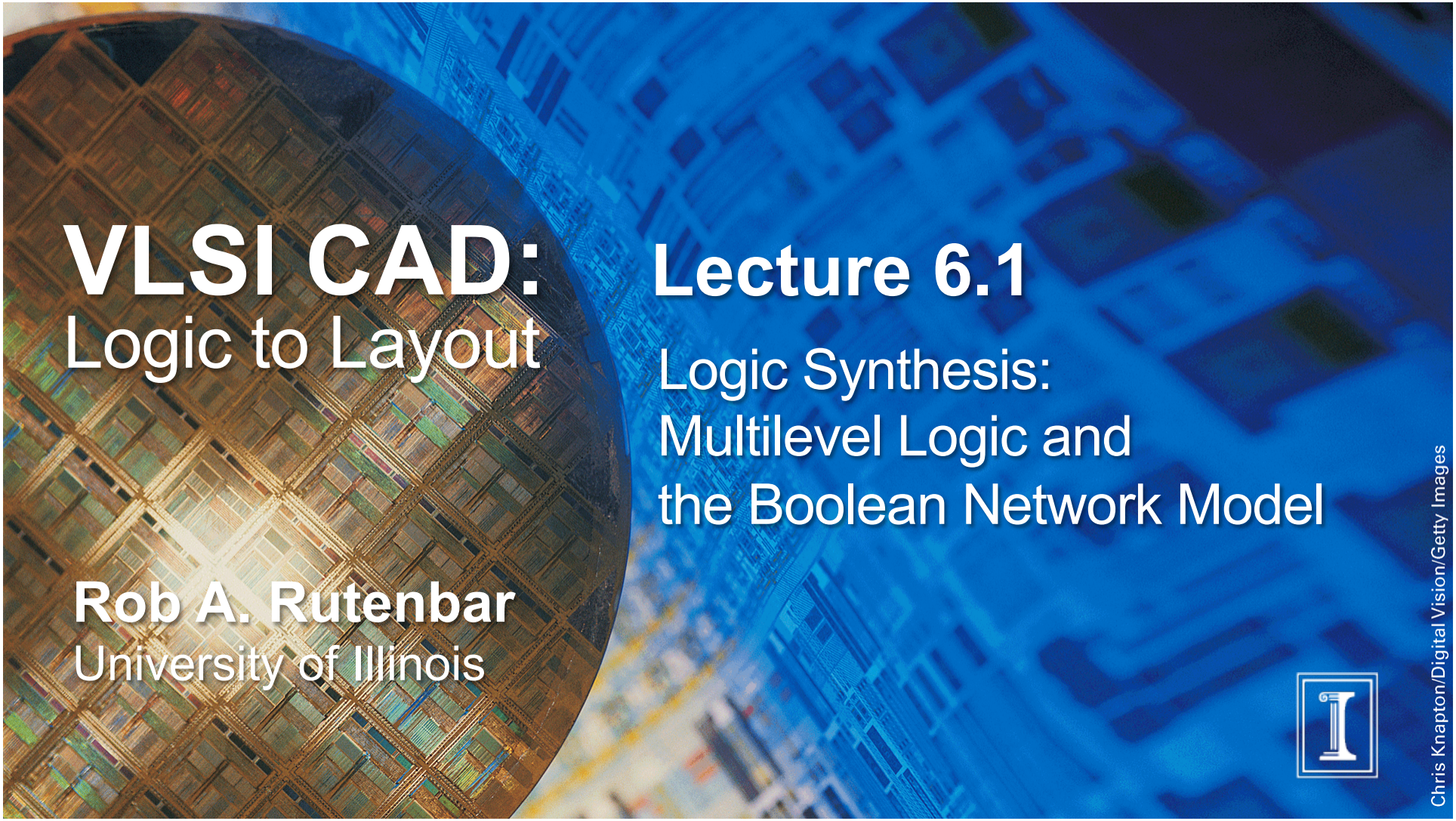


VLSI CAD: Logic to Layout

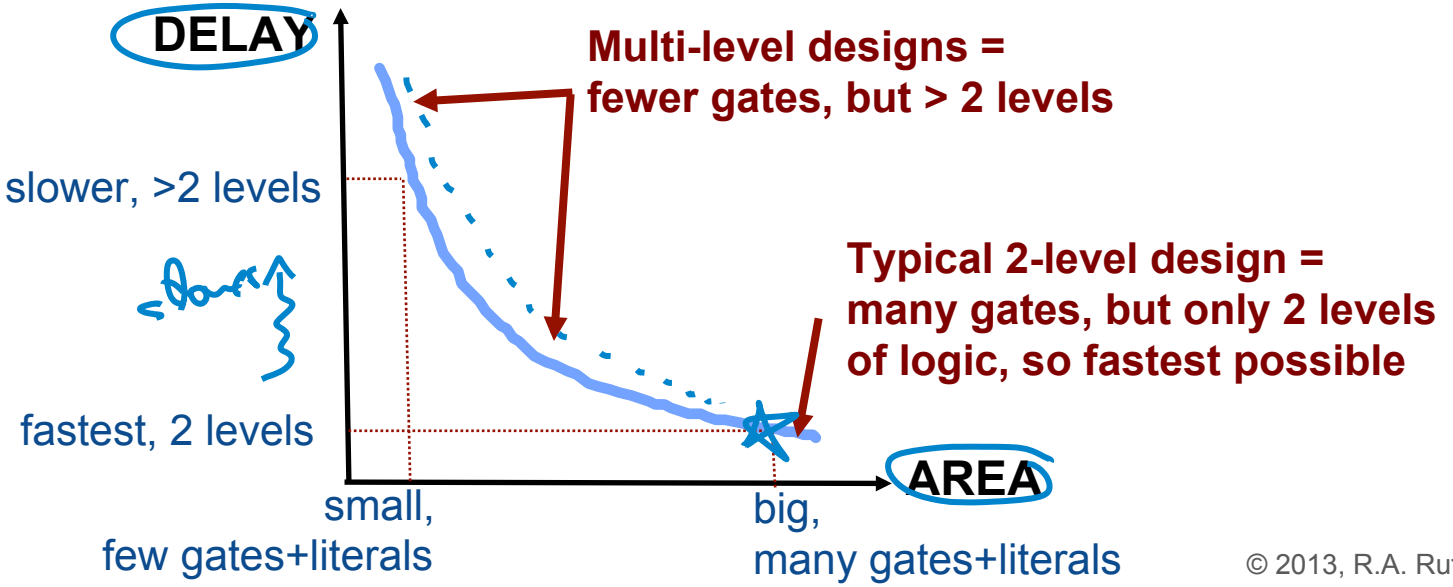
Lecture 6.1

Logic Synthesis: Multilevel Logic and the Boolean Network Model

Rob A. Rutenbar
University of Illinois

Why *Multi*-level Logic?

- **2-level forms are too *restrictive*: specific area vs. delay tradeoff**
  - **Area** = gates + literals (wires), i.e., things that take space on a chip
  - **Delay** = maximum levels of logic gates required to compute function
  - 2-level is *minimum* gate delay possible, but usually *worst* on area

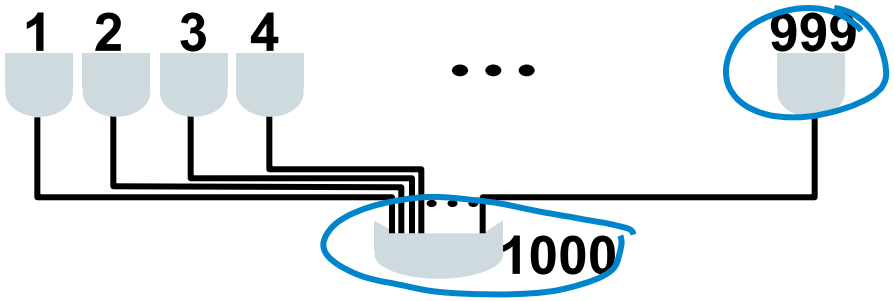


Why Multilevel?

