

CSE 620: Selective Topics

Introduction to Formal Verification



Master Studies in CSE
Winter 2017
Lecture #5



Dr. Hazem Ibrahim Shehata
Assistant Professor
Dept. of Computer & Systems Engineering

Course Outline

- Computational Boolean Algebra

- Basics

- Shannon Expansion
 - Boolean Difference
 - Quantification Operators
 - + Application to Logic Network Repair

- Validity Checking (Tautology Checking)

- Binary Decision Diagrams (BDD's)

- Satisfiability Checking (SAT solving)

- Model Checking

- Temporal Logics → LTL - CTL

- SMV: Symbolic Model Verifier

- Model Checking Algorithms → Explicit CTL





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 4.1

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

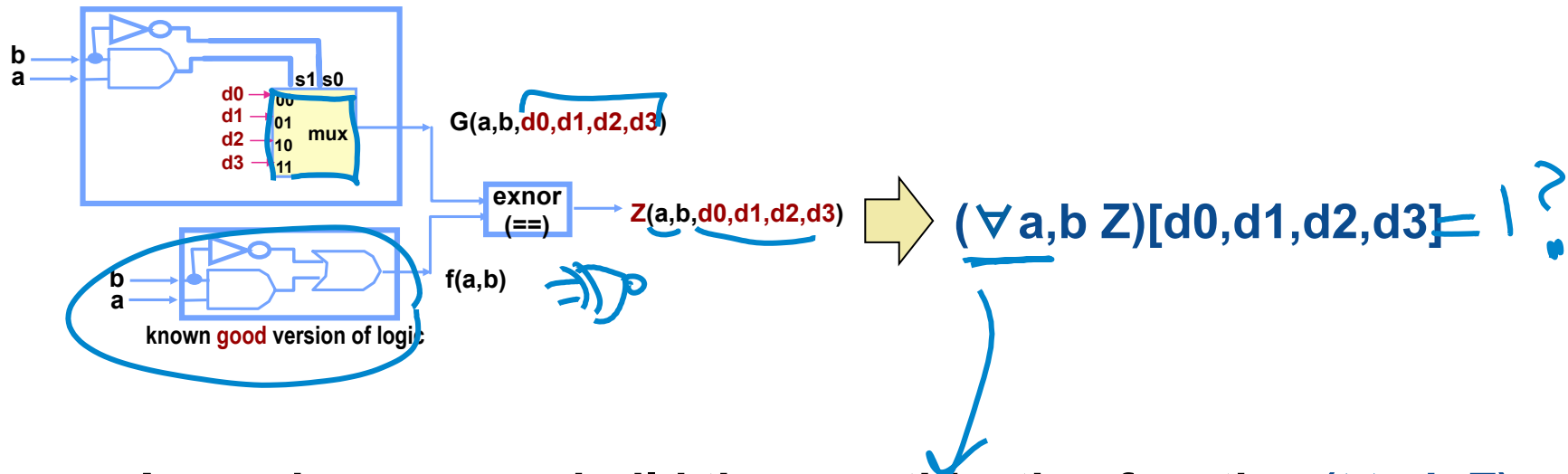


Some Terminology

- **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)

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

$\neg = \text{not}$
 $= \bar{} = \text{c}'$

clause

positive
literal

negative
literal

- **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"**.

