Outline

Division is central in many operations.
- What is it (in the context of Boolean functions)?
- What to divide (divisor) with?
- How to divide (quotient and remainder)?

Applications: factoring, resubstitution, extraction
Product

Definition 1: An algebraic expression is a SOP representation of a logic function which is minimal w.r.t. 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 \( \Sigma c_id_j \) where \( \{c_i\} = f \), \( \{d_j\} = g \), made irredundant w.r.t. single cube containment 
\( e.g. \ ab + a = a \)
Product

Algebraic product: defined only 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 \( f = gh + r \), \( gh \neq 0 \).

\( g \) is said to be a factor of \( f \) if, in addition, \( r = 0 \), 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).
Division (f = gh+r)

If h ≠ 0, and h can be obtained using algebraic division, then g is an algebraic divisor of f. Otherwise, g is a Boolean divisor of f.

Example:
\[ f = ad + ae + bcd + j \]
\[ g_1 = a + bc \]
\[ g_2 = a + b \]

- 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. \(f = (a+b)(a+c)d + ae + j\)
Division

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. $fg' = 0$, i.e. $g' \subseteq f'$).
Division

Proof:
⇒: g is a Boolean factor of f. Then ∃h such that
f = gh; Hence, f ⊆ g (as well as h).

⇐: f ⊆ g ⇒ f = gf = g(f + r) = gh.
(Here r is any function r ⊆ g').

Notes:
• h = f works fine for the proof.
• Given f and g, h is not unique.
• To get a small h is the same as getting a small f + r.
  Since rg = 0, 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 $fg \neq 0$. 
Division

Proof:
\[ \Rightarrow: f = gh + r, \; gh \neq 0 \Rightarrow fg = gh + gr. \; \text{Since } gh \neq 0, \; fg \neq 0. \]

\[ \Leftarrow: \text{Assume that } fg \neq 0. \; f = fg + fg' = g(f + k) + fg'. \]
\[ \text{(Here } k \subseteq g'.) \; \text{Then } f = gh + r, \; \text{with } h = f + k, \; r = fg'. \]
\[ \text{Since } gh = fg \neq 0, \; \text{then } gh \neq 0. \]

Note: \( f \) has many divisors. We are looking for a \( g \)
such that \( f = gh + r \), where \( g, \; h, \; r \) are simple
functions. (simplify \( f \) with \( DC = g' \))
Incompletely Specified Functions

\[ F = (f, d, r) \]

Definition 7: A completely specified logic function \( g \) is a Boolean divisor of \( F \) 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 \( F \) if there exists \( h \) such that
\[ f \subseteq gh \subseteq f + d \]
Incompletely Specified Functions

Lemma 9: \( f \leq g \) if and only if \( g \) is a Boolean factor of \( F \).

Proof.

\[ \Rightarrow: \text{Assume that } f \leq g. \text{ Let } h = f + k \text{ where } kg \leq d. \text{ Then } \]
\[ h g = (f + k) g \leq (f + d). \text{ Since } f \leq g, \ f g = f \text{ and thus } \]
\[ f \leq (f + k) g = gh. \]
Thus
\[ f \leq (f + k) g \leq f + d \]

\[ \Leftarrow: \text{Assume the } f = gh. \text{ Suppose } \exists \text{ minterm } m \text{ such that } f(m) = 1 \]
\[ \text{but } g(m) = 0. \text{ Then } f(m) = 1 \text{ but } g(m)h(m) = 0 \text{ implying that } \]
\[ f \not\leq gh. \text{ Thus } f(m) = 1 \text{ implies } g(m) = 1, \text{ i.e. } f \leq g \]

Note: Since \( kg \leq d, k \leq (d + g'). \text{ Hence obtain } \]
\[ h = f + k \text{ by simplifying } f \text{ with } DC = (d + g'). \]
Incompletely Specified Functions

Lemma 10: \( fg \neq 0 \) if and only if \( g \) is a Boolean divisor of \( F \).

Proof.

