

VLSI CAD: Logic to Layout

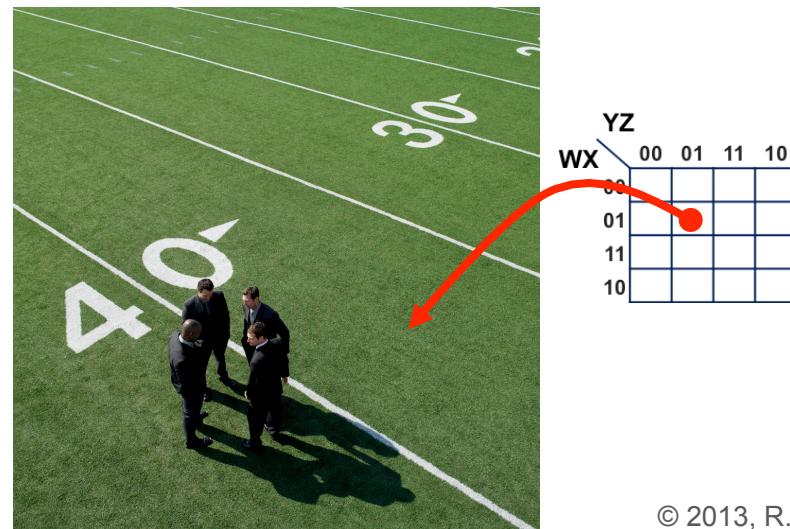
Rob A. Rutenbar
University of Illinois

Lecture 2.1 Computational Boolean Algebra: Basics



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...



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



Advanced Boolean Algebra

- **Useful analogy to calculus...**
 - You can represent complex functions like $\exp(x)$ using simpler functions
 - If you only get to use $1, x, x^2, x^3, x^4, \dots$ as the pieces...
 - ...turns out $\exp(x) = 1 + x + x^2/2! + x^3/3! + \dots$
 - In Calculus, we tell you the general formula, the **Taylor series expansion**
 - $f(x) = f(0) + f'(0)/1! x + f''(0)/2! x^2 + f'''(0)/3! x^3 + \dots$
 - 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?**



Boolean Decompositions

- Yes. Called the *Shannon Expansion*



WIKIPEDIA
The Free Encyclopedia

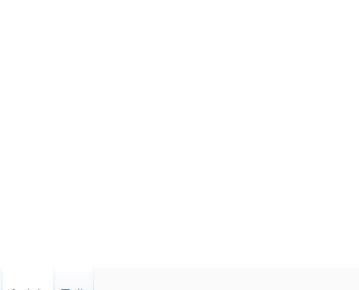
Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export
Languages

Slide 5



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

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



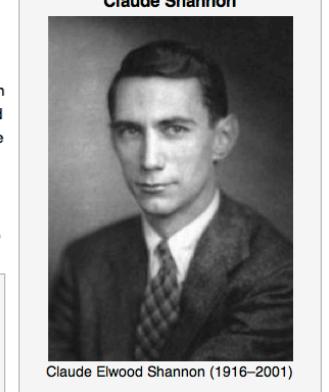
Create account Log in Article Talk Read Edit View history Search

Claude Shannon

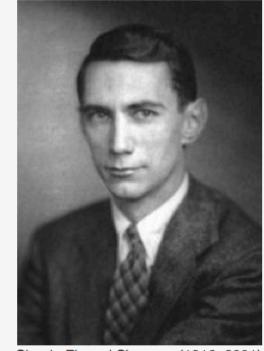
From Wikipedia, the free encyclopedia

Claude Elwood Shannon (April 30, 1916 – February 24, 2001) was an American, electronic engineer, and cryptographer known as "the father of information theory".^{[1][2]}

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



Claude Shannon



Claude Elwood Shannon (1916–2001)

Contents [hide]

Early and beyond

Search

Contributions

© 2013, R.A. Rutenbar



Shannon Expansion

- Suppose we have a function $F(x_1, x_2, \dots, x_n)$
- Define a new function if we set one of the $x_i = \text{constant}$
 - Example: $F(x_1, x_2, \dots, x_i=1, \dots, x_n)$
 - Example: $F(x_1, x_2, \dots, x_i=0, \dots, x_n)$
- Easy to do one by hand

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

$$F(x=1, y, z) =$$

$$F(x, y=0, z) =$$

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



Shannon Expansion: Cofactors

- Turns out to be an incredibly useful idea
 - Several alternative names and notations
 - Shannon Cofactor with respect to x_i
 - Write $F(x_1, x_2, \dots, x_i=1, \dots x_n)$ as:
 - Write $F(x_1, x_2, \dots, x_i=0, \dots x_n)$ as:
 - Often write this as just $F(x_i=1) F(x_i=0)$ which is easier to type
 - Why are these useful functions to get from F ?



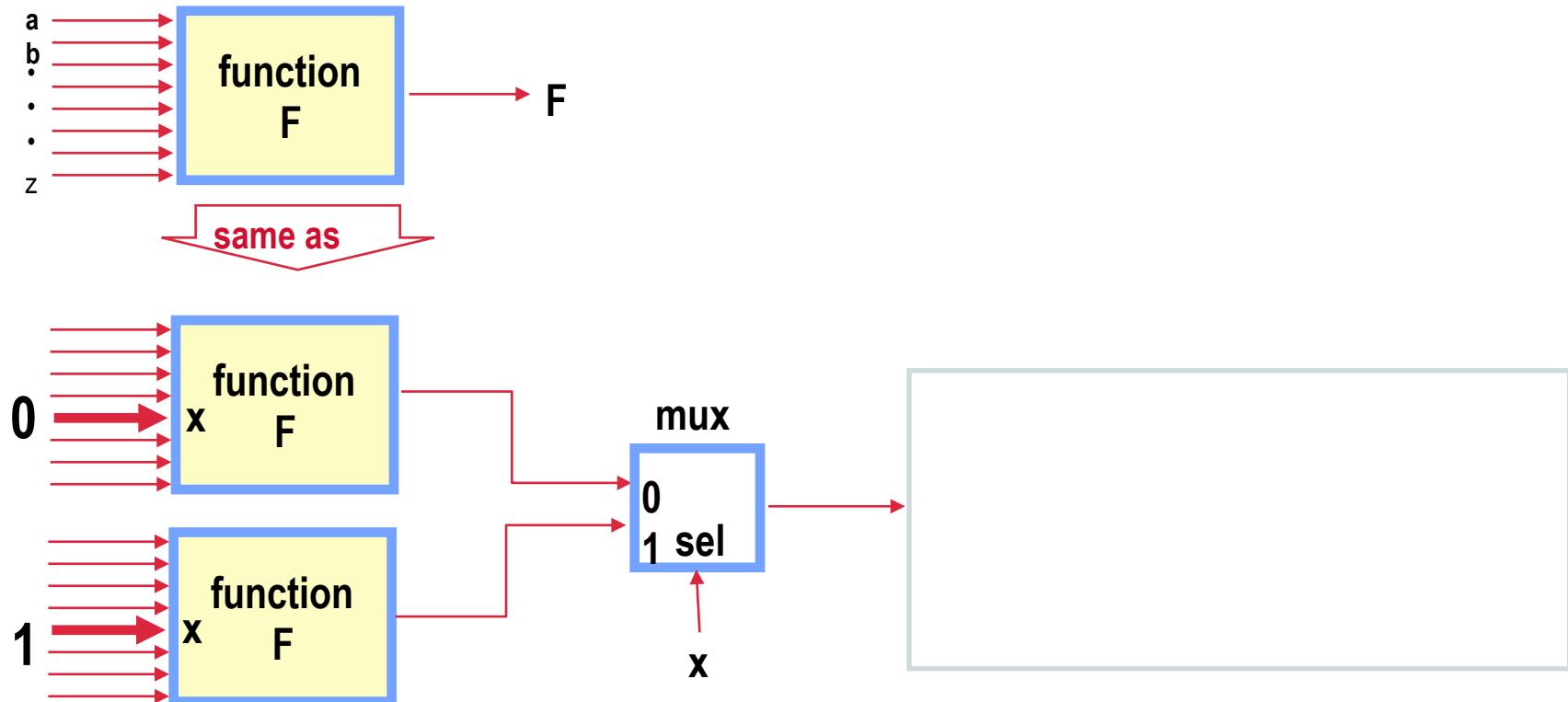
Shannon Expansion Theorem

- Why we care: **Shannon Expansion Theorem**
 - Given any Boolean function $F(x_1, x_2, \dots, x_n)$ and pick any x_i in $F(\)$'s inputs $F()$ can be represented as

$$F(x_1, x_2, \dots, x_i, \dots, x_n) = x_i \cdot F(x_i=1) + x_i' \cdot F(x_i=0)$$

- Pretty easy to prove...

Shannon Expansion: Another View



Shannon Expansion: Multiple Variables

- Can do it on *more than one* variable, too

- Just keep on applying the theorem

- Example $F(x,y,z,w) = x \cdot F(x=1) + x' \cdot F(x=0)$ expanded around x

Expand each
cofactor
around y

$$F(x,y,z,w) =$$

= expanded around variables x and y



Shannon Cofactors: Multiple Variables

- BTW, there is notation for these as well

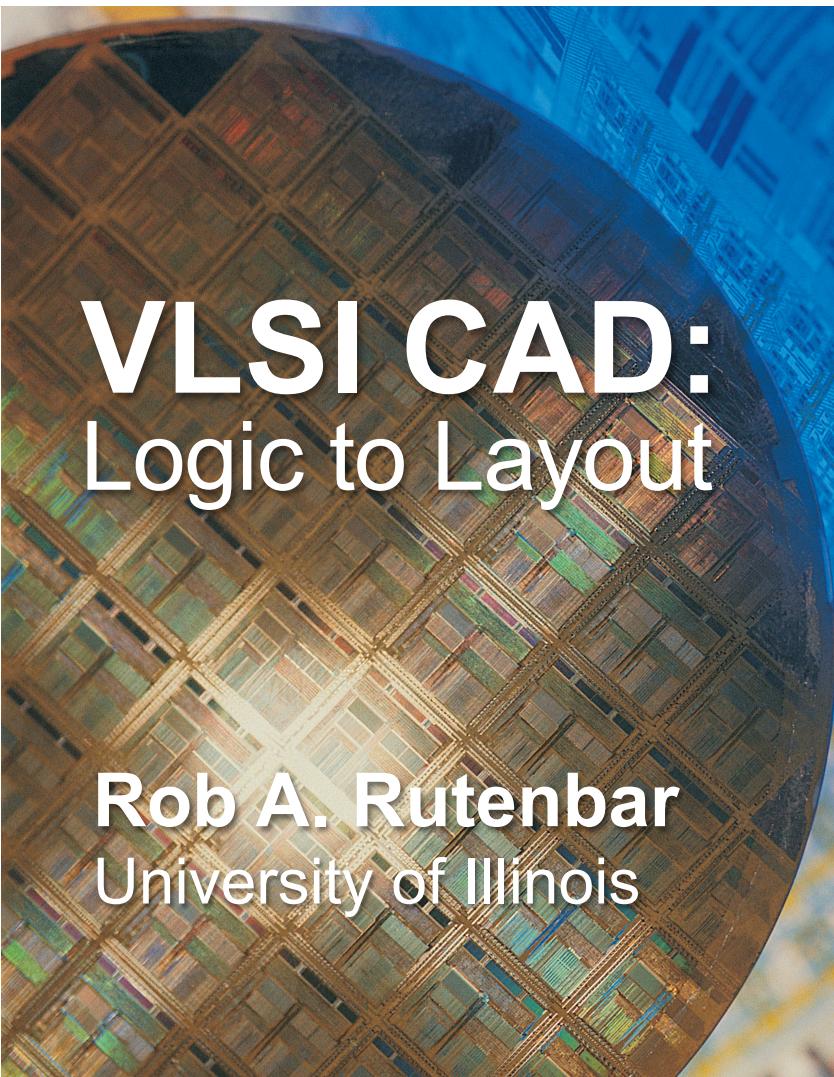
- Shannon Cofactor with respect to x_i and x_j
 - Write $F(x_1, x_2, \dots, x_i=1, \dots, x_j=0, \dots, x_n)$ as $F_{x_i} x_j'$ or $F_{x_i} \bar{x_j}$
 - Ditto for any number of variables x_i, x_j, x_k, \dots
 - Notice also that order does **not** matter: $(F_x)_y = (F_y)_x = F_{xy}$
- For our example

