Verification Trends

- The Verification Gap
  - Many companies still using 1990’s verification technologies
  - Traditional verification techniques can’t keep up

Based on data from the 2003 ITRS study and Collett International 2004 FV Survey
The Verification Gap
Directed Test
State-of-the-Art Verification Circa 1990

- Imagine verifying a car using a directed-test approach
  - Requirement: Fuse will not blow under any normal operation
  - Scenario 1: accelerate to 37 mph, pop in the new Lady GaGa CD, and turn on the windshield wipers

A Few Weeks Later. . . .
The Verification Gap

Directed Test

State-of-the-Art Verification Circa 1990

- Imagine verifying a car using a directed-test approach
  - Requirement: Fuse will not blow under any normal operation
  - Scenario 714: accelerate to 48 mph, roll down the window, and turn on the left-turn signal

Concurrency Challenge

- A purely directed-test methodology does not scale
  - Imagine writing a directed test for this scenario!
  - Truly heroic effort—but not practical
Concurrent Challenge

Packet-Based Design

Transaction Layer Packets
Data-Link Layer Packets
Retry Buffer
Arbiter

From Fabric → Tx
Rx → From Rx Channel

ABV Industry Adoption

Assertions

Property Checking

0% 20% 40% 60% 80% 100%
37%
19%
Published Data on Assertions Use

<table>
<thead>
<tr>
<th>Assertion Monitors</th>
<th>34%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Coherency Checkers</td>
<td>9%</td>
</tr>
<tr>
<td>Register File Trace Compare</td>
<td>8%</td>
</tr>
<tr>
<td>Memory State Compare</td>
<td>7%</td>
</tr>
<tr>
<td>End-of-Run State Compare</td>
<td>6%</td>
</tr>
<tr>
<td>PC Trace Compare</td>
<td>4%</td>
</tr>
<tr>
<td>Self-Checking Test</td>
<td>11%</td>
</tr>
<tr>
<td>Simulation Output Inspection</td>
<td>7%</td>
</tr>
<tr>
<td>Simulation Hang</td>
<td>6%</td>
</tr>
<tr>
<td>Other</td>
<td>8%</td>
</tr>
</tbody>
</table>

Kantrowitz and Noack [DAC 1998]

<table>
<thead>
<tr>
<th>Assertion Monitors</th>
<th>25%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Miscompare</td>
<td>22%</td>
</tr>
<tr>
<td>Simulation “No Progress”</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>

Taylor et al. [DAC 1998]

- 17% of bugs found by assertions on Cyrix M3(p1) project [Krolnik ‘98]
- 50% of bugs found by assertions on Cyrix M3(p2) project [Krolnik ‘98]
- 85% of bugs found using over 4000 assertions on an HP server chipset project [Foster and Coelho HDLCon 2001]
- Thousands of assertions in Intel Pentium project [Bentley 2001]
- 10,000 OVL assertion in Cisco project [Sean Smith 2002]

Sun
Assertion-Based Verification of a 32 thread SPARC™ CMT Processor [Turumella, Sharma, DAC 2008]

<table>
<thead>
<tr>
<th>Category</th>
<th>Unique</th>
<th>Instantiated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-Level</td>
<td>3912</td>
<td>132773</td>
</tr>
<tr>
<td>Interface</td>
<td>5004</td>
<td>44756</td>
</tr>
<tr>
<td>High-Level</td>
<td>1930</td>
<td>18618</td>
</tr>
</tbody>
</table>

Bugs Found Using Assertions

Bugs Found by Type of Assertion

Low-Level
Interface
High-Level
Sun
Assertion-Based Verification of a 32 thread SPARC™ CMT Processor
DAC 2008, Turumella, Sharma

<table>
<thead>
<tr>
<th>Category</th>
<th>Unique</th>
<th>Instantiated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-Level</td>
<td>3912</td>
<td>132773</td>
</tr>
<tr>
<td>Interface</td>
<td>5004</td>
<td>44756</td>
</tr>
<tr>
<td>High-Level</td>
<td>1930</td>
<td>18618</td>
</tr>
</tbody>
</table>

