# VE527

## Computer-Aided Design of Integrated Circuits

Computational Boolean Algebra

# Outline

- Cofactor and Shannon Expansion
- Combinations of Cofactors
  - Boolean Difference
  - Quantification
  - Quantification Application: Network Repair
- Application of Computational Boolean Algebra: Tautology Checking

# Roadmap

- Going forward: Logic synthesis and verification
  - E.g., how to implement a Boolean function by a digital circuit? how to verify two digital circuits implement the same thing?
  - They deal with Boolean stuffs

- Begin with computational Boolean algebra

# Computational Boolean Algebra

## Motivation

- Background
  - You've done Boolean algebra, hand manipulations, Karnaugh maps to simplify…
  - But this is **not sufficient** for real designs!

- Example: a multiplier of two 16-bit numbers
  - It has 32 inputs.
  - Its Karnaugh map has $2^{32} = 4{,}294{,}967{,}296$ squares
    - This is too big!
  - There must be a better way…

# Need a Computational Approach

- Need **algorithmic**, **computational** strategies for Boolean stuff.
  - Need to be able to think of Boolean objects as **data structures + operators**

- What will we study?
  - **Decomposition strategies**
    - Ways of decomposing complex functions into simpler pieces.
    - A set of advanced concepts you need to be able to do this.
  - **Computational strategies**
    - Ways to manipulate Boolean functions by programs.
  - **Interesting applications**
    - When you have new tools, there are some useful new things to do.

# Advanced Boolean Algebra

## Useful Analogy to Calculus

- In calculus, you can represent complex functions like $e^x$ using simpler functions.
  - If you can only use $1, x, x^2, x^3, \ldots$ as the pieces …
  - … turns out $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$
- It corresponds to the **Taylor series expansion**.
  - $f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \cdots$

Question: Anything like this for Boolean functions?

Yes. It is called **Shannon Expansion**.

# Shannon Expansion

- Proposed by **Claude Shannon**, the father of information theory.

- Suppose we have a function $F(x_1, x_2, \ldots, x_n)$.
- Define **a new function** if we set one of the $x_i = const$
  - $F(x_1, x_2, \ldots, x_i = 1, \ldots, x_n)$
  - $F(x_1, x_2, \ldots, x_i = 0, \ldots, x_n)$
- Example: $F(x, y, z) = xy + x\bar{z} + y(\bar{x}z + \bar{z})$
  - $F(x = 1, y, z) = \ y + \bar{z} + y\bar{z}$
  - $F(x, y = 0, z) = \ x\bar{z}$

Note: this is a new function that no longer depends on the variable $x_i$.

# Shannon Expansion: Cofactors

- Turns out to be an incredibly useful idea.
- It is also known as **Shannon cofactor** with respect to $x_i$.
  - We write $F(x_1, x_2, \ldots, x_i = 1, \ldots, x_n)$ as $F_{x_i}$. We call it **positive cofactor**.
  - We write $F(x_1, x_2, \ldots, x_i = 0, \ldots, x_n)$ as $F_{\overline{x_i}}$. We call it **negative cofactor**.
  - Often, just write them as $F(x_i = 1)$ and $F(x_i = 0)$.

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

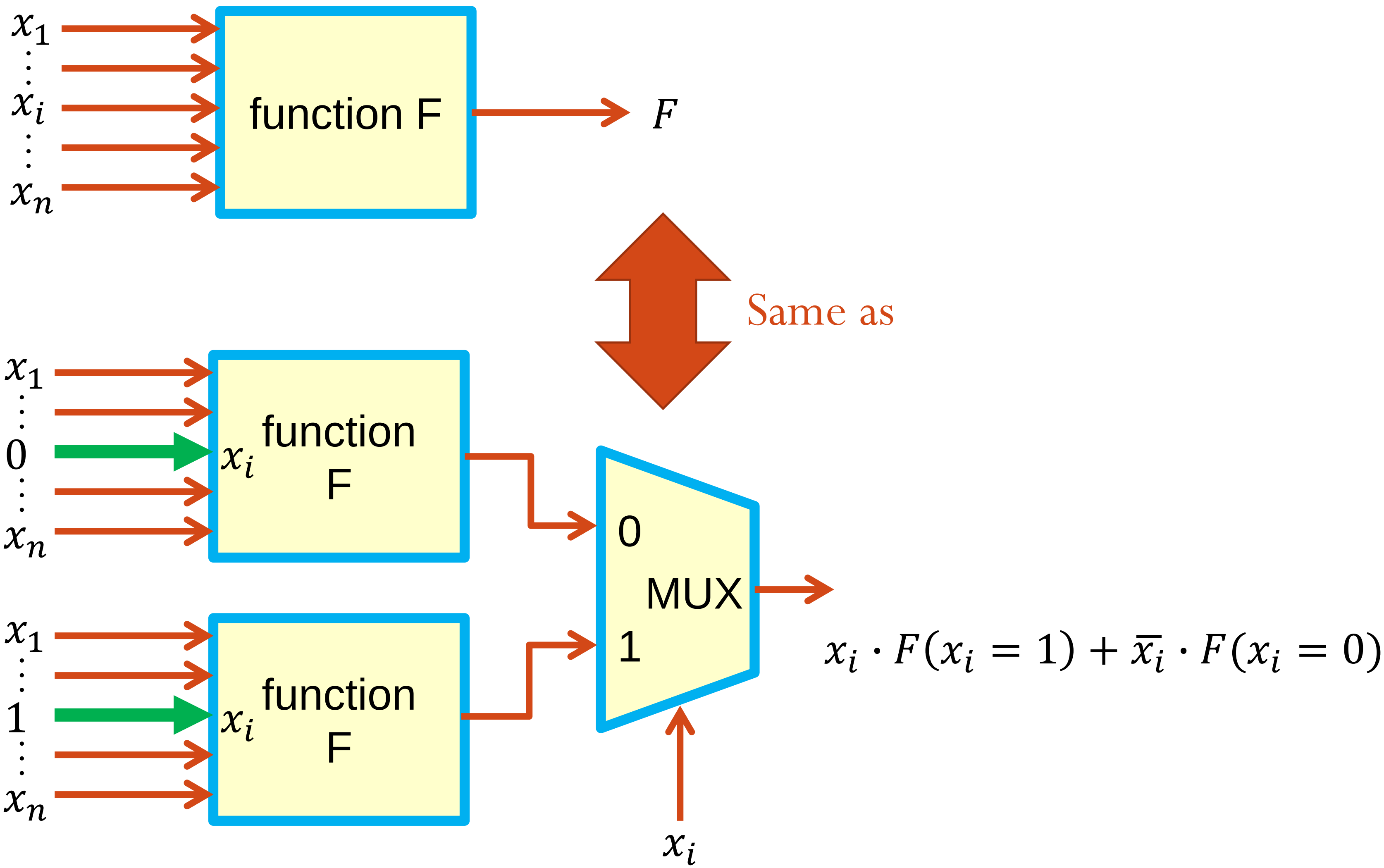
# Shannon Expansion Theorem

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

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

- Proof:
  - Consider any $(x_1, x_2, \dots, x_n) \in \{0,1\}^n$
    - If $x_i = 1$:
    - If $x_i = 0$:

# Shannon Expansion: Another View



# Shannon Expansion: Multiple Variables

- Can do it on **more than one** variable, too.
  - Just keep on applying the theorem on each variable.

- Example: Expand $F(x, y, z, w)$ around $x$ and $y$
  - First, expand around x:
    $$F(x, y, z, w) = x \cdot F(x = 1) + \bar{x} \cdot F(x = 0)$$
  - Then, expand cofactors $F(x = 1)$ and $F(x = 0)$ around $y$:
    $$F(x = 1) = y \cdot F(x = 1, y = 1) + \bar{y} \cdot F(x = 1, y = 0)$$
    $$F(x = 0) = y \cdot F(x = 0, y = 1) + \bar{y} \cdot F(x = 0, y = 0)$$
  - Final result:
    $$\begin{aligned} &F(x, y, z, w) \\ &= xy \cdot F(x = 1, y = 1) + x\bar{y} \cdot F(x = 1, y = 0) \\ &\quad + \bar{x}y \cdot F(x = 0, y = 1) + \bar{x}\bar{y} \cdot F(x = 0, y = 0) \end{aligned}$$

# Shannon Cofactors: Multiple Variables

- There is notation for these multiple-variable expansions as well.
- Shannon cofactor with respect to $x_i$ and $x_j$:
  - Write $F(x_1, \dots, x_i = 1, \dots, x_j = 0, \dots, x_n)$ as $F_{x_i\overline{x_j}}$.
  - The same for any number of variables $x_i, x_j, x_k, \dots$
  - Notice that order does **not** matter: $(F_x)_y = (F_y)_x = F_{xy}$.

- For the previous example:

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

- Again, remember: each of the cofactors is a **function**, not a number.
  - $F_{xy} = F(x = 1, y = 1, z, w)$ is a Boolean **function** of $z$ and $w$.

# Next Question: Properties of Cofactors

- What ***else*** can you do with cofactors?
- Suppose you have 2 functions $F(X)$ and $G(X)$, where $X = (x_1, x_2, \dots, x_n)$.
- Suppose you make a new function $H$, from $F$ and $G$, say…
  - $H = \bar{F}$
  - $H = F \cdot G$, i.e., $H(X) = F(X) \cdot G(X)$
  - $H = F + G$, i.e., $H(X) = F(X) + G(X)$
  - $H = F \oplus G$, i.e., $H(X) = F(X) \oplus G(X)$
- Question: can you tell anything about $H$'s cofactors from those of $F$ and $G$?
  - $(F \cdot G)_x =$ **what**? $(\bar{F})_x =$ **what**?

# Nice Properties of Cofactors

- Cofactors of $F$ and $G$ tell you everything you need to know.
- Complements
  - $(\bar{F})_x = \overline{(F_x)}$
  - In English: cofactor of complement is complement of cofactor.
- Binary Boolean operators
  - $(F \cdot G)_x = F_x \cdot G_x$ cofactor of AND is AND of cofactors
  - $(F + G)_x = F_x + G_x$ cofactor of OR is OR of cofactors
  - $(F \oplus G)_x = F_x \oplus G_x$ cofactor of XOR is XOR of cofactors
- **Very useful**! Can often help in getting cofactors of complex formulas.

