

VE527

Computer-Aided Design of Integrated Circuits

Multi-level Logic Synthesis:
Factor Operation

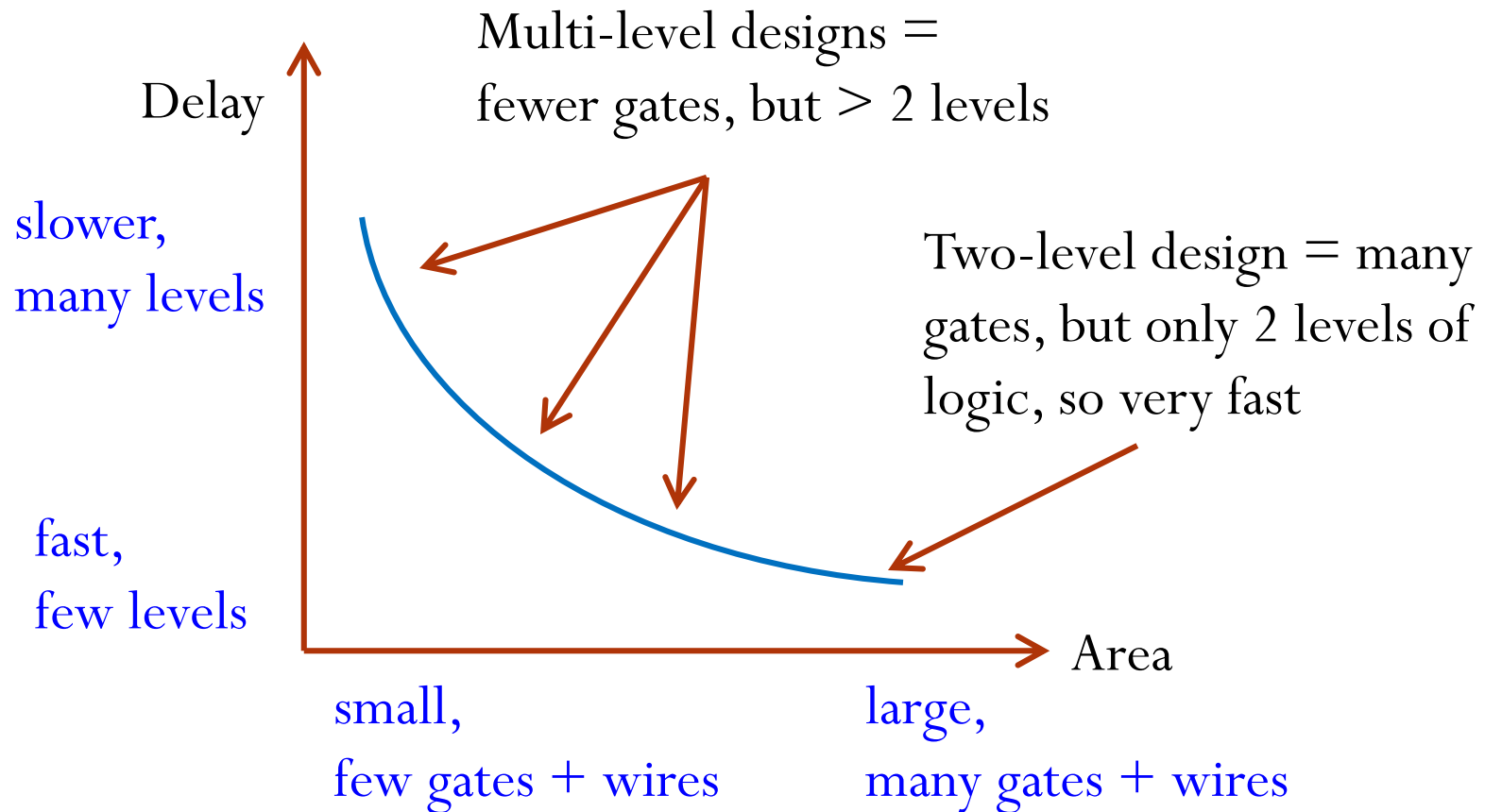
Outline

- Introduction to Multi-Level Logic Synthesis
- Factor operation
 - Algebraic Division
 - Kernels
 - Obtaining Kernels

Why Multi-level Logic?

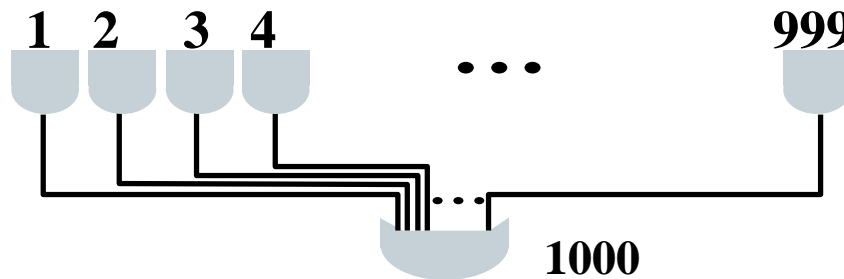
- Two-level forms are too **restrictive**
- It has small delay but large area
 - **Area** = gates + literals (wires), i.e., things that take space on a chip
 - **Delay** = maximum levels of logic gates

Area versus Delay Tradeoff



Why Multi-level Logic?

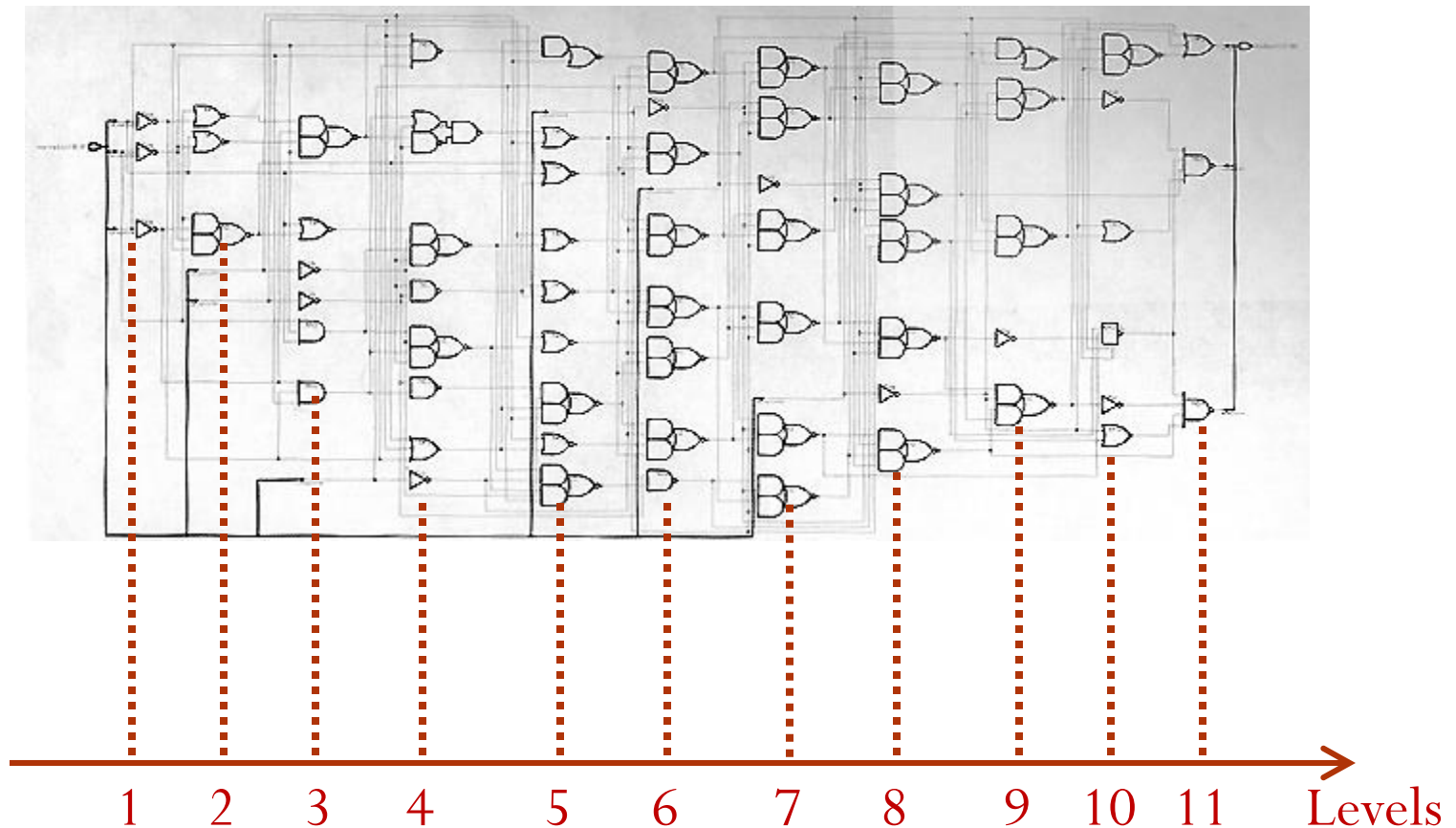
- **Rarely** see 2-level designs for really big things...
 - We use 2-level logic mostly for **components** of bigger things.
 - Even small things routinely done as **multi-level**.
- What does a 2-level design with 1000 gates look like?



