Outline

Division is central in many operations.

- What is it (in the context of Boolean functions)?
- How to divide?
- What to divide with?
- Applications: factoring, resubstitution, extraction

Product

DEFINITION 1 *An* **algebraic expression** *is a sum of products representation of a logic function which is minimal with respect to single cube containment.*

Example: $ab + abc + cd$ is not an algebraic expression, $ab + cd$ is.

DEFINITION 2 *The* **product** *of two algebraic expressions f and g, fg is a $\sum c_i d_j$ where $\{c_i\} = f, \{d_j\} = g$, made irredundant with respect to single cube containment e.g. $ab + a = a$).*

- Algebraic product: when f and g have disjoint supports.
- Boolean product: otherwise.

- Example:

-$(a+b)(c+d) = ac+ad+bc+bd$ is an algebraic product

-$(a+b)(a+c) = aa+ac+ab+bc = a+bc$ is a Boolean product.

Division

DEFINITION 3 *g is a* **Boolean divisor** *of f if h and r exist such that $f = gh + r$, $gh \neq \emptyset$.*

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

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

- If gh is restricted to an algebraic product, h is the **algebraic quotient**, denoted $f//g$. Otherwise, h is a (non-unique) **Boolean quotient** denoted $f \div g$. (*We will reserve the notation f/g for the more useful "weak-division", defined later*).

- If $h \neq \emptyset$, and h can be obtained using algebraic division, then g is an **algebraic divisor** of f. Otherwise, g is a **Boolean divisor** of f.

Division

- Example:

$$
\begin{aligned}
f &= ad + ae + bcd + j \\
g_1 &= a + bc \\
g_2 &= a + b
\end{aligned}
$$

- Algebraic division $f//a = d + e, f//(bc) = d$
  (*Also,* $f//a = d$ *or* $f//a = e$*, i.e. algebraic division is not unique*)
  $h_1 = f//g_1 = d, r_1 = ae + j$

- Boolean division: $h_2 = f \div g_2 = (a + c)d$, $r_2 = ae + j$. i.e. $f = (a + b)(a + c)d + ae + j$

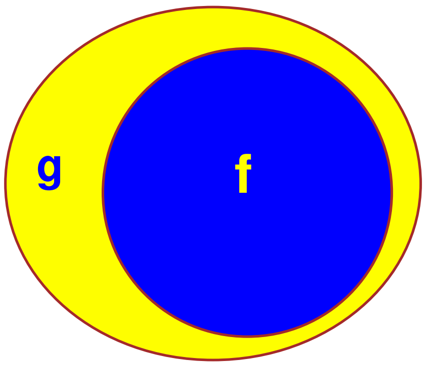
DEFINITION 4 *g is an* **algebraic factor** *of f if there exists an algebraic expression h such that $f = gh$ (using algebraic multiplication).*

Why Use Algebraic Methods

- need spectrum of operations - Algebraic methods provide fast algorithms
- treat logic function like a polynomial
- fast methods for manipulation of polynomials available
- loss of optimality, but results quite good
- can iterate and interleave with Boolean operations

DIVISION

THEOREM 5 *A logic function g is a Boolean factor of a logic function f if and only if $f \subseteq g$ (i.e. $fg' = \emptyset$, i.e. $g' \subseteq f'$).*



Proof.
$\Rightarrow$: g is a Boolean factor of f. Then $\exists h$ such that $f = gh$; Hence, $f \subseteq g$ (as well as h).

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

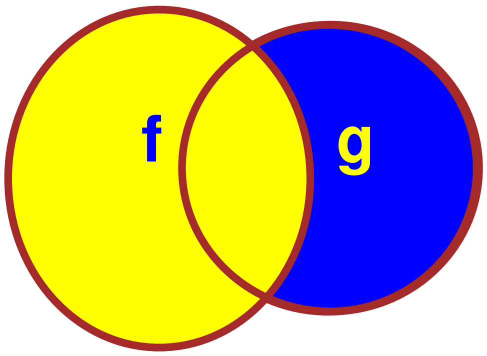
Notes:

1. $h = f$ works fine for the proof.

2. Given f and g, h is not unique. To get a small h is the same as getting a small $f + r$. Since $rg = \emptyset$, this is the same as minimizing (simplifying) f with DC $= g'$.

DIVISION

THEOREM 6 *g is a Boolean divisor of f if and only if $fg \neq \emptyset$.*



Proof.
$\Rightarrow$: $f = gh + r, gh \neq \emptyset \Rightarrow fg = gh + gr$. Since $gh \neq \emptyset, fg \neq \emptyset$.

