CHAPTER

6

ASSERTION PATTERNS

Patterns, which has emerged as a popular topic of discussion within the contemporary software design community, is a convenient medium for documenting and communicating design insight as well as design decisions (for example, design assumptions, structures, dynamics, and consequences). The origin of this notion is actually rooted in contemporary architecture (that is, the design of buildings and urban planning [Alexander 1979]). However, their descriptive problem-solving form also makes patterns applicable (and useful) across the broad and varied field of engineering. In this chapter, we introduce patterns as a system for documenting (in a consistent form) and describing commonly occurring assertions found in today's RTL designs. We propose a pattern format that is ideal as a quick reference for various classes of assertions, and throughout the remainder of the book we use it in our assertion descriptions. In addition, the format we propose is useful when documenting your own assertion patterns and increases their worth when they are shared among multiple stakeholders.

6.1 Introduction to patterns

A *pattern,* by definition, is an observable characteristic that recurs. Furthermore, it often serves as a form or model proposal for imitation.

In *Software Patterns,* James Coplien [2000] describes a pattern as follows:

> *I like to relate this definition to dress patterns. I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the*

right thing when you are finished The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself.

A succinct and intuitive definition of a *pattern,* in the context of conveying design intent, was provided by Appleton [2000]:

pattern definition

A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns.

Appleton goes on to say:

A pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. But a pattern does more than just identify a solution, it explains why the solution is needed!

Hence, you will see in the next section that a pattern format (sometimes referred to as a pattern language) provides a systematic and consistent method of documentation describing a *recurring solution* to a *recurring problem.*

6.1.1 What are assertion patterns?

Applying pattern-based approaches to present, codify, and reuse property specification for finite-state verification was originally proposed by Dwyer et al. [1998]. In this section, we build on the Dwyer discussion by categorizing the assertion patterns related to the three distinct phases of design, *specification, architect/design,* and *RTL implementation,* introduced in section 1.4 "Phases of the design process" on page 14.

property structure and rationale

The goal in creating an *assertion pattern* is to present two interdependent components (that is, *property structure* and *rationale*) about a particular characteristic or verification concern associated with a particular design. The property structure serves as a model for possible implementations, while the rationale tells you under what conditions the property should be used and examines the various trade-offs and variations. Their interdependence is essential, for without a property structure, the assertion pattern's rationale is superficial or meaningless, and without a rationale, the assertion pattern's property structure is perplexing and of little use.

Pattern categories. Just as specifying assertions can range from high-level system properties and block-level interfaces down to lower-level RTL implementation concerns—assertion patterns can be categorized in a similar fashion [Riehle and Zullighoven 1996].

categories of assertion patterns

- *Conceptual patterns* express higher-level global or system-level properties that concern large-scale components of an application domain.
- *Design patterns* are a refinement of the higher-level architectural patterns into medium-scale subsystem properties. Typically, these patterns describe block-level component relationships and their interfaces.
- *Programming patterns* are lower-level patterns specific to the RTL implementation. These patterns describe how to implement particular aspects or details of a component using the features of a given hardware description language.

Notice that these categories map into the three distinct phases of the design process, discussed in section 1.4 "Phases of the design process" on page 14. In other words, defining *conceptual patterns* is appropriate during the *specification phase,* while *design patterns* should be defined during the *architect/design phase,* and similarly *programming patterns* must be considered during the *RTL implementation phase.*

6.1.2 Elements of an assertion pattern

problem, solution, and context

Fundamental to a pattern is the format used to describe (document) the *solution* to a *problem* in a *context.* A number of pattern formats have been proposed, such as the *Alexander form* [Alexander 1977] and the *Gang of Four form* [Gamma et al. 1995]. Coplien [2000] provides a comprehensive survey of various proposed forms as well as a detailed description of the elements contained within a pattern form. Many pattern experts (and critics) will argue the advantages of one format over another. For our purposes, consistency in documenting and conveying the assertion intent is the primary goal. Hence, the pattern format we propose for assertions draws from the various formats to suit our needs. The pattern elements (or sections) we recommend are:

recommended pattern elements

Pattern name. This is important since it identifies an assertion solution and quickly becomes a part of the design team's vocabulary, which aids communication between engineers. We recommend a meaningful pattern name that is either a single word or short phrase.

Problem. Describe the problem to be solved. We recommend a concise statement, which helps engineers decide if this particular problem is applicable to their own (that is, whether to read further).

Motivation. Describe a scenario that illustrates the design problem.

Context. Describe (in a broader sense than the motivation section) the situations in which the problem recurs, and to which the solution applies.

Solution. Provide the details used to solve the stated problem. We recommend a solution that is detailed enough for the reader to know what to do, but general enough to be applied to a broader class of similar problems.

Considerations. This section identifies caveats for usage or suggests alternative patterns for certain situations. In addition, this section recommends alternative applications for the pattern or novel usage.

applying patterns

The remainder of this chapter discusses a number of common assertion patterns found in a typical RTL design. Hopefully, by reading through these examples, you will see that the power of this pattern format is its ability to clearly describe assertions. We also expect that the assertion patterns will aid in application of assertions for your logic.