\( \Rightarrow \): Assume \( fg \neq 0 \). 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, \( 0 \neq fg \subseteq gh \rightarrow ghf \neq 0 \). Now \( gh \not\subset d \), since otherwise \( ghf = 0 \) (since \( fd = 0 \)), verifying the conditions of Boolean division.

\( \Leftarrow \): Assume that \( g \) is a Boolean divisor. Then \( \exists h \) such that \( gh \not\subset d \) and

\[ f \subseteq gh + e \subseteq f + d \]

Since \( gh = (ghf + ghd) \not\subset d \), then \( fgh \neq 0 \) implying that \( fg \neq 0 \).
Incompletely Specified Functions

\[ f g \subseteq h \subseteq (f + d + g') \quad \text{and} \quad f g' \subseteq e \subseteq (f + d) \]

Recipe: \(( f \subseteq gh + e \subseteq f + d )\)

- Choose \( g \) such that \( fg \neq 0 \).
- Simplify \( fg \) with \( DC = (d + g') \) to get \( h \).
- Simplify \( fg' \) with \( DC = (d + fg) \) to get \( e \). (could use \( DC = d + gh \) )

Thus

\[ f g \subseteq h \subseteq f + g' + d \]
\[ f g' \subseteq e \subseteq f g' + d + f g = f + d \]
Incompletely Specified Functions

\( F = (f, d, r) \)

Lemma 11. Suppose \( g \) is an algebraic divisor of \( F \), a cover of \( F \). If \( f \not\subset e \) (where \( e \) is the remainder in the algebraic division, i.e. \( F = gh + e \)) then \( g \) is a Boolean divisor of \( F \).

Proof. Assume \( F = gh + e \), \( gh \neq 0 \), \( f \not\subset e \). Since \( f \subset gh + e \) and \( f \not\subset e \), then \( fgh \neq 0 \) implying that \( fg \neq 0 \). Therefore, by the Lemma 10, \( g \) is a Boolean divisor of \( f \).

Lemma 12. If \( g \) is an algebraic factor of \( F \), a cover of \( F \), then \( g \) is a Boolean factor of \( F \).

Proof. Assume \( F = gh \). Since \( f \subset F \), then 
\[
f \subset gh \Rightarrow f \subset g.
\]
By Lemma 9, \( g \) is a Boolean factor of \( F \).
Algorithm for Boolean Division

Given $F = (f, d, r)$, write a cover for $F$ in the form $gh + e$ where
$h, e$ are minimal in some sense.
Minimal may be minimum factored form.

An algorithm:

- Create a new variable $x$ to “represent” $g$.
- Form the don’t care set $	ilde{d} = \chi g' + x'g$. (Since $\chi = g$ we don’t care if $\chi \neq g$).
- Minimize $(f \tilde{d}', d + \tilde{d}, r \tilde{d}')$ to get $\tilde{f}$.
- Return $(h = \tilde{f}/x, e)$ where $e$ is the remainder of $\tilde{f}$. (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.
Algorithm for Boolean Division

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

Example:
\[
\begin{align*}
    f &= a + bc \\
    g &= a + b
\end{align*}
\]

\[
\tilde{d} = xa'b' + x'(a+b)\text{ where } x = g = (a+b)
\]
- Minimize \((a + bc) \tilde{d}' = (a + bc)(x'a'b' + x(a+b)) = xa + xbc\) 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. This yields \(f = a + xc = a + (a + b)c\)

Heuristic: Try to find answer with \(x\) in it and which also uses the least variables (or literals)
Two Algorithms for Boolean Division

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

First Algorithm:

$$(H, E) \leftarrow \text{Boolean-Division}(F, D, g)$$

$D_1 = D + xg' + x'g$ (don't care)

$F_1 = FD'_1$ (on-set)

$R_1 = (F_1 + D_1)' = F'_1 D'_1 = F'D'_1$ (off-set)

$F_2 = \text{remove } x' \text{ from } F_1$

$F_3 = \text{Minimum Literal}(F_2, R_1, x)$

/* (minimum literal support including $x$) */

$F_4 = \text{ESPRESSO}(F_3, D_1, R_1)$

$H = F_4/x$ (quotient)