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

- 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





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 2.2

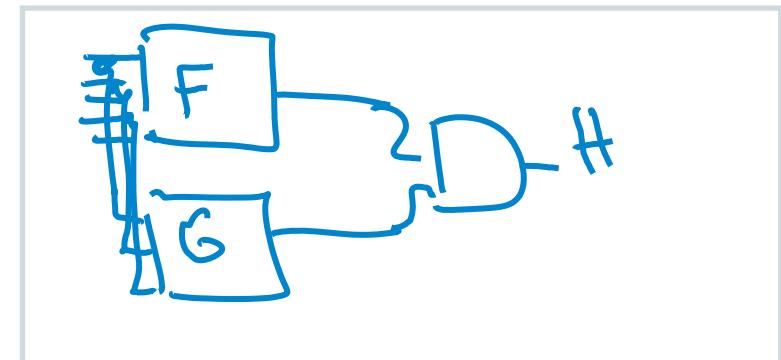
Computational Boolean Algebra: Boolean Difference



Next Question: Properties of Cofactors

- **What else can you do with cofactors?**

- Suppose you have 2 functions $F(X)$ and $G(X)$, where $X=(x_1, x_2, \dots, x_n)$
- Suppose you make a new function H , from F and G , say...
 - $H = \overline{F}$
 - $H = (F \cdot G)$ ie, $H(X) = F(X) \cdot G(X)$
 - $H = (F + G)$ ie, $H(X) = F(X) + G(X)$
 - $H = (F \oplus G)$ ie, $H(X) = F(X) \oplus G(X)$



- **Interesting question**

- Can you tell anything about H 's cofactors from those of F , G ...?

$$(F \cdot G)_x = \text{what?} \quad (F')_x = \text{what?} \quad \text{etc.}$$



Nice Properties of Cofactors

- Cofactors of **F** and **G** tell you *everything* you need to know

- Complements

- $(F')_x = \overline{(F_x)}' = \overline{\overline{(F_x)}}$
- 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 cofactors*

- **Very useful:** can often help in getting cofactors of complex formulas



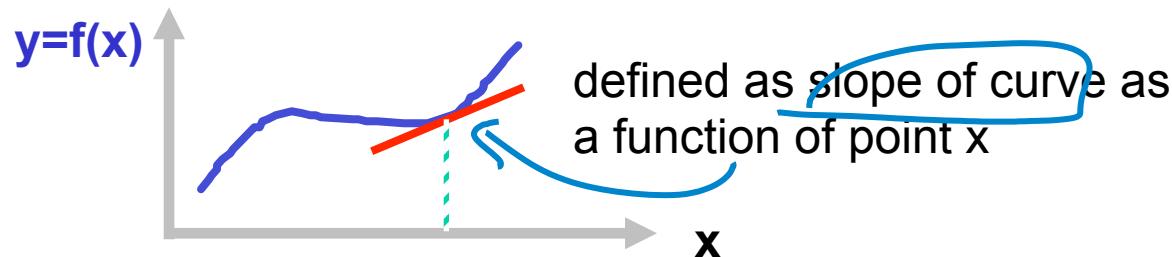
Combinations of Cofactors

- 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



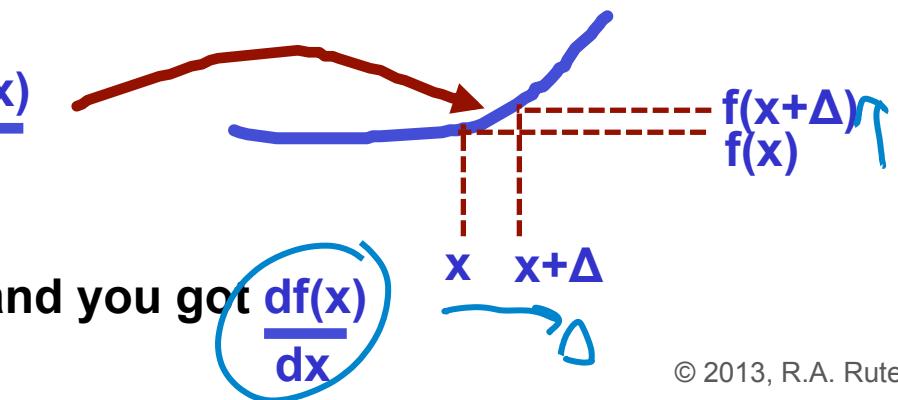
Calculus Revisited: Derivatives

- 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 $\frac{df(x)}{dx}$



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 $0 \leftrightarrow 1$, but can still ask how f changes with x ...

$$\begin{aligned} f_{x_0} \oplus a \oplus b \\ = \bar{a} \bar{b} + \bar{a} b \\ = 1 \quad \text{if} \\ a \neq b \end{aligned}$$

Boolean $f(x)$:

$$\partial f / \partial x =$$

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

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



It's Got a Name: Boolean *Difference*

- Hey, we have seen these pieces before!
 - $\partial f / \partial x$ = 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
 - $\partial f / \partial x \partial y = \partial f / \partial y \partial x$
 - Derivative of exor is exor of derivatives
 - $\partial(f \oplus g) / \partial x = \partial f / \partial x \oplus \partial g / \partial x$
 - If function f is actually constant ($f=1$ or $f=0$, always, for all inputs)
 - $\partial f / \partial x = 0$ for any x

like addition

like conj



Boolean Difference

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

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

!!

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



Boolean Difference: Gate-level View

- Try the obvious “simple” examples for $\frac{\partial f}{\partial x} = f_x \oplus \bar{f}_x$
- $f = \bar{x}$ $f_x = 0$ $\bar{f}_x = 1$ $\frac{\partial f}{\partial x} = 0 \oplus 1 = 1$

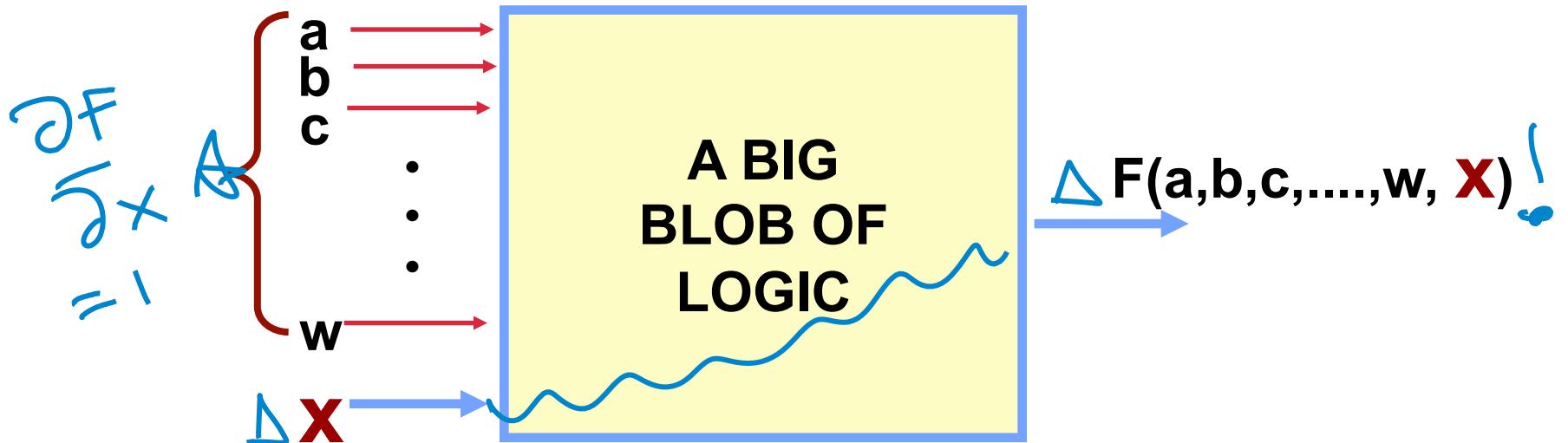
$f = xy$ $f_x = y$ $f_{\bar{x}} = 0$ $\frac{\partial f}{\partial x} = y \oplus \bar{y} \rightarrow 1$
- $f = x \oplus y$ $f_x = 1$ $f_{\bar{x}} = y$ $\frac{\partial f}{\partial x} = 1 \oplus y \rightarrow \bar{y}$

$f = x \otimes y$ $f_x = \bar{y}$ $f_{\bar{x}} = y$ $\frac{\partial f}{\partial x} = y \oplus \bar{y} = 1$

Meaning: when $\frac{\partial f}{\partial x} = 1$, then f changes if x changes(!)



Interpreting the Boolean Difference



When $\partial F / \partial x (a, b, c, \dots, w) = 1$, 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 $F()$



Boolean Difference: Example



$$\partial \text{Cout} / \partial \text{Cin} = ?$$

$$\text{Cout}_{\text{cin}} = ab + (\overline{a} \overline{b}) = ab$$

$$\text{Cout}_{\overline{\text{cin}}} = ab$$

$$, \text{ if } a \neq b$$

$$\Delta \text{cin} \rightarrow \Delta \text{cout}$$

$$!$$

$$\frac{\partial \text{Cout}}{\partial \text{cin}} = \frac{(ab) \ominus ab}{(ab)ab + (\overline{a}\overline{b})\overline{ab}}$$

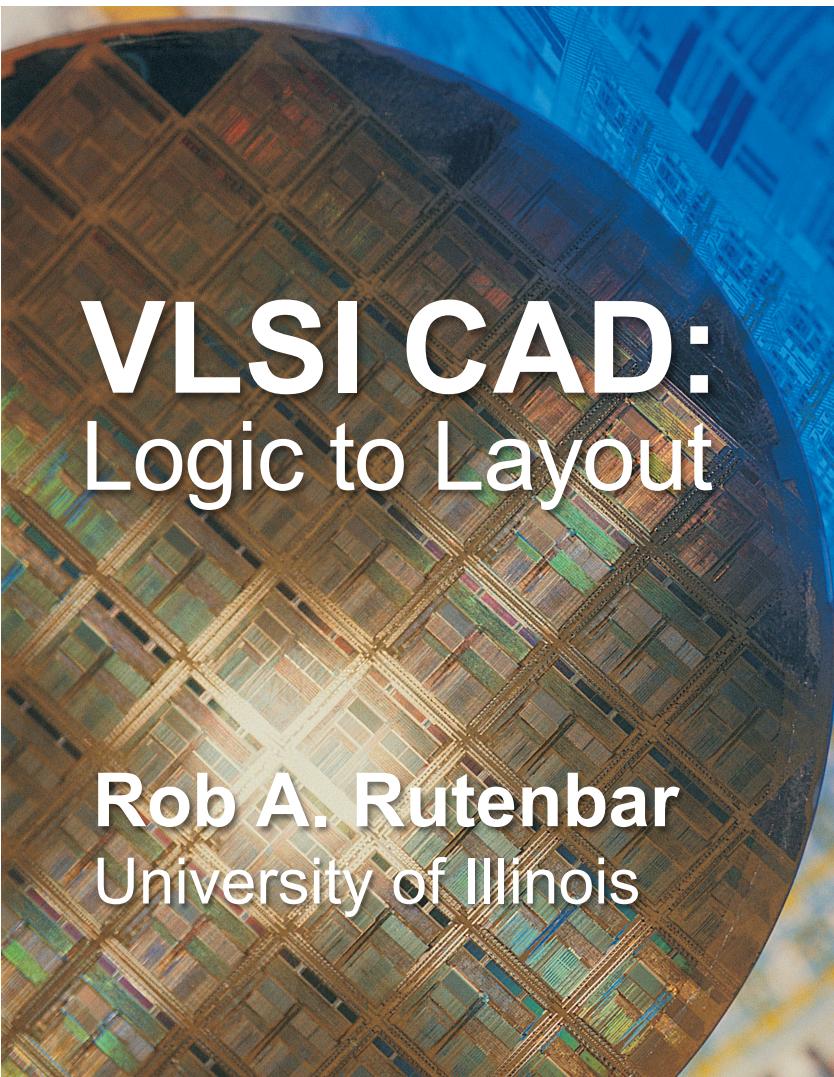
$$\underbrace{a \ominus b}$$



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 $F()$ 
 - $\partial f / \partial x$ 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...)**





