CSE 620: Selective Topics

# Introduction to Formal Verification

Master Studies in CSE

Fall 2016

**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

## Lecture 4.1

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

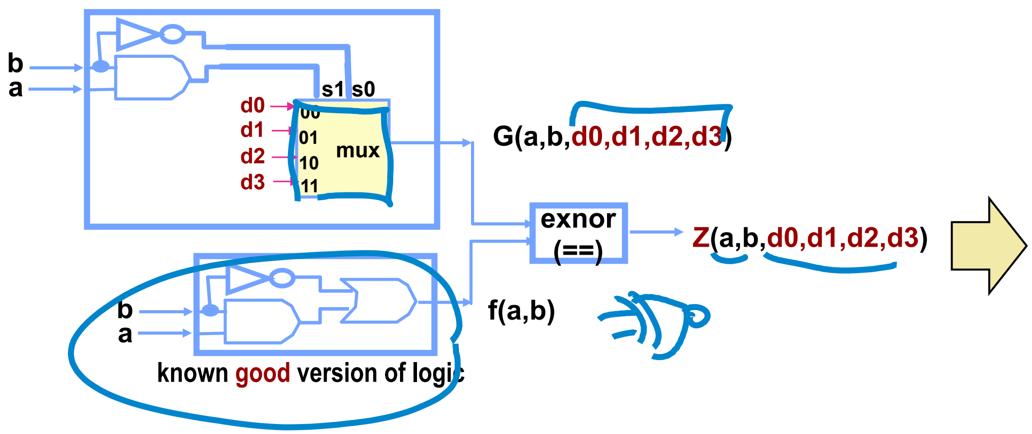
**Rob A. Rutenbar**
University of Illinois

# 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**

# Example:  Network Repair




$(\forall a,b\ Z)[d0,d1,d2,d3] = 1$ ?

- **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

(SOP)

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

$$\Phi = \underbrace{(a + c)}_{\text{clause}}\,(b + \underbrace{c}_{\text{positive literal}})\,(\neg a + \neg b + \underbrace{\neg c}_{\text{negative literal}})$$