- **Rarely see 2-level designs for really big things…**
  - We use 2-level logic mostly for **pieces** of bigger things
  - Even smallish things routinely done as multi-level

~1000 gate "block" of logic

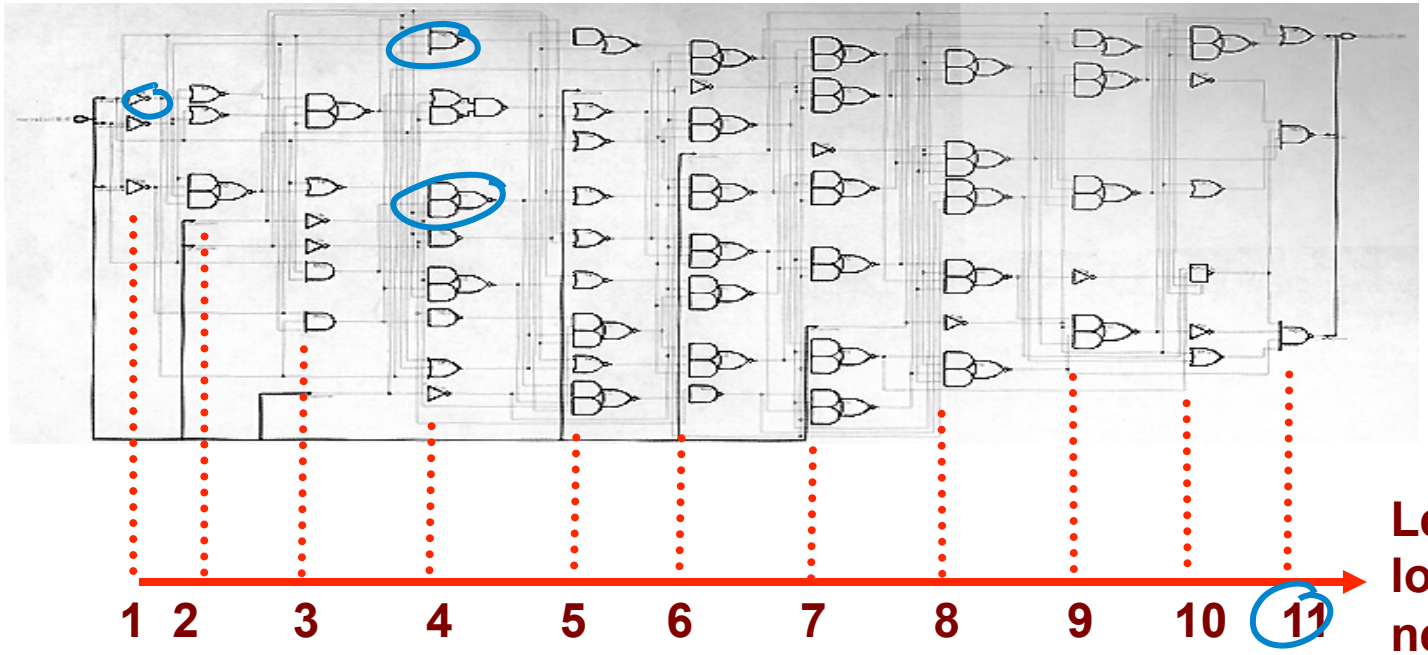
?



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

Real Multilevel Example

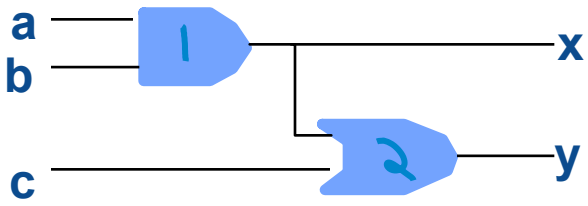
- **…this is a *small* design, done by commercial synthesis tool**



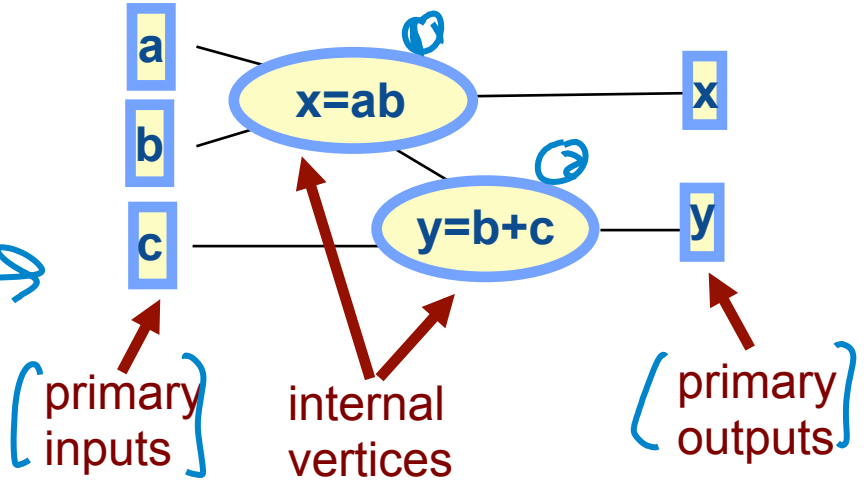
Boolean Logic Network Model

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

Ordinary gate logic

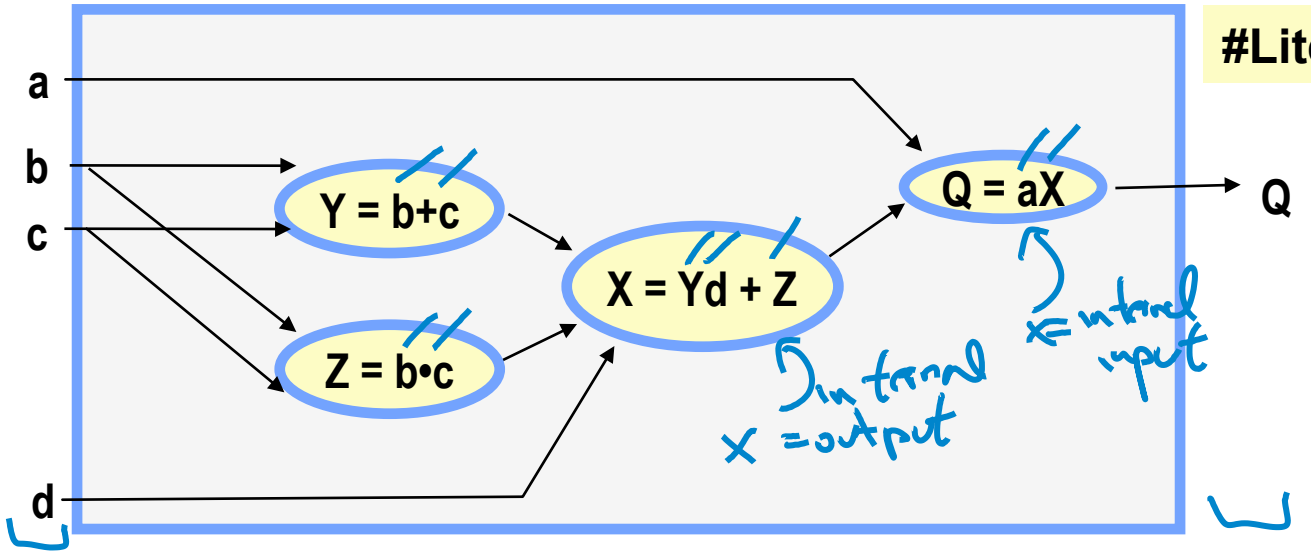


…now as a **Boolean logic network**, **x, y** are now Boolean functions



Multilevel Logic: What to Optimize?

- **This is simplistic but surprisingly useful: Total literal count**
  - Count *every* appearance of *every* variable on right hand side of "=" in *every* node
  - (Yes, delays matter too, but for this class, only focus on logic *complexity*)



#Literals = 9

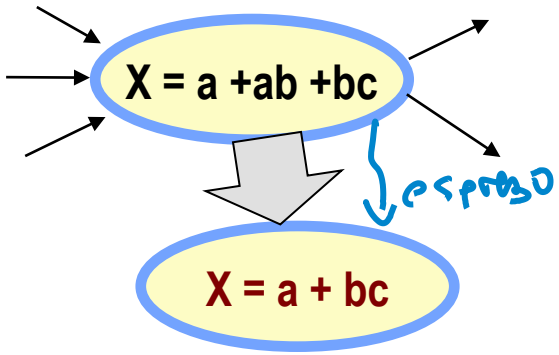
Optimizing Multilevel Logic: Big Ideas

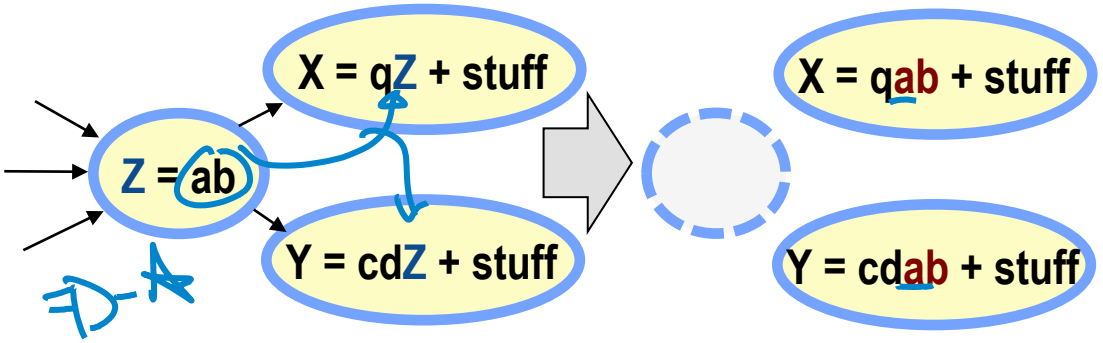
- **Again: this is a *data structure*. What *operators* do we need?**

