Reading/Hand-on Assignment 1

- Survey of 3 SAT solvers
  - **MiniSAT**, Sweden.
  - **CHAFF**, Princeton University.
  - **GRASP**, University of Michigan.
- 3 groups, 1 group per solver.
- Oral presentation (**April 14th**, in class)
  - Technical details.
  - Your test run of the solvers + results.
- Written report (due **April 19th**)
  - One copy per group.

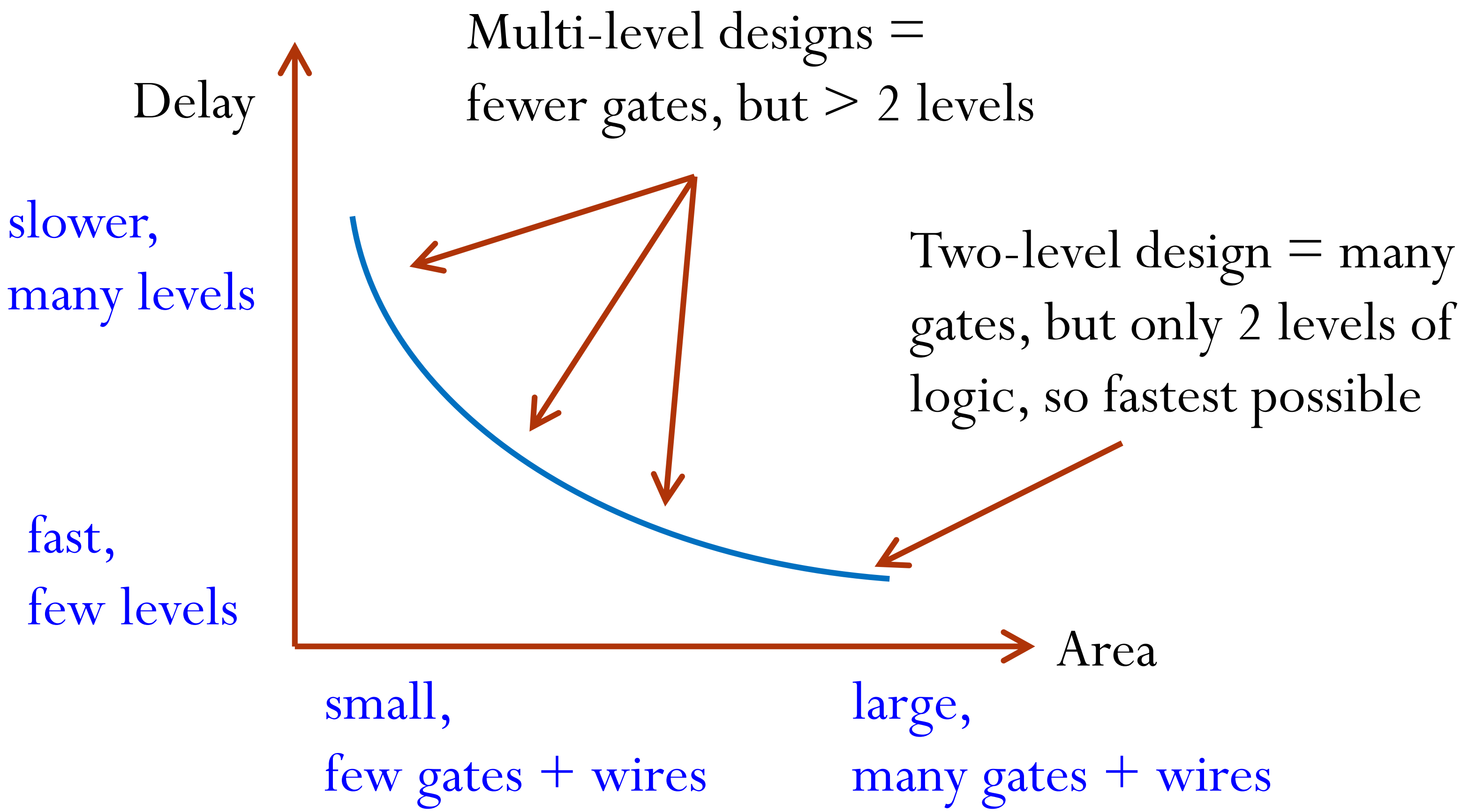
Multi-Level Logic Synthesis

Pingqiang Zhou
ShanghaiTech University

Why Multi-level Logic?

- Two-level forms are too **restrictive**.
- It has small delay but large area.
  - **Area** = gates + literals (wires), i.e., things that take space on a chip.
  - **Delay** = maximum levels of logic gates required to compute function.
  - Two-level is **minimum** gate delay possible, but usually **worst** on area.

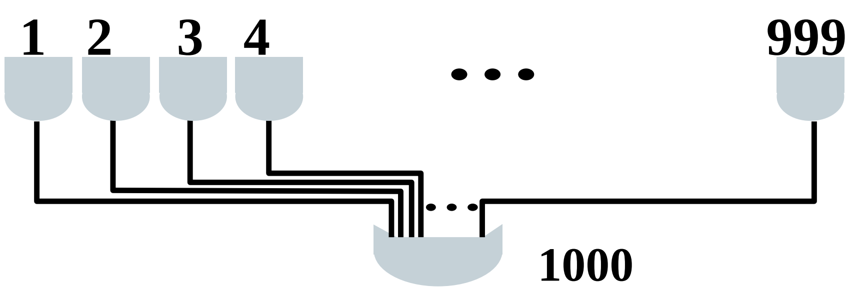
Area versus Delay Tradeoff



Why Multi-level Logic?

- **<u>Rarely</u>** see 2-level designs for really big things…
  - We use 2-level logic mostly for **pieces** of bigger things.
  - Even small things routinely done as **multi-level**.

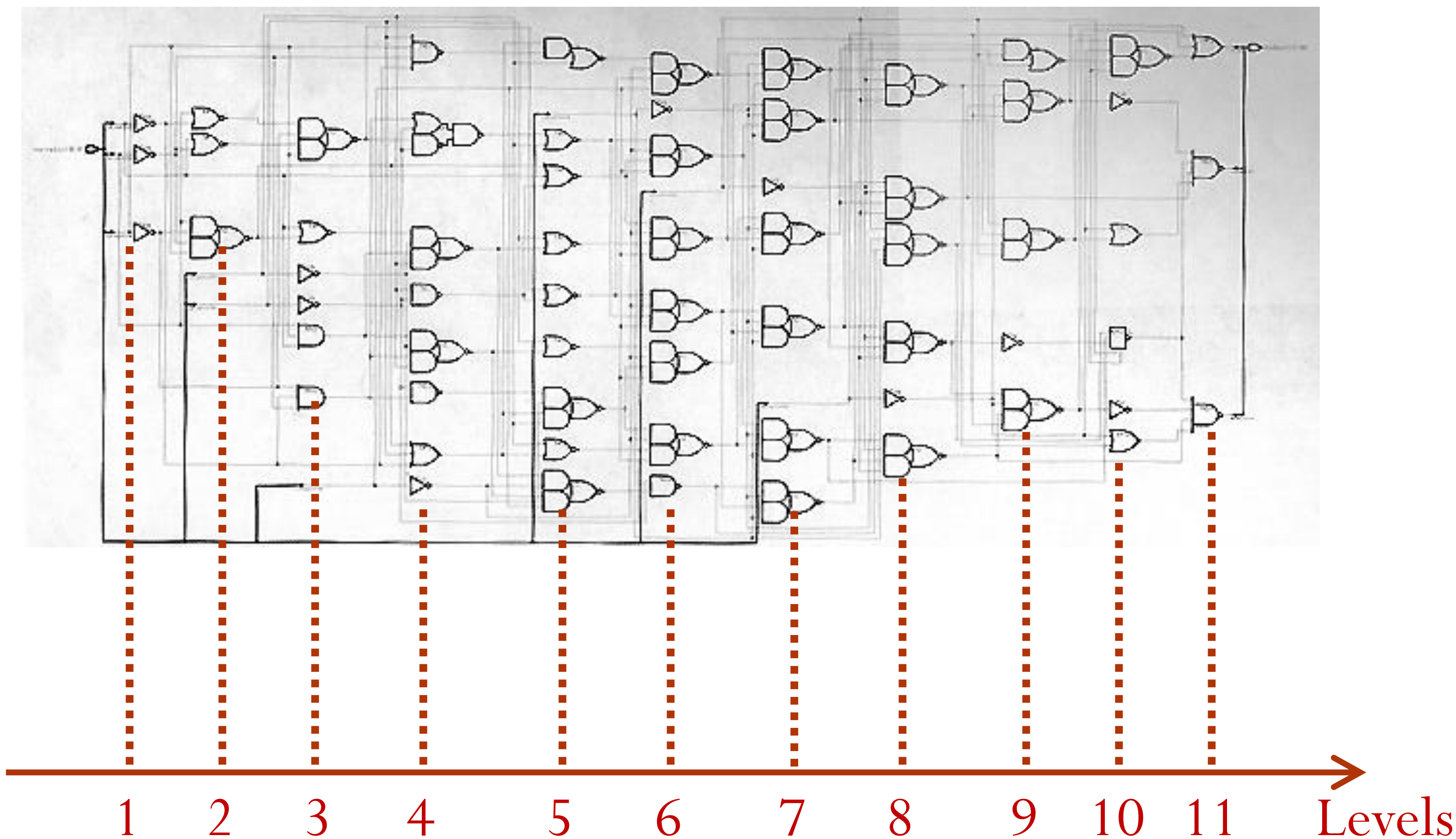
- What does a 2-level design with 1000 gates look like?



This is just **NOT** going to be the preferred logic network structure…

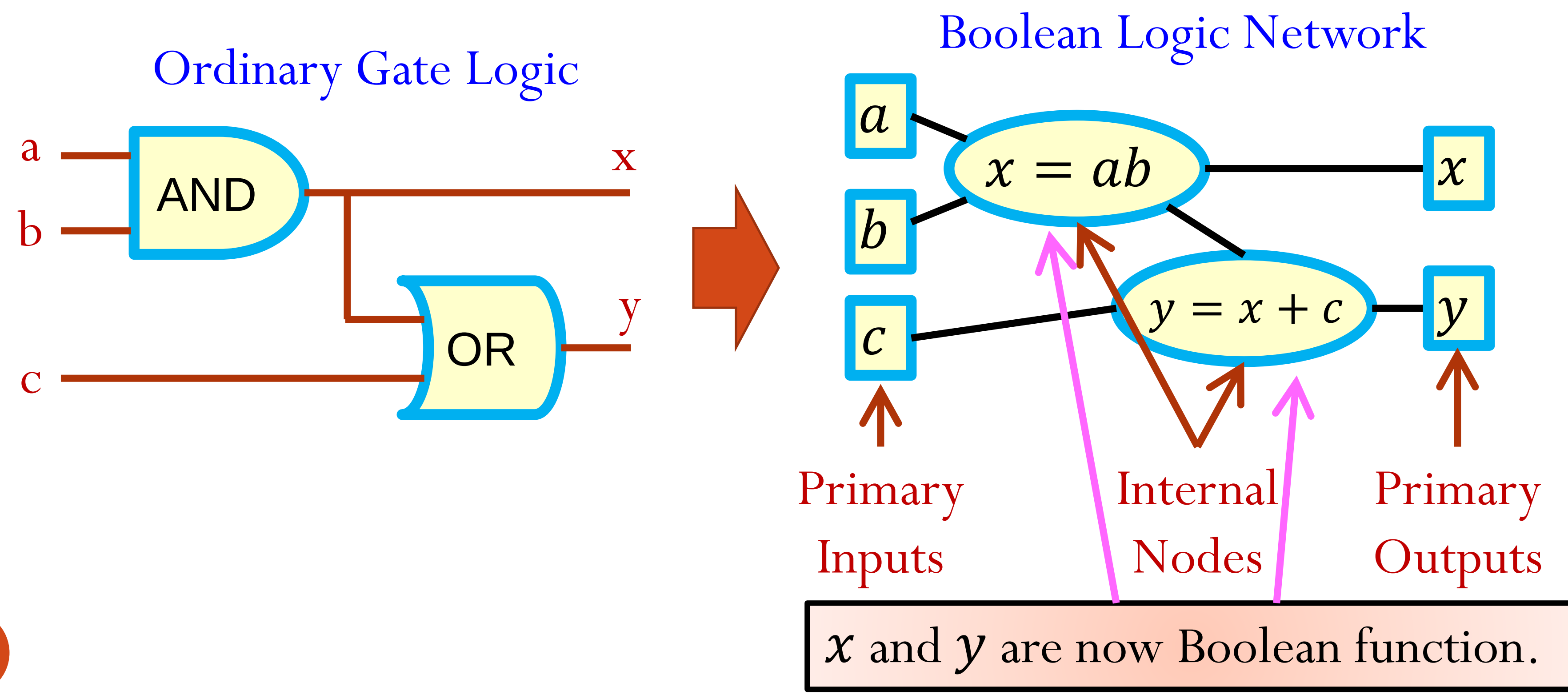
Real Multilevel Example

- A small design, done by commercial synthesis tool.



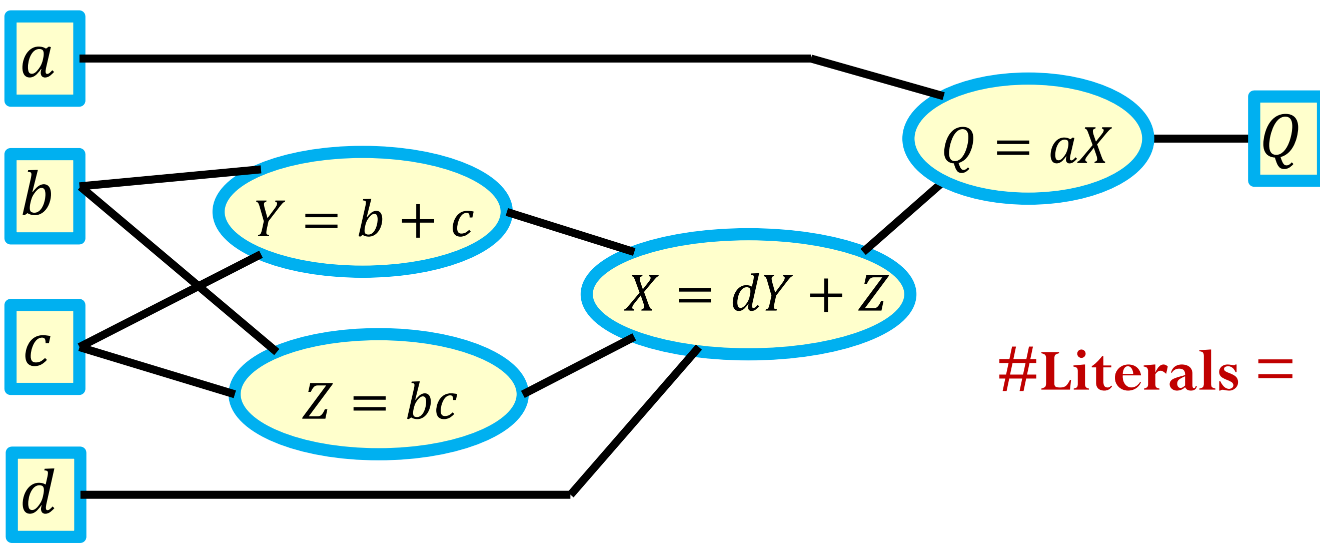
Boolean Logic Network Model

- Need more sophisticated model: **Boolean Logic Network**
  - Idea: it's a **graph** of connected blocks, like any logic diagram, but now individual component blocks can be **2-level Boolean functions in SOP form**.



Multilevel Logic: What to Optimize?

- A simplistic but surprisingly useful metric:
  Total literal count
  - Count **every** appearance of **every** variable on **right hand side** of "=" in **every internal node**.
  - Delays also matter, but for this class, only focus on **logic complexity**.



#Literals = 9

Optimizing Multilevel Logic: Big Ideas

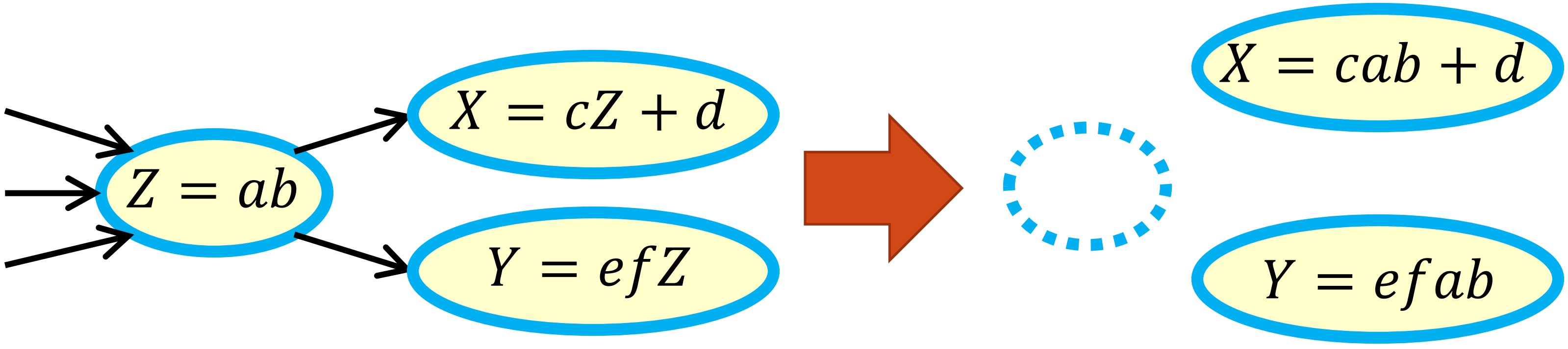
- Again: **Boolean logic network** is a **data structure**. What **operators** do we need?
- 3 basic kinds of operators:
  - **Simplify** network nodes: **no change** in # of nodes, just simplify insides, which are **SOP form**.
  - **Remove** network nodes: take "too small" nodes, **substitute them back** into fanouts.
    - This is not too hard. This is mostly manipulating the graph, simple SOP edits.
  - **Add** new network nodes: this is **factoring**. Take big nodes, split into smaller nodes.
    - This is a **big deal**. This is new. This is what we need to teach you…

Simplifying a Node

- You already know this! This is **2-level synthesis**.
- Just run ESPRESSO on 2-level form **inside** the node, to reduce # literals.
- As structural changes happen across network, "**insides**" of nodes may present opportunity to simplify.

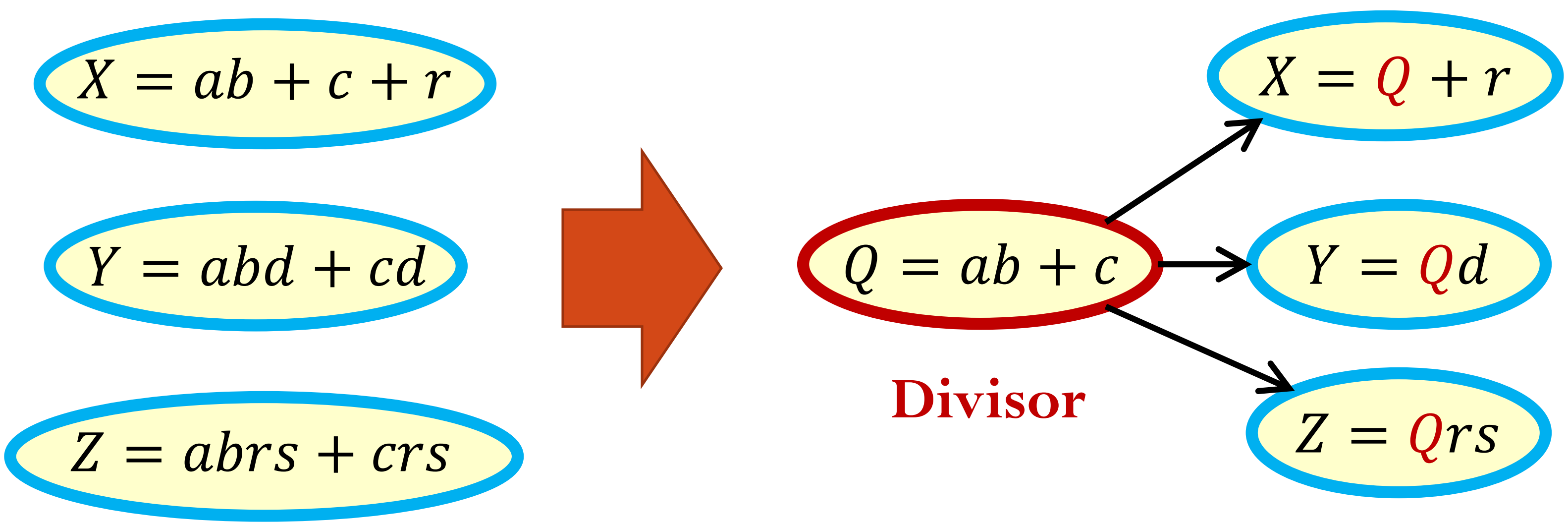
Removing a Node

- Typical case is you have a "**small**" factor which doesn't seem to be worth making it a **separate** node.
- "**Push**" it back into its fanouts, make those nodes bigger, and hope you can use 2-level simplification on them.



