

# Logic Synthesis

## Algebraic Division

# Factored Forms (contd...)

---

- Any sub-tree of a factoring tree is a **factor** (any sum or product term in the factored form)
- Two factored forms (FFs) are **equivalent** if they represent the same logic function, and they are **syntactically equivalent** if their trees are isomorphic

- Equivalent:

$$a(b+c) + bc \quad \text{and} \quad ab + c(a+b)$$

- Syntactically Equivalent:

$$(a+b)(c+d)e \quad \text{and} \quad (c+d)e(a+b)$$

# Factorization

---

- Given a SOP, how do we generate a “good” factored form
  - find a good divisor
  - apply the actual division
    - results in quotient and remainder
- Analogous to Division:
  - is central in many operations, including
    - factoring
    - decomposition
    - substitution
    - extraction

# Product

---

- A Boolean cube (implicant) is a set of literals
- SOP is a set of cubes
- A **product** of two cubes C and D is a cube

$$C D = \begin{cases} \phi & \text{if } \exists x (x \in C \cup D \text{ and } x' \in C \cup D) \\ C \cup D & \text{otherwise} \end{cases}$$

- Product of two SOP expressions, F and G, is an SOP expression s.t.

$$FG = F \times G = \{ CD \text{ such that } C \in F \text{ and } D \in G \text{ and } CD \neq \phi \}$$

- If F and G have disjoint support, then FG is an **algebraic product**, otherwise it is a **Boolean product**

(a+b)(c+d) is an algebraic product

(a+b)(a+c) and (a+b)(b'+c) are Boolean products

- FG contains precisely  $|F| \cdot |G|$  cubes if FG is an algebraic product

# Division

---

## Definition:

An operation OP is called division if, given two SOP expressions F and G, it generates expressions H and R ( $\langle H, R \rangle = OP(F, G)$ ) 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  $F // G$ , G is an **algebraic divisor**)

otherwise GH is a Boolean product and OP is called a **Boolean division**

(denoted  $F \div G$ , G is a **Boolean divisor**).

# 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

$$F = GH \text{ (using algebraic multiplication).}$$

## Definition:

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

$$F = GH \text{ (using Boolean multiplication).}$$

## Example:

$$f = ac + ad + bc + bd$$

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

$$f = \sim ab + ac + bc$$

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

# Divisors

---

- Need to find a good candidate divisor to perform division
- Division:
  - Given  $F$  and  $G$ , need to determine  $Q$  and  $R$
- Algebraic divisors are a subset of Boolean divisors
- Algebraic divisors alone can't be a basis for an optimum choice
- Boolean divisors:
  - Higher quality division at higher expense
- Algebraic divisors
  - Lower quality at cheaper CPU utilization

# 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 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 example of algebraic division.

## DEFINITION:

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

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

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

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

# 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:

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

$$G = ae + b$$

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

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

$$H = c + d = F/G$$

$$R = be + a'b + ab$$

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

$$R = F \setminus GH$$

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

# 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 (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:

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

kernels

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

co-kernels

$df, ef$   
 $af, bf, cf$   
 $1$

# 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

- $K^0(F) \subset K^1(F) \subset K^2(F) \subset \dots \subset K^n(F) \subset K(F)$ .
- 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:

$$\begin{aligned} F &= (a + b(c + d))(e + g) \\ k_1 &= a + b(c + d) \in K^1 \\ &\quad \notin K^0 \implies \text{level-1} \\ k_2 &= c + d \in K^0 \\ k_3 &= e + g \in K^0 \end{aligned}$$

# Kerneling Algorithm

---

```
Algorithm KERNEL(j, G) {  
  R =  $\emptyset$   
  if(CUBE_FREE(G)) R = {G}  
  for(i=j+1, ..., n) {  
    if( $l_i$  appears only in one term)      continue  
    if( $\exists k \leq i, l_k \in$  all cubes of  $G/l_i$ ) continue  
    R = R  $\cup$  KERNEL(i, MAKE_CUBE_FREE( $G/l_i$ ))  
  }  
  return R  
}
```

**MAKE\_CUBE\_FREE**(F) removes algebraic cube factor from F



# Kerneling Illustrated

---

co-kernels

1  
a  
ab  
abc  
abd  
abe  
ac  
acd

kernels

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

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

# 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( $f_i$ ) {  
   $k = \mathbf{CHOOSE\_KERNEL}(f_i)$   
  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( $f_{m+j}$ )  
}
```

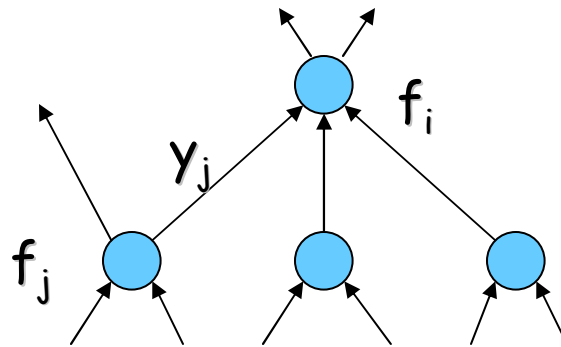
Similar to factoring, we can define

**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

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

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

# Extraction

---

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

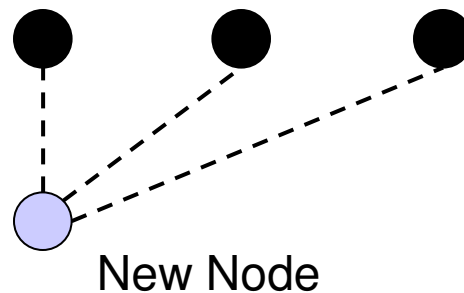
```
Algorithm EXTRACT
  foreach node n {
    DECOMP(n)          // decompose all network nodes
  }
  foreach node n {
    RESUB(n)          // resubstitute using existing nodes
  }
  ELIMINATE_NODES_WITH_SMALL_VALUE
}
```

# 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  $\leq$  threshold



# Example-Extraction

---

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

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

(only level-0 kernels used in this example)

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

$$l = d + e$$
$$f_1 = ab(cl + f + g) + h$$
$$f_2 = ai(cl + f + j) + k$$

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

$$m = cl + f$$
$$f_1 = ab(m + g) + h$$
$$f_2 = ai(m + j) + k$$

No kernel intersections anymore!!

3. Cube extraction:

$$n = am$$
$$f_1 = b(n + ag) + h$$
$$f_2 = i(n + aj) + k$$

# Node Elimination

---

- Function  $F_i$  is a fanout of  $F_j$  represented by

$$F_i = \tilde{F}_i y_j + \hat{F}_i y_j + R_i$$

- Increase in literal count when we substitute SOP expression of  $F_j$  into this representation of  $F_i$
- If repeated for every node in direct fanout of  $F_j$ , we can eliminate  $j$  from the network
  - Number of literals tradeoff with levels of logic
- Eliminate nodes with small value
- Consider a single node  $i$  in the fanout of node  $j$ 
  - $n_i$  – number of times  $y_j$  or  $y_j'$  appears in the FF representation of  $F_i$
  - $L_i$  – number of literals in FF of  $F_i$

# Node Elimination (contd...)

---

- After elimination
  - Increase of literals in FF of  $F_i = n_i L_j - n_i$
- Cost of elimination
  - Sum over fanout of node j
  - Subtract literal count of node j, which will be eliminated

$$e\_value_j = \left( \sum_{i \in \text{fanout of } j} n_i (L_j - 1) \right) - L_j$$

- This is the literal savings when products are algebraic
- Not necessarily true in real circuits
- Approximation to true elimination value