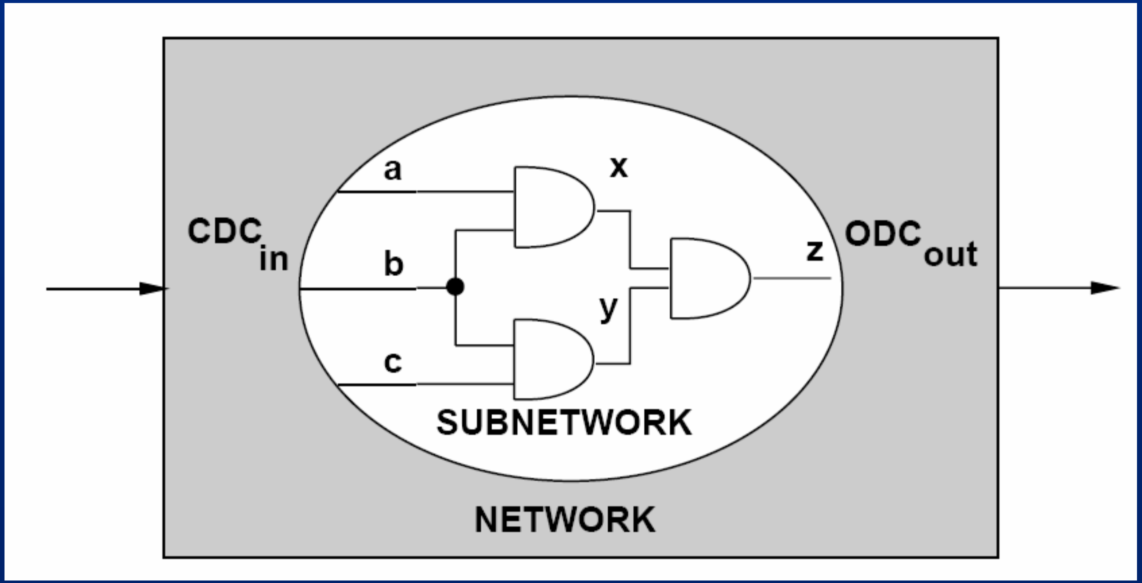
- Inputs driven by a decoder.
- $\mathrm{CDC}_{in} = x_1'x_2'x_3'x_4' + x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4$.
- Outputs observed when $x_1+x_4=1$.

$$\mathrm{ODC}_{out} = \begin{bmatrix} x_1' \\ x_1' \\ x_4' \\ x_4' \end{bmatrix}$$

$$\mathrm{DC}_{ext} = \mathrm{CDC}_{in} + \mathrm{ODC}_{out} = \begin{bmatrix} x_1' + x_2 + x_3 + x_4 \\ x_1' + x_2 + x_3 + x_4 \\ x_4' + x_2 + x_3 + x_1 \\ x_4' + x_2 + x_3 + x_1 \end{bmatrix}$$

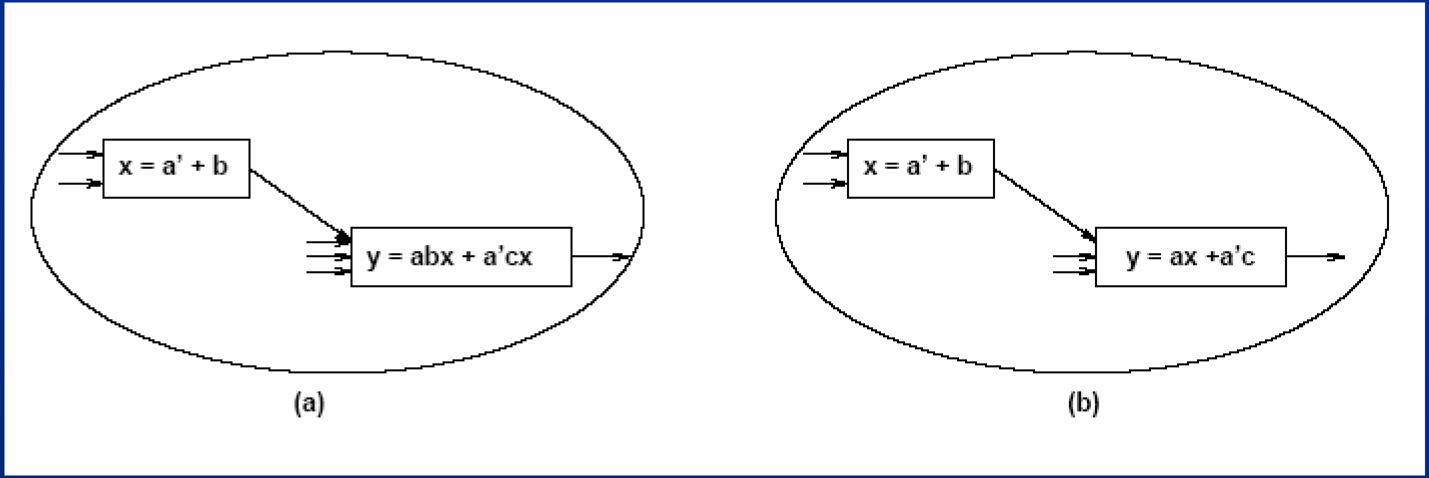
Internal *Don't Care* Conditions …

- **Induced by the network structure.**
- **Controllability *don't care* conditions**
  - Patterns never produced at the inputs of a subnetwork.
- **Observability *don't care* conditions**
  - Patterns such that the outputs of a subnetwork are not observed.



… Internal *Don't Care* Conditions

- Example: x = a'+b; y= abx + a'cx



- CDC of v_y includes ab'x+a'x'.
  - ab'⇒x=0; ab'x is a don't care condition
  - a' ⇒ x=1; a'x' is a don't care condition
- Minimize f_y to obtain: f_y = ax+a'c.

Satisfiability *Don't Care* Conditions

- **Invariant of the network**
  - $x = f_x \rightarrow x \neq f_x \subseteq$ SDC.

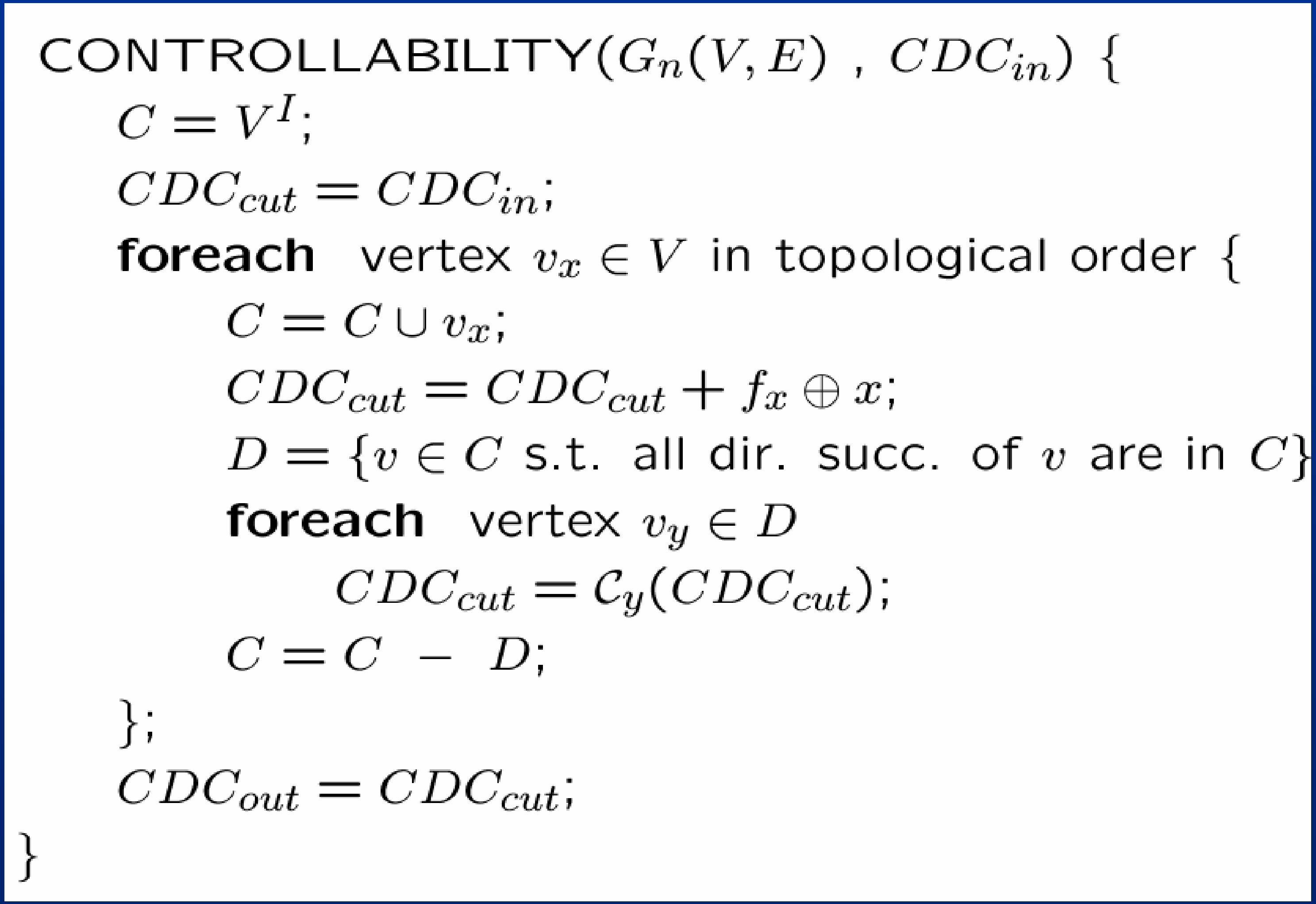
$$SDC = \sum_{v_x \in V^G} x \oplus f_x$$

