(*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

Copyright Notice

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 $1, x, x^2, x^3, x^4, \ldots$ as the pieces…
    - …turns out $\exp(x) = 1 + x + x^2/2! + x^3/3! + \ldots$
  - 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 polyomials
- 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 x1, x2, …, xn
  - Let B = {0,1} then we write formally:
  - We often refer to the variables x1,x2,..xn by lumping them together in a set {x1, x2, …, xn} called the **support** of F, or **sup(F)**.

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) =

F(x,y=0,z) =

- Important to remember that result is a *new function*
  - Note that new function no longer depends on this variable

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
    - Write F(x1, x2, ..., xi=0, ...xn) as
    - Often see as just which is easier to type

  - *Restriction of F on variable xi*
    - Write F(x1, x2, ..., xi=1, ...xn) as
    - Write F(x1, x2, ..., xi=0, ...xn) as

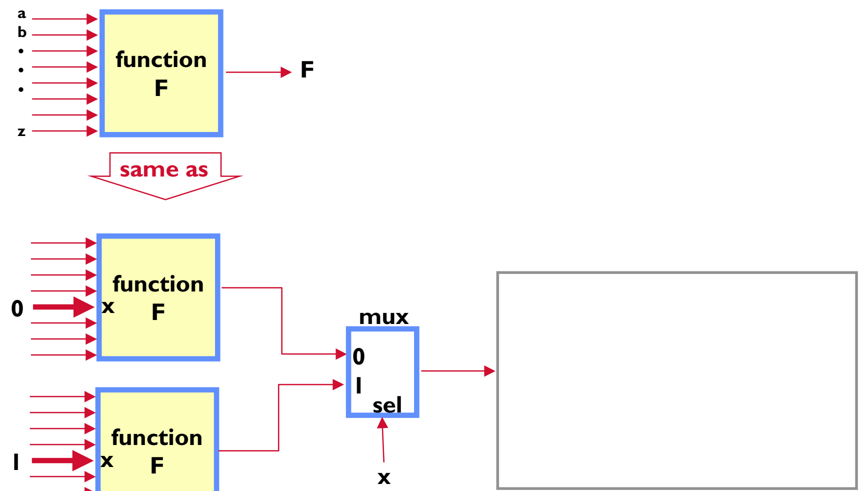
- Why are these useful functions to get from F?

Shannon Expansion Theorem

- Shannon Expansion Theorem
  - Given any Boolean function F(x1, x2, ..., xn) and any xi in the support of F( ), F( ) can be represented as

  - 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

$F(x,y,z,w) = x \bullet \boxed{F(x=1)} + x' \bullet \boxed{F(x=0)}$ 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's 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 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) = a Boolean *function* of z and w**

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?**
  - **For example,**

$(F \bullet G)_x$ **= what?** $(F\,')_x$ **= what?** **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

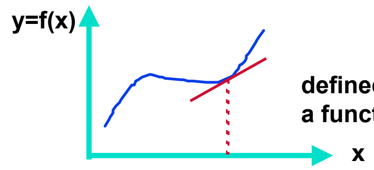
    - *In fact, true for ANY binary operator on Boolean functions*
    - Very useful: can often help in getting cofactors of complex formulas

Combinations of Cofactors

- OK, now consider operations on cofactors *themselves*
- Suppose we have F(X), and get F_x and $F_{x'}$
  - $F_x \oplus F_{x'}$ = ?
  - $F_x \bullet F_{x'}$ = ?
  - $F_x + F_{x'}$ = ?
- Turns out these are all useful *new* functions
  - We'll start with most obvious one to get…
  - Need to remember some more calculus…

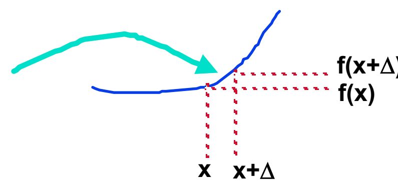
Derivatives

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



defined as slope of curve as a function of point x

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



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

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 Δ
  - Can only change 0 <-> 1, but can still ask how f changes with x ...

