EE 579: Design Verification

The Problem

• Asks the question: Has a system been designed correctly?
• Hardware debugging and testing now account for half or more of system design costs
• The Pentium’s FDIV design error cost Intel about $475 million. Other errors have cost billions of dollars.
• Development of verification methodologies is handicapped by:
  + Computational complexity of the problem
  + Lack of efficient and/or comprehensive verification methodologies
  + Lack of data on the nature of actual design errors

Design Verification Methods

Formal Verification

• Uses methods of mathematical logic to prove \( spec \equiv \text{impl} \)
  + Requires two detailed and precise descriptions of design
  + Will not detect specification errors

Correctness by Construction

• (Re)uses only “correct” components and design methods/tools
  + Current tools have limited abilities and quality
  + Will not detect specification errors

Simulation

• Exercises computer model of implementation using random or deterministic test patterns
  + Practical but usually time-consuming and incomplete
  + Can detect some specification errors
Design Verification Methods

Formal Methods

- Verify correctness via mathematical proofs
- Basic question: Implementation = Specification?
- Representative methods:
  + Equivalence checking
  + Model (property) checking
  + Theorem proving
- Limitations:
  + Computationally complex
  + Difficult for designers to use (not automated)
  + Inadequacy of specification

Formal Verification

Example [McFarland 1993]

- Description of implementation
  - NOR gate: nor(p,q,r): \( r = \neg (p \lor q) \)
  - Inverter: inv(p,q): \( q = \neg p \)
  - Circuit designed: \( C(a,b,c): \exists x \ nnor(a,b,x) \text{ and } \text{inv}(x,c) \)
- A formal proof must prove the “theorem”
  \( \exists x \ nnor(a,b,x) \text{ and } \text{inv}(x,c) \equiv (c = a \lor b) \)
  - using “first-order” logic (Boolean algebra plus quantifiers like \( \exists x \)
**Formal Verification**

**Example (cont’d)**

\[ \exists x \quad \text{nor}(a, b, x) \text{ and } \text{inv}(x, c) \]

\[ \equiv \exists x \quad (x = \text{not}(a \lor b) \text{ and } c = \text{not}(x)) \]

\[ \equiv \exists x \quad (x = \text{not}(a \lor b) \text{ and } \text{not}(c) = x) \]

\[ \equiv \exists x \quad (\text{not}(c) = \text{not}(a \lor b) \text{ and } \text{not}(c) = x) \]

\[ \equiv \exists x \quad (c = (a \lor b) \text{ and } \text{not}(c) = x) \]

\[ \equiv (c = (a \lor b) \text{ and } \exists x \quad \text{not}(c) = x) \]

\[ \equiv (c = (a \lor b) \text{ and } \text{True}) \]

\[ \equiv c = (a \lor b) \quad \text{[End of proof]} \]

---

**Design Verification Methods**

**Correctness by Construction**

- Uses only “correct” components and design methods/tools
- Basis for design reuse with predesigned and verified components known as intellectual property (IP) circuits or cores
  + Hard cores: fully implemented and laid-out designs
  + Soft cores: high-level synthesizable designs
- **Advantages**
  + Lowers design, verification, and testing costs
  + Supports fast time-to-market
- **Disadvantages**
  + Current tools have limited abilities and quality
  + Designers have limited information on the design components
  + Cores can interact in complex (an erroneous) ways
Design Verification Methods

**Simulation**

- Apply tests (simulation vectors, test programs) to uncover faulty behavior

  - **Representative Methods**
    - Exercising with real programs
    - Focused testing for difficult, “corner” cases
    - Pseudorandom testing

  - **Limitations**
    - Inefficient hit-or-miss process
    - Can be extremely time-consuming
    - Hard to say when is the simulation sufficient
Design Verification Methods

Error Modeling

- Formal methods and simulation do not generally target specific design errors

- An alternative approach is to use synthetic design error models to guide verification

- Few design error models have been proposed

- Synthetic models of incorrect behavior exist in other domains:
  - Hardware testing of logic circuits
  - Software testing

Design Verification Methods

Hardware Testing

- Has a well-accepted fault model at the gate level of abstraction:
  - The single-stuck line (SSL) model: Any logic line in a circuit can be stuck at 1 or 0

- SSL model features:
  - Low complexity
  - Provides a standard coverage measure
  - Efficient automatic test generation tools exist
  - Captures the behavior of only a few actual faults
  - But SSL-based tests give high coverage of actual faults
  - Not easily extensible to higher levels of abstraction
### Design Verification vs. Hardware Testing

- **Error model**
  - Verification tests
  - Design errors

- **Fault model**
  - Physical fault tests
  - Physical faults

- Prototype system
  - Design
  - Manufacturing

- Operational system
  - Residual design errors

### Design Verification Methods

#### Software Testing

- A related approach exists in software testing called **mutation testing**
- Program “mutants” differ from program under test by a small error, e.g. using add instead of subtract
- Supported by two hypotheses:
  - Programmers write programs close to correct ones
  - Tests that distinguishes a program from its mutants are sensitive to more complex errors
- Features of mutation testing:
  - Provides a useful coverage measure (mutation score)
  - Considered too costly for general industrial use
  - Above hypotheses are not widely accepted
  - Potential to provide automated software testing
Design Error Collection

- Detailed error data are useful for:
  + Verification via error-based test generation
  + Benchmarks to compare verification methods
  + Statistical reliability analysis

- Industry often records design errors, but rarely publishes them

- Notable exceptions:
  + MIPS errata for the R4000 microprocessor
  + Intel and others in the wake of the Pentium’s FDIV bug
Design Error Collection

• Industrial bug lists are inadequate for error model construction for several reasons:
  + They provide a programmer’s view that is largely independent of the design implementation
  + Errata lists apply only to the final product and are not representative of all design errors

• The best point to collect data is immediately after a design error is discovered and corrected
  + Feasible in an academic setting using design courses and research projects

Collection Methodology

• Design were created by students are created in a hardware design language (Verilog)
• Participation in the collection effort was voluntary
• Designers were given a handout explaining our objectives to elicit their cooperation
• A revision management tool CVS supported the archiving of successive design revisions
• The designers are asked to submit a new revision of their design to CVS whenever a bug was fixed or their work was interrupted
• Design errors were detected by designer inspection and simulation
• On each revision, the designers were prompted to fill in a brief questionnaire
Bug Report Form

Replace the _ with X where appropriate

MOTIVATION:
- X bug correction
- _ design modification
- _ design continuation
- _ performance optimization
- _ synthesis simplification
- _ documentation

BUG DETECTED BY:
- _ inspection
- _ compilation
- X simulation
- _ synthesis

BUG CLASSIFICATION:
Please try to identify the primary source of the error. If in doubt, check all categories that apply.
- _ verilog syntax error
- _ conceptual error
- X combinational logic:
  - X wrong signal source
  - _ missing input(s)
  - _ unconnected (floating) input(s)
  - _ unconnected (floating) output(s)
  - _ conflicting outputs
  - _ wrong gate/module type
  - _ missing instance of gate/module

- _ sequential logic:
  - _ extra latch/flip-flop
  - _ missing latch/flip-flop
  - _ extra state
  - _ missing state
  - _ wrong next state
  - _ other finite state machine error

- _ statement:
  - _ case statement
  - _ always statement
  - _ declaration
  - _ port list of module declaration

- _ expression (RHS of assignment):
  - _ missing term/factor
  - _ extra term/factor
  - _ missing inversion
  - _ extra inversion
  - _ wrong operator
  - _ wrong constant
  - _ completely wrong

- _ buses:
  - _ wrong bus width
  - _ wrong bit order

- _ new category (describe below)

BUG DESCRIPTION:

Used wrong field from instruction

Design input
Simulate design
Detect bug
Correct bug
Fill out questionnaire
CVS revision database
### Basic Error Models

- **Bus SSL error**: A bus of $n > 1$ lines is totally stuck-at-0 or stuck-at-1
- **Bus source error** (BSE): connecting a module input to a wrong source
- **Module substitution error** (MSE): replacing a module by one with the same number of I/O lines
- **Bus order error** (BOE): incorrectly ordering the bits in a bus

### Test Generation

- ATPG for the preceding error models is relatively easy
- Errors are usually easy to activate; their effects of errors are usually difficult to observe (propagate)
- Test generation for design errors is not supported by CAD tools
- Experiments use manual test generation which limits size and scope
**Example: c880 ALU**

- Based on a Verilog design with 6 modules (383 gates.)
- Error models used: BOEs, BSEs, and MSEs composed of inversion errors on 1-bit signals
- Tried to find to small set of assignments to detect each error, noting that some errors are redundant
  - **BOEs**: 10 tests detect all 22 detectable BOEs.
    - Two BOEs were found to be redundant (undetectable)
  - **BSEs**: Adding 3 tests to the 10 BOE tests detect all 27 BSEs on the c880’s multibit buses
  - **MSEs**: Most 1-bit errors are detected by the BOE and BSE tests.
- The experiment suggests that good coverage of the simple errors can be achieved with few tests
Basic Error Models

• Simple models of the preceding type can be used with RTL circuits, and give high, but incomplete, error coverage with small test sets
• To increase error coverage to the extremely high levels needed for DV, additional error models are needed to guide test generation.
• More complex error models can be derived directly from the actual design error data to supplement the previous error types
  + Bus count error (BCE): using a module with more or fewer input buses than required
  + Module count error (MCE): adding or removing a module
  + Label count error (LCE): incorrectly labels on a case statement
  + Expression error (ESE): deviations from a correct expression, such as extra/missing terms or wrong operator
  + State count error (SCE): FSM with an extra or missing state
  + Next state error (NSE): FSM with an incorrect next-state function

Basic Error Models

• This extended set of models significantly increases the set of actual errors that can be covered
• However, they are too complex for use in manual or automated test generation
• The more difficult-to-detect actual errors often consist of multiple interacting basic errors,
• A test for such an error must be much more specific than a test for its component basic errors.
• New error models should be comparable in number and complexity to the simpler error models
• Trade-offs should be possible between test generation effort and verification confidence
Conditional Error Model

- Conditional error \((C,E)\): \(C\) is a condition (predicate) over circuit signals. \(E\) is a basic error which is only active when \(C\) is satisfied.
- We restrict \(C\) to the form \((y_i = w_i)\), where \(y_i\) is a logic (bus) signal and \(w_i\) is an all-0s or all-1s constant
- The number of terms (condition variables) appearing in \(C\) is the order of \((C,E)\). Some special cases:
  + Conditional single-stuck line (CSSL\(n\)) error of order \(n\);
  + Conditional bus order error (CBOE\(n\)) of order \(n\);
  + Conditional bus source error (CBSE\(n\)) of order \(n\).
- When \(n = 0\), \((C,E)\) reduces to the basic error \(E\)
- CSSL\(n\) errors with \(n > 2\) do not seem practical

Case Study: DLX Microprocessor
**DLX Microprocessor**

- This is a student-written design that implements 44 instructions and a 5-stage pipeline
- It contains about 1500 lines of structural Verilog (excluding library module details)
- Several basic and conditional error models were considered
- The actual errors reported by the designer were analyzed and a modeled error found that it dominated, wherever possible
- A test consisting of a short DLX assembly program was constructed manually for each modeled error

---

### Example Error

<table>
<thead>
<tr>
<th>S1, S0</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>X0</td>
<td>X0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>X1</td>
<td>X1</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>X2</td>
<td>X2</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>X3</td>
<td>X3</td>
<td></td>
</tr>
</tbody>
</table>

- Detection requirements: S1 = 0, S0 = 1, X0 ≠ X1, and Y3 sensitized
- Dominated CSSL1 error: (jump_to_reg_instr = 1, S0 stuck-at-0)
DLX Microprocessor

Example Test

- Targeted at SSL error in the branch prediction logic
- Requires a conditional branch instruction to cause a hit in the branch target buffer when the branch prediction is wrong

```c
// TARGET ERROR: dbp_a5.IN0 stuck-at-0
//
// or3 dbp_a5 (.Y(btb_squash),
// .IN0(hit_but_not_taken),
// .IN1(bw_not_taken),
// .IN2(fw_taken)) ;
//
@C00
main:
    ADDUIr1, r0, #2
loop:
    SUBIr1, r1, #1
    BNEZr1, loop
    NOP
    HALT
```

DLX Microprocessor

Experimental Results

<table>
<thead>
<tr>
<th>Category</th>
<th>ISA</th>
<th>ISAb</th>
<th>Total</th>
<th>INV</th>
<th>SSL</th>
<th>BSE</th>
<th>CSSL1</th>
<th>CBOE</th>
<th>CSSL2</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing module (2)</td>
<td>8</td>
<td>2</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Wrong singal source (1)</td>
<td>9</td>
<td>2</td>
<td>11</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Complex (2)</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inversion (5)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Missing input (4)</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unconnected input (4)</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Missing minterm (2)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Extra input (2)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>27</td>
<td>6</td>
<td>39</td>
<td>7</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Error-Based DV Methodology**

**Conclusions**

- Results to date support the effectiveness of model-based verification:
  - High coverage of actual design errors can be obtained by tests for a few synthetic error types
  - Design error coverage can be quantified at various levels of confidence
- The method can be used to construct focused, incremental tests that detect a wide range of actual bugs
- The conditional error models proved to be especially useful, e.g., for detecting missing logic
- The scope of the experiments is small, and further validation using industrial-size designs is desirable,
- More CAD support for error test generation is needed

**References**

