APPENDIX

A

QUICK TUTORIAL FOR SVA

In this appendix, we provide a quick tutorial for SystemVerilog Assertions (SVA). It is not our objective to present a comprehensive overview of SVA. Our goal is to provide you with enough information so that you can understand the examples presented in this book. For a complete overview and reference of the SVA language, we recommend the following sources: [Cohen 2005], [Haque et al., 2006], and [Vijayaraghavan and Ramanathan 2005].

A.1 SVA fundamentals

SVA, an assertion sublanguage of the IEEE Std 1800-2005 SystemVerilog standard [IEEE 1800-2005], is a linear-time temporal logic that is intended to be used to specify assertions and functional coverage properties for the validation and verification of concurrent systems.

Using SVA, we are able to specify the design intent (that is, expected behavior) in terms of properties. Once defined, we can check our properties as assertions or use them as verification constraints (assumptions), or they can describe behavior that must be observed during the course of simulation (that is, coverage properties).

SVA contains two types of assertions: *immediate* and *concurrent*. Immediate assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block. Immediate assertions are primarily intended to be used with simulation. In contrast, concurrent assertions are based on clock semantics and use sampled values of variables (that is, they exist outside the context of procedural code).

A.1.1 Immediate assertions

In many respects, an immediate assertion is similar to an if statement. They check to see if a conditional expression holds, and if not, then the simulator generates an error message.

For instance, Example A-1 demonstrates a portion of some procedural code associated with an arbiter design where a procedural `if` statement checks that two of the arbiter's `grant` signals are never active at the same time.

Example A-1 Immediate check for mutually exclusive grants

```
module my_arb (. . . )

 always @* begin // arbiter code
    . . .
    if (grant[0] & grant[1]))
      $display ("Mutex violation for grant[0] and grant[1]");
    . . .
  end

endmodule
```

A key observation is that the Boolean condition for the `if` statement is immediately checked within the procedural context of the code (that is, when the `if` statement is visited during the procedural code execution).

Example A-2 demonstrates the same check, except in this case, a SystemVerilog immediate assertion is used.

Example A-2 Immediate check for mutually exclusive grants

```
module my_arb (. . . )

 always @* begin // arbiter code
    . . .
    assert (!(grant[0] & grant[1]));
    . . .
  end

endmodule
```

For this example, the Boolean condition

```
!(grant[0] & grant[1])
```

is immediately checked only when the assert statement is visited within the procedural context of the SystemVerilog `always` block. If the Boolean condition does not evaluate true during the procedural execution of the assertion, then an error message is automatically generated.

Immediate assertions may be used within SystemVerilog `initial` and `always` blocks or as a `task` or `function`.

A.1.2 Concurrent assertions

There are properties of a design that must evaluate true globally, and not just during a particular instance in which a line of procedural code executes. Hence, SVA provides support for *concurrent assertions*, which may be checked outside the execution of a procedural block. That is, the checks are performed concurrently with all other procedural blocks in the verification environment.

Concurrent assertions are easily identified by observing the presence of the SVA `property` keyword combined with the `assert` directive. Example A-3 situates a concurrent assertion for our arbiter example. In this case, the concurrent assertion exists outside the context of a procedural always block.

Example A-3 Immediate check for mutually exclusive grants

```
module my_arb (. . . )

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

  assert property (@(posedge clk) !(grant[0] & grant[1]));

endmodule
```

Concurrent assertions are based on clock semantics, use sampled values of variables, and can be specified in always blocks, initial blocks, or stand alone outside a procedural block.

All of the assertion-based IP examples we present in this book consist of a set of concurrent assertions with a specific sample clock associated with them to prevent false firings due to race conditions. One of the challenges associated with immediate (unclocked) assertions contained within procedural code is that they can suffer the same race condition problems as other design code, which can result in false firings during simulation. Hence, you must be careful when using them in this context.

A.1.3 Resets

SystemVerilog provides a mechanism to asynchronously disable an assertion during a reset using the SVA `disable iff` clause. In Example A-4, we demonstrate the same assertion expressed in Example A-3, except we have added a reset signal. Even though our concurrent assertion is clocked (`@(posedge clk)`) the assertion is asynchronously disabled when the `disable` iff clause expression evaluates true.

Example A-4 Asynchronously disabling an assertion with a reset

```
module my_arb (. . . )

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

  assert property (@(posedge clk) disable iff (reset)
     !(grant[0] & grant[1]));

endmodule
```

A.1.4 Action blocks

An SVA action block specifies the actions that are taken upon success or failure of the assertion (see the BNF fragment in Example A-5). The statement associated with the success of the assert statement is the first statement of an action block. It is called the *pass statement* and is executed if the expression evaluates to true. The pass statement can, for example, record the number of successes for a coverage log but can be omitted altogether. If the pass statement is omitted, then no user-specified action is taken when the assert expression holds. The statement associated with the assertion's `else` clause is called a *fail statement* and is executed if the expression evaluates to false. The fail statement can also be omitted. The action block, if specified, is executed immediately after the evaluation of the assert expression.

Example A-5 SystemVerilog concurrent assertion syntax

```
concurrent_assertion_statement ::=
   assert property ( property_spec ) action_block

property_spec ::=
   [ clocking_event ] [ disable iff ( expression_or_dist ) ]
      property_expr

action_block ::=
       pass_statement_or_null
     | [ pass_statement ] else fail_statement
```

Example A-6 demonstrates an assertion where the action block's pass statement has been omitted, yet the fail statement exists and is used to pass failure information out of the assertion-based IP's analysis port.

Example A-6 SystemVerilog concurrent assertion syntax

```
assert property
  ( @(posedge clk) disable iff (reset) !(grant[0] & grant[1]) )
else begin // action block fail statement
// See Appendix B, "Complete OVM/AVM Testbench Example" for OVM
details
  status = new();
  status.set_err_grant_mutex();
  status_ap.write(status);
end
```

A.2 SystemVerilog sequences

In the previous section, we demonstrated simple assertions based on the evaluation of a single Boolean expression that must hold true at every sample point in time. In this section, we introduce SVA sequences, which are Boolean expressions that are evaluated in a linear order of increasing time.

The temporal delay operator "##" constructs sequences by combining Boolean expressions (or smaller sequences). For instance, Example A-7 demonstrates a set of simple sequences using the temporal delay operator.

Example A-7 Construction of sequences with temporal delay

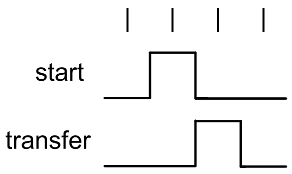
```
start ##1 transfer     // a sequence in which the Boolean variable
                       // transfer holds on the clock after start

start ##2 transfer     // a sequence in which the Boolean variable
                       // transfer holds two clocks after start

start ##[0:2] transfer // a sequence in which the Boolean variable
                       // transfer holds between zero to two clocks
                       // after start
```

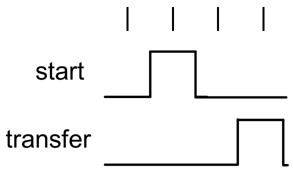
Figure A-1 illustrates a sequence in which the Boolean variable `transfer` holds exactly one clock after `start` holds.

Figure A-1. Trace on which the sequence (start ##1 transfer) **holds**



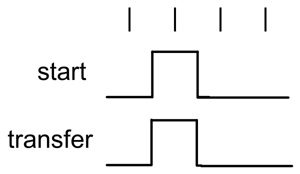
Similarly, Figure A-2 illustrates a sequence in which the Boolean variable `transfer` holds exactly two clocks after `start` holds.

Figure A-2. Trace on which the sequence (start ##2 transfer) **holds**



Finally, Figure A-3 illustrates a sequence in which the Boolean variable `transfer` holds between zero and two clocks after `start` holds. In fact, Figure A-1 and Figure A-2 also hold on the sequence defined by `(start ##[0:2] transfer)`.