- **3 basic kinds of operators**
  - **Simplify network nodes:** no change in # of nodes, just simplify insides, SOP form
    - → You already know this! This is **2-level synthesis**, this is ESPRESSO!
  - **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 need to teach you…

Optimizing Boolean Logic Network

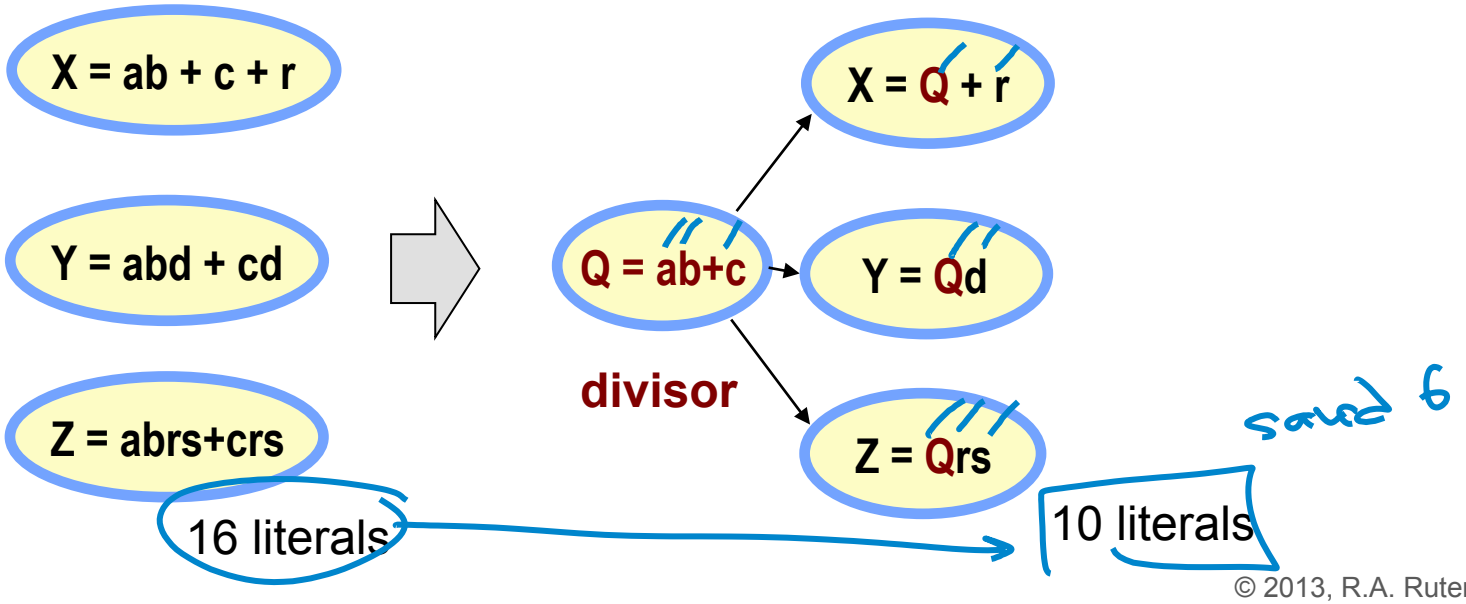
- **Simplifying a node**
  - Just run ESPRESSO on 2-level form **inside** the node, to reduce # literals
  - As structural changes happen across network, "insides" of nodes may present opportunity to simplify
- **Removing a node**
  - Typical case is you have a "small" factor which doesn't seem to be 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

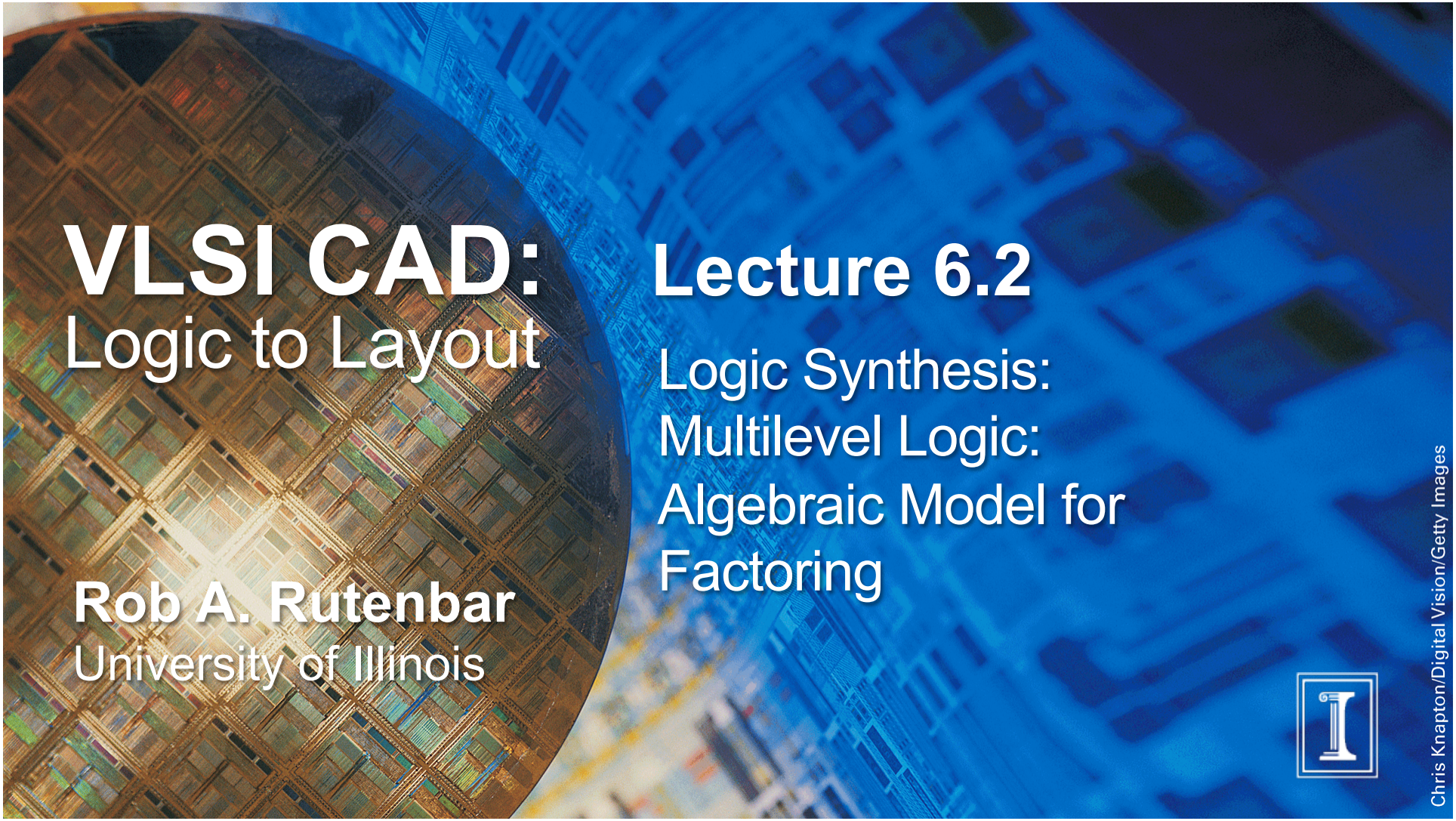




Optimizing Boolean Logic Network

- **Adding new node(s): 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)





VLSI CAD: Logic to Layout

Lecture 6.2

Logic Synthesis: Multilevel Logic: Algebraic Model for Factoring

Rob A. Rutenbar
University of Illinois

Aside: 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 them)
  - Look for "easy" factors, just sitting around as existing nodes, and try to use them
  - Look for "hard" factors, do the work to extract them, try them, keep the good ones
  - Do 2-level optimization of insides of each logic node in network (ESPRESSO)
  - Lots of "art" in the engineering of these scripts
- **For us, the one big thing you don't know: How to factor…**

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**
  - **Surprise**: Term "algebraic" comes from pretending that Boolean expressions behave like **polynomials of real numbers**, not like Boolean algebra
  - Big new Boolean operator: **Algebraic Division** (or, also **"Weak" Division**)

Algebraic Model

- **Idea: keep just those rules (axioms) that work for polynomials of reals AND Boolean algebra, *dump the rest***

Real numbers

$a \cdot b = b \cdot a$
$a + b = b + a$
$a \cdot (b \cdot c) = (a \cdot b) \cdot c$
$a + (b + c) = (a + b) + c$
$a \cdot (b + c) = a \cdot b + a \cdot c$
$a \cdot 1 = a \quad a \cdot 0 = 0$
$a + 0 = a$

SAME

Boolean algebra

$a \cdot b = b \cdot a$
$a + b = b + a$
$a \cdot (b \cdot c) = (a \cdot b) \cdot c$
$a + (b + c) = (a + b) + c$
$a \cdot (b + c) = a \cdot b + a \cdot c$
$a \cdot 1 = a \quad a \cdot 0 = 0$
$a + 0 = a$

