Project Help Slides

系統晶片驗證
SoC Verification

Sep, 2004

To alleviate your pressure in taking this class,

and

To encourage practical experience after learning various verification techniques…

Grading Policy Changed

- Homework 25%
- Final project 75%
  - Initial proposal 10% (of 75%)
  - Project planning 15% (of 75%)
  - Final report 75% (of 75%)
- Bonus and/or Quiz TBD

The final grade will be linearly adjusted.

Instructor will determine the average and standard deviation

Important Dates for Final Project

- Oct 8
  - Detailed rules announcement
  - Project topics announcement
- Oct 25
  - Project group formation
  - Initial proposal due
- Nov 15
  - Project planning due
- Jan 7 (and/or Jan 14)
  - Oral report
- Jan 14
  - Final written report due

Project Topics

1. Verification tools implementation
   - SAT (zChaff) engine
   - BDD engine
2. Design verification practices
   - Apply new techniques you learn in this class to your design(s)
3. Field studies
   - Thorough survey on specific verification techniques, tools, markets, companies, etc.
4. Self-defined topic
   - With the approval of instructor

Verification tools implementation is highly encouraged

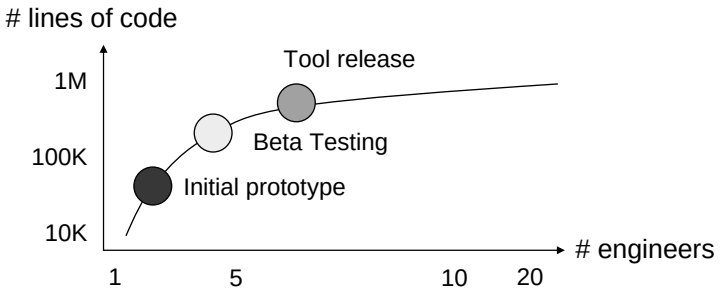
However, you may need the following background…

EDA Tool Implementation

- Mission impossible?
- How many lines of codes (C/C++) can you handle? (100? 1000? 10000? Or?)
- How/where to start?
- What are required know-how's?
- Why should I learn it?

EDA Tool Implementation --- Mission Impossible?

- A typical EDA tool
  - 100K ~ 1M+ lines of code
- Initial prototype (proof of concept)
  - 10K ~ 100K lines of code

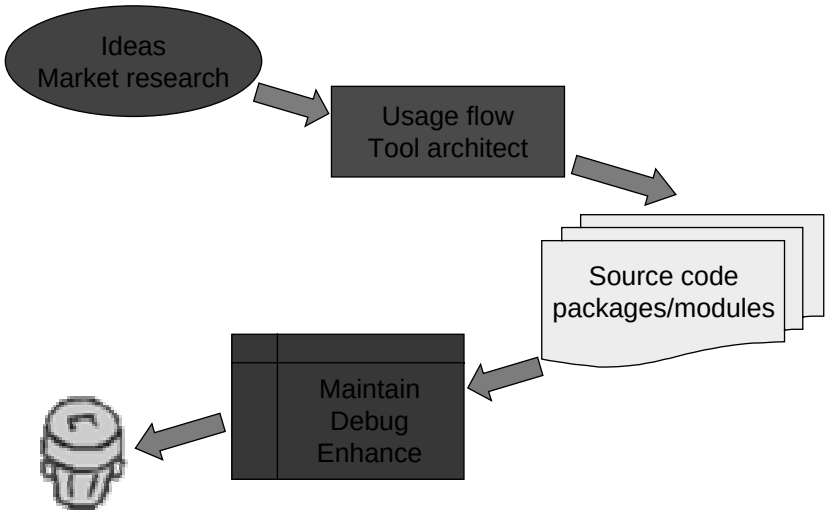


"My school project is about
1000 lines of code,
and it drives me nuts already…"

