Assertion-Based Verification using SystemVerilog

Mark Litterick (Verification Consultant)

mark.litterick@verilab.com
Introduction

• Overview of ABV
• Overview of SystemVerilog Assertions
  – general syntax and components
  – formal arguments
  – local variables
  – multiple clocks
• Detailed analysis of complex worked examples
  – combinations of SVA constructs
  – demonstrate power and capability of SVA
• Conclusion
• Related reading
Assertion-Based Verification

• **Assertion-Based Verification** is a *methodology* for improving the *effectiveness* of a *verification* environment
  – define properties that specify expected behavior of design
  – check property assertions by simulation or formal analysis
  – ABV does not provide alternative testbench stimulus

• Assertions are used to:
  – clarify specification requirements
  – capture design intent of implementation
  – validate correct operation and usage of design

• Benefits of ABV include:
  – improved error detection and reduced debug time due to observability
  – improved integration due to built-in self-checking
  – improved communication and documentation
ABV Methodology

• ABV involves:
  – *analysis* of design to determine key targets for assertions
  – *implementation* of appropriate properties, assertions and coverage
  – *validation* by formal analysis (static), simulation (dynamic) or mixture

• ABV can be performed during the following project phases:
  – *specification* and planning (both design and verification)
  – *design* architecting and implementation
  – *verification* environment architecting, implementation and execution

• ABV can be applied to:
  – an existing design with known problems or planned derivatives
  – new designs during or prior to development
  – individual parts of a system or the full project
SystemVerilog Assertions

- SVA is concise and powerful
  - intuitive for RTL-heads
  - but practice required
  - code carefully for maintenance
  - avoid cryptic regular expressions
  - test assertions for pass/fail
- Example: handshake interface
  - req gets ack before other req

```systemverilog
clock req ack data
sequence s_transfer;
req ##1 !req [*1:max] ##0 ack;
endsequence

property p_transfer;
@ (posedge clk)
disable iff (reset)
req |-> s_transfer;
endproperty

a_transfer :
assert property (p_transfer)
else $error("illegal transfer");
```
Formal Arguments

- Generic properties and sequences can use formal arguments
- Actual values taken from property binding when triggered
  - like an instance of the property
- Not limited to signal connections
  - can be events, variables, constants
  - more later...
- Example: handshake interface
  - used for valid/done protocol

```verilog
clock ___________
valid ___________
done ___________
data □□□□□□□□□□

property p_handshake(clk,req,ack);
  @(posedge clk)
  req |=> !req [*1:max] ##0 ack;
endproperty

assert property
(p_handshake(clock,valid,done));
```
Local Variables

- Variables can be declared locally within property
  - syntax = (sub-sequence, variable assignment)
  - variable only assigned if sub-sequence evaluates to true
- Each assertion thread has private variable
- Example: out-of-order tagged bus protocol

```verilog
property p_tag;
byte tag;
@(posedge clk)
(MCmd==RD,tag=MTag) |-> ##[1:5] (SResp==DVA)&(STag==tag);
endproperty
assert property(p_tag);
```
Multiple Clocks

- SVA supports multiple clock events
  - sequences use `#1`
  - properties use non-overlapped implication `|=>`
  - can use any valid timing events (more later…)
- Example: pulse synchronizer
  - every input pulse results in output pulse

```
property p_in_out;
  @(posedge clk_i) p_in |=>
  @(posedge clk_o) ##[2:3] p_out;
endproperty
assert property (p_in_out);
```
Data Integrity

• Example: dual-clock asynchronous FIFO
  – data integrity => correct value and order
int wcnt, rcnt;
always @(posedge wclk) if (write) wcnt = wcnt + 1;
always @(posedge rclk) if (read) rcnt = rcnt + 1;

property p_data_integrity;
  int cnt;
  logic data;
  @(posedge wclk)
    (write, cnt=wcnt, data=wdata) |=>
  @(posedge rclk)
    first_match(##[0:$] (read & (rcnt==cnt)))
    ##0 (rdata==data);
endproperty

assert property (p_data_integrity);
Timing Diagram for
\textit{p\_data\_integrity}

\begin{verbatim}
@\texttt{posedge wclk)
  (write, cnt=wcnt, data=wdata) |=>
@\texttt{posedge rclk)
  first_match(##[0:$$] (read & (rcnt==cnt)))
  ##0 (rdata==data);
\end{verbatim}
Glitch Detection

