1. Introduction to Verification

Jacob Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin

Verification of Digital Systems
Spring 2020

January 23, 2020

Goals of This Course

Learn the principles of verification

- Verification is a key task in designing complex chips (as well as software and systems, for that matter)
  - Takes the majority of effort in the design cycle
- We will focus on digital hardware in this class
- Class will cover both simulation-based and formal verification

Apply techniques from the lectures to designs in the lab

- Use commercial software (Cadence, Mentor Graphics)
- Formal equivalence checking
- Specification and application of assertions in simulation
- Portable constrained random tests
- Formal verification of assertions

Course Information

Instructor

- Jacob A. Abraham
- +1-512-471-8983
- jaa@cerc.utexas.edu
- `http://www.cerc.utexas.edu/~jaa`

More on the course

- Course Web Page: `http://www.cerc.utexas.edu/~jaa/verification/`
- Prerequisites: VLSI I (or equivalent), some programming experience, computer architecture
- Lectures and discussion in class will cover basics of course
- Homework, Laboratory exercises will help you gain a deep understanding of the subject

Topics

- Introduction
- Machine learning in verification
- Formal equivalence checking
  - Binary decision diagrams, satisfiability engines
  - Use of term rewriting
  - Sequential equivalence checking
- Dynamic (simulation-based) verification
  - Simulation environments, coverage metrics
  - Assertion-based verification
  - UVM
- Formal property checking
  - Introduction to model checking and comparing finite-state machines
  - Techniques to detect subtle bugs
- Post-Silicon validation
- Verification challenges
- Abstractions to reduce complexity

Lectures in the course

- Introduction
- Example of verification flow in industry (Alan Hunter, ARM)
- Machine learning and AI in verification (Monika Farkash, AMD)
- Formal equivalence checking (combinational)
- Finite-state machines and temporal logic
- Assertion-based verification and SystemVerilog assertions (Harry Foster, Siemens)
- Verification testbenches and UVM (Nagesh Loke, ARM)
- Sequential equivalence checking (Shaun Feng, Samsung)
- Model checking (Amit Goel, Apple)
- Quick Error Detection
- Verifying cache coherency
- Semi-formal verification (Hary Mony, RealIntent)

Lectures in the course, Cont'd

- CPU verification Challenges (Tse-Yu Yeh, Apple)
- GPU verification Challenges (John Coers, Apple)
- SoC verification
- Techniques to extend tool capacity
- Am I ready to be a verification engineer? (Ram Narayan, ARM)
- New directions in verification

Work in the Course

- Lectures
  - Cover fundamentals of the topics
  - Notes posted on the web page
  - Supplemental notes and papers on Canvas
- Homework problems
  - Solve problems posted on Canvas
- Laboratory exercises
  - Use commercial tools to apply techniques to realistic designs
- Project
  - Your opportunity to delve into a verification-related topic of interest to you
  - 2 – 3 person teams
  - Project report and presentation to class at the end of the semester
  - **Work on project throughout the semester**

Laboratory Exercises

Lab. 1 – Logic Equivalence Checking (LEC)

- Formally check logical equivalence between a simple RTL module and its synthesized version
- Example, after DFT insertion
- Cadence *Conformal LEC*

Lab. 2 – Assertion Based Verification (ABV)

- Add assertions to a testbench to verify that the implementation correctly implements design intent
- Document the functional coverage
- Mentor *Questa*

Laboratory Exercises, Cont'd

Lab. 3 – Universal Verification Methodology (UVM)

- Standardized verification methodology
- Testbench in SystemVerilog for a given design
- Design tested for functional bugs
- Mentor *Questa*

Lab. 4 – Formal Property Checking

- Check specified properties in all possible states
- Study effect of improperly specified properties
- Techniques to detect subtle bugs: QED
- Cadence *JasperGold*

Project

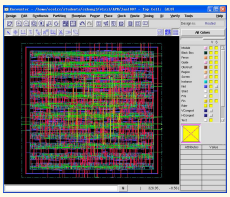
Topics

- Research different areas in verification to pick a topic
- Project can focus on a particular aspect of verification
  - Analysis and comparison of different verification techniques
  - Application of verification to a real design (targets include the SUN OpenSparc (`http://www.opensparc.net/`), the Illinois Verilog Model for the DEC Alpha (`http://www.crhc.illinois.edu/ACS/tools/ivm/about.html`), designs from `http://www.opencores.org/projects/` including Amber ARM (`http://opencores.org/project,amber`), RISC-V (`http://riscv.org/`), Ridecore (`https://github.com/ridecore/ridecore/`), Pulpino (`https://github.com/pulp-platform/pulpino`))