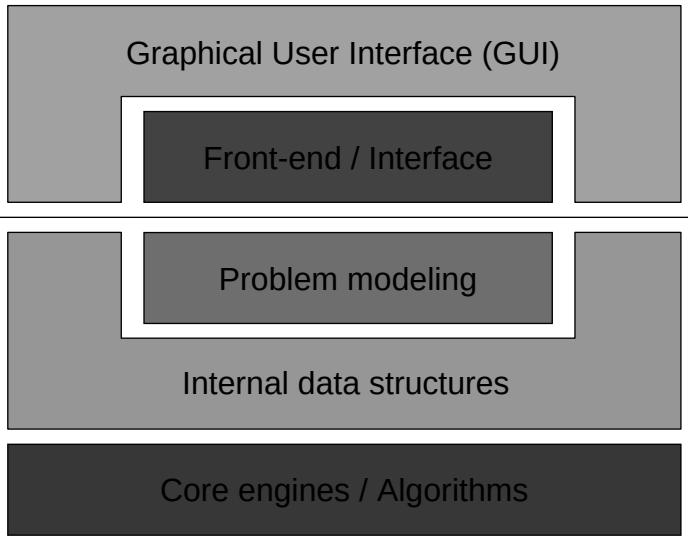
Practice, practice, practice…

- 1K → 10K
- Data structure
  - Dynamic array, hash, set/map, etc
  - Classes, enum, etc
- Algorithms
  - Non-greedy algorithms
  - Heuristic / cost functions
  - Complexity analysis
- Software engineering

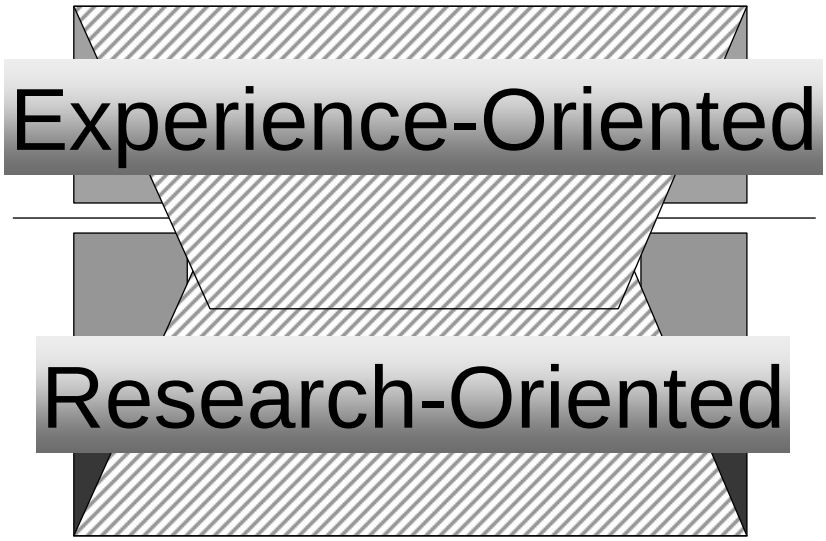
Life of an EDA tool



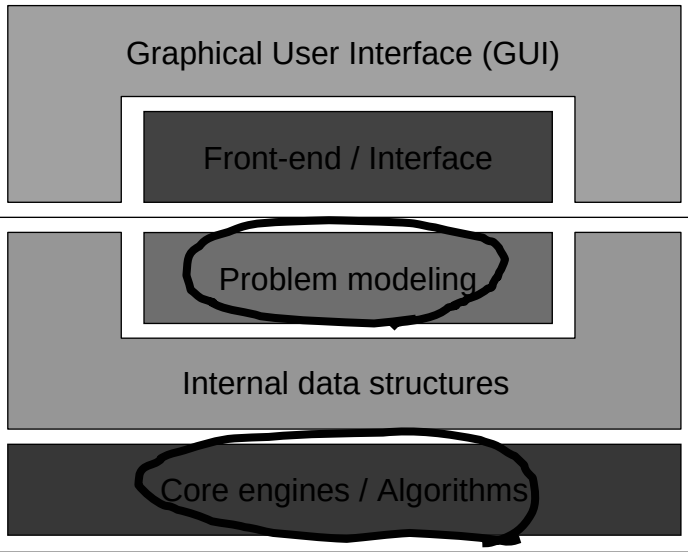
Typical EDA Tool Architecture



Typical EDA Tool Architecture



Typical EDA Research Project



Core Engines

- Boolean Satisfiability (SAT)
- Binary Decision Diagram (BDD)

SAT --- a Special Constraint Satisfiability Problem (CSP)

- Conventional CSP applications
  - AI, planning, automated deduction
- Constraints in CSP
  - Linear (+/-, constant), non-linear
  - Equality, non-equality
  - Integer, real number, imaginary number
- CSP solutions
  - Satisfiability
  - Unsatisfiability
  - Inconclusive

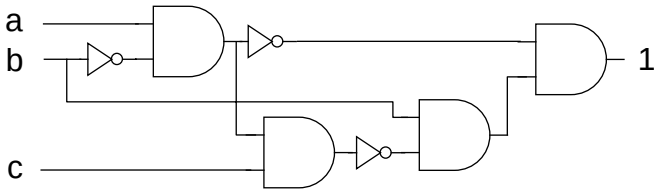
CSP in EDA (VLSI) Applications

- Variables are digitalized (0/1)
  - ➔ Boolean Satisfiability
- Operators can be reduced to { not, and }
  - (a || b) = ! (!a && !b) … DeMorgan
  - (a → b) = (!a || b) = !(a && !b)
  - (a > b) = (a && !b)
  - f(+, -, *, /) = g(not, and)
  - Integer → Binary number system
- Applications: verification, optimization, simulation

Boolean Constraints

- Variables: Boolean (0/1)
- Operators: { not, and }
- Formula: multi-level logic
  - e.g.

    (!(a && !b))&&(b && !((a && !b) && c)) = 1

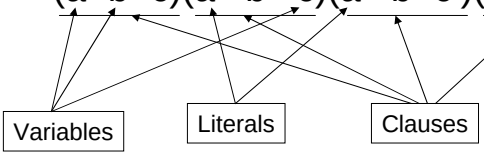


Given a Boolean Constraint,
How to solve it?

Converting Boolean Constraint to Conjunctive Normal Form (CNF)

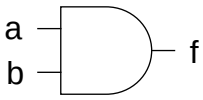
- A Product of Sum (PoS) format
  - e.g.

(a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')

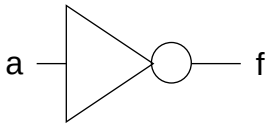


- For the constraints to be satisfied, all the clauses should = 1

Converting to CNF



- Implications
  - !a → !f
  - !b → !f
  - (a && b) → f
- Clauses
  - (a + !f)
  - (b + !f)
  - (!a + !b + f)



- Implications
  - !a → f
  - a → !f
- Clauses
  - (a + f)
  - (!a + !f)

In other words…

- We can convert all the constraints in a circuit
  - ➔ Boolean constraints
  - ➔ CNF

How to solve it?

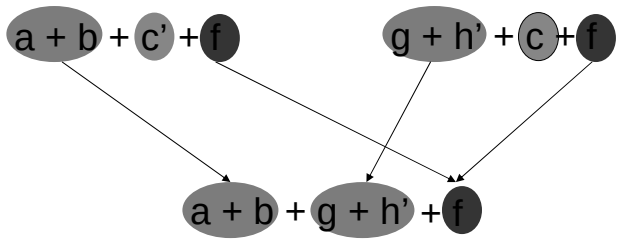
Boolean Satisfiability (SAT) Algorithm

- NP-Complete problem (Cook, 1971)
  - But can often be solved very efficiently
  - ~10k Variables, ~100k Clauses

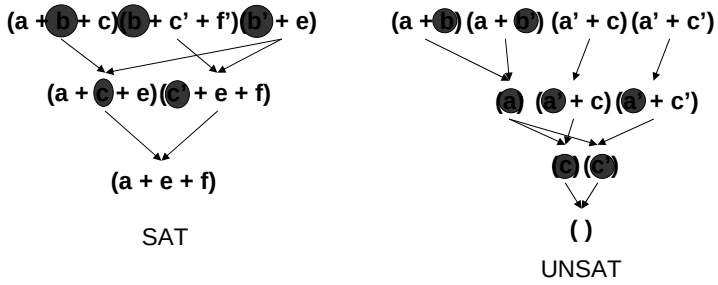
1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, Logemann, Loveland, 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. Stålmarcks algorithm, 1980s
   - Proprietary algorithm. Patented.
   - Commercial versions available
4. Stochastic Methods, 1992
   - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
   - Local search and hill climbing

Resolution

- Resolution of a pair of clauses with exactly ONE incompatible variable
  - Two clauses are said to have distance 1



Davis Putnam Algorithm



Potential memory explosion problem!

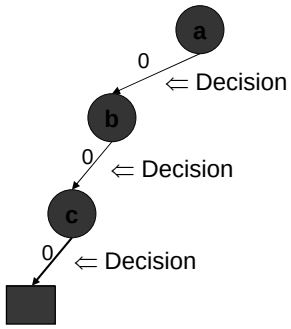
Boolean Satisfiability (SAT) Algorithm

- NP-Complete problem (Cook, 1971)
  - But can often be solved very efficiently
  - ~10k Variables, ~100k Clauses

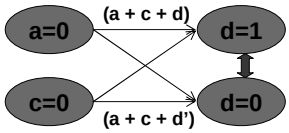
1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, Logemann, Loveland, 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. Stålmarcks algorithm, 1980s
   - Proprietary algorithm. Patented.
   - Commercial versions available
4. Stochastic Methods, 1992
   - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
   - Local search and hill climbing

Basic DLL Procedure - DFS

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



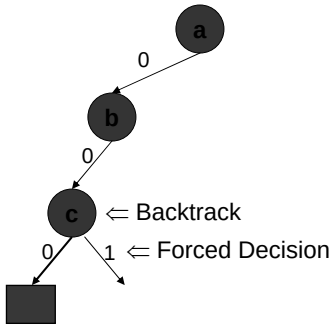
Implication Graph



Conflict!

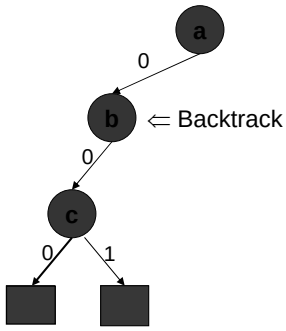
Basic DLL Procedure - DFS

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)

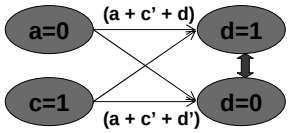


Basic DLL Procedure - DFS

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



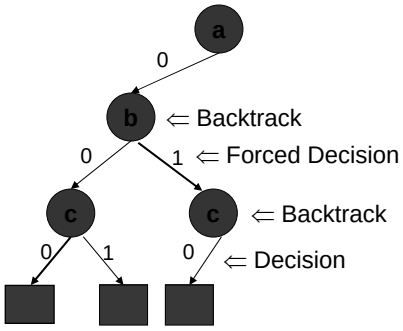
Implication Graph



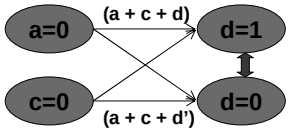
Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



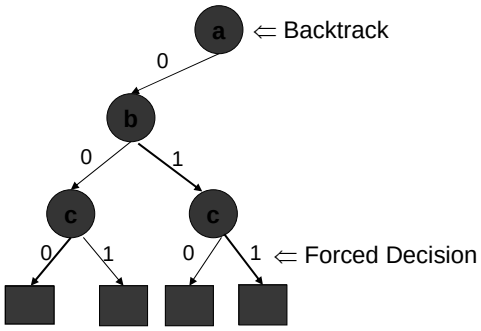
Implication Graph



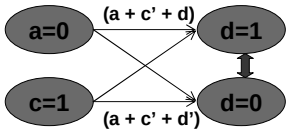
Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



