《芯片设计自动化与智能优化》Boolean Algebra

The slides are based on Prof. Weikang Qian’s lecture notes at SJTU and Prof. Rob Rutenbar’s lecture notes at UIUC

Yibo Lin

Peking University

Outline

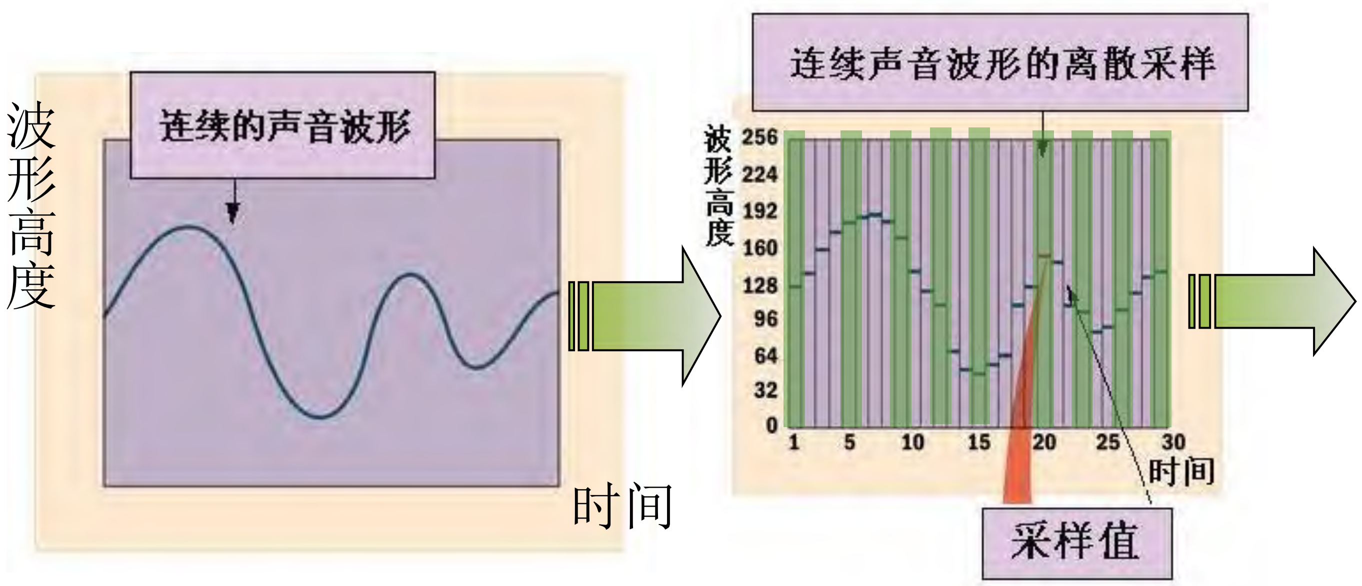
- Computational Boolean Algebra
  - Cofactors
  - Shannon expansion
  - Combinations of cofactors
  - Quantification
  - ~~Tautology checking~~
  - ~~Unateness~~
- Tautology and circuit representation
  - ~~Recursive Tautology checking~~
  - Represent combinational circuits with DAG
  - Traversal combinational circuits: topological sorting
  - Circuit netlist format

Information Encoded to Boolean representation

图像编码

真彩色（24位）

声音编码



波形高度值

采样点序号（时间）

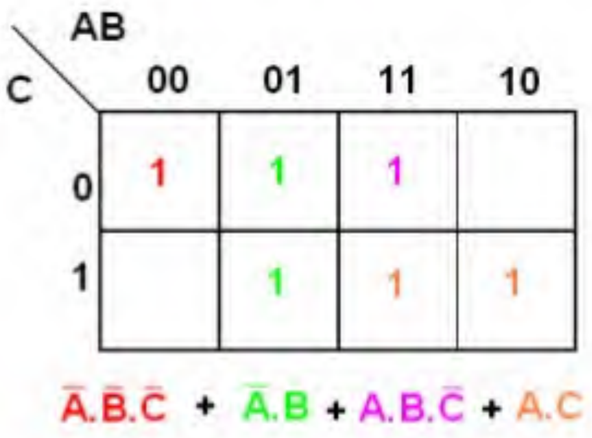
采样点序号	波形编码
1	10000000
2	11001010
3	11100101
4	11101100
5	11111001
…	…
30	10000011

Boolean Algebra

- Karnaugh maps
  - Given a truth table, simplify the Boolean expression

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$Z = f(A, B, C) = \overline{A}.\overline{B}.\overline{C} + \overline{A}.B + A.B.\overline{C} + A.C$$



Boolean Algebra

- Karnaugh maps
  - Given a truth table, simplify the Boolean expression
  - **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…

Computational Boolean Algebra

- 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 and terms you need to be able to do this.
  - **Computational strategies**
    - Ways to think about Boolean functions that let them be manipulated by programs.
  - **Interesting applications**
    - When you have new tools, there are some useful new things to do.

Advanced Boolean Algebra – Analogy to Calculus

- In calculus, you can represent complex functions like e^x using simpler functions.
  - If you can only use $1, x, x^2, x^3$, … 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**.
  - $f(x) = f(0) + \frac{f'(0)}{1!} + \frac{f''(0)}{2!} + \frac{f'''(0)}{3!} + \cdots$

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 $F(x_1, x_2, \dots, x_n)$.
- Define **a new function** if we set one of the $x_i = const$
  - $F(x_1, x_2, \dots, x_i = 1, \dots, x_n)$
  - $F(x_1, x_2, \dots, x_i = 0, \dots, x_n)$
- 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 $F(x_1, x_2, \dots, x_i = 1, \dots, x_n)$ as F_{x_i}. We call it **positive cofactor**.
  - We write $F(x_1, x_2, \dots, x_i = 0, \dots, x_n)$ 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?

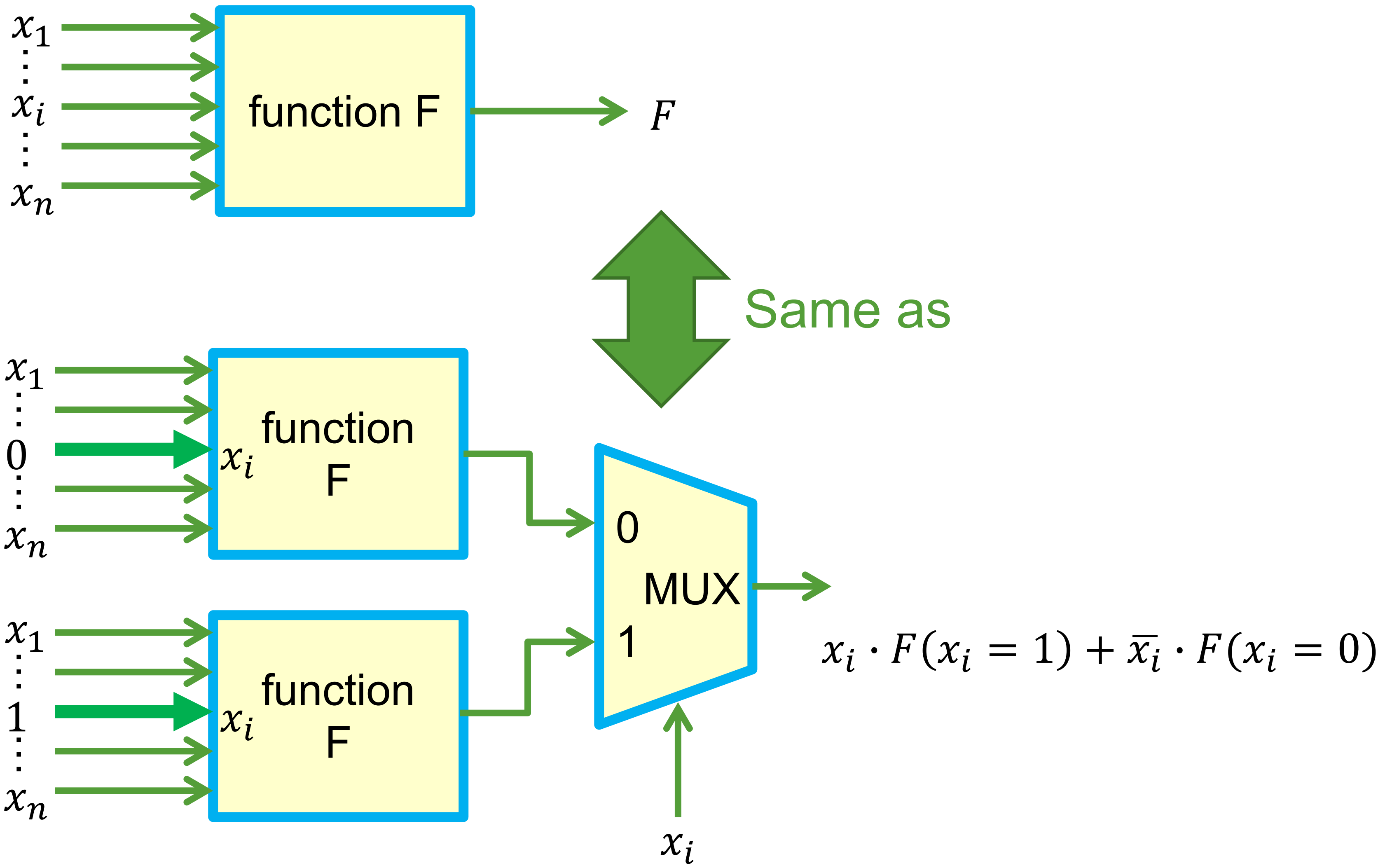
Shannon Expansion Theorem

- 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

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

- 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:

$$\begin{aligned} &F(x, y, z, w) \\ &= xy \cdot F(x = 1, y = 1) + x\bar{y} \cdot F(x = 1, y = 0) \\ &+\bar{x}y \cdot F(x = 0, y = 1) + \bar{x}\bar{y} \cdot F(x = 0, y = 0) \end{aligned}$$

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.