Presentation/Report

- Team presents results to the class (during the last few classes)
- A concise report on the project is due at the end of the course

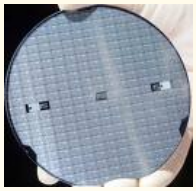
Reliability in the Life of an Integrated Circuit – I



Design

Design "bugs"
Verification (Simulation, Formal)

Fabrication



Wafer

Process variations, defects
Process Monitors

ECE Department, University of Texas at Austin | Lecture 1. Introduction to Verification | Jacob Abraham, January 23, 2020 10 / 44

Reliability in the Life of an Integrated Circuit – II

Wafer Probe

Package

Tester

Identify residual bugs, Test cost, coverage
Post-Silicon Validation, Design for Test, Built-In Self Test

System

Application

Test escapes, wearout, environment
System Self-Test, Error Detection, Fault Tolerance

ECE Department, University of Texas at Austin | Lecture 1. Introduction to Verification | Jacob Abraham, January 23, 2020 11 / 44

Verification versus Validation – From IEEE "PMBOK guide"

Verification

"The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process."

Are we designing the system right?

Validation

"The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers."

Are we designing the right system?

Historical Interest in Verification

A saga of "correct" software

- In 1969, Naur published a technique for constructing and proving software, and applied it to a text processing problem
  - Informally proved correctness of about 25 lines of ALGOL 60
- Leavenworth in a 1970 review pointed out that the first line of the output would be preceded by a blank unless the first word had exactly the maximum number of possible characters in a line (MAXPOS)
- London found three additional faults in 1971 (e.g., procedure would not terminate unless word with more than MAXPOS characters encountered)
  - Presented a corrected version and proved it formally
- Goodenough and Gerhart found three further faults in 1975 that London had not detected (included the fact that the last word would not be output unless it is followed by a BLANK or NIL)

Historical Interest, Cont'd

A saga of "correct" software, Cont'd

- Of the total of seven faults detected by the above researchers, four could have been detected simply by running the procedure on test data!
- Difficult to capture the specifications and requirements against which an implementation is proved correct

National Computer Conference, 1978

- Panel session: "Formal methods in programming – When will they be practical?"

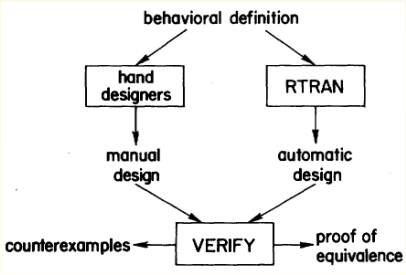
Don Knuth:

"Beware of bugs in the above code; I have only proved it correct, not tried it"

Historical Interest in Verification, Cont'd

Hardware

- J. Paul Roth, "Hardware Verification", *IEEE Transactions on Computers, December 1977.*



Analyzing Complex Designs

Need to (implicitly) search a very large state space

- Find bugs in a design
- Generate tests for faults in a manufactured chip

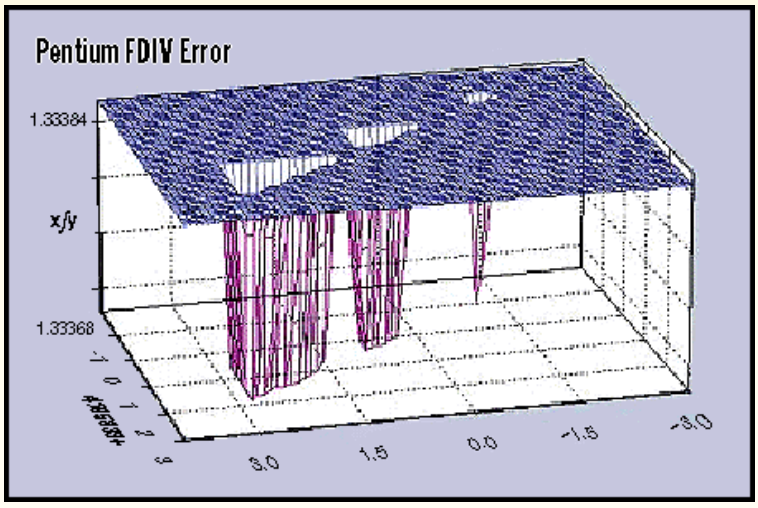
Basic algorithms for even combinational blocks (SAT, ATPG) are NP-complete Approaches to deal with real designs

- Exploit hierarchy in the design
- Develop abstractions for parts of a design

State-space explosion: A design with 300 state variables has more states than the number of protons in the universe (10^{80})!

The (In)Famous Pentium FDIV Problem

Graph of x, y, x/y in a small region by Larry Hoyle



What is a "Bug"?

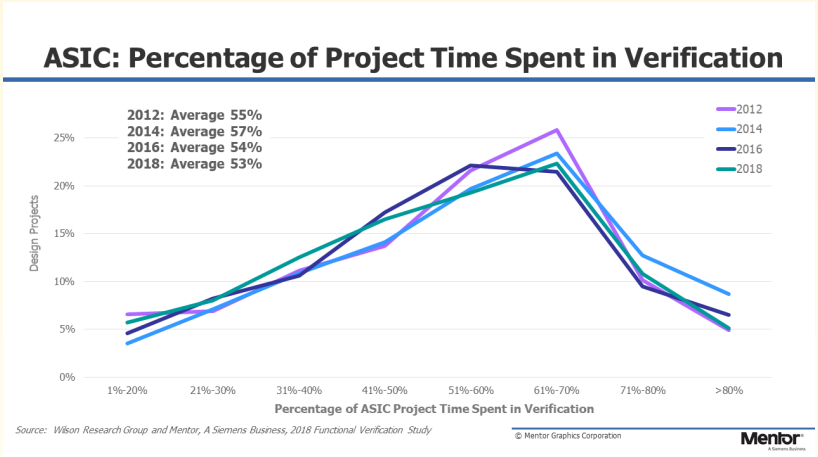
Design does not Match the Specification

- One problem: complete (and consistent) specifications may not exist for many products
- For example, the difficulty in designing an X86 compatible chip is not in implementing the X-86 instruction set architecture, but in matching the behavior with Intel chips

Something which the customer will complain about

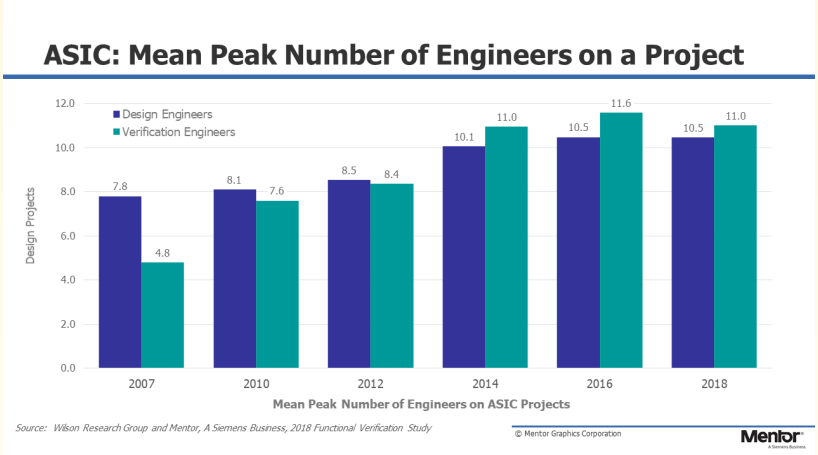
- "It's not a bug, it's a feature"

Verification Consumes the Majority of Project Time



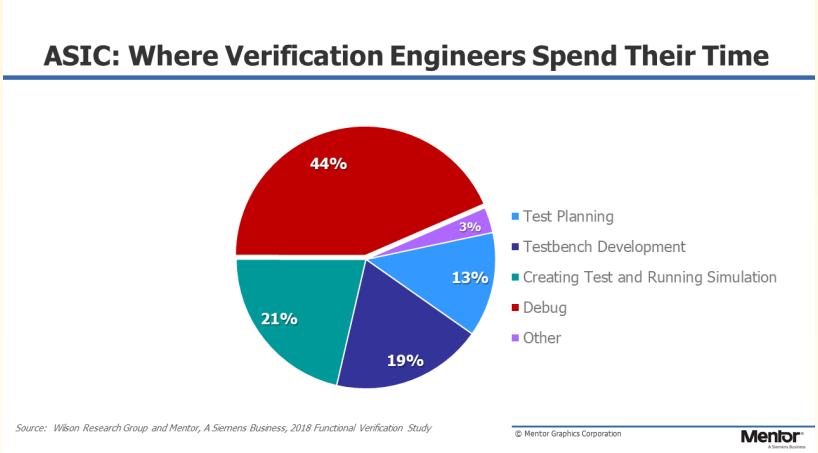
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Number of Engineers on a Project: Design vs. Verification



