Lecture 8: SAT
Logic Design Automation for VLSI 046880

- Reminder of last lectures: BDDs - Binary Decision Diagrams
  - Provide a canonical structure that represents function and sets
  - Used for
    - Verification: “Combinatorial” or “Sequential” equivalence checking
    - Synthesis: represent sets of “don’t cares”
    - Test: false path identification
  - But may “explode” memory on some structures

- Today’s lesson: the modern alternative
  - SAT problem
  - Solvers
  - And applications

Sources:
Prof. Karem A. Sakallah EECS Department University of Michigan
Prof. Ofer Strichman, IE Department Technion
Prof. Javier Larrosa, Universitat Polit’ecnica de Catalunya, Spain
Boolean Satisfiability

- Given a suitable representation for a Boolean function \( f(X) \):
  - Find an assignment \( X^* \) such that \( f(X^*) = 1 \)
  - Or prove that such an assignment does not exist
    (i.e. \( f(X) = 0 \) for all possible assignments)

- In the “classical” SAT problem, \( f(X) \) is represented as product-of-sums (POS) a.k.a. “conjunctive normal form” (CNF)
  - example: \((x_1)(x_1+x_2)(x_2+x_3)(x_3+x_4)(x_5)=1\)

- The classical example: Embassy Ball.
  - “King wants to invite Peru or exclude Qatar
    Queen wants to invite Qatar or Romania
    Prince wants to exclude Romania or Peru
    Is there a guest-list that will satisfy the whims of the entire royal family?”
  - Represented as CNF: \((P \mid \sim Q) \& (Q \mid R) \& (\sim R \mid \sim P)\)
  - Is satisfied by
    P=true, Q=true, R=false
    P=false, Q=false, R=true
Satisfiability: Mother of all NP-complete problems

- Proven to be NP-complete (by Cook’s theorem)
- Other problems are shown to be intractable by transforming satisfiability into them
- Note that the tautology problem is the complement of satisfiability: Find if the function is TRUE for all possible assignments of variables
  - Universality vs. existence of a truth assignment
How to formulate Logic Equivalence Checking as a SAT problem?

- BDD-based approach
  
  \[ \text{XOR}(f_1, f_2) \equiv 0 \]
  
  Reduce the XOR to 0

- SAT-based approach
  
  Disprove \( \text{XOR}(f_1, f_2) = 1 \)
  
  XOR is 0 for all assignments
Petrick’s method for solving satisfiability

- Write the equation as a Product Of Sums.
  e.g. \((p_1)(p_1+p_2)(p_2+p_3)(p_3+p_4)(p_5)=1\)

- Carry out the products.....
  Using algebra to get a Sum Of Products,
  e.g. \(p_1 \ p_2 \ p_4 + p_1 \ p_3 \ p_5 + \ldots = 1\)

- Now each product term is a solution!

- Looks “innocent”, but opening all the parentheses involves an exponential number of operations!
  - So this method is practical just for small problems
Conjunctive Normal Form (CNF) Definitions

- Clause
- Positive Literal
- Negative Literal

\[ \varphi = (a + c) (b + c) (\neg a + \neg b + \neg c) \]
Basics

- **Implication**
  
  \[ x \rightarrow y = \neg x + y \]
  
  \[ = \neg(\neg y) + (\neg x) \]
  
  \[ = \neg y \rightarrow \neg x \text{ (contra positive)} \]

- **Assignments**: e.g. \( \{a = 0, b = 1\} = \neg a \, b \)
  
  - Partial (some variables still unassigned)
  - Complete (all variables assigned)
  - Conflicting (imply \( \neg \varphi \))

  \[ \varphi = (a + c)(b + c)(\neg a + \neg b + \neg c) \]
  
  \[ \varphi \rightarrow (a + c) \]
  
  \[ \neg(a + c) \rightarrow \neg \varphi \]
  
  \[ \neg a \, \neg c \rightarrow \neg \varphi \]
Literal & Clause Classification

- Variables can be assigned 0 or 1 or unassigned yet

\[ \varphi = (a + \neg b)(\neg a + b + \neg c)(a + c + d)(\neg a + \neg b + \neg c) \]

