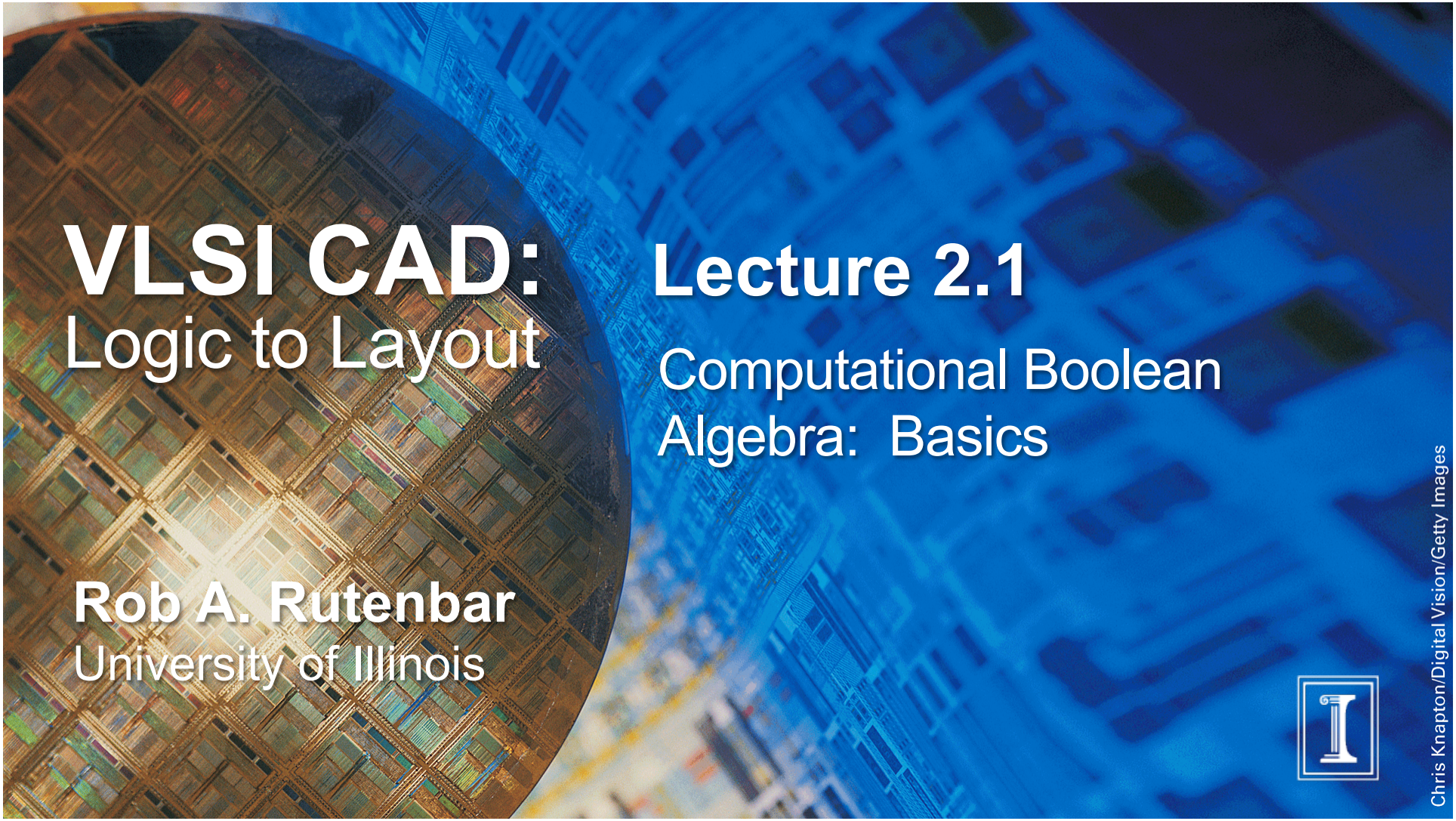


VLSI CAD: Logic to Layout

Lecture 2.1

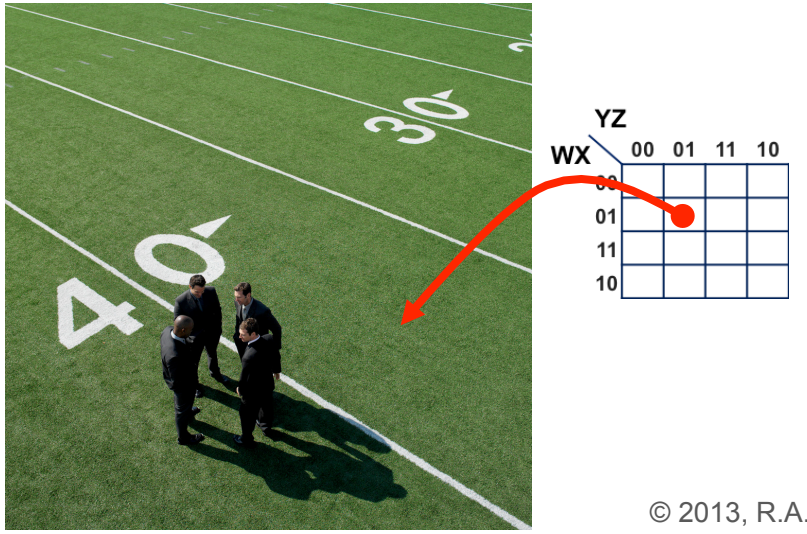
Computational Boolean Algebra: Basics

Rob A. Rutenbar
University of Illinois

Chris Knapton/Digital Vision/Getty Images

Computational Boolean Algebra…?

- **Background…**
  - You've done Boolean algebra, hand manipulations, Karnaugh maps to simplify…
  - But this is **not sufficient** for real designs
- **Example: simplify Boolean function of 40 variables via Kmap**
  - It has **1,099,511,627,776** squares
  - You could fit this on an American style football field...
  - …but each Kmap square would be just **60 x 60** microns!
  - There must be a **better** way…



© 2013, R.A. Rutenbar

Need a *Computational* Approach

- **Need *algorithmic, computational* strategies for Boolean stuff**
  - Need to be able to think of Boolean objects as **data structures + operators**
- **What will we study?**
  - **Decomposition strategies**
    - Ways of taking apart complex functions into simpler pieces
    - A set of advanced concepts, terms you need to be able to do this
  - **Computational strategies**
    - Ways to think about Boolean functions that let them be manipulated by programs
  - **Interesting applications**
    - When you have new tools, there are some useful new things to do

© 2013, R.A. Rutenbar

Advanced Boolean Algebra

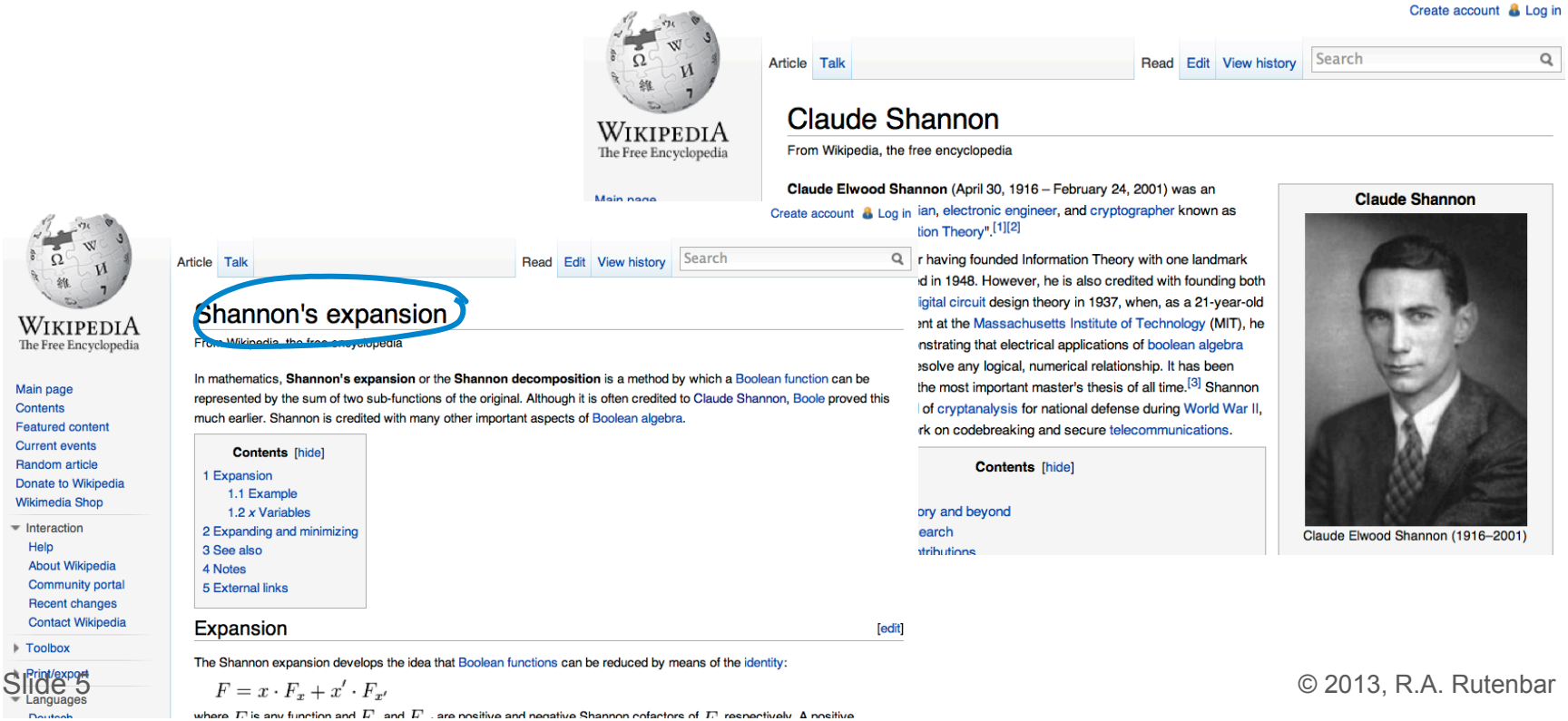
- **Useful analogy to calculus…**
  - You can represent complex functions like **exp(x)** using simpler functions
    - If you only get to use **$1, x, x^2, x^3, x^4, \ldots$** as the pieces...
    - ...turns out **$\exp(x) = 1 + x + x^2/2! + x^3/3! + \ldots$**
  - In Calculus, we tell you the general formula, the **Taylor series expansion**
    - **$f(x) = f(0) + f'(0)/1!\, x + f''(0)/2!\, x^2 + f'''(0)/3!\, x^3 + \ldots$**
  - If you take more math, you might find out several other ways:
    - If it's a periodic function, can use a **Fourier series**

- **Question: *Anything like this for Boolean functions?***

© 2013, R.A. Rutenbar

Boolean Decompositions

- **Yes. Called the *Shannon Expansion***



Create account Log in

WIKIPEDIA
The Free Encyclopedia

Article Talk Read Edit View history Search

Claude Shannon

From Wikipedia, the free encyclopedia

Claude Elwood Shannon (April 30, 1916 – February 24, 2001) was an ian, electronic engineer, and cryptographer known as tion Theory".[1][2]

r having founded Information Theory with one landmark d in 1948. However, he is also credited with founding both igital circuit design theory in 1937, when, as a 21-year-old nt at the Massachusetts Institute of Technology (MIT), he nstrating that electrical applications of boolean algebra solve any logical, numerical relationship. It has been the most important master's thesis of all time.[3] Shannon of cryptanalysis for national defense during World War II, rk on codebreaking and secure telecommunications.

Contents [hide]

ory and beyond
earch
tributions

Claude Shannon

Claude Elwood Shannon (1916–2001)

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages

Article Talk Read Edit View history Search

Shannon's expansion

From Wikipedia, the free encyclopedia

In mathematics, **Shannon's expansion** or the **Shannon decomposition** is a method by which a Boolean function can be represented by the sum of two sub-functions of the original. Although it is often credited to Claude Shannon, Boole proved this much earlier. Shannon is credited with many other important aspects of Boolean algebra.