$\Leftarrow$: $f = fg + fg' = g(f + k) + fg'$. Here $k \subseteq g'$. Then $f = gh + r$, with $h = f + k$, $r = fg'$. Since $gh = fg \neq \emptyset$, $gh \neq \emptyset$. ∎

Note: f has many divisors. We are looking for a g such that $f = gh + r$, where g, h, r are **simple** functions.

Incompletely Specified Functions

$\mathcal{F} = (f, d, r)$

DEFINITION 7 *A completely specified logic function g is a* **Boolean divisor** *of $\mathcal{F}$ if there exist h, e (completely specified) such that*

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

and $gh \not\subseteq d$.

DEFINITION 8 *g is a* **Boolean factor** *of $\mathcal{F}$ if there exists h such that*

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

LEMMA 9 *$f \subseteq g$ if and only if g is a Boolean factor of $\mathcal{F}$.*

Proof.
$\Rightarrow$: Let $h = f + k$ where $kg \subseteq d$. Then $hg = (f + k)g \subseteq (f + d)$. Since $f \subseteq g, fg = f$ and thus $f \subseteq (f + k)g = gh$. Thus

$$f \subseteq (f + k)g \subseteq f + d$$

$\Leftarrow$: Suppose $\exists$ minterm m such that $f(m) = 1$ but $g(m) = 0$. Then $f(m) = 1$ but $g(m)h(m) = 0$ implying that $f \not\subseteq gh$. ∎

Note: Since $kg \subseteq d, k \subseteq (d + g')$. Hence obtain $h = f + k$ by simplifying f with DC $(d + g')$.

Incompletely Specified Functions

LEMMA 10 *$fg \neq \emptyset$ if and only if g is a Boolean divisor of $\mathcal{F}$.*

Proof.
$\Rightarrow$: Assume $fg \neq \emptyset$. Let $fg \subseteq h \subseteq (f + d + g')$ and $fg' \subseteq e \subseteq (f + d)$. Then

$$f = fg + fg' \subseteq gh + e \subseteq g(f + d + g') + f + d = f + d$$

Also, $\emptyset \neq fg \subseteq gh \rightarrow ghf \neq \emptyset$. Since $fd = \emptyset$ then $gh \not\subseteq d$, verifying the conditions for Boolean division.

$\Leftarrow$: Suppose $\exists h$ such that $gh \not\subseteq d$ and

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

Since $gh \not\subseteq d$, then $gh \neq \emptyset$. Hence $ghf \neq \emptyset$ implying that $fg \neq \emptyset$. ∎

Recipe:

1. Choose g such that $fg \neq \emptyset$.

2. Simplify fg with DC $(g' + d)$ to get h. Simplify fg' with DC $(d + fg)$ to get e. (*actually could simplify $f(g' + h')$ with DC $(d + fgh)$ here*)

Thus

$$\begin{aligned} fg &\subseteq h \subseteq g' + d \\ fg' &\subseteq e \subseteq fg' + d + fg = f + d \end{aligned}$$

$$\mathcal{F} = (f, d, r)$$

LEMMA 11 *Suppose g is an algebraic divisor of F, a cover of $\mathcal{F}$. If $f \not\subseteq e$* (where e is the remainder in the algebraic division, i.e. $F = gh + e$) *then g is a Boolean divisor of $\mathcal{F}$.*

Proof. Assume $F = gh + e, gh \neq 0, f \not\subseteq e$. Since $f \subseteq gh + e$ and $f \not\subseteq e$, then $fgh \neq \emptyset$ implying that $fg \neq \emptyset$. Therefore, by the previous Lemma, g is a Boolean divisor of f. ∎

LEMMA 12 *If g is an algebraic factor of F a cover of $\mathcal{F}$, then g is a Boolean factor of $\mathcal{F}$.*

Proof. Assume $F = gh$. Since $f \subseteq F$, then

$$f \subseteq gh \Rightarrow f \subseteq g.$$

By a previous lemma, g is a Boolean factor of $\mathcal{F}$. ∎

Algorithm for Boolean Division

Given $\mathcal{F} = (f, d, r)$, write a cover for $\mathcal{F}$ in the form $gh + e$ where h, e are minimal in some sense.

Minimal may be minimum factored form.

An algorithm:

1. Create a **new** variable x to "represent" g.

2. Form the don't care set $\widetilde{d} = xg' + x'g$. (Since $x = g$ we don't care if $x \neq g$).

3. Minimize $(f\widetilde{d}', d + \widetilde{d}, r\widetilde{d}')$ to get $\widetilde{f}$.

4. Return $(h = \widetilde{f}/x, e)$ where e is the remainder of $\widetilde{f}$: these are simply the terms not containing x.

