VLSI CAD: Logic to Layout

Lecture 4.1

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

Rob A. Rutenbar
University of Illinois

Chris Knapton/Digital Vision/Getty Images

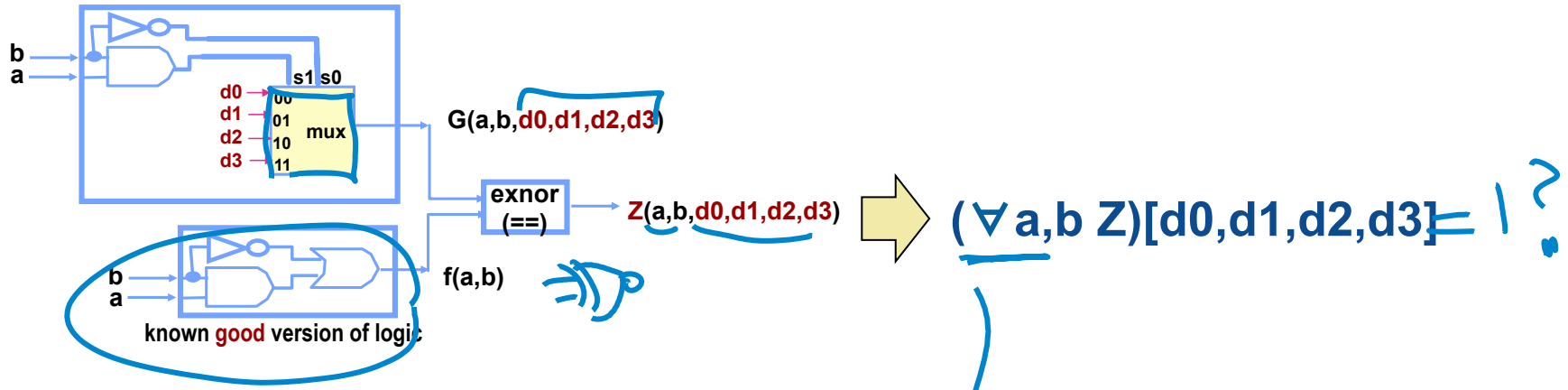
Some Terminology

- **Satisfiability (called "SAT" for short)**
  - Give me an appropriate representation of function **F(x1, x2, … xn)**
  - Find an assignment of the variables ("**vars**" for short) **(x1, x2, … xn)** 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**

© 2013, R.A. Rutenbar

Example: Network Repair



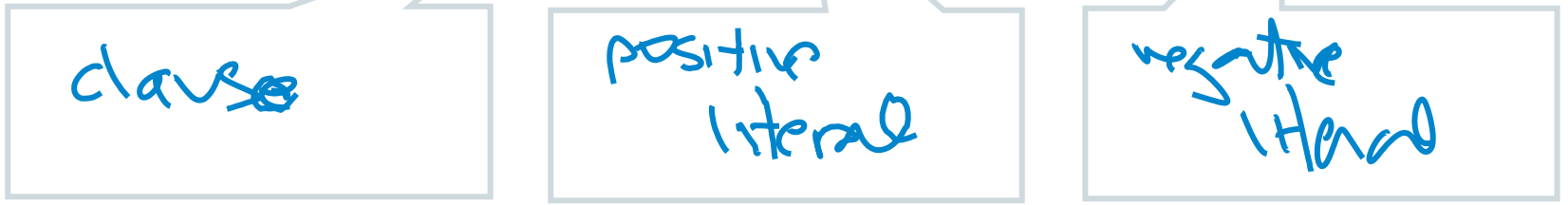
- **Assuming you can build the quantification function (∀a,b Z)**
  - Go find a SAT assignment to get you the **d** values to repair the network
  - Or, if unSAT, know that there is no network repair possible

© 2013, R.A. Rutenbar

Standard SAT Form: CNF

- **Conjunctive Normal Form (CNF) = Standard POS form**

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



- **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"**.

© 2013, R.A. Rutenbar

Assignment to a CNF Formula

- **An *assignment*…**
  - …gives values to some, not necessarily all, of variables (vars) **xi** in **(x1, x2, … , xn)**.
  - ***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**

$$\Phi = (a + \neg b)(\neg a + b + \neg c)(a + c + d)(\neg a + \neg b + \neg c)$$