X

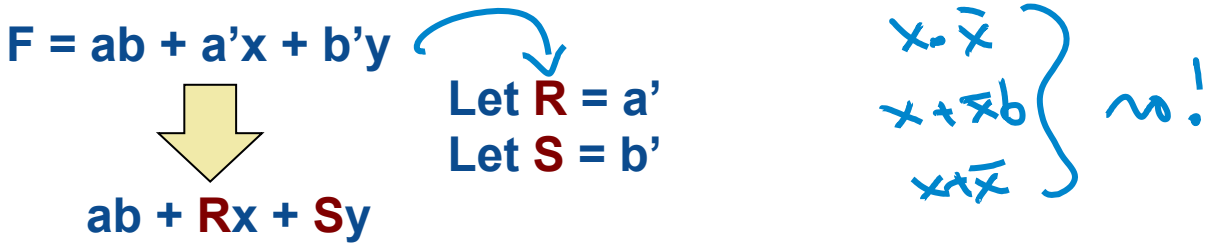
NOT ALLOWED

$a + a' = 1$
$a \cdot a = a$
$a + 1 = 1$
$a + (b \cdot c) = (a + b) \cdot (a + c)$

$a \cdot a' = 0$
$a + a = a$

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***
  - **Aside**: this is one of the losses of "expressive power" for Booleans we just tolerate



- **Idea**
  - Boolean functions manipulated as **SOP expressions–like polynomials**
  - Each product term in such an expression is just a **set of variables,** e.g., **abRy**
  - SOP expression itself is just a **list of these products (cubes)**, e.g. **ab + Rx = ab,Rx**

Algebraic Division: Our Model for Factoring

- **Given function D we want to factor as:**

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

divisor — quotient — remainder (if =0, then we say the divisor is 'technically' called a *Factor*)

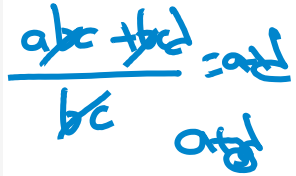
Example with REAL

15 = 7•2 + 1
7 = D = divisor
2 = Q = quotient
1 = R = remainder

Example with BOOLEAN SOP

F = ac + ad + bc + bd + e
= (a+b)•(c+d) + e

(a+b) = D = divisor
(c+d) = Q = quotient
e = R = remainder



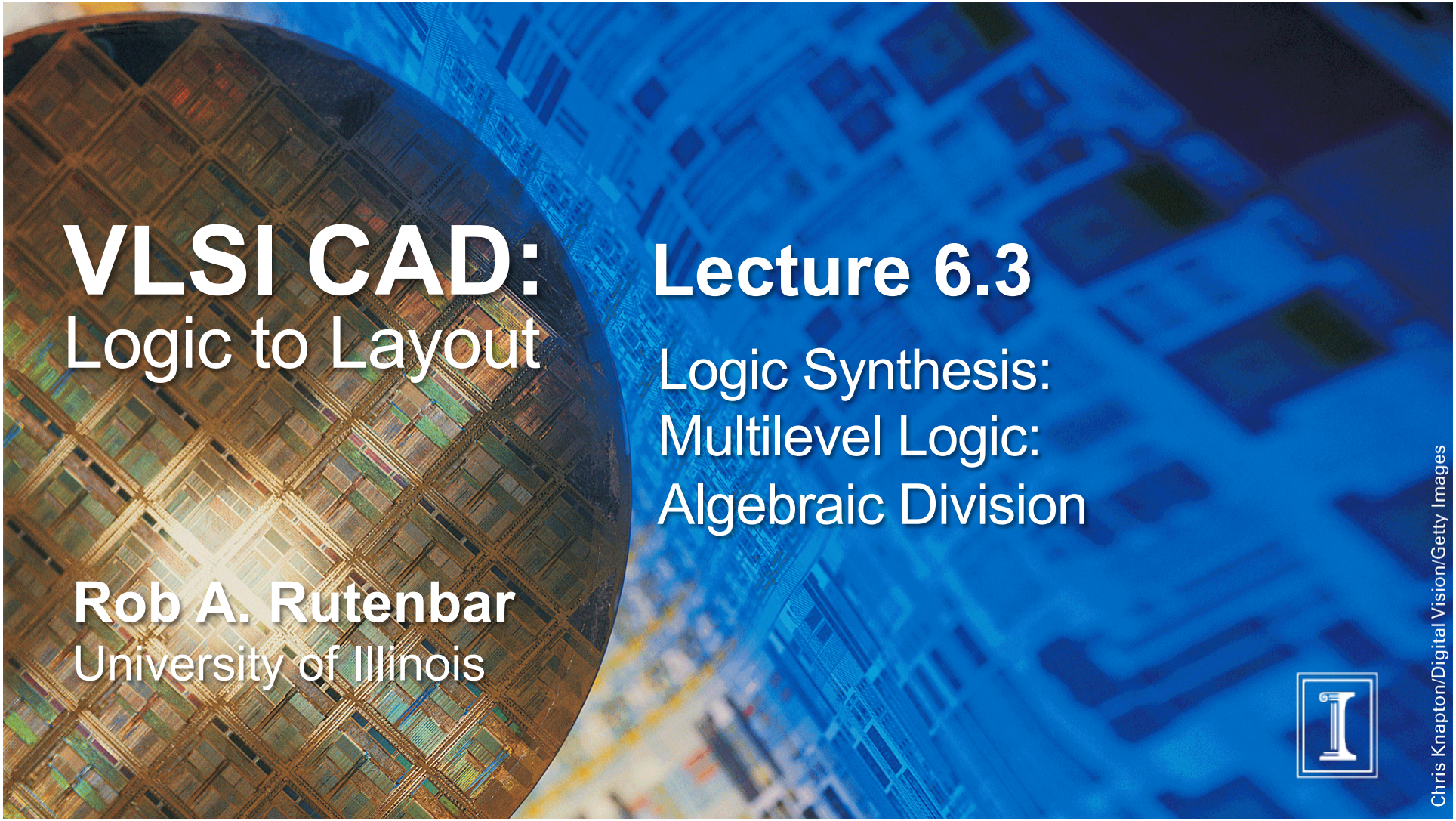
Algebraic Division

- **Example** **F = ac + ad + bc + bd + e** want **F = D • Q + R**
  - **Reminder**: **Quotient** is only called a **"Factor"** if remainder is **0**

F ÷ D

Divisors (D)	Quotient (D)	Remainder (R)	Factor?
ac+ad+bc+bd+e	1	0	Yes (trivial)
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

★ How to do it?



VLSI CAD: Logic to Layout

Lecture 6.3

Logic Synthesis: Multilevel Logic: Algebraic Division

Rob A. Rutenbar
University of Illinois

Algebraic Division: Very Nice Algorithm

- **Inputs**
  - A Boolean expression **F** and a divisor (to divide by) **D**, represented as lists of cubes (and each cube a set of literals)
- **Output**
  - Quotient **Q = F/D** = cubes in quotient, or **0 (empty)** if none
  - Remainder **R** = cubes in remainder, or **0 (empty)** if **D** was a factor
  - i.e., figures out **Q, R** so that **F = D•Q+ R = D•(F/D) + R**
- **Strategy**
  - Cubewise walk thru cubes in divisor **D**, trying to divide each of them into **F**
  - ...being careful to track which cubes ***do*** divide into **F**

Algebraic Division Algorithm

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

   for ( each cube d in divisor D ) {
      let C = { cubes in F 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 ∩ C;
   }
   R = F - ( Q • 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

Example:
Suppose Q = xyz + yzw +pqyz and C = "xyz". Then Q ∩ C is just the cubes that are in ***both*** Q and C in this case: xyz

Algebraic Division: Example

F/D: F = axc + axd + axe + bc + bd + de **D = ax + b**

F cube	D cube: ax C = … (1)	D cube: b C = … (3)
axc	axc→~~a~~xc→c	--
axd	axd→~~a~~xd→d	--
axe	axd→~~a~~xe→e	--
bc	--	bc→~~b~~c→c
bd	--	bd→~~b~~d→d
de	--	--
	(2) Q = c+d+e	(4) Q = (c+d) ∩ (c+d+e) = c+d

Easiest way manually is to make this table: one row per cube in F, one column per cube in D, bottom row to evolve Quotient Q and, when done, remember to get remainder R