$E = F_4 - \{xH\}$ (remainder)

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

19
Two Algorithms for Boolean Division

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

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

$$(H1, H0, e) \leftarrow \text{Boolean-Division}(F, D, g)$$

Thus we obtain a cover $F = xH1 + x' H0 + E$
Two Algorithms for Boolean Division

Second Algorithm:

\[ D_1 = D + xg' + x'g \]  (don't care)
\[ F_1 = FD'_1 \]  (on-set)
\[ R_1 = (F_1 + D_1)' = F'_1D'_1 = F'D'_1 \]  (off-set)
\[ /* F_2 = remove x' from F_1 */ \]  (this line is deleted)
\[ F_3 = \text{Minimum Literal}(F_1, R_1, x, x') \]
\[ /* (minimum literal support including} x}) */
\[ F_4 = \text{ESPRESSO}(F_3, D_1, R_1) \]
\[ H_1 = F_4/x \]
\[ H_0 = F_4/x' \]
\[ E = F_4 \setminus (\{xH_1\} + (x'H_0)) \]

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

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

Definitions: minimum supports for $F = $ some cover of $F$

$v\_sup(F) = \{ v | v \in c \text{ or } v' \in c \text{ for some } c \in F\}$

$l\_sup(F) = \{ l | l \in c \text{ for some } c \in F\}$

Definitions: minimum supports for $F$

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

$l\_sup(F) = \min\{ l\_sup(F) | F \text{ a cover of } F\}$
**Minimum_Literal()**

\[ F = (f,d,r), \quad F \text{ is any prime cover} \]

Lemma 13: If \( d = 0 \), then \( \nu_{\text{sup}} (F) = \nu_{\text{sup}} (F) \) and \( \nu_{\text{sup}} (F) = \nu_{\text{sup}} (F) \) for any prime cover \( F \) of \( F \).

Proof. Suppose \( F_1 \) and \( F_2 \) are two prime covers of \( F \). Suppose \( x \) appears in \( F_2 \) but not in \( F_1 \). Let \( xc \in F_2 \). Let \( c = m_1 + m_2 + \ldots + m_k \). Since \( d = 0 \) and \( xm_i \) is an implicant of \( F \), it is present in \( F_1 \). However, \( F_1 \) is independent of \( x \). Hence \( x'm_i \) is also an implicant of \( F \). Hence \( m_i \) is an implicant of \( F \) for all \( i \). So \( xc \) can be raised to \( c \), contradicting the fact that \( xc \) is prime.
Minimum Variable Algorithm (MINVAR)

Given:

\[ F = (f, d, r) \]
\[ F = \{c^1, c^2, \ldots, c^k\} \quad \text{(a cover of } F) \]
\[ R = \{r^1, r^2, \ldots, r^m\} \quad \text{(a cover of } r) \]

1. Construct blocking matrix \( B_i \) for each \( c^i \).
2. Form “super” blocking matrix

\[
B = \begin{bmatrix}
B_1^1 \\
B_2^2 \\
\vdots \\
B_M^m \\
B_3^3
\end{bmatrix}
\]
Minimum Variable Algorithm (MINVAR)

MINVAR continued

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

\[
(\widetilde{c}^i)_j = \begin{cases} 
(\widetilde{c}^i)_j & \text{if } j \in S \\
\{0,1\} = 2 & \text{otherwise}
\end{cases}
\]
End of lecture 9
**Minimum Variable Algorithm**

Theorem 14: The set \( \{x_j \mid j \in S\} \) is a minimum variable set in the sense that no other cover of \( F \), obtained by expanding \( F \), has fewer variables.

