

VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 4.1

Computational Boolean Algebra Representations: Satisfiability (SAT), Part 1



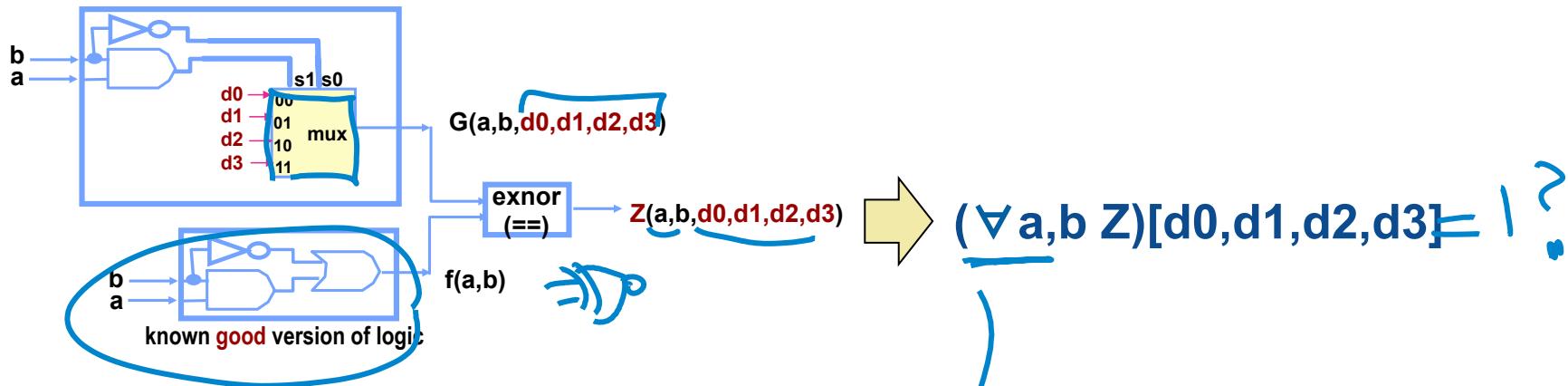
Chris Knott/Digital Vision/Getty Images

Some Terminology

- **Satisfiability (called “SAT” for short)**
 - Give me an appropriate representation of function $F(x_1, x_2, \dots x_n)$
 - Find an assignment of the variables (“vars” for short) $(x_1, x_2, \dots x_n)$ so $F(\) = 1$
 - Note – this assignment need not be unique. Could be many satisfying solutions
 - But if there are no satisfying assignments at all – prove it, and return this info
- **Some things you can do with BDDs, can do easier with SAT**
 - SAT is aimed at scenarios where you just need one satisfying assignment...
 - ...or prove that there is no such satisfying assignment



Example: Network Repair



- Assuming you can build the quantification function $(\forall a,b Z)$
 - Go find a SAT assignment to get you the d values to repair the network
 - Or, if unSAT, know that there is no network repair possible



Standard SAT Form: CNF

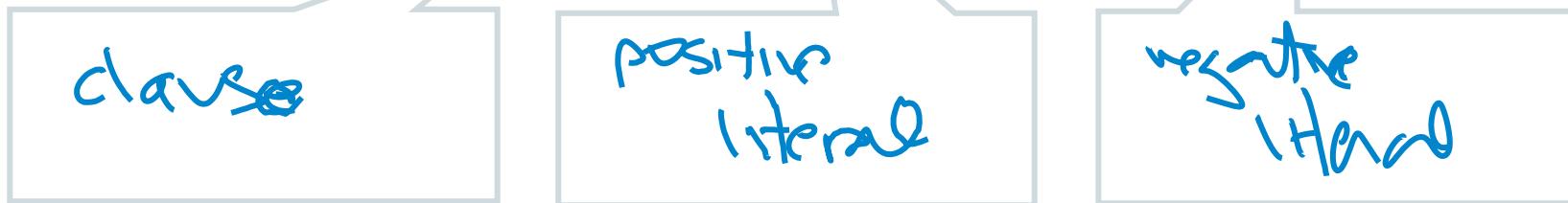
- Conjunctive Normal Form (CNF) = Standard POS form

(SOP)

$$\phi = (a + c)(b + c)(\neg a + \neg b + \neg c)$$

$$\neg = \text{not}$$

$$= \bar{c} = c'$$



- Why CNF is useful

- Need only determine that **one** clause evaluates to **“0”** to know whole formula = **“0”**
- Of course, to satisfy the whole formula, you must make **all** clauses identically **“1”**.



Assignment to a CNF Formula

- An **assignment**...
 - ...gives values to some, not necessarily all, of variables (vars) x_i in (x_1, x_2, \dots, x_n) .
 - **Complete** assignment: assigns value to all vars. **Partial**: some, not all, have values
- Assignment means we can evaluate **status** of the clauses
 - Suppose $a=0, b=1$ but c, d are unassigned

$$\Phi = (a^{\circ} + \neg b^{\circ}) (\neg a^{\circ} + b^{\circ} + \neg c^{\circ}) (a^{\circ} + c^{\circ} + d^{\circ}) (\neg a^{\circ} + \neg b^{\circ} + \neg c^{\circ})$$

clause = 0
conflicting

clause = 1
satisfied

unresolved

SAT



How Do We “Solve” This?