clause = 0 conflicting

clause = 1 satisfied

unresolved

SAT

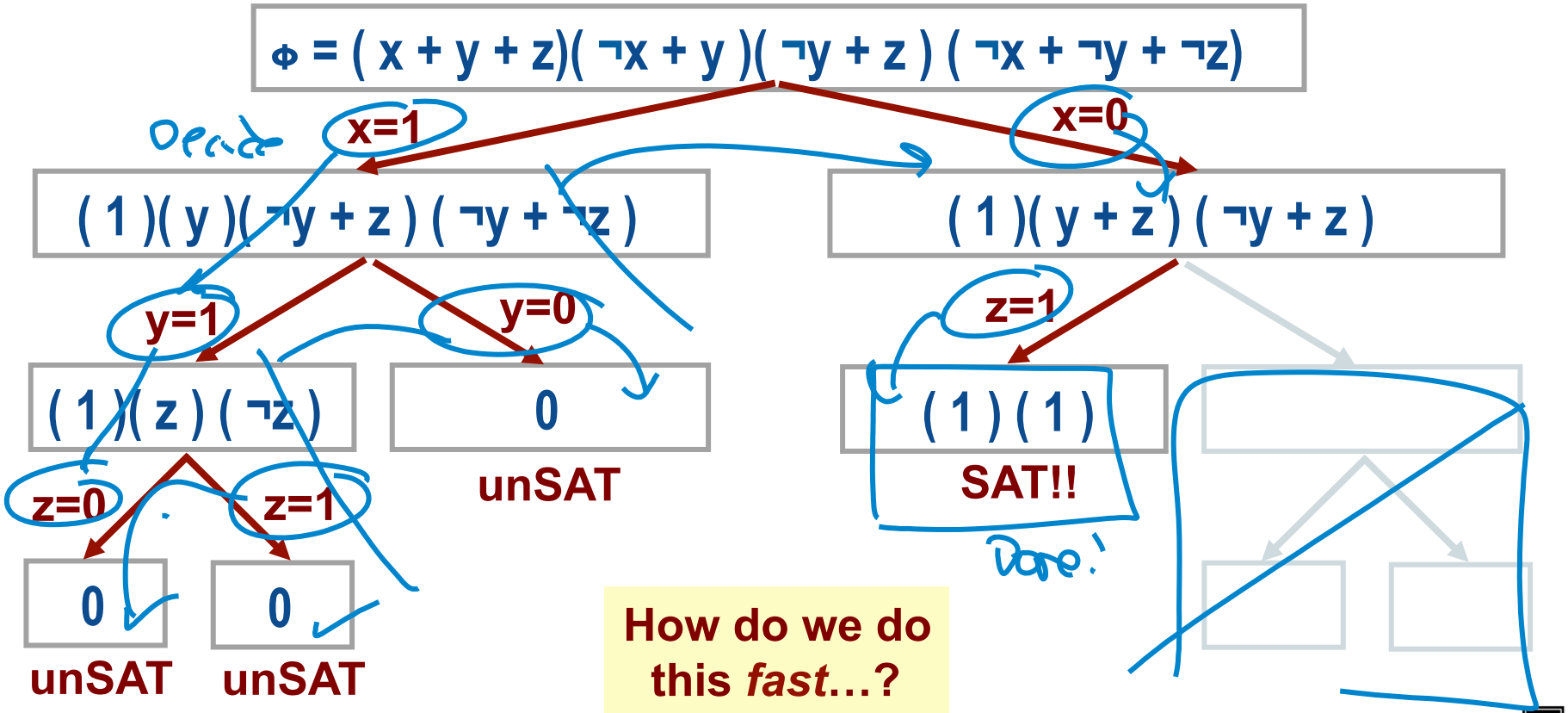
© 2013, R.A. Rutenbar

How Do We "Solve" This?

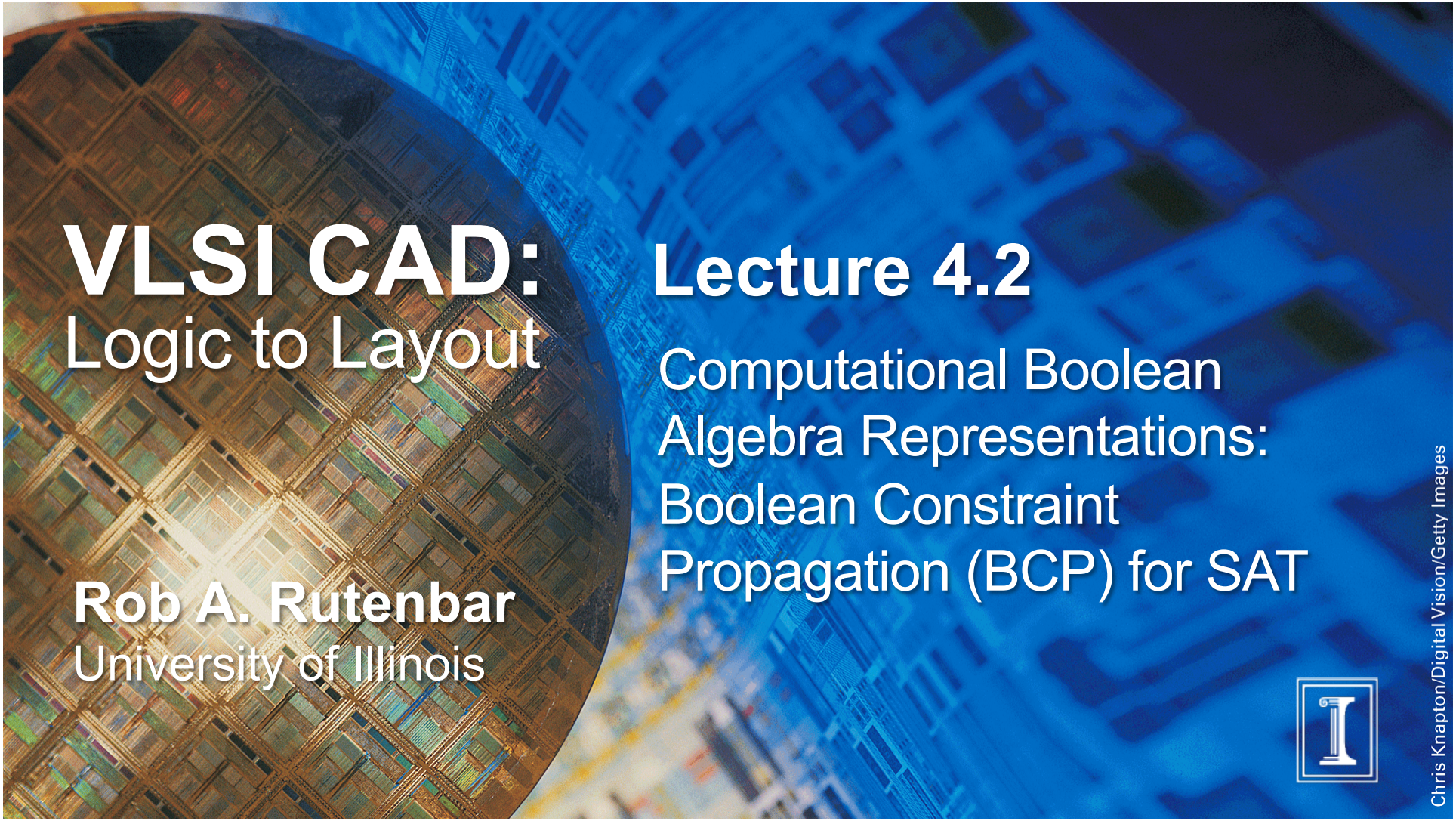
- **Recursively (…surprised?)**
  - Strategy has two big ideas
  - **DECISION:**
    - ***Select*** a variable and ***assign*** its value; ***simplify*** CNF formula as far as you can
    - Hope you can decide if it's SAT, yes/no, without further work
  - **DEDUCTION:**
    - Look at the newly simplified 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

© 2013, R.A. Rutenbar

How Do We "Solve" This?



