Advanced Verification Methodology
ECE 4280/5280
Wed 4:45pm-7:20pm

Nathan Pihlstrom
My Background

B.S.E.E. from Colorado State University
M.S.E.E. from Colorado State University
M.B.A. from UCCS

Ford Microelectronics, Inc./Visteon 1985 - 2000
Intel Corp. 2000-2006
Marvell 2006-2009
Hittite Microwave 2009-2011
Covidien 2011 - 2013
LSI/Avago 2013 - present

UCCS Fall Semester 2013 – ECE4280/5280
UCCS Spring Semester 2014 – ECE4211/5211
Course Description

Advanced Verification Methodology. Verification of electronic systems consumes **70% of the development cycle.** This course teaches students how to develop high-quality verification environments with **SystemVerilog** and how to use advanced verification techniques such as **assertions** and **coverage** for digital systems.

Prerequisite: ECE 4242/5242.
Assumptions

• Prerequisite is Advanced Digital Design Methodology (4242/5242) or equivalent
• Strong knowledge of Verilog
• Familiarity with a simulator (preferably QuestaSim)
• Object oriented exposure helpful but not required
References

Spear, C. and Tumbush, G.,  

Tools

QuestaSim 10.1b 64-bit
Semester Topics

- Introduction to Verification
- SystemVerilog Language Constructs
- Test Bench Environments
- Interfaces
- Clocking blocks and programs
- Basic OOP
- Randomization
- Threads and Interprocess Communication
- Advanced OOP
- Functional Coverage
- Advanced Interfaces
- Interfacing with C/C++
Class 1 Topics

• What is verification?
• Why verify?
• The verification process
• Verification plan exercise
• Directed testing
• Testbench exercise
• Randomization
• Functional coverage
• Testbench components
• Black/white/gray box verification methods
• HW 1
Verification versus Validation

PMBOK Guide book (adopted by IEEE) states:

**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. Contrast with *validation*.

Are we building it 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. Contrast with *verification*.

Are we building the right thing?
What is verification?

• Verification ensures that the RTL performs the correct function.
• What defines the correct function of the RTL?

What is the format of the specification?
1. Document
2. Executable specification:
   1. SystemC
   2. C++
   3. Matlab
   4. etc.
Why Verify?

• **Medical**
  • IEC 62304 is vague w.r.t. FPGA HDL
    • Covidien considers Verilog/VHDL as software and thus, require independent verification

• **Aviation**
  • DO-254
  • Verification results (simulation waveforms, regression status, coverage figures) must be traceable and linked to the formal requirements.

• **Automotive**
  • ISO 26262

• **Industrial**
  • IEC 61508
Why Verify?

The later in the product cycle a bug is found the more costly it is.

<table>
<thead>
<tr>
<th>Product Cycle</th>
<th>Cost of Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Level</td>
<td></td>
</tr>
<tr>
<td>Subsystem Level</td>
<td></td>
</tr>
<tr>
<td>System Level</td>
<td></td>
</tr>
<tr>
<td>Post Silicon</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td></td>
</tr>
</tbody>
</table>

Intel Pentium FPU Bug Cost > $475M
FPGAs Need Verification!

Case Study 1:
LOA Technology
2 fully custom FPGAs

45 man weeks of verification effort 131 bugs found
10 man-weeks of lab debug 9 bugs found
100% statement coverage
Case Study 2:
• Fore River Group
• Hardware acceleration for network security.
• Existing testbench with 100 directed tests.
• Shipping to customers.
• 7 man-months of effort.
• Using random verification found 40 bugs

Case Study 3:
• Fore River Group
• Packet Switching device
• Verification considered complete
• 6 man-months of effort.
• Using random verification found 42 bugs
1.1 The Verification Process

Specification

- Designer interprets the spec
  - Designer creates a design spec
    - Designer creates the logic
- Verifier interprets the spec
  - Verifier creates a verification plan
    - Verifier creates tests

BUGS!
1.1.1 Testing at Different Levels

Block Level
- Maximum control
- Need to emulate peripheral blocks
- Might need to emulate system I/O
- Easiest to detect/debug
- Highest simulation performance