# Outline

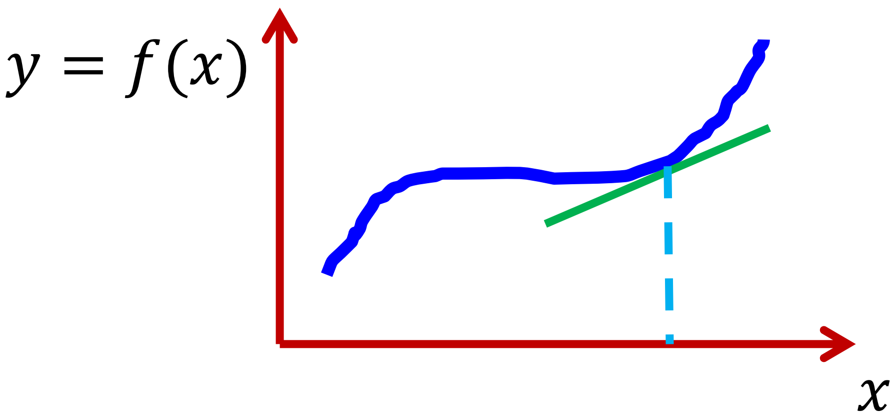
- Cofactor and Shannon Expansion
- Combinations of Cofactors
  - Boolean Difference
  - Quantification
  - Quantification Application: Network Repair
- Application of Computational Boolean Algebra: Tautology Checking

# Combinations of Cofactors

- Now consider **operations** on cofactors themselves.
- Suppose we have $F(X)$, and get $F_x$ and $F_{\overline{x}}$.
  - $F_x \oplus F_{\overline{x}} = ?$
  - $F_x \cdot F_{\overline{x}} = ?$
  - $F_x + F_{\overline{x}} = ?$
- Turns out these are all useful **new** functions.
  - Indeed, they even have **names**!
- Next: let's look at these interesting, useful new functions.

# Calculus Revisited: Derivatives

- Remember how you defined derivatives?
  - Suppose you have $y = f(x)$.




Defined as slope of curve at point $x$.

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

# Boolean Derivatives

- So, do Boolean functions have "**derivatives**"?
  - Actually, yes. Trick is how to define them…

- Basic idea
  - For real-valued $f(x)$, $\frac{df}{dx}$ tells how $f$ changes when $x$ changes.
  - For 0,1-valued Boolean function, we cannot change $x$ by small delta.
  - Can only change 0 $\leftarrow\rightarrow$ 1, but can still ask how $f$ changes with $x$ …
  - For Boolean function $f(x)$, define

$$\frac{\partial f}{\partial x} = f_x \oplus f_{\overline{x}}$$

# Boolean Derivatives

$$\frac{\partial f}{\partial x} = f_x \oplus f_{\overline{x}}$$

- Compare value of $f$ when $x = 0$ against when $x = 1$.
- $\frac{\partial f}{\partial x} == 1$ if and only if $f(x = 0)$ is different from $f(x = 1)$.
- $\frac{\partial f}{\partial x}$ is also known as **Boolean difference**.

# Boolean Difference

- Boolean difference also behaves sort of like regular derivatives…
- Can do on multiple vars. Order of variables does not matter
$$(\partial f/\partial x)/\partial y = (\partial f/\partial y)/\partial x$$
- Derivative of XOR is XOR of derivatives
$$\frac{\partial (f \oplus g)}{\partial x} = \frac{\partial f}{\partial x} \oplus \frac{\partial g}{\partial x}$$
  - Like addition
- If function $f$ is constant ($f = 1$ or $f = 0$ for all inputs), then $\partial f/\partial x = 0$ for any $x$.

# Boolean Difference

- But some things are just more complex
  - Derivatives of $(f \cdot g)$ and $(f + g)$ do not work the same…

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

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

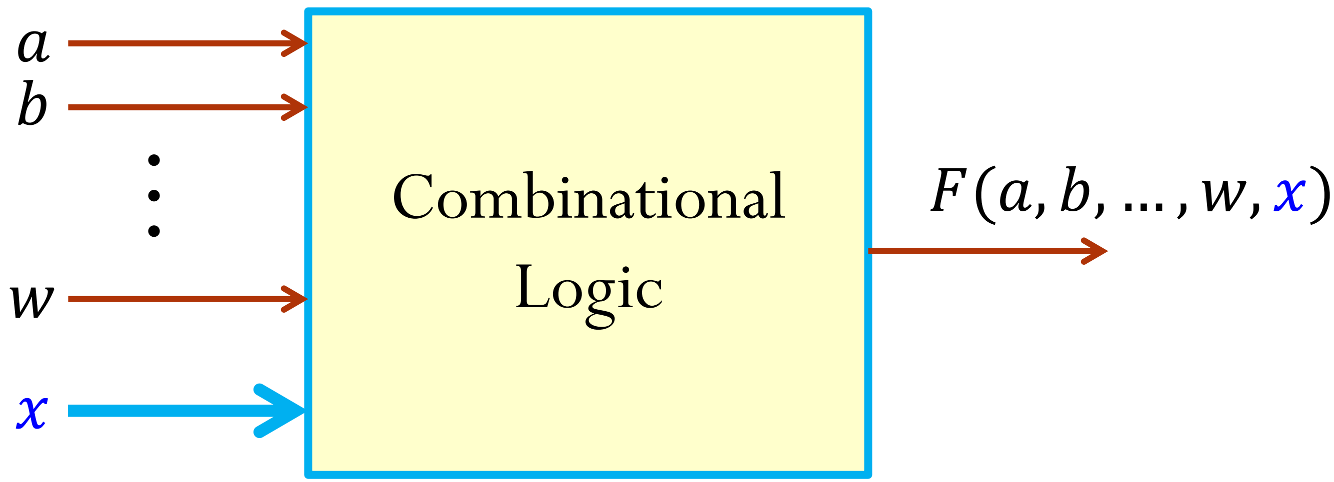
- Why?
  - Because AND and OR on Boolean values do not always behave like **ADDITION** and **MULTIPLICATION** on real numbers.

# Boolean Difference: Gate-Level View

- Consider simple examples for $\partial f / \partial x$.
- Inverter: $f = \bar{x}$
  - $f_x = 0, \; f_{\bar{x}} = 1, \; \partial f / \partial x = f_x \oplus f_{\bar{x}} = 1$
- AND: $f = xy$
  - $f_x = y, \; f_{\bar{x}} = 0, \; \partial f / \partial x = f_x \oplus f_{\bar{x}} = y$
- OR: $f = x + y$
  - $f_x = 1, \; f_{\bar{x}} = y, \; \partial f / \partial x = f_x \oplus f_{\bar{x}} = \bar{y}$
- XOR: $f = x \oplus y$
  - $f_x = \bar{y}, \; f_{\bar{x}} = y, \; \partial f / \partial x = f_x \oplus f_{\bar{x}} = 1$

Meaning: When $\partial f / \partial x = 1$, then $f$ changes if $x$ changes!

# Interpreting the Boolean Difference




- What does $\partial F(a, b, \dots, w, x)/\partial x = 1$ mean?
  - If you apply a pattern of inputs $(a, b, \dots, w)$ that makes $\partial F/\partial x = 1$, then any change in $x$ will force a change in output $F$.

# Boolean Difference: Example


$$s = a \oplus b \oplus c_{in}$$

$$c_{out} = ab + ac_{in} + bc_{in}$$

- When is $\partial c_{out} / \partial c_{in} = 1$?
  - $c_{out}(c_{in} = 1) = a + b$
  - $c_{out}(c_{in} = 0) = ab$
  - $\partial c_{out} / \partial c_{in} = c_{out}(c_{in} = 1) \oplus c_{out}(c_{in} = 0)$
    $= (a + b) \oplus (ab) = a \oplus b$
- Make sense?
  - $a \oplus b = 1 \Rightarrow a \neq b$

# Boolean Difference: Summary

- Boolean difference explains under what situations an input-change can cause output-change for a Boolean function $f$.

- $\partial f/\partial x$ is another Boolean function, but it does not depend on $x$!
  - It cannot, because it is made out of cofactors with respect to $x$, which eliminate all the $x$ and $\bar{x}$ terms by setting them to constants.

- **Very useful!** (we will see more, later…)

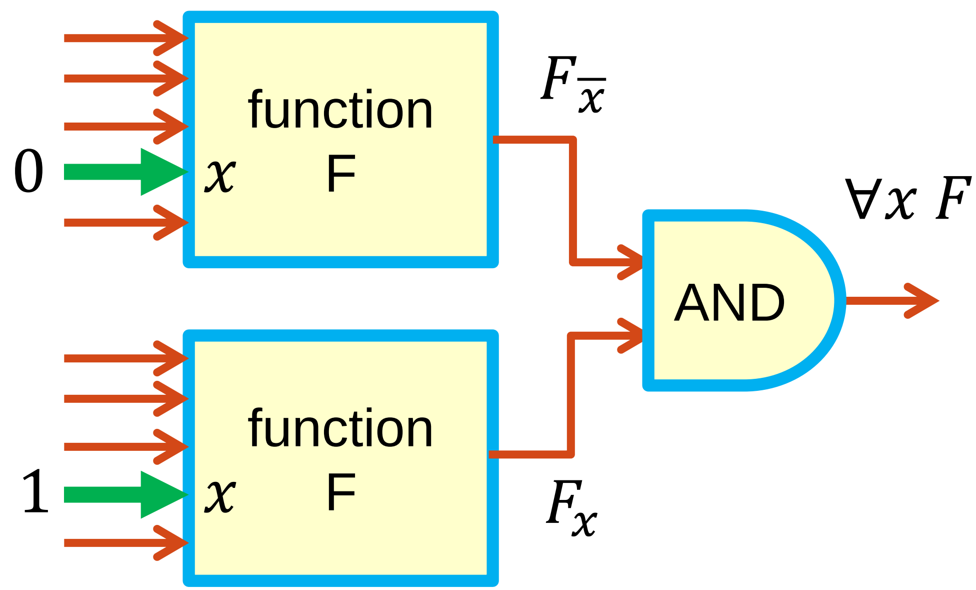
# Outline

- Cofactor and Shannon Expansion
- Combinations of Cofactors
  - Boolean Difference
  - Quantification
  - Quantification Application: Network Repair
- Application of Computational Boolean Algebra: Tautology Checking

# AND of $F_x$ and $F_{\bar{x}}$: Universal Quantification

- AND the cofactors: $F_{x_i} \cdot F_{\overline{x}_i}$
  - Name: **Universal Quantification** of function $F$ with respect to variable $x_i$.
  - Represented as: $(\forall x_i\ F)(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$