© 2013, R.A. Rutenbar



VLSI CAD: Logic to Layout

Lecture 4.2

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

Rob A. Rutenbar
University of Illinois

Chris Knapton/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"**

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

SAT SAT

UNIT: 1 unassigned literal = c

c must be 0 → SAT !

Assume: a=1, b=1

© 2013, R.A. Rutenbar

BCP Iterative – Go Until No More Implications

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

$\omega_1 = (\neg x1 + x2)$

$\omega_2 = (\neg x1 + x3 + x9)$ 0

$\omega_3 = (\neg x2 + \neg x3 + x4)$

$\omega_4 = (\neg x4 + x5 + x10)$ 0

$\omega_5 = (\neg x4 + x6 + x11)$ 0

$\omega_6 = (\neg x5 + \neg x6)$

$\omega_7 = (x1 + x7 + \neg x12)$ 0

$\omega_8 = (x1 + x8)$

$\omega_9 = (\neg x7 + \neg x8 + \neg x13)$ 0

No SAT
No BCP
Now what?

- **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:**
  - **x9=0 x10=0 x11=0 x12=1 x13=1**
- **To start…**
  - What are ***obvious*** simplifications when we assign these variables?

© 2013, R.A. Rutenbar

BCP Iterative – Go Until No More Implications

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

$\omega_1 = (\neg x1 + x2)$ — 0

$\omega_2 = (\neg x1 + x3 + x9)$ — 0, 0

Unit

$\omega_3 = (\neg x2 + \neg x3 + x4)$

$\omega_4 = (\neg x4 + x5 + x10)$ — 0

$\omega_5 = (\neg x4 + x6 + x11)$ — 0

$\omega_6 = (\neg x5 + \neg x6)$

$\omega_7 = (x1 + x7 + \neg x12)$ — 1, 0

$\omega_8 = (x1 + x8)$ — 1

SAT

$\omega_9 = (\neg x7 + \neg x8 + \neg x13)$ — 0

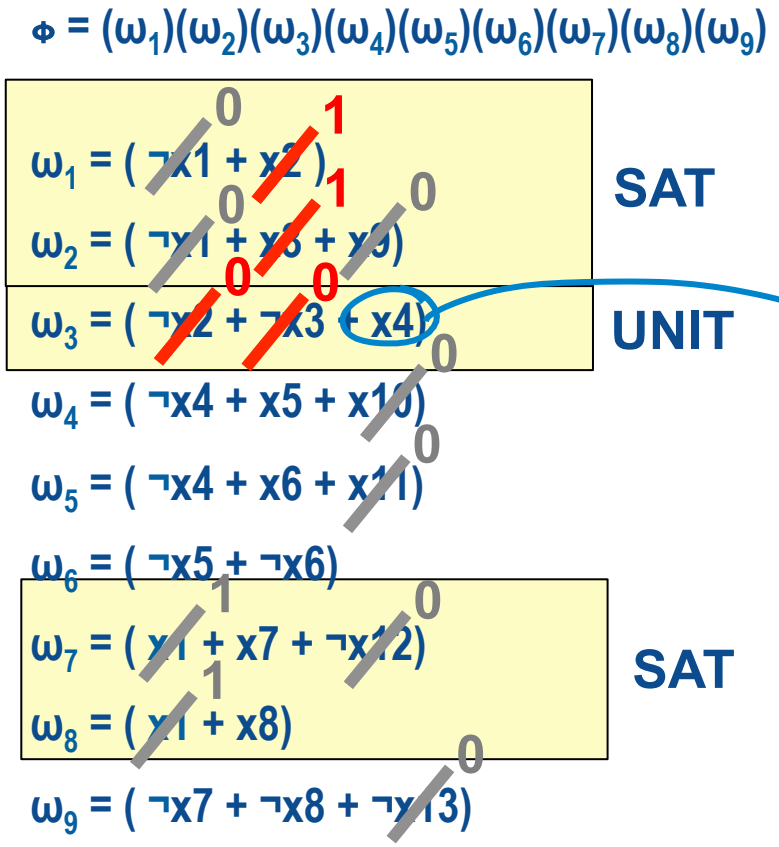
- **Partial assignment is:**
  - x9=0 x10=0 x11=0 x12=1 x13=1
- **Next: Assign a variable to value**
  - Assign x1=1

Implications!

x1=1 → x2=1 && x3=1

© 2013, R.A. Rutenbar

BCP Iterative – Go Until No More Implications

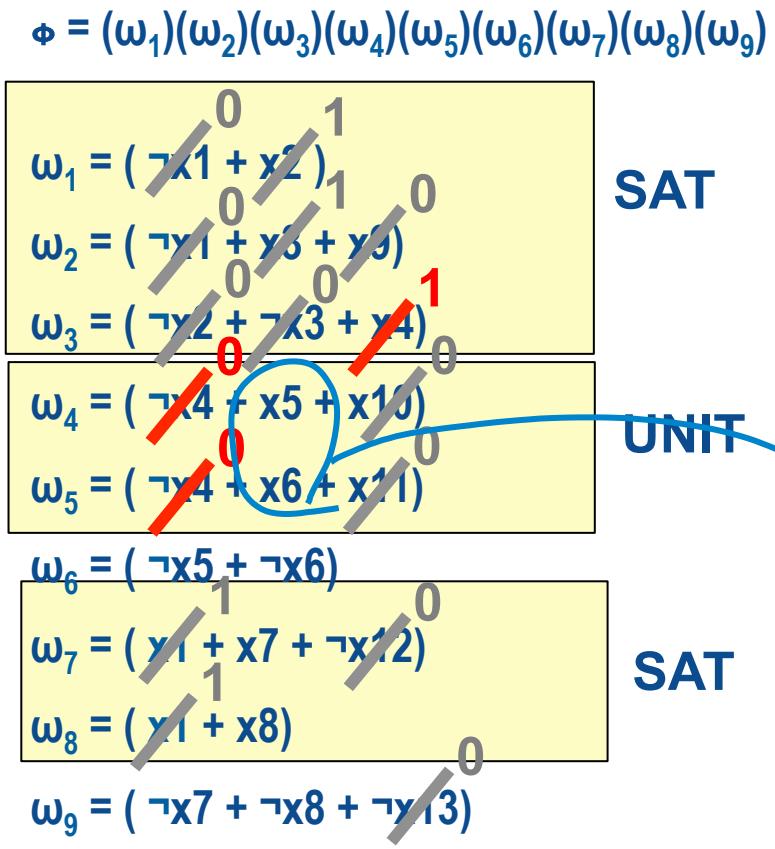


- **Partial assignment is:**
  - x9=0 x10=0 x11=0 x12=1 x13=1
- **Next: Assign a var to value**
  - Assign x1=1
  - **Assign (implied): x2=1, x3=1**

