IP Core Design

Lecture 10
Property/Assertion-Based Verification

Juinn-Dar Huang, Ph.D.
Assistant Professor
jdhuang@mail.nctu.edu.tw

October 2004
Outline

• Verification challenge
• Introduction to PBV
• Property languages
• PBV-enabled applications
• Conclusion
Verification Challenge (1/3)

- Design productivity has risen 10x since 1990 due to synthesis
- Effort to verify new designs doubles every 6-9 months

The challenge of functional verification is getting bigger
Verification Challenge (2/3)

- 50% of chips require 1 or more re-spins
- 74% of re-spins are due to **functional defects**

![Chart showing reasons of re-spins](image)
It is reported that …

- Average 1.5 functional bugs per 200 lines in RTL
  - 1500 bugs in a 200K-line design
- Average 1.5 man-day required to find a bug
  - 2250 man-days required for a 200K-line design

Do you find all bugs before tape-out?
How do you find bugs quicker and easier?
A USD$ 475 Million Bug

• Intel’s Pentium Bug in 1994
  – 3.3 million transistors on chip
  – 1 trillion $10^{12}$ cycles were simulated before tape-out
  – a bug in floating-point division hardware was revealed
  – fixed just by adding 5 more transistors

• Intel offered replacements to users
  – $475 Million USD penalty at Q4’94

Simulation after Simulation … Endless Simulation

Besides simulation, what can you do more?
Today’s Functional Verification

- Verification starts late
- Simulation process scales, but quality of coverage does not scale
- Low controllability
  - verification vectors are not enough and effective
- Low observability
  - internal errors may not propagate to outputs
- Low reusability

*Verification capacity lags far behind design capacity*
Today’s Real Verification Challenge

• An ad-hoc verification approach might not work effectively no longer

• New methodology is required to provide the equivalent of “synthesis breakthrough for design”
What We Really Need Is …

A reusable and scalable verification methodology
• Increase controllability and observability
• A more effective block(IP)-level verification
• Verification reuse
  – from block-level to system-level
  – from IP developer to IP integrator
  – from design to design

⇒ Property-Based Verification (PBV)
Outline

- Verification challenge
- **Introduction to PBV**
- Property languages
- PBV-enabled applications
- Conclusion
Property-Based Verification (PBV)

• Property
  – a statement about a designer’s intended behavior

• Property-based verification
  – to ensure consistency between the designer’s intention and what he/she creates
  – to guard the design properties

“The 8-bit input should range from 0 to 200.”;
“After req raises, gnt is expected within 10 clock cycles.”

In fact, property is the real soul

Historically, it is named Assertion-Based Verification (ABV)
Neither New Nor Innovative Idea

• The property/assertion concept has been widely used in software development for decades

```c
#include <assert.h>

char hex_to_char(unsigned int num)
{
    assert(num <= 15);
    if(num < 10)
        return num + '0';
    else
        return num - 10 + 'A';
}
```

Writing Solid Code, Microsoft Press, 1993
Topics of PBV

• What’s are the benefits of PBV?
• How to describe properties?
  – OVL, PSL, and SVA
• How to ensure properties are guarded?
  – dynamic simulation with assertions
  – constraint-driven stimulus generation
  – static formal verification (model checking)
  – semi-formal verification (dynamic + static)
Benefits of PBV (1/2)

- Capture design intent succinctly
  - executable comments
- Improve observability
  - enable white-box verification
    - easier to find deeply embedded bugs
- Improve controllability
  - more effective vectors by constraint-driven stimulus generator
  - enable formal technology
- Improve verification efficiency
  - shorter debug time (50%, Abarbanel et al., IBM, CAV2000)
Benefits of PBV (2/2)

• Specify once, use in many tools and flows later
  – dynamic simulation
  – constraint-driven stimulus generation
  – static formal property verification
  – semi-formal property verification
  – ...

• Verification reuse
  – properties can be reused
    • block to system
    • developer to integrator
    • design to design
Outline

- Verification challenge
- Introduction to PBV
- Property languages
- PBV-enabled applications
- Conclusion
Writing Properties

- Pragma-based method
  - a.k.a. directive-based or comment-based
    - // company_abcs fsm one_hot, // company_xyz always(a == !b)

- Property template library in native HDL
  - Open Verification Library (OVL) in Verilog and VHDL

- Property languages
  - Property Specification Language (PSL)
  - SystemVerilog Assertion (SVA)
Open Verification Library (OVL)

- An open-source library of property templates
  - 31 templates available in the latest release
- Templates in HDLs (both Verilog/VHDL available)
  - no new tool required (an existing simulator is enough)
  - virtually no further learning required
- Free download from www.verificationlib.org
  - hosted by Accellera

<table>
<thead>
<tr>
<th>Property</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert_always</td>
<td>assert_one_hot</td>
</tr>
<tr>
<td>assert_never</td>
<td>assert_odd_parity</td>
</tr>
<tr>
<td>assert_proposition</td>
<td>assert_even_parity</td>
</tr>
<tr>
<td>assert_delta</td>
<td>assert_range</td>
</tr>
<tr>
<td>assert_decrement</td>
<td>assert_increment</td>
</tr>
<tr>
<td>assert_no_overflow</td>
<td>assert_no_underflow</td>
</tr>
<tr>
<td>assert_zero_one_hot</td>
<td>assert_one_cold</td>
</tr>
<tr>
<td>assert_change</td>
<td>assert_unchange</td>
</tr>
<tr>
<td>assert_time</td>
<td>assert_handshake</td>
</tr>
<tr>
<td>assert_frame</td>
<td>assert_implication</td>
</tr>
<tr>
<td>assert_window</td>
<td>assert_transition</td>
</tr>
<tr>
<td>assert_no_transition</td>
<td>assert_win_unchange</td>
</tr>
<tr>
<td>assert_win_change</td>
<td>assert_quiescent_state</td>
</tr>
</tbody>
</table>

...
OVL Example

Property Template

assert_frame #(severity_level, min, max, flag, opt, msg)
inst_name(clk, rst_n, start_event, test_expr);

Property Instance

assert_frame #(0, 3, 7) check_req_ack(clk, rst_n, req, ack);
Property Languages (1/2)

- Proposals submitted to Accellera
  - IBM’s Sugar, adopted as the standard, April 2002
  - Intel’s ForSpec
  - Verisity’s Temporal e
  - Motorola’s CBV (Clock-Based Verification)

  ➔ Property Specification Language (PSL)

- SystemVerilog Assertion (SVA)
  - along with SystemVerilog 3.1
Property Languages (2/2)

- The battle of property languages is still hot
- Today, the major battle is between PSL and SVA
- OVL is complementary
  - less powerful;
  - however, virtually no cost in dynamic PBV
Taste Different Flavors

• In English
  “If \texttt{rst\_n} is deasserted, \texttt{ack} is expected within next 2\textasciitilde5
  \texttt{clk} cycles after \texttt{req} raises.”

• In OVL
  \texttt{assert\_handshake \#(0,2,5) hs\_1(clk, rst\_n, req, ack);}

• In PSL
  \texttt{assert always (req -> \{ack\} within [*2:5] abort !rst\_n) @rose(clk);} 

• In SVA
  \texttt{always @(posedge clk) disable iff !rst\_n}
  \texttt{assert property ( req |=> ##[2:5] ack );}
Common Fallacies of Properties (1/2)

- I don’t have time to write properties. I’ve to get my design done
  - If you write comments, you should write properties
- I’ll spend more time debugging the properties than my code, therefore, it’s really a burden
  - You mean you don’t debug your testbench?
- I cannot think of any properties to put in my code
  - Just learn it
  - Does your design need no simulation?
- Dynamic PBV slow down simulations
  - That’s true (15%-30%).
  - However, debugging takes more verification time and assertions significantly reduce the debugging time
Common Fallacies of Properties (2/2)

- Designers should not check their code. Thus, writing properties violates this rule.
  - Actually, RTL and properties are two different formats to describe the same design intent
    - double confirmation, easier to catch bugs
  - Dedicated verification engineers
    - usually focus on interface/feature verification (black-box verification)
    - hardly care about the implementation details
  - Software engineer has found assertions very useful
Where Are Properties?

