CSE 620: Selective Topics
Introduction to Formal Verification

Master Studies in CSE
Fall 2016
Lecture #2

Dr. Hazem Ibrahim Shehata
Assistant Professor
Dept. of Computer & Systems Engineering

Course Outline

- **Computational Boolean Algebra**
  - Basics
    - Shannon Expansion
    - Boolean Difference
    - Quantification Operators
      - + Application to Logic Network Repair
  - Validity Checking (Tautology Checking)
  - Satisfiability Checking (SAT solving)
  - Binary Decision Diagrams (BDD's)

- **Model Checking**
  - Temporal Logics ➔ LTL - CTL
  - SMV: Symbolic Model Verifier
  - Model Checking Algorithms ➔ Explicit CTL

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 $\dfrac{\partial f}{\partial x} \oplus$

- **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)
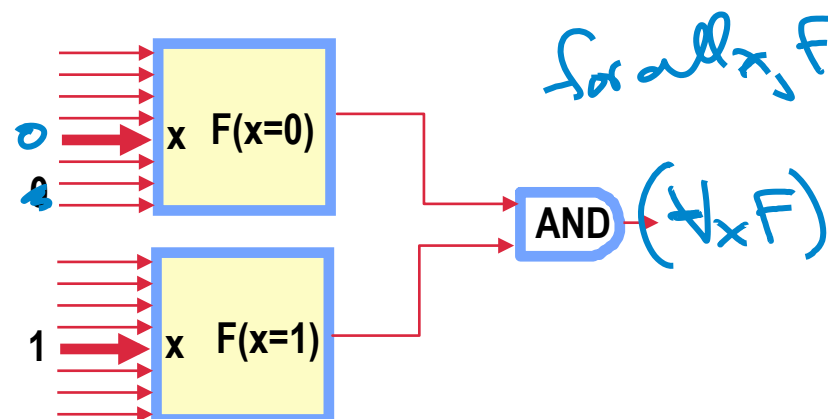
AND: Fx • Fx' is *Universal Quantification*

- **Have** $F(x_1, x_2, \ldots, x_i, \ldots x_n)$

- **AND** cofactors: $F_{x_i} \cdot F_{x_i'}$

    - Name: **Universal Quantification** of function **F** with respect to (wrt) variable x_i

    $(\forall x_i \ F) \ [x_1, x_2, \ldots, x_i\text{-}1, x_i\text{+}1, \ldots x_n]$

- **"$(\forall x_i \ F)$" 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…

for all x, F

$(\forall_x F)$

OR: Fx + Fx' is *Existential Quantification*

- **Have F(x1, x2, …, xi, … xn)**

- **OR the cofactors: $F_{xi} + F_{xi}'$**

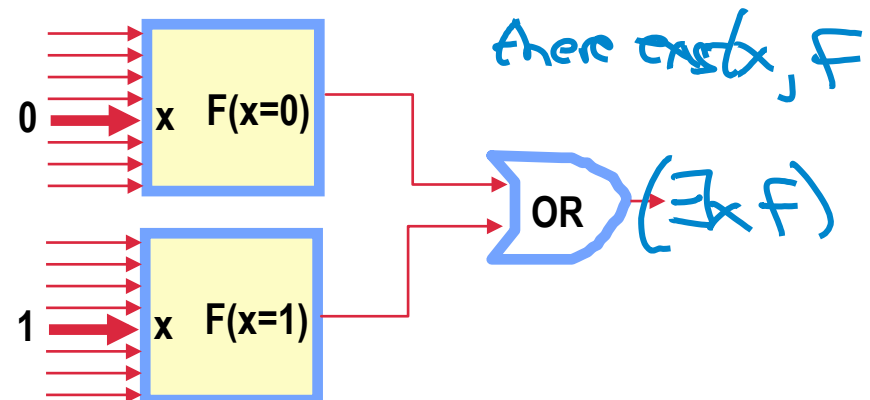 - Name: **Existential Quantification** of function F wrt variable xi

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

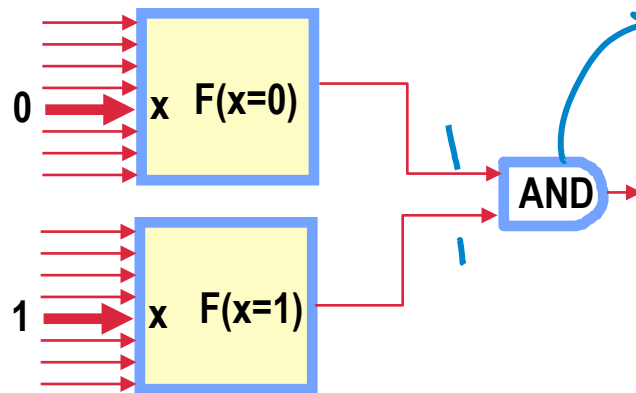- **"$(\exists xi \; F)$" is a new function**

  - "$\exists$" sign is "there exists" from logic; and function also does not depend on **xi**

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

So: $(\forall xi \; F)$ and $(\exists xi \; F)$ both omit **xi**

0 → x F(x=0)
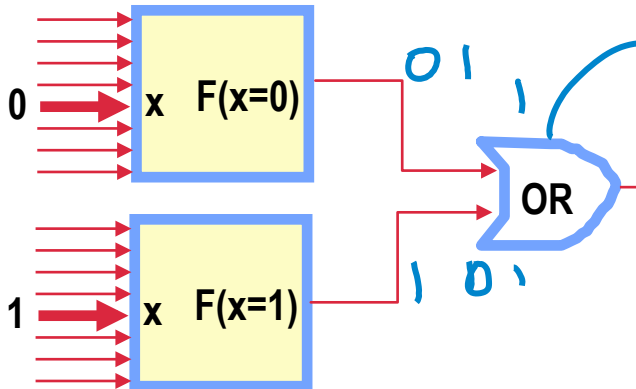
1 → x F(x=1)

OR

there exists$_x$ F

$(\exists_x F)$

Quantification Notation Makes Sense...

