

## (Lec 1) Advanced Boolean Algebra

### ▼ Assumptions

- ▶ You've seen basic Boolean algebra, and manipulations
- ▶ You've seen simplification-related ideas
  - ▶ Kmaps, Quine-McCluskey simplification, minterms, SOP, etc

### ▼ What's left...? Actually, a lot...

- ▶ Decomposition strategies
  - ▶ Ways of taking apart complex functions into simpler pieces
  - ▶ A set of standard advanced concepts, terms you need to see to be able to read the DeMicheli book (or the literature)
- ▶ Computational strategies
  - ▶ Ways to think about Boolean functions that allow them to be manipulated by programs
- ▶ Interesting applications
  - ▶ When you have new tools, there are some neat new things to do

© R. Rutenbar 2001 Fall 18-760 Page 1

## Copyright Notice

© Rob A. Rutenbar, 2001  
All rights reserved.

You may not make copies of this material in any form without my express permission.

© R. Rutenbar 2001 Fall 18-760 Page 2

# Handouts

## Physical

- ▶ Lecture 01 -- Advanced Boolean Algebra

## Electronic

- ▶ Nothing today

# Where Are We?

## Doing the Boolean background you need...

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

- Introduction
- Advanced Boolean algebra**
- JAVA Review
- Formal verification
- 2-Level logic synthesis
- Multi-level logic synthesis
- Technology mapping
- Placement
- Routing
- Static timing analysis
- Electrical timing analysis
- Geometric data structs & apps

## Readings

### ▼ De Micheli

- ▶ Chapter 1 -- once over, lightly
- ▶ Chapter 2 -- just Section 2.7
- ▶ Chapter 7 -- just Section 7.3
- ▶ Don't worry if it doesn't all make sense yet, the notes will explain

## Advanced Boolean Algebra

### ▼ Useful analogy to calculus...

- ▶ At some point somebody told you that you could 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  $\exp(x) = 1 + x + x^2/2! + x^3/3! + \dots$

- ▶ Later, somebody told you there was a *general* formula, called the *Taylor series* expansion

- ▶ If you took some more math, or EE, you might have found out that there were several other ways of representing arbitrary  $f(x)$ 
  - ▶ If it's a periodic function, can use a Fourier series
  - ▶ Other polynomials, eg, Legendre polynomials

### ▼ Question: *Anything like this for Boolean functions?*

## Boolean Decompositions

▼ Yes. Called the *Shannon Expansion*

▼ A little refresher in notation first...

- ▶ F is a Boolean function of n variables  $x_1, x_2, \dots, x_n$
- ▶ Let  $B = \{0,1\}$  then we write formally:



- ▶ We often refer to the variables  $x_1, x_2, \dots, x_n$  by lumping them together in a set  $\{x_1, x_2, \dots, x_n\}$  called the **support** of F, or **sup(F)**.

© R. Rutenbar 2001 Fall 18-760 Page 7

## 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

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

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

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

▼ Important to remember that result is a *new function*

- ▶ Note that new function no longer depends on this variable

© R. Rutenbar 2001 Fall 18-760 Page 8

## 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 see as just  which is easier to type

- ▶ Restriction of  $F$  on variable  $x_i$

- ▶ 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

### Why are these useful functions to get from $F$ ?

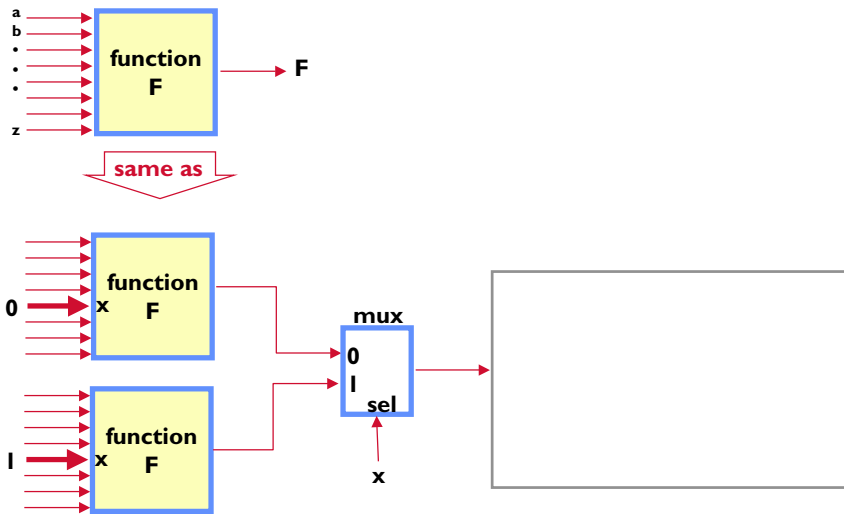
## Shannon Expansion Theorem

### Shannon Expansion Theorem

- ▶ Given any Boolean function  $F(x_1, x_2, \dots, x_n)$  and any  $x_i$  in the support of  $F()$ ,  $F()$  can be represented as

- ▶ Pretty easy to prove...

## Shannon Expansion: Another View



© R. Rutenbar 2001 Fall 18-760 Page 11

## Shannon Expansion: Multiple Variables

▼ Can do it on more than one variable, too

- ▶ Just keep on applying the theorem
- ▶ Example

$$F(x,y,z,w) = x \cdot F(x=1) + x' \cdot F(x=0) \text{ expanded around } x$$

Expand each cofactor around y

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

= expanded around variables x and y

© R. Rutenbar 2001 Fall 18-760 Page 12

## Shannon Cofactors: Multiple Variables

▼ BTW, there's 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  $F_{x_i \overline{x_j}}$

▶ Ditto for any number of variables  $x_i, x_j, x_k, \dots$

▶ Notice also that order doesn't matter:  $(F_x)_y = (F_y)_x = F_{xy}$

▶ For our example

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

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

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

## 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  $F, G$ ?

▶ For example,

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

## Properties of Cofactors

### ▼ More nice properties...

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

- ▶ Complements

- ▶ In English: *cofactor of complement is complement of cofactor*

- ▶ Binary boolean operators

*cofactor of AND is AND of cofactors*

*cofactor of OR is OR of cofactors*

*cofactor of EXOR is EXOR of cofactors*

- ▶ In fact, true for *ANY* binary operator on Boolean functions
- ▶ 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 $F_{x'}$

- ▶  $F_x \oplus F_{x'} = ?$

- ▶  $F_x \cdot F_{x'} = ?$

- ▶  $F_x + F_{x'} = ?$

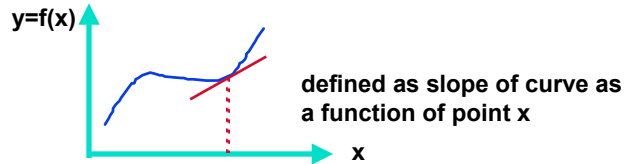
### ▼ Turns out these are all useful *new* functions

- ▶ We'll start with most obvious one to get...
- ▶ Need to remember some more calculus...