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 2.3

Computational Boolean Algebra: Quantification Operators



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



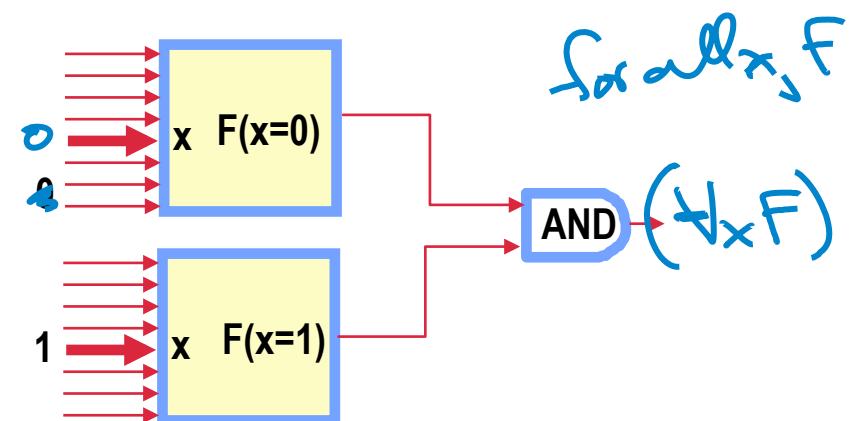
- **What you don't know**

- Other combinations of cofactors that do useful things
 - The big ones: **Quantification operators** (this lecture)
 - **Applications:** Being able to do something impressive (next lecture)



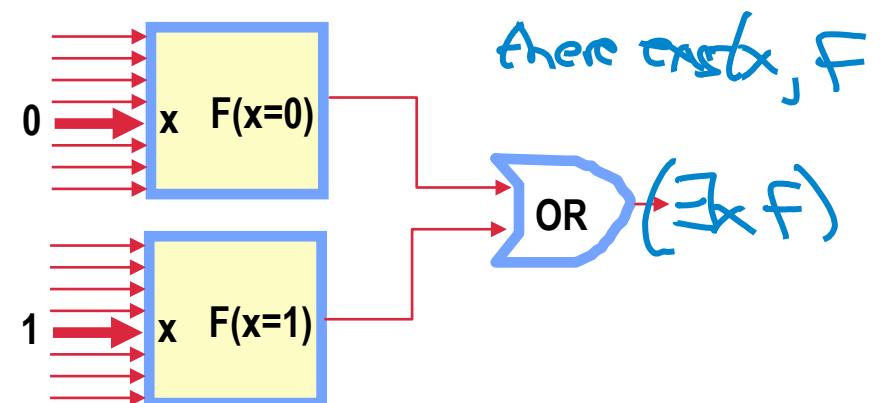
AND: $Fx \cdot Fx'$ is *Universal Quantification*

- Have $F(x_1, x_2, \dots, x_i, \dots x_n)$
- AND cofactors: $F_{xi} \cdot F_{xi'}$
 - Name: **Universal Quantification** of function F with respect to (wrt) variable x_i
- $(\forall x_i F) [x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots x_n]$
- “ $(\forall x_i F)$ ” is a **new function**
 - Yes, the ‘ \forall ’ sign is the “for all” symbol from logic (predicate calculus)
 - And, it does not depend on x_i ...



OR: $Fx + Fx'$ is *Existential Quantification*

- Have $F(x_1, x_2, \dots, x_i, \dots x_n)$
- OR the cofactors: $F_{xi} + F_{xi'}$
 - Name: **Existential Quantification** of function F wrt variable x_i
 - $(\exists x_i F) [x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots x_n]$
- “ $(\exists x_i F)$ ” is a **new** function
 - “ \exists ” sign is “there exists” from logic; and function also does not depend on x_i

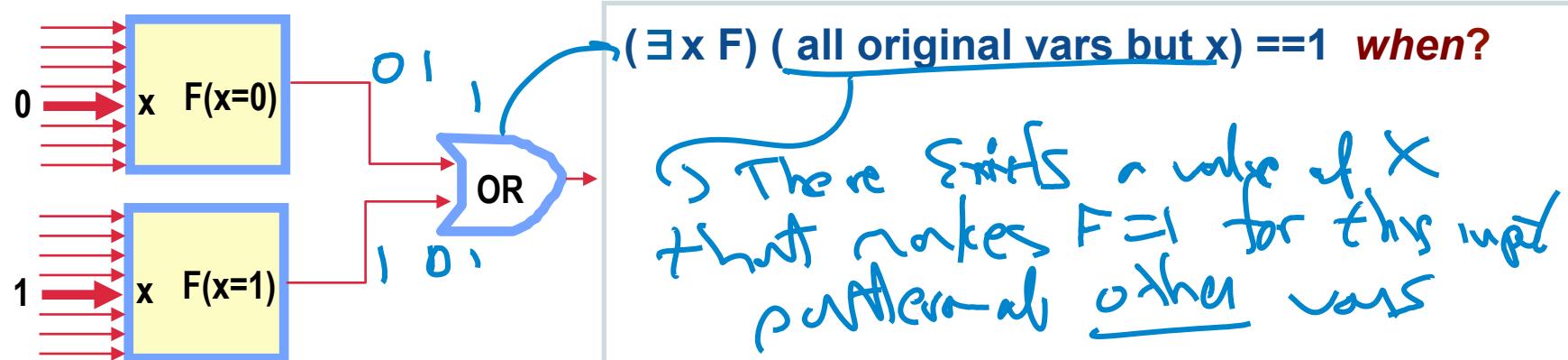
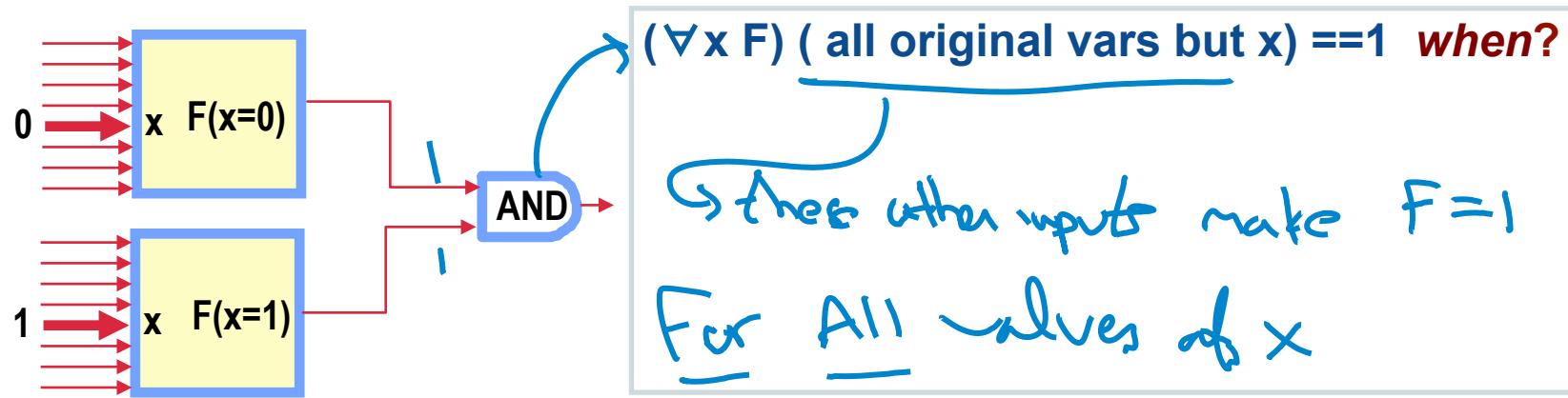


Note, like anything involving cofactors, both these new functions **do not** depend on x_i

So: $(\forall x_i F)$ and $(\exists x_i F)$ both omit x_i



Quantification Notation Makes Sense...



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: $(\forall xy F)[z,w] = (\forall x (\forall y F)) = F_{xy} \cdot F_{x'y} \cdot F_{xy'} \cdot F_{x'y'} \text{ AND}$
- Example: $(\exists xy F)[z,w] = (\exists x (\exists y F)) = F_{xy} + F_{x'y} + F_{xy'} + F_{x'y'} \text{ OR}$

- **Remember!**

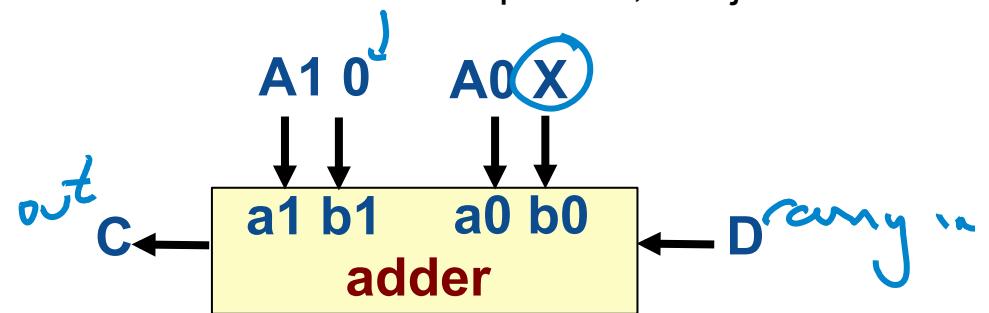
- $(\forall x F)$, $(\exists x F)$, and $\partial F / \partial x$ are all **functions**...
- ..but they are functions of all the vars **except x**
- We got rid of variable x and made 3 **new** functions



Quantification Example

- Consider this circuit, it adds $X=0$ or $X=1$ to a 2-bit number A_1A_0

- It's just a 2-bit adder, but instead of B_1B_0 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 $(\forall A_1, A_0 \ C)[X, D] \dots ?$
 - 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 A_1A_0 , i.e., makes a carry $C=1$ for all values of A_1A_0
- What is $(\exists A_1, A_0 \ C)[X, D] \dots ?$
 - A function of just X, D . Makes a **1** for values of X, D that make carry $C=1$, for **some value** of A_1A_0 , i.e., **there exists** some A_1A_0 that, for this X, D makes $C=1$



Quantification Example



Do the math:
Need all 4 cofactors:

$$C = A_1 A_0 X + A_1 (A_0 + X) D$$

$C_{A_1 A_0}$	$C_{A_1' A_0}$	$C_{A_1 A_0'}$	$C_{A_1' A_0'}$
$X+D$	0	XD	0

- Compute $(\forall A_1, A_0 C)[X, D]$
 - $C_{A_1 A_0} \cdot C_{A_1' A_0} \cdot C_{A_1 A_0'} \cdot C_{A_1' A_0'}$ ↗ N ↘
- In words: No values of X, D that make $C=1$ independent of A_1, A_0



- Compute $(\exists A_1, A_0 C)[X, D]$

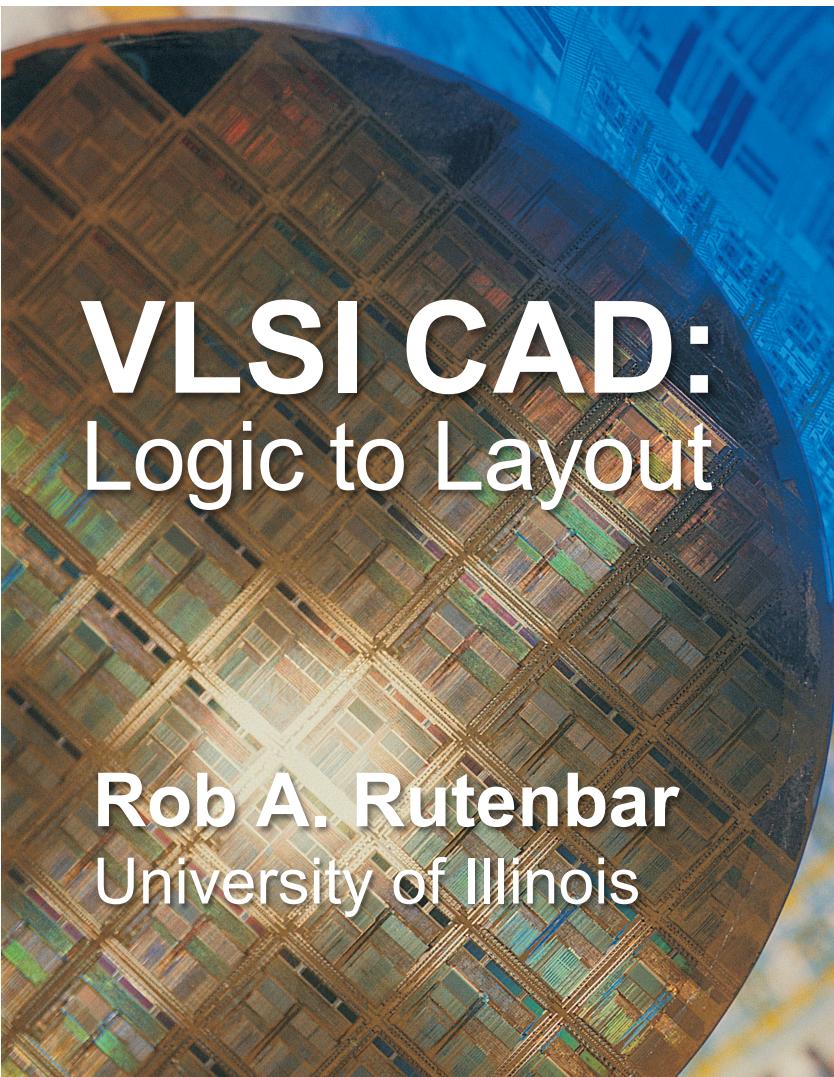
- $C_{A_1 A_0} + C_{A_1' A_0} + C_{A_1 A_0'} + C_{A_1' A_0'}$

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

BUG fix.
 $C=1$ for
some
values
of A_1, A_0

- In words: Yes, if at least one of $X, D = 1 \rightarrow C=1$ for some A_1, A_0





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 2.4

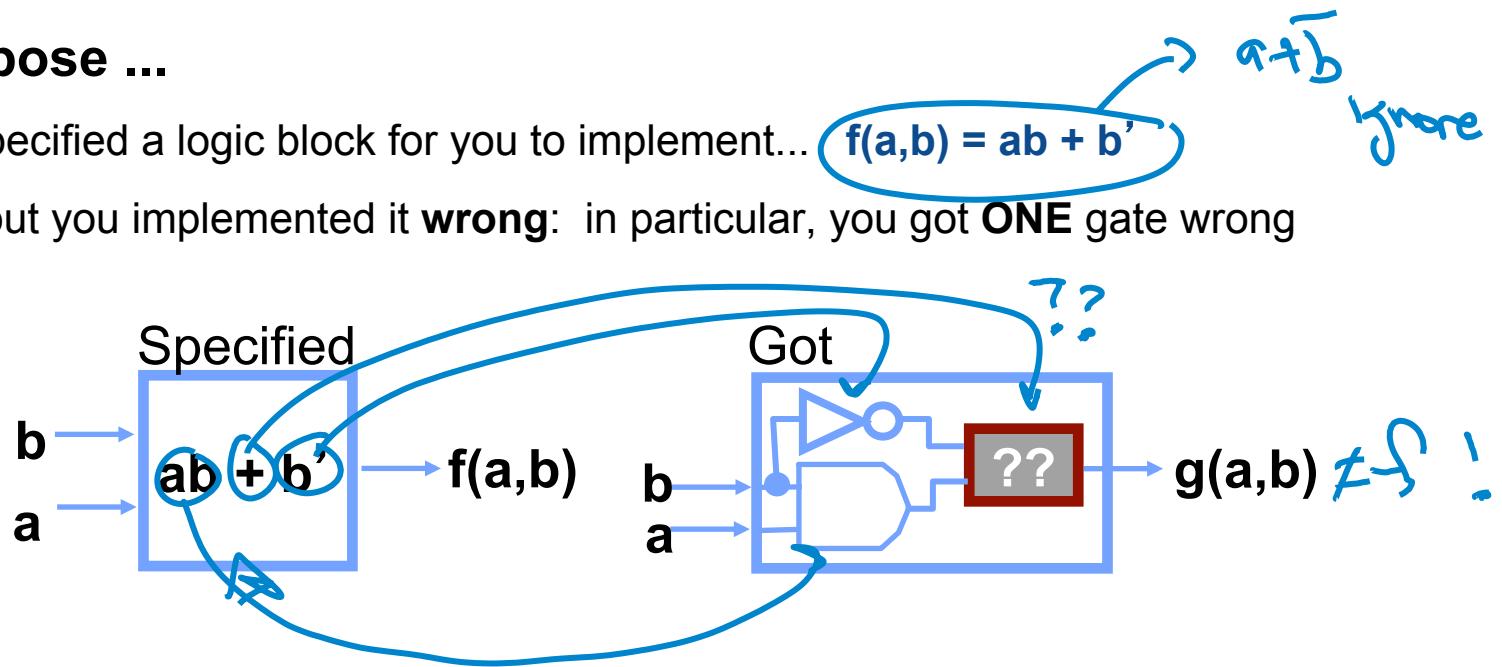
Computational Boolean Algebra: Application to Logic Network Repair



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**

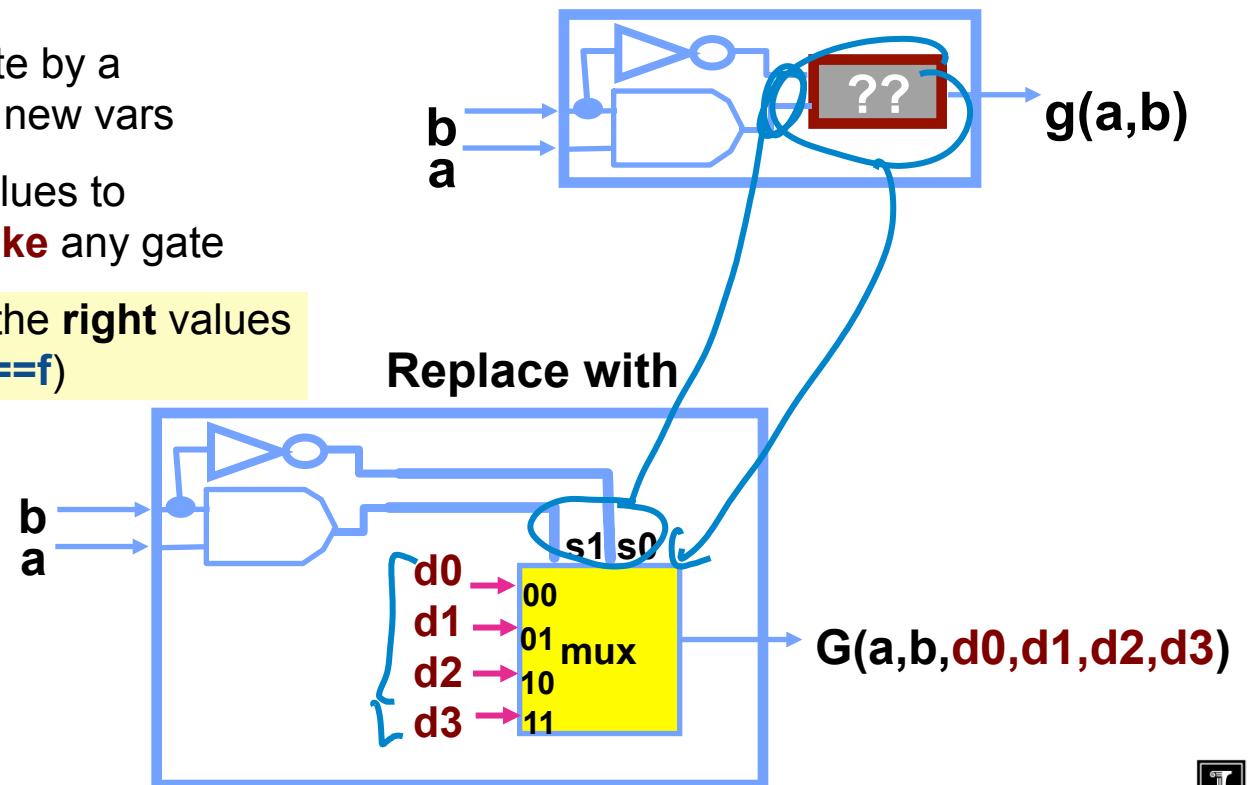
- Can we deduce how precisely to **change this gate** to restore correct function?
- Lets go with this very trivial test case to see how mechanics work...