Implications!
x2=1,x3=1 → x4=1

© 2013, R.A. Rutenbar

BCP Iterative – Go Until No More Implications



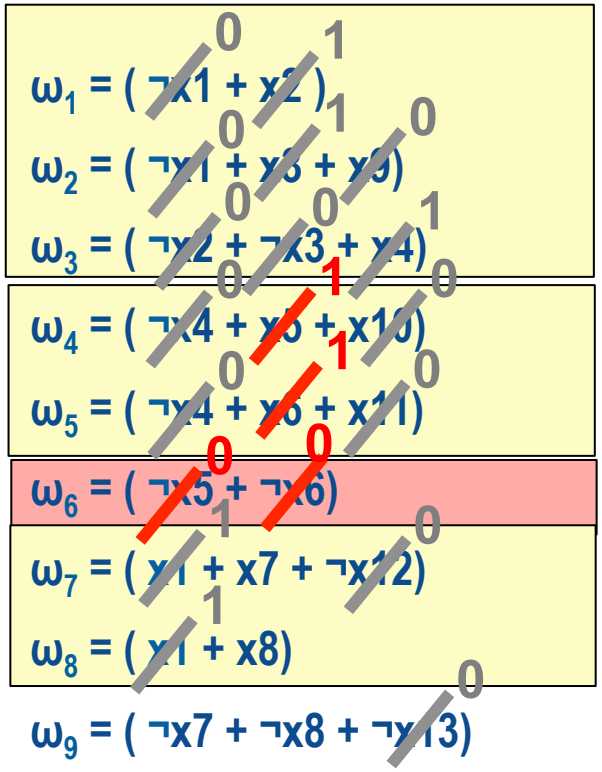
- **Partial assignment is:**
  - **x9=0 x10=0 x11=0 x12=1 x13=1**
- **Next: Assign a var to value**
  - Assign **x1=1**
  - **Assign (implied): x2=1, x3=1**
  - **Assign (implied): x4=1**

Implications!
x4=1 → x5=1 && x6=1

© 2013, R.A. Rutenbar

BCP Iterative – Go Until No More Implications

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



SAT

UNIT

CONFLICT!

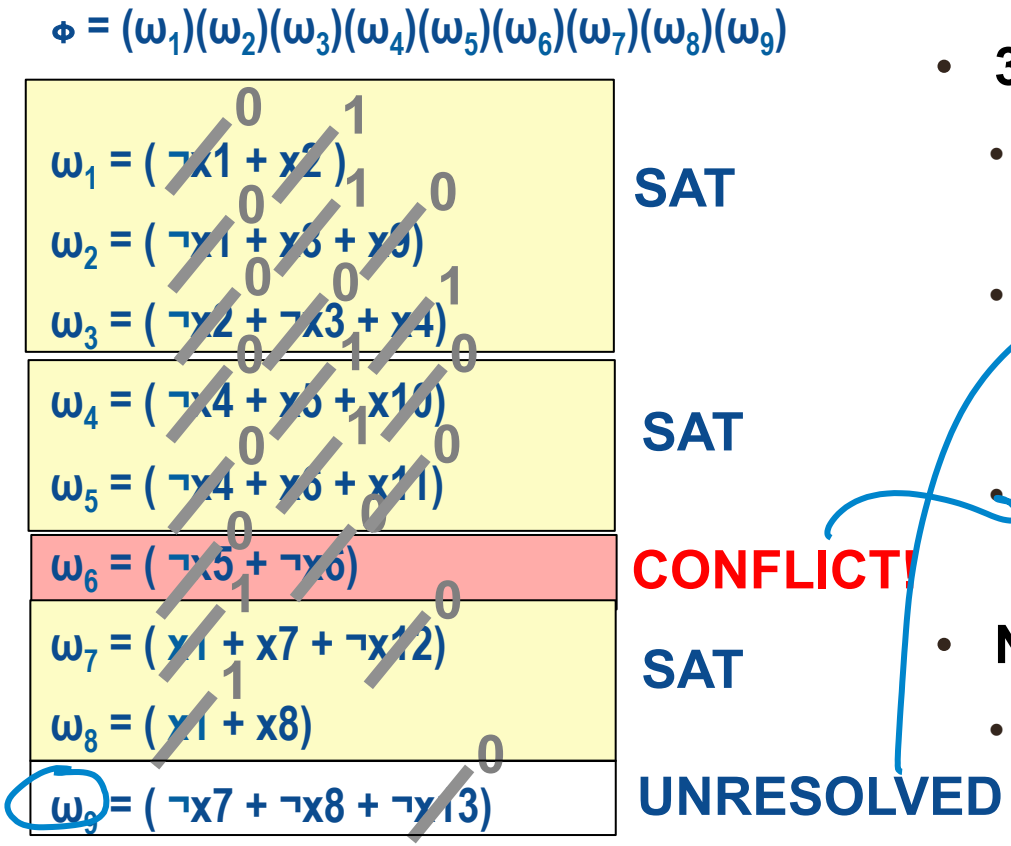
SAT

- **Partial assignment is:**
  - **x9=0 x10=0 x11=0 x12=1 x13=1**
- **Next: Assign a var to value**
  - Assign **x1=1**
  - **Assign (implied): x2=1, x3=1**
  - **Assign (implied): x4=1**
  - **Assign (implied): x5=1 && x6=1**

Conflict →unSAT
x5=1 && x6=1 → clause ω6==0!

© 2013, R.A. Rutenbar

BCP Iterative – Go Until No More Implications



- **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…

© 2013, R.A. Rutenbar

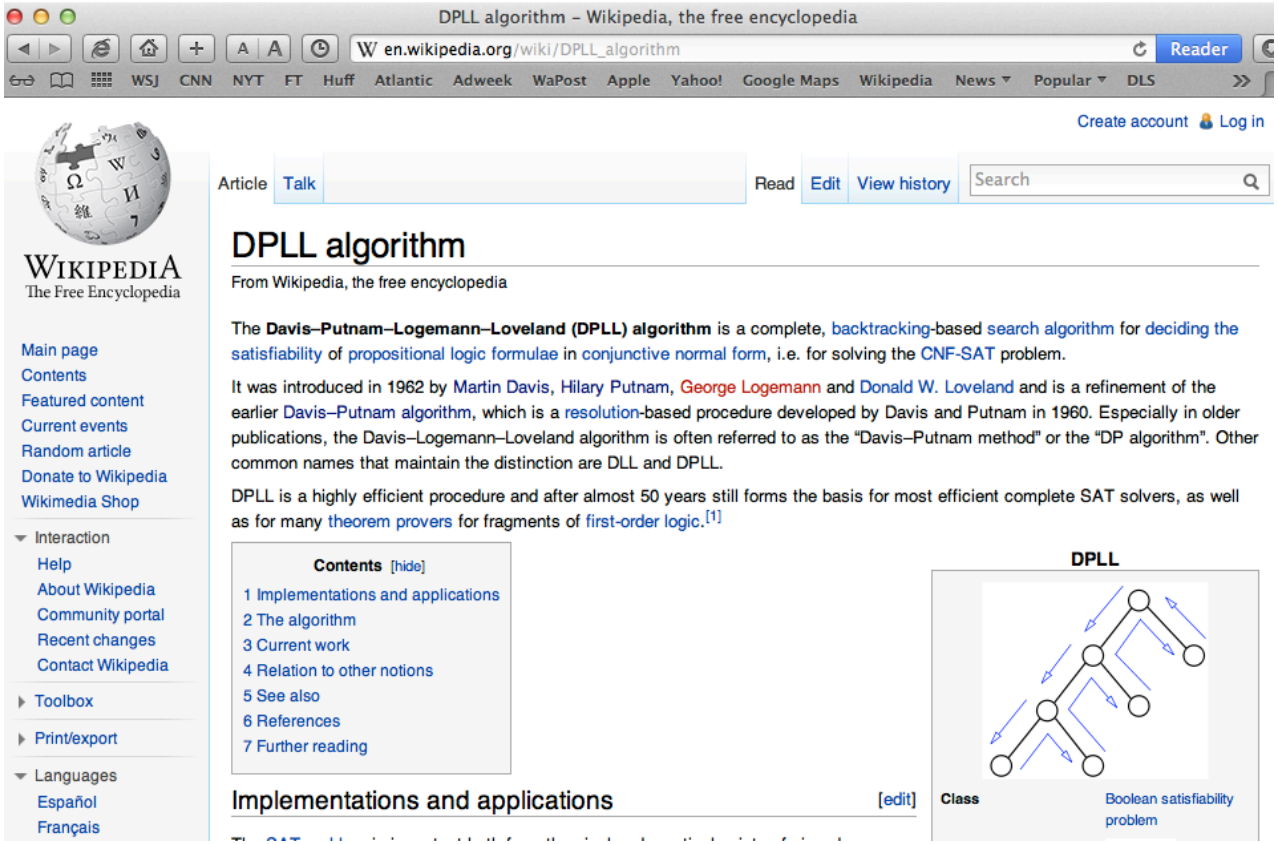
This Has a Famous Name: DPLL

- **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

© 2013, R.A. Rutenbar


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

WIKIPEDIA
The Free Encyclopedia

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
Print/export
Languages
Español
Français

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

DPLL

Class Boolean satisfiability problem

Implementations and applications [edit]

© 2013, R.A. Rutenbar

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 (!!)