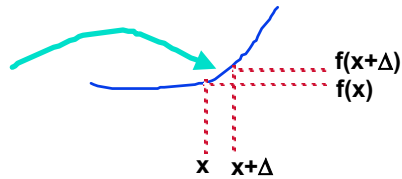
# Derivatives

## Remember way back to how you defined derivatives?

- Suppose you have  $y = f(x)$



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



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

# Boolean Derivatives

## OK, do Boolean functions have derivatives?

- Actually, yes. Trick is how to *define* them...

### Basic idea

- For real-valued  $f(x)$ ,  $df/dx$  tell hows **f changes** when **x changes**
- For 0,1-valued Boolean function, we can't *change*  $x$  by small  $\Delta$
- Can only change  $0 \leftrightarrow 1$ , but can still ask how  $f$  changes with  $x$  ...

Real-valued  $f(x)$ :

$$\frac{df}{dx} = \lim_{\Delta \rightarrow 0} \frac{f(x + \Delta) - f(x)}{\Delta}$$

Binary-valued  $f(x)$ :

$$\frac{df}{dx} = \boxed{\phantom{0}}$$

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

## Boolean Difference

### ▼ Hey, we've seen these pieces before!

- ▶  $df/dx$  = exor of the Shannon cofactors with respect to  $x$
- ▶ Also often written as  $\partial f/\partial x$
- ▶ Called the **Boolean Difference** of function  $f$  wrt variable  $x$

### ▼ It also behaves sort of like regular derivatives...

- ▶ Order of vars doesn't matter

$$\partial f / \partial x \partial y = \text{_____}$$

- ▶ Derivative of exor is exor of derivatives

$$\partial(f \oplus g) / \partial x = \text{_____}$$

- ▶ If function  $f$  is actually constant ( $f=1$  or  $f=0$ , always, for all inputs)

$$\partial f / \partial x = \text{_____}$$

- ▶ If function  $f$  doesn't depend on var  $x$  (ie change  $x$ ,  $f$  never changes)

$$\partial f / \partial x = \text{_____}$$

$$\partial(f \cdot g) / \partial x = \text{_____}$$

$$\partial(f + g) / \partial x = \text{_____}$$

© R. Rutenbar 2001 Fall 18-760 Page 19

## Boolean Difference

### ▼ continued...

- ▶ If  $f$  is a function only of  $x$  (only changing  $x$  can change  $f$ )

$$\partial f / \partial x = \text{_____}$$

### ▼ But some things are just *more* complex, though...

- ▶ Derivatives of  $(f \cdot g)$  and  $(f + g)$  don't work the same...

$$\partial(f \cdot g) / \partial x =$$

$$\partial(f + g) / \partial x =$$

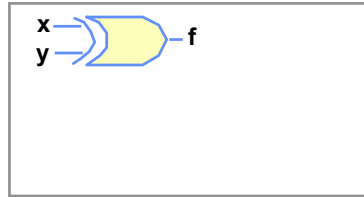
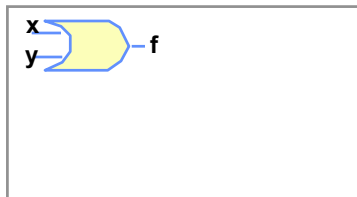
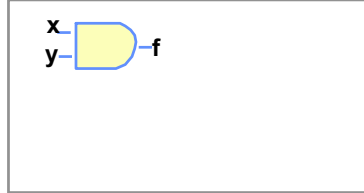
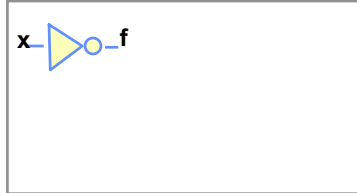
- ▶ Why?

- ▶ Because AND and OR on Boolean values don't always behave like ADDITION and MULTIPLICATION on real numbers

© R. Rutenbar 2001 Fall 18-760 Page 20

## Boolean Difference: Gate-level View

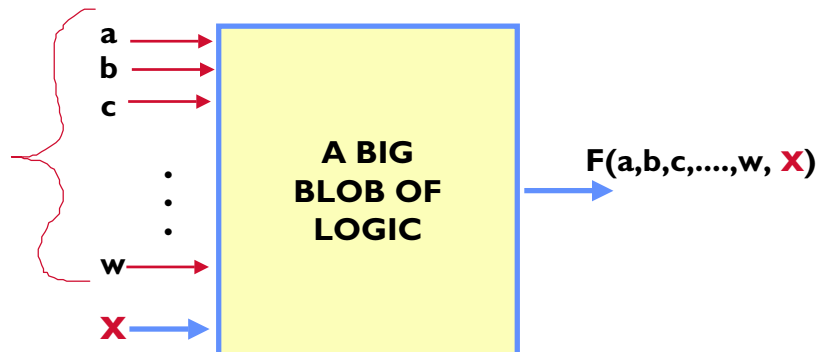
▼ Try the obvious “simple” examples for  $\partial f / \partial x$



Interpretation: when  $\partial f / \partial x = 1$ , then  $f$  changes as  $x$  changes

© R. Rutenbar 2001 Fall 18-760 Page 21

## Interpreting the Boolean Difference



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

© R. Rutenbar 2001 Fall 18-760 Page 22

## Boolean Difference: Example

▼ Try another example



$$s = a \oplus b \oplus cin$$
$$cout = ab + (a + b)cin$$

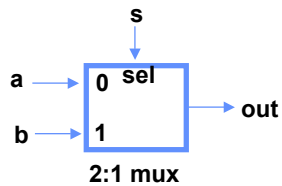
$$\partial s / \partial a =$$

$$\partial cout / \partial cin =$$

© R. Rutenbar 2001 Fall 18-760 Page 23

## Boolean Difference: Example

▼ Example



$$out = as' + bs$$

$$\partial out / \partial s =$$

$$\partial out / \partial a =$$

© R. Rutenbar 2001 Fall 18-760 Page 24

## Boolean Difference

### ▼ Things to remember about Boolean Difference

- ▶ Not as easy to assign a physical interpretation like ordinary derivative (ie, no “slope of the curve” sort of stuff)
- ▶  $\partial f / \partial x$  is another Boolean **function**, but it does **not** depend on  $x$ 
  - ▶ It **can't**, it's made out of the cofactors wrt  $x$ , and they eliminate all the  $x$  and  $x'$  terms by setting them to constants

### ▼ OK, it's cute, but is it useful...?

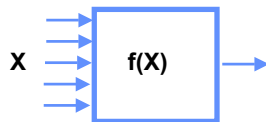
- ▶ Sure, let's look at a simplified application in **testing**

© R. Rutenbar 2001 Fall 18-760 Page 25

## Application: Testing

### ▼ Suppose you want to test some logic

- ▶ You want to figure out what inputs to apply to figure out if it works



You can't afford to apply all possible inputs

Example: 50 inputs = 1000 trillion patterns

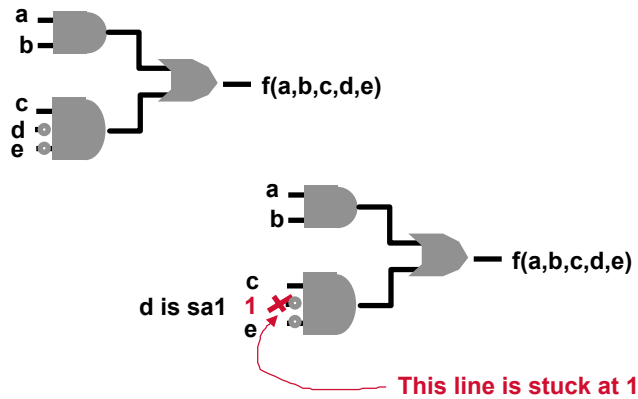
- ▶ So, you look at the manufacturing process and try to figure out what actually breaks
  - ▶ These are called **defects**
  - ▶ The effect of the defect is called a **fault**
  - ▶ How you model it in your testing procedure is the **fault model**