Contents [hide]

1 Expansion
1.1 Example
1.2 *x* Variables
2 Expanding and minimizing
3 See also
4 Notes
5 External links

Expansion [edit]

The Shannon expansion develops the idea that Boolean functions can be reduced by means of the identity:

$$F = x \cdot F_x + x' \cdot F_{x'}$$

where F is any function and F_x and $F_{x'}$ are positive and negative Shannon cofactors of F, respectively. A positive

© 2013, R.A. Rutenbar

Shannon Expansion

- **Suppose we have a function F(x1,x2, ..., xn)**
- **Define a new function if we set one of the xi=*constant***
  - Example: **F(x1, x2, ..., xi=1, ..., xn)**
  - Example: **F(x1, x2, ..., xi=0, ..., xn)**
- **Easy to do one by hand**

F(x,y,z) = xy + xz' +y(x'z +z')

F(x=1,y,z) = $y + \bar{z} + y\bar{z}$ **F(x,y=0,z) =** $0 + x\bar{z} + 0 = x\bar{z}$

- **Note: this is a new function, that no longer depends on this variable (var)**

© 2013, R.A. Rutenbar

Shannon Expansion: Cofactors

- **Turns out to be an incredibly useful idea**
  - Several alternative names and notations
  - Shannon Cofactor with respect to **xi**
    - Write **F(x1, x2, ..., xi=1, ...xn)** as: F_{x_i} = positive cofactor
    - Write **F(x1, x2, ..., xi=0, ...xn)** as: $F_{\bar{x}_i}$ - negative cofactor
    - Often write this as just **F(xi=1) F(xi=0)** which is easier to type
- **Why are these useful functions to get from F?**

© 2013, R.A. Rutenbar

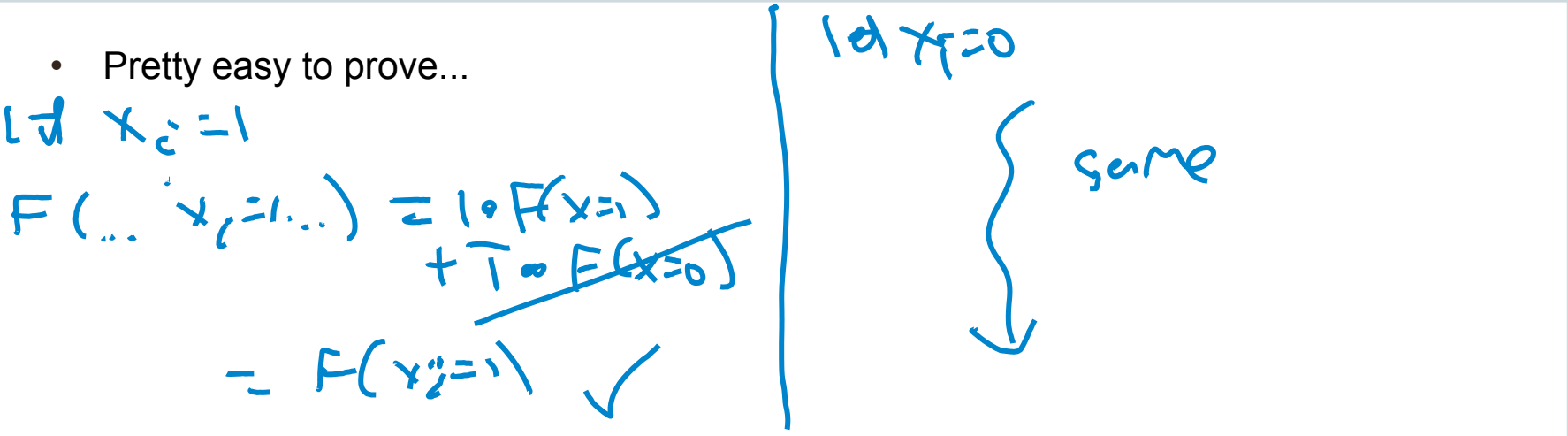
Shannon Expansion Theorem

- **Why we care: Shannon Expansion Theorem**
  - Given any Boolean function **F(x1, x2, ..., xn)** and pick any **xi** in **F( )'s** inputs
    F( ) can be represented as

$$F(x1, x2, \ldots, xi, \ldots\ldots, xn) = xi \bullet F(xi=1) + xi' \bullet F(xi=0)$$

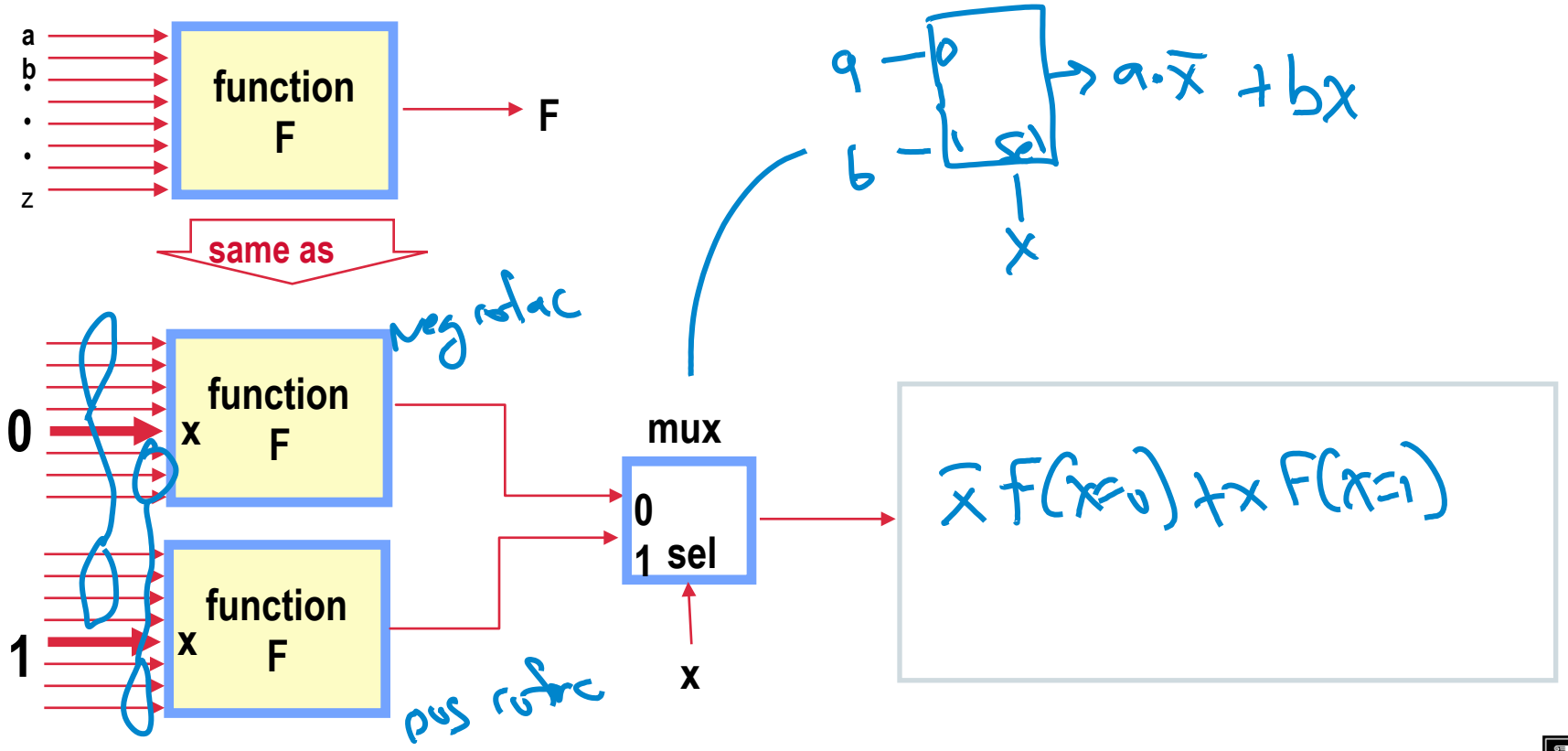
pos neg

- Pretty easy to prove...



© 2013, R.A. Rutenbar

Shannon Expansion: Another View



© 2013, R.A. Rutenbar

Shannon Expansion: Multiple Variables

- **Can do it on *more than one* variable, too**
  - Just keep on applying the theorem
  - Example **F(x,y,z,w) = x • F(x=1) + x' • F(x=0) expanded around x**

Expand each cofactor around y

$F(x=1) = y \cdot F(x=1, y=1) + \bar{y} F(x=1, y=0)$

$F(x=0) = y \cdot F(x=0, y=1) + \bar{y} F(x=0, y=0)$

F(x,y,z,w) =

$xy\,F(x=1, y=1) + x\bar{y}\,F(x=1, y=0) + \bar{x}y\,F(x=0, y=1) + \bar{x}\bar{y}\,F(x=0, y=0)$

= expanded around variables x and y

© 2013, R.A. Rutenbar

Shannon Cofactors: Multiple Variables

- **BTW, there is notation for these as well**
  - Shannon Cofactor with respect to **xi** and **xj**
    - Write **F(x1, x2, ..., xi=1, ..., xj=0, ..., xn)** as $F_{xi\ xj'}$ or $F_{xi\ \overline{xj}}$
    - Ditto for any number of variables **xi, xj, xk**, ...
    - Notice also that order does **not** matter: $(F_x)_y = (F_y)_x = F_{xy}$
  - For our example

$$F(x,y,z,w) = xy \bullet F_{xy} + x'y \bullet F_{x'y} + xy' \bullet F_{xy'} + x'y' \bullet F_{x'y'}$$

  - Again, **remember**: each of the cofactors is a ***function***, not a number

F_{xy} = F(x=1, y=1, z, w) = **a Boolean *function* of z and w**

© 2013, R.A. Rutenbar

VLSI CAD: Logic to Layout

Lecture 2.2

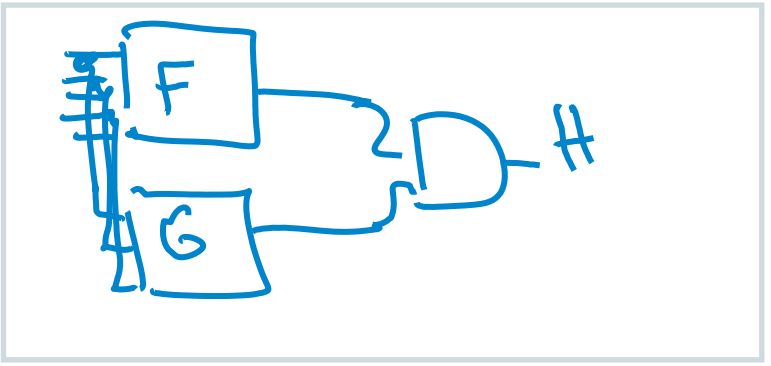
Computational Boolean Algebra: Boolean Difference

Rob A. Rutenbar
University of Illinois

Chris Knapton/Digital Vision/Getty Images

Next Question: Properties of Cofactors

- **What *else* can you do with cofactors?**
  - Suppose you have 2 functions **F(X)** and **G(X)**, where **X=(x1,x2,...xn)**
  - Suppose you make a new function **H**, from **F** and **G**, say…
    - **$H = \overline{F}$**
    - **H = (F • G)** ie, **H(X) = F(X) • G(X)**
    - **H = (F + G)** ie, **H(X) = F(X) + G(X)**
    - **H = (F ⊕ G)** ie, **H(X) = F(X)⊕G(X)**



- **Interesting question**
  - Can you tell anything about **H's** cofactors from those of **F, G**…?

$(F \cdot G)_x$ = what? $(F')_x$ = what? etc.

© 2013, R.A. Rutenbar

Nice Properties of Cofactors

- **Cofactors of F and G tell you *everything* you need to know**
  - **Complements**
    - $(F')_x =$ $(F_x)' = \overline{(F_x)}$
    - In English: *cofactor of complement is complement of cofactor*
  - **Binary boolean operators**
    - $(F \cdot G)_x = F_x \cdot G_x$ *cofactor of AND is AND of cofactors*
    - $(F + G)_x = F_x + G_x$ *cofactor of OR is OR of cofactors*
    - $(F \oplus G)_x = F_x \oplus G_x$ *cofactor of EXOR is EXOR of cofactor*
    - **Very useful:** can often help in getting cofactors of complex formulas

© 2013, R.A. Rutenbar

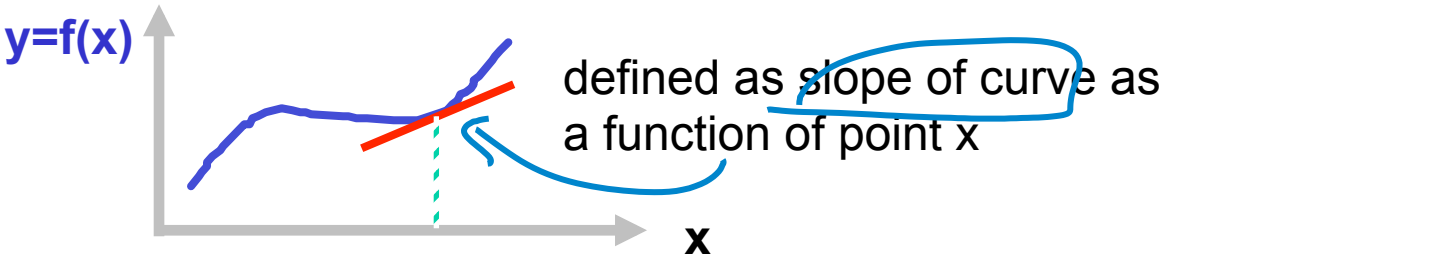
Combinations of Cofactors

- **OK, now consider *operations* on cofactors themselves**
- **Suppose we have F(X), and get F_x and F_x'**
  - **$F_x \oplus F_x' = ?$**
  - **$F_x \bullet F_x' = ?$**
  - **$F_x + F_x' = ?$**
- **Turns out these are all useful *new* functions**
  - Indeed – they even have **names!**
- **Next: let's go look at these interesting, useful new things**
  - First up: the **EXOR** of the cofactors

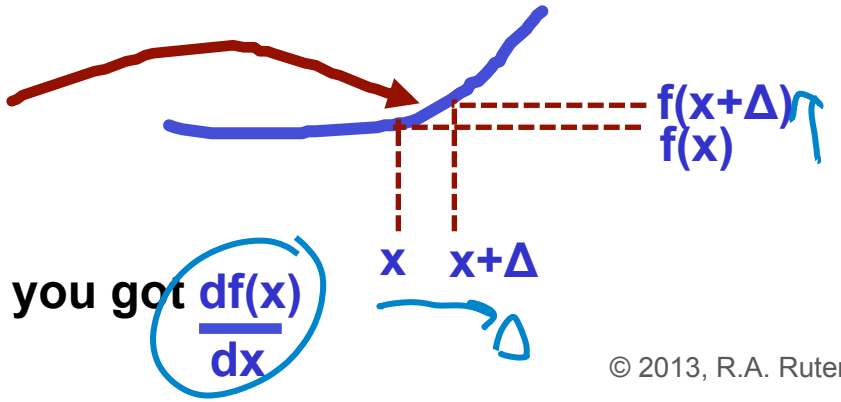
© 2013, R.A. Rutenbar

Calculus Revisited: Derivatives

- **Remember way back to how you defined derivatives?**
  - Suppose you have **y = f(x)**



Considered $\frac{f(x + \Delta) - f(x)}{\Delta}$



Let Δ go to 0 in the limit and you got $\frac{df(x)}{dx}$

© 2013, R.A. Rutenbar

Boolean Derivatives

- **So, do Boolean functions have "derivatives"…?**
  - Actually, **yes**. Trick is how to **define** them...

