ECE 697B (667)
Spring 2003

Synthesis and Verification of Digital Systems

Multi-level Minimization

- Algebraic division

Slides adopted (with permission) from A. Kuehlmann, UC Berkeley, 2003

Outline

- Division and factorization
  - Definitions
  - Algebraic vs Boolean
- Algebraic division
  - Algorithm
  - Applications
- Finding good divisors
  - Kernels and co-kernels
- Generation of all Kernels – algorithm
- Extraction: rectangle covering method

Factorization

- Given an F in SOP form, how do we generate a "good" factored form

- Division operation:
  - central in many operations
  - need to find a good divisor *D*
  - apply the actual division
    - results in *quotient Q* and *remainder R*

- Applications:
  - factoring
  - substitution
  - extraction

Division

Definition:

An operation *OP* is called *division* if, given two SOP expressions *F* and *G,* it generates expressions *H* and *R,* such that:

$$F = GH + R$$

- *G* is called the *divisor*
- *H* is called the *quotient*
- *R* is called the *remainder*

Definition:

If *GH* is an *algebraic product*, then *OP* is called an *algebraic division* (denoted F // G)

otherwise *GH* is a *Boolean product* and OP is a *Boolean division* (denoted $F \div G$).

Division ($f = gh+r$)

Example:

$f = ad + ae + bcd + j$
$g_1 = a + bc$
$g_2 = a + b$

- Algebraic division:
  - $f // a = d + e, r = bcd + j$
  - $f // (bc) = d, r = ad + ae + j$
  - (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:

G is an *algebraic factor* of F if there exists an algebraic expression H such that

$F = GH$ using *algebraic* multiplication.

Definition:

G is an Boolean factor of F if there exists an expression H such that

$F = GH$ using *Boolean* multiplication.

Example:

$f = ac + ad + bc + bd$

$(a+b)$ is an algebraic factor of f since $f = (a+b)(c+d)$

$f = a'b + ac + bc$

$(a+b)$ is a Boolean factor of f since f = (a+b)(a'+c)

Why Use Algebraic Methods?

- Need spectrum of operations
  - algebraic methods provide fast algorithms
- Treat logic function like a polynomial
  - efficient data structures
  - fast methods for manipulation of polynomials available
- Loss of optimality, but results are quite good
- Can iterate and interleave with Boolean operations
- In specific instances slight extensions available to include Boolean methods

Weak Division

Weak division is a specific case 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, expressions H and R generated by weak division are unique.

Algorithm

```
ALGORITHM WEAK_DIV(F,G) {  // G={g1,g2,...},
    f=(f1,f2,...}
  foreach gi {
    Vgi=∅
    foreach fj {
      if(fj contains all literals of gi) {
        vij=fj - literals of gi
        Vgi=Vgi ∪ vij
      }
    }
  }
  H = ∩i Vgi
  R = F - GH
  return (H,R);
}
```


Example of *WEAK_DIV*

Example: divide F by G

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

$G = ae + b$

$V^{ae} = c + d$

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

$H = c + d = F/G$ $\qquad$ $H = \cap V^{g_i}$

$R = be + a'b + ab$ $\qquad$ $R = F \setminus GH$

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

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.

Division - What do we divide with?

- *Weak_Div* provides a methods to divide an expression for a given divisor
- How do we find a “good” divisor?
  - Restrict to algebraic divisors
  - Generalize to Boolean divisors
- Problem:
  - Given a set of functions { F_i }, find common weak divisors (algebraic divisors).

Kernels and Kernel Intersections

Definition:

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).

$(ab + c)$ is cube-free

$(ab + ac)$ and abc are not cube-free

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

Definition:

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

Definition:

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:

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:

$$x = adf + aef + bdf + bef + cdf + cef + g$$
$$= (a + b + c)(d + e)f + g$$

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

Fundamental Theorem

THEOREM:

If two expressions F and G have the property that

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

(i.e., 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*

Definition:

A kernel is of level 0 (K^0) if it contains no kernels except itself.

A kernel is of level n (K^n) if it contains at least one kernel of level (n-1), but no kernels (except itself) of level n or greater

- $K^0(F) \subset K^1(F) \subset K^2(F) \subset ... \subset K^n(F) \subset K(F)$.
- level-n kernels = $K^n(F) \setminus K^{n-1}(F)$
- $K^n(F)$ is the set of kernels of level k or less.

Example:

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

$k_1 = a + b(c + d) \quad \in K^{1,} \notin K^0$ *(level 1)*

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

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

Kerneling Algorithm

```
Algorithm KERNEL(j, G) {
  R = ∅
  if(CUBE_FREE(G)) R = {G}
  for(i=j+1,...,n) {
    if(l_i appears only in one term)      continue
    if(∃k ≤ i, l_k ∈ all cubes of G/l_i)   continue
    R = R ∪ KERNEL(i,MAKE_CUBE_FREE(G/l_i))
  }
  return R
}
```

MAKE_CUBE_FREE(F) removes algebraic cube factor from F

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.
- Can be used to generate all co-kernels.

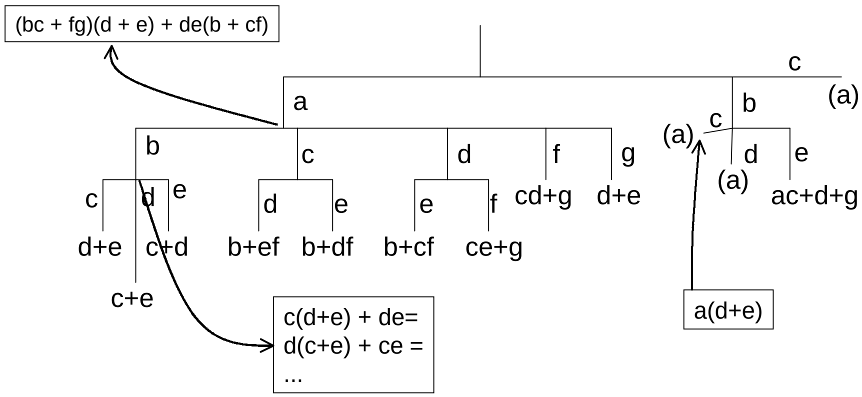
Kernel Generation - example

F = ace + bce + de + g   n = 6 variables

- Call *KERNEL(0,F)*
  - $i=1$, $l_1=a$, literal appears only once; continue
  - $i=2$, $l_2=b$, …….. ; continue
  - $i=3$, $l_3=c$,
    - *make_cube_free(F/c) = (a+b)*
    - recursive call to *KERNEL(3,(a+b))*
      - *the call considers variables 4,5,6 = {d,e,g}* – No Kernels
    - Return *R = {(a+b)}*
  - $i=4$, $l_4=d$, literal appears only once; continue
  - $i=5$, $l_5=e$,
    - *make_cube_free(F/e) = (ac+bc+d)*
    - recursive call to *KERNEL(5,(ac+bc+d))*
      - *the call considers variable 6 = {g}* – No Kernels
    - Return $R = R \cup \{(a+b), (ac+bc+d)\}$
  - $i=6$, $l_6=g$, *appears only once; continue; stop.*
    - Return $R = R \cup \{(a+b), (ac+bc+d), (ace + bce + de + g)\}$

Kerneling Illustrated

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



Kerneling Illustrated

co-kernels	kernels
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), so that *f/bca = (d+e),*
but *f/bca = (d+e)* has been already found (no repetition).

Applications - Factoring

```
Algorithm FACTOR(F) {
  if(F has no factor) return F
    // e.g. if |F|=1, or F is an OR of single literals
    // or if no literal appears more than once
  D     = CHOOSE_DIVISOR(F)
  (Q,R) = DIVIDE(F,D)
  return FACTOR(Q)·FACTOR(D) + FACTOR(R)  // recur
}
```

– Different heuristics can be applied for *CHOOSE_DIVISOR*
– Different *DIVIDE* routines may be applied (e.g. also Boolean divide)

Example and Problems of *Factor*

Example:

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

Notation:
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.