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

   for ( each cube d in divisor D ) {
      let C = { cubes in F 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 ∩ C;
   }
   R = F - ( Q • D );
   return (  quotient = Q,  remainder = R)
}
```

Remainder **R = F–Q•D:**
"—" means remove cubes in **Q•D** that appear *same* in **F**

R = (axc + axd + axe + bc + bd + de) – (**c+d**)•(ax+b) (5)
(~~axc~~ + ~~axd~~ + axe + ~~bc~~ + ~~bd~~ + de) – (~~axc~~ + ~~axd~~ + ~~bc~~ + ~~bd~~)

= (axe + de) = remainder = R

Algebraic Division: Warning

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

    F = ab'c' + ab + ac + bc **G = ab + c'** **want F / G**

  - You must transform it to something like this...

    Let X=b', Y=c'
    → F = aXY + ab + ac + bc **G = ab + Y** ***then*** **can do F / G**

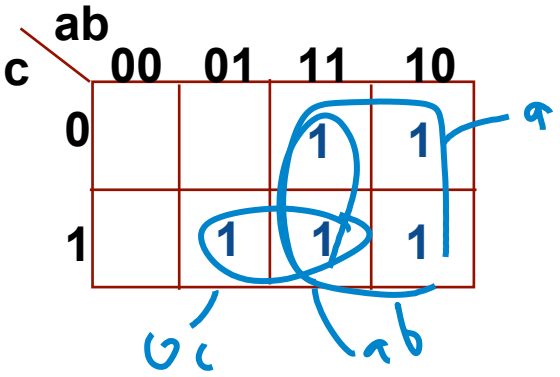
  - Because **must** treat the true and complement forms of variable as **different**

One More Constraint: Redundant Cubes

- **To do F/D, function F must have *no redundant* cubes**
  - Technical term is: **"minimal with respect to single-cube containment"**
  - **In words:** no one cube is ***completely*** covered by other cubes in SOP cover

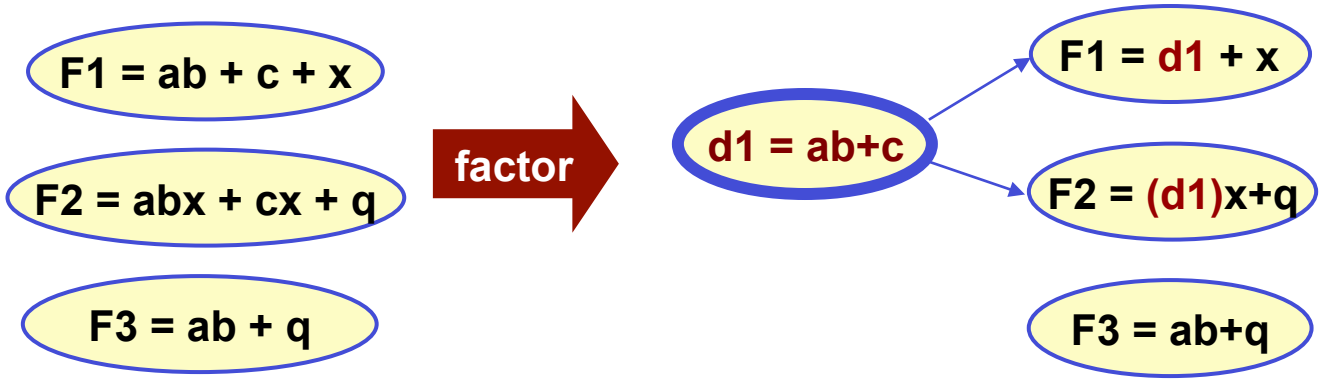
F = a + ab + bc is redundant quotient
D = a is the divisor
Now: compute **F / D**, i.e., **F / a**
use our algebraic division algorithm

→ Problem: **F / D = 1 + b,** remainder = **bc**
→ **"1+b"** is **not** operation in Algebraic model!

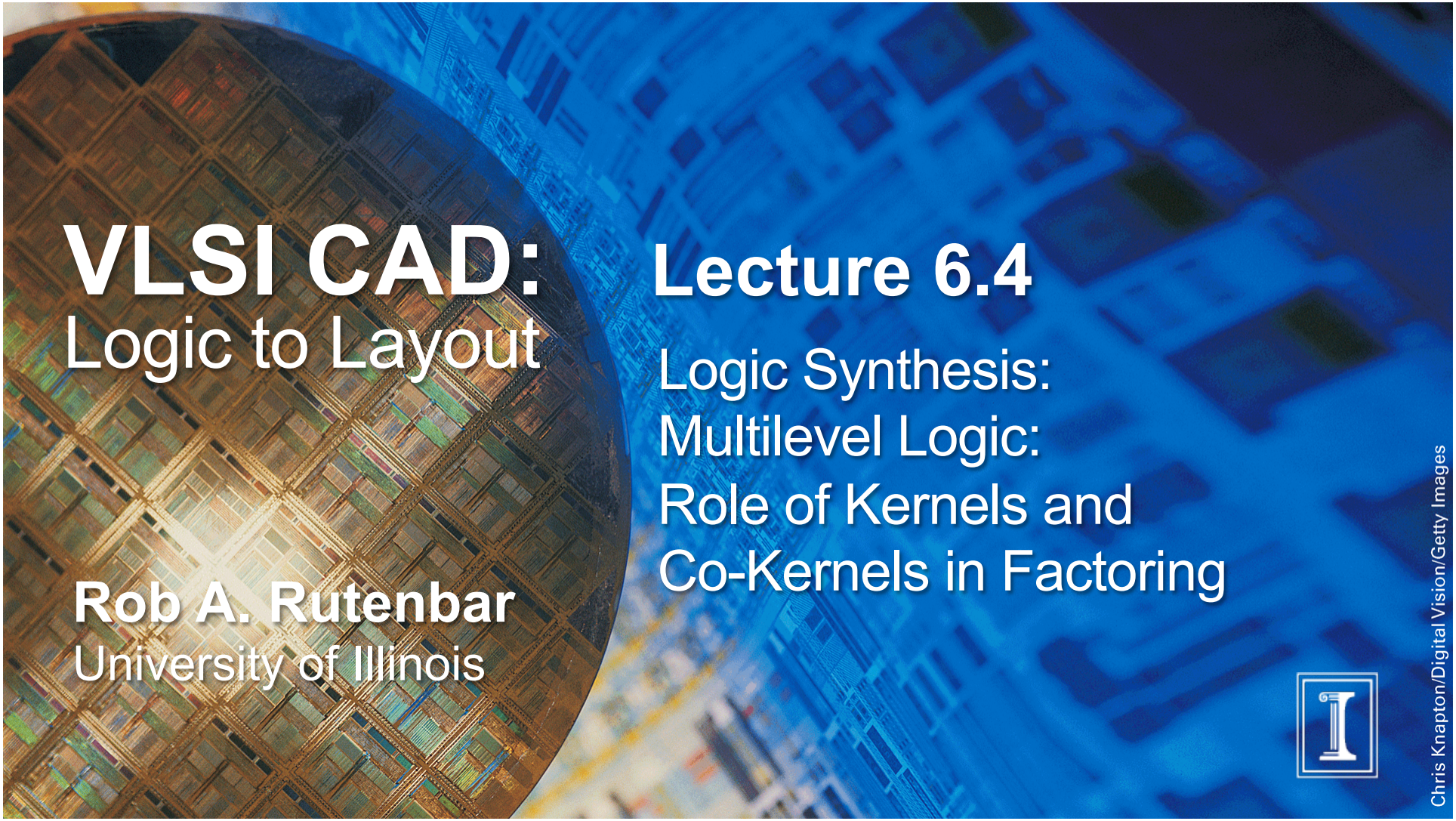


Multilevel Synthesis Models: Where are We?

- **For Boolean F, D, can compute F = Q•D + R via algebraic model**
  - This is great—but it's still not enough. Don't know how to **find** these divisors.
  - Real problem: **n** functions **F1, F2, … Fn**, find a **set of good common divisors di**



- **What are we looking for?**
  - **Case 1:** divisors **d** that are just 1 cube (1 product term), e.g., **d = ab**
  - **Case 2:** "bigger" multiple-cube divisors, e.g. **d = ab + cd + e**



VLSI CAD: Logic to Layout

Lecture 6.4

Logic Synthesis:
Multilevel Logic:
Role of Kernels and
Co-Kernels in Factoring

Rob A. Rutenbar
University of Illinois

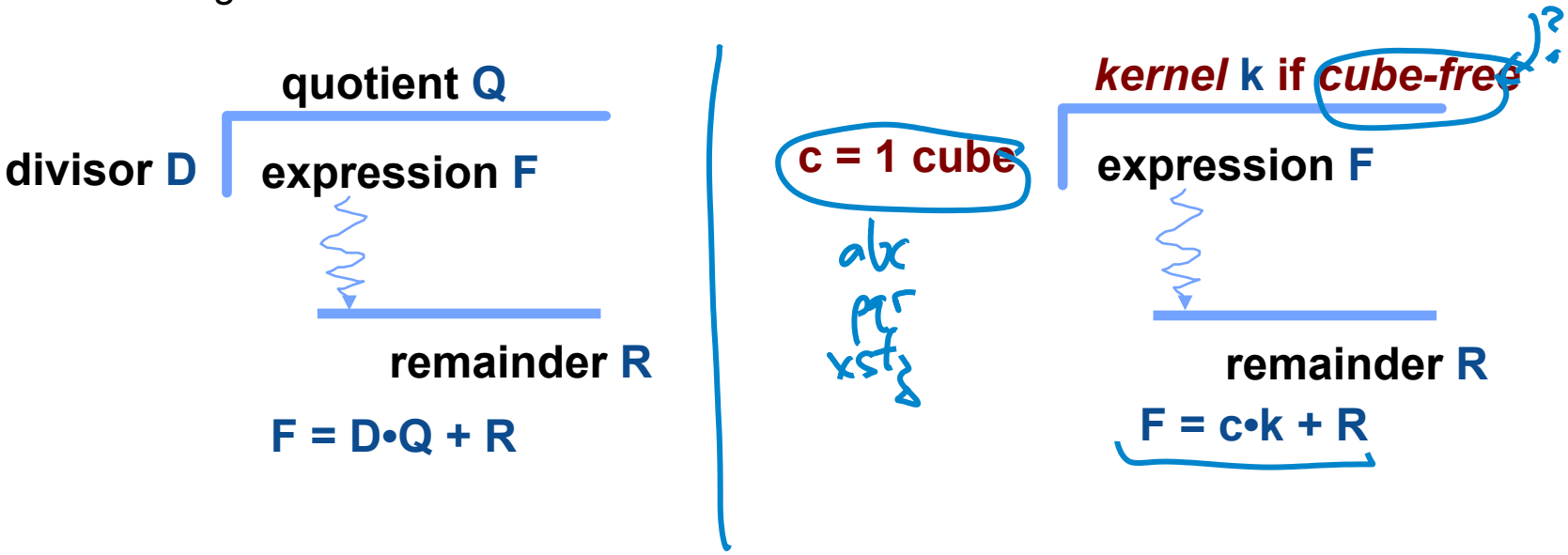
Where To Look For Good Divisors?

- **Surprisingly, the Algebraic Model has a *beautiful* answer**
  - One more reason we like it: Has some surprising and elegant "deep structure"

- **Where to look for divisors of function F?** **In the *kernels* of F**
  - Denoted **K(F)**. Pronounced like "Colonel." Spelled with TWO e's (remember!)