- **(Recursively) (...surprised?)**

- Strategy has two big ideas

- **DECISION:**

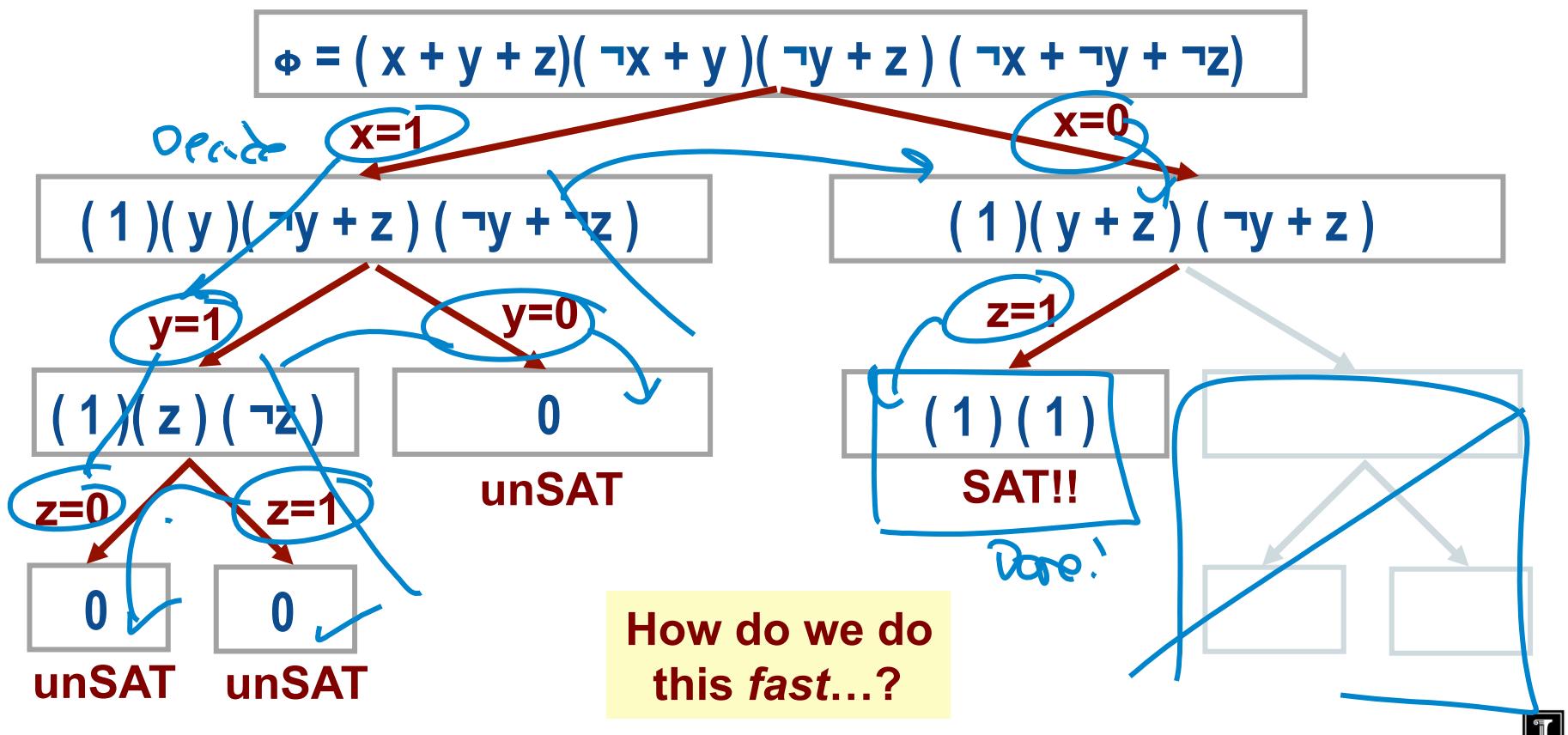
- **Select** a variable and **assign** its value; **simplify** CNF formula as far as you can
 - Hope you can decide if it's SAT, yes/no, without further work

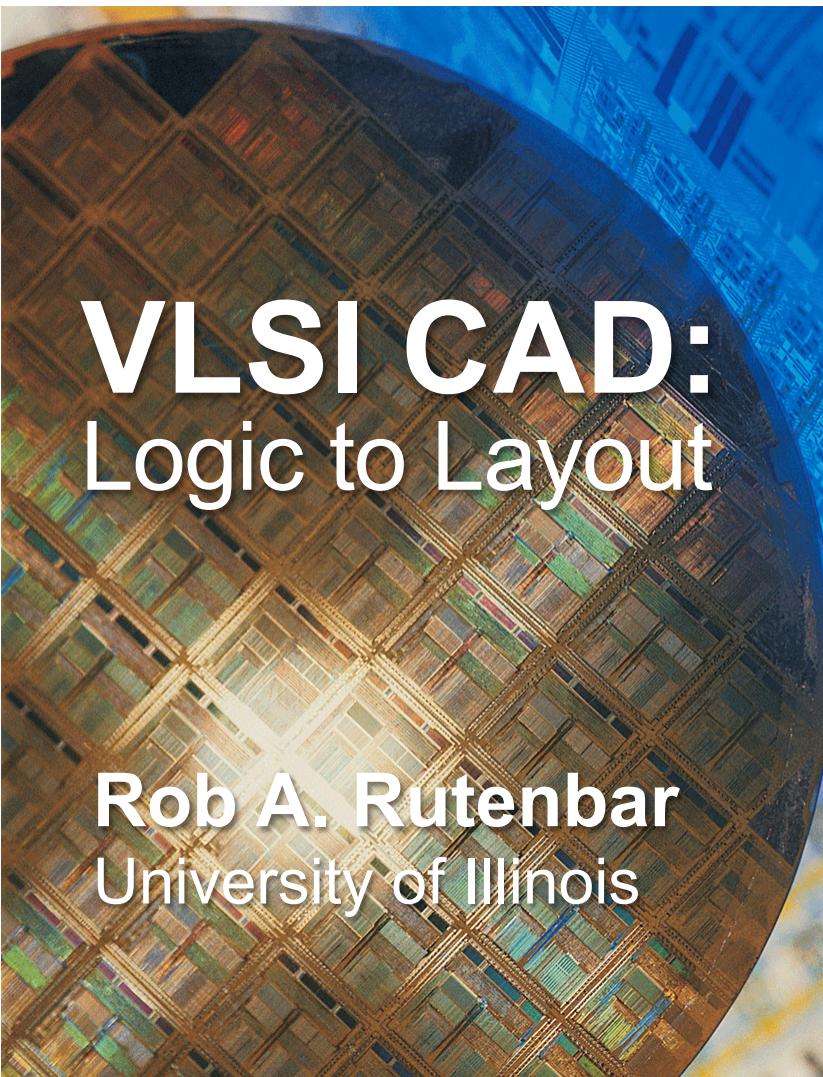
- **DEDUCTION:**

- Look at the newly simplified clauses
 - **Iteratively simplify**, based on structure of clauses, and value of partial assignment
 - Do this until nothing simplifies. If you can decide SAT yes/no, great.
 - If not, then you have to **recurse** some more, back up to DECIDE



How Do We “Solve” This?





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 4.2

Computational Boolean Algebra Representations: Boolean Constraint Propagation (BCP) for SAT



Chris Knott/Digital Vision/Getty Images

BCP: Boolean Constraint Propagation

- To do “deduction”, use **BCP**
 - Given a set of **fixed** variable assignments, what else can you “deduce” about necessary assignments by “**propagating constraints**”
- Most famous BCP strategy is “**Unit Clause Rule**”
 - A clause is said to be “**unit**” if it has exactly one unassigned literal
 - Unit clause has exactly **one** way to be satisfied, ie, pick polarity that makes clause=“1”
 - This choice is called an “**implication**”

$$\phi = (\overline{a} + c) (\overline{b} + c) (\overline{\overline{a}} + \overline{\overline{b}} + \overline{\overline{c}})$$

Assume:
 $a=1, b=1$

Unit: 1 unassigned literal = \overline{c}
 c must be 0 \rightarrow SAT!



BCP Iterative – Go Until No More Implications

$$\Phi = (\omega_1)(\omega_2)(\omega_3)(\omega_4)(\omega_5)(\omega_6)(\omega_7)(\omega_8)(\omega_9)$$

$$\omega_1 = (\neg x_1 + x_2) \quad 0$$

$$\omega_2 = (\neg x_1 + x_3 + \cancel{x_5}) \quad 0$$

$$\omega_3 = (\neg x_2 + \neg x_3 + x_4) \quad 0$$

$$\omega_4 = (\neg x_4 + x_5 + x_{10}) \quad 0$$

$$\omega_5 = (\neg x_4 + x_6 + x_{11}) \quad 0$$

$$\omega_6 = (\neg x_5 + \neg x_6)$$

$$\omega_7 = (x_1 + x_7 + \neg x_{12}) \quad 0$$

$$\omega_8 = (x_1 + x_8)$$

$$\omega_9 = (\neg x_7 + \neg x_8 + \neg x_{13}) \quad 0$$

No SAT
No BCP
Now what?

Slide 10

- **Example from**

J.P. Marques-Silva and K. Sakallah,
“GRASP: A Search Algorithm for
Propositional Satisfiability”, *IEEE
Trans. Computers*, Vol 8, No 5, May’99

- **Partial assignment is:**

$x_9=0 \quad x_{10}=0 \quad x_{11}=0 \quad x_{12}=1 \quad x_{13}=1$

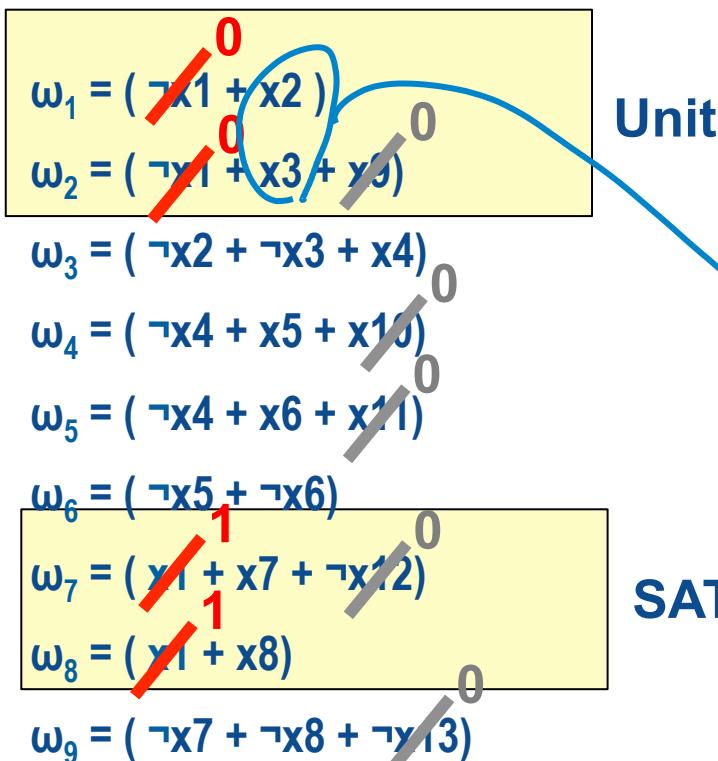
- **To start...**

- What are **obvious** simplifications when we assign these variables?



BCP Iterative – Go Until No More Implications

$$\phi = (\omega_1)(\omega_2)(\omega_3)(\omega_4)(\omega_5)(\omega_6)(\omega_7)(\omega_8)(\omega_9)$$



- **Partial assignment is:**
 - $x_9=0 \quad x_{10}=0 \quad x_{11}=0 \quad x_{12}=1 \quad x_{13}=1$
- **Next: Assign a variable to value**
 - { Assign $x_1=1$ }

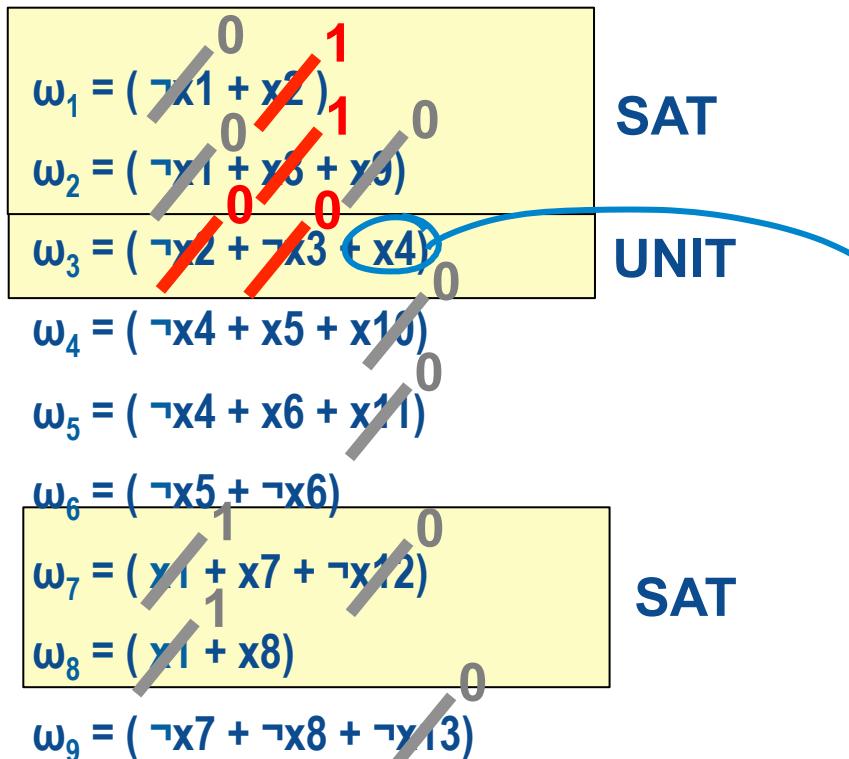
Implications!

$x_1=1 \rightarrow x_2=1 \quad \&& \quad x_3=1$



BCP Iterative – Go Until No More Implications

$$\Phi = (\omega_1)(\omega_2)(\omega_3)(\omega_4)(\omega_5)(\omega_6)(\omega_7)(\omega_8)(\omega_9)$$



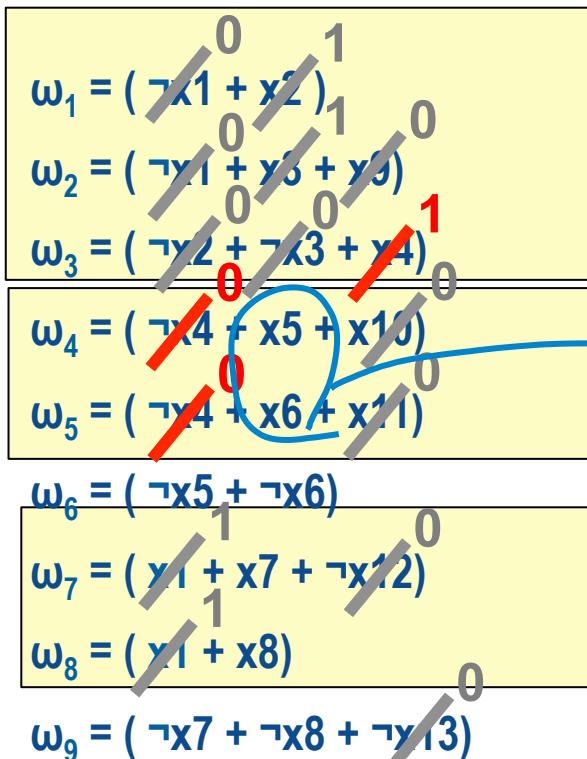
- **Partial assignment is:**
 - $x_9=0 \quad x_{10}=0 \quad x_{11}=0 \quad x_{12}=1 \quad x_{13}=1$
- **Next: Assign a var to value**
 - Assign $x_1=1$
 - Assign (implied). $x_2=1, x_3=1$