• Design assumptions
• Legal states/transitions of FSMs
• Allowed FIFO operations
• Range checking
• Permissible bus req/ack/transfer sequences
• Illegal/Undefined commands/operations
• …
Outline

- Verification challenge
- Introduction to PBV
- Property languages
- **PBV-enabled applications**
- Conclusion
Dynamic Simulation (Dynamic PBV)

All properties are treated as assertions during simulation.
Increase observability and facilitate *functional coverage*.
Simulation without Internal Assertions

- Without internal assertions (black-box verification)
  - bugs must propagate to an output
  - backward traversal to find the bug is required even if you find an error at an output

Diagram:

- Poor Observability
Simulation with Internal Assertions

• With internal assertions (white-box verification)
  – bugs no longer have to propagate to outputs
  – assertions take you closer to the source of the bug
**Power of Assertion (1/3)**

DECS Alpha 21164 project [Kantrowitz et al., DAC 1996]

**Assertion Checkers**

- Cache Coherency Checkers: 9%
- Reference Model Comparison
  - Register File Trace Compare: 8%
  - Memory State Compare: 7%
  - End-of-Run State Compare: 6%
  - PC Trace Compare: 4%
- Self-Checking Test: 11%
- Manual Inspection of Simulation Output: 7%
- Simulation hang: 6%
- Other: 8%
# Power of Assertion (2/3)

DEC Alpha 21264 project [Taylor et al., DAC 1998]

<table>
<thead>
<tr>
<th>Assertion Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion Checker</td>
<td>25%</td>
</tr>
<tr>
<td>Register Miscompare</td>
<td>22%</td>
</tr>
<tr>
<td>Simulation &quot;No Progress&quot;</td>
<td>15%</td>
</tr>
<tr>
<td>PC Miscompare</td>
<td>14%</td>
</tr>
<tr>
<td>Memory State Miscompare</td>
<td>8%</td>
</tr>
<tr>
<td>Manual Inspection</td>
<td>6%</td>
</tr>
<tr>
<td>Self-Checking Test</td>
<td>5%</td>
</tr>
<tr>
<td>Cache Coherency Check</td>
<td>3%</td>
</tr>
<tr>
<td>SAVES Check</td>
<td>2%</td>
</tr>
</tbody>
</table>
Want more?

- 17% of bugs were found by assertions on Cyrix M3 (p1) project
- 50% of bugs were found by assertions on Cyrix M3 (p2) project
- 85% of bugs were found by over 4000 assertions on HP [Foster et al., HDLCon2000]
- 400 bugs were found from formal proofs of assertions on Intel Pentium 4 project [Bentley, DAC2001]
Property/Assertion Reuse

- Assertions developed in IP block-level can be reused
  - from block-level to system-level
  - from developer to integrator
  - from design to design
PBV Enables New Applications

- Constraint-driven stimulus generation
  - improve controllability
- Formal property checking
  - scale to large capacity by limiting the search space
- Semi-formal verification
  - dynamic simulation + static property checking
- Property-based functional coverage
Interface Property vs. Constraint

Property: mem_adr cannot change during a read cycle

always (rose(mem_rd) -> next (stable(mem_adr) until rose(mem_done))

An output property of Block A can be an input constraint of Block B
Constraint-Driven Stimulus Generation

Stimulus Generator

- `mem_rd`
- `mem_adr`
- `mem_done`

Simulation paths guided by constraints

Block B (DUV)