- **Basic idea**
  - For real-valued **f(x)**, **df/dx** tell how **f** changes when **x** changes
  - For **0,1**-valued Boolean function, we cannot change **x** by small delta
  - Can only change **0 ↔ 1**, but can still ask how **f** changes with **x** ...

Boolean f(x):

∂f/∂x =

$f_x \oplus f_{\bar{x}}$

Compares value of **f( )** when **x=0** against when **x=1** ; **==1** just if these are *different*

© 2013, R.A. Rutenbar

It's Got a Name: Boolean *Difference*

- **Hey, we have seen these pieces before!**
  - **∂f/∂x** = **exor** of the Shannon cofactors with respect to **x**
  - But… for Boolean variables, it's usually written with the "∂" symbol
- **It also behaves *sort* of like regular derivatives...**
  - Order of variables (vars) does not matter
    - **∂f / ∂x∂y = ∂f / ∂y∂x**
  - Derivative of exor is exor of derivatives
    - **∂(f ⊕ g) / ∂x = ∂f/∂x ⊕ ∂g/∂x**
  - If function **f** is actually *constant* (**f=1** or **f=0**, always, for **all** inputs)
    - **∂f/∂x = 0** for any **x**

like addition

like 0

© 2013, R.A. Rutenbar

Boolean Difference

- **But some things are just more complex, though...**
  - Derivatives of **(f •g)** and **(f + g)** ***do not*** work the same...

$$\frac{\partial}{\partial x}(f \bullet g) = \left[f \bullet \frac{\partial g}{\partial x}\right] \oplus \left[g \bullet \frac{\partial f}{\partial x}\right] \oplus \left[\frac{\partial f}{\partial x} \bullet \frac{\partial g}{\partial x}\right]$$

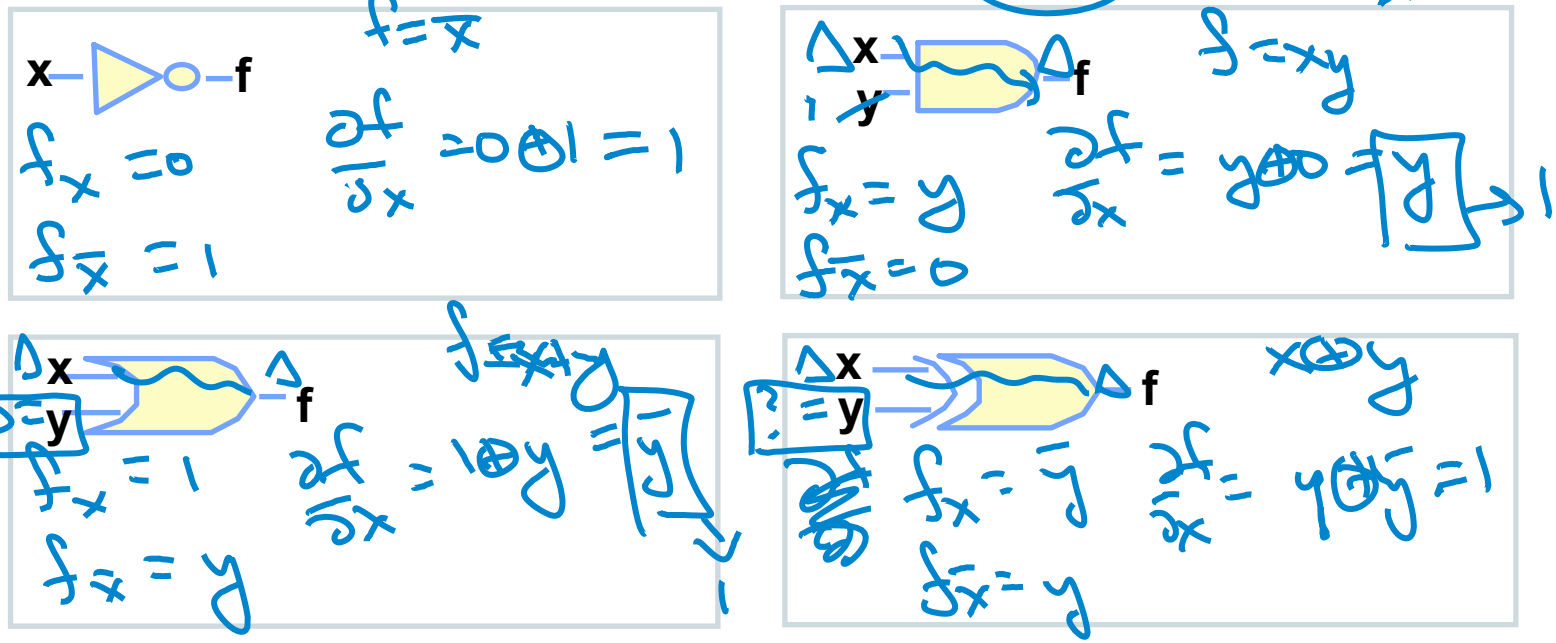
$$\frac{\partial}{\partial x}(f + g) = \left[\overline{f} \bullet \frac{\partial g}{\partial x}\right] \oplus \left[\overline{g} \bullet \frac{\partial f}{\partial x}\right] \oplus \left[\frac{\partial f}{\partial x} \bullet \frac{\partial g}{\partial x}\right]$$

- **Why?**
  - Because AND and OR on Boolean values do **not** always behave like ADDITION and MULTIPLICATION on real numbers

© 2013, R.A. Rutenbar

Boolean Difference: Gate-level View

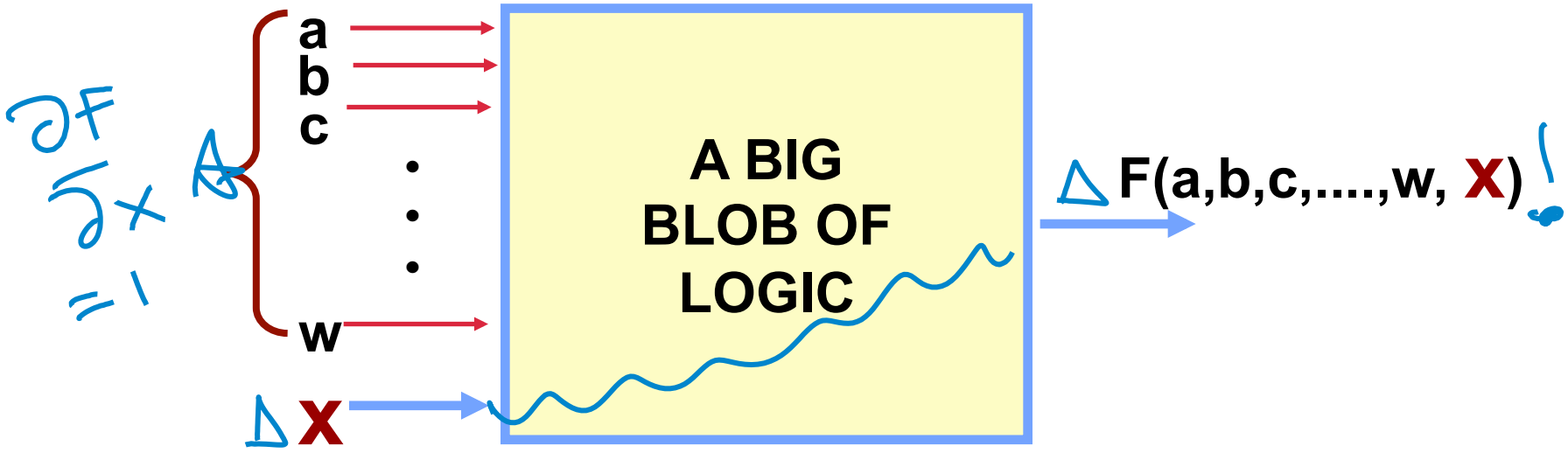
- **Try the obvious "simple" examples for ∂f/∂x**



Meaning: when ∂f/∂x = 1, then f changes if x changes(!)

© 2013, R.A. Rutenbar

Interpreting the Boolean Difference



When $\partial F / \partial x$ (a,b,c, ..., w) = 1, it means that ...

If you apply a pattern of other inputs (not X) that makes $\partial F / \partial x = 1$, Any change in X will force a change in output F( )

© 2013, R.A. Rutenbar

Boolean Difference: Example

$s = a \oplus b \oplus cin$

$Cout = ab + (a + b)Cin$

$\partial Cout / \partial Cin = ?$

$Cout_{Cin} = ab + (a+b) = a+b$

$Cout_{\overline{Cin}} = ab$

$\frac{\partial Cout}{\partial Cin} = (a+b) \oplus ab$

$(a+b)\overline{ab} + (a+b)ab$

$\downarrow$

$a \oplus b$

If $a \neq b$

$\Delta Cin \rightarrow \Delta Cout$!

© 2013, R.A. Rutenbar

Boolean Difference

- **Things to remember about Boolean Difference**
  - Not like the physical interpretation of the ordinary calculus derivative (ie, no "slope of the curve" sort of stuff)…
  - …but it explains how an input-change can cause output-change for a Boolean F( )
  - **∂f / ∂x** is another Boolean **function**, but it does **not** depend on **x**
    - It cannot; it is made out of the cofactors with respect to ("wrt") **x**
    - …and they eliminate all the **x** and **x'** terms by setting them to constants
- **Surprisingly useful (we will see more, later…)**

© 2013, R.A. Rutenbar

VLSI CAD: Logic to Layout

Lecture 2.3

Computational Boolean Algebra: Quantification Operators

Rob A. Rutenbar
University of Illinois

Chris Knapton/Digital Vision/Getty Images

Computational Boolean Algebra, Cont…

- **What you know**
  - Shannon expansion lets you decompose a Boolean function
  - Combinations of cofactors do interesting things, e.g., the Boolean difference $\frac{\partial f}{\partial x}$ ⊕

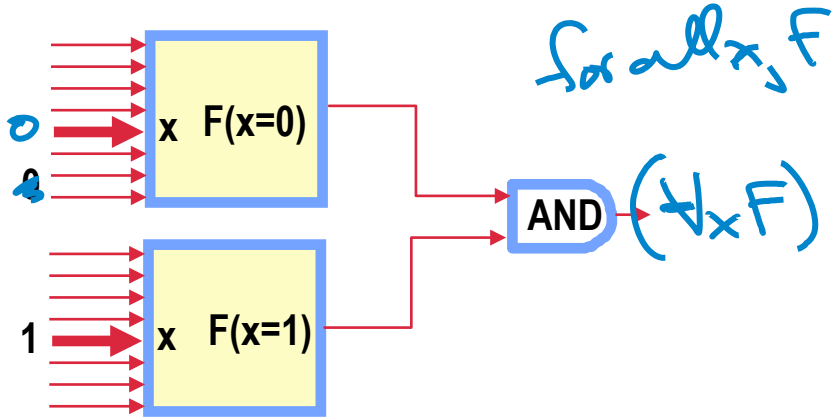
- **What you don't know**
  - Other combinations of cofactors that do useful things
    - The big ones: **Quantification operators** (this lecture)
    - **Applications:** Being able to do something impressive (next lecture)

© 2013, R.A. Rutenbar

AND: Fx • Fx' is *Universal Quantification*

- **Have F(x1, x2, …, xi, … xn)**
- **AND cofactors: $F_{xi} \cdot F_{xi'}$**
  - Name: **Universal Quantification** of function **F** with respect to (wrt) variable **xi**

$(\forall xi\ F)$ [x1,x2, …, xi-1, xi+1, … xn]

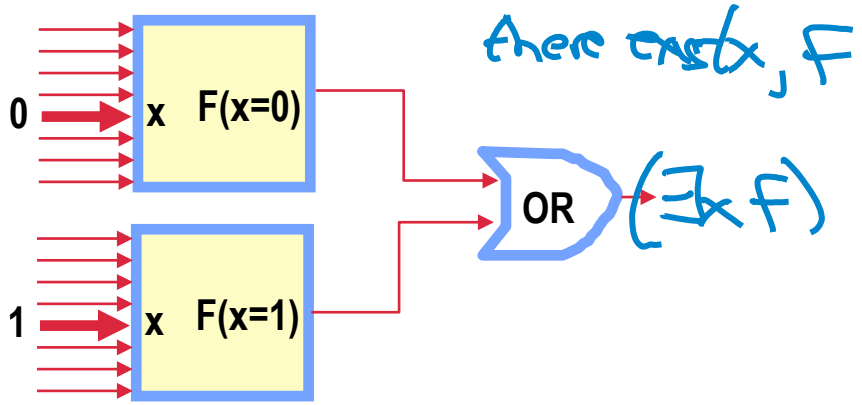


- **"$(\forall xi\ F)$" is a new function**
  - Yes, the '$\forall$' sign is the "for all" symbol from logic (predicate calculus)
  - And, it does not depend on **xi**…