  - **K(F)** is another set of two-level SOP forms which are the special, foundational structure of any function **F,** being interpreted in our algebraic model.
- **How to find a kernel k∈K(F)? Algebraically divide F by one of its *co-kernels*, c.**
  - Wow – lots of new concepts and terms here. Let us go develop these ideas…

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**



Kernels Are *Cube-Free*…

- **Cube-free means...?**
  - You cannot factor out a single cube (product term) divisor that leaves no remainder
  - Technically -- has no **one cube (product)** that is a **factor** of expression
  - Do **F / cube**, look at result, if you can 'cross out' some cube in **each** term, **not** a kernel

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

Some Kernel Examples

- **Kernels of F denoted K(F). Suppose F = abc + abd + bcd**

Divisor cube d	F= d•Q + R	Is it a Kernel of f?
1	(1)(abc+abd+bcd)+0	No, has cube = b as factor
a	(a)(bc+bd)+bcd	No, also has cube = b as factor
b	(b)(ac+ad+cd)+0	Yes, a kernel; co-kernel = (b)
ab	(ab)(c+d)+bcd	Yes, a kernel; co-kernel = (ab)
… etc		

- **So, any Boolean F can have many different kernels k∈K(F)**

Kernels: Why Are They Important?

- **…, if they are important, how do we actually compute them?**
- **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.

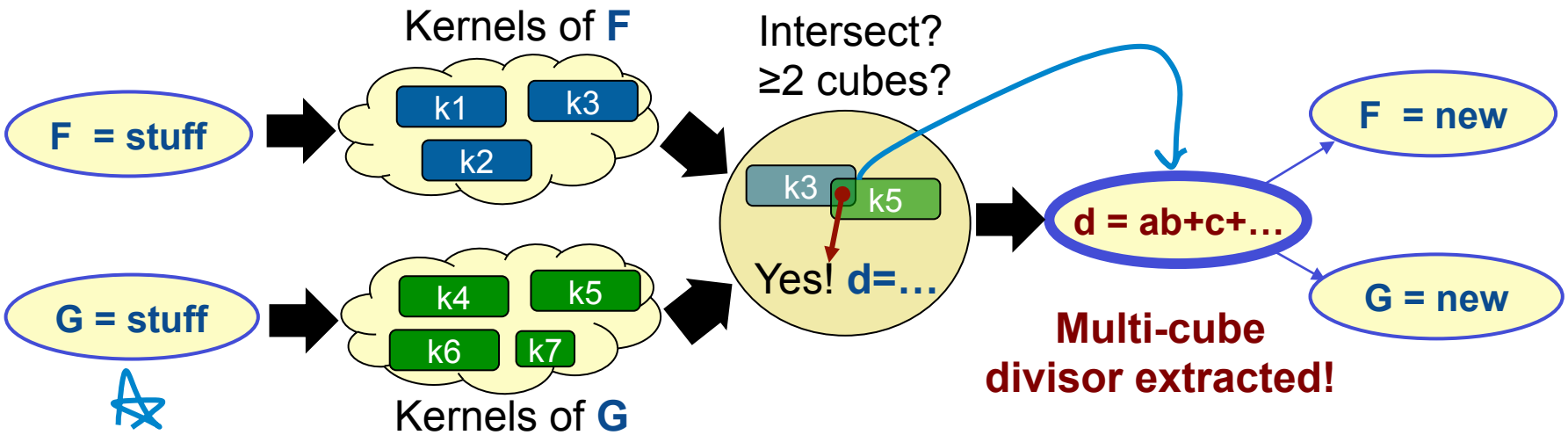
Expressions **F, G** have a **common multiple-cube divisor d**

if and only if

there are kernels **k1 ∈ K(F), k2 ∈ K(G)**
such that **d = k1 ∩ k2** (ie, SOP form with **common cubes** in it)
and **d** is an expression with at least **2** cubes in it

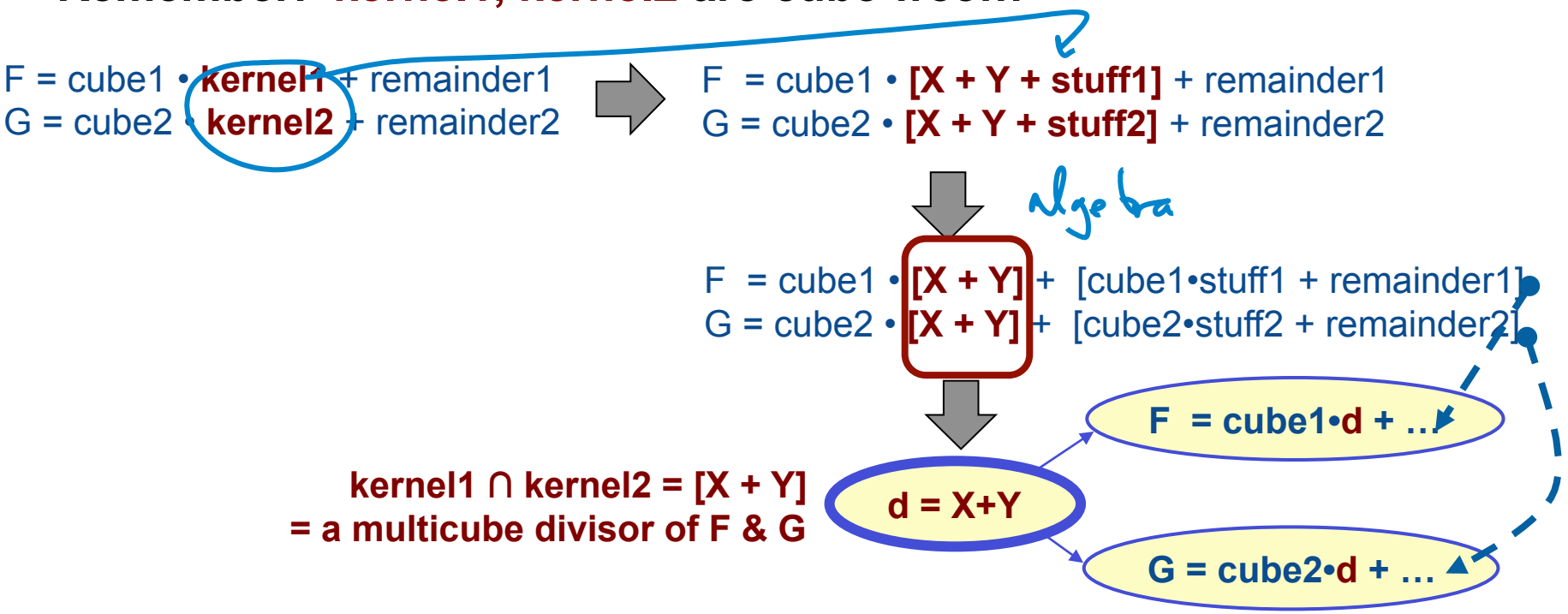
Multiple-Cube Divisors and Kernels

- **In words:**
  - The ***only*** place to look for multiple-cube divisors is in the **intersection of kernels**!
  - Further: this intersection of kernels **is** the divisor, there are **no** others



Brayton-McMullen: Informal Illustration

- **Remember: kernel1, kernel2 are cube-free…**



Kernels: Real Example

- **Consider this F, G**

F = ae + be + cde + ab

K(F) Kernel	Co-kernel
a+b+cd	e
b+e	a
a+e	b
ae+be+cde+ab	1

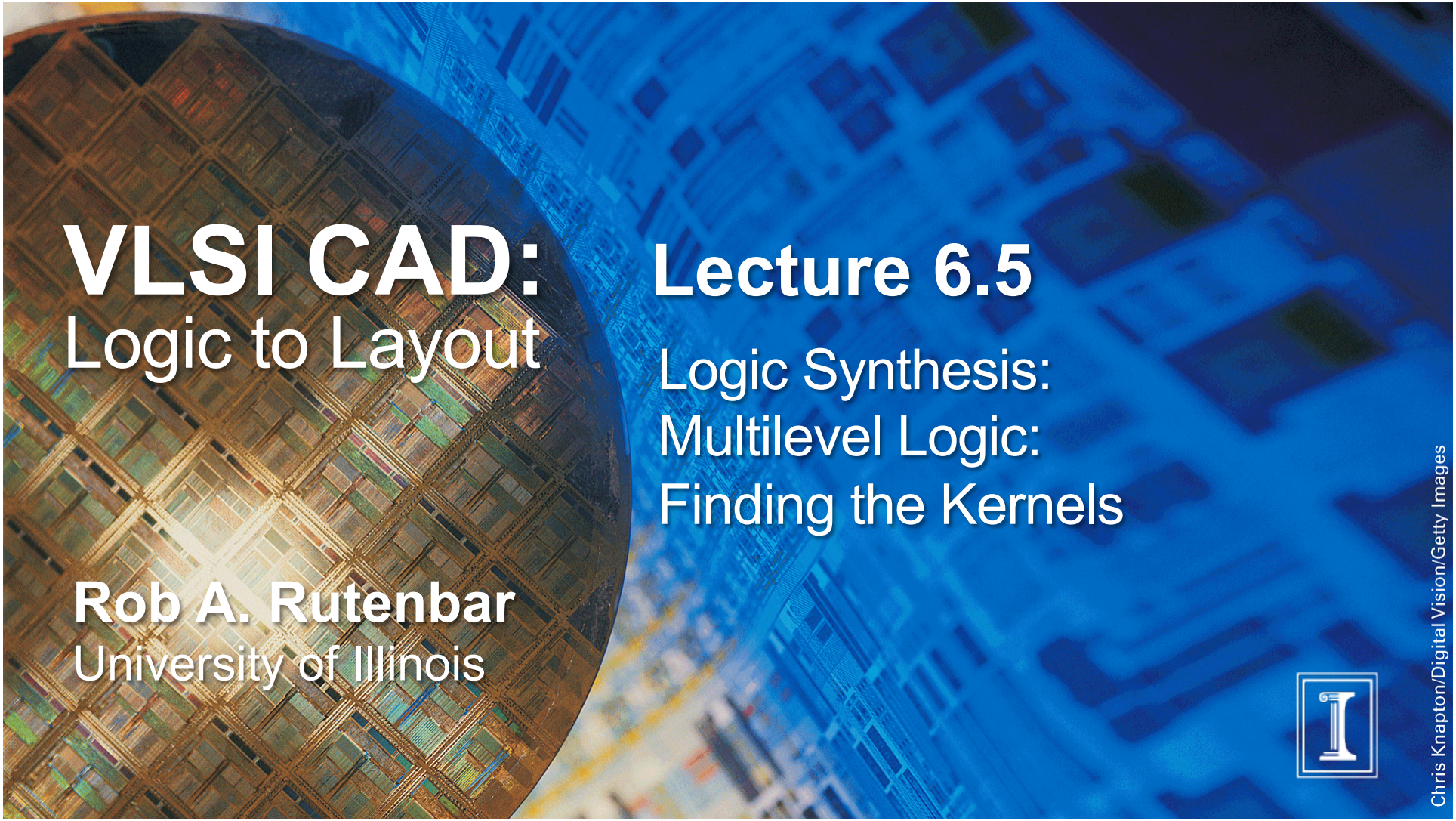
G= ad + ae + bd + be + bc

K(G) Kernel	Co-kernel
a+b	d or e
d+e	a or b
d+e+c	b
ad+ae+bd+be+bc	1

Intersecting these 2 kernels: **(a+b+cd) ∩ (a+b) =** a+b

So, this is workable **multicube divisor** we can consider for both **F,G**

How to find kernels?



VLSI CAD: Logic to Layout

Lecture 6.5

Logic Synthesis: Multilevel Logic: Finding the Kernels