This is just **NOT** going to be the preferred logic network structure...

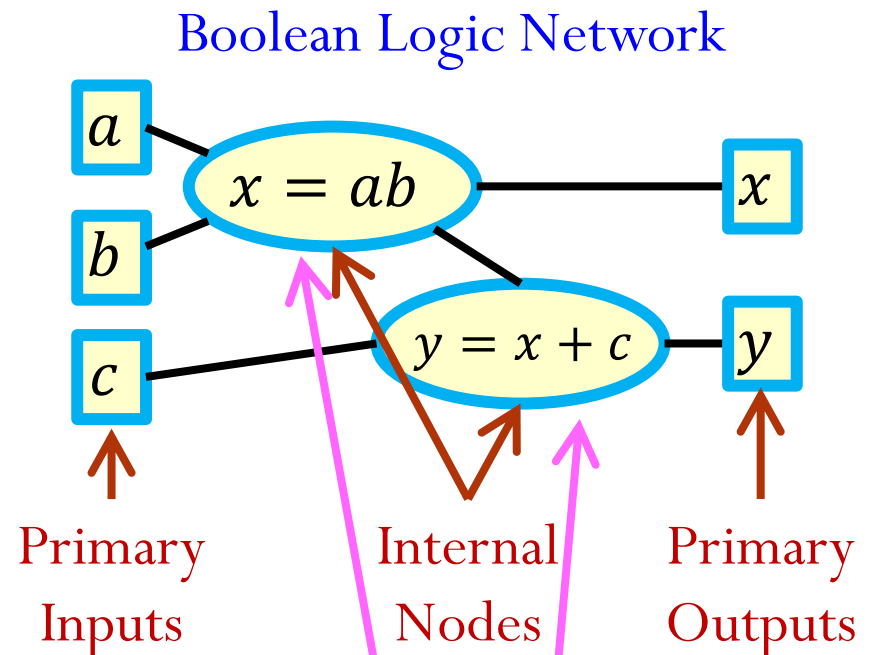
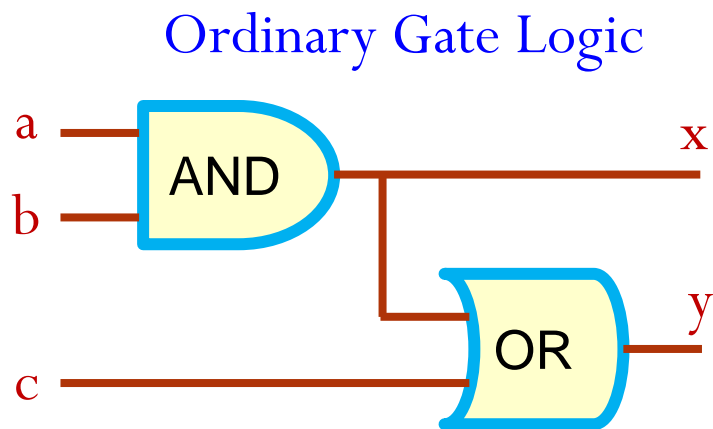
Real Multilevel Example

- A small design, done by commercial synthesis tool.



Boolean Logic Network Model

- Need more sophisticated model: **Boolean Logic Network**
 - Idea: it's a **graph** of connected blocks, like any logic diagram, but now individual component blocks is **2-level Boolean functions in SOP form**.



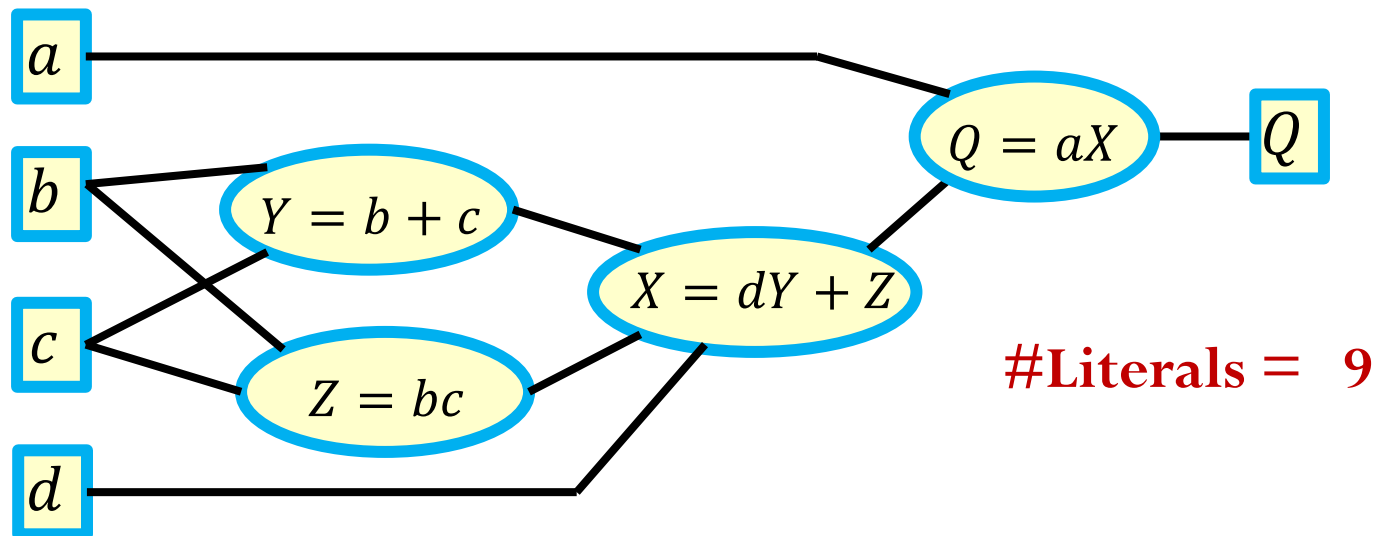
x and y are now Boolean function.

