ECE 667

Synthesis and Verification of Digital Circuits

Boolean SAT

Slides adapted from students presentation, Girish Paladugu (ECE 667 Spring 2011)
Overview

• Introduction: motivation, brief history
• Conjunctive Normal Form (CNF)
• DPLL Method for solving SAT problems
• Backtrack search algorithm
• Applications
• Conclusion
INTRODUCTION

- **SAT**: Given a Boolean formula (propositional logic formula)
  - find a variable assignment such that the formula evaluates to 1 (SAT), or
  - prove that no such assignment exists (unSAT).
- For $n$ variables, there are $2^n$ possible truth assignments to be checked.

The first established **NP-Complete** problem.

Conjunctive Normal Form

- Conjunctive Normal Form (CNF):

  A conjunctive normal form (CNF) formula $\phi$ on $n$ binary variables $x_1, \ldots, x_n$ is the conjunction of $m$ clauses $\omega_1, \ldots, \omega_m$.

- **Literal**: it is the occurrence of a variable $x$ or its complement $x'$.

- **Clause**: It is the disjunction of one or more literals.

- **Example**
  - CNF formula is satisfiable if each clause is satisfiable (evaluate to true) else it is unsatisfiable.

  $$\phi = (a + c) (b + c) (\neg a + \neg b + \neg c)$$
Applications

• Core computational engine for major applications
  – EDA
    • Testing and Verification
    • Logic synthesis
    • FPGA routing
    • Path delay analysis
    • Test Pattern Generation
  – Artificial Intelligence
    • Knowledge base deduction
    • Automatic theorem proving
Equivalence Checking

If \( z = 1 \) is unsatisfiable, the two circuits are equivalent!
Automatic Test Pattern Generation (ATPG)
Development of SAT: a Brief History

• Main contributions in SAT research:
  – **1960**: Davis and Putnam – resolution based - dealing with ~10 variables
  – **1962**: Davis, Logemann and Loveland – DFS-based – dealing with ~10 variables
    • Basic framework for many modern SAT solvers
  – **1986**: R. E. Bryant – BDD-based – dealing with ~100 variables
  – **1996**: Silva and Sakallah – GRASP (conflict-driven learning and non-chronological backtracking) – dealing with ~1k variables
  – **2001**: Malik *et al.* – Efficient BCP and decision making – dealing with ~10k vars

Adopted from S. Malik, Princeton University
Deriving CNF for Logic Gate

\[ \varphi_d = [d = \neg(a \land b)] \]
\[ = \neg[d \oplus \neg(a \land b)] \]
\[ = \neg[(\neg(a \land b) \neg d + a \land b \land d)] \]
\[ = \neg[(\neg a \neg d + \neg b \neg d + a \land b \land d)] \]
\[ = (a + d)(b + d)(\neg a + \neg b + \neg d) \]

\[ \varphi_d = [d = \neg(a \land b)] [\neg d = a \land b] \]
\[ = [d = \neg a + \neg b] [\neg d = a \land b] \]
\[ = (\neg a \rightarrow d)(\neg b \rightarrow d)(a \land b \rightarrow \neg d) \]
\[ = (a + d)(b + d)(\neg a + \neg b + \neg d) \]

- Show alternate method (*characteristic function*)
- Point to the difference between
  - *satisfying the CNF formula*
  - *solving for the output value*
## CNF Formulas for simple gates

<table>
<thead>
<tr>
<th>Gate type</th>
<th>Gate function</th>
<th>$\varphi_x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>$x = \text{AND}(w_1, \ldots, w_j)$</td>
<td>$\prod_{i=1}^{j} (w_i + \neg x) \cdot \left( \sum_{i=1}^{j} \neg w_i + x \right)$</td>
</tr>
<tr>
<td>NAND</td>
<td>$x = \text{NAND}(w_1, \ldots, w_j)$</td>
<td>$\prod_{i=1}^{j} (w_i + x) \cdot \left( \sum_{i=1}^{j} \neg w_i + \neg x \right)$</td>
</tr>
<tr>
<td>OR</td>
<td>$x = \text{OR}(w_1, \ldots, w_j)$</td>
<td>$\prod_{i=1}^{j} (\neg w_i + x) \cdot \left( \sum_{i=1}^{j} w_i + \neg x \right)$</td>
</tr>
<tr>
<td>NOR</td>
<td>$x = \text{NOR}(w_1, \ldots, w_j)$</td>
<td>$\prod_{i=1}^{j} (\neg w_i + \neg x) \cdot \left( \sum_{i=1}^{j} w_i + x \right)$</td>
</tr>
<tr>
<td>NOT</td>
<td>$x = \text{NOT}(w_1)$</td>
<td>$(x + w_1) \cdot (\neg x + \neg w_1)$</td>
</tr>
<tr>
<td>BUFFER</td>
<td>$x = \text{BUFFER}(w_1)$</td>
<td>$(\neg x + w_1) \cdot (x + \neg w_1)$</td>
</tr>
</tbody>
</table>
Circuit - CNF conversion

\[ \varphi = (x_1 + x_3) \cdot (x_2 + x_3) \cdot (\neg x_1 + \neg x_2 + \neg x_3) \cdot (\neg x_3 + z) \cdot (\neg x_4 + z) \cdot (x_3 + x_4 + \neg z) \]

(a) Consistent assignments

\[ \varphi' = (x_1 + x_3) \cdot (x_2 + x_3) \cdot (\neg x_1 + \neg x_2 + \neg x_3) \cdot (\neg x_3 + z) \cdot (\neg x_4 + z) \cdot (x_3 + x_4 + \neg z) \cdot (\neg z) \]

(b) With property \( z = 0 \)
DLL Algorithm

• Davis, Logemann and Loveland
  

• Also known as **DPLL** for historical reasons (P = Putman)