Implications!
 $x_2=1, x_3=1 \rightarrow x_4=1$



BCP Iterative – Go Until No More Implications

$$\Phi = (\omega_1)(\omega_2)(\omega_3)(\omega_4)(\omega_5)(\omega_6)(\omega_7)(\omega_8)(\omega_9)$$



- **Partial assignment is:**
 - $x9=0 \quad x10=0 \quad x11=0 \quad x12=1 \quad x13=1$
- **Next: Assign a var to value**
 - Assign $x1=1$
 - Assign (implied): $x2=1, x3=1$
 - Assign (implied): $x4=1$

Implications!
 $x4=1 \rightarrow x5=1 \text{ && } x6=1$



BCP Iterative – Go Until No More Implications

$$\Phi = (\omega_1)(\omega_2)(\omega_3)(\omega_4)(\omega_5)(\omega_6)(\omega_7)(\omega_8)(\omega_9)$$

| | | | |
|--|---|---|---|
| $\omega_1 = (\neg x_1 + x_2)$ | 0 | 1 | |
| $\omega_2 = (\neg x_1 + x_5 + x_6)$ | 0 | 1 | 0 |
| $\omega_3 = (\neg x_2 + \neg x_3 + x_4)$ | 0 | 0 | 1 |
| $\omega_4 = (\neg x_4 + x_5 + x_{10})$ | 0 | 1 | 0 |
| $\omega_5 = (\neg x_4 + x_6 + x_{11})$ | 0 | 1 | 0 |
| $\omega_6 = (\neg x_5 + \neg x_6)$ | 0 | 0 | |
| $\omega_7 = (x_1 + x_7 + \neg x_{12})$ | 1 | 0 | |
| $\omega_8 = (x_1 + x_8)$ | 1 | 0 | |
| $\omega_9 = (\neg x_7 + \neg x_8 + \neg x_{13})$ | 0 | | |

SAT

UNIT

CONFLICT!

SAT

- **Partial assignment is:**

- $x_9=0 \quad x_{10}=0 \quad x_{11}=0 \quad x_{12}=1 \quad x_{13}=1$

- **Next: Assign a var to value**

- Assign $x_1=1$

- Assign (implied): $x_2=1, x_3=1$

- Assign (implied): $x_4=1$

- Assign (implied): $x_5=1 \text{ & } x_6=1$

Conflict \rightarrow unSAT

$x_5=1 \text{ & } x_6=1 \rightarrow \text{clause } \omega_6==0!$



BCP Iterative – Go Until No More Implications

$$\phi = (\omega_1)(\omega_2)(\omega_3)(\omega_4)(\omega_5)(\omega_6)(\omega_7)(\omega_8)(\omega_9)$$

| | | | |
|--|---|---|------------|
| $\omega_1 = (\neg x_1 + x_2)$ | 0 | 1 | SAT |
| $\omega_2 = (\neg x_1 + x_5 + x_9)$ | 0 | 1 | SAT |
| $\omega_3 = (\neg x_2 + \neg x_3 + x_4)$ | 0 | 0 | SAT |
| $\omega_4 = (\neg x_4 + x_5 + x_{10})$ | 0 | 1 | SAT |
| $\omega_5 = (\neg x_4 + x_6 + x_{11})$ | 0 | 1 | SAT |
| $\omega_6 = (\neg x_5 + \neg x_6)$ | 1 | 0 | CONFLICT! |
| $\omega_7 = (x_1 + x_7 + \neg x_{12})$ | 1 | 0 | SAT |
| $\omega_8 = (x_1 + x_8)$ | 1 | 0 | SAT |
| $\omega_9 = (\neg x_7 + \neg x_8 + \neg x_{13})$ | 0 | 0 | UNRESOLVED |

- **3 cases when BCP finishes**
 - **SAT:** Find a SAT assignment, all clauses resolve to “1”. Return it.
 - **UNRESOLVED:** One or more clauses unresolved. Pick another unassigned var, and recurse more.
 - **UNSAT:** Like this. Found **conflict**, one or more clauses eval to “0”
- **Now what?**
 - You need to **undo** one of our variable assignments, try again...



This Has a Famous Name: DPLL

- **Davis-Putnam-Logemann-Loveland Algorithm**
 - Davis, Putnam published the basic recursive framework in 1960 (!)
 - Davis, Logemann, Loveland: found smarter BCP, eg, **unit-clause rule**, in 1962
 - Often called “Davis-Putnam” or “DP” in honor of the first paper in 1960, or (inaccurately) DPLL (all four of them never did publish this stuff together)
- **Big ideas**
 - A complete, systematic search of variable assignments
 - Useful **CNF** form for efficiency
 - BCP makes search stop earlier, “resolving” more assignments w/o recursing more



DPLL: Famous Stuff...

DPLL algorithm – Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/DPLL_algorithm

Create account Log in

Article Talk Read Edit View history Search

DPLL algorithm

From Wikipedia, the free encyclopedia

The **Davis–Putnam–Logemann–Loveland (DPLL) algorithm** is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. for solving the CNF-SAT problem.

It was introduced in 1962 by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland and is a refinement of the earlier Davis–Putnam algorithm, which is a resolution-based procedure developed by Davis and Putnam in 1960. Especially in older publications, the Davis–Logemann–Loveland algorithm is often referred to as the "Davis–Putnam method" or the "DP algorithm". Other common names that maintain the distinction are DLL and DPLL.

DPLL is a highly efficient procedure and after almost 50 years still forms the basis for most efficient complete SAT solvers, as well as for many theorem provers for fragments of first-order logic.^[1]

Contents [hide]

- 1 Implementations and applications
- 2 The algorithm
- 3 Current work
- 4 Relation to other notions
- 5 See also
- 6 References
- 7 Further reading

Implementations and applications

[edit]

DPLL

Class Boolean satisfiability problem



SAT: Huge Progress Last ~20 Years

- But: DPLL is only the start...
- SAT has been subject of intense work and great progress
 - Efficient data structures for clauses (so can search them fast) ✓
 - Efficient variable selection heuristics (so search smart, find lots of implications) ✓
 - Efficient BCP mechanisms (because SAT spends MOST of its time here)
 - Learning mechanisms (find patterns of vars that NEVER lead to SAT, avoid them) ✓
- Results: Good SAT codes that can do huge problems, fast
 - Huge means? 50,000 vars; 25,000,000 literals; 50,000,000 clauses (!!)