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 the Problem

Solving this problem:

- Check if the quotient Q is not a single cube, then done, else,
- Pick a literal l_1 in Q which occurs most frequently in cubes of F.
- 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:

$F = ace + ade + bce + bde + cf + df$
$D = a + b$
$Q = ce + de$
$P = (ce + de)(a + b) + (c + d) f$
$O = e(c + d)(a + b) + (c + d)f$

Notation:
F = the original function,
D = the divisor,
Q = the quotient,
P = the partial factored form,
O = the final factored form by FACTOR.

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:

- Essentially, we reverse D and Q !
- 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.

Improved Factoring

```
Algorithm 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)
  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)
}}
```

Improved Factoring

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

Example: QUICK_FACTOR

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

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

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

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

Application - Decomposition

Decomposition is the same as factoring except:

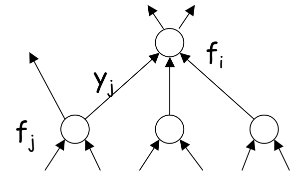
- *divisors* are added as *new nodes* in the network.
- the new nodes may *fan out* elsewhere in the network in both positive and negative phases

```
Algorithm DECOMP(f_i) {
  k = 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 using new
                                 // node for kernel
  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.

Re-substitution



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

$$f_i = qy_j + r \quad (\text{or } f_i = q_1 y_j + q_0 y'_j + r)$$

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

Extraction

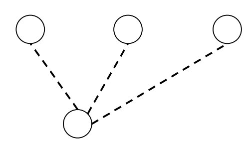
- Recall: Extraction operation identifies common sub-expressions and manipulates the Boolean network.
- Combine decomposition and substitution to provide an effective extraction algorithm.