Boundary Level
- Block designer interpretation of interface
- Need to emulate system I/O
- Harder to debug

System Level
- Minimum control
- Need to emulate system I/O
- Hardest to debug
- Lowest simulation performance
- Test system introducing errors
1.1.2 The Verification Plan

1. Binds a requirement in the specification to the test(s)

Specification:
The I²C bus will read/write the config registers

Test Plan:
• Check reset values
• Write a writable config reg
• Write a read-only config reg
• Read a config reg
• Read a memory mapped input
• Stall an I²C transaction

2. Technique for testing feature
   a. Block, hybrid, or System level (2nd party master)
   b. Directed or random
   c. Assertions
   d. Emulation (FPGA, accelerator, etc.)
   e. Verification IP
   f. Self checking or visual (avoid visual if possible)
Evolution of the Verification Plan

Design and Verification follow the “Waterfall” flow

Continuously updated

V-Model

Source: Washington State DOT
1.2 The Verification Process

Vendor Tools available:

Mentor Graphics

ReqTracer – Automates the capture of requirements and matches verification results to those requirements
Works with QuestaSim, DOORS, Excel, etc.

Synopsys

HVP – Hierarchical Verification Plan
VMM – Verification Methodology Manual
Free download of VMM Golden Ref. Guide at vmmcentral.com

IBM Rational Tools

Doors – Requirements Management
Change – Change Management
Verification Plan Exercise

Write a verification plan for an ALU with:

1. Asynchronous active high input reset
2. Input clock
3. 4-bit signed inputs, A and B
4. 5-bit registered signed output C
5. 4 opcodes
   i. Add
   ii. Sub
   iii. bitwise invert input A
   iv. reduction OR input B
Traditional Testbench

- Application Specific – non reusable
- One interface
- Good for simple designs
- For complex designs
  - Very difficult to predict all input sequences
  - Can’t generate stimulus by hand
  - How to determine when done?
1.3 Basic Testbench Functionality

1. Generate stimulus
2. Apply stimulus to DUT
3. Capture the responses
4. Check for correctness
5. Measure progress
1.4 Directed Testing

Most (all) probably specified directed testing in their test plan

- Steady progress
- Little up-front infrastructure development
- Small design changes could mean massive test changes

![Graph showing test plan completion over time]
Verification Exercise

Write a self-checking testbench to verify your ALU. Assume the following encoding of the opcodes.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>2'b00</td>
</tr>
<tr>
<td>sub</td>
<td>2'b01</td>
</tr>
<tr>
<td>bitwise invert input A</td>
<td>2'b10</td>
</tr>
<tr>
<td>reduction OR input B</td>
<td>2'b11</td>
</tr>
</tbody>
</table>
Our verification environments will use the following principles:

1. Constrained random stimulus
2. Functional coverage
3. Layered testbench using transactors
   1. Writing takes more time, thus the initial delay.
4. Common testbench for all tests
5. Test-specific code kept separate from testbench
1.7 What should you randomize?

Much more than data
1. Device configuration
2. Environment configuration
3. Protocol exceptions
4. Errors and violations
5. Delays
6. Test order
7. Seed for the random test
$random

• Available since Verilog 1995.
• $random creates a 32-bit \textit{signed} random number.
• If your range is a power of 2 use implicit bit selection

```verilog
logic [3:0] addr;
addr = $random;
```

• If your range is not a power of 2 use modulus (%).
  Example: generate a random addr between 0 and 5

```verilog
addr = $unsigned($random) \% 6;
```

• New to SystemVerilog is $urandom and $urandom\_range

```verilog
$urandom\_range(5,0);
```

• Unless your code changes the random generation will be the same. Use a seed to change the generation

```verilog
integer r_seed = 2;
addr = $unsigned($random(r_seed)) \% 6;
```
1.8 Functional Coverage

How do you know your random testbench is doing anything useful?

Functional coverage measures how many items in your test plan have been tested.

For ALU Example:
1. Have all opcodes been exercised?
2. Have operands taken values of max pos, max neg, 0?
3. Have all permutation of operands been exercised?

Functional coverage can be collected manually or by writing SystemVerilog coverage statements.
1.8.1 Feedback from Functional Coverage

Tests might need to be modified or additional tests written due to feedback from functional coverage.
1.9 Testbench Components

• Testbench wraps around the Design Under Test
  • Generate stimulus
  • Capture response
  • Check for correctness
  • Measure progress through coverage numbers

• Features of an effective testbench
  • Reusable and easy to modify for different DUTs ← Object oriented
  • Testbench should be layered to enable reuse
    • Flat testbenches are hard to expand.
    • Separate code into smaller pieces that can be developed separately and combine common actions together
  • Catches bugs and achieves coverage quickly ← Randomize!
1.10 Layered Testbench

Flat testbench

```verilog
//Write 32’hFFFF to 16’h50 @(posedge clk);
PAddr <= 16’h50;
PWData <= 32’hFFFF;
PWrite <= 1’b1;
PSel <= 1’b1;

// Toggle PEnable @(posedge clk);
PEnable <= 1’b1;
@ (posedge clk);
PEnable <= 1’b0;
```

Layered testbench

```verilog
task write(reg[15:0] addr,
          reg [31:0] data);
    @(posedge clk);
PAddr <= addr;
PWData <= data;
PWrite <= 1’b1;
PSel <= 1’b1;

    // Toggle PEnable @(posedge clk);
PEnable <= 1’b1;
@ (posedge clk);
PEnable <= 1’b0;
endtask
```
1.10.2 Signal and Command Layers

- Signal Layer
  - DUT and all pin level connections
- Command Layer
  - Driver converts low level commands to inputs of DUT
  - Assertions look at I/O or internal signals of DUT
  - Monitor converts outputs of DUT to low level transaction results
1.10.3 Functional Layer

- **Agent**
  - Breaks down higher level commands into low level commands
  - Informs the Scoreboard of commands sent to the DUT
- **Scoreboard**
  - Predicts the results of the commands
- **Checker**
  - Compares result predicted by scoreboard with the result collected by the Monitor
1.10.4 Scenario Layer

- Generator
  - Breaks down a scenario into high level commands
- Scenario examples for an MP3 player
  - play music from storage
  - download new music
  - adjust volume
  - change tracks, etc.
1.10.5 Test Layer and Functional Coverage

- Test block determines:
  - What scenarios to run
  - Timing of scenarios
  - Random constraints

- Functional Coverage
  - Measures progress of tests
  - Changes throughout project

---

**Diagram:**
- **Test**
- **Scenario Generator**
- **Driver**
- **Agent**
- **Scoreboard**
- **Assertions**
- **Checker**
- **Monitor**
- **DUT**
- **Environment**
- **Functional Coverage**

---

Chapter 1 Copyright 2012 G. Tumbush, C. Spear v1.2
1.12 Simulation Environment Phases

• **Build**
  - Generate DUT configuration and environment configuration
  - Build testbench environment
  - Reset the DUT
  - Configure the DUT

• **Run**
  - Start environment
  - Run the test

• **Wrap-up**
  - Sweep
  - Report
1.13 Maximum Code Reuse

- Put your effort into your testbench not into your tests.
- Write 100’s or 1000’s of directed tests or far fewer random tests.
- Use directed test to cover missed functional coverage
1.14 Testbench Performance

- Directed tests run quicker than random tests
- Random testing explores a wide range of the state space
- Simulation time is cheap
  - Your time is not
- Avoid visually verified tests
  - Examining waveforms are error prone and hard to replicate
- Test maintenance can be a huge effort
Black/White/Grey Box testing methods

• Blackbox verification
  • Use I/O only for determining bugs
  • Difficult to fully verify and debug

  ![Blackbox Verification Diagram]

• Whitebox verification
  • Use I/O and signals in the DUT for determining bugs
  • Easy to debug

  ![Whitebox Verification Diagram]

• Greybox verification
  • A hybrid of blackbox and white box verification

  ![Greybox Verification Diagram]