Adding new Nodes

- This is Factoring, this is new, and hard.
  - Look at existing nodes, identify **common divisors**, extract them, connect as **<u>fan-ins</u>**.
  - Tradeoff: **more** delay (another level of logic), but **fewer literals** (less gate area).



16 Literals

10 Literals

Multi-Level Logic Synthesis

- A more common design style.
  - Small area, but may have large delay.
- More sophisticated model: **Boolean logic network**
- 3 kinds of optimizing step:
  - Simplify a node by **2-level minimization**.
  - Remove a node by **substituting**.
  - Add a node by **factoring**.

Multilevel Synthesis Scripts

- Multilevel synthesis like 2-level synthesis is **heuristic**.
- …but it's also more complex. Write **scripts of basic operators**.
  - Do **several passes** of different optimizations over the Boolean logic network.
  - Do some "cleanup" steps to get rid of "too small" nodes (**remove nodes**).
  - Look for "easy" **factors**: just look at existing nodes, and try to use them.
  - Look for "hard" **factors**: do some work to extract them, try them, and keep the good ones.
  - Do 2-level optimization of insides of each logic node in network (**simplify nodes** by ESPRESSO).
  - Lots of "art" in the engineering of these scripts.
- For us, the one big thing you don't know: **How to factor**…

Multi-Level Logic Synthesis

- We need a new operator: **factoring**
- Problem #1: how to do division?
  - **Solution**: Algebraic model and algebraic division
  - **Algebraic model**: **Pretending** that Boolean expressions behave like **polynomials of real numbers**, not like Boolean algebra.
  - **Algebraic division**: Given a Boolean expression F and a divisor D, obtain quotient Q and remainder R, such that

$$F = D \cdot Q + R$$

- Problem #2: how to find good divisors?
  - **Solution**: **Kernels**.

Another New Model: Algebraic Model

- Factoring: How do we really do it?
  - Develop another model for Boolean functions, cleverly designed to let us do this
  - Tradeoff: lose some "expressivity" – some aspects of Boolean behavior and some Boolean optimizations we just **cannot do**, but we **gain practical factoring**.

- **New model**: **Algebraic model**
  - Term "algebraic" comes from **pretending** that Boolean expressions behave like **polynomials of real numbers**, not like Boolean algebra.
  - **Big new Boolean operator**: **Algebraic Division** (or, also **"Weak" Division**).

Algebraic Model

- Idea: keep just those **rules** that work for **BOTH** polynomials of reals **AND** Boolean algebra, but **get rid of** the rest.

Real numbers

$$a \cdot b = b \cdot a \quad a + b = b + a$$
$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$
$$a + (b + c) = (a + b) + c$$
$$a \cdot (b + c) = a \cdot b + a \cdot c$$
$$a \cdot 1 = a \quad a \cdot 0 = 0$$
$$a + 0 = a$$

Same

Boolean algebra

$$a \cdot b = b \cdot a \quad a + b = b + a$$
$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$
$$a + (b + c) = (a + b) + c$$
$$a \cdot (b + c) = a \cdot b + a \cdot c$$
$$a \cdot 1 = a \quad a \cdot 0 = 0$$
$$a + 0 = a$$

$$a \cdot \bar{a} = 0 \quad a + \bar{a} = 1$$
$$a \cdot a = a \quad a + a = a$$
$$a + 1 = 1$$
$$(a + b)(a + c) = a + b \cdot c$$

Not Allowed

Algebraic Model

- If we only get to use algebra rules from real numbers…
  - **Consequence**: A variable and its complement must be treated as **totally unrelated**.
    - Since no expression like $a + \bar{a} = 1$ allowed.

$$F = ab + \bar{a}x + \bar{b}y$$

Let $R = \bar{a}, S = \bar{b}$

$$F = ab + Rx + Sy$$

  - **Aside**: this is one of the losses of "expressive power" of Boolean algebra.

Algebraic Model

- Idea
  - Boolean functions manipulated in SOP form like **polynomials**.
  - Each product term in such an expression is just **a set of variables**, e.g., $abRy$ is the set (a, b, R, y).
  - SOP expression itself is just a **list of these products (cubes)**, e.g., $ab + Rx$ is the list (ab, Rx).

Algebraic Division: Our Model for Factoring

- Given function F we want to factor as:

$$F = D \cdot Q + R$$

divisor quotient remainder

  - If remainder $R = 0$, we call the divisor as a "**factor**".

Example: $F = ac + ad + bc + bd + e$

$$= (a + b)(c + d) + e$$

divisor quotient remainder

Algebraic Division

Divisor is a **factor** if $R = 0$.

- Example: $F = ac + ad + bc + bd + e$
  - **Want**: $F = D \cdot Q + R$.

Divisor (D)	Quotient (Q)	Remainder (R)	Is D Factor?
$ac + ad + bc$ $+bd + e$	1	0	Yes
$a + b$	$c + d$	e	No
$c + d$	$a + b$	e	No
a	$c + d$	$bc + bd + e$	No
b	$c + d$	$ac + ad + e$	No
c	$a + b$	$ad + bd + e$	No
d	$a + b$	$ac + bc + e$	No
e	1	$ac + ad + bc + bd$	No

Algebraic Division: Very Nice Algorithm

- **Inputs**: A Boolean expression F and a divisor D, represented as lists of cubes (and each cube as a set of literals).
- **Output**
  - Quotient $Q = F/D$ as a cube list, or **empty** if $Q = 0$.
  - Remainder R as a cube list, or **empty** if D was a **factor**.

- **Strategy**
  - Cube-wise walk through cubes in divisor D, trying to divide each of them into F.
  - … intersect all the division results.

Algebraic Division Algorithm

```
AlgebraicDivision( F, D ) {  // divide D into F
   for ( each cube d in divisor D ) {
      let C = { cubes in F that contain this product term d };
      if ( C is empty )  return ( quotient = 0,  remainder = F );
      let C = cross out literals of cube d in each cube of C;
      if ( d is the first cube we have looked at in divisor D )
            let Q = C;
      else  Q = Q ∩ C;
   }
   R = F − ( Q • D );
   return (quotient = Q,  remainder = R);
}
```

Example:
Cube **xyzw** contains product term **yz**

Example:
Suppose **C = xyz + yzw +pqyz** and **d = yz**. Then crossing out all the **yz** parts yields **x + w + pq**

Algebraic Division: Example

$$F = axc + axd + axe + bc + bd + de, D = ax + b$$

F cube	D cube: ax	D cube: b
axc	$axc \rightarrow c$	$-$
axd	$axd \rightarrow d$	$-$
axe	$axe \rightarrow e$	$-$
bc	$-$	$bc \rightarrow c$
bd	$-$	$bd \rightarrow d$
de	$-$	$-$

$$C = c + d + e \qquad C = c + d$$

$$Q = (c + d + e) \cap (c + d) = c + d$$

$$R = F - QD = axe + de$$

Algebraic Division: Warning

- Remember: No "**Boolean**" simplification, only "**algebraic**".
  - So what? Well, suppose you have this
    $$F = a\bar{b}\bar{c} + ab + ac + bc, D = ab + \bar{c}$$
    and you want F/D.
  - You must let $X = \bar{b}$ and $Y = \bar{c}$ and transform F and D to something like
    $$F = aXY + ab + ac + bc, D = ab + Y$$
  - Because we must treat the true and complement forms of variables as **totally unrelated**.

One More Constraint: Redundant Cubes

- To do F/D, function F must have no **redundant** cubes
  - Technical definition is: **minimal** with respect to **single-cube containment**.
  - **Means**: no one cube is **completely** covered by **one of the other** cubes in SOP cover.
    - E.g., $abcd$ is completely covered by ab.

- Why **no** redundant cubes?
  - **Consider**: $F = a + ab + bc$ and $D = a$.
    - **Note**: F has redundant cube ab.
  - What is F/D by our **algebraic division algorithm**?

$$\boxed{Q = F/D = 1 + b}$$

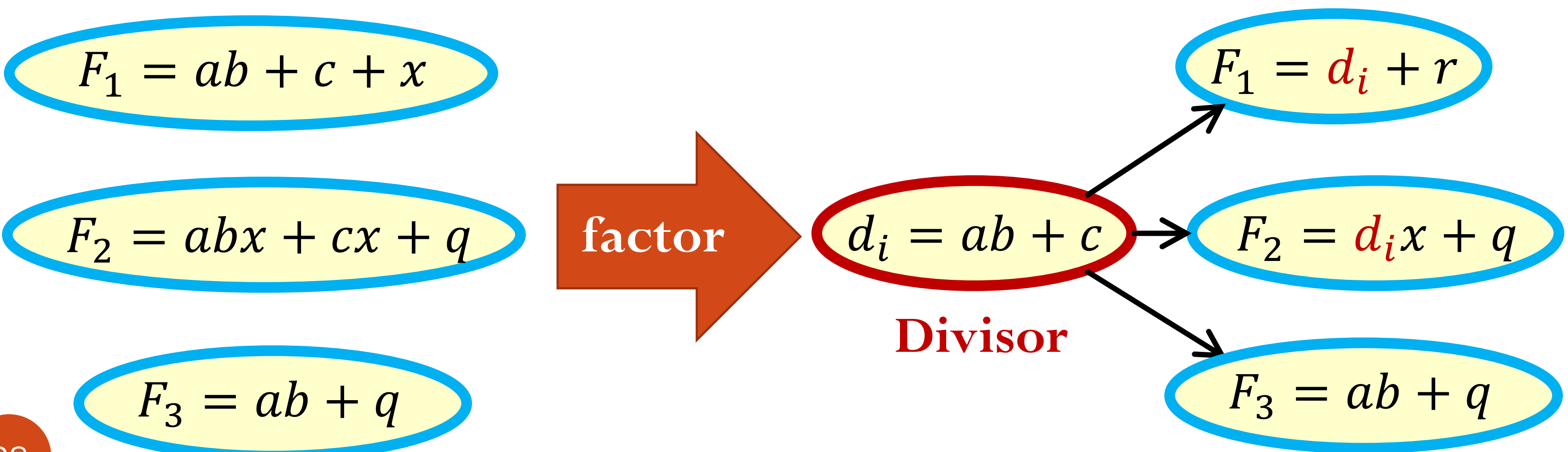
However, we don't have 1+stuff operation in algebraic model!

One More Constraint: Redundant Cubes

- … So, we should remove redundant cubes to make the SOP **minimal** with respect to **single-cube containment**.
  - Not hard.

Multilevel Logic Synthesis: Where are We?

- For Boolean function F and D, can compute $F = Q \cdot D + R$ via **algebraic model**.
  - It is great, but still <u>**not enough**</u>: don't know how to **find** a good divisor D.
  - <u>**Another problem**</u>: given n functions $F_1, F_2, \ldots, F_n$, find **a set of good common divisors**.



Where to Look for Good Divisors?

- Surprisingly, the **algebraic model** has a beautiful answer.
  - One more reason we like it: Has some surprising and elegant "deep structure".

- Where to look for divisors of function F?
  - In the set of **kernels** of F, denoted $K(F)$.
  - $K(F)$ is another set of 2-level SOP forms which are the special, foundational structure of any function F, being interpreted in our algebraic model.
- How to find a kernel $k \in K(F)$?
  - **Algebraically divide** F by one of its **co-kernels**, c.

Kernels and Co-Kernels of Function F

- **Kernel** of a Boolean expression F is:
  - A **cube-free** quotient k obtained by **algebraically dividing** F by a **single cube** c.
  - This single cube c also has a name: it is a **co-kernel** of function F.

quotient Q

divisor D | expression F

⋮

remainder R

$$F = D \cdot Q + R$$

kernel k (cube-free)

$c = 1$ cube | expression F

⋮

remainder R

$$F = c \cdot k + R$$

Kernels Are Cube-Free…

- **Cube-free** means…?
  - You cannot factor out a **<u>single</u>** cube divisor that **leaves no remainder**.
  - Technically: has no **one cube** that is a **factor** of expression.
    - Pick a cube c. If you can "**cross out**" c in **<u>each</u>** product term of F, then F is **not** a **kernel**.

Expression F	$F = D \cdot Q + R$	F Cube-free?
a	$a \cdot 1 + 0$	No
$a + b$	--	Yes
$ab + ac$	$a \cdot (b + c) + 0$	No
$abc + abd$	$ab \cdot (c + d) + 0$	No
$ab + acd + bd$	--	Yes

Some Kernel Examples

- Suppose $F = abc + abd + bcd$

Divisor cube d	$Q = F/d$	Is Q a kernel of F?
1	$abc + abd + bcd$	No, has cube $= b$ as factor
a	$bc + bd$	No, has cube $= b$ as factor
b	$\boxed{ac + ad + cd}$	Yes! co-kernel $= b$
ab	$\boxed{c + d}$	Yes! co-kernel $= ab$

- Any Boolean function F can have many different kernels.
  - The **set** of kernels of F is denoted as $K(F)$.

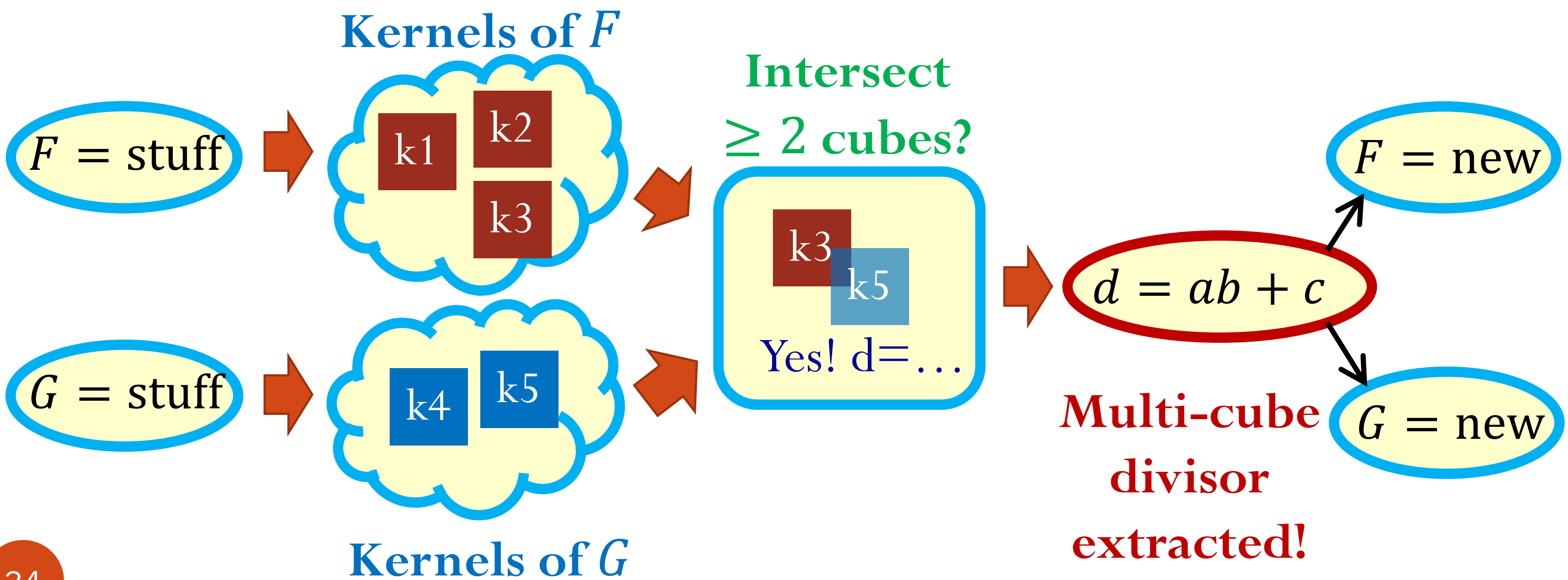
Kernels: Why Are They Important?

- Big result: Brayton & McMullen Theorem
  - **<u>From</u>**: R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions." In *IEEE International Symposium on Circuits and Systems*, pages 49–54, 1982.

Expressions F and G have a **common multiple-cube divisor** d
if and only if:
there are kernels $k1 \in K(F)$ and $k2 \in K(G)$ such that $d = k1 \cap k2$ and d is an expression with **at least 2** cubes in it (i.e., $k1$ and $k2$ have **common cubes**).

Multiple-Cube Divisors and Kernels

- Brayton & McMullen Theorem **<u>in words</u>**:
  - The **<u>only</u>** place to look for **multiple-cube divisors** is in the **intersection of kernels**!
    - Indeed, this intersection of kernels contains **<u>all</u>** divisors.



Brayton-McMullen: Informal Illustration

F = cube1 • **kernel1** + remainder1

G = cube2 • **kernel2** + remainder2

Assume:

kernel1 ∩ kernel2 = X + Y

F = cube1 • **[X + Y + stuff1]** + remainder1

G = cube2 • **[X + Y + stuff2]** + remainder2

F = cube1 • **[X + Y]** + [cube1•stuff1 + remainder1]

G = cube2 • **[X + Y]** + [cube2•stuff2 + remainder2]

d=**X+Y**

F = cube1•**d** + …

G = cube2•**d** + …

Kernels: Real Example

$F = ae + be + cde + ab$

$G = ad + ae + bd + be + bc$

Kernels	Co-kernel
$\boxed{a + b + cd}$	e
$b + e$	a
$a + e$	b
$ae + be + cde + ab$	1

Kernels	Co-kernel
$\boxed{a + b}$	d or e
$d + e$	a
$c + d + e$	b
$ab + ae + bd + be + bc$	1

Intersecting these 2 kernels: $(a + b + cd) \cap (a + b) = a + b$

Kernels: Very Useful, But How To Find?

- Another **recursive** algorithm ("recursive" again!)
  - There are 2 more useful properties of kernels we need to see first…

- Start with a function F and a kernel $k1$ in $K(F)$

$$F = \text{cube1} \bullet \text{k1} + \text{remainder1}$$

- Then: a new, interesting question: what about $K(k1)$?
  - $k1$ is a perfectly nice Boolean expression, so it has got **its own** kernels.
  - Do these $k1$'s kernels have anything interesting to say about $K(F)$?

How $K(k1)$ Relates to $K(F)$...

- We know this: F = cube1•k1 + remainder1
- Suppose k2 is a kernel in K(k1), then we know

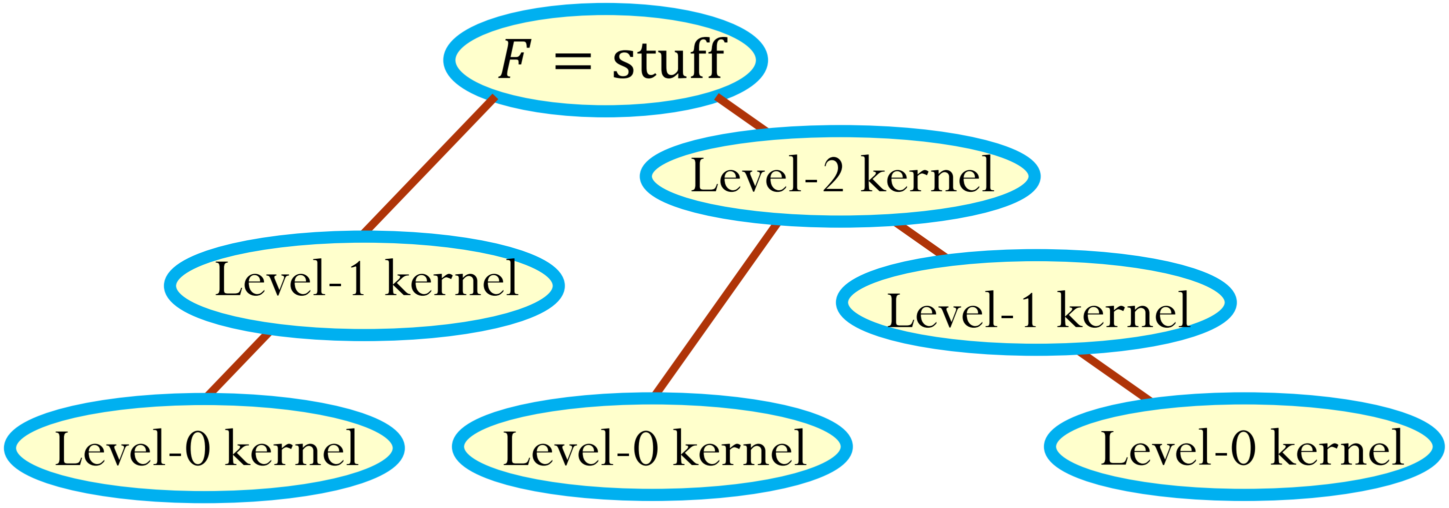
  k1 = cube2•k2 + remainder2
- Substitute this expression for k1 in original expression for F

  F = cube1•[cube2•k2 + remainder2] + remainder1
- Since cube1•cube2 is itself just another **single cube**, we have:

  F = (cube1•cube2)•[k2] + [cube1•remainder2 + remainder1]
- **Conclusion**: k2 also a **kernel** of original F (with **co-kernel** cube1•cube2)

There is a Hierarchy of Kernels Inside F

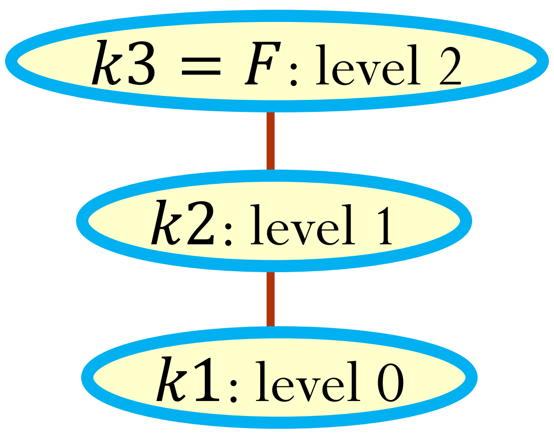
- **Definition**: $k \in K(F)$ is
  - A **level-0 kernel** if it contains no kernels inside it except itself.
    - In words: Only cube you can pull out, get a cube-free quotient is "1".
  - A **level-n kernel** if it contains **at least one** level-(n-1) kernel, and no other level-n kernels except itself.
    - In words: a level-1 kernel only has level-0 kernels inside it. A level-2 kernel only has level-1 and level-0 kernels in it, etc…



Kernel Hierarchy: Example

- $F = abe + ace + de + gh$ has three kernels:
  - $k1 = b + c$, with co-kernel ae.
  - $k2 = ab + ac + d$, with co-kernel e.
  - $k3 = F$ with co-kernel 1.

- Note: $k1$ is level 0, $k2$ is level 1, and $k3$ is level 2.



Kernels

- Second useful result [by Brayton et al.]:
  - **Co-kernels** of a Boolean expression in SOP form correspond to **intersections of 2 or more cubes** in this SOP form.
- **Note**: **Intersections** here means that we regard a cube as **a set of literals**, and look at common subsets of literals.
  - This is not like "AND" for products. This just extracts **common literals**.
  - Example: ace + bce + de + g

    ace ∩ bce = ce → ce is a potential co-kernel

    ace ∩ bce ∩ de = e → e is a potential co-kernel

How to Find Kernels Using These 2 Results?

- Find the kernels **recursively**.
  - Whenever find one kernel, call **FindKernels()** on it, to find (if any) lower level kernels inside.
- Use **algebraic division** to divide function by potential co-kernels, to drive recursion.
  - Use 2nd result – co-kernels are **intersections** of the cubes: If there're at least 2 cubes, then look at the intersection of those cubes, and use that intersected result as our potential co-kernel cube.

- One technical point: need to start with a **cube-free function** F to make things work right.
  - If not cube-free, just divide by biggest common cube to simplify F.

Kernel Algorithm

```
FindKernels( cube-free SOP expression F ) {
  K = empty;
  for ( each variable x in  F  ) {
     if ( there are at least 2 cubes in F that have variable x ) {
        let S = { cubes in F that have variable x in them };
        let co =  cube that results from intersection of all cubes in S,
                      this will be the product of just those literals
                      that appear in each of these cubes in S;
        K = K ∪ FindKernels( F / co) ;
     }
  }
  K = K ∪ F ;
  return( K );
}
```

Cube-free **F** is always its own kernel, with trivial co-kernel = **1**

Kernelling Example

```
FindKernels( F ):
for (each var x in  F ) {
    if (x in ≥ 2 cubes in F) {
        co = intersection of cubes;
        K=K ∪ FindKernels(F/co) ;
    }
}
K = K ∪ F ;
return( K );
```

$$F = ace + bce + de + g$$

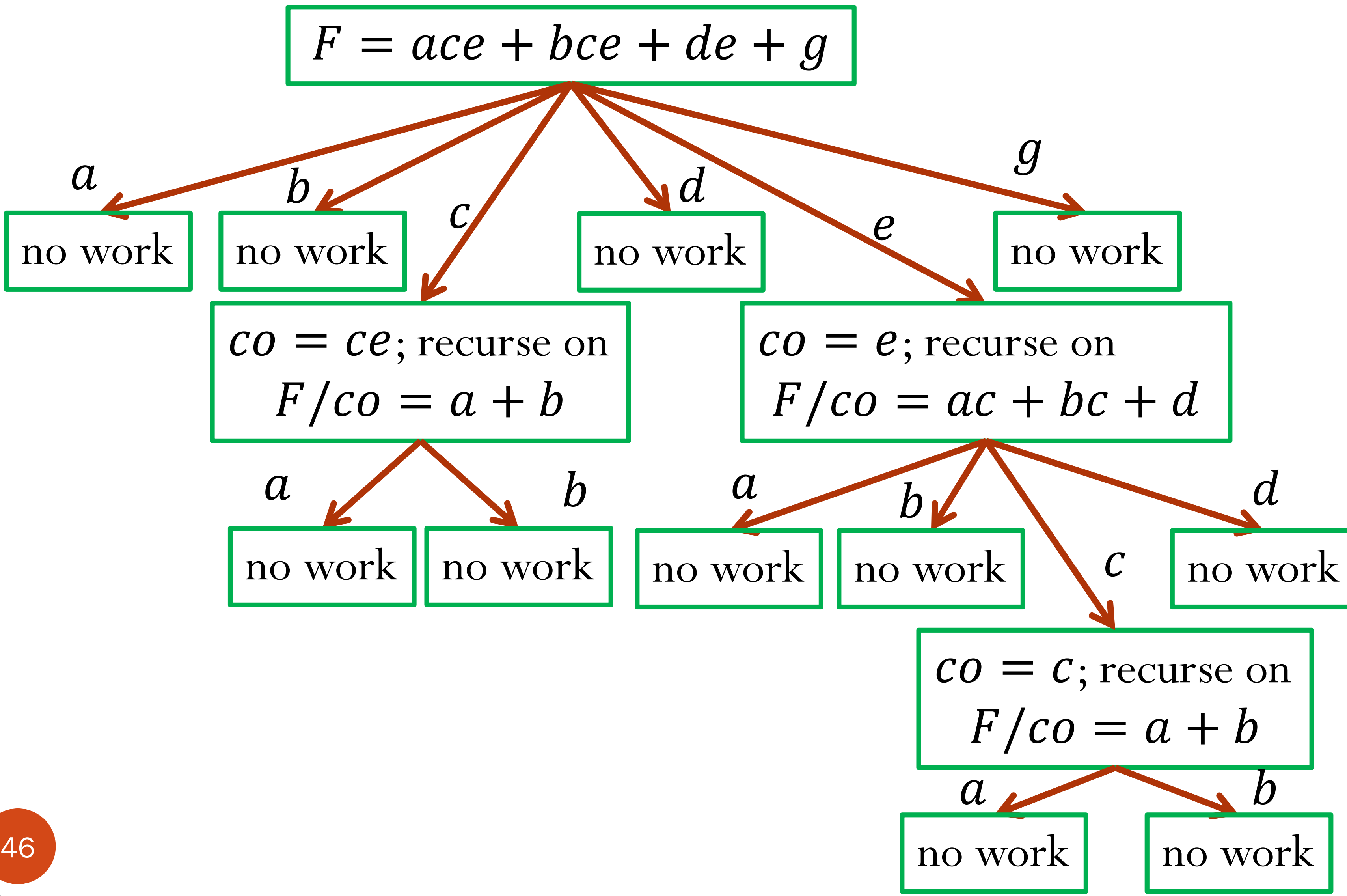
- a: only 1 cube with a, no work.
- b: only 1 cube with b, no work.
- c: two cubes ace and bce with c.
  - $co = ace \cap bce = ce$
  - $F/co = a + b$
  - **Recurse** on $a + b$
- d: only 1 cube with d, no work.
- e: three cubes ace, bce, and de with e.
  - $co = ace \cap bce \cap de = e$
  - $F/co = ac + bc + d$
  - **Recurse** on $ac + bc + d$
- g: only 1 cube with g, no work.

Kernelling Example (cont.)

- **Recurse** on $a + b$
  - No work for variables a and b, since one cube with a/b.

- **Recurse** on $ac + bc + d$
  - No work for variables a, b, d, since one cube with $a/b/d$.
  - c: two cubes ac and bc with c.
    - $co = ac \cap bc = c$
    - $F/co = a + b$
    - **Recurse** on $a + b$ (the same as above)

Kernelling Example (cont.)



Kernelling Example (cont.)

FindKernels(F):
for (each var **x** in **F**) {
    …
}
K = **K** ∪ **F** ;
return(**K**);

$F = ace + bce + de + g$

Kernels $K = \{a + b,$
$ac + bc + d,$
$ace + bce + de + g\}$

return $K = \{a + b,$
$ac + bc + d\}$

e

$co = e$; recurse on
$F/co = ac + bc + d$

c

return $K = \{a + b\}$

$co = ce$; recurse on
$F/co = a + b$

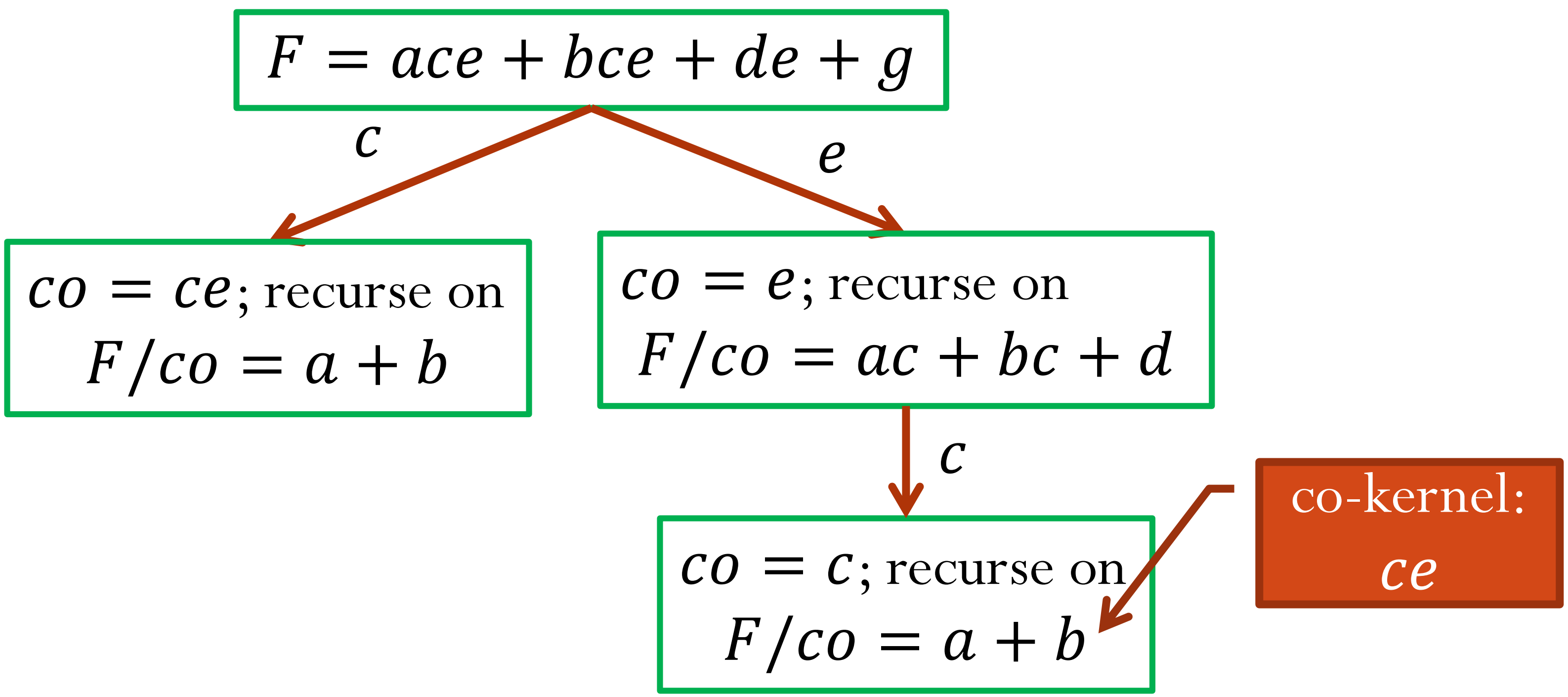
return $K = \{a + b\}$

c

$co = c$; recurse on
$F/co = a + b$

Get Co-Kernels

- With this algorithm …
  - Can find **all** the kernels and **co-kernels** too.
  - Get co-kernels by **ANDing** the divisor co cubes **up recursion tree**.



One Tiny Problem

$$\begin{tikzcd} & \boxed{F = ace + bce + de + g} \arrow[dl, "c"'] \arrow[dr, "e"] & \\ \boxed{F/co = a + b} & & \boxed{F/co = ac + bc + d} \arrow[d, "c"] \\ & & \boxed{F/co = a + b} \end{tikzcd}$$

- The algorithm will revisit same kernel **multiple** times.
  - Why? Kernel you get for co-kernel abc is same as for cba, but current algorithm **doesn't know this** and will find same kernel for both co-kernels.
- <u>Solution</u>: remember which variables already tried in co-kernels. A little extrabook keeping solves this.

Multilevel Synthesis Models: Summary

- **Boolean network model**
  - Like a gate network, but each node in network is an SOP form.
  - Supports many operations to add, reduce, simplify nodes in network.
- **Algebraic model** & **algebraic division**
  - Simplifies Boolean functions to behave like polynomials of reals.
  - Divides one Boolean function by another:

$$F = (\text{divisor } D)\bullet(\text{quotient } Q) + \text{remainder } R$$

- **Kernels** / **Co-kernels** of a function F
  - **Kernel** = **cube-free** quotient obtained by dividing by a single cube (**co-kernel**)
  - **Intersections** of kernels of two functions give all **multiple-cube common divisors** (Brayton & McMullen theorem).

Notes

- The **algebraic model** (and **division**) are not the only options.
  - There are also "**Boolean division**" models and algorithms that don't lose expressivity.
  - ..but they are more complex.
  - Rich universe of models & methods here.

Good References

- R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on CAD of ICs*, vol. CAD-6, no. 6, November 1987, pp. 1062-1081.
- Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

Next question:
what are the **best** common divisors to get?

How Do We Find Good Divisors?

- The operator is called **extraction**.
  - Want to extract either **single-cube divisor** or **multiple-cube divisor** from multiple expressions.
- How do we **extract** good divisors?
- Solution:
  - When you want a **single-cube divisor**, go look for **co-kernels**.
  - When you want a **multiple-cube divisor**, go look for **kernels**.

Approach Overview

- For **single cube extraction**
  - Build a very large matrix of **0s and 1s**
  - Heuristically look for "**prime rectangles**" in this matrix
  - Each such "prime" is a good common single-cube divisor
- For **multiple cube extraction**
  - Build a (different) very large matrix of **0s and 1s**
  - Heuristically look for "**prime rectangles**" in this matrix
  - Each such "prime" is a good multiple-cube divisor
- Surprisingly, a lot like Karnaugh maps!
  - Except we do it all algorithmically.

Single Cube Extract: Matrix Representation

- **<u>Given</u>**: a set of SOP Boolean equations (P,Q,R).
- Construct the **cube-literal matrix** as follows:
  - One row for each unique product term.
  - One column for each unique literal.
  - A "1" in the matrix if this product term uses this literal, else a "-".

$$P = abc + abd + eg$$
$$Q = abfg$$
$$R = bd + ef$$

		a 1	b 2	c 3	d 4	e 5	f 6	g 7
abc	1	1	1	1	-	-	-	-
abd	2	1	1	-	1	-	-	-
eg	3	-	-	-	-	1	-	1
abfg	4	1	1	-	-	-	1	1
bd	5	-	1	-	1	-	-	-
ef	6	-	-	-	-	1	1	-

Covering this Matrix: Prime Rectangles

- A **rectangle** of a cube-literal matrix is a set of rows R and columns C that has a '1' in **every** **row/column intersection**.
  - Need not be contiguous rows or columns in matrix. Any set of rows or columns is fine.

	a 1	b 2	c 3	d 4	e 5	f 6	g 7
abc 1	1	1	1	-	-	-	-
abd 2	1	1	-	1	-	-	-
eg 3	-	-	-	-	1	-	1
abfg 4	1	1	-	-	-	1	1
bd 5	-	1	-	1	-	-	-
ef 6	-	-	-	-	1	1	-

Covering this Matrix: Prime Rectangles

- A **prime rectangle** is a rectangle that cannot be made any bigger by adding another row or a column.

		a 1	b 2	c 3	d 4	e 5	f 6	g 7
abc	1	1	1	1	-	-	-	-
abd	2	1	1	-	1	-	-	-
eg	3	-	-	-	-	1	-	1
abfg	4	1	1	-	-	-	1	1
bd	5	-	1	-	1	-	-	-
ef	6	-	-	-	-	1	1	-

Prime Rectangle Columns = Divisor!

- **Primes** are "biggest possible" common single-cube divisors.
  - **Makes sense**: columns of the prime rectangle tell you the literals in the single-cube divisor, while rows tell you which product terms you can divide!

		a 1	b 2	c 3	d 4	e 5	f 6	g 7
abc	1	1	1	1	-	-	-	-
abd	2	1	1	-	1	-	-	-
eg	3	-	-	-	-	1	-	1
abfg	4	1	1	-	-	-	1	1
bd	5	-	1	-	1	-	-	-
ef	6	-	-	-	-	1	1	-

Single-cube divisor:

$$X = ab$$

Prime Rectangle Columns = Divisor!

	a 1	b 2	c 3	d 4	e 5	f 6	g 7
abc 1	1	1	1	-	-	-	-
abd 2	1	1	-	1	-	-	-
eg 3	-	-	-	-	1	-	1
abfg 4	1	1	-	-	-	1	1
bd 5	-	1	-	1	-	-	-
ef 6	-	-	-	-	1	1	-

Single-cube divisor:

$X = ab$

$$P = abc + abd + eg$$
$$Q = abfg$$
$$R = bd + ef$$

$$P = Xc + Xd + eg$$
$$Q = Xfg$$
$$R = bd + ef$$
$$X = ab$$

Simple Bookkeeping to Track # Literals

- Recall: we factor & extract to **reduce literals** in logic network.
  - Would be nice if there was a simple formula to compute this.
- Indeed, there is:
  - Start with a prime rectangle.
  - Let C = # columns in rectangle.
  - For each row r in rectangle: let $\text{Weight}(r)$ = # times this product appears in network.
  - Compute $L = (C-1) \times [\sum_{\text{rows } r} \text{Weight}(r)] - C$.
- **Nice result**: for a prime rectangle, L = **# literals saved**
  - **To be precise**: if you count literals before extracting this single-cube divisor, and after, L is how many literals are saved.

Compute Saved Literals: Example

$R = abw + wz$

$S = abw + aby$

Build Matrix

	a 1	b 2	w 3	y 4	z 5
abw 1	1	1	1	-	-
wz 2	-	-	1	-	1
aby 3	1	1	-	1	-

Extraction

Original # literals: 11

$X = ab$

$R = Xw + wz$

$S = Xw + Xy$

 # saved: 1

After extraction # literals: 10

Compute Saved Literals: Example

$R = abw + wz$

$S = abw + aby$

	a 1	b 2	w 3	y 4	z 5
abw 1	1	1	1	-	-
wz 2	-	-	1	-	1
aby 3	1	1	-	1	-

Result by Counting:
 # saved: 1

- Now apply formula $L = (C - 1) \times [\sum_{\text{rows } r} \text{Weight}(r)] - C$
  - C = # columns in rectangle $\Rightarrow 2$
  - Weight(abw) $\Rightarrow 2$ (appear twice in the network)
  - Weight(aby) $\Rightarrow 1$ (appear once in the network)
  - $L = (2 - 1) \times (2 + 1) - 2 = 1$ **Correct!**

How About Multiple-Cube Factors?