Network Repair

- **Clever trick**

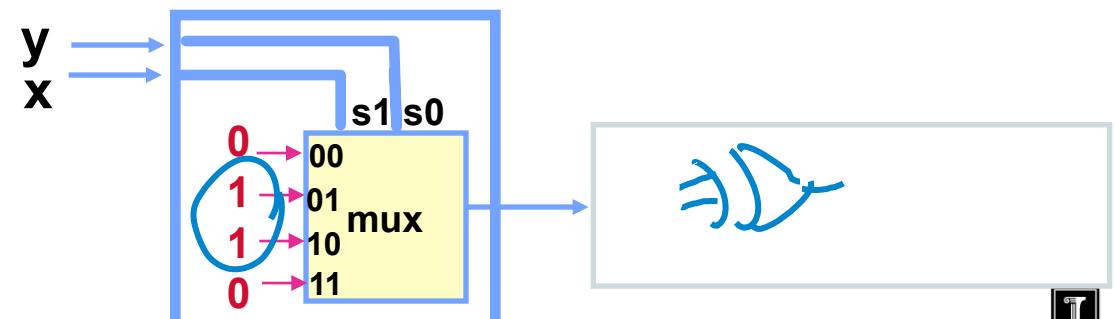
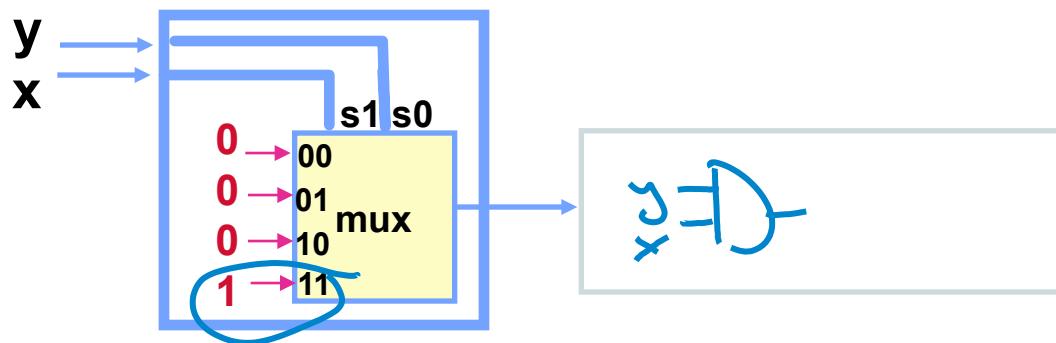
- Replace our suspect gate by a **4:1 mux** with **4** arbitrary new vars
- By cleverly assigning values to **d0 d1 d2 d3**, we can **fake** any gate
- **Question is:** what are the **right** values of **d's** so **g** is repaired ($\equiv f$)



Aside: Faking a Gate with a MUX

- **Remember...**

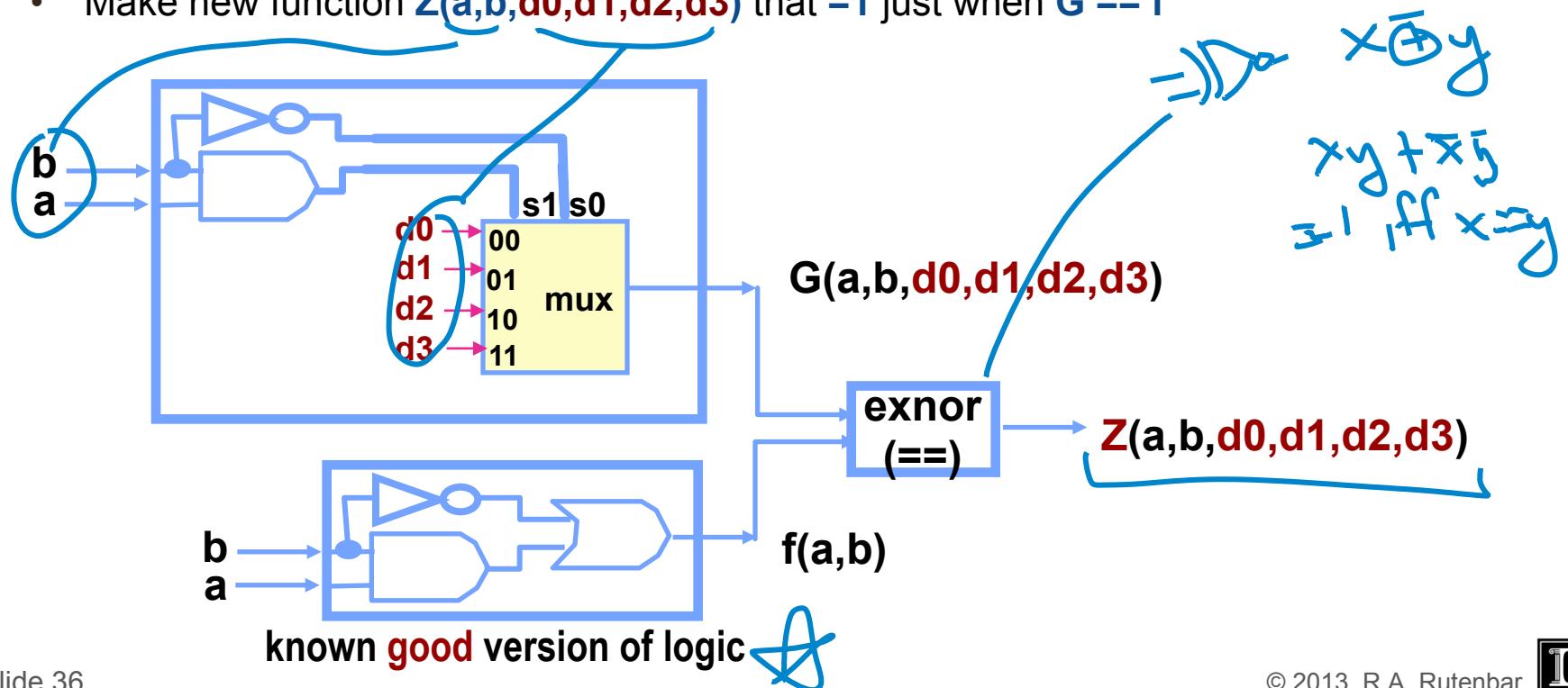
- You can do **any** function of 2 vars with one 4 input multiplexor (MUX)



Network Repair: Using Quantification

- Next trick

- Make new function $Z(a,b,d_0,d_1,d_2,d_3)$ that =1 just when $G == f$



Using Quantification

- **What now?**

- Think hard about exactly what we want:

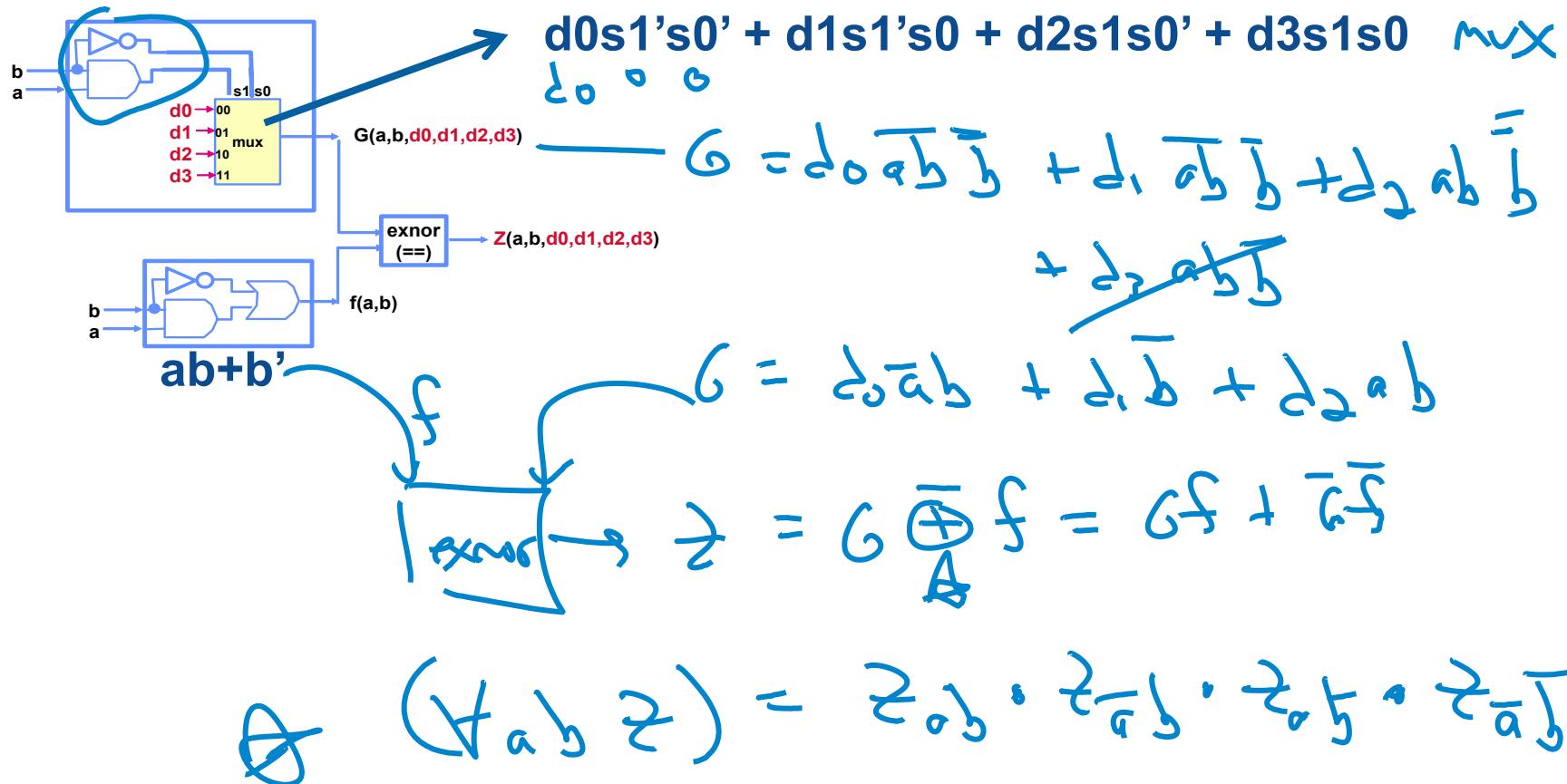
values of $d_0 d_1 d_2 d_3$ that make $Z == 1$
for all possible values of inputs a, b .
 $(\forall ab Z)(d_0, d_1, d_2, d_3) == 1 ?$

- **But this is something we have seen!**

- **Universal quantification** of function Z wrt variables a, b !
- Any pattern of $(d_0 d_1 d_2 d_3)$ that makes $(\forall ab Z)(d_0, d_1, d_2, d_3) == 1$ will do it!
- (Aside: do you know where a, b went??)



Network Repair via Quantification: Try It...



Network Repair via Quantification: Continued

$$Z = \overbrace{[d_0a'b + d_1b' + d_2ab]}^G \oplus \overbrace{[ab + b']}^f = G \text{ exnor } f$$

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

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

$$Z_{a'b'} = G_{a'b'} \oplus f_{a'b'} \rightarrow \text{set } a=0, b=0 \rightarrow$$

$$\overline{a}\overline{b}' = 1$$

$$Z_{a'b} = G_{a'b} \oplus f_{a'b} \rightarrow \text{set } a=0, b=1 \rightarrow$$

$$\overline{a}\overline{b} = 0$$

$$Z_{ab'} = G_{ab'} \oplus f_{ab'} \rightarrow \text{set } a=1, b=0 \rightarrow$$

$$\overline{a}b' = 1$$

$$Z_{ab} = G_{ab} \oplus f_{ab} \rightarrow \text{set } a=1, b=1 \rightarrow$$

$$ab = 1$$

$$[(\exists a \exists b) (\overline{a}b' \neq 1 \wedge ab' \neq 1)] \wedge [(\exists a \exists b) (\overline{a}\overline{b} \neq 1 \wedge \overline{a}b' \neq 1)] = A \wedge B = \overline{A} \overline{B}$$



Repair via Quantification: Continued

- So, we got this: $(\forall ab Z)[d_0, d_1, d_2, d_3] \neq d_0' \cdot d_1 \cdot d_2$
- And know this: if can solve $(\forall ab Z)[d_0, d_1, d_2, d_3] == 1 \rightarrow$ repaired!
- Hey – this one is not very hard...