**Assume**
always (\(\text{rose}(\text{mem}_\text{rd})\) -> next (\(\text{stable}(\text{mem}_\text{adr})\) until \(\text{rose}(\text{mem}_\text{done})\)))

Constraint-driven random stimulus generation based on
*input constraints*

\(\Rightarrow\) No illegal input vectors

*Not blindly random!*
Static Formal Verification (Static PBV)

Formally verify whether Block B’s internal properties are preserved

search space limited by constraints

Block B (DUV)

legal states

all states

Constraints specify the valid search sub-space

Smaller search space!
Handle larger designs!

No input vector required!

Stimulus Generator

assume always (rose(mem_rd) -> next (stable(mem_adr) until rose(mem_done)))

mem_rd

mem_adr

mem_done
Dynamic vs. Static (1/2)

- **Reachable State Space**
- **Starting States**
- **Simulation (Dynamic PBV)**
- **Bugs triggered with simulation**
- **Model Checking (static PBV)**
  - Explore all possible states
Dynamic vs. Static (2/2)

- **Static ABV**
  - (hard to find bugs in simulation)
  - (start sooner!!)

- **Dynamic ABV**
  - (based on multiple studies)
  - 50% Reduction in Debug

- **Simulation (NO assertions)**
  - As published by IBM in CAV 2000
Semi-Formal Verification (Dynamic + Static)

Take a simulation snapshot as an initial state
Apply formal verification with a limited radius
Functional Coverage

• Assertions in dynamic simulation
  – detection of prohibited behavior
    • true or false (是非題)
    • e.g., cannot grant to more than 1 request, or the design breaks
    • usually named as (assertion) checkers
  – coverage of expected behaviors
    • multiple desired manners (多選題)
    • e.g., Do all types of bus transactions (single, continuous burst, 4-beat burst, 4-beat wrap-around, …) occur?
    • usually named as (functional/protocol) monitors
    • help complete the verification of lower-level implementation details
Properties Grouped as Verification IP (VIP)

• Properties can be used
  – to guard internal design behaviors in white-box verification
  – to guard external interface behaviors in black-box verification

• VIP can save your efforts for writing properties for
  – standard communication/interface protocols
    • AMBA, AXI, PCI, PCI-E, USB, USB-OTG, 1394, …

Verification Reuse!
Automatic Assertion Extraction

• Some assertions can be automatically extracted from RTL
  – semantic consistency check
    • full_case and parallel_case
    • X-assignments
  – structural consistency check
    • bus contention and floating (one-hot property)
    • simultaneous set/reset
    • dead-code (if/case branch reachability)
    • clock-domain crossing (metastability)
PBV Framework

Custom Properties
Existing Libraries
Verification IPs

Design-Specific Property Specifications

Simulation
Stimulus Generation
Formal
Semi-Formal

Supported by many EDA vendors
(e.g., PSL support @ www.pslsugar.org)

Single property specification drives multiple verification tools
Outline

- Verification challenge
- Introduction to PBV
- Property languages
- PBV-enabled applications
- Conclusion
Conclusion

• PBV enables new applications
  – constraint-driven stimuli generation
  – formal property checking
  – semi-formal verification

• PBV enables true verification reuse
  – specify properties once, use them ever after
    • various tools and flows
    • block to system, developer to integrator, and design to design

• PBV increases verification quality and productivity

*Specify properties of your designs today!*
Reference Information

- “Principles of Verifiable RTL Design”, 2nd Ed., L. Bening and H. Foster
- “Assertion-Based Design”, 2nd Ed., H. Foster et al.
- IBM Sugar
- PSL LRM (Version 1.1)
  - www.eda.org/vfv/docs/PSL-v1.1.pdf
- SystemVerilog LRM (Version 3.1)
- Open Verification Library (OVL)
  - www.verificationlib.org
- Lots of articles on assertion and ABV in www.eedesign.com