© 2013, R.A. Rutenbar

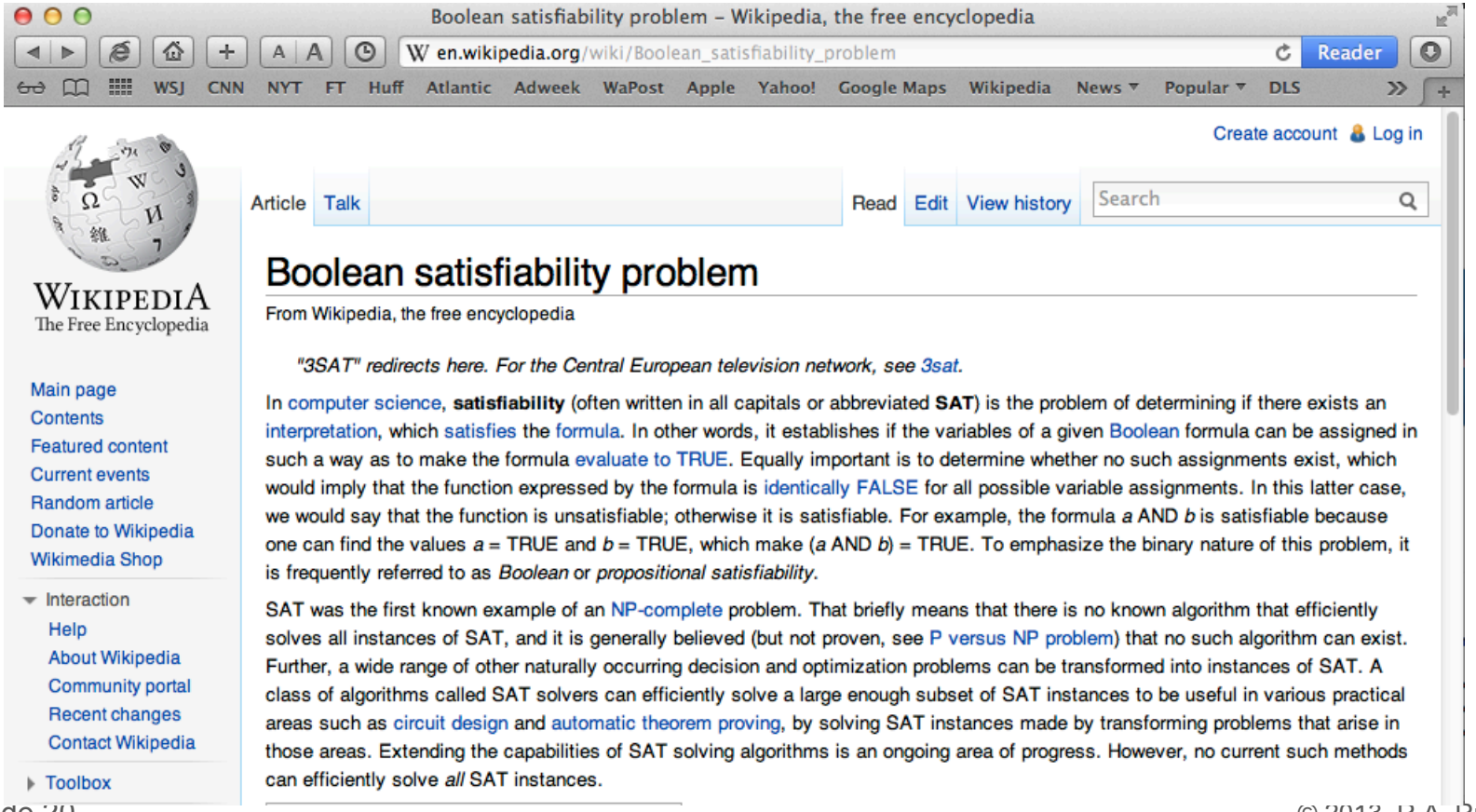
SAT Solvers

- **Many good solvers available online, open source**

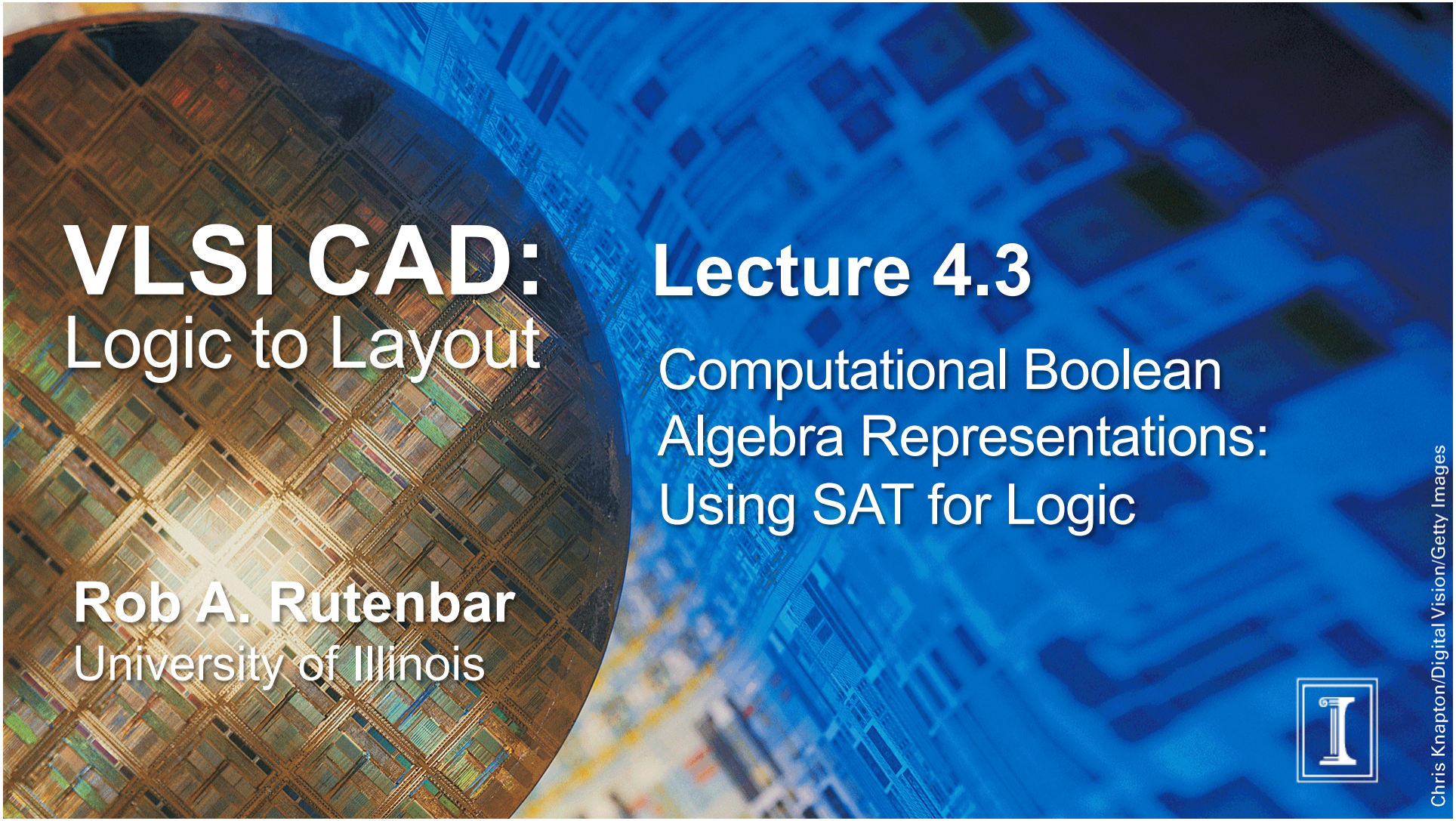
- **Examples**
  - **MiniSAT**, from Niklas Eén, Niklas Sörensson in Sweden.
    - **We are using this one for our MOOC** ✓
  - CHAFF, from Sharad Malik and students, Princeton University
  - GRASP, from Joao Marques-Silva and Karem Sakallah, University of Michigan
  - …and many others too. Go google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