• It is basically an intelligent
  – Depth First Search (DFS)
  – Binary Covering Problem (BCP)

• Basic framework for many modern SAT solvers
Example

Single gate

\[(^a+^b+ c)(a+^c)(b+^c)\]

Circuit in AIG form: network of connected AND gates

Note:
- Nodes 1,2,3: PI
- Nodes 4,5,6,7,8,9: AND gates
Circuit Satisfiability

\[ \varphi = h \ [d=\neg(ab)] \ [e=\neg(b+c)] \ [f=\neg d] \ [g=d+e] \ [h=fg] \]

\[ = h \]

\[ (a + d)(b + d)(\neg a + \neg b + \neg d) \]
\[ (\neg b + \neg e)(\neg c + \neg e)(b + c + e) \]
\[ (\neg d + \neg f)(d + f) \]
\[ (\neg d + g)(\neg e + g)(d + e + \neg g) \]
\[ (f + \neg h)(g + \neg h)(\neg f + \neg g + h) \]
Conjunctive Normal Form (CNF)

\[ \phi = \left( a + c \right) \left( b + c \right) \left( \neg a + \neg b + \neg c \right) \]
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)
Basic DLL Procedure - DFS

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

0  ⇐ Decision

Slide adapted from ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
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)

(-- Decision

Slide adapted From ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
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)

Slide adapted from ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

- a
- b
- c
- d

Implication Graph

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

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!

ECE 667 - Synthesis & Verification - SAT
Slide adapted From ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

\[
\begin{align*}
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
(a + c' + d) \\
(a + c' + d') \\
(b' + c' + d) \\
(a' + b + c') \\
(a' + b' + c) \\
\end{align*}
\]

Slide adapted from ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(a' + b + c)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

\[d = 1\]
\[c = 1\]
\[a = 0\]
\[b = 0\]

Conflict!

\[\text{Forced Decision}\]

Slide adapted from ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

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

Slide adapted from ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
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)

Forced Decision

Slide adapted From ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

\[ (a' + b + c) \]
\[ (a + c + d) \]
\[ (a' + c + d') \]
\[ (a + c' + d) \]
\[ (a' + c' + d') \]
\[ (a' + b + c) \]
\[ (a' + b' + c) \]

\( c = 0 \)
\( d = 1 \)

\( a = 0 \)
\( d = 1 \)
\( c = 0 \)
\( d = 0 \)

Conflict!

Slide adapted From ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

\[\text{Backtrack}\]
Basic DLL Procedure - DFS

\((a' + b + c)\)
\((a + c + d)\)
\((a + c + d')\)
\((a + c' + d)\)
\((a + c' + d')\)
\((b' + c' + d)\)
\((a' + b + c')\)
\((a' + b' + c)\)

Slide adapted From ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
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)

Slide adapted From ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

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

\[ (a + c + d) \]
\[ (a + c + d') \]
\[ (a + c' + d) \]
\[ (a + c' + d') \]
\[ (b' + c' + d) \]
\[ (a' + b + c') \]
\[ (a' + b' + c) \]

\[ \iff \text{Forced Decision} \]

Slide adapted from 'The Quest for Efficient Boolean Satisfiability Solvers' – Sharad Malik
Basic DLL Procedure - DFS

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

Slide adapted from ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

\((a' + b + c)\)
\((a + c + d)\)
\((a + c' + d)\)
\((a + c' + d')\)
\((b' + c' + d)\)
\((a' + b + c')\)
\((a' + b' + c)\)

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)

⇐ Backtrack

[Diagram of a SAT problem with nodes labeled a, b, c, and d with assignments 0 and 1, and backtracking indicated]

Slide adapted from ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

\(a=1\)
\(b=1\)
\(c=1\)

Forced Decision

Slide adapted from ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

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

ECE 667 - Synthesis & Verification - SAT
Slide adapted From ‘The Quest for Efficient Boolean Satisfiability Solvers’ – Sharad Malik
Basic DLL Procedure - DFS

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

\(a=1\)
\(b=1\)
\(c=1\)
\(d=1\)

\(\text{SAT}\)
Backtrack Search Algorithm

- The algorithm conducts a search through the space of the possible assignments to the problem instance variables.

- The search is improved by
  - Resolution
  - Clause recording
  - Recursive learning

- Backtrack search has proven useful for solving instances of SAT from EDA applications, in particular for applications where the objective is to prove unsatisfiability.
// Input arg: Current decision level d
// Output arg: Backtrack decision level β
// Return value: SATISFIABLE or UNSATISFIABLE

SAT (d, &β) {
    if (Decide (d) != DECISION)
        return SATISFIABLE;
    while (TRUE) {
        if (Deduce (d) != CONFLICT) {
            if (SAT (d + 1, β) == SATISFIABLE)
                return SATISFIABLE;
            else if (β != d || d == 0) {
                Erase (d); return UNSATISFIABLE;
            }
        }
        if (Diagnose (d, β) == CONFLICT) {
            return UNSATISFIABLE;
        }
    }
}
Methods in the Backtrack SAT Algorithm

- **Decide( )**: Identify the necessary assignments
- **Deduce( )**: Returns a conflict indication whenever a clause becomes unSAT
- **Diagnose( )**: Analyzes the conflict and returns a decision level to which the search process is required to backtrack
- **Erase( )**: Clears implied assignments that results from each assignment selection
- **Preprocess( )**: Preprocessing before running SAT algorithm.
Summary of SAT Techniques

Techniques for Backtrack Search

• Formula simplification & clause inference
• Conflict analysis
  – Clause/implicate recording
  – Non-chronological backtracking
• Resolution
• Recursive learning
• Randomization & Restarts