© R. Rutenbar 2001 Fall 18-760 Page 26

## Testing

### ▼ Most common fault model is *stuck-at model*

- ▶ Assume individual wires are stuck at logic 1 (**sa1**) or logic 0 (**sa0**)

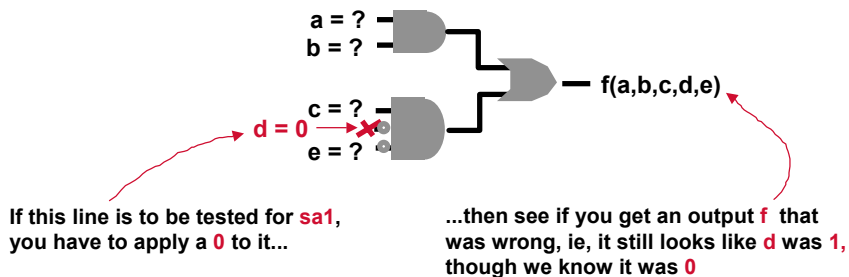


© R. Rutenbar 2001 Fall 18-760 Page 27

## Testing

### ▼ So, what's a *test* here?

- ▶ A pattern of inputs that makes an output that is wrong, ie, different what it *should* be if the fault was not present
- ▶ Usually try to generate tests to detect specific faults
- ▶ If you have a big list of possible faults, you generate a **test set** and ask what fraction of all the faults will get detected, called the **fault coverage**



© R. Rutenbar 2001 Fall 18-760 Page 28

## Testing

### ▼ OK, so how do you do it?

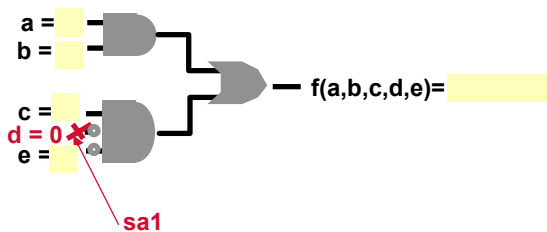
- ▶ For this simplified problem, easy
  - ▶ What do you want...?
    - ▶ An input pattern **abcde**...
- 
- ▶ ...which means you can see if the line is stuck
  - ▶ Note: to test **sa1** you apply **ab0de**, for **sa0** you apply **ablde**
- 
- ▶ We already know how to write this more clearly
    - ▶ Want an input pattern where  $F(a,b,c,d=0,e) \neq F(a,b,c,d=1,e)$
  - ▶ Same as asking for
- 
- ▶ Really want a pattern that makes cofactors wrt d *different*

© R. Rutenbar 2001 Fall 18-760 Page 29

## Testing

### ▼ For our example...

$$f(a,b,c,d,e) = ab + cd'e'$$
$$\partial f / \partial d =$$



© R. Rutenbar 2001 Fall 18-760 Page 30

## Back to Combinations of Cofactors

### Other combinations of cofactors also important

- ▶  $F_x \cdot F_{x'} = ?$
- ▶  $F_x + F_{x'} = ?$
- ▶ Look at example to get some insight

$$f(a,b,c) = ab + bc + ac$$

$$f_a =$$

$$f_a \cdot f_{a'} =$$

$$f_{a'} =$$

$$f_a + f_{a'} =$$

a b c	f(a,b,c)	f <sub>a</sub>	f <sub>a'</sub>	f <sub>a</sub> · f <sub>a'</sub>	f <sub>a</sub> + f <sub>a'</sub>
0 0 0	0	0	0		
0 0 1	0	1	0		
0 1 0	0	1	0		
0 1 1	1	1	1		
1 0 0	0	0	0		
1 0 1	1	1	0		
1 1 0	1	1	0		
1 1 1	1	1	1		

© R. Rutenbar 2001 Fall 18-760 Page 31

## Combinations of Cofactors

### Observe

- ▶  $f_a \cdot f_{a'} = 1$  just in places where  $f$  would be 1 *independent of value of a*
- ▶  $f_a + f_{a'} = 1$  if either  $f(a=0,b,c)=1$  or  $f(a=1,b,c)=1$

a b c	f(a,b,c)	f <sub>a</sub>	f <sub>a'</sub>	f <sub>a</sub> · f <sub>a'</sub>	f <sub>a</sub> + f <sub>a'</sub>
0 0 0	0	0	0		
0 0 1	0	1	0		
0 1 0	0	1	0		
0 1 1	1	1	1		
1 0 0	0	0	0		
1 0 1	1	1	0		
1 1 0	1	1	0		
1 1 1	1	1	1		

### But...this idea is hard to see in a truth table

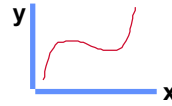
© R. Rutenbar 2001 Fall 18-760 Page 32

## Understanding Combinations of Cofactors

▼ Alternative perspective that offers some more insight

▼ Recall *Boolean cubes*

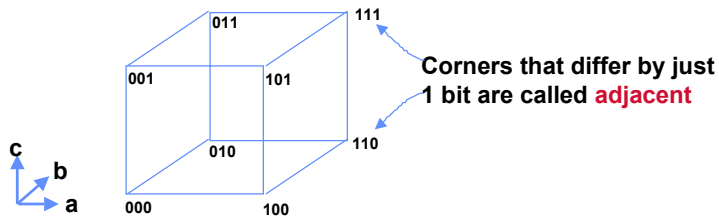
▶ You can plot a real function  $y=f(x)$



▶ You can also “plot” a Boolean function  $f(a,b,c)$

▶ Of course, each axis only goes from 0 to 1

▶ And, the only points are at the corners of this cube

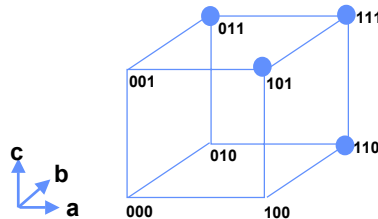


© R. Rutenbar 2001 Fall 18-760 Page 33

## Boolean Cubes

▼ Represent a function by corners where  $f=1$

▶  $f(a,b,c) = ab + bc + ac$



▶ Note that **product terms** appear on cube as sets of  $2^k$  **adjacent** corners

▶ product = 0 literals, eg “1” => all 8 corners

▶ product = 1 literal eg  $a$  => 4 corners

▶ product = 2 literals eg  $ab'$  => 2 corners

▶ product = 3 literals eg  $ab'c'$  => 1 corner, ie, a minterm

© R. Rutenbar 2001 Fall 18-760 Page 34

## Aside: Literals and other Terms

▼ Consider this function:

$$ab'c + a'bc + acd + e + af'$$

► How many **literals** are there?