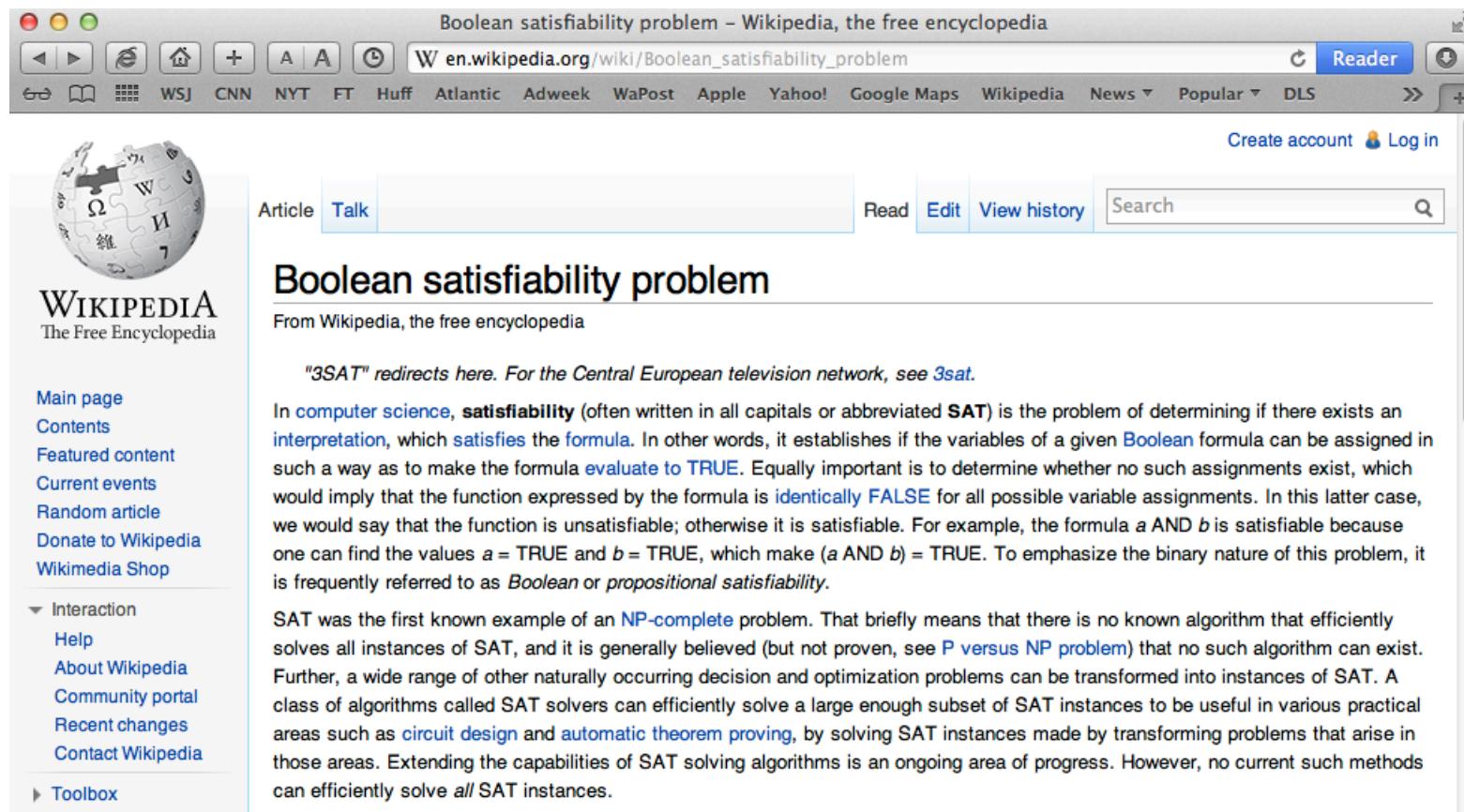


SAT Solvers

- **Many good solvers available online, open source**
- **Examples**
 - **MiniSAT**, from Niklas Eén, Niklas Sörensson in Sweden.
 - **We are using this one for our MOOC ✓**
 - CHAFF, from Sharad Malik and students, Princeton University
 - GRASP, from Joao Marques-Silva and Karem Sakallah, University of Michigan
 - ...and many others too. Go google around for them...



Lots of Information on SAT...



The screenshot shows a web browser window displaying the Wikipedia article on Boolean satisfiability problem. The title bar reads "Boolean satisfiability problem – Wikipedia, the free encyclopedia". The address bar shows the URL "en.wikipedia.org/wiki/Boolean_satisfiability_problem". The page content starts with the heading "Boolean satisfiability problem" and a brief summary from Wikipedia. Below the summary, there is a detailed explanation of the NP-complete problem, mentioning SAT solvers and their applications in circuit design and automatic theorem proving. On the left side, there is a sidebar with links to the main page, contents, featured content, current events, random article, donate, and various interaction and toolbox options.

Boolean satisfiability problem – Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/Boolean_satisfiability_problem

WSJ CNN NYT FT Huff Atlantic Adweek WaPost Apple Yahoo! Google Maps Wikipedia News Popular DLS

Create account Log in

Article Talk Read Edit View history Search

Boolean satisfiability problem

From Wikipedia, the free encyclopedia

"3SAT" redirects here. For the Central European television network, see [3sat](#).

In computer science, **satisfiability** (often written in all capitals or abbreviated **SAT**) is the problem of determining if there exists an interpretation, which satisfies the formula. In other words, it establishes if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable. For example, the formula $a \text{ AND } b$ is satisfiable because one can find the values $a = \text{TRUE}$ and $b = \text{TRUE}$, which make $(a \text{ AND } b) = \text{TRUE}$. To emphasize the binary nature of this problem, it is frequently referred to as *Boolean* or *propositional satisfiability*.

SAT was the first known example of an **NP-complete** problem. That briefly means that there is no known algorithm that efficiently solves all instances of SAT, and it is generally believed (but not proven, see [P versus NP problem](#)) that no such algorithm can exist. Further, a wide range of other naturally occurring decision and optimization problems can be transformed into instances of SAT. A class of algorithms called SAT solvers can efficiently solve a large enough subset of SAT instances to be useful in various practical areas such as [circuit design](#) and [automatic theorem proving](#), by solving SAT instances made by transforming problems that arise in those areas. Extending the capabilities of SAT solving algorithms is an ongoing area of progress. However, no current such methods can efficiently solve *all* SAT instances.