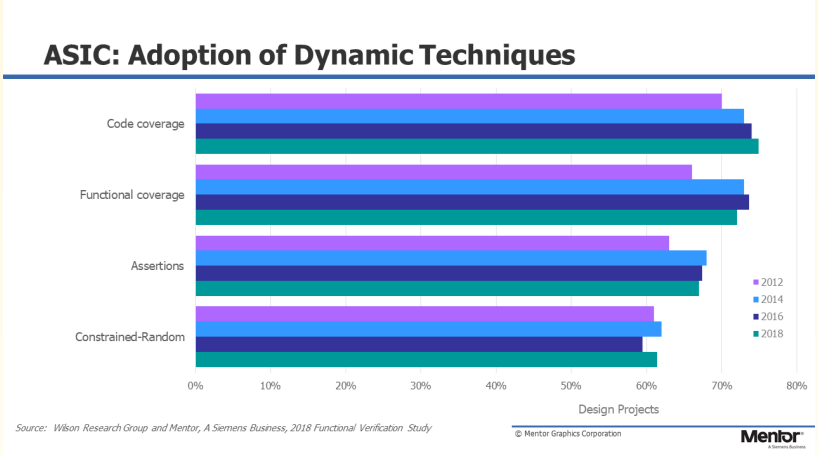
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Where Verification Engineers Spend Their Time



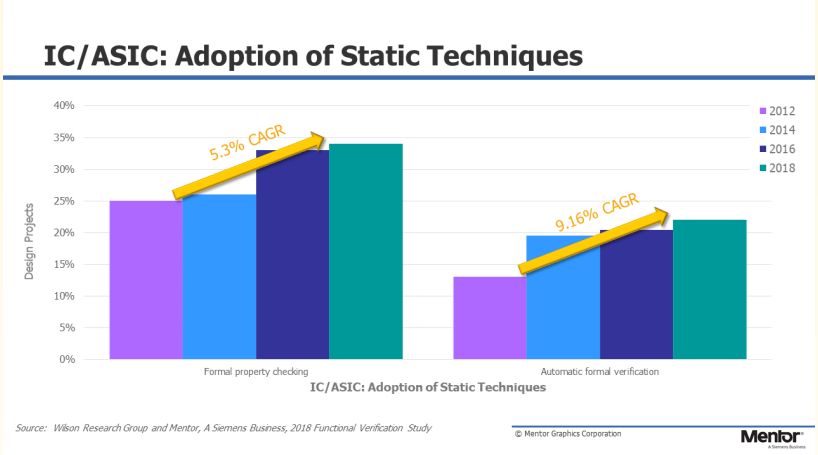
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Adoption of Dynamic Verification Techniques



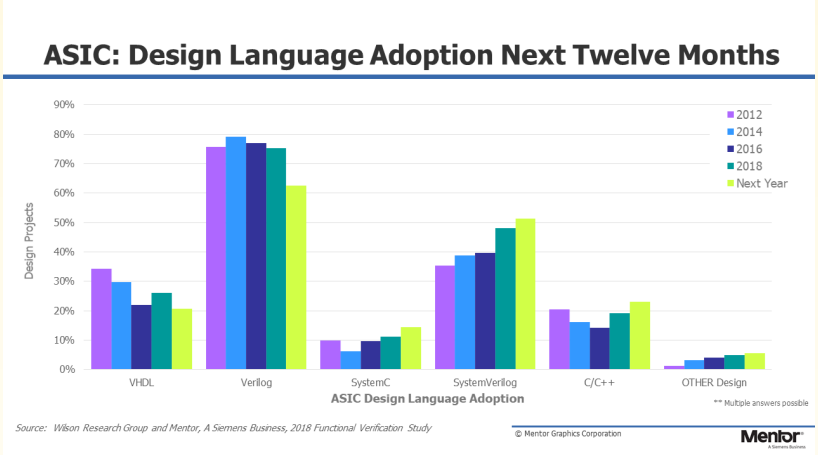
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Adoption of Static (Formal) Verification Techniques



