Lecture 4: Dynamic Verification

Environment – a common methodology

Logic Design Automation for VLSI 046880

- We learned about key principles of automation, structure of the design and about logic simulators

- Today’s we learn
  - How to validate the Functionality of the RTL model?
  - How to deal with the design exponential complexity?
  - We describe a clever methodology for doing so
    - By introducing its concepts
  - And some challenges and technologies to help

This lecture was written with Dror Bohrer, Senior Director of Chip Design, of Mellanox Technologies 2013
Different Kind of Verification

- Verifying the specification
  - “is what I think I want - what I really want?”
  - Some times called VALIDATION

- Verifying an implementation
  - “is this design consistent with the specification?”

- Verifying implementation quality
  - “is there something suspected as unreliable, inefficient, or otherwise undesirable in this design?”
  - “was design methodology adhered to?”

- Manufacturing test
  - “is this particular unit a good one or a bad one?”
This Lecture is about: Dynamic Logic Verification

Dynamic Logic Verification is

- The set of methods, tools, models (a.k.a. code) that stimulates a Device Under Test (DUT) *Functional Model* and verifies it responds according to spec
- In contrast to testing the manufactured device…

Like spice simulation but

- Spice calculates the analog waveforms of voltages and currents
- Dynamic Logic Verification – uses digital abstraction
- Spice (with reduced accuracy) may model ~100K transistors
- Functional model represent entire device as large as it gets
The Device Complexity Challenge

- Number of transistors per VLSI devices keep doubling
- Logic complexity doubles every ~2-3 years
- An example project (@40nm)

<table>
<thead>
<tr>
<th>Measureable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of RTL code lines</td>
<td>876,414</td>
</tr>
<tr>
<td>Number of Verification Environment code lines</td>
<td>709,031</td>
</tr>
<tr>
<td>Number of functional description items (features)</td>
<td>2,200</td>
</tr>
<tr>
<td>Number of defined tests</td>
<td>300</td>
</tr>
<tr>
<td>Number of tests run nightly</td>
<td>25,000</td>
</tr>
<tr>
<td>Number of bugs found</td>
<td>2,341</td>
</tr>
</tbody>
</table>
The Device State Transitions Space

- Assume the device is a set of state machines
  - True for synchronous design
- One way to define the function of a device is to express transition conditions from one state to the other
- Number of possible states $2^{\text{num-ffs}}$
- Number of state transitions EXPLOADS
- Not enough runtime to cover every transition

Space of State Transitions
Our Example: A System of Elevators

- I2C is a bus of 1 data and 1 clock lines (operates ~400KHz)
- Supports multiple masters, addresses and read/write transaction
State Transition Space Example

- What is the state space of the Elevators Brain?
- Assume 4 elevators and 6 floors
- First we count states:
  - Keep track of each elevator current floor – $6 \times 4$
  - Keep track of each elevator requests – $2^6 \times 4$
  - Keep track of each floor requests (up/down) – $2 \times 6$
- So state space size is
  - $6^4 (2^6)^4 2^6 = 3^4 2^{34}$
- Are these states cross connected?
  - Yes / but not ALL
    - The decision of what elevator need to go where is affected by all
    - But some state transitions are impossible. For example same Elevator Levels
Verification Environment Components

- Generators
  - Generates the specific test(s)
- Stubs
  - Hide test environment complexity
- Injectors
  - Feed the test to the DUT – hides device specific
- Monitors/Watchers/Assertions
  - Inspect the external (internal) model signals
  - Classify what is going on
- Verifiers
  - Make sure expected results are reached
- Coverage Analyzers
  - Tracks what happened at a high level
TEST GENERATION

Diagram:
- Gen
- STUB
- Injector
- DUT
  - Assert
- Monitor
- Coverage
- Verifier
Directed Tests

- A specific scenario and stimuli
- Benefits
  - Aimed to cover particular “corner” case
  - For example:
    - A well known “constant”
    - Exhausting specific hardware resource
    - Exercise specific condition
- Limitations
  - Focus on where designers think the bug is. I.e. miss most of them
  - Can’t create complex state interactions
- Directed tests cover “dots” on the verification space!
Random Testing

- Instead of “hard coding” specific scenario – randomize it
- Each “system input” have a valid range
- Some inter-relations of inputs may have dependency too
- So a random test is:
  - A \textit{LEGAL} random generation of stimuli for the system
  - When run multiple times, covers a space around some points

- A perfect random test environment may cover entire space if ran infinite number of times… (Ergodic system)
A Single Test

- What is better?
  - Each test randomizing a different aspect of the possible stimuli
  - A single monumental test – randomizing all aspects?

- From complexity point of view the former
- From coverage point of view the latter

*All possible interactions may be tested only when all possible stimuli are randomized together*
Good-Flow / Bad-Flow

- Good-Flow:
  - When the device is performing its “normal” flow of things

- Bad-Flow:
  - When the device has to do some “exception” flows (bad data)

- DUT “Linearity” assumption:
  - Imposing randomness on multiple parameters has small cross-product – i.e. features do not contradict each other
  - This assumption does not hold for “bad flow” in most systems
    - Why?
    - Since hardware resources to treat bad-flows are usually minimized
HIERARCHY
Testing at Various Levels of Hierarchy

- Unit, Cluster and Chip-Level Verification
  - Unit – a basic logical partition
  - Cluster – A collection of Units
  - Chip Level – the full system

- Why?
  - Smaller = easier and faster = better coverage
  - But unit test has issues of environment correctness
    - In most cases the unit is over-stressed
    - Why? Since the chip imposes constraints on the inputs
    - E.g. rate of changes on parallel BRAIN inputs vs. I2C
Why Full-System?