- Example: clock-domain crossing (fast-to-slow)
  - signal changes must be sampled by destination clock
  - glitch is any transition that is missed by destination clock domain (could be many source clocks wide)

```verilog
property p_no_glitch;
logic data;
@(d_in)
(1, data = !d_in) |=>
@(posedge clk)
(d_in == data);
endproperty

assert property(p_no_glitch);
```
Timing Diagram for
\( p_{\text{no\_glitch}} \)

```
property p_no_glitch;
    logic data;
    @(d_in)
        (1, data = !d_in) |=>
    @(posedge clk)
        (d_in == data);
endproperty
```

\[
\begin{align*}
\text{clk} & \quad \text{d\_in} \\
\text{S} & \quad \text{S}
\end{align*}
\]

\[
\begin{align*}
@\text{posedge d\_in} \, \text{sample}(d\_in) & \equiv 0 \\
@\text{negedge d\_in} \, \text{sample}(d\_in) & \equiv 1
\end{align*}
\]
Timing Checks

- Gate-level models have timing checks built-in (specify/vital)
  - not all timing defects fail built-in timing checks
  - e.g. narrow pulse in worst-case, but not narrow enough to violate $width$
- RTL does not normally have timing checks
  - sometimes timing checks are appropriate verification objectives
  - e.g. clock mux glitch detection based on current clock periods
SVA Timing Checks

- SVA allows *time* and *EventExpression* formal arguments
- Actual arguments can be static values or dynamic variables
- Can be used for:
  - pulse width violations (e.g. for clock quality checks at multiplexer)
  - dynamic period checks (e.g. for Multi-Voltage)
  - etc.

```vhdl
property p_min_time(start,stop,duration);
  time start_time;
  @(start)
    (1,start_time = $time) |=>
  @(stop)
    (($time - start_time) >= duration);
endproperty

property p_min_high;
  p_min_time(posedge clk, negedge clk, 2ns);
endproperty

a_min_high : assert property (p_min_high);

time minp = min_period_current_voltage(volt_v);
property p_min_period;
  p_min_time(posedge clk, posedge clk, minp);
endproperty

a_min_period : assert property (p_min_period);
```
SVA Timing Coverage

• SVA timing checks can also be used for functional coverage
  • *cover property* statement records a *hit* if property evaluates to *true*
  • Can be used to ensure testbench environment created the required timing relationships
    - gate-level stress case
    - high-level protocol checks
    - etc.

```verilog
property p_max_time(start,stop,duration); 
time start_time;
@(start)
(1,start_time = $time) |=>
@(stop)
($time - start_time) < duration);
endproperty

property p_max_time(data,posedge clk, 30ps);
endproperty

property p_just_before;
    p_max_time(data, posedge clk, 30ps);
endproperty

property p_just_after;
    p_max_time(posedge clk, data, 30ps);
endproperty

c JB : cover property (p_just_before);
c JA : cover property (p_just_after);
```
Conclusion

• SVA is cool!
  – Powerful
  – Flexible

• Formal arguments, local variables and multiple clocks
  – Enable complex checks
  – Not restricted to low-level protocol checks

• Shown some examples, but most important:
  – Ideas
  – Concepts
  – Take away and adapt for your own applications
Related Reading

• Pragmatic Simulation-Based Verification of Clock Domain Crossing Signals and Jitter using SystemVerilog Assertions
  – DVCon 2006
• Using SystemVerilog Assertions in Gate-Level Verification Environments
  – DVCon 2006
• Focusing Assertion Based Verification Effort for Best Results
  – Mentor Solutions Expo 2005
• Using SystemVerilog Assertions for Functional Coverage
  – DAC 2005

www.verilab.com/resources/papers-and-presentations