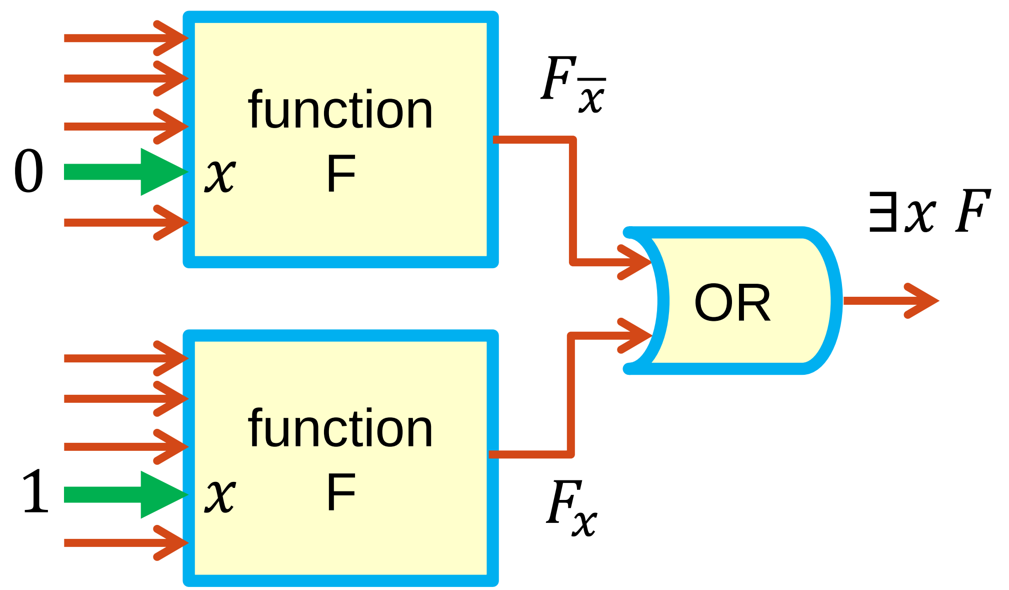
- $(\forall x_i\ F)$ is a new function
  - It does not depend on $x_i$!
  - "∀" sign is the "for all" symbol from logic.



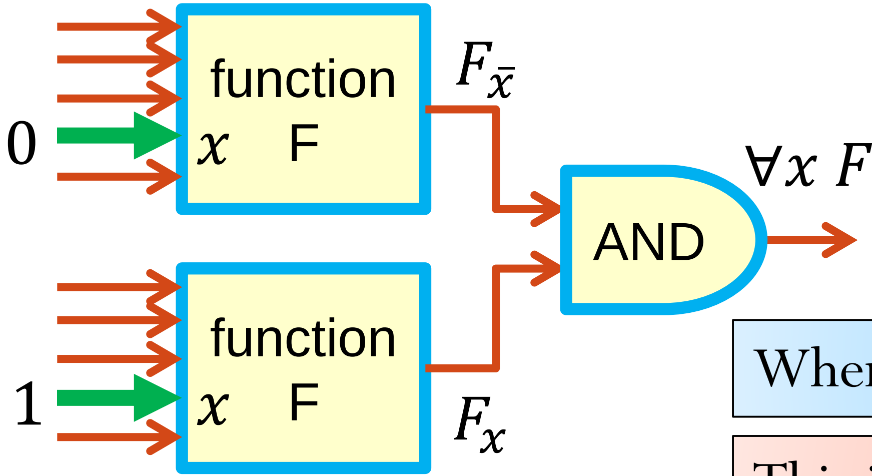
# OR of $F_x$ and $F_{\bar{x}}$: Existential Quantification

- OR the cofactors: $F_{x_i} + F_{\overline{x_i}}$
  - Name: **Existential Quantification** of function $F$ with respect to variable $x_i$.
  - Represented as: $(\exists x_i\ F)(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$

- $(\exists x_i\ F)$ is a new function
  - It does not depend on $x_i$!
  - "∃" sign is the "there exists" symbol from logic.

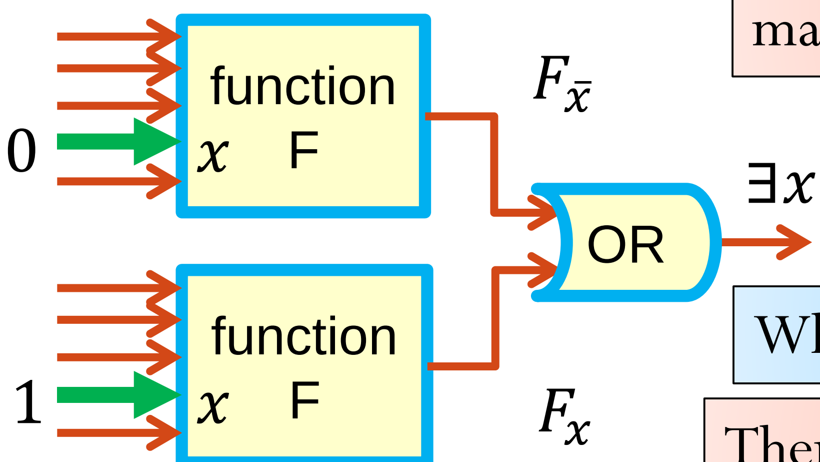


# Quantification Notation Makes Sense…




When does $\forall x\ F$(all vars except $x$) $= 1$?

This input pattern of **the other** vars makes $F = 1$ <u>for all</u> values of $x$.




When does $\exists x\ F$(all vars except $x$) $= 1$?

<u>There exists</u> a value of $x$ that makes $F = 1$ for this input pattern of **the other** vars.

# Quantification: Gate-Level View

- Consider simple examples for $(\forall x\, f)$ and $(\exists x\, f)$.
- Inverter: $f = \bar{x}$
  - $f_x = 0,\ f_{\bar{x}} = 1, (\forall x\, f) = f_x f_{\bar{x}} = 0, (\exists x\, f) = f_x + f_{\bar{x}} = 1$
- AND: $f = xy$
  - $f_x = y,\ f_{\bar{x}} = 0, (\forall x\, f) = f_x f_{\bar{x}} = 0, (\exists x\, f) = f_x + f_{\bar{x}} = y$
- OR: $f = x + y$
  - $f_x = 1,\ f_{\bar{x}} = y, (\forall x\, f) = f_x f_{\bar{x}} = y, (\exists x\, f) = f_x + f_{\bar{x}} = 1$
- XOR: $f = x \oplus y$
  - $f_x = \bar{y},\ f_{\bar{x}} = y, (\forall x\, f) = f_x f_{\bar{x}} = 0, (\exists x\, f) = f_x + f_{\bar{x}} = 1$

Make sense?

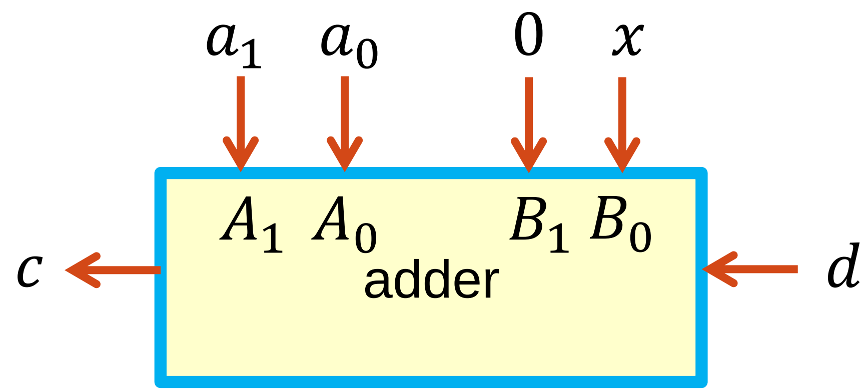
# Extends to More Variables in Obvious Way

- Like Boolean difference, can do with respect to more than 1 variable
  - Suppose we have $F(x, y, z, w)$.
  - $(\forall xy\ F)(z, w) = \big(\forall x\ (\forall y\ F)\big) = F_{xy} \cdot F_{x\bar{y}} \cdot F_{\bar{x}y} \cdot F_{\bar{x}\bar{y}}$
  - $(\exists xy\ F)(z, w) = \big(\exists x\ (\exists y\ F)\big) = F_{xy} + F_{x\bar{y}} + F_{\bar{x}y} + F_{\bar{x}\bar{y}}$

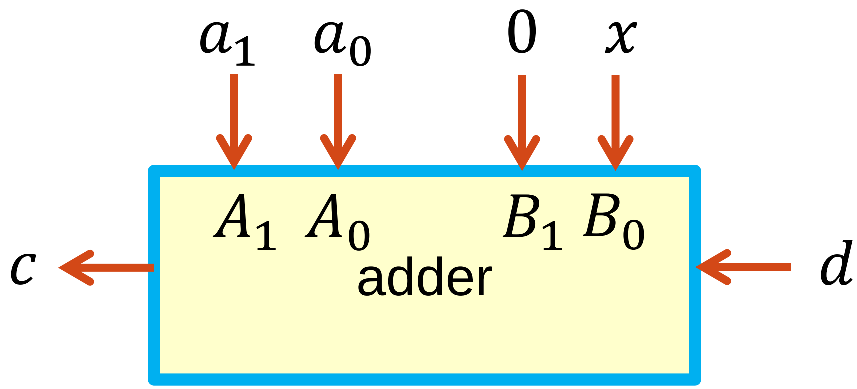
# Quantification Example

- Consider the following circuit, it adds $x = 0$ or $x = 1$ to a 2-bit number $a_1 a_0$.
  - It's just a 2-bit adder, but instead of $b_1 b_0$ for the second operand, it is just $0x$.
  - It has a carry-in $d$ and produces a carry-out $c$.
  - Hence, $c$ is function of $a_1, a_0, d$ and $x$.

- <u>Questions</u>:
  - What is $(\forall a_1 a_0\ c)(x, d)$?
  - What is $(\exists a_1 a_0\ c)(x, d)$?

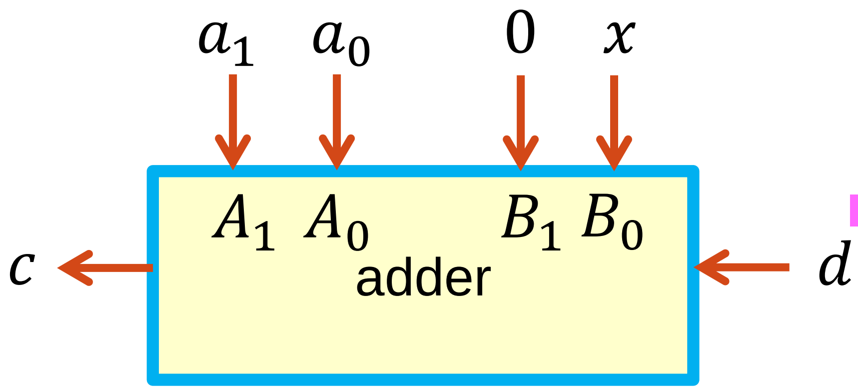


# Quantification Example




- What is $(\forall a_1 a_0\ c)(x, d)$?
  - A function of only $x$ and $d$. $x$ and $d$ that let this function be 1 should make carry $c = 1$ for **all values** of inputs $a_1$ and $a_0$.
- What is $(\exists a_1 a_0\ c)(x, d)$?
  - A function of only $x$ and $d$. $x$ and $d$ that let this function be 1 should make carry $c = 1$ for **some value** of inputs $a_1$ and $a_0$, i.e., there exists some $a_1$ and $a_0$ that for this $x$ and $d$, $c = 1$.