► How many **product terms**?

► Is this **SOP** or **POS**?

© R. Rutenbar 2001 Fall 18-760 Page 35

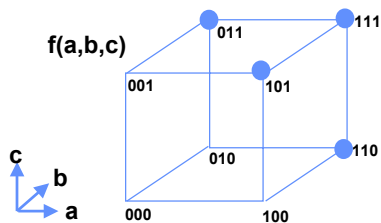
## Plotting the Cofactors

▼ Look at cubes here...

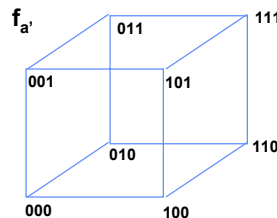
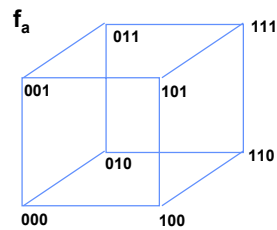
►  $f = ab + bc + ac$

►  $f_a = b+c$

►  $f_{a'} = bc$



Remember that  
this is 'a' axis

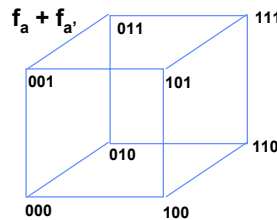
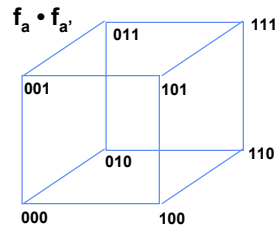
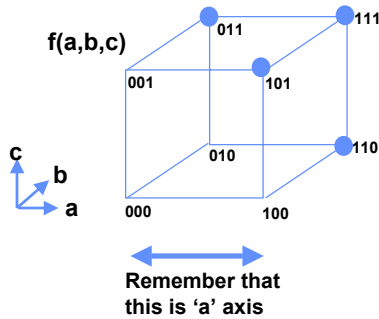


© R. Rutenbar 2001 Fall 18-760 Page 36

## Plotting Combinations of Cofactors

▼ Look at cubes here...

- ▶  $f = ab + bc + ac$
- ▶  $f_a \cdot f_{a'} = bc$
- ▶  $f_a + f_{a'} = b+c$

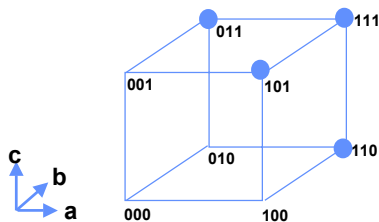


© R. Rutenbar 2001 Fall 18-760 Page 37

## Interpreting Combinations of Cofactors

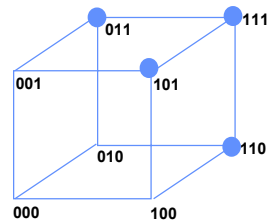
▼ Nice geometric interpretation

$f_a \cdot f_{a'}$  is called the **consensus** of  $f$  wrt  $a$ ,  $C_a(f)(b,c)$



Interpretation: keep corners where  $f=1$  independent of "a"

$f_a + f_{a'}$  is called the **smoothing** of  $f$  wrt  $a$ ,  $S_a(f)(b,c)$



Interpretation: add corners where adjacent "a" corner = 1

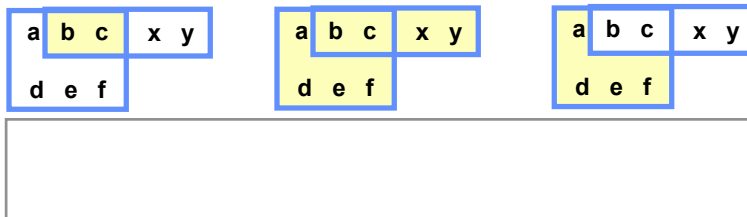
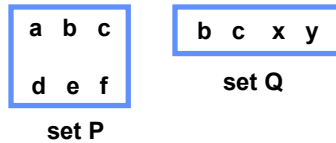
© R. Rutenbar 2001 Fall 18-760 Page 38

## Containment Props for Combinations of Cofac

▼ One more perspective on  $C_x(f)$ ,  $S_x(f)$ , and  $\partial f / \partial x$

*Containment properties*

► Remember basic set theory...?



© R. Rutenbar 2001 Fall 18-760 Page 39

## Containment Properties

▼ Think about Boolean functions as *sets of minterms*

► Not entirely unfamiliar, you should have seen somewhere notation like:

$$f(a,b,c) = \sum m(0,3,5,7) = m_0 + m_3 + m_5 + m_7 \\ = a'b'c' + a'bc + ab'c + abc$$

► Now, we just explicitly reformulate this as

▼ Who cares?

► We can now ask **containment** questions about functions...

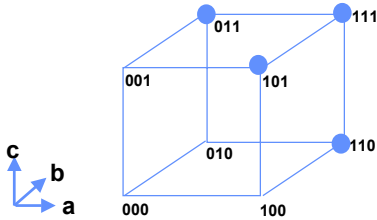
► Like: is  $f$  "bigger than"  $g$ , ie,  $f \supseteq g$

© R. Rutenbar 2001 Fall 18-760 Page 40

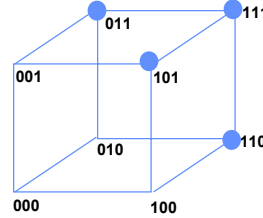
## Containment Properties

▼ Interesting containment property:  $C_a(f) \subseteq f \subseteq S_a(f)$

Consensus  $C_a(f)$  is



Smoothing  $S_a(f)$  is



Actually: consensus is function independent of var 'a' that's still 'inside' f

Actually: smoothing is function independent of var 'a' that contains f

© R. Rutenbar 2001 Fall 18-760 Page 41

## Containment Properties

▼ How would you *prove* something like this...?

Consensus is **biggest** function independent of var 'a' that's still 'inside' f

© R. Rutenbar 2001 Fall 18-760 Page 42

## Containment Properties

▼ How would you *prove* something like this...? cont.



