
EE/CE 6301: Advanced Digital Logic

Bill Swartz

**Dept. of EE
Univ. of Texas at Dallas**

Session 07

Multi-Level Optimization

Adapted from the work of M Nourani to whom I am grateful

Motivation for Multi-Level Optimization

General Form

- General circuits may have
 - Sums within products
 - Products within sums
 - Arbitrary depth
- Example: $z = ((a \cdot (b + c) + e) \cdot f \cdot g + h) \cdot i$

Example – Two-Level vs. Multi-Level

- Consider the minimal 2-level SOP form

$$Z = adf + aef + bdf + bef + cdf + cef + g$$

- # AND3 = 6; # OR7 = 1 → Total = 7 gates
- # gate inputs = 25

- This can be improved if we replace the 2-level form with a multi-level, *factored form* ...

$$Z = (ad + ae + bd + be + cd + ce)f + g$$

$$Z = [(a + b + c)d + (a + b + c)e]f + g$$

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



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

$$x = a + b + c$$

$$y = d + e$$

- # AND3 = 1; # OR2 = 2; # OR3 = 1 → Total = 4 gates!
- # gate inputs = 10

Consideration of Cost

- Two-level implementation for most practical circuits will be too expensive.
 - Example: 16-bit adder (32 bits inputs, huge number 2^{16} to 2^{32} product terms)
- Two-level is natural for Programmable Logic Array (PLA). Inefficient, hence folded PLAs.
- Many of datapath logic are designed by integrating/interfaces modules. A multi-level architecture will be created.

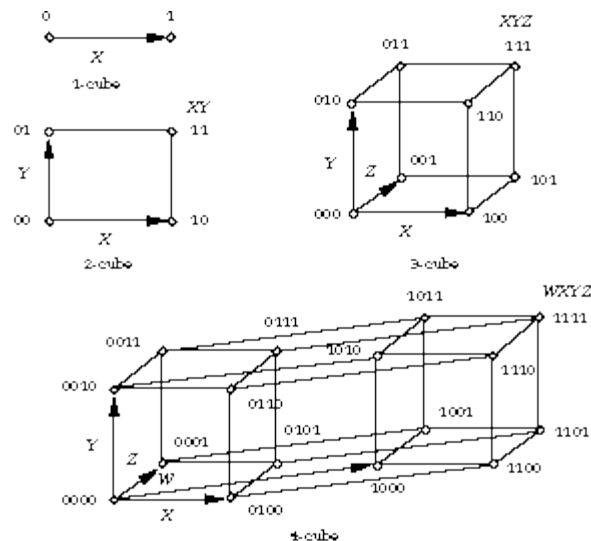
Multi-Level Optimization Methods

Fundamental Techniques

- Cubes
- Transformations
 - Factorization
 - Decomposition
 - Extraction
 - Substitution
 - Collapsing
- Shannon Expansion
- Division
 - Weak (Algebraic)
 - Strong (Boolean)

Boolean Cubes

- We can represent the truth table of an n-input Boolean function as a n-dimensional cube
- Each axis represent one variable or literal and can be either 0 or 1



Some example cubes

Boolean Cubes

- Cubes are denoted by their *literals* or the positive logic variables
- Examples
 - a
 - ab
 - xyz
 - wxyz

Transformations

Factorization – Example 1

- Multi-level logic can be less expensive
- Original form

$$z = abcf + abdf + abef + abcg + abdg + abeg$$

—7 gates: 6 AND4, 1 OR6

- After factorization:

$$z = ab(c + d + e)(f + g)$$

—3 gates: 1 OR2, 1 OR3, 1 AND4

Factorization – Example 2

- Multi-level logic can be less expensive
- Original form

$$z = ac + ad + bc + bd + e$$

—5 gates: 4 AND2, 1 OR5, 9 literals

- After factorization:

$$z = (a + b)(c + d) + e$$

—4 gates: 3 OR2, 1 AND2, 5 literals

Decomposition – Example 1

- Original form

$$f = abc + abd + \bar{a}\bar{c}\bar{d} + \bar{b}\bar{c}\bar{d}$$

—5 gates: 4 AND3, 1 OR4

- After decomposition:
$$\begin{cases} x = ab \\ y = c + d \\ f = xy + \bar{x}\bar{y} \end{cases}$$

—5 gates: 3 AND2, 2 OR2 not quite! Why?

Decomposition – Example 2

- Multi-level logic can be less expensive
- Original form

$$z = \bar{a}bcd + a\bar{b}cd + ab\bar{c}d + abcd\bar{d} + a\bar{b}\bar{c}\bar{d} + \bar{a}b\bar{c}\bar{d} + \bar{a}\bar{b}c\bar{d} + \bar{a}\bar{b}\bar{c}d$$

- After identifying the sub-expressions

$$\begin{cases} x_1 = a \oplus b = \bar{a}b + a\bar{b} \\ x_2 = x_1 \oplus c = \bar{x}_1c + x_1\bar{c} \\ z = x_2 \oplus d = \bar{x}_2d + x_2\bar{d} \end{cases}$$

Extraction – Example 1

- Having multi-level and multi-output optimization may be advantageous

- Original form

$$\begin{cases} f_1 = AB + AC + AD \\ f_2 = \bar{A}B + \bar{A}C + \bar{A}E \end{cases} \quad \text{6 And2 6 Or2 1inv}$$

—2 levels, 6 product terms, 24 transistors in static CMOS implementation (not counting inverters)

- After extraction:

$$\begin{cases} K = B + C \\ f_1 = AK + AD \\ f_2 = \bar{A}K + \bar{A}E \end{cases} \quad \text{4 And2 3 Or2 1inv}$$

—3 levels, 20 transistors in static CMOS implementation (not counting inverters)

Extraction – Example 2

- Extracting common sub-expression over multiple outputs may be advantageous

- Original form
$$\begin{cases} f = (a + b)cd + e \\ g = (a + b)\bar{e} \\ h = cde \end{cases}$$

—6 gates: 4 AND2/OR2, 2 AND3

- After extraction:
$$\begin{cases} x = a + b \\ y = cd \\ f = xy + e \\ g = x\bar{e} \\ h = ye \end{cases}$$

—6 gates: 6 AND2/OR2

Substitution – Example 1

- When appropriate, a new expression can be formed by substitution
- Original form
$$\begin{cases} f = a + bc \\ g = a + b \end{cases}$$
 - 3 gates: 1 AND2, 2 OR2
- After substituting g into f :
$$\begin{cases} f = a + bc = (a + b)(a + c) = g(a + c) \\ g = a + b \end{cases}$$
 - 3 gates: 1 AND2, 2 OR2
- In this example, it is the same but in general g can be another available signal already produced

Collapsing – Example 1

- Collapsing is the opposite of substitution

- Original form

$$\begin{cases} f = ga + \bar{g}b \\ g = c + d \end{cases}$$

—4 gates: 2 AND2, 2 OR2

- After collapsing g in f:

$$f = (c + d)a + \overline{(c + d)}b = ac + ad + b\bar{c}\bar{d}$$

—4 gates: 2 AND2, 1 AND3, 1 OR3

- In this example, it may not be less costly but a series of collapsing and refactoring sometimes produce a better result

Shannon Expansion

Shannon Expansion

- Partitioning of the variables in Boolean equations
- Literal and its complement

Cofactors in Shannon Formulas

- SOP form (with respect to x_i)

$$\begin{aligned} f(x_{n-1}, \dots, x_0) &= \overline{x_i} f(x_{n-1}, \dots, x_i = 0, \dots, x_0) + x_i f(x_{n-1}, \dots, x_i = 1, \dots, x_0) \\ &= \overline{x_i} f_{\overline{x_i}} + x_i f_{x_i} \end{aligned}$$

- POS form (with respect to x_i)

$$\begin{aligned} f(x_{n-1}, \dots, x_0) &= [\overline{x_i} + f(x_{n-1}, \dots, x_i = 1, \dots, x_0)][x_i + f(x_{n-1}, \dots, x_i = 0, \dots, x_0)] \\ &= [\overline{x_i} + f_{x_i}][x_i + f_{\overline{x_i}}] \end{aligned}$$

Shannon Expansion – Example 1

$$f(x, y, z) = xyz + x\bar{y}z + x\bar{y}\bar{z} + x\bar{y}z + x\bar{y}\bar{z}$$

- Rewrite in terms of the variable x and its complement

$$f(x, y, z) = x g_x + \bar{x} g_{\bar{x}}$$

- Use distributive property for the variable and its complement

$$f(x, y, z) = x(yz + \bar{y}z) + \bar{x}(\bar{y}z + yz + \bar{y}\bar{z})$$

Shannon Expansion – Missing variables

$$f(x, y, z) = yz + x\bar{y}z + \bar{x}\bar{y}z$$

- Rewrite in terms of the variable x and its complement using identity $(x+x') = 1$

$$f(x, y, z) = yz(x + \bar{x}) + x\bar{y}z + \bar{x}\bar{y}z$$

- Expand

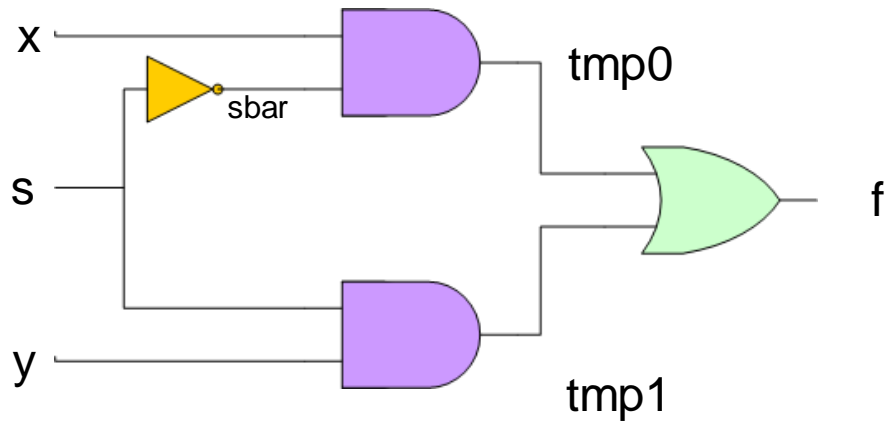
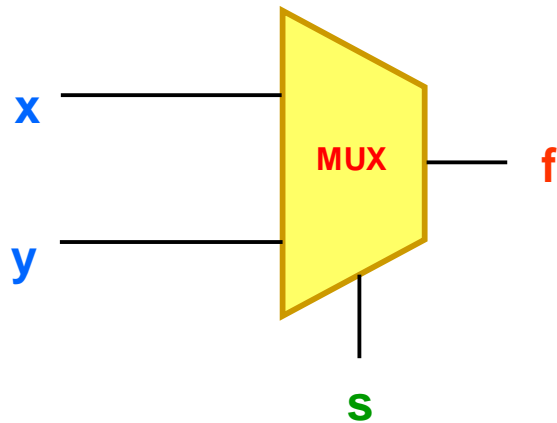
$$f(x, y, z) = \bar{x}yz + x\bar{y}z + x\bar{y}\bar{z} + \bar{x}\bar{y}z$$

- Rearrange using distributive property for the variable and its complement

$$f(x, y, z) = x(yz + \bar{y}z) + \bar{x}(yz + \bar{y}z)$$

Multiplexor-Based Implementation

- MUX-based implementation is naturally a multi-level logical structure
- Implementation of a 2x1 MUX
 - Function: $f = s' x + s y$



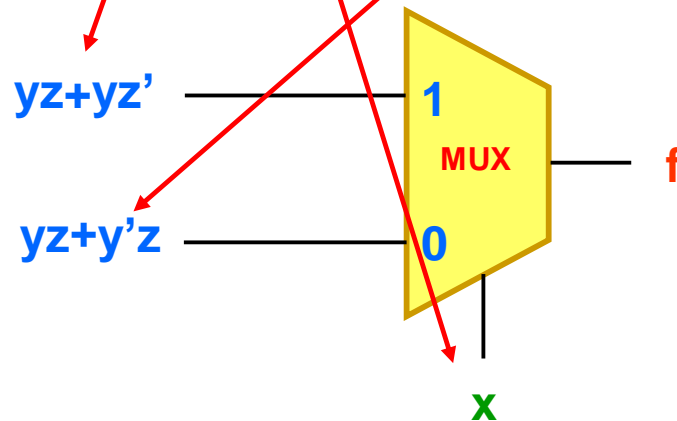
Multiplexor-Based Example

- Implementation of a 2x1 MUX

—Function: $f = s(y) + s'(x)$

$$f(x, y, z) = x(yz + y\bar{z}) + \bar{x}(yz + \bar{y}z)$$

- Let the selector $s = x$:



