Introduction to PSL for VHDL 200x

Erich Marschner
Senior Architect, Advanced Verification

July 2003
Boolean Logic Review

• If $P$, $P_1$, $P_2$ are predicates, then the following are also:
  
  $- (P)$
  $- \text{not } P$
  $- P_1 \text{ and } P_2$
  $- P_1 \text{ or } P_2 = (\text{not }((\text{not } P_1) \text{ and } (\text{not } P_2)))$
  $- P_1 \rightarrow P_2 = ((\text{not } P_1) \text{ or } P_2))$
  $- P_1 \leftrightarrow P_2 = ((P_1 \rightarrow P_2) \text{ and } (P_2 \rightarrow P_1))$
  $- \text{true} = (P \text{ or } (\text{not } P)) \text{ for any } P$
  $- \text{false} = (P \text{ and } (\text{not } P)) \text{ for any } P$
Temporal Logics

• Computation Tree Logic (CTL)
  – Expresses properties of states
    – e.g., “for all states reachable in one step from this state, ....”
    – leads to branching-time statements such as AGEFp:
      – “for all states globally, there exists a future state in which p holds”

• Linear-Time Temporal Logic (LTL)
  – Expresses properties of paths, or sequences of states
    – e.g., “at every state along the path, ....”
    – leads to linear-time statements such as Ap->Xq:
      – “whenever p holds along a given path, q holds in the next state of the path.”
Structure of PSL

• **Linear-Time (LTL) part**
  – for formal verification
  – for simulation
  – Simple Subset
    – LHS of certain binary operators must be boolean
    – easy to implement in simulation

• **Branching-Time (CTL) part**
  – for formal verification

---

© 2003, Cadence Design Systems. All rights reserved.
Basic LTL Operators

- \( P_t = \text{true iff predicate } P \text{ is true at time } t \)
- \( \text{next } P = \text{true iff, at time } t, P_{t+1} = \text{true} \)
- \( \text{always } P = (P \text{ and next } (P \text{ and next } (P \text{ and } \ldots ))) \) 
- \( \text{eventually } P = (P \text{ or next } (P \text{ or next } (P \text{ or } \ldots ))) \) 
- \( \text{never } P = (\neg P \text{ and next } (\neg P \text{ and next } (\neg P \text{ and } \ldots ))) \) 
- \( \forall t \ P_t \) 
- \( \exists t \ P_t \) 
- \( \forall t \ \neg P_t \) 
- \( \exists t \ \neg P_t \) 

\( \neg \text{always } \)
Sequences, SEREs, and Suffix Implication

• A “Sequence” is a brace-enclosed series of Boolean expressions that are considered in successive timesteps
  
  - \{B1; B2; B3; ... \}

• A “Sequential Regular Expression” (SERE) is an element of a Sequence:

  - \(B\) \(B[*]\) \(B[=n]\) \(B[*n:m] \) \(B[+]\)
  
  - \(B[=0]\) \(B[=n]\) \(B[=n:m] \) \(B[->]\) \(B[->n]\) \(B[->n:m]\)
  
  - \(r1 \; ; \; r2\) \(\{r1\} : \{r2\}\) \(\{r1\} | \{r2\}\) \(\{r1\} \& \{r2\}\) \(\{r1\} \&\& \{r2\}\)

• General use of ‘next’

  - \(B1 \rightarrow \text{next} (B2 \rightarrow \text{next} (B3 \rightarrow ... \rightarrow (Bn-2 \text{ and next} (Bn-1 \text{ and next} (Bn))))..))))\)

• Equivalent to suffix implication:

  - \{B1; B2; B3; ...\} \Rightarrow \{...; Bn-2; Bn-1; Bn\}
Reasoning over Finite Traces

• LTL is usually applied to infinite traces.
• Simulation deals with finite traces.
• How should we interpret temporal operators in simulation?

```
eventually b

never b
```

Pass

Fail

© 2003, Cadence Design Systems. All rights reserved.
Overview of PSL

• Boolean Expressions
  – HDL expressions
  – PSL/Sugar functions `rose()`, `fell()`, `prev()`, ...