Multilevel Logic: What to Optimize?

- A simplistic but surprisingly useful metric:

Total literal count

- Count **every** appearance of **every** variable on right hand side of “=” in **every internal node**.
- Delays also matter, but for our lecture, only focus on **area**.



Optimizing Multilevel Logic: Big Ideas

- **Boolean logic network** is a **data structure**. So, what **operators** do we need?
- 3 basic kinds of operators:
 - **Simplify** network nodes: no change in # of nodes, just simplify insides, which are **SOP form**.
 - **Remove** network nodes: take “too small” nodes, **substitute them back** into fanouts.
 - This is not too hard. This is mostly manipulating the graph, simple SOP edits.
 - **Add** new network nodes: this is **factoring**. Take big nodes, split into smaller nodes.
 - This is a **big deal**. This is new. This is what we will show you...

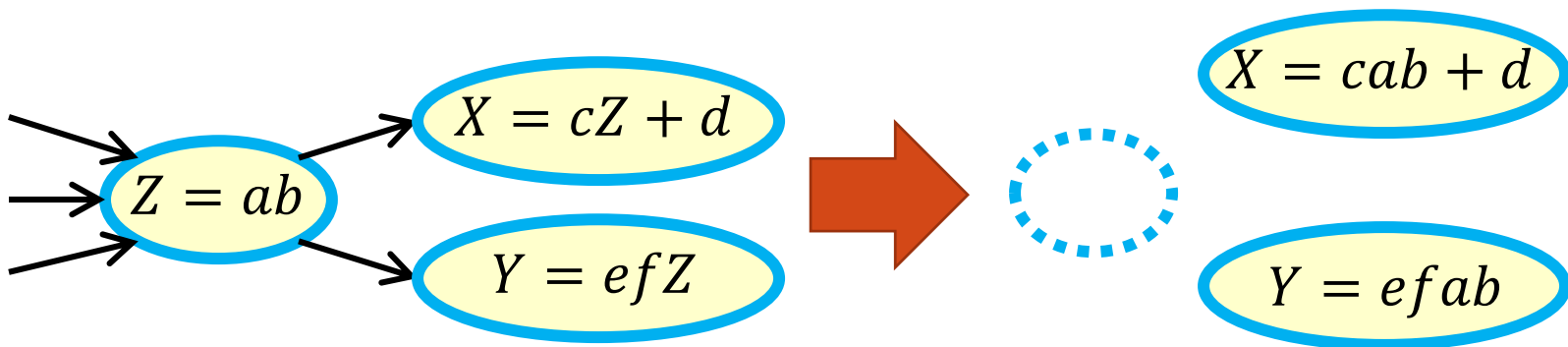
Simplifying a Node

- You already know this! This is **2-level synthesis**.
- Just run ESPRESSO on 2-level form **inside** the node, to reduce # literals.
- As structural changes happen across network, “**insides**” of nodes may present opportunity to simplify.



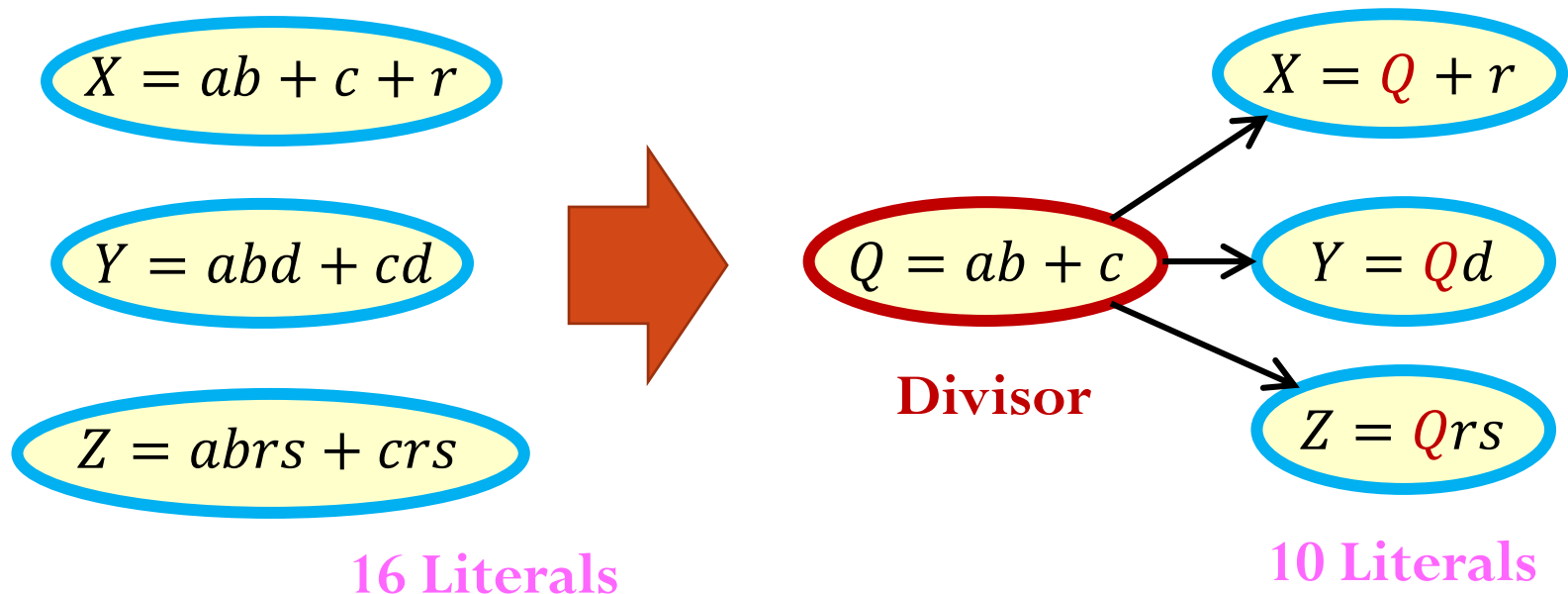
Removing a Node

- Typical case is you have a “**small**” factor which doesn’t seem to be worth making it a **separate** node.
- “**Push**” it back into its fanouts, make those nodes bigger, and hope you can use 2-level simplification on them.



Adding new Nodes

- This is Factoring, this is new, and hard.
 - Look at existing nodes, identify **common divisors**, extract them, connect as **fanins**.
 - Tradeoff: **more** delay (another level of logic), but **fewer literals** (less gate area).



Multilevel Synthesis Scripts

- Multilevel synthesis like 2-level synthesis is **heuristic**.
- ...but it's also more complex. Write **scripts of basic operators**.
 - Do **several passes** of different optimizations over the Boolean logic network.
 - Do some “cleanup” steps to get rid of “too small” nodes (**remove nodes**).
 - Look for “easy” **factors**: just look at existing nodes, and try to use them.
 - Look for “hard” **factors**: do some work to extract them, try them, and keep the good ones.
 - Do 2-level optimization of insides of each logic node in network (**simplify nodes** by ESPRESSO).
 - Lots of “art” in the engineering of these scripts.
- For us, the one big thing you don't know: **How to factor...**

Outline

- Introduction to Multi-Level Logic Synthesis
- Factor operation
 - Algebraic Division
 - Kernels
 - Obtaining Kernels

Another New Model: Algebraic Model

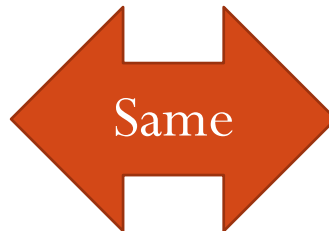
- Factoring: How do we really do it?
 - Develop another model for Boolean functions, cleverly designed to let us do this
 - Tradeoff: lose some “expressivity” – some aspects of Boolean behavior and some Boolean optimizations we just **cannot do**, but we **gain practical factoring**.
- New model: **Algebraic model**
 - Term “algebraic” comes from pretending that Boolean expressions behave like **polynomials of real numbers**, not like Boolean algebra.
 - **Big new Boolean operator**: **Algebraic Division** (also known as **“Weak” Division**).

Algebraic Model

- Idea: keep just those **rules** that work for **BOTH** polynomials of reals **AND** Boolean algebra, but **get rid of** the rest.

Real numbers

$$\begin{aligned}a \cdot b &= b \cdot a & a + b &= b + a \\a \cdot (b \cdot c) &= (a \cdot b) \cdot c \\a + (b + c) &= (a + b) + c \\a \cdot (b + c) &= a \cdot b + a \cdot c \\a \cdot 1 &= a & a \cdot 0 &= 0 \\a + 0 &= a\end{aligned}$$



Boolean algebra

$$\begin{aligned}a \cdot b &= b \cdot a & a + b &= b + a \\a \cdot (b \cdot c) &= (a \cdot b) \cdot c \\a + (b + c) &= (a + b) + c \\a \cdot (b + c) &= a \cdot b + a \cdot c \\a \cdot 1 &= a & a \cdot 0 &= 0 \\a + 0 &= a\end{aligned}$$



$$\begin{aligned}a \cdot \bar{a} &= 0 & a + \bar{a} &= 1 \\a \cdot a &= a & a + a &= a \\a + 1 &= 1 \\(a + b)(a + c) &= a + b \cdot c\end{aligned}$$

Algebraic Model

- If we only get to use algebra rules from real numbers...
- **Consequence**: A variable and its complement must be treated as **totally unrelated**.
 - Since no expression like $a + \bar{a} = 1$ allowed.

$$F = ab + \bar{a}x + \bar{b}y$$



$$\text{Let } R = \bar{a}, S = \bar{b}$$

$$F = ab + Rx + Sy$$

- **Aside**: this is one of the losses of “expressive power” of Boolean algebra.

Algebraic Model: Representation

- Boolean functions manipulated in SOP form like **polynomials**.
- Each product term in such an expression is just **a set of variables**, e.g., $abRy$ is the set (a, b, R, y) .
- SOP expression itself is just a **list of these products (cubes)**, e.g., $ab + Rx$ is the list $((a, b), (R, x))$.

Algebraic Division: Our Model for Factoring

- Given function F , we want to factor it as:

$$F = D \cdot Q + R$$

divisor quotient remainder

- If remainder $R = 0$, we call the divisor as a “**factor**”.

Example: $F = ac + ad + bc + bd + e$


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

divisor quotient remainder

Algebraic Division

- Example: $F = ac + ad + bc + bd + e$
- Want: $F = D \cdot Q + R$.

Divisor is a **factor**
if $R = 0$.



Divisor (D)	Quotient (Q)	Remainder (R)	Is D Factor?
$ac + ad + bc + bd + e$	1	0	Yes
$a + b$	$c + d$	e	No
$c + d$	$a + b$	e	No
a	$c + d$	$bc + bd + e$	No
b	$c + d$	$ac + ad + e$	No
c	$a + b$	$ad + bd + e$	No
d	$a + b$	$ac + bc + e$	No
e	1	$ac + ad + bc + bd$	No

Algebraic Division: Very Nice Algorithm

- **Inputs**: A Boolean expression F and a divisor D , represented as lists of cubes (and each cube as a set of literals).
- **Output**
 - Quotient $Q = F/D$ as a cube list, or **empty** if $Q = 0$.
 - Remainder R as a cube list, or **empty** if D was a **factor**.
- **Strategy**
 - Cube-wise walk through cubes in divisor D , trying to divide each of them into F .
 - ... intersect all the division results.

Algebraic Division Algorithm

AlgebraicDivision(F , D) { // divide D into F

for (each cube d in divisor D) {

let $C = \{ \text{cubes in } F \text{ that contain this product term } d \};$

if (C is empty) return (quotient = 0, remainder = F);

let $C =$ cross out literals of cube d in each cube of C ;

if (d is the first cube we have looked at in divisor D) let $Q = C$;

else $Q = Q \cap C$;

}

$R = F - (Q \cdot D);$

return (quotient = Q , remainder = R);

}

Example:

Cube $xyzw$ contains product term yz

Example:

Suppose $C = xyz + yzw + pqyz$ and $d = yz$. Then crossing out all the yz parts yields $x + w + pq$

Algebraic Division: Example

$$F = axc + axd + axe + bc + bd + de, D = ax + b$$

F cube	D cube: ax	D cube: b
axc	$axc \rightarrow c$	—
axd	$axd \rightarrow d$	—
axe	$axe \rightarrow e$	—
bc	—	$bc \rightarrow c$
bd	—	$bd \rightarrow d$
de	—	—

$$C = c + d + e \quad C = c + d$$

$$Q = (c + d + e) \cap (c + d) = c + d$$

$$R = F - QD = axe + de$$

Algebraic Division: Warning

- Remember: No “**Boolean**” simplification, only “**algebraic**”.
- So what? Well, suppose you have this

$$F = a\bar{b}\bar{c} + ab + ac + bc, D = ab + \bar{c}$$

and you want F/D .

- You must let $X = \bar{b}$ and $Y = \bar{c}$ and transform F and D to something like

$$F = aXY + ab + ac + bc, D = ab + Y$$

- Because we must treat the true and complement forms of variables as **totally unrelated**.

One More Constraint: Redundant Cubes

- To do F/D , function F must have no **redundant** cubes
 - Technical definition is: **minimal** with respect to **single-cube containment**.
 - Means: no one cube is **completely** covered by **one of the other** cubes in SOP cover.
 - E.g., $abcd$ is completely covered by ab .
- Why no redundant cubes?
 - Consider: $F = a + ab + bc$ and $D = a$.
 - Note: F has redundant cube ab .
 - What is F/D by our **algebraic division algorithm**?

$$Q = F/D = 1 + b$$

However, we don't have $1 + \text{stuff}$ operation in algebraic model!