- Remarkably, a very similar matrix-rectangle-prime concept.
  - Make an appropriate **matrix**. Find **prime rectangle**. Do literal count **bookkeeping** with numbers associated with rows/columns.
- Given: A set of Boolean functions (nodes in a network)

$$P = af + bf + ag + cg + ade + bde + cde$$
$$Q = af + bf + ace + bce$$
$$R = ade + cde$$

- First: find **kernels** of each of these functions.
  - Why? Brayton-McMullen theorem: Multiple-cube factors are **intersections of the product terms in the kernels** for each of these functions.

Kernels / Co-Kernels of P,Q,R Example

- $P = af + bf + ag + cg + ade + bde + cde$
  - Co-kernel a, kernel $de + f + g$
  - Co-kernel b, kernel $de + f$
  - Co-kernel c, kernel $de + g$
  - Co-kernel de, kernel $a + b + c$
  - Co-kernel f, kernel $a + b$
  - Co-kernel g, kernel $a + c$
  - Co-kernel 1, kernel $af + bf + ag + cg + ade + bde + cde$ **(trivial, ignore)**

Kernels / Co-Kernels of P,Q,R Example

- $Q = af + bf + ace + bce$
  - Co-kernel a, kernel $ce + f$
  - Co-kernel b, kernel $ce + f$
  - Co-kernel ce, kernel $a + b$
  - Co-kernel f, kernel $a + b$
  - Co-kernel 1, kernel $af + bf + ace + bce$ **(trivial, ignore)**

- $R = ade + cde$
  - Co-kernel de, kernel $a + c$
  - Note: R is not its own kernel, why?

New Matrix: Co-Kernel-Cube Matrix

- One row for each **unique** (function, co-kernel) **pair** in problem.
- One column for each **unique cube** among all kernels in problem.

P: co-kernel a, kernel $de + f + g$
P: co-kernel b, kernel $de + f$
P: co-kernel c, kernel $de + g$
P: co-kernel de, kernel $a + b + c$
P: co-kernel f, kernel $a + b$
P: co-kernel g, kernel $a + c$
Q: co-kernel a, kernel $ce + f$
Q: co-kernel b, kernel $ce + f$
Q: co-kernel ce, kernel $a + b$
Q: co-kernel f, kernel $a + b$
R: co-kernel de, kernel $a + c$

			a 1	b 2	c 3	ce 4	de 5	f 6	g 7
P	a	1							
P	b	2							
P	c	3							
P	de	4							
P	f	5				?			
P	g	6							
Q	a	7							
Q	b	8							
Q	ce	9							
Q	f	10							
R	de	11							

Entries in the Co-Kernel-Cube Matrix

- For each **row**, take the co-kernel, go look at the associated kernel.
- Look at **cubes** in this kernel: put "1" in columns that are cubes in this kernel; else put "-"

P: co-kernel a, kernel $de + f + g$
P: co-kernel b, kernel $de + f$
P: co-kernel c, kernel $de + g$
P: co-kernel de, kernel $a + b + c$
P: co-kernel f, kernel $a + b$
P: co-kernel g, kernel $a + c$
Q: co-kernel a, kernel $ce + f$
Q: co-kernel b, kernel $ce + f$
Q: co-kernel ce, kernel $a + b$
Q: co-kernel f, kernel $a + b$
R: co-kernel de, kernel $a + c$

			a 1	b 2	c 3	ce 4	de 5	f 6	g 7
P	a	1	-	-	-	-	1	1	1
P	b	2	-	-	-	-	1	1	-
P	c	3	-	-	-	-	1	-	1
P	de	4	1	1	1	-	-	-	-
P	f	5	1	1	-	-	-	-	-
P	g	6	1	-	1	-	-	-	-
Q	a	7	-	-	-	1	-	1	-
Q	b	8	-	-	-	1	-	1	-
Q	ce	9	1	1	-	-	-	-	-
Q	f	10	1	1	-	-	-	-	-
R	de	11	1	-	1	-	-	-	-

Entries in the Co-Kernel-Cube Matrix

- Each row gives the kernel of the function (e.g., P) obtained by dividing the co-kernel (e.g., a).

P: co-kernel a, kernel $de + f + g$

			a 1	b 2	c 3	ce 4	de 5	f 6	g 7
P	a	1	-	-	-	-	1	1	1
P	b	2	-	-	-	-	1	1	-
P	c	3	-	-	-	-	1	-	1
P	de	4	1	1	1	-	-	-	-
P	f	5	1	1	-	-	-	-	-
P	g	6	1	-	1	-	-	-	-
Q	a	7	-	-	-	1	-	1	-
Q	b	8	-	-	-	1	-	1	-
Q	ce	9	1	1	-	-	-	-	-
Q	f	10	1	1	-	-	-	-	-
R	de	11	1	-	1	-	-	-	-

Prime Rectangles in Co-Kernel-Cube Matrix

- Prime rectangle is again a good divisor: now multiple cube
  - Create the multiple cube divisor as **sum** (OR) of cubes of prime rectangle **columns**.

	a 1	b 2	c 3	ce 4	de 5	f 6	g 7
P a 1	-	-	-	-	1	1	1
P b 2	-	-	-	-	1	1	-
P c 3	-	-	-	-	1	-	1
P de 4	1	1	1	-	-	-	
P f 5	1	1	-	-	-	-	-
P g 6	1	-	1	-	-	-	-
Q a 7	-	-	-	1	-	1	-
Q b 8	-	-	-	1	-	1	-
Q ce 9	1	1	-	-	-	-	-
Q f 10	1	1	-	-	-	-	-
R de 11	1	-	1	-	-	-	-

(a+b) is the multiple cube divisor!

P = (de)•(a+b+stuff1) + rem1

P = (f)•(a+b+stuff2) + rem2

Q = (ce)•(a+b+stuff3) + rem3

Q = (f)•(a+b+stuff4) + rem4

Simple Formula to Get # Literals Saved

- For each column c in rectangle: let $\text{Weight}(c) = \#$ literals in column cube.
- For each row r in rectangle: let $\text{Weight}(r) = 1 + \#$ literals in co-kernel label.
- For each "1" covered at row r and column c: AND row co-kernel and column cube; let $\text{Value}(r, c) = \#$ literals in this new ANDed product.
- **# literals saved** is

$$L = \sum_{\text{row } r} \sum_{\text{col } c} \text{Value}(r, c) - \sum_{\text{row } r} \text{Weight}(r) - \sum_{\text{col } c} \text{Weight}(c)$$

Compute Saved Literals: Example

$P = af + bf + ag + cg + ade + bde + cde$

$Q = af + bf + ace + bce$

$R = ade + cde$

Original # literals: 33

Build Matrix

			a 1	b 2	c 3	ce 4	de 5	f 6	g 7
P	a	1	-	-	-	-	1	1	1
P	b	2	-	-	-	-	1	1	-
P	c	3	-	-	-	-	1	-	1
P	de	4	1	1	1	-	-	-	-
P	f	5	1	1	-	-	-	-	-
P	g	6	1	-	1	-	-	-	-
Q	a	7	-	-	-	1	-	1	-
Q	b	8	-	-	-	1	-	1	-
Q	ce	9	1	1	-	-	-	-	-
Q	f	10	1	1	-	-	-	-	-
R	de	11	1	-	1	-	-	-	-

Extraction

$X = a + b$

$P = Xf + Xde + ag + cg + cde$

$Q = Xf + Xce$

$R = ade + cde$

After extraction # literals: 25

 # saved: 8

Compute Saved Literals: Example

			a 1	b 2	c 3	ce 4	de 5	f 6	g 7
P	a	1	-	-	-	-	1	1	1
P	b	2	-	-	-	-	1	1	-
P	c	3	-	-	-	-	1	-	1
P	de	4	1	1	1	-	-	-	-
P	f	5	1	1	-	-	-	-	-
P	g	6	1	-	1	-	-	-	-
Q	a	7	-	-	-	1	-	1	-
Q	b	8	-	-	-	1	-	1	-
Q	ce	9	1	1	-	-	-	-	-
Q	f	10	1	1	-	-	-	-	-
R	de	11	1	-	1	-	-	-	-

$P = af + bf + ag + cg + ade + bde + cde$

$Q = af + bf + ace + bce$

$R = ade + cde$

 # saved: 8

- Column weight
  - Weight(a) = #literals in "a" ⇒ 1
  - Weight(b) = #literals in "b" ⇒ 1
- Row weight
  - Weight((P, de)) = 1+ #literals in "de" ⇒ 3
  - Weight((P, f)) = 1+ #literals in "f" ⇒ 2
  - Weight((Q, ce)) = 1+ #literals in "ce" ⇒ 3
  - Weight((Q, f)) = 1+ #literals in "f" ⇒ 2

Compute Saved Literals: Example

			a 1	b 2	c 3	ce 4	de 5	f 6	g 7
P	a	1	-	-	-	-	1	1	1
P	b	2	-	-	-			-	-
P	c	3	-	-	-		1		1
P	de	4	1	1	1	3	3		-
P	f	5	1	1	-	2	2		-
P	g	6	1	-	1	-	-	-	-
Q	a	7	-	-	-	1	-	1	-
Q	b	8	-	-	-	1	-	1	-
Q	ce	9	1	1	-	3	3		-
Q	f	10	1	1	-	2	2		-
R	de	11	1	-	1	-	-	-	-

Values

 # saved: 8

- Column weight
  - Weight(a) = 1; Weight(b) = 1
- Row weight
  - Weight((P, de)) = 3; Weight((P, f)) = 2
  - Weight((Q, ce)) = 3; Weight((Q, f)) = 2
- Value(r,c): # literals in the **product** of **row co-kernel** and **column cube**.
- Apply formula $L = \sum_{\text{row } r} \sum_{\text{col } c} \text{Value}(r, c) - \sum_{\text{row } r} \text{Weight}(r) - \sum_{\text{col } c} \text{Weight}(c)$
  $= 20 - 10 - 2 = 8$

Correct!

Details for Both Single/Multiple Cube Extraction

- You can extract a **second**, **third**, etc., divisor using same matrix.
  - Works for both single-cube and multiple-cube divisors.

- …but must **update** matrix to reflect new Boolean logic network.
  - Because the node contents are different, and there is a new divisor node in network.
  - For multiple-cube case, must **kernel** new divisor nodes to update matrix.
  - All mechanical. A bit tedious. Just skip it…
  - For us: just know how to **extract first good divisor** is good enough.

How to Find Prime Rectangle in Matrix?

- **Greedy heuristics** work well for this rectangle covering problem.
  - Start with a single row rectangle with "good #literal savings".
  - Grow the rectangle alternatively by adding more rows, more columns.

- Example: **Rudell's Ping Pong heuristic**.
  - From his Berkeley PhD dissertation in 1989.
  - **Very good** heuristic:
    - **< 1%** of optimal result.
    - **10~100x faster** than brute force approach.

Rudell's Ping Pong Heuristic

1. Pick the **best single row** (the 1-row rectangle with best #literals saved).
2. Look at other rows **with 1s in same places** (may have more 1s). Add the one that **maximizes** #literals saved. Iterate until can't find any more.
3. Look at other columns **with 1s in same places** (may have more 1s). Add the one that **maximizes** # literals saved. Iterate until can't find any more.
4. Go to 2.
5. Quit when can't grow rectangle any more in any direction.

Extraction: Summary

- **Single cube extraction**
  - Build the cube-literal matrix.
  - Each prime rectangle is a good **single cube divisor**.
  - Simple bookkeeping lets us obtain savings in #literals.
- **Multiple cube extraction**
  - Kernel all the expressions in network; build the co-kernel-cube matrix.
  - Each prime rectangle is a good **multiple cube divisor**.
  - Simple bookkeeping lets us obtain savings in #literals.
- Mechanically, both are **rectangle covering** problems (very like Karnaugh maps!)
  - Good **heuristics** to obtain a good prime rectangle, fast and effective.

Aside: How to We Really Do This?

- Do **not** use rectangle covering on **all** kernels/co-kernels
  - Too expensive to do rectangle problem on big circuits (>20K gates)
  - Too expensive to go compute **complete** set of kernels, co-kernels

- Often use heuristics to find a "**quick**" set of likely divisors.
  - Don't fully kernel each node of network: too many cubes to consider. Instead, can extract a **subset** of useful kernels quickly.
  - Then, can either do rectangle cover on these smaller problems (smaller since fewer things to consider in covering problem)…
  - …or, try to do simpler overall network restructuring, e.g., try all pairwise **substitutions** of one node into another node: keep good ones, continue in a greedy way.

References

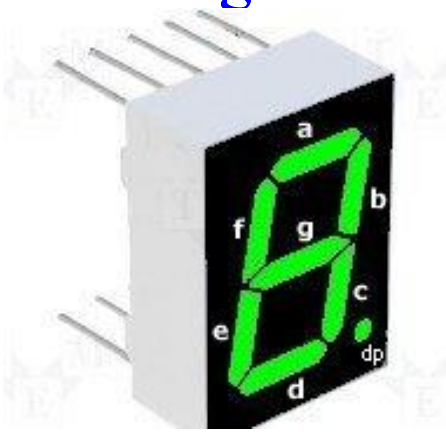
- R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on CAD of ICs*, vol. CAD-6, no. 6, November 1987, pp. 1062-1081.
- Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
- R.K. Brayton, R. Rudell, A.S. Vincentelli, and A. Wang, "Multi-Level Logic Optimization and the Rectangular Covering Problem," Proceedings of the International Conference on Computer Aided Design, pp. 66-69, 1987.
- Richard Rudell, *Logic Synthesis for VLSI Design*, PhD Thesis, Dept of EECS, University of California at Berkeley, 1989.
- Srinivas Devadas, Abhijit Gosh, Kurt Keutzer, *Logic Synthesis*, McGraw Hill Inc., 1994.

Don’t Cares

- We made progress on multi-level logic by **simplifying** the model.
  - Algebraic model: we **get rid of** a lot of “difficult” Boolean behaviors.
  - But we lost some optimality in the process.
- How do we put it back? One surprising answer: **Don’t cares**
  - To help this, **extract** don’t cares from “surrounding logic,” use them **inside each node**.
- The big difference in multi-level logic
  - Don’t cares happen as a natural byproduct of Boolean network model: called **Implicit Don’t Cares**.
  - They are all over the place, in fact. Very useful for simplification.
  - But they are **not explicit**. We have to **go hunt for them**…

Don't Cares Review: 2-Level

- In basic digital design…
  - Don't Care (DC) = an input pattern that **can never happen** or you don't care the output if it happens.
  - Example: use **binary-coded decimals (BCD)** to control **seven-segment digital tube**.



How about input (x,y,z,w) =(1,0,1,0),(1,0,1,1) …?

Don't care!

x y z w	decimal value	segment a
0 0 0 0	0	1
0 0 0 1	1	0
0 0 1 0	2	1
0 0 1 1	3	1
0 1 0 0	4	0
0 1 0 1	5	1
0 1 1 0	6	1
0 1 1 1	7	1
1 0 0 0	8	1
1 0 0 1	9	1

Don't Cares Review: 2-Level

- Since patterns (x,y,z,w)=(1,0,1,0), (1,0,1,1), (1,1,0,0), (1,1,0,1), (1,1,1,0), (1,1,1,1) are don't cares, we are **free to decide** whether F=1 or 0, to better **optimize** F.

x y z w	decimal value	segment a
0 0 0 0	0	1
0 0 0 1	1	0
0 0 1 0	2	1
0 0 1 1	3	1
0 1 0 0	4	0
0 1 0 1	5	1
0 1 1 0	6	1
0 1 1 1	7	1
1 0 0 0	8	1
1 0 0 1	9	1

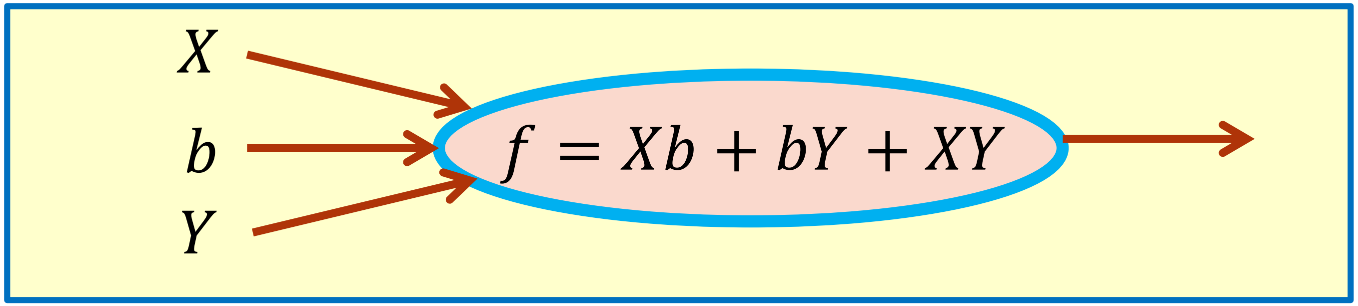
zw \ xy	00	01	11	10
00	1	0	d	1
01	0	1	d	1
11	1	1	d	d
10	1	1	d	d

Don’t Cares (DCs): Multi-level

- What’s different in multi-level?
  - DCs arise **implicitly**, as a result of the **Boolean logic network structure**.
  - We must go find these implicit don’t cares – we must search for them explicitly.

Multi-level DCs: Informal Tour

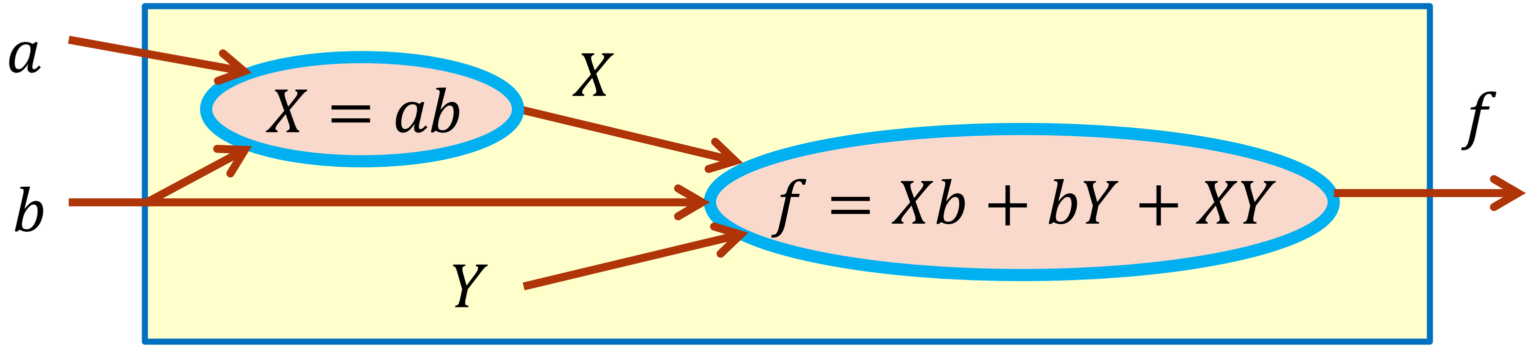
- Suppose we have a Boolean network and a node f in the network.



- Can we say anything about **don't cares** for node f?
  - No. We don't know any "context" for surrounding parts of network.
  - As far as we can tell, all patterns of inputs (X,b,Y) are possible.
  - We **cannot further simplify** the expression for f.

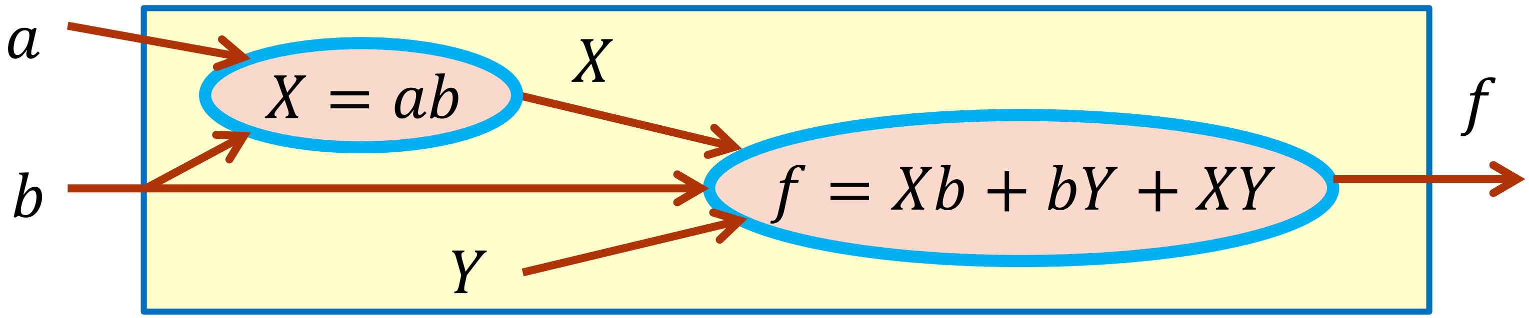
Multi-level DCs: Informal Tour

- Now suppose we know something about input X to f:
  - Node $X = ab$.
  - Also assume a and b are **primary inputs (PIs)** and f is **primary output (PO)**.



- Now can we say something about DCs for node f …?
  - **YES!**
  - Because there are some **impossible patterns** of (X, b, Y).

Multi-level DCs: Informal Tour



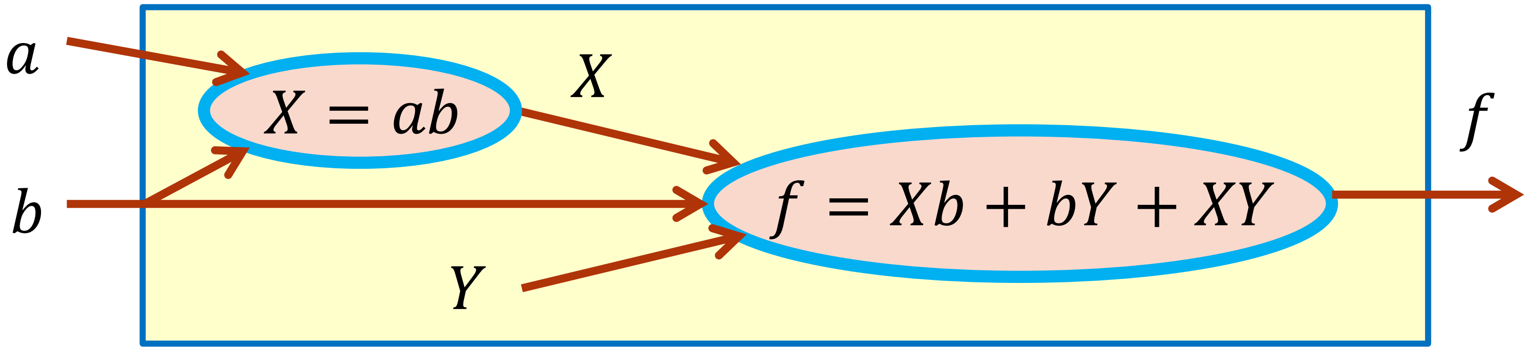
The possible input/output patterns for node X

a	b	X	Can it occur?
0	0	0	Yes
0	0	1	No
0	1	0	Yes
0	1	1	No
1	0	0	Yes
1	0	1	No
1	1	0	No
1	1	1	Yes

b	X	Can it occur?
0	0	Yes
0	1	No
1	0	Yes
1	1	Yes

Impossible patterns for (X, b, Y) are: (1, 0, 0) and (1, 0, 1)

Multi-level DCs: Informal Tour



- Impossible patterns for (X, b, Y) are (1, 0, 0) and (1, 0, 1).
  - With them, we can simplify f.

Can be simplified as
$f = X + bY$

Kmap for $f = Xb + bY + XY$

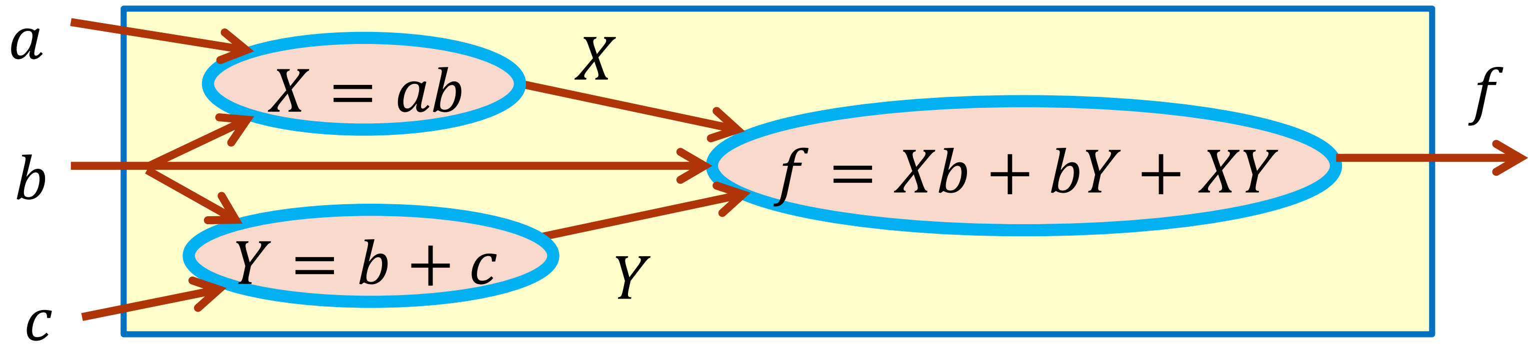
Y \ Xb	00	01	11	10
0			1	
1		1	1	1

With don't cares

Y \ Xb	00	01	11	10
0			1	d
1		1	1	d

Multi-level DCs: Informal Tour

- Now further suppose $Y = b + c$. What will happen?

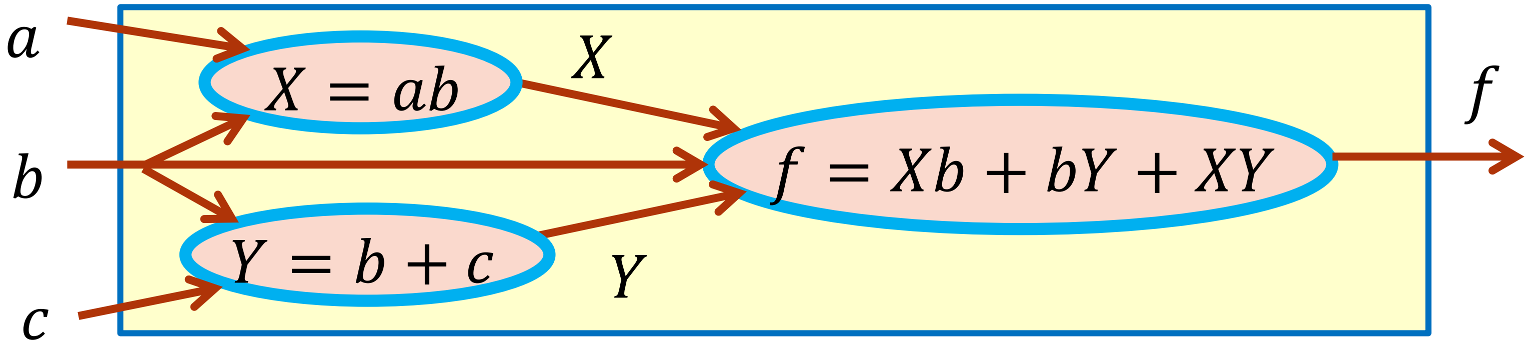


b	c	Y	Can it occur?
0	0	0	Yes
0	0	1	No
0	1	0	No
0	1	1	Yes
1	0	0	No
1	0	1	Yes
1	1	0	No
1	1	1	Yes

b	Y	Can it occur?
0	0	Yes
0	1	Yes
1	0	No
1	1	Yes

Impossible patterns for (X, b, Y) are:
(0, 1, 0) and (1, 1, 0)

Multi-level DCs: Informal Tour



- Impossible patterns for (X, b, Y) are
  - (1,0,0), (1,0,1) (From $X = ab$)
  - (0,1,0), (1,1,0) (From $Y = b + c$)

f can be simplified as $f = b$

Kmap for $f = Xb + bY + XY$

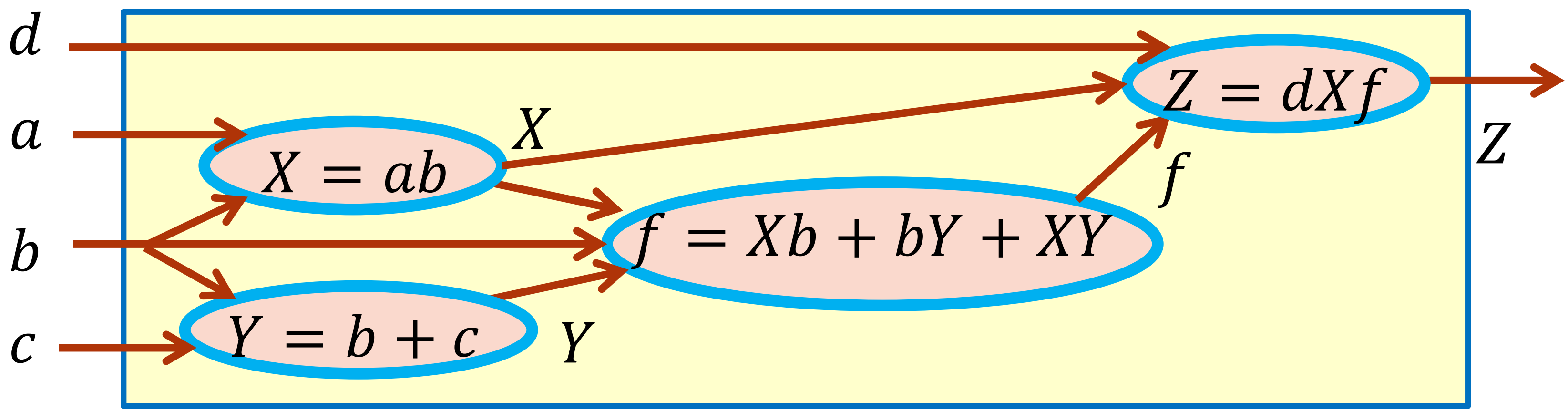
Y \ Xb	00	01	11	10
0			1	
1		1	1	1

With don't cares

Y \ Xb	00	01	11	10
0		d	d	d
1		1	1	d

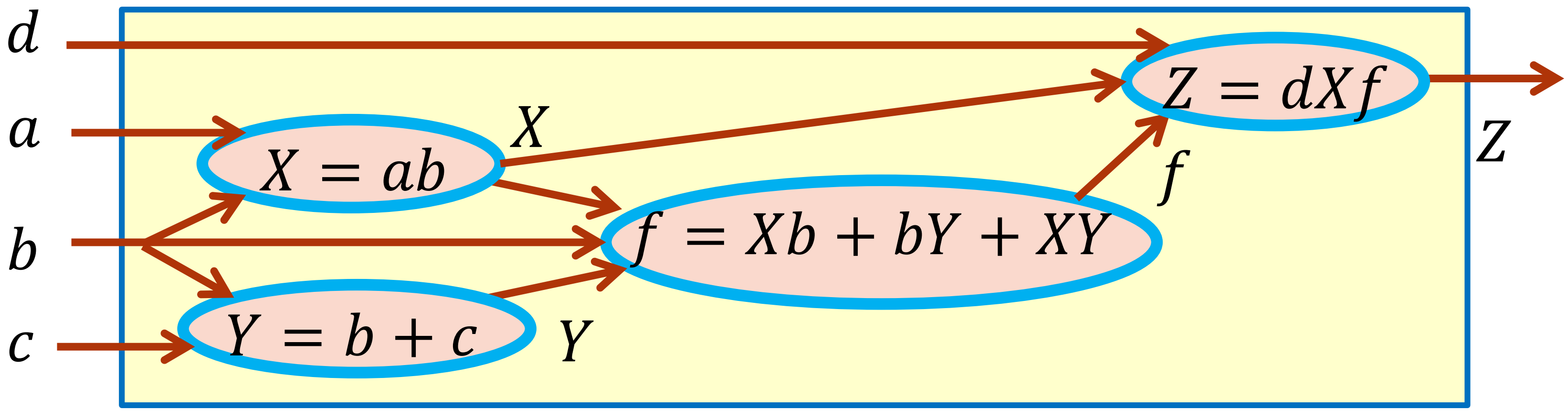
Multi-level DCs: Informal Tour

- Now suppose f is **<u>not</u>** a **primary output**, Z is.



- **<u>Question</u>**: when does the value of the output of node f actually **affect** the primary output Z?
  - Or, said **<u>conversely</u>**: When does it **not matter** what f is?
  - Let's go look at patterns of (f, X, d) at node Z…

When Is Z "Sensitive" to Value of f?

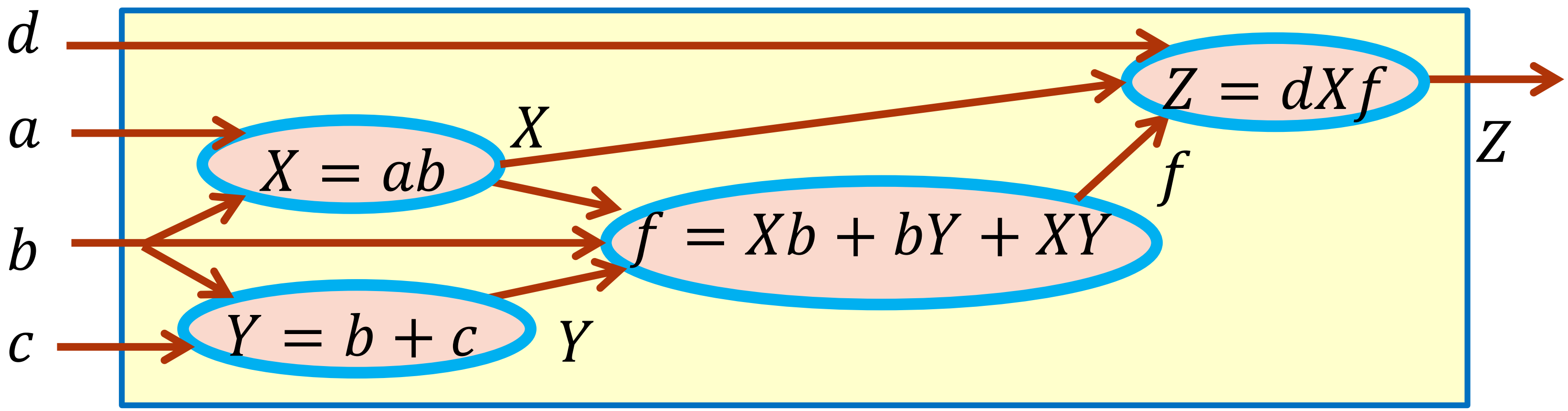


f	X	d	Z	Does f affect Z?
0	0	0	0	No
1	0	0	0	
0	0	1	0	No
1	0	1	0	
0	1	0	0	No
1	1	0	0	
0	1	1	0	Yes
1	1	1	1	

Can we use this information to find new patterns of (X, b, Y) to help us simplify f further?

YES!

When Is Z “Sensitive” to Value of f?



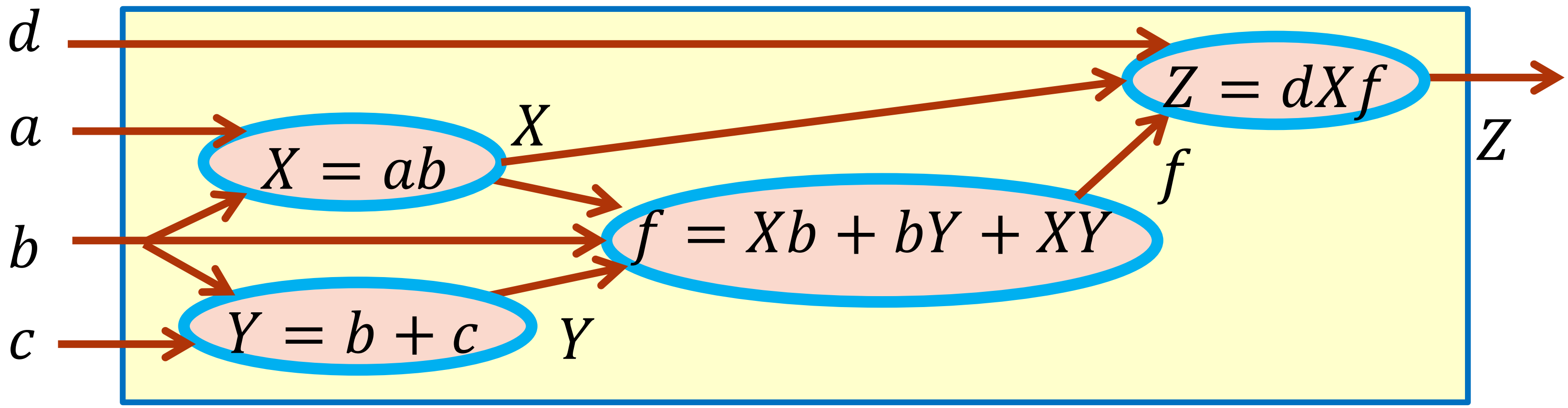
f	X	d	Z	Dose f affect Z?
0	0	0	0	No
1	0	0	0	
0	0	1	0	No
1	0	1	0	
0	1	0	0	No
1	1	0	0	
0	1	1	0	Yes
1	1	1	1	

What patterns at **input to** f node (i.e., (X, b, Y)) are DCs, because those patterns make Z output **insensitive** to changes in f?

$$(X, b, Y) = (0, -, -)$$

This means when $X = 0$, we can set f to any value – it **won’t change** Z.
So $(X, b, Y) = (0, -, -)$ is DC of f!

Multi-level DCs: Informal Tour



- So, we can use this **new** DC pattern $(0, -, -)$ to simplify f further…
  - … with previous DC patterns (1,0,0), (1,0,1), (0,1,0), (1,1,0).

Kmap for $f = Xb + bY + XY$

Y \ Xb	00	01	11	10
0			1	
1		1	1	1

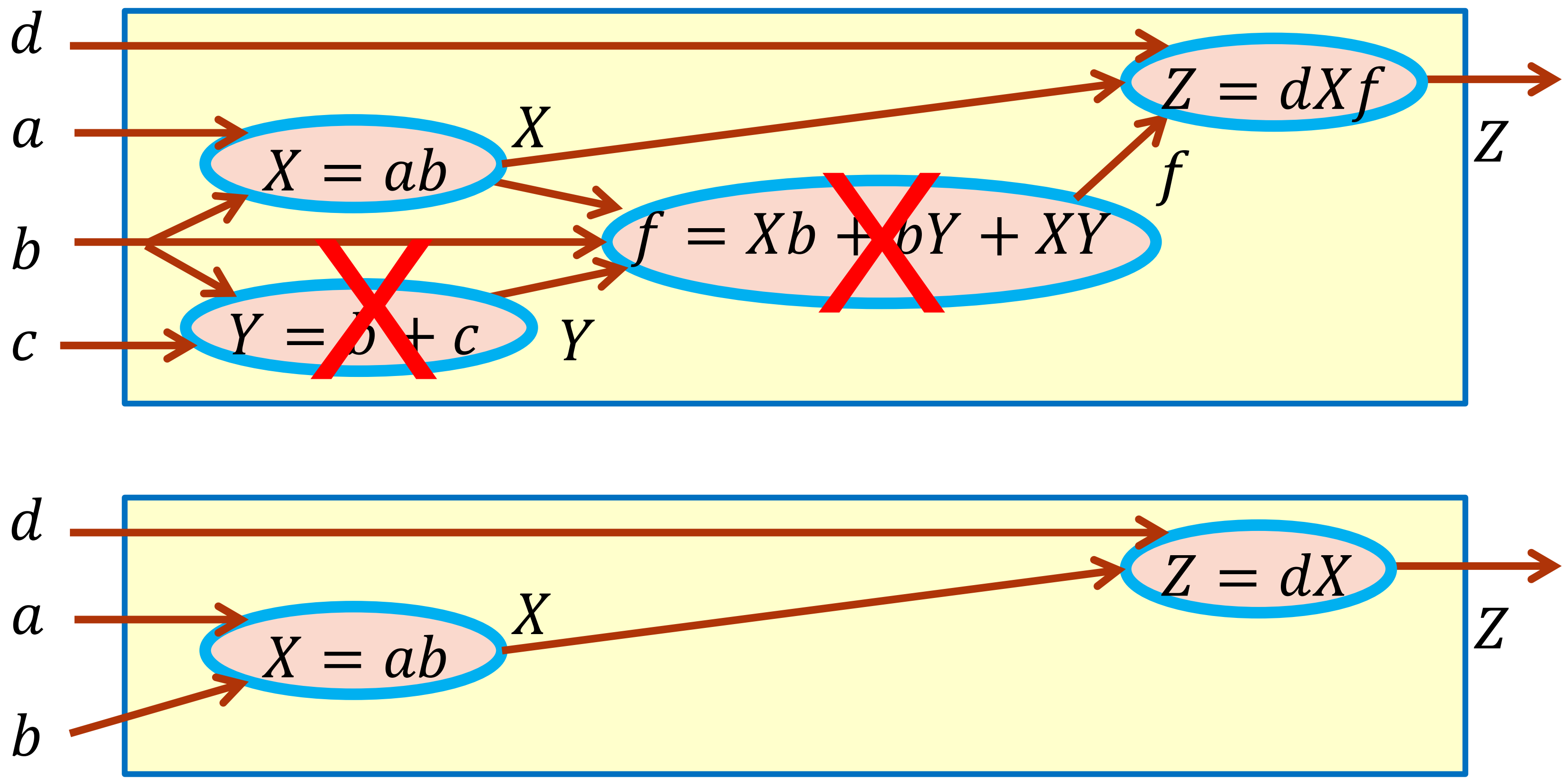
With don't cares

f simplified as 1

Y \ Xb	00	01	11	10
0	d	d	d	d
1	d	1	1	d

Final Result: Multi-level DC Tour

- What happened to f?
  - Due to network **context**, it **disappeared** ($f = 1$)!



Summary

- Don't Cares are **implicit** in the Boolean network model.
  - They arise from the **graph structure** of the multilevel Boolean network model itself.
- Implicit Don't Cares are **powerful**.
  - They can greatly help simplify the 2-level SOP structure of any node.
- Implicit Don't Cares require **computational work** to find.
  - For this example, we just "stared at the logic" to find the DC patterns.
  - We need some **algorithms** to do this automatically!
  - This is what we need to study next …

Multi-Level Don't Cares

- Don't Cares are **implicit** in the Boolean network model.
  - They arise from the **graph structure** of the multilevel Boolean network model itself.
- Implicit Don't Cares are **powerful**.
  - They can greatly help simplify the 2-level SOP structure of any node.
- Implicit Don't Cares require **computational work** to find.
  - We need some **algorithms** to do this automatically!

3 Types of Implicit DCs

- **Satisfiability** don't cares: **SDCs**
  - Belong to the **wires** inside the Boolean logic network.
  - Used to compute **controllability** don't cares (below).

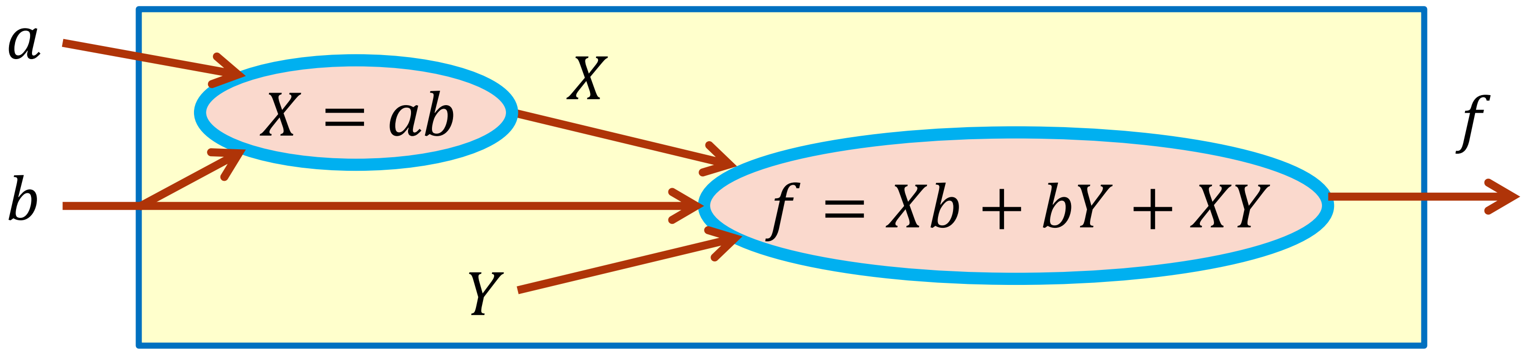
- **Controllability** don't cares: **CDCs**
  - Patterns that **cannot happen at inputs** to a network node.

- **Observability** don't cares: **ODCs**
  - Patterns that "**mask**" outputs.

Controllability don’t cares: CDCs

- Patterns that **cannot happen at inputs** to a network node.

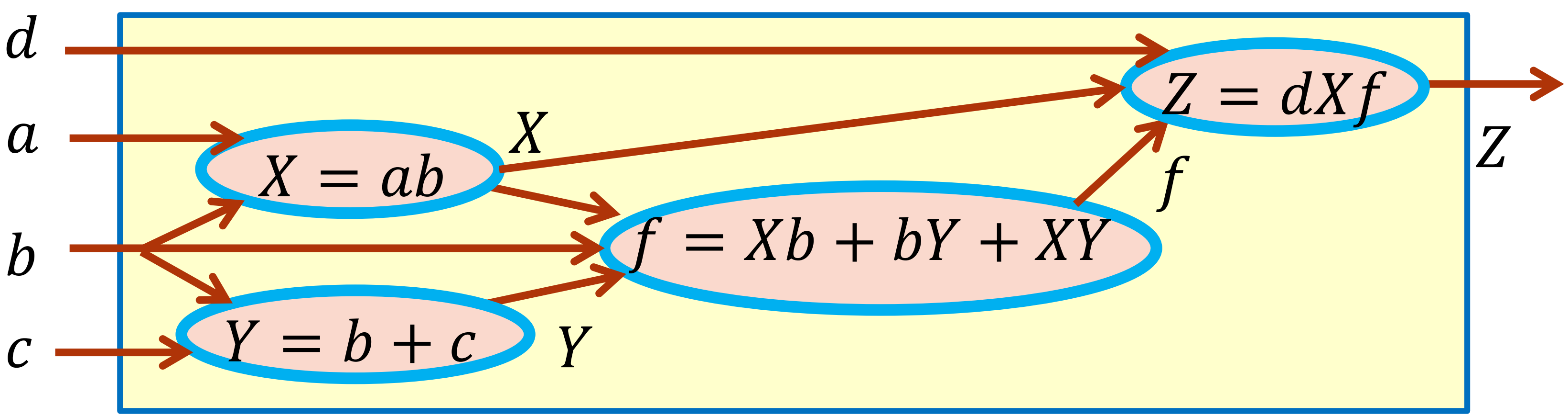
- **Example**
  - For node f, $(X, b, Y) = (1,0,0), (1,0,1)$ are CDCs.



Observability don't cares: ODCs

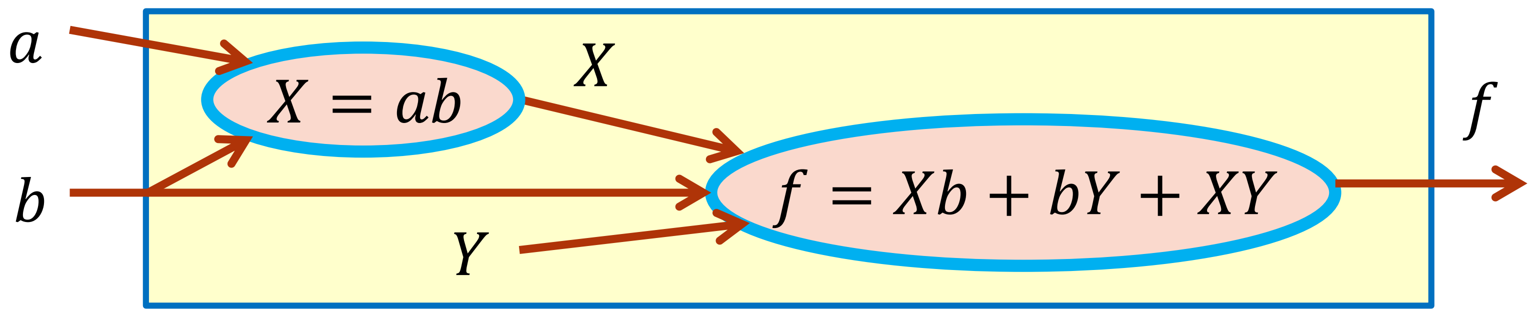
- Input patterns to node that make primary outputs **insensitive** to output of the node.
  - Patterns that "**mask**" outputs.

- **Example**
  - For node f, $(X, b, Y) = (0, -, -)$ is ODC.



Background: Representing DC Patterns

- How shall we **represent** DC patterns at a node?
  - **Answer**: As a **Boolean function** that makes a 1 when the inputs are **these DCs**.
  - This is often called a **Don't Care Cover**.



Don't care pattern of (X,b,Y)=(1,0,0), (1,0,1)

The don't care cover is $X\bar{b}\bar{Y} + X\bar{b}Y = X\bar{b}$

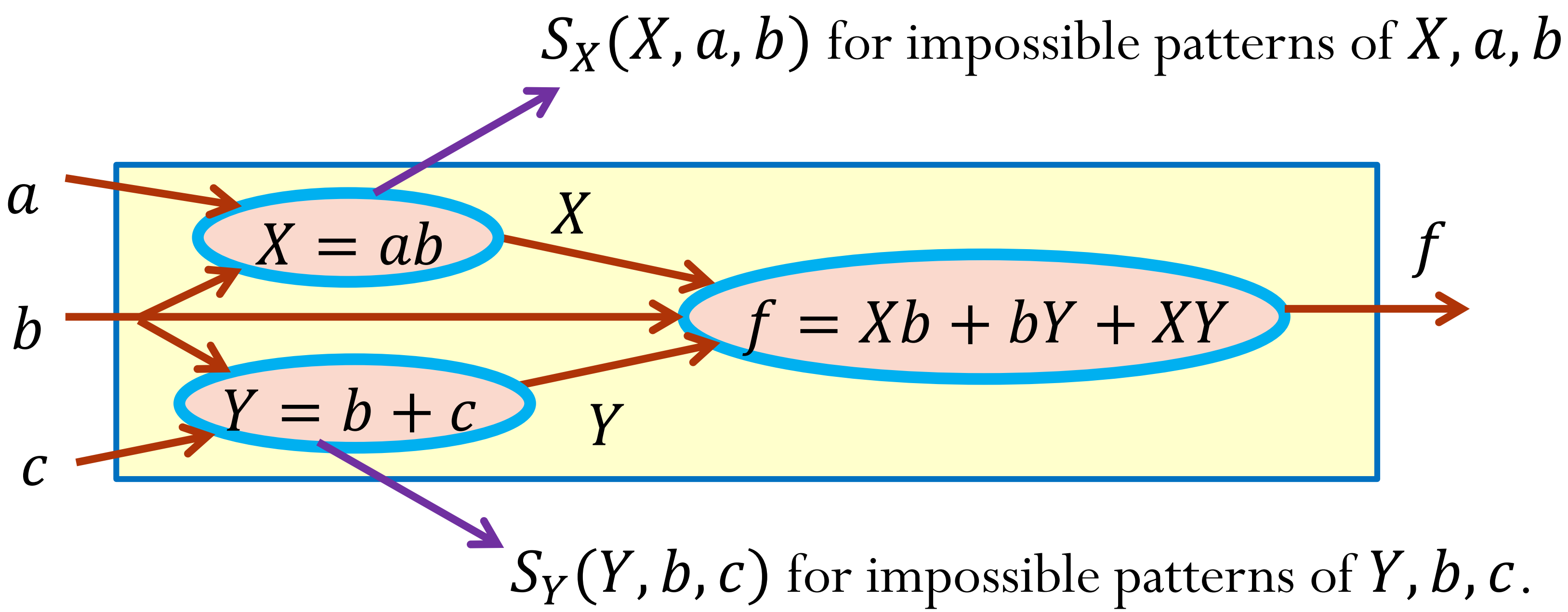
Background: Representing DC Patterns

- So, each SDC, CDC, ODC is really just another Boolean function, in this strategy.

- Why do it like this?
  - Because we can use all the other **computational Boolean algebra** techniques we know (e.g., BDDs), to **solve** for, and to **manipulate** the DC patterns.
  - This turns out to be hugely important to making the computation practical.

SDCs: They "Belong" to the Wires

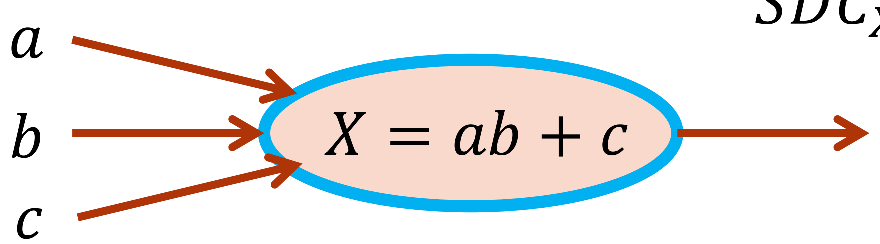
- One SDC for every **internal wire** in Boolean logic network.
  - The SDC represents **impossible** patterns of **inputs to, and output of**, each node.
  - If the node function is F, with inputs a, b, c, write as: $S_F(F, a, b, c)$.



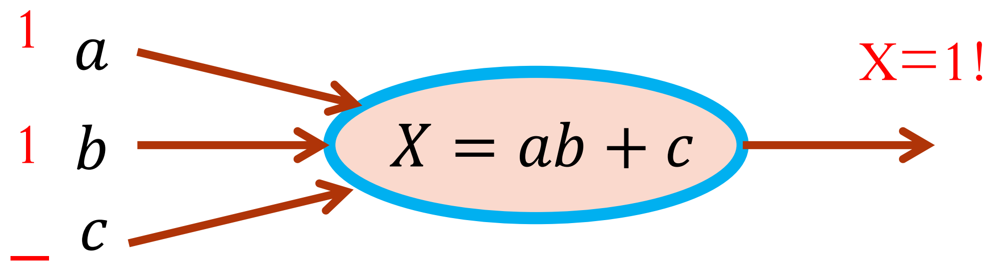
SDCs: How to Compute

- Compute an SDC for each output wire from each internal Boolean node.
- You want an expression that is 1 when output X **does not equal** the Boolean expression for X.
  - This is just: $X \oplus$ (expression for X)
    - **Note #1**: expression for X doesn't have X in it!
    - **Note #2**: this is the **complement** of the gate consistency function from SAT.
- Example

$$SDC_X = X \oplus (ab + c)$$



SDCs: Example



- $SDC_X = X \oplus (ab + c) = \bar{X}ab + \bar{X}c + X\bar{a}\bar{c} + X\bar{b}\bar{c}$

One **impossible pattern**: $Xabc = 011-$

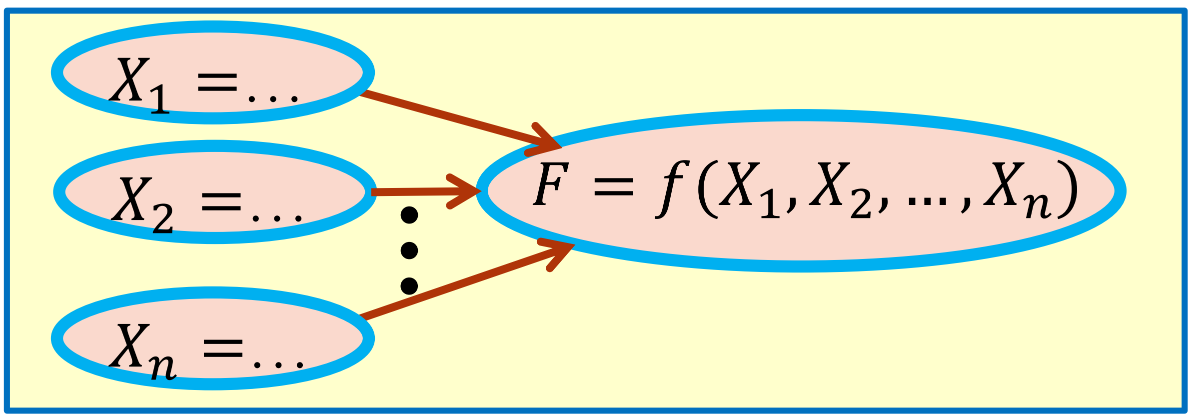
SDCs: Summary

- SDCs are associated with every **internal wire** in Boolean logic network.
  - SDCs explain **impossible patterns** of input to, and output of, each node.
  - SDCs are easy to compute.

- SDCs alone are **not** the Don't Cares used to simplify nodes.
  - We use SDCs to **build CDCs**, which give impossible patterns at input of nodes.

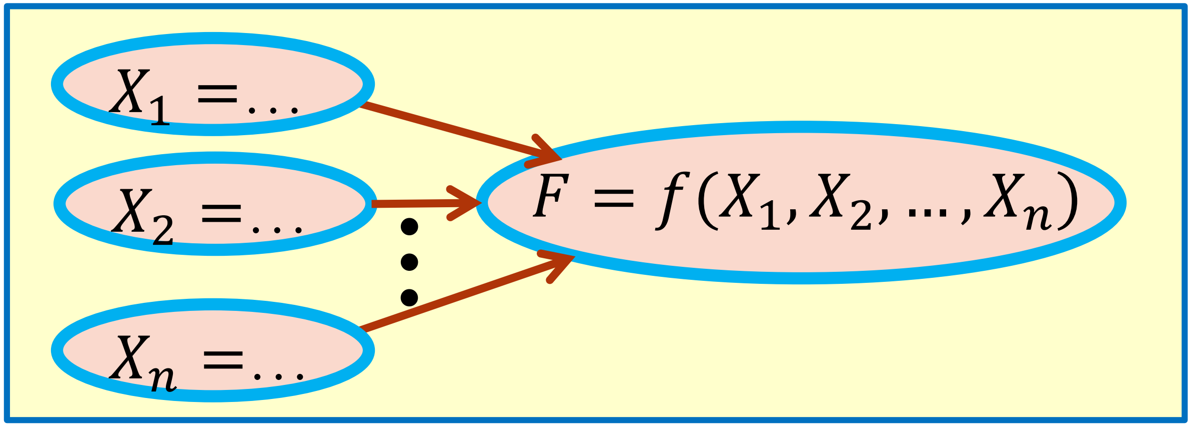
How to Compute CDCs?

- Computational recipe:
  1. Get all the **SDCs** on the wires **input to** this node in Boolean logic network.
  2. **OR** together all these SDCs.
  3. **Universally Quantify** away all variables that are **NOT** used inside this node.



$$CDC_F(X_1, \ldots, X_n) = (\forall \text{ vars not used in } F)\left[\sum_{\text{input } X_i \text{ to } F} SDC_{X_i}\right]$$

How to Compute CDCs?



$$CDC_F(X_1, \dots, X_n) = (\forall \text{ vars not used in } F)\left[\sum_{\text{input } X_i \text{ to } F} SDC_{X_i}\right]$$

- **<u>Result</u>**: Inputs that let $CDC_F = 1$ are **impossible patterns** at input to node!

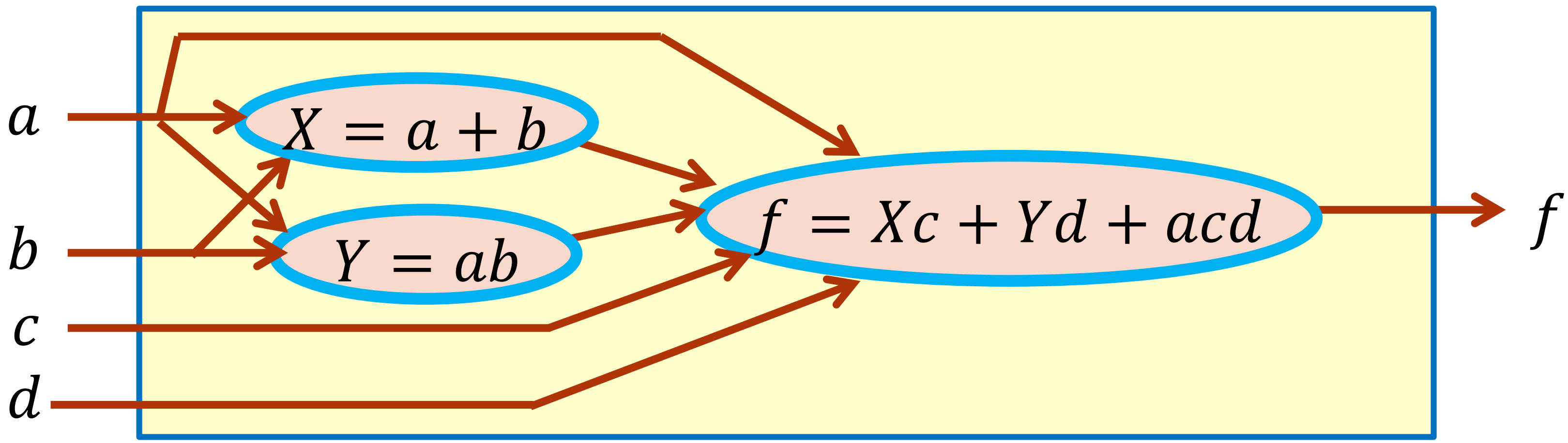
CDCs: Why Does This Work?

$$CDC_F(X_1, \ldots, X_n) = (\forall \text{ vars not used in } F)\left[\sum_{\text{input } X_i \text{ to } F} SDC_{X_i}\right]$$

- Roughly speaking…
  - SDC_{X_i}'s explain all the impossible patterns involving X_i wire input to the F node.
  - **OR** operation is just the "**union**" of all these impossible patterns involving X_i's.
  - **Universal Quantify** removes variables **not** used by F, and does so in the right way: we want patterns that are impossible **FOR ALL** values of these removed variables.