- \( a \) assigned 0
- \( b \) assigned 1
- \( c \) and \( d \) unassigned
Definitions Summary

- **Variable**
  - Positive / Negative
  - Unassigned

- **Assignment**
  - Full / Partial
  - Conflicting

- **Clause**
  - Satisfied / Violated
  - Unresolved
SAT Algorithms Overview

- Incomplete Algorithms (can only prove satisfiability)
  - GSAT (hill climbing …)
  - Genetic algorithms
  - Simulated annealing
  - …

- Complete Algorithms
  - Backtrack search / DPLL
    - Conflict-Driven clause learning (CDCL)
  - Proof systems
    - Resolution
    - Stalmarck’s method
    - Recursive learning
  - BDDs
Incomplete - GSAT

- *Usually* it cannot prove unsatisfiability
  - But it is very effective in many situations
- A generic algorithm:
  
  Given a CNF formula $\phi$, choose max_tries and max_flips

  for $i = 1$ to max_tries {
    $T :=$ randomly generated truth assignment
    for $j = 1$ to max_flips {
      if $T$ satisfies $\phi$ return TRUE
      randomly choose $v$ from those $v$ that
        flipping $v$’s value that is best according to some *progressing heuristic*
        e.g. largest increase in the # of satisfied clauses
      $T := T$ with $v$’s assignment flipped
    }
  }

  Many alternative progress heuristics
Many Progressing Heuristics

- Hill-climbing
- Tabu-list
- Simulated-annealing
- Random-Walk
- Min-conflicts
E.g. Clause Weights

- **Basic:**
  - Initial weight of each clause: 1
  - Increase by $k$ the weight of unsatisfied clauses
  - Choose $v$ according to max increase in weight

- **Improvement:**
  - Can we reuse information gathered in previous tries in order to speed up the search?
  - Yes!
    - Rather than choosing $T$ randomly each time, repeat ‘good assignments’ and choose randomly the rest
Summary Optimization Approach

- There are many more ideas for heuristics
- But we want to KNOW for **sure** if there exists an assignment that satisfies F
Basic Rules: Pure Literal Rule

- A variable is **pure** if its literals are either all positive or all negative

- Pure literal rule:
  
  *Clauses containing pure literals can be removed from the formula*

- Why?
  - Just assign pure literals to the values that satisfy the clauses

- E.g.
  \[ \phi = (a + c)(b + c)(b + \neg d)(\neg a + \neg b + d) \]

  Set \( c \) to 1; if \( \phi \) becomes unsatisfiable, then it is also unsatisfiable when \( c \) is set to 0.

  \[ \phi' = (b + \neg d)(\neg a + \neg b + d) \]
Basic Rules: Unit Propagation

- A unit clause is an unresolved clause that has exactly one unassigned literal.

- Unit clause rule:
  
  A unit clause, unassigned literal must be assigned value 1 for the clause to be satisfied.

- Unit propagation:
  Apply the unit clause rule iteratively.

- Example:
  \[ \varphi = (a + c)(b + c)(\neg a + \neg b + \neg c) \]
  
  \[ a \, b \rightarrow \neg c \]

- But unit propagation can create conflicts too.
Resolution [Davis&Putnam’60]

■ Resolution Rule

*If \( \varphi \) contains clauses \((x + a)(\neg x + b)\)

one can replace them by \((a + b)\)*

■ Resolution can be used as **complete** algorithm for SAT

- Iteratively apply the following steps:
  - Select a variable \( x \)
  - Apply resolution rule between every pair of clauses matching \((x + a), (\neg x + b)\)
  - Remove ALL clauses containing \( x \) or \( \neg x \)
  - Apply pure literal and unit clause rules
- Terminate when the formula contains only pure literals
- Worst case exponential – of course
Resolution Example