6.2 Signal patterns

In this section we define a set of patterns related to signal use within an RTL model. One pattern that may not immediately come to the designer's mind, but does immediately affect design operation, is undriven inputs that evaluate to Z or signals derived from unconnected ports that evaluate to X. Other patterns related to signals include multi-bit range checks as well as one-hot checks and gray codes.

6.2.1 X detection pattern

Pattern name. X detection

Problem. Detect unconnected ports and undriven signals, as well as X assignment propagation.

Motivation. During RTL development (that is, initial coding or code modifications and edits) the engineer often leaves an unconnected input port to a module, defines a new variable without an assignment, or neglects to drive a signal within a testbench. The *X detection* pattern is useful for identifying and isolating this class of problem.

Context. In addition to unconnected signals, RTL modeling that contains X assignment and detection is problematic and should be avoided [Bening and Foster 2001].[1] Problems typically encountered include missing functional bugs associated with startup (that is, during the reset process) as well as reduced performance of the RTL simulation model. Nonetheless, the development of today's complex chips often involves multiple designers and verification engineers and a significant amount of design reuse (that is, internally- and externally-developed IP). Hence, IP consumers often have little or no control over the coding of the RTL they choose to use (that is, reuse). Detecting X or Z values on block boundaries can significantly reduce debug during system-level integration of multiple blocks or IP. The *X detection* pattern is useful for detecting X propagation as well as unconnected ports.

Solution. Detect unconnected bits, undriven bits, and X propagation in Verilog as follows:

```
^<expression> === 1'bX
```

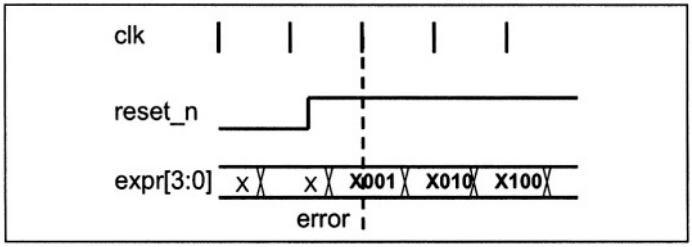
For example, using the OVL assert_never monitor, you can detect any bit for the Verilog expression `expr[3:0]` that is unconnected or undriven as follows:

Example 6-1 *OVL* X detection

```
assert_never invalid (clk, reset_n, ^expr[3:0] === 1'bX);
```

Figure 6-1 demonstrates that `expr[3]` is unknown after an active low reset.

Figure 6-1 expr[3] is undriven or unconnected



The OVL assertion in Example 6-1 quickly detects, isolates, and reports this problem. For instance, if the violation occurs at time 150 and is detected in the hierarchy `top.my_mod` (with the assert_never OVL example shown in Example 6-1), then the following default message prints during simulation:

1. Lint tools are superior to assertions for detecting unconnected signals and, in general, are the authors' recommended method for identifying these problems.

OVL_FATAL : ASSERT_NEVER : VIOLATION : : severity 0 : time 150 : top.my_mod. invalid

Note that you can override the default error message by creating a unique (that is, customized) error message for each instantiated OVL assertion module. The customized error message is passed into the instantiated OVL as a string through the *msg* parameter as shown in Example 6-2. Note, the first parameter in this example is the *severity level,* while the second parameter is an *options* parameter. For additional details on the OVL parameter values, see Appendix A.

Example 6-2 *OVL* X detection with customized message

```
assert_never #(1,0,"X detected")
  invalid (clk, reset_n, ^expr[3:0] === 1'bX);
```

The customized message in Example 6-2 is:

OVL_ERROR : ASSERT_NEVER : X detected : : severity 1: time 150 : top.my_mod.invalid

Refer to Appendix A for additional ways to customize OVL messages.

methodology guideline

Notice in this example that the *severity level* 1 changed the severity to ERROR. We recommend that only assertions associated with the testbench have a *severity level* of 0 (FATAL), which causes simulation to halt.

SystemVerilog **$isunknown** system task

Alternatively, detecting unconnected ports or signals that were assigned an X value is accomplished in SystemVerilog using the newly defined system task $isunknown(<*expression*>), which returns true if any bit of the *expression* is X or Z. This is equivalent to:

```
^<expression> === 1'bX
```

Example 6-3 demonstrates how to code the same OVL assertion shown in Example 6-1 using the SystemVerilog assert construct and the $isunknown system task.

Example 6-3 *SystemVerilog* undriven signal detection

```
always @ (posedge clk) begin
  if (reset_n)
    invalid: assert property (!$isunknown(expr));
end
```

Considerations. Instead of specifying an assertion for each input signal, you should group an appropriate set of signals and check them as a single assertion, as shown in Example 6-4

Example 6-4 *SystemVerilog* unknown check for multiple signals

```
`ifdef X_DETECTION

always @(posedge clk) begin
  if  (reset_n)
    x_prob: assert(!$isunknown({req,tras_start,addr,burst,we})) else
      $error("undriven cpu input signal req=%h trans_start=%h addr=%hburst=%h we=%h",
              req,  tras_start, addr, burst, we);
end

`endif
```

methodology guideline

For improved simulation performance, the engineer might consider bracketing all *X detection* checks with a Verilog `ifdef compiler directive. After the model has reached a stable point during verification (that is, X values are no longer propagating), disable these assertions.