Here we are using f/x to denote "weak division" a maximal form of algebraic division. This will be defined later.

Example: Boolean Division

- Note that $(f\widetilde{d}', d + \widetilde{d}, r\widetilde{d}')$ is a partition. We can use ESPRESSO to minimize it. But the objective there is to minimize **number of cubes** - not completely appropriate.

- Example:
  $f = a + bc$
  $g = a + b$
  $\widetilde{d} = xa'b' + x'(a + b)$ where $x = g = (a + b)$

- Minimize $(a + bc)\widetilde{d}' = (a + bc)(x'a'b' + x(a + b)) = xa + xbc$ with DC $= xa'b' + x'(a + b)$.

- A minimum cover is $a + bc$ but it does not use x or x'!!

- Force x in the cover. It yields $f = a + xc = a + (a + b)c$.

- Heuristic: Try to find answer **with** x **in it** and which uses the **least variables** (or literals).

Two Algorithms for Boolean Division

Assume F is a cover for $\mathcal{F} = (f, d, r)$ and D is a cover for d.

First Algorithm:
$(H, E) \leftarrow$ Boolean-Division(F, D, g)

$$
\begin{array}{rcl}
D_1 & = & D + xg' + x'g \;\; \text{(don't care)} \\
F_1 & = & FD_1' \;\; \text{(on-set)} \\
R_1 & = & (F_1 + D_1)' = F_1' D_1' = F' D_1' \;\; \text{(off-set)} \\
F_2 & = & \text{remove } x' \text{ from } F_1. \\
F_3 & = & \text{Minimum\_Literal}(F_2, R_1, x) \\
 & /* & \text{(minimum literal support including } x) */ \\
F_4 & = & \text{ESPRESSO}(F_3, D_1, R_1) \\
H & = & F_4/x \;\; \text{(quotient)} \\
E & = & F_4 - \{H\} \;\; \text{(remainder)}
\end{array}
$$

Thus $GH + E$ is a cover for (f, d, r).

Two Algorithms for Boolean Division

Assume F is a cover for $\mathcal{F} = (f, d, r)$ and D is a cover for d.

Second Algorithm:
This is a slight variation of the first one. It uses x' also while dividing.

$$(H_1, H_0, e) \leftarrow \text{Boolean-Division}(F, D, g)$$

Thus we obtaine a cover $F = xH_1 + x'H_0 + E$.

$$
\begin{aligned}
D_1 &= D + xg' + x'g \text{ (don't care)} \\
F_1 &= FD_1' \text{ (on-set)} \\
R_1 &= (F_1 + D_1)' = F_1'D_1' = F'D_1' \text{ (off-set)} \\
/ * F_2 &= \text{remove } x' \text{ from } F_1 * / \\
F_3 &= \text{Minimum\_Literal}(F_1, R_1, x, x') \\
&/* \text{ (minimum literal support including } x, x') * / \\
F_4 &= \text{ESPRESSO}(F_3, D_1, r_1) \\
H_1 &= F_4/x \\
H_0 &= F_4/x' \\
E &= F_4 - \{H_1\} - \{H_0\}
\end{aligned}
$$

Thus $GH_1 + G'H_0 + E$ is a cover for (f, d, r).

Minimum_Literal()

Given $\mathcal{F} = (f, d, r)$, find a cover which has the smallest variable support (literal support).

Let v_sup = variable support, ℓ_sup = literal support, F = some cover of $\mathcal{F}$.

$$\begin{aligned} v\_sup(F) &= \{v | v \in c \text{ or } v' \in c \text{ for some } c \in F\} \\ \ell\_sup(F) &= \{\ell | \ell \in c \text{ for some } c \in F\} \end{aligned}$$

Let

$$v\_sup(\mathcal{F}) = \min\{v\_sup(F) | F \text{ a cover of } \mathcal{F}\}$$

Let

$$\ell\_sup(\mathcal{F}) = \min\{\ell\_sup(F) | F \text{ a cover of } \mathcal{F}\}$$

LEMMA 13 *If $d = \emptyset$, the $v\_sup(F) = v\_sup(\mathcal{F})$ and $\ell\_sup(F) = \ell\_sup(\mathcal{F})$ for any prime cover F of $\mathcal{F}$.*

Proof. Suppose F_1 and F_2 be two prime covers of $\mathcal{F}$. Let F_1 be independent of x, but x be in F_2. Let $xc \in F_2$. Let $c = m_1 + m_2 + \ldots m_k$. Since xm_i is an implicant of $\mathcal{F}$, it is present in F_1. However, F_1 is independent of x. Hence $x'm_i$ is also an implicant of $\mathcal{F}$. Hence m_i is an implicant $\mathcal{F}$ for all i. So xc can be raised to c, contradicting the fact that xc is a prime. ∎