© R. Rutenbar 2001 Fall 18-760 Page 43

## Consensus and Smoothing

▼ Additional properties

- ▶ Like Boolean difference, can do with respect to more than 1 var
- ▶ Example:  $C_{xy}(f) = C_y(C_x(f)) = f_{xy'} \cdot f_{x'y'} \cdot f_{xy} \cdot f_{x'y}$
- ▶ Example:  $S_{xy}(f) = S_y(S_x(f)) = f_{xy'} + f_{x'y'} + f_{xy} + f_{x'y}$

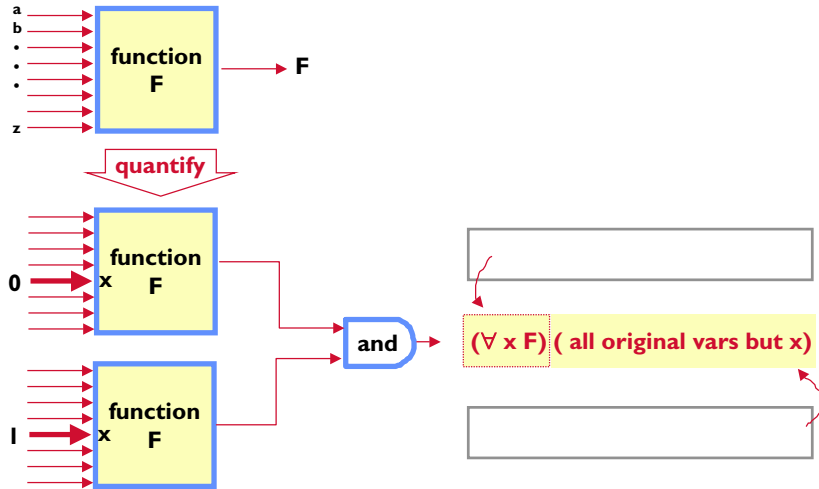
▼ Alternative names: Quantification

- ▶ In logic (predicate calculus over truth values) when you have a formula and want to get rid of a variable, the term is “quantification”
- ▶ Two kinds of quantifiers
  - ▶ “For all x”  $\forall x$  called *universal* quantification
  - ▶ “There exists x”  $\exists x$  called *existential* quantification
- ▶ Back to cofactors...
  - ▶ Consensus  $C_x(f)$  is also written  $(\forall xf)$ , called **universal quantification** of function f wrt variable x.
  - ▶ Smoothing  $S_x(f)$  is also written  $(\exists xf)$ , called **existential quantification** of function f wrt variable x
  - ▶ Both of these things --  $(\forall xf)$ ,  $(\exists xf)$  -- are new functions

© R. Rutenbar 2001 Fall 18-760 Page 44

## Quantification...?

▼ What does this really mean? Look at *Universal* quantify



© R. Rutenbar 2001 Fall 18-760 Page 45

## Quantification

▼ Why "quantification"...?

► Quantification is about "abstracting away" variables

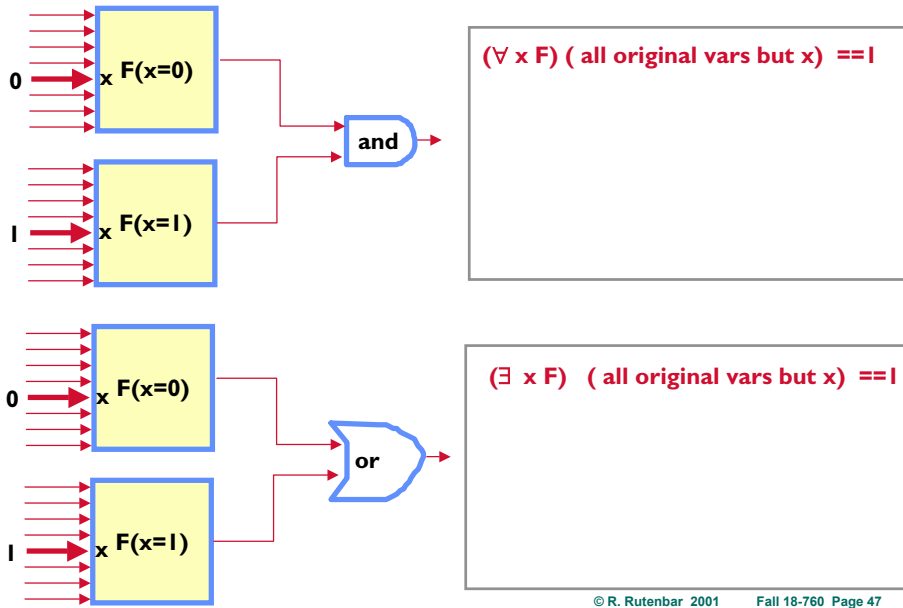
▼ Ponder the names a little...

►  $(\forall x F)$  (vars except  $x$ )

►  $(\exists x F)$  (vars except  $x$ )

© R. Rutenbar 2001 Fall 18-760 Page 46

## Quantification



## Quantification

### Remember!

- ▶  $C_x(f)$ ,  $S_x(f)$ , and  $\partial f / \partial x$  are all functions...
- ▶ ..but they are functions of all the vars in support of  $f$  except  $x$
- ▶ There are no 'x' vars anywhere in expressions for  $C_x(f)$ ,  $S_x(f)$ ,  $\partial f / \partial x$
- ▶ We got rid of variable  $x$  and made 3 new functions

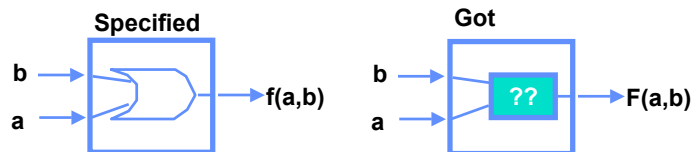
### So, are these any good for anything...?

- ▶ Sure, look at an example in logic network debugging

## Application: Network Repair

### Suppose ...

- ▶ I specified a logic block for you to implement
- ▶ ...but you implemented it wrong.
- ▶ In particular, you got **ONE** gate wrong



### Goal

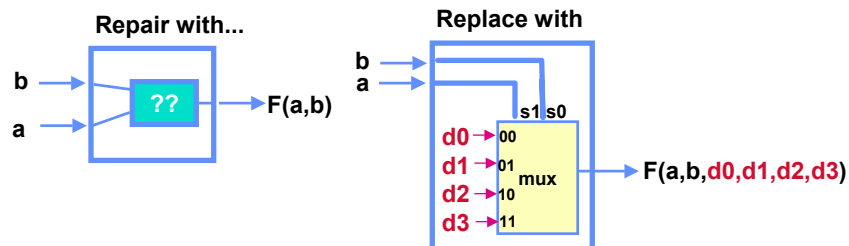
- ▶ Can we deduce how precisely to *change this gate* to restore the correct functionality?
- ▶ Let's go with this very trivial test case to see how mechanics work...

© R. Rutenbar 2001 Fall 18-760 Page 49

## Network Repair

### Clever trick

- ▶ Replace our suspect gate by a 4:1 mux with 4 arbitrary new vars
- ▶ By cleverly assigning values to  $d_0$   $d_1$   $d_2$   $d_3$ , we can *fake* any gate
- ▶ Question is: what are the right values of  $d$ 's so  $F$  is repaired ( $=f$ )