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

→ these other inputs make F=1

For All values of x

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

→ There exists a value of X that makes F=1 for this input particular other vars

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: $(\forall xy \ F)[z,w] = (\forall x \ (\forall y \ F)) = Fxy \cdot Fx'y \cdot Fxy' \cdot Fx'y'$ AND

  - Example: $(\exists xy \ F)[z,w] = (\exists x \ (\exists y \ F)) = Fxy + Fx'y + Fxy' + Fx'y'$ 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 A1A0**

  - It's just a 2-bit adder, but instead of **B1B0** for the second operand, it is just **0X**

  - It produces a carry out called **C** and also has a carry in called **D**

    A1 0   **A0 X**

    out   **C** ← | **a1 b1**   **a0 b0** | ← **D** carry in
                      adder
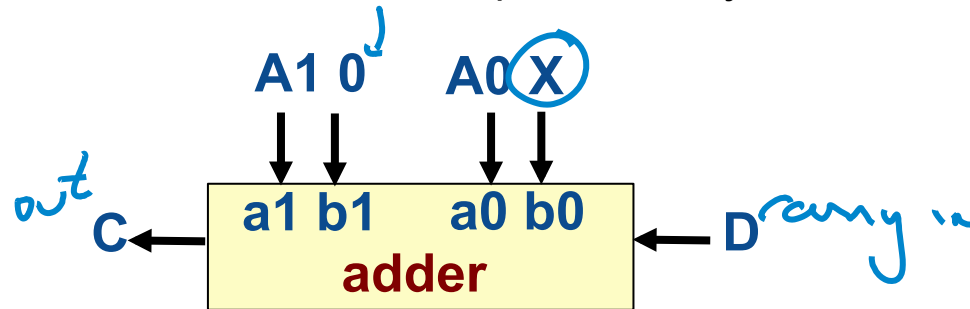
  - What is $(\forall A1, A0 \ C)[X,D]...$?

    - A function of only **X,D**. Makes a **1** for values of **X,D** that make carry **C=1** for *all values* of operand input **A1A0**, i.e., makes a carry **C=1** for all values of **A1A0**

  - What is $(\exists A1, A0 \ C)[X,D]...$?

    interpreted

    - A function of just **X,D**. Makes a **1** for values of **X,D** that make carry **C=1**, for *some value* of **A1A0**, i.e., *there exists* some **A1A0** that, for this **X,D** makes **C=1**

Quantification Example

A1 0   A0 X

↓ ↓   ↓ ↓

| a1 b1   a0 b0 | ← D |
| adder | |

C ←
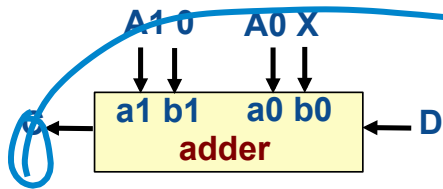
Do the math. Need all 4 cofactors:

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

C_{A1A0} $C_{A1'A0}$ $C_{A1A0'}$ $C_{A1'A0'}$

X+D   0   XD   0

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

  0

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

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

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

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

VLSI CAD:
Logic to Layout

Rob A. Rutenbar
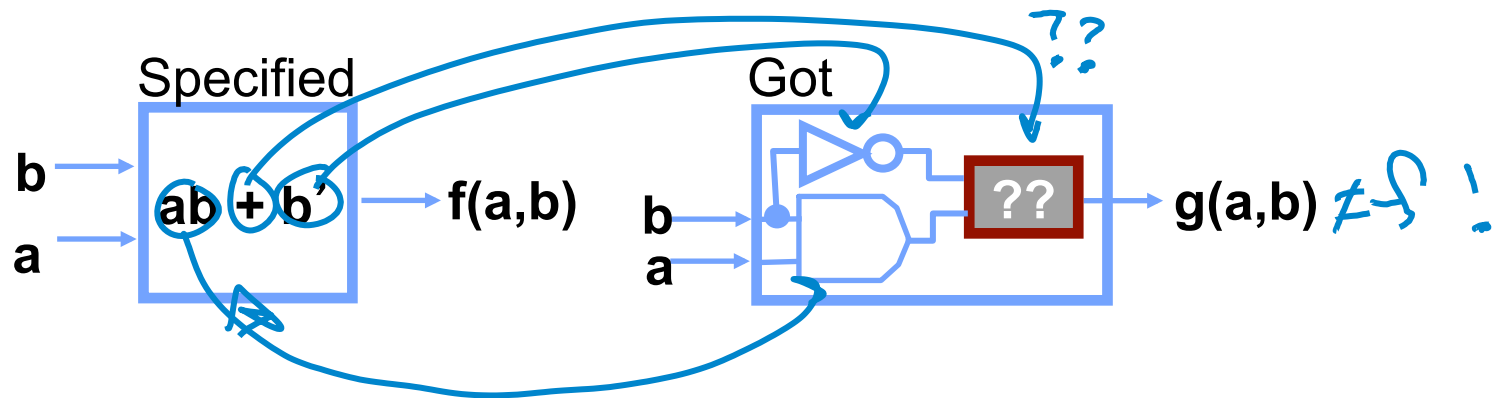University of Illinois

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... $f(a,b) = ab + b'$ → $a+b$ ignore

  - …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 (**==f**)

g(a,b)

Replace with
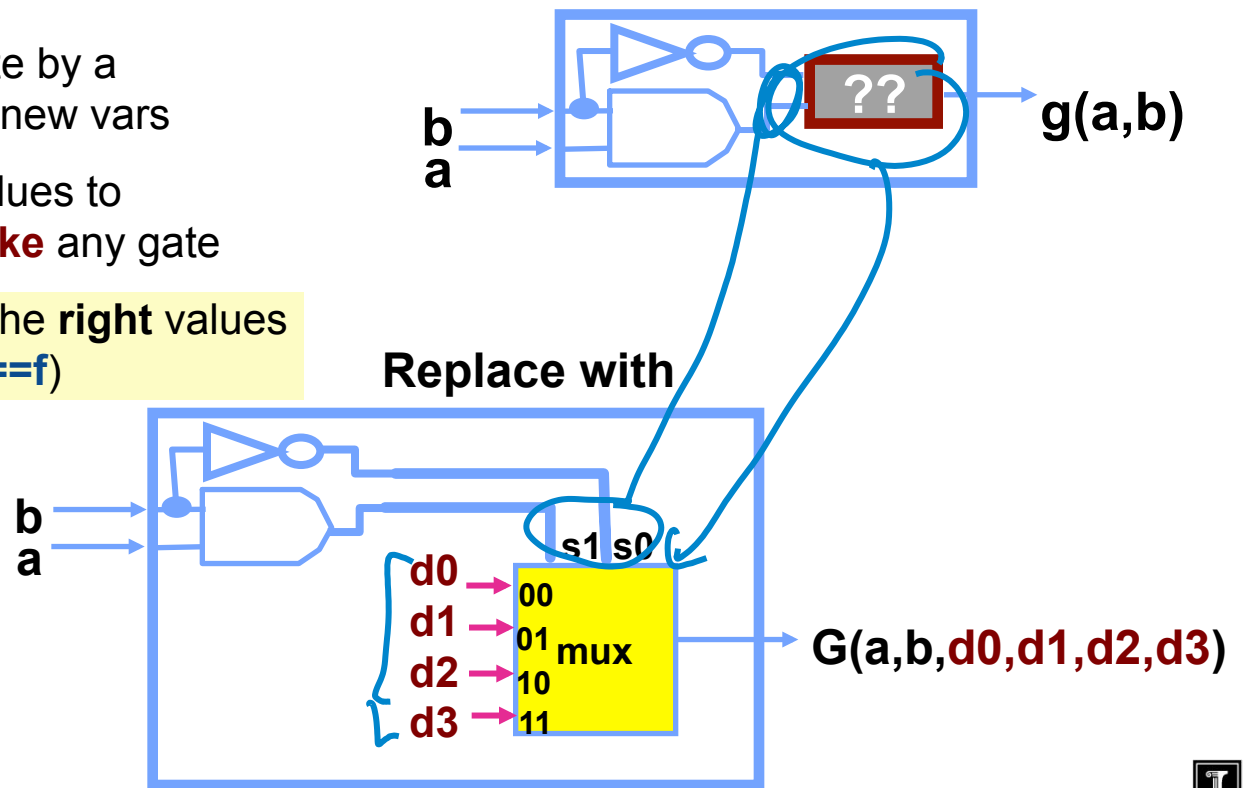
G(a,b,d0,d1,d2,d3)

Aside: Faking a Gate with a MUX

- **Remember...**

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

© 2013, R.A. Rutenbar
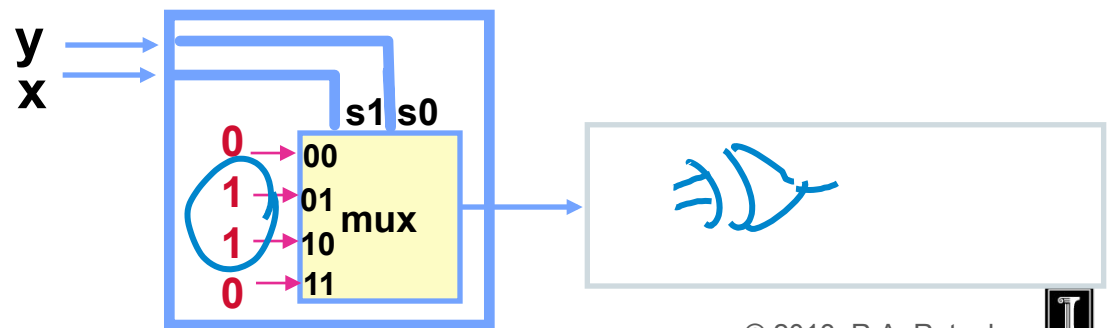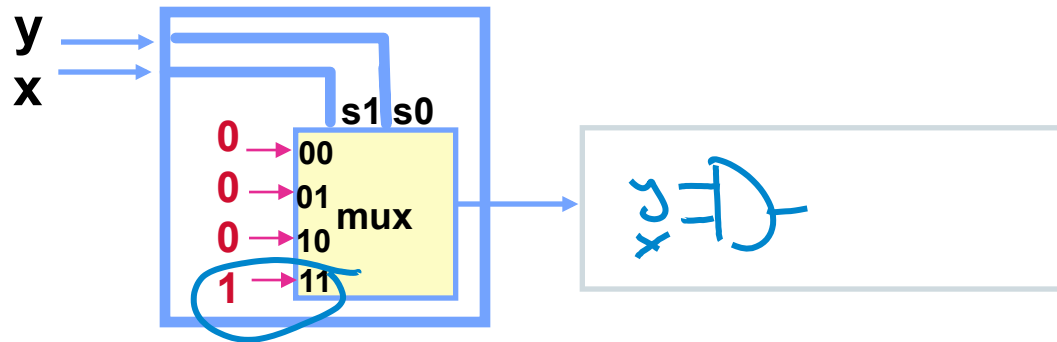
Network Repair: Using Quantification

- **Next trick**

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

$x \oplus y$

$xy + \bar{x}\bar{y}$

$= 1 \text{ iff } x = y$

G(a,b,d0,d1,d2,d3)

s1 s0

d0 → 00
d1 → 01   mux
d2 → 10
d3 → 11

exnor (==)

Z(a,b,d0,d1,d2,d3)

b
a

f(a,b)

known good version of logic
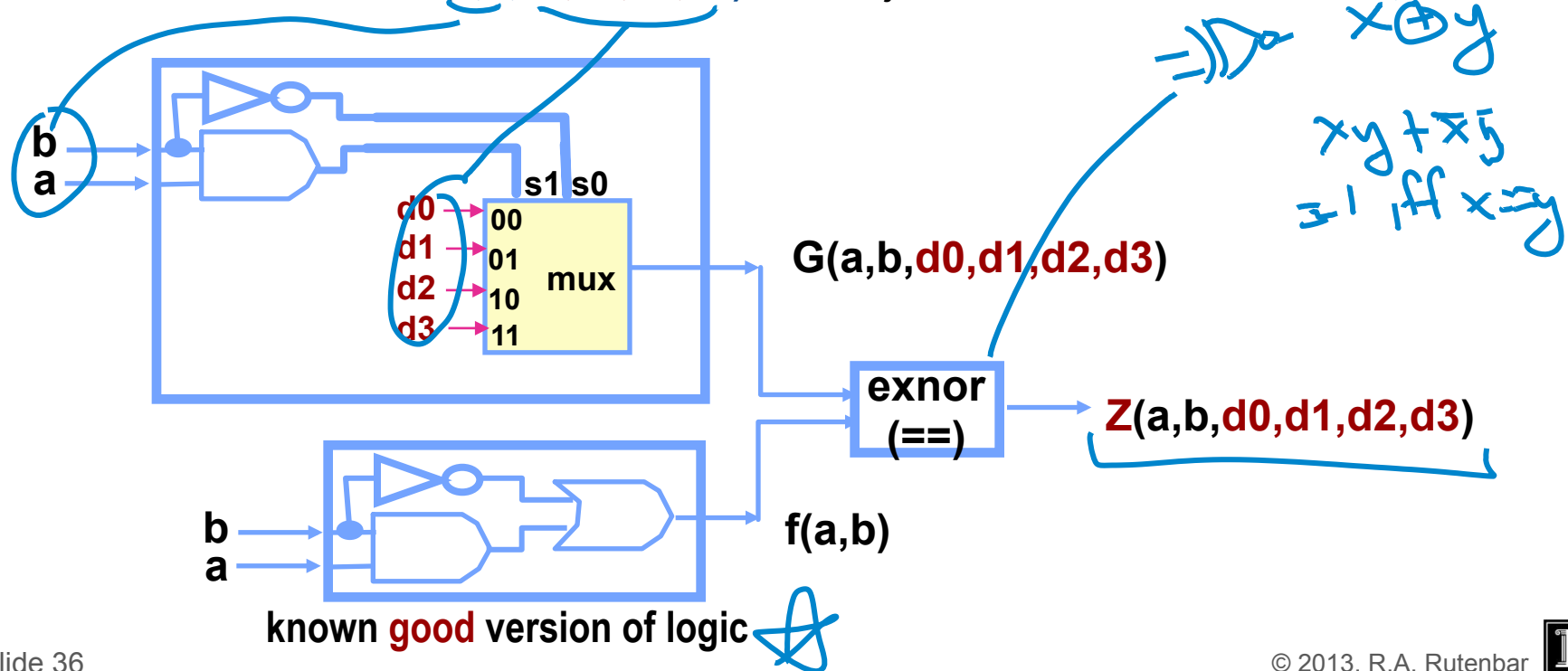
Using Quantification