- **Useful to compute controllability *don't cares*.**
- **Example**
  - Assume x = a’ + b
  - Since x ≠ (a’ + b) is not possible, x ⊕ (a’ + b)=x’a’ + x’b + xab’ is a *don’t care* condition.

CDC Computation …

- **Network traversal algorithm**
  - Consider different cuts moving from input to output.
- **Initial CDC is CDC_{in}.**
- **Move cut forward.**
  - Consider SDC contributions of predecessors.
  - Remove unneeded variables by consensus.
- **Consensus of a function f with respect to variable x is $f_x \cdot f_{x'}$**

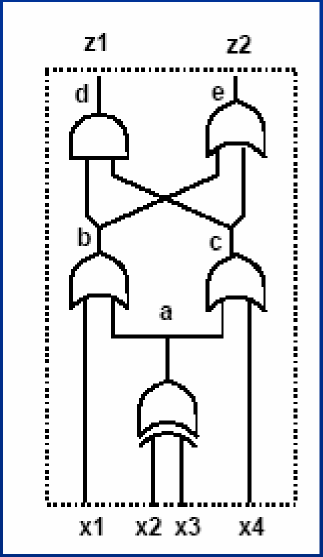
… CDC Computation …

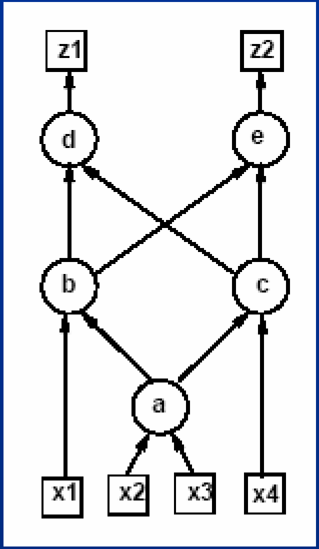


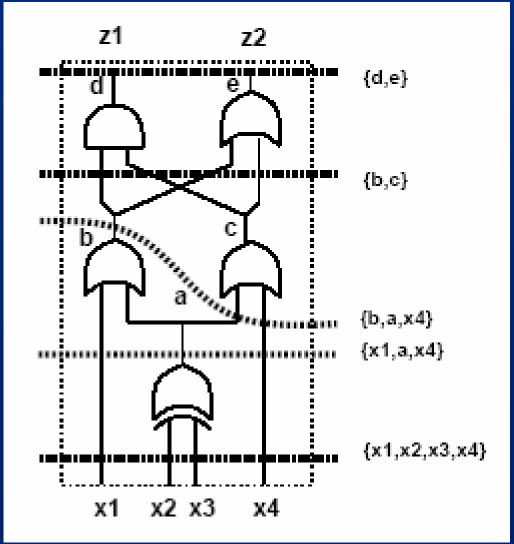
```
CONTROLLABILITY($G_n(V,E)$ , $CDC_{in}$) {
    $C = V^I$;
    $CDC_{cut} = CDC_{in}$;
    foreach vertex $v_x \in V$ in topological order {
        $C = C \cup v_x$;
        $CDC_{cut} = CDC_{cut} + f_x \oplus x$;
        $D = \{v \in C$ s.t. all dir. succ. of $v$ are in $C\}$
        foreach vertex $v_y \in D$
            $CDC_{cut} = \mathcal{C}_y(CDC_{cut})$;
        $C = C - D$;
    };
    $CDC_{out} = CDC_{cut}$;
}
```

… CDC Computation …

- **Assume $CDC_{in} = x_1'x_4'$.**
- **Select vertex v_a**
  - Contribution to CDC_{cut}: $a \oplus (x_2 \oplus x_3)$.
    - $CDC_{cut} = x_1'x_4' + a \oplus (x_2 \oplus x_3)$.
  - Drop variables $D = \{x_2, x_3\}$
  - $CDC_{cut} = x_1'x_4'$.
- **Select vertex v_b**
  - Contribution to CDC_{cut} : $b \oplus (x_1 + a)$.
    - $CDC_{cut} = x_1'x_4' + b \oplus (x_1 + a)$.
  - Drop variable $D = \{x_1\}$
  - $CDC_{cut} = b'x_4' + b'a$.

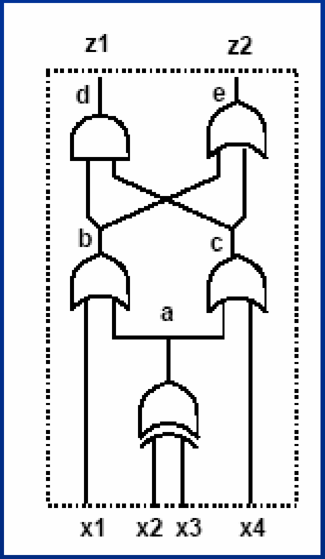


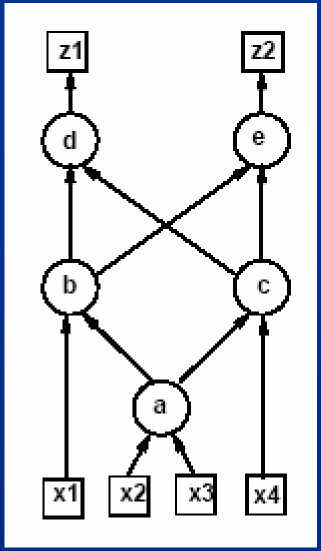


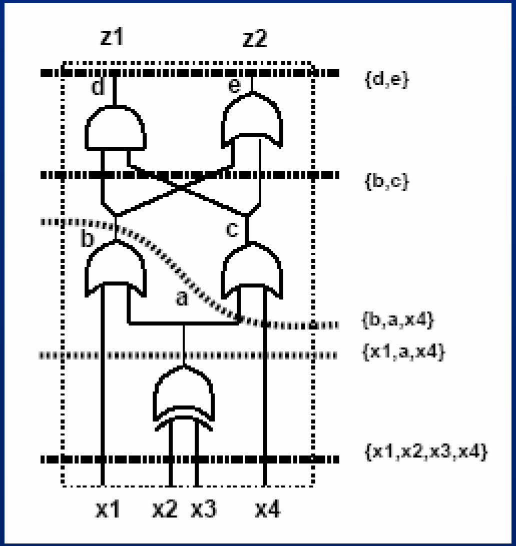


… CDC Computation

- **Select vertex v_c**
  - Contribution to CDC_{cut}: $c \oplus (x_4 + a)$.
    - $CDC_{cut} = b'x_4' + b'a + c \oplus (x_4 + a)$.
  - Drop variables $D = \{a, x_4\}$
  - $CDC_{cut} = b'c'$.
- **Select vertex v_d**
  - Contribution to CDC_{cut}: $d \oplus (bc)$.
  - $CDC_{cut} = b'c' + d \oplus (bc)$.
- **Select vertex v_e**
  - Contribution to CDC_{cut}: $e \oplus (b + c)$.
    - $CDC_{cut} = b'c' + d \oplus (bc) + e \oplus (b + c)$.
  - Drop variables $D = \{b, c\}$
  - $CDC_{cut} = e'$.
- **$CDC_{cut} = e' = z_2'$.**

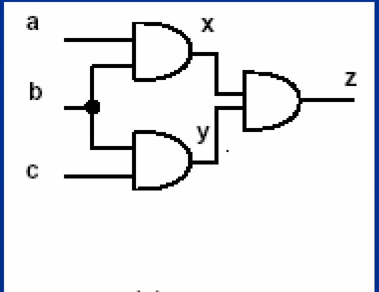




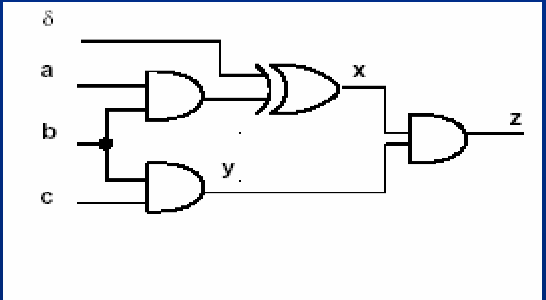


Network Perturbation

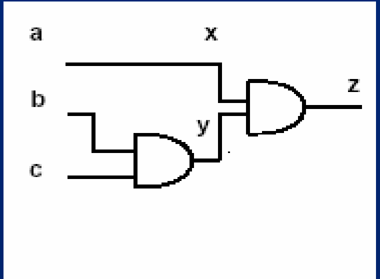
- Modify network by adding an extra input δ.
- Extra input can flip polarity of a signal x.
- Replace local function f_x by $f_x \oplus \delta$.
- Perturbed terminal behavior: $f^x(\delta)$.
- A variable is observable if a change in its polarity is perceived at an output.
- Observability *don't-care* set ODC for variable x is $(f^x(0) \oplus f^x(1))'$
  - $f^x(0)=abc$
  - $f^x(1)=a'bc$
  - $ODC_x= (abc \oplus a'bc)' = b'+c'$
  - Minimizing $f_x=ab$ with $ODC_x= b'+c'$ leads to $f_x=a$.



(a)



(b)



(c)

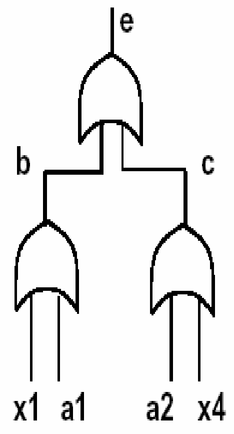
Observability *Don't Care* Conditions

- Conditions under which a change in polarity of a signal x is not perceived at the outputs.
- Complement of the Boolean Difference
  - $\partial f/\partial x = f|_{x=1} \oplus f|_{x=0}$
- Equivalence of perturbed function: $(f^x(0) \oplus f^x(1))'$.
- Observability *don't care* computation
  - Problem
    - Outputs are not expressed as function of all variables.
    - If network is flattened to obtain f, it may explode in size.
  - Requirement
    - Local rules for ODC computation.
    - Network traversal.

Observability *Don't Care* Computation …

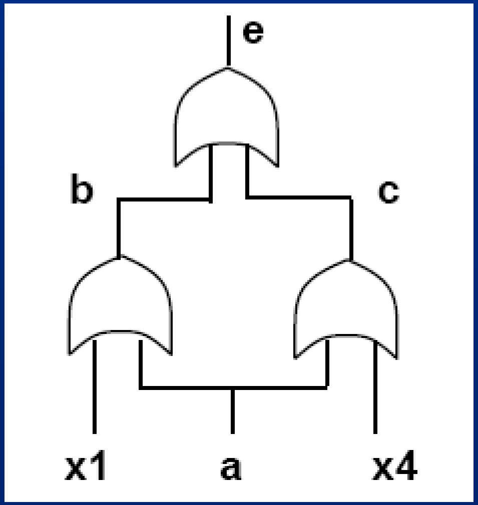
- **Assume single-output network with tree structure.**
- **Traverse network tree.**
- **At root**
  - ODC_{out} is given.
- **At internal vertices assuming y is the output of x**
  - $ODC_x = (\partial f_y/\partial x)' + ODC_y = (f_y|_{x=1} \oplus f_y|_{x=0})' + ODC_y$
- Example
  - Assume $ODC_{out} = ODC_e = 0$.
  - $ODC_b = (\partial f_e/\partial b)'$
    $= ((b+c)|_{b=1} \oplus (b+c)|_{b=0})' = c$.
  - $ODC_c = (\partial f_e/\partial c)' = b$.
  - $ODC_{x1} = ODC_b + (\partial f_b/\partial x_1)' = c+a1$.
  - …

$$\begin{aligned} e &= b+c \\ b &= x_1+a_1 \\ c &= x_4+a_2 \end{aligned}$$



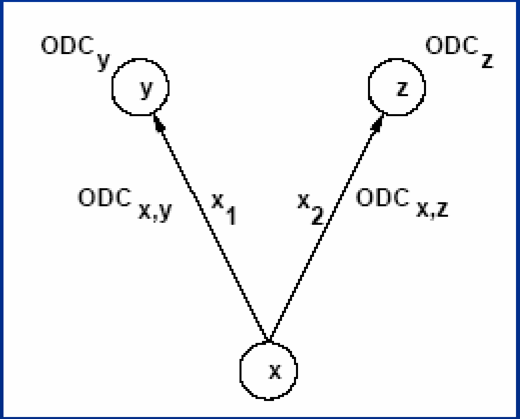
… Observability Don't Care Computation

- **General networks have fanout re-convergence.**
- **For each vertex with two (or more) fanout stems**
  - The contribution of the ODC along the stems cannot be added.
  - Wrong assumption is intersecting them
    - $ODC_{a,b}=x_1+c=x_1+a+x_4$
    - $ODC_{a,c}=x_4+b=x_4+a+x_1$
    - $ODC_{a,b} \cap ODC_{a,c}=x_1+a+x_4$
    - Variable a is not redundant
  - Interplay of different paths.
- **More elaborate analysis.**



Two-way Fanout Stem …

- **Compute ODC sets associated with edges.**
- **Combine ODCs at vertex.**
- **Formula derivation**
  - Assume two equal perturbations on the edges.



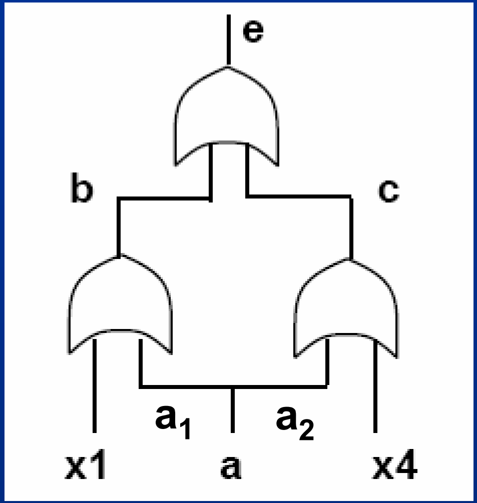
$$\begin{aligned}
\mathbf{ODC}_x &= \mathbf{f}^{x1,x2}(1,1) \;\overline{\oplus}\; \mathbf{f}^{x1,x2}(0,0) \\
&= \mathbf{f}^{x1,x2}(1,1) \;\overline{\oplus}\; \mathbf{f}^{x1,x2}(0,0) \\
&\quad \overline{\oplus}\; (\mathbf{f}^{x1,x2}(0,1) \;\overline{\oplus}\; \mathbf{f}^{x1,x2}(0,1)) \\
&= (\mathbf{f}^{x1,x2}(1,1) \;\overline{\oplus}\; \mathbf{f}^{x1,x2}(0,1)) \\
&\quad \overline{\oplus}\; (\mathbf{f}^{x1,x2}(0,1) \;\overline{\oplus}\; \mathbf{f}^{x1,x2}(0,0)) \\
&= \mathbf{ODC}_{x,y}|_{\delta_2=1} \;\overline{\oplus}\; \mathbf{ODC}_{x,z}|_{\delta_1=0} \\
&= \mathbf{ODC}_{x,y}|_{x_2=x'} \;\overline{\oplus}\; \mathbf{ODC}_{x,z}|_{x_1=x} \\
&= \mathbf{ODC}_{x,y}|_{x=x'} \;\overline{\oplus}\; \mathbf{ODC}_{x,z}
\end{aligned}$$

… Two-way Fanout Stem

- $ODC_{a,b} = x_1+c = x_1+a_2+x_4$
- $ODC_{a,c} = x_4+b = x_4+a_1+x_1$
- $ODC_a = (ODC_{a,b\,|a2=a'} \oplus ODC_{a,c})'$

  $= ((x_1+a'+x_4) \oplus (x_4+a+x_1))'$

  $= x_1+x_4$



Multi-Way Stems Theorem

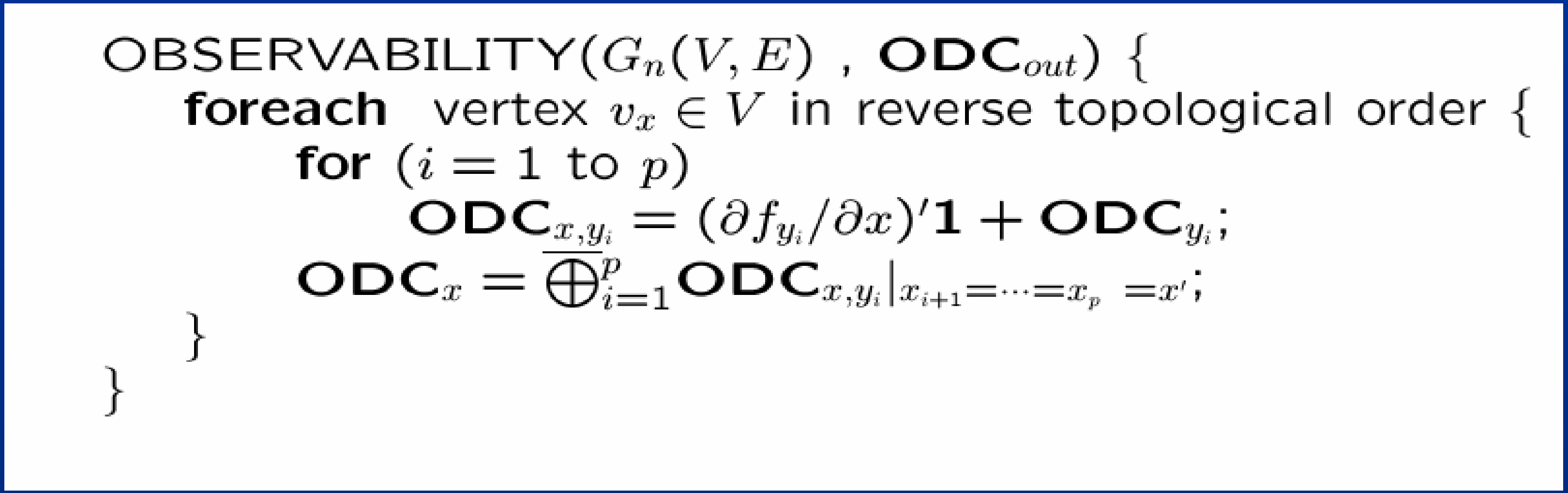
- Let $v_x \in V$ be any internal or input vertex.
- Let $\{x_i;\ i = 1, 2, \ldots, p\}$ be the edge variables corresponding to $\{(x, y_i)\ ;\ i = 1, 2, \ldots, p\}$.
- Let ODC_{x,y_i} ; $i = 1, 2, \ldots, p$ be the edge ODCs.

$$\mathbf{ODC}_x = \overline{\bigoplus}_{i=1}^{p} \mathbf{ODC}_{x,y_i}|_{x_{i+1}=\cdots=x_p\ =x'}$$

- For a 3-fanout stem variable x:

$$\mathrm{ODC}_x = \mathrm{ODC}_{x,y1\ |x2=x3=x'} \oplus \mathrm{ODC}_{x,y2\ |x3=x'} \oplus \mathrm{ODC}_{x,y3}$$

Observability *Don't Care* Algorithm …



```
OBSERVABILITY($G_n(V,E)$ , ODC$_{out}$) {
    foreach vertex $v_x \in V$ in reverse topological order {
        for ($i = 1$ to $p$)
            ODC$_{x,y_i}$ $= (\partial f_{y_i}/\partial x)'\mathbf{1} +$ ODC$_{y_i}$;
        ODC$_x$ $= \overline{\bigoplus}_{i=1}^{p}$ ODC$_{x,y_i}|_{x_{i+1}=\cdots=x_p\ =x'}$;
    }
}
```

- **For each variable, intersection of ODC at all outputs yields condition under which output is not observed**
  - Global ODC of a variable
- **The global ODC conditions of the input variables is the input observability *don't care* set ODC$_{in}$.**
  - May be used as external ODC sets for optimizing a network feeding the one under consideration

… Observability *Don't Care* Algorithm

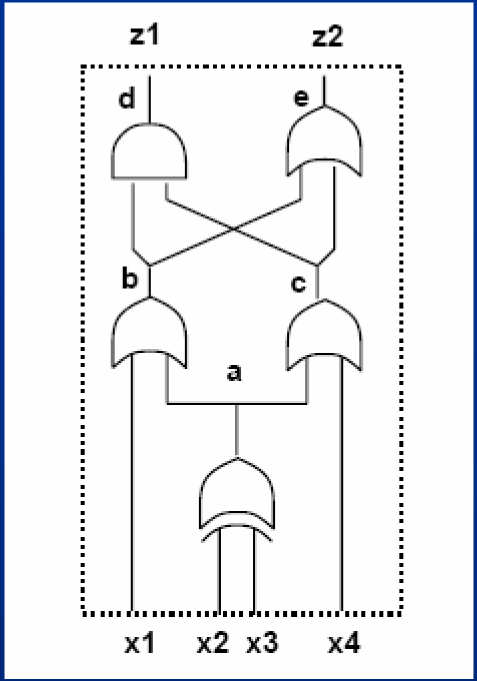
$$\text{ODC}_d = \begin{pmatrix} 0 \\ 1 \end{pmatrix} ;\text{ODC}_e = \begin{pmatrix} 1 \\ 0 \end{pmatrix} ;\text{ODC}_c = \begin{pmatrix} b' \\ b \end{pmatrix} ;\text{ODC}_b = \begin{pmatrix} c' \\ c \end{pmatrix}$$

$$\text{ODC}_{a,b} = \begin{pmatrix} c' + x_1 \\ c + x_1 \end{pmatrix} = \begin{pmatrix} a'x_4' + x_1 \\ a + x_4 + x_1 \end{pmatrix}$$

$$\text{ODC}_{a,c} = \begin{pmatrix} b' + x_4 \\ b + x_4 \end{pmatrix} = \begin{pmatrix} a'x_1' + x_4 \\ a + x_1 + x_4 \end{pmatrix}$$

$$\text{ODC}_a = \text{ODC}_{a,b}|_{a=a'} \overline{\oplus} \text{ODC}_{a,c} = \begin{pmatrix} ax_4' + x_1 \\ a' + x_4 + x_1 \end{pmatrix} \overline{\oplus} \begin{pmatrix} a'x_1' + x_4 \\ a + x_1 + x_4 \end{pmatrix} =$$

$$= \begin{pmatrix} x_1x_4 \\ x_1 + x_4 \end{pmatrix}$$



Global ODC of a is (x1x4)(x1+x4)=x1x4

Transformations with *Don't Cares*

- **Boolean simplification**
  - Use standard minimizer (Espresso).
  - Minimize the number of literals.
- **Boolean substitution**
  - Simplify a function by adding an extra input.
  - Equivalent to simplification with global *don't care* conditions.
- **Example**
  - Substitute $q = a+cd$ into $f_h = a+bcd+e$ to get $f_h = a+bq+e$.
  - SDC set: $q \oplus (a+cd) = q'a+q'cd+qa'(cd)'$.
  - Simplify $f_h = a+bcd+e$ with $q'a+q'cd+qa'(cd)'$ as *don't care*.
  - Simplication yields $f_h = a+bq+e$.
  - One literal less by changing the support of f_h.

Single-Vertex Optimization

```
$SIMPLIFY\_SV(\ G_n(V,E)\ )\{$
    **repeat** {
        $v_x$ = selected vertex ;
        Compute the local *don't care* set $DC_x$;
        Optimize the function $f_x$ ;
    }**until** (no more reduction is possible)
}
```

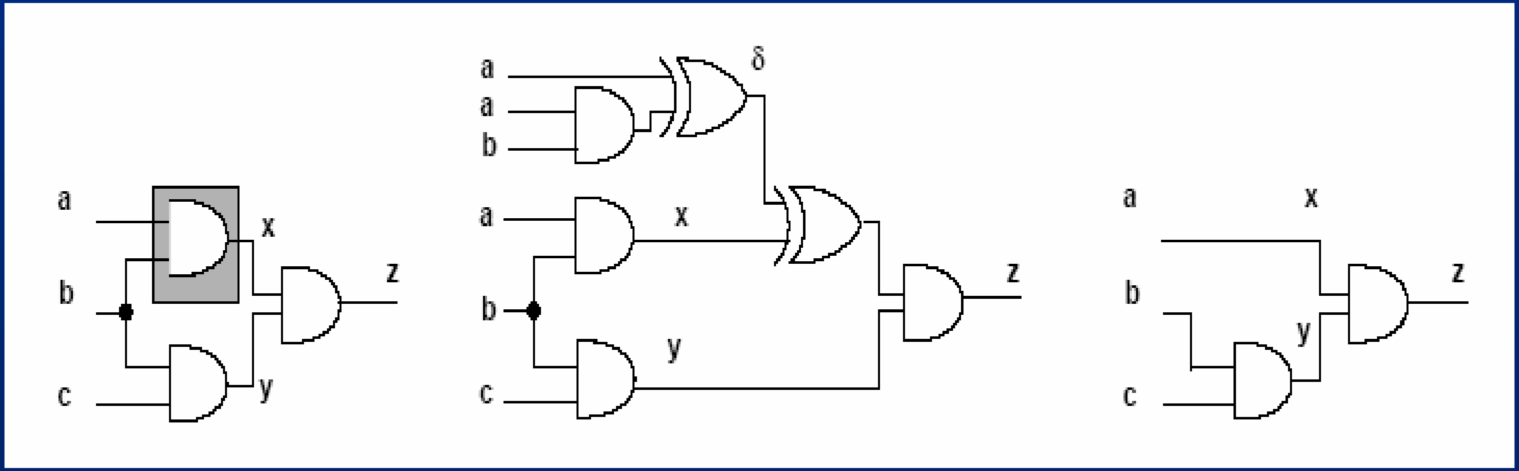
Optimization and Perturbations …

- **Replace f_x by g_x.**
- **Perturbation $\delta_x = f_x \oplus g_x$.**
- **Condition for feasible replacement**
  - Perturbation bounded by local don't care set
  - $\delta_x \subseteq DC_{ext} + ODC_x$
  - If f_x and g_x have the same support set S(x) then
    - $\delta_x \subseteq DC_{ext} + ODC_x + CDC_{S(x)}$
  - If $S(g_x)$ includes network variables
    - $\delta_x \subseteq DC_{ext} + ODC_x + SDC_x$

$$\boxed{SDC_x = \sum_{v_y \in V : v_y \neq v_x} y \oplus f_y}$$

… Optimization and Perturbations

- No external don't care set.
- Replace AND by wire: $g_x = a$
- Analysis
  - $\delta_x = f_x \oplus g_x = ab \oplus a = ab'$.
  - $ODC_x = y' = b' + c'$.
  - $\delta_x = ab' \subseteq DC_x = b' + c' \Rightarrow$ feasible!



Synthesis and Testability

- **Testability**
  - Ease of testing a circuit.
- **Assumptions**
  - Combinational circuit.
  - Single or multiple stuck-at faults.
- **Full testability**
  - Possible to generate test set for all faults.
- Synergy between synthesis and testing.
- Testable networks correlate to small-area networks.
- Don't care conditions play a major role.

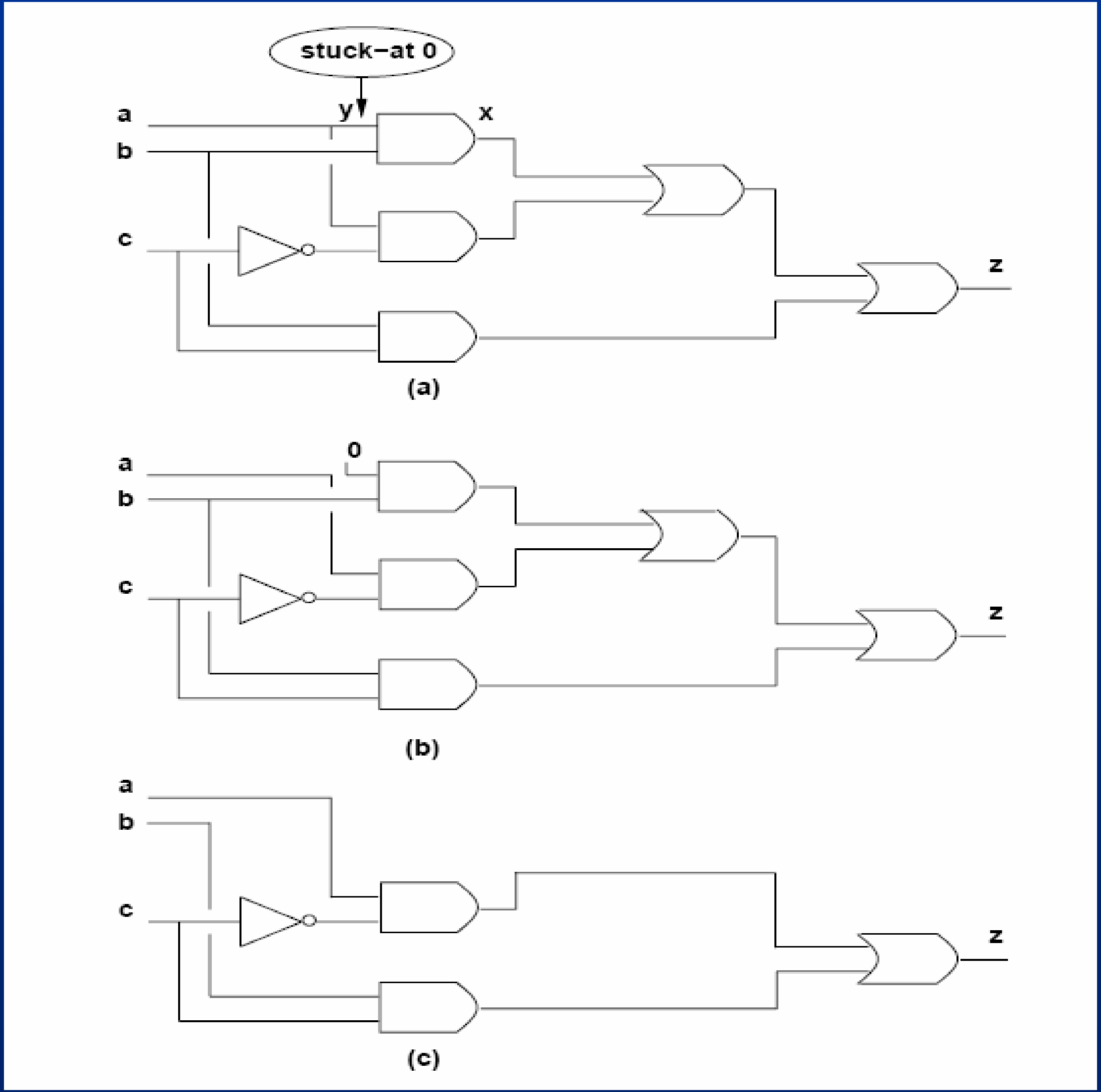
Test for Stuck-at-Faults

- **Net y stuck-at 0**
  - Input pattern that sets y to true.
  - Observe output.
  - Output of faulty circuit differs.
  - $\{t \mid y(t) \,.\, ODC'_y(t) = 1\}$.
- **Net y stuck-at 1**
  - Same, but set y to false.
  - $\{t \mid y'(t) \,.\, ODC'_y(t) = 1\}$.
- **Need controllability and observability.**

Using Testing Methods for Synthesis …

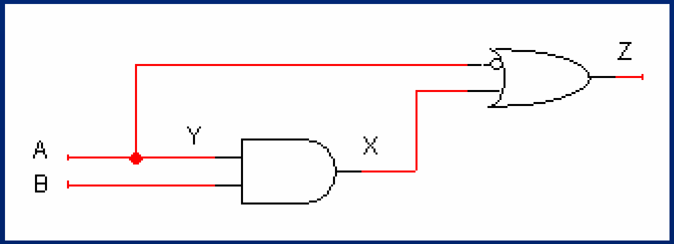
- **Redundancy removal.**
  - Use ATPG to search for untestable faults.
- **If stuck-at 0 on net y is untestable**
  - Set y = 0.
  - Propagate constant.
- **If stuck-at 1 on y is untestable**
  - Set y = 1.
  - Propagate constant.

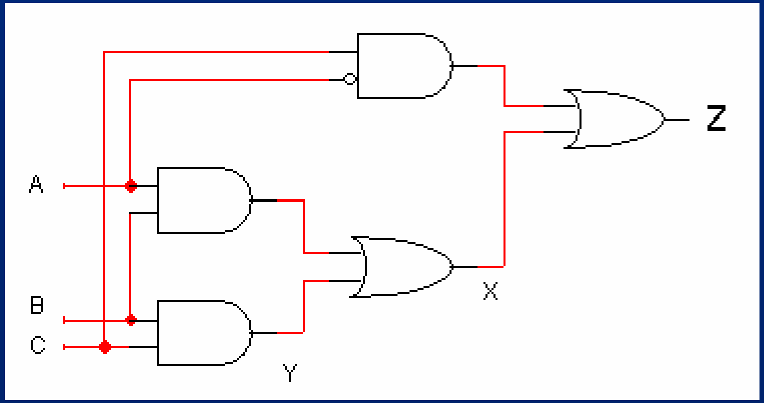
… Using Testing Methods for Synthesis



Redundancy Removal and Perturbation Analysis

- Stuck-at 0 on y.
  - y set to 0. Namely $g_x = f_x|_{y=0}$.
  - Perturbation
    - $\delta = f_x \oplus f_x|_{y=0} = y \cdot \partial f_x / \partial y$.
- Perturbation is feasible $\Leftrightarrow$ fault is untestable.
- $\delta = y \cdot \partial f_x / \partial y \subseteq DC_x \Leftrightarrow$ fault is untestable
- Making f_x prime and irredundant with respect to DC_x guarantees that all single stuck-at faults in f_x are testable.





Synthesis for Testability

- **Synthesize networks that are fully testable.**
  - Single stuck-at faults.
  - Multiple stuck-at faults.
- **Two-level forms**
  - Full testability for single stuck-at faults
    - Prime and irredundant cover.
  - Full testability for multiple stuck-at faults
    - Prime and irredundant cover when
      - Single-output function.
      - No product term sharing.
      - Each component is PI.

… Synthesis for Testability

- **A complete single-stuck-at fault test set for a single-output sum-of-product circuit is a complete test set for all multiple stuck-at faults.**
- **Single stuck-at fault testability of multiple-level network does not imply multiple stuck-at fault testability.**
- **Fast extraction transformations are single stuck-at fault test-set preserving transformations.**
- **Algebraic transformations preserve multiple stuck-at fault testability but not single stuck-at fault testability**
  - Factorization
  - Substitution (without complement)
  - Cube and kernel extraction

Synthesis of Testable Multiple-Level Networks …

- A logic network $G_n(V, E)$, with local functions in sum of product form.
- Prime and irredundant (PI)
  - No literal nor implicant of any local function can be dropped.
- Simultaneously prime and irredundant (SPI)
  - No subset of literals and/or implicants can be dropped.
- A logic network is PI if and only if
  - its AND-OR implementation is fully testable for single stuck-at faults.
- A logic network is SPI if and only if
  - its AND-OR implementation is fully testable for multiple stuck-at faults.

… Synthesis of Testable Multiple-Level Networks

- **Compute full local *don't care* sets.**
  - Make all local functions PI w.r. to *don't care* sets.
- **Pitfall**
  - Don't cares change as functions change.
- **Solution**
  - Iteration (Espresso-MLD).
  - If iteration converges, network is fully testable.
- **Flatten to two-level form.**
  - When possible -- no size explosion.
- **Make SPI by disjoint logic minimization.**
- **Reconstruct multiple-level network**
  - Algebraic transformations preserve multifault testability.

Timing Issues in Multiple-Level Logic Optimization

- **Timing optimization is crucial for achieving competitive logic design.**
- **Timing verification: Check that a circuit runs at speed**
  - Satisfies I/O delay constraints.
  - Satisfies cycle-time constraints.
  - Delay modeling.
  - Critical paths.
  - The false path problem.
- **Algorithms for timing optimization.**
  - Minimum area subject to delay constraints.
  - Minimum delay (subject to area constraints).

Delay Modeling

- **Gate delay modeling**
  - Straightforward for bound networks.
  - Approximations for unbound networks.
- **Network delay modeling**
  - Compute signal propagation
    - Topological methods.
    - Logic/topological methods.
- **Gate delay modeling for unbound networks**
  - Virtual gates: Logic expressions.
  - Stage delay model: Unit delay per vertex.
  - Refined models: Depending on size and fanout.

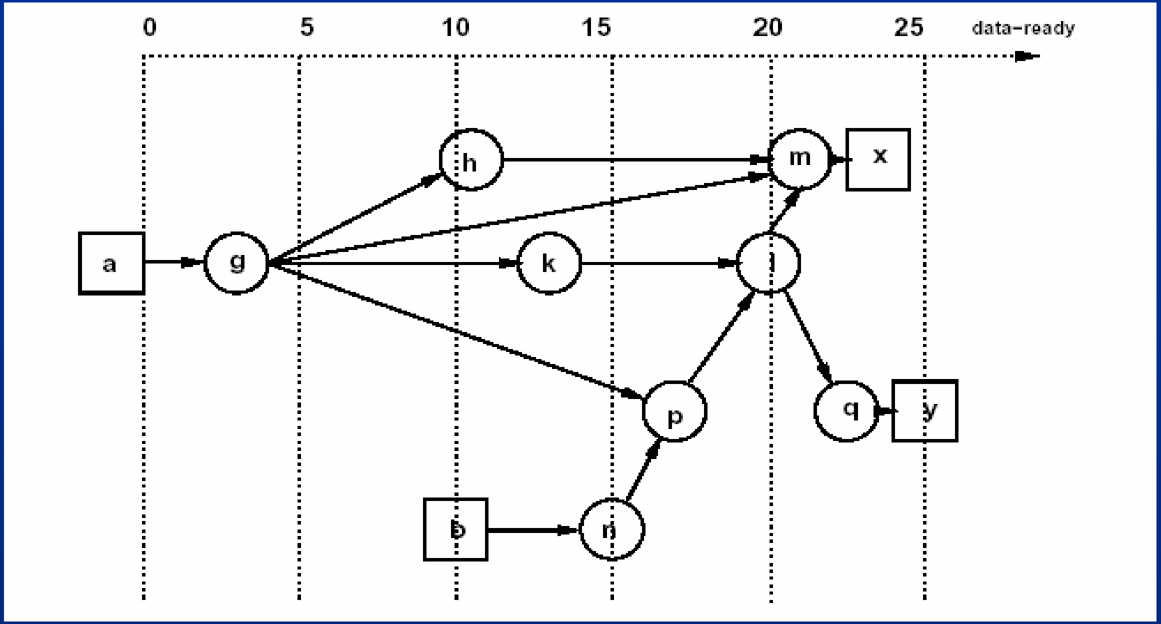
Network Delay Modeling …

- **For each vertex v_i.**
  - Propagation delay d_i.
- **Data-ready time t_i.**
  - Denotes the time at which the data is ready at the output.
  - Input data-ready times denote when inputs are available.
  - Computed elsewhere by forward traversal

$$t_i = d_i + \max_{j|(v_j,v_i)\in E} t_j$$

- **The maximum data-ready time occurring at an output vertex**
  - Corresponds to the longest propagation delay path
  - Called topological critical path

… Network Delay Modeling …



$t_g= 3+0=3$
$t_h= 8+3=11$
$t_k= 10+3=13$
$t_n= 5+10=15$
$t_p= 2+\max\{15,3\}=17$
$t_l= 3+\max\{13,17\}=20$
$t_m= 1+\max\{3,11,20\}=21$
$t_x= 2+21=23$
$t_q= 2+20=22$
$t_y= 3+22=25$

- Assume $t_a=0$ and $t_b=10$.
- Propagation delays
  - $d_g = 3$; $d_h = 8$; $d_m = 1$; $d_k = 10$; $d_l = 3$;
  - $d_n = 5$; $d_p = 2$; $d_q = 2$; $d_x = 2$; $d_y = 3$;
  - Maximum data-ready time is $t_y=25$
  - Topological critical path: $(v_b, v_n, v_p, v_l, v_q, v_y)$.

… Network Delay Modeling …

- For each vertex v_i.
  - **Required data-ready time $\bar{t}_i$.**
    - Specified at the primary outputs.
    - Computed elsewhere by backward traversal

$$\bar{t}_i = \min_{j|(v_i,v_j)\in E} \bar{t}_j - d_j$$

  - Slack s_i.
    - Difference between required and actual data-ready times

$$s_i = \bar{t}_i - t_i.$$

… Network Delay Modeling

- **Required data-ready times**
  - $\bar{t}_x = 25$ and $\bar{t}_y = 25$.

Propagation Delays :

$d_g = 3$; $d_h = 8$; $d_m = 1$; $d_k = 10$; $d_l = 3$; $d_n = 5$; $d_p = 2$; $d_q = 2$; $d_x = 2$; $d_y = 3$

Data-Ready Times:

$t_g = 3+0=3$
$t_h = 8+3=11$
$t_k = 10+3=13$
$t_n = 5+10=15$
$t_p = 2+\max\{15,3\}=17$
$t_l = 3+\max\{13,17\}=20$
$t_m = 1+\max\{3,11,20\}=21$
$t_x = 2+21=23$
$t_q = 2+20=22$
$t_y = 3+22=25$

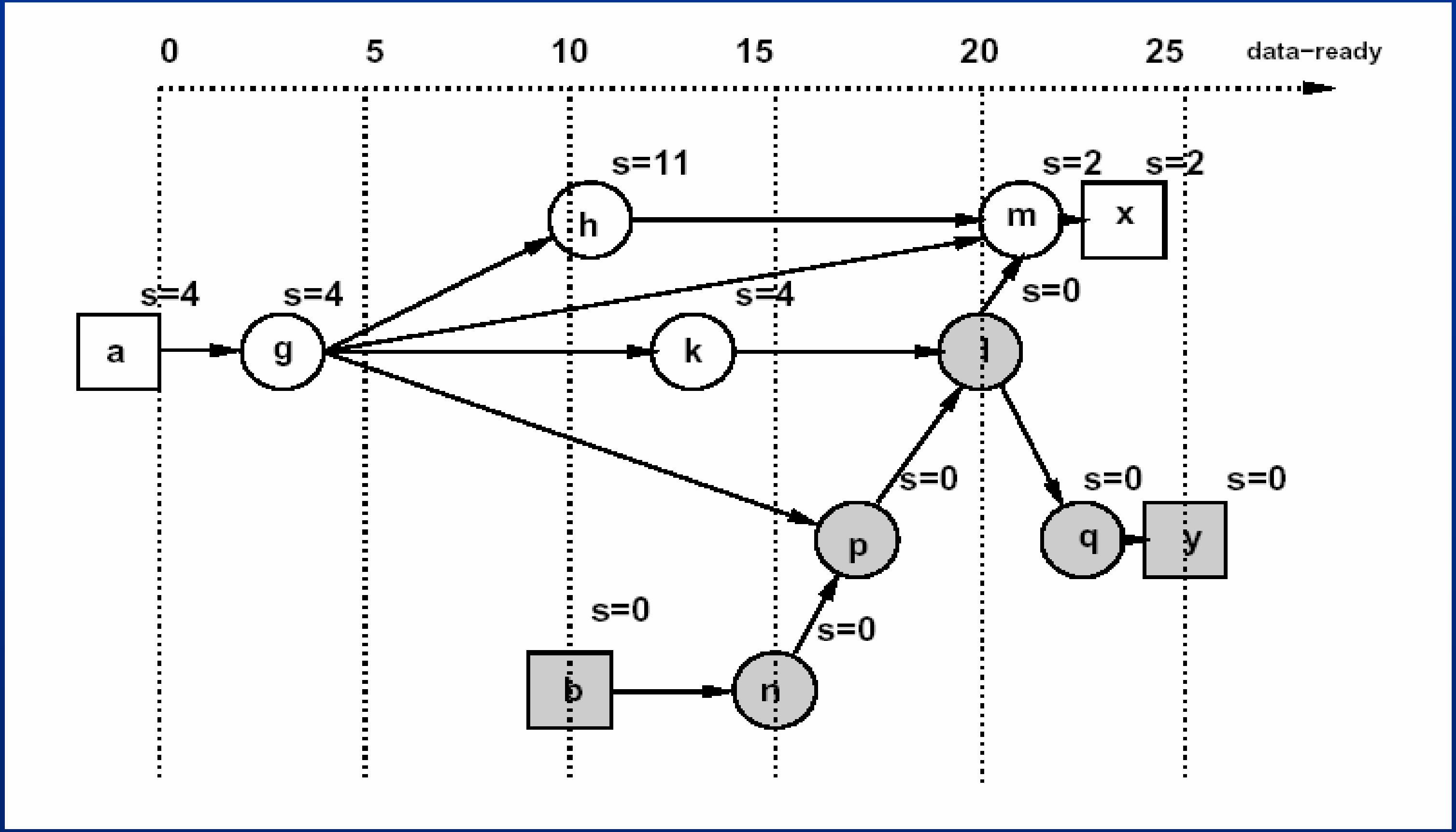
Required Times & Slack:

$s_x = 2$; $s_y=0$
$\bar{t}_m = 25-2=23$; $s_m=23-21=2$
$\bar{t}_q = 25-3=22$; $s_q=22-22=0$
$\bar{t}_l = \min\{23-1,22-2\}=20$; $s_l=0$
$\bar{t}_h = 23-1=22$; $s_h=22-11=11$
$\bar{t}_k = 20-3=17$; $s_k=17-13=4$
$\bar{t}_p = 20-3=17$; $s_p=17-17=0$
$\bar{t}_n = 17-2=15$; $s_n=15-15=0$
$\bar{t}_b = 15-5=10$; $s_b=10-10=0$
$\bar{t}_g = \min\{22-8;17-10;17-2\}=7$; $s_g=4$
$\bar{t}_a=7-3=4$; $s_a=4-0=4$

Topological Critical Path …

- **Assume topologic computation of**
  - Data-ready by forward traversal.
  - Required data-ready by backward traversal.
- **Topological critical path**
  - Input/output path with zero slacks.
  - Any increase in the vertex propagation delay affects the output data-ready time.
- **A topological critical path may be false.**
  - No event can propagate along that path.
  - False path does not affect performance

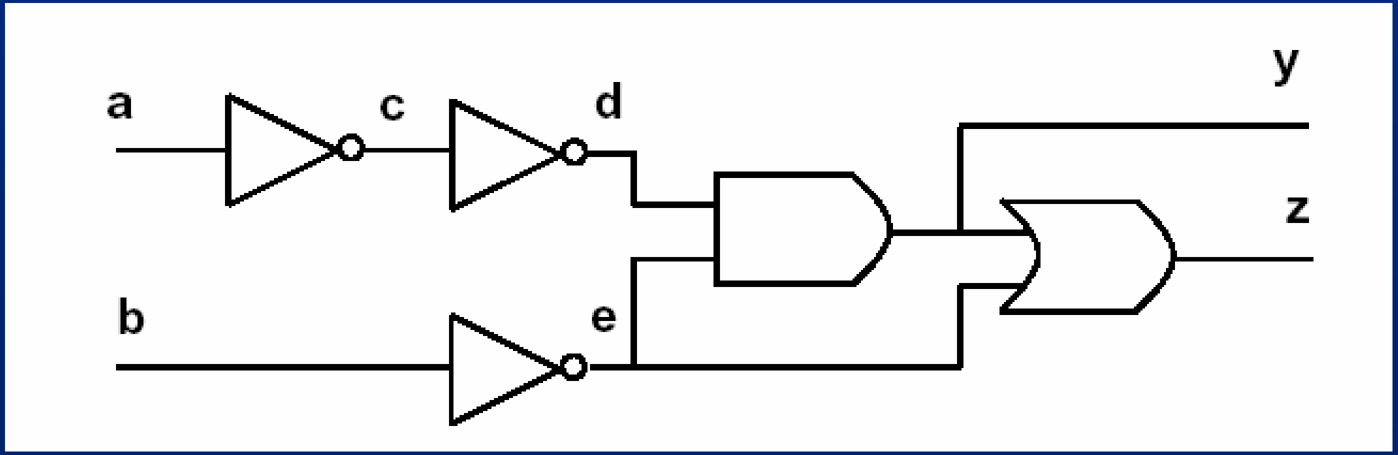
… Topological Critical Path



Topological critical path: $(v_b, v_n, v_p, v_l, v_q, v_y)$.

False Path Example

- **All gates have unit delay.**
- **All inputs ready at time 0.**
- **Longest topological path: (v_a, v_c, v_d, v_y, v_z).**
  - Path delay: 4 units.
  - False path: event cannot propagate through it
- **Critical true path: (v_a, v_c, v_d, v_y).**
  - Path delay: 3 units.



Algorithms for Delay Minimization …

- **Alternate**
  - Critical path computation.
  - Logic transformation on critical vertices.
- **Consider quasi critical paths**
  - Paths with near-critical delay.
  - Small slacks.
- **Small difference between critical paths and largest delay of a non-critical path leads to smaller gain in speeding up critical paths only.**