# Quantification Example

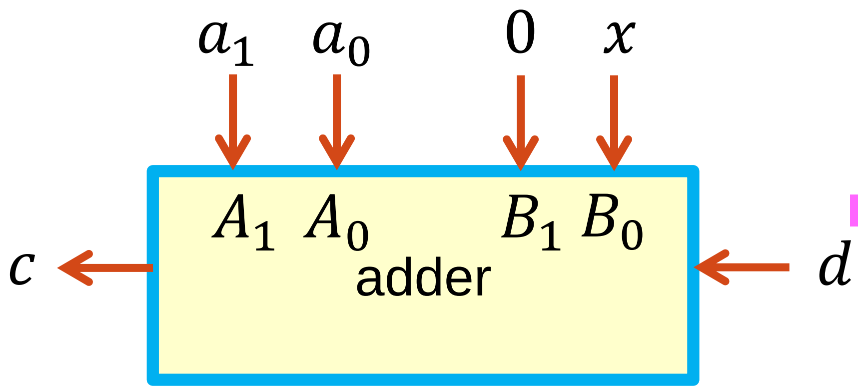



$$c = a_1(a_0 x + a_0 d + xd)$$

- Compute $(\forall a_1 a_0\ c)(x, d)$
  - $c_{a_1 a_0} \cdot c_{a_1 \bar{a}_0} \cdot c_{\bar{a}_1 a_0} \cdot c_{\bar{a}_1 \bar{a}_0} = 0$
- Compute $(\exists a_1 a_0\ c)(x, d)$
  - $c_{a_1 a_0} + c_{a_1 \bar{a}_0} + c_{\bar{a}_1 a_0} + c_{\bar{a}_1 \bar{a}_0} = x + d$

Need four cofactors:
- $c_{a_1 a_0} = x + d$
- $c_{a_1 \bar{a}_0} = xd$
- $c_{\bar{a}_1 a_0} = 0$
- $c_{\bar{a}_1 \bar{a}_0} = 0$

# Quantification Example




$$c = a_1(a_0x + a_0d + xd)$$

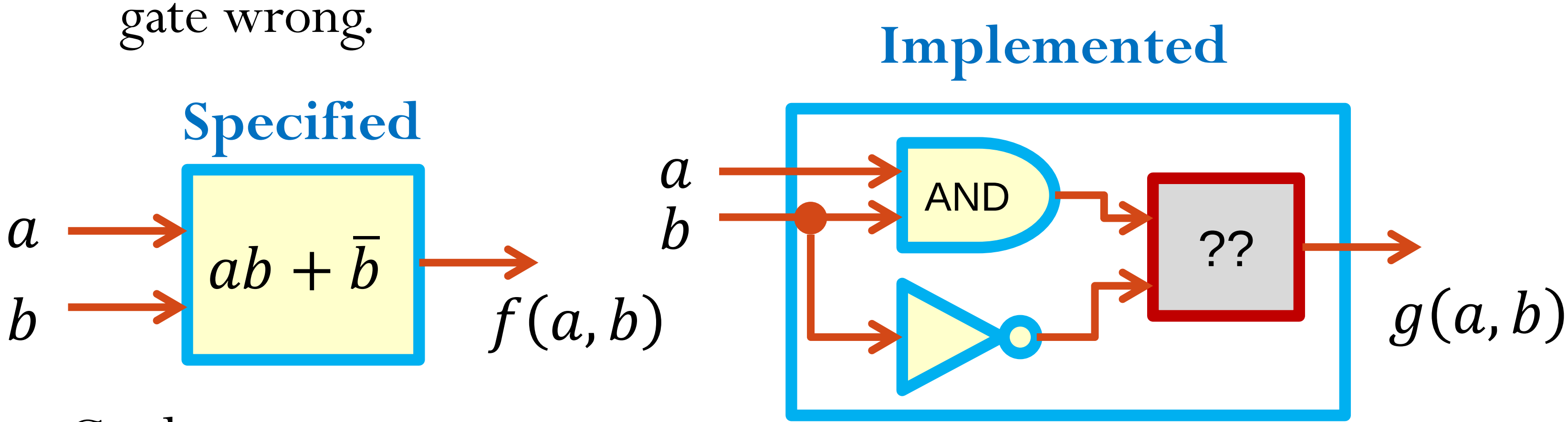
- $(\forall a_1 a_0\ c)(x, d) = 0$
  - Make sense: **No** values of $x$ and $d$ that make $c = 1$ **independent of** $a_1$ and $a_0$
- $(\exists a_1 a_0\ c)(x, d) = x + d$
  - Make sense: If **at least one** of $x$ and $d = 1$, then **there exist** $a_1$ and $a_0$ that let $c = 1$.

# Outline

- Cofactor and Shannon Expansion
- Combinations of Cofactors
  - Boolean Difference
  - Quantification
  - Quantification Application: Network Repair
- Application of Computational Boolean Algebra: Tautology Checking

# Quantification Application: Network Repair

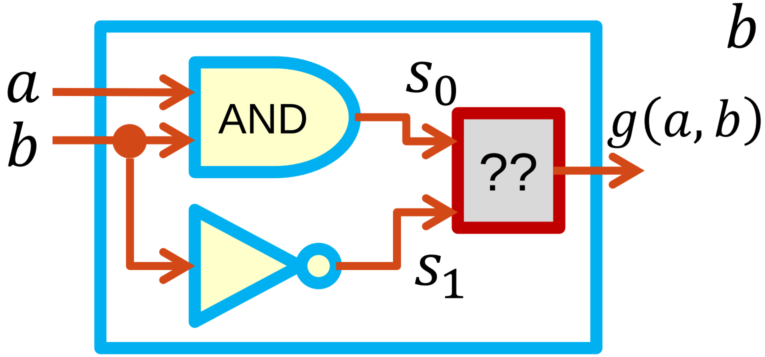
- Suppose that some one specified a logic block for you to implement: $f(a,b) = ab + \bar{b}$
  - …but you implemented it **wrong**: in particular, you got ONE gate wrong.

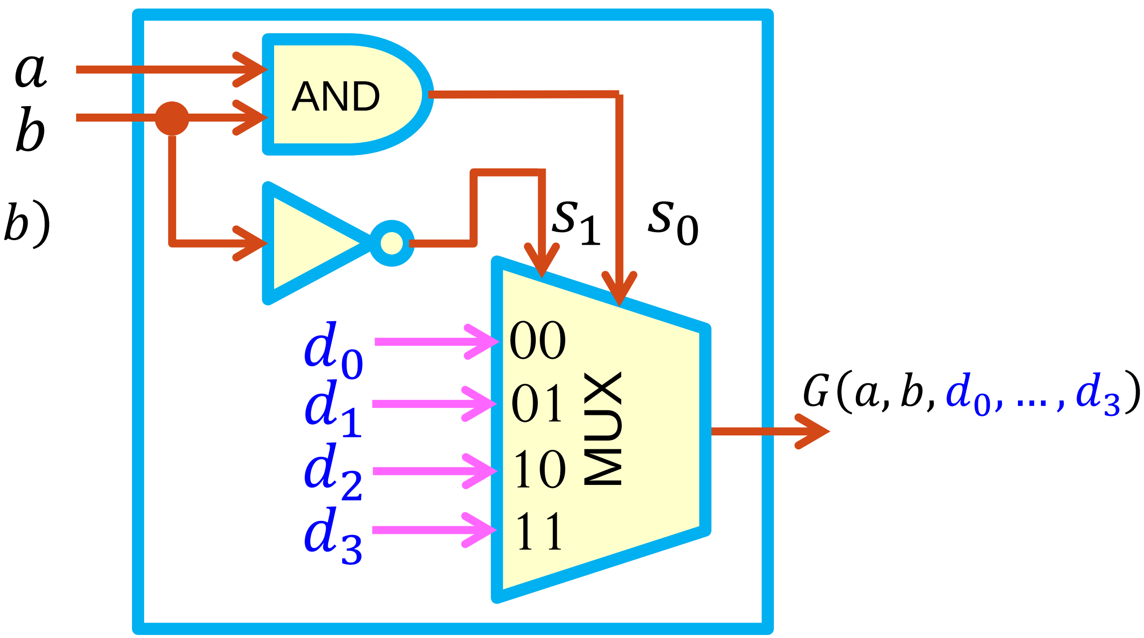



- <u>Goal</u>
  - Can we deduce how precisely to **change this gate** to restore correct function?
  - Go with this very trivial test case to see how mechanics work…

# Network Repair

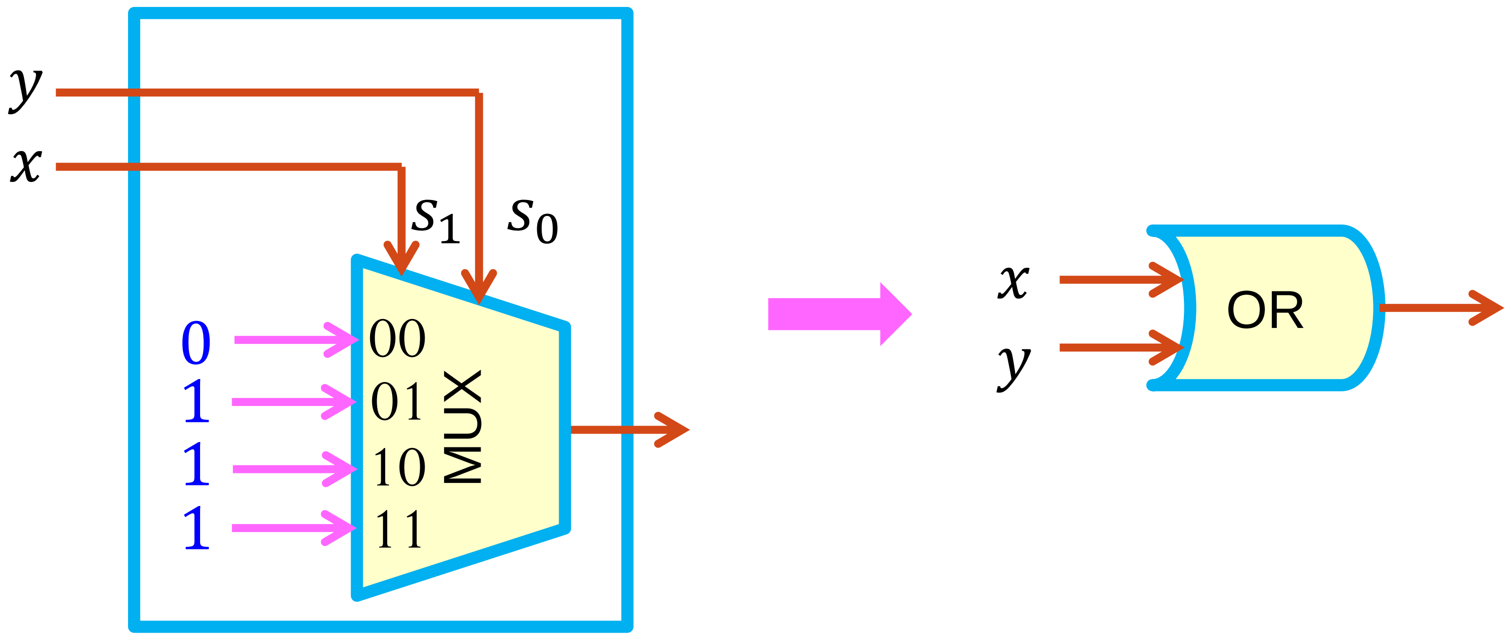
- Clever trick: Replace our suspect gate by a 4-to-1 MUX with 4 arbitrary new variables $d_0, d_1, d_2, d_3$.
  - By cleverly assigning values to $d_0, d_1, d_2, d_3$, we can **fake** any gate.
  - Question is: what are the right values of $d_i$'s so $g$ is repaired, i.e., $g = f$?



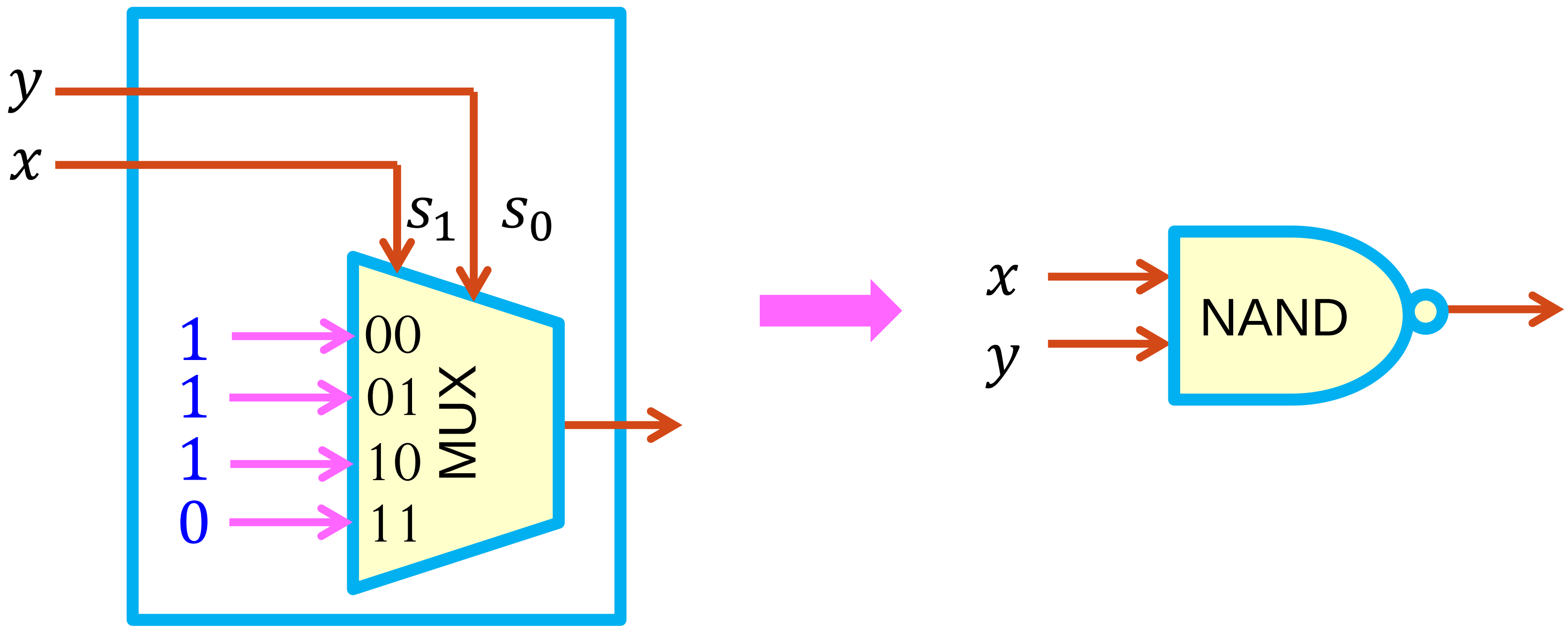
# Aside: Faking a Gate with a MUX

- You can do **any** function of 2 variables with one 4-to-1 multiplexor (MUX).



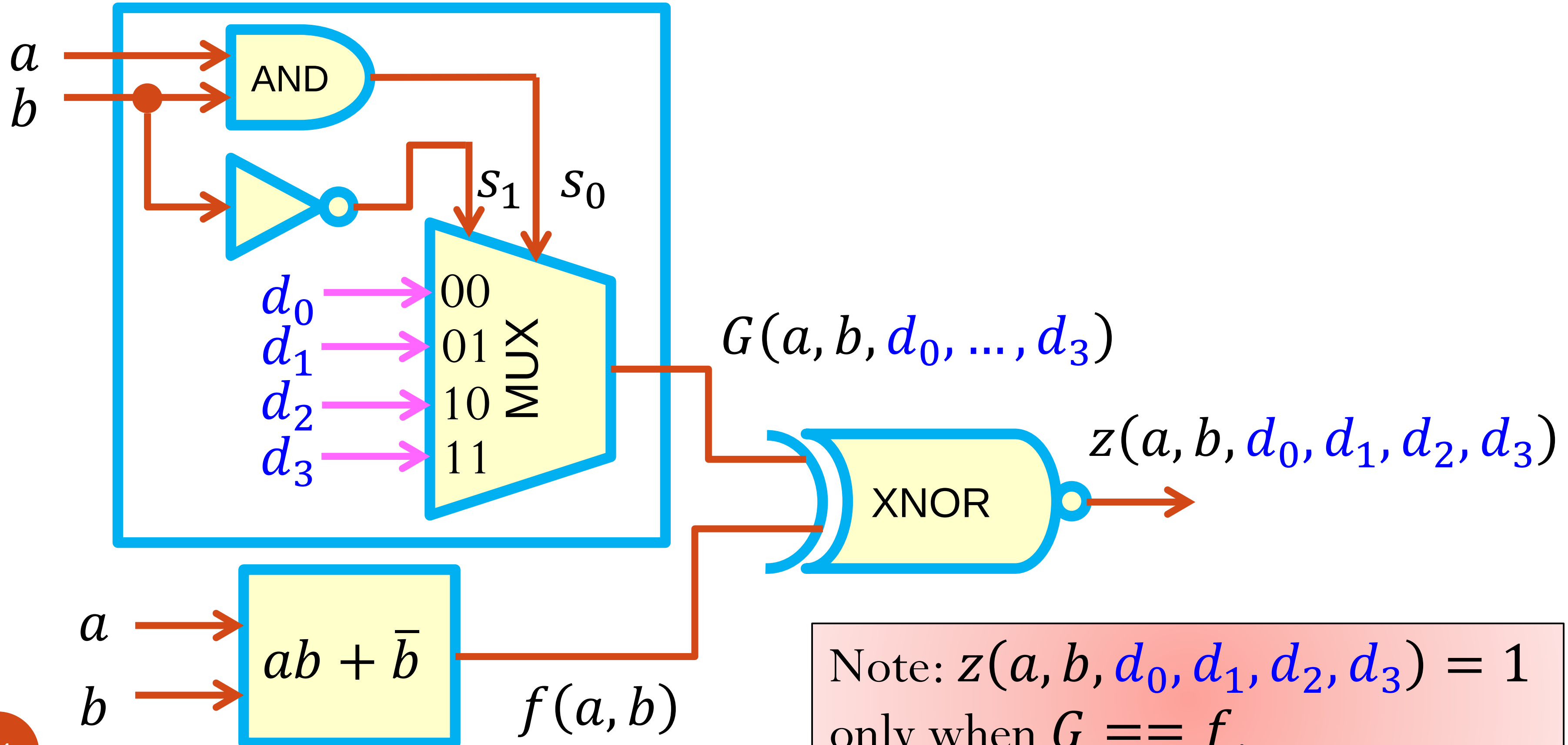
# Aside: Faking a Gate with a MUX

- You can do **any** function of 2 variables with one 4-to-1 multiplexor (MUX).



# Network Repair: Using Quantification

- Next trick: XNOR $G(a, b, d_0, \dots, d_3)$ with the specification $f(a, b)$.




Note: $z(a, b, d_0, d_1, d_2, d_3) = 1$ only when $G == f$.