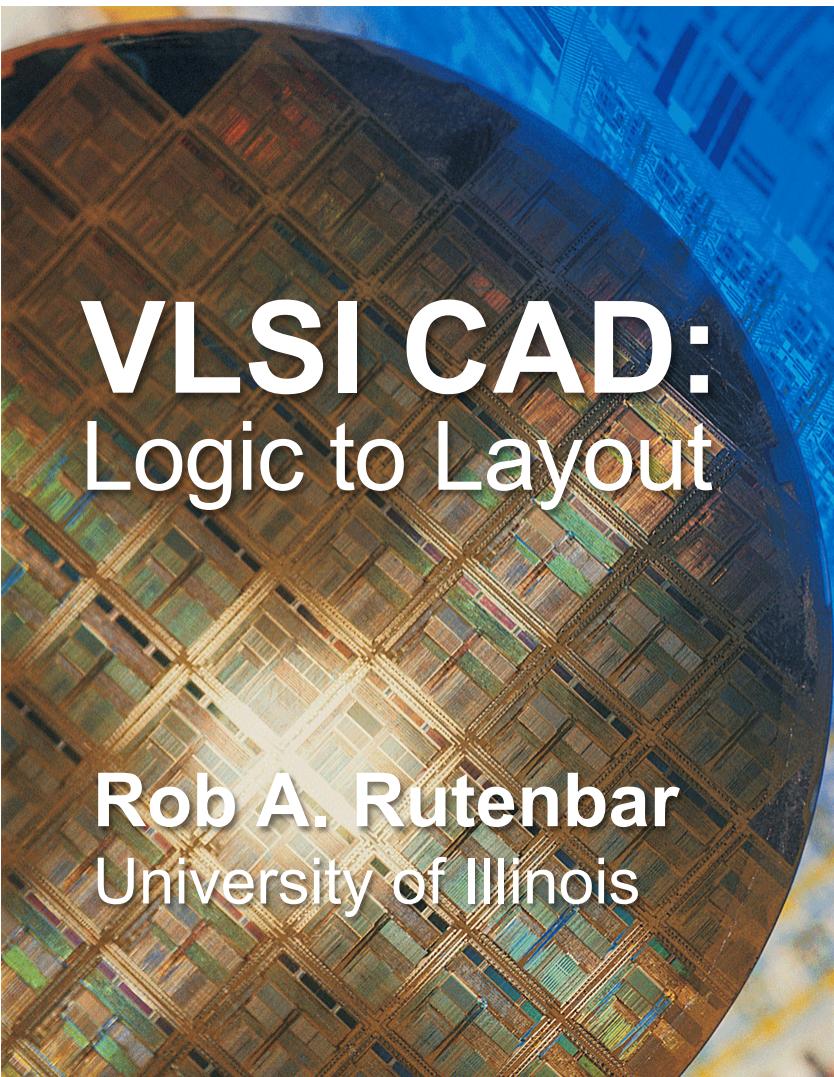
Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

© 2013, R.A. Rutenbar





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 4.3

Computational Boolean Algebra Representations: Using SAT for Logic



Chris Knott/Digital Vision/Getty Images

BDDs vs SAT Functionality

- **BDDs**



- Often work well for many problems
- But — no guarantee it will always work
- Can build BDD to *represent* function ϕ
- But—sometimes **cannot build BDD** with reasonable computer resources (run out of memory **SPACE**)

Yes -- builds a full representation of ϕ

- Can do a big set of Boolean manipulations on data structure
- Can build $(\exists xyz F)$ and $(\forall xyz F)$

- **SAT**

- Often works well for many problems
- But — no guarantee it will always work
- Can solve for SAT (y/n) on function ϕ
- But—sometimes **cannot find SAT** with reasonable computer resources (run out of **TIME** doing search)

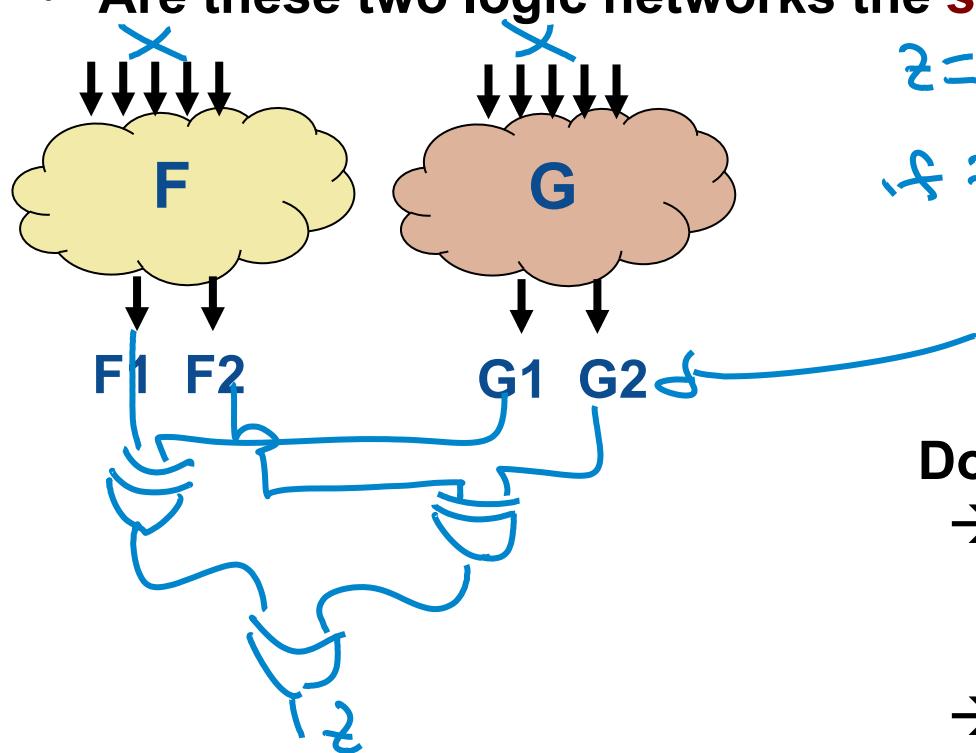
No – does **not represent** all of ϕ

- Can *solve* for SAT, but does not support big set of operators
- There are versions of **Quantified SAT** that solve SAT on $(\exists xyz F)$, $(\forall xyz F)$



Typical Practical SAT Problem

- Are these two logic networks the **same** Boolean function(s)? $f=6$?



$z = 1 \text{ SAT} \iff F \neq G$

$\neg z = 0 \text{ SAT: find } x \text{ such that}$

$$F(x) \neq G(x)$$

Do SAT solve on this new network

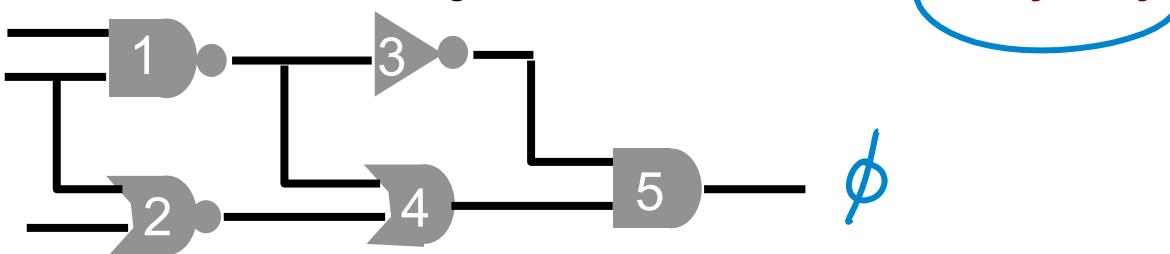
- If **SAT**: networks **not same**, and this pattern makes them give different outputs
- If **unSAT**: yes, **same!**



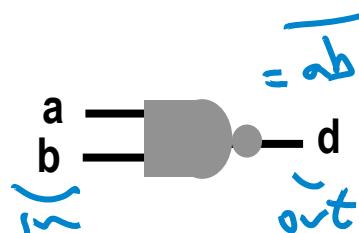
Final Topic: Gates → CNF

- How do I **start** with a **gate-level description** and get **CNF**?