$\neg$ = not

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

- **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

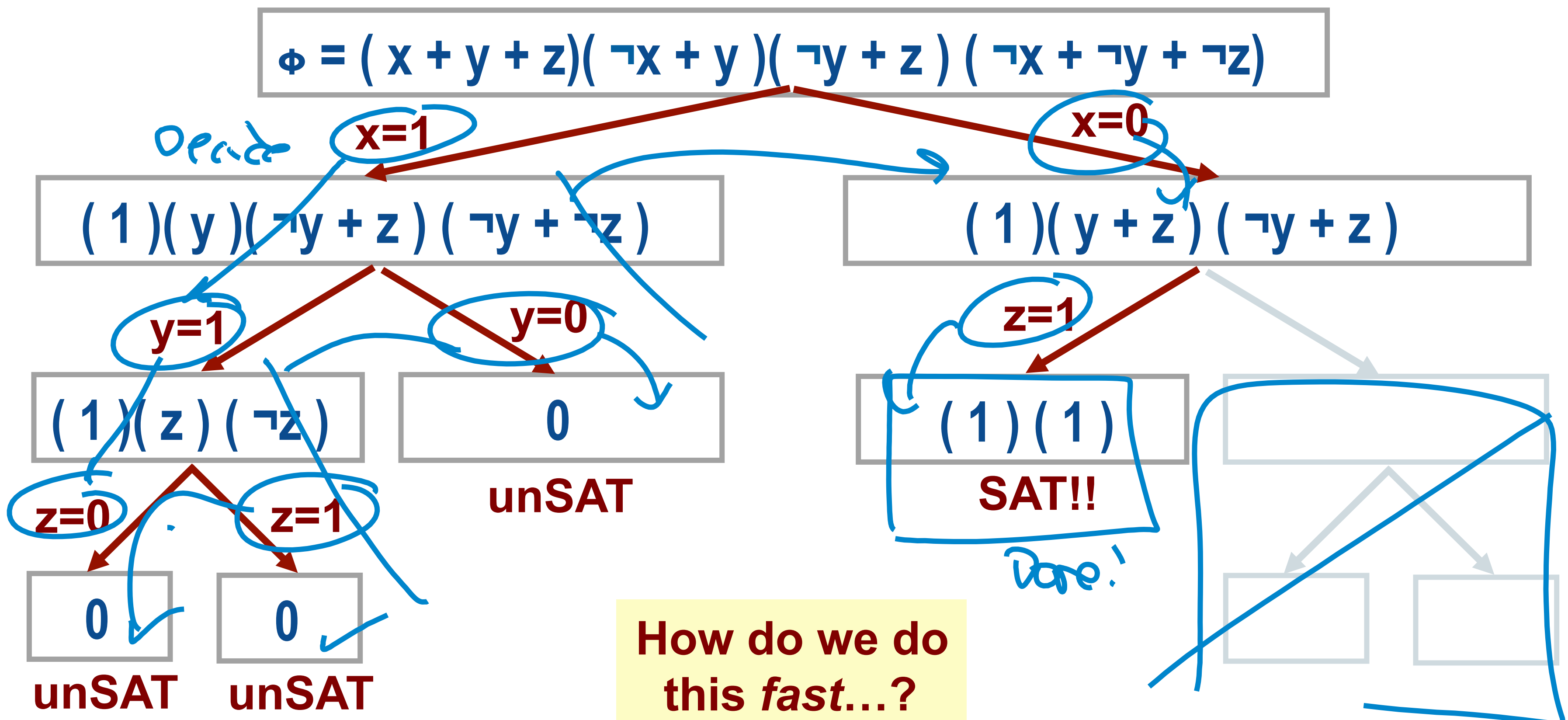
- **An *assignment*…**
  - …gives values to some, not necessarily all, of variables (vars) **xi** in **(x1, x2, … , xn)**.
  - ***Complete*** assignment: assigns value to all vars. ***Partial***: some, not all, have values
- **Assignment means we can evaluate status of the clauses**
  - Suppose $\boxed{a=0,\ b=1 \text{ but } c,\ d}$ are **unassigned**

$$\Phi = \underbrace{(\cancel{a} + \cancel{\neg b})}_{\text{clause = 0 conflicting}}\ \underbrace{(\cancel{\neg a} + \cancel{b} + \neg c)}_{\text{clause = 1 Satisfied}}\ \underbrace{(\cancel{a} + c + d)}_{\text{unresolved}}\ \underbrace{(\cancel{\neg a} + \cancel{\neg b} + \neg c)}_{\text{SAT}}$$

# 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?




**How do we do this *fast*…?**

# 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"**

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

SAT SAT

**Assume:**
**a=1, b=1**

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

# BCP Iterative – Go Until No More Implications

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

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

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

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

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

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

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

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

$\omega_8 = ( x1 + x8)$

$\omega_9 = ( \neg x7 + \neg x8 + \cancel{\neg x13})$ 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?

# BCP Iterative – Go Until No More Implications

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

$\omega_1 = ( \overset{0}{\cancel{\neg x1}} + x2 )$

$\omega_2 = ( \overset{0}{\cancel{\neg x1}} + x3 + \overset{0}{\cancel{x9}})$

**Unit**

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

$\omega_4 = ( \neg x4 + x5 + \overset{0}{\cancel{x10}})$

$\omega_5 = ( \neg x4 + x6 + \overset{0}{\cancel{x11}})$

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

$\omega_7 = ( \overset{1}{\cancel{x1}} + x7 + \overset{0}{\cancel{\neg x12}})$

$\omega_8 = ( \overset{1}{\cancel{x1}} + x8)$

**SAT**

$\omega_9 = ( \neg x7 + \neg x8 + \overset{0}{\cancel{\neg x13}})$

- **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

Slide 11

# BCP Iterative – Go Until No More Implications

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

$\omega_1 = ( \cancel{\neg x1}^{0} + \cancel{x2}^{1} )$ — SAT

$\omega_2 = ( \cancel{\neg x1}^{0} + \cancel{x3}^{1} + \cancel{x9}^{0} )$ — SAT

$\omega_3 = ( \cancel{\neg x2}^{0} + \cancel{\neg x3}^{0} + x4 )$ — UNIT

$\omega_4 = ( \neg x4 + x5 + \cancel{x10}^{0} )$

$\omega_5 = ( \neg x4 + x6 + \cancel{x11}^{0} )$

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

$\omega_7 = ( \cancel{x1}^{1} + x7 + \cancel{\neg x12}^{0} )$ — SAT

$\omega_8 = ( \cancel{x1}^{1} + x8 )$ — SAT

$\omega_9 = ( \neg x7 + \neg x8 + \cancel{\neg x13}^{0} )$

- **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**

# BCP Iterative – Go Until No More Implications

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

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

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

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

SAT

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

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

UNIT

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

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

$\omega_8 = ( x1 + x8)$

SAT

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

- **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**

# BCP Iterative – Go Until No More Implications

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

$$\omega_1 = (\overset{0}{\cancel{\neg x1}} + \overset{1}{\cancel{x2}})$$
$$\omega_2 = (\overset{0}{\cancel{\neg x1}} + \overset{1}{\cancel{x3}} + \overset{0}{\cancel{x9}})$$
$$\omega_3 = (\overset{0}{\cancel{\neg x2}} + \overset{0}{\cancel{\neg x3}} + \overset{1}{\cancel{x4}})$$

**SAT**

$$\omega_4 = (\overset{0}{\cancel{\neg x4}} + \overset{1}{\cancel{x5}} + \overset{0}{\cancel{x10}})$$
$$\omega_5 = (\overset{0}{\cancel{\neg x4}} + \overset{1}{\cancel{x6}} + \overset{0}{\cancel{x11}})$$

**UNIT**

$$\omega_6 = (\overset{0}{\cancel{\neg x5}} + \overset{0}{\cancel{\neg x6}})$$

**CONFLICT!**

$$\omega_7 = (\overset{1}{\cancel{x1}} + x7 + \overset{0}{\cancel{\neg x12}})$$
$$\omega_8 = (\overset{1}{\cancel{x1}} + x8)$$

**SAT**

$$\omega_9 = (\neg x7 + \neg x8 + \overset{0}{\cancel{\neg x13}})$$

- **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!**

# BCP Iterative – Go Until No More Implications

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

$\omega_1 = (\neg x1 + x2)$
$\omega_2 = (\neg x1 + x3 + x9)$
$\omega_3 = (\neg x2 + \neg x3 + x4)$