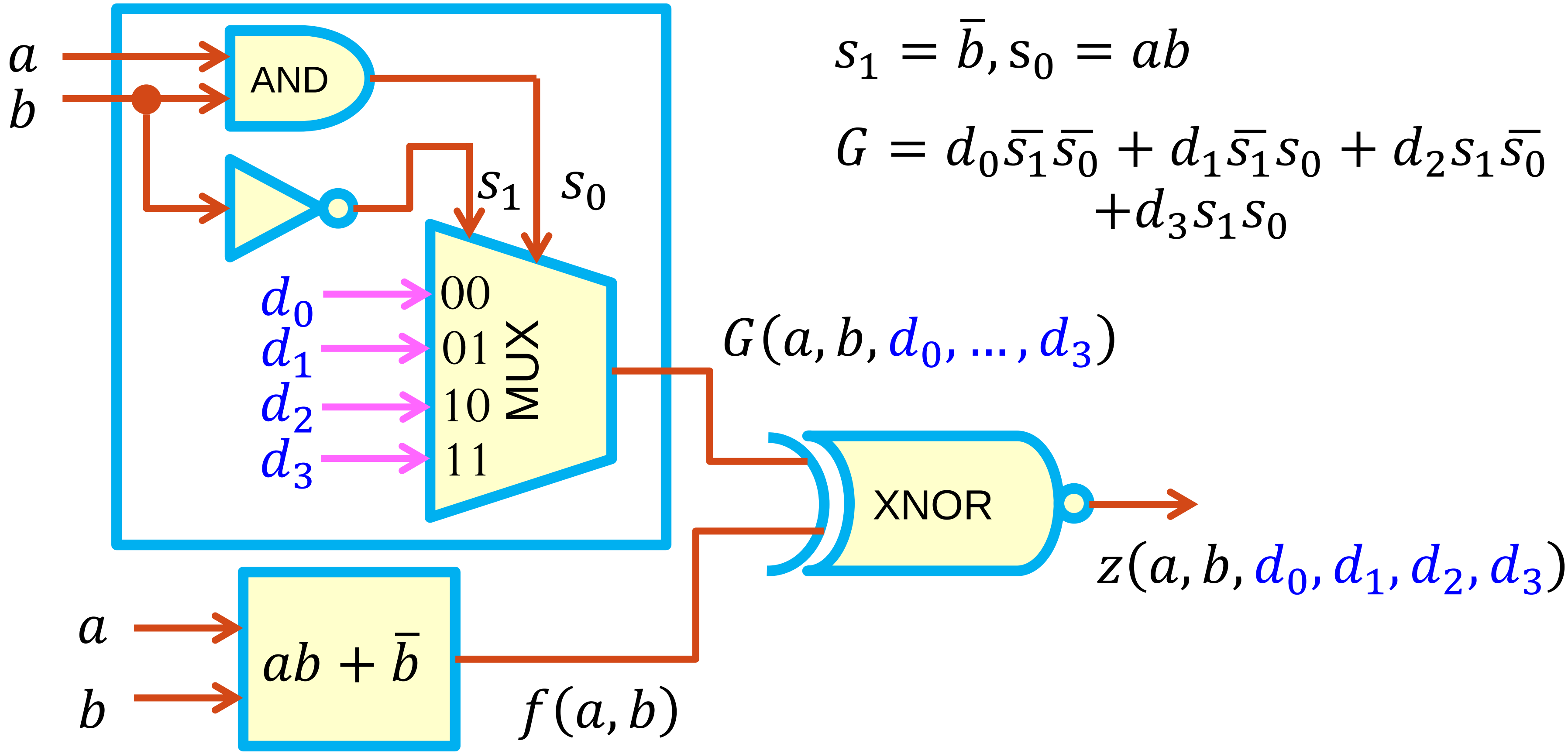
# Using Quantification

- What do we need?
  - Values of $d_0, d_1, d_2, d_3$ that make $z = 1$ **for all** possible values of inputs $a, b$.
  - They are values of $d_0, d_1, d_2, d_3$ that let
$$(\forall ab\ z)(d_0, d_1, d_2, d_3) = 1$$
  - The above equation is **universal quantification** of function $z$ with respect to $a, b$!
  - Any pattern of $(d_0, d_1, d_2, d_3)$ that makes
$$(\forall ab\ z)(d_0, d_1, d_2, d_3) = 1$$
will do the repair!

# Network Repair via Quantification

$$s_1 = \bar{b}, s_0 = ab$$

$$G = d_0\bar{s_1}\bar{s_0} + d_1\bar{s_1}s_0 + d_2 s_1\bar{s_0} + d_3 s_1 s_0$$



# Network Repair via Quantification

- As a result
  - $G(a,b,d_0,\dots,d_3) = d_0\bar{a}b + d_1ab + d_2\bar{b}$
  - $f(a,b) = ab + \bar{b}$
  - $z(a,b,d_0,\dots,d_3) = G(a,b,d_0,\dots,d_3)\overline{\oplus}f(a,b)$

- We want to get

$$(\forall ab\ z)(d_0,d_1,d_2,d_3)$$
$$= z_{\bar{a}\bar{b}} \cdot z_{\bar{a}b} \cdot z_{a\bar{b}} \cdot z_{ab}$$

- To simplify the computation, we will apply the relation:

$$z_{ab} = G_{ab}\overline{\oplus}f_{ab}$$

# Network Repair via Quantification

- $G(a,b,d_0,\dots,d_3) = d_0\bar{a}b + d_1 ab + d_2\bar{b}$
- $f(a,b) = ab + \bar{b}$
- $z(a,b,d_0,\dots,d_3) = G(a,b,d_0,\dots,d_3)\overline{\oplus}f(a,b)$

- $z_{\bar{a}\bar{b}} = G_{\bar{a}\bar{b}}\overline{\oplus}f_{\bar{a}\bar{b}} = d_2\overline{\oplus}1 = d_2$
- $z_{\bar{a}b} = G_{\bar{a}b}\overline{\oplus}f_{\bar{a}b} = d_0\overline{\oplus}0 = \overline{d_0}$
- $z_{a\bar{b}} = G_{a\bar{b}}\overline{\oplus}f_{a\bar{b}} = d_2\overline{\oplus}1 = d_2$
- $z_{ab} = G_{ab}\overline{\oplus}f_{ab} = d_1\overline{\oplus}1 = d_1$

- $(\forall ab\, z)(d_0,d_1,d_2,d_3) = z_{\bar{a}\bar{b}} \cdot z_{\bar{a}b} \cdot z_{a\bar{b}} \cdot z_{ab} = \overline{d_0}d_1 d_2$

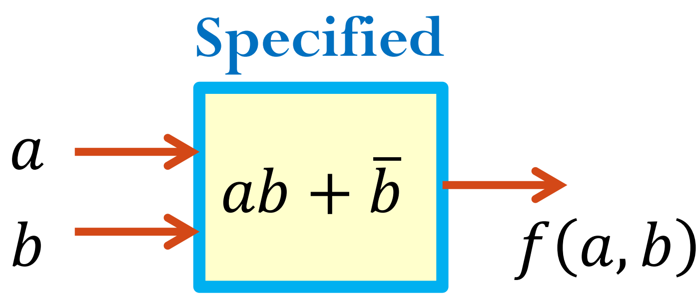
# Network Repair via Quantification

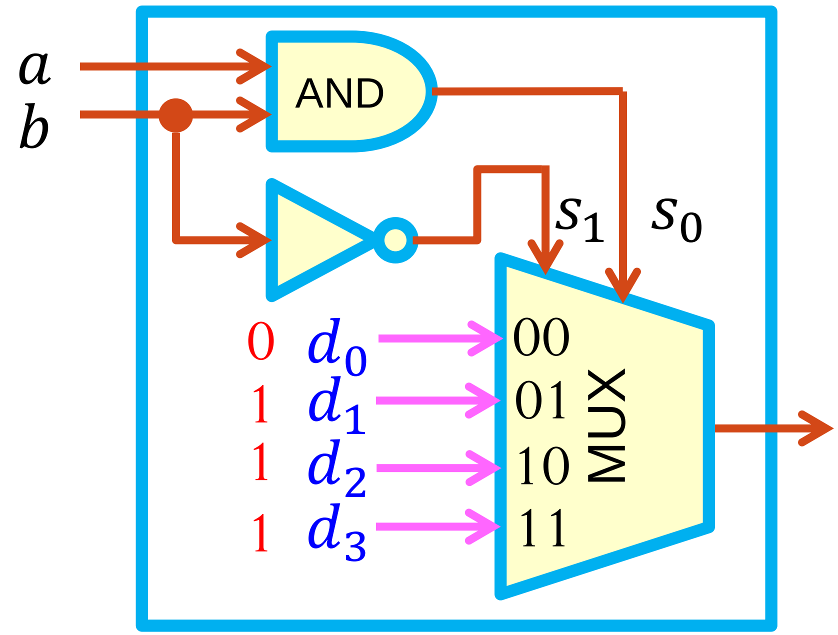
- Finally, we obtain $(\forall ab\ z)(d_0, d_1, d_2, d_3) = \overline{d_0} d_1 d_2$

- To repair, we should find values of $d_0, d_1, d_2, d_3$ so that
$$(\forall ab\ z)(d_0, d_1, d_2, d_3) = 1$$
  - Not hard: $d_0 = 0, d_1 = 1, d_2 = 1, d_3 = X(\text{don}'\text{t care})$

# Network Repair

- Does $d_0 = 0, d_1 = 1, d_2 = 1, d_3 = X$ work?
  - Case 1: $d_0 = 0, d_1 = 1, d_2 = 1, d_3 = 1$

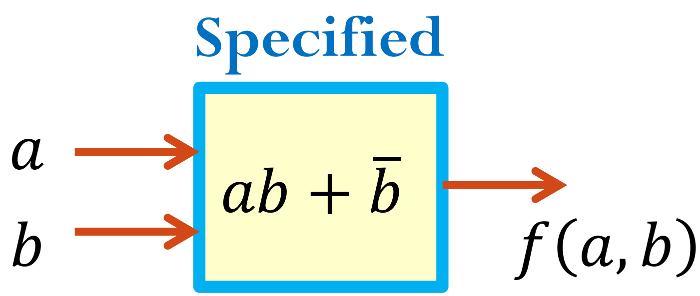


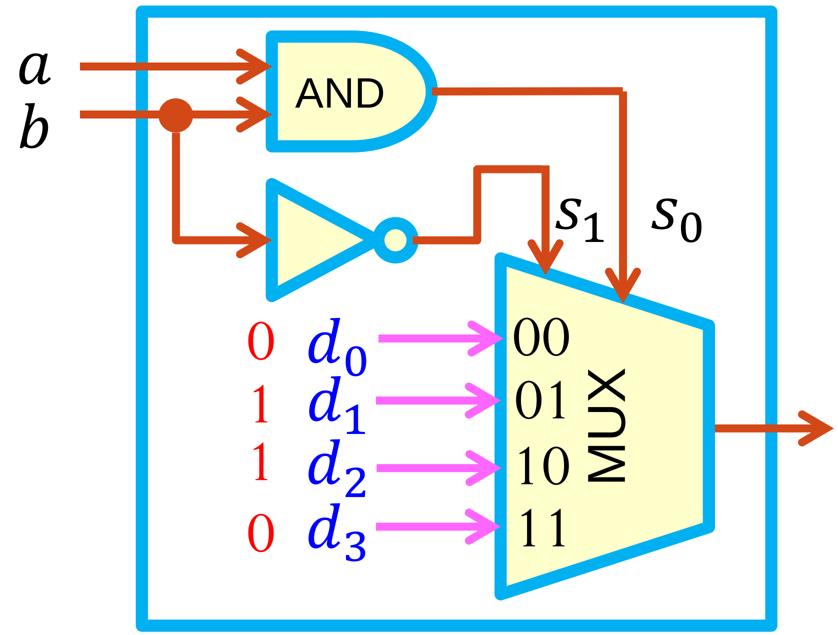




MUX is an OR gate. Expected!