- Isn't this hard? Don't I need Boolean algebra or BDDs? No – it's **really easy**



- Trick: build up CNF one gate at a time



Gate **consistency** function (or gate **satisfiability** function)

$$\Phi_d = [d == (\bar{a}b)] = \overline{d} \oplus \overline{(\bar{a}b)} \rightarrow \\ (\overline{a} + \overline{d})(\overline{b} + \overline{d}) (\overline{\overline{a}} + \overline{b} + \overline{d}) = \overline{d}$$



ASIDE: EXOR vs EXNOR

- **EXOR: Exclusive OR**

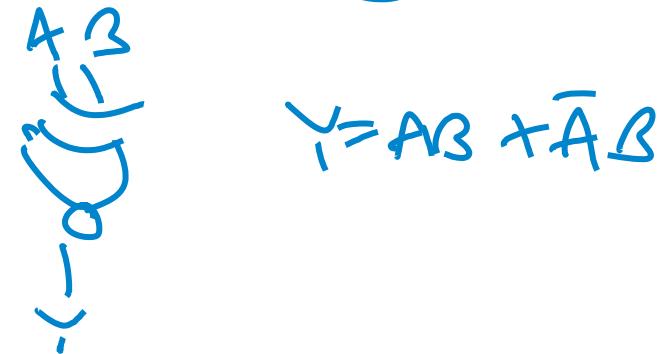
- Write is as: $Y = A \oplus B$
- Output is **1** just if $A \neq B$,
ie, if A is **different** than B



A hand-drawn logic gate diagram for an EXOR gate. It features two inputs labeled A and B at the top, each with a blue arrow pointing down towards the gate. The gate itself is a small circle with a blue output line extending downwards. To the right of the gate, the equation $Y = \bar{A}B + A\bar{B}$ is written in blue.

- **EXNOR: Exclusive NOR**

- Write is as: $\underline{Y = A \oplus B}$ (**I like this**)
- Or like this: $Y = A \odot B$
- Output is **1** just if $A == B$, ie,
if A is **same as, equal to** B

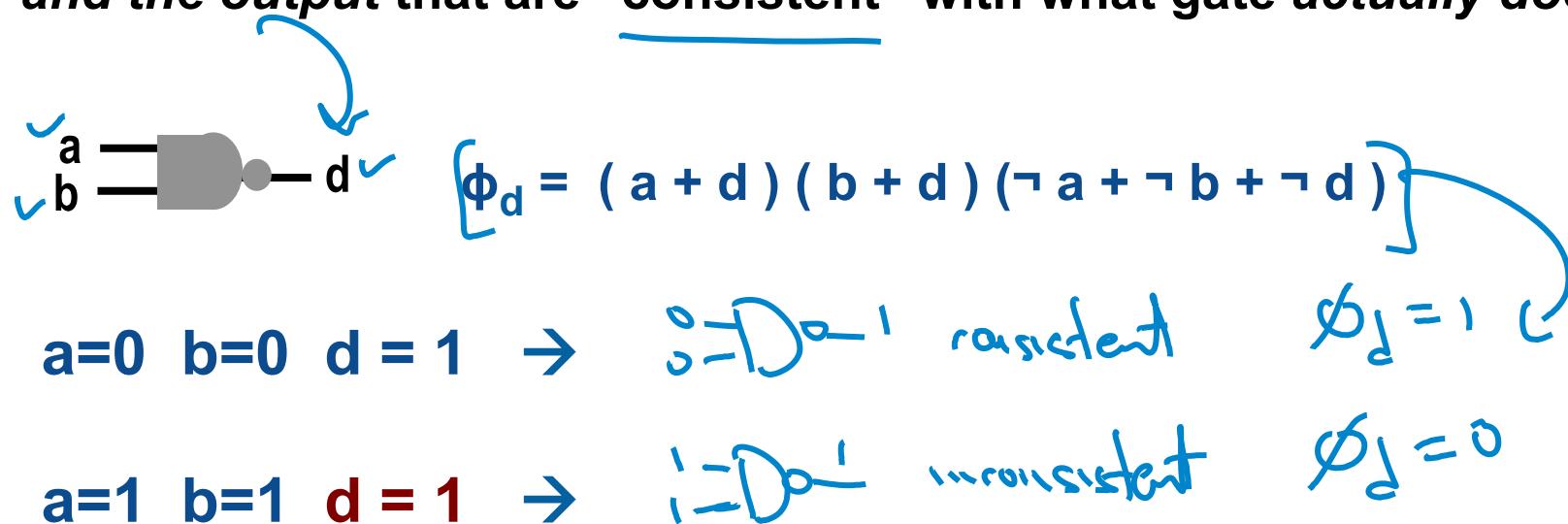


A hand-drawn logic gate diagram for an EXNOR gate. It features two inputs labeled A and B at the top, each with a blue arrow pointing down towards the gate. The gate is a small circle with a blue output line extending downwards. To the right of the gate, the equation $Y = AB + \bar{A}\bar{B}$ is written in blue.



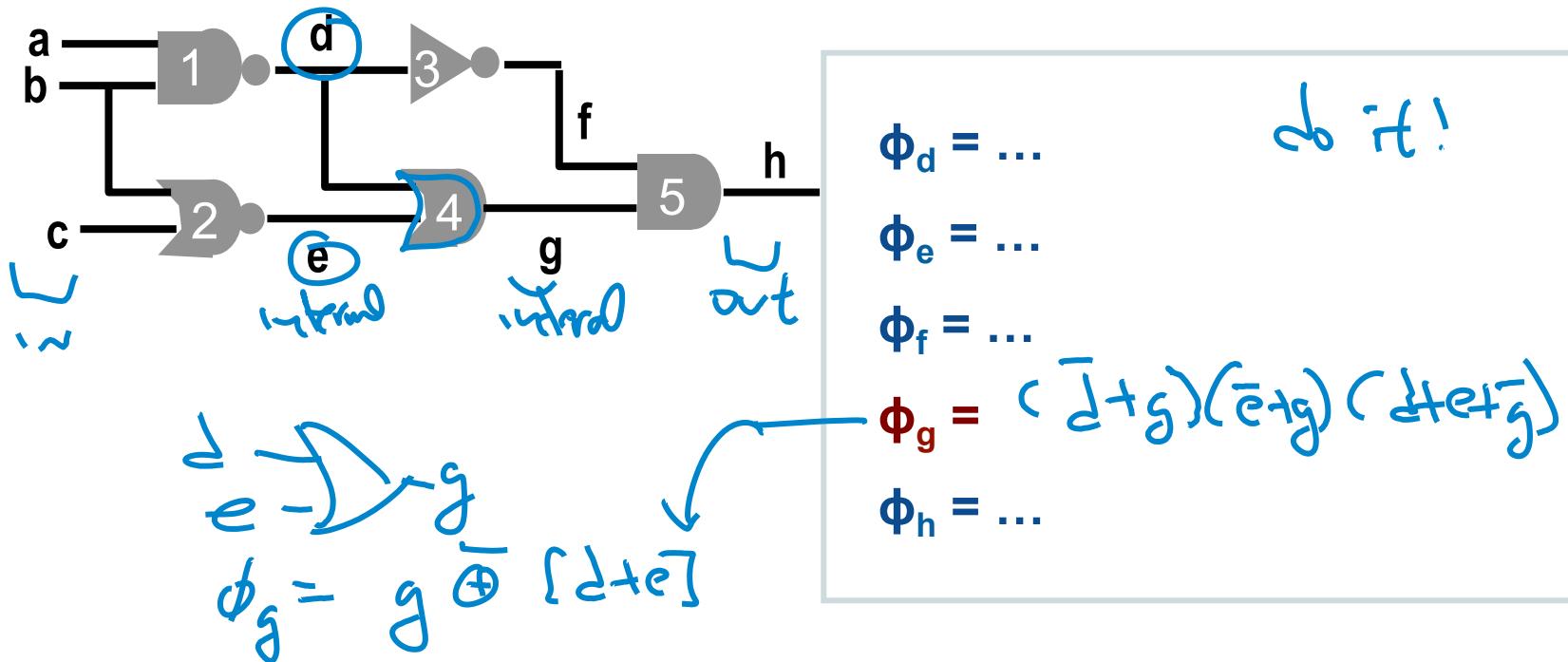
Final Topic: Gates \rightarrow CNF

- Gate consistency function == “1” just for combinations of inputs *and the output* that are “consistent” with what gate *actually does*



Final Topic: Gates → CNF

- For a network: label **each wire**, build **all** gate consistency funcs

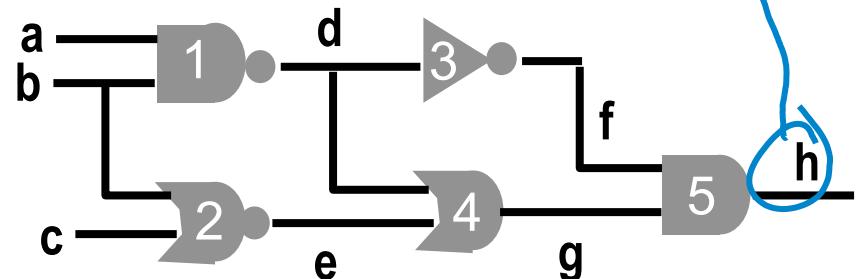


Final Topic: Gates → CNF

- SAT **CNF** for network is simple:
 - Any pattern of **abch** that satisfies this, also makes the gate network output **h=1**

$$\varphi = (\text{output var}) \quad \boxed{W} \quad (\text{k is gate output wire}) \quad \Phi_k$$

$$\begin{aligned}\varphi = h \\ & (a + d) (b + d) (\neg a + \neg b + \neg d) \\ & (\neg b + \neg e) (\neg c + \neg e) (b + c + e) \\ & (\neg d + \neg f) (d + f) \\ & (\neg d + g) (\neg e + g) (d + e + \neg g) \\ & (f + \neg h) (g + \neg h) (\neg f + \neg g + h)\end{aligned}$$



- Only need Boolean algebra/simplification for each **individual** gate-level function
- At network level, just **AND** them all together to get **CNF**



Rules for ALL Kinds of Basic Gates

- **Gate consistency rules from:**

- Fadi Aloul, Igor L. Markov, Karem Sakallah, “MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation,” *J. of Universal Computer Sci.*, vol. 10, no. 12 (2004), 1562-1596.

$$z = (x)$$

(yes this is just a wire)

$$[\bar{x} + z][x + \bar{z}]$$

$$z = \text{NOR}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^n (\bar{x}_i + \bar{z}) \right] \left[\left(\sum_{i=1}^n x_i \right) + z \right]$$

product sum

$$z = \text{OR}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^n (\bar{x}_i + z) \right] \left[\left(\sum_{i=1}^n x_i \right) + \bar{z} \right]$$

$$z = \text{NOT}(x)$$

$$[x + z][\bar{x} + \bar{z}]$$

$$z = \text{NAND}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^n (x_i + z) \right] \left[\left(\sum_{i=1}^n \bar{x}_i \right) + \bar{z} \right]$$

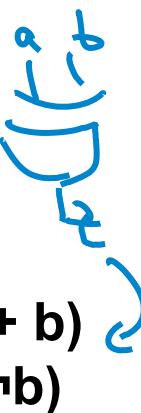
$$Z = \text{AND}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^n (x_i + \bar{z}) \right] \left[\left(\sum_{i=1}^n \bar{x}_i \right) + z \right]$$



Rules for ALL Kinds of Basic Gates

- EXOR/EXNOR gates are rather *unpleasant* for SAT
 - And the basic “either-or” structure makes for some tough SAT search, often
 - And, they have rather large gate consistency functions too
 - Even small 2-input gates create a lot of terms, like this:

$$\begin{aligned} z &= \text{EXOR}(a, b) \\ \Phi_z &= z \overline{\oplus} (a \oplus b) \\ &= (\neg z + \neg a + \neg b) \cdot (\neg z + a + b) \\ &\quad \cdot (z + \neg a + b) \cdot (z + a + \neg b) \end{aligned}$$


$z = \text{EXNOR}(a, b)$

Do it –
good practice!



Using the Rules...

$$z = \text{NAND}(x_1, x_2, \dots, x_n)$$

$$\left[\prod_{i=1}^{n=2} (x_i + z) \right] \left[\left(\sum_{i=1}^{n=2} \bar{x}_i \right) + \bar{z} \right]$$

product

$$n=2: z = \text{NAND}(x_1, x_2)$$



$$\phi_z = (x_1 + z)(x_2 + z)(\neg x_1 + \neg x_2 + \neg z)$$

For these formulae,

\prod means “AND”

\sum means “OR”



Summary

- SAT has largely displaced BDDs for “just solve it” apps 
 - Reason is scalability: can do very large problems faster, more reliably
 - Still, SAT, like BDDs, not guaranteed to find a solution in reasonable time or space
- 40 years old, but still “the” big idea: DPLL
 - Many recent engineering advances make it *stupendously* fast
- Acknowledgements for help with earlier versions of this SAT lec
 - Karem Sakallah
U of Michigan
 - Joao Marques-Silva
University College, Dublin