**SAT**

$\omega_4 = (\neg x4 + x5 + x10)$
$\omega_5 = (\neg x4 + x6 + x11)$

**SAT**

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

**CONFLICT!**

$\omega_7 = (x1 + x7 + \neg x12)$
$\omega_8 = (x1 + x8)$

**SAT**

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

**UNRESOLVED**

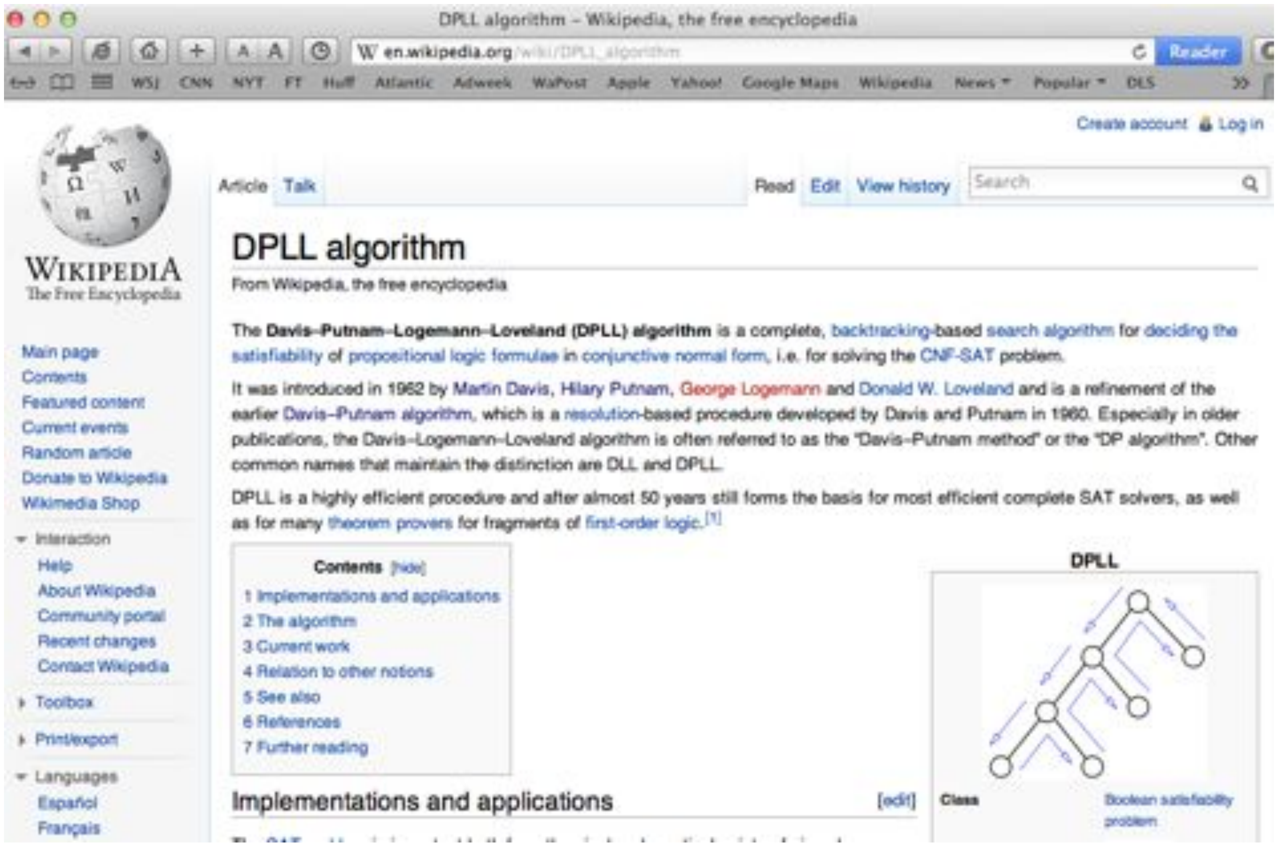
- **3 cases when BCP finishes**
  - **SAT:** Find a SAT assignment, all clauses resolve to "**1**". Return it.
  - **UNRESOLVED:** One or more clauses unresolved. Pick another unassigned var, and recurse more.
  - **UNSAT:** Like this. Found **conflict,** one or more clauses eval to "**0**"
- **Now what?**
  - 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

## 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]

# 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…


Boolean satisfiability problem – Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/Boolean_satisfiability_problem

Create account Log in

Article Talk Read Edit View history

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

## 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* AND *b* is satisfiable because one can find the values *a* = TRUE and *b* = TRUE, which make (*a* AND *b*) = 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.