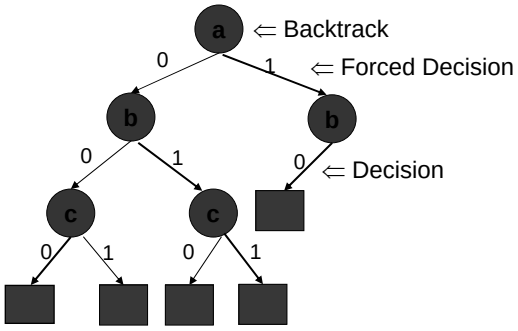
Implication Graph



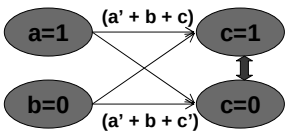
Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Implication Graph



Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

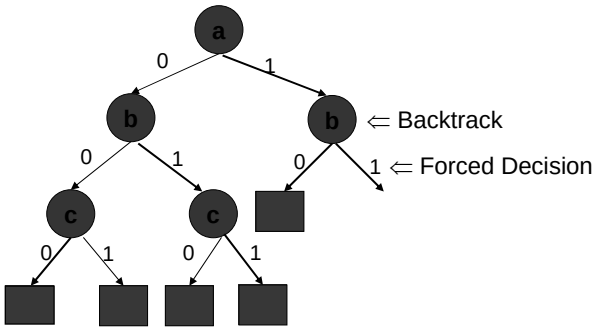
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

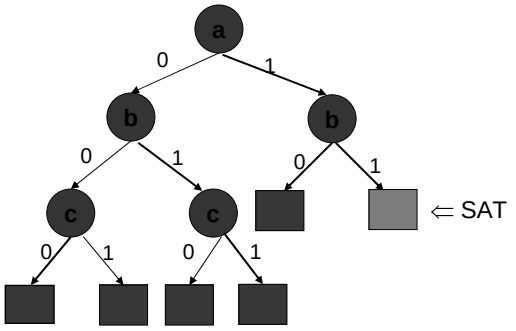
(a + c’ + d)

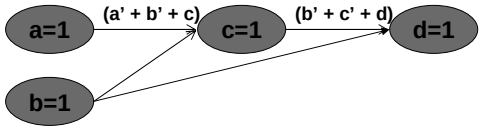
(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)





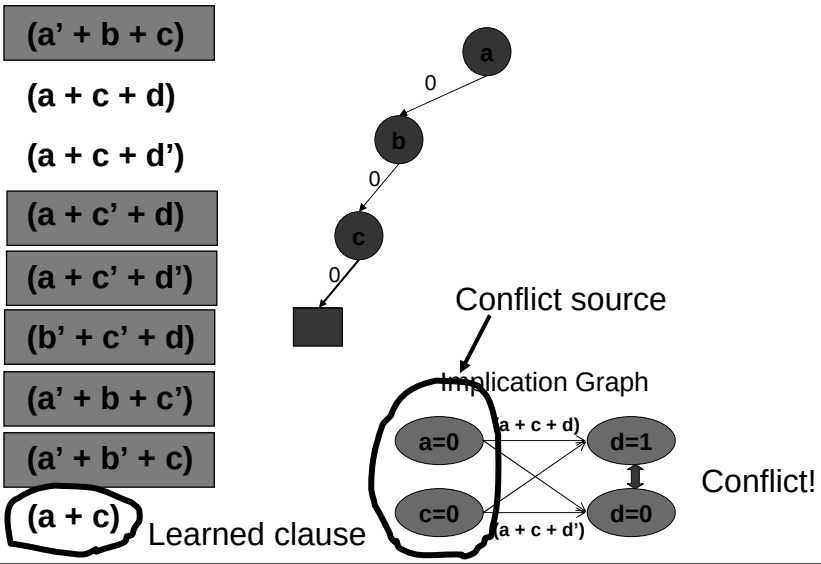
Potentially exponential complexity!!

Did you see any unnecessary work?

SAT Improvements

1. Conflict-driven learning
   - Once we encounter a conflict
     ➔ Figure out the cause(s) of this conflict and prevent to see this conflict again!!