Real-valued f(x):

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

Binary-valued f(x):

df/dx = $\boxed{}$

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
    $\partial f / \partial x \, \partial y =$
  - Derivative of exor is exor of derivatives
    $\partial(f \oplus g) / \partial x =$
  - If function f is actually constant (f=1 or f=0, always, for all inputs)
    $\partial f / \partial x =$
  - If function f doesn't *depend on* var x (ie change x, f never changes)
    $\partial f / \partial x =$
    $\partial(f \bullet g) / \partial x =$
    $\partial(f + g) / \partial x =$

Boolean Difference

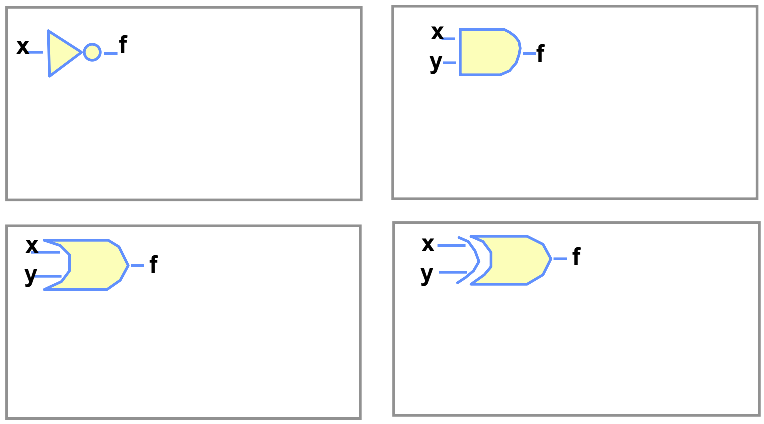
- continued...
  - If f is a function only of x (only changing x can change f)
    - $\partial f / \partial x =$
- But some things are just *more* complex, though...
  - Derivatives of (f •g) and (f + g) *don't* work the same...

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

    $\partial(f + g) / \partial x =$
  - Why?
    - Because AND and OR on Boolean values don't always behave like ADDITION and MULTIPLICATION on real numbers

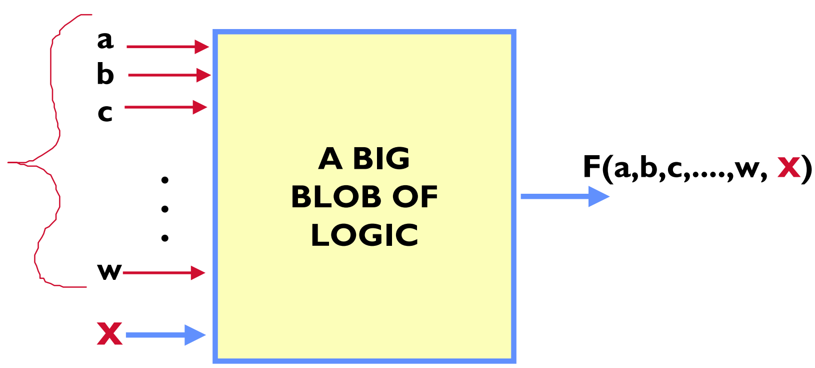
Boolean Difference: Gate-level View

- Try the obvious “simple” examples for ∂f / ∂x



Interpretation: when ∂f / ∂x = 1, then f changes as x changes

Interpreting the Boolean Difference



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

Boolean Difference: Example

- Try another example

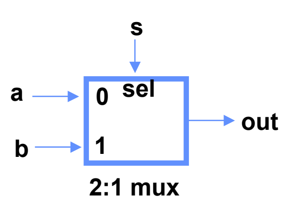
s = a ⊕ b ⊕ cin
cout = ab + (a + b)cin

∂s / ∂a =

∂cout / ∂cin =

Boolean Difference: Example

- Example



out = as ’ + bs

∂out / ∂s =

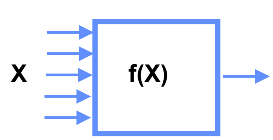
∂out / ∂a =

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**

Application: Testing

- Suppose you want to test some logic
  - **You want to figure out what inputs to apply to figure out if it works**



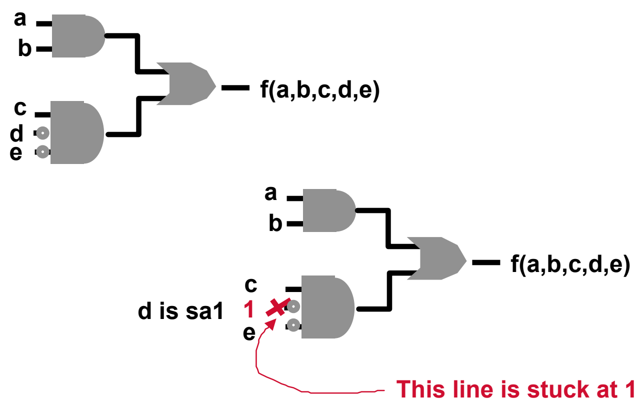
You can't afford to apply all possible inputs

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**

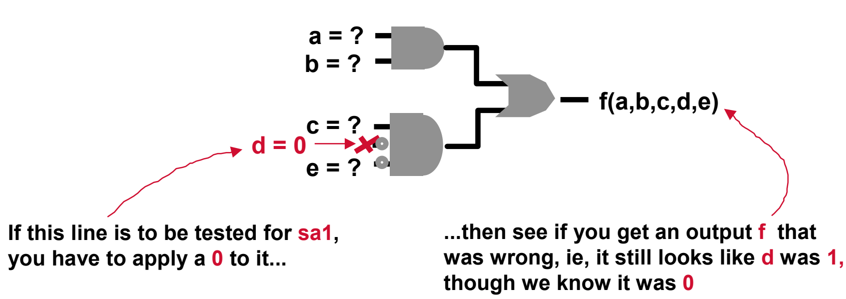
Testing

- **Most common fault model is *stuck-at model***
  - **Assume individual wires are stuck at logic 1 (sa1) or logic 0 (sa0)**



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**



Testing

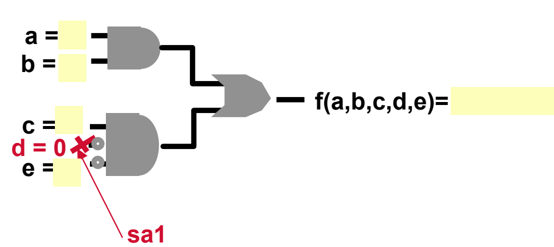
- **OK, so how do you do it?**
  - For this simplified problem, easy
  - What do you want...?
    - An input pattern abcde...
    - ...which means you can see if the line is stuck
    - Note: to test sa1 you apply ab0de, for sa0 you apply ab1de
  - We already know how to write this more clearly
    - Want an input pattern where F(a,b,c,d=0,e) != F(a,b,c,d=1,e)
    - Same as asking for
    - Really want a pattern that makes cofactors wrt d *different*

Testing

- **For our example...**

f(a,b,c,d,e) = ab + cd'e'

$\partial f / \partial d =$



Back to Combinations of Cofactors

- *Other* combinations of cofactors also important
  - $F_x \bullet F_{x'} = ?$
  - $F_x + F_{x'} = ?$
  - Look at example to get some insight

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

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

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

a b c	f(a,b,c)	f_a	$f_{a'}$	$f_a \bullet f_{a'}$	$f_a + f_{a'}$
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		

Combinations of Cofactors

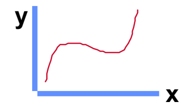
- Observe
  - $f_a \bullet f_{a'} = 1$ just in places where f would be 1 *independent* of value of a
  - $f_a + f_{a'} = 1$ if *either* f(a=0,b,c)=1 or f(a=1,b,c)=1

a b c	f(a,b,c)	f_a	$f_{a'}$	$f_a \bullet f_{a'}$	$f_a + f_{a'}$
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		

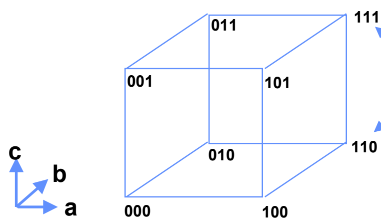
- But...this idea is hard to *see* in a truth table

Understanding Combinations of Cofactors

- Alternative perspective that offers some more insight
- Recall *Boolean cubes*
  - You can plot a real function y=f(x)



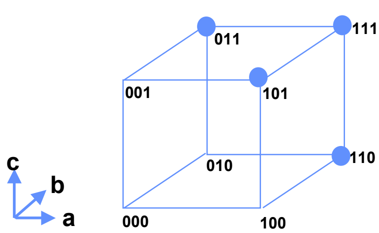
  - You can also "plot" a Boolean function f(a,b,c)
    - Of course, each axis only goes from 0 to 1
    - And, the only points are at the corners of this cube



Corners that differ by just 1 bit are called adjacent

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

Aside: Literals and other Terms

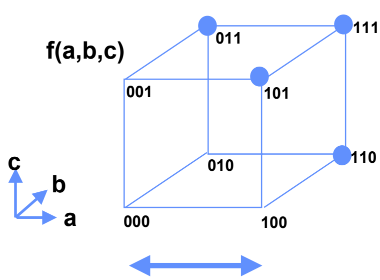
- **Consider this function:**

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

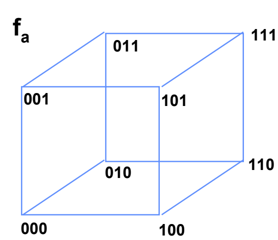
  - **How many literals are there?**
  - **How many product terms?**
  - **Is this SOP or POS?**

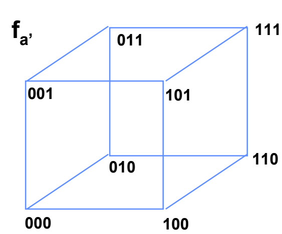
Plotting the Cofactors

- **Look at cubes here...**
  - **f = ab + bc + ac**
  - **f_a = b+c**
  - **$f_{a'}$ = bc**



Remember that this is 'a' axis

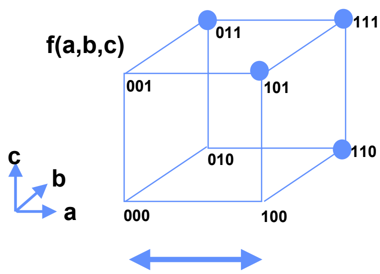




Plotting Combinations of Cofactors

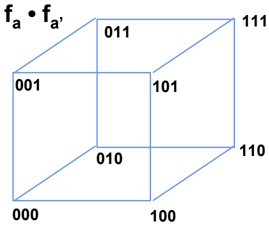
- Look at cubes here…
  - $f = ab + bc + ac$
  - $f_a \bullet f_{a'} = bc$
  - $f_a + f_{a'} = b{+}c$

f(a,b,c)

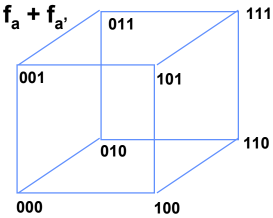


Remember that this is ‘a’ axis

$f_a \bullet f_{a'}$



$f_a + f_{a'}$

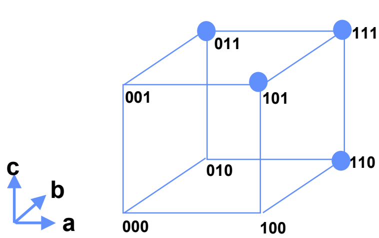


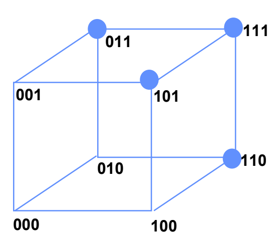
Interpreting Combinations of Cofactors

- Nice geometric interpretation

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

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



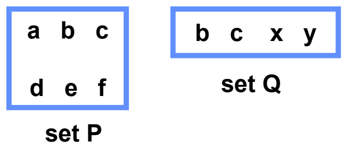


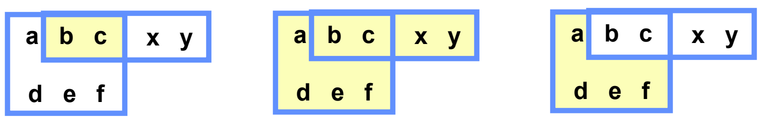
Interpretation: keep corners where f=1 independent of “a”

Interpretation: add corners where adjacent “a” corner = 1

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...?