Example - MUX-Based Implementation

Equation

$$f(x, y, z) = \bar{x}yz + x\bar{y}z + xy\bar{z} + \bar{x}\bar{y}z$$

Minterms

$$f(x, y, z) = \sum (1, 3, 6, 7)$$

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

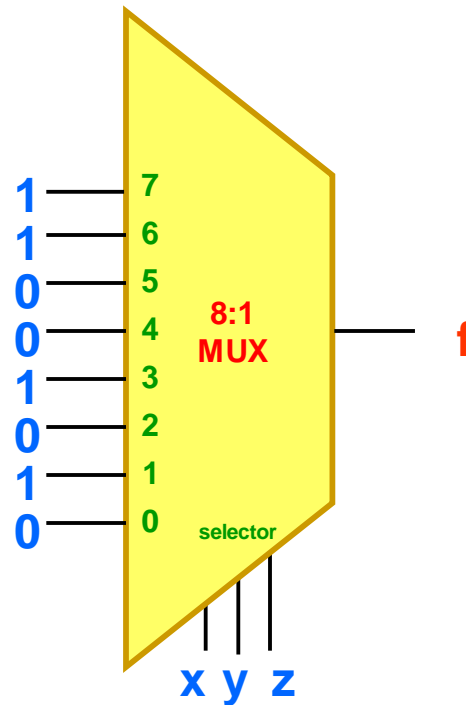
$$2^3 = 8$$

Example - MUX-Based Implementation

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1

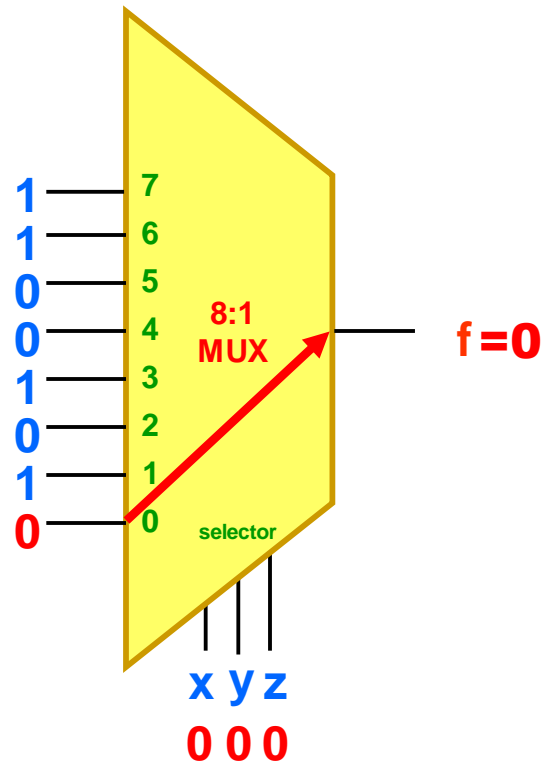


MUXes are Routers

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1

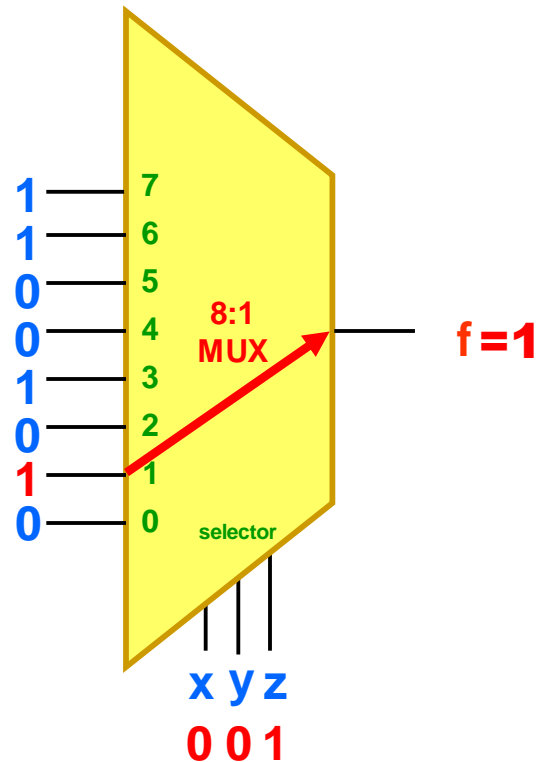


MUXes are Routers

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1

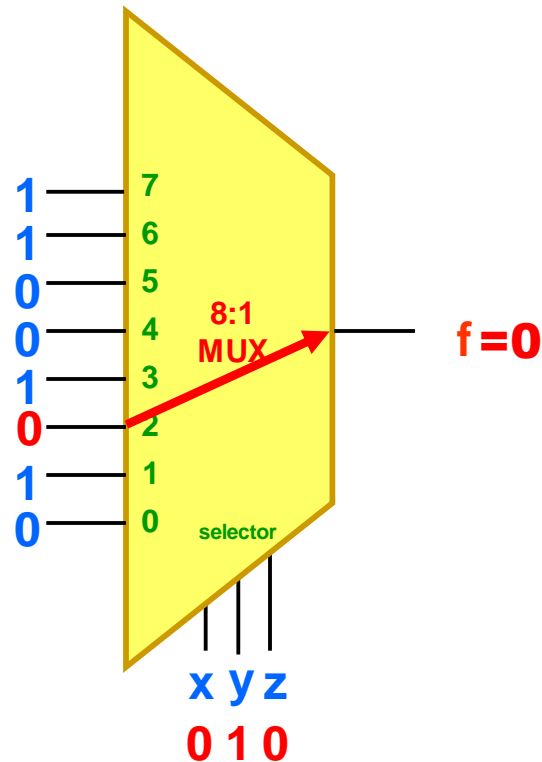


MUXes are Routers

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1

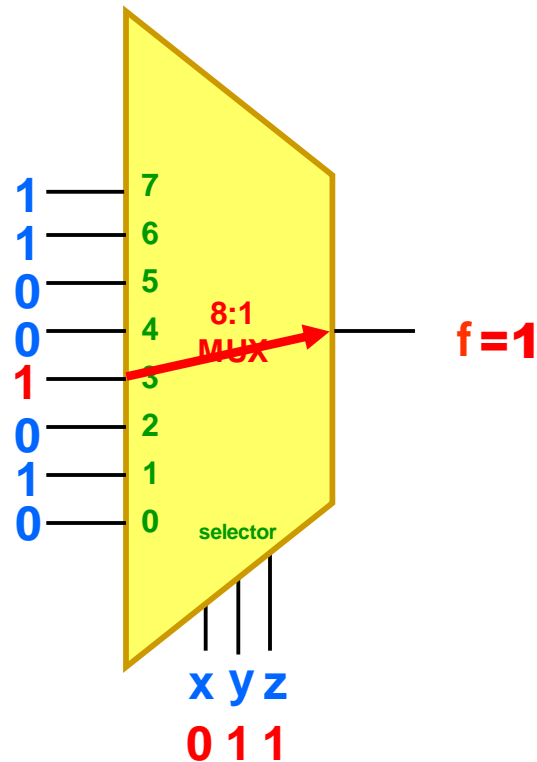


MUXes are Routers

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1

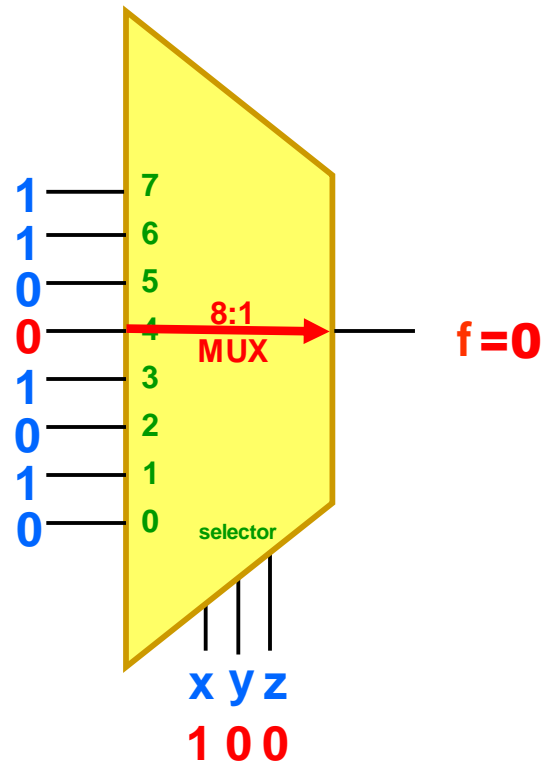


MUXes are Routers

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1

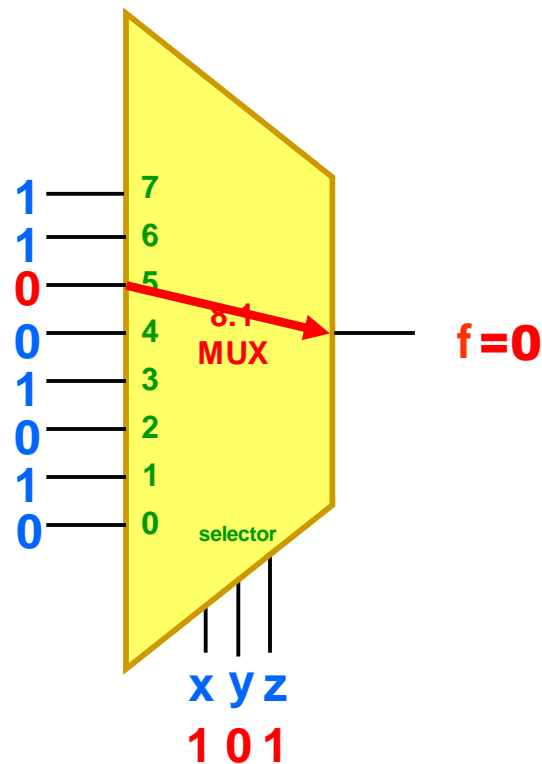


MUXes are Routers

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1

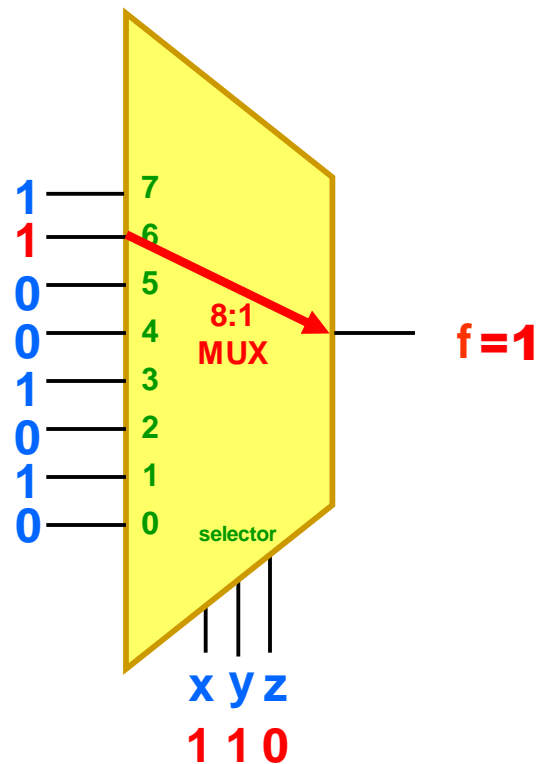


MUXes are Routers

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1

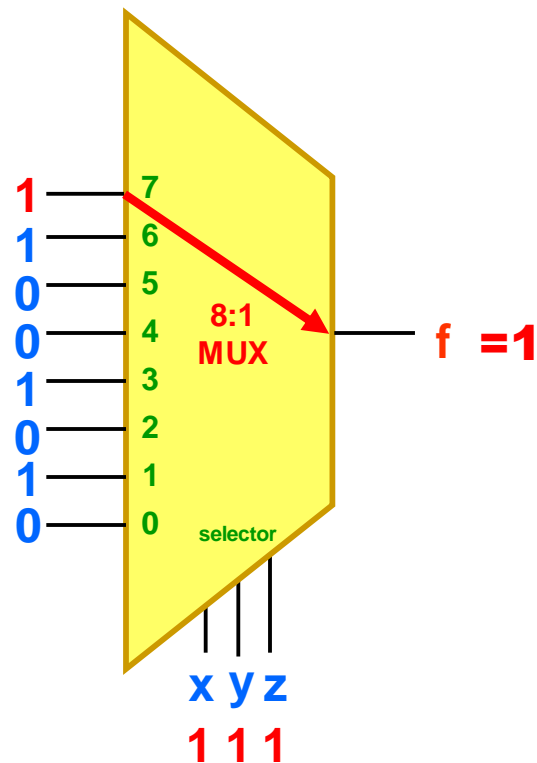


MUXes are Routers

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #1



Generalization of Shannon Expansions

- SOP form (with respect to $(x_{k-1}, \dots, x_1, x_0)$)

$$f(x_{n-1}, \dots, x_0) = \sum_{i=0}^{2^k-1} m_i f(x_{n-1}, \dots, x_k, \underline{m_i})$$

Minterm m_i



**Binary code
corresponding to m_i**



- POS form (with respect to $(x_{k-1}, \dots, x_1, x_0)$)

$$f(x_{n-1}, \dots, x_0) = \prod_{i=0}^{2^k-1} [M_i f(x_{n-1}, \dots, x_k, \underline{M_i})]$$

Maxterm M_i



**Binary code
corresponding to M_i**



Shannon Expansion – Example 8:1

- Expand (decompose) $f(x_{n-1}, \dots, x_0)$ with respect to (x_2, x_1, x_0)

$$\begin{aligned} f(x_{n-1}, \dots, x_0) &= \overline{x_2} \overline{x_1} \overline{x_0} f(x_{n-1}, \dots, x_3, 0, 0, 0) + \\ &\quad \overline{x_2} \overline{x_1} x_0 f(x_{n-1}, \dots, x_3, 0, 0, 1) + \\ &\quad \dots + \\ &\quad x_2 x_1 x_0 f(x_{n-1}, \dots, x_3, 1, 1, 1) \end{aligned}$$

Example – Using Shannon's Expansion

- Instead of a 8:1 Mux, expand to use a 4:1 Mux
- Expand using one variable
- 2 inputs as selectors to 4:1 Mux
- Three choices: MSB , middle bit, LSB
- One is easier on the eyes

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Attempt 1:

Let $\{x,y\}$ be the bit vector for the 2-bit selector

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

When selector $s = \{0,0\}$ or 0 decimal

$$f = z$$

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

When selector $s = \{0,1\}$ or 1 decimal

$$f = z$$

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

When selector $s = \{1,0\}$ or 2 decimal

$f = 0$ (does not depend on z)

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

When selector $s = \{1,1\}$ or 3 decimal

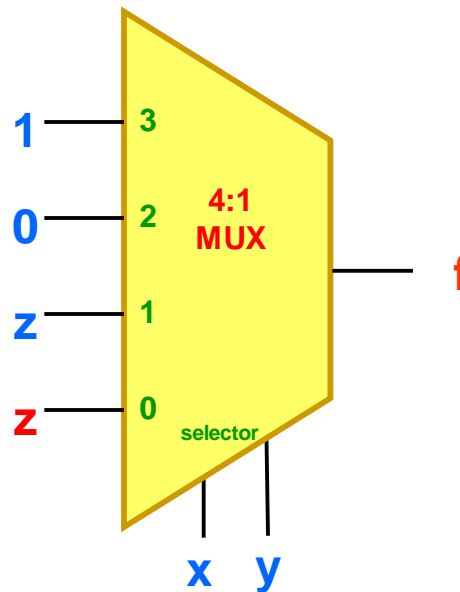
$f = 1$ (does not depend on z)

Resulting circuit using 4:1 MUX

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #2



4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Attempt 2:

Let $\{y,z\}$ be the bit vector for the 2-bit selector

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

When selector $s = \{0,0\}$ or 0 decimal

$f = 0$ (does not depend on x)

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

When selector $s = \{0,1\}$ or 1 decimal

$$f = x'$$

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

When selector $s = \{1,0\}$ or 2 decimal

$$f = x$$

4:1 Mux – Using Shannon's Expansion

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

When selector $s = \{1,1\}$ or 3 decimal

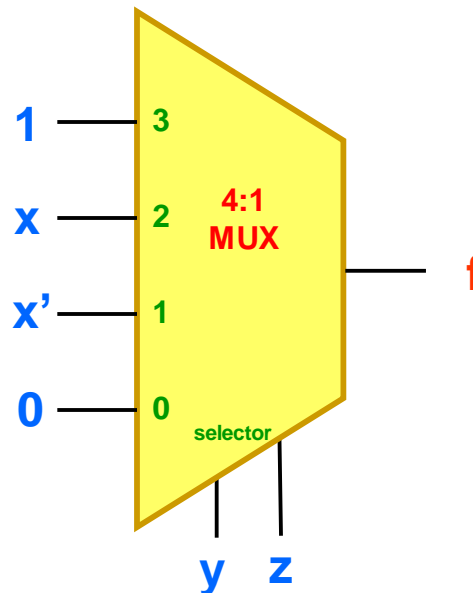
$f = 1$ (does not depend on x)

Resulting circuit using 4:1 MUX

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implementation #2b



Same complexity
but harder to align
visually in table

Easier to read table
from left to right

Example – Using Shannon's Expansion

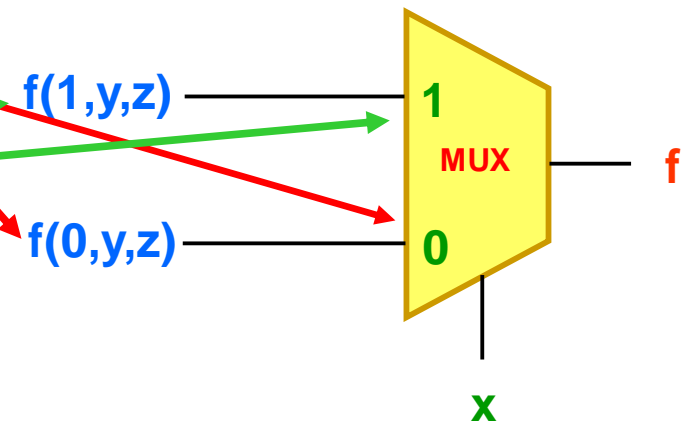
- Instead of a 4:1 Mux, expand to use 2:1 Muxes
- Expand one variable at a time
- 1 inputs as selectors to 2:1 Mux
- Expand most significant to least significant because of proximity in truth table due to the nature of binary numbers (Arithmetic is Big Endian format)

2:1 Mux – X First

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Two cases: $x = 0$ and $x = 1$

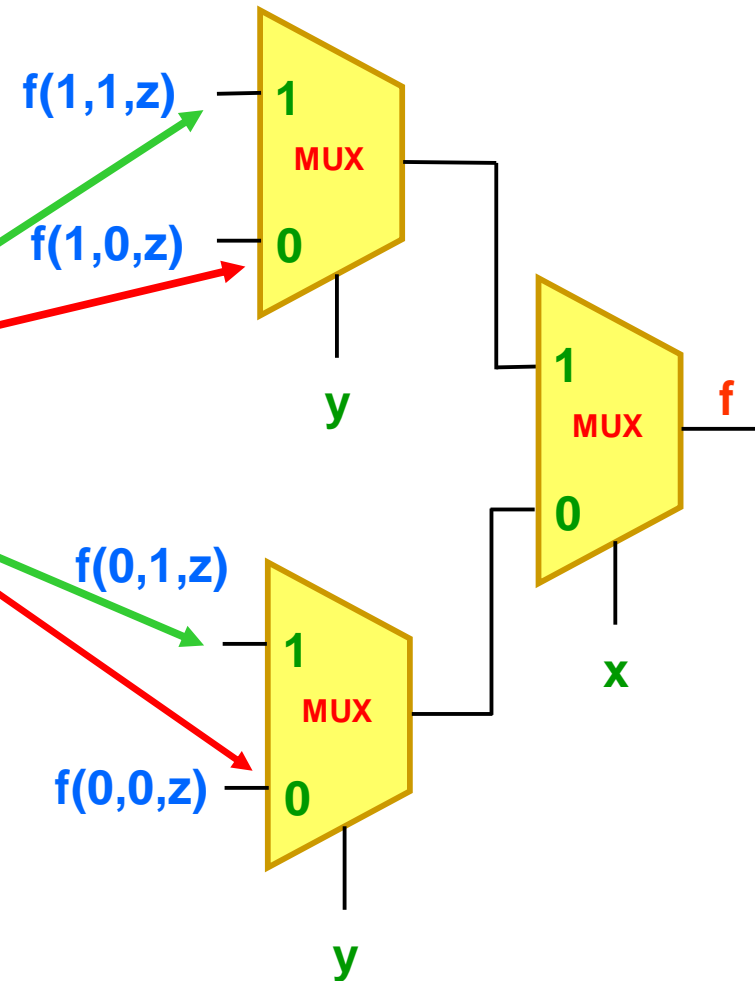


2:1 Mux – Y Next

Truth Table

Two cases: $y = 0$ and $y = 1$

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

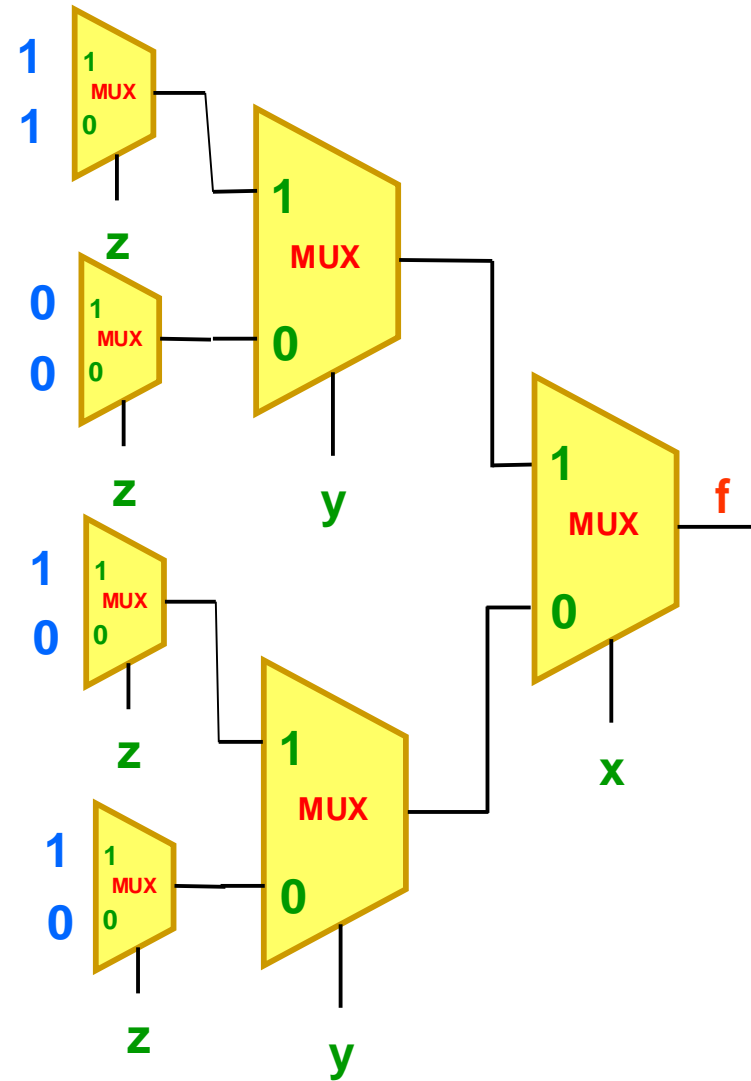


2:1 Mux – Z Expanded Too

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

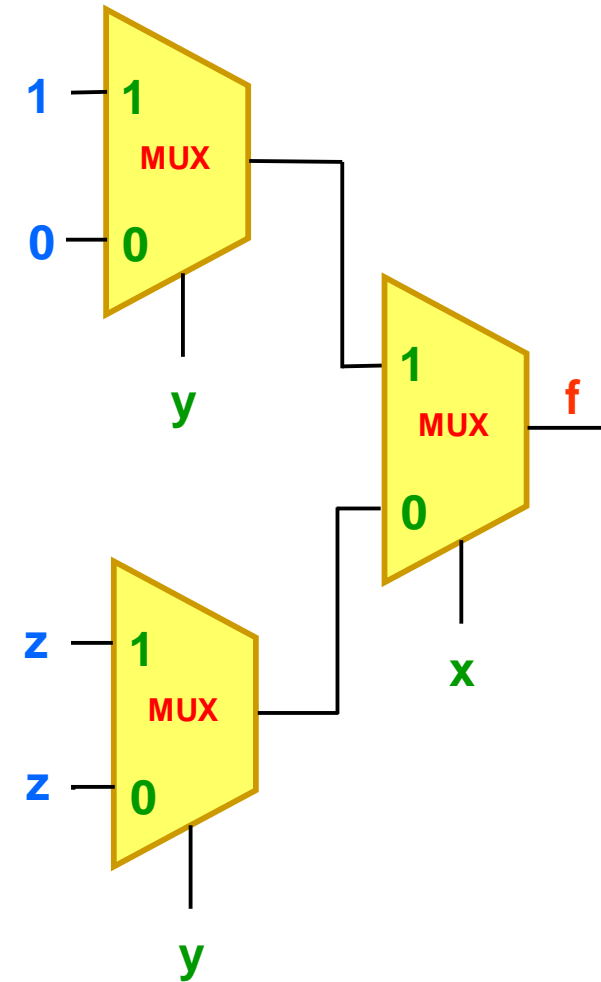
Decision Tree



2:1 Mux – Z Simplified

Truth Table

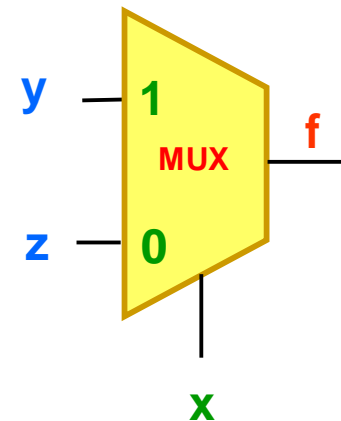
x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



2:1 Mux – Continued Simplification

Truth Table

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Division

Main Concept of Division

- Given a function (f), a divisor (p), find quotient (q) and remainder (r) such that: $f=pq+r$
- Example: $z=abc+abd+ef$, $p=ab$
 - $q=(c+d)$ and $r=ef$
 - Checking: $z=pq+r=(ab)(c+d)+ef=abc+abd+ef$
- Types of division
 - **Weak (algebraic)** division: some of the basic rules of algebra will be used
 - **Strong (Boolean)** division: only Boolean algebra will be used
- Weak and strong divisions may produce different results. Sometimes one may not exist.

Weak (Algebraic) Division

- **Definition:** Support of f
 - $\text{sup}(f) = \{\text{all variables } v \text{ that appear in } f \text{ as } v \text{ or } v'\}$
 - Example: $f = AB' + C \rightarrow \text{sup}(f) = \{A, B, C\}$
- **Definition:** f is orthogonal to g (shown as $f \perp g$) if $\text{sup}(f) \cap \text{sup}(g) = \emptyset$
 - Example: $f = A + B, g = C + D \rightarrow f$ and g are orthogonal since $\{A, B\} \cap \{C, D\} = \emptyset$
- **Definition:** g divides f weakly if there exist h and r such that: $f = gh + r$, where $h \neq 0$ and $g \perp h$
 - Example: $f = ab + ac + d, g = b + c \rightarrow f = a(b + c) + d \rightarrow$ quotient becomes $h = a$ and remainder is $r = d$
- **Definition:** g divides f evenly if $r = 0$

Computing f/g in Weak Division

- Suppose $f=\{c_i\}$ and $g=\{a_i\}$ denote the list of cubes (product terms) in function f and divisor g .
- Division relationship (finding quotient):

$$f / g = \bigcap_{i=1}^{|g|} h_i = h_1 \cap h_2 \cap \dots \cap h_{|g|}$$

Such that: $h_i = f / a_i = \{b_i \mid a_i b_i \in f\} \forall i$

h_i shows the product terms in f that have a_i without literal a_i

- The complexity of the computation is $O(|f|.|g|)$ where $|f|$ and $|g|$ indicate number of product terms in f and g , respectively.

Weak Division – Example

- Function and divisor:
$$\begin{cases} f = abc + abde + abh + bcd \\ g = c + de + h \end{cases}$$

- Calculating h_i values:
$$\begin{cases} f / c = \{ab, bd\} \\ f / de = \{ab\} \\ f / h = \{ab\} \end{cases}$$

- Computing quotient f/g :

$$f / g = f / c \cap f / de \cap f / h = \{ab, bd\} \cap \{ab\} \cap \{ab\} = \{ab\}$$

- Computing the remainder:

$$\begin{aligned} f &= g \cdot (f / g) + r \Rightarrow r = f - g \cdot (f / g) \\ &= abc + abde + abh + bcd - (ab)(c + de + h) \\ &= abc + abde + abh + bcd - abc - abde - abh \\ &= bcd \end{aligned}$$

- Final result:

$$f = g \cdot (f / g) + r = (c + de + h) \cdot ab + bcd$$

Weak Division

Weak division is a specific example of algebraic division.

DEFINITION:

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

- it is algebraic and
- R has as few cubes as possible.

The quotient H resulting from weak division is denoted by F/G .

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

Computing f/g in Weak Division

- Suppose $f=\{c_i\}$ and $g=\{a_i\}$ denote the list of cubes (product terms) in function f and divisor g .
- Division relationship (finding quotient):

$$f / g = \bigcap_{i=1}^{|g|} h_i = h_1 \cap h_2 \cap \dots \cap h_{|g|}$$

Such that: $h_i = f / a_i = \{b_i \mid a_i b_i \in f\} \forall i$

h_i shows the product terms in f that have a_i without literal a_i

- The complexity of the computation is $O(|f|.|g|)$ where $|f|$ and $|g|$ indicate number of product terms in f and g , respectively.

Weak Division – Example

- Given

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

$$G = ae + b$$

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

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

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

$$R = F \setminus GH$$

$$H = c + d = F/G \quad R = be + a'b + ab$$

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

Efficiency Issues

- We use filters to prevent trying a division

G is not an algebraic divisor of F if

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

Boolean Network: As a Digraph

$G=(V,E)$: DAG

- V : each function is a node (node $i \Leftrightarrow f_i \Leftrightarrow y_i$).
- E : there is a directed edge from node i to node j if $y_i \in \text{supp}(f_j)$, denoted by $(i,j) \in E$.

If $(i,j) \in E$, node i is a predecessor (input, fanin) of node j , and node j is a successor (output, fanout) of node i ;

If there is a path from node i to node j , node i is a transitive predecessor (transitive fanin) of node j , and node j is a transitive successor (transitive fanout) of node i .

$$P_i = \{j \in V \mid (j,i) \in E\} \quad S_i = \{j \in V \mid (i,j) \in E\}$$

$$P_i^* = \{j \in V \mid \text{node } j \text{ is a transitive fanin of node } i\}$$

$$S_i^* = \{j \in V \mid \text{node } j \text{ is a transitive fanout of node } i\}$$

Division - What do we divide with?

So far, we learned **how** to divide a given expression F by another expression G .

But how do we find **G** ?

- Too many Boolean divisors
- Restrict to algebraic divisors.

Problem: Given a set of functions $\{ F_i \}$, find **common** weak **(algebraic)** divisors.

Kernels and Kernel Intersections

DEFINITION 18: An expression is **cube-free** if no cube divides the expression evenly (i.e. there is no literal that is common to all the cubes).

(e.g., $ab + c$ is cube-free; $ab + ac$ and abc are not cube-free).

Note: a cube-free expression **must** have more than one cube.

DEFINITION 19: The **primary divisors** of an expression F are the set of expressions

$$D(F) = \{F/c \mid c \text{ is a cube}\}.$$

Kernels and Kernel Intersections

DEFINITIONS 20: The **kernels** of an expression F are the set of expressions

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

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

DEFINITION 21: A cube c used to obtain the kernel $K = F/c$ is called a **co-kernel** of K .

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

Example

Example:

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

kernels

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

co-kernels

df, ef
 af, bf, cf
 1

Fundamental Theorem

THEOREM 22: If two expressions F and G have the property that

$$\forall k_F \in K(F), \forall k_G \in K(G) \rightarrow |k_G \cap k_F| \leq 1$$

(k_G and k_F have at most one term in common),

then F and G have no common algebraic multiple divisors (i.e. with more than one cube).

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

The Level of a Kernel

It is nearly as effective to compute a certain subset of $K(F)$. This leads to the definition for the level of a kernel:

$$K^n(F) = \left\{ \begin{array}{l} (n = 0) \Rightarrow \{k \in K(F) \mid K(k) = \{k\}\} \\ (n > 0) \Rightarrow \{k \in K(F) \mid \forall_{k_1 \in K(k)}, (k_1 \neq k) \Rightarrow k_1 \in K^{n-1}(F)\} \end{array} \right\}$$

Notes:

1. $K^0(F) \subset K^1(F) \subset K^2(F) \subset \dots \subset K^n(F) \subset K(F)$.
2. level- n kernels = $K^n(F) \setminus K^{n-1}(F)$
3. $K^n(F)$ is the set of kernels of level k or less.
4. A level-0 kernel has no kernels except itself.
5. A level- n kernel has at least one level $n-1$ kernel but no kernels (except itself) of level n or greater.

Level of a Kernel Example

Example:

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

$$k_1 = a + b(c + d) \in K^1$$

$\notin K^0 \implies$ level-1

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

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

Kerneling Algorithm

$R \leftarrow \text{KERNEL}(j, G)$

$R \leftarrow \emptyset$

if (G is cube-free) $R \leftarrow \{G\}$

For $i = j + 1, \dots, n$ {

 if (l_i appears only in one term) continue

 else

 if ($\exists k \leq i, l_k \in \text{all cubes of } G/l_i$), continue

 else,

$R \leftarrow R \cup \text{KERNEL}(i, \text{cube_free}(G/l_i))$

 }

return R

Kerneling Algorithm

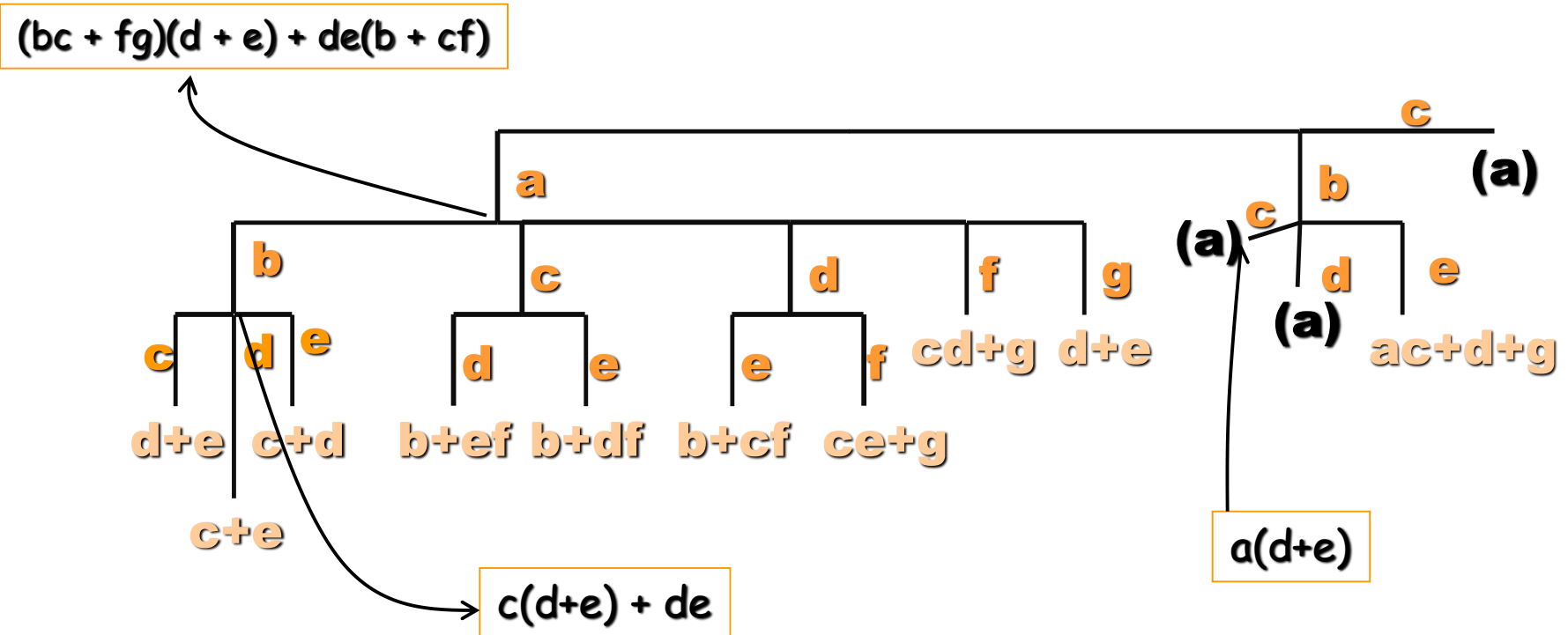
$\text{KERNEL}(0, F)$ returns all the kernels of F .

Notes:

1. The test $(\exists k \leq i, l_k \in \text{all cubes of } G/l_i)$ is a **major** efficiency factor. It also guarantees that no co-kernel is tried more than once.
2. This algorithm has stood up to all attempts to find faster ones.
3. Can be used to generate all co-kernels.

Kerneling Illustrated

$abcd + abce + adfg + aefg + adbe + acdef + beg$



Kerneling Illustrated

co-kernels

kernels

1

a

ab

abc

abd

abe

ac

acd

$a((bc + fg)(d + e) + de(b + cf))) + beg$

$(bc + fg)(d + e) + de(b + cf)$

$c(d+e) + de$

$d + e$

$c + e$

$c + d$

$b(d + e) + def$

$b + ef$

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

Applications - Factoring

FACTOR (F):

1. if (F has no factor) return F
/* e.g. if $|F| = 1$, or an OR of literals
or no literal appears more than once */
1. $D = \text{CHOOSE_DIVISOR}(F)$
2. $(Q, R) = \text{DIVIDE}(F, D)$
3. return
 $\text{FACTOR}(Q) \text{FACTOR}(D) + \text{FACTOR}(R)$

Problems with FACTOR

Notation in following examples:

F = the original function,

D = the divisor,

Q = the quotient,

P = the partial factored form,

O = the final factored form by FACTOR.

Restrict to algebraic operations only.

Example and Problems

Example 1:

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

$$D = c + d$$

$$Q = ab$$

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

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

O is **not** optimal since not maximally factored.

Can be further factored to

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

The problem **occurs** when

1. quotient Q is a **single** cube, and
2. some of the literals of Q also appear in the remainder R.

Solving this Problem

1. Check if the quotient Q is not a single cube, then done, else,
2. Pick a literal l_1 in Q which occurs most frequently in cubes of F .
3. Divide F by l_1 to obtain a new divisor D_1 .
Now, F has a new partial factored form
$$(l_1)(D_1) + (R_1)$$

and literal l_1 does not appear in R_1 .

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

Second Problem with FACTOR

Example 2:

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

$$D = a + b$$

$$Q = ce + de$$

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

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

Again, O is not maximally factored because $(c + d)$ is common to both products $e(c + d)(a + b)$ and remainder $(c + d)f$. The final factored form should have been

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

Second Problem with FACTOR

Solving the problem:

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

Improving Vanilla Factoring

```
GFACTOR(F, DIVISOR, DIVIDE)
  D = DIVISOR(F)
  if (D = 0) return F
  Q = DIVIDE(F,D)
  if (|Q| = 1) return LF(F, Q, DIVISOR, DIVIDE)
  else Q = make_cube_free(Q)
    (D, R) = DIVIDE(F,Q)
    if (cube_free(D)) {
      Q = GFACTOR(Q, DIVISOR, DIVIDE)
      D = GFACTOR(D, DIVISOR, DIVIDE)
      R = GFACTOR(R, DIVISOR, DIVIDE)
      return Q x D + R    }
  else {
    C = common_cube(D)
    return LF(F, C, DIVISOR, DIVIDE)
  }
```

Improving Vanilla Factoring

```
LF(F, C, DIVISOR, DIVIDE)
  L = best_literal(F, C)      /* most frequent */
  (Q, R) = DIVIDE(F, L)
  C = common_cube(Q)         /* largest one */
  Q = cube_free(Q)
  Q = GFACTOR(Q, DIVISOR, DIVIDE)
  R = GFACTOR(R, DIVISOR, DIVIDE)
  return  L C Q + R
```

Improving the Divisor

Various kinds of factoring can be obtained by choosing **different** forms of DIVISOR and DIVIDE.

CHOOSE_DIVISOR:

LITERAL - chooses most frequent literal

QUICK_DIVISOR - chooses the first level-0 kernel

BEST_DIVISOR - chooses the best kernel

DIVIDE:

Algebraic Division

Boolean Division

Factoring algorithms

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

LITERAL_FACTOR:

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

QUICK_FACTOR:

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

GOOD_FACTOR:

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

QUICK_FACTOR

QUICK_FACTOR uses

1. GFACTOR,
2. First level-0 kernel DIVISOR, and
3. WEAK_DIV.

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

$$D = c + d \text{ ---- level-0 kernel (hastily chosen)}$$

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

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

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

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

$$x = QD + R \qquad R = 0$$

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

Application - Decomposition

Recall: decomposition is the same as factoring **except:**

1. divisors are added as **new** nodes in the network.
2. the new nodes may **fan out** elsewhere in the network in both positive and **negative** phases

DECOMP(f_i)

$k = \text{CHOOSE_KERNEL}(f_i)$

if ($k == 0$) return

$f_{m+j} = k$ /* create new node $m + j$ */

$f_i = (f_i/k) y_{m+j} + (f_i/k') y'_{m+j} + r$ /* change node i */

DECOMP(f_i)

DECOMP(f_{m+j})

Similar to factoring, we can define

QUICK_DECOMP: pick a level 0 kernel and improve it.

GOOD_DECOMP: pick the best kernel.

decomp

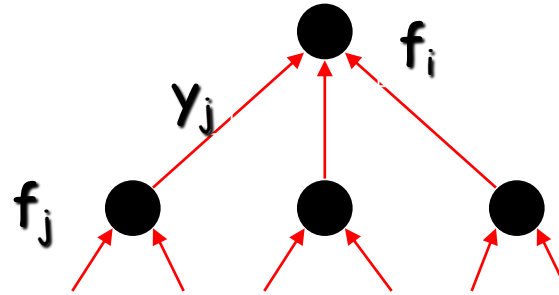
decomp[-gqd] [node-list]

decompose all nodes in the *node-list*.

If the node-list not specified, all nodes in network will be decomposed.

- q (default) is specified, the quick decomp algorithm is use which extracts out an arbitrary kernel successfully. Because of the fast algorithm for generating an arbitrary kernel, decomp -q is very fast compared with decomp -g. In most cases, the results ure very close. This command is recommended at the early phase of the optimization.
- g the good decomp algorithm is used which successively extracts out the best kernel until the function is factor free, and apply the same algorithm to all the kernels just extracted. This operation will give the best algebraic decomposition for the nodes. But, since it generates all the kernels at each step, it takes more CPU time. In general, decomp -q should be used in the early stage of the optimization. Only at the end of the optimization, should decomp -g be used.
- d disjoint decomposition is performed. It partitions the cubes into sets of cubes having disjoint variable support, creates one node for each partition, and a root node, the OR of the partitions.

Re-substitution (resub)



Idea: An **existing** node in a network may be a useful divisor in another node. If so, no loss in using it (**unless delay is a factor**).

Algebraic substitution consists of the process of algebraically dividing the function f_i at node i in the network by the function f_j (**or by f'_j**) at node j . During substitution, if f_j is an algebraic divisor of f_i , then f_i is transformed into

$$f_i = qy_j + r \quad (\text{or } f_i = q_1y_j + q_0y'_j + r)$$

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

Boolean Re-substitution Example

Substituting x into F .

$$x = ab + cd + e$$

$$F = abf + a'cd + cdf + a'de + ef$$

Incompletely specified function $F = (F_1, D, R_1)$

$$D = x'(cad + cd + e) + x(ab + cd + e)$$

$$F_1 = xef + xa'cd + xa'de + xabf + xcdf$$

$$R_1 = abf'x + aef'x + d'ef'x + a'c'e'x' + b'c'e'x' + a'd'e'x' + b'd'e'x' + acdf'$$

Sufficient variables a, d, f, x (minvar).

$$F_3 = xf + xa'd + xa'd + xaf + xdf$$

$$= xf + xa'd$$

$$= x(f + a'd)$$

$$F = (ab + cd + e)(f + a'd)$$

resub

```
resub [-ab] [node-list]
```

Resubstitute each node in the node-list into all the nodes in the network. The resubstitution will try to use both the positive and negative phase of the node. If node-list is not specified, the resubstitution will be done for **every** node in the network and this operation will keep looping until no more changes of the network can be made. Note the difference between resub * and resub. The former will apply the resubstitution to each node only once.

- a (**default**) option uses **algebraic** division when substituting one node into another. The division is performed on both the divisor and it's complement.
- b uses **Boolean** division when substituting one node into another.

Extraction-I

Recall: Extraction operation identifies **common** sub-expressions and manipulates the Boolean network.

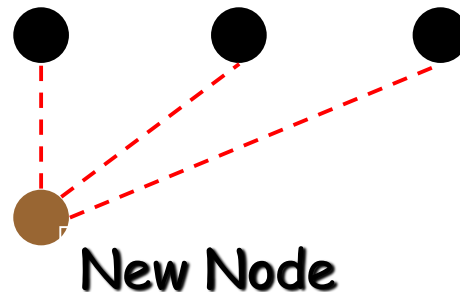
We can combine **decomposition** and **substitution** to provide an effective extraction algorithm.

```
EXTRACT( $\eta$ )
  For each node  $n$  {
     $\eta = \text{DECOMP}(n, \eta)$ 
  }
  For each node  $n$  {
    RESUB( $n, \eta$ )
  }
  Eliminate nodes with small value
```

Extraction-II

Kernel Extraction:

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



Example-Extraction

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

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

only level-0 kernels used in this example

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

$$l = d + e$$

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

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

Extraction:

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

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

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

$$m = cl + f$$

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

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

Note: no kernel intersections at this point.

cube extraction:

$$n = am$$

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

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

gkx

gkx [-1abcdfo] [-t threshold]

Extract multiple-cube common divisors from the network.

- a generates **all** kernels of all function in the network.
- b chooses the **best** kernel intersection as the new factor at each step of the algorithm; this is done by enumerating and considering each possible kernel intersection, and choosing the best.
- c uses the new factor and its **complement** when attempting to introduce the new factor into the network.
- d enables **debugging** information which traces the execution of the kernel extract algorithm.
- f uses the number of literals in the **factored** form for the network as the cost function when determining the value of a kernel intersection.
- o allows for **overlapping** factors.
- t sets a **threshold** such that divisors are extracted only while their value exceeds the threshold.
- 1 performs only a **single** pass over the network

script.algebraic

```
sweep  
eliminate 5  
simplify -m nocomp -d  
resub -a
```

```
gkx -abt 30  
resub -a; sweep  
gcx -bt 30  
resub -a; sweep
```

```
gkx -abt 10  
resub -a; sweep  
gcx -bt 10  
resub -a; sweep
```

```
gkx -ab  
resub -a; sweep  
gcx -b  
resub -a; sweep
```

```
eliminate 0  
decomp -g *
```

Faster “Kernel” Extraction

Non-robustness of kernel extraction

- Recomputation of kernels after every substitution: expensive
- Some functions may have many kernels (e.g. symmetric functions).

Two-cube “kernel” extraction [Rajski et al '90]

- Objects:
 - 2-cube divisors
 - 2-literal cube divisors
- **Example:** $f = abd + a'b'd + a'cd$
 - $ab + a'b'$, $b' + c$ and $ab + a'c$ are 2-cube divisors.
 - $a'd$ is a 2-literal cube divisor.

Fast Divisor Extraction

Features

$O(n^2)$ number of 2-cube divisors in an n -cube Boolean expression.

Concurrent extraction of 2-cube divisors and 2-literal cube divisors.

Some complement divisors recognized in each step during the synthesis, thus no algebraic resubstitution needed.

Example: $f = abd + a'b'd + a'cd$.

$k = ab + a'b'$, $k' = ab' + a'b$ (both 2-cube divisors)

$j = ab + a'c$, $j' = a'b' + ac'$ (both 2-cube divisors)

$c = ab$ (2-literal cube), $c' = a' + b'$ (2-cube divisor)

Generating all 2-cube divisors

$$F = \{c_i\}$$

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

This just takes all pairs of cubes in F and makes them cube-free.

c_i, c_j are any pair of cubes of cubes in F

Divisor generation is $O(n^2)$,

where n = number of cubes in F

Example: $F = axe + ag + bcxe + bcg$

$$\text{make_cube_free}(c_i + c_j) =$$

$$\{xe + g, a + bc, axe + bcg, ag + bcxe\}$$

Notes: (1) the function F is made an algebraic expression before generating double-cube divisors.

(2) not all 2-cube divisors are kernels.

Key result for 2-cube divisors

THEOREM 23: Expressions F and G have a common multiple-cube divisors if and only if $D(F) \cap D(G) \neq 0$.

Proof.

If part: If $D(F) \cap D(G) \neq 0$ then $\exists d \in D(F) \cap D(G)$ which is a double-cube divisor of F and G . d is a multiple-cube divisor of F and of G .

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

Key result for 2-cube divisors

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

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

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

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

Fast Kernel Extraction

Algorithm:

Generate and store all 2-cube kernels (2-literal cubes) and recognize complement divisors.

Find the **best** 2-cube kernel or 2-literal cube divisor at each stage and extract it.

Update 2-cube divisor set after extraction

Iterate extraction of divisors until no more improvement

Results:

Much **faster**.

Quality as good as that of kernel extraction.

On a set of examples:

- fast extract gives 9563 literals, general kernel extraction 9732,
- fast extract was 20 times faster.

fast_extract

```
fx [-o] [-b limit] [-l] [-z]
```

Greedy concurrent algorithm for finding the best double cube divisors and single cube divisors. Finds all the double cube and single cube divisors of the nodes in the network. It associates a value to each node, and extracts the node with the best value greedily.

- o only looks for **0-level** two-cube divisors.
- b reads an upper **bound** for the number of divisors generated.
- l changes the **level** of each node in the network as allowed by the slack between the required time and arrival time at that node.
- z uses **zero**-value divisors (in addition to divisors with a larger weight). This means that divisors that contribute zero gain to the overall decomposition are extracted. This may result in an overall better decomposition but takes an exorbitant amount of time.

script.rugged

```
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
```

Boolean Division

- Algebraic division:
- Example: $f = abd + cd + abe + ace$, assumed $g = ab + c$
 - $= d(ab + c) + abe + ace$
 - $h = d$ here
 - More optimal: $f = (ab + c)(ae + d)$
 - Why? No remainder!
- Algebraic division: $f = h.g + r$
- h and g are orthogonal
- Boolean division
 - $g = ab + c$ and $h = ae + d$
 - g and h are not orthogonal!

Boolean Division

Boolean division: $f = h.g + r$

- **h** and **g** can share:
- A common literal **x**
 - $a + bc = (a + b).(a + c)$
- Literal **x** and **x'**
 - $ab + a'c + bc = (a + b)(a' + c)$

Strong (Boolean) Division

- In strong division, no algebraic type of calculation (e.g. set computation, subtraction etc.) will be used.
- We use only Boolean relationships and methods
- Use logic optimization to identify portion of function f that can be constructed using divisor g

Strong (Boolean) Division

- What if we add the don't care set to improve ability to optimize?

$$\tilde{f} = f + f_{DC}$$

- Our new function may be simpler

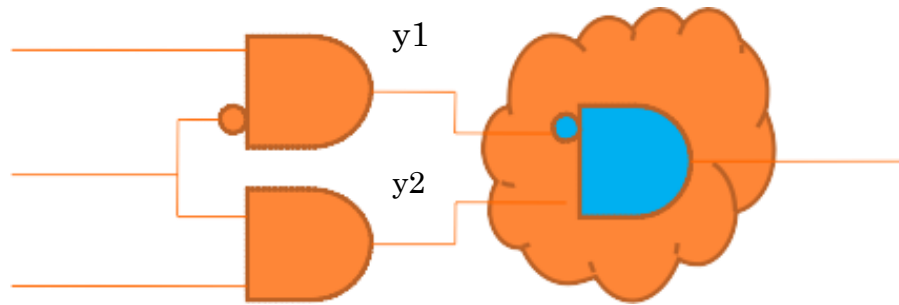
Don't Care based optimization

Two types of Don't Care conditions:

- External Don't Cares: defined by user, example:
the DC-set
- Internal Don't Cares: exist because of the structure of the boolean network. Two types of internal don't cares:
 - Satisfiability don't care
 - Observability don't care

SATISFIABILITY DON'T CARES

If input cannot occur, don't care the output



a b c

y1	y2	f
0	0	0
0	1	1
1	0	0
1	1	0



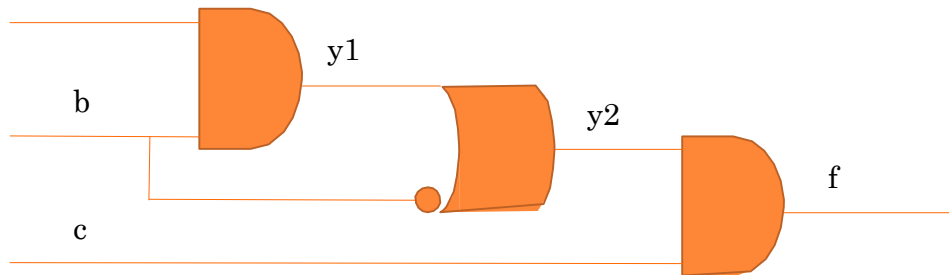
Never occurs

y1	y2	f
0	0	0
0	1	1
0	0	0
1	1	x



$$f = y2$$

OBSERVABILITY DON'T CARES



- o Says: The output F can observe the input X (X can be observed at output F) if changes of X make F changed
- o Example: if $c = 1$ then $y2$ can be observed at F
if $c = 0$ then $y2$ cannot be observed at F

Boolean Division

- Provided f, g ; reexpress $f = h.g + r$ (minimized)
- Label g as an input G (Add another input to f)
- Construct a function $F = \begin{cases} \text{Don't care, } G \neq g \\ f, G = g \end{cases}$
-
- DC set of $F = G \wedge g' = G \cdot g' + G' \cdot g$
- ON set of $F = f \cdot (G \cdot g + G' \cdot g') = f \cdot (G \cdot g' + G' \cdot g)'$
- Apply some known optimization algorithm on F
- Note: $F === f$, because $G = g$

Computing f/g in Strong Division

- Given a function f , to be strongly divided by g :
- Add an extra input to f corresponding to g , namely G

This is really 0 as G and g are the same

- Obtain function f^{\sim}

1. Define the don't care set function:

$$f_{DC} = g \oplus G = g\overline{G} + \overline{g}G$$

2. Find

$$\tilde{f} = f + f_{DC}$$

- Minimize \tilde{f} using two-level optimization
 - The product terms that have g in them will form the quotient
 - The remaining terms form remainder

Strong Division – Example

- Function and divisor:
$$\begin{cases} f = a + bcd \\ g = a + cd \end{cases}$$
- Calculate f_{DC} and f^{\sim} :
$$\begin{cases} f_{DC} = g \oplus (a + cd) = \bar{g}a + \bar{g}cd + g\bar{a}\bar{c} + g\bar{a}\bar{d} \\ \tilde{f} = f + f_{DC} \end{cases}$$
- Computing quotient f/g and remainder is done by doing the 2-level optimization: copy over K-map for two cases

$ab \backslash cd$	00	01	11	10
00				
01			1	
11	1	1	1	1
10	1	1	1	1

K-map for $f(a,b,c,d)$

$ab \backslash cd$	00	01	11	10
00				
01			1	
11	1	1	1	1
10	1	1	1	1

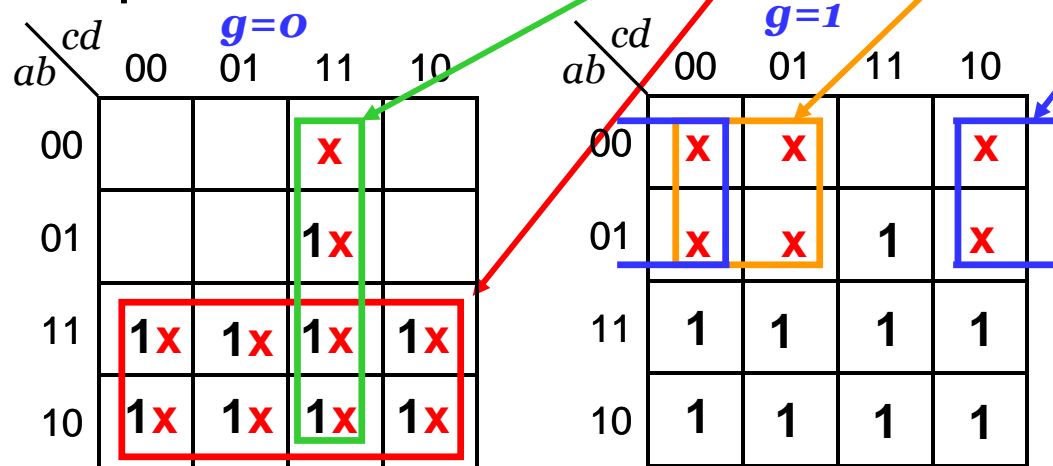
K-map for $f(a,b,c,d,g)$

$ab \backslash cd$	00	01	11	10
00				
01			1	
11	1	1	1	1
10	1	1	1	1

Strong Division – Example (cont.)

- Calculate f_{DC} and f^{\sim} :

$$\begin{cases} f_{DC} = g \oplus (a + cd) = \bar{g}a + \bar{g}cd + g\bar{a}\bar{c} + g\bar{a}\bar{d} \\ \tilde{f} = f + f_{DC} \end{cases}$$
- Computing quotient f/g and remainder is done by doing the 2-level optimization:



- When 1 and x overlap, we treat them as x.

Strong Division – Example (cont.)

- Calculate f_{DC} and f^{\sim} :

$$\begin{cases} f_{DC} = g \oplus (a + cd) = \bar{g}a + \bar{g}cd + g\bar{a}\bar{c} + g\bar{a}\bar{d} \\ \tilde{f} = f + f_{DC} \end{cases}$$
- Computing quotient f/g and remainder is done by doing the 2-level optimization:

$g=0$

$ab \backslash cd$	00	01	11	10
00			x	
01			x	
11	x	x	x	x
10	x	x	x	x

$g=1$

$ab \backslash cd$	00	01	11	10
00	x	x		x
01	x	x	1	x
11	1	1	1	1
10	1	1	1	1

- Cover the 1s
- Final Result:

quotient Remainder

↓ ↓

$$\tilde{f} = f + f_{DC} = f = b \cdot g + a(g + g') = b \cdot (a + cd) + a$$

XOR/XNOR Implementation

Laws of XOR/XNOR Algebra

- Laws for XOR and XNOR are dual of each other
- XNOR also known as EQV function

XOR
Truth Table

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

Good for test

XNOR
EQV.
Truth Table

X	Y	$X \odot Y$
0	0	1
0	1	0
1	0	0
1	1	1

XOR Laws

$$X \oplus 0 = X$$

$$X \oplus 1 = \bar{X}$$

$$X \oplus X = 0$$

$$X \oplus \bar{X} = 1$$

By
Duality

EQV Laws

$$X \odot 1 = X$$

$$X \odot 0 = \bar{X}$$

$$X \odot X = 1$$

$$X \odot \bar{X} = 0$$

Laws of XOR/XNOR Algebra (cont.)