# Network Repair

- Does $d_0 = 0, d_1 = 1, d_2 = 1, d_3 = X$ work?
  - Case 2: $d_0 = 0, d_1 = 1, d_2 = 1, d_3 = 0$




MUX is an XOR gate.
Unexpected but works!

# Network Repair: Summary

- This example is **tiny**…
  - But in a real example, you have a big network – 100 inputs, 50,000 gates.
  - When the design doesn't work, it is a major hassle to go through the design to fix it.
  - This gives a mechanical procedure to answer: Can we change 1 gate to repair?
- What we haven't seen yet: **Computation strategy** to mechanically find inputs to make

$$(\forall ab\ z)(d_0, d_1, d_2, d_3) = 1$$

  - This computation is called **Boolean Satisfiability (SAT)**.
  - We will see how to solve Boolean SAT problem efficiently later.

# Outline

- Cofactor and Shannon Expansion
- Combinations of Cofactors
  - Boolean Difference
  - Quantification
  - Quantification Application: Network Repair
- Application of Computational Boolean Algebra: Tautology Checking

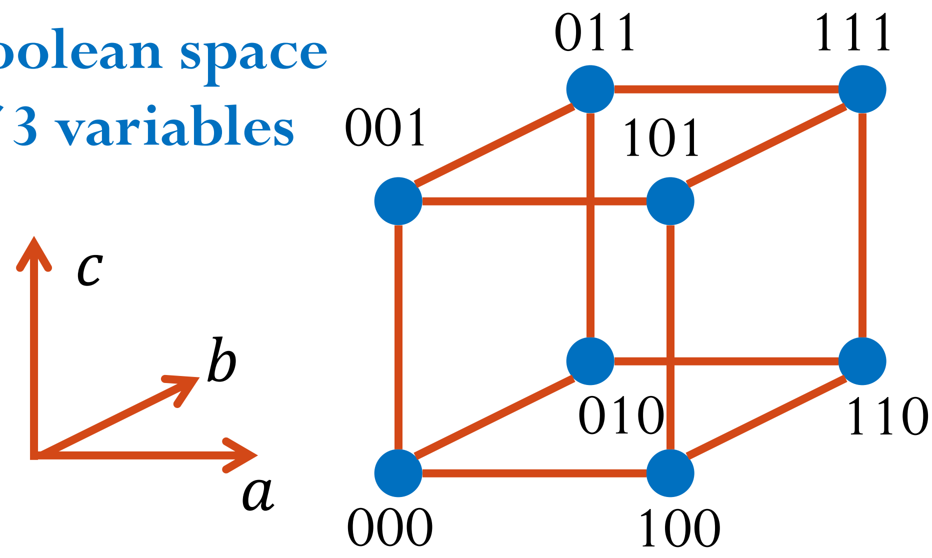
# Important Example of Computation: Tautology Checking

- **Tautology**: a Boolean function is 1 for every input.
- We are going to look at how to do **tautology checking**, i.e. whether a Boolean function $f$ is 1 for every input.
- Why study this problem?
  - To show a **representation**, i.e., **a data structure**, for a Boolean function $f$.
  - To show an important **computational strategy**: recursion
- How **hard** is this problem?
  - Very, very hard!
  - What happens if you are given a sum-of-product expression with 50 variables and 800 products?

# Start with: Representation

- We use a simple representation scheme for functions
  - Represent a function as a set of OR'ed product terms, i.e., **a sum of products (SOP)**.
- Each product term is also called a **cube**, e.g., $a\bar{b}c$ is a cube.
  - Why call it cube?

**Boolean space of 3 variables**

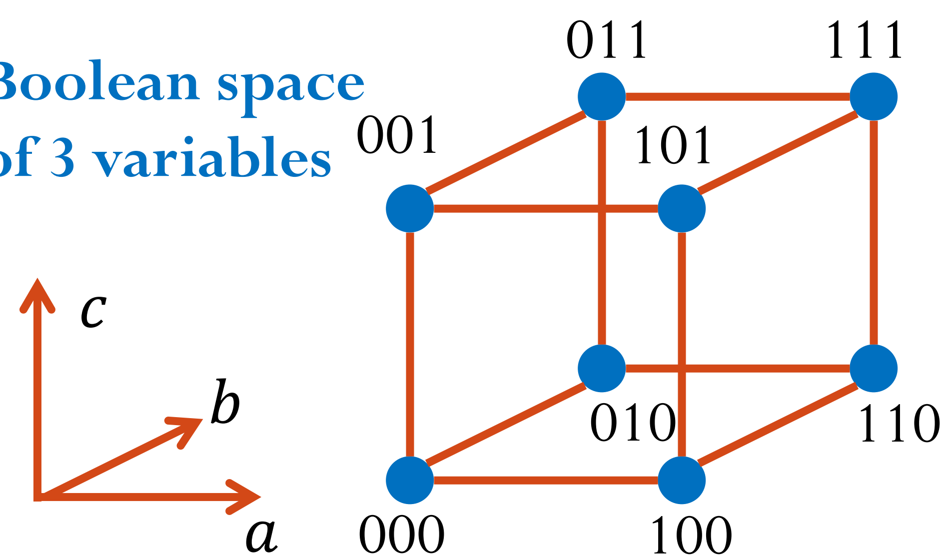



How does $f$ as a product terms look like?

# Properties of Cubes

- In what follows, we refer to **product term** as **cube**.
- For each variable $x$, $x$ and $\bar{x}$ do not appear simultaneously.
  - However, for each variable $x$, it is possible that none of $x$ and $\bar{x}$ appears in the cube.
  - If **for each** variable $x$, one of $x$ and $\bar{x}$ appears in the cube. The cube is a **minterm**.
- The number of vertices in a cube is $2^k$.

**Boolean space of 3 variables**



# Positional Cube Notation (PCN)

- We represent a cube using <u>Positional Cube Notation (PCN)</u>.
  - One slot per variable.
  - In slot for variable $x$:
    - put "1", if cube has $x$ in it;
    - put "0", if cube has $\bar{x}$ in it;
    - put "−", if cube has no $x$ or $\bar{x}$ in it.

- Example: In a Boolean space on three variables $a, b, c$

| | | $a$ | $b$ | $c$ | |
|---|---|---|---|---|---|
| • $\bar{b}$: | [ | − | 0 | − | ] |
| • $a\bar{c}$: | [ | 1 | − | 0 | ] |

# Positional Cube Notation (PCN)

- To represent cube in program, we need to encode "`1`", "`0`", and "`−`".
  - We need at least two bits to encode three values.
  - One encoding: "01" to encode "`1`"; "10" to encode "`0`"; "11" to encode "`−`".

- Example: In a Boolean space on three variables $a, b, c$

$$\begin{array}{lccc}
 & \begin{array}{ccc} a & b & c \end{array} & & \begin{array}{ccc} a & b & c \end{array} \\
\bullet\ \bar{b}: & [\ -\ \ 0\ \ -\ ] & \Rightarrow & [11\ \ 10\ \ 11] \\
\bullet\ a\bar{c}: & [\ 1\ \ -\ \ 0\ ] & \Rightarrow & [01\ \ 11\ \ 10]
\end{array}$$

# PCN Cube List

- A **sum-of-products (SOP)** expression of a Boolean function is also called as a **cover of cubes**.
  - We present a **cover of cubes** using **a list of cubes** in **positional cube notation**.

- Example: $f = \bar{a} + bc + ab$

$$\begin{array}{cc} & \begin{array}{ccc} a & b & c \end{array} \\ \bar{a} & [10 \ \ 11 \ \ 11] \\ bc & [11 \ \ 01 \ \ 01] \\ ab & [01 \ \ 01 \ \ 11] \end{array}$$

# Tautology Checking

- How do we approach **tautology checking** as a **computation**?
  - Input: a list of cube in PCN representing an SOP expression of $f$
  - Output: Yes, when $f$ is always 1; No, when $f$ is not always 1.

- Cofactors to rescue
  - Great result: $f$ is a tautology if and only if $f_x$ and $f_{\bar{x}}$ are both tautologies.
  - This makes sense:
    - If function $f = 1$, then cofactors both obviously $= 1$.
    - If both cofactors $= 1$, then $f = x \cdot f_x + \bar{x} \cdot f_{\bar{x}} = x + \bar{x} = 1$

# Recursive Tautology Checking

- Suggests a **recursive computation** strategy:
  - If you cannot tell immediately that f is a tautology, go try to see if each cofactor is a tautology.

$$f = 1\text{??} \Rightarrow \text{Split on } x: \ (f_x = 1 \text{ AND } f_{\overline{x}} = 1)\text{??}$$

$$f_x = 1 \qquad\qquad f_{\overline{x}} = 1$$

- What else do we need?
  - Selection rules: which $x$ is good to pick to split on?
  - Termination rules: how do we know when to stop splitting, so we can answer that the function at this node of tree is tautology or not?
  - Mechanics: how hard is it to actually obtain the cofactors?

# Recursive Cofactoring

- Do mechanics first (easy!). For each cube in the list:
  - If you want **positive** cofactor w.r.t. var $x$, look at $x$ slot in each cube:
    - [… 10 …] ➔ just remove this cube from list, since it is a term with $\bar{x}$.
    - [… 01 …] ➔ just make this slot 11 (don't care), since we will strike $x$ from the product term.
    - [… 11 …] ➔ just leave this alone, since this term doesn't have any $x/\bar{x}$ in it.

# Recursive Cofactoring

- Do mechanics first (easy!). For each cube in the list:
  - If you want **negative** cofactor w.r.t. var $x$, look at $x$ slot in each cube:
    - [… 01 …] ➔ just remove this cube from list, since it is a term with $x$.
    - [… 10 …] ➔ just make this slot 11 (don't care), since we will strike $\bar{x}$ from the product term.
    - [… 11 …] ➔ just leave this alone, since this term doesn't have any $x/\bar{x}$ in it.