Rob A. Rutenbar
University of Illinois

Kernels: Very Useful, But How To Find?

- **Another recursive algorithm (are we surprised...?)**
  - 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 = cube1 • k1 + 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)…

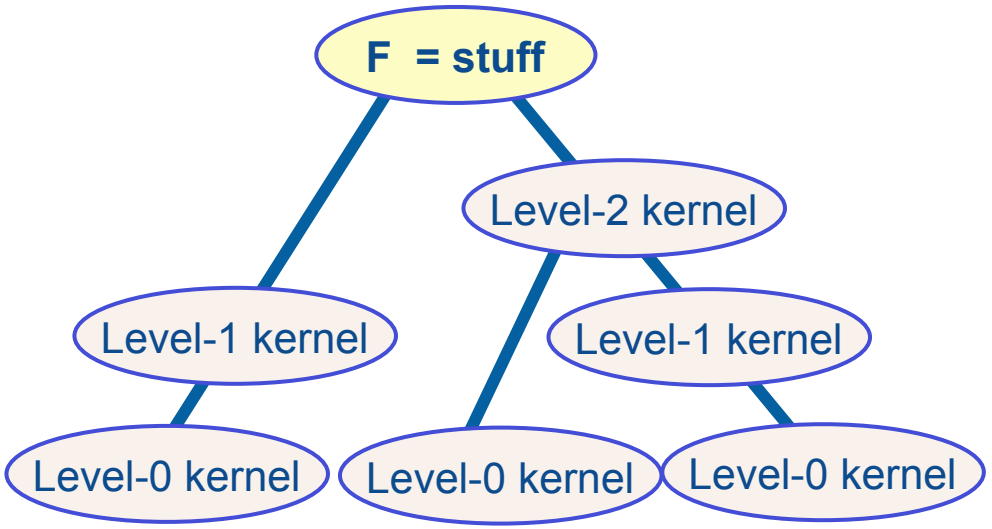
- **We know this: F = cube1•k1 + remainder1**
  - Suppose **k2** is a kernel in **K(k1)**, then we know
    - **k1 = cube2•k2 + remainder2**
  - Substitute this expression for **k1** in original expression for **F**
    - **F = cube1•[cube2•k2 + remainder2] + remainder1**
  - Neat trick: **cube1•cube2** is itself just another ***single cube***, rewrite to emphasize:
    - **F = (cube1•cube2)•[k2] + [cube1•remainder2 + remainder1]**

= (a cube) • [cube free quotient] + [other stuff]

  - **Lovely result:** **k2** also a ***kernel*** of original **F** (with co-kernel **cube1•cube2**)

There is a *Hierarchy* of Kernels Inside F

- **Terminology: k∈K(F) is:**
  - A ***level-0 kernel*** if it contains no kernels inside it except itself
    - **In words:** Only cube you can pull out, 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

- **2nd useful result [Brayton et al]**
  - **Co-kernels** of a Boolean expression in SOP form correspond to **intersections of 2 or more of the cubes** in this constituent SOP form
- **Note: *Intersections* here means specifically 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 is simple common sub-expressions.
  - Example **ace + bce + de + g**

ace ∩ bce = ce → **ce is a potential co-kernel**

ace ∩ bce ∩ de = e → **e is a potential co-kernel**

Kernel Hierarchy

- **How do we use these 2 results?**
  - Find the kernels **recursively.** Whenever we find one:
    - Call **FindKernels( )** on it, to find (if any) lower level kernels are inside
  - Use **algebraic division** to divide function by potential co-kernels, to drive recursion
    - …but be smart: co-kernels are *intersections* of the cubes
    - ...if there's 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 the biggest common cube to simplify **F**…

Kernel Algorithm

- **Algorithm is then...**

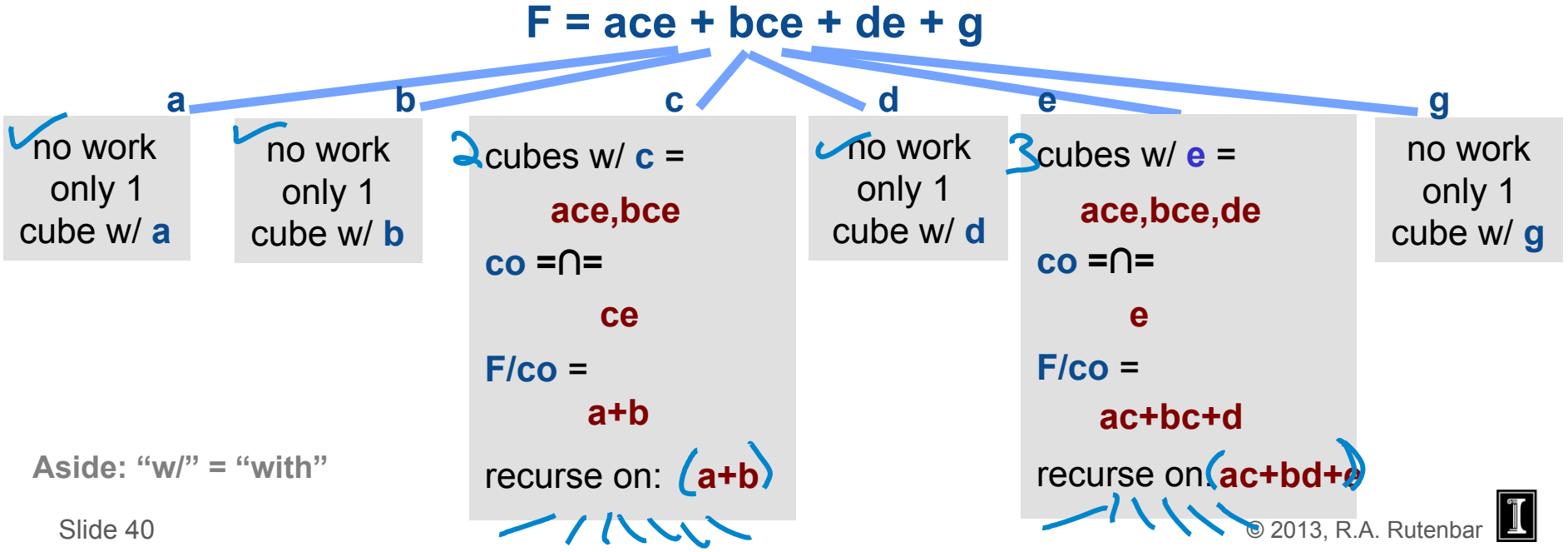
```
FindKernels( cube-free SOP expression F) {
    K = empty;
    for ( each variable x in  F  ) {
        if ( there are at least 2 cubes in F that have variable x ) {
            let S = { cubes in F that have variable x in them };
            let co =  cube that results from intersection of all cubes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