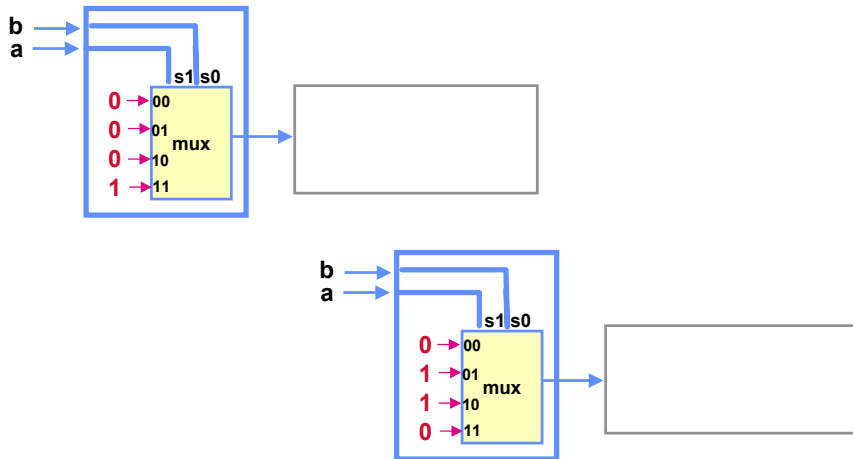


© R. Rutenbar 2001 Fall 18-760 Page 50

## Aside: Faking a Gate with a MUX

### Remember...

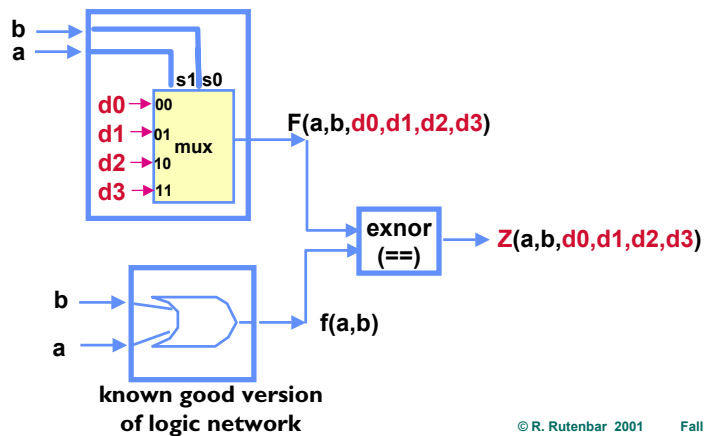
- You can do any function of 2 vars with one 4 input MUX



## Network Repair: Using Quantification

### Next trick

- Make new function  $Z(a,b,d0,d1,d2,d3)$  that = 1 just when  $F == f$



## Using Quantification

### What now?

- ▶ Think hard about exactly what we want:

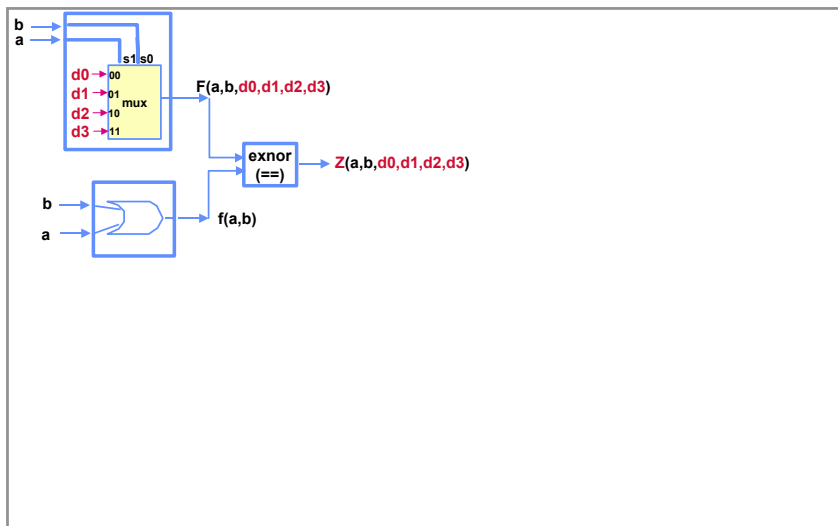
### But this is something we've seen!

- ▶ Consensus of function  $Z$  wrt variables  $a, b$ !
- ▶ This is just any pattern of  $(d0\ d1\ d2\ d3)$  that makes  $C_{ab}(Z)(d0\ d1\ d2\ d3) == 1$  (do you know where  $a, b$  went??)
- ▶ Can also write as quantification:  $(\forall_{ab} Z)(d0, d1, d2, d3)$ 
  - ▶ Note: these are both functions of just the  $d$ 's
  - ▶ We want any pattern of  $d$ 's that makes  $C_{ab}(Z) == 1$
  - ▶ This pattern is guaranteed to make the mux behave like the correct gate, independent of what's going on with  $a, b$

© R. Rutenbar 2001 Fall 18-760 Page 53

## Network Repair via Quantification: Example

### Don't believe it...? Try it... It's all mechanics



© R. Rutenbar 2001 Fall 18-760 Page 54

## Network Repair via Quantification: Example

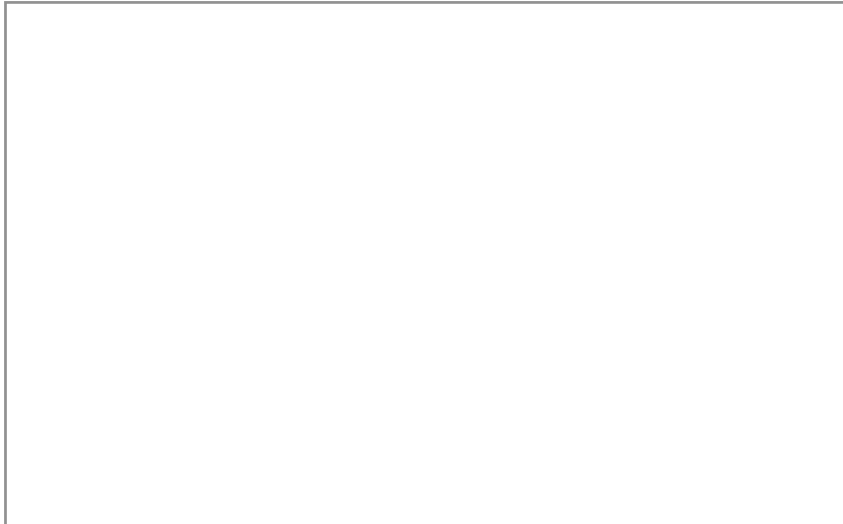
▼ ...mechanics, cont.



© R. Rutenbar 2001 Fall 18-760 Page 55

## Repair via Quantification

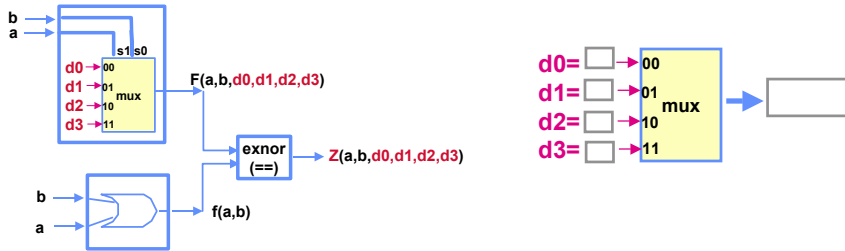
▼ Mechanical foo, cont.