Conflict-Driven Learning



SAT Improvements

2. Non-chronological backtracking
   - Since we get a learned clause from the conflict analysis…

     ➔ Instead of backtracking 1 decision at a time, backtrack to the "next-to-the-last" variable in the learned clause

Non-Chronological Backtracking

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

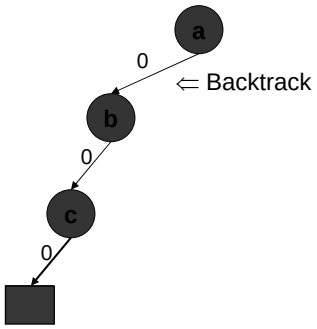
(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c) Learned clause



- ‘a’ is the next-to-the-last variable in the learned clause
- Backtrack c = 0 && b = 0

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

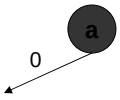
(a + c’ + d’)

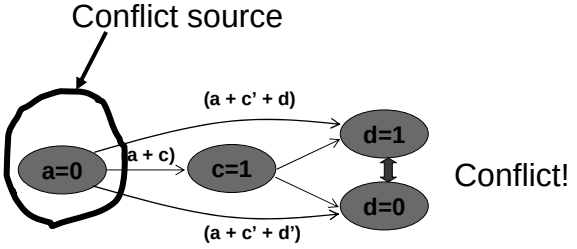
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c)





Conflict!

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c)

(a) Learned clause

- Since there is only one variable in the learned clause

  → No one is the next-to-the-last variable

- Backtrack all decisions

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

(a + c’ + d’)

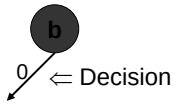
(b’ + c’ + d)

(a’ + b + c’)

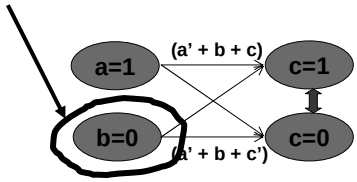
(a’ + b’ + c)

(a + c)

(a)

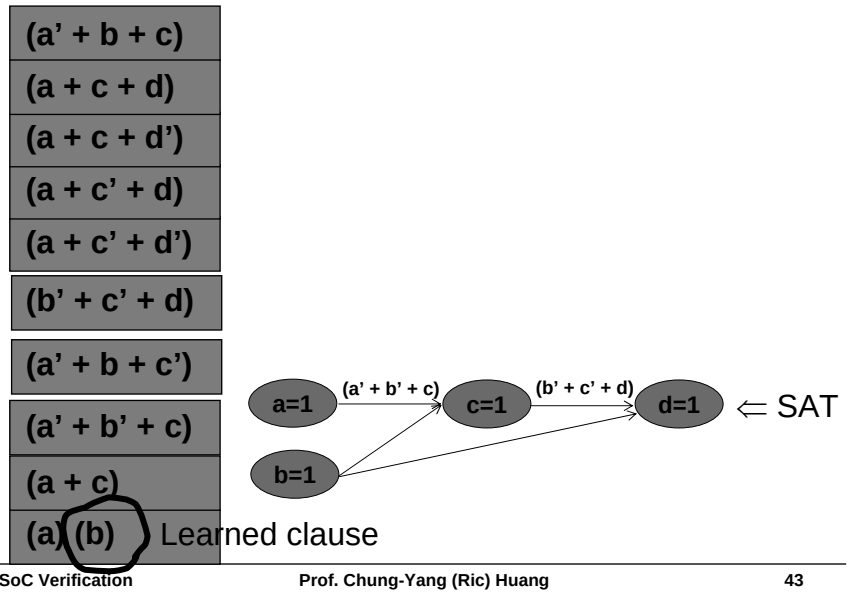


Conflict source



Conflict!

Deduced Implication from Learned Clause



zChaff SAT Engine

- A very efficient C++-based SAT engine developed in Princeton University
  - Good engineering work
- Source codes free to download
- Can be used as a stand-alone SAT solver, or be compiled as a library and linked with other applications

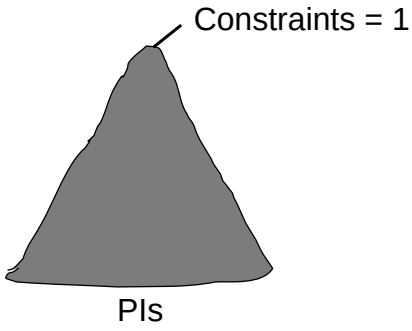
Efficiency of SAT Engine

- The memory usage of SAT engine is almost proportional to the number of (implications + learned clauses)
  - ➔ Memory usage is relatively small
  - ➔ Can apply to large circuit
- However, since it is making one decision at a time, its runtime is almost proportional to the number of decisions it makes
  - ➔ Could be inefficient if making bad decisions

Any alternative??