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

F/co = a kernel, returned at end. This is algebraic division, but simpler since it always just divides by **exactly 1 cube, a simple product term**

Kerneling Example

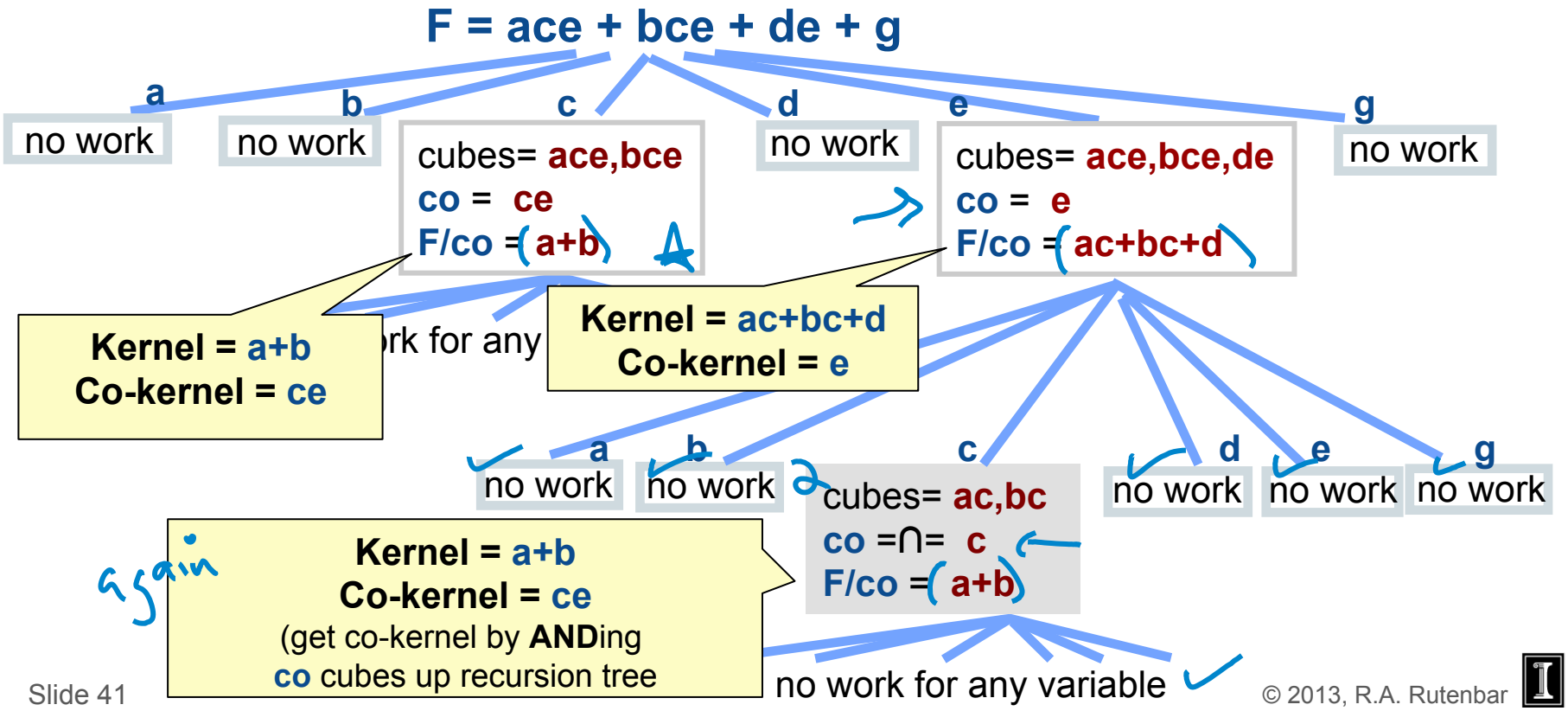
- **To start, divide F by each of the variables, and use to recurse**
  - We are looking for co-kernels that start with this **ONE** variable in them
  - But—be smart, it cannot be a co-kernel unless it's in at least 2 cubes



Aside: "w/" = "with"

Kernel Hierarchy, Example Continued

- **With this algorithm, overall recursion tree looks like this**

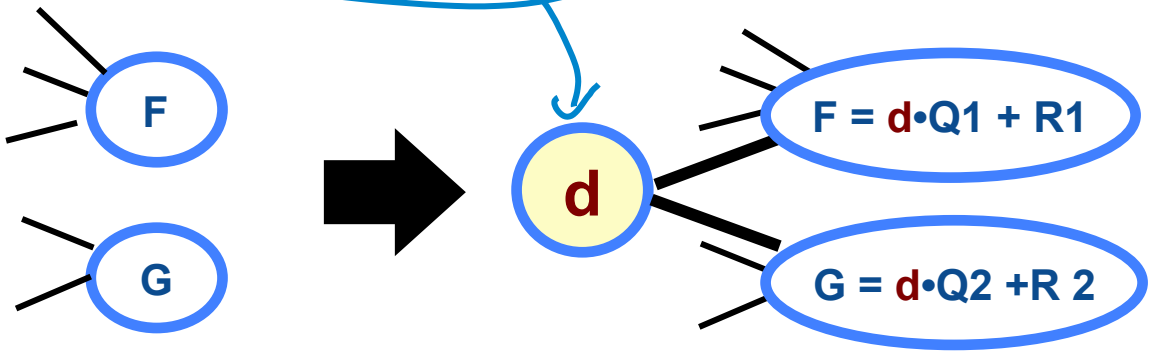


Kernel Hierarchy

- **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**
  - Will revisit *same* kernel ***multiple*** times 
- **Solution: *remember* which variables already tried in co-kernels**
  - **Problem**: 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 cubes
  - A little extra book keeping solves this

Using Kernels and Co-Kernels

- **These are *exactly* the right component pieces for...**
  - Extraction of a **single-cube divisor** from multiple expressions
  - Extraction of a **multiple-cube divisor** from multiple expressions



- When you want a single-cube divisor: **Go look for co-kernels**
- When you want a multiple-cube divisor: **Go look for kernels**

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 real numbers
  - Divides one Boolean function by another: **F = (divisor D)•(quotient Q) + remainder R**
- **Kernels / Co-kernels of a function F**
  - ***Kernel*** = cube-free quotient obtained by dividing by a single cube (***co-kernel***)
  - Intersections of **F, G** kernels → all multiple-cube common divisors (Brayton-McMullen)
- **Next: what are *best* common divisors to get, given these ideas?**

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 to read about all these ideas**
  - 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.