# Recursive Cofactoring: Example

| | $f = acd + b\bar{c}$ | $f_a$ | $f_{\bar{c}}$ |
|---|---|---|---|
| $acd$ | [01 11 01 01] | [11 11 01 01] | |
| $b\bar{c}$ | [11 01 10 11] | [11 01 10 11] | [11 01 11 11] |

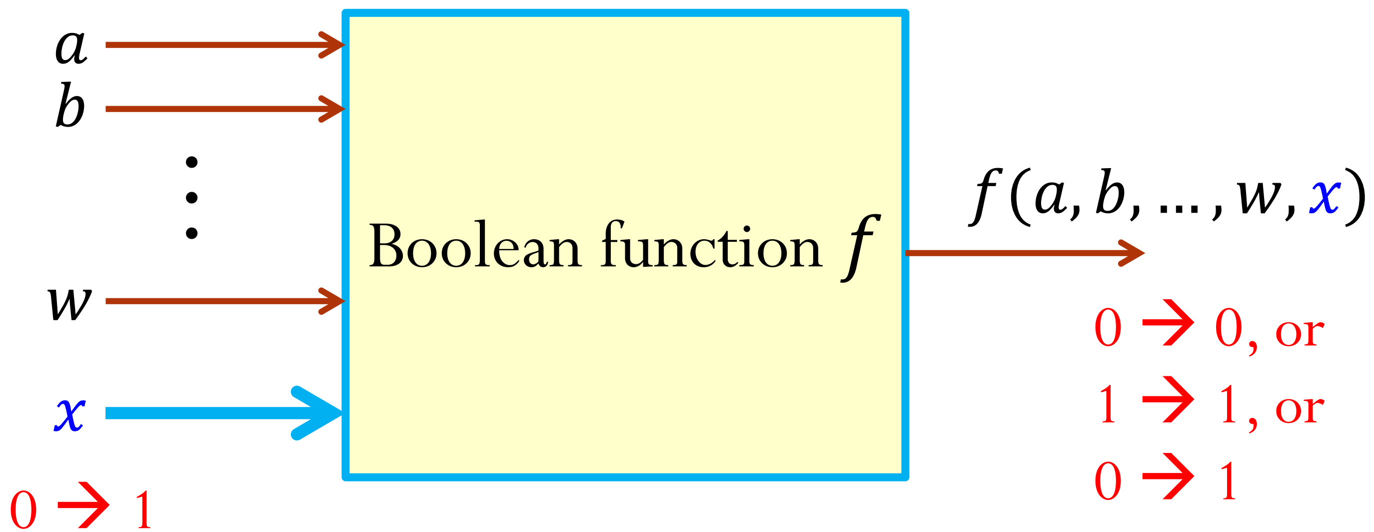
# Unate Functions

- Selection / termination, another trick: **Unate functions**
  - Special class of Boolean functions
  - $f$ is **positive unate** in var $x$: if changing x 0→1 but keeping other variables constant (no matter what values they are), keeps $f$ constant or makes $f$ change 0→1.
  - $f$ is **negative unate** in var $x$: if changing x 0→1 but keeping other variables constant (no matter what values they are), keeps $f$ constant or makes $f$ change1→0.
  - $f$ is **unate** in var $x$ if $f$ is either **positive unate** in var $x$ or **negative unate** in var $x$.
- E.g., $f = ab$ is positive unate in $a$
- E.g., $f = \bar{a}b + a\bar{b}$ is not unate in $a$.

# Unate Functions

- Analogous to **monotonic continuous functions**.
  - A **monotonically non-decreasing** function: whenever $x_2 \geq x_1$, we have $f(x_2) \geq f(x_1)$.

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



# Checking Unateness

- How to check unateness?
  - Not easy if $f$ is represented using truth table.
  - Very easy if $f$ is represented as an SOP.
- Suppose that $f$ is represented as an SOP. Then $f$ is **unate** in var $x$ if …
  - For all the cubes that contain var $x$, $x$ appears in exactly **one polarity**, either all true, or all complemented.
    - E.g., $f = ab + a\bar{c}d + \bar{c}d\bar{e}$ is unate in $a, b, c, d, e$.
  - A **sufficient** condition only!! If not satisfied, may be either unate or not unate.
    - E.g., $f = xy + \bar{x}y + \bar{x}y\bar{z} + \bar{z}$ is unate in $y, z$, but may or may NOT be unate in $x$. (Actually, it is unate in $x$!)

# Unate Functions

- If **<u>for each</u>** var $x$, $f$ is either positive or negative unate in that var $x$, then $f$ is said to be **unate**.
  - If **<u>for each</u>** var $x$, $f$ is positive unate in that var $x$, then $f$ is said to be **positive unate**.
  - If **<u>for each</u>** var $x$, $f$ is negative unate in that var $x$, then $f$ is said to be **negative unate**.
- Function that is not unate is called **binate**.

- E.g., $f = ab + a\bar{c}d + \bar{c}d\bar{e}$ is unate.
- E.g., $f = x\bar{y} + \bar{x}y$ is NOT unate.

# Unate Cube-List

- A **sufficient condition** on cube list: A **<u>cube-list</u>** is **unate** if <u>for each</u> var $x$ and <u>for all</u> the cubes that contain var $x$, $x$ only appears in one polarity, not both.
- Easier to see if draw the cube-list vertically.

$$a + bc + ac$$

$$\begin{array}{ll} a & [01\ \cancel{11}\ \cancel{11}] \\ bc & [\cancel{11}\ 01\ \ 01] \\ ac & [01\ \cancel{11}\ 01] \end{array}$$

Unate cube list

$$a + \bar{b}c + bc$$

$$\begin{array}{ll} a & [01\ \cancel{11}\ \cancel{11}] \\ \bar{b}c & [\cancel{11}\ 10\ \ 01] \\ bc & [\cancel{11}\ 01\ \ 01] \end{array}$$

Cannot tell unateness

- A unate cube-list corresponds to a unate function

## Using Unate Functions in Tautology Checking

- It is <u>very easy</u> to check a **unate** cube-list for **tautology**:
  - Unate cube-list for $f$ is tautology iff it contains a cube whose elements are all don't care: [11 11 … 11].
  - Question: what exactly is [11 11 … 11] as a product term?
- This result actually makes sense…
  - If without [11 11 … 11], then the SOP looks like $a + a\overline{b} + \overline{b}c$
  - It will be 0 for value which lets the variable be 0 (i.e., $a = 0, b = 1, c = 0$).

# Termination Rules Using Unateness

- If we have a **unate** cube-list, we can check for tautology directly.
  - Rule 1: The function is a **tautology** if cube-list has an all-don't-care cube [11 11 … 11].
  - Rule 2: The function is **NOT tautology** if cube-list does not have any all-don't-care cube [11 11 … 11].

- There are some more possible termination rules. For example:
  - Rule 3: The function is **tautology** if cube list has **single var cube** that appears in <u>both polarities</u>.
    - Why? function $= x + \bar{x} +$ stuff $= 1$

# Selection Rule

- We can't use easy termination rules <u>unless</u> cube-list is unate
- Selection rule…? Pick splitting var to make unate cofactors
  - Strategy: pick "most non-unate (binate)" var as split var
  - Pick binate var with the **most** product terms dependent on it
    - Why? A product independent of a var is duplicated twice
  - If a tie, pick var with minimum |#true_var – #compl_var|
    - Why? Left subtree and right subtree are balanced

| x | y | z | w |
|---|---|---|---|
| 01 | 01 | 01 | 01 |
| 10 | 11 | 01 | 01 |
| 10 | 11 | 11 | 10 |
| 01 | 01 | 11 | 01 |

✓ x: binate, in 4 cubes, |true-compl| = |2-2|=0

y: unate

z: unate

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

# Recursive Tautology Checking: Algorithm

```
tautology(f represented as cubelist) {
   /* check if we can terminate recursion */
   if (f is unate)  {
      if (f has all-don't-care cube) return 1
      else return 0
   }
   else if (any other termination rules, like rule 3, work?) {
      return the appropriate value (1 or 0)
   }
   else {    /* cannot tell -- find splitting variable */
      x = most binate variable in f
      return (  tautology( f(x=1) ) && tautology( f(x=0) ) )
   }
}
```

# Recursive Tautology Checking

## Example

- Example: $f = ab + ac + a\bar{b}\bar{c} + \bar{a}$

$f$

| a | b | c |
|---|---|---|
| 01 | 01 | 11 |
| 01 | 11 | 01 |
| 01 | 10 | 10 |
| 10 | 11 | 11 |

- $f$ unate?
  - No
- Which var to pick?
  - Most binate var: $a$

$f_a$

| a | b | c |
|---|---|---|
| 11 | 01 | 11 |
| 11 | 11 | 01 |
| 11 | 10 | 10 |
| ---- | ---- | ---- |

- $f_a$ unate?
  - No
  - Need to further split

$f_{\bar{a}}$

| a | b | c |
|---|---|---|
| ---- | ---- | ---- |
| ---- | ---- | ---- |
| ---- | ---- | ---- |
| 11 | 11 | 11 |

- $f_{\bar{a}}$ unate?
  - Yes!
- Tautology?
  - Yes!
  - Terminate!

# Recursive Tautology Checking
## Example

$f_a$

| a | b | c |
|---|---|---|
| 11 | 01 | 11 |
| 11 | 11 | 01 |
| 11 | 10 | 10 |

- $f_a$ unate?
  - No
- Which var to pick?
  - Either $b$ or $c$
  - For example, we pick $b$

$f_{ab}$

| a | b | c |
|---|---|---|
| 11 | 11 | 11 |
| 11 | 11 | 01 |
| ---- | ---- | ---- |

- $f_{ab}$ unate?
  - Yes!
- Tautology?
  - Yes! (contain all-don't-care cube)
  - Terminate!

$f_{a\bar{b}}$

| a | b | c |
|---|---|---|
| ---- | ---- | ---- |
| 11 | 11 | 01 |
| 11 | 11 | 10 |

- $f_{a\bar{b}}$ unate?
  - No
- Can we terminate?
  - Yes, due to Rule 3

# Recursive Tautology Checking

## Example

- The recursion tree we finally get is

$$
\begin{tikzcd}
 & & f \arrow[dl] \arrow[dr] & \\
 & f_a \arrow[dl] \arrow[dr] & & f_{\bar{a}} = 1 \\
f_{ab} = 1 & & f_{a\bar{b}} = 1 &
\end{tikzcd}
$$

- The tree has tautologies at all leaves, so $f$ is tautology.

- Note: if any leaf is NOT tautology, $f$ is NOT!

# Recursive Tautology Checking
## Summary

- This strategy is so general and useful it has a name: **Unate Recursive Paradigm (URP)**.

- Again, we see that **cofactors** are important and useful.

- **Representations (data structures)** for Boolean functions are critical.
  - Truth tables, Karnaugh maps, Boolean expressions cannot be manipulated by software.
  - See one real representation: **positional cube notation**