© R. Rutenbar 2001 Fall 18-760 Page 56

# Network Repair

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



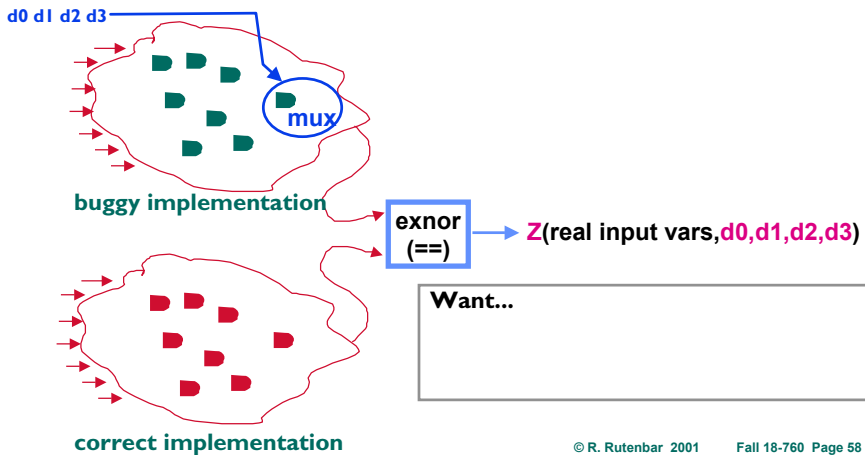
▼ This example is small...

- ▶ But in a real example, you have a *big* network-- 500 inputs, 50,000 gates
- ▶ When it doesn't work, it's a major hassle to go thru in detail
- ▶ This is a mechanical procedure that can answer this:
  - ▶ Is there a way to change this one gate to make it right?

# Network Repair

▼ Realistic Case

- ▶ Oops, it doesn't work! Let us guess there is a one-gate error someplace..but we don't know where...
- ▶ For **each gate** in network, try to do this repair procedure..



## Computational Strategies

### ▼ What haven't we seen yet? *Computational strategies*

- ▶ In several places we sort of assumed you could figure something out once you got the right function...
  - ▶ Example: find inputs to make  $\partial f / \partial x == 1$  for testing
  - ▶ Example: find inputs to make  $C_{ab}(Z) == 1$  for gate debug
  - ▶ This computation is called **satisfiability**
  - ▶ We'll see a bunch of such strategies later in course

### ▼ Common computation theme: *divide & conquer*

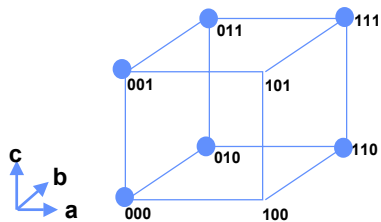
- ▶ You want to do something hard on a Boolean function...
- ▶ ...so you try to do it with the cofactors, glue answer back together
- ▶ Let's look at one simplified example to get some experience...

© R. Rutenbar 2001 Fall 18-760 Page 59

## Representation Issues

### ▼ First, let's look at a simple, historically early representation scheme for functions

- ▶ Represent a function as a set of OR'ed product terms
- ▶ Remember: each product term is a cube with  $2^k$  corners when plotted

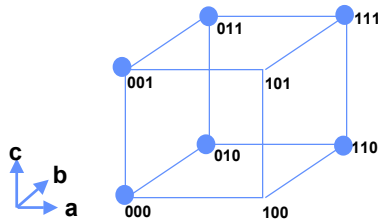


© R. Rutenbar 2001 Fall 18-760 Page 60

## Positional Cube Notation

### Remember, we say “cube” and mean “product term”

- ▶ So, how to represent each cube?
- ▶ **Positional cube notation:** one slot per variable, 2 bits per slot
- ▶ Can write down each cube very simply 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





© R. Rutenbar 2001 Fall 18-760 Page 61

## Positional Cube Notation: Tautology

### So, we represent a *cover* of a function...

- ▶ ...as a list of cubes in positional cube notation
- ▶ Ex:  $f(a,b,c) = a + bc + ac' \Rightarrow$

### Look at an application: *Tautology testing*

- ▶ We say a function  $f$  is a **tautology** when  $f == 1$  for all inputs
- ▶ Turns out to be many computational uses for this
- ▶ But you might be thinking “Hey, how hard can this be...?”
  - ▶ Actually, pretty hard for a big complex function represented in some POS form like a cube-list

Ex:  $f(a,b,c) = ab + ac + ab'c' + a'$   
is it or isn't it  $== 1$  always?

© R. Rutenbar 2001 Fall 18-760 Page 62

# Tautology Checking

## How do we approach tautology as a *computation*?

- ▶ Input = cube-list representing products in an SOP form of  $f$
- ▶ Output = yes/no,  $f == 1$  always or not

## Cofactors to the rescue

Nice result:  $f$  is a tautology if and only if  $f_x$  and  $f_{x'}$  are both tautologies

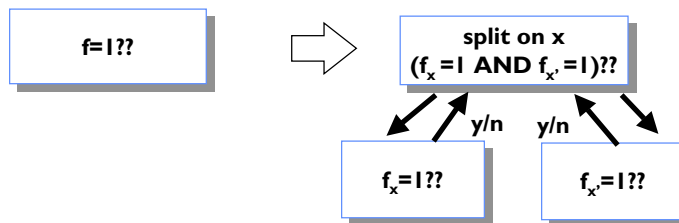
Proof again not too hard:

© R. Rutenbar 2001 Fall 18-760 Page 63

# Recursive Tautology Checking

## Suggests a recursive strategy:

- ▶ If you can't 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?

© R. Rutenbar 2001 Fall 18-760 Page 64

## Recursive Cofactoring

### ▼ Do mechanics first -- they're *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

#### ► Examples

$$f = ab+ac'd+bc'$$

$f_a$                        $f_c$

```
[01 01 11 11]
[01 11 10 01]
[11 01 10 11]
```

© R. Rutenbar 2001 Fall 18-760 Page 65

## Unate Functions

### ▼ Selection / termination, another trick: *Unate functions*

#### ► Special class of Boolean functions

#### ► $f$ is unate if a SOP representation only has each literal appearing in exactly one polarity, either all true, or all complemented



#### ► $f$ is **positive unate** in variable $x$ if changing $x$ 0->1 keeps $f$ constant or makes it change 0->1

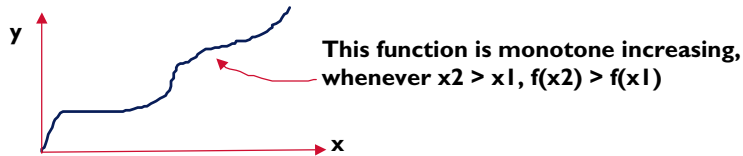
#### ► $f$ is **negative unate** in variable $x$ if changing $x$ 0->1 keeps $f$ constant or makes it change 1->0

#### ► Function that's not unate is called **binate**

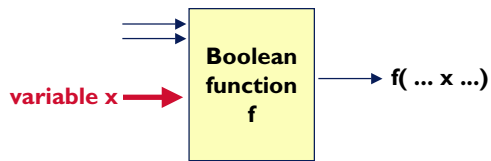
© R. Rutenbar 2001 Fall 18-760 Page 66

# Unate Functions

## ▼ Analogous to *monotone* continuous functions



## ▼ Boolean function positive unate in x



Really just same idea as above...

# Using Unate Functions For Computation

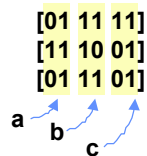
## ▼ Who cares?

- ▶ Unate schmunate--we need easy tautology checking!
- ▶ But this helps...