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

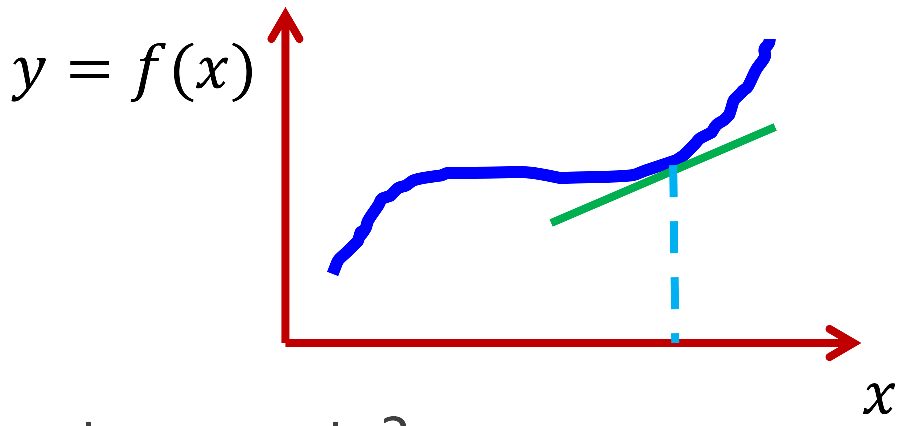
- Cofactors of F and G tell you everything you need to know.
- Complements
  - $(\bar{F})_x = \overline{(F_x)}$
  - In English: cofactor of complement is complement of cofactor.
- Binary Boolean operators
  - $(F \cdot G)_x = F_x \cdot G_x$ cofactor of AND is AND of cofactors
  - $(F + G)_x = F_x + G_x$ 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.

Combinations of Cofactors

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

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

Boolean Derivatives

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

- 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...
- 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...

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

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

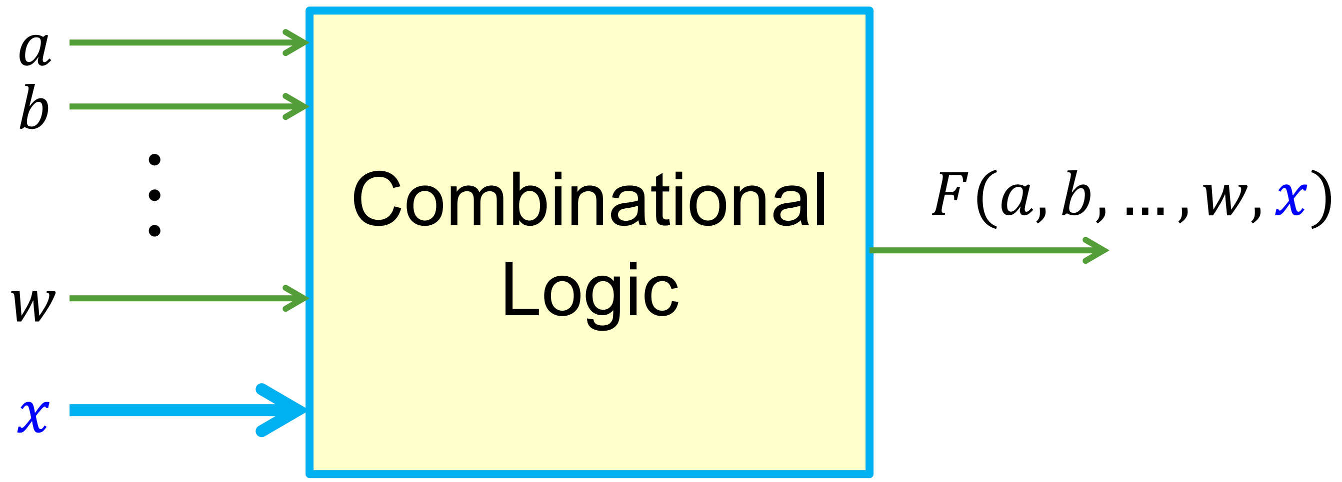
- Why?
  - Because AND and OR on Boolean values do not always behave like **ADDITION** and **MULTIPLICATION** on real numbers.

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

Boolean Difference: Example

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

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

- What 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.

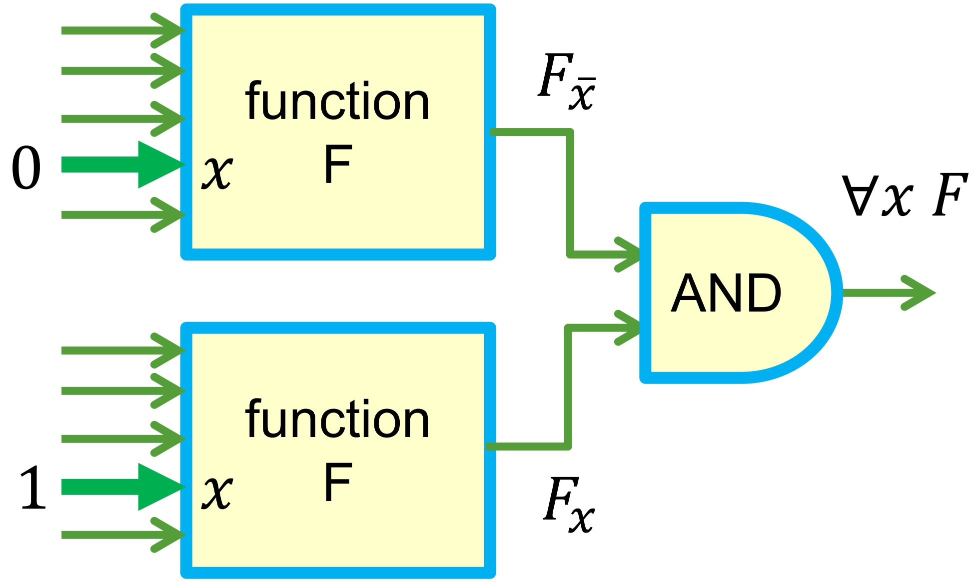
- $\partial f / \partial x$ is another Boolean function, but it does not depend on x!
  - It cannot, because it is made out of cofactors with respect to x, which eliminate all the x and $\bar{x}$ terms by setting them to constants.

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

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.

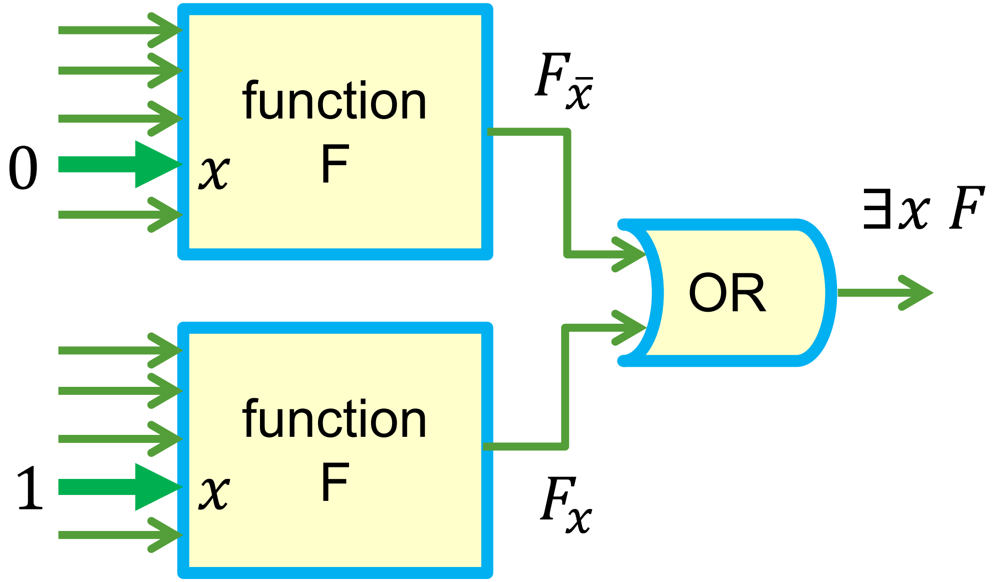


OR of F_x and $F_{\bar{x}}$: Existential Quantification

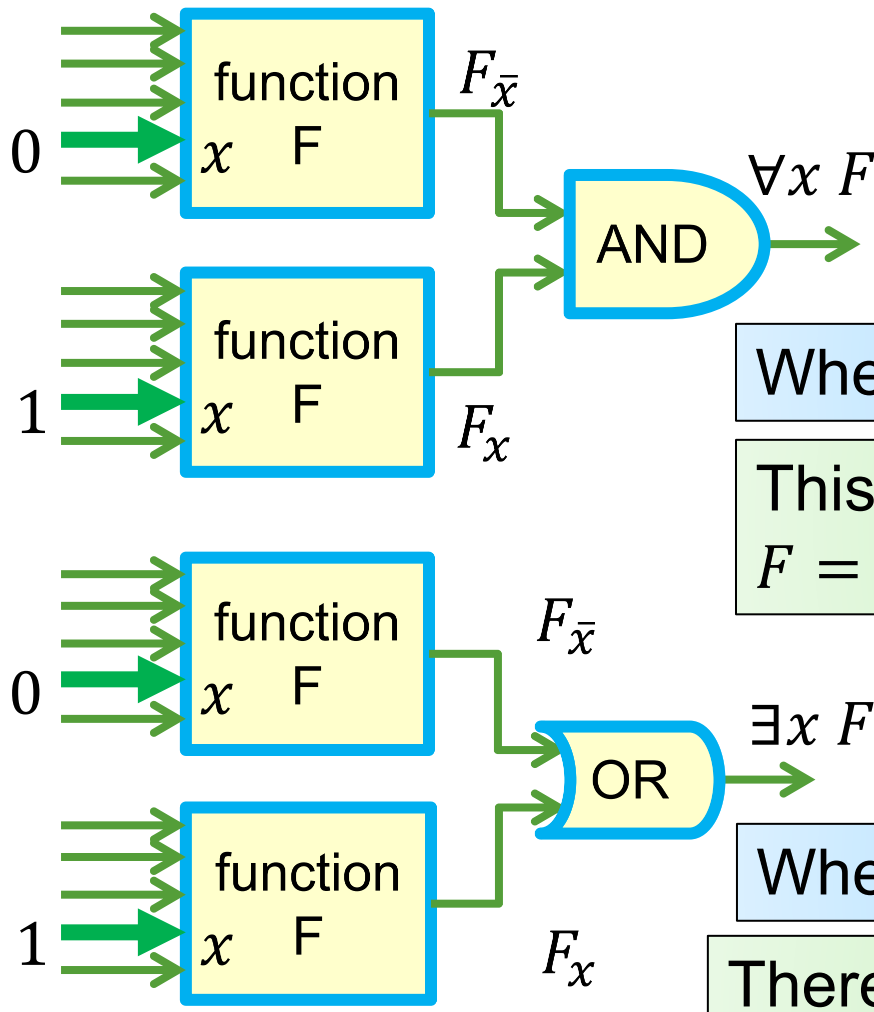
- OR the cofactors: $F_{x_i} + F_{\bar{x}_i}$
  - Name: **Existential Quantification** of function F with respect to variable x_i.
  - Represented as: $(\exists x_i\ F)(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$