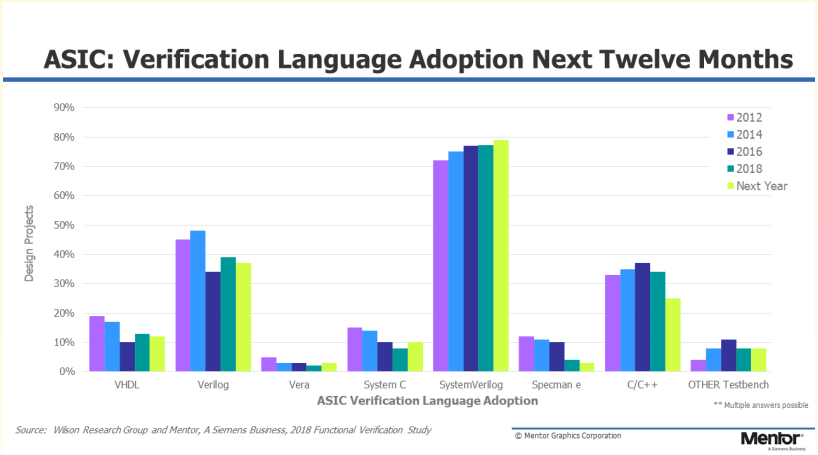
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

RTL Design Language Adoption



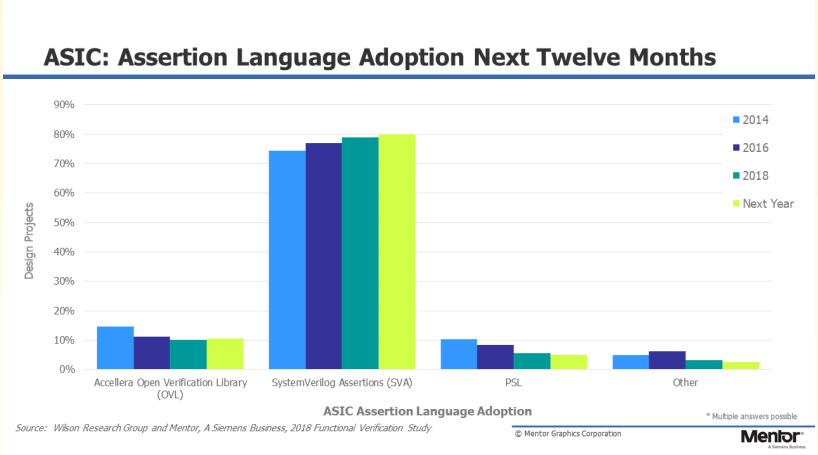
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Verification Language (Testbench) Adoption



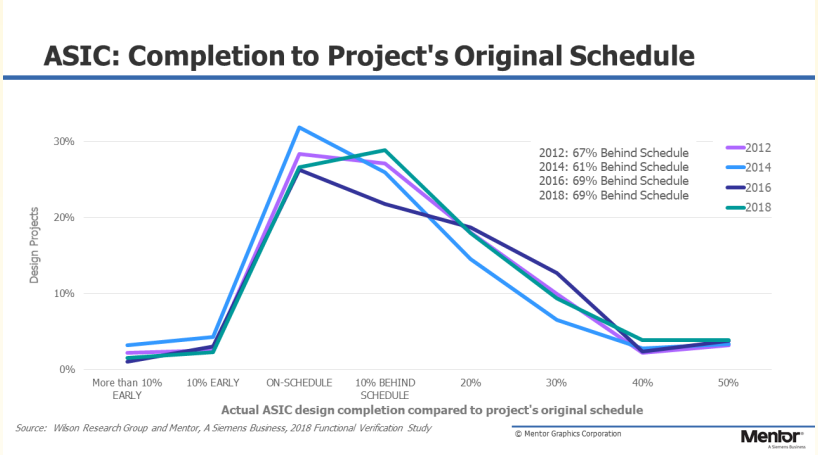
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Assertion Language Adoption