# VLSI CAD:
## Logic to Layout

# Lecture 4.3

## Computational Boolean Algebra Representations: Using SAT for Logic

**Rob A. Rutenbar**
University of Illinois

# 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)**

$\neq$

- **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)**

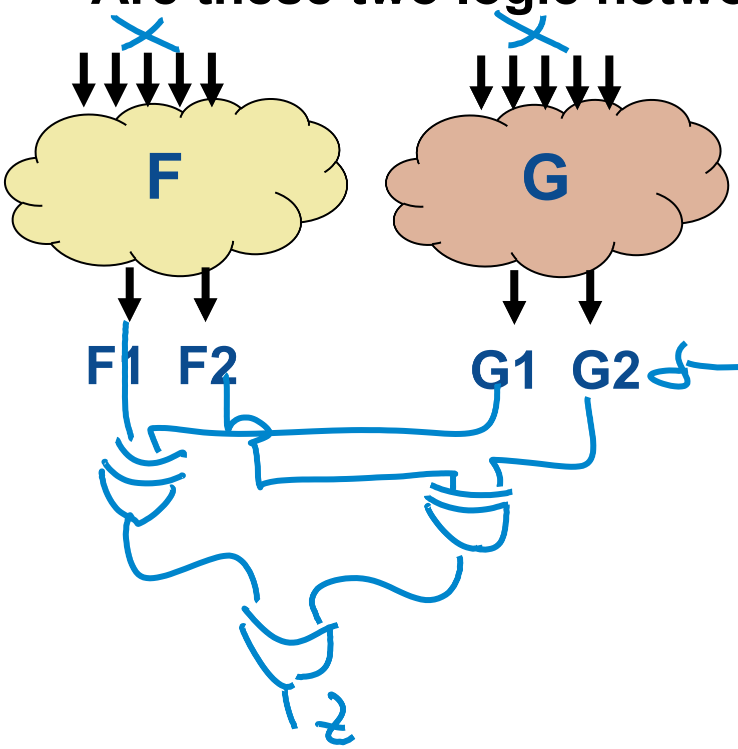
# Typical Practical SAT Problem

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

$z = 1$ SAT iff $F \neq G$

if $z$ SAT: find $x$ such that

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



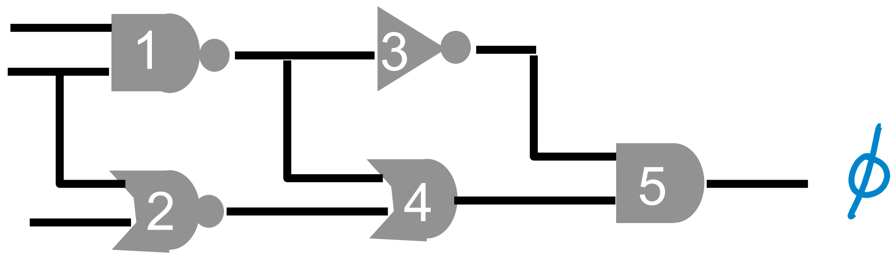

**Do SAT solve on this new network**

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

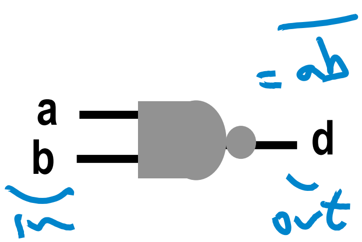
→ If **unSAT**: yes, **same**!

# 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)**

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

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

# 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**

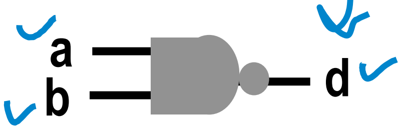

$Y = \bar{A}B + A\bar{B}$

- **EXNOR: Exclusive NOR**
  - Write is as: $Y = A \overline{\oplus} B$ **( 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**