Compute CDCs: Example

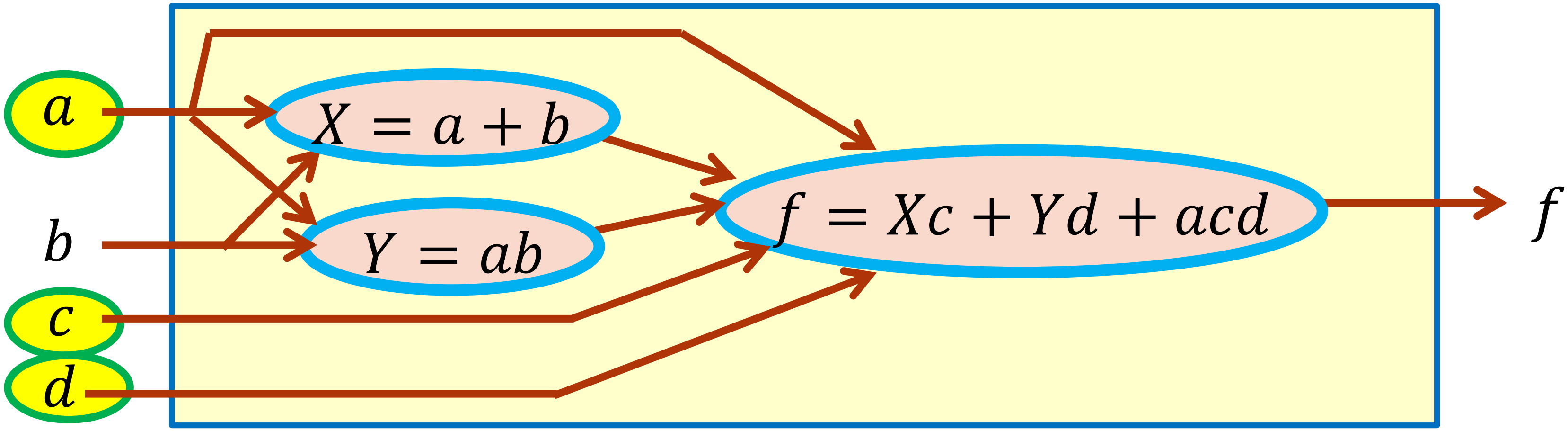
Obtain CDCs for the node f



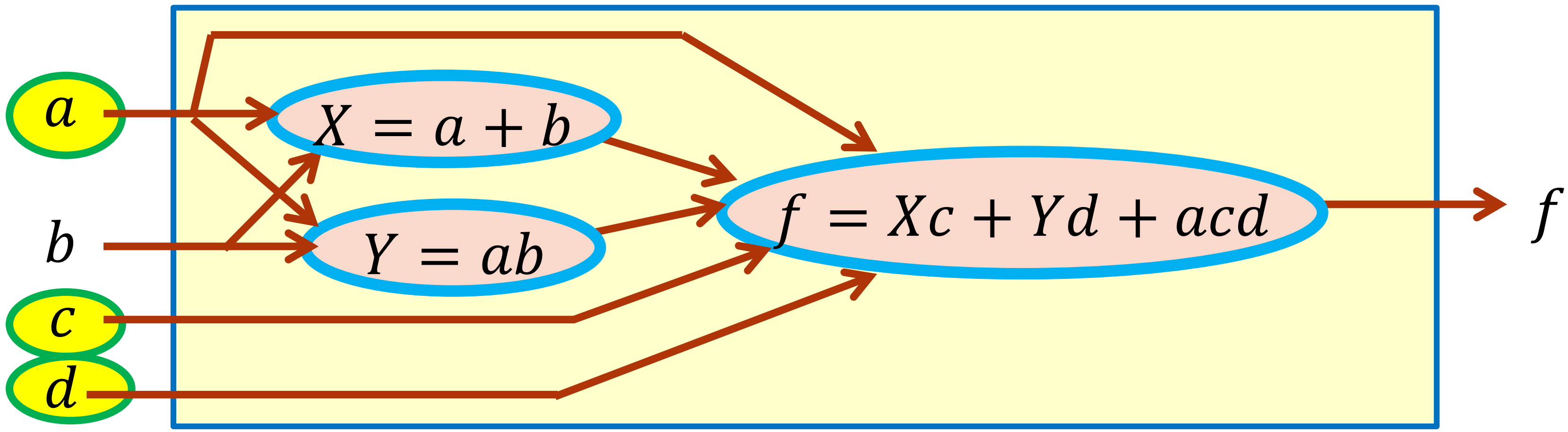
$$CDC_f(X_1, \dots, X_n) = \underbrace{(\forall \text{ vars not used in } f)}_{\text{This is } b} \left[\underbrace{\sum_{\text{input } X_i \text{ to } f}}_{\substack{\text{Input variables to } f \\ \text{are } a,\, c,\, d,\, X,\, Y}} SDC_{X_i} \right]$$

Compute CDCs: Example

- What about SDCs on **primary inputs**?
  - They are just 0.
  - Why? $SDC_a = a \oplus (\text{expression for } a) = a \oplus a = 0$.
- **Thus**: SDCs on primary inputs have no impact on OR. We can **ignore primary inputs**.



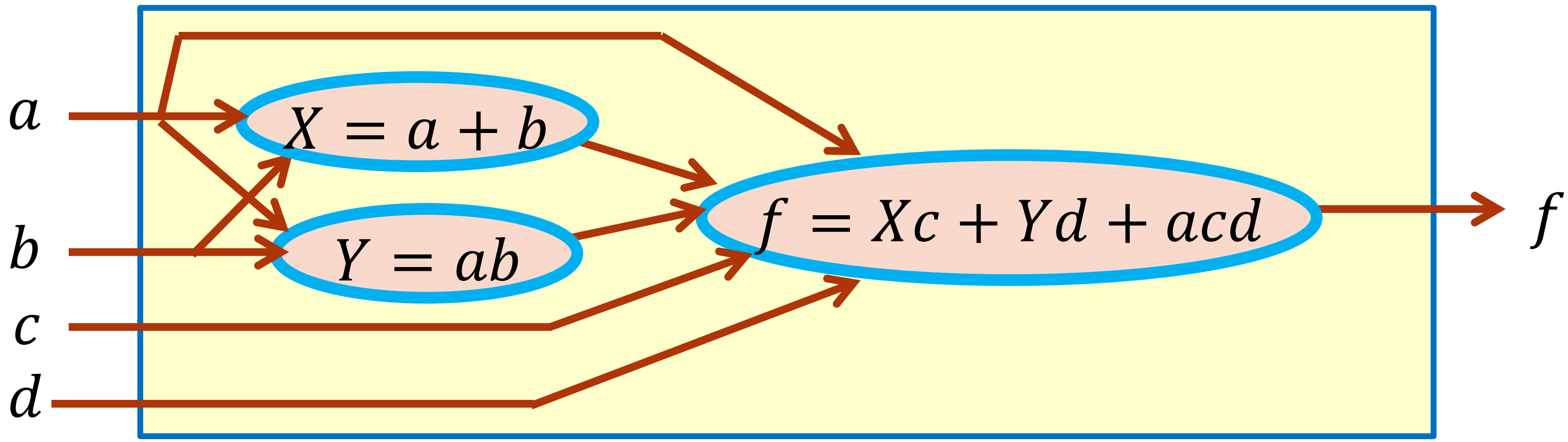
Compute CDCs: Example



- Since we ignore primary inputs, we have …

$$CDC_f(X_1, \dots, X_n) = \underbrace{(\forall \text{ vars not used in } f)}_{\text{This is } b} \left[\underbrace{\sum_{\text{input } X_i \text{ to } f}}_{\boxed{\text{Only } X, Y}} SDC_{X_i} \right]$$

Compute CDCs: Example



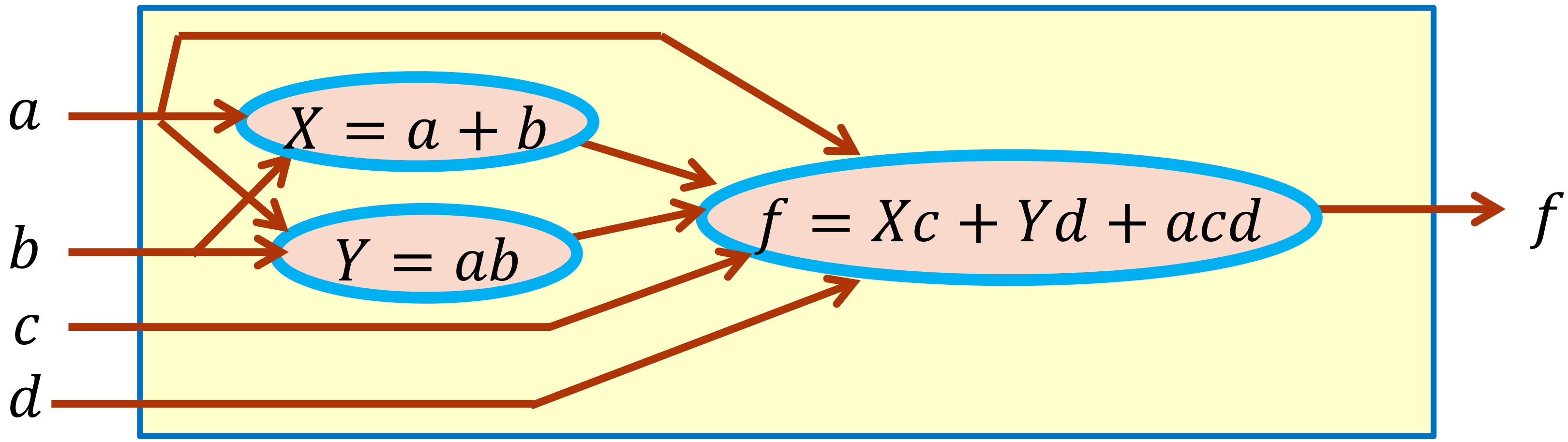
- Thus, we have:

$$CDC_f = (\forall b)[SDC_X + SDC_Y] = (\forall b)\big[[X \oplus (a+b)] + [Y \oplus ab]\big]$$

$$= \big[[X \oplus (a+b)] + [Y \oplus ab]\big]_{b=1} \cdot \big[[X \oplus (a+b)] + [Y \oplus ab]\big]_{b=0}$$

$$= [\bar{X} + (Y \oplus a)] \cdot [(X \oplus a) + Y] = \bar{X}a + Y\bar{a} + \bar{X}Y$$

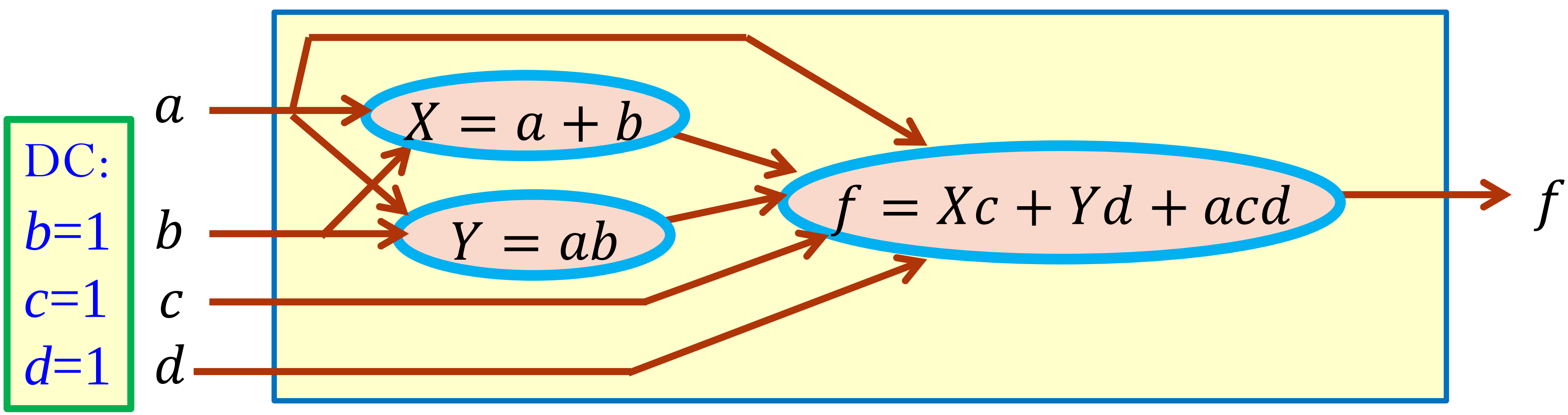
Compute CDCs: Example



- $CDC_f = \bar{X}a + Y\bar{a} + \bar{X}Y$
- Does it **make sense**?
  - From CDC_f, **impossible patterns** are
    - $(X, a) = (0,1)$ $\quad a = 1 \Rightarrow X = 1$
    - $(Y, a) = (1,0)$ $\quad a = 0 \Rightarrow Y = 0$
    - $(X, Y) = (0,1)$ $\quad X = 0 \Rightarrow a = 0 \,\&\&\, b = 0 \Rightarrow Y = 0$

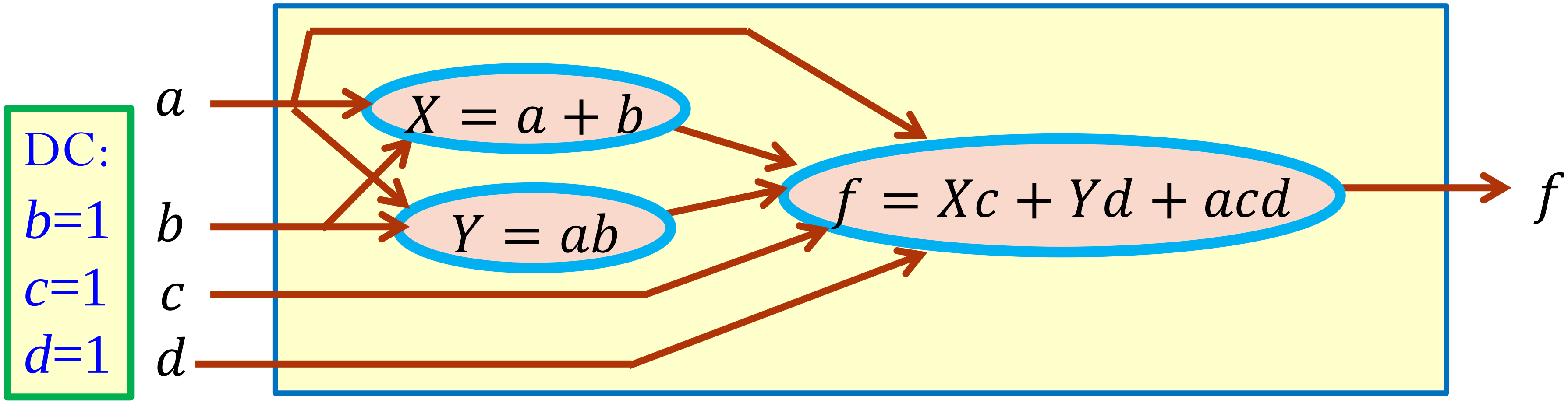
How to Handle External CDCs?

- What if there are **external DCs** for primary inputs a, b, c, d for which we just **don't care** what f does?
  - **Answer**: Just **OR** these DCs in $(\sum SDC_i)$ part of CDC expression.
  - Represent these DCs as a **Boolean function** that makes a 1 when the inputs are **these DCs**.



Handling External CDCs: Example

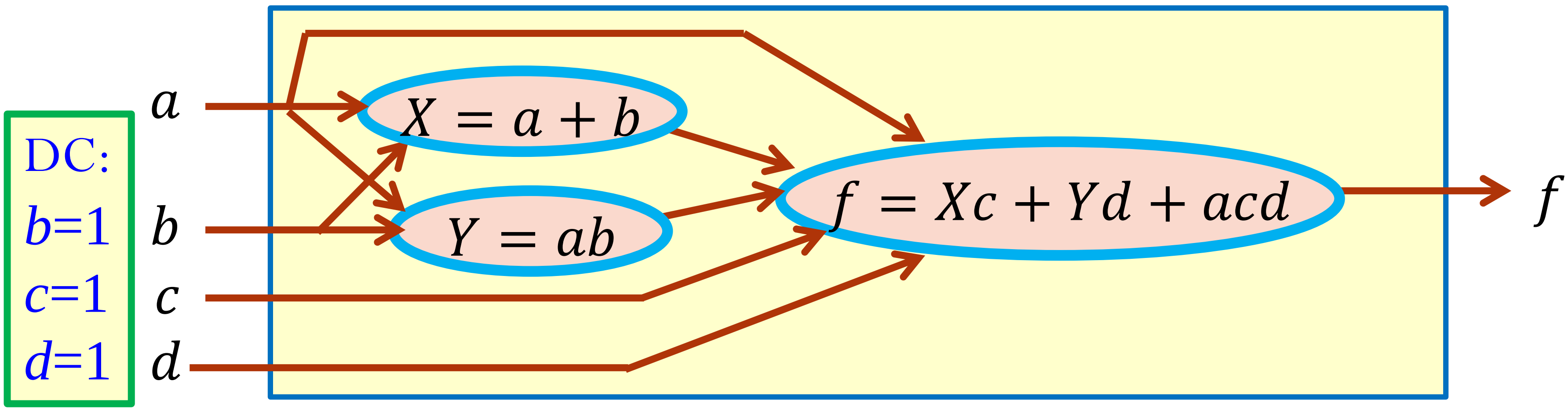
- Suppose $(b, c, d) = (1,1,1)$ cannot happen.
  - How to compute CDC_f now?



$$CDC_f = (\forall b)\Big[[X \oplus (a+b)] + [Y \oplus ab] \underbrace{+\ bcd}\Big]$$

External DCs as a **Boolean function** that makes a 1 when the pattern is **impossible**.

Handling External CDCs: Example



$$CDC_f = (\forall b)\big[[X \oplus (a + b)] + [Y \oplus ab] + bcd\big]$$

$$= \bar{X}a + Y\bar{a} + \bar{X}Y + \bar{a}cdX + cdY$$

- **New impossible patterns** are

Make sense?

  - $(a, c, d, X) = (0,1,1,1)$ $\quad a = 0 \,\&\&\, X = 1 \Rightarrow b = 1$

    Thus, $b = c = d = 1$

  - $(c, d, Y) = (1,1,1)$ $\quad Y = 1 \Rightarrow b = 1$

    Thus, $b = c = d = 1$

CDCs: Summary

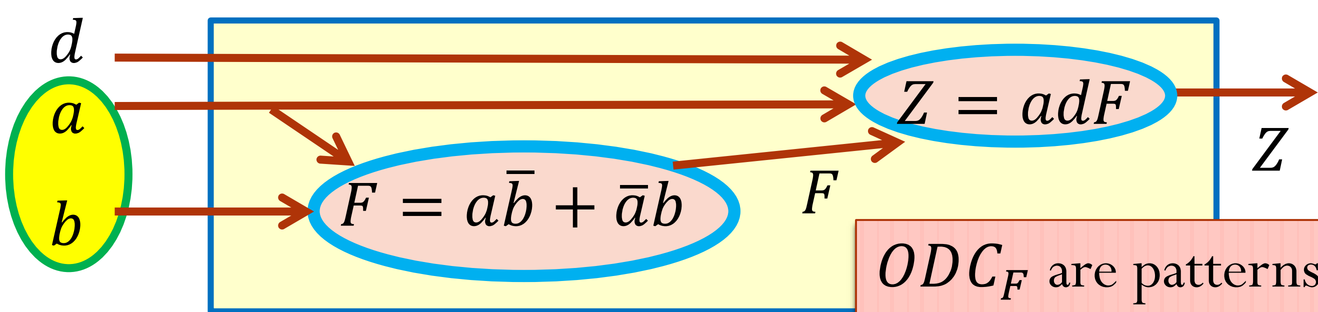
- CDCs give **impossible patterns** at input to node F – use as DCs.
  - Impossible because of the network structure of the nodes **feeding** node F.
  - CDCs can be computed mechanically from **SDCs** on wires input to F.
    - **Internal local CDCs**: computed just from SDCs on wires into F.
    - **External global CDCs**: include DC patterns at primary inputs.

CDCs: Summary (cont.)

- But CDCs still **not all** the Don't Cares available to simplify nodes.
  - CDC_F derived from the structure of nodes "**before**" node F.
  - We need to look at DCs that derive form nodes "**after**" node F.
  - These are nodes between the **output** of F and **primary outputs** of overall network.
  - These are ODCs.