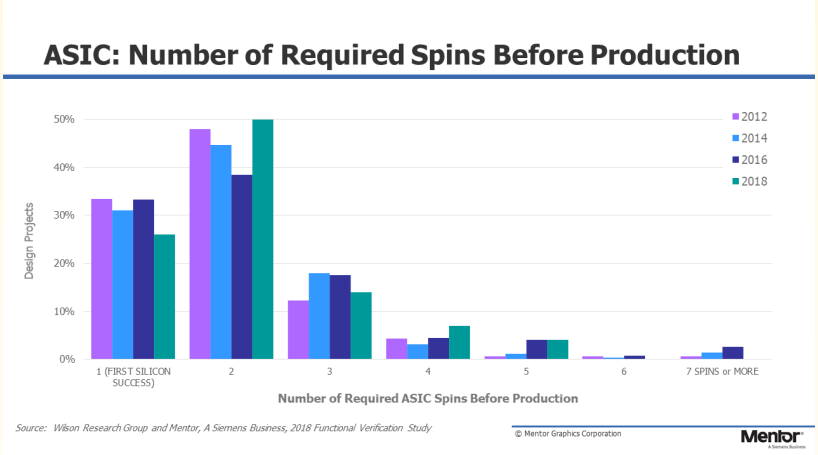
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Many Projects Miss Schedule



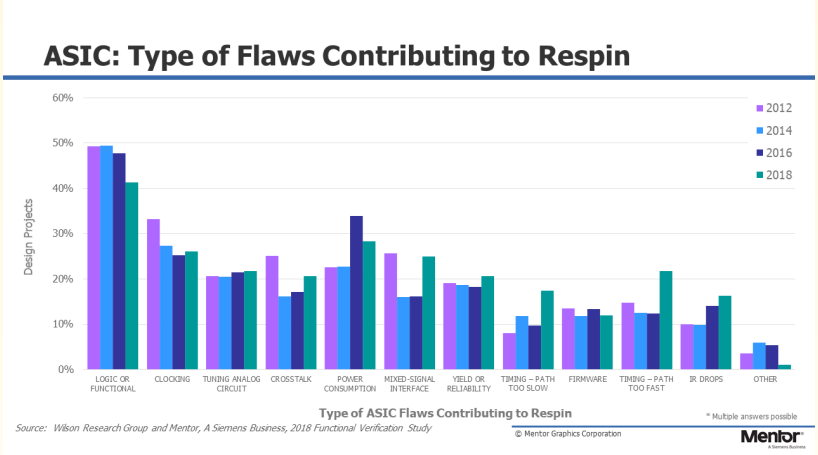
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Required Spins Before Production



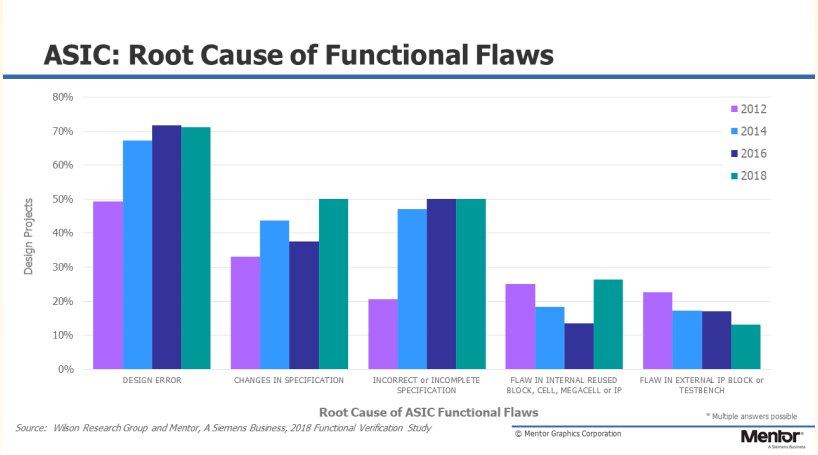
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Flaws Contributing to Respin



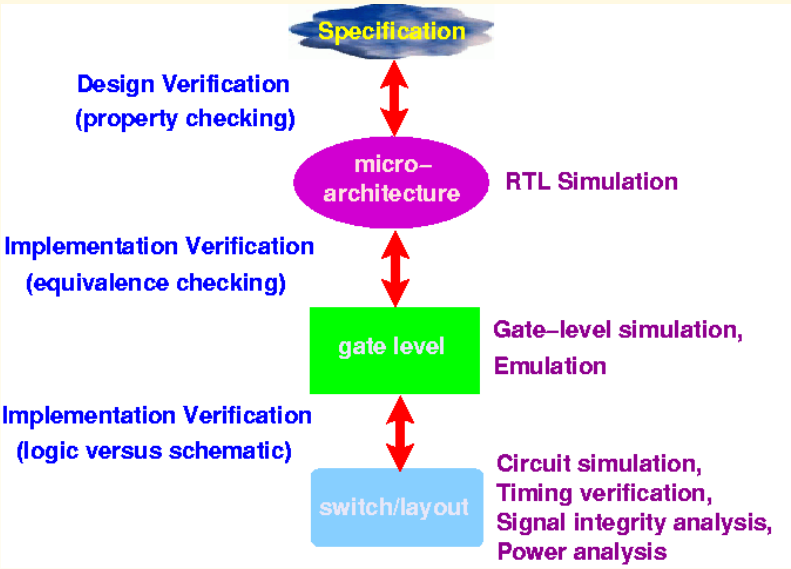
Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Root Cause of Functional Flaws



Source: Wilson Research Group and Mentor Graphics, 2018 Functional Verification Study

Design and Implementation Verification



Verification Approaches

- Simulation (the most popular verification method)
  - Cycle based, functional simulation for billions of cycles
  - Good coverage metrics usually not available
  - *Assertions* used to specify behavior
  - Emulation
    - Capital intensive
    - Map design to be verified on FPGAs
    - Run OS and application at MHz rates
- Formal verification
  - Exhaustive verification of small modules
  - Formal equivalence checking
  - Property checking
- Techniques to manage complexity
  - Compositional techniques
  - Make use of symmetry
  - Abstractions

Evaluating the Complete Design

- Is there a verification technique which can be applied to the entire chip?
- Only one approach which scales with the design: **Simulation**
- Most common technique now used in industry
- **Cycle-based simulation** can exercise the design for millions of cycles
  - Unfortunately, the question of when to stop simulation is open
  - No good measures of **coverage**
- **Emulation**
  - Used to verify the first Pentium (windows booted on FPGA system)
  - Developing another accurate model is an issue

When are we Done Simulating?

When do you tape out?

- Motorola criteria (EE Times, July 4, 2001)
- 40 billion random cycles without finding a bug
- Directed tests in verification plan are completed
- Source code and/or functional coverage goals are met
- Diminishing bug rate is observed
- A certain date on the calendar is reached

Coverage-Driven Verification

Attempt to Verify that the Design Meets Verification Goals

- Define all the verification goals up front in terms of "functional coverage points"
  - Each bit of functionality required to be tested in the design is described in terms of events, values and combinations
- Functional coverage points are coded into the verification environment
  - Simulation runs can be measured for the coverage they accomplish
- Focus on tests that will accomplishing the coverage ("coverage driven testing")
  - Then fix bugs, release constraints, improve the test environment
  - Measurable metric for verification effort

Open Questions

Are There Better Measures of Coverage?

- Coverage of statements in RTL would be a **necessary** but not **sufficient**
- Coverage of all states is impractical even for a design with a few hundred state variables
- Is there a way to identify a **subset of state variables** that would be tractable, and would lead to better bug detection?
- How would these variables be related to the **behavior** of the design?

Assertions

- Assertions capture knowledge about how a design should behave
- Used in coverage-based verification techniques in a simulation environment as well as in formal verification
- Assertions help to increase observability into a design, as well as the controllability of a design
- Each assertion specifies
  - legal behavior of some part of the design, or
  - illegal behavior of part of the design
- Examples of assertions (will be specified in a formal language)
  - The fifo should not overflow
  - Some set of signals should be "one-hot"
  - If a signal occurs, then ...