© 2013, R.A. Rutenbar


Lots of Information on SAT…



© 2013, R.A. Rutenbar



VLSI CAD: Logic to Layout

Lecture 4.3

Computational Boolean Algebra Representations: Using SAT for Logic

Rob A. Rutenbar
University of Illinois

Chris Knapton/Digital Vision/Getty Images

BDDs vs SAT Functionality

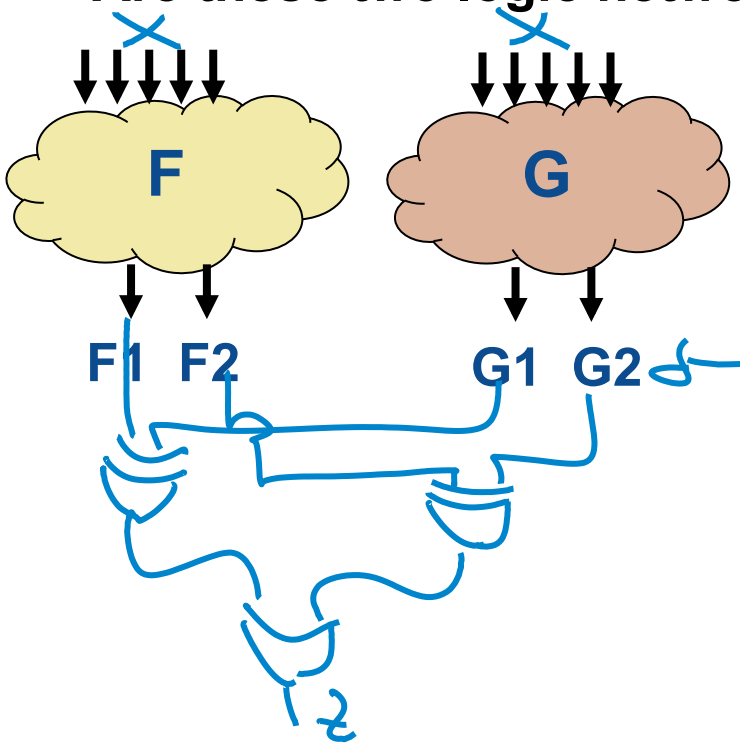
- **BDDs**
  - Often work well for many problems
  - But – no guarantee it will always work
  - Can build BDD to *represent* function **ϕ**
  - But—sometimes **cannot build BDD** with reasonable computer resources (run out of memory **SPACE)**
  - Yes -- builds a **full** representation of **ϕ**
    - Can do a big set of Boolean *manipulations* on data structure
  - Can build **(∃xyz F)** and **(∀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 **(∃xyz F), (∀xyz F)**

© 2013, R.A. Rutenbar

Typical Practical SAT Problem

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



z = 1 SAT iff F ≠ G

if z SAT: find X such that

F(X) ≠ G(X)

Do SAT solve on this new network

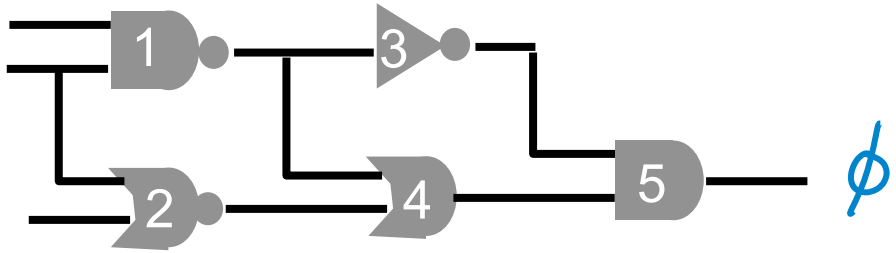
→ If **SAT**: networks **not same**, and this pattern makes them give different outputs

→ If **unSAT**: yes, **same**!

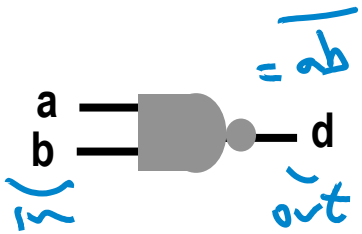
© 2013, R.A. Rutenbar

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)

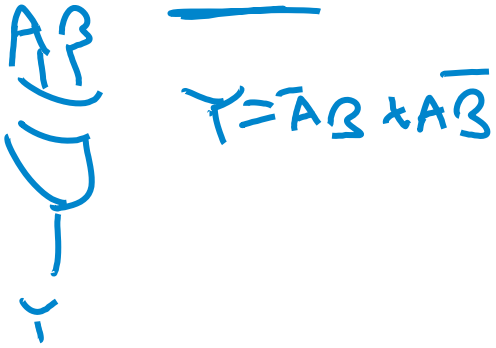
$\Phi_d = [d == (\overline{ab})] =$

$(a+d)(b+d)(\bar{a}+\bar{b}+\bar{d})$

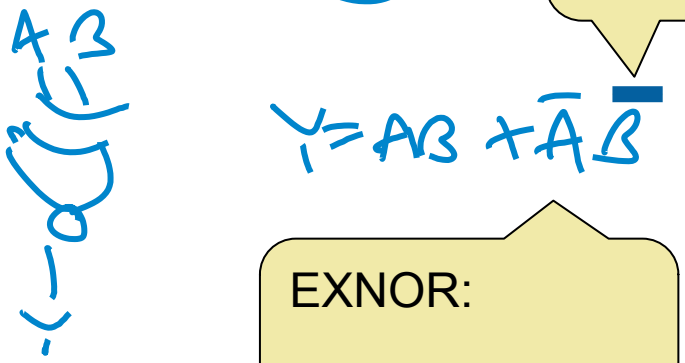
© 2013, R.A. Rutenbar

ASIDE: EXOR vs EXNOR

- **EXOR: Exclusive OR**
  - Write is as: **Y = A ⊕ B**
  - Output is **1** just if **A ≠ B**, ie, if **A** is ***different*** than **B**