- **What now?**

  - Think hard about exactly what we want:

  $$\text{values of } d_0 \, d_1 \, d_2 \, d_3 \text{ that make } z == 1$$
  $$\underline{\text{for}} \; \underline{\text{all}} \text{ possible values of inputs } a, b.$$
  $$(\forall_{ab}, z) \left[d_0 \, d_1 \, d_2 \, d_3 \right] == 1 \, ?$$

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

  - **Universal quantification** of function **Z** wrt variables **a,b**!

  - Any pattern of **(d0 d1 d2 d3)** that makes **($\forall$ab Z)(d0,d1,d2,d3)==1** will do it!

  - (Aside: do you know where **a, b** went??)

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

$G(a,b,d0,d1,d2,d3)$

$Z(a,b,d0,d1,d2,d3)$

$f(a,b)$

ab+b'

$$\{0\ 0\ 0\}$$

$$G = d_0 a\bar{b}\,\bar{s} + d_1 \bar{a}\bar{b}\,\bar{s} + d_2 ab\,\bar{s}$$
$$+ d_2 ab\,\bar{s}$$

$$G = d_0 \bar{a}b + d_1 \bar{b} + d_2 ab$$

$$Z = G \oplus f = Gf + \bar{G}\bar{f}$$

$$(\forall ab\ Z) = Z_{ab} \cdot Z_{\bar{a}b} \cdot Z_{a\bar{b}} \cdot Z_{\bar{a}\bar{b}}$$

Network Repair via Quantification: Continued

G   f

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

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

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

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

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

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

$d1 \overline{0}' = d1$

$d0 \overline{0} 0 = \overline{d0}$

$d1 \overline{0} 1 = d1$

$d2 \overline{0} 1 = d2$

$[\forall ab \ Z](d0d_1, d2d_2) = AND = \overline{d_0} d d_2 \ !$

Repair via Quantification: Continued

- **So, we got this: $(\forall ab\ Z)[d0,d1,d2,d3] = \boxed{d0' \cdot d1 \cdot d2}$**

- **And know this: if can solve $(\forall ab\ Z)[d0,d1,d2,d3]==1 \rightarrow$ repaired!**

- **Hey – this one is not very hard…**

$$d_0 = 0$$
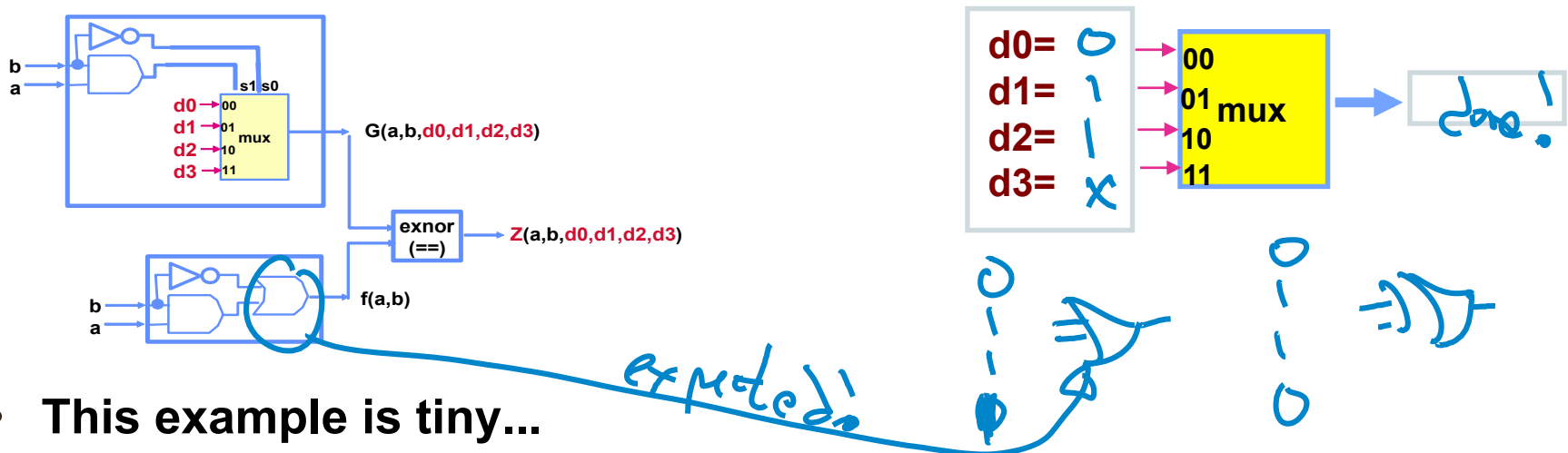$$d_1 = 1$$
$$d_2 = 1$$
$$d_3 = X \quad \text{don't care!}$$

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)(d0,d1,d2,d3)==1** for gate debug