Minimum Variable Algorithm (MINVAR)

Given
$\mathcal{F} = (f, d, r)$
$F = \{c^1, c^2, \ldots, c^k\}$ (a cover of $\mathcal{F}$)
$F = \{r^1, r^2, \ldots, r^m\}$ (a cover of r)

1. Construct blocking matrix B^i for each c^i.

2. Form "super" blocking matrix

$$B = \begin{bmatrix} B^1 \\ B^2 \\ \vdots \\ B^k \end{bmatrix}$$

3. Find a minimum cover S of B, $S = \{j_1, j_2, \ldots, j_v\}$.

4. Modify $\widetilde{F} \leftarrow \{\widetilde{c}^1, \widetilde{c}^2, \ldots, \widetilde{c}^k\}$ where

$$(\widetilde{c}^i)_j = \begin{cases} (c^i)_j & \text{if } j \in S \\ \{0, 1\} = 2 & \text{otherwise} \end{cases}$$

Minimum Variable Algorithm

THEOREM 14 *The set $\{x^j | j \in S\}$ is a minimum set of variables which appear in any cover of $\mathcal{F}$ obtained by expanding F*

Proof. Expand treats each $c^i \in F$ and builds B^i. Let $\widetilde{c}^i$ be any prime containing c^i. Then the variables in $\widetilde{c}^i$ "cover" B^i. Thus the union of the set of variables in $\widetilde{c}^i$ taken over all i cover B. Hence this set cannot be smaller than a minimum cover of B. ∎

Note: In general, there could exist another cover of $\mathcal{F}$ which has less variables, but such could not be one which can be obtained by expanding F

Minimum Literal Support

Given
$\mathcal{F} = (f, d, r)$
$F = \{c^1, c^2, \ldots, c^k\}$ (a cover of $\mathcal{F}$)
$R = \{r^1, r^2, \ldots, r^m\}$ (a cover of r)

Literal Blocking Matrix

$$(\hat{B}^i)_{qj} = \begin{cases} 1 & \text{if } v_j \in c^i \text{ and } v'_j \in r^q \\ 0 & \text{otherwise} \end{cases}$$

$$(\hat{B}^i)_{q,j+n} = \begin{cases} 1 & \text{if } v'_j \in c^i \text{ and } v_j \in r^q \\ 0 & \text{otherwise} \end{cases}$$

Example: $c^i = ad'e', \quad r^q = a'ce.$

$$\begin{array}{cccccccccccc} & a & b & c & d & e & a' & b' & c' & d' & e' \\ \hat{B}^i_q = & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

- Use the same method - construct "super" blocking matrix and get its row cover J.

THEOREM 15 *If x is a row cover of $\hat{B}$, then a representation (expression or factored form) exists using only the literals $\{\ell_i i \in J\} \cup \{\ell'_i | i + n \in J\}$*

Proof. Left as an exercise. ∎

Example of Blocking Matrix

---

on-set cube $c^i = ab'd$.
off-set $r = a'b'd' + abd' + acd' + bcd + c'd'$.

	a	b	c	d	a'	b'	c'	d'
$a'b'd'$	1	0	0	1	0	0	0	0
abd'	0	0	0	1	0	1	0	0
acd'	0	0	0	1	0	0	0	0
bcd	0	0	0	0	0	1	0	0
$c'd'$	0	0	0	1	0	0	0	0

Minimum row cover = $\{d, b'\}$.
Thus $b'd$ is the maximum prime covering $ab'd$.

Note: For one cube, minimum literal support is the same as minimum variable support.

Boolean Division: Example

$F = a + bc$
algebraic division: $F/(a+b) = \emptyset$
Boolean division: $F \div (a+b) = a + c$.

- Let $x = a + b$.
- Generate don't care set: $D_1 = x'(a+b) + xa'b'$.
- Generate care on-set: $F_1 = F \cap D_1' = (a+bc)(xa + xb + x'a'b') = xa + bcx$. Let $\mathcal{C} = \{c^1 = ax, c^2 = bcx\}$.
- Generate care off-set: $R_1 = F'D_1' = (a'b' + a'c')(xa + xb + x'a'b') = a'bc'x + a'b'x'$. Let $R = \{r^1 = a'bc'x, r^2 = a'b'x'\}$.
- Form super-variable blocking matrix using column order (a, b, c, x).

$$B = \begin{bmatrix} B^1 \\ B^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ - & - & - & - \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{1}$$

- Find minimum row cover $= \{a, c, x\}$.