Observability Don't Cares (ODCs)

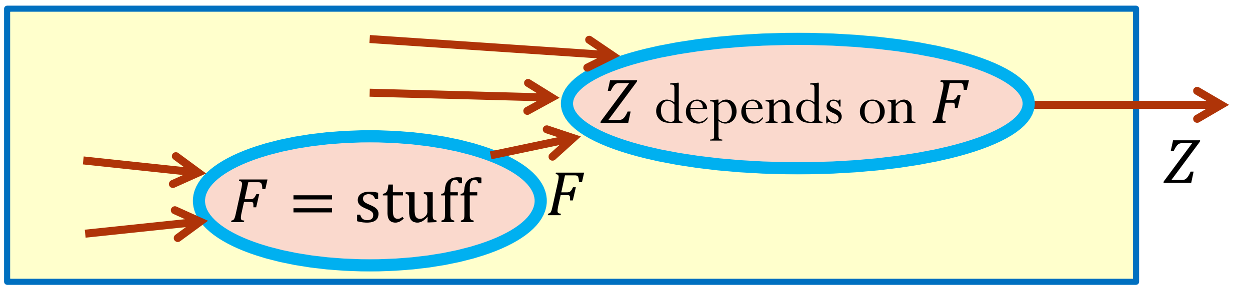
- **ODCs**: patterns that **mask** a node's output at primary output (PO) of the network.
  - So, these are **not** impossible patterns – these patterns **can occur** at node input.
  - These patterns make this node's output **not observable at primary output**.
  - "**Not observable**" for an input pattern means: Boolean value of node output **does not affect** **<u>ANY</u>** primary output.

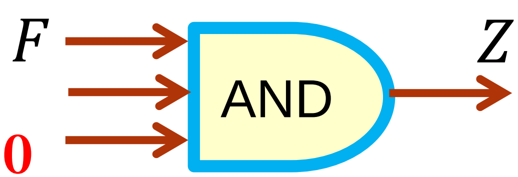


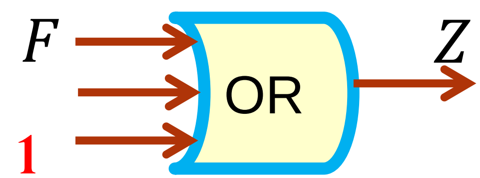
ODC_F are patterns of (a, b) that make Z **insensitive** to F's value.

Primary Output Insensitive to F

- When is primary output Z **insensitive** to internal variable F?
  - Means Z **independent** of value of F, given other inputs to Z.





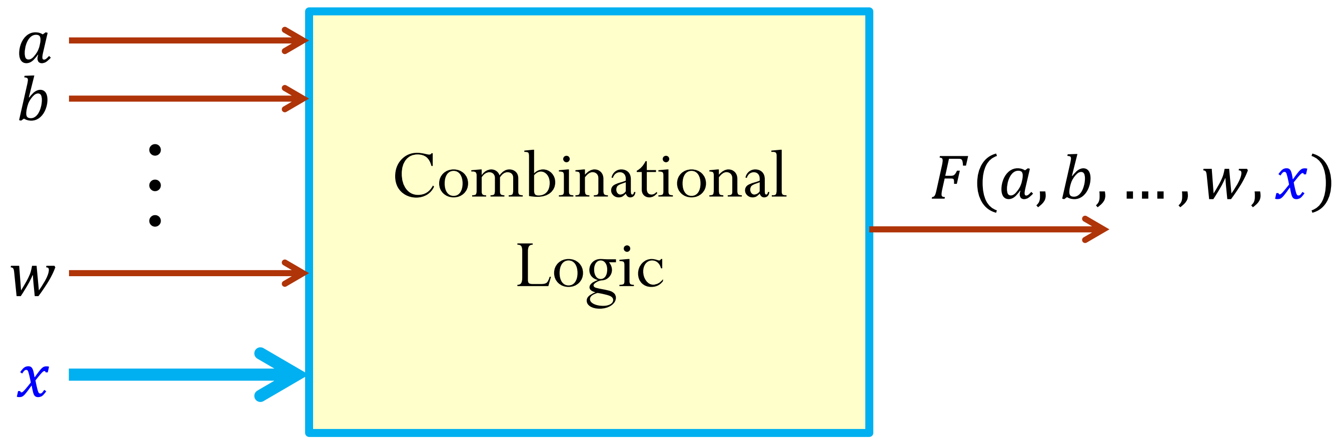


Z **insensitive** to F if
any other input $= 0$

Z **insensitive** to F if
any other input $= 1$

How about the general case?

Recall: Boolean Difference



- What does **Boolean difference** $\partial F(a, b, \ldots, w, x)/\partial x = F_x \oplus F_{\overline{x}} = 1$ mean?
  - If you apply an input pattern $(a, b, \ldots, w)$ that makes $\partial F/\partial x = 1$, then **any change** in x will **force a change** in output F.
- What makes output F **sensitive** to input x?
  - **Answer**: Any pattern that makes $\frac{\partial F}{\partial x} = F_x \oplus F_{\overline{x}} = 1$.

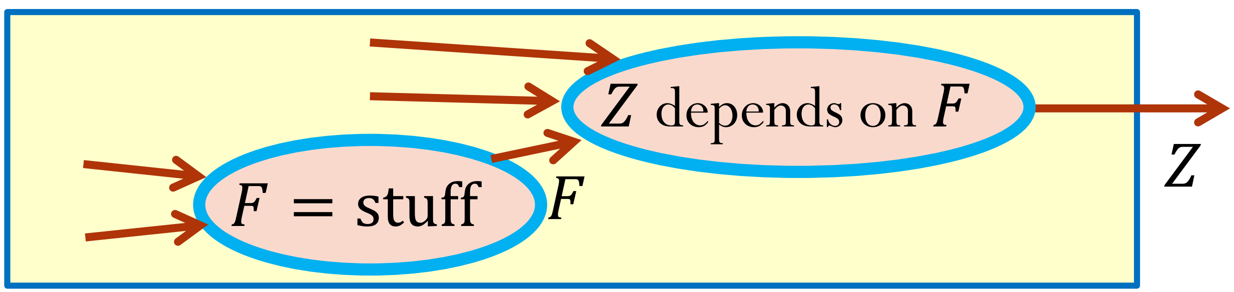
Z Insensitive to F

- When is primary output Z **insensitive** to internal variable F?
  - **Answer**: when inputs (other than F) to Z make cofactors $Z_F = Z_{\bar{F}}$.
  - **Make sense**: if cofactors with respect to F are **same**, Z does not depend on F!

- How to find when cofactors are the same?
  - **Answer**: Solve for $Z_F \overline{\oplus} Z_{\bar{F}} = 1$
  - **Note**: $Z_F \overline{\oplus} Z_{\bar{F}} = 1 \Rightarrow \overline{Z_F \oplus Z_{\bar{F}}} = 1 \Rightarrow \overline{\frac{\partial Z}{\partial F}} = 1$

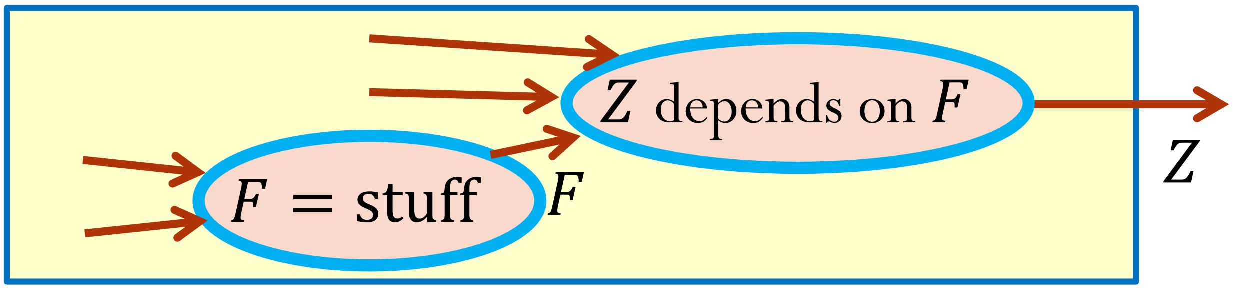
How to Compute ODCs?

- A nice computational recipe:
  1. **Compute** $\overline{\partial Z/\partial F}$. Any patterns that make $\overline{\partial Z/\partial F} = 1$ **mask** output F for Z.
  2. **Universally Quantify** away all variables that are **NOT** inputs to the F node.



$$ODC_F(X_1, \dots, X_n) = (\forall \text{ vars not used in } F)\left[\overline{\partial Z/\partial F}\right]$$

How to Compute ODCs?

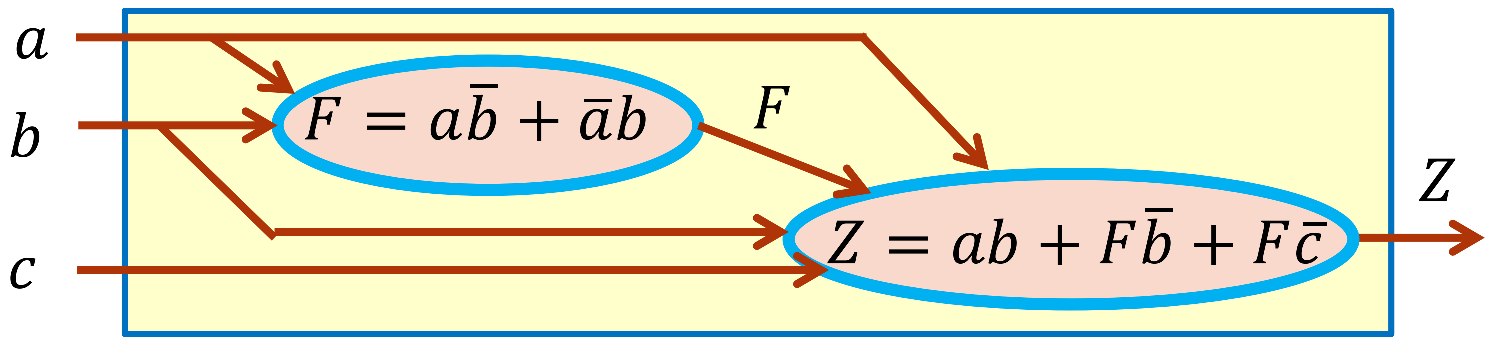


$$ODC_F(X_1, \ldots, X_n) = (\forall \text{ vars not used in } F)\left[\overline{\partial Z / \partial F}\right]$$

- **Result**: Inputs that let $ODC_F = 1$ **mask** output F for Z, i.e., make Z **insensitive** to F.

Compute ODCs: Example

- Obtain the ODCs for node F.

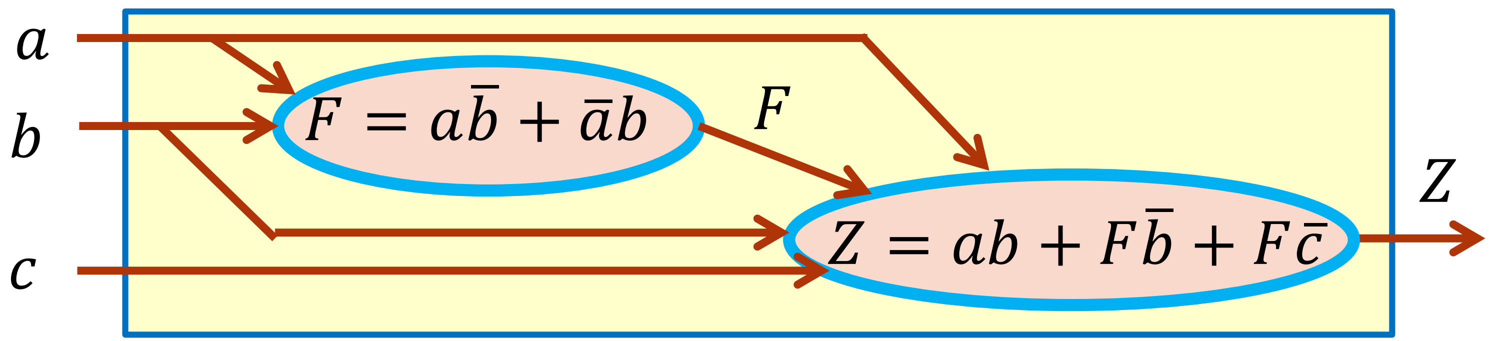


$$ODC_F(\underbrace{X_1, \ldots, X_n}_{\text{They are } a, b}) = \underbrace{(\forall \text{ vars not used in } F)}_{\text{This is } c}\left[\overline{\partial Z/\partial F}\right]$$

$$= (\forall c)\left[\overline{\left(ab + F\bar{b} + F\bar{c}\right)_{F=1} \oplus \left(ab + F\bar{b} + F\bar{c}\right)_{F=0}}\right]$$

$$= (\forall c)\left[\overline{(ab + \bar{c}) \oplus (ab)}\right] = ab$$

Check: Does this ODC Make Sense?

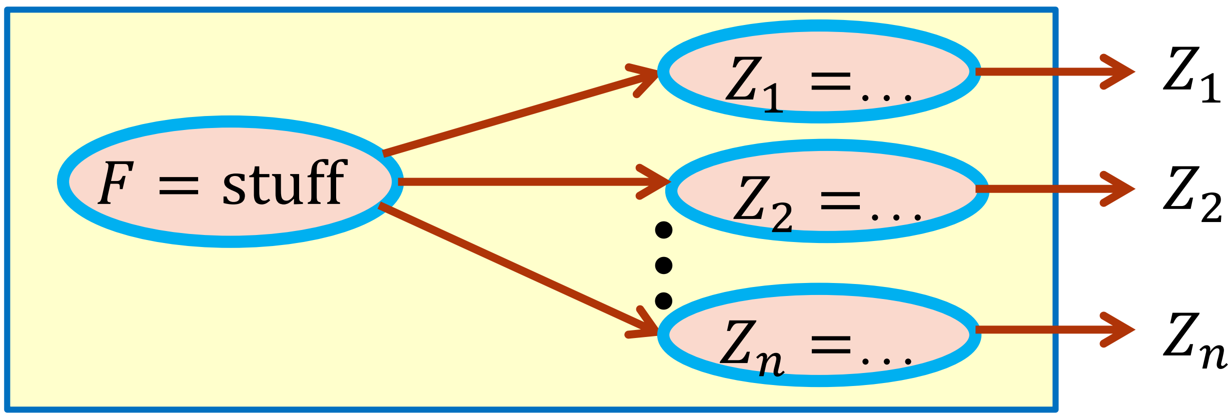


- $ODC_F = ab$
  - ODC pattern is $(a, b) = (1,1)$

- Make sense! Because when $(a, b) = (1,1), Z = 1$ independent of F.

ODCs: More General Case

- **<u>Question</u>**: what if F feeds to **many** primary outputs?
  - **<u>Answer</u>**: Only patterns that are **unobservable** at **ALL** outputs can be ODCs.



- Computational recipe:

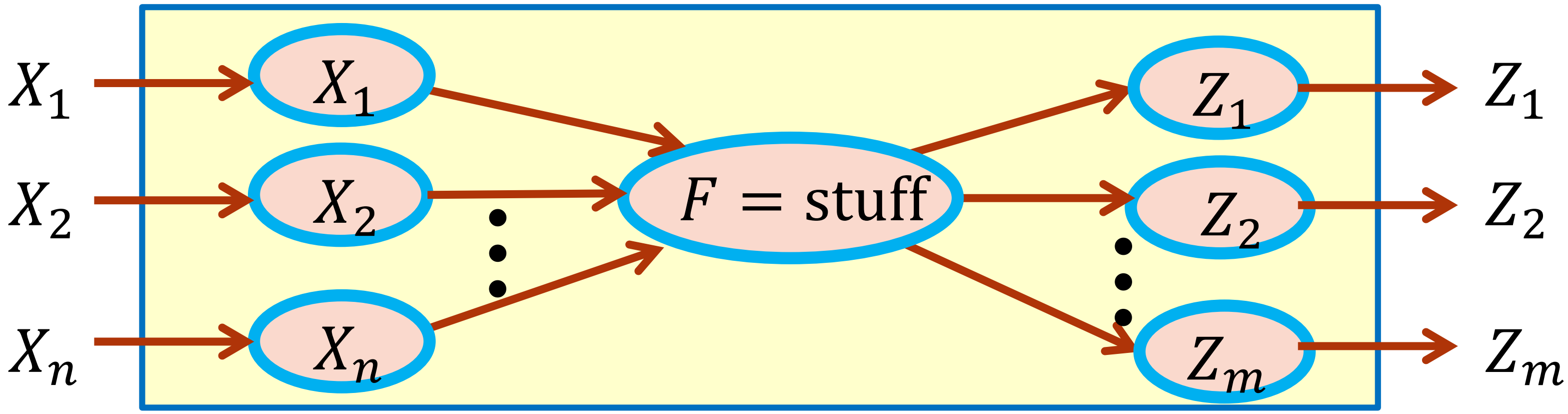
$$ODC_F = (\forall \text{ vars not used in } F)\left[\underbrace{\prod_{\text{Output } Z_i} \overline{\partial Z_i / \partial F}}_{\textbf{AND} \text{ all } n \text{ differences for each output } Z_i.}\right]$$

ODCs: Summary

- ODCs give input patterns of node F that **mask** F at **primary outputs**.
  - **Not** impossible patterns – they **can occur**.
  - Don't cares because primary output "**doesn't care**" what F is, for these patterns.
  - ODCs are can be computed mechanically from $\overline{\partial Z_i / \partial F}$ on all outputs connected to F.

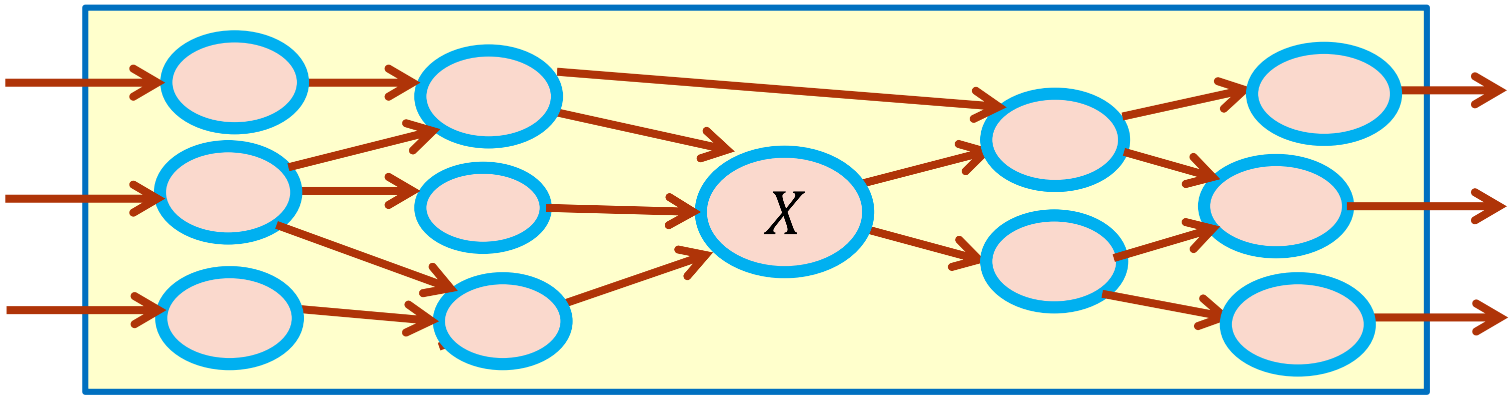
- CDCs + ODCs give the "**full**" don't care set used to simplify F.
  - With these patterns, you can call something like ESPRESSO to simplify F.

Multi-Level Don't Cares: Are We Done?



- Yes, if your networks look **just like above**.
  - More precisely, if you only want to get CDCs from nodes **immediately** "before" you.
  - And if you only want to get ODCs for **one layer of nodes** between you and output.

Don't Cares, In General



- But, this is what **real** multi-level logic can look like!
  - CDCs are function of **all nodes** "before" *X*.
  - ODCs are function of **all nodes** between *X* and any output.
  - In general, we can **never get all** the DCs for node *X* in a big network.
  - Representing all this stuff can be **explosively** large, even with BDDs

Summary: Getting Network DCs

- How we really do it? generally **do not get all** the DCs.
  - Lots of tricks that trade off effort (time, memory) with quality (how many DCs).
  - Example: Can just extract "**local CDCs**", which requires looking at outputs of **immediate precedent** vertices and computing from the SDC patterns, which is easy.
  - There are also **incremental**, node-by-node algorithms that walk the network to compute more of the CDC and ODC set for X, but these are more **complex**.

- For us, knowing these "limited" DC recipes is **sufficient**.