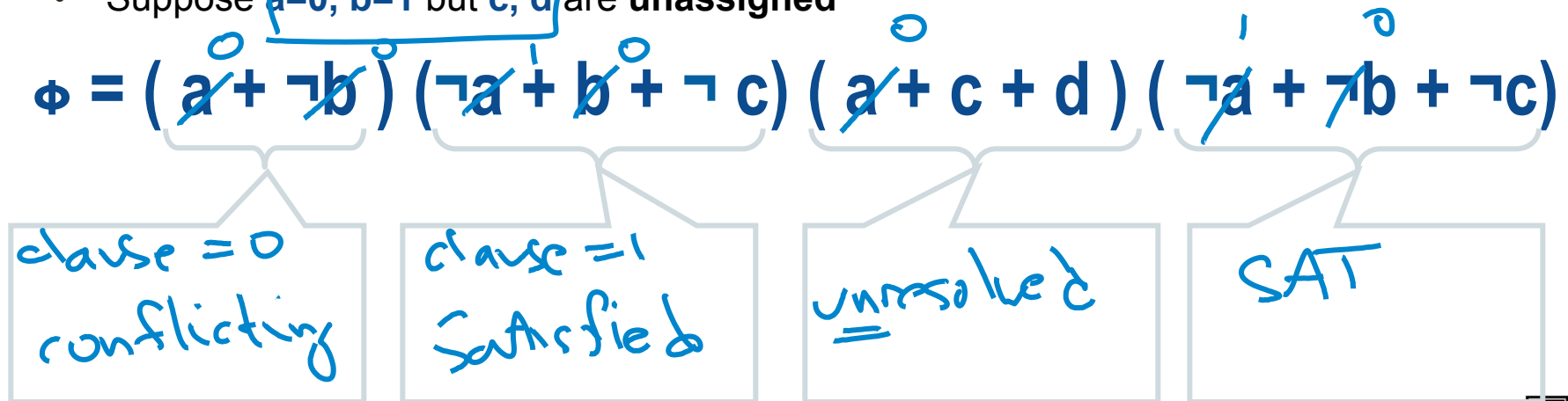
Assignment to a CNF Formula

- 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

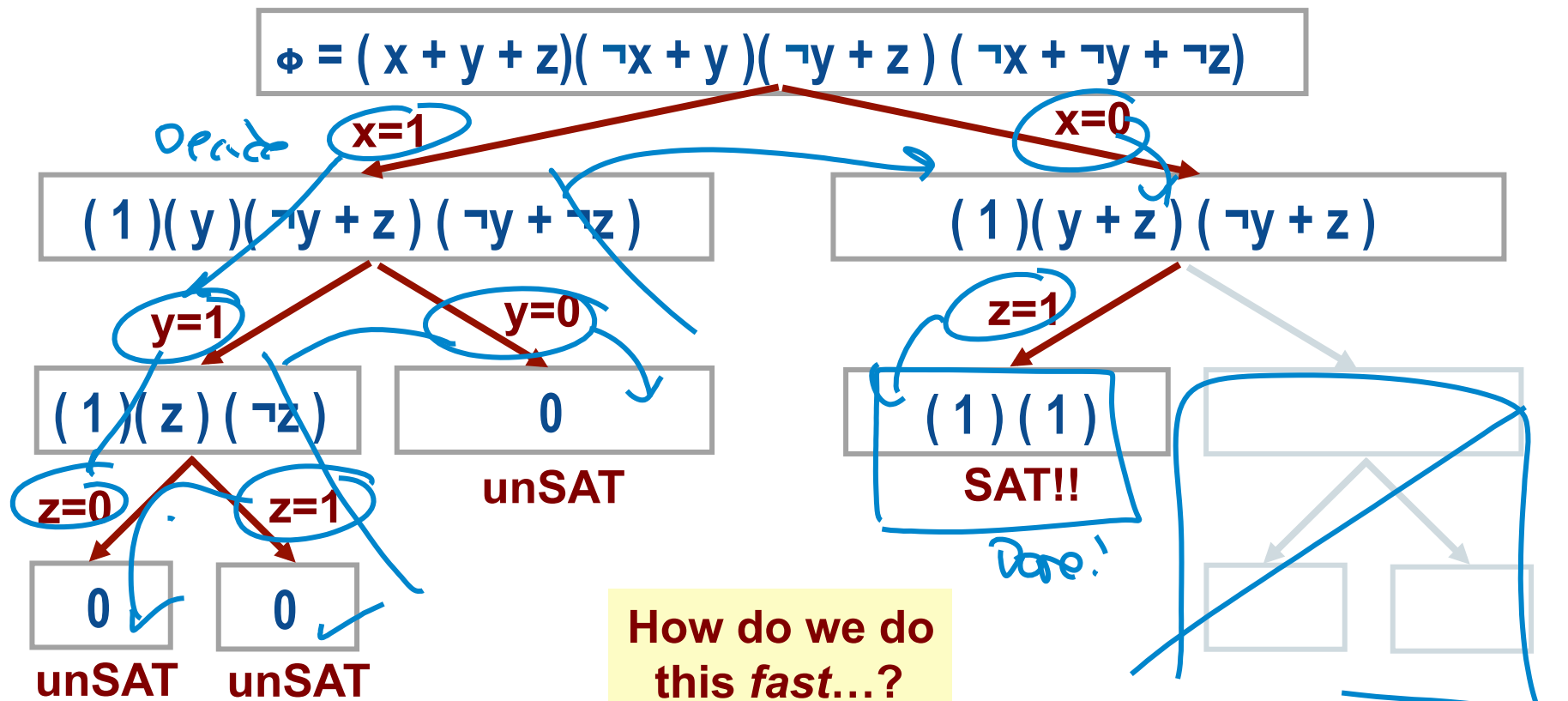


How Do We “Solve” This?

- **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



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**”

$$\Phi = (\overset{1}{\cancel{a}} + c) (\overset{1}{\cancel{b}} + c) (\overset{0}{\cancel{\neg a}} + \overset{0}{\cancel{\neg b}} + \textcircled{\neg c})$$

SAT SAT

Assume:
a=1, b=1

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

BCP Iterative – Go Until No More Implications

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

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

$$\omega_2 = (\neg x_1 + x_3 + x_9)$$

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

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

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

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

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

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

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

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:**

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

- **To start...**

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



BCP Iterative – Go Until No More Implications

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

Unit

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

$$\omega_2 = (\neg x_1 + x_3 + x_9)$$

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

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

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

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

SAT

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

