SAT-Based Design Debugging

Sharad Malik
Georg Weissenbacher
Princeton University
Verification vs. Debugging

- **Verification**
  - Make sure that there are no bugs

- **Debugging**
  - Observe error
  - What went wrong?
    - Root-cause analysis

- Understanding bugs is often more time consuming than finding them
Hardware bugs are expensive

- Intel Pentium FDIV Bug 1994
  - incorrect results for division
  - due to errors in the entries in the lookup table used by the digital divide operation algorithm

- Total cost associated with replacement: $475 million
Outline

- Hardware Design/Verification Flow

- SAT Background
  - SAT Encoding of Logic Designs
  - Satisfying Assignments for Debugging
  - Unsatisfiable Instances for Debugging

- Pre-Silicon Debug
  - Localizing faults in Register-Transfer-Level (RTL) designs

- Post-Silicon Validation
  - Localizing faults in manufactured prototypes

- Outlook: Fault Localization in Software

- Summary
Specification

RTL Design

Logic Circuit

Physical Design

Chip Prototype

[Smith, Veneris, Ali, Viglas 2004]
Verification Techniques

- Testing
  - White Box Testing
    - Logic Simulation
  - Black Box Testing

- Formal Verification
  - Automated Theorem Proving (ATP)
  - Model Checking
    - (Partial) Specifications in Temporal Logic
  - Bounded Model Checking
  - Equivalence Checking

- ...
# Applicability of Verification Techniques

<table>
<thead>
<tr>
<th></th>
<th>Logic Simulation</th>
<th>Model Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pre-silicon</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Full observability)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>+ ease of use</td>
<td>+ complete coverage</td>
</tr>
<tr>
<td></td>
<td>- limited coverage</td>
<td>- scalability issues</td>
</tr>
<tr>
<td></td>
<td>- simulation is slow</td>
<td>- complex formalisms</td>
</tr>
<tr>
<td><strong>Post-silicon</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Limited observability)</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>+ real-time execution</td>
<td>not applicable</td>
</tr>
<tr>
<td></td>
<td>- still limited coverage</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- manufacturing cost</td>
<td></td>
</tr>
</tbody>
</table>

- ✓: Advantage
- ×: Not applicable
Find Errors, Localize Faults

- **Error:**
  Discrepancy between observed and expected behavior

- **Fault:**
  Abnormal condition, may cause an error
  - Electrical: signal interference in manufactured prototypes
  - Logical: missing case statement in hardware design

- Temporal and spatial localization of faults
  “typically dominate[s] the effort expended during the debug process for a bug” [Josephson 2006]

- This talk’s focus:
  **Automated Fault Localization**
Outline

- Hardware Design/Verification Flow
- **SAT Background**
  - SAT Encoding of Logic Designs
  - Satisfying Assignments for Debugging
  - Unsatisfiable Instances for Debugging
- Pre-Silicon Debug
  - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
  - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary
module seq_moore(i, o, clk, reset)
    input [2:0] i;
    input clk;
    input reset;
    output o;
    reg [1:0] o;

    reg [1:0] Q; // state variables
    reg [1:0] D; // next state output

    always @(posedge clk)
        Q = D;

    always @(Q or reset or ...)
    begin
        case (Q)
            ...

Huffman Model

Register Transfer Level
Iterative Logic Array

[ Abramovici, Breuer, Friedman 1990 ]
Iterative Logic Array (Example)

\[
\begin{align*}
\text{cycle 1:} & \quad (\overline{r} + i_1^1) \cdot (\overline{r} + s) \cdot (\overline{i_1^1} + s + r) \\
\text{cycle 2:} & \quad (t + i_1^2) \cdot (\overline{t} + r) \cdot (\overline{i_1^2} + \overline{r} + t)
\end{align*}
\]
Outline

- Hardware Design/Verification Flow
  - SAT Background
    - SAT Encoding of Logic Designs
    - Satisfying Assignments for Debugging
    - Unsatisfiable Instances for Debugging
  - Pre-Silicon Debug
    - Localizing faults in Register-Transfer-Level (RTL) designs
  - Post-Silicon Validation
    - Localizing faults in manufactured prototypes
  - Outlook: Fault Localization in Software
  - Summary
Applications of SAT Encoding

Property Checking

- How do we know that the RTL design is correct?
- Check whether certain specified properties hold
- If not, we want a counterexample

"output remains 0 as long as initial state (s) and input 1 (i_1) are 0"
Applications of SAT Encoding

Property Checking

- Bounded Model Checking
  Check whether given property holds for \( k \) cycles

"output remains 0 as long as initial state (s) and input 1 (i_1) are 0"

\[
\begin{align*}
(s \cdot \overline{i_1}) & \rightarrow o^1 \\
(s \cdot \overline{i_1} \cdot \overline{i_2}) & \rightarrow o^2
\end{align*}
\]
Applications of SAT Encoding

Property Checking

- Property holds if unfolding and negated property **UNSAT**
- Any satisfying assignment is a **counterexample** to the claim

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\overline{r} + i_1^1) \cdot (\overline{r} + s) \cdot (\overline{i_1^1} + s + r))</td>
<td>((i_2^1 + o^1) \cdot (s + o^1) \cdot (o^1 + i_2^1 + s))</td>
</tr>
<tr>
<td>((\overline{t} + i_1^2) \cdot (\overline{t} + r) \cdot (\overline{i_1^2} + r + t))</td>
<td>((i_2^2 + o^2) \cdot (\overline{r} + o^2) \cdot (o^2 + i_2^2 + r))</td>
</tr>
</tbody>
</table>

\[\left\{ \left( \overline{s} \cdot \overline{i_1^1} \right) \rightarrow o^1 \right\} + \left\{ \left( \overline{s} \cdot i_1^1 \cdot i_1^2 \right) \rightarrow o^2 \right\}\]
Applications of SAT Encoding

Property Checking

- Any satisfying assignment is a counterexample to the claim
Counterexamples

More valuable than correctness proofs?

“One of the most important advantages of model checking [...] is its counterexample facility. [...] The counterexamples can be essential in finding subtle errors in designs.”

[Clarke, Grumberg, McMillan, Zhao 1995]

- However, a counterexample still doesn’t tell us
  - what went wrong, and
  - where it went wrong.

Root Cause Analysis or Fault Localization
Outline

- Hardware Design/Verification Flow

- **SAT Background**
  - SAT Encoding of Logic Designs
  - Satisfying Assignments for Debugging
  - *Unsatisfiable Instances for Debugging*

- Pre-Silicon Debug
  - Localizing faults in Register-Transfer-Level (RTL) designs

- Post-Silicon Validation
  - Localizing faults in manufactured prototypes

- Outlook: Fault Localization in Software

- Summary
 Unsatisfiable Instances

- Constrain a faulty design with the correct input/output
- The resulting formula is unsatisfiable
- Locate gates that are inconsistent with desired behavior
Minimal Correction Set (MCS)

- Given an UNSAT instance
  - Minimal subset of clauses that must be dropped to make instance satisfiable
  - Any subset of an MCS is not a correction set

\[
(r + s + t) \quad (\overline{r} + s) \quad (r) \quad (s) \quad (t)
\]

dropping both \((\overline{r} + s)\) and \((s)\) “corrects” the formula

- Remaining clauses are consistent

- Is there more than one MCS in our example?
Minimal Correction Set (MCS)

- The following formula has a single minimum MCS:
  - Least cardinality

\[ (\overline{s}) \quad (\overline{r} + s) \quad (r) \quad (s) \]

- \{(r), (s)\} is minimal but not minimum.

- The formula has three different MCSes.
The complement of a minimum correction set
- Largest subset of clauses that can be satisfied

Given an **UNSAT** instance

\[
(r' + s' + t') \quad (r' + s') \quad (r) \quad (s) \quad (t)
\]

What is the largest subset of clauses that can be satisfiable?