- Initialization
  - Due to **reset** de-assertion “staging” or lack of those
  - Boot sequence dependencies
  - X propagation (and X loops) at the top level
  - Where many bugs are … Why? Humans need to communicate.

- Performance modeling
  - Especially when a central resource is shared

- Power consumption
  - Why? Normally not all the machine is active at the same time.
    - E.g. error handling
INJECTORS
Most Dynamic Verification Environments provides an efficient method to drive Device Under Test (DUT) interface signals.

Injectors simplify the details of DUT signals exposing a generic set
- E.g. Allows same verification IP to be used on different devices
- Act as a user friendly “Driver” for the Generator

Injecting to internal signals?
- A “clever way” to hide bugs – don’t do it!
- But sometimes required to overcome test initialization time
  - When? Memory, Arbiters, bad states …
Transactional Level Models (TLM)

- Hides the details of DUT interface protocols
- Instead of exercising each bus signal to perform the protocol, tests are defined as bus “transactions”

In our example I2C:
- A TLM transaction of Write(B) translates to the below sequence

![I2C Transaction Diagram]
White box Black box Testing …

- **Black Box Testing** = only utilize DUT interface
- **White Box Testing** = look inside = use some state
- **When to use what?**
  - **De Jure** – Black box is enough and closer to device usage
    - Why would you need to know how does the device work?
  - **De Facto** – Many tests are easier to describe in “internal” terms
    - For example “Arbitration Fairness”, “Performance”,

Assertions

- Captures designers assumptions
  - Makes the code readable to others
  - Protect against “human interface” bugs

- Evaluated with the model – during simulation
  - Errors are thrown when assertions not met

- Many times – over constraint
  - Failed assertion does not mean the model will fail

- Example – Elevators Core Algorithm may assume
  - No out of range floor numbers are passed
  - “Stop at Floor” request arrive to the brain – one at a time
COVERAGE
What is Coverage?

- Answers: “How much of the state transitions space was tested?”
- At the architecture level
  - Function – a particular feature exercised
    - Requested floor was cancelled -> don’t stop
  - Event – a combination of feature and state
    - Request floor was cancelled but it was also a floor that “called” the elevator
- At the code level
  - Expression – exact sub expression evaluated to specific value
    - Including “states” of a machine
  - Line
  - Block (between branches or jumps)
  - Path – the combinations of the above
Feature Plan and Functional Coverage

- Feature plan
  - The full set of the device functional and performance features
  - The “plan” provides order (and dates) for covering each feature

- Test plan
  - By what means are these features tested

- When can you say a feature is covered?
TECHNOLOGY
Constraint Solvers

- Devices operate in a complex but constrained inputs
  - E.g. Ethernet switch should handle all types of legal traffic…

- Random test generators should impose these relationships and data dependency to produce legal data

- Once “legal” is defined the constraint solver should randomize as much as possible but apply the “legality” constraints

- SAT solvers used will be described later in our course
Constraint Solving Example

- While generating tests for the Elevators Brain
  - The Elevator Controller never cancels non outstanding request
  - Similarly it does not send double requests

- These properties of the Elevator Controller immediately translates into test generation constraints

- Do you have more constraints for our example system?
Aspect Programming - AP

- A generic concept in Object Oriented Programming
- Best to maintain cross-cutting concerns
  - Other functionalities overlay on the functional model. E.g. logging, assertions, performance, power model, etc...

Aspect Programming Terminology:

- **Advice**
  - The actual code that handles specific concern
- **Pointcut**
  - The point of execution at which cross-cutting concern are applied
- **Aspect**
  - The combination of the pointcut and the advice
Aspect Programming in Verification

- Given a DUT model and its environment
- Describe each feature as a concern on its own plane
- Attach to the relevant objects as advices
- Result in robust independent verification environment
- Which of our 4 DA principles is AP matching?

- VERA (Synopsys) and “e” (Cadence) are the de-facto languages used

Further reading:


Job Queuing Systems

The problem
- Dynamic Verification space is huge
- A single random test covers a *single point* in this space
- How many tests are required?

The solution
- Batch queuing systems over computer farms
  - One of the first Cloud applications…
- Verification environments indeed simulate millions of tests
  - 10K nightly or more
Future of Dynamic Verification

- Coverage driven random test generation
  - Influence random test generation towards missing events

- Bug injection
  - Test the verification environment – how many are found?

- Performance testing
  - Automatically generate performance bugs

- Power testing
  - Automatically generate max power viruses/tests
In Contrast: Formal Model Checking

- What if we could ask questions like:
  - Is every elevator request handled?
  - Aren’t two elevators sent to same request?
- These are Properties of our design
- A huge number of simulations required to verify them

- Formal Model Checking and Theorem Proving can either
  - Provide a counter example
  - Prove they are correct
Summary

- Dynamic verification faces a complexity challenge
- Verification Environment methodology is
  - Based on a holistic random test approach
  - Coverage is the metric
  - An extensive environment
  - Relies on abstraction, hierarchy and reuse

- Next steps: Static Verification – Formal Model Check
  - Is an alternative
  - Can it really cover the state and functional space?
  - To be continued…