Proof. Expand treats each \( c^i \in F \) and builds \( B^i \). Let be any prime containing \( c^i \). Then the variables in “cover” \( B^i \). Thus the union of the set of variables in 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 \( F \) which has less variables, but that cover could not be obtained by expanding \( F \).
Minimum Literal Support

Given:
\[ F = (f, d, r) \]
\[ F = \{c^1, c^2, \ldots, c^k\} \quad \text{(a cover of } F) \]
\[ R = \{r^1, r^2, \ldots, r^m\} \quad \text{(a cover of } r) \]

Literal Blocking Matrix
\[
\begin{align*}
(\hat{B}^i)_{qi} &= \begin{cases} 
1 & \text{if } v_j \in c^i \text{ and } v'_j \in r^q \\
0 & \text{otherwise}
\end{cases} \\
(\hat{B}^i)_{q,j+n} &= \begin{cases} 
1 & \text{if } v'_j \in c^i \text{ and } v_j \in r^q \\
0 & \text{otherwise}
\end{cases}
\end{align*}
\]

Example: \( c^i = ad'e' \), \( r^q = a'ce \)

\[
\hat{B}^i_q = \begin{bmatrix} a & b & c & d & e & a'b'c'd'e' \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}
\]
Minimum Literal Support

Use the same method - construct “super” blocking matrix and get its row cover \( J \).

Theorem 15: If \( x \) is a row cover of \( \hat{a} \), then a representation (expression or factored form) exists using only the literals
\[
\{ \overline{i} \mid i \in J \} \cup \{ \overline{i} \mid i + n \in J \}
\]

Proof. Left as an exercise.
Example of Literal Blocking Matrix

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>a'</th>
<th>b'</th>
<th>c'</th>
<th>d'</th>
</tr>
</thead>
<tbody>
<tr>
<td>a'b'd'</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>abd'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>acd'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bcd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c'd'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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: \( F / (a + b) = 0 \)

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:
\[ F_1 = F \cap D'_1 = (a + bc)(xa + xb + x'a'b') = ax + bcx. \]
Let \( C = \{c^1 = ax, c^2 = bcx\}\)

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 \((a, b, c, x)\).

\[
B = \begin{bmatrix}
B^1 \\
B^2
\end{bmatrix} = 
\begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1
\end{bmatrix}
\]
Boolean Division: Example

\[
B = \begin{bmatrix}
  B^1 \\
  B^2
\end{bmatrix} = \begin{bmatrix}
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

- Find minimum 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 where

- GH is an algebraic product and
- GH + R and F are the same expression (having the same set of cubes).

**Weak division** is a specific example of algebraic division.

**DEFINITION 16:** 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 17:** Given expressions F and G, H and R generated by weak division are unique.
Algorithm

WEAK_DIV(F,G): \( G=\{ \, g_1, g_2, \ldots, \, \} \)
- \( U = \{u_j\} \) - cubes of \( F \) but only literals in \( G \) kept.
- \( V = \{v_j\} \) - cubes of \( F \) but literals in \( G \) removed.
  /* note that \( u_j\) is the \( j \)-th cube of \( F \) */
- \( V^g_i = \{ v_j \in V : u_j = g_i \} \)
  /* one set for each cube of \( G \) */
- \( H = \bigcap V^g_i \) /* those cubes found in all \( V^g_i \) */
- \( R = F \setminus GH \)
- return \( (H,R) \)

Note: Time complexity of \( \text{WEAK}_\text{DIV} = O(n \log n) \),
n = number of product terms in \( F \) and \( G \).
Example of \textsc{Weak\_Div}

Example:
\[
\begin{align*}
F &= ace + ade + bc + bd + be + \text{a'\text{b}} + \text{ab} \\
G &= ae + b \\
U &= ae + ae + b + b + b + b + \text{ab} \\
V &= c + d + c + d + 1 + \text{a'} + 1 \\
V^{ae} &= c + d \\
V^{b} &= c + d + 1 + \text{a'} \\
H &= c + d = \frac{F}{G} \\
R &= be + \text{a'\text{b}} + \text{ab} \\
F &= (ae + b)(c + d) + be + \text{a'\text{b}} + \text{ab}
\end{align*}
\]
Efficiency Issues

We use filters to prevent trying a division.

The cover $G$ 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 $G$.

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 \textit{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 \textit{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:
\[ x = a df + a ef + b df + b ef + c df + c ef + g \]
\[ = (a + b + c)(d + e)f + g \]

<table>
<thead>
<tr>
<th>kernels</th>
<th>co-kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+c</td>
<td>df, ef</td>
</tr>
<tr>
<td>d+e</td>
<td>af, bf, cf</td>
</tr>
<tr>
<td>(a+b+c)(d+e)f+g</td>
<td>1</td>
</tr>
</tbody>
</table>
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_F and k_G 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:

$$K^n(F) = \begin{cases} 
(n = 0) & \Rightarrow \{ k \in K(F) \mid K(k) = \{k\} \} \\
(n > 0) & \Rightarrow \{ k \in K(F) \mid \forall k_1 \in K(k), (k_1 \neq k) \Rightarrow k_1 \in K^{n-1}(F) \} 
\end{cases}$$

Notes:

- $K^0(F) \subset K^1(F) \subset K^2(F) \subset \ldots \subset K^n(F) \subset K(F)$.
- level-$n$ kernels $= K^n(F) \setminus K^{n-1}(F)$
- $K^n(F)$ is the set of kernels of level $k$ or less.
- A level-0 kernel has no kernels except itself.
- 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:

\[ F = (a + b(c + d))(e + g) \]

\[ k_1 = a + b(c + d) \in K^1 \quad \notin K^0 \implies \text{level-1} \]

\[ k_2 = c + d \in K^0 \]

\[ k_3 = e + g \in K^0 \]
Kerneling Algorithm

\[
R \leftarrow \text{KERNEL}(j, G)
\]
\[
R \leftarrow 0
\]
\[
\text{if (G is cube-free) } R \leftarrow \{G\}
\]
\[
\text{For } i = j + 1, \ldots, n \{
\]
\[
\quad \text{if (l}_i \text{ appears only in one term) continue}
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{if (}\exists k \leq i, l_k \in \text{ all cubes of } G/l_i\text{), continue}
\]
\[
\quad \quad \text{else,}
\]
\[
\quad \quad \quad \quad R \leftarrow R \cup \text{KERNEL}(i, \text{cube_free}(G/l_i))
\]
\[
\}
\]
\[
\text{return } R
\]
Kerneling Algorithm

KERNEL(0, F) returns all the kernels of F.

Notes:

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

\[ a \cdot b \cdot c \cdot d + a \cdot b \cdot c \cdot e + a \cdot d \cdot f \cdot g + a \cdot e \cdot f \cdot g + a \cdot d \cdot b \cdot e + a \cdot c \cdot d \cdot e \cdot f + b \cdot e \cdot g \]

\[(bc + fg)(d + e) + de(b + cf)\]
**Kerneling Illustrated**

<table>
<thead>
<tr>
<th>co-kernels</th>
<th>kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a(((bc + fg)(d + e) + de(b + cf))) + beg</td>
</tr>
<tr>
<td>a</td>
<td>(bc + fg)(d + e) + de(b + cf)</td>
</tr>
<tr>
<td>ab</td>
<td>c(d+e) + de</td>
</tr>
<tr>
<td>abc</td>
<td>d + e</td>
</tr>
<tr>
<td>abd</td>
<td>c + e</td>
</tr>
<tr>
<td>abe</td>
<td>c + d</td>
</tr>
<tr>
<td>ac</td>
<td>b(d + e) + def</td>
</tr>
<tr>
<td>acd</td>
<td>b + ef</td>
</tr>
</tbody>
</table>

Note: \( f/bc = ad + ae = a(d + e) \)
Applications - Factoring

FACTOR (F):

- if (F has no factor) return F
  /* e.g. if |F| = 1, or an OR of literals
      or no literal appears more that once */
- D = CHOOSE_DIVISOR (F)
- (Q, R) = DIVIDE (F, D)
- 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 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
- quotient \( Q \) is a single cube, and
- some of the literals of \( Q \) also appear in the remainder \( R \).
Solving this Problem

- Check if the quotient $Q$ is not a single cube, then done, else,
- Pick a literal $/_{i}$ in $Q$ which occurs most frequently in cubes of $F$.
- Divide $F$ by $/_{i}$ to obtain a new divisor $D_{1}$.
  Now, $F$ has a new partial factored form
  \[(l_{1})(D_{1}) + (R_{1})\]
  and literal $/_{i}$ does not appear in $R_{1}$.

Note: the new divisor $D_{1}$ contains the original $D$ as a divisor because $/_{i}$ 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:

• Make $Q$ cube-free to get $Q_1$
• Obtain a new divisor $D_1$ by dividing $F$ by $Q_1$
• 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 $R_1$
• 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 $R_1$. 


End of lecture 10
Improving Vanilla Factoring

GFACCTOR(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 * 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
  • GFACTOR,
  • First level-0 kernel DIVISOR, and
  • WEAK_DIV.

\[ x = ae + afg + afh + bce + bcfg + bcfh + bde + bdfg + bcfh \]
\[ D = c + d \quad \text{---- level-0 kernel (hastily chosen)} \]
\[ Q = x/D = b(e + f(g + h)) \quad \text{---- weak division} \]
\[ Q = e + f(g + h) \quad \text{---- make cube-free} \]
\[ (D, R) = \text{WEAK_DIV}(x, Q) \quad \text{---- second division} \]
\[ D = a + b(c + d) \]
\[ x = QD + R \quad \text{----- } R = 0 \]
\[ x = (e + f(g + h))(a + b(c + d)) \]
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 both positive and negative phases

DECOMP(f_i)
  k = CHOOSEKERNEL(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

deomp[-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,
deomp -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

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

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 \( F \).

\[
\begin{align*}
x &= ab + cd + e \\
F &= abf + a'cd + cdf + a'de + ef
\end{align*}
\]

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

\[
\begin{align*}
D &= x'(cad + cd + e) + x(ab + cd + e) \\
F_1 &= xef + xa'cd + xa'de + xabf + xcdf \\
R_1 &= abf'x + aef'x + d'ef'x + a'c'e'x' + \\
&\quad b'c'e'x' + a'd'e'x' + b'd'e'x' + acdf'
\end{align*}
\]

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

\[
\begin{align*}
F_3 &= xf + xa'd + xa'd + xaf + xdf \\
&= xf + xa'd \\
&= x(f + a'd) \\
F &= (ab + cd + e)(f + a'd)
\end{align*}
\]
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 subexpressions and manipulates the Boolean network. We can combine decomposition and substitution to provide an effective extraction algorithm.

\[
\text{EXTRACT}(\eta) \\
\quad \text{For each node } n \{ \\
\quad \quad \eta = \text{DECOMP}(n, \eta) \\
\quad \} \\
\quad \text{For each node } n \{ \\
\quad \quad \text{RESUB}(n, \eta) \\
\quad \} \\
\text{Eliminate nodes with small value}
\]
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 ≤ threshold

New Node
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:**
\[ K^0(f_1) = \{cl + f + g\} \]
\[ K^0(f_2) = \{cl + f + j\} \]
\[ K^0(f_1) \cap K^0(f_2) = cl + f \]
\[ m = cl + f \]
\[ f_1 = ab(m + g) + h \]
\[ f_2 = ai(m + j) + k \]
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
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}_\text{cube}_\text{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

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 $C = \{c_1, c_2, ..., c_m\}$ 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 $d = make\_cube\_free(c_i + c_j)$. 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**

```shell
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.
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
End
Complexity of Algebraic Operations

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

\[ ab'de = 01 10 11 01 01 \text{ (binary number)} \]

Can we find a \textbf{linear} time algorithm for computing F/G and R produced in sorted order?

\textbf{Answer:} Yes (see McGeer and Brayton VLSI’87)

In fact, the operations, algebraic division, multiplication, addition and subtraction, and equality test are all linear and \textbf{stable}.

\textbf{Definition:} A \textbf{stable} algorithm receives its input in \textbf{sorted} order and produces its outputs in \textbf{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.

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