- Other properties:

Associative Law $\left\{ \begin{array}{l} (X \odot Y) \odot Z = X \odot (Y \odot Z) = X \odot Y \odot Z \\ (X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z \end{array} \right\}$

Commutative Laws $\left\{ \begin{array}{l} X \odot Y \odot Z = X \odot Z \odot Y = Z \odot X \odot Y = \dots \\ X \oplus Y \oplus Z = X \oplus Z \oplus Y = Z \oplus X \oplus Y = \dots \end{array} \right\}$

Distributive Laws $\left\{ \begin{array}{ll} (X \cdot Y) \oplus (X \cdot Z) = X \cdot (Y \oplus Z) & \text{Factoring Law} \\ (X + Y) \odot (X + Z) = X + (Y \odot Z) & \text{Distributive Law} \end{array} \right\}$

Absorptive Laws $\left\{ \begin{array}{l} X \cdot (\bar{X} \oplus Y) = X \cdot Y \\ X + (\bar{X} \odot Y) = X + Y \end{array} \right\}$

remember

$$xy + \bar{x}z + yz = xy + \bar{x}z$$

Consensus Laws $\left\{ \begin{array}{l} (X \cdot Y) \oplus (\bar{X} \cdot Z) + (Y \cdot Z) = (X \cdot Y) \oplus (\bar{X} \cdot Z) \\ (X + Y) \odot (\bar{X} + Z) \cdot (Y + Z) = (X + Y) \odot (\bar{X} + Z) \end{array} \right\}$

DeMorgan's Laws $\left\{ \begin{array}{l} \overline{X \odot Y} = \bar{X} \oplus \bar{Y} = X \oplus Y \\ \overline{X \oplus Y} = \bar{X} \odot \bar{Y} = X \odot Y \end{array} \right\}$

Note carefully

Examples of XOR/XNOR Properties

- Ex. 1: Generalization of DeMorgan's XOR law

$$\overline{X \odot Y \odot Z \odot \dots \odot N} = \bar{X} \oplus \bar{Y} \oplus \bar{Z} \oplus \dots \oplus \bar{N}$$

$$\overline{X \oplus Y \oplus Z \oplus \dots \oplus N} = \bar{X} \odot \bar{Y} \odot \bar{Z} \odot \dots \odot \bar{N}.$$

- Ex. 2: XOR Distributive Law

X	Y	Z	$X + Y$	$X + Z$	$Y \odot Z$	$(X + Y) \odot (X + Z)$	$X + (Y \odot Z)$
0	0	0	0	0	1	1	1
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1
1	0	1	1	1	0	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

Examples of XOR/XNOR Properties (cont.)

- Ex. 3:

$$\begin{aligned} [(XY) \oplus (XZ)] &= [(\overline{XY})](XZ) + (XY)[(\overline{XZ})] \\ &= [(\bar{X} + \bar{Y})(XZ)] + [(XY)(\bar{X} + \bar{Z})] \\ &= [\bar{X}XZ + X\bar{Y}Z] + [X\bar{X}Y + XY\bar{Z}] \\ &= [X\bar{Y}Z + XY\bar{Z}] \\ &= X[\bar{Y}Z + Y\bar{Z}] \\ &= X(Y \oplus Z). \end{aligned}$$

Examples of XOR/XNOR Properties (cont.)

- Ex. 4:

$$\begin{aligned}X + [(\bar{X} \odot Y)] &= X + (X\bar{Y}) + (\bar{X}Y) \\&= [X + (X\bar{Y})] + \bar{X}Y \\&= [X(1 + \bar{Y}) + \bar{X}Y] \\&= [X + \bar{X}Y] \\&= X + Y.\end{aligned}$$

- Ex. 5:

$$\begin{aligned}X \cdot [(\bar{X} \oplus Y)] &= X \cdot (\bar{X}\bar{Y} + XY) \\&= [X \cdot (\bar{X}\bar{Y} + XY)] \\&= [X \cdot \bar{X}\bar{Y} + X \cdot XY] \\&= XY,\end{aligned}$$

Important Property

- In any string of terms interconnected only with XOR and/or EQV operators:
 - an odd number of complementations (variable or operator complementations) complements the function
 - an even number of complementations preserves the function

• Ex. 1:

$$\begin{aligned}
 F &= W \oplus X \oplus Y \oplus Z = W \oplus \bar{X} \oplus Y \odot Z \\
 &= W \odot \bar{X} \oplus \bar{Y} \oplus \bar{Z} = W \odot X \odot Y \oplus Z = \dots \\
 \bar{F} &= W \oplus \bar{X} \oplus Y \oplus Z = W \oplus \bar{X} \oplus Y \odot \bar{Z} \\
 &= W \odot X \odot Y \odot Z = \bar{W} \odot X \oplus \bar{Y} \oplus Z = \dots
 \end{aligned}$$

Handwritten notes: "4 changes" with arrows pointing to the four complementations in the first two lines; "1 change" with an arrow pointing to the complementation of Z in the third line.

• Ex. 2:

$$\begin{aligned}
 A \odot (A \odot D + \bar{C}) \odot \bar{B} &= A \odot [(A \oplus D)C] \odot B \\
 &= A \oplus [(A \oplus D)C] \oplus B,
 \end{aligned}$$

Corollary I

- If two functions α and β never take the logic 1 value at the same time, then:

$$\alpha \cdot \beta = 0 \text{ and } \alpha + \beta = \alpha \oplus \beta$$

and the logic operators "+" and \oplus are interchangeable.

a	b	$a+b$	$a \oplus b$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

Same



Corollary II

- If two functions α and β never take the logic 0 value at the same time, then:

$$\alpha + \beta = 1 \text{ and } \alpha \cdot \beta = \alpha \odot \beta$$

and the logic operators "." and \odot are interchangeable.

a	b	ab	$a \odot b$
0	0	0	1
0	1	0	0
1	0	0	0
1	1	1	1

Same

Examples of Corollaries I and II

- Ex. 1: $AB + \bar{B}C = (AB) \oplus (\bar{B}C),$
 where $\alpha = AB, \beta = \bar{B}C,$ and $\alpha \cdot \beta = 0$ by Corollary I.
- Ex. 2: $(\bar{A} + B + X) \cdot (A + B + C + Y) = (\bar{A} + B + X) \odot (A + B + C + Y),$
 where $\alpha = (\bar{A} + B + X), \beta = (A + B + C + Y)$ and $\alpha + \beta = 1$ according to Corollary II.
- Ex. 3: $a + \bar{b} \oplus bc = a + \bar{b} + bc = a + \bar{b} + c$
 where Corollary I has been applied followed by the absorptive law i
- Ex. 4: $(X\bar{Y}) \odot (\bar{X} + Y + \bar{Z}) = (X\bar{Y})(\bar{X} + Y + \bar{Z}) = X\bar{Y}\bar{Z}$
 Corollary II is applicable since $\overline{X\bar{Y}} = \bar{X} + Y,$

Useful Identities

$$\left\{ \begin{aligned} \bar{X} \oplus Y &= X \oplus \bar{Y} = \overline{X \oplus Y} = \bar{X} \odot \bar{Y} = X \odot Y \\ \bar{X} \odot Y &= X \odot \bar{Y} = \overline{X \odot Y} = \bar{X} \oplus \bar{Y} = X \oplus Y \end{aligned} \right\}$$

$$\left\{ \begin{aligned} X \oplus \bar{X} &= X \odot X = 1 \\ X \odot \bar{X} &= X \oplus X = 0 \end{aligned} \right\}$$

$$\left\{ \begin{aligned} 1 \oplus X &= \bar{X} \\ 0 \odot X &= \bar{X} \end{aligned} \right\} \quad \left\{ \begin{aligned} 0 \oplus X &= X \\ 1 \odot X &= X \end{aligned} \right\}$$

$$\left\{ \begin{aligned} X\bar{Y} \oplus X &= \bar{X}Y \oplus Y = XY \\ (X + \bar{Y}) \odot X &= (\bar{X} + Y) \odot Y = X + Y \end{aligned} \right\}$$

$$\left\{ \begin{aligned} \bar{X}Y \oplus X &= X\bar{Y} \oplus Y = 1 \oplus \bar{X}\bar{Y} \\ (\bar{X} + Y) \odot X &= (X + \bar{Y}) \odot Y = 0 \odot (\bar{X} + \bar{Y}) \end{aligned} \right\}$$

$$\left\{ \begin{aligned} (XY) \oplus (X + Y) &= X \oplus Y \\ (X + Y) \odot (XY) &= X \odot Y \end{aligned} \right\}$$

$$\left\{ \begin{aligned} XY + YZ + XZ &= XY \oplus YZ \oplus XZ \\ (X + Y)(Y + Z)(X + Z) &= (X + Y) \odot (Y + Z) \odot (X + Z) \end{aligned} \right\}$$

Example

- Using identities to simplify to XOR of product terms (XSOP):

$$\begin{aligned}XY + YZ + XZ &= XY(\bar{Z} + Z) + (\bar{X} + X)YZ + X(\bar{Y} + Y)X \\&= XYZ + XY\bar{Z} + XYZ + \bar{X}YZ + XYZ + X\bar{Y}Z \\&= XYZ \oplus XY\bar{Z} \oplus (\bar{X}YZ + X\bar{Y}Z) \\&= XY \oplus (Y \oplus X)Z \\&= XY \oplus YZ \oplus XZ.\end{aligned}$$

Generalization of Corollary I

- Generalization in canonical form

$$\begin{aligned} F_n(x_0, x_1, \dots, x_{n-1}) &= \sum_{i=0}^{2^n-1} (m_i \cdot f_i) \\ &= \bigoplus_{i=0}^{2^n-1} (m_i \cdot f_i) \\ &= (m_0 f_0) \oplus (m_1 f_1) \oplus (m_2 f_2) \oplus \dots \oplus (m_{2^n-1} f_{2^n-1}), \end{aligned}$$

Where the 2^n m_i terms represent minterms read in minterm codes and f_i values represent their respective coefficients whose values are 0 or 1.

- Minterms are, by definition mutually disjoint. i.e. only one minterm can be 1 for the same values of inputs. So, OR \rightarrow XOR (SOP \rightarrow EXSOP) is permissible.

Reed-Muller Expansion for EXSOP

- In the previous generalized formula, replace x_i' with $x_i \oplus 1$. This will eliminate all complements and results in all positive polarities. After considerable Boolean manipulation, especially using distributive law, it is recast as Reed-Muller expansion relationship (1954):

$$F_n(x_0, x_1, \dots, x_{n-1}) = g_0 \oplus g_1 x_{n-1} \oplus g_2 x_{n-2} \oplus g_3 x_{n-2} x_{n-1} \\ \oplus g_4 x_{n-3} \dots \oplus g_{2^n-1} x_0 x_1 \dots x_{n-1},$$

$$g_i = \bigoplus_{j \subseteq i} f_j.$$

subnumbers

where the g_i values are called the Reed-Muller (R-M) coefficients for a positive polarity.

R-M Coefficients

- Each R-M coefficient g_i is obtained from the subnumbers of i by replacing m 1's with 0's in 2^m possible ways in the binary number corresponding to decimal i : $g_i = \bigoplus_{j \subseteq i} f_j$. *subnumbers*
- $g_i = 1$ if an odd number of f coefficients are logic 1
- $g_i = 0$ if an even number of f coefficients are logic 1

$$g_0 = f_0$$

...000

$$g_1 = \oplus f(1, 0) = f_1 \oplus f_0$$

...001 ... 000

$$g_2 = \oplus f(2, 0) = f_2 \oplus f_0$$

...010 ... 000

$$g_3 = \oplus f(3, 2, 1, 0) = f_3 \oplus f_2 \oplus f_1 \oplus f_0$$

...011 ... 010 ... 001 ... 000

$$g_4 = \oplus f(4, 0) = f_4 \oplus f_0$$

...100 ... 000


$$g_5 = \oplus f(5, 4, 1, 0) = f_5 \oplus f_4 \oplus f_1 \oplus f_0$$

... 101 ... 100 ... 001 ... 000

$$g_{2^n-1} = \bigoplus_{i=0}^{2^n-1} f_i$$

All combinations in positions 0 and 2 should be exercised.

EXSOP Example

- Find the optimized EXSOP form of $F = \bar{A}BC + A\bar{B} + A\bar{C}$
- Use corollary I: $F = \bar{A}BC + A\bar{B} + A\bar{C} = \sum m(3,4,5,6)$
 $= \oplus m(3,4,5,6)$
 $= \bar{A}BC \oplus A\bar{B}\bar{C} \oplus A\bar{B}C \oplus ABC\bar{C}$

Non-Optimized
- $f_0=f_1=f_2=f_7=0$ and $f_3=f_4=f_5=f_6=1$
- The g_i values, therefore are:

$$g_0 = f_0 = 0$$

$$g_1 = \oplus f_1(1, 0) = 0$$

$$g_2 = \oplus f_2(2, 0) = 0$$

$$g_3 = \oplus f_3(3, 2, 1, 0) = 1$$

$$g_4 = \oplus f_4(4, 0) = 1$$

$$g_5 = \oplus f_5(5, 4, 1, 0) = 0$$

$$g_6 = \oplus f_6(6, 4, 2, 0) = 0$$

$$g_7 = \oplus f_7(7-0) = 0.$$

- Final result (note that 3=011 and 4=100):

$$F = g_3BC \oplus g_4A = BC \oplus A \quad \longleftarrow \text{Optimized}$$

Generalization of Corollary II

- Generalization in canonical form

$$\begin{aligned}F_n(x_0, x_1, x_2, \dots, x_{n-1}) &= \prod_{i=0}^{2^n-1} (M_i + f_i) \\&= \bigodot_{i=0}^{2^n-1} (M_i + f_i) \\&= (M_0 + f_0) \odot (M_1 + f_1) \odot (M_2 + f_2) \\&\quad \odot \dots \odot (M_{2^n-1} + f_{2^n-1}),\end{aligned}$$

Where the 2^n m_i terms represent maxterms read in maxterm codes and f_i values represent their respective coefficients whose values are 0 or 1.

- Maxterms are, by definition mutually disjoint. i.e. only one maxterm can be 1 for the same values of inputs. So, AND \rightarrow XNOR (POS \rightarrow EQSOP) is permissible.

Reed-Muller Expansion for EQPOS

- In the previous generalized formula, replace x_i with $x_i' \odot 0$. This will eliminate all complements and results in all negative polarities. After considerable Boolean manipulation, especially using distributive law, it is recast as Reed-Muller expansion relationship:

$$F_n(x_0, x_1, x_2, \dots, x_{n-1}) = g_0 \odot (g_1 + \bar{x}_{n-1}) \odot (g_2 + \bar{x}_{n-2}) \odot (g_3 + \bar{x}_{n-2} + \bar{x}_{n-1}) \\ \odot (g_4 + \bar{x}_{n-3}) \odot \dots \odot (g_{2^n-1} + \bar{x}_0 + \bar{x}_1 + \dots + \bar{x}_{n-1})$$

$$g_i = \bigodot_{j \subseteq i} f_j$$

where the g_i values are called the Reed-Muller (R-M) coefficients for a negative polarity.

Reed-Muller Expansion Generalized

- See Saul thesis – mixed form best for logic synthesis

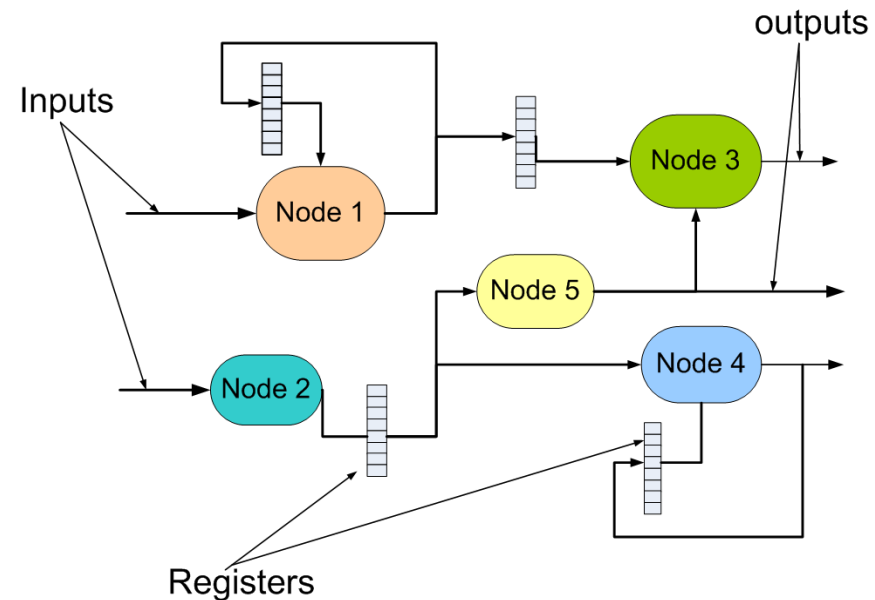
FPGA Synthesis

Agenda

- Brief tour in RTL synthesis
 - Basic concepts and representations
- LUT-based technology mapping
 - The chortle algorithm
 - The FlowMap approach

RTL Representation

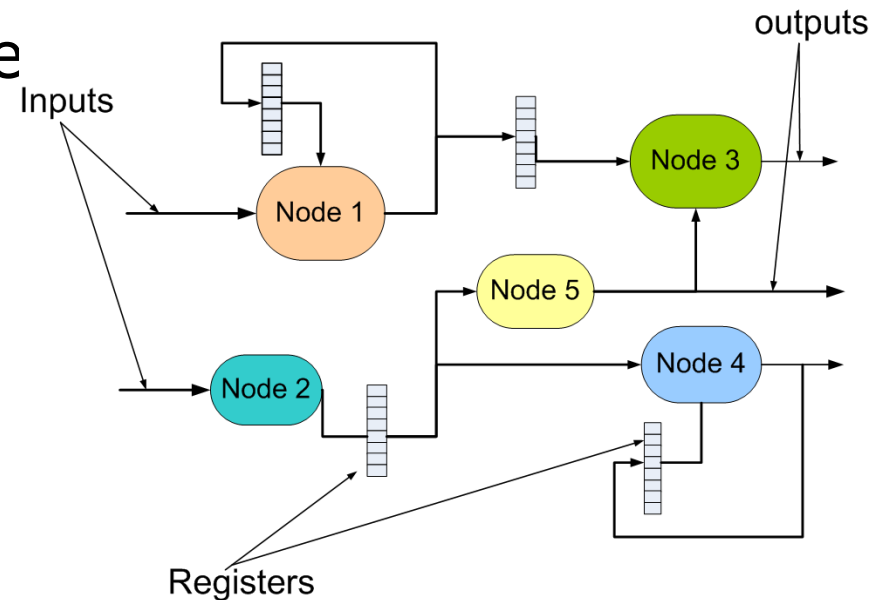
- A structured system:
 - Made upon a set of combinational parts separated by memory elements
- RTL
 - Computation:
 - Performed in the combinational parts
 - Results
 - Might be temporarily stored in registers (placed between the combinational blocks)
 - Clock:
 - Synchronizes the transfer of data from register to register via combinational parts



A structured digital system

Synthesis Goal

- RTL synthesis goal:
 - To provide an optimal implementation of a structure system
 - for a given hardware platform
- FPGA-Goal:
 - Generation of configuration data
- Factors:
 - Area,
 - Delay,
 - Power consumption,
 - Testability,