6.2.2 Valid range pattern

Pattern name. Valid range

Problem. Ensure that a multi-bit signal or expression evaluates to a value within a valid min/max range.

Motivation. Signals (and expressions) within the RTL model may incorrectly evaluate to values that are not supported within the structure of the model. For example, consider a simple FIFO with a maximum depth of six elements. If we use a three-bit pointer to track the current number of valid elements contained within the FIFO, then the pointer should never evaluate to seven. In general, range checks are specific (and critical) to a given RTL implementation.

Context. Signals within many RTL control structures, such as counters, memory address circuits, and finite-state machines (FSM), are often limited to a specified valid range. In addition, datapath circuits often require variables or expressions to evaluate within the allowed range.

Solution. You can detect multi-bit variables or expressions outside of a valid minimum or maximum range by writing the following Verilog expression:

```
expr >= min_val & & expr <= max_val
```

PSL range check

For instance, in Example 6-5, we have written a PSL assertion to validate that a three-bit `fifo_depth` variable evaluates to a value between zero and six.[2]

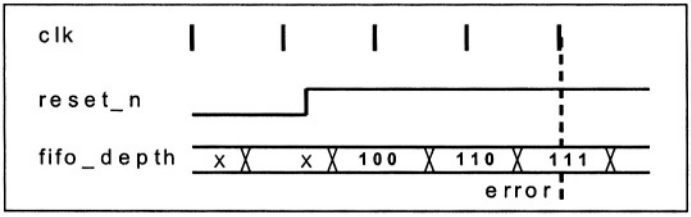
Example 6-5 *PSL* valid range check

```
assert always (fifo_depth < 7) @ (posedge clk) abort (!reset_n);
```

Notice the Verilog conditional expression in Example 6-5, which is used to prevent a false evaluation of the assertion during reset. In the next section, we demonstrate a simpler way to code assertions containing conditional expressions using the PSL implication operator.

Figure 6-2 illustrates a range violation that the PSL assertion in Example 6-5 would identify.

Figure 6-2 fifo_depth variable out of valid range zero thru six



OVL range check

The OVL assert_range monitor detects range violations. In Example 6-6, the monitor reports an error if the three-bit `fifo_depth` variable evaluates outside the *min/max* range. In this case, the *min* and *max* values (which are parameters in the OVL module) are zero and six, respectively.

Example 6-6 *OVL* valid range check

```
assert_range #(0,3,0,6) above_full (clk, reset_n, fifo_depth);
```

The assert_range assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock c*lk*. It contends that a specified *test_expr* will always have a value within a legal *min/max* range; otherwise, an assertion will fire (that is, an error condition will be detected in the code). The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using). The *min* and *max* should be a valid parameter and *min* must be less than or equal to *max*.

2. We do not check for the case of `fifo_depth` less than zero, since all Verilog unsigned registers are greater than or equal to zero.

OVL Verilog Syntax

assert_range [#(severity_level, width, min, max, options, msg)] inst_name (clk, reset_n, test_expr);

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*.
min	Minimum value allowed for range check. Default to 0.
max	Maximum value allowed for range check. Default to (2***width* - 1).
options	Vendor options.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

For additional details of the OVL **assert_range,** see Appendix A.

SystemVerilog range check

Alternatively, the range check previously demonstrated using the OVL in Example 6-6 for a three-bit `fifo_depth` variable can be expressed in SystemVerilog using the assert construct as shown in Example 6-7.

Example 6-7 *SystemVerilog* valid range check

```
// procedural assertion

always @ (posedge clk) begin
  if (reset_n)
    full: assert property (fifo_depth < 7) else
            $error ( "fifo_com Fifo64 Internal Failure, send mail to support@fifo.com." ) ;
end
```

Considerations. Valid range patterns are useful in any design where a physical address limit has been established on some fully addressable space.

6.2.3 One-hot pattern

Pattern name. One-hot

Problem. Ensure that no more than one bit of a multi-bit variable or expression is active high at a time (that is, all other bits are active low).

Motivation. Often signals in RTL designs have a specific or required encoding. For example, one-hot encodings are common in high-speed designs. A common use of one-hot encoding is associated with multiplexers, as shown in Example 6-8.

Note that if a condition occurs where multiple select bits are active high in Example 6-8, then the RTL model during simulation will not reflect the actual circuit behavior (that is, the first match within the casez statement will take effect). In fact, for some vendor ASIC multiplexer cells (such as a pass-through mux), the circuit can be damaged (that is, burn up) if more than one select line is active at a time.

Context. The one-hot pattern is most useful in control circuits. It ensures that the state variable of a finite state machine (FSM) implemented with one-hot encoding will maintain proper behavior (that is, exactly one bit is asserted high). In datapath circuits, one-hot checks ensure that the enabling signals of bus-based designs do not generate bus contention.

Example 6-8 one-hot multiplexer

```
module dmux4 (o, sel, i0, i1, i2, i3);
  parