CENG4120

January 15, 2025

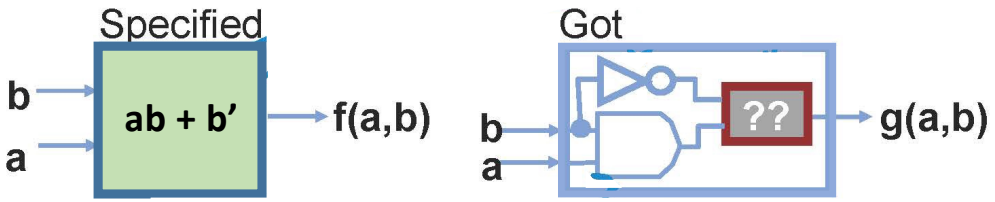
Lecture 3

Application of Computational Boolean Algebra

(This set of slides sources from R.A. Rutenbar @ UIUC)

Quantification App: Network Repair

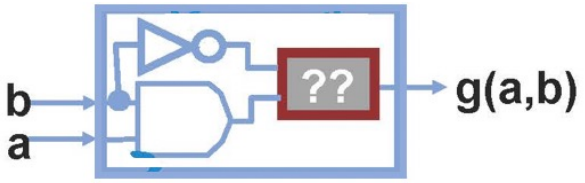
- **Suppose …**
  - I specified a logic block for you to implement …, **f(a,b) = ab + b'**
  - … but you implemented it **wrong**: in particular, you got **ONE** gate wrong

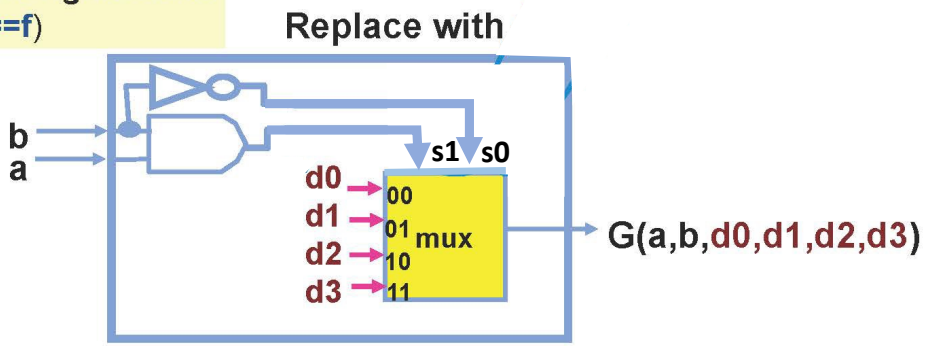


- **Goal**
  - Can we deduce how precisely to **change this gate** to restore correct function?
  - Lets go with this very trivial test case to see how mechanics work…

Network Repair

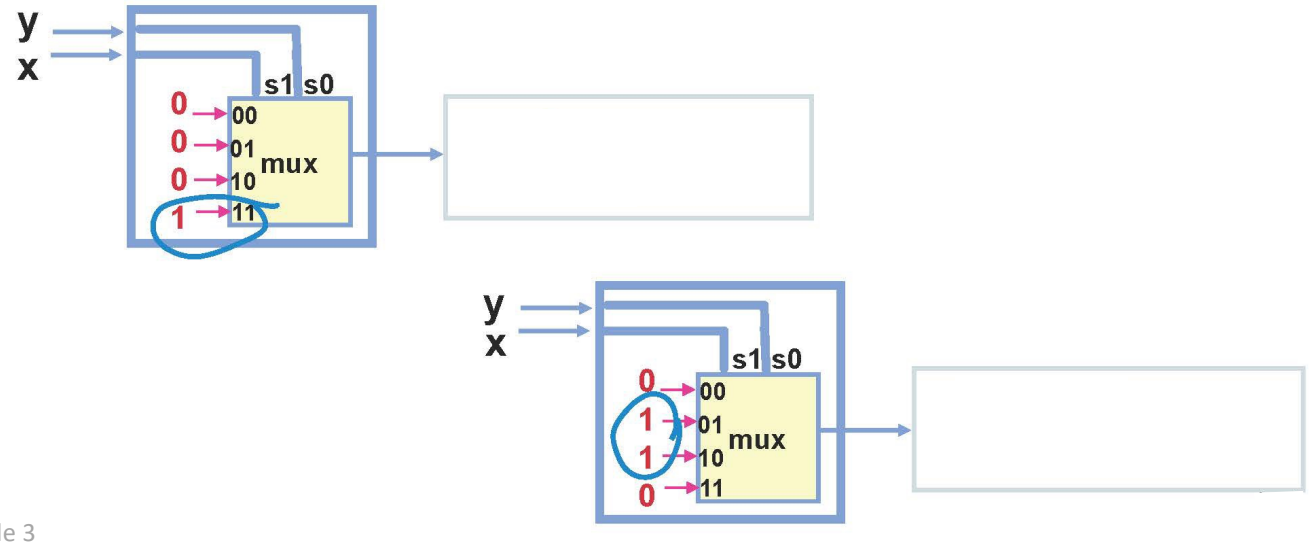
- **Clever trick**
  - Replace our suspect gate by a **4:1** mux with **4** arbitrary new vars
  - By cleverly assigning values to **d0 d1 d2 d3**, we can **fake** any gate
- **Question is:** what are the **right** values of **d's** so **g** is repaired (**==f**)





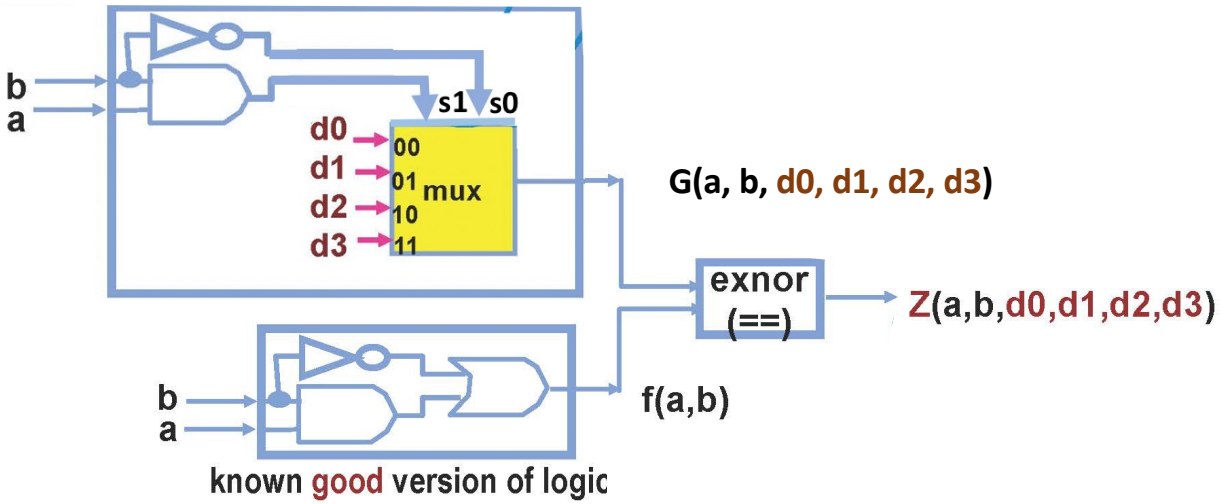
Aside: Faking a Gate with a MUX

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



Network Repair: Using Quantification

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

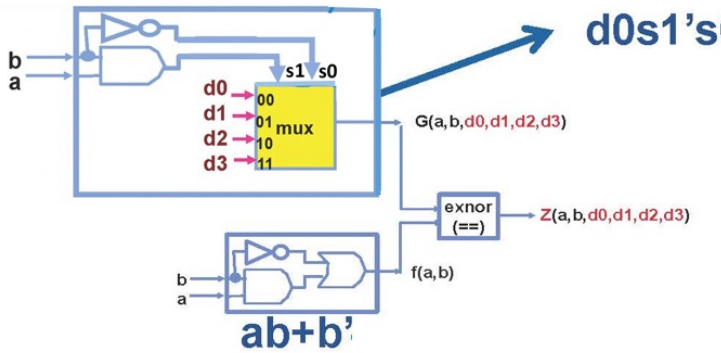


Using Quantification

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

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

Network Repair via Quantification: Try It…



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

Work out these s1's0', s1's0, s1s0' and s1s0.

Network Repair via Quantification: Continued

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

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

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

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

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

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

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

Class Exercise:
Can you work these out?

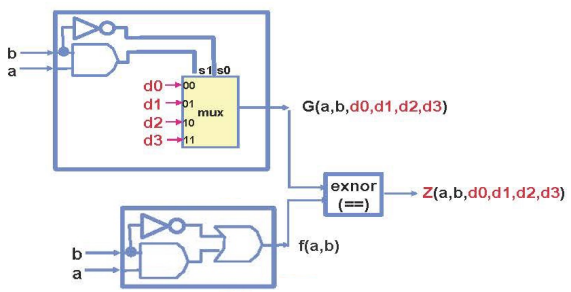
Repair via Quantification: Continued

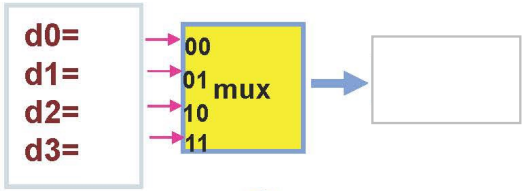
- **So, we got this:** $(\forall\ ab\ Z)[d0, d1, d2, d3] = d0' \cdot d1 \cdot d2$
- **And know this: if can solve** $(\forall\ ab\ Z)[d0, d1, d2, d3] == 1$ **→ repaired**

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

Network Repair

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





Class Exercise:
What happen to d3?

- **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 Algebra

- **What haven't we seen yet? Computational strategies in general**
  - Example: find inputs to make **(∀ab Z)(d0,d1,d2,d3)==1** for gate debug
  - This computation is called **Boolean Satisfiability (also called SAT)**
- **Ability to do Boolean SAT efficiently is a big goal for us**
  - **We will see how to do this in later lectures…**

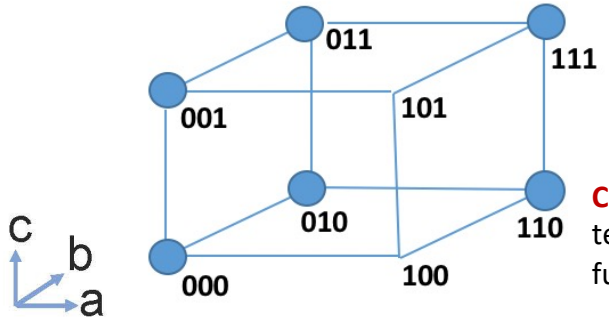
Important Computation: Tautology

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

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

Start with: Representation

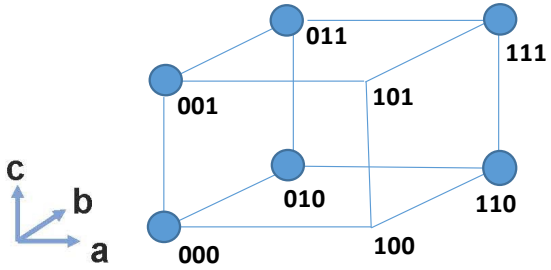
- **We use a simple, early representation scheme for functions**
  - Represent a function as a set of OR'ed product terms (i.e., a sum of products)
  - **Simple visual:** use a 3-var **Boolean cube**, with solid circles where **f() = 1**
  - So: each product term (circle in a Kmap) called a "**cube**"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



Cube is a product term of a Boolean function.

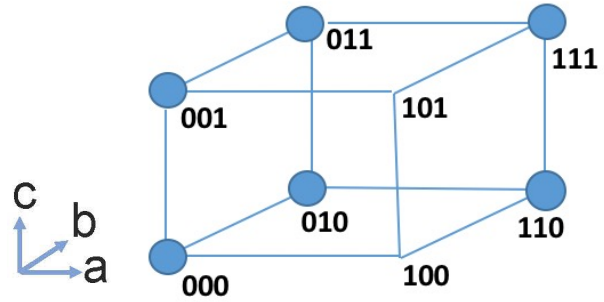
Positional Cube Notation (PCN)

- **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



PCN Cube List = Our Representation

- **So, we represent a function as a cover of cubes (circle its 1's)**
  - This is a **list of cubes** (this is the 'sum') in **positional cube notation** (of products)
  - Ex: **f(a,b,c)=a′ +bc +ab => [10 11 11], [11 01 01], [01 01 11]**

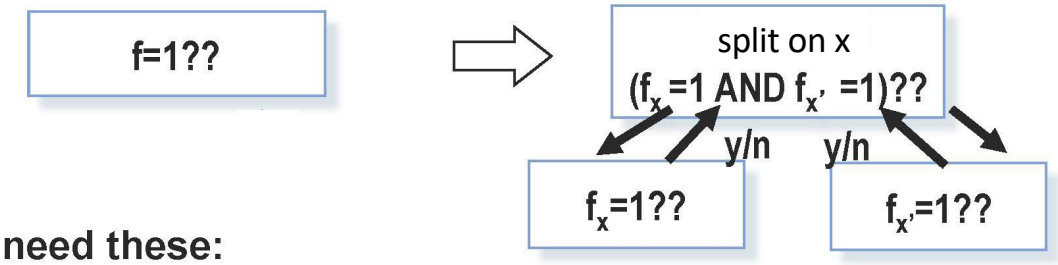


Tautology Checking

- **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:
    - If function **f() = 1** → then cofactors both obviously = 1
    - If **both** cofactors = 1 → $x \cdot \mathrm{f}(\mathrm{x} = 1) + \mathrm{x}' \cdot f(x = 0) = x \cdot 1 + x' \cdot 1 = x + x' = 1$

Recursive Tautology Checking

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



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

Recursive Cofactoring

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

$f = abd+bc'$	f_a	f_c
[01 01 11 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 in **exactly one polarity**, either all true, or all complemented

    Ex: ab + ac'd + c'de' is …

    Ex: xy + x'y + xyz' + z' is …

- **Terminology**
  - **f** is **positive unate** in var **x** -- **x** appears in positive polarity, *in this context*
  - **f** is **negative unate** in var **x** -- **x** appears in negative polarity, *in this context*
  - Function that is not unate is called **binate**

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

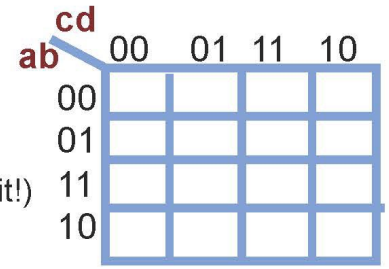
a+b'c+bc NOT

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

Using Unate Functions in Tautology Checking

- **Beautiful result** **(Can you prove this? The *if* can be easily seen. The *only-if* can be proved by induction on the number of variables.)**
  - It is very **easy** to check a unate cube-list for tautology
  - **Unate cube-list for f is tautology iff it contains an *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] =**

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

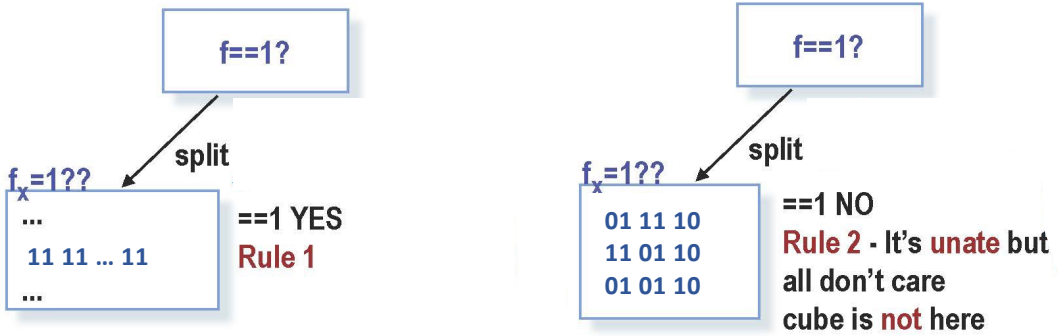


So, Unateness Gives Us *Termination Rules*

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

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

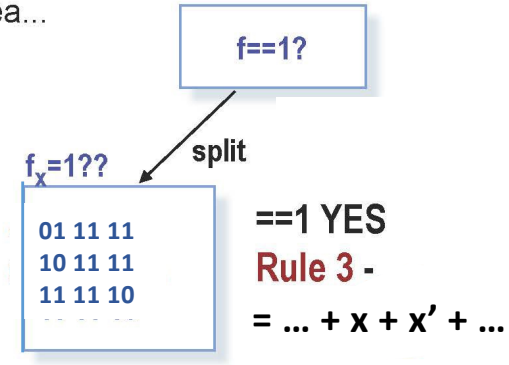
Rule 2: **!=1** if cube-list unate and all don't care cube **missing**
Why: unate ==1 if and only if has [11 11 ... 11] cube



Recursive Tautology Checking

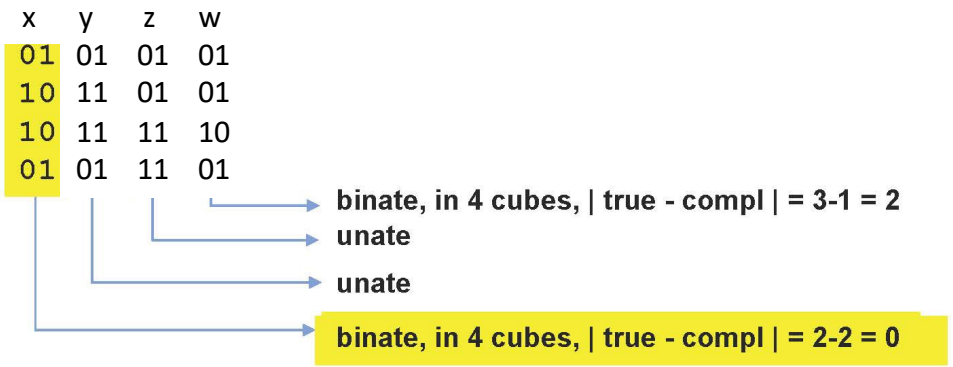
- **Lots more possible rules...**
  - **Rule 3:** **==1** if cube list has single var cube that appears in both polarities
    Why: function at this leaf is **(stuff + x + x' + stuff) == 1**

  - You get the idea...



Recursive Tautology Checking

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



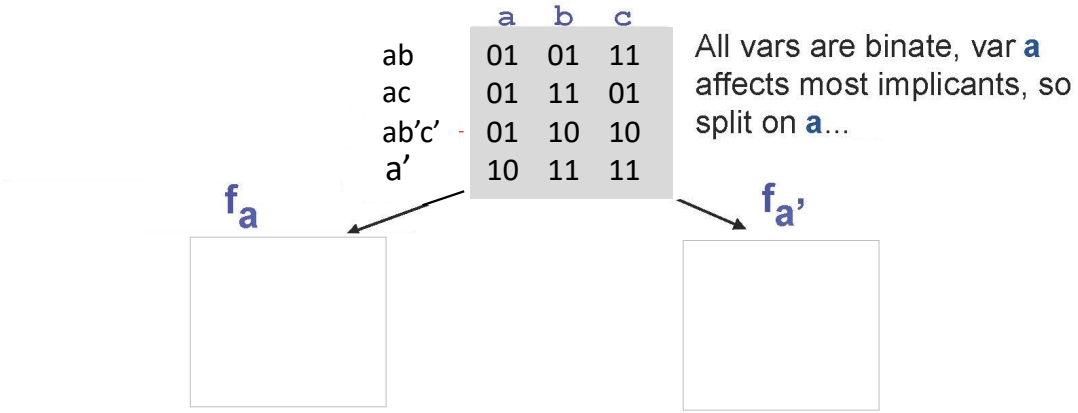
Recursive Tautology Checking: *Done*!

- **Algorithm**

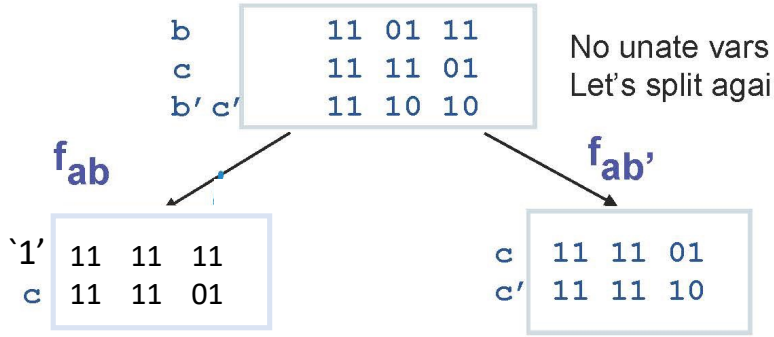
```
tautology(f represented as cubelist) {
    /* check if we can terminate recursion */
    if (f is unate)  {
        apply unate tautology termination rules directly
        if (==1) return (1)
        else return (0)
    }
    else if (any other termination rules, like rule 3, work?) {
        return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

Recursive Tautology Checking: Example

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

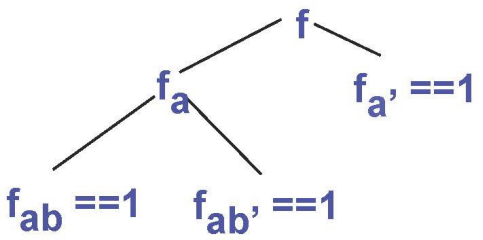


Recursive Tautology Checking: Example



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

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



Computational Boolean Algebra

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

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

Class Exercise

Determine if **ab'+bc'+a'd'+c'd** is a tautology using the URP method.

(Please work on it and submit.)