A structured digital system

RTL Synthesis

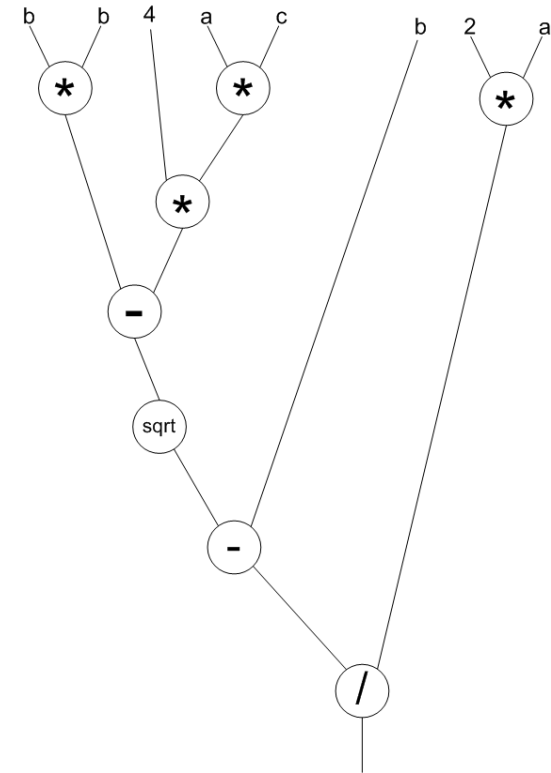
- Inputs:
 - A number of data path components,
 - Binding of operations to data path components,
 - A controller (FSM) that contains the detailed schedule (related to clock edge) of
 - computational
 - I/O
 - memory operations
- Output:
 - Logic-level implementations of the design

RTL Synthesis

- For FPGA, can use existing ASIC RTL synthesis techniques
 - Relatively matured area
- But must consider:
 - FPGA special circuitry features in logic blocks:
 - Fast carry chains (→ better performance)
 - FPGA hard structures:
 - Blocks of memory
 - Multipliers
 - FPGA flexible soft blocks
- Two stages:
 1. Datapath synthesis
 2. Control synthesis

Dataflow Graph

- Input:
 - DFG or CDFG
- DFG:
 - Means to describe a computing task in a streaming mode
 - **Operators:** nodes
 - **Operands:** Inputs of the nodes
 - Node's output can be used as input to other nodes
 - Data dependency in the graph
- Given a set of tasks $\{T_1, \dots, T_k\}$, a **DFG** is a DAG $G(V, E)$, where
 - $V (= T)$: the set of nodes representing operators and
 - E : the set of edges representing data dependency between tasks



DFG for the quadratic root computation using:

$$x = \frac{(b^2 - 4ac)^{\frac{1}{2}} - b}{2a}$$

DFG

- Any high-level program can be compiled into a DFG, provided that:
 - No loops
 - No branching instructions.
- Therefore:
 - DFG is used for parts of a program.
 - Extend to other models:
 - Sequencing graph
 - FSMD
 - CDFG (Control DFG)

Datapath Synthesis

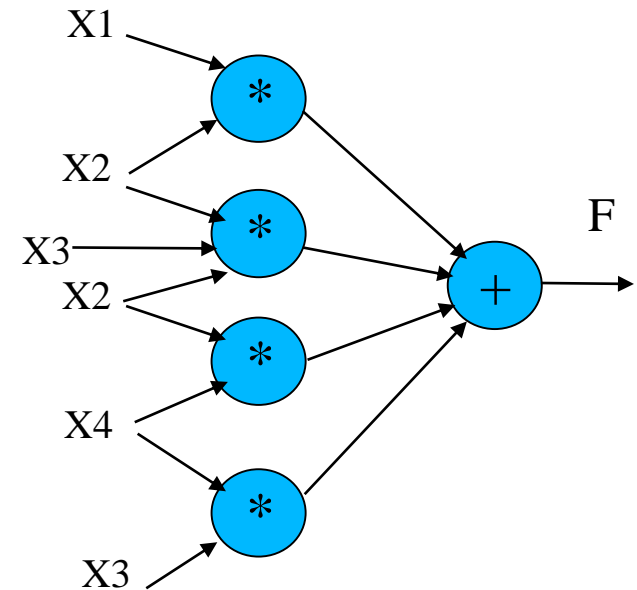
- Steps:
 - Implement each datapath node with a datapath component
 - Flatten the datapath components to gates
 - Problems:
 - + Discards information about regularity of design (bit-slice)
 - + Usually not possible to rediscover uses of specialized features of the CLBs (e.g., fast carry chain)
 - Solution:
 - + Preserve special datapath structures
 - Feed the resulting netlist to the gate-level design flow

Datapath Synthesis

- Technology-independent optimization
 - Very much like ASIC synthesis techniques
 - Not focus of this course:
 - Just a general taste

Two-level logic

- Two approaches to logic synthesis:
 1. Two-level logic synthesis:
 - targets designs represented in two-level SOP
 - Products are implemented on the first level and the sum on the second level
 - **Advantages:**
 - Natural representation of Boolean functions
 - Well understood and easy manipulation
 - **Drawbacks:**
 - Bad estimator of complexity during logic optimization
 - Initially developed for PALs and PLAs, later for CPLDs



Two-level logic

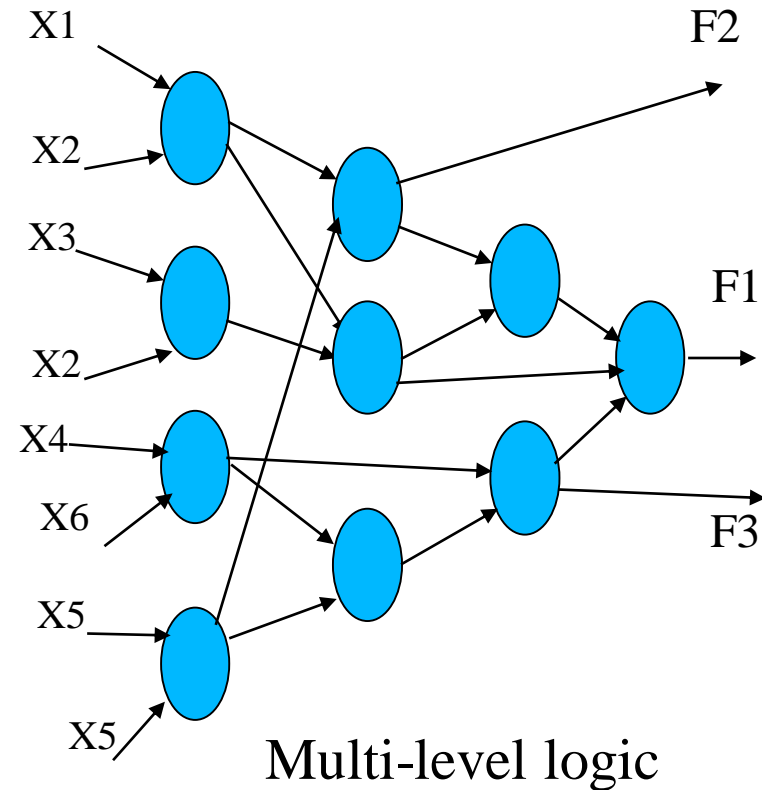
Multi-level-logic

2. Multi-level logic synthesis:

- Targets multi-level designs
- Many Boolean functions on the path from the inputs to the outputs

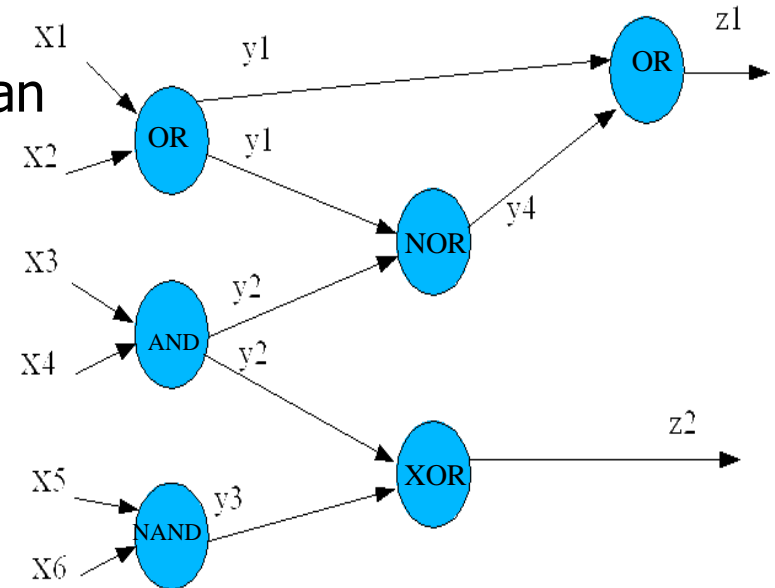
- Appropriate for mask-programmable or
- field programmable devices

- Multi-level will therefore be considered in this course



Boolean Networks

- Multi-level logic:
 - are usually represented using Boolean networks
- A Boolean network: a DAG:
 - Node:
 - an arbitrary Boolean function
 - Edge:
 - data dependency between nodes
- Proper representation is required for Manipulation
- Important factors:
 - memory efficiency
 - correlation with the final representation



$$\begin{aligned}y_1 &= x_1 + x_2 \\y_2 &= x_3 \cdot x_4 \\y_3 &= (x_5 \cdot x_6)' \\y_4 &= (y_1 + y_2)' \\z_1 &= y_1 + y_4 \\z_2 &= y_2 \oplus y_3\end{aligned}$$

Synthesis Approaches

- Synthesis Approaches:
 1. Technology dependent:
 - Only valid gates (from the target library) are used in the node representation.
 2. Technology independent:
 - The design is not tied to any library.
 - A final mapping must be done to the final library.
 - Mostly used
- For FPGA devices: in two steps:
 1. All the Boolean equations are minimized, independent of the logic blocks.
 2. Technology mapping maps the parts of the Boolean network to LUTs.

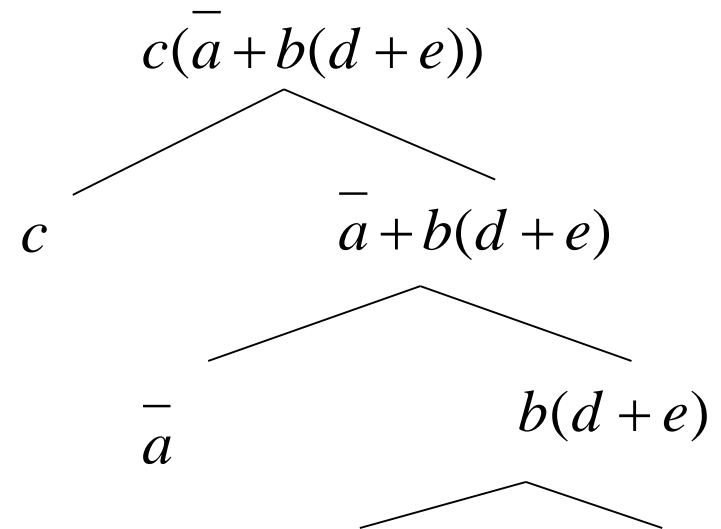
Representation

- Node representation choices:
 1. Sum-Of-Products (SOP)
 2. Factored Form (FF)
 3. Binary Decision Diagram (BDD)
- Sum-Of-Products:
 - ✓ Well understood
 - ✓ Easy to manipulate
 - ✓ Many optimization algorithms available
 - ✗ Not representative of the logic complexity
 - Estimation of progress during logic minimization is difficult

Representation: FF

- Factored form (FF):
 - Defined recursively:
 - (FF = **product**) or (FF = **sum**).
 - (product = **literal**) or (product = **FF1*FF2**).
 - (sum = **literal**) or (sum = **FF1+FF2**).

- Example:



✖ Few manipulation algorithms.

Representation: BDD

BDD:

— A rooted DAG

— Nodes:

— **Variable node:**

— non-terminal:

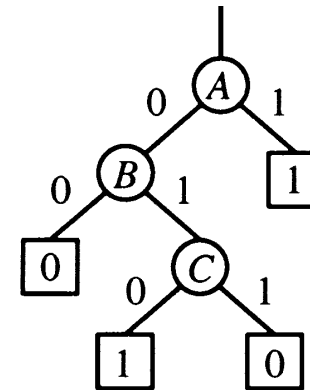
+ $index(v) \in \{1, \dots, n\}$ (for variable x_i)

+ Two children $low(v)$, $high(v)$

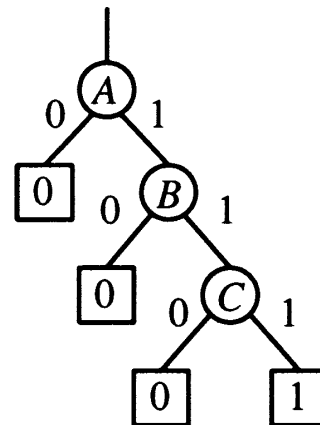
— **Constant nodes:**

— terminal node:

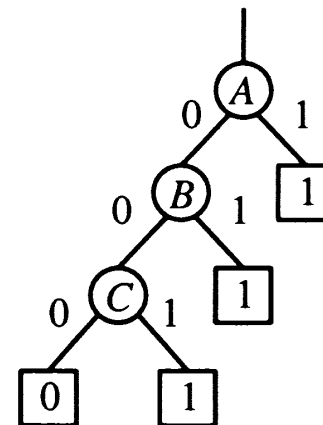
+ $value(v) \in \{0, 1\}$



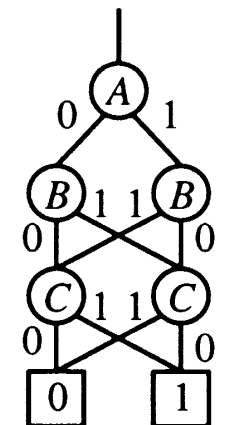
$A + B \cdot C'$



(a) $A \cdot B \cdot C$



(b) $A + B + C$



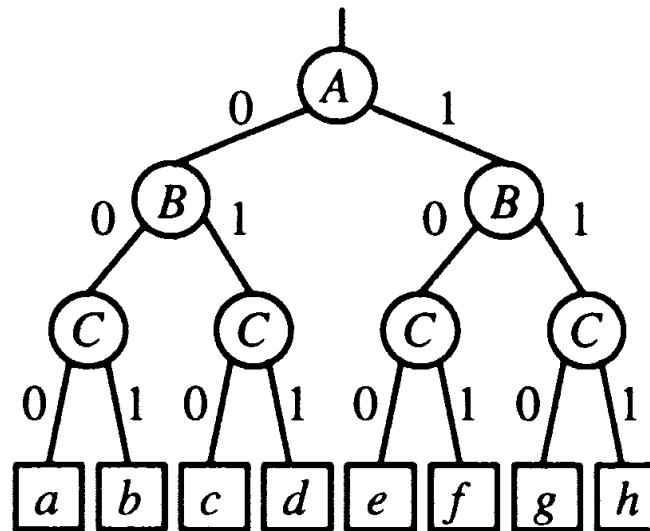
(c) $A \oplus B \oplus C$

Representation: BDD

- BDD from Truth Table:

<i>A</i>	<i>B</i>	<i>C</i>	<i>f</i>
0	0	0	<i>a</i>
0	0	1	<i>b</i>
0	1	0	<i>c</i>
0	1	1	<i>d</i>
1	0	0	<i>e</i>
1	0	1	<i>f</i>
1	1	0	<i>g</i>
1	1	1	<i>h</i>

(a)

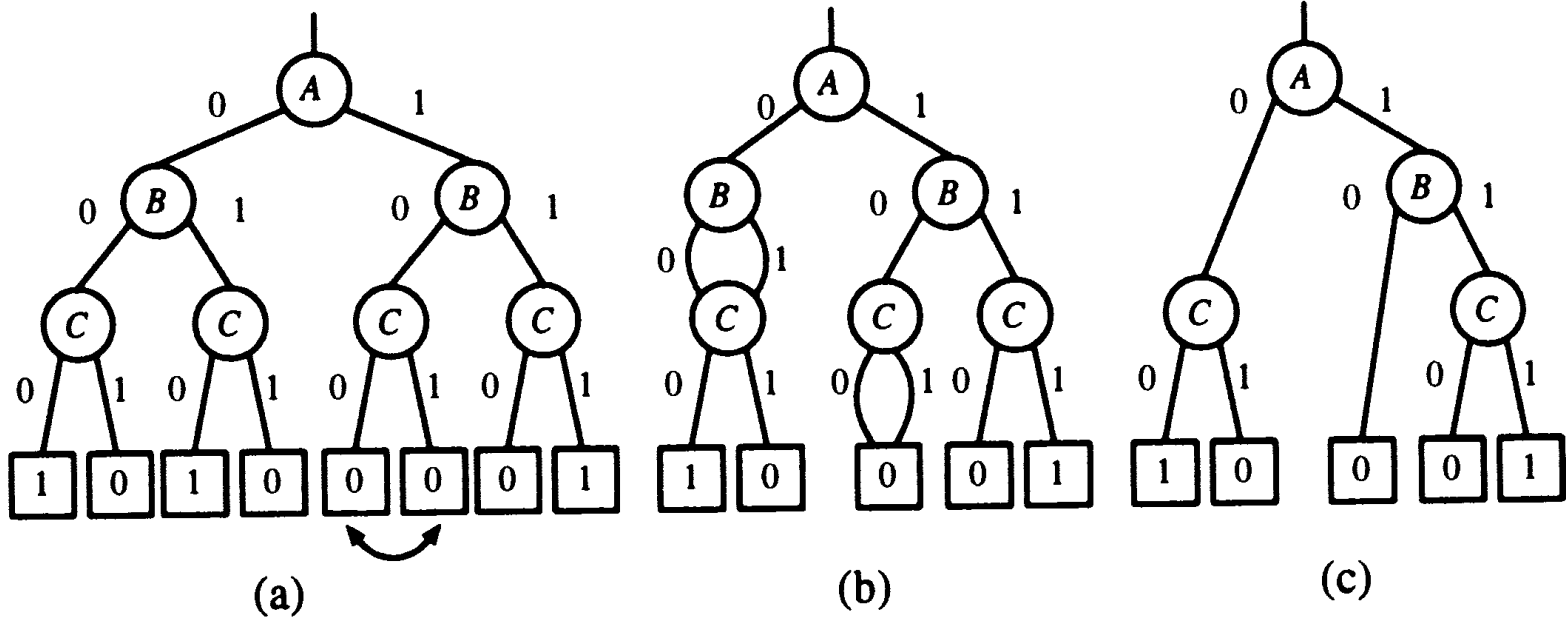


(b)