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

Both $\forall x_i\ F$ and $\exists x_i\ F$ do not depend on x_i



Quantification Notation Makes Sense…



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

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

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

There exists 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

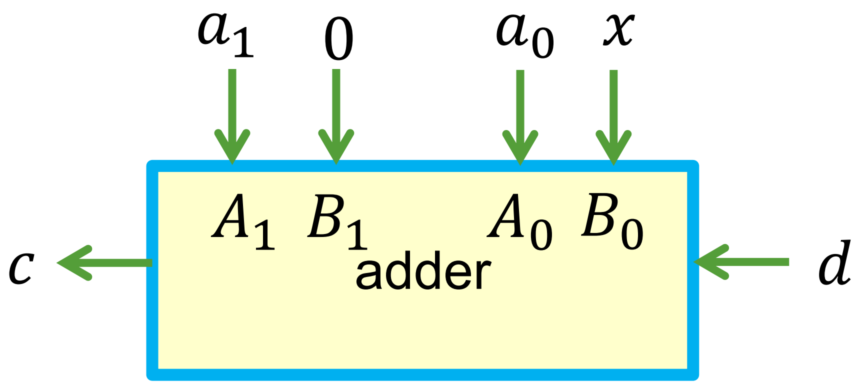
- Like Boolean difference, can do with respect to more than 1 variable
  - Suppose we have $F(x, y, z, w)$.
  - $(\forall xy\ F)(z, w) = \big(\forall x\ (\forall y\ F)\big) = F_{xy} \cdot F_{x\bar{y}} \cdot F_{\bar{x}y} \cdot F_{\bar{x}\bar{y}}$
  - $(\exists xy\ F)(z, w) = \big(\exists x\ (\exists y\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

- Remember:
  - $(\forall x\ F)$, $(\exists x\ F)$, and $\partial F / \partial x$ are all functions.
  - … but they are functions of all the variables **except** x.

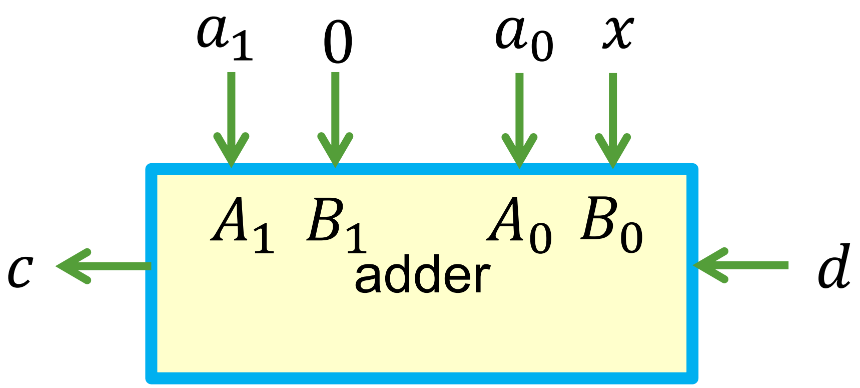
Quantification Example

- Consider the following circuit, it adds x =0 or x =1 to a 2-bit number a_1a_0.
  - It's just a 2-bit adder, but instead of b_1b_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 $(\exists a_1 a_0\ c)(x, d)$?

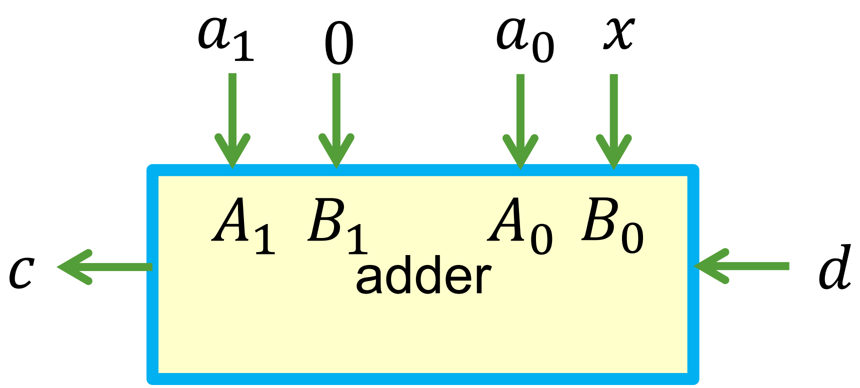


Quantification Example



- What is $(\forall a_1 a_0\ c)(x, d)$?
  - A function of only x and d. Makes a 1 for values of x and d that make a 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. Makes a 1 for values of x and d that make a 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



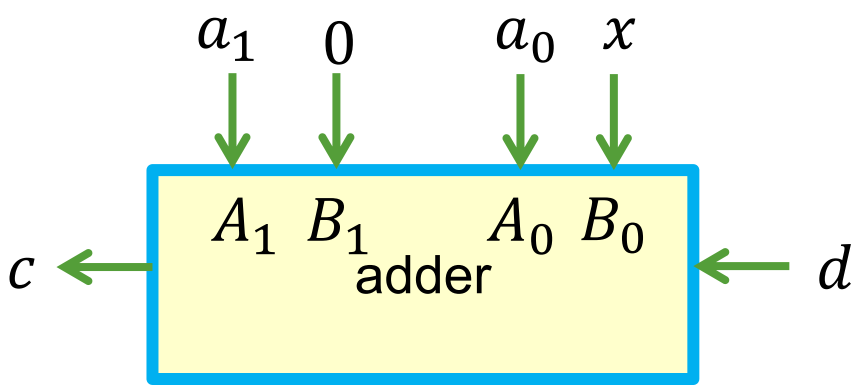
$$c = a_1(a_0x + a_0d + xd)$$

- Compute $(\forall a_1 a_0\ c)(x, d)$
  - $c_{a_1a_0} \cdot c_{a_1\bar{a}_0} \cdot c_{\bar{a}_1a_0} \cdot c_{\bar{a}_1\bar{a}_0}$
    $= 0$
- Compute $(\exists a_1 a_0\ c)(x, d)$
  - $c_{a_1a_0} + c_{a_1\bar{a}_0} + c_{\bar{a}_1a_0} + c_{\bar{a}_1\bar{a}_0}$
    $= x + d$

Need four cofactors:

- $c_{a_1a_0} = x + d$
- $c_{a_1\bar{a}_0} = xd$
- $c_{\bar{a}_1a_0} = 0$
- $c_{\bar{a}_1\bar{a}_0} = 0$

Quantification Example

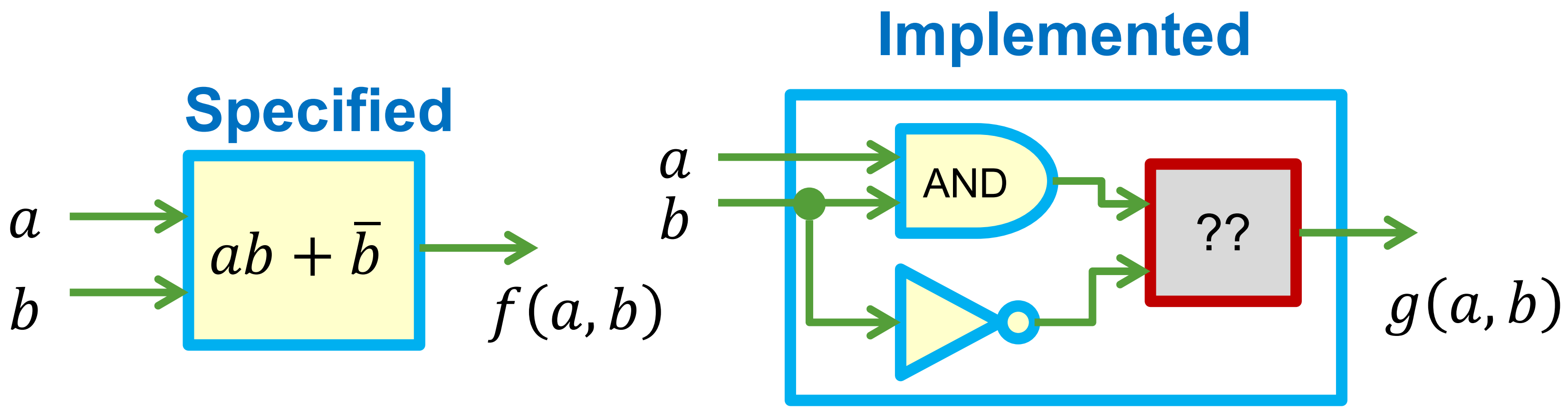


$c = a_1(a_0x + a_0d + xd)$

- $(\forall a_1 a_0\ c)(x, d) = 0$
  - 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 exists** a_1 and a_0 that let $c = 1$.

Quantification Application: Network Repair

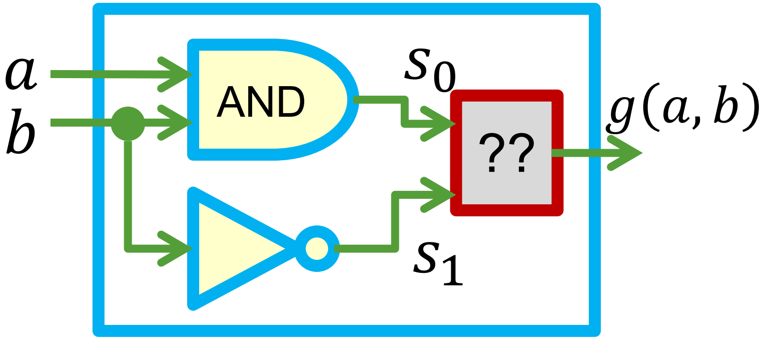
- Suppose that someone specified a logic block for you to implement: $f(a,b) = ab + \bar{b}$
  - ...but you implemented it **wrong**: in particular, you got ONE gate wrong.