© 2013, R.A. Rutenbar

OR: Fx + Fx' is *Existential Quantification*

- **Have F(x1, x2, …, xi, … xn)**
- **OR the cofactors: $F_{xi} + F_{xi'}$**
  - Name: **Existential Quantification** of function F wrt variable xi
  - **(∃xi F) [x1, x2, …, xi-1, xi+1, … xn]**
- **"(∃xi F)" is a new function**
  - "∃" sign is "there exists" from logic; and function also does not depend on **xi**

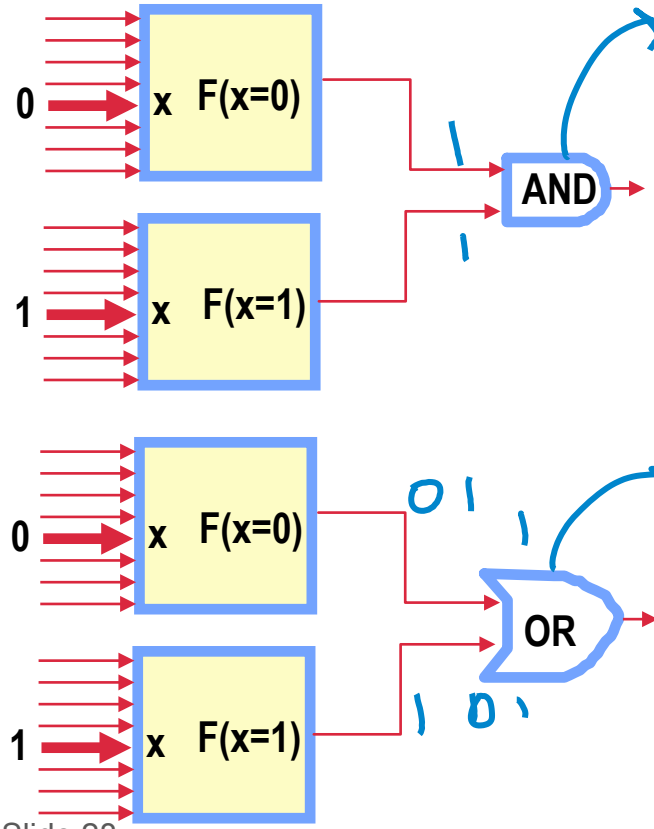


Note, like anything involving cofactors, both these new functions **do not** depend on **xi**

So: **(∀xi F)** and **(∃xi F)** both omit **xi**

© 2013, R.A. Rutenbar

Quantification Notation Makes Sense…



(∀x F) (all original vars but x) ==1 *when?*

→ these other inputs make F=1

For All values of x

(∃x F) (all original vars but x) ==1 *when?*

→ There Exists a value of x that makes F=1 for this input pattern of other vars

© 2013, R.A. Rutenbar

Extends to More Variables in Obvious Way

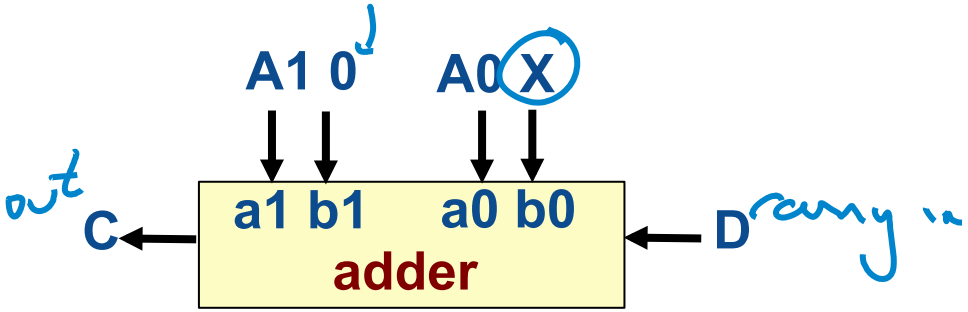
- **Additional properties**
  - Like Boolean difference, can do with respect to **more** than 1 var
  - Suppose we have **F(x,y,z,w)**
  - Example: **(∀xy F)[z,w] = (∀x (∀y F)) = Fxy • Fx'y • Fxy' • Fx'y'**
  - Example: **(∃xy F)[z,w] = (∃x (∃y F)) = Fxy + Fx'y + Fxy' + Fx'y'**

- **Remember!**
  - **(∀x F), (∃x F),** and **∂F / ∂x** are all **functions**...
  - ..but they are functions of all the vars **except x**
  - We got rid of variable x and made 3 **new** functions

© 2013, R.A. Rutenbar

Quantification Example

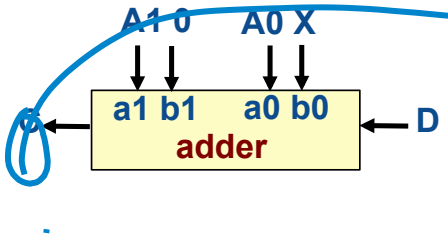
- **Consider this circuit, it adds X=0 or X=1 to a 2-bit number A1A0**
  - It's just a 2-bit adder, but instead of **B1B0** for the second operand, it is just **0X**
  - It produces a carry out called **C** and also has a carry in called **D**



  - What is **(∀A1,A0 C)[X,D]…?**
    - A function of only **X,D**. Makes a **1** for values of **X,D** that make carry **C=1** for ***all values*** of operand input **A1A0**, i.e., makes a carry **C=1** for all values of **A1A0**
  - What is **(∃A1,A0 C)[X,D]…?**
    - A function of just **X,D**. Makes a **1** for values of **X,D** that make carry **C=1**, for ***some value*** of **A1A0**, i.e., ***there exists*** some **A1A0** that, for this **X**,**D** makes **C=1**

© 2013, R.A. Rutenbar

Quantification Example



Do the math: Need all 4 cofactors:

$C = A1A0X + A1(A0+X)D$

C_{A1A0}	$C_{A1'A0}$	$C_{A1A0'}$	$C_{A1'A0'}$
X+D	0	XD	0

- **Compute $(\forall A1,A0\ C)[X,D]$**
  - $C_{A1A0} \bullet C_{A1'A0} \bullet C_{A1A0'} \bullet C_{A1'A0'}$ AND

0

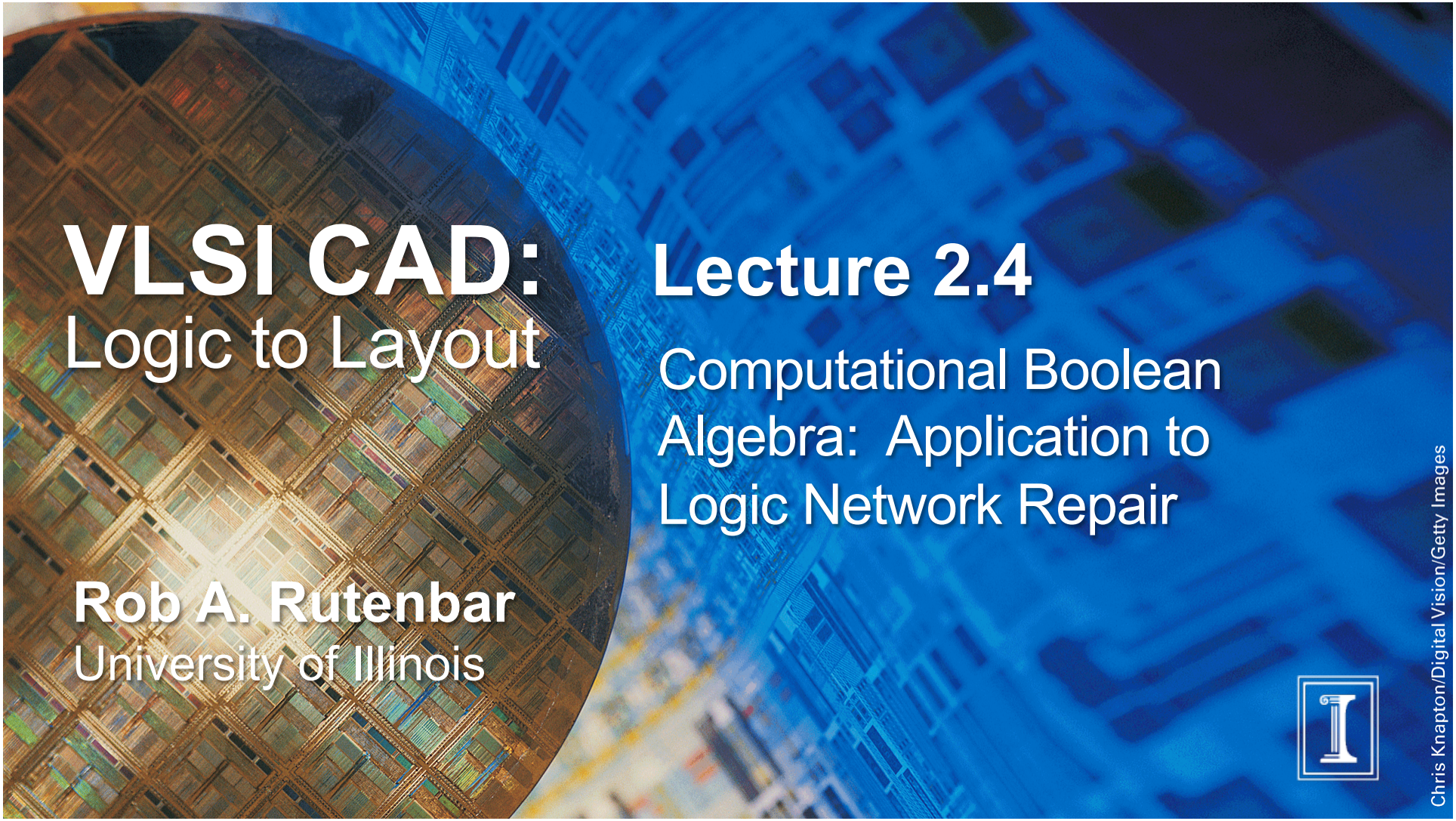
- **In words: No** values of **X,D** that make **C=1** independent of **A1,A0**

- **Compute $(\exists A1,A0\ C)[X,D]$**
  - $C_{A1A0} + C_{A1'A0} + C_{A1A0'} + C_{A1'A0'}$

X+D + 0 + XD + 0 = X+D

- **In words: Yes**, if at least one of **X,D =1 → C=1** indep of **A1,A0**

© 2013, R.A. Rutenbar



VLSI CAD: Logic to Layout

Lecture 2.4

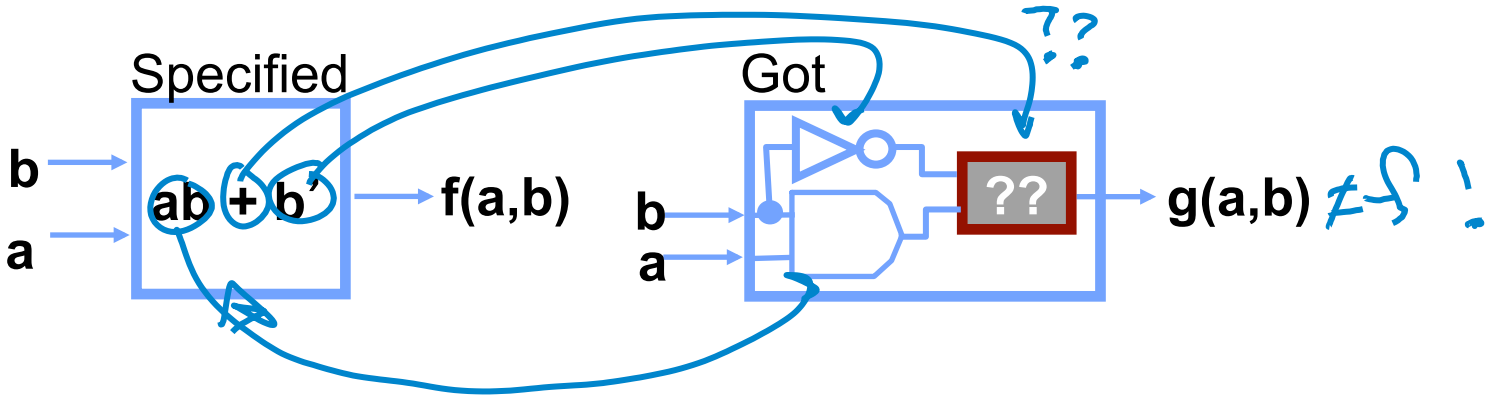
Computational Boolean Algebra: Application to Logic Network Repair

Rob A. Rutenbar
University of Illinois

Chris Knapton/Digital Vision/Getty Images

Quantification App: Network Repair

- **Suppose ...**
  - I specified a logic block for you to implement... **f(a,b) = ab + b'**
  - …but you implemented it **wrong**: in particular, you got **ONE** gate wrong

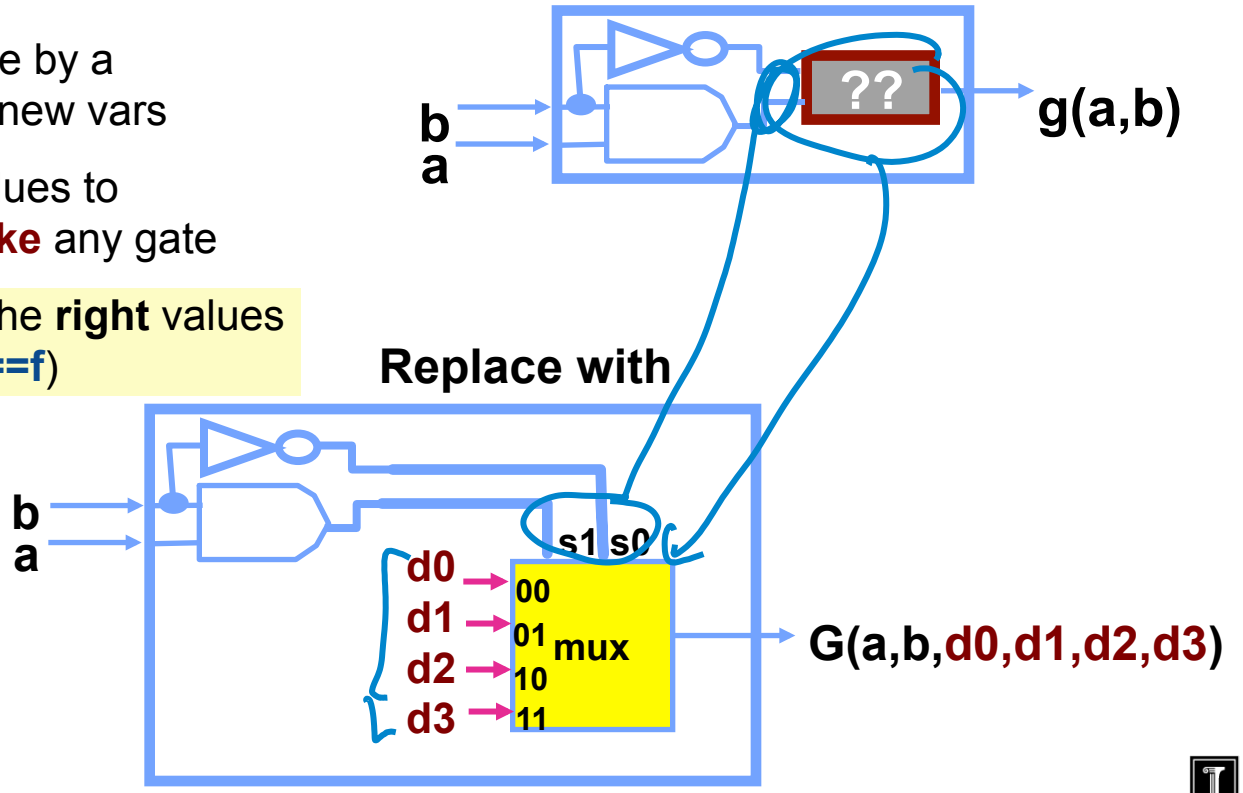


- **Goal**
  - Can we deduce how precisely to **change this gate** to restore correct function?
  - Lets go with this very trivial test case to see how mechanics work...

© 2013, R.A. Rutenbar

Network Repair

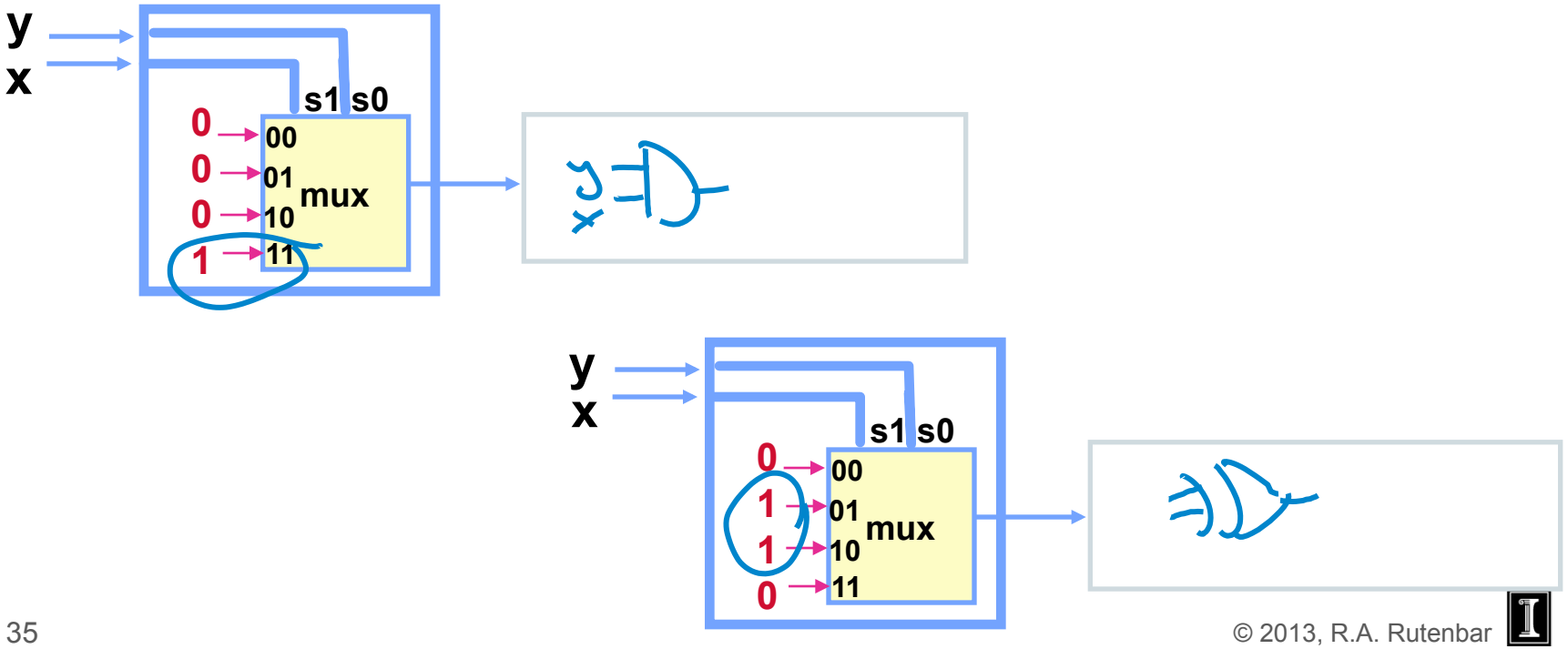
- **Clever trick**
  - Replace our suspect gate by a **4:1** mux with **4** arbitrary new vars
  - By cleverly assigning values to **d0 d1 d2 d3**, we can **fake** any gate
  - **Question is:** what are the **right** values of **d's** so **g** is repaired (**==f**)



© 2013, R.A. Rutenbar

Aside: Faking a Gate with a MUX

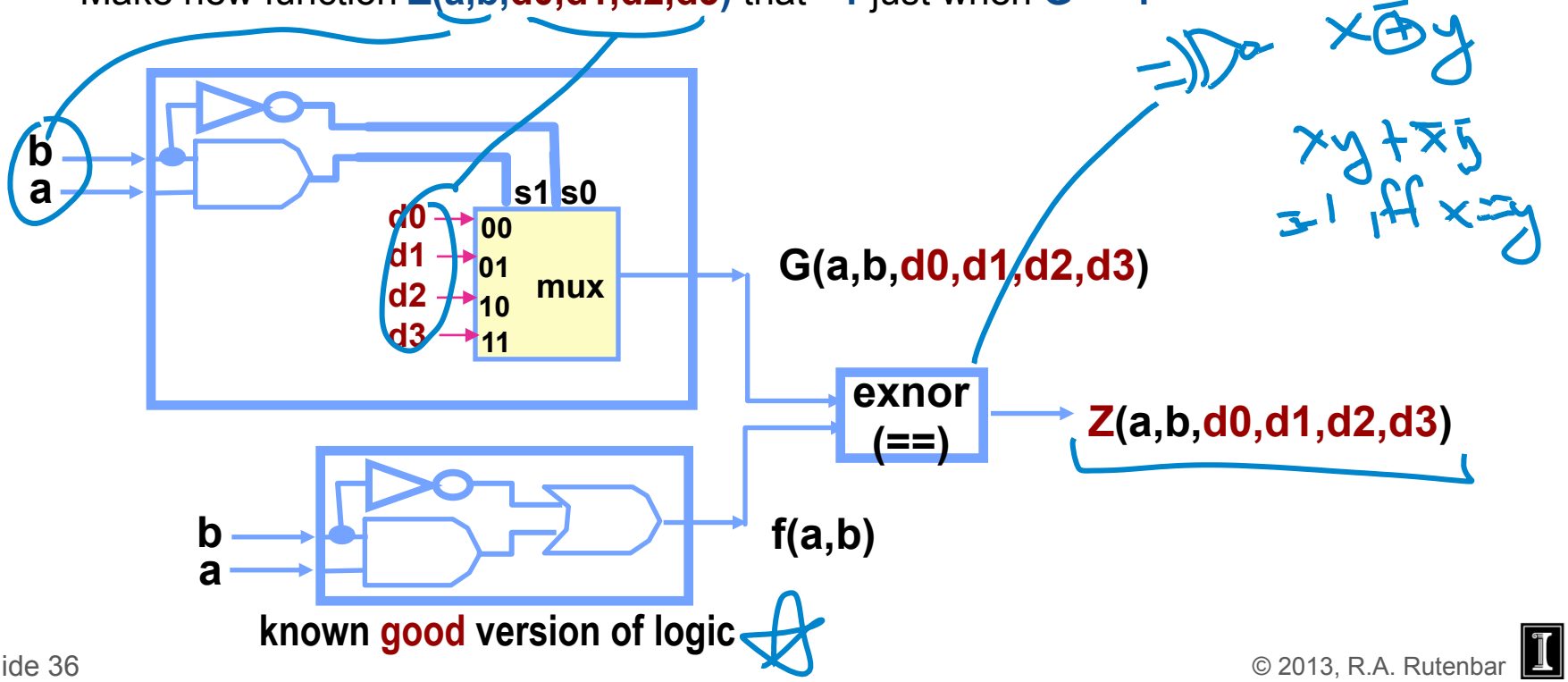
- **Remember...**
  - You can do **any** function of 2 vars with one 4 input multiplexor (MUX)



© 2013, R.A. Rutenbar

Network Repair: Using Quantification

- **Next trick**
  - Make new function **Z(a,b,d0,d1,d2,d3)** that **=1** just when **G == f**



© 2013, R.A. Rutenbar

Using Quantification

- **What now?**
  - Think hard about exactly what we want:

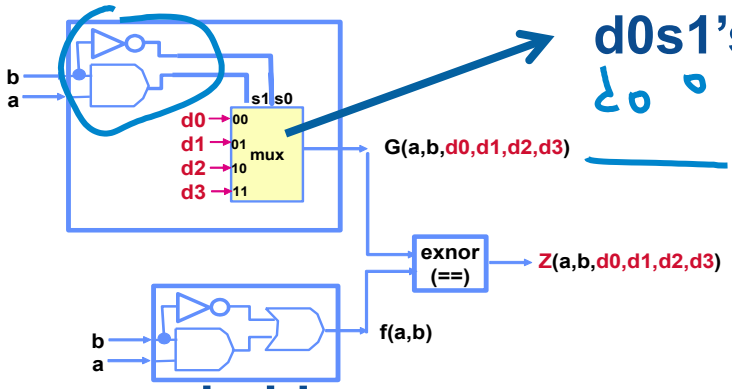
values of $d_0 d_1 d_2 d_3$ that make $Z==1$ for all possible values of inputs a,b.

$(\forall a, b)[d_0 d_1 d_2 d_3] == 1$?

- **But this is something we have seen!**
  - **Universal quantification** of function **Z** wrt variables **a,b**!
  - Any pattern of **(d0 d1 d2 d3)** that makes **(∀ab Z)(d0,d1,d2,d3)==1** will do it!
  - (Aside: do you know where **a, b** went??)

© 2013, R.A. Rutenbar

Network Repair via Quantification: Try It…



d0s1's0' + d1s1's0 + d2s1s0' + d3s1s0

© 2013, R.A. Rutenbar

Network Repair via Quantification: Continued

$$Z = \overbrace{[d0a'b + d1b' + d2ab]}^{G} \overline{\oplus} \overbrace{[ab + b']}^{f} = G \text{ exnor } f$$

Reminder:
Q exnor 0 = **Q'**
Q exnor 1 = **Q**

Use nice property: *cofactor of exnor is exnor of cofactors!*

$Z_{a'b'} = G_{a'b'} \overline{\oplus} f_{a'b'}$ → **set a=0,b=0** →

$Z_{a'b} = G_{a'b} \overline{\oplus} f_{a'b}$ → **set a=0,b=1** →

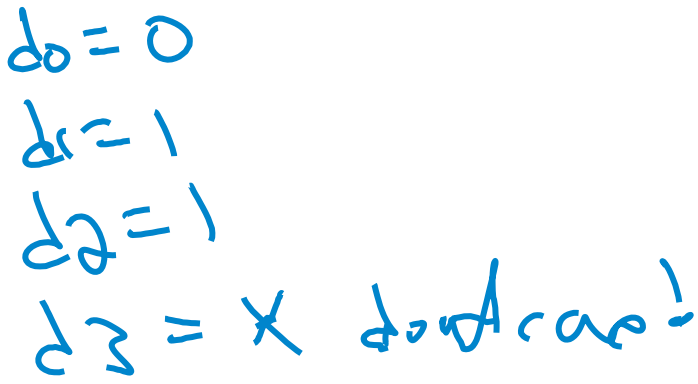
$Z_{ab'} = G_{ab'} \overline{\oplus} f_{ab'}$ → **set a=1,b=0** →

$Z_{ab} = G_{ab} \overline{\oplus} f_{ab}$ → **set a=1,b=1** →

© 2013, R.A. Rutenbar

Repair via Quantification: Continued

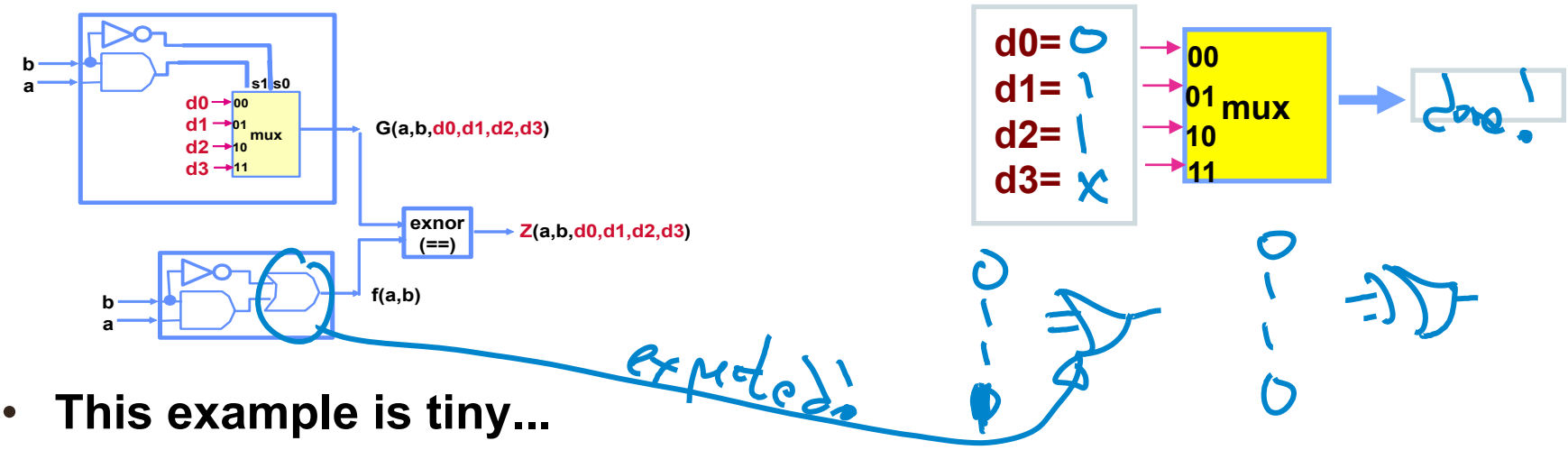
- **So, we got this: (∀ab Z)[d0,d1,d2,d3] = d0' •d1•d2**
- **And know this: if can solve (∀ab Z)[d0,d1,d2,d3]==1 → repaired!**
- **Hey – this one is not very hard…**



© 2013, R.A. Rutenbar

Network Repair

- **Does it work? What do these d's represent?**



- **This example is tiny...**
  - But in a real example, you have a **big** network-- 100 inputs, 50,000 gates
  - When it doesn't work, it's a major hassle to go thru in detail
  - This is a mechanical procedure to answer: Can we change 1 gate to **repair**?

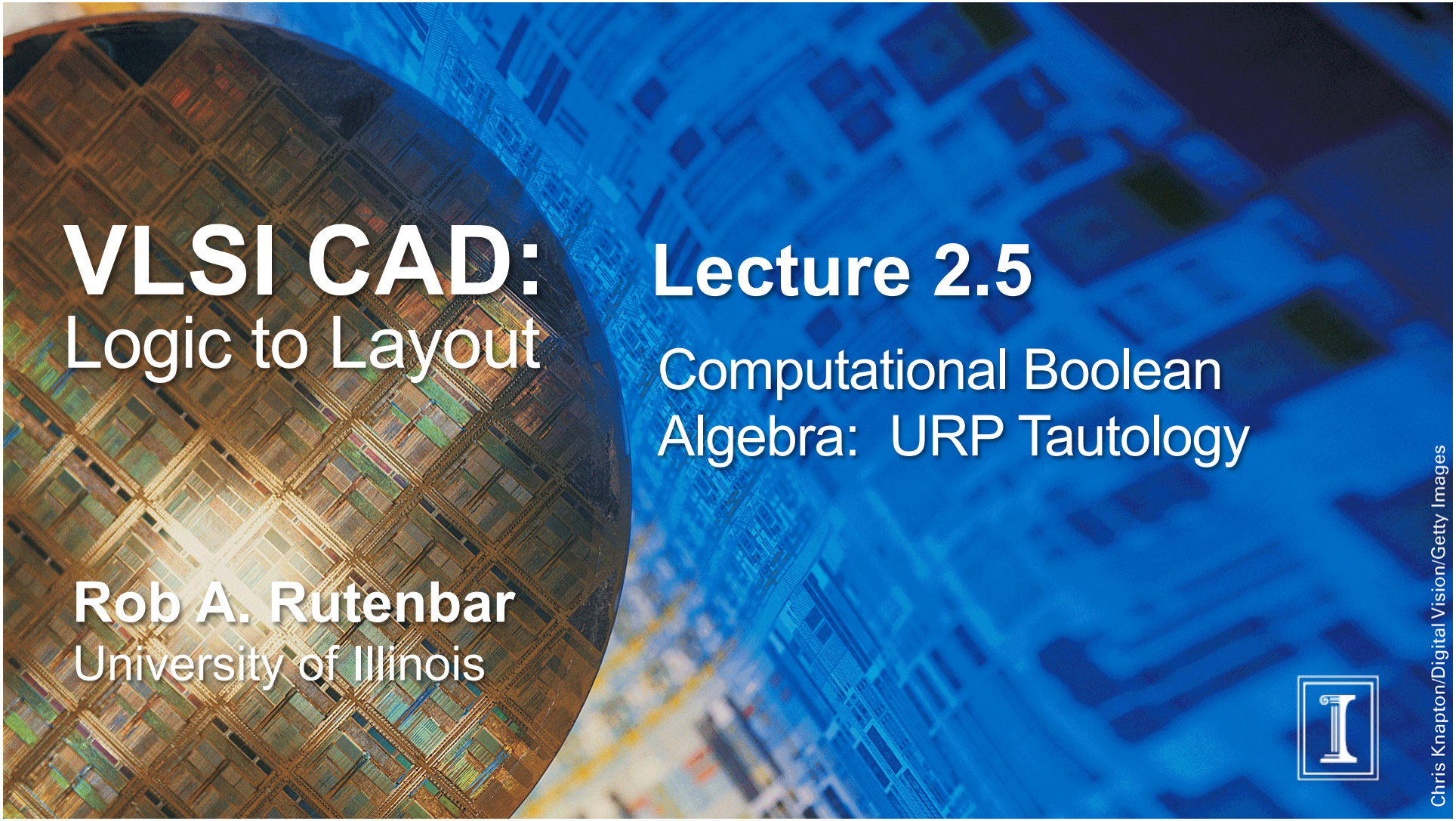
© 2013, R.A. Rutenbar

Computational Boolean Alg Strategies

- **What haven't we seen yet? Computational strategies**
  - Example: find inputs to make **(∀ab Z)(d0,d1,d2,d3)==1** for gate debug
  - This computation is called **Boolean Satisfiability (also called SAT)**

- **Ability to do Boolean SAT efficiently is a big goal for us**
  - **We will see how to do this in later lectures…**

© 2013, R.A. Rutenbar



VLSI CAD: Logic to Layout

Lecture 2.5

Computational Boolean Algebra: URP Tautology

Rob A. Rutenbar
University of Illinois

Chris Knapton/Digital Vision/Getty Images

Important Ex Computation: Tautology

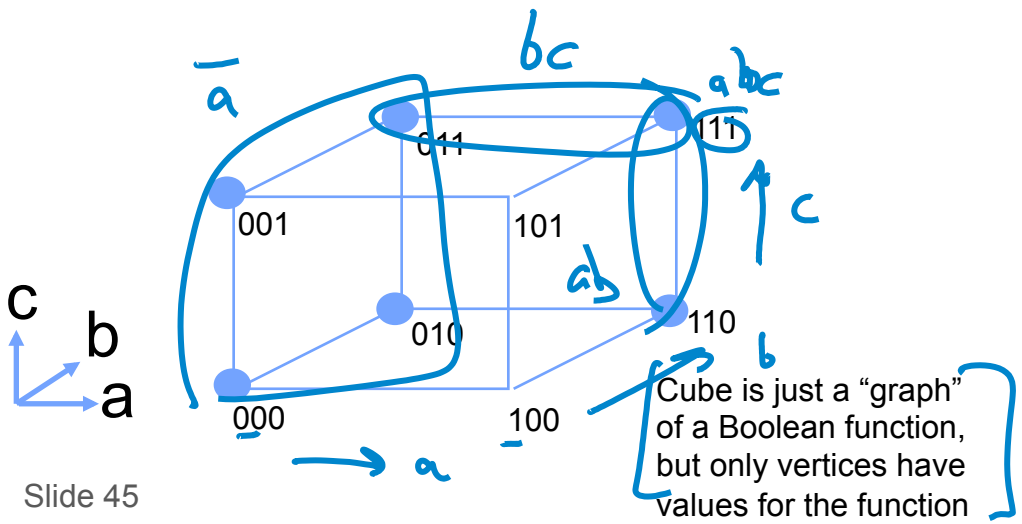
- **Let's build a real computational strategy for a real problem**
- **Tautology:**
  - I will give you a **representation – a *data structure*** -- for a Boolean function **f( )**
  - You build an **algorithm** to tell– yes or no – if this function **f( ) == 1** for every input

- **You might be thinking: Hey, how *hard* can that be…??**
  - Very, *very* hard.
  - What happens if I give you a function with **50 variables**…?
  - Turns out this is a great example: illustrates all the stuff we need to know...

© 2013, R.A. Rutenbar

Start with: Representation

- **We use a simple, early representation scheme for functions**
  - Represent a function as a set of OR'ed product terms (i.e., a sum of products)
  - **Simple visual:** use a 3-var **Boolean cube**, with solid circles where **f() = 1**
  - So: each product term (circle in a Kmap) called a "**cube**" == 2^k corners circled

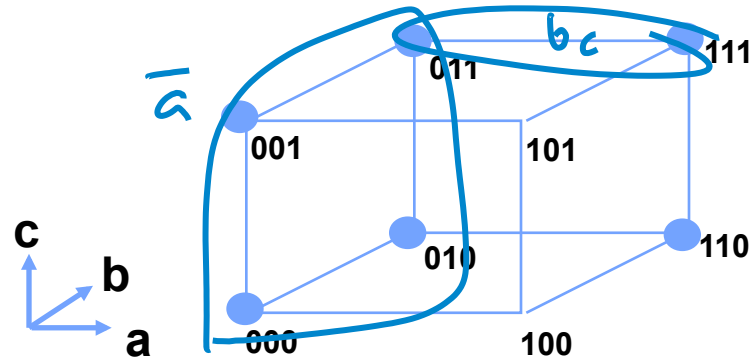


Cube is just a "graph" of a Boolean function, but only vertices have values for the function

© 2013, R.A. Rutenbar

Positional Cube Notation (PCN)

- **So, we say 'cube' and mean 'product term'**
  - So, how to represent each cube? **PCN:** one slot per variable, 2 bits per slot
  - Write each cube by just noting which variables are true, complemented, or absent
    - In slot for var **x**: put **01** if product term has **...x...** in it
    - In slot for var **x**: put **10** if product term has **...x'...** in it
    - In slot for var **x**: put **11** if product terms has **no x or x'** in it



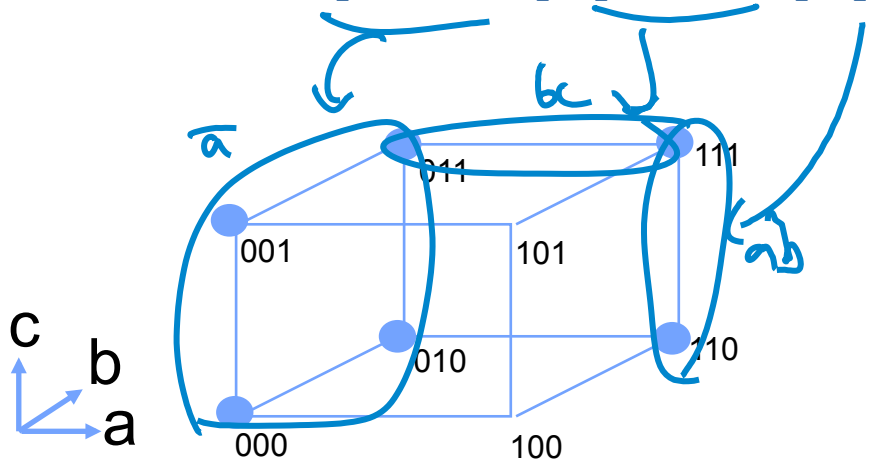
$\bar{a} = [10 \;\; 11 \;\; 11]$

$bc = [11 \;\; 01 \;\; 01]$

© 2013, R.A. Rutenbar

PCN Cube List = Our Representation

- **So, we represent a function as a cover of cubes (circle its 1's)**
  - This is a **list of cubes** (this is the 'sum') in **positional cube notation** (of products)
  - Ex: **f(a,b,c)=$\bar{a}$ +bc +ab => [01 11 11], [11 01 01], [01 01 11]**