Average Debug Time

ABV Bus-Based Design Example
Using SystemVerilog Assertions (SVA)
**Assertion Building Blocks**

*Specifying A Sequence in SVA*

- **Sequence**
  - Temporal delay `##n` (where `n` is an integer)

```
start   ##1 transfer
```

```
clk      | | | | | |
start    | | | | |
transfer | | |
```

---

```
start   ##2 transfer
```

```
clk      | | | | | |
start    | | | | |
transfer | | |
```
Assertion Building Blocks

**SVA Properties**

- **Properties**
  - Overlapping sequence implication operator |->

```plaintext
ready  ##1  start |-> go  ##1  done
```

- **NOTE:** `start |=> go` is the same as `start |-> ##1 go`

---

SVA Language Structure

- **Properties**
  - Non-overlapping sequence implication operator |=>

```plaintext
ready  ##1  start |=> go  ##1  done
```

**NOTE:** `start |=> go` is the same as `start |-> ##1 go`
Building Blocks
SVA Properties

- Properties
  - Creating named properties

```verilog
class p_valid_transaction;
@ (posedge clk)
    ready #1 start |=> go #1 done;
endclass
```

```verilog
assert property (p_valid_transaction);
```

Nonpipelined Bus Interface
Non-Burst Write Cycle

The Only Legal Behavior!

```
0        1        2        3        4
addr      I         Addr 1
write     /         /
se[0]     /         /
en        |         |
wdata     I         Data 1
```

Notice that !(se[0]==0 & en==1) is not a legal state of the bus!

Non-Burst Read Cycle

The only legal behavior!

```
0        1        2        3        4
addr      I         Addr 1
write     /         /
se[0]     |         |
en        |         |
rdata     I         Data 1
```

Notice that !(se[0]==0 & en==1) is not a legal state of the bus!
### Conceptual Bus State Transitions

- **INACTIVE**
  - sel[0] == 0
  - en == 0
- **START**
  - sel[0] == 1
  - en == 0
- **ACTIVE**
  - sel[0] == 1
  - en == 1

### Bus Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_valid_inactive_transition</td>
<td>Only INACTIVE or START states follows INACTIVE</td>
</tr>
<tr>
<td>p_valid_start_transition</td>
<td>Only ACTIVE state follows START</td>
</tr>
<tr>
<td>p_valid_active_transition</td>
<td>Only INACTIVE or START states follows ACTIVE</td>
</tr>
<tr>
<td>p_no_error_state</td>
<td>Bus state must be valid: (sel[0]==0 &amp; en==1)</td>
</tr>
</tbody>
</table>
Use Modeling Code to Simplify Coding

\[
\begin{align*}
\text{bus_inactive} & = \neg \text{sel}[0] \, \& \, \neg \text{en}; \\
\text{bus_start} & = \text{sel}[0] \, \& \, \neg \text{en}; \\
\text{bus_active} & = \text{sel}[0] \, \& \, \text{en}; \\
\text{bus_error} & = \neg \text{sel}[0] \, \& \, \text{en};
\end{align*}
\]

Formalize Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus legal transitions</td>
<td>Only INACTIVE or START states follows INACTIVE</td>
</tr>
</tbody>
</table>

\[
\text{property p_valid_inactive_transition;}
\]
\[
\text{ @(posedge clk) disable if (bus_reset)}
\]
\[
\text{ ( bus_inactive) } \implies
\]
\[
\text{ ((bus_inactive) || (bus_start))};
\]
\[
\text{endproperty}
\]
\[
\text{assert property (p_valid_inactive_transition);}
\]
Formalize Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus legal transitions</td>
<td>Only ACTIVE state follows START</td>
</tr>
</tbody>
</table>

property p_valid_start_transition;
 @(posedge clk) disable iff (bus_reset)
    ( bus_start ) |=> ( bus_active );
endproperty

assert property (p_valid_start_transition);

