EECS 4340: Computer Hardware Design
Unit 4: Validation

Prof. Simha Sethumadhavan

Reference Book: System Verilog for Verification
Agenda

- Last Unit
  - Design abstractions
  - Basic primitives

- This Unit
  - Validation

- Forthcoming
  - Design Tips
  - Power, layout, floorplanning
Unit Outline

• Importance of Validation
  • Cost

• Defensive HDL Programming: Expecting the unexpected
  • Assertions
  • Coding Style/Reviews

• Different types and levels of test benches

• How to architect a random test bench?

• Validation Planning
  • Creating a test plan
  • Bug handling
  • Calling it done
Validation

- Process used to demonstrate that intent of design is preserved in its implementation

- When?

- Importance
  - 70% of the design effort is spent in Validation
  - A design bug can cause billions of dollars
    - Recall costs, Opportunity costs, Re-spin costs, Re-design costs see U#1.
  - Validation is on the critical path
Assertions (Pg. 107)

• Assertions: Things that must be true about the design
  • Guard against potentially dangerous scenarios
  • Can be specified by designer as part of the model, or by verification engineer as part of validation

• Typical uses
  • Can be used to validate assumptions made by other designers
    • *E.g.,* read_valid & write_valid should not simultaneously asserted
  • Check functionality provided by 3rd party IP or local state machine
    • *E.g.,* Memory controller should provide output 4 cycles after input
  • Check that something eventually happens
    • *E.g.,* After reset, the clock goes high again

• Highly valuable
  • Understand your design better
  • Can be used for proving design properties
System Verilog Assertions

- **Immediate Assertions**
  - Statements that execute at the current time
  
  ```verilog
  [name:] assert ( expression ) [pass_stmt] [else fail_stmt];
  always_comb assert ($onehot(state)) else $fatal("Not one hot");
  ```

  `$info, $warning, $error, $fatal`

- **Concurrent Assertions**
  - Assertions that can operate over a period of time
  
  ```verilog
  [name:] assert property ( property ) [pass_stmt] [else fail_stmt];
  assert_request_2state: assert property (request_2state);
  ```

  ```verilog
  property request_2state;
      @(posedge clk) disable iff (reset)
          request ##3 grant ##1 !request ##1 !grant;
  endproperty Request must be true immediately, grant must be true 3 clock cycles later, followed by request going false, and then grant goes false
  ```
Tips

• Keep it simple
  • Invariant assertions and sequential assertions are common
    • Most common: Invariant: Assert “something doesn’t happen”
    • Sequential: Assert some sequence of events happens
    • Least common: Eventual: Something eventually happens
  • Assertion errors can also be productivity hogs

• Good places to use assertions
  • One hot state machines ($\text{onehot}$, $\text{onehot0}$)
  • Distributed bus protocols (sequential)
  • Checking that values should not go to X or Z ($\text{isunknown}$)

• Let us practice some assertions
Defensive Coding

- Code inspection is effective at catching bugs
  - Two types: Automated or Manual

- Automated analysis
  - Sophisticated static analysis based tools
  - Pre-packaged rules to check HDL code for synthesizability, simulatability, testability, reusability, and RTL/gate signoff
  - Similar to Lint tools (bit-type mismatch, false positives)
  - Synopsys Leda tools (demo)

- Manual analysis: Source code peer reviews
  - Identify problems that cannot be found by automated tools
  - Provide feedback on structure, organization, comments
Defensive programming
!=
sub-optimal, underperforming design

“Intellectual Complexity” of Design
!=
Validation Complexity of Design
Levels of Validation

Module level
- Ad-hoc testing by designer “directed tests”
- `$display`, .stim files etc., (unsophisticated)

Design Unit level
- Random unit level transactional testing
- Cycle accurate checker Stress test

Full design
- Integration testing
- Few cycles but full components

Validation Time
- ~10 cycles
- ~10^8 cycles
- ~10^4 cycles
Design Flow

TIME

Specification → RTL Coding → Physical Design → Design Validation → Tapeout/Fabrication → Deployment
Random Validation

- **Principle:** Reduce the chance of same errors
  - Two different people
  - Two different language subsets
  - Random inputs too!
Abstract Hardware Unit

Control
- Read Valid
- Write Valid
- Read addr
- Write addr

Data

Input Interfaces

Output Interfaces

Hardware Unit
Transactional Testing

- Each cycle
  - Pick a random subset of interfaces to excite
  - Setup values for those interfaces
  - Interfaces and values may be restricted by state of unit

- Each cycle
  - Check all of the output interfaces

- Controlling randomness
  - Say you expect interface X to be busy 50% of the time
  - Set likelihood of transaction X to 0.50
  - Setting transaction X likelihood to 0.0 => ?
  - Setting transaction X likelihood to 1.0 => ?
What should you randomize?

- Device Configuration
- Environment Configuration
- Input Data
- Protocol Exceptions
- Errors and Violations
- Delays
Testbench Desiderata

- Constrained-random stimulus
- Functional coverage
- Layered testbench
- Reuseable testbench
- Separate test specific code
• **Reference state**
  - All the “data structures” implemented in hardware
  - FIFO would be a queue, Register file is an array
  - CAM is an associative array
  - Pipeline stages are array elements!

• **Checkers**
  - Compare outputs
Incremental Validation Strategy

Test Everything Randomly

Test Functionality

Test Data Interfaces

Test Control Interfaces

Test Clock and Reset

Coverage
Step 1: Testing Clock and Reset

- Run an empty clock with reset asserted high for a few cycles
- When reset is high check the values for output control interfaces
- Almost instantly start validation

```plaintext
max_cycles 5000

density_reset 0.002 // between 0 and 1

density_read 1.0 // between 0 and 1
density_write 0.0 // between 0 and 1
density_search 0.0 // between 0 and 1

index_mask_read 0x01 // 5 bit number in hexadecimal format
index_mask_write 0x01 // 5 bit number in hexadecimal format

data_mask_write 0x01 // 32 bit number in hexadecimal format
data_mask_search 0x01 // 32 bit number in hexadecimal format

auto_configure 0 // can be 1 or 0
```
2: Testing input control interfaces

• Set up controls in the test generator to turn on/off individual control signals for all of the interfaces

• For the CAM:

```
max_cycles 5000

density_reset 0.002 // between 0 and 1

density_read 1.0 // between 0 and 1
density_write 0.0 // between 0 and 1
density_search 0.0 // between 0 and 1

index_mask_read 0x01 // 5 bit number in hexadecimal format
index_mask_write 0x01 // 5 bit number in hexadecimal format

data_mask_write 0x01 // 32 bit number in hexadecimal format
data_mask_search 0x01 // 32 bit number in hexadecimal format

auto_configure 0 // can be 1 or 0
```
3: Test functionality

- Set up possible data ranges for the input interfaces

```c
max_cycles 5000

density_reset 0.002 // between 0 and 1

density_read 1.0 // between 0 and 1
density_write 0.0 // between 0 and 1
density_search 0.0 // between 0 and 1

index_mask_read 0x01 // 5 bit number in hexadecimal format
index_mask_write 0x01 // 5 bit number in hexadecimal format

data_mask_write 0x01 // 32 bit number in hexadecimal format
data_mask_search 0x01 // 32 bit number in hexadecimal format

auto_configure 0 // can be 1 or 0
```
4: Test corner cases explicitly

- CAM: All data in the CAM is the same!

```plaintext
max_cycles 5000

density_reset 0.002 // between 0 and 1
density_read 1.0 // between 0 and 1
density_write 0.0 // between 0 and 1
density_search 0.0 // between 0 and 1

index_mask_read 0x01 // 5 bit number in hexadecimal format
index_mask_write 0x01 // 5 bit number in hexadecimal format

data_mask_write 0x01 // 32 bit number in hexadecimal format
data_mask_search 0x01 // 32 bit number in hexadecimal format

auto_configure 0 // can be 1 or 0
```
5: Finally

- Excite all interfaces randomly in any combination
- And pick values for the configuration parameters randomly!

```plaintext
max_cycles 5000

density_reset 0.002 // between 0 and 1
density_read 1.0 // between 0 and 1
density_write 0.0 // between 0 and 1
density_search 0.0 // between 0 and 1

index_mask_read 0x01 // 5 bit number in hexadecimal format
index_mask_write 0x01 // 5 bit number in hexadecimal format

data_mask_write 0x01 // 32 bit number in hexadecimal format
data_mask_search 0x01 // 32 bit number in hexadecimal format

auto_configure 0 // can be 1 or 0
```
Validation Strategy Summary

- Test all interfaces (individually and in combinations)
  - System interface: Reset, Clock
  - Input control interfaces, on/off signals
  - Input data interfaces; ranges of values, subset of values
- Test all functionality provided by the unit
  - Allow all functions to be tested individually
  - Allow all functions to be tested in groups
- Test corner cases
  - Cases specified by the designer during the specification
  - Things that tanked the design the last time 😊
- Also do this
  - Interface, functionality, errors, to be randomly interleaved
  - Interleaved with no work, and electrically bad inputs
Some Other “Patterns” for Validation

- **Backpressure from the DUT**
  - *Transaction generator and transactor have to be modified*

- **Sometimes 3rd party tools do not have visibility**

Validation Sign-off

- **Code coverage**
  - Check if all the lines of source code have been covered
  - `vcs -cm +line+cond+fsm+tgl+path *.sv`

- **Event coverage**
  - Check if all events of interest have been covered
  - E.g., `pipestall X in pipestage Y has been covered N times`
  - Include non-synthesizeable, removable code to check

- **Interface coverage**
  - All interfaces have been sufficiently stressed
  - By construction this should be true barring test bench bugs
  - Check using `covergroup` construct (Chapter 9, SVV book)

- **Performance coverage**
  - Make sure the unit meets the performance goals

- You will do all four in Lab 2.
Lifecycle of a Bug

- Every little weirdness is a lurking symptom.
- Bugzilla demo

1. Enter New Bug (Reporter)
2. Assign or Accept (Component Owner)
3. Resolve or Fix (Component owner)
4. Verify Fix (Reported or Verif Team)
5. Close (Reporter)

Re-open (optional)
Bug Rate Over Time

- Simulation Cycles
- Design Coding
- Testbench Devel
- Bug Rate
- Tapeout!
System Verilog Test Bench Demo