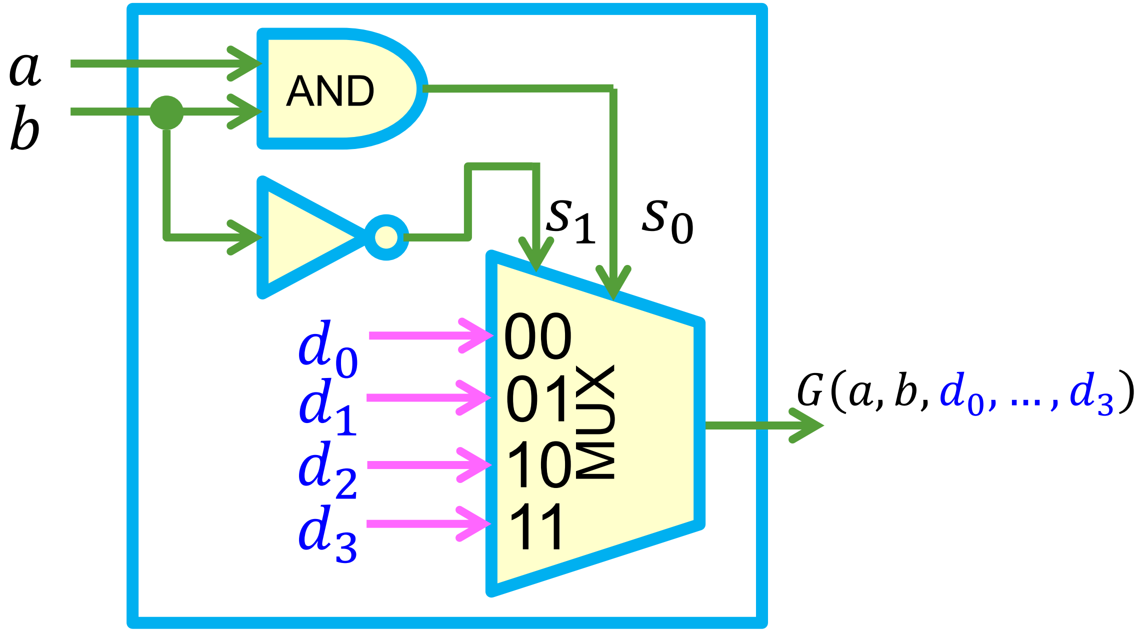


- <u>Goal</u>
  - Can we deduce how precisely to **change this gate** to restore correct function?
  - Go with this very trivial test case to see how mechanics work...

Network Repair

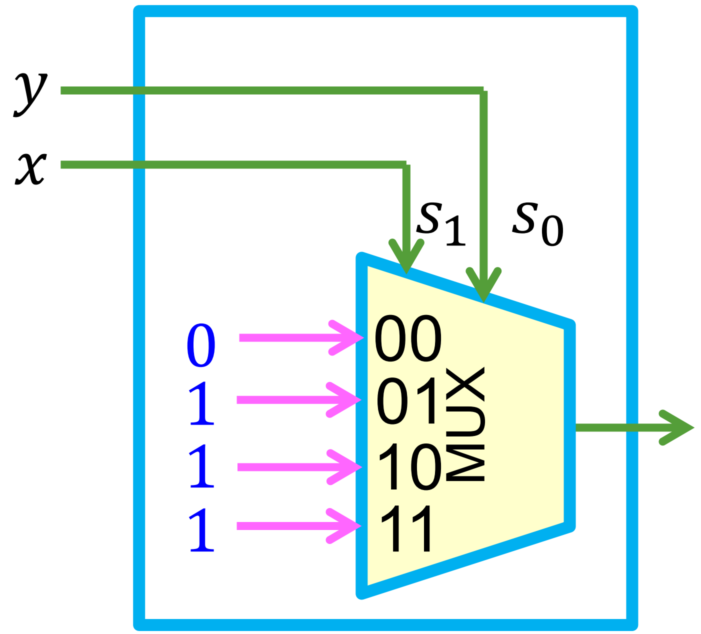
- Clever trick: Replace our suspect gate by a 4-to-1 MUX with 4 arbitrary new vars d_0, d_1, d_2, d_3.
  - By cleverly assigning values to d_0, d_1, d_2, d_3, we can **fake** any gate.
  - <u>Question is</u>: what are the right values of d_i's so g is repaired, i.e., $g = f$?

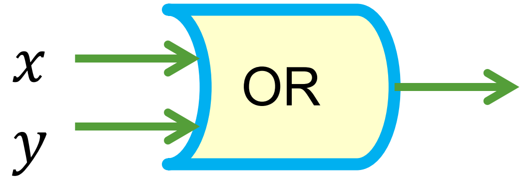




Aside: Faking a Gate with a MUX

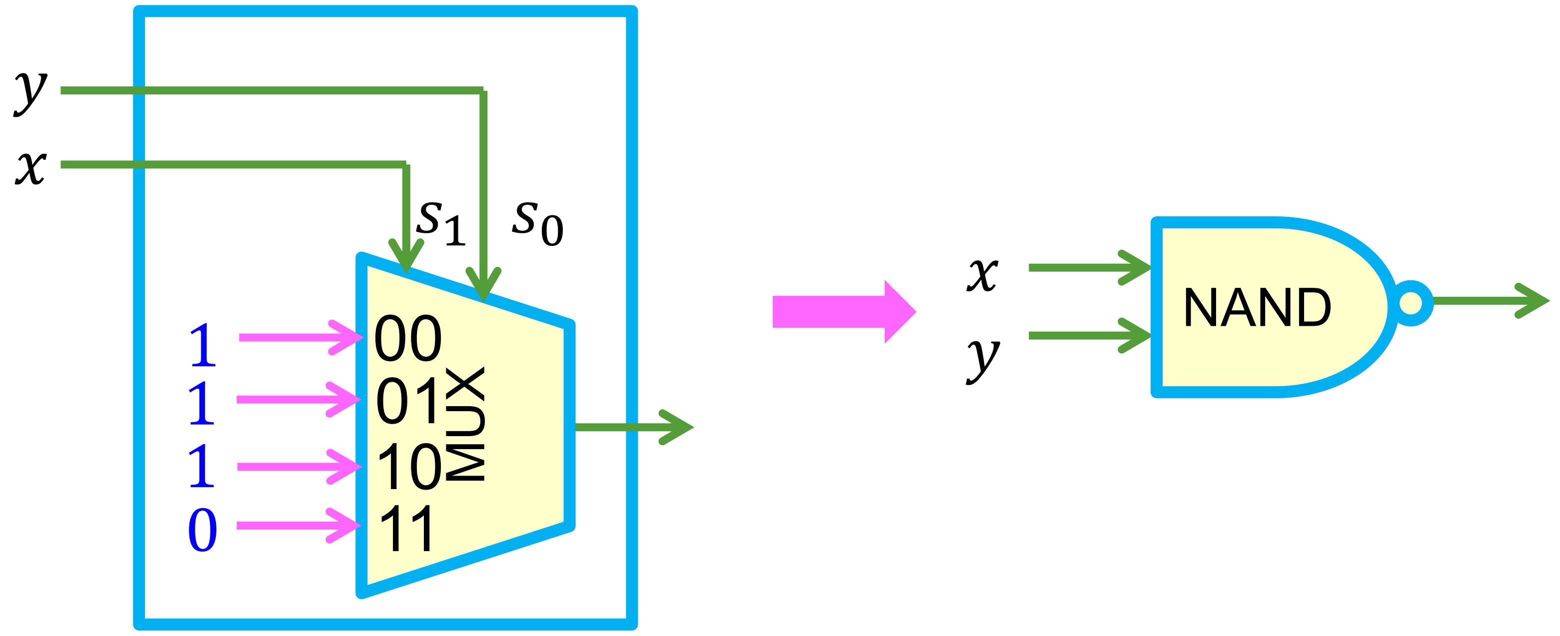
- You can do **any** function of 2 vars with one 4-to-1 multiplexor (MUX).





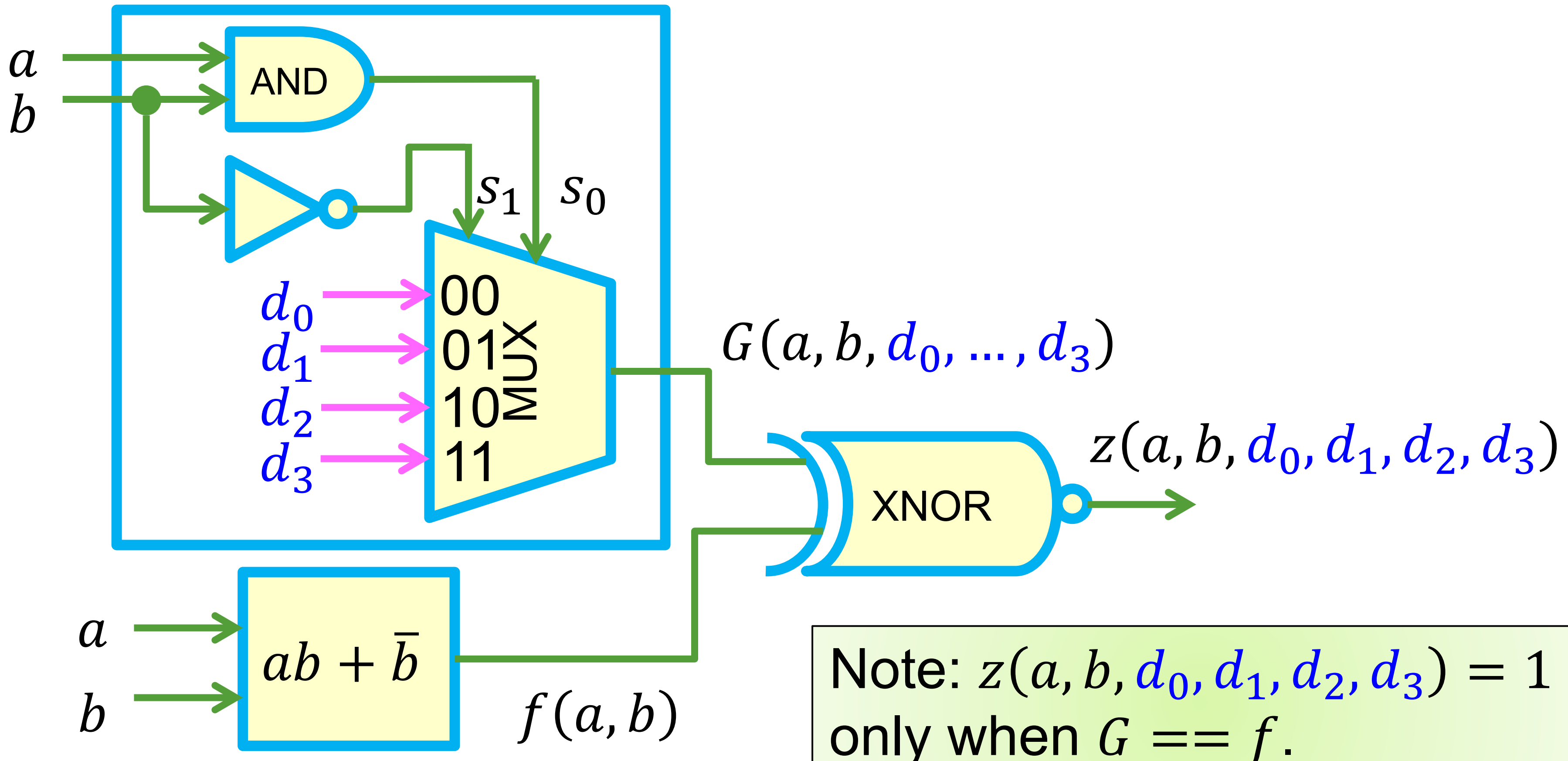
Aside: Faking a Gate with a MUX

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



Network Repair: Using Quantification

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

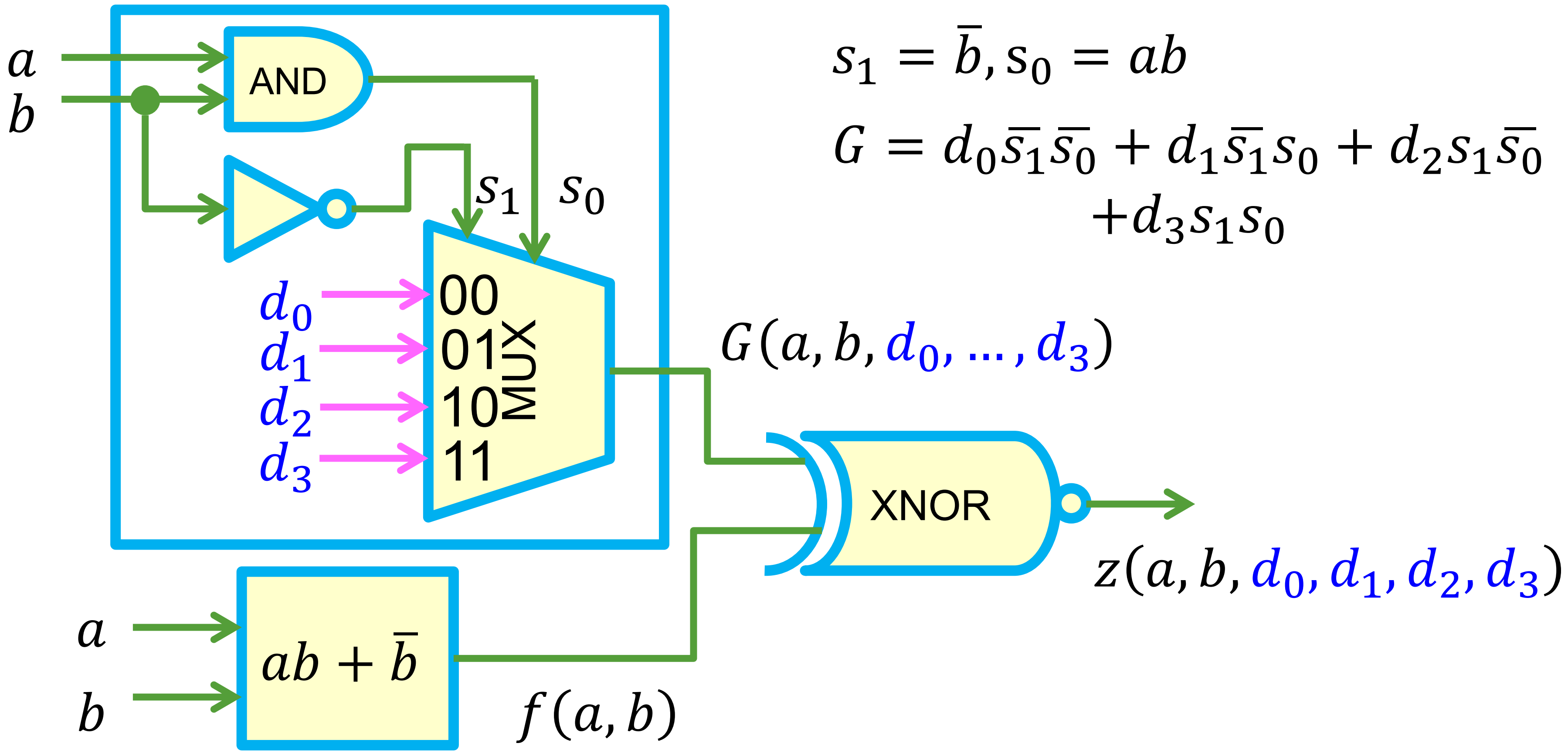


Note: $z(a, b, d_0, d_1, d_2, d_3) = 1$ only when $G == f$.

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 it!

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

- 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

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

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

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

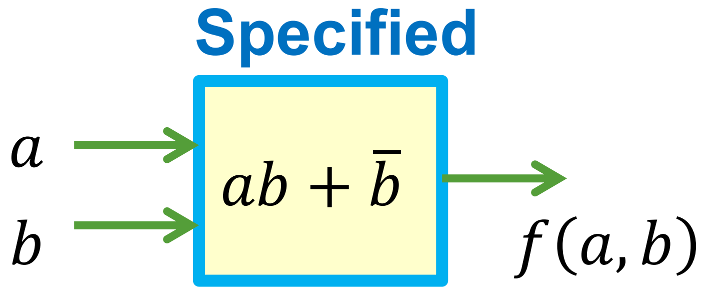
Network Repair via Quantification

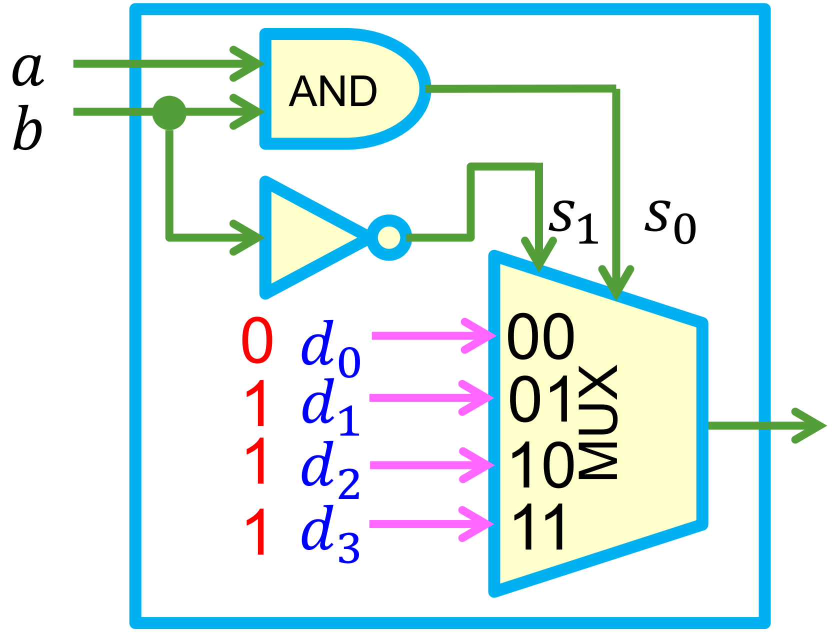
- Finally, we obtain $(\forall ab\ z)(d_0, d_1, d_2, d_3) = \overline{d_0} d_1 d_2$

- 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: $d_0 = 0, d_1 = 1, d_2 = 1, d_3 = X(\mathrm{don't\ care})$

Network Repair

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

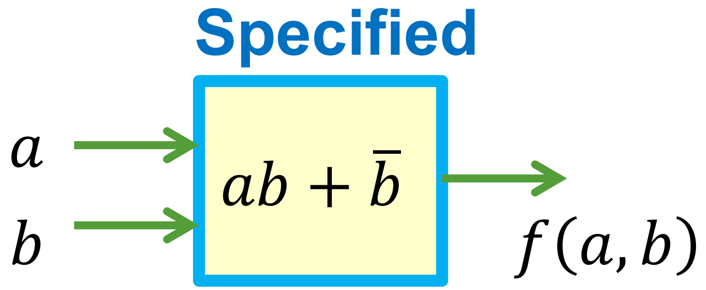


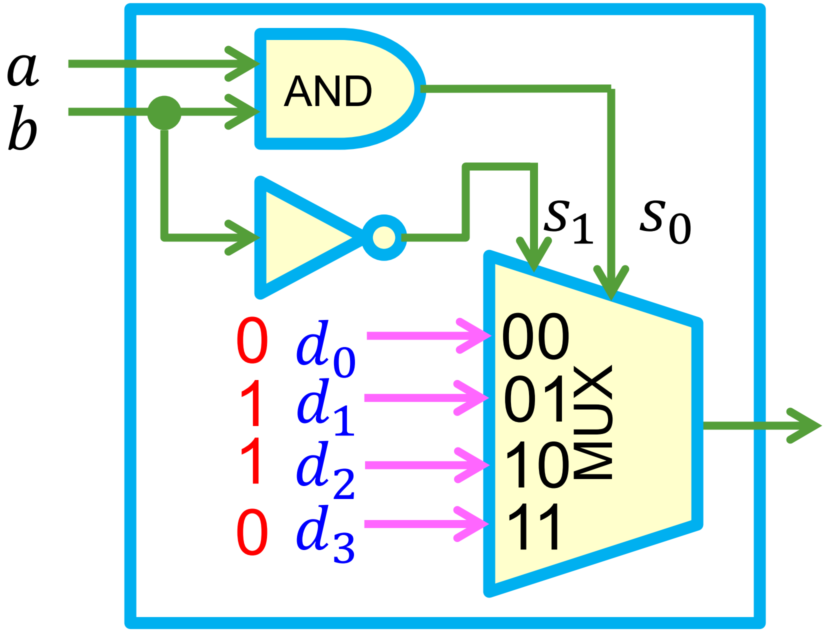


MUX is an OR gate. Expected!