Improved Observability

Master
- clk
- rst_n
- sel[0]
- en
- addr
- write
- rdata
- wdata

Slave 0
- I/F
- wdata
- rdata
- addr
- write
- en
- sel[0]
- rst_n
- clk

I/F
Introduction to SVA

- Assertions allow us to specify design intent in a way that lends itself to automation

```
// Assert that the grants for our simple arbiter are mutually exclusive

(grant0 & grant1) // error condition
```
module arbiter (clk, rst_n, req0, req1, grant0, grant1);

always @(posedge clk or negedge rst_n) begin
  if (rst_n != 1'b0)
    if (grant0 & grant1)
      $display("ERROR: Grants not mutex");
endmodule

module arbiter (clk, rst_n, req0, req1, grant0, grant1);

... assert property (@(posedge clk) disable iff (~rst_n) ! (grant0 & grant1));

... endmodule
SVA Language Structure

- **Assertion Units**
  - Checker packaging
- **Directives** (assert, cover)
  - `assert`, `assume`, `cover`
- **Properties**
  - Specification of behavior; desired or undesired
    - How Boolean events are related over time
      - True or false

---

**Boolean Expressions**

`assert property (@(posedge clk) disable iff (~rst_n) !(grant0 & grant1));`

---

**SVA Language Structure**

```
clk  |  |  |  |  |  |  |  |
rst_n
!(grant0 & grant1)

error
```
SVA Language Structure

- SVA contains two types of assertions:
  - *Immediate* and *concurrent*

- Immediate assertions evaluate within the context of a procedural block
  - Similar to an if statement
  - Check to see if a conditional expression holds
    - If not, the verification tool generates an error message

module my_arb (. . .);
  . . .
  always @* begin // arbiter code
  . . .
  if (rst_n)
    assert (!(grant0 & grant1));
  . . .
  end
endmodule
Concurrent assertions evaluate concurrently with respect to other procedural blocks

- Unlike immediate assertions that evaluate during a particular instance in which the line of procedural code containing the assertion is executed

module my_arb (. . . )

always @* begin // arbiter code
  . . .
  end

assert property (@(posedge clk)
  !(grant0 & grant1 ));
endmodule
SVA Language Structure

- SVA provides a mechanism to asynchronously disable an assertion during a reset using the SVA `disable iff` clause

```vcs
assert property (@(posedge clk) disable iff (~rst_n) !(grant0 & grant1));
```

SVA Language Structure

- **Sequences**
  - So far we have examined simple assertions
    - A Boolean expression must hold at every clock
  - Now we introduce SVA sequences
    - Multiple Boolean expressions that are valuated in a linear order of increasing time
SVA Language Structure

- **Sequence**
  - Temporal delay **##n** — where \( n \) is an integer

```
start    ##1    transfer

clk       |

start__    

transfer______
```

SVA Language Structure

- **Sequence**
  - Temporal delay **##n** — where \( n \) is an integer

```
start    ##2    transfer

clk       |

start__    

transfer______
```
SVA Language Structure

- Sequence
  - Temporal delay with range \([m:n]\)
  
  \[\text{start \[\#[0:2]\] transfer}\]

- Consecutive repetition \([^*m]\) or \([^*m:n]\)
  
  \[\text{start[^2] \[\#1\] transfer}\]
SVA Language Structure

- **Sequence**
  - Consecutive repetition [*m] or range [*m:n]

```
start[*1:2] ##1 transfer
```

```
clk  | | | | |
start _____________
transfer __________
```
SVA Language Structure

- **Sequence**
  - Non-consecutive repetition \([=m]\) or \([=m:n]\)

\[
\text{start}[=2] ##1 \text{ transfer}
\]

\[
\begin{array}{c}
\text{clk} \\
\text{start} \\
\text{transfer}
\end{array}
\]

\[
\text{start}[=2] == !\text{start}[0:] ##1 \text{ start} ##1 !\text{start}[0:] ##1 \text{ start} ##1 !\text{start}[0:] ##1
\]

- **Sequence**
  - Goto non-consecutive repetition \([->m]\) or \([->m:n]\)

\[
\text{start}[->2] ##1 \text{ transfer}
\]

\[
\begin{array}{c}
\text{clk} \\
\text{start} \\
\text{transfer}
\end{array}
\]

\[
\text{start}[->2] \rightarrow !\text{start}[0:] ##1 \text{ start} ##1 !\text{start}[0:] ##1 \text{ start}
\]

\[
[^0:] \text{ represents zero to infinity}
\]
SVA Language Structure

- Properties
  - Overlapping sequence implication operator |->

```
ready ##1 start |-> go ##1 done
```

```
assertion property (@(posedge clk) ready ##1 start |-> go ##1 done);
```
SVA Language Structure

- Properties
  - Non-overlapping sequence implication operator
    \[ \Rightarrow \]
    \[
    \text{ready} \ #1 \ 
    \text{start} \ |\Rightarrow \ 
    \text{go} \ #1 \ 
    \text{done} \\
    \]
    
    \[\text{clk} \ |\ 
    \text{ready} \ |\ |\ |\ |\ |
    \text{start} \ |
    \text{go} \ |
    \text{done} \ |
    \]
    
    NOTE: \( A \Rightarrow B \) is the same as \( A \Rightarrow ##1 B \)

- Named properties and sequences
  - To facilitate reuse, properties and sequences can be declared and then referenced by name
  - Can be declared with or without parameters

```sвязь
sequence op_retry;
  (req #1 retry);
endsequence

sequence cache_fill(req, done, fill);
  (req #1 done [=1] #1 fill);
endsequence
```
SVA Language Structure

- Named properties and sequences

```sverilog
sequence s_op_retry;
  (my_req ##1 my_retry);
endsequence
sequence s_cache_fill(rdy, done, fill);
  (rdy ##1 done =1 ##1 fill);
endsequence
assert property ( @(posedge clk) disable iff (!reset_n) 
  s_op_retry |=> s_cache_fill (my_rdy,my_done,my_fill));
```

- Named properties and sequences

```sverilog
property p_en_mutex(en0, en1);
  @(posedge clk) disable iff (~reset_n) 
  ~(en0 & en1);
endproperty
assert property (p_en_mutex(en[0], en[1]));
```
SVA Language Structure

- **System functions**
  - `$onehot (<expression>)`
    - Returns true if only one bit of the expression is high
  - `$onehot0 (<expression>)`
    - Returns true if at most one bit of the expression is high
  - `$isunknown (<expression>)`
    - Returns true if any bit of the expression is X or Z
    - This is equivalent to `^<expression> === 'bx`

SVA Language Structure

- **System functions**
  - `$rose( expression )$
  - `$fell( expression )$
  - `$stable( expression )$
  - `$past( expression [, number_of_ticks] )$
Some assertions require additional modeling code
   – In addition to the assertion constructs

```
// assertion modeling code – not part of the design
ifndef ASSERT_ON
  int cnt = 0;
  always @(posedge clk)
    if (!rst_n)
      cnt <= 0;
    else
      cnt <= cnt + push - pop;
  // assert no overflow
  assert property (@posedge clk disable iff (!rst_n)
    !((cnt + push - pop) > `DEPTH));
  // assert no underflow
  assert property (@posedge clk disable iff (!rst_n) !(cnt + push - pop < 0));
endif
```
Introduction to SVA

- SVA provides *local variable* support within properties and sequences
  - Can eliminate the need for extra modeling code

```verilog
property req_gnt;
logic [3:0] lv_id;
 @(posedge clk) disable iff (~rst_n)
($rose(req), lv_id=req_id) |=> (gnt, gnt_id==lv_id);
endproperty
```

Summarize SVA Structure

```verilog
assert property (@(posedge clk) disable iff (~rst_n)
! (grant0 & grant1));
```
SVA Summary

- Language structure
- ## sequence operator
- Repetition operators
- Overlapping |-> and non-overlapping |=> implication operator
- Named properties
- Directives: assert