```
Algorithm EXTRACT
  foreach node n {
     DECOMP(n)       // decompose all network nodes
   }
  foreach node n {
    RESUB(n)        // resubstitute using existing nodes
  }
  ELIMINATE_NODES_WITH_SMALL_VALUE
}
```


Extraction

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

$f_1 = ab(c(d + e) + f + g) + h, \quad f_2 = ai(c(d + e) + f + j) + k$

(only level-0 kernels used in this example)

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

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

$l = d + e$ $\quad f_1 = ab(cl + f + g) + h$

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

2. Extraction: $K^0(f_1) = \{cl + f + g\}; K^0(f_2) = \{cl + f + j\}$

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

$m = cl + f$ $\quad f_1 = ab(m + g) + h$

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

No kernel intersections anymore !

3. Cube extraction:

$n = am$ $\quad f_1 = b(n + ag) + h$

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

Rectangle Covering

Alternative method for extraction

- Build co-kernel cube matrix $M = R \times C$
  - rows correspond to co-kernels of individual functions
  - columns correspond to individual cubes of kernel

$$m_{ij} = \begin{cases} 1 \text{ (cubes of functions)} \\ 0 \text{ if cube is not there} \end{cases}$$

- Rectangle covering:
  - identify sub-matrix $M' = R' \times C'$, where $R' \subseteq R$, $C' \subseteq C$, and $m'_{ij} \neq 0$
  - construct divisor D corresponding to M' as new node
  - extract D from all functions

Example for Rectangle Covering

$F = af + bf + ag + cg + ade + bde + cde$

$G = af + bf + ace + bce$

$H = ade + cde$

Kernels/Co-kernels:

F: (de+f+g)/a

(de + f)/b

(a+b+c/de

(a + b)/f

(de+g)/c

(a+c)/g

G: (ce+f)/{a,b}

(a+b)/{f,ce}

H: (a+c)/de

M =

		a	b	c	ce	de	f	g
F	a					ade	af	ag
F	b					bde	bf	
F	de	ade	bde	cde				
F	f	af	bf					
F	c					cde		cg
F	g	ag		cg				
G	a				ace		af	
G	b				bce		bf	
G	ce	ace	bce					
G	f	af	bf					
H	de	ade		cde				

Example for Rectangle Covering

$F = af + bf + ag + cg + ade + bde + cde$

$G = af + bf + ace + bce$

$H = ade + cde$

- Pick sub-matrix (rectangle) M'
- Extract new expression X

$X = a + b$

$F = fx + ag + cg + dex + cde$

$G = fx + cex$

$H = ade + cde$

- Update M

M =

		a	b	c	ce	de	f	g
F	a					ade	af	ag
F	b					bde	bf	
F	de	ade	bde	cde				
F	f	af	bf					
F	c					cde		cg
F	g	ag		cg				
G	a				ace		af	
G	b				bce		bf	
G	ce	ace	bce					
G	f	af	bf					
H	de	ade		cde				

Value of a Sub-Matrix

- Number literals before - Number of literals after

$$V(R',C') = \sum_{i\in R,\, j\in C} v_{ij} - \sum_{i\in R'} w_i^r - \sum_{j\in C} w_j^c$$

v_{ij} : Number of literals of cube m_{ij}

w_i^r: (Number of literals of the cube associated with row i) + 1

w_j^c: Number of literals of the cube associated with column j

- For the example

V = 20 - 10 - 2 = 8

$M =$

		a	b	c	ce	de	f	g
F	a					ade	af	ag
F	b					bde	bf	
F	de	ade	bde	cde				
F	f	af	bf					
F	c					cde		cg
F	g	ag		cg				
G	a				ace		af	
G	b				bce		bf	
G	ce	ace	bce					
G	f	af	bf					
H	de	ade		cde				

Pseudo-Boolean Division

- Idea: consider entries in covering matrix that are don't cares
  – overlap of rectangles (*a*+*a* = *a*)
  – product that cancel each other out (*a*+*a'* = *0*)

- Example: *F = ab' + ac' + a'b + a'c + bc' + b'c*

- Result:
  X = a' + b' + c'
  F = ax + bx + cx

$M =$

		a	b	c	a'	b'	c'
F	a				*	ab'	ac
F	b				a'b	*	bc
F	c				a'c	b'c	*
F	a'	*	a'b	a'c			
F	b'	ab'	*	b'c			
F	c'	ac'	bc'	*			

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)
- Cannot measure if kernel can be used as complemented node

- Solution: compute only subset of kernels:
  – 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

Properties of fast divisor (kernel) extraction:
- $O(n^2)$ number of 2-cube divisors in an *n*-cube Boolean expression.
- Concurrent extraction of 2-cube divisors and 2-literal cube divisors.
- Handle divisor and complemented divisor simultaneously

- 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

$F = \{c_i\}$, $D(F) = \{d \mid d = make\_cube\_free(c_i + c_j)\}$

This 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:

$F = axe + ag + bcxe + bcg$

$make\_cube\_free(c_i + c_j) = \{xe + g, a + bc, axe + bcg, ag + bcxe\}$

Note:

- the function *F* is made into an algebraic expression before generating double-cube divisors
- not all 2-cube divisors are kernels (why ?)

Key Result For 2-cube Divisors

THEOREM:

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

Proof:

If:

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:

Suppose $C = \{c_1, c_2, ..., c_m\}$ is a multiple-cube divisor of *F* and of *G*. Take any $e = (c_i + c_j)$. If e is cube-free, then $e \in D(F) \cap D(G)$. If *e* is not cube-free, then let $d = make\_cube\_free(c_i + c_j)$. Then *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 Divisor 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 (2-literal cubes) set after extraction
- Iteratate extraction of divisors until no more improvement

- Results:
  - Much faster
  - Quality as good as that of kernel extraction