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

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

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

Implications!
 $x_1=1 \rightarrow x_2=1 \ \&\& \ x_3=1$

BCP Iterative – Go Until No More Implications

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

$\omega_1 = (\neg x_1 + x_2)$ $\omega_2 = (\neg x_1 + x_5 + x_9)$ $\omega_3 = (\neg x_2 + \neg x_3 + x_4)$ $\omega_4 = (\neg x_4 + x_5 + x_{10})$ $\omega_5 = (\neg x_4 + x_6 + x_{11})$ $\omega_6 = (\neg x_5 + \neg x_6)$ $\omega_7 = (x_1 + x_7 + \neg x_{12})$ $\omega_8 = (x_1 + x_8)$ $\omega_9 = (\neg x_7 + \neg x_8 + \neg x_{13})$	<p>SAT</p> <p>UNIT</p> <p>SAT</p>
--	-----------------------------------

- Partial assignment is:
 - $x_9=0$ $x_{10}=0$ $x_{11}=0$ $x_{12}=1$ $x_{13}=1$
- 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

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

$$\begin{aligned}\omega_1 &= (\neg x_1 + x_2) \\ \omega_2 &= (\neg x_1 + x_5 + x_9) \\ \omega_3 &= (\neg x_2 + \neg x_3 + x_4)\end{aligned}$$

SAT

$$\begin{aligned}\omega_4 &= (\neg x_4 + x_5 + x_{10}) \\ \omega_5 &= (\neg x_4 + x_6 + x_{11})\end{aligned}$$

UNIT

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

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

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

SAT

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

- Partial assignment is:
 - $x_9=0$ $x_{10}=0$ $x_{11}=0$ $x_{12}=1$ $x_{13}=1$
- Next: Assign a var to value
 - Assign $x_1=1$
 - Assign (implied): $x_2=1$, $x_3=1$
 - Assign (implied): $x_4=1$

Implications!

$$x_4=1 \rightarrow x_5=1 \ \&\& \ x_6=1$$

BCP Iterative – Go Until No More Implications

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

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

SAT

UNIT

CONFLICT!