  - This computation is called **Boolean Satisfiability (also called SAT)**

- **Ability to do Boolean SAT efficiently is a big goal for us**

  - **We will see how to do this in later lectures…**

VLSI CAD:
Logic to Layout

Lecture 2.5

Computational Boolean Algebra: URP Tautology

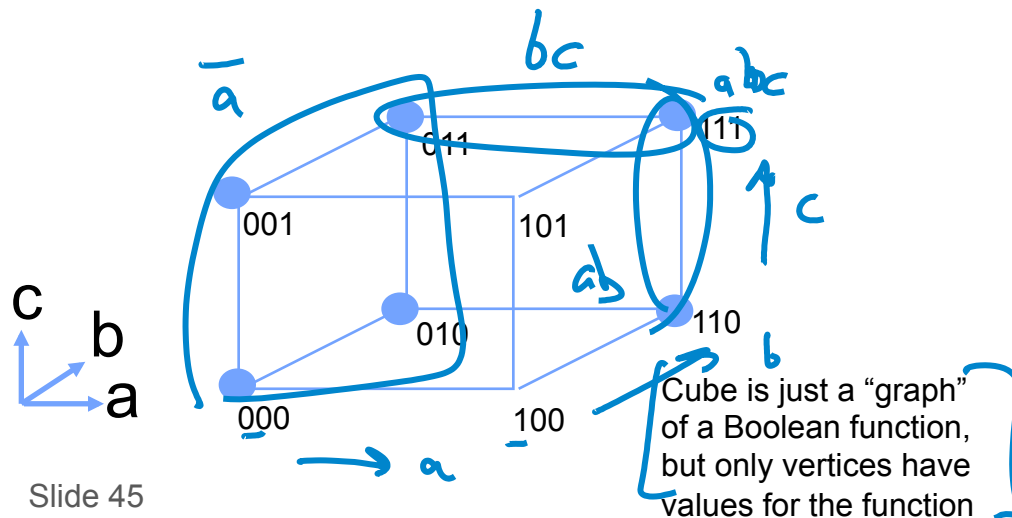**Rob A. Rutenbar**
University of Illinois

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...

- **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

$\bar{a} \perp bc + ab$

$\not\Rightarrow$ NOT NEC minimal

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

Positional Cube Notation (PCN)

- **So, we say 'cube' and mean 'product term'**
    - So, how to represent each cube? (**PCN:**) one slot per variable, 2 bits per slot
    - Write each cube by just noting which variables are true, complemented, or absent
        - In slot for var **x**:   put **01**   if product term has **...x...** in it
        - In slot for var **x**:   put **10**   if product term has **...x'...** in it
        - In slot for var **x**:   put **11**   if product terms has **no x or x'** in it
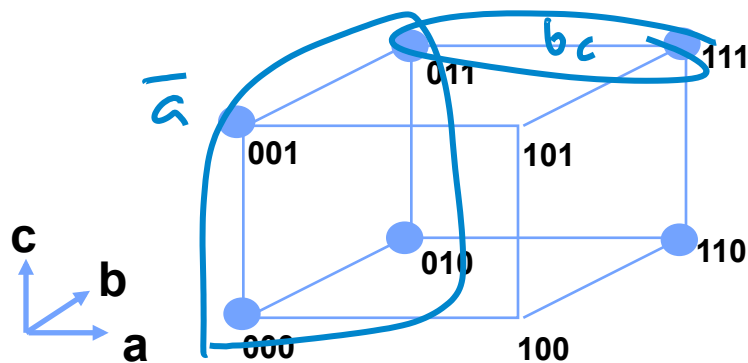
$$\bar{a} = \begin{bmatrix} \overset{a}{10} & \overset{b}{11} & \overset{c}{11} \end{bmatrix}$$

$$bc = \begin{bmatrix} 11 & 01 & 01 \end{bmatrix}$$

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)=a' +bc +ab => [01 11 11], [11 01 01], [01 01 11]**
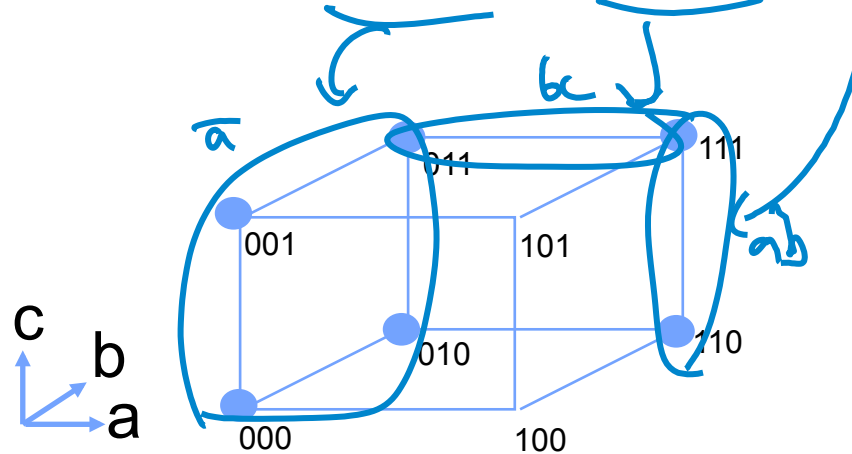
Tautology Checking

- **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* fx and fx' are *both* tautologies**
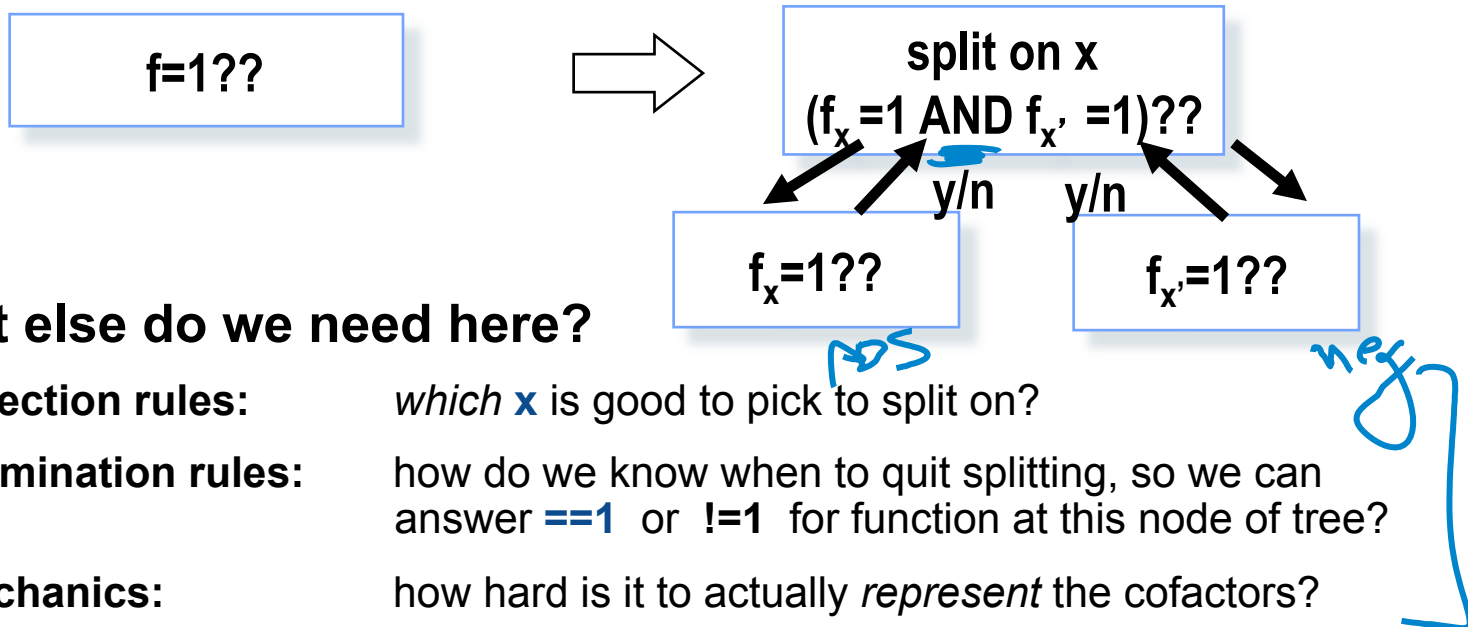
  - This makes sense: $f(x=0), f(x=1) = 1$