- Eliminate in F_1 all variables associated with b. So $F_1 = ax + bcx = ax + cx = x(a + c)$.

- Simplifying (applying expand, irredundant on) F_1, we get $F_1 = a + xc$.

- Thus quotient $= F_1/x = c$, remainder $= a$.

- $F = a + bc = a + cx = a + c(a + b)$.

Question: How to force x in the cover?

Algebraic Division

Algebraic division, Alg_Div, is any operation, given F, G, returns H, R such that GH is an algebraic product and $GH + R$ and F are the same expression (having the same set of cubes).

DEFINITION 16 **Weak division** *is a specific example of algebraic division. Given two algebraic expressions F and G, a division is called* **weak division** *if*

1. *it is algebraic and*
2. *R has as few cubes as possible.*

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

THEOREM 17 *Given expressions F and G, H and R generated by weak division are unique.*

Algorithm

WEAK_DIV(F,G):
$U = \{u_j\}$ - all cubes of F but where only literals in G have been kept.
$V = \{v_j\}$ - all cubes of F but literals in G removed.
/* note that $u_j v_j$ is the j-th cube of F */
$V^i = \{ \ v_j \in \ V \ \ : \ \ u_j = G_i \ \}$.
/* one set for each cube of G */
$H = \cap \ V^i$. /* those cubes found in all V_i */
$R = F - GH$
return (H, R)
Note: Time complexity of WEAK_DIV $= O(n \log n)$, n = number of product terms in F and G.
Example

$$
\begin{aligned}
F &= ace + ade + bc + bd + be + a'b + ab \\
G &= ae + b \\
U &= ae + ae + b + b + b + b + ab \\
V &= c + d + c + d + 1 + a' + 1 \\
V^{ae} &= c + d \\
V^{b} &= c + d + 1 + a' \\
H &= c + d = F/G \\
R &= be + a'b + ab \\
F &= (ae + b)(c + d) + be + a'b + ab
\end{aligned}
$$

Complexity of Algebraic Operations

What if F, G were already given as sorted cubes, i.e. in the order of their binary encoding?

$ab'de =$ 01 10 11 01 01 (binary number)

Can we find a **linear** time algorithm for computing F/G and R produced in sorted order?

Answer: Yes (see McGeer and Brayton VLSI'87)
In fact, the operations, algebraic division, multiplication, addition, subtraction, and equality test are all linear and **stable.**

Definition: A **stable** algorithm receives its input in sorted order and produces its outputs in sorted order.

If all algorithms are stable, then we can start with a Boolean network, do an initial sort on each node, and then use only stable operations.

Note: This is not implemented in MIS since most functions F and G are small.

Efficiency Issues

We use filters to prevent trying a division. The cover G is not an algebraic divisor of F if

1. G contains a literal not in F.

2. G has more terms than F.

3. For any literal, its count in G exceeds that in F.

4. F is in the transitive fanin of G.

Proof. If G were an algebraic divisor of F, $F = GH + E$. ∎

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

Notice that a cube-free expression must have more than one cube.

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

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

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

$$\mathcal{K}(F) = \{G \mid G \in \mathcal{D}(F) \text{ and } G \text{ is cubefree } \}.$$

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, and $\mathcal{C}(F)$ is used to denote the set of co-kernels of F.*

Example

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

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

Fundamental Theorem

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

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

(k_G and k_F have at most one term in common), then F and G have **no common algebraic divisors with more than one term.**

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 $\mathcal{K}(F)$. This leads to the definition for the **level** of a kernel:

$$\mathcal{K}^n(F) = \begin{cases} \{k \in \mathcal{K}(F) \mid \mathcal{K}(k) = \{k\}\} & n = 0 \\ \{k \in \mathcal{K}(F) \mid \forall k_1 \in \mathcal{K}(k) & n > 0 \\ \qquad k_1 \neq k \text{ such that } k_1 \in \mathcal{K}^{n-1}(F)\} & \end{cases}$$

Note: a kernel is level-0 if it has no kernels except itself. Similarly, a kernel is of level n if it has at least one level n-1 kernel but no kernels (except itself) of level n or greater. Thus

$$\mathcal{K}^0(F) \subset \mathcal{K}^1(F) \subset \mathcal{K}^2(F) \subset \cdots \subset \mathcal{K}^n(F) \subset \mathcal{K}(F).$$

Example:

$$\begin{aligned} F &= (a + b(c + d))(e + g) \\ k_1 &= a + b(c + d) \in \mathcal{K}^1 \\ &\qquad\qquad \notin \mathcal{K}^0 \\ k_2 &= c + d \in \mathcal{K}^0 \end{aligned}$$

Kerneling Algorithm

$R \Leftarrow$ KERNEL(j, G)

$R \Leftarrow \emptyset$
if (G is cube-free) $R \Leftarrow \{G\}$
For $i = j + 1, ..., n$ {
    If ℓ_i appears only in one term, continue
    $c \Leftarrow$ largest cube dividing G/ℓ_i evenly
    If ($\ell_k \in c$, for some $k \leq i$), continue
    else $R \Leftarrow R \cup$ KERNEL$(i, G/(\ell_i c))$
}
return R

So KERNEL$(0, F)$ returns all the kernels.

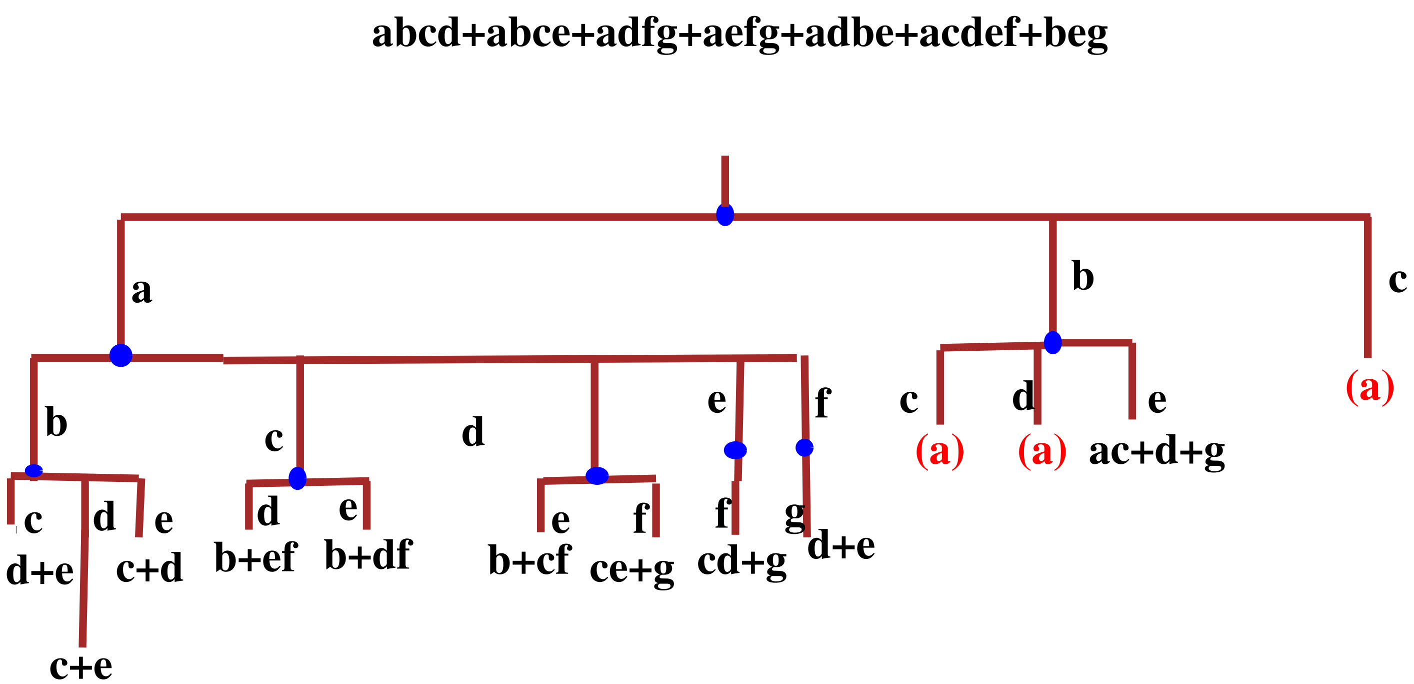
Notes:

1. The test $(l_k \in c)$ 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



co-kernel | **kernel**

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

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

Applications - Factoring

$FACTOR(F)$:

1. if F has no factor, return F
   /* e.g. if $|F| = 1$, or an OR of literals*/

2. $D = CHOOSE\_DIVISOR(F)$

3. $(Q, R) = DIVIDE(F, D)$

4. return $FACTOR(Q)FACTOR(D)+FACTOR(R)$

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 1

$$
\begin{aligned}
F &= abc + abd + ae + af + g \\
D &= c + d \\
Q &= ab \\
P &= ab(c + d) + ae + af + g \\
O &= ab(c + d) + a(e + f) + g
\end{aligned}
$$

O is not optimal since not maximally factored. Can be further factored to

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

The problem occurs when quotient Q is a single cube, and 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 the cube which occurs in the greatest number of cubes of F.