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

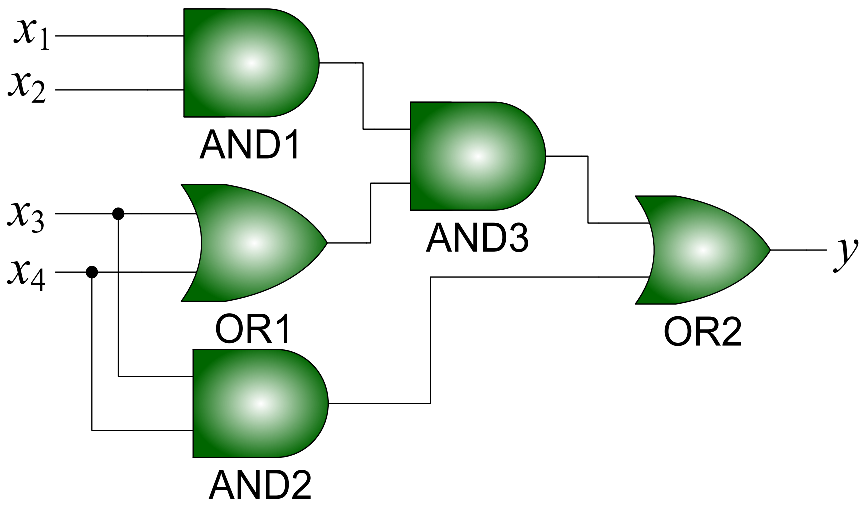
- This example is **tiny**...
  - But in a real example, you have a big network – 100 inputs, 50,000 gates.
  - When the design doesn't work, it is a major hassle to go through the design to fix it.
  - This gives a mechanical procedure to answer: Can we change 1 gate to repair?
- What we haven't seen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

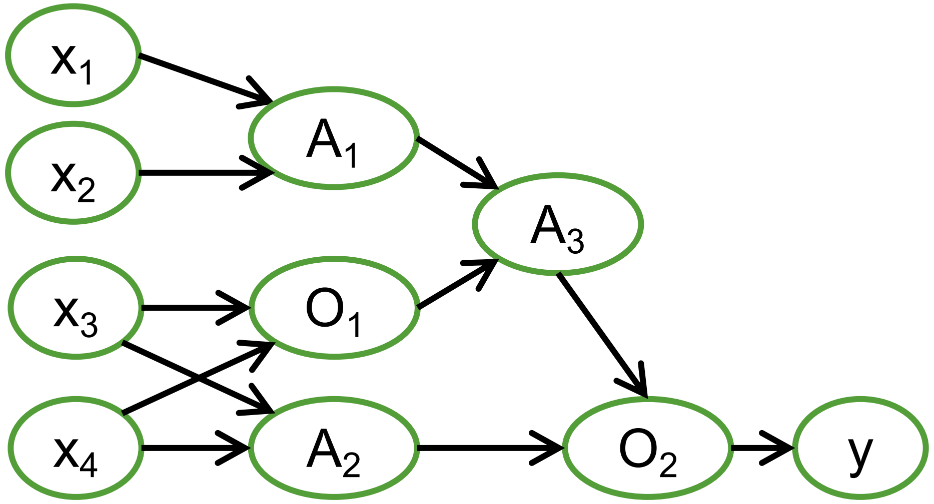
$$(\forall ab\ z)(d_0, d_1, d_2, d_3) = 1$$

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

Representation of Combinational Circuits

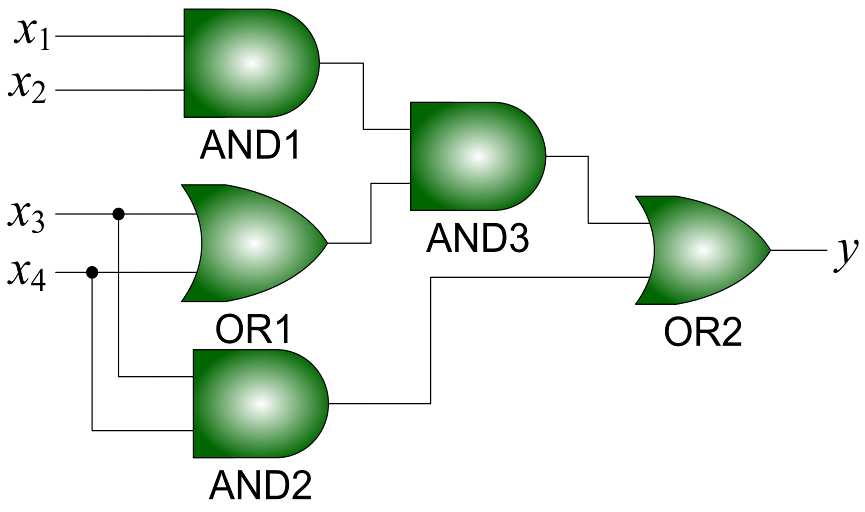
- Represented as a directed graph.
  - Inputs, outputs, and gates ➔ nodes
  - Wires ➔ directed edges.
    - Why directed edges? Signal flow has direction.

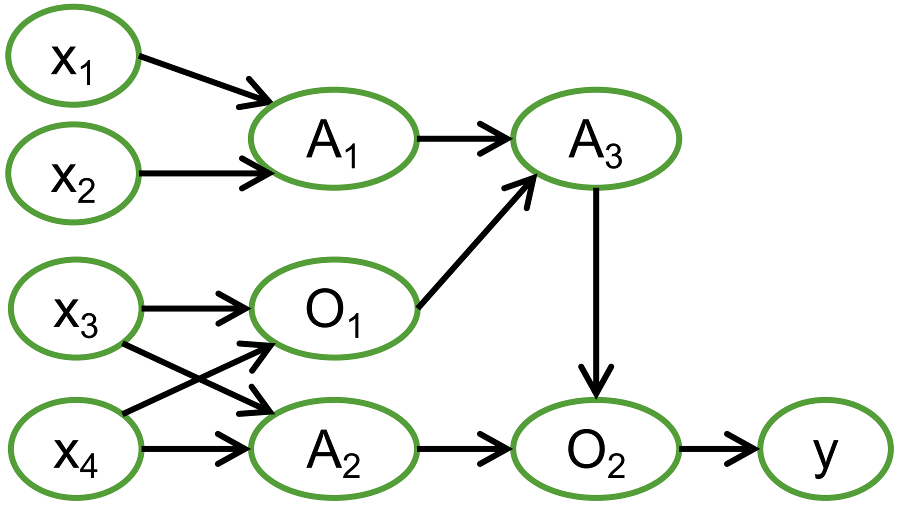




Representation of Combinational Circuits

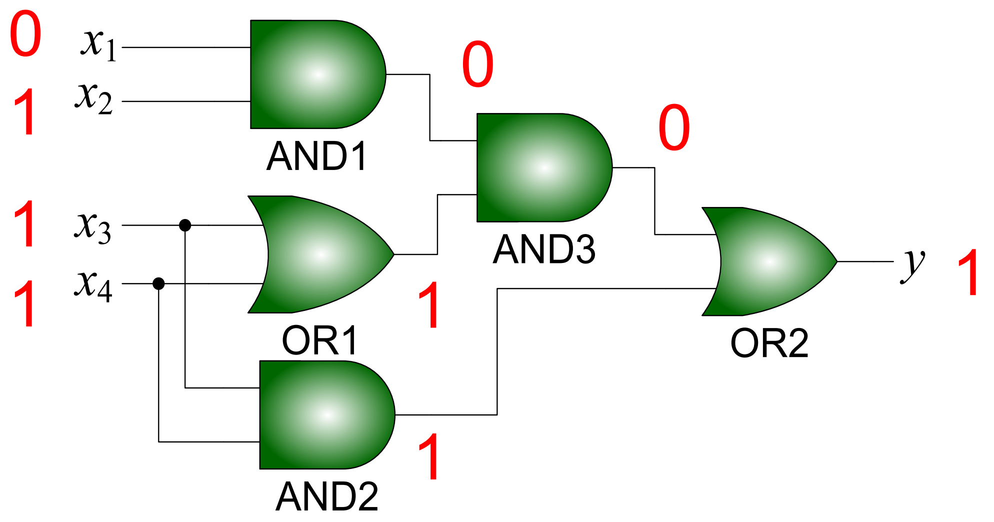
- Since a combinational circuit has no loops, the corresponding graph is a **directed acyclic graphs (DAG)**.
  - DAG: A directed graph with no cycles.





Traversal of Combinational Circuits

- Many operations on combinational circuit need to traverse it.
- Example: obtain the output given an input pattern.

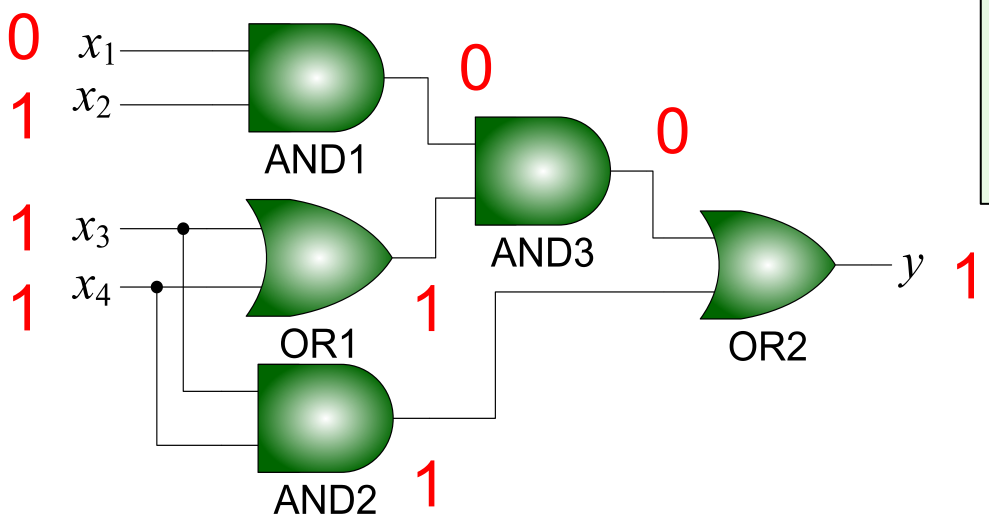


How to approach this kind of traversal as a computation?

Answer: Topological Sorting.