© 2013, R.A. Rutenbar

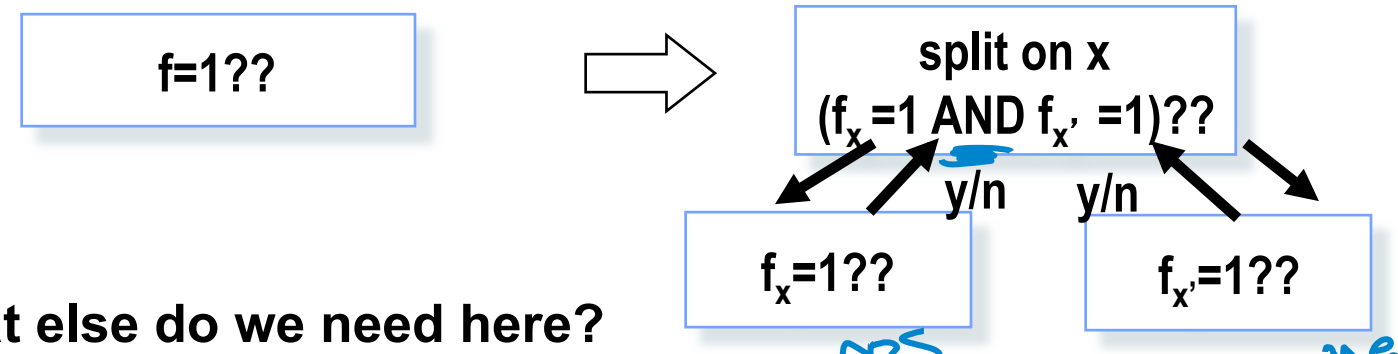
Tautology Checking

- **How do we approach tautology as a computation?**
  - **Input** = cube-list representing products in an SOP cover of **f**
  - **Output** = yes/no, **f ==1** always or not
- **Cofactors to the rescue**
  - Great result: **f is a tautology *if and only if* fx and fx' are *both* tautologies**
  - This makes sense:
    - If function **f() = 1** → then cofactors both obviously **= 1** ✓
    - If **both** cofactors **= 1** → **x•F(x=1) + x'•F(x=0) = x•1 + x'•1 = x+x' = 1** ✓

© 2013, R.A. Rutenbar

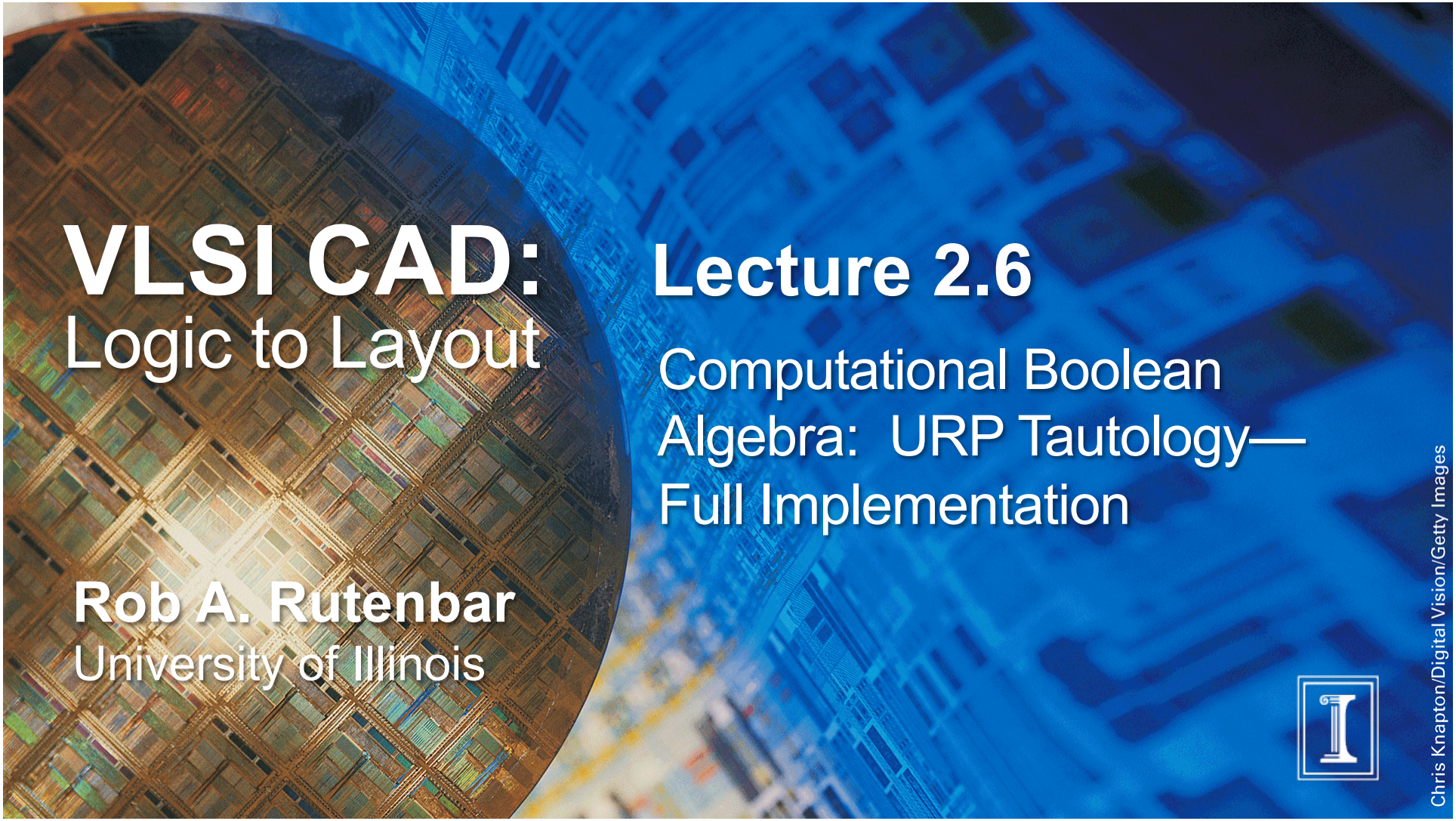
Recursive Tautology Checking

- **Suggests a recursive computation strategy:**
  - If you cannot tell immediately that **f==1** ...go try to see if each **cofactor == 1** !



- **What else do we need here?**
  - **Selection rules:** *which* **x** is good to pick to split on?
  - **Termination rules:** how do we know when to quit splitting, so we can answer **==1** or **!=1** for function at this node of tree?
  - **Mechanics:** how hard is it to actually *represent* the cofactors?

© 2013, R.A. Rutenbar



VLSI CAD: Logic to Layout

Lecture 2.6

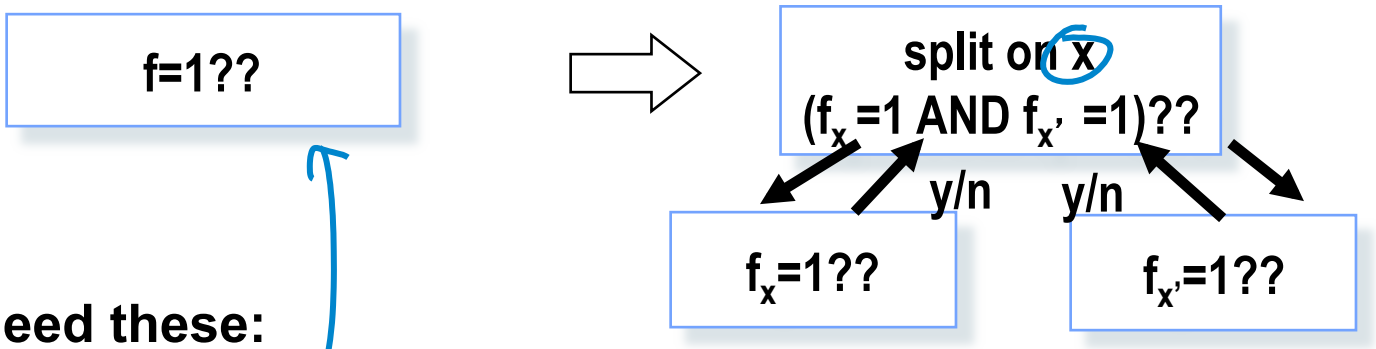
Computational Boolean Algebra: URP Tautology—Full Implementation

Rob A. Rutenbar
University of Illinois

Chris Knapton/Digital Vision/Getty Images

Recursive Tautology Checking

- **So, we think a recursive computation strategy will do it:**
  - If you cannot tell immediately that **f==1** ...go try to see if each **cofactor == 1** !



- **We need these:**
  - **Selection rules:** *which* **x** is good to pick to split on?
  - **Termination rules:** how do we know when to quit splitting, so we can answer **==1** or **!=1** for function at this node of tree?
  - **Mechanics:** how hard is it to actually *represent* the cofactors?

© 2013, R.A. Rutenbar

Recursive Cofactoring

- **Do mechanics first (easy!). For each cube in your list:**
  - If you want cofactor wrt var x=1, look at **x** slot in each cube:
    - [... **10** ...] => just remove this cube from list, since it's a term with an **x'**
    - [... **01** ...] => just make this slot **11** == don't care, strike the **x** from product term
    - [... **11** ...] => just leave this alone, this term doesn't have any **x** in it
  - If you want cofactor wrt var x=0, look at **x** slot in each cube:
    - [...**01** ...] => just remove this cube from list, since it's a term with an **x**
    - [...**10** ...] => just make this slot **11** == don't care, strike the **x'** from product term
    - [...**11** ...] => just leave this alone, this term doesn't have any **x** in it

f = abd+bc'	f_a	f_c
[01 01 11 01]		
[11 01 10 11]		

© 2013, R.A. Rutenbar

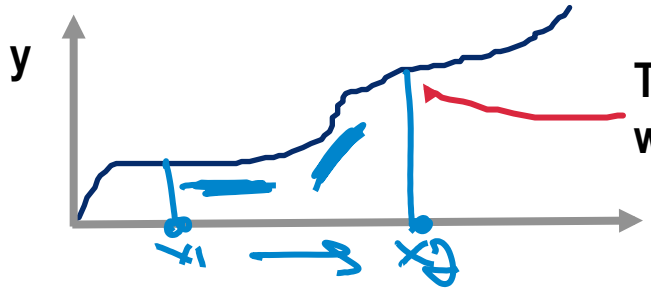
Unate Functions

- **Selection / termination, another trick: Unate functions**
  - Special class of Boolean functions
  - **f** is ***unate*** if a SOP representation only has each literal in **exactly one polarity**, either all true, or all complemented
    - **Ex:** **ab + ac'd + c'de'** is … unate: $a, b, \bar{c}, d, \bar{e}$
    - **Ex:** **xy + x'y + xyz' + z'** is ... not unate: $x, y, \bar{x}$ is unate in y
- **Terminology**
  - **f** is **positive unate** in var **x** -- if changing **x 0→1** keeps **f constant** or makes **f: 0→1**
  - **f** is **negative unate** in var **x** -- if changing **x 0→1** keeps **f constant** or makes **f: 1→0**
  - Function that is not unate is called **binate**

© 2013, R.A. Rutenbar

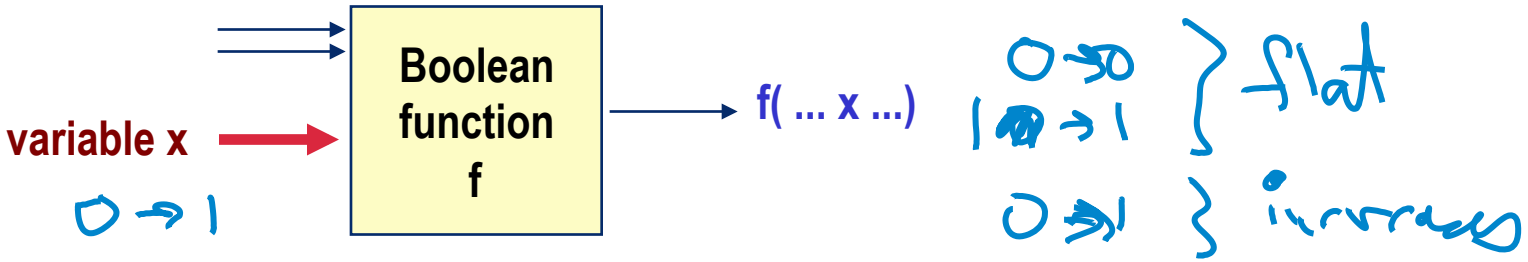
Unate Functions

- **Analogous to monotone continuous functions**



This function is monotone non-decreasing,
whenever x2 ≥ x1, then f(x2) ≥ f(x1)

- **Example: for a Boolean function f positive unate in x**

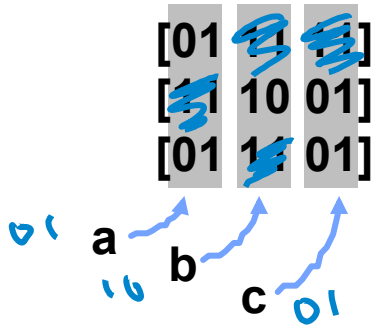


© 2013, R.A. Rutenbar

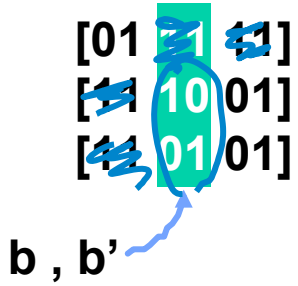
Can Exploit Unate Func's For Computation

- **Suppose you have a cube-list for f**
  - A cube-list is **unate** if each var in each cube only appears in **one** polarity, not both
  - Ex: **f(a,b,c)=a +bc +ac → [01 11 11], [11 01 01], [01 11 01]** is **unate**
  - Ex: **f(a,b,c)=a +b'c +bc → [01 11 11], [11 10 01], [11 01 01]** is **not**
  - Easier to see if draw vertically

a+b'c+ac UNATE



a+b'c+bc NOT



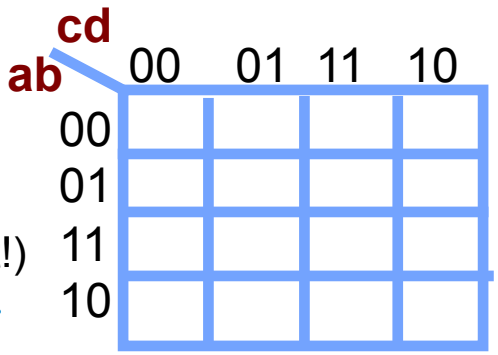
© 2013, R.A. Rutenbar

Using Unate Functions in Tautology Checking

- **Beautiful result**
  - It is very **easy** to check a unate cube-list for tautology
  - **Unate cube-list for f is tautology iff it contains *all don't care* cube = [11 ... 11]**
  - Reminder: what exactly is **[11 11 11 ... 11]** as a product term?