$$d_0 = 0$$

$$d_1 = 1$$

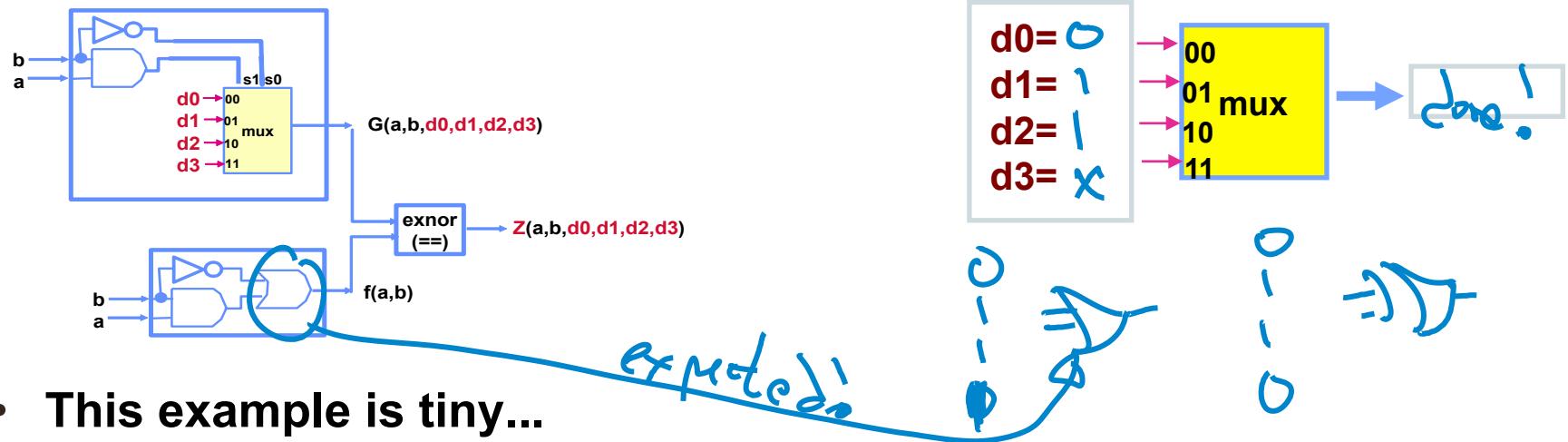
$$d_2 = 1$$

$$d_3 = \times \text{ doesn't care!}$$



Network Repair

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



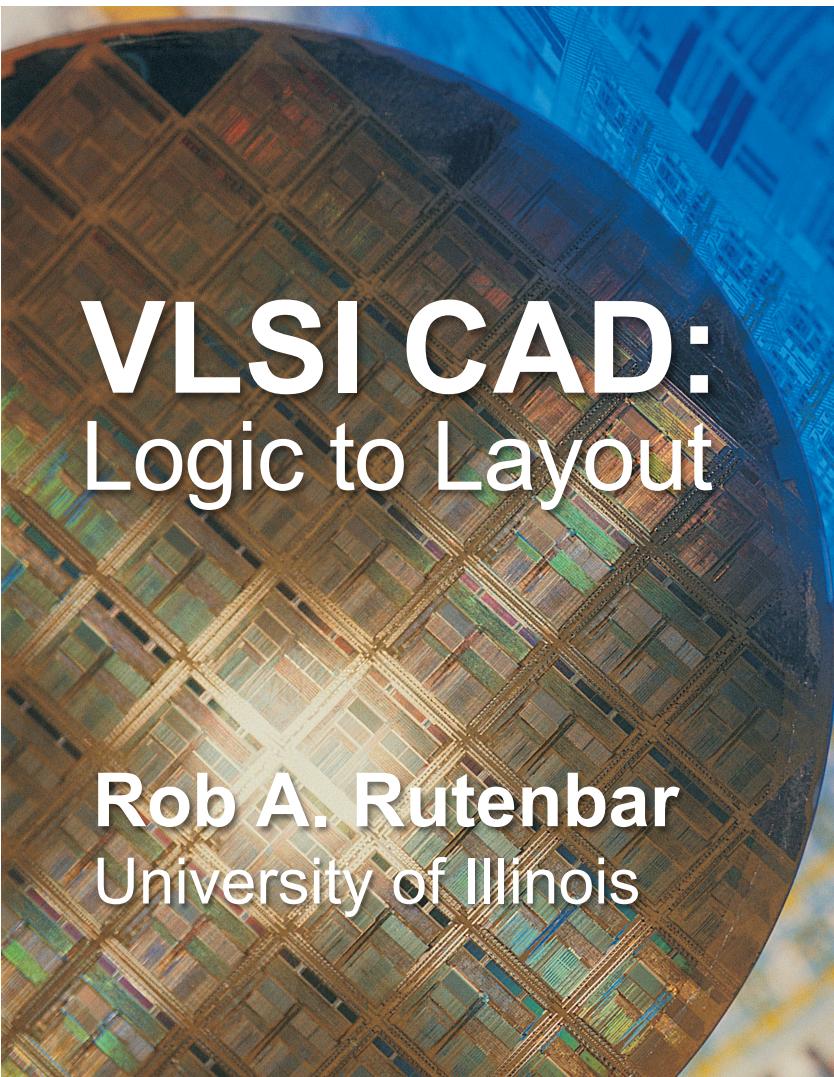
- This example is tiny...
- But in a real example, you have a **big** network-- 100 inputs, 50,000 gates
- When it doesn't work, it's a major hassle to go thru in detail
- This is a mechanical procedure to answer: Can we change 1 gate to **repair**?



Computational Boolean Alg Strategies

- What haven't we seen yet? **Computational strategies**
 - Example: find inputs to make $(\forall ab Z)(d_0, d_1, d_2, d_3) == 1$ 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...





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 2.5

Computational Boolean Algebra: Recursive Tautology



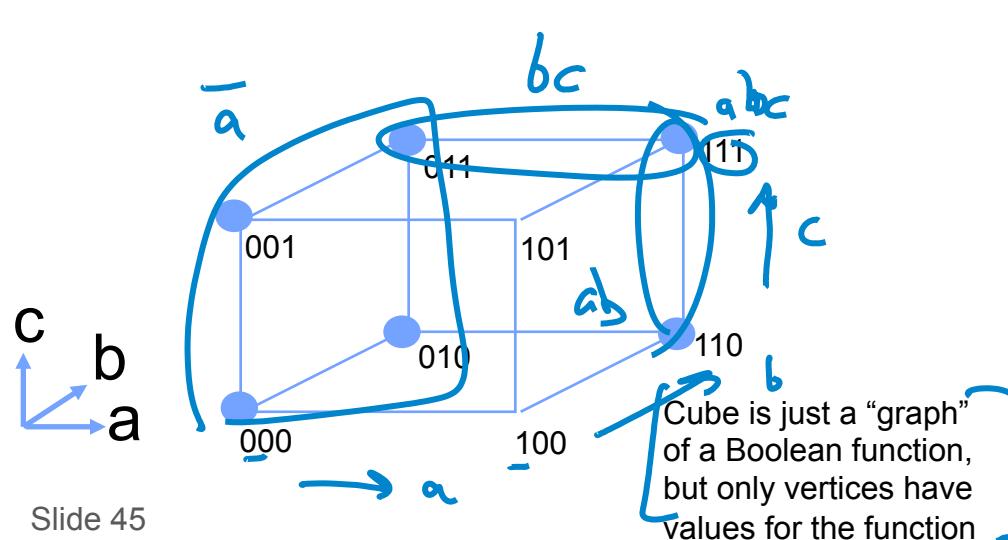
Important Ex Computation: Tautology

- 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...



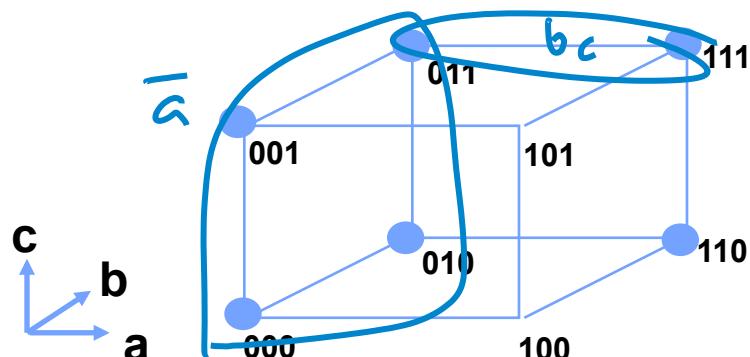
Start with: Representation

- 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



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



Slide 46

$$\bar{a} = [\bar{0} \quad \bar{1} \quad \bar{1}]$$

$$b_c = [1 \quad 01 \quad \alpha]$$

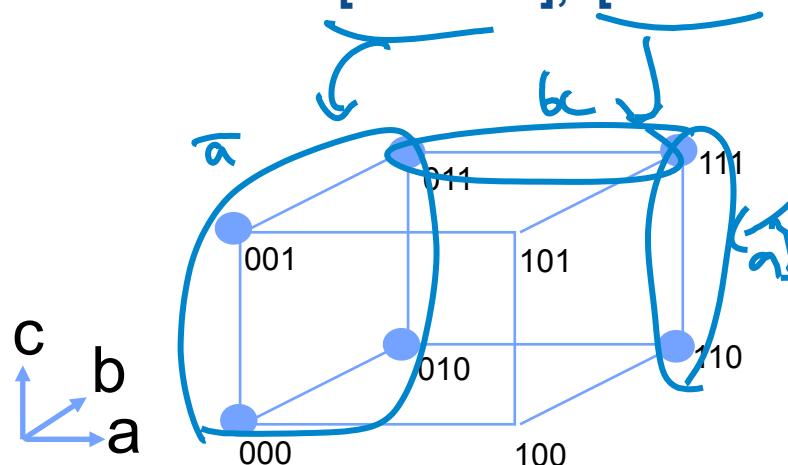
! *FOO*

© 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)
or
products
 - Ex: $f(a,b,c) = \bar{a}' + bc + ab \Rightarrow [01\ 11\ 11], [11\ 01\ 01], [01\ 01\ 11]$



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 f_x and $f_{x'}$ are both tautologies

- This makes sense:

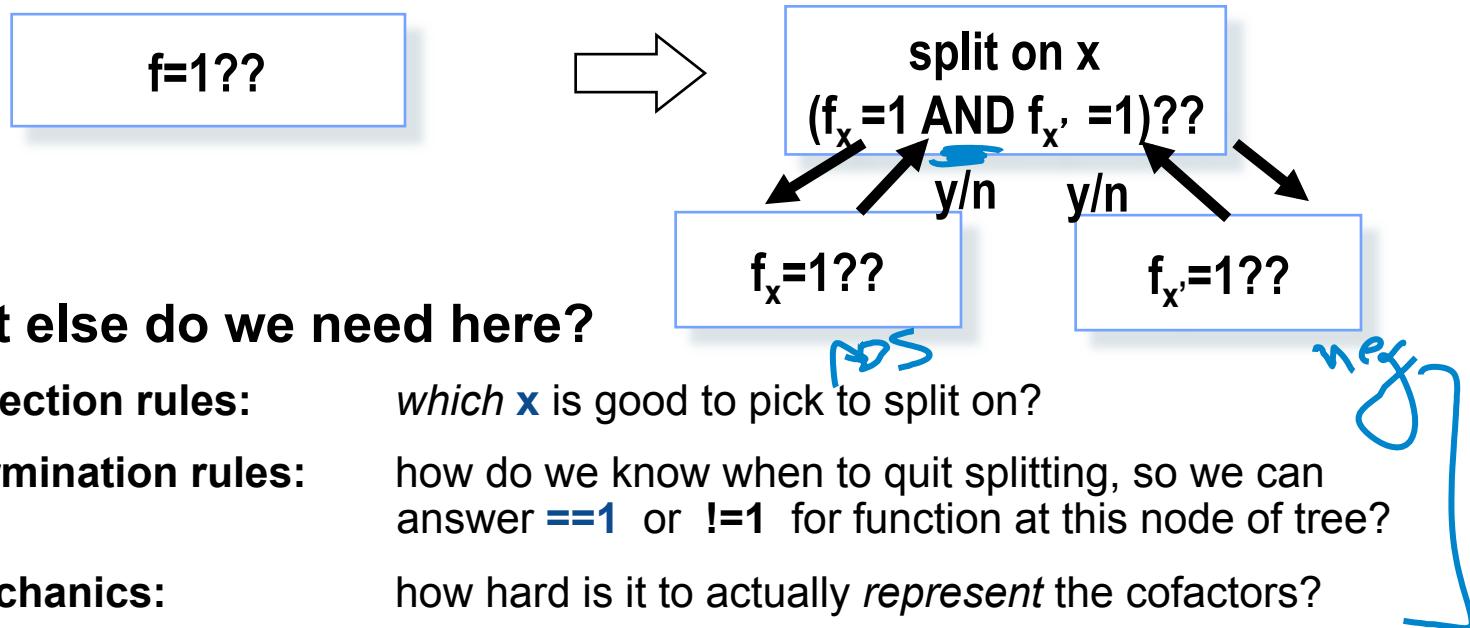
$$f(x=0), f(x=1) = 1$$

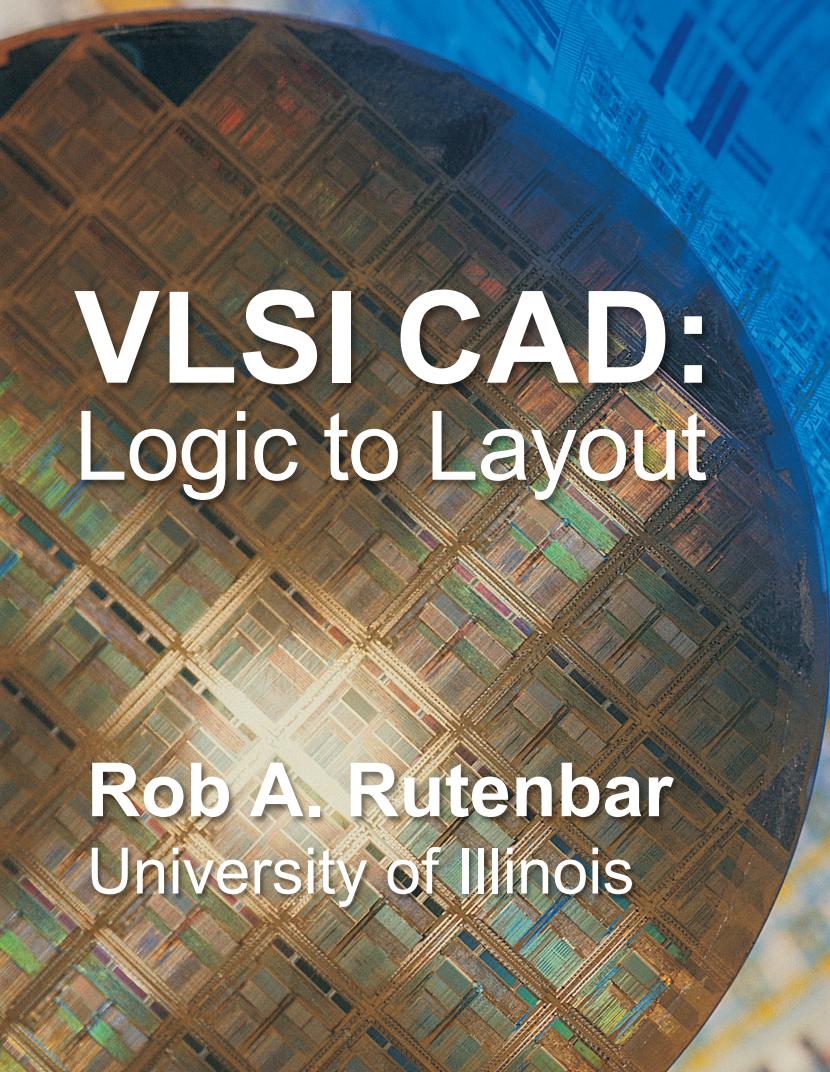
- If function $f() = 1$ → then cofactors both obviously = 1 ✓
- If both cofactors = 1 → $x \cdot F(x=1) + x' \cdot F(x=0) = x \cdot 1 + x' \cdot 1 = x + x' = 1$ ✓



Recursive Tautology Checking

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





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

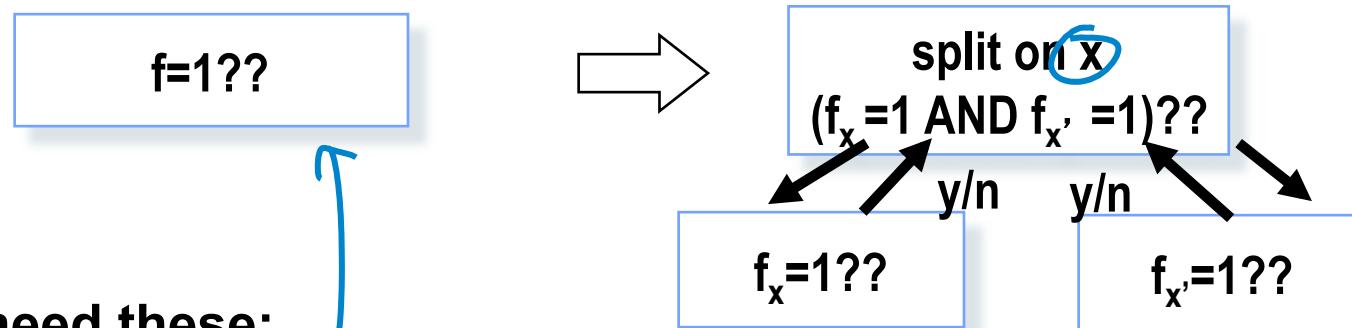
Lecture 2.6

Computational Boolean Algebra: Recursive Tautology— URP Implementation



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:
 - Termination rules:
 - Mechanics:

which x is good to pick to split on?

how do we know when to quit splitting, so we can answer $\equiv 1$ or $\neq 1$ for function at this node of tree?

how hard is it to actually *represent* the cofactors?



Recursive Cofactoring

- **Do mechanics first (easy!). For each cube in your list:** *recipe*
 - 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 $\textcircled{11}$ == 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 $\textcircled{11}$ == 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	$a \rightarrow 1$	f_c	$c \rightarrow 1$
$a\bar{b}\}$ $b\bar{c}$	$b\bar{c} [11\ 01\ 1\ 01]$	$b\bar{c} [11\ 01\ 1\ 01]$	$a\bar{b} [01\ 01\ 11\ 01]$	
$[11\ 01\ 10\ 11]$	$b\bar{c} [11\ 01\ 10\ 11]$			



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: $\underline{xy} + \underline{x'y} + \underline{xyz'} + z'$ is ... ~~not unate~~: x, y, \bar{x}
is unate in var y

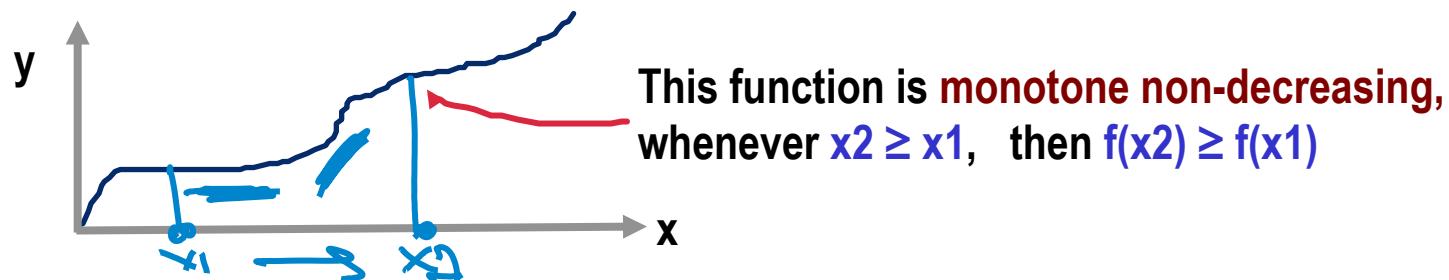
- **Terminology**

- f is **positive unate** in var x -- if changing $x 0 \rightarrow 1$ keeps f **constant** or makes $f: 0 \rightarrow 1$
- f is **negative unate** in var x -- if changing $x 0 \rightarrow 1$ keeps f **constant** or makes $f: 1 \rightarrow 0$
- Function that is not unate is called **binate**

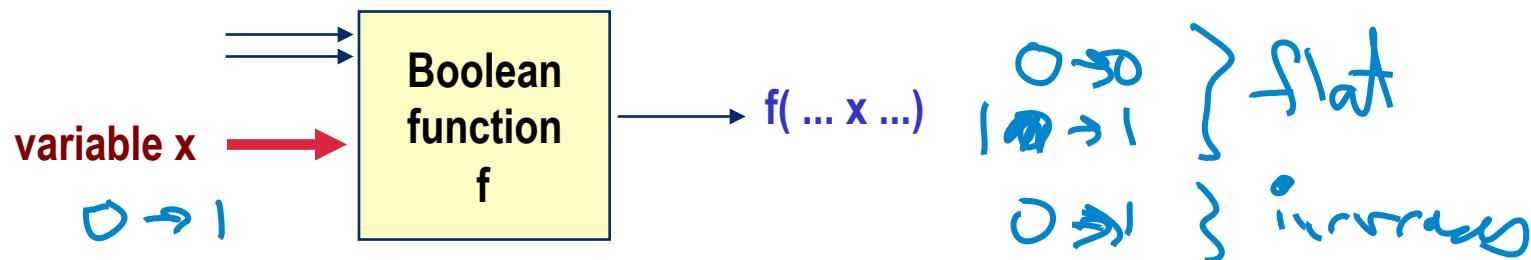


Unate Functions

- Analogous to **monotone continuous functions**



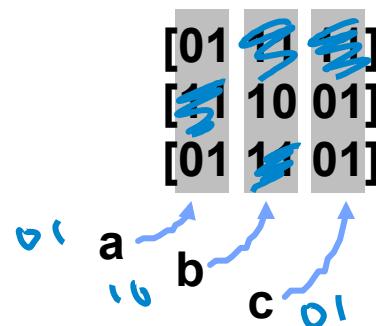
- Example: for a Boolean function f **positive unate** in x



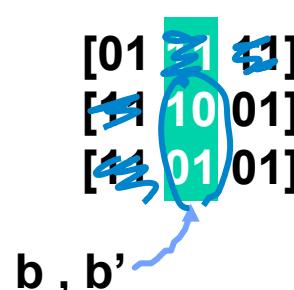
Can Exploit Unate Func's For Computation

- 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: $f(a,b,c)=a +bc +ac \rightarrow [01\ 11\ 11], [11\ 01\ 01], [01\ 11\ 01]$ is **unate**
 - Ex: $f(a,b,c)=a +b'c +bc \rightarrow [01\ 11\ 11], [11\ 10\ 01], [11\ 01\ 01]$ is **not**
 - Easier to see if draw vertically

$a+b'c+ac$ UNATE



$a+b'c+bc$ NOT



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 = $[11 \dots 11]$**

- Reminder: what exactly is $[11 11 11 \dots 11]$ as a product term?

$$[01 01 01] = abc \quad [01 01 11] = ab \quad [01 11 11] = a \quad [11 11 11] = 1$$

- **This result actually makes sense...**

- Cannot make a “1” with only product terms where all literals are in just **one polarity**. (Try it!)