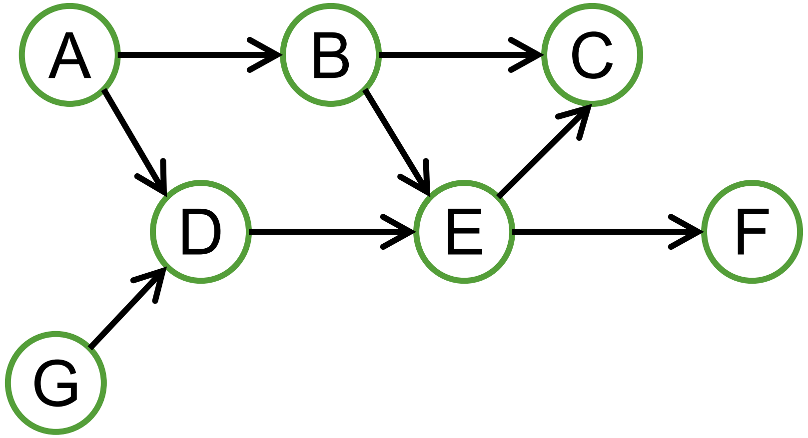
Topological Sorting

- Observation on computing the output of a combinational circuit:
  - The output of each gate can be computed only when all of its input are known. ➔ We have to obtain its inputs before obtaining output.
  - In other words, if there is a wire (edge) from gate (node) u to v, then gate (node) u should be visited before gate (node) v.
    - This is exactly **topological sorting**.



<u>One possible visiting order</u>:
x_1, x_2, x_3, x_4, AND1, OR1, AND2, AND3, OR2, y.

Topological Sorting

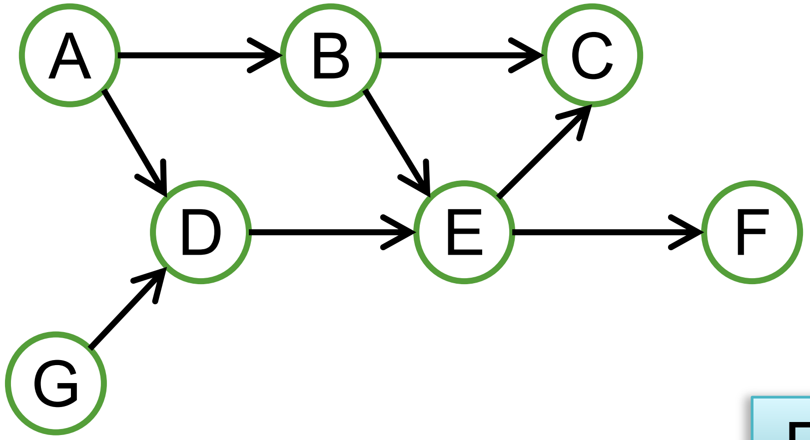


- Topological sorting is not necessarily unique:
  - A, G, D, B, E, C, F and A, B, G, D, E, F, C are both topological sorting.
- Are the following orderings topological sorting?
  - A, B, E, G, D, C, F
  - A, G, B, D, E, F, C

Topological Sorting: Algorithm

- Based on a **queue**.
- Algorithm:
  1. Compute the in-degrees of all nodes. (**in-degree**: number of **incoming** edges of a node.)
  2. Enqueue all in-degree 0 nodes into a queue.
  3. While queue is not empty
     1. Dequeue a node v from the queue and visit it.
     2. Decrement in-degrees of node v's neighbors.
     3. If any neighbor's in-degree becomes 0, enqueue it into the queue.

Topological Sorting Algorithm: Example



Queue

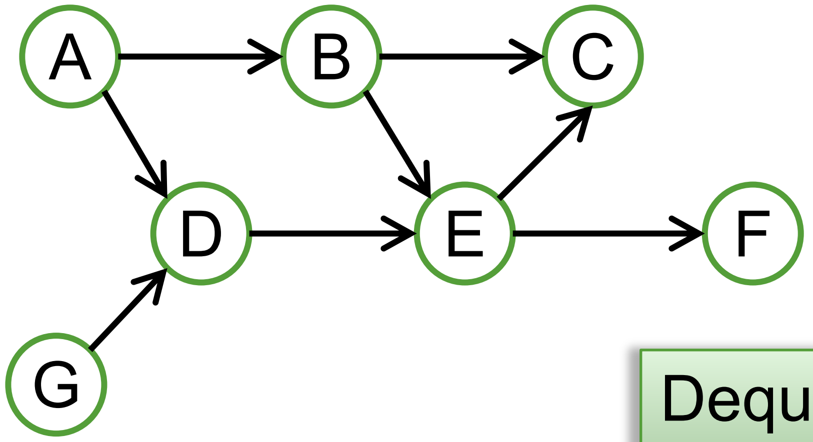
Enqueue A and G

In-degrees

A	B	C	D	E	F	G
0	1	2	2	2	1	0

Order

Topological Sorting Algorithm: Example



Queue

A
G

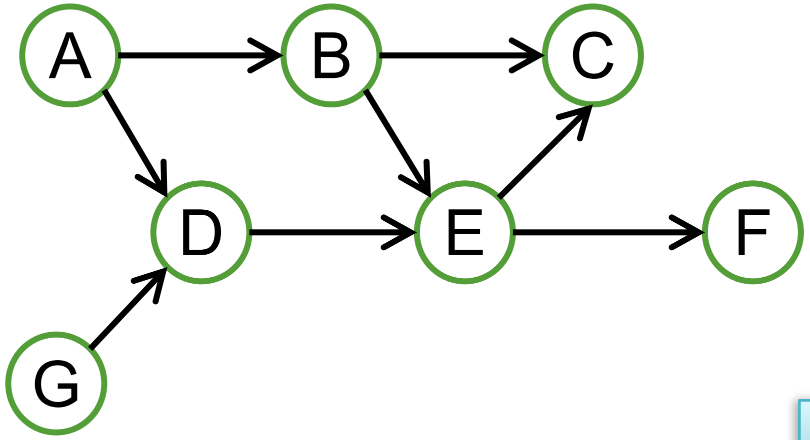
Dequeue A, visit A, and decrement in-degrees of A's neighbors.

In-degrees

A	B	C	D	E	F	G
0	1	2	2	2	1	0

Order

Topological Sorting Algorithm: Example



Queue

G

Enqueue B

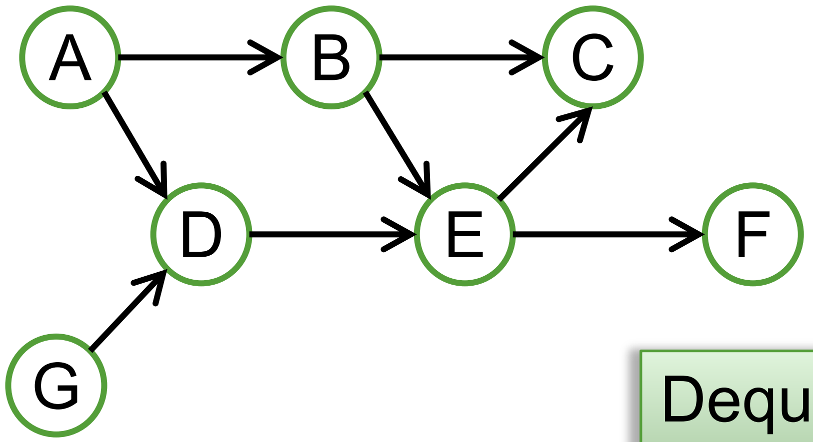
In-degrees

A	B	C	D	E	F	G
0	~~1~~ 0	2	~~2~~ 1	2	1	0

Order

A						

Topological Sorting Algorithm: Example



Queue

G
B

Dequeue G, visit G, and decrement in-degrees of G’s neighbors.

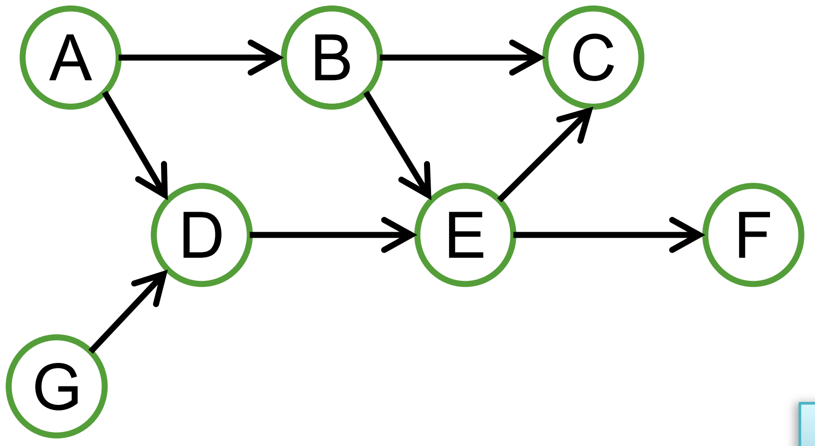
In-degrees

A	B	C	D	E	F	G
0	0	2	1	2	1	0

Order

A						

Topological Sorting Algorithm: Example



Queue

B

Enqueue D

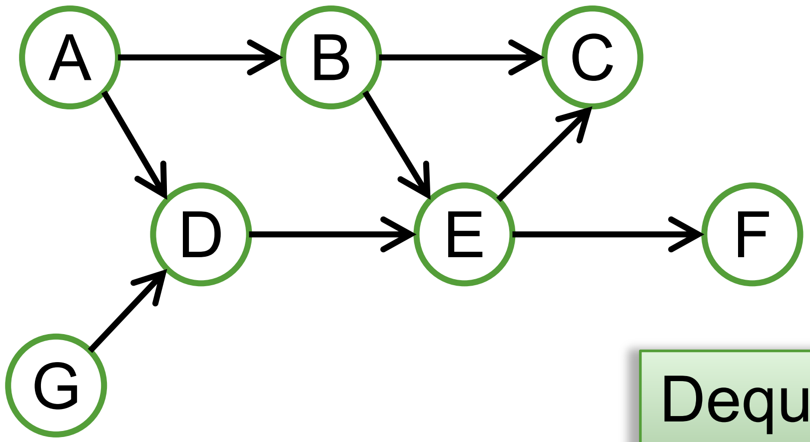
In-degrees

A	B	C	D	E	F	G
0	0	2	~~1~~ 0	2	1	0

Order

A	**G**					

Topological Sorting Algorithm: Example



Queue

B
D

Dequeue B, visit B, and decrement in-degrees of B's neighbors.