Containment Properties

- Think about Boolean functions as *sets of minterms*
  - Not entirely unfamiliar, you should have seen somewhere notation like:

$$f(a,b,c) = \Sigma\, m(0,3,5,7) = m0 + m3 + m5 + m7$$
$$= a'b'c' + a'bc + ab'c + abc$$

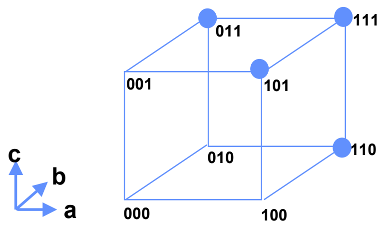
  - Now, we just explicitly reformulate this as

- Who cares?
  - We can now ask containment questions about functions...
  - Like: is f "bigger than" g, ie, $f \supseteq g$

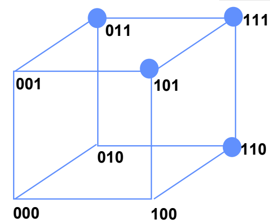
Containment Properties

- 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

Containment Properties

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

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

Containment Properties

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

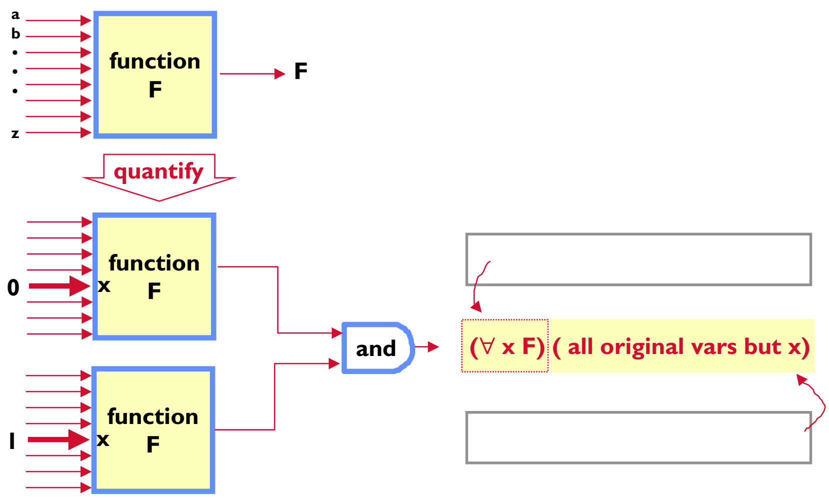
Consensus and Smoothing

- Additional properties
  - Like Boolean difference, can do with respect to more than 1 var
  - Example: $C_{xy}(f) = C_y(\ C_x(f)\ ) = f_{xy'} \bullet f_{x'y'} \bullet f_{xy} \bullet f_{x'y}$
  - 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

Quantification...?

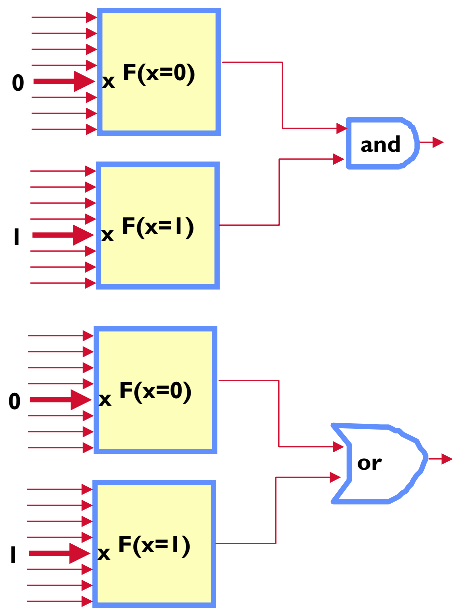
- What does this really mean? Look at *Universal* quantify



Quantification

- Why "quantification"...?
  - Quantification is about "abstracting away" variables
- Ponder the names a little...
  - (∀ x F)(vars except x)
  - (∃ x F) (vars except x)

Quantification



($\forall$ x F) (all original vars but x) ==1

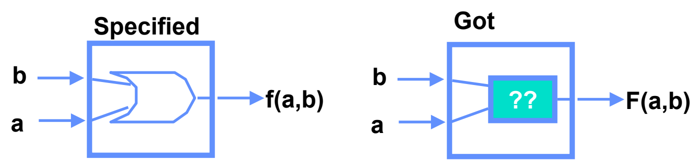
($\exists$ x F) (all original vars but x) ==1

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

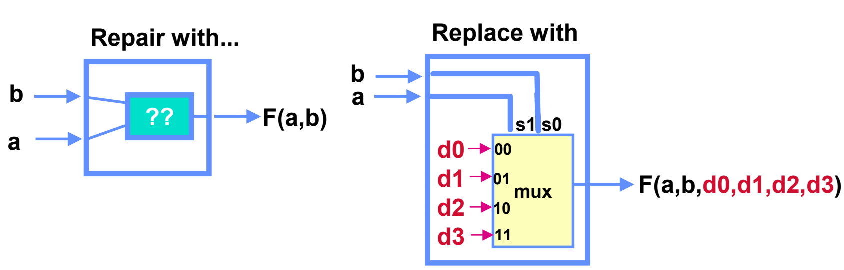
- **Suppose ...**
  - I specified a logic block for you to implement
  - ...but you implemented it wrong.
  - In particular, you got ONE gate wrong



- **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...

Network Repair

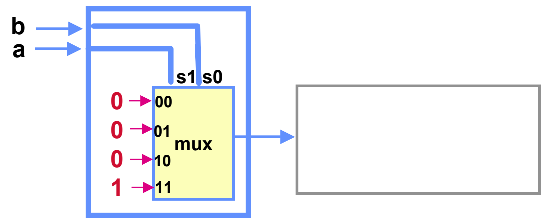
- **Clever trick**
  - Replace our suspect gate by a 4:1 mux with 4 arbitrary new vars
  - By cleverly assigning values to d0 d1 d2 d3, we can *fake* any gate
  - Question is: what are the right values of d's so F is repaired (==f)

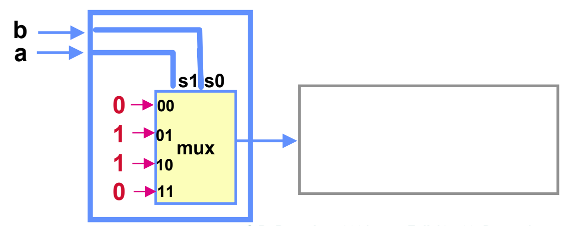


Aside: Faking a Gate with a MUX

Remember...

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

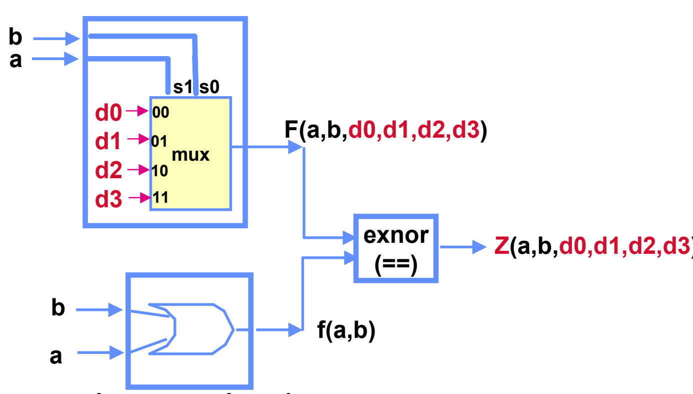




Network Repair: Using Quantification

Next trick

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



known good version
of logic network

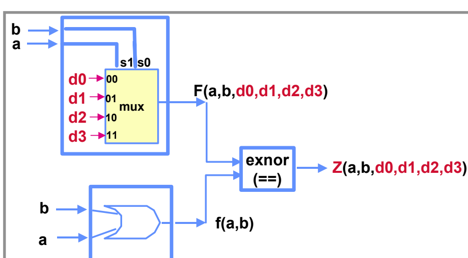
Using Quantification

- 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

Network Repair via Quantification: Example

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



Network Repair via Quantification: Example

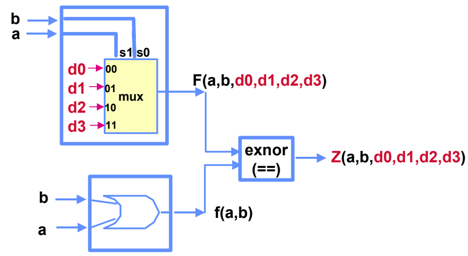
- ...mechanics, cont.

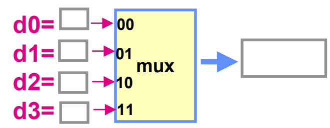
Repair via Quantification

- Mechanical foo, cont.

Network Repair

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

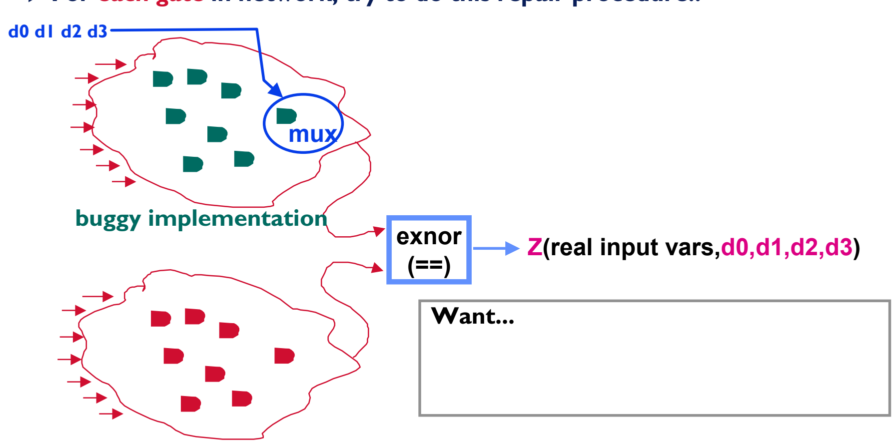




- **This example is small...**
  - But in a real example, you have a *big* network-- 500 inputs, 50,000 gates
  - When it doesn't work, it's a major hassle to go thru in detail
  - This is a mechanical procedure that can answer this:
    - Is there a way to change this one gate to make it right?

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..



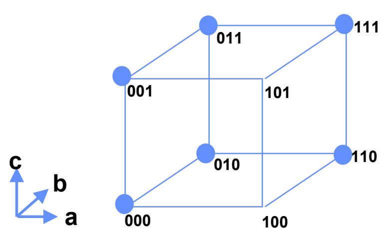
Computational Strategies

- 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...

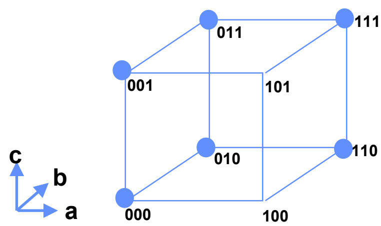
Representation Issues

- 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



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



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' =>
- 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?

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

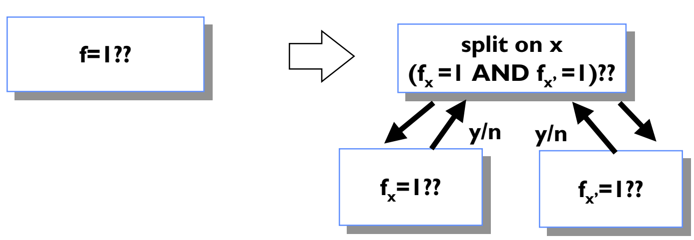
- **Cofactors to the rescue**

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

  Proof again not too hard:

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?

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

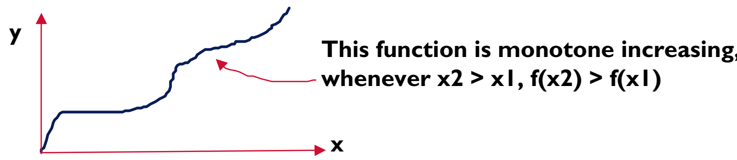
f = ab+ac'd+bc'	f_a	f_c
[01 01 11 11] [01 11 10 01] [11 01 10 11]		

Unate Functions

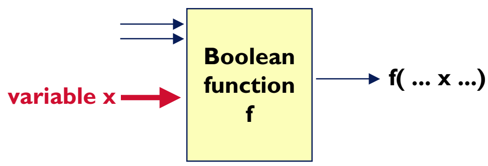
- Selection / termination, another trick: *Unate functions*
  - Special class of Boolean functions
  - f is unate if a SOP representation only has each literal appearing in exactly one polarity, either all true, or all complemented
  - f is **positive unate** in variable x if changing x 0-->1 keeps f 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**

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 => [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

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

a b c

a+b'c+bc NOT

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

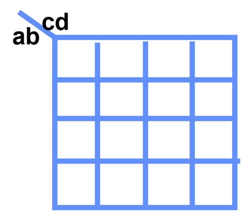
b , b'

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 ... 11] as a product term?

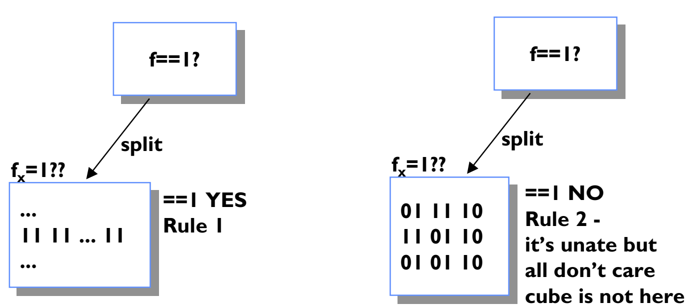
    [01 01 01] = abc   [01 01 11] = ab   [01 11 11] = a   [11 11 11] =

- 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...



Recursive Tautology Checking

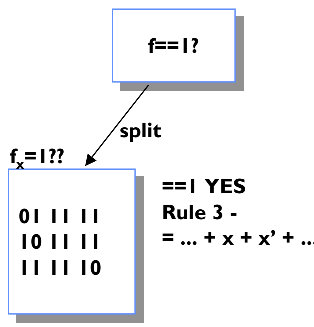
- So, unateness gives us some *termination* rules
  - We can look for tautology directly, if we have a unate cube-list
  - If match rule, know immediately if ==1, or not
    - **Rule 1:** ==1 if cube-list has all don't care cube [11 11 ... 11]
      Why: function at this leaf is (stuff + 1 + stuff) == 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



Recursive Tautology Checking

- Lots more 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...



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 |

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

- w: binate, in 4 cubes, | true - compl | = 3-1 = 2
- z: unate
- y: unate
- x: binate, in 4 cubes, | true - compl | = 2-2 = 0

Recursive Tautology Checking

- And that's it!
- Algorithm

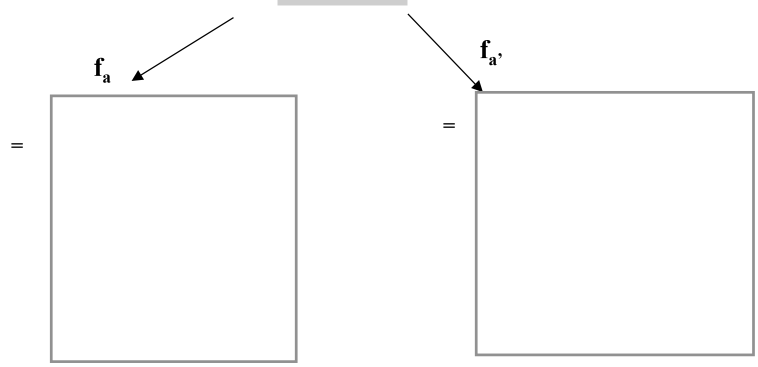
```
tautology( f represented as cubelist) {
   /* check if we can terminate recursion */
   if (f is unate)  {
      apply unate tautology termination rules directly
      if (==1) return (1)
      else return (0)
   }
   else if (any other termination rules, like rule 3, work) {
       return the appropriate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


Recursive Tautology Checking: Example

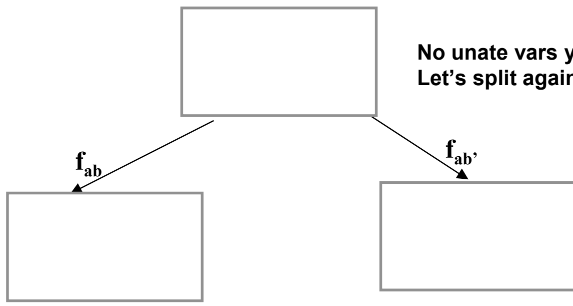
- Tautology example: f = ab + ac + ab'c' + a'

	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...

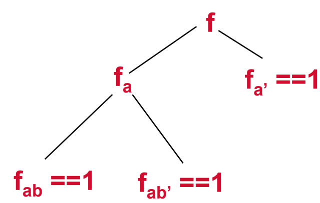


Recursive Tautology Checking: Example



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

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



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

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