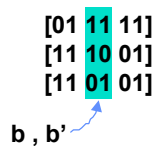
## ▼ Suppose you have a cube-list for f

- ▶ That cube-list is **unate** if each var in each cube only appears in one polarity, but 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

### Nice result

- ▶ It's pretty easy to check a unate cube-list for tautology

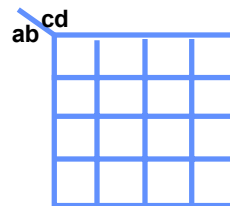


- ▶ Reminder: what exactly is  $[11 \ 11 \ 11 \ \dots \ 11]$  as a product term?

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

### This result actually makes sense...

- ▶ You can't make a "1" with only product terms where all literals are in just one polarity
- ▶ Try to do it on a Kmap...



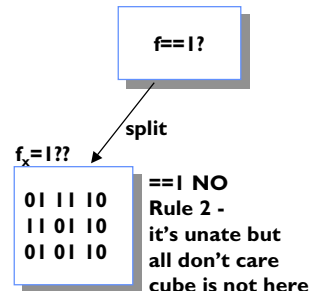
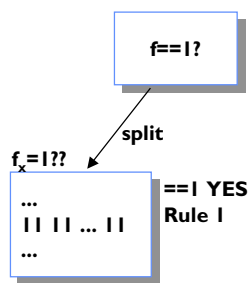
© R. Rutenbar 2001   Fall 18-760 Page 69

## Recursive Tautology Checking

### So, unateness gives us some *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  $[11 \ 11 \ \dots \ 11]$   
**Why:** function at this leaf is (stuff + 1 + stuff)  $= 1$
- ▶ **Rule 2:**  $\neq 1$  if cube-list unate and all don't care cube *missing*  
**Why:** unate  $=1$  if and only if has  $[11 \ 11 \ \dots \ 11]$  cube



© R. Rutenbar 2001   Fall 18-760 Page 70

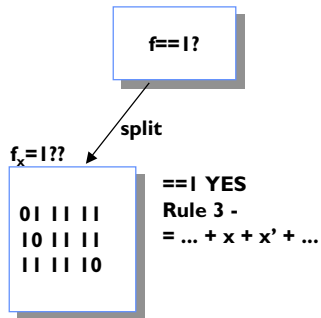
# Recursive Tautology Checking

► Lots more rules...

► **Rule 3:**  $f == 1$  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...



# Recursive Tautology Checking

▼ But can't use easy termination rules unless *unate* cubelist

▼ Selection rule...?

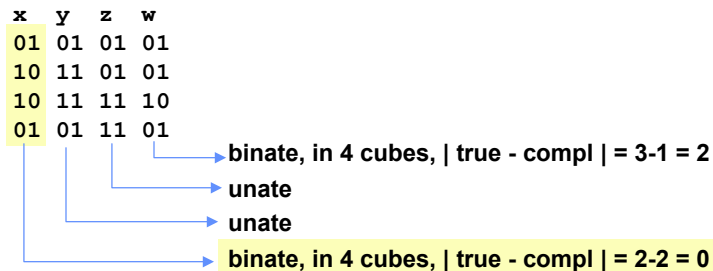
► Hey, pick the splitting var to try to make unate cofactors!

► Strategy: pick "most not-unate" (binate) var as split var

▼ Implementation

► Pick binate var with **most** product terms dependent on it

► If a tie, pick one with **min** | true var - complement var |



## Recursive Tautology Checking

▼ And that's it!

▼ 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( fx ) && tautology( fx' ) )
  }
}
    
```

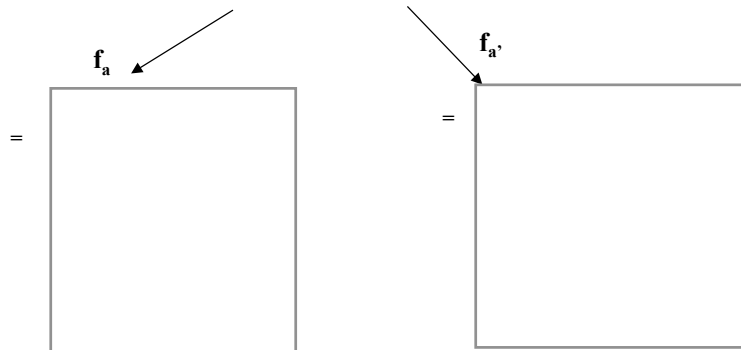
© R. Rutenbar 2001 Fall 18-760 Page 73

## Recursive Tautology Checking: Example

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

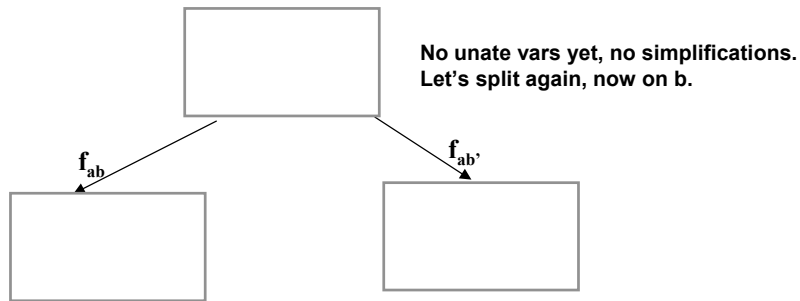
	a	b	c
ab	01	01	11
ac	01	11	01
ab'c'	01	10	10
a'	10	11	11

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



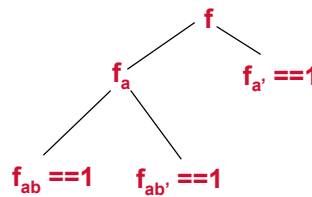
© R. Rutenbar 2001 Fall 18-760 Page 74

## Recursive Tautology Checking: Example



▼ So we are done:

- ▶ Our tree has tautologies at all the leaves!
- ▶ Note - if any leaf ends up  $\neq 1$ , then  $f \neq 1$  too, this is how tautology fails



© R. Rutenbar 2001 Fall 18-760 Page 75

## Computational Boolean Algebra

▼ Computational philosophy revisited

- ▶ This strategy is so general and useful it has a name

- ▶ **Paradigm:** a general strategy of broad application, power
- ▶ **Recursive:** use Shannon cofactoring as basis for progress
- ▶ **Unate:** strive to make cofactors unate, since unate = simpler, and lots of properties are just easier to find with unate f's

© R. Rutenbar 2001 Fall 18-760 Page 76

# Advanced Boolean Algebra

## ▼ Summary

- ▶ **Cofactors, and functions of cofactors interesting and useful**
  - ▶ **Boolean difference, consensus, smoothing (quantification)**
  - ▶ **Real applications: test, gate debugging, etc.**
- ▶ **Representation for Boolean functions will end up being critical**
  - ▶ **Truth tables, Kmaps, equations not manipulable by software**
  - ▶ **Saw one real representation: cube-list, positional cube notation**
- ▶ **Emphasis on computational strategies to answer questions about Boolean functions**
  - ▶ **Ex: is  $f=1$ ? does  $f$  cover this product term? what values of inputs makes  $f=1$ ?**
  - ▶ **Saw an example of a strategy: Unate Recursive Paradigm**