    - If function **f() = 1** → then cofactors both obviously **= 1** ✓

    - If **both** cofactors **= 1** → **x•F(x=1) + x'•F(x=0) = x•1 + x'•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** !

- **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?

VLSI CAD:
Logic to Layout
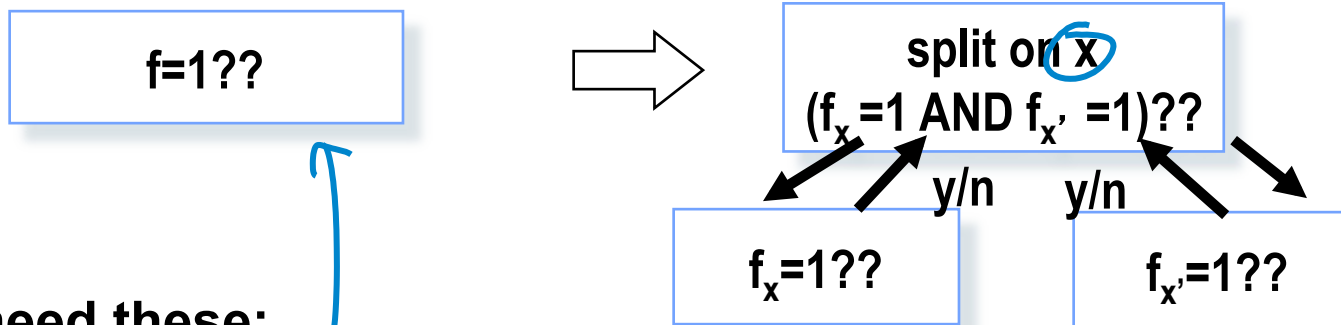
Rob A. Rutenbar
University of Illinois

Lecture 2.6

Computational Boolean
Algebra: URP Tautology—
Full 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** !

| f=1?? | $\Rightarrow$ | split on x
 $(f_x = 1 \text{ AND } f_{x'} = 1)??$ |

y/n   y/n

| $f_x = 1??$ | $f_{x'} = 1??$ |

- **We need these:**

  - **Selection rules:**   *which* **x** is good to pick to split on?

  - **Termination rules:**   how do we know when to quit splitting, so we can answer **==1** or **!=1** for function at this node of tree?

  - **Mechanics:**   how hard is it to actually *represent* the cofactors?

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

f = abd+bc'	f_a a->1	f_c c->1
abd [01 01 11 01]	bd [11 01 11 01]	abd [01 01 11 01]
bc [11 01 10 11]	bc [11 01 10 11]	b [11 01 11 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: xy + x'y + 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→1** keeps **f constant** or makes **f: 0→1**

  - **f** is **negative unate** in var **x** -- if changing **x 0→1** keeps **f constant** or makes **f: 1→0**

  - Function that is not unate is called **binate**

Unate Functions

- **Analogous to monotone continuous functions**

y

This function is monotone non-decreasing,
whenever $x2 \geq x1$, then $f(x2) \geq f(x1)$

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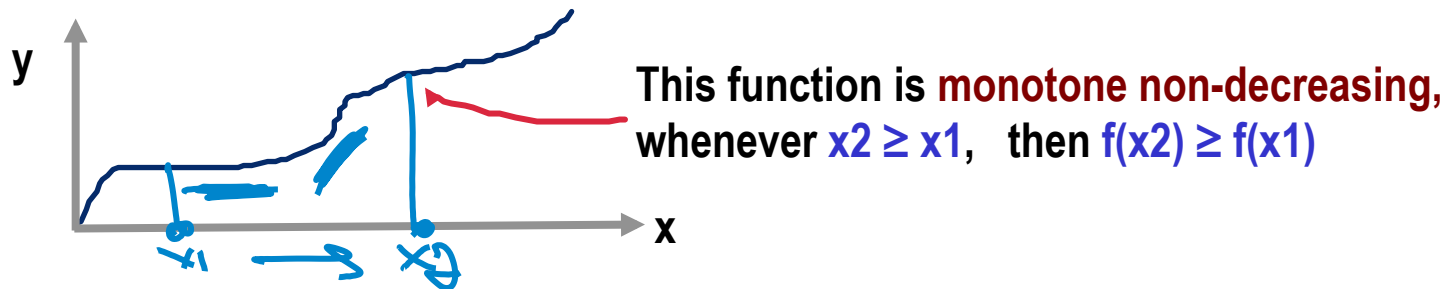
- **Example: for a Boolean function f positive unate in x**

variable x → | Boolean function f | → f(... x ...)
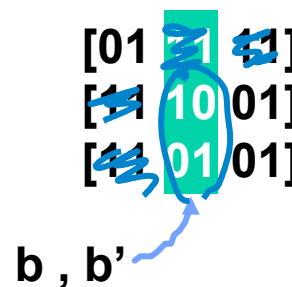
© 2013, R.A. Rutenbar
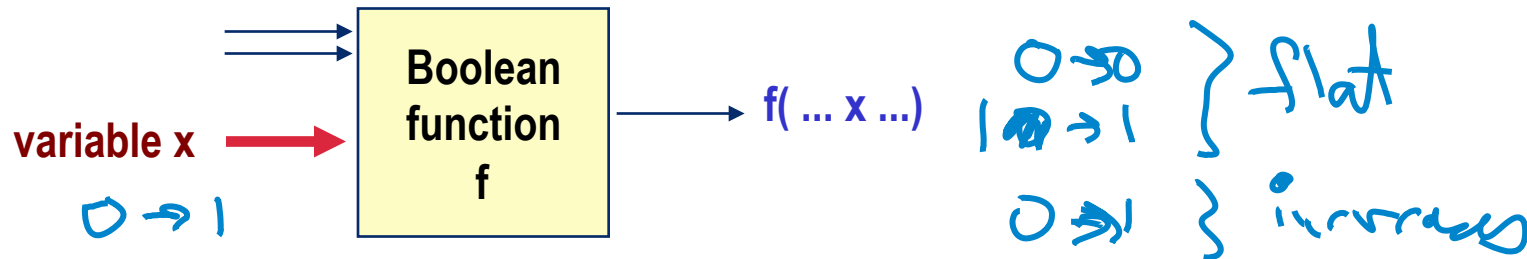
Can Exploit Unate Func's For Computation

- **Suppose you have a cube-list for f**

  - A cube-list is **unate** if each var in each cube only appears in **one** polarity, not both

  - Ex: **f(a,b,c)=a +bc +ac** → **[01 11 11], [11 01 01], [01 11 01]** is **unate**

  - Ex: **f(a,b,c)=a +b'c +bc** → **[01 11 11], [11 10 01], [11 01 01]** is **not**

  - Easier to see if draw vertically

a+b'c+ac UNATE

[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'
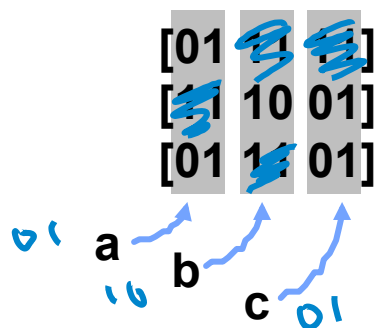
Using Unate Functions in Tautology Checking

- **Beautiful result**

  - It is very **easy** to check a unate cube-list for tautology

  - **Unate cube-list for f is tautology iff it contains *all don't care* cube = [11 ... 11]**

  - Reminder: what exactly is **[11 11 11 ... 11]** as a product term?

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

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

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

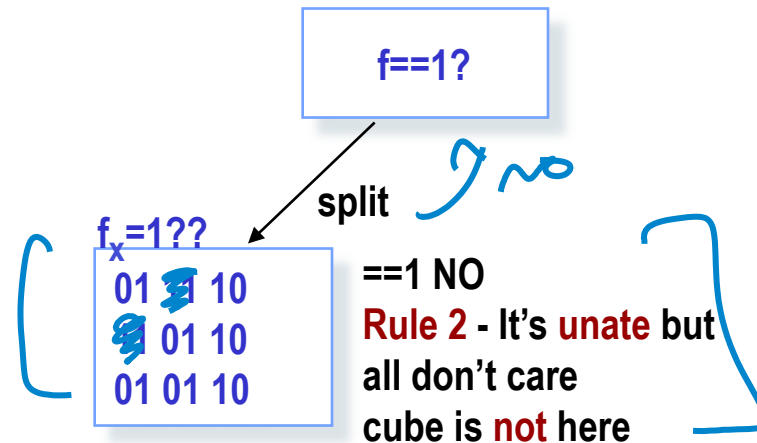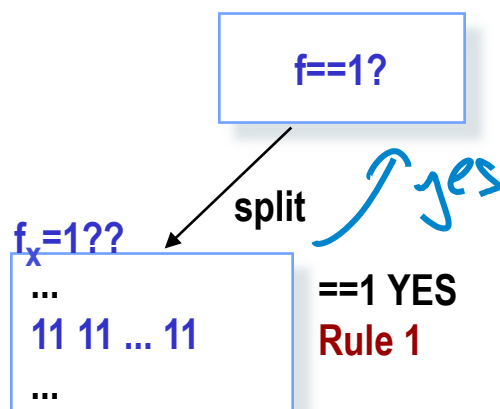| cd	00	01	11	10
ab 00				
01				
11				
10				

So, Unateness Gives Us *Termination Rules*

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

    Rule 1: **==1** if cube-list has all don't care cube **[11 11 ... 11]**
    Why: function at this leaf is **(stuff + 1 + stuff) == 1**   *easy-linear*

    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

f==1?

f_x=1??
...
11 11 ... 11
...

yes

==1 YES
Rule 1

f==1?

f_x=1??
01 11 10
01 01 10
01 01 10

no

==1 NO
Rule 2 - It's **unate** but
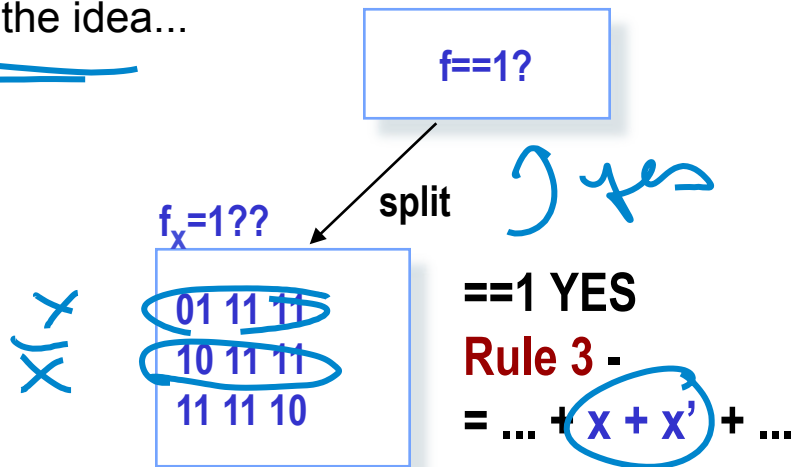all don't care
cube is **not** here

Recursive Tautology Checking

- **Lots more possible rules...**

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

  - You get the idea...

$f==1?$

$f_x=1??$

split

01 11 11
10 11 11
11 11 10

==1 YES
Rule 3 -
$= ... + x + x' + ...$

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*)

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

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

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

unate

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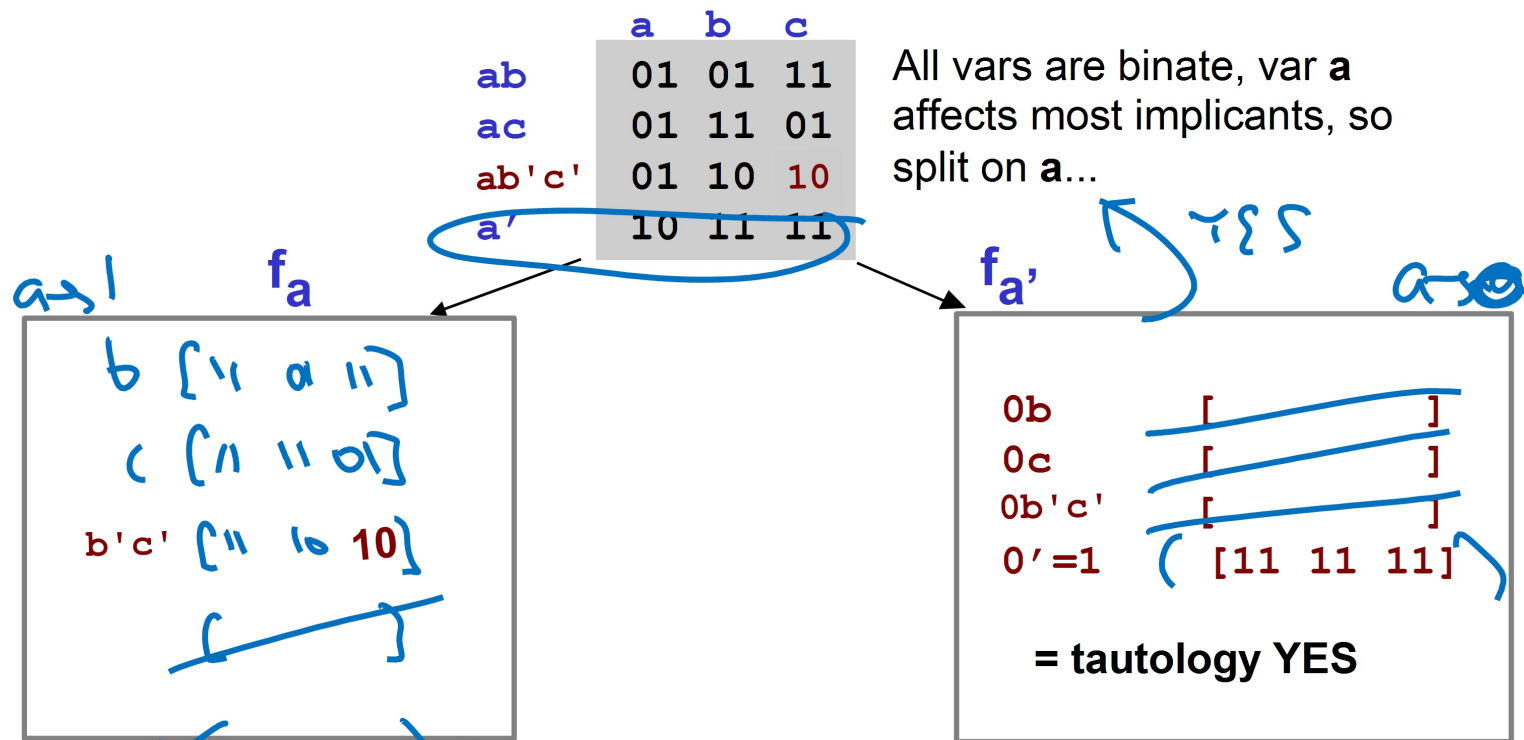
binate, in 4 cubes, | true - compl | = 2-2 = 0

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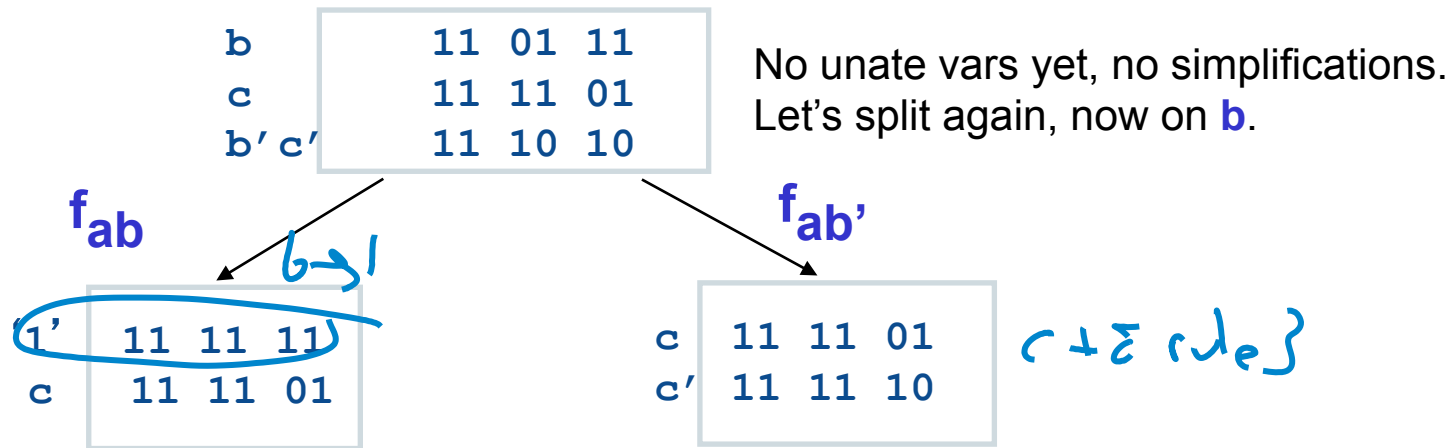
- **Algorithm**   **tautology**(f represented as cubelist) {
    /* check if we can terminate recursion */
    if (**f** is unate) {
        apply unate tautology termination rules directly
        if (**==1**) return (**1**)
        else return (**0**)
    }
    else if (any other termination rules,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

terminate

select

recurse

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**...

f_a

$f_{a'}$

(handwritten annotations on left box)

b [11 a 11]
c [10 11 01]
b'c' [11 10 **10**]

(handwritten annotations on right box)

0b [   ]
0c [   ]
0b'c' [   ]
0'=1 ([11 11 11])

= tautology YES

Recursive Tautology Checking: Example

```
b          11 01 11
c          11 11 01
b'c'       11 10 10
```

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

f_ab

$b \to 1$

```
1'   11 11 11
c    11 11 01
```

f_ab'

```
c    11 11 01
c'   11 11 10
```

$c + \bar{c}$ rule}

- **So we are done:**

  - Our tree has tautologies at all leaves!

  - Note -- if any leaf **!=1**, then **f != 1** too, this is how tautology fails

f
├── f_a
│   ├── f_ab ==1
│   └── f_ab' ==1
└── f_a' ==1
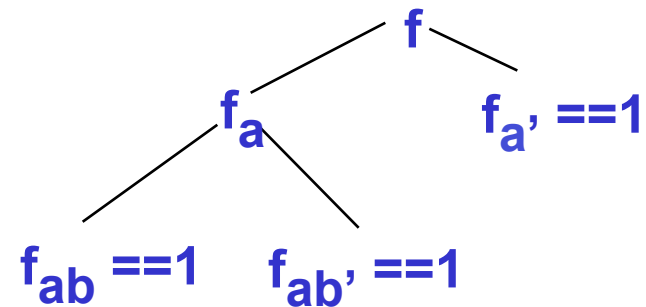
Computational Boolean Algebra

- **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