$Y = AB + \bar{A}\bar{B}$

# Final Topic: Gates → CNF

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



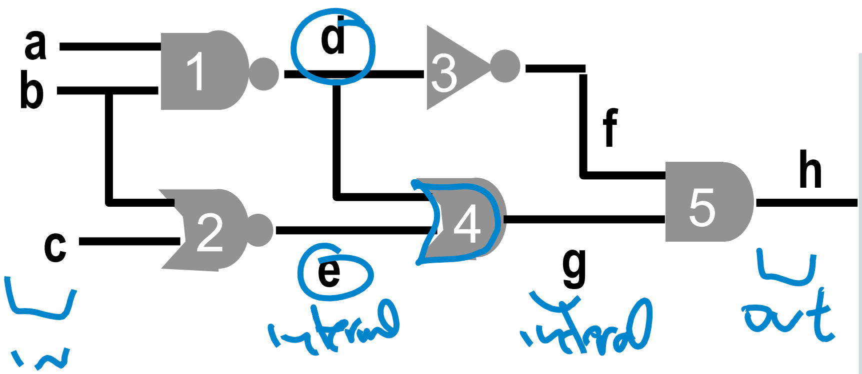

$$\boxed{\phi_d = (a + d)(b + d)(\neg a + \neg b + \neg d)}$$

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

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

# Final Topic: Gates → CNF

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





$\phi_g = g \overline{\oplus} [d+e]$

$\phi_d = \ldots$ do it!

$\phi_e = \ldots$

$\phi_f = \ldots$

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

$\phi_h = \ldots$

# 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**

**$\varphi$ = (output var) $\bigwedge$ (k is gate output wire) $\Phi_k$**

**$\varphi$ = h**

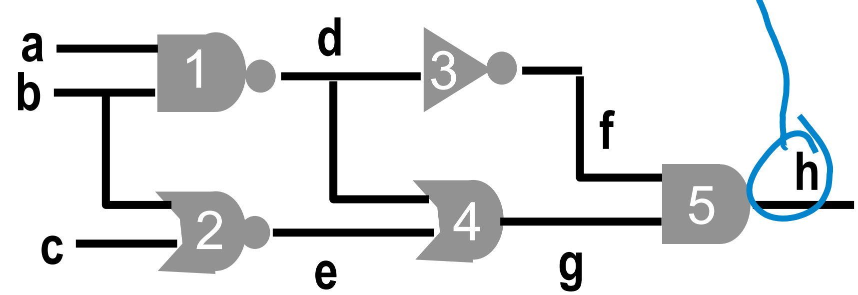
**( a + d ) ( b + d ) ( ¬a + ¬b + ¬d )** 1

**( ¬b + ¬e ) ( ¬c + ¬e ) ( b + c + e )** 2

**( ¬d + ¬f ) ( d + f )** 3

**( ¬d + g ) ( ¬e + g ) ( d + e + ¬g )** 4

**( f + ¬h ) ( g + ¬h ) ( ¬f + ¬g + h )** 5




- 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)

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

**z=NOR(x1, x2, … xn)**

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

**z=OR(x1, x2, … xn)**

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

**z=NOT(x)**

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

**z=NAND(x1, x2, … xn)**

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

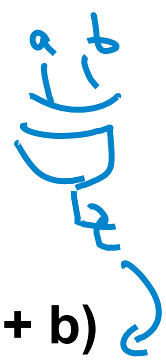
**Z=AND(x1, x2, … xn)**

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

# 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=EXOR(a, b)**




$$\phi_z = z \overline{\oplus} (a \oplus b)$$

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

**z=EXNOR(a, b)**

**Do it –**
**good practice!**

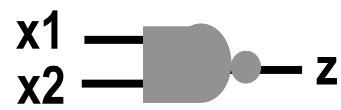
# Using the Rules…

**z=NAND(x1, x2, … xn)**

$$\left[\prod_{i=1}^{n=2}(xi+z)\right]\left[\left(\sum_{i=1}^{n=2}\overline{xi}\right)+\overline{z}\right]$$

product

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




$$\phi_z = (x1 + z)(x2 + z)(\neg x1 + \neg x2 + \neg z)$$

**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