	cd	00	01	11	10
ab	00				
00	00				
01	01				
11	11				
10	10				

© 2013, R.A. Rutenbar

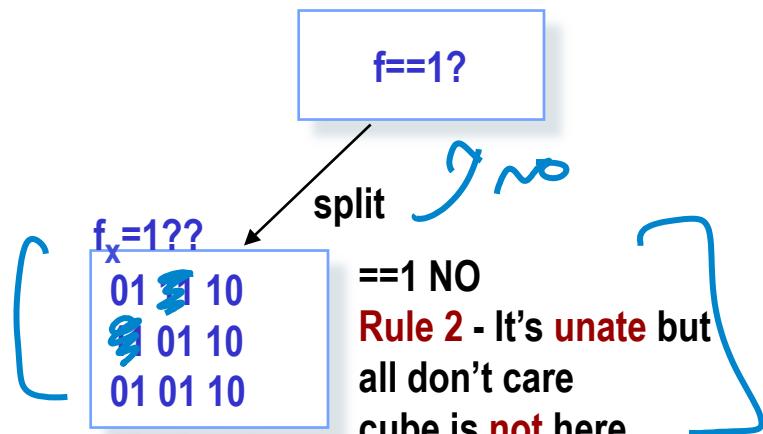
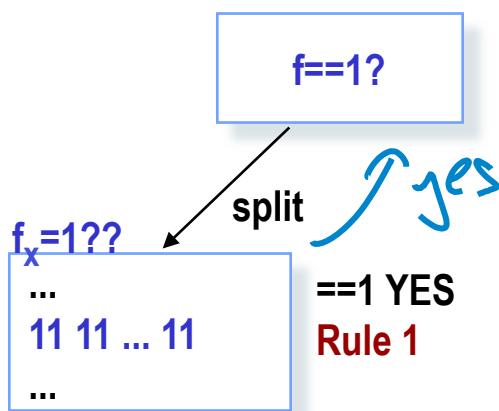


So, Unateness Gives Us Termination Rules

- We can look for tautology directly, if we have a unate cube-list
 - If match rule, know immediately if $\equiv 1$, or not

Rule 1: $\equiv 1$ if cube-list has all don't care cube [11 11 ... 11] easy - trivial
Why: function at this leaf is $(\text{stuff} + 1 + \text{stuff}) \equiv 1$

Rule 2: $\equiv 1$ if cube-list unate and all don't care cube missing
Why: unate $\equiv 1$ if and only if has [11 11 ... 11] cube

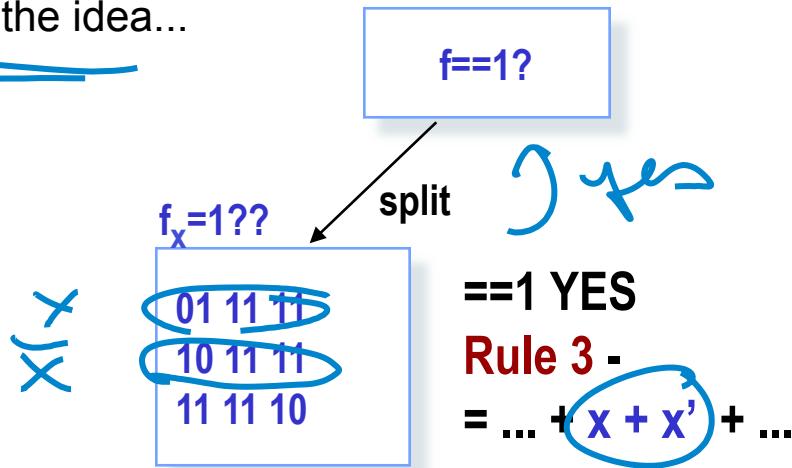


Recursive Tautology Checking

- Lots more possible rules...

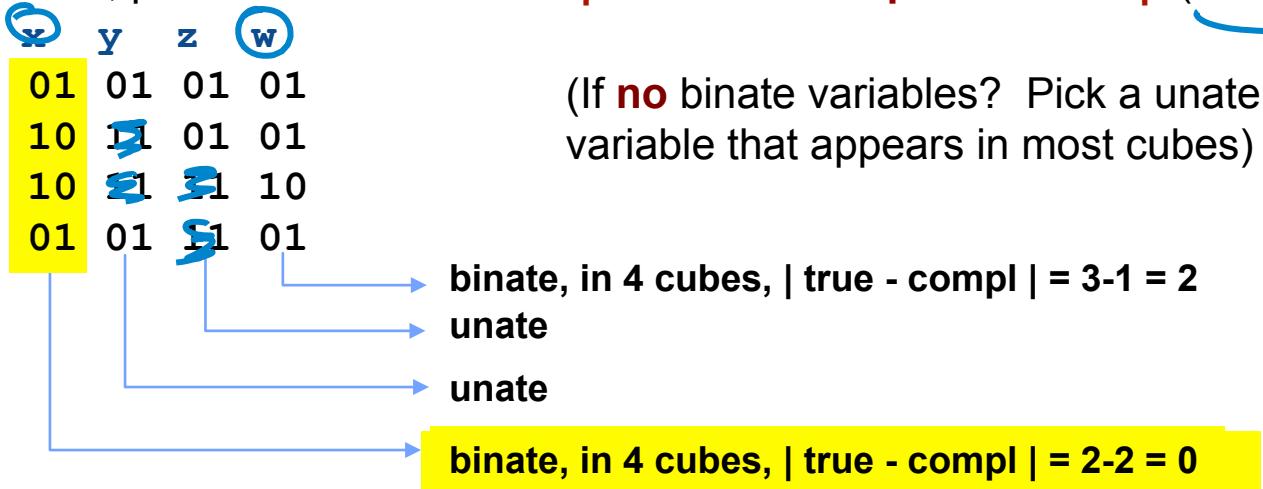
- Rule 3: ==1 if cube list has single var cube that appears in both polarities
Why: function at this leaf is $(\text{stuff} + x + x' + \text{stuff}) == 1$

- You get the idea...



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 ↗ *heuristic!!*
 - 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*)



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 ($\equiv 1$) return (1)
 else return (0)
 }
 else if (any other termination rules, like rule 3, work?) {
 return the appropriate value if $\equiv 1$ or $\equiv 0$
 }
 else { /* can't tell from this -- find splitting variable */
 x = most-not-unate variable in f
 return (**tautology**(f_x) && **tautology**($f_{x'}$))
 }
}
- terminates*
- select* *recurse*



Recursive Tautology Checking: Example

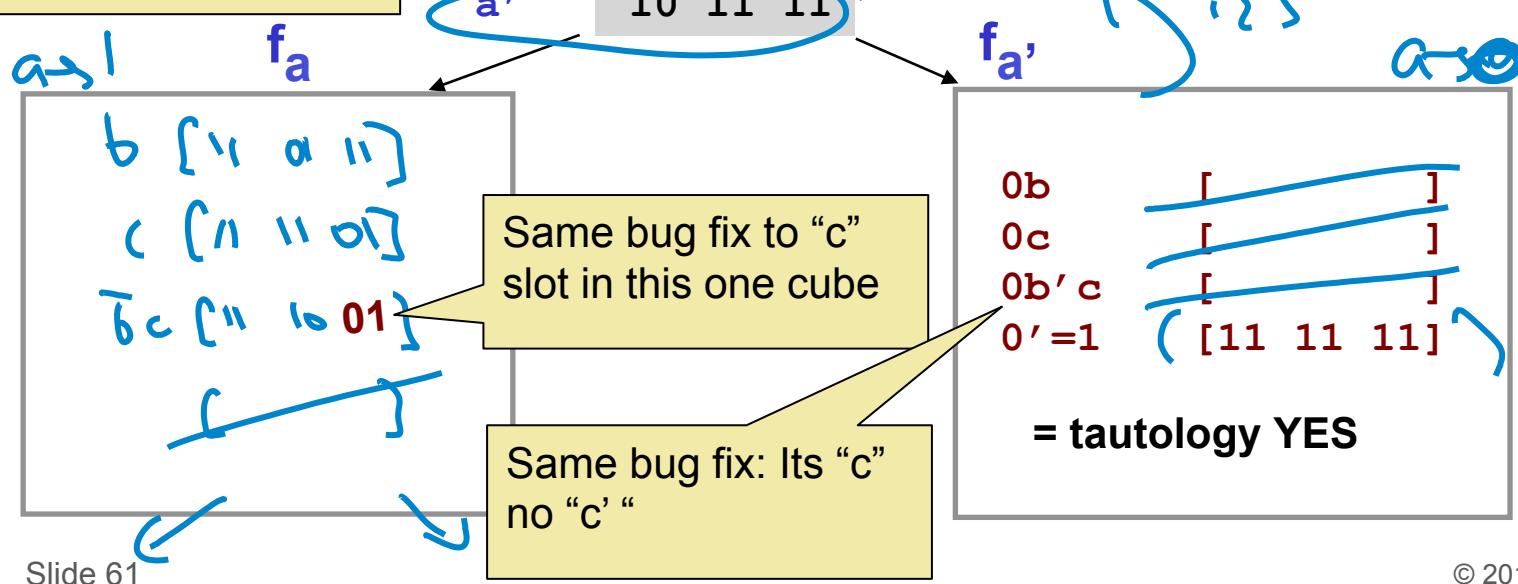
- Tautology example: $f = ab + ac + ab'c' + a'$

Dear Courserians:

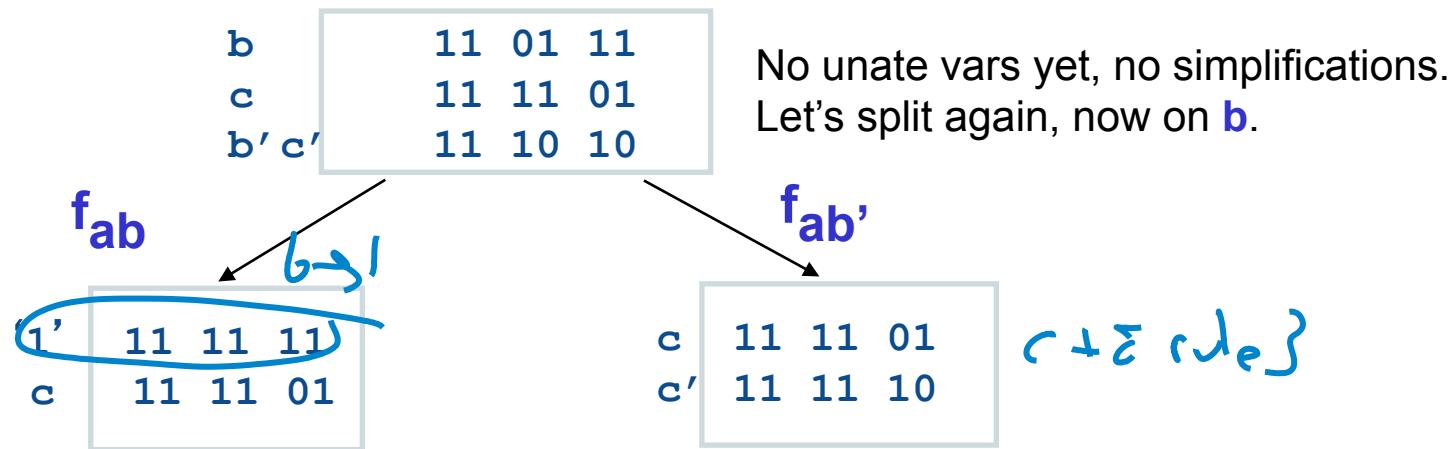
Small bug fix to "c" slot in this one cube

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

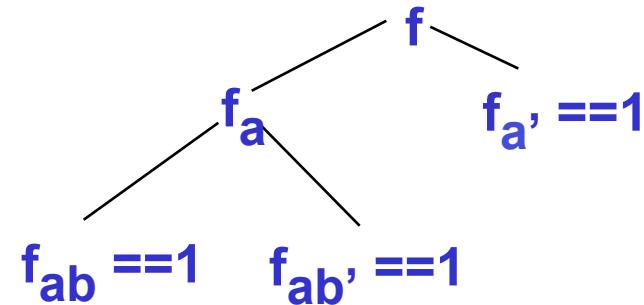
All vars are binate, var **a** affects most implicants, so split on **a**...



Recursive Tautology Checking: Example



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



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