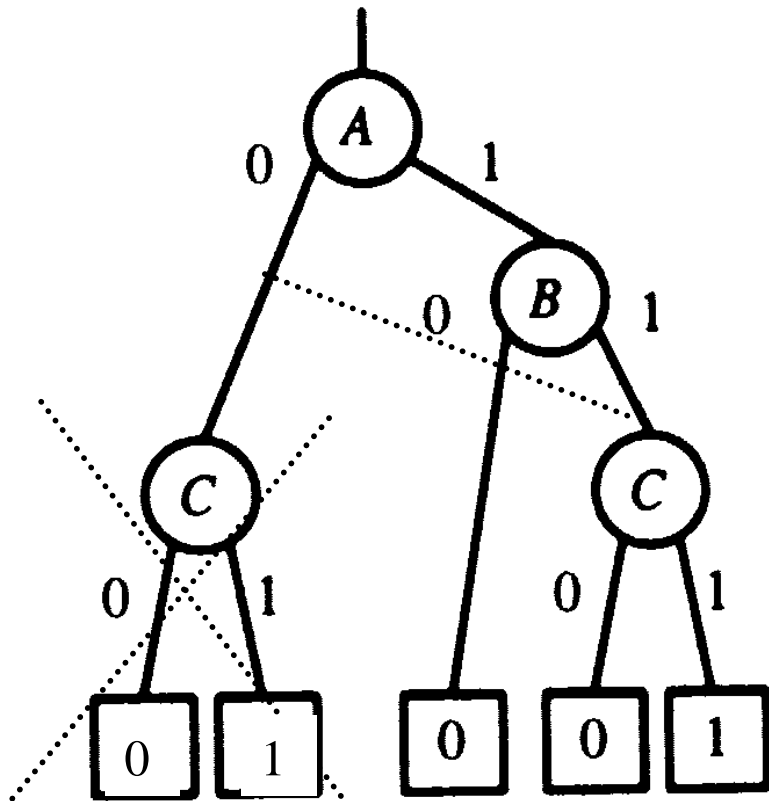
Representation: BDD

- BDD Simplification

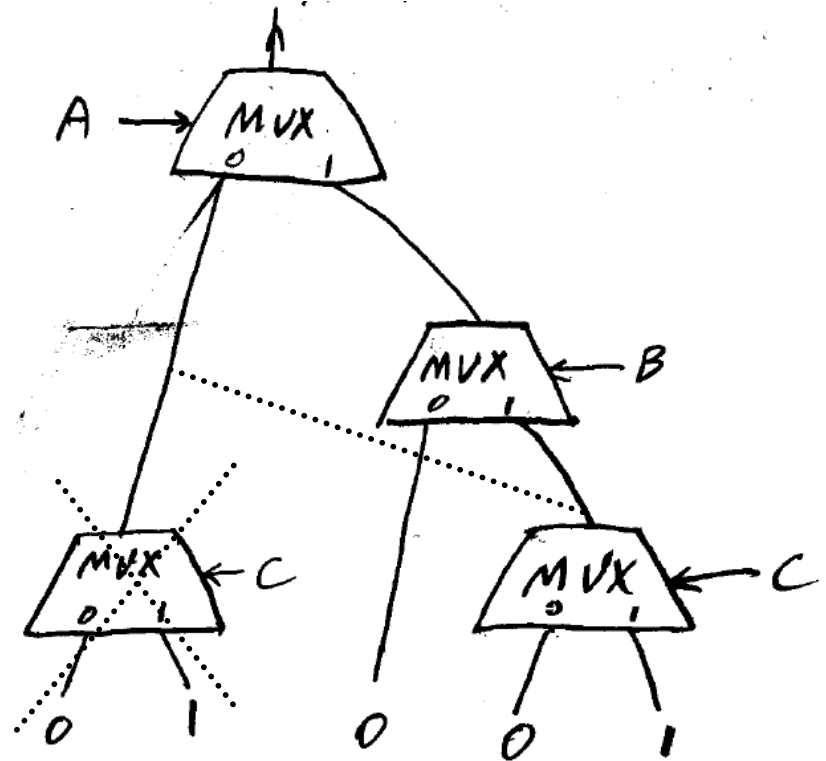
$$ABC + A'C'$$



- Implementation by MUX



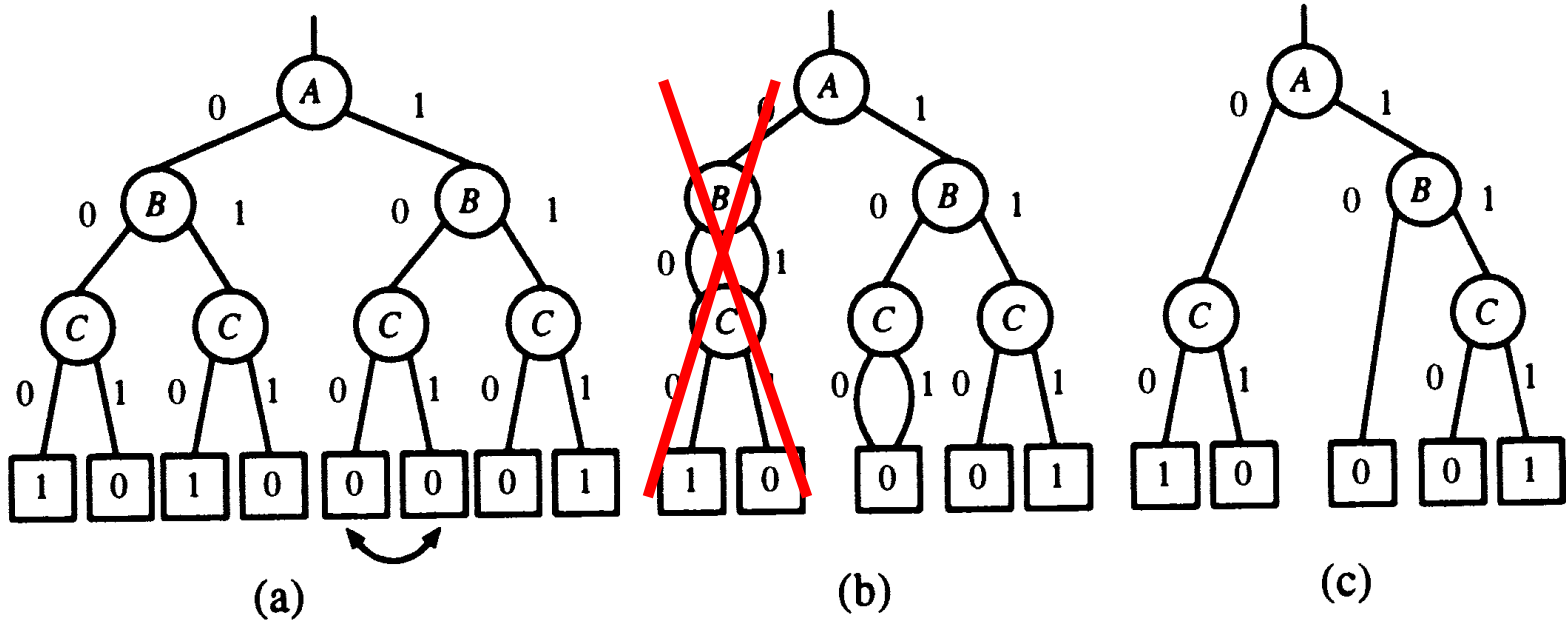
$$F = ABC + \bar{A}C$$



Representation: BDD

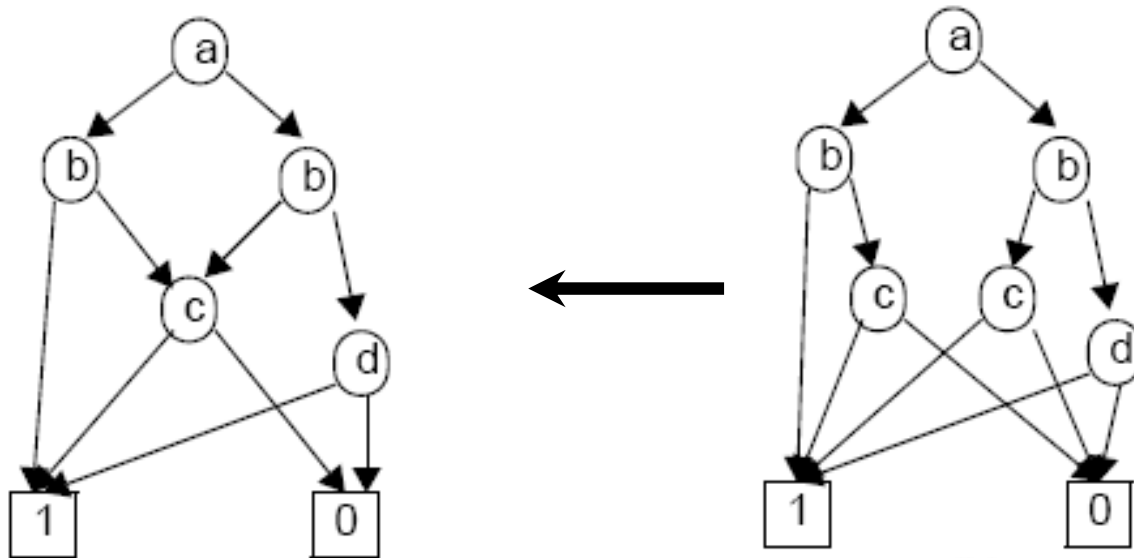
- BDD Simplification

$$ABC + A'C'$$



Representation: ROBDD

- BDD \rightarrow ROBDD
 - Fix order of variables
 - Delete redundant nodes
 - Share subgraphs that represent same function

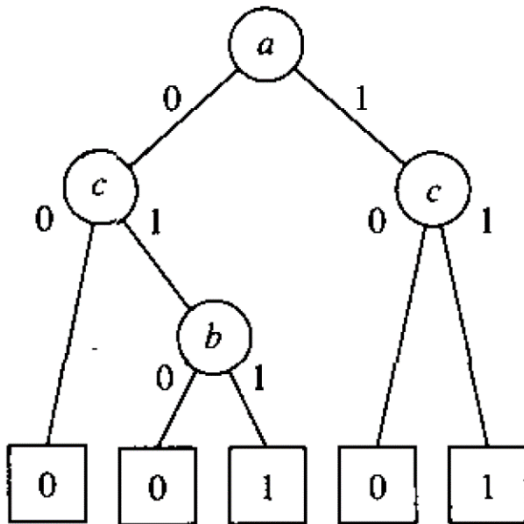


ROBDD is canonical

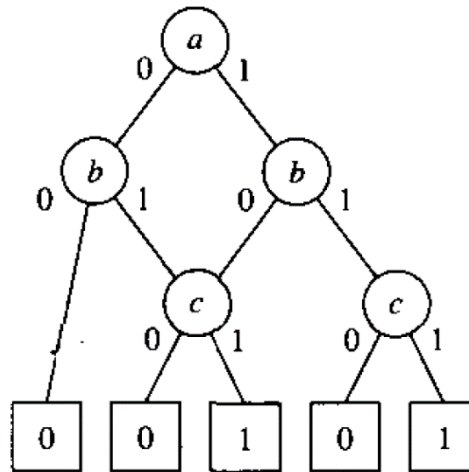
ROBDD Example

- $f = (a + b) \cdot c$

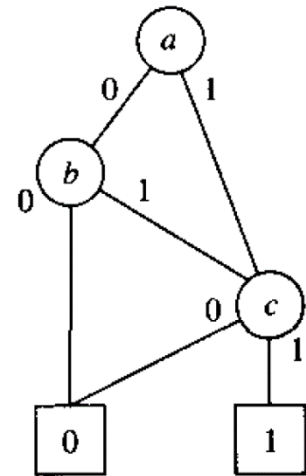
OBDD for order
(a,c,b)



OBDD for order
(a,b,c)



ROBDD for order
(a,b,c)



DDBDD: Delay-Driven BDD Synthesis for FPGAs

Lei Cheng

Deming Chen

Martin D.F. Wong

**Coordinated Science Laboratory
University of Illinois at Urbana-Champaign**

DAC07: CD43\49-1

Node Manipulation

- Node Manipulation:
 - Input:
 - A suitable node representation
 - Goal:
 - Generation of an equivalent and cost effective simplified function
- Operations:
 - Decomposition
 - Extraction
 - Factoring
 - Substitution
 - Collapsing (Elimination)

Node Manipulation: Decomposition

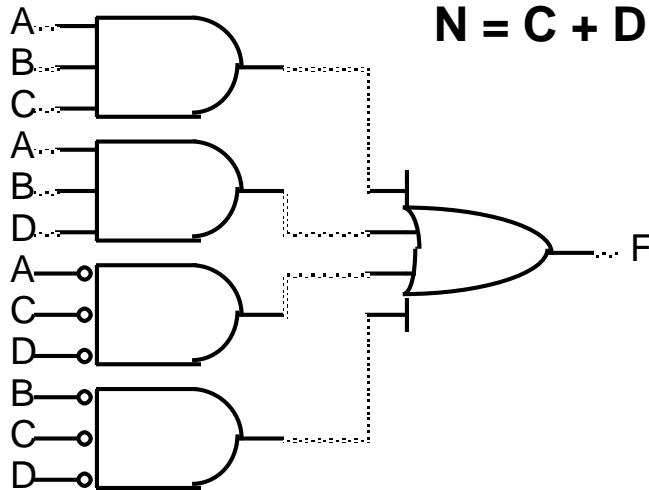
- Decomposition:

- Take a single Boolean expression and replace with collection of new expressions:
- A Boolean function $f(X)$ is decomposable if we can find a function $g(X)$ such that $f(X) = f'(g(X), X)$.

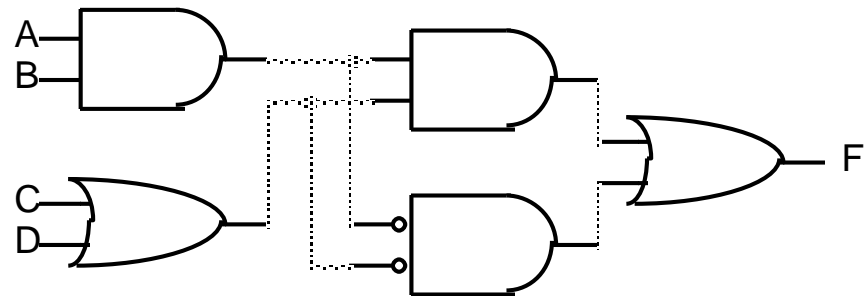
$$F = A B C + A B D + A' C' D' + B' C' D' \quad (12 \text{ literals})$$

$$\begin{aligned} F &= M N + M' N' \\ M &= A B \\ N &= C + D \end{aligned}$$

(8 literals)



Before Decomposition

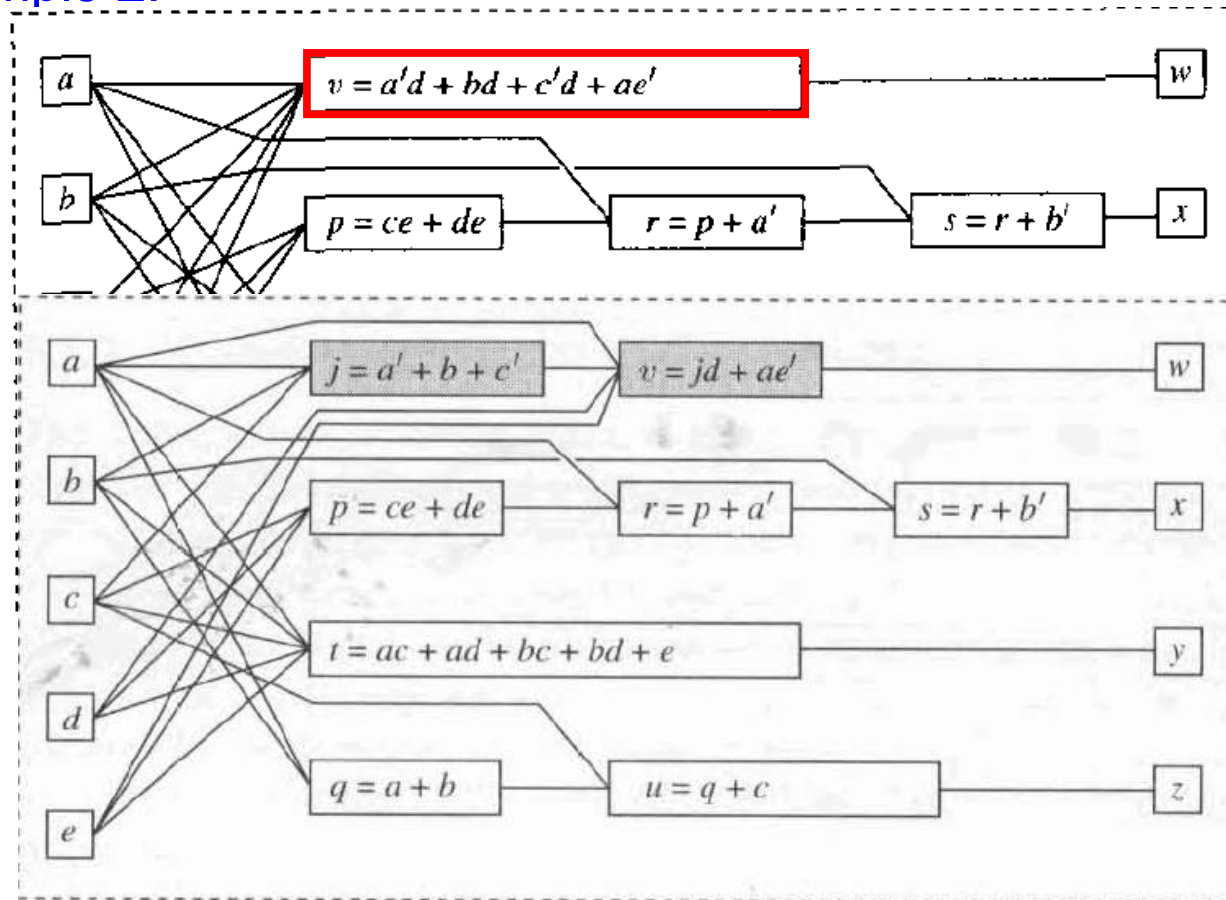


After Decomposition

Node Manipulation: Decomposition

- Decomposition:

- Example 2:



Node Manipulation: Extraction

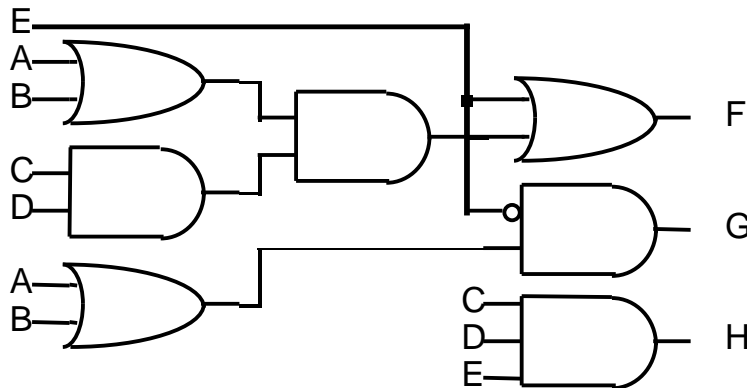
• Extraction:

- Identify common intermediate sub-functions from a set of given functions.