SAT

- Partial assignment is:
 - $x_9=0$ $x_{10}=0$ $x_{11}=0$ $x_{12}=1$ $x_{13}=1$
- 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): $x_5=1 \ \&\& \ x_6=1$

Conflict \rightarrow unSAT

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



BCP Iterative – Go Until No More Implications

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

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

The screenshot shows the Wikipedia page for the DPLL algorithm. The browser window title is "DPLL algorithm - Wikipedia, the free encyclopedia". The URL is "en.wikipedia.org/wiki/DPLL_algorithm". The page features the Wikipedia logo on the left, a navigation sidebar with links like "Main page", "Contents", and "Interaction", and a main content area. The article title is "DPLL algorithm". Below the title, it states "From Wikipedia, the free encyclopedia". The text describes the DPLL algorithm as a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form. It mentions its introduction in 1962 by Martin Davis, Hilary Putnam, George Logemann, and Donald W. Loveland, and its refinement of the earlier Davis-Putnam algorithm. A "Contents" box lists sections: 1 Implementations and applications, 2 The algorithm, 3 Current work, 4 Relation to other notions, 5 See also, 6 References, and 7 Further reading. The first section, "Implementations and applications", is currently selected. To the right of the text is a diagram titled "DPLL" showing a search tree with nodes and edges, illustrating the algorithm's backtracking process. Below the diagram, the text "Class" and "Boolean satisfiability problem" are visible.

Slide 17

© 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 (!!)