One More Constraint: Redundant Cubes

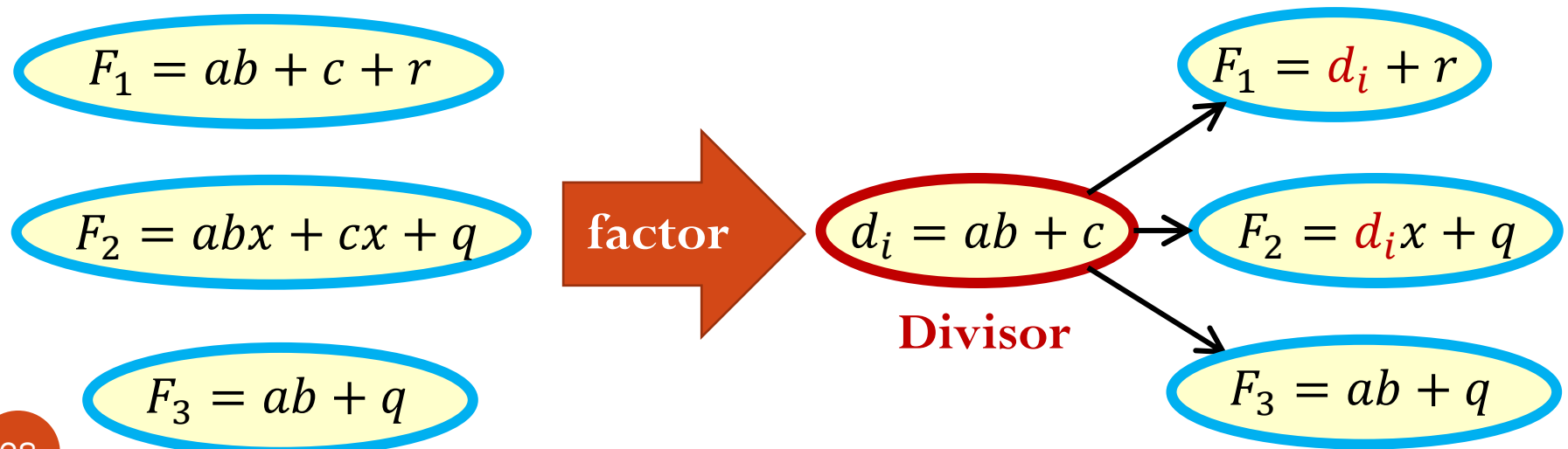
- ... So, we should remove redundant cubes to make the SOP **minimal** with respect to **single-cube containment**.
 - Not hard.

Outline

- Introduction to Multi-Level Logic Synthesis
- Factor operation
 - Algebraic Division
 - Kernels
 - Obtaining Kernels

Multilevel Logic Synthesis: Where are We?

- For Boolean function F and D , can compute $F = Q \cdot D + R$ via **algebraic model**.
 - It is great, but still **not enough**: don't know how to **find** a good divisor D .
 - **Another problem**: given n functions F_1, F_2, \dots, F_n , find **a set of good common divisors**.



Where To Look For Good Divisors?

- Surprisingly, the **algebraic model** has a beautiful answer.
- Where to look for divisors of function F ?
 - In the set of **kernels** of F , denoted $K(F)$.
 - $K(F)$ is another set of 2-level SOP forms which are the special, foundational structure of any function F
- How to find a kernel $k \in K(F)$?
 - **Algebraically divide** F by one of its **co-kernels**, c .

Kernels and Co-Kernels of Function F

- **Kernel** of a Boolean expression F is:
 - A **cube-free** quotient k obtained by **algebraically dividing** F by a **single cube** c .
 - This single cube c also has a name: it is a **co-kernel** of function F .

$$\begin{array}{r}
 \text{quotient } Q \\
 \hline
 \text{divisor } D \left[\begin{array}{c} \text{expression } F \\ \vdots \end{array} \right. \\
 \hline
 \text{remainder } R \\
 F = D \cdot Q + R
 \end{array}$$

$$\begin{array}{r}
 \text{kernel } k \text{ (cube-free)} \\
 \hline
 c = 1 \text{ cube} \left[\begin{array}{c} \text{expression } F \\ \vdots \end{array} \right. \\
 \hline
 \text{remainder } R \\
 F = c \cdot k + R
 \end{array}$$

Kernels Are Cube-Free...

- **Cube-free** means...?
 - You cannot factor out a single cube divisor that **leaves no remainder**.
 - Technically: has no **one cube** that is a **factor** of expression.
 - Pick a cube C . If you can “**cross out**” C in each product term of F , then F is **not** a **kernel**.
 - Note: cubes “1” and “ a ” are **NOT** cube-free

Expression F	$F = D \cdot Q + R$	F Cube-free?
a	$a \cdot 1 + 0$	No
$a + b$	--	Yes
$ab + ac$	$a \cdot (b + c) + 0$	No
$abc + abd$	$ab \cdot (c + d) + 0$	No
$ab + acd + bd$	--	Yes

Some Kernel Examples

- Suppose $F = abc + abd + bcd$

Divisor cube d	$Q = F/d$	Is Q a kernel of F ?
1	$abc + abd + bcd$	No, has cube = b as factor
a	$bc + bd$	No, has cube = b as factor
b	$ac + ad + cd$	Yes! co-kernel = b
ab	$c + d$	Yes! co-kernel = ab

- Any Boolean function F can have many different kernels.
 - The set of kernels of F is denoted as $K(F)$.

Kernels: Why Are They Important?

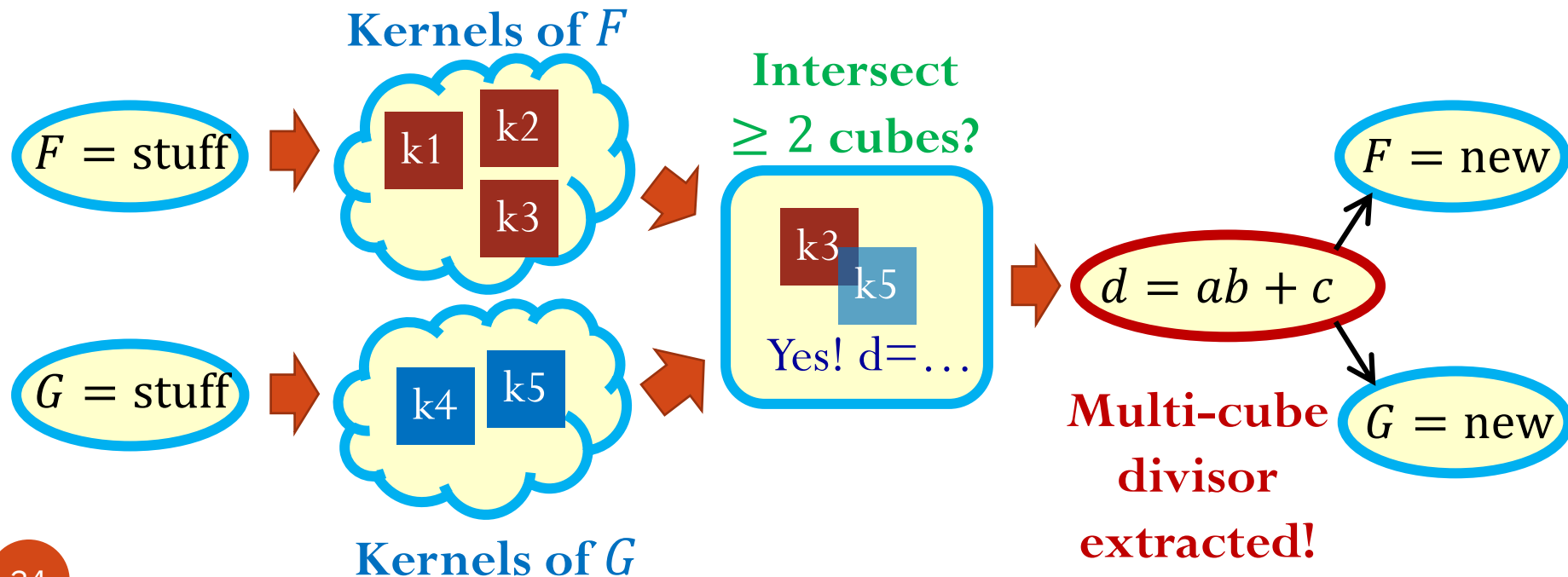
- Big result: Brayton & McMullen Theorem
 - **From**: R. Brayton and C. McMullen, “The decomposition and factorization of Boolean expressions.” In *IEEE International Symposium on Circuits and Systems*, pages 49–54, 1982.

Expressions F and G have a **common multiple-cube divisor** d **if and only if**:

there are kernels $k_1 \in K(F)$ and $k_2 \in K(G)$ such that $d = k_1 \cap k_2$ and d is an expression with **at least 2** cubes in it (i.e., k_1 and k_2 have **common cubes**).

Multiple-Cube Divisors and Kernels

- Brayton & McMullen Theorem in words:
 - The only place to look for **multiple-cube divisors** is in the **intersection of kernels**!
 - Indeed, this intersection of kernels contains all divisors.



Brayton-McMullen: Informal Illustration

$$F = \text{cube1} \cdot \text{kernel1} + \text{remainder1}$$

$$G = \text{cube2} \cdot \text{kernel2} + \text{remainder2}$$

Assume:

$$\text{kernel1} \cap \text{kernel2} = X + Y$$



$$F = \text{cube1} \cdot [X + Y + \text{stuff1}] + \text{remainder1}$$

$$G = \text{cube2} \cdot [X + Y + \text{stuff2}] + \text{remainder2}$$



$$F = \text{cube1} \cdot [X + Y] + [\text{cube1} \cdot \text{stuff1} + \text{remainder1}]$$

$$G = \text{cube2} \cdot [X + Y] + [\text{cube2} \cdot \text{stuff2} + \text{remainder2}]$$



$$d = X + Y$$



$$F = \text{cube1} \cdot d + \dots$$



$$G = \text{cube2} \cdot d + \dots$$

Kernels: Real Example

$$F = ae + be + cde + ab$$

$$G = ad + ae + bd + be + bc$$

Kernels	Co-kernel
$a + b + cd$	e
$b + e$	a
$a + e$	b
$ae + be + cde + ab$	1

Kernels	Co-kernel
$a + b$	$d \text{ or } e$
$d + e$	a
$c + d + e$	b
$ab + ae + bd + be + bc$	1

Intersecting these 2 kernels: $(a + b + cd) \cap (a + b) = a + b$

Outline

- Introduction to Multi-Level Logic Synthesis
- Factor operation
 - Algebraic Division
 - Kernels
 - Obtaining Kernels

Kernels: Very Useful, But How To Find?

- Another **recursive** algorithm (“recursive” again!)
 - There are 2 more useful properties of kernels we need to see first...
- Start with a function F and a kernel $k1$ in $K(F)$
$$F = \text{cube1} \cdot k1 + \text{remainder1}$$
- Then: a new, interesting question: what about $K(k1)$?
 - $k1$ is a perfectly nice Boolean expression, so it has got **its own** kernels.
 - Do these $k1$ ’s kernels have anything interesting to say about $K(F)$?

How $K(k1)$ Relates to $K(F)$...

- We know this: $F = \text{cube1} \bullet k1 + \text{remainder1}$
- Suppose $k2$ is a kernel in $K(k1)$, then we know

$$k1 = \text{cube2} \bullet k2 + \text{remainder2}$$

- Substitute this expression for $k1$ in original expression for F

$$F = \text{cube1} \bullet [\text{cube2} \bullet k2 + \text{remainder2}] + \text{remainder1}$$

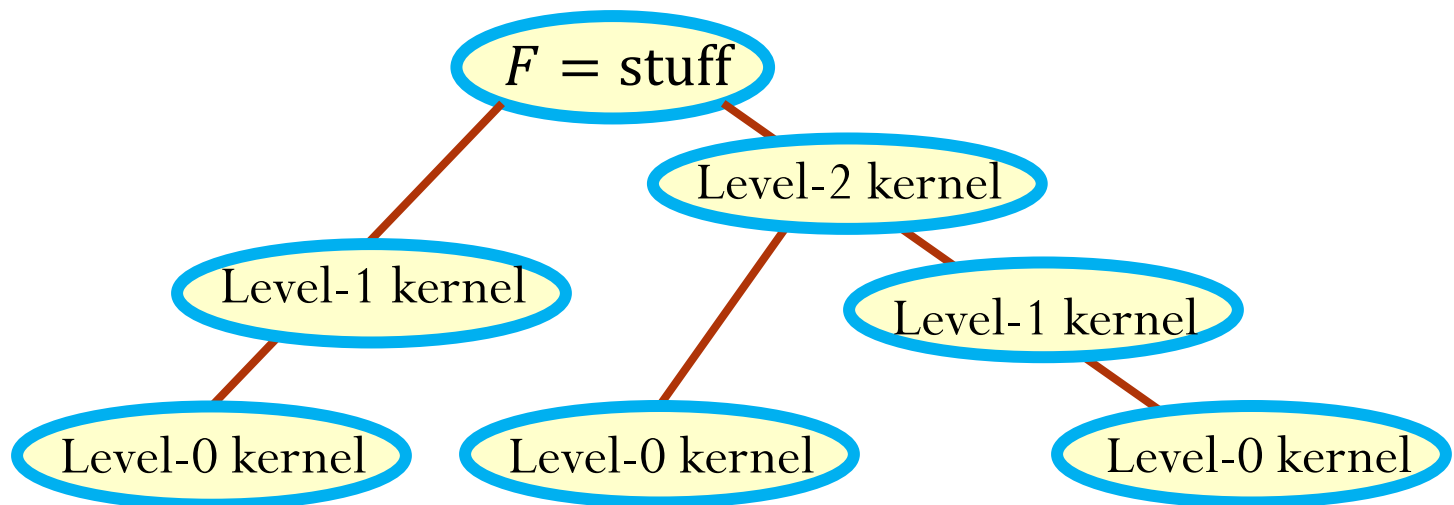
- Since $\text{cube1} \bullet \text{cube2}$ is itself just another **single cube**, we have:

$$F = (\text{cube1} \bullet \text{cube2}) \bullet [k2] + [\text{cube1} \bullet \text{remainder2} + \text{remainder1}]$$

- **Conclusion:** $k2$ is also a **kernel** of original F (with **co-kernel** $\text{cube1} \bullet \text{cube2}$)

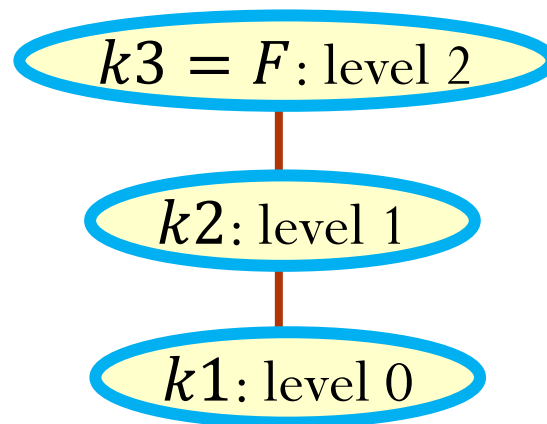
There is a Hierarchy of Kernels Inside F

- **Definition:** $k \in K(F)$ is
 - A **level-0 kernel** if it contains no kernels inside it except itself.
 - In words: Only cube you can pull out to get a cube-free quotient is “1”.
 - A **level- n kernel** if it contains **at least one** level- $(n-1)$ kernel, and no other level- n kernels except itself.
 - In words: a level-1 kernel only has level-0 kernels inside it. A level-2 kernel only has level-1 and level-0 kernels in it, etc...



Kernel Hierarchy: Example

- $F = abe + ace + de + gh$ has three kernels:
 - $k1 = b + c$, with co-kernel ae .
 - $k2 = ab + ac + d$, with co-kernel e .
 - $k3 = F$ with co-kernel 1.
- Note: $k1$ is level 0, $k2$ is level 1, and $k3$ is level 2.



Kernels

- Second useful result [by Brayton et al.]:
 - **Co-kernels** of a Boolean expression in SOP form correspond to **intersections of 2 or more cubes** in this SOP form.
- **Note:** **Intersections** here means that we regard a cube as **a set of literals**, and look at common subsets of literals.
 - This is not like “AND” for products. This just extracts **common literals**.
 - Example: $ace + bce + de + g$

$ace \cap bce = ce \rightarrow ce$ is a potential co-kernel


$ace \cap bce \cap de = e \rightarrow e$ is a potential co-kernel

How to Find Kernels Using These 2 Results?

- Find the kernels **recursively**.
 - Whenever find one kernel, call **FindKernels()** on that kernel, to find (if any) lower level kernels inside.
- Use **algebraic division** to divide function by potential co-kernels, to drive recursion.
 - Use 2nd result – co-kernels are **intersections** of the cubes: If there're at least 2 cubes, then look at the intersection of those cubes, and use that intersected result as our potential co-kernel cube.
- One technical point: need to start with a **cube-free function** F to make things work right.
 - If not cube-free, just divide by biggest common cube to simplify F .

Kernel Algorithm

```
FindKernels( cube-free SOP expression F ) {  
  K = empty;  
  for ( each variable x in F ) {  
    if ( there are at least 2 cubes in F that have variable x ) {  
      let S = { cubes in F that have variable x in them };  
      let co = cube that results from intersection of all cubes in S,  
               this will be the product of those literals  
               that appear in each of these cubes in S;  
      K = K  $\cup$  FindKernels( F / co );  
    }  
  }  
  K = K  $\cup$  F ;  
  return( K );  
}
```



Cube-free **F** is always its own kernel, with trivial co-kernel = **1**

Kernelling Example

FindKernels(**F**):

```
for (each var x in F ) {  
    if (x in  $\geq 2$  cubes in F) {  
        co = intersection of cubes;  
        K = K  $\cup$  FindKernels(F/co) ;  
    }  
}  
K = K  $\cup$  F ;  
return( K );
```

$$F = ace + bce + de + g$$

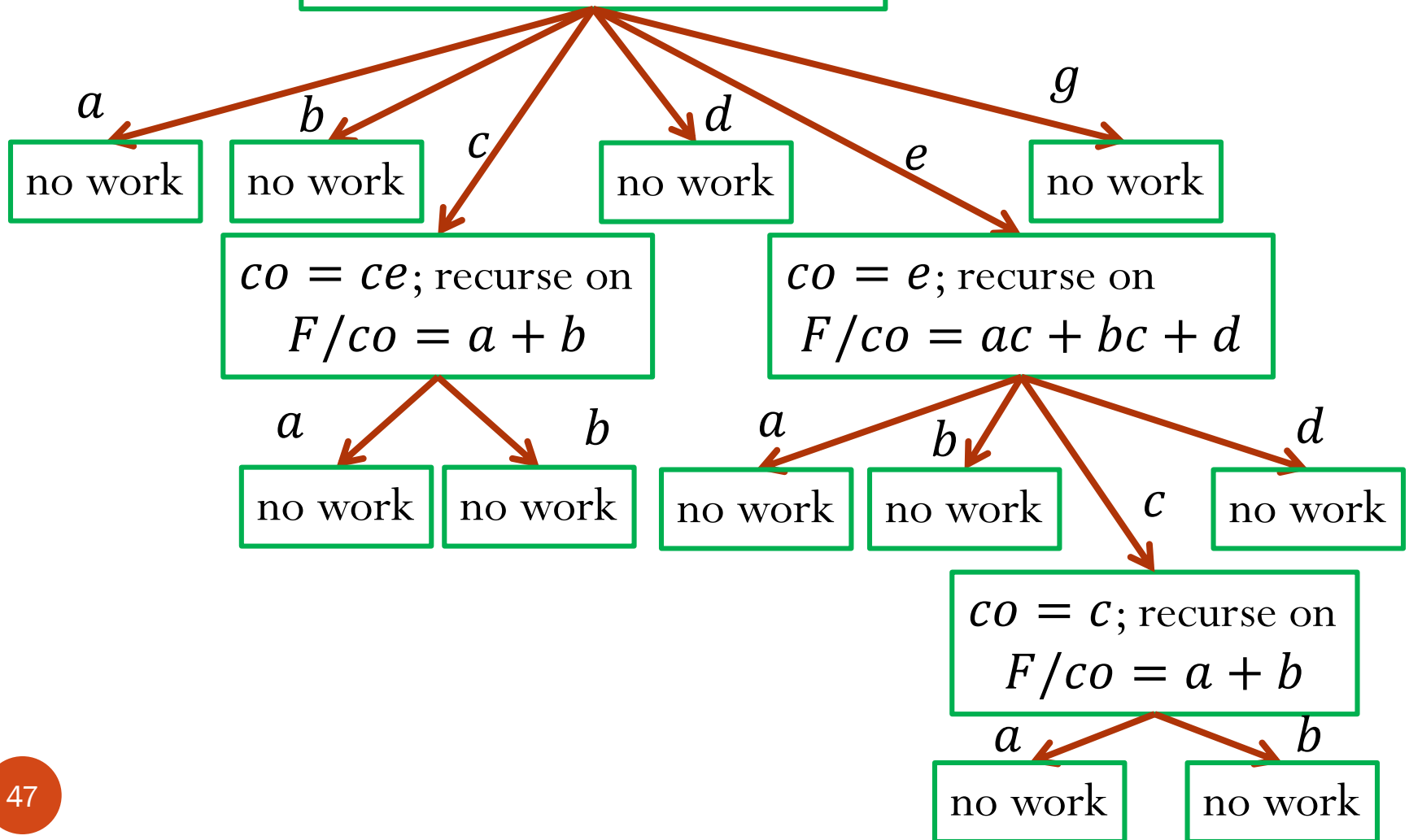
- *a*: only 1 cube with *a*, no work.
- *b*: only 1 cube with *b*, no work.
- *c*: two cubes *ace* and *bce* with *c*.
 - $co = ace \cap bce = ce$
 - $F/co = a + b$
 - **Recurse** on $a + b$
- *d*: only 1 cube with *d*, no work.
- *e*: three cubes *ace*, *bce*, and *de* with *e*.
 - $co = ace \cap bce \cap de = e$
 - $F/co = ac + bc + d$
 - **Recurse** on $ac + bc + d$
- *g*: only 1 cube with *g*, no work.

Kernelling Example (cont.)