The complement of any MAX-SAT solution is an MCS

Converse doesn’t hold:
- Complement of MCS is maximal set of satisfiable clauses
Partial MAX SAT

- Maximum subset of clauses that can be satisfied given certain clauses cannot be dropped

- Given an UNSAT instance

\[(\overline{r} + \overline{s} + t) (\overline{r} + s) (r) (s) (\overline{t})\]

which clauses do we have to drop to make it satisfiable? (the pinned clauses can’t be dropped)
Minimal Unsatisfiable Subsets (MUS) and Unsatisfiable Cores

- An *unsatisfiable subset* is an inconsistent subset of the clauses of the original formula.
- Also, referred to as an UNSAT core.

\[
(\overline{s}) \ (\overline{r} + s) \ (r) \ (s)
\]

- An unsatisfiable subset is *minimal* if dropping one of its clauses makes it satisfiable.
MCSes and MUSes

- Generate all MCSes for a set of clauses

\[
\begin{align*}
\{ & \overline{s} \} \\
\{ & (r), (s) \} \\
\{ & (s), (\overline{r} + s) \}
\end{align*}
\]

\[
\begin{align*}
(\overline{s}) & \quad (\overline{r} + s) & (r) & (s) \\
\{ & \overline{s} \} & & & \\
\{ & (r), (s) \} & & & \\
\{ & (s), (\overline{r} + s) \} & & & \\
\end{align*}
\]

- Each hitting set of the MCSes is an MUS
- Each hitting set of all MUSes is an MCS
- Therefore, dropping the clauses of an MCS “deactivates” all cores
Outline

- Hardware Design/Verification Flow
- SAT Background
  - SAT Encoding of Logic Designs
  - Satisfying Assignments for Debugging
  - Unsatisfiable Instances for Debugging
- Pre-Silicon Debug
  - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
  - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary
Pre-Silicon Debug

Problem Definition

Golden Model

Specification

Requirements, UML, Use Cases, SystemC, ...

Logic net-list

Golden Model

Derive

Test Scenario

Consistent?

Iterative Logic Array

Unfolding
Test Case as Circuit Constraints

- Test scenario is modeled as constraint for iterative logic array.
Test Scenarios as Constraints

Example

**Specification**

“output remains 0 as long as initial state (s) and input 1 \((i_1)\) are 0”

**Logic net-list**

<table>
<thead>
<tr>
<th>Time-frame 1</th>
<th>Time-frame 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s = 0)</td>
<td>(i_1 = 0)</td>
</tr>
<tr>
<td>(i_1 = 0)</td>
<td>(i_2 = 1)</td>
</tr>
<tr>
<td>(i_2 = 0)</td>
<td>(o = 0)</td>
</tr>
<tr>
<td>(o = 0)</td>
<td>(t = 0)</td>
</tr>
</tbody>
</table>
Test Scenarios as Constraints

Example

- Add test-scenario as constraints to circuit
- The corresponding CNF formula is inconsistent
Test Scenarios as Constraints

Example

- *Detecting* the error is only half the story
- Manually *localizing* the fault causing a known error is tedious
Fault Localization Using MCSes

Example

- Use MCSes to identify error location.
- Input/output values are hard constraints (we're not interested in MCSes including them).

Example:

\[ (\overline{s}) (i_1^1) (i_2^1) (o^1) \quad (t) (i_1^2) (i_2^2) (o^2) \]

Cycle 1:
- \( (\overline{r} + i_1^1) \cdot (\overline{r} + s) \cdot (i_1^1 + \overline{s} + \overline{r}) \)

Cycle 2:
- \( (t + i_1^2) \cdot (\overline{t} + r) \cdot (i_1^2 + \overline{r} + t) \)
Fault Localization Using MCSes

General Methodology

- Generate ILA constrained with test case (in CNF)
- Compute all MCSes: Each MCS represents a set of potential fault locations
Outline

- Hardware Design/Verification Flow
- SAT Background
  - SAT Encoding of Logic Designs
  - Satisfying Assignments for Debugging
  - Unsatisfiable Instances for Debugging
- Pre-Silicon Debug
  - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
  - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary
Post-Silicon Validation

Problem Definition

Golden Model

Logic Circuit

Unfolding

Iterative Logic Array

Consistent?

Golden Model

Chip Prototype

Testing

Test Result
Post-Silicon Validation

What has changed compared to pre-silicon?

- Include a different class of faults: electrical faults
  - Not in every time-frame: may be transient or intermittent
- The test result represents the erroneous behavior
- The net-list is the fault-free golden model
  - We assume functional correctness when considering electrical bugs
Post-Silicon Validation

What has changed compared to pre-silicon?

- Limited observability of signals in manufactured chip
  - Trace buffers: Limited recording of select signals
  - Scan chains: Read-out after chip execution stopped
Test Results as Circuit Constraints

- Test results used as constraint for iterative logic array
- ? = information was not recorded
Test Results as Constraints

Example

time-frame 1

s = 0
i_1 = ?
i_2 = 0
o = 0
r = ?

time-frame 2

r = ?
i_1 = ?
i_2 = 0
o = 1
t = ?
Test Results as Constraints

Example

- Add test results as constraints to circuit
- The corresponding CNF formula is inconsistent
Fault Localization Using MCSes

Example

- Use MCSes to identify error location.
- Recorded test results:

\[
(s) \quad (i_1^1) \quad (o^1) \quad (i_2^2) \quad (o^2)
\]

\[
(\bar{s}) \quad (\bar{i}_2^1) \quad (\bar{o}^1) \quad (\bar{i}_2^2) \quad (\bar{o}^2)
\]

\[
\begin{align*}
\text{cycle } 1 & : (r + i_1^1) \cdot (\bar{r} + s) \cdot (i_1^1 + s + \bar{s} + r) \\
\text{cycle } 2 & : (t + i_2^2) \cdot (\bar{t} + r) \cdot (i_2^2 + r + t)
\end{align*}
\]

\[
\begin{align*}
\text{cycle } 1 & : (\bar{i}_2^1 + o^1) \cdot (s + o^1) \cdot (\bar{o}^1 + i_2^1 + s) \\
\text{cycle } 2 & : (i_2^2 + o^2) \cdot (\bar{r} + o^2) \cdot (\bar{o}^2 + i_2^2 + r)
\end{align*}
\]
Post-Silicon Faults and MCSes

Limits of Scalability

- Limited observability results in harder decision problems
  - In Pre-Silicon: Full information about signals in each cycle

- Analysing ILA with thousands of time-frames becomes computationally infeasible
Post-Silicon Faults and MCSes

Limits of Scalability

- Analysis limited to small (contiguous) sequence of cycles
- Scalability of decision procedure determines window size
- Slide window backwards in time to cover different cycles
Sliding windows may fail to locate fault.

Approach is incomplete due to limited information.

In this particular example: we don’t know the value of \( r \).
Would like to propagate information across windows

- At a reasonable computational cost

- Maybe we can infer the value of $r$ in the first window?
Reconstructing Information
With Inferred Values

\[ r = 1 \]

[Diagram of a circuit with labeled inputs and outputs, showing the reconstruction process.]
Backbones

Inferring “Fixed” Signals for Satisfiable Instances

- Backbone of a satisfiable formula:
  Set of variables that have same value in all satisfying assignments

- Consider the satisfiable formula

\[
(r \oplus t) \cdot (r + s) \cdot (r)
\]

- Satisfying assignments:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>s</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Computing Backbones

Given a Boolean formula $F$

1. Obtain initial satisfying assignment $A_0$

2. For each literal $p$ such that $A_0[p]=1$
   - variable of $p$ is part of backbone iff $(F \cdot \overline{p})$ is UNSAT

Optimization (Filtering):

1. If $(F \cdot \overline{p})$ is satisfiable, look at this satisfying assignment $A_1$

2. Variables differing in value in $A_0$ and $A_1$ are not in backbone

[Marques-Silva, Janota, Lynce 2010]
Propagating Backbones Across Sliding Windows

[Zhu, Weissenbacher, Sethi, Malik 2011]
Propagating Backbones Across Sliding Windows