Figure A-3. Trace on which sequence (start ##[0:2] transfer) **holds**



A temporal delay may begin a sequence. The range may be a single constant amount or a range of time. All times may be used to match the following sequence. The range is interpreted as follows:

Example A-8 Construction of sequences with temporal delay

```
s##0 a     - same as  (a)

##1 a      - same as  (1'b1 ##1 a)

##[0:1] a - same as  a  or  (1'b1 ##1 a)
```

When the symbol `$` is used, the range is infinite. For example, `req ##[1:$] gnt` specifies a sequence in which a `gnt` signal occurs at some point after a `req`.

Assertion 6-1, "A requesting client will eventually be served" on page 115 demonstrates the use of a delay operator.

A.2.1 Consecutive repetition

SystemVerilog provides syntactic sugar to simplify expressing the repetition for Boolean expressions or a sequence expression. For example, the sequence `(a ##1 a)` can be expressed as sequence `a[*2]`.

The `[*n]` construct is used to represent a repetition, where `n` is a constant. A repetition range can be expressed using `[*m:n]`, where both `m` and `n` are constants.

Sequence repetition examples include:

Example A-9 Construction of sequences with temporal delay

```
a [*0] ##1 b        same as (b), a is not repeated
a [*2] ##1 b        same as (a ##1 a ##1 b)
a [*1:2] ##1 b      same as (a ##1 b) or (a ##1 a ##1 b)
(a ##1 b) [*2]      same as (a ##1 b ##1 a ##1 b)
```

The first example demonstrates that a repetition of zero is equivalent to the following:

`a[*0] ##n b` *is equivalent to* `##(n-1) b`

where `n` is any constant greater than zero.

Assertion 5-4, "Valid transfer size property" on page 74 demonstrates a consecutive repetition.

A.2.2 Goto repetition

The goto repetition is syntactic sugar that allows for space (absence of the term) between the repetition of the terms. For example, a[->2] is a shortcut for the following sequence:

```
~a[*0:$] ##1 a ##1 ~a[*0:$] ##1 a
```

Assertion 6-2, "Two-client fair arbitration assertion for client 0" on page 116 demonstrate an SVA goto repetition.

A.2.3 Nonconsecutive repetition

The nonconsectutive repetition operator is syntactic sugar that allows for space (absence of a term) between the repetition of the terms. For example, a[=2] is a shortcut for the following sequence:

```
~a[*0:$] ##1 a ##1 ~a[*0:$] ##1 a ##1 ~a[*0:$]
```

Note the difference in the goto repetition and nonconsecutive repetition operators. The nonconsecutive operators do not require the repeating term to be true at the end of the sequence. A nonconsecutive repetition is often followed by another element in the sequence. For example:

```
a[=2] ##1 b
```

This expression describes a sequence of two nonconsecutive occurrences of `a`, followed eventually by an occurrence of `b`.

Assertion 5-11, “Bus wdata properties” on page 104 demonstrates a nonconsecutive repetition.

A.2.4 Declared named sequences

To facilitate reuse, sequences can be declared and then referenced by name. A sequence can be declared in the following:

- a module
- an interface
- a program
- a clocking block
- a package
- a compilation-unit scope

Sequences can be declared with or without parameters, as demonstrated in Example A-10.

Example A-10 Declared sequences

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

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

A.3 Property declarations

SystemVerilog allows for declarations of properties to facilitate reuse. A property differs from a sequence in that it contains a clock specification for the entire property and an optional asynchronous term to disable the property evaluations.

Named properties can be used to construct more complex properties, or they can be directly used during verification as an assertion, assumption, or coverage item. Properties can be declared with or without parameters, as demonstrated in Example A-11.

Example A-11 Declared properties

```
property req_t1_start;
  @(posedge clk)  req && req_tag == t1;
endproperty

property illegal_op;
  @(posedge clk)  ~(req && cmd == 4);
endproperty

property en_mutex(en[0], en[1], reset_n);
  @(posedge clk)  disable iff  (~reset_n)    // asynch reset
    ~(en[0] & en[1]);
endproperty
```

Properties may reference other properties in their definition. They may even reference themselves recursively. Properties may also be written using multiple clocks to define a sequence that transitions from one clock to another as it matches the elements of the sequence.

A.4 Sequence and property operators

The sequence operators defined for SystemVerilog allow us to compose expressions into temporal sequences. These sequences are the building blocks of properties and concurrent assertions. The first four allow us to construct sequences, while the remaining operators allow us to perform operations on a sequence as well as compose a complex sequence from simpler sequences.

A.4.1 AND

Both SVA properties and sequences can be operands of the and operator. The operation on two sequences produces a match when both sequences produce a match (the end point may not match). A match occurs until the endpoint of the longer sequence (provided the shorter sequence produces one match).

Examples of sequence and are:

Example A-12 Sequence AND

```
(a ##1 b) and ()        same as (), which is a no match
(a ##1 b) and (c ##1 d) same as (a & c ##1 b & d)
(a ##[1:2] b) and (c ##3 d)
                          same as (a & c ##1 b ##1 1 ##1 d)
                          or (a & c ##1 1 ##1 b ##1 d)
```

A.4.2 Sequence intersection

An intersection of two sequences is like an and of two sequences (both sequences produce a match). This operator also requires the length of the sequences to match. That is, the match point of both sequences must be the same time. With multiple matches of each sequence, a match occurs each time both sequences produce a match.

Example A-13 demonstrates the SVA sequence intersect operator.

Example A-13 Sequence intersect

```
(1) intersect ()              same as (), which is a no match

##1 a intersect ##2 b         same as (), which is a no match

##2 a intersect ##2 b         match if ##2 (a & b))

##[1:2] a intersect ##[2:3] b  match if (1 ##1 a&b )
                                     or (1 ##1 a&b ##1 b)

##[1:3] a intersect ## [1:3] b match if (##1 a&b)
                                     or (##2 a&b)
                                     or (##3 a&b)
```

A.4.3 OR

SVA provides a means to or sequences. For or-ed sequences, a match on either sequence results in a match for the operation

Example A-14 demonstrates the SVA sequence or operator.

Example A-14 Sequence OR

```
() or ()                 same as (), which is a no match

(## 2 a or ## [1:2] b)   match if (b)
                               or (##1 b) or (## 2 a) or ( ##2 b)
```

A.4.4 Boolean throughout

This operator is used to specify a Boolean value throughout a sequence. The operator produces a match if the sequence matches and the Boolean expression holds until the end of the sequence.

Example A-15 demonstrates the SVA sequence throughout operator.

Example A-15 Sequence throughout

```
0 throughout  (1)          same as (), which is a no match

1 throughout  ##1 a        same as ##1 a

a throughout  ##2 b        same as (a ##1 a & b)

a throughout (b ##[1:3] c)  same as (a&b ##1 a ##1 a &c)
                               or  (a&b ##1 a ##1 a ##1 a&c)
                               or  (a&b ##1 a ##1 a ##1 a ##1 a&c)
```

A.4.5 Sequence within

The within operator determines if one sequence matches within the length of another sequence.

Example A-16 demonstrates the SVA sequence within operator.

Example A-16 Sequence within

```
() within (1)             same as (), which is a no match

(1) within ()             same as (), which is a no match

(a) within ## [1:2] b     same as (a&b) or (a ##1 b) or (##1 a&b)
```

A.4.6 Sequence ended

The method ended returns true at the end of the sequence. This is in contrast to matching a sequence from the beginning time point, which is obtained when we use only the sequence name.

Example A-17 demonstrates the SVA sequence ended method.

Example A-17 Sequence ended

```
sequence a1;
   @(posedge clk) (c ##1 b ##1 d);
endsequence

(a ##1  a1.ended)       same as (c ##1 b & a ##1 d)

(a ##2  a1.ended)       same as (c & a ## b ##1 d)
```

A.4.7 Sequence first_match

The first_match operator returns true only for the first occurrence of a multiple occurrence match for a sequence.

Example A-18 demonstrates the SVA sequence first_match method.

Example A-18 Sequence ended

```
first_match (1 [*1:5])        same as (1)

first_match (##[0:4] 1)       same as (1)

first_match (##[0:1] a)       same as (a) or (!a ##1 a)

first_match (b throughout s1) same as
                                  (b throughout first_match(s1))

first_match(s1 within s2)     same as
                                  (s1 within first_match (s2))
```

A.4.8 Implication

SystemVerilog supports implication of properties and sequences. The left-hand operand of the *implication* is called the antecedent and the right-hand operand is called the *consequent*. Evaluation of an implication starts through repeated attempts to evaluate the antecedent. When the antecedent succeeds, the consequent is required to succeed for the property to hold.

As a convenience, there are two forms of implication, overlapping (|->) and non-overlapping (|=>). The overlap occurs between the final cycle of the left-hand side (the antecedent) and the starting cycle of the right-hand side (the consequent) operands. For the overlapping form, the consequent starts on the current cycle (that the antecedent matched). The non-overlapping form has the consequent start the subsequent cycle. Implication is similar in concept to an if statement.

Example A-19 demonstrates the SVA implication operators.

Example A-19 Implication operators

```
a |-> b                   same as   a ? b : 1'b1

(a ##1 b) |-> (c)        conceptually same as  (a ##1 b) ? c : 1'b1

(a ##1 b) |=> (c ##1 d)  same as   (a ##1 b) |-> ##1 c ##1 d)
```

A.5 SVA system functions and task

SVA provides various system functions to facilitate specifying common functionality. This section describes a few of SVA's more commonly used system functions.

A.5.1 Sampled value functions

Sampled value functions include the capability to access the current or past sampled value, or detect changes in the sampled value of an expression. The following functions are provided:

```
$sampled(expression [, clocking_event])

$rose( expression [, clocking_event])

$fell( expression [, clocking_event])

$stable( expression [, clocking_event])

$past( expression1 [, number_of_ticks]
            [, expression2] [, clocking_event])
```

Using these functions is not limited to assertion features; they can be used as expressions in procedural code as well. The clocking event, although optional as an explicit argument to the functions, is required for semantics. The clocking event is used to sample the value of the argument expression.

The clocking event must be explicitly specified as an argument or inferred from the code where it is used. The following rules are used to infer the clocking event:

- If used in an assertion, the appropriate clocking event from the assertion is used.
- If used in an action block of a singly clocked assertion, the clock of the assertion is used.
- If used in a procedural block, the inferred clock, if any, for the procedural code is used.

Otherwise, default clocking is used.

In addition to accessing value changes, the past values can be accessed with the $past function. The following three optional arguments are provided:

- *expression2* is used as a gating expression for the clocking event

- *number_of_ticks* specifies the number of clock ticks in the past
- *clocking_event* specifies the clocking event for sampling expression1

The *expression1* and *expression2* can be any expression allowed in assertions.

Assertion 5-12, “Bus slave ready assertions” on page 105 demonstrates both a `$rose` and `$fell`. Assertion 5-5, “Bus must reset to INACTIVE state property” on page 86 demonstrates `$past`.

A.5.2 Useful functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is “one-hot.” The following system functions are included to facilitate such common assertion functionality:

- **$onehot** (<expression>) returns true if only one bit of the expression is high.
- **$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.

All of the above system functions have a return type of bit. A return value of 1’b1 indicates true, and a return value of 1’b0 indicates false.

Another useful function provided for the Boolean expression is $countones, to count the number of ones in a bit vector expression.

- **$countones** (expression)

An X and Z value of a bit is not counted towards the number of ones.

Assertion 6-6, “Built-in function to check mutually exclusive grants” on page 119 demonstrates a $onehot system function.

A.5.3 System tasks

SystemVerilog has defined several severity system tasks for use in reporting messages. These system tasks are defined as follows:

$fatal(finish_num [, message {, message_argument }]);

This system task reports the error message provided and terminates the simulation with the `finish_num` error code. This system task is best used to report fatal errors related to testbench/OS system failures (for example, can’t open, read, or write to files) The message and argument format are the same as the **$display()** system task.

$error(message {, message_argument });

This system task reports the error message as a run-time error in a tool-specific manner. However, it provides the following information:

- severity of the message
- file and line number of the system task call
- hierarchical name of the scope or assertion or property
- simulation time of the failure

$warning(message {, message_argument });

This system task reports the *warning* message as a run-time warning in a format similar to $error and with similar information.

$info(message {, message_argument });

This system task reports the *informational* message in a format similar to $error and with similar information.

$asserton(levels, [list_of_modules_or_assertions])

This system task will reenable assertion and coverage statements. This allows sequences and properties to match elements. If a `level` of 0 is given, all statements in the design are affected. If a list of *modules* is given, then that module and modules instantiated to a depth of the `level` parameter are affected. If specifically named assertion statements are listed, then they are affected.

$assertkill(levels, [list_of_modules_or_assertions])

This system task stops the execution of all assertion and cover statements. These statements will not begin matching until re-enabled with **$asserton**(). Use the arguments in the same way as $asserton uses them.

$assertoff(levels, [list_of_modules_or_assertions])

This system task prevents matching of assertion and cover statements. Sequences and properties in the process of matching sequences will continue. Assertion and cover statements will not begin matching again until re-enabled with **$asserton**(). Use the arguments in the same way as $asserton uses them.

A.6 Dynamic data within sequences

SVA includes the ability to call a routine or sample data within the sequence for later analysis. The ability to call a routine (tasks, void functions, methods, and system tasks) allows you to record data for analysis or debugging.

Example A-20 demonstrates the SVA sequences with system task calls.

Example A-20 Calling system task within a sequence

```
sequence track_it;
       (req[->1],  $display("Found the request at cycle %0d\n",
                             'cycle))
    ##1 (sent[->1], $display("Sent occurred at cycle %0d\n",
                              'cycle))
   ##3  done;
endsequence
assert property (start |=> track_it);
```

SVA sequences allow you to declare *local variables* that can be used to sample data at a specific point within a sequence. Example 5-12, "Encapsulate coverage properties inside a module" on page 111 demonstrates an SVA sequence with a local variable. Long *et. al* [2007] provide an excellent overview and set of guidelines for coding local variables.

A.7 SVA directives

SVA directives specify how properties are to be used. Example A-21 describes the syntax for SVA directives.

Example A-21 Directives

```
immediate_assert_statement ::=
    assert ( expression ) action_block

concurrent_assert_statement ::=
  assert property '(' property_spec ')' action_block

concurrent_assume_statement ::=
  assume property '(' property_spec ')' ';'

concurrent_cover_statement ::=
  cover property '(' property_spec ')' statement_or_null

action_block ::=
    statement   [ else statement_or_null ]
  | [statement_or_null] else statement_or_null

statement_or_null ::=
    statement   |  ';'
```

A.8 Useful named property examples

This section defines a set of useful, named property declarations we have created with the intent to be shared by an assertion-based IP design team. Our goal is to create a set of named properties that is similar to a some of the expressive IEEE 1850 Std. Property Specification Language (PSL) [IEEE 1850-2005] linear-time temporal operators. This allows us to express complex properties with minimum code through reuse.

Example A-22 Named property declarations

```
// file property_declarations.h

`ifndef PROPERTY_DECLARATIONS
`define PROPERTY_DECLARATIONS

// P -> next (Q until R)

property P_impl_Q_weak_until_R(ck,rst,P,Q,R);
  @(posedge ck) disable iff (rst)
    $rose(P) ##1 (~R)[*1:$] |-> Q;
endproperty

// P -> next (Q until! R)

property P_impl_Q_strong_until_R(ck,rst,P,Q,R);
  @(posedge ck) disable iff (rst)
    $rose(P) |=> (~R throughout (Q[*0:$])) ##1 R;
endproperty

// P -> next (Q before R)

property P_impl_Q_weak_before_R(ck,rst,P,Q,R);
  @(posedge ck) disable iff (rst)
    $rose(P) ##1 (~(Q & ~R))[*1:$] |-> ~R;
endproperty

// P -> next ( Q before! R)

property P_impl_Q_strong_before_R(ck,rst,P,Q,R);
  @(posedge ck) disable iff (rst)
    $rose(P) |=> (~R throughout Q[=1:$]) ##1 (R & ~Q);
endproperty

`endif
```

APPENDIX B

COMPLETE OVM/AVM TESTBENCH EXAMPLE

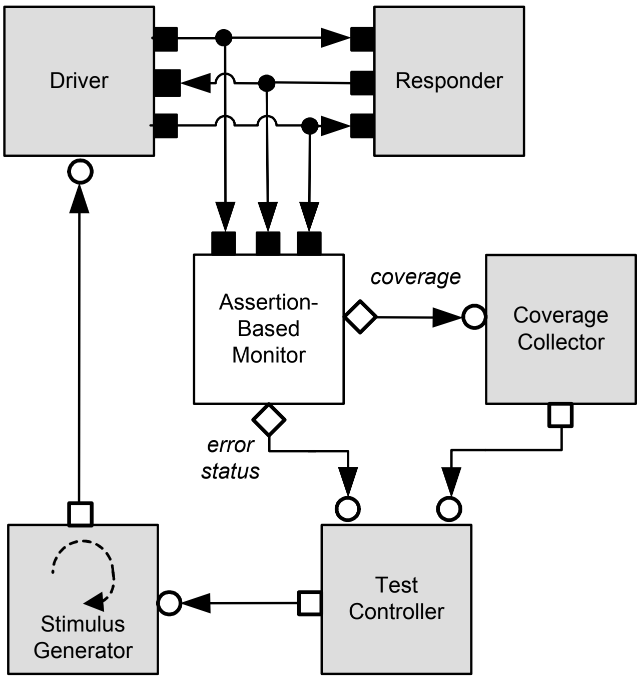
In this appendix, we provide a complete OVM/AVM example to illustrate how to integrate assertion-based IP into a *transaction-level* testbench. We have divided this appendix into two parts. The first part provides the source code for our simple nonpipelined bus interface example (previously discussed in Section 5.2 on page 75). The second part provides a high-level reference for many of the potentially less obvious OVM/AVM classes used on our testbench example.

The Open Verification Methodology (or OVM) is an open-source and class-based library that is freely available under an Apache License, Version 2.0 open-source license. OVM is a superset of the Cadence Design System's Unified Reuse Methodology (URM) and Mentor Graphic's Advanced Verification Methodology (AVM) [Glasser et al., 2007], with some additional features. We chose the Open Verification Methodology as the basis for our testbench example because the source code for the OVM library is openly available and can be downloaded at *http:// www.mentor.com/go/cookbook*. Assuredly, there are other testbench base-class libraries available. The general ideas, processes, and techniques we present in this appendix to demonstrate our assertion-based IP within a transaction-level testbench can be extended to the specific implementation details of other verification environment methodologies, such as the VMM [Bergeron et al., 2006].

B.1 OVM/AVM Example Source Code

Figure B-1 illustrates our testbench example for the simple nonpipelined bus interface example (previously discussed in Section 5.2 on page 75).

Figure B-1



You might note that our example testbench is lacking a DUV. For our example, we are emulating the bus behavior using a responder verification transactor. For this case, the responder could be replaced with an actual DUV at a point in the design cycle when the RTL is available. However, the current testbench environment allows us to debug our assertion-based IP module prior to completion of the DUV. The source code for the various verification components is defined in the following sections.

Example B-1 Testbench top module

```
interface clk_rst_if;
  bit clk , rst;
endinterface

module top;

  import avm_pkg::*;
  import tb_tr_pkg::*;
  import tb_env_pkg::*;

  clk_rst_if clk_rst_bus();

  tb_clock_reset clock_reset_gen( clk_rst_bus );

  pins_if #(.DATA_SIZE(8),.ADDR_SIZE(8))
    nonpiped_bus (
     .clk( clk_rst_bus.clk ),
     .rst( clk_rst_bus.rst)
    );

// assertion-based IP module

  tb_monitor_mod tb_monitor(
    .bus_if( nonpiped_bus )
  );

// class-based components instantiated within the environment class

  tb_env env;

  initial begin
    env = new( nonpiped_bus,
               tb_monitor.cov_ap,
               tb_monitor.status_ap );
    fork
      clock_reset_gen.run;
    join_none
    env.do_test;
    $finish;
  end

endmodule
```

B.1.1 top module

The top module shown in Example B-1 (see page 277) instantiates all SystemVerilog interfaces, modules, and class-based components.

B.1.2 tb_clock_reset module

The SystemVerilog `tb_clock_reset` module is responsible for generating all testbench clock and reset activity.

Module-based reference of an interface

Example B-2 Clock and reset generator

```
module tb_clock_reset( interface i );

  parameter bit ACTIVE_RESET = 1;
  bit       stop;

  initial begin
    stop = 0;
  end

  task run( int reset_hold = 2 ,
            int half_period = 10 ,
            int count = 0 );

    i.clk = 0;
    i.rst = ACTIVE_RESET;

    for( int i = 0; i < reset_hold;i++ ) begin
      tick( half_period );
      tick( half_period );
    end

    i.rst = !i.rst;

    for( int i = 0; (i < count || count == 0) && !stop; i++ ) begin
      tick( half_period );
    end
  endtask

  task tick( int half_period );
    # half_period;
    i.clk = !i.clk;
  endtask
endmodule
```

B.1.3 pins_if interface

The SystemVerilog pins_if interface defines the testbench interconnect for the various components that connect to our simple nonpipelined bus.

Module-based reference of an interface

Example B-3 SystemVerilog interface definition for pins_if

```
import avm_pkg::*;
import tb_tr_pkg::*;

interface pins_if( input clk , input rst );

  parameter int DATA_SIZE = 8;
  parameter int ADDR_SIZE = 8;

  bit       sel;
  bit       en;
  bit       write;
  bit [DATA_SIZE-1:0] wdata;
  bit [DATA_SIZE-1:0] rdata;
  bit [ADDR_SIZE-1:0] addr;

  modport driver_mp (
   input         clk , rst ,
   output        sel , en , write , addr ,
   output        wdata ,
   input         rdata
  );

  modport responder_mp (
   input         clk , rst ,
   input         sel , en , write , addr ,
   input         wdata ,
   output        rdata
  );

  modport monitor_mp (
   input         clk , rst ,
   input         sel , en , write , addr ,
   input         wdata ,
   input         rdata
  );

endinterface
```

B.1.4 tb_monitor module

The `tb_monitor` module, which is illustrated as the Assertion-Based Monitor in Figure B-1, is our assertion-based IP for the nonpipelined bus.

Example B-4 Assertion-based IP tb_monitor module

```
import avm_pkg::*;
import tb_tr_pkg::*;

module tb_monitor_mod(
  interface bus_if
);

  avm_analysis_port #(tb_transaction)
        cov_ap = new("cov_ap", null);
  avm_analysis_port #(tb_status)
        status_ap = new("status_ap", null);

  parameter DATA_SIZE  = 8;
  parameter ADDR_SIZE  = 8;

  tb_status status; // See Section B.1.7 (see page 287) for details

// Used to decode bus control signals

  bit  [ADDR_SIZE-1:0] bus_addr;
  bit  [DATA_SIZE-1:0] bus_wdata;
  bit  [DATA_SIZE-1:0] bus_rdata;
  bit                  bus_write;
  bit                  bus_sel;
  bit                  bus_en;

  bit                 bus_reset;
  bit                 bus_inactive;
  bit                 bus_start;
  bit                 bus_active;
  bit                 bus_error;

// Identify conceptual states from bus control signals

  always @(posedge bus_if.monitor_mp.clk) begin

   bus_addr  = bus_if.monitor_mp.addr;
   bus_wdata = bus_if.monitor_mp.wdata;
   bus_rdata = bus_if.monitor_mp.rdata;
   bus_write = bus_if.monitor_mp.write;

   bus_sel = bus_if.monitor_mp.sel;
   bus_en = bus_if.monitor_mp.en;

// Continued on next page
```