3. Divide F by ℓ_1 to obtain a new divisor D_1. Now, F has a new partial factored form

$$(\ell_1)(D_1) + (R_1)$$

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

$$\begin{aligned}
F &= ace + ade + bce + bde + cf + df \\
D &= a + b \\
Q &= ce + de \\
P &= (ce + de)(a + b) + (c + d)f \\
O &= e(c + d)(a + b) + (c + d)f
\end{aligned}$$

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

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

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 $D_1 = cD_2$. Let $D_3 = Q_1 D_2$. We have the partial factoring $F = cD_3 + R_1$. Now recursively factor D_3 and R_1.

Improving Hastily Chosen Divisors

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

LF(F, C, DIVISOR, DIVIDE)
    L = best_literal(F, C) /* most common */
    (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 a literal, or the best literal
- QUICK_DIVISOR - chooses one level-0 kernel
- BEST_DIVISOR - chooses the best kernel

DIVIDE:

- Algebraic Division
- Boolean Division

QUICK_FACTOR

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

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

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

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

QUICK_FACTOR uses GFACTOR, level-0 kernel DIVISOR, and WEAK_DIV.

$x = (a + b(c + d))(e + f(g + h))$
$D = c + d$ —— level-0 kernel
$Q = x/D = b(e + f(g + h))$ —– weak division
$Q = e + f(g + h)$ —– make cube-free
$(D, R) = WEAK\_DIV(x, Q)$ —– second division
$D = a + b(c + d)$
$x = DQ + R = (a + b(c + d))(e + f(g + h))$
$R = \emptyset$

Application - Decomposition

Recall: decomposition is the same as factoring except:

- divisors are added as new nodes in the network.
- the new nodes may fan out elsewhere in the network in positive or negative phases.

$DECOMP(f_i)$
  $k = CHOOSE\_KERNEL(f_i)$
  if ($k == 0$) return
  $f_{m+\ell} = k$ /* create new node $m + \ell$ */
  $f_i = (f_i/k)y_{m+\ell} + (f_i/k')y'_{m+\ell} + r$
  $DECOMP(f_i)$
  $DECOMP(f_{m+\ell})$

Similar to factoring, we can define

- QUICK_DECOMP: pick a level 0 kernel.
- GOOD_DECOMP: pick the best kernel.

decomp

```
decomp [-gqd] [node-list]
```

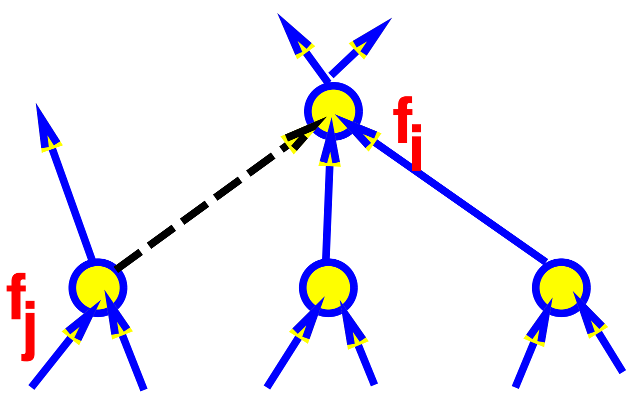
Decompose all nodes in the node-list. If the node-list not specified, all nodes in network will be decomposed.

If the -q option (the default) is specified, the quick decomp algorithm is used which extracts out an arbitrary kernel successively. Because of the fast algorithm for generating an arbitrary kernel, decomp -q is very fast compared with the decomp -g. In most cases, the result is very close. This command is recommanded at the early phase of the optimization.

If the -g option is specified, 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 optimiza- tion, should decomp -g be used.

If the -d option is specified, 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



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

Algebraic substitution consists of the process of 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

$$f_i = qy_j + r;$$

Similarly for f'_j.

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

Boolean (Re)substitution - Example

Substituting x into F.

$$\begin{aligned}
x &= ab + cd + e \\
F &= abf + a'cd + cdf + a'de + ef \\
Don'tcareset &: \\
D &= x'(cab + cd + e) + x(ab + cd + e)' \\
F_1 &= xef + xa'cd + xa'de + xabf + xcdf \\
R_1 &= abf'x + aef'x + d'ef'x + a'c'e'x' + \\
&\quad b'c'e'x' + a'd'e'x' + b'd'e'x' + acdf'x
\end{aligned}$$

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

$$\begin{aligned}
F_3 &= xf + xa'd + xa'd + xaf + xdf \\
&= xf + xa'd \\
&= x(f + a'd)
\end{aligned}$$

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.