Simulation Monitors and Assertions

```
assert_never underflow ( clk, reset_n,
    (q_valid==1'b1) && (q_underflow==1'b1));
```

RTL Design

```
module assert_never (clk, reset_
 input clk, reset_n, test_expr;
 parameter severity_level = 0;
 parameter msg="ASSERT NEVER VIOLATION";
// ASSERT: PRAGMA HERE
 //synopsys translate_off
 `ifdef ASSERT_ON
  integer error_count;
  initial error_count = 0;
  always @(posedge clk) begin
   `ifdef ASSERT_GLOBAL_RESET
    if (`ASSERT_GLOBAL_RESET != 1'b0) begin
   `else
    if (reset_n != 0) begin // active low reset_n
   `endif
     if (test_expr == 1'b1) begin
       error_count = error_count + 1;
       `ifdef ASSERT_MAX_REPORT_ERROR
        if (error_count<=`ASSERT_MAX_REPORT_ERROR)
       `endif
         $display("%s : severity %0d : time %0t : %m", msg, severity_level, $time);
       if (severity_level == 0) $finish;
     end
    end
  end
 `endif
//synopsys translate_on
endmodule
```

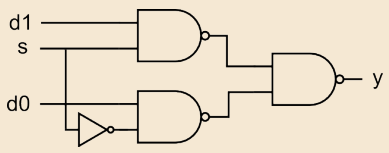

Equivalence Checking

- Validate that the implementation of a module is consistent with the specification
  - Can use simulation or formal techniques
  - Combinational or sequential modules

Example: Specification in RTL

```
module mux(input s, d0, d1,
           output y);
assign y = s ? d1 : d0;
endmodule
```

Example: Implementation at the gate level



Design Verification

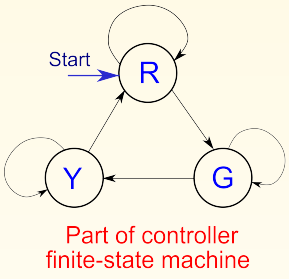
- Digital systems similar to **reactive programs**
- Digital systems receive inputs and produce outputs in a continuous interaction with their environment
- Behavior of digital systems is concurrent because each gate in the system simultaneously evaluating its output as a function of its inputs

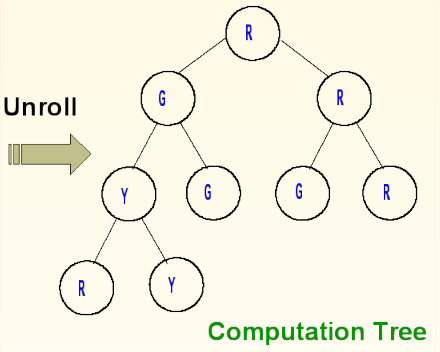
Check Properties of Design

- Since specification is usually not formal, check design for properties that would be consistent with the specification
- Safety "something bad will never happen"
- Liveness Property: "something good will eventually happen"
- Temporal Logic and variations commonly used to specify properties
- Example: Linear Temporal Logic (LTL) or Computation Tree Logic (CTL)

Example of Computation Tree

Traffic light controller





Dealing with State Explosion

Verification is a Very Difficult Problem

- Even combinational equivalence checking problems (ATPG, SAT) are NP-complete
- Checking sequential properties is only possible for small designs
- Additional problem of generating correct "wrappers" for the module being verified

How can we deal with the complexity?

- Use more powerful computers?
  - Computers double in capability (assuming we can program multi-core processors) every couple of years
  - Adding one state variable to a design doubles its states
- Exploit hierarchy in the design
- Develop powerful abstractions

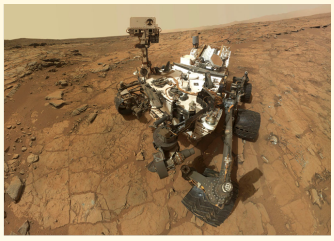
Program Slicing

A Slice of a Design

- Represents behavior of the design with respect to a given set of variables (or slicing criterion)
- Proposed for use in software in 1984 (Weiser)
- Slice generated by a control/data flow analysis of the program code
- Slicing is done on the **structure** of the design, so scales well
- "Static analysis"

Dramatic Example of Design Bug Detection and Recovery

BAE RAD750 (133 MHz)



Science Lab on Mars

- Mars Science Laboratory – MSL ("Curiosity") relies extensively on BAE RAD750 chip running at 133 MHz
- During cruise to Mars (circa January 2012), MSL processes are unexpectedly reset – no code bug is found (7 months remaining before landing)
- Misbehavior eventually traced to processor hardware malfunction: instruction flow depends on processor temperature
- Only possible fix was via software – done successfully

ECE Department, University of Texas at Austin | Lecture 1. Introduction to Verification | Jacob Abraham, January 23, 2020 44 / 44