Example B-4 Assertion-based IP tb_monitor module

```
if (bus_if.monitor_mp.rst) begin
     bus_reset    = 1;
     bus_inactive = 1;
     bus_start    = 0;
     bus_active   = 0;
     bus_error   = 0;
   end
else begin
     bus_reset    = 0;

     bus_inactive = ~bus_if.monitor_mp.sel &
                    ~bus_if.monitor_mp.en;
     bus_start    =  bus_if.monitor_mp.sel &
                    ~bus_if.monitor_mp.en;
     bus_active   =  bus_if.monitor_mp.sel &
                     bus_if.monitor_mp.en;
     bus_error    = ~bus_if.monitor_mp.sel &
                     bus_if.monitor_mp.en;

    end
  end

 // ---------------------------------------------
 // REQUIREMENT: Bus legal states
 // ---------------------------------------------

 property p_reset_inactive;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
      $past(bus_reset) |-> (bus_inactive);
 endproperty
 assert property (p_reset_inactive) else begin
   status = new();
   status.set_err_trans_reset();
   status_ap.write(status);
 end

 property p_valid_inact_trans;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
     ( bus_inactive) |=>
       (( bus_inactive) || (bus_start));
 endproperty
 assert property (p_valid_inact_trans) else begin
   status = new();
   status.set_err_trans_inactive();
   status_ap.write(status);
 end

// Continued on next page
```

Example B-4 Assertion-based IP tb_monitor module

```
property p_valid_start_trans;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
      (bus_start) |=> (bus_active);
 endproperty
 assert property (p_valid_start_trans) else begin
   status = new();
   status.set_err_trans_start();
   status_ap.write(status);
 end

 property p_valid_active_trans;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
      (bus_active) |=>
        (( bus_inactive | bus_start));
 endproperty
 assert property (p_valid_active_trans) else begin
   status = new();
   status.set_err_trans_active();
   status_ap.write(status);
 end

 property p_valid_error_trans;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
      (~bus_error);
 endproperty
 assert property (p_valid_error_trans) else begin
   status = new();
   status.set_err_trans_error();
   status_ap.write(status);
 end

 // ---------------------------------------------
 // REQUIREMENT: Bus must remain stable
 // ---------------------------------------------

 property p_bsel_stable;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
      (bus_start) |=> ($stable(bus_sel));
 endproperty
 assert property (p_bsel_stable) else begin
   status = new();
   status.set_err_stable_sel();
   status_ap.write(status);
 end

// Continued on next page
```

Example B-4 Assertion-based IP tb_monitor module

```
property p_baddr_stable;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
      (bus_start) |=> $stable(bus_addr);
 endproperty
 assert property (p_baddr_stable) else begin
   status = new();
   status.set_err_stable_addr();
   status_ap.write(status);
 end

 property p_bwrite_stable;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
      (bus_start) |=> $stable(bus_write);
 endproperty
 assert property (p_bwrite_stable) else begin
   status = new();
   status.set_err_stable_write();
   status_ap.write(status);
 end

 property p_bwdata_stable;
   @(posedge bus_if.monitor_mp.clk)
    disable iff (bus_reset)
      (bus_start) && (bus_write) |=>
        $stable(bus_wdata);
 endproperty
 assert property (p_bwdata_stable) else begin
   status = new();
   status.set_err_stable_wdata();
   status_ap.write(status);
 end

 property p_burst_size;
   int psize;

  @(posedge bus_if.monitor_mp.clk)
      ((bus_inactive), psize=0)
  ##1 ((bus_start, psize++, build_tr(psize))
       ##1 (bus_active))[*1:$]
  ##1 (bus_inactive);
 endproperty

 cover property (p_burst_size);

// Continued on next page
```

Example B-4 Assertion-based IP tb_monitor module

```
function void build_tr(int psize);
   tb_transaction tr;

   tr = new();
   if (bus_write) begin
     tr.set_write();
     tr.data = bus_wdata;
   end
   else begin
     tr.set_read();
     tr.data = bus_rdata;
   end
   tr.burst_count = psize;
   tr.addr        = bus_addr;
   cov_ap.write(tr);
 endfunction

initial begin
   bus_reset = 0;
   bus_inactive = 0;
   bus_start = 0;
   bus_active = 0;
 end

endmodule
```

B.1.5 tb_env class

The environment class is the topmost container of an AVM testbench. It contains all the class-based components that comprise the testbench and orchestrate test execution of the testbench.

Example B-5 Testbench environment class

```
package tb_env_pkg;

 import avm_pkg::*;
 import tb_tr_pkg::*;
 import tb_transactor_pkg::*;

 class tb_env extends avm_env;

   protected tb_stimulus  p_stimulus;
   protected tb_driver    p_driver;
   protected tb_responder p_responder;
   protected tb_coverage  p_coverage;

   analysis_fifo #( tb_status ) p_status_af;
   avm_analysis_port #( tb_status ) p_status_ap;
   avm_analysis_port #( tb_transaction ) p_cov_ap;

   local process p_stimulus_process;

   virtual pins_if #( .DATA_SIZE(8), .ADDR_SIZE(8) )
             p_nonpiped_bus;

 // Environment class constructor

   function new(
      virtual pins_if #(.DATA_SIZE(8),.ADDR_SIZE(8))
                   nonpiped_bus,
      avm_analysis_port #( tb_transaction ) cov_ap,
      avm_analysis_port #( tb_status ) status_ap
   );

     p_nonpiped_bus = nonpiped_bus;
     p_cov_ap = cov_ap;
     p_status_ap = status_ap;

     p_driver = new(“driver ”, this);
     p_responder = new(“responder”, this);
     p_coverage = new(“coverage “, this);
     p_stimulus = new(“stimulus “, this);

     p_status_af = new(“status_fifo”);
   endfunction

// Continued on next page
```

Example B-5 Testbench environment class

```
   function void connect;

     p_stimulus.blocking_put_port.connect (
        p_driver.blocking_put_export
     );

     p_driver.m_bus_if = p_nonpiped_bus;
     p_responder.p_bus_if = p_nonpiped_bus;

     p_cov_ap.register(
        p_coverage.analysis_export
     );
     p_status_ap.register(
        p_status_af.analysis_export
     );

   endfunction

// Test Controller

   task run;
    fork
      begin
       p_stimulus_process = process::self;
       p_stimulus.generate_stimulus;
      end
      terminate_when_timedout;
      terminate_on_error;
      terminate_when_covered;
    join
   endtask

   task terminate_when_timedout;
    #200000;
    p_stimulus_process.kill;
    avm_report_message("terminate_when_timedout" , "" );
    $finish;
   endtask

   task terminate_on_error;
    string s;
    tb_status status;

    p_status_af.get( status );
    p_stimulus_process.kill;
    $sformat (s, "%s", status.convert2string);
    avm_report_error("terminate_on_error" , s );
    $finish;
   endtask

// Continued on next page
```

Example B-5 Testbench environment class

```
   task terminate_when_covered;
    wait( p_coverage.p_is_covered );
    p_coverage.report;
    p_stimulus_process.kill;
    avm_report_warning(“terminate_when_covered” , “” );
    $finish;
   endtask

  endclass

endpackage
```

B.1.6 tb_tr_pkg package

The `tb_tr_pkg` contains the class definitions for coverage transactions and error status, which is referenced in Example B-5.

Module-based reference of an interface

Example B-6 tb_tr_pkg package

```
package tb_tr_pkg;

  import avm_pkg::*;
  ‘include “tb_transaction.svh”
  ‘include “tb_status.svh”

endpackage
```

B.1.7 tb_transaction class

The `tb_transaction` class defines the nonpipelined bus address and data for both stimulus generation and coverage detection.

Example B-7 tb_transaction class

```
class tb_transaction extends avm_transaction;

  typedef enum {IDLE, WRITE, READ} bus_trans_t;

  rand bus_trans_t  bus_trans_type;
  rand bit[4:0]     burst_count;
  rand bit[7:0]     data;
  rand bit[7:0]     addr;


  function avm_transaction clone();
    tb_transaction t = new;
    t.copy( this );
    return t;
  endfunction

  function void copy( input tb_transaction t );
    data            = t.data;
    addr            = t.addr;
    burst_count     = t.burst_count;
    bus_trans_type  = t.bus_trans_type;
  endfunction

  function bit comp( input tb_transaction t );
    avm_report_message( t.convert2string , convert2string );
    return ((t.data == data) & (t.addr == addr) &
            (t.burst_count == burst_count) &
            (t.bus_trans_type == bus_trans_type));
  endfunction

  function string bus_transaction_type;
    if (bus_trans_type == IDLE)
      return ("IDLE ");
    else if (bus_trans_type == WRITE)
      return ("WRITE");
    else
      return ("READ ");
  endfunction

  function string convert2string;
    string s;
    $sformat( s , "Bus type = %s , data = %d , addr = %d" ,
              bus_transaction_type(), data , addr);
    return s;
  endfunction

  function bit is_idle;
    return (bus_trans_type == IDLE);
  endfunction

// Continued on next page
```

Example B-7 tb_transaction class

```
  function bit is_write;
    return (bus_trans_type == WRITE);
  endfunction

  function bit is_read;
    return (bus_trans_type == READ);
  endfunction

  function bit is_done;
    return (burst_count == 1);
  endfunction

  function void set_idle();
    bus_trans_type = IDLE;
    return ;
  endfunction

  function void set_write();
    bus_trans_type = WRITE;
    return ;
  endfunction

  function void set_read();
    bus_trans_type = READ;
    return ;
  endfunction

  function void set_data( bit [7:0] D );
    data = D;
    return ;
  endfunction

  function void set_addr( bit [7:0] A );
    addr = A;
    return ;
  endfunction

  function bit[7:0] get_data;
    return data ;
  endfunction

  function bit[7:0] get_addr;
    return addr ;
  endfunction

  function bit[4:0] get_burst_count;
    return burst_count ;
  endfunction

// Continued on next page
```

Example B-7 tb_transaction class

```
  function void decrement_burst_count;
    burst_count = burst_count - 1 ;
    return;
  endfunction
endclass
```

B.1.8 tb_status class

The `tb_status` class is broadcast through an `avm_analysis_port` to identify a specific nonpipelined bus error.

Example B-8 tb_status class

```
class tb_status extends avm_transaction;

  typedef enum { ERR_TRANS_RESET ,
                 ERR_TRANS_INACTIVE ,
                 ERR_TRANS_START ,
                 ERR_TRANS_ACTIVE ,
                 ERR_TRANS_ERROR ,
                 ERR_STABLE_SEL ,
                 ERR_STABLE_ADDR ,
                 ERR_STABLE_WRITE ,
                 ERR_STABLE_WDATA } bus_status_t;

  bus_status_t bus_status;

  function void set_err_trans_reset;
    bus_status = ERR_TRANS_RESET;
  endfunction

  function void set_err_trans_inactive;
    bus_status = ERR_TRANS_INACTIVE;
  endfunction

  function void set_err_trans_start;
    bus_status = ERR_TRANS_START;
  endfunction

// Continued on next page
```

Example B-8 tb_status class

```
  function void set_err_trans_active;
    bus_status = ERR_TRANS_ACTIVE;
  endfunction

  function void set_err_trans_error;
    bus_status = ERR_TRANS_ERROR;
  endfunction

  function void set_err_stable_sel;
    bus_status = ERR_STABLE_SEL;
  endfunction

  function void set_err_stable_addr;
    bus_status = ERR_STABLE_ADDR;
  endfunction

  function void set_err_stable_write;
    bus_status = ERR_STABLE_WRITE;
  endfunction

  function void set_err_stable_wdata;
    bus_status = ERR_STABLE_WDATA;
  endfunction

  function avm_transaction clone();
    tb_status t = new;
    t.copy( this );
    return t;
  endfunction

  function void copy( input tb_status t );
    bus_status = t.bus_status;
  endfunction

  function bit comp( input tb_status t );
    avm_report_message( t.convert2string , convert2string );
    return (t.bus_status == bus_status);
  endfunction

  function string convert2string;
    string s;
    $sformat( s , "Assertion error %s" , bus_status_type());
    return s;
  endfunction

  function string bus_status_type;
    return bus_status.name();
  endfunction

endclass
```

B.1.9 tb_transactor package

The `tb_transactor` package, which is referenced in Example B-5, includes all the SystemVerilog class-based components.

Example B-9 tb_transactor package

```
package tb_transactor_pkg;

  import avm_pkg::*;
  import tb_tr_pkg::*;

  'include "tb_stimulus.svh"
  'include "tb_driver.svh"
  'include "tb_responder.svh"
  'include "tb_coverage.svh"

endpackage
```

B.1.10 tb_stimulus class

The `tb_stimulus` class is responsible for generating random stimulus in our example testbench.

Example B-10 tb_stimulus class

```
class tb_stimulus extends avm_random_stimulus #( tb_transaction );

  function new( string name = "" ,
                avm_named_component parent = null );
    super.new( name , parent );
  endfunction

  task generate_stimulus( tb_transaction t = null ,
                                   input int max_count = 0 );

    tb_transaction m_temp;
    int burst_count = 0;

    if (t == null) t = new;

    for( int i = 0;
        (max_count == 0 || i < max_count);
        i++ ) begin

// Continued on next page
```

Example B-10 tb_stimulus class

```
// Get new random transaction type and burst count

      assert( t.randomize() );

      burst_count = t.get_burst_count();

      for ( int j = 1; j<=burst_count; j++) begin

// Keep same transaction type and burst count
// but randomize data and address within burst count loop

        t.data = $random % 256 ;
        t.addr = $random % 256 ;

        $cast( m_temp , t.clone() );
        blocking_put_port.put( m_temp );

        t.decrement_burst_count();
      end
    end
  endtask

endclass
```

B.1.11 tb_driver class

The `tb_driver` class is responsible for converting an untimed transaction into the appropriate timed pin activity defined by the nonpipelined bus protocol.

Example B-11 tb_driver class

```
class tb_driver extends avm_threaded_component;

  typedef enum {
    INACTIVE, START, ACTIVE, ERROR
  } tb_driver_state_e;

  virtual pins_if #( .DATA_SIZE( 8 ),
                     .ADDR_SIZE ( 8 ) ) m_bus_if;

// Continued on next page
```

Example B-11 tb_driver class

```
  avm_blocking_put_export #( tb_transaction )
      blocking_put_export;

  local tlm_fifo #( tb_transaction ) p_fifo;
  local tb_driver_state_e m_state;
  local tb_transaction m_transaction;

// Generate cycle accurate bus controls for protocol

  function new( string name ,
    avm_named_component parent );

    super.new( name , parent );

    p_fifo = new("fifo" , this );

    m_state = INACTIVE;
    blocking_put_export =
       new("blocking_put_export", this);

    set_report_verbosity_level( 400 );

  endfunction

  function void export_connections;
    blocking_put_export.connect
          ( p_fifo.blocking_put_export );
  endfunction

  task run;

    string m_trans_str;

    m_bus_if.driver_mp.en = 0;
    m_bus_if.driver_mp.sel = 0;

    forever begin
      @( posedge m_bus_if.driver_mp.clk );

      if( m_bus_if.driver_mp.rst == 1 ) begin

        m_bus_if.driver_mp.en <= 0;
        m_bus_if.driver_mp.sel <= 0;
        m_state = INACTIVE;

      end
      else begin

// Continued on next page
```

Example B-11 tb_driver class

```
// Conceptual state-machine to
// emulate bus protocol activity

        case( m_state )
          INACTIVE : begin

// Get stimulus generator output

            if( p_fifo.try_get ( m_transaction ) ) begin

// If not idle, set bus controls to transition to a START state

              m_bus_if.driver_mp.write <= 0;
              m_bus_if.driver_mp.en <= 0;

              if ( ~m_transaction.is_idle() ) begin
                m_bus_if.driver_mp.sel <= 1;
                m_state = START;
              end
              else begin
                m_bus_if.driver_mp.sel <= 0;
                m_state = INACTIVE;
              end

             m_bus_if.driver_mp.addr  <= m_transaction.get_addr() ;

// Set bus controls for write

              if ( m_transaction.is_write() ) begin
                m_bus_if.driver_mp.write <= 1;
               m_bus_if.driver_mp.wdata <= m_transaction.get_data()
;
              end
            end
            else begin
            avm_report_error (“DRIVER reqest_fifo.try_get failed” ,
                                “”);
            end
          end // INACTIVE

          START : begin

// Set bus controls to transition to an ACTIVE state

            m_bus_if.driver_mp.sel <= 1;
            m_bus_if.driver_mp.en <= 1;

            m_state = ACTIVE;

// Continued on next page
```

Example B-11 tb_driver class

```
          if (m_bus_if.driver_mp.write) begin
            avm_report_message("DRIVER sending      " ,
                          m_transaction.convert2string );
          end
        end // START

        ACTIVE : begin

          if (~m_bus_if.driver_mp.write) begin
            $sformat( m_trans_str ,
                    "Bus type = READ  , data = %d , addr = %d" ,
                    m_bus_if.driver_mp.rdata ,
                    m_bus_if.driver_mp.addr );

             avm_report_message("DRIVER receiving " ,
                                      m_trans_str );
          end

// Determine if burst is done

          m_bus_if.driver_mp.en <= 0;

          if( m_transaction.is_done() ) begin

// Set bus controls to transition to an INACTIVE state

            m_bus_if.driver_mp.sel <= 0;
            m_state = INACTIVE;

          end
          else begin

// Set bus controls to transition to a START state for burst

            m_bus_if.driver_mp.sel <= 1;
            m_state = START;

// Get next transaction

            if( p_fifo.try_get( m_transaction ) ) begin

             m_bus_if.driver_mp.addr  <= m_transaction.get_addr()
;

// Set bus controls for write

// Continued on next page
```

Example B-11 tb_driver class

```
                if ( m_transaction.is_write() ) begin
                  m_bus_if.driver_mp.write <= 1;
                  m_bus_if.driver_mp.wdata <=
                                        m_transaction.get_data()
;
                end
                else begin
                  m_bus_if.driver_mp.write <= 0;
                end

              end
              else begin
              avm_report_error("DRIVER reqest_fifo.try_get failed",
                                 "");
              end
            end
          end // ACTIVE
        endcase
      end
    end
  endtask

endclass
```

B.1.12 tb_responder class

The `tb_responder` class emulates the DUV and responds to bus activity initiated from the `tb_driver`, as illustrated in Figure B-1.

Example B-12 tb_responder class

```
class tb_responder extends avm_threaded_component;

  virtual pins_if #( .DATA_SIZE( 8 ), .ADDR_SIZE( 8 ) ) p_bus_if;

  function new( string name ,
    avm_named_component parent = null );

    super.new( name , parent );

    set_report_verbosity_level( 400 );
  endfunction

// Continued on next page
```

Example B-12 tb_responder class

```
  task run;

// Used to decode bus control signals

    localparam INACTIVE = 2'b00;
    localparam START    = 2'b10;
    localparam ACTIVE   = 2'b11;
    localparam ERROR    = 2'b01;

    string s_trans_str;

// Evaluate cycle accurate bus controls for protocol

    forever begin
      @( posedge p_bus_if.responder_mp.clk );

      if (p_bus_if.responder_mp.rst) continue;

// Conceptual state-machine to decode bus protocol transitions

      case( {p_bus_if.responder_mp.sel,p_bus_if.responder_mp.en} )
        INACTIVE : begin
        end // INACTIVE

        START : begin

// If read transaction, get generator output for transaction

          if (~p_bus_if.responder_mp.write) begin

// Get stimulus generator output for read transaction

            p_bus_if.responder_mp.rdata = $random % 256;

            $sformat( s_trans_str ,
                    "Bus type = READ  , data = %d , addr = %d" ,
                    p_bus_if.responder_mp.rdata,
                    p_bus_if.responder_mp.addr);

            avm_report_message("RESPONDER sending   " ,
                               s_trans_str );
          end
        end // START

        ACTIVE : begin

// If write transaction, print data and addr sent from driver

// Continued on next page
```

Example B-12 tb_responder class

```
          if (p_bus_if.responder_mp.write) begin
            $sformat( s_trans_str ,
                    "Bus type = WRITE , data = %d , addr = %d" ,
                    p_bus_if.responder_mp.wdata,
                    p_bus_if.responder_mp.addr);
            avm_report_message("RESPONDER receiving " , s_trans_str
);
          end
        end // ACTIVE
      endcase
    end
  endtask
endclass
```

B.1.13 tb_coverage class

The `tb_coverage` class is the coverage collector, which receives transactions detected by our assertion-based monitor as illustrated in Figure B-1, and uses a SystemVerilog `covergroup` construct to measure various bus burst lengths.

Example B-13 tb_coverage class

```
class tb_coverage extends avm_threaded_component;

  bit  p_is_covered;

  local bit [4:0] tsize;
  local bit       ttype;
  local tb_transaction t;

  covergroup p_size_cov;
    csize : coverpoint tsize;
    ctype : coverpoint ttype;
  endgroup

  analysis_fifo #(tb_transaction) af;
  analysis_if #(tb_transaction) analysis_export;

// Continued on next page
```

Example B-13 tb_coverage class

```
  function new( string name ,
                avm_named_component parent = null );
    super.new( name , parent );

    p_size_cov = new;
    tsize = 0;
    p_is_covered = 0;
    set_report_verbosity_level( 4 );
    af = new("trans_fifo",this);
  endfunction

  function void export_connections;
    analysis_export = af.analysis_export;
  endfunction

  task run;

    forever begin
      af.get(t);

      this.avm_report_message ("COLLECTOR receiving ",
                               t.convert2string);

      ttype = t.is_write();
      tsize++;

// If done bursting (burst_count == 1),
// sample final burst size and type

      if (t.get_burst_count() == 1) begin

        p_size_cov.sample;

        if( p_size_cov.get_inst_coverage > 75 ) begin
          p_is_covered = 1;
        end
        tsize = 0;
      end
    end
  endtask

  function void report;
    string report_str;

    $sformat( report_str , "%d percent covered" ,
              p_size_cov.get_inst_coverage );

    this.avm_report_message( "coverage report" , report_str );

   endfunction

endclass
```

B.2 OVM/AVM high-level reference guide

This section contains a high-level reference guide, which provides an overview description for many of the potentially less obvious OVM/AVM classes used in our complete testbench example. For a comprehensive list of all OVM/AVM classes, we suggest you reference the OVM/AVM Encyclopedia appendix in [Glasser et al., 2007], which documents all of the classes in the OVM/AVM library. For each class, the OVM/AVM Encyclopedia provides a description of the class and what it is used for along with a listing of all the members and methods.[1]

`avm_env` `extends avm_named_component`

A subclass of `avm_env` is the top-level class in any class-based AVM verification environment. All the components of the testbench are children of this top-level class.

`avm_named_component` `extends avm_report_client`

This is the fundamental building block of the AVM. All structural classes (for example, `avm_env`, `avm_threaded_component`, `avm_random_stimulus`, and so forth) inherit from `avm_named_component`.

```
avm_random_stimulus #(type
    trans_type=avm_transaction)
    extends avm_named_component
```

This is a general purpose unidirectional random stimulus generator. It is a useful component in its own right, but can also be used as a template to define other stimulus generators or extended to add stimulus generation methods to simplify test writing.

The `avm_random_stimulus` class generates streams of `trans_type` transactions. These streams may be generated by the `randomize()` method of `trans_type`,

1. The OVM is a superset of the 3.0 version of the AVM. The names in this appendix are discussed in terms of avm_* prefix, which are backwards compatible, versus the ovm_* names that are being proposed at the time of this writing.

or the `randomize()` method of one of its subclasses, depending on the type of the argument passed into the `generate_stimulus()` method. The stream may go indefinitely until terminated by a call to `stop_stimulus_generation()`, or you may specify the maximum number of transactions to be generated.