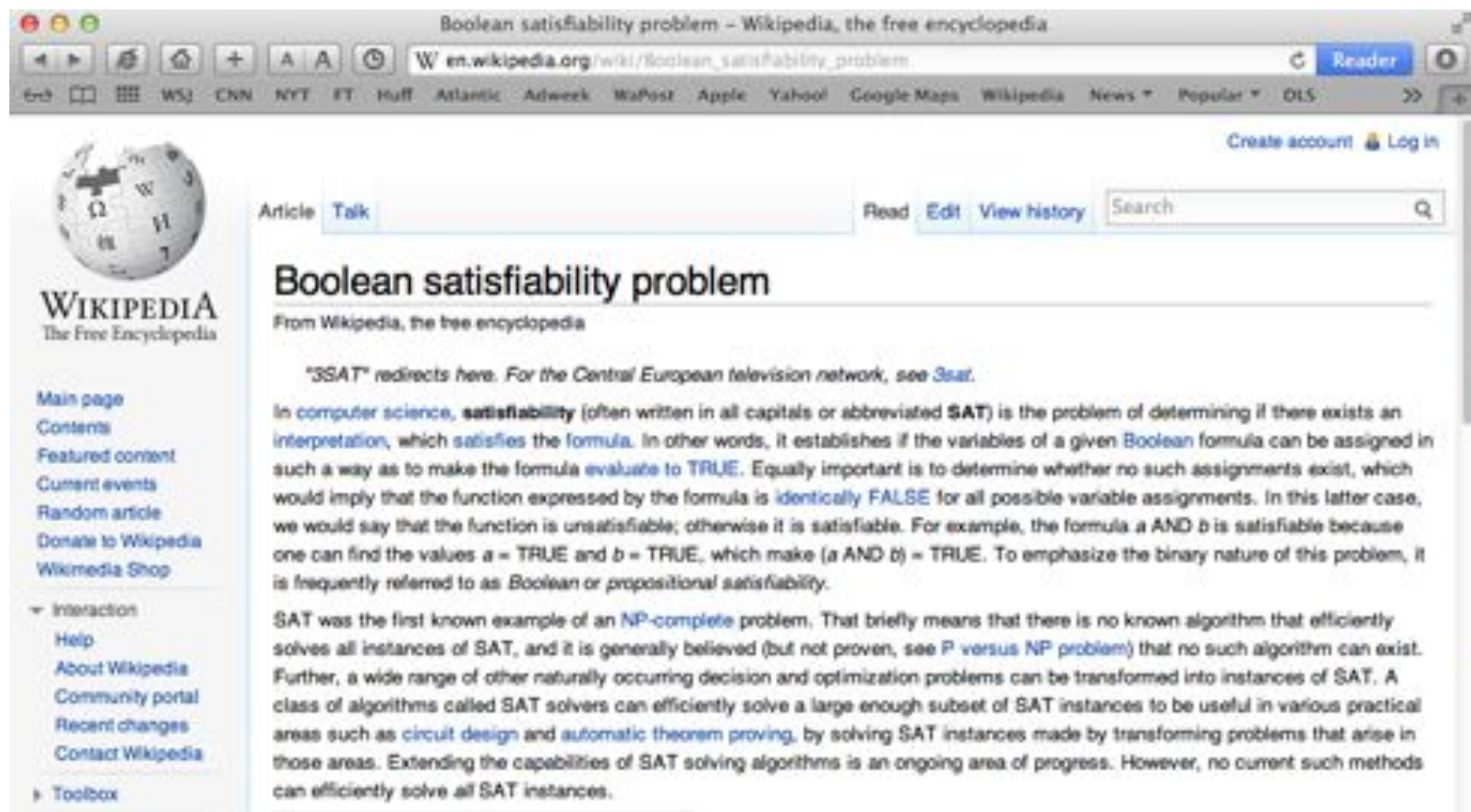


SAT Solvers

- **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...





VLSI CAD: Logic to Layout

Rob A. Rutenbar
University of Illinois

Lecture 4.3

Computational Boolean
Algebra Representations:
Using SAT for Logic



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

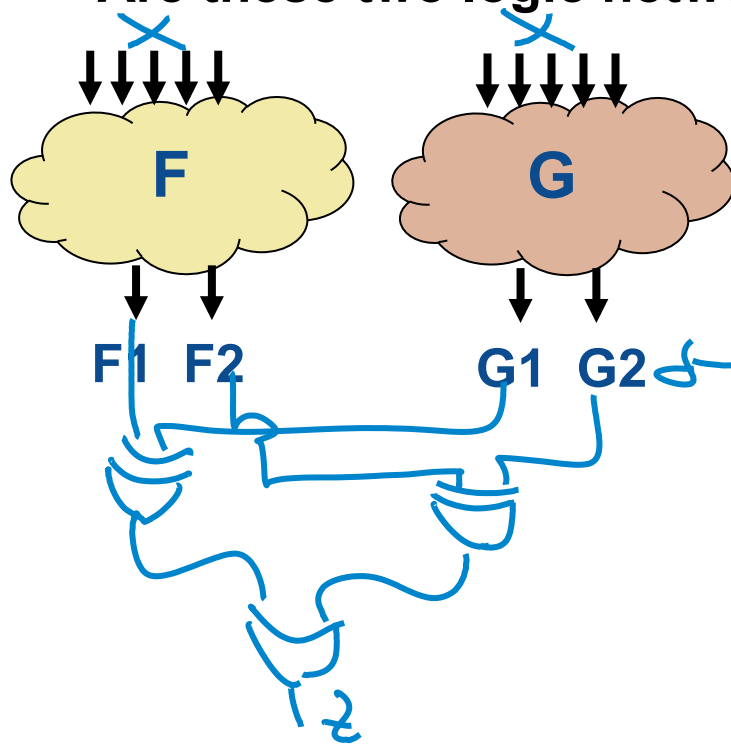
- Often works well for many problems
- But no guarantee it will always work
- Can solve for SAT (y/n) on function ϕ
- But—sometimes **cannot find SAT** with reasonable computer resources (run out of **TIME** doing search)

[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)? $f=6?$



$z=1 \text{ SAT} \iff F \neq 6$

$\text{if } z \text{ SAT: find } x \text{ such that}$

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

Do SAT solve on this new network

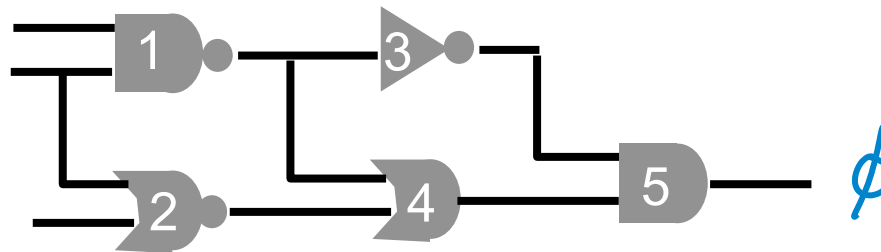
→ If **SAT**: networks **not same**,
and this pattern makes them
give different outputs

→ If **unSAT**: yes, **same**!