… Algorithms for Delay Minimization

- **Most critical delay optimization algorithms have the following framework:**

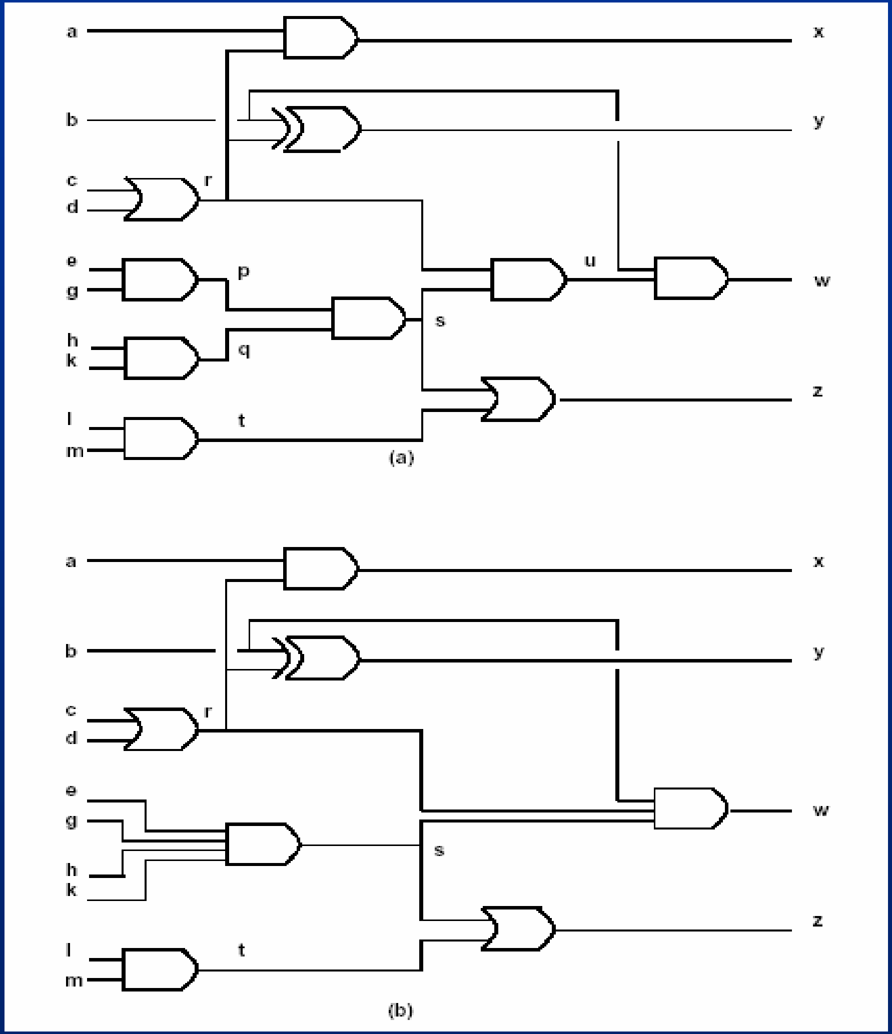
```
$REDUCE\_DELAY( G_n(V,E) ,\epsilon)\{$
    repeat {
        Compute critical paths and critical delay $\tau$;
        Set output required data-ready times to $\tau$;
        Compute slacks;
        $U$ = vertex subset with slack lower than $\epsilon$;
        $W$ = select vertices in $U$;
        Apply transformations to vertices $W$;
    }until (no transformation can reduce $\tau$ );
}
```

Transformations for Delay Reduction …

- **Reduce propagation delay.**
- **Reduce dependencies from critical inputs.**
- **Favorable transformation**
  - Reduces local data-ready time.
  - Any data-ready time increase at other vertices is bounded by the local slack.
- **Example**
  - Unit gate delay.
  - Transformation: Elimination.
    - Always favorable.
    - Obtain several area/delay trade-off points.

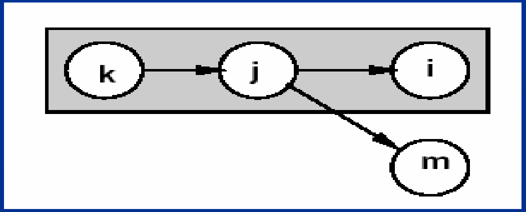
… Transformations for Delay Reduction

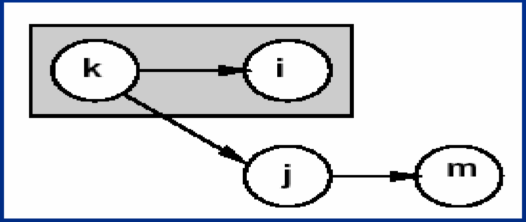
- W is a minimum-weight separation set from U.
- Iteration 1
  - Values of v_p, v_q, v_u = -1
  - Value of v_s=0.
  - Eliminate v_p, v_q. (No literal increase.)
- Iteration 2
  - Value of v_s=2, value of v_u=-1.
  - Eliminate v_u. (No literal increase.)
- Iteration 3
  - Eliminate v_r , v_s, v_t. (Literals increase.)

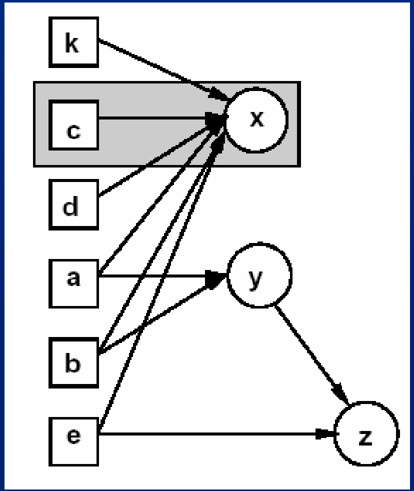


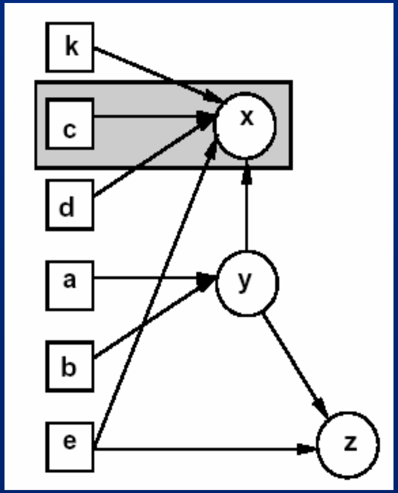
More Refined Delay Models

- **Propagation delay grows with the size of the expression and with fanout load.**
- **Elimination**
  - Reduces one stage.
  - Yields more complex and slower gates.
  - May slow other paths.
- **Substitution**
  - Adds one dependency.
  - Loads and slows a gate.
  - May slow other paths.
  - Useful if arrival time of critical input is larger than other inputs







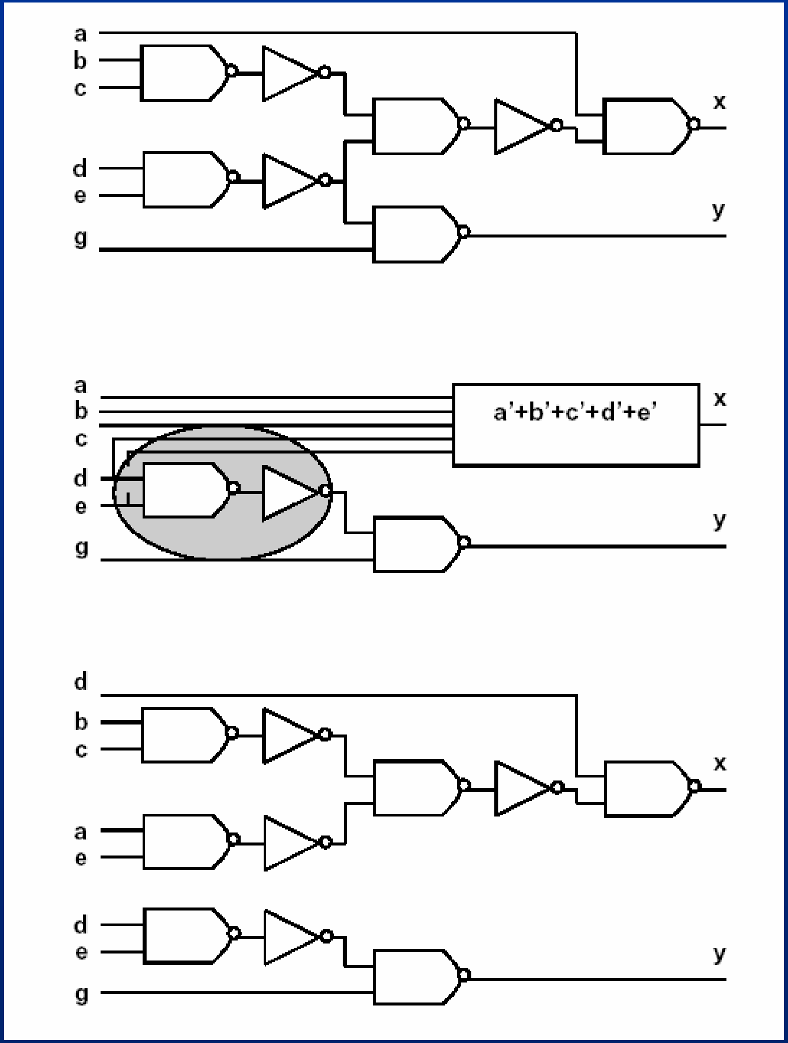


Speed-Up Algorithm …

- **Decompose network into two-input NAND gates and inverters.**
- **Determine a subnetwork W of depth d.**
- **Collapse subnetwork by elimination.**
- **Duplicate input vertices with successors outside W**
  - Record area penalty.
  - Resynthesize W by timing-driven decomposition.
- **Heuristics**
  - Choice of W.
  - Monitor area penalty and potential speed-up.

… Speed-Up Algorithm

- **Example**
  - NAND delay =2.
  - INVERTER delay =1.
  - All input data-ready=0 except t_d=3.
  - Critical Path: from V_d to V_x (11 delay units)
  - Assume V_x is selected and d=5.
  - New critical path: 8 delay units.


a
b
c
d
e
g
x
y
a
b
c
d
e
g
a'+b'+c'+d'+e'
x
y
d
b
c
a
e
d
e
g
x
y