`avm_threaded_component` `extends avm_named_component`

A threaded component inherits from `avm_named_component` and adds the ability to spawn a `run()` task at the beginning of the simulation.

`avm_*_export` `#(type T=int) extends avm_port_base #(tlm_*_if #(T))`

An `avm_*_export` is a connector that provides interfaces to other components. It gets these interfaces by connecting to an `avm_*_export` or `avm_*_imp` in a child component.

`avm_*_port` `#(type T=int) extends avm_port_base #(tlm_*_if #(T))`

An `avm_*_port` is a connector that requires interfaces be supplied to it. It may get these interfaces by connecting to a parent's `avm_*_port` or an `avm_*_export` or `avm_*_imp` in a sibling.

`avm_analysis_port` `#(type T=int) extends avm_port_base #(analysis_if #(T))`

`avm_analysis_port` is used by a component such as a monitor to publish a transaction to zero, one, or more subscribers. Typically, it will be used inside a monitor to publish a transaction observed on a bus to scoreboards and coverage objects.

`avm_port_base``#(type IF=avm_virtual_class) extends IF`

`avm_port_base` is the base class for all ports, exports, and implementations (`avm_*_port`, `avm_*_export`, and `avm_*_imp`). `avm_port_base` extends `IF`, which is the type of the interface required or provided by the port, export, or implementation.

`analysis_fifo` `#(type T=int) extends tlm_fifo #(T)`

An `analysis_fifo` is a `tlm_fifo` with an unbounded size and a `write()` interface. It can be used any place an `avm_subscriber` is used. Typical usage is as a buffer between an `analysis_port` in a monitor and an analysis component (that is, a component derived from `avm_subscriber`).

`tlm_fifo` `#(type T=int) extends avm_named_component`

`tlm_fifo` is a FIFO that implements all the unidirectional TLM interfaces.

`analysis_if` `#(type T=int)`

The analysis interface is a nonblocking, non-negotiable, unidirectional interface. It is typically used to transfer a transaction from a monitor, which cannot block, to a scoreboard or coverage object.

`avm_transaction`

This is the base class for all AVM transactions.

BIBLIOGRAPHY

[Accellera OVL 2007] *Accellera Standard Open Verification Library Reference Manual*, 2007.

[Alur and Henzinger 1998] R. Alur, T. Henzinger, “Finitary fairness,” *ACM Trans. Program. Lang. Syst.*, 20, 6 Nov. 1998.

[AMBA 1999] *ARM, AMBA specification version 2.0*, May 1999.

[Bailey *et al.*, 2005] B. Bailey, G. Martin, T. Anderson, *Taxonomies for the Development and Verification of Digital Systems*, Springer, 2005.

[Bailey *et al.*, 2007] B. Bailey, G. Martin, A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*, Morgan Kaufmann, 2007.

[Bergeron *et al.*, 2006] J. Bergeron, E. Cerny, A. Hunter, A. Nightingale, *Verification Methodology Manual for SystemVerilog*, Springer, 2006.

[Chattterjee 2005] P. Chattterjee, “Streamline Verification Process with Formal Property Verification to Meet Highly Compressed Design Cycle,” *Proc. Design Automation Conference*, 2005.

[Cohen 2005] B. Cohen, S. Venkataramanan, A. Kumar, *SystemVerilog Assertions Handbook...for Formal and Dynamic Verification*, VhdlCohen Publishing, 2005.

[Dasgupta 2006] P. Dasgupta, *A Roadmap for Formal Property Verification*, Springer 2006.

[Ecker *et al.*, 2006] W., Ecker, V. Esen, M. Hull, “Execution semantics and formalisms for multi-abstraction TLM assertions,” *Proc. MEMOCODE*, 2006.

[Eisner and Fisman 2006] C. Eisner, D. Fisman, *A Practical Introduction to PSL*, Springer, 2006.

[eRM 2005] *Scalable Testbench White Paper*, Cadence Design Systems, 2005.

[Foster and Coelho 2001] H. Foster, C. Coelho, “Assertions Targeting A Diverse Set of Verification Tools,” *Proc. Intn’l HDL Conference*, March, 2001.

[Foster *et al.*, 2004] H. Foster, A. Krolnik, D. Lacey, *Assertion-Based Design*, 2nd *Edition*, Kluwer Academic Publishers, 2004.

[Foster *et al.*, 2006a] H. Foster, K. Larsen, M. Turpin, “Introducing The New Accellera Open Verification Library Standard,” *Proc. DVCon*, 2006.

[Foster *et al.*, 2006b] H. Foster, L. Loh, B. Rabii, V. Singhal, "Guidelines for creating a formal verification testplan," *Proc. DVCON*, 2006.

[Glasser *et al.*, 2007] M. Glasser, A. Rose, T. Fitzpatrick, D. Rich, H. Foster, *The Verification Cookbook: Advanced Verification Methodology (AVM)*, Mentor Graphics Corp, 2007. http://www.mentor.com/go/cookbook.

[Haque *et al.*, 2006] F. Haque, J. Michelson, K. Khan, *The Art of Verification with SystemVerilog Assertions*, Verification Central, 2006.

[I^{2}C 2000] *I^{2}C Bus Specification*, version 2.1, January 2000, http://www.semiconductors.philips.com/buses/i2c.

[IEEE 1800-2005] *IEEE Standard 1800-2005 SystemVerilog: Unified Hardware Design, Specification and Verification Language*, IEEE, Inc., New York, NY, USA, 2005

[IEEE 1850-2005] *IEEE Standard 1850-2005 Property Specification Language (PSL)*, IEEE, Inc., New York, NY, USA, 2005.

[ITRS 2003] *The International Technology Roadmap for Semiconductors, 2003 Edition*, Retrieved July 5, 2007 from the World Wide Web: http://www.itrs.net/Links/2003ITRS/Home2003.htm.

[Kariniemi and Nurmi 2005] H. Kariniemi and J. Nurmi, *Arbitration and Routing Schemes for on-Chip Packet Networks,* Springer, 2005, http://www.tkt.cs.tut.fi/kurssit/9636/K05/Chapter18.pdf.

[Keating and Bricaud 2002] M. Keating and P. Bricaud, *Reuse Methodology Manual*, Kluwer Academic Publishers, 2002.

[Long *et al.*, 2007] J. Long, A, Seawright, H. Foster, "SVA Local Variable Coding Guidelines for Effective Use," *Proc. DVCon*, 2007.

[Marschner 2002] E. Marschner, B. Deadman, G. Martin, "IP Reuse Hardening via Embedded Sugar Assertions," *Proc. International Workshop on IP-Based SoC Design*, 2002.

[Martin 2002] G. Martin, "UML for embedded systems specification and design: motivation and overview," *Proc. Design, Automation and Test in Europe*, 2002.

[OCP 2003] *Open Core Protocol Specification 2.0*, Document Revision 1.1, Part number: 161-000125-0002, 2003, www.ocpip.org.

[Piziali 2004] A. Piziali, *Functional Verification Coverage Measurement and Analysis*, Kluwer Academic Publishers, 2004.

[Richards 2003] J. Richards, "Creative assertion and constraint methods for formal design verification," *Proceedings of DVCon*, 2003.

[Ruah *et al.*, 2005] S. Ruah, A. Fedeli, C. Eisner, "Property-Driven Specification of VLSI design," Property Based System Design (PROSYD) methodology document FP6-IST-507219, 2005.

[Susantol and Melham 2003] K. W. Susanto1, T. Melham, “An AMBA-ARM7 Formal Verification Platform,” *Lecture Notes in Computer Science*, Springer, 2003.

[Vijayaraghavan and Ramanathan 2005] S. Vijayaraghavan, M. Ramanathan, *A Practical Guide for SystemVerilog Assertion*, Springer, 2005.

[Wilcox 2004] P. Wilcox, *Professional Verification: A Guide to Advanced Functional Verification*, Kluwer Academic Publishers, 2004.

Index

Continued from page ii