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

The -b option uses Boolean division when substitute one node into another.

Extraction

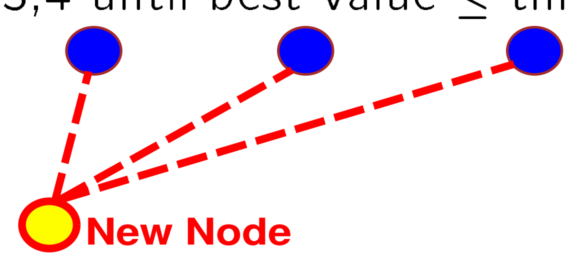
Recall: Extraction operation identifies common subexpressions and manipulates the Boolean network.

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

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

Kernel Extraction:

1. Find **all** kernels of **all** functions
2. Choose one 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 $\leq$ threshhold



Example - Extraction

$$
\begin{aligned}
f_1 &= ab(c(d+e)+f+g)+h \\
f_2 &= ai(c(d+e)+f+j)+k
\end{aligned}
$$

extraction $\mathcal{K}^0(f_1) = \mathcal{K}^0(f_2) = \{d+e\}$

$$
\begin{aligned}
l &= d+e \\
f_1 &= ab(cl+f+g)+h \\
f_2 &= ai(cl+f+j)+k
\end{aligned}
$$

extraction $\mathcal{K}^0(f_1) = \{cl+f+g\}$
$\mathcal{K}^0(f_2) = \{cl+f+j\}$
$\mathcal{K}^0(f_1) \cap \mathcal{K}^0(f_2) = cl+f$

$$
\begin{aligned}
m &= cl+f \\
f_1 &= ab(m+g)+h \\
f_2 &= ai(m+j)+k
\end{aligned}
$$

Note: no kernel intersections at this point.
cube extraction

$$
\begin{aligned}
n &= am \\
f_1 &= b(n+ag)+h \\
f_2 &= i(n+aj)+k
\end{aligned}
$$

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 *
```

Fast 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 kernels

- 2-literal cube divisors

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

- $ab + a'b', b' + c$ and $ab + a'c$ are 2-cube kernels.

- $a'd$ 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 or 2-literal cube divisors.

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

Generating all double-cube divisors

$F = \{c_i\}$
$\mathcal{D}(F) = \{d \mid d = \text{make\_cube\_free}(c_i + c_j)\}$
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 $\mathcal{O}(n^2)$,
where $n =$ #cubes in F

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

make_cube_free$(axe + ag) = xe + g$
make_cube_free$(axe + bcxe) = a + bc$
make_cube_free$(axe + bcg) = axe + bcg$
make_cube_free$(ag + bcxe) = ag + bcxe$

Note: the functions are made algebraic expressions before generating double-cube divisors. Also, note that not all 2-cube divisors are kernels.

Key result for double-cube divisors

THEOREM 23 *Expressions F and G have a common multiple-cube divisor if and only if $\mathcal{D}(F) \cap \mathcal{D}(G) \neq 0$.*

Proof. **If part:** If $\mathcal{D}(F) \cap \mathcal{D}(G) \neq 0$ then $\exists d \in \mathcal{D}(F) \cap \mathcal{D}(G)$ 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 $C = \{c_1, c_2, \ldots, c_m\}$ 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 $e \in \mathcal{D}(F) \cap \mathcal{D}(G)$. If e is not cube-free, then let $\widetilde{e} =$ make_cube_free$(c_i + c_j)$. $\widetilde{e}$ exists since F and G are algebraic expressions. Hence $\widetilde{e} \in \mathcal{D}(F) \cap \mathcal{D}(G)$. ∎

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

If $e = ac + d$, e is cube-free and $e \in \mathcal{D}(F) \cap \mathcal{D}(G)$

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

As a result of the Theorem, all multiple-cube divisors can be "discovered" from double-cube divisors.

Fast Kernel Extraction

Algorithm:

- Generate and store all 2-cube kernels and recognize complement divisors.
- Find the best 2-cube kernel or 2-literal cube divisor at each stage and extract it.
- Update 2-cube kernel set after extraction
- Iteratively extract divisors until no more improvement.

Result:

- Much faster.
- Quality as good as that of kernel extraction.
- On a set of examples: fast extract gives 9563 literals, general kernel extraction 9732, and is 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 cost function to each node,
and extracts the node with the best cost function greedily.

The -o option only looks for 0-level two-cube divisors.

The -b option reads an upper bound for the number of divisors
generated.

The -l option changes the level of each node in the network
as allowed by the slack between the required time and
arrival time at that node.

The -z option uses zero-weight 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 take an exhorbitant 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
```