[01 01 01] = abc **[01 01 11] = ab** **[01 11 11] = a** **[11 11 11] =** 1

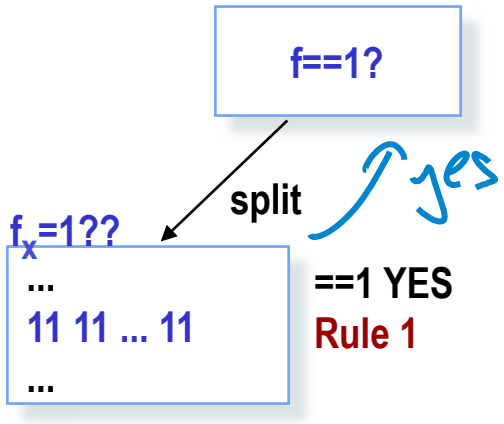
- **This result actually makes sense...**
  - Cannot make a "1" with only product terms where all literals are in just **one polarity**. (Try it!)

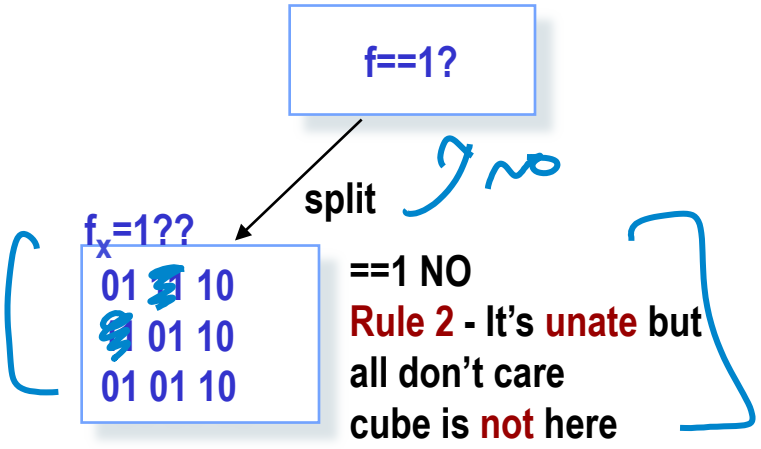


© 2013, R.A. Rutenbar

So, Unateness Gives Us *Termination Rules*

- **We can look for tautology directly, if we have a unate cube-list**
  - If match rule, know immediately if **==1**, or not
    - **Rule 1:** **==1** if cube-list has all don't care cube **[11 11 ... 11]**
      Why: function at this leaf is **(stuff + 1 + stuff) == 1**
    - **Rule 2:** **!=1** if cube-list unate and all don't care cube **missing**
      Why: unate ==1 if and only if has [11 11 ... 11] cube

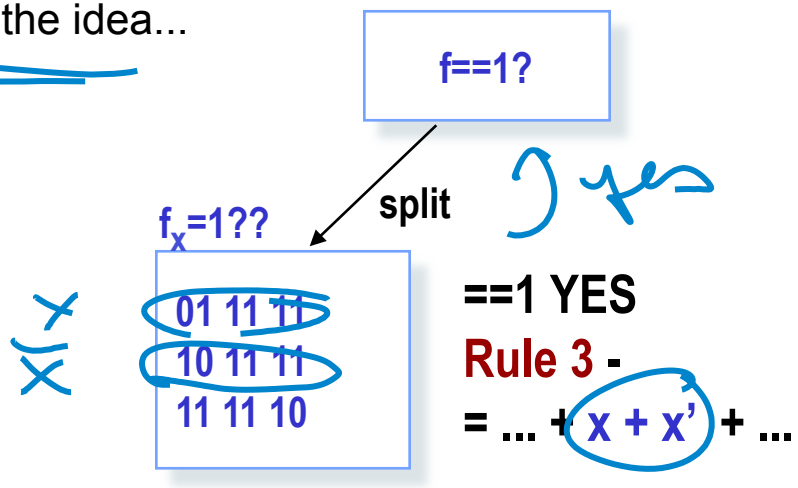




© 2013, R.A. Rutenbar

Recursive Tautology Checking

- **Lots more possible rules...**
  - **Rule 3:** **==1** if cube list has single var cube that appears in both polarities
    Why: function at this leaf is **(stuff + x + x' + stuff) == 1**
- You get the idea...



© 2013, R.A. Rutenbar

Recursive Tautology Checking

- **But can't use easy termination rules *unless* unate cubelist**
- **Selection rule...? Pick splitting var to *make* unate cofactors!**
  - **Strategy:** pick "most not-unate" (binate) var as split var
  - Pick binate var with **most** product terms dependent on it (Why? *Simplify more cubes*)
  - If a tie, pick var with **minimum | true var - complement var |** (L-R *subtree balance*)

x	y	z	w
01	01	01	01
10	11	01	01
10	01	11	10
01	01	11	01

(If **no** binate variables? Pick a unate variable that appears in most cubes)

binate, in 4 cubes, | true - compl | = 3-1 = 2

unate

unate

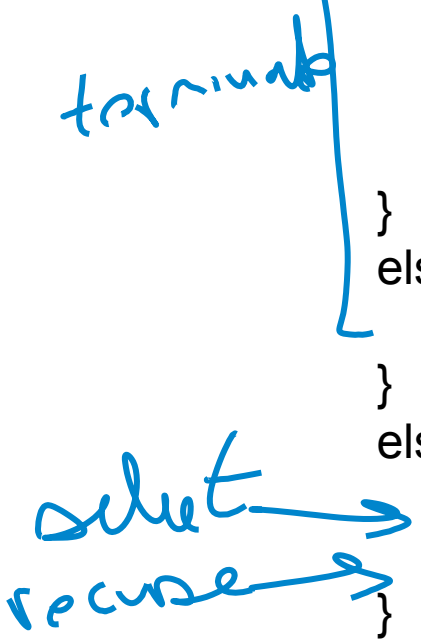
binate, in 4 cubes, | true - compl | = 2-2 = 0

© 2013, R.A. Rutenbar

Recursive Tautology Checking: *Done*!

- **Algorithm**

```
tautology(f represented as cubelist) {
    /* check if we can terminate recursion */
    if (f is unate)  {
        apply unate tautology termination rules directly
        if (==1) return (1)
        else return (0)
    }
    else if (any other termination rules, like rule 3, work?) {
        return the appropriate value if ==1 or ==0
    }
    else {   /* can't tell from this -- find splitting variable */
        x = most-not-unate variable in f
        return (  tautology( f_x ) && tautology( f_x' )   )
    }
}
```



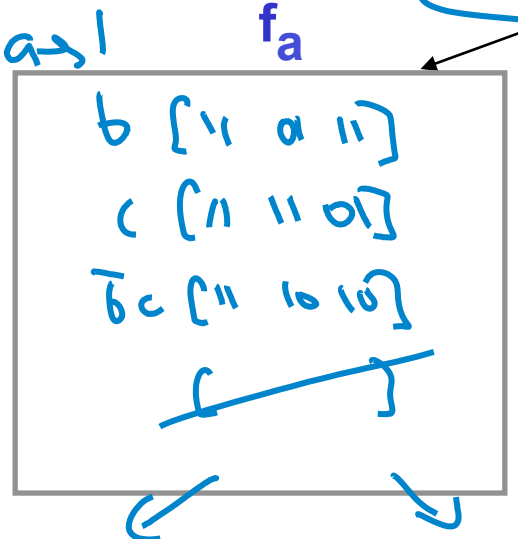
© 2013, R.A. Rutenbar

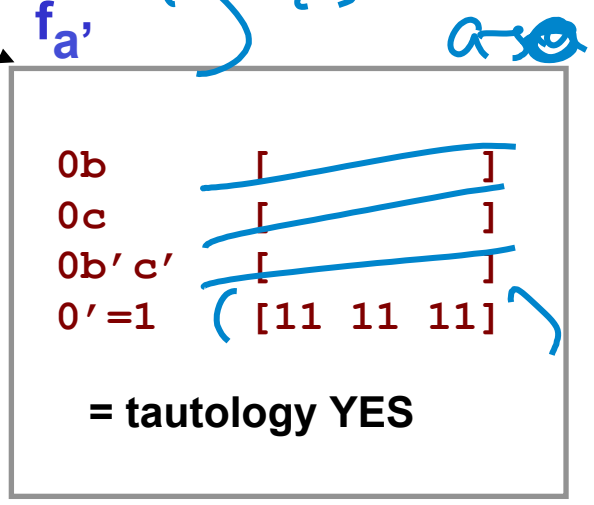
Recursive Tautology Checking: Example

- **Tautology example: f = ab + ac + ab'c' + a'**

	a	b	c
ab	01	01	11
ac	01	11	01
ab'c	01	10	10
a'	10	11	11

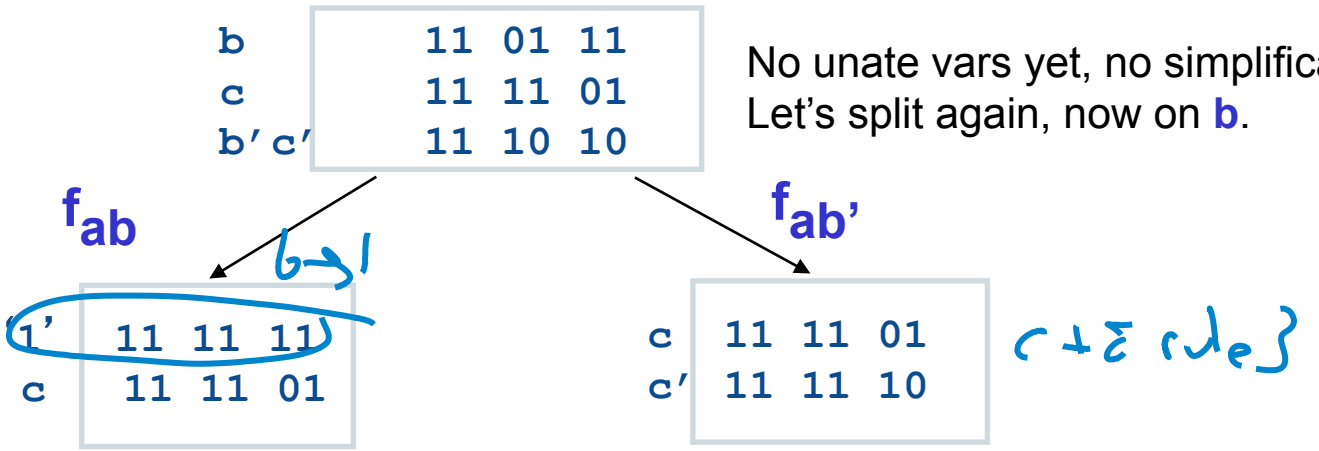
All vars are binate, var **a** affects most implicants, so split on **a**...





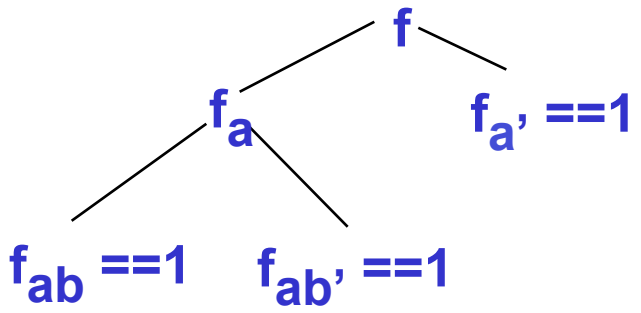
© 2013, R.A. Rutenbar

Recursive Tautology Checking: Example



No unate vars yet, no simplifications.
Let's split again, now on **b**.

- **So we are done:**
  - Our tree has tautologies at all leaves!
  - Note -- if any leaf **!=1**, then **f != 1** too, this is how tautology fails



© 2013, R.A. Rutenbar

Computational Boolean Algebra

- **Computational philosophy revisited**
  - Strategy is so general and useful it has a name: **Unate Recursive Paradigm**
  - Abbreviated usually as **"URP"**

- **Summary**
  - **Cofactors** and **functions of cofactors** are important and useful
    - Boolean difference, quantifications; real applications like network repair
  - **Representations (data structures)** for Boolean functions are critical
    - Truth tables, Kmaps, equations ***cannot be manipulated*** by software
    - Saw one real representation: Cube-list, positional cube notation

© 2013, R.A. Rutenbar