- logged information
- inferred information
- time
- cycle $i$
- cycle $n-1$
- cycle $n$
- crash
- 1st window
- 1st backbone
- 2nd window
- 2nd backbone
Outline

- Hardware Design/Verification Flow
- SAT Background
  - SAT Encoding of Logic Designs
  - Satisfying Assignments for Debugging
  - Unsatisfiable Instances for Debugging
- Pre-Silicon Debug
  - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
  - Localizing faults in manufactured prototypes
- **Outlook: Fault Localization in Software**
- Summary
Fault Localization in Software

Methodology

1. Observe an assertion violation
2. Unwind loops of the program, obtain symbolic representation
3. Constrain program with expected input/output values
4. Compute MCSes to locate faults

[Jose, Majumdar 2011]
Fault Localization in Software

Problem Definition

Golden Model

Specification

e.g., requirements

Software Implementation

Unwinding

Assertions in Program

Consistent?

Symbolic Representation
Symbolic Representation of Software

Static Single Assignment Form
[Cytron, Ferrante, Rosen, Wegman, Zadeck 1991]

```c
int Array[3];

int testme(int index)
{
    if (index != 1)
        index = 2;
    else
        index = index + 2;
    i = index;
    assert (i >= 0 && i < 3); //
    return Array[i];
}
```

guard_1 = (index_1 ≠ 1)  
index_2 = 2  
index_3 = index_1 + 2  
i = guard_1 ? index_2 : index_3
Computing MCSes for Program

test input

\[
\text{index}_1 = 1
\]

\[
\text{guard}_1 = (\text{index}_1 \neq 1)
\]

\[
\text{index}_2 = 2
\]

\[
\text{index}_3 = \text{index}_1 + 2
\]

\[
i = \text{guard}_1 \land \text{index}_2 \lor \text{index}_3
\]

\[
(i < 3)
\]

violated assertion
MCSes Indicate Potential Errors

```c
int Array[3];

int testme(int index)
{
    if (index != 1)
        index = 2;
    else
        index = index + 2;
    i = index;

    assert (i >= 0 && i < 3); // array bounds
    return Array[i];
}
```
Outline

- Hardware Design/Verification Flow
- SAT Background
  - SAT Encoding of Logic Designs
  - Satisfying Assignments for Debugging
  - Unsatisfiable Instances for Debugging
- Pre-Silicon Debug
  - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
  - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary
Summary

- Locating faults is tedious

- Minimal Correction Sets enable fault localization in
  - logic design
  - manufactured chip prototypes
  - software

- Backbones enable partial recovery of information
References (for Hardware)

- [Abramovici, Breuer, Friedman 1990] *Digital Systems Testing and Testable Design*  
  Computer Science Press

- [Smith, Veneris, Ali, Viglas 2004] *Fault Diagnosis and Logic Debugging Using Boolean Satisfiability*  
  IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

- [Josephson 2006] *The Good, the Bad, and the Ugly of Silicon Debug*  
  Design Automation Conference

- [Liffiton, Sakallah 2009] *Generalizing Core-Guided MAX-SAT*  
  Theory and Applications of Satisfiability Testing

- [Chen, Safarpour, Marques-Silva, Veneris 2009] *Automated Design Debugging with Maximum Satisfiability*  
  IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

  European Conference on Artificial Intelligence

- [Zhu, Weissenbacher, Sethi, Malik 2011] *Post-Silicon Fault Localisation Using Maximum Satisfiability and Backbones*  
References (for Software)

- [Clarke, Kroening, Lerda 2004]
  *A Tool for Checking ANSI-C Programs*
  Tools and Algorithms for the Construction and Analysis of Systems

- [Groce, Chaki, Kroening, Strichman 2006]
  *Error Explanation with Distance Metrics*
  International Journal on Software Tools for Technology Transfer

- [Jose, Majumdar 2011] (Tool paper)
  *BugAssist: Assisting Fault Localization in ANSI-C Programs*
  Computer Aided Verification

- [Jose, Majumdar 2011]
  *Clause Cue Clauses: Error Localization Using Maximum Satisfiability*
  Programming Languages Design and Implementation