- **Recurse** on $a + b$
 - No work for variables a and b , since one cube with a/b .
- **Recurse** on $ac + bc + d$
 - No work for variables a, b, d , since one cube with $a/b/d$.
 - c : two cubes ac and bc with c .
 - $co = ac \cap bc = c$
 - $F/co = a + b$
 - **Recurse** on $a + b$ (the same as above)

Kernelling Example (cont.)

$$F = ace + bce + de + g$$



Kernelling Example (cont.)

```

FindKernels( F ):
  for (each var x in F ) {
    ...
    K = K  $\cup$  FindKernels(F/co) ;
  }
  K = K  $\cup$  F ;
  return( K );
    
```

$$F = ace + bce + de + g$$

Kernels $K = \{a + b,$
 $ac + bc + d,$
 $ace + bce + de + g\}$

return $K = \{a + b,$
 $ac + bc + d\}$

c

return $K = \{a + b\}$

$co = ce$; recurse on
 $F/co = a + b$

e

$co = e$; recurse on
 $F/co = ac + bc + d$

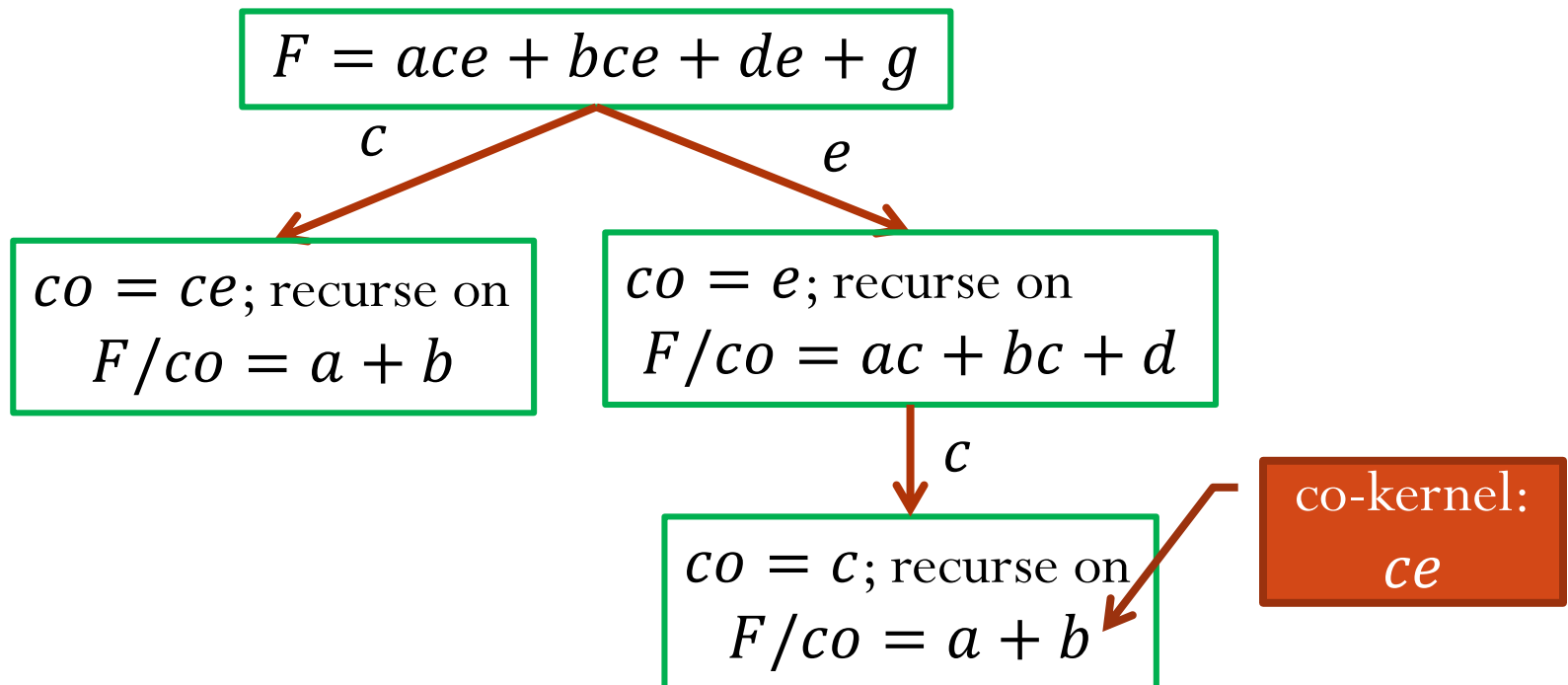
return $K = \{a + b\}$

c

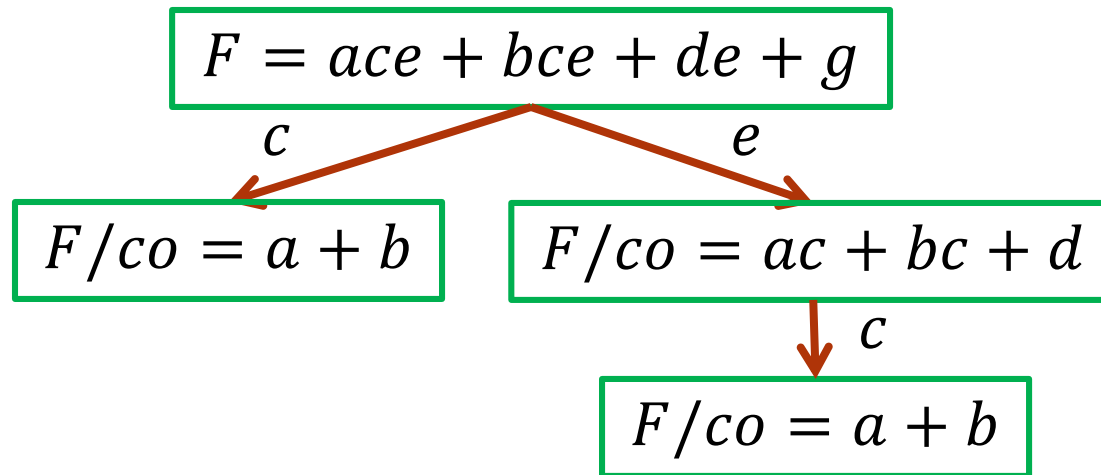
$co = c$; recurse on
 $F/co = a + b$

Get Co-Kernels

- With this algorithm ...
 - Can find **all** the kernels and **co-kernels** too.
 - Get co-kernels by **ANDing** the divisor co cubes up recursion tree.



One Tiny Problem



- The algorithm will revisit same (co-kernel, kernel) pair **multiple** times.
 - Why? Kernel you get for co-kernel abc is same as for cba , but current algorithm **doesn't know this** and will find same kernel for both co-kernels.
- Solution: remember which variables already tried in co-kernels. A little extra book keeping solves this.

Multilevel Synthesis Models: Summary

- **Boolean network model**

- Like a gate network, but each node in network is an SOP form.
- Supports many operations to add, reduce, simplify nodes in network.

- **Algebraic model & algebraic division**

- Simplifies Boolean functions to behave like polynomials of reals.
- Divides one Boolean function by another:

$$F = (\text{divisor } D) \cdot (\text{quotient } Q) + \text{remainder } R$$

- **Kernels / Co-kernels** of a function F

- **Kernel** = **cube-free** quotient obtained by dividing by a single cube (**co-kernel**)
- **Intersections** of kernels of two functions give all **multiple-cube common divisors** (Brayton & McMullen theorem).

Notes

- The **algebraic model** (and **division**) are not the only options.
 - There are also “**Boolean division**” models and algorithms that don’t lose expressivity.
 - ..but they are more complex.
 - Rich universe of models & methods here.

Good References

- R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, “MIS: A Multiple-Level Logic Optimization System,” *IEEE Transactions on CAD of ICs*, vol. CAD-6, no. 6, November 1987, pp. 1062-1081.
- Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.