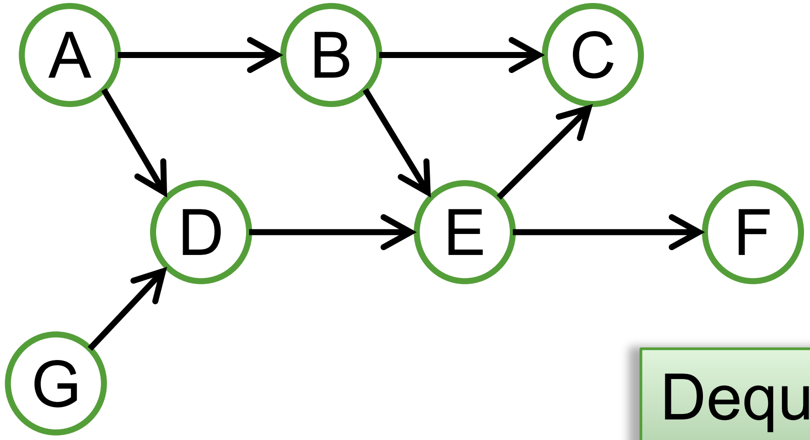
In-degrees

A	B	C	D	E	F	G
0	0	2	0	2	1	0

Order

A	G					

Topological Sorting Algorithm: Example



Queue

D

Dequeue D, visit D, and decrement in-degrees of D’s neighbors.

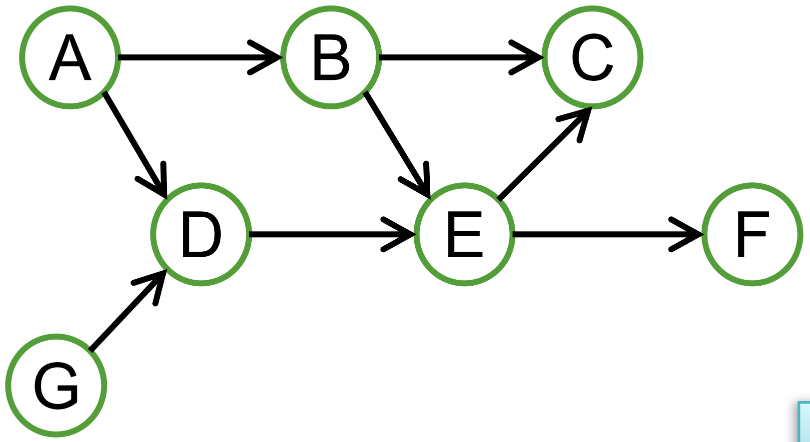
In-degrees

A	B	C	D	E	F	G
0	0	~~2~~ 1	0	~~2~~ 1	1	0

Order

A	G	**B**				

Topological Sorting Algorithm: Example



Queue

Enqueue E

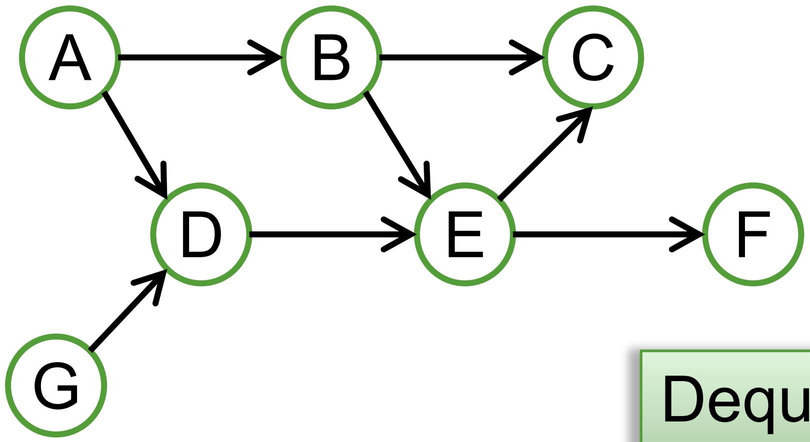
In-degrees

A	B	C	D	E	F	G
0	0	1	0	~~1~~ 0	1	0

Order

A	G	B	**D**			

Topological Sorting Algorithm: Example



Queue

E

Dequeue E, visit E, and decrement in-degrees of E’s neighbors.

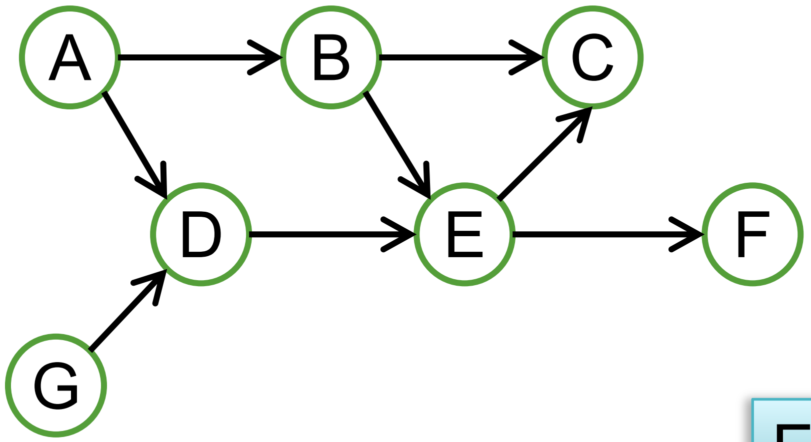
In-degrees

A	B	C	D	E	F	G
0	0	1	0	0	1	0

Order

A	G	B	D			

Topological Sorting Algorithm: Example



Queue

Enqueue C and F

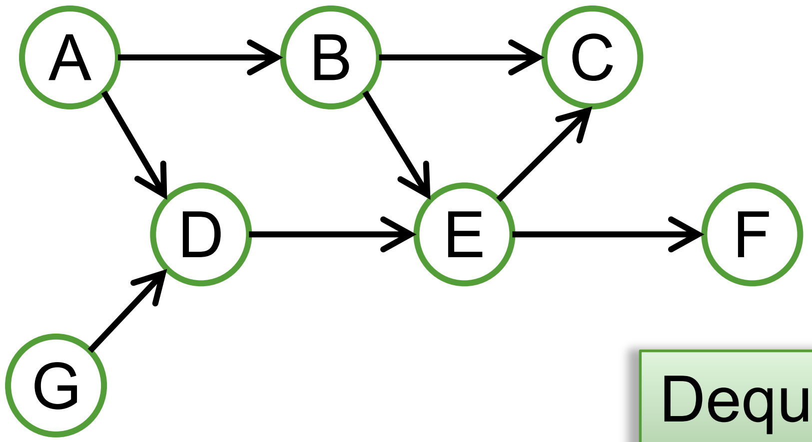
In-degrees

A	B	C	D	E	F	G
0	0	~~1~~ 0	0	0	~~1~~ 0	0

Order

A	G	B	D	**E**		

Topological Sorting Algorithm: Example



Queue

C
F

Dequeue C, visit C, and decrement in-degrees of C's neighbors.

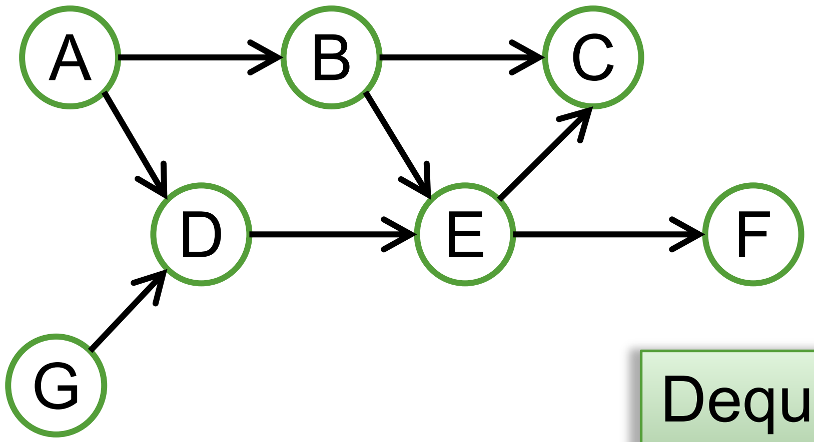
In-degrees

A	B	C	D	E	F	G
0	0	0	0	0	0	0

Order

A	G	B	D	E		

Topological Sorting Algorithm: Example



Queue

F

Dequeue F, visit F, and decrement in-degrees of F's neighbors.

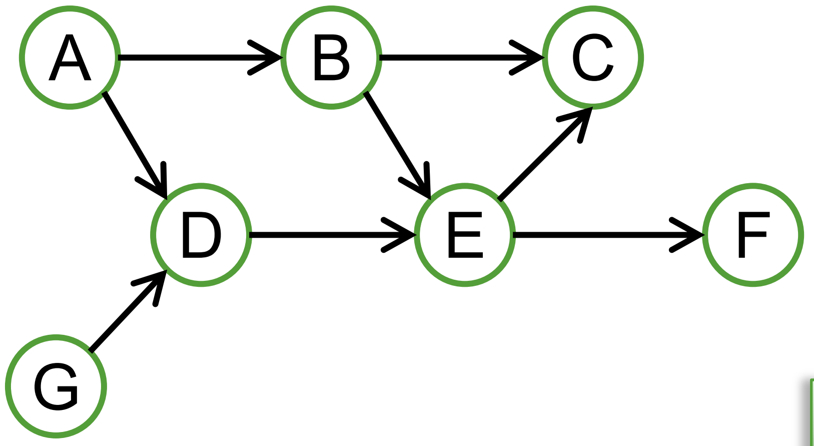
In-degrees

A	B	C	D	E	F	G
0	0	0	0	0	0	0

Order

A	G	B	D	E	C	

Topological Sorting Algorithm: Example



Queue

Queue is now empty. Done!

In-degrees

A	B	C	D	E	F	G
0	0	0	0	0	0	0

Order

A	G	B	D	E	C	**F**

Circuit Netlist Format

- We also need to store circuits as a text file.
  - … to be processed by different programs, e.g., layout synthesis tool, SPICE, schematic viewer, etc.
  - Such files essentially store a netlist of gates.

- Many formats exist:
  - Berkeley Logic Interchange Format (BLIF)
  - Structured Verilog Format
  - Benchmark Format

Example: Benchmark Format

INPUT(x1)
INPUT(x2)
INPUT(x3)
INPUT(x4)
OUTPUT(y)
g1=AND(x1,x2)
g2=OR(x3,x4)
g3=AND(x3,x4)
g4=AND(g1,g2)
y=OR(g3,g4)