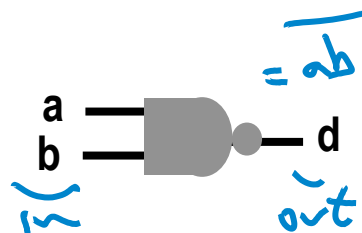
Final Topic: Gates \rightarrow CNF

- 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)

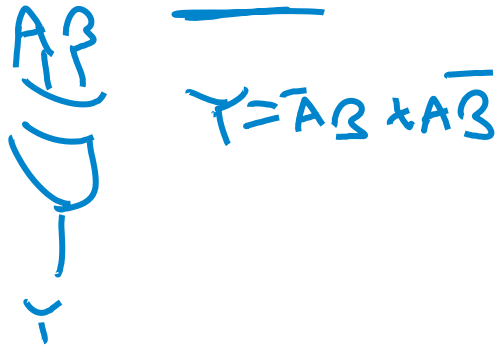
$$\phi_d = [d == (\overline{a}b)] = d \oplus (\overline{a}b)$$

$$(a+d)(b+d)(\overline{a}+\overline{b}+1) = \phi_d$$

ASIDE: EXOR vs EXNOR

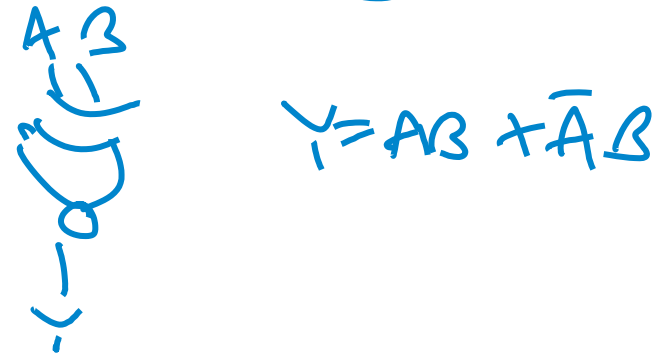
- **EXOR: Exclusive OR**

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



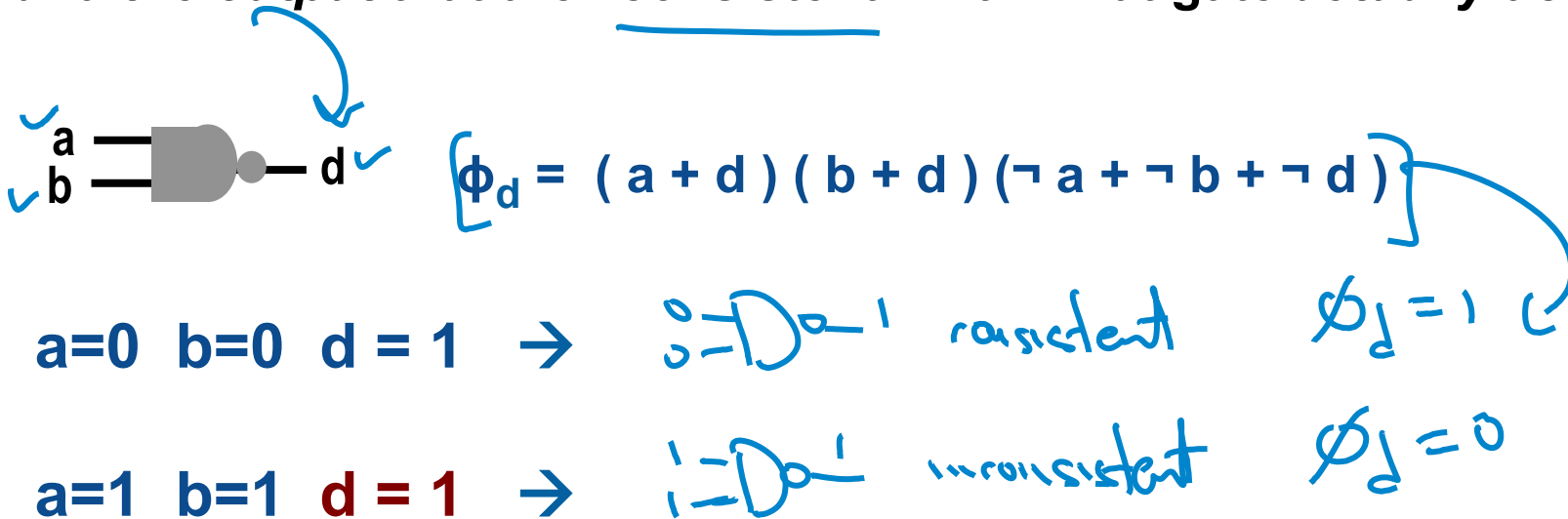
- **EXNOR: Exclusive NOR**

- Write is as: $Y = A \oplus \bar{B}$ (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**



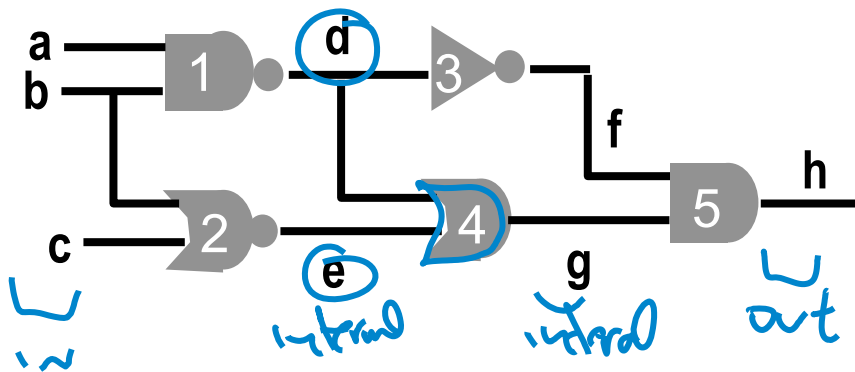
Final Topic: Gates \rightarrow CNF

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



Final Topic: Gates \rightarrow CNF

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



$$\phi_g = g \oplus [d+e]$$

$$\phi_d = \dots$$

$$\phi_e = \dots$$

$$\phi_f = \dots$$

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

$$\phi_h = \dots$$

do it!

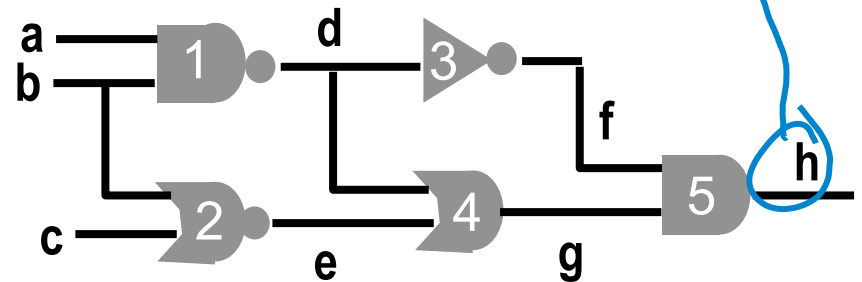
Final Topic: Gates \rightarrow CNF

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

$\phi = (\text{output var}) \bigwedge_{k \text{ is gate output wire}} \phi_k$

$\phi = h$

$(a + d)(b + d)(\neg a + \neg b + \neg d)$
 $(\neg b + \neg e)(\neg c + \neg e)(b + c + e)$
 $(\neg d + \neg f)(d + f)$
 $(\neg d + g)(\neg e + g)(d + e + \neg g)$
 $(f + \neg h)(g + \neg h)(\neg f + \neg g + h)$



- 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:

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

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

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



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

**Do it –
good practice!**

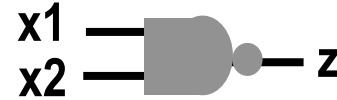


Using the Rules...

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

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

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




product

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

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