\((a \lor \neg b \lor \neg c) \land (\neg a \lor \neg b \lor \neg c) \land (b \lor c) \land (c \lor d) \land (c \lor \neg d) \vdash\)

resolution on \(a:\)
\((\neg b \lor \neg c) \land (b \lor c) \land (c \lor d) \land (c \lor \neg d) \vdash\)

resolution on \(b:\)
\((c \lor \neg c) \land (c \lor d) \land (c \lor \neg d) \vdash\)

remove trivial clause:
\((c \lor d) \land (c \lor \neg d) \vdash\)

resolution on \(d:\)
\((c)\)
Stålmarck’s Method

- Recursive application of branch-merge rule to each variable to identify common assignments

- Branch merge
  - Recursively apply basic rules once assuming positive assignment and once for negative assignment
  - If both assumptions infer same value to a variable its value is necessary

- E.g.

  \[ \varphi = (a \lor b) (\neg a \lor c) (\neg b \lor d) (\neg c \lor d) \]

  \[(a = 0) \rightarrow (b = 1) \rightarrow (d = 1)\]

  \[\text{UP}(a = 0) = \{a = 0, b = 1, d = 1\}\]

  \[(a = 1) \rightarrow (c = 1) \rightarrow (d = 1)\]

  \[\text{UP}(a = 1) = \{a = 1, c = 1, d = 1\}\]

  \[\text{UP}(a = 0) \cap \text{UP}(a = 1) = \{d = 1\}\]
Recursive Learning

- Recursive evaluation of clause satisfiability requirements for identifying common assignments

E.g.

\[ \varphi = (a \lor b)(\neg a \lor c)(\neg b \lor d)(\neg c \lor d) \]

\[(a = 1) \rightarrow (c = 1) \rightarrow (d = 1) \]

\[ \text{UP}(a = 1) = \{a = 1, c = 1, d = 1\} \]

\[(b = 1) \rightarrow (d = 1) \]

\[ \text{UP}(b = 1) = \{b = 1, d = 1\} \]

\[ \text{UP}(a = 1) \cap \text{UP}(b = 1) = \{d = 1\} \]

So \(d = 1\) is necessary assignment
Backtrack Search – DPLL*

- A Generic Iterative Algorithm
- At each step:
  - (1) [DECIDE] Select decision assignment
  - (2) [DEDUCE] Apply unit propagation and the pure literal rule
  - (3) [DIAGNOSIS] If conflict identified, then backtrack
    - If cannot backtrack further, return UNSAT
    - Otherwise, proceed with unit propagation (2)
  - (4) If formula satisfied, return SAT
  - (5) Otherwise, proceed with another decision (1)

*DPLL: Davis–Putnam–Logemann–Loveland
DPLL Example

\[ \phi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land \\
(\neg b \lor \neg d \lor \neg e) \land \\
(a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land \\
(a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]
DPLL Example

\[ \phi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land (\neg b \lor \neg d \lor \neg e) \land (a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land (a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]

d and e are implied but conflict!
DPLL Example

\[ \phi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land \\
(\neg b \lor \neg d \lor \neg e) \land \\
(a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land \\
(a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]

d and \( \neg d \) are implied but conflict!
DPLL Example

$$\phi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land (\neg b \lor \neg d \lor \neg e) \land (a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land (a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e)$$

e and ¬e are implied but conflict!
DPLL Example

\[ \phi = (a \lor \neg b \lor d) \land (a \lor \neg b \lor e) \land 
\neg b \lor \neg d \lor \neg e \land 
(a \lor b \lor c \lor d) \land (a \lor b \lor c \lor \neg d) \land 
(a \lor b \lor \neg c \lor e) \land (a \lor b \lor \neg c \lor \neg e) \]

All clauses are met! \( \phi \) is SAT
Conflict Driven Learning

SAT Solvers

- Introduced in the 90’s
  - [Marques-Silva&Sakallah’96][Bayardo&Schrag’97]
- Inspired by DPLL
  - Must be able to prove both satisfiability and unsatisfiability
- New clauses are learnt from conflicts
- Structure of conflicts exploited (UIPs)
- Backtracking can be non-chronological
- Efficient data structures [Moskewicz&al’01]
  - Compact and reduced maintenance overhead
- Backtrack search is periodically restarted [Gomes&al’98]

Can solve instances with hundreds of thousand variables and tens of million clauses
Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

\[ \phi = (a \lor b) \land (\neg b \lor c \lor d) \land (\neg b \lor e) \land (\neg d \lor \neg e \lor f) \ldots \]

- Assume decisions \( c = 0 \) and \( f = 0 \)
- Assign \( a = 0 \) and imply assignments (\( b \), \( d \), \( e \))
- A conflict is reached: \( (\neg d \lor \neg e \lor f) \) is unsatisfied
- \( (a = 0) \land (c = 0) \land (f = 0) \Rightarrow (\phi = 0) \)
- \( (\phi = 1) \Rightarrow (a = 1) \lor (c = 1) \lor (f = 1) \)
- Learn new clause \( (a \lor c \lor f) \)
More Techniques

- Backtracking
  - Most Recent Assignment Backtracking
  - Non-Chronologic Assignment Backtracking
  - 2nd highest decision clause

- Implication Graph base
  - Unique Implication Points

- Many more ideas keep advancing SAT performance and capacity as we speak…
Converting Circuit to CNF

Tseitin’s encoding

- $\phi = x_1 \lor \neg(x_2 \land x_3)$

- $\phi' = a_0 \land (a_0 \leftrightarrow x_1 \lor a_1) \land (a_1 \leftrightarrow \neg a_2) \land (a_2 \leftrightarrow x_2 \land x_3)$

- It is left to transform $\phi'$ to CNF.
Tseitin’s encoding: CNF encodings of gates

- **And gate:**
  - e.g. for \( a_i \leftrightarrow x_1 \land x_2 \) add to \( S \)
    \[
    (a_i \lor \neg x_1 \lor \neg x_2), \\
    (\neg a_i \lor x_1), \\
    (\neg a_i \lor x_2)
    \]

- **Or gate:**
  - e.g. for \( a_i \leftrightarrow x_1 \lor x_2 \) add to \( S \)
    \[
    (\neg a_i \lor x_1 \lor x_2), \\
    (a_i \lor \neg x_1), \\
    (a_i \lor \neg x_2)
    \]

- **Not gate:**
  - e.g. for \( a_i \leftrightarrow \neg x_1 \) add to \( S \)
    \[
    (\neg a_i \lor \neg x_1), \\
    (a_i \lor x_1)
    \]

- **Why is that true?**
Tseitin’s Encoding

- For each Boolean gate instance $g_i$ in $\phi$, add a new auxiliary variable $a_i$, and add to a stack $S$ the CNF clauses encoding $a_i \leftrightarrow g_i$.

- Let $a_0$ denote the auxiliary variable encoding the main operator of $\phi$.

- Let

  $$\phi' = \bigwedge_{cl \in S} cl \land a_0$$

- Theorem (Tseitin): $\phi$ is satisfiable iff $\phi'$ is satisfiable.
SAT Summary

- Satisfiability is the mother of all NP-Complete problems
- In the context of Logic DA, it solves BDDs memory explosion problem
- It is sometimes “easy” to prove Satisfiability by example
  - A search for a satisfying assignment may use many “optimization” heuristics and engines
- But proving Un-Satisfiability may require visiting all possible inputs
- Unless a smart way to deduce some combinations are masked by others

- The art of proving Un-Satisfiability keeps evolving and improving
- In Logic Design Automation it is used for
  - Combinatorial and Sequential Equivalence checking
  - Automatic Pattern Generation for Device Testing
  - Sequential Model checking