• Temporal Operators
  – `always`, `never`, `next`, `eventually`, `until`, `before`, `abort`, ...
  – `@` -> `<->` ; `{}` `[*]` `[=]` `[->]` `&&` `&` `|` `:`

• Verification Directives
  – `assert`, `cover`, ...

• Modeling Constructs
  – **HDL statements** used to model the environment
Kinds of Assertions

- **Interface Assertions**
- **Structural Assertions**
- **Protocol Assertions**

Diagram showing interfaces and components:
- FIFO
- FSM
- PCI Bus
- AHB Bus
- Arbiter
- Processor

© 2003, Cadence Design Systems. All rights reserved.
A Simple Example

- Two blocks A, B exchange data via a common bus.

- A and/or B sends ‘Req’ to the Arbiter.

- Arbiter does round-robin scheduling between A, B.

- Arbiter sends ‘Gnt’ back to A or B, making it Master.

- Arbiter sets ‘Busy’ while A or B is Master.

- Master sets ‘DRdy’ when Data is on the bus.

- Master sets ‘Done’ in the last cycle of a grant.

- ‘Reset’ resets the bus.
Some Assertions to Check

• A Grant never occurs without a Request.
  – assert never \( GntA \) and not \( ReqA \)

• If A (B) receives a Grant, then B (A) does not.
  – assert always \( GntA \rightarrow not GntB \)

• A (B) never receives a Grant in two successive cycles.
  – assert never \( GntA \) and next \( GntA \)

• A Grant is always followed by Busy.
  – assert always \( GntA \) or \( GntB \rightarrow next Busy \)

• A Request is eventually followed by a Grant.
  – assert always \( ReqA \rightarrow eventually GntA \)
A Grant never occurs without a Request.

Verification Directive

Temporal Operator

Boolean Expression

assert never GntA and not ReqA ;

This assertion should hold at every time step

ReqA

GntA
Unclocked Invariants

A Grant never occurs without a Request.

assert never GntA and not ReqA ;

Timing of transitions might result in a failure...
Clocked Invariants

A Grant never occurs without a Request.

... but assertions can be clocked ...

... which causes them to ignore glitches
Clocked Invariants

A Grant never occurs without a Request.

Verification Directive  Temporal Operator  Boolean Expression  Clock Expression
assert  never  GntA and not ReqA  @rose(clk) ;

Clock can be level-sensitive or edge-sensitive

ReqA  GntA  clk
@clk  @(not clk)
@rose(clk)  @felling(clk)
@rising(clk)  @falling(clk)
Conditional Behavior

If A receives a Grant, then B does not.

assert always (GntA -> not GntB) @rose(clk) ;

Implication (->) expresses “if...then”

At the rising clk, if GntA is high, then GntB must be low
Multi-Cycle Behavior

A (B) never receives a Grant in two successive cycles.

assert never GntA and next GntA @rose(clk) ;

If GntA is high for two cycles, the assertion fails

‘next’ refers to the next time step (or clock edge)
Multi-Cycle Behavior using Sequences

A (B) never receives a Grant in two successive cycles.

assert never GntA and next GntA @rose(clk) ;

assert never \{GntA ; GntA\} @rose(clk) ;

A sequence is a shorthand for a series of ‘next’s
Multi-Cycle Conditional Behavior

A Grant is always followed by Busy.

assert always GntA or GntB -> next Busy @rose(clk) ;

Implication (->) and ‘next’ together express multi-cycle conditional behavior

Now there is a one-cycle delay from ‘if’ to ‘then’
Multi-Cycle Conditional Behavior

• A Request is eventually followed by a Grant.

\[
\text{assert always ReqA} \rightarrow \text{eventually}! \ GntA \ @\text{rose(clk)}
\]

‘Eventually’ refers to now or some future cycle.

One Grant satisfies all related ‘if-then’ requirements.

ReqA

GntA

clk
More Assertions to Check

• If Request is followed by Grant, then next is Busy, and next is Done.
  – assert always (ReqA -> next (GntA -> next (Busy and next Done)))
  – assert always {ReqA; GntA} |=> {Busy; Done}

• If Request is followed by Grant, then next Busy is high until Done.
  – assert always (ReqB -> next (GntB -> next (Busy until Done)))
  – assert always {ReqB; GntB} |=> {Busy[*]; Done}

• A Grant is always followed by Busy until, and overlapping with, Done.
  – assert always (GntA or GntB) -> next (Busy until _ Done)
  – assert always {GntA or GntB} |=> {Busy[*]; Busy and Done}

• If A has a Request outstanding when B receives a Grant, then A will receive a Grant before B receives another Grant.
  – assert always (ReqA and GntB) -> next (GntA before GntB)
  – assert always {ReqA and GntB} |=> {[*]; GntA} && {GntB[=0]}
Sequences and Suffix Implication

assert always 
(ReqA -> next (GntA -> next (Busy and next Done)))@rose(clk);

assert always ({ReqA; GntA} |=> {Busy; Done})@rose(clk);

The left-hand side (LHS) sequence is the ‘enabling’ sequence

The right-hand side (RHS) sequence is the ‘fulfilling’ sequence

The suffix implication operator says “if LHS, then RHS”
Compound Assertions

If Request is followed by Grant, then next is Busy, and next is Done.

```plaintext
assert always (ReqA -> next (GntA -> next (Busy and next Done))) @rose(clk) ;
assert always {ReqA; GntA} |=> {Busy; Done} @rose(clk) ;
```

The two assertions are equivalent.

Evaluation starts again in each cycle, overlapping with previous evaluations.
More Precise Specification

If Request is followed by Grant, then next Busy is high until Done.

```vHDL
assert always (ReqB -> next (GntB -> next (Busy until Done))) @rose(clk) ;
assert always {ReqB; GntB} |=> {Busy[*]; Done} @rose(clk) ;
```

[*] means zero or more occurrences

Busy can stay high for any number of cycles
**Even More Precise Specification**

A Grant is always followed by Busy until, and overlapping with, Done.

\[
\text{assert always } ((\text{GntA or GntB}) \rightarrow \text{next (Busy until Done)}) \; \text{@rose(clk)} ;
\]

\[
\text{assert always } \{\text{GntA or GntB}\} \Rightarrow \{\text{Busy[*]; Busy and Done}\} \; \text{@rose(clk)} ;
\]

---

- **GntA**
- **GntB**
- **Busy**
- **Done**
- **clk**

'until\_\_' requires that Busy be high when Done occurs
Still More Assertions

If A has a Request outstanding when B receives a Grant, then A will receive a Grant before B receives another Grant.

assert always (ReqA and GntB) -> next (GntA before GntB) @rose(clk) ;
assert always {ReqA and GntB} |=> {[*]; GntA} && {GntB[=0]} @rose(clk) ;
Assertions and Coverage

- Assertions only catch bugs that occur during verification.
  - Conditional assertions must be enabled before they can fail.

- Coverage Monitors ensure that verification is thorough.
  - Monitors check that all interesting behavior is exercised.
  - Functional coverage is more effective than code coverage.

- PSL includes both
  - Assert Directives (assertions)
  - Cover Directives (coverage monitors)
Some Scenarios to Cover

• Test the case in which a transfer includes from 1 to 3 successive data ready cycles.
  – cover \{(GntA or GntB); \{Busy[*] && DRdy[*1:3]\} : \{Done\}\}

• Test the case in which a transfer includes exactly 4 data ready cycles (but not necessarily in succession).
  – cover \{(GntA or GntB); \{Busy[*] && DRdy[=4]\} : \{Done\}\}

• Test the case in which a transfer completes without having the bus reset.
  – cover \{(GntA or GntB); \{Busy[*] && Reset[=0]\} : \{Done\}\}

• Test the case in which a transfer is interrupted by a bus reset.
  – cover \{(GntA or GntB); \{Busy[*] && Reset[->1]\}\}