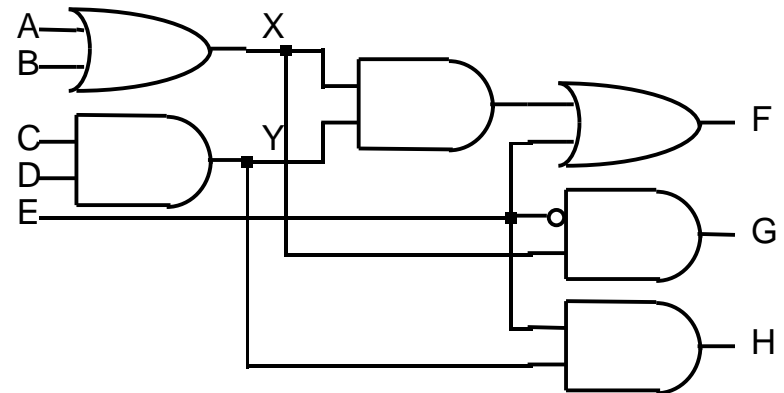
$$\begin{aligned} F &= (A + B) C D + E & (11 \text{ literals \& 8 gates}) \\ G &= (A + B) E' \\ H &= C D E \end{aligned}$$

$$\begin{aligned} F &= X Y + E & (11 \text{ literals \& 7 gates}) \\ G &= X E' \\ H &= Y E \\ X &= A + B \\ Y &= C D \end{aligned}$$

"Kernels":
primary divisors



Before Extraction

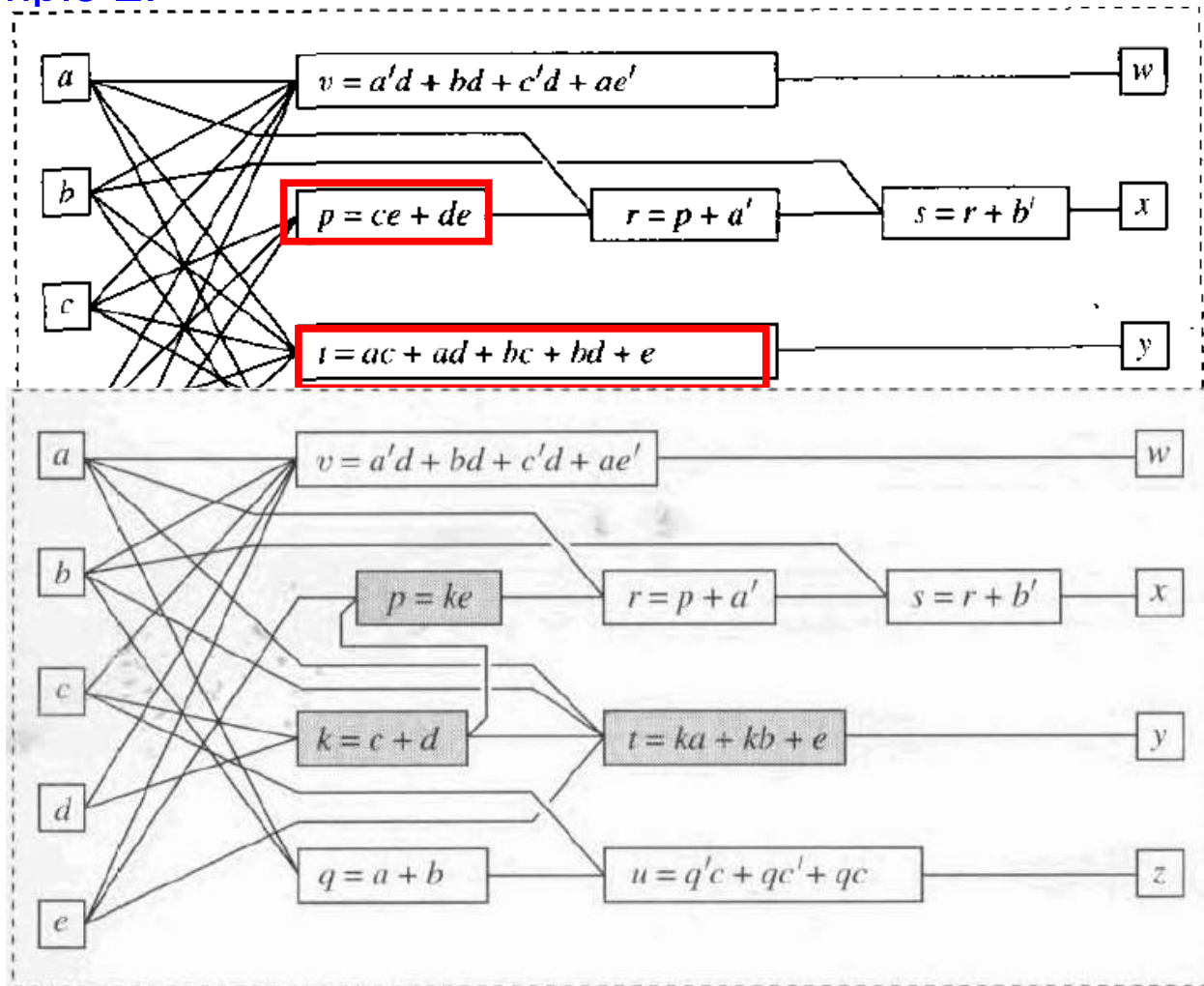


After Extraction

Node Manipulation: Extraction

- Extraction:

- Example 2:



Node Manipulation: Factoring

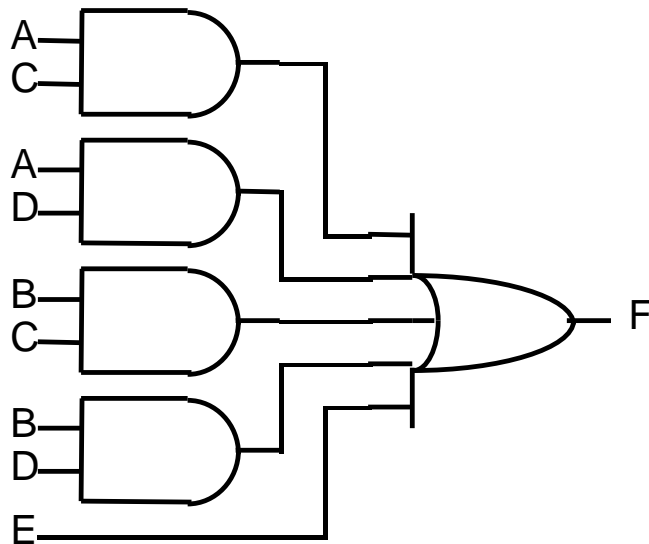
- Factoring:

- Transformation of SOP-expressions in factored form

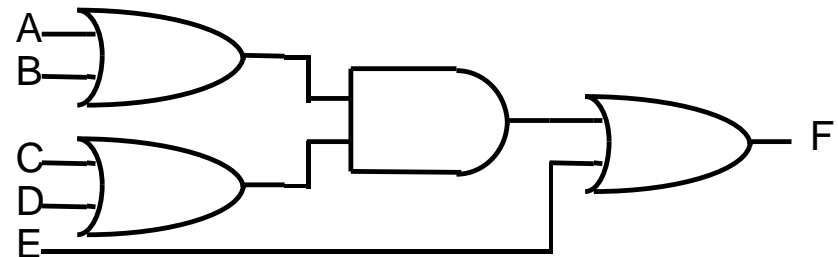
$$F = A C + A D + B C + B D + E \quad (9 \text{ literals \& 5 gates})$$

$$F = (A + B) (C + D) + E$$

(7 literals & 4 gates)



Before Factoring



After Factoring

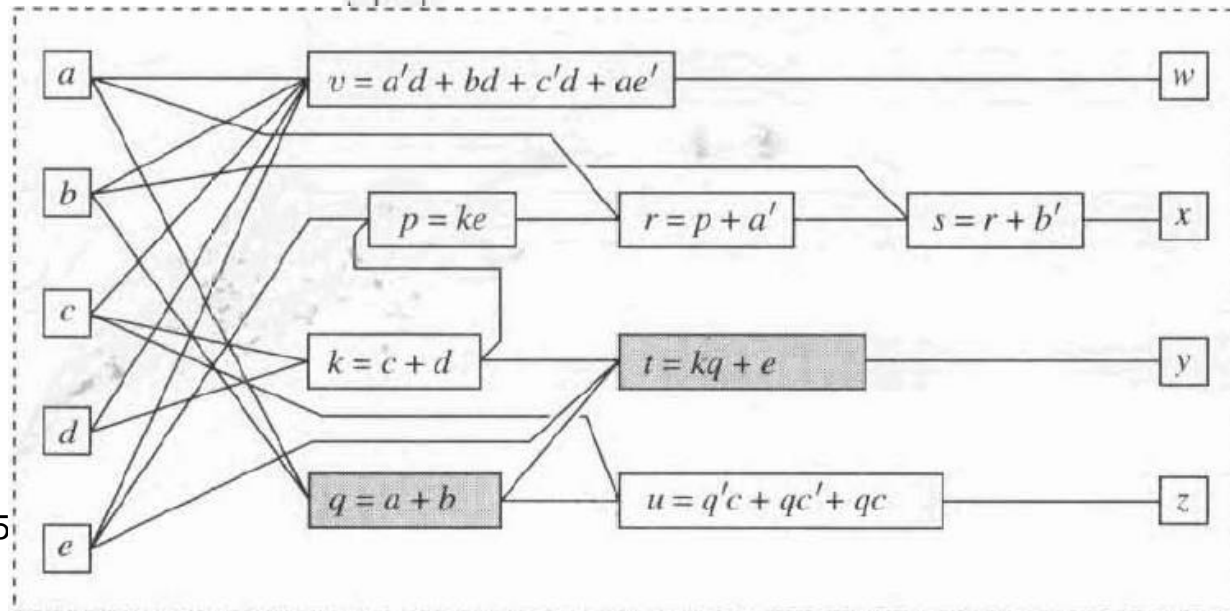
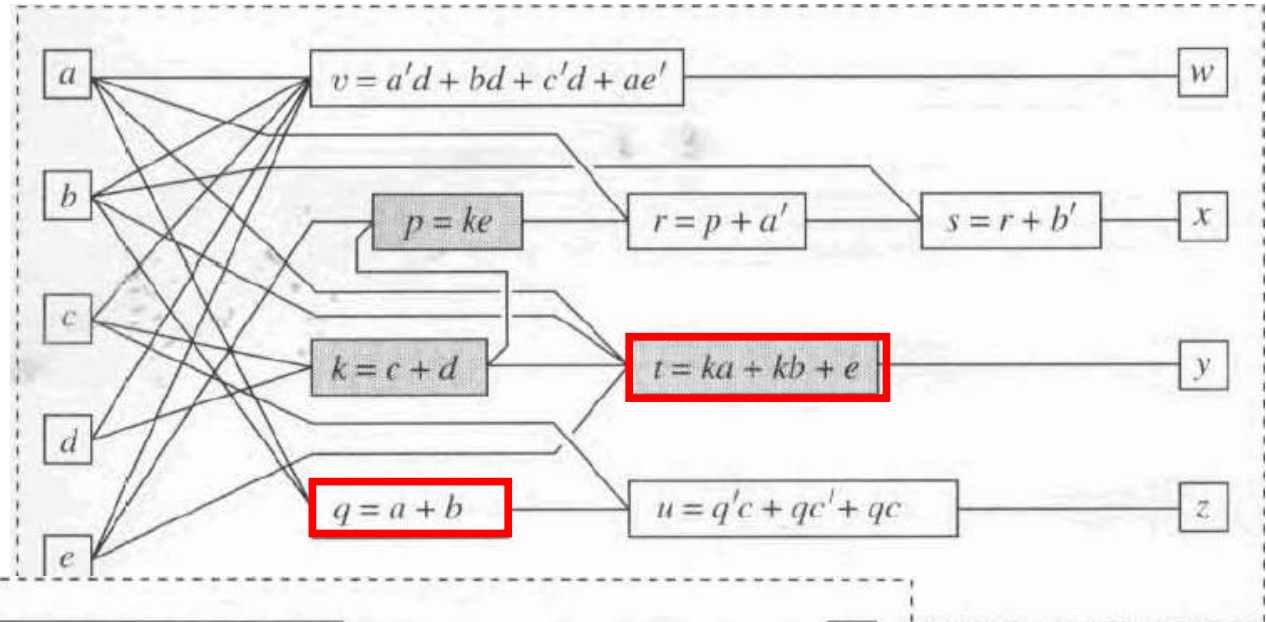
Node Manipulation: Substitution

- **Substitution:**
 - A function is reduced in complexity by using an additional input that was not previously in its support set.
 - The transformation requires the creation of a **dependency**,
 - but it may also lead to dropping others.

Node Manipulation: Substitution

- Substitution:

- Example:



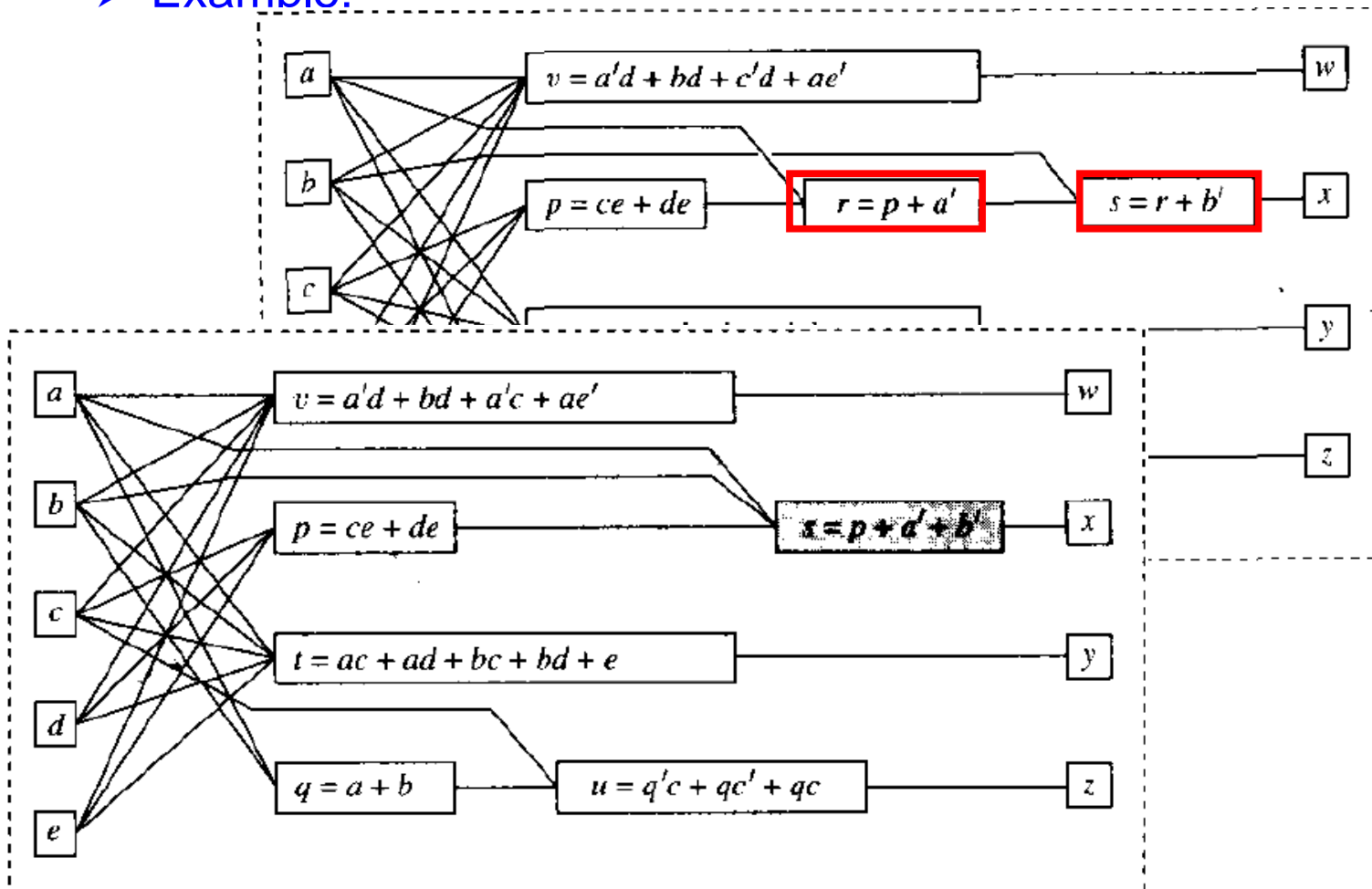
Node Manipulation: Collapsing

- **Collapsing (Elimination):**
 - The elimination of an internal vertex is its removal from the network.
 - The variable corresponding to the vertex is replaced by the corresponding expression.
 - Reverse of substitution.
 - To eliminate levels to meet timing constraints.

Node Manipulation: Collapsing

- Collapsing:

- Example:



References

- [Bobda07] C. Bobda, “Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications,” Springer, 2007.
- I. J. Teich, “Reconfigurable computing design and implementation,” Lecture Slides.
- [Cong06] D. Chen, J. Cong and P. Pan, “FPGA Design Automation: A Survey,” Foundations and Trends in Electronic Design Automation, Vol. 1, No. 3 (2006) 195–330.
- [DeMicheli94] G. De Micheli, “Synthesis and optimization of digital circuits,” McGraw-Hill, 1994.