- **EXNOR: Exclusive NOR**
  - Write is as: **Y = A ⊕ B** **(I like *this*)**
  - Or like this: **Y = A ⊙ B**
  - Output is **1** just if **A == B**, ie, if **A** is ***same as, equal to*** **B**



Bug fix: add this NEG to B

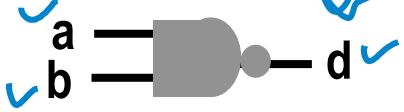
EXNOR:

AB + A' B'

© 2013, R.A. Rutenbar

Final Topic: Gates → CNF

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



$\phi_d = (a + d)(b + d)(\neg a + \neg b + \neg d)$

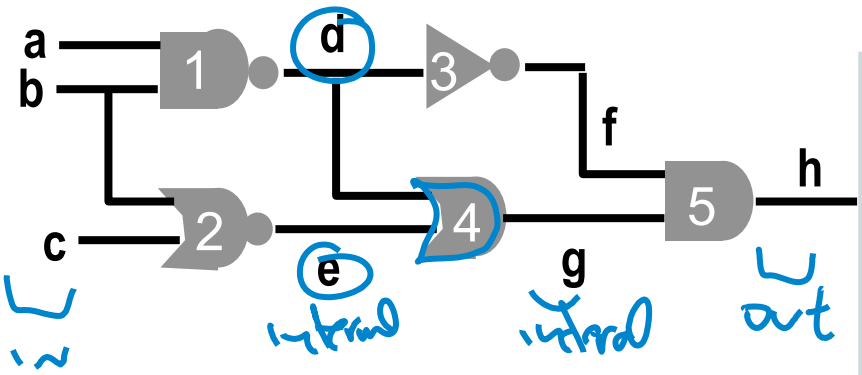
a=0 b=0 d = 1 → 0 0 1 consistent $\phi_d = 1$

a=1 b=1 d = 1 → 1 1 1 inconsistent $\phi_d = 0$

© 2013, R.A. Rutenbar

Final Topic: Gates → CNF

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



Φ_d = …

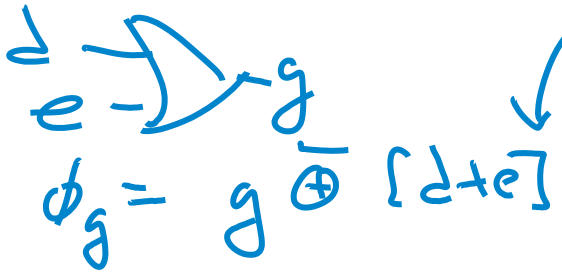
Φ_e = …

Φ_f = …

Φ_g = $(\bar{d}+g)(\bar{e}+g)(d+e+\bar{g})$

Φ_h = …

do it!



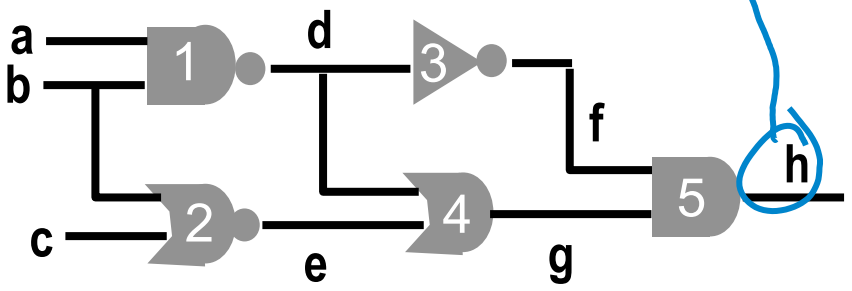
© 2013, R.A. Rutenbar

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**

φ = (output var) $\bigwedge$ (k is gate output wire) Φ_k

φ = h
(a + d) (b + d) (¬a + ¬b + ¬d)
(¬b + ¬e) (¬c + ¬e) (b + c + e)
(¬d + ¬f) (d + f)
(¬d + g) (¬e + g) (d + e + ¬g)
(f + ¬h) (g + ¬h) (¬f + ¬g + h)



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

© 2013, R.A. Rutenbar

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)

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

z=NOR(x1, x2, … xn)

$$\left[\prod_{i=1}^{n}\left(\overline{xi}+\overline{z}\right)\right]\left[\left(\sum_{i=1}^{n}xi\right)+z\right]$$

product sum

z=OR(x1, x2, … xn)

$$\left[\prod_{i=1}^{n}\left(\overline{xi}+z\right)\right]\left[\left(\sum_{i=1}^{n}xi\right)+\overline{z}\right]$$

z=NOT(x)

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

z=NAND(x1, x2, … xn)

$$\left[\prod_{i=1}^{n}\left(xi+z\right)\right]\left[\left(\sum_{i=1}^{n}\overline{xi}\right)+\overline{z}\right]$$

Z=AND(x1, x2, … xn)

$$\left[\prod_{i=1}^{n}\left(xi+\overline{z}\right)\right]\left[\left(\sum_{i=1}^{n}\overline{xi}\right)+z\right]$$

© 2013, R.A. Rutenbar

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:

z=EXOR(a, b)

$$\phi_z = z \oplus (a \oplus b)$$

$$= (\neg z + \neg a + \neg b)\bullet(\neg z + a + b)$$
$$\bullet(z + \neg a + b)\bullet(z + a + \neg b)$$

z=EXNOR(a, b)

Do it –
good practice!

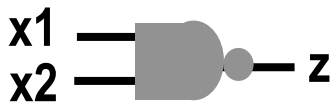
© 2013, R.A. Rutenbar

Using the Rules…

z=NAND(x1, x2, … xn)

$$\left[\prod_{i=1}^{n}(xi+z)\right]\left[\left(\sum_{i=1}^{n}\overline{xi}\right)+\overline{z}\right]$$

n==2: z=NAND(x1, x2)



product

ϕ_z = (x1 + z) (x2 + z) (¬ x1 + ¬ x2 + ¬ z)

For these formulae, $\prod$ means "AND" $\sum$ means "OR"

© 2013, R.A. Rutenbar

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

© 2013, R.A. Rutenbar