If the constraints are represented as a logic circuit…

➔ Represent the logic function using PIs…



Function Representation

- In general…
  - f = (a && b || ((c + d) > e)) ^ !g && (a > b)?...
  - Not canonical

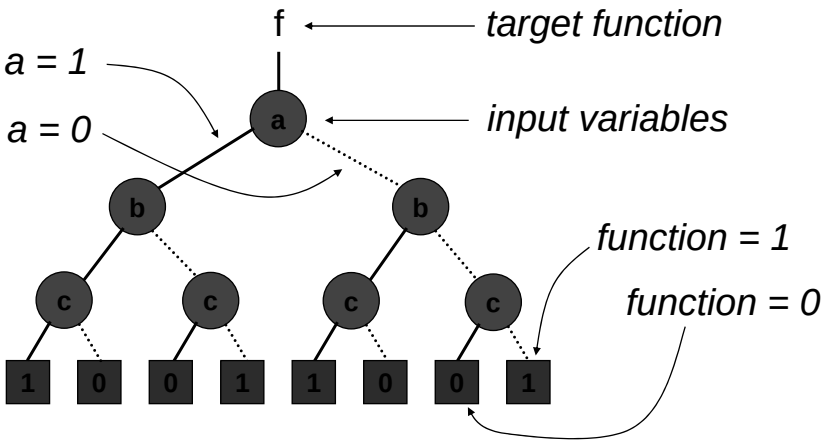
- Enumeration (Truth table)
  - e.g.

a1	a0	b1	b0	f
0	0	0	0	0
0	0	0	1	1
.....				
1	1	1	0	1
1	1	1	1	0

  - Exponential growth in size (like simulation)
  - But, once we have the table, finding an assignment is easy
  - Canonical

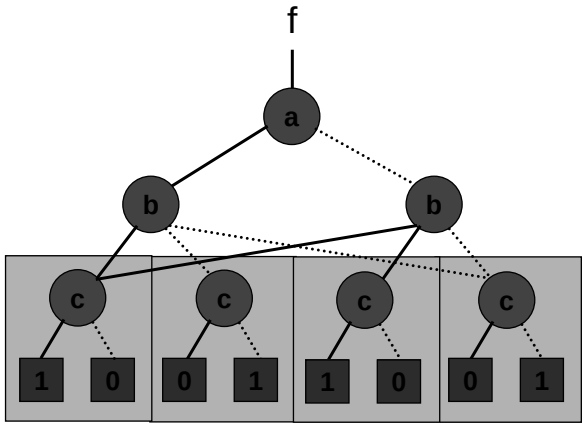
A better data structure to represent truth table?

Binary Decision Tree

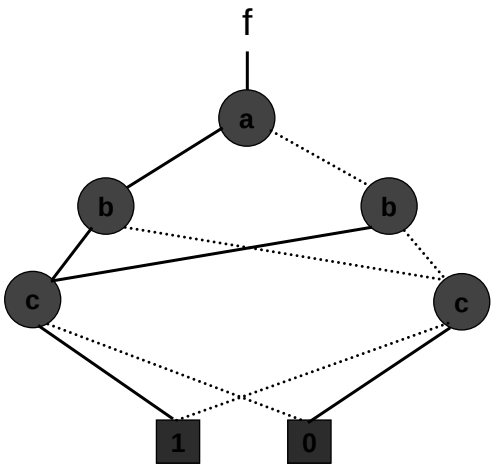


◆Still exponential in size

Binary Decision Diagram



Reduced Binary Decision Diagram

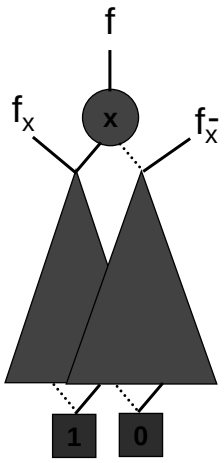


Binary Decision Diagram (BDD)

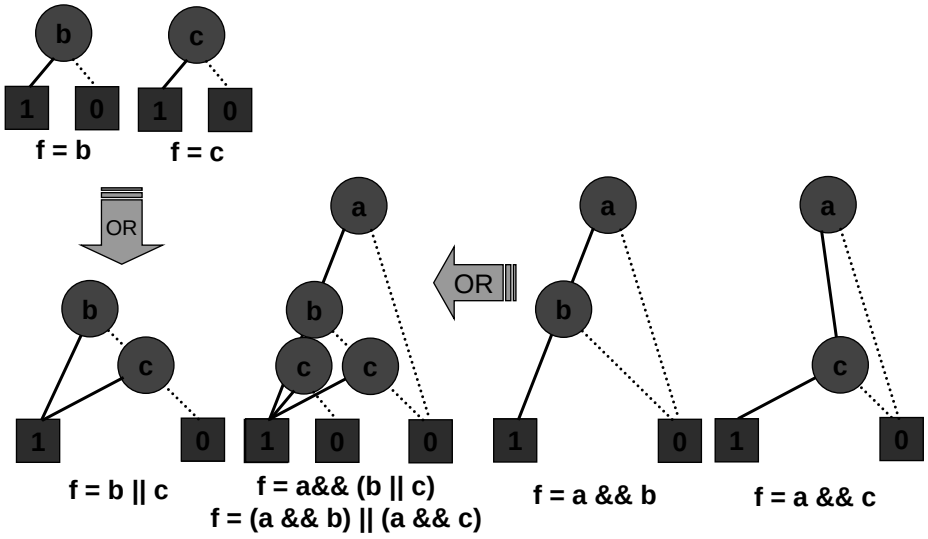
- A graphical representation of truth table
  - f = func(a, b, c, d, …) is a logic function
  - Each level corresponds to an input variable
    → Set of inputs is called “support”
  - Functions with identical functions are merged together
    - Always canonical
  - Each node (and its sub-graph) represents a function
  - Each path represents a cube of the function

Basic BDD Operations

- Shannon expansion of f
  - $f = x * f_x + \overline{x} * f_{\overline{x}}$
- $f * g$
  $= (x * f_x + \overline{x} * f_{\overline{x}}) *$
  $(x * g_x + \overline{x} * g_{\overline{x}})$
  $= x * (f_x * g_x) + \overline{x} * (f_{\overline{x}} * g_{\overline{x}})$
- $f + g$
  $= x * (f_x + g_x) + \overline{x} * (f_{\overline{x}} + g_{\overline{x}})$
- Operation: perform on cofactors individually



Basic BDD Operations



In short,

1. f = a && (b || c)
2. f = (a && b) || (a && c)

➔ will result in same BDD

➔ independent of building orders

Therefore, to build BDDs for a circuit…

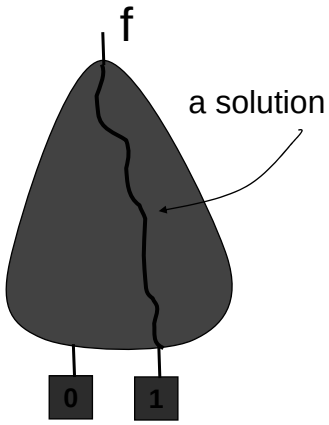
1. Order circuit topologically from PIs to target gate
2. Build BDDs from PIs to target

To solve a satifiability problem by BDDs

➔ Build BDDs for for the corresponding circuit f

Null BDD; f is always = 0

➔ Constraints are UNSAT



Sounds too good to be true…

Any problem ??

BDD Complexity

- In general, the size of BDD nodes is still exponential to the size of input supports
  - Usually can only build BDDs for circuit with #input = 100 ~ 200

The problem?

- BDDs find all the assignments at once
  - → while we only need one…
  - → many many optimization techniques…

CUDD Package

- Implemented in University of Colorado at Boulder
- The most widely used BDD package in various researches and EDA tools