Hewlett Packard
Enterprise

27th Jan 2016

Verification Challenges

Verification Futures 2016

Jon Buckingham

Challenges

1. Coverage from FPGA Prototypes
2. Real System Use Verification
3. Catching All the Show Stoppers

Coverage from FPGA Prototypes

The situation

- **FPGA prototypes are expensive**
  - Engineering time for FPGA, PCA, Software
  - Part costs
- **Typically used for**
  - Accelerating Software turn on
  - Finding ASIC defects early
- **Anticipated extra verification**
  - Large counter rollover
  - System level scenarios
  - “Doing so much more” with the faster run times: large volumes of data/traffic/scenarios

Coverage from FPGA Prototypes #2

The challenge: to find defects

- Does the software used emulate real system use?
  - Is it production code or special test code?
- How can we measure coverage?
  - Are large test runs delivering new coverage?
  - What about synthesising *cover* properties (including *assertion preconditions*)?
- Can we find defects in time – before netlist release?
  - Needs to be working very early to be beneficial in finding defects
- How different are the compromises between FPGA and ASIC?
  - Performance
  - Interaction with other system components
  - Different IP
  - Different RTL due to FPGA synthesis
- Debug can be difficult!

Getting good return on investment is challenging!

Real System Use Verification

- **Constrained random is hard**
  - Separate verification teams
  - Methodologies and languages are complex
  - Test benches for random with good coverage is a very challenging design problem
- **Error scenarios are hardest**
  - Random errors
  - Abort and restart
  - Does it work ok afterwards – did all your scoreboards clean up and restart properly?
- **Do we know the system architecture well enough**
  - Are documentation methods good enough?
    - Ambiguous? Too wordy? Unreadable?
  - How do we write system-wide properties?
    - Are PSL, SVA practical for this?
- **Whilst doing all the above,**
  - Does the verification team truly understand the system?
  - What about asynchronous interactions within the system (especially during aborts and subsequent bring ups)
  - Can deadlocks occur in the system?

Catching All the Show Stoppers

Finding all the defects that will force a respin

- **System Level Defects**
  - Know system requirements: find system level defects
  - Are we spending too much time measuring detailed coverage?
  - OK, the specs are verified, but are the specs correct?
  - Are we stimulating and recovering from all error situations – especially with asynchronous system components
    - Random helps a lot, but only if well designed
- **Are the RTL design languages good enough?**
  - Verilog/SystemVerilog too loosely defined?
  - Are EDA tools consistently interpreting them?
- **Other tools**
  - How much gate level simulation should we do?
  - Are linting rules good enough?
  - Are all those synthesis warnings really ok?

Catching All the Show Stoppers #2

People and Teams

- **This is also a “human/team” problem**
  - System knowledge and experience of team members is critical
  - the tools are only part of the solution
- **Design quality in**
  - Testing quality in later is **so** much harder
  - Beware relying too much on state of the art verification methodologies – give designers time and tools too!
- **Keep Design and Verification teams together**
  - Verification engineers get to understand the requirements (not just the spec)
  - Designers get to review test plans and coverage
  - Designers should review tests, and also write some
    - “e” sequences aren’t that hard!

Backup Slides

Backup Slides: System Use and specification

Lifecycle, operations and scenarios – (*Fusion method)*

```
  Reset
.  (    (  configure
          . Activate
          . SystemTest
        )
      . (   ReceiveStuff*
          || TransmitStuff*
          || etc
        )

    )*
```

operation: *SystemTest*

1. Write reg SYSTEM_TEST_TYPE = test1
2. Write reg SYSTEM_TEST_CTRL.go = 1
3. Wait for interrupt SYSTEM_TEST_IRQ

etc

Can write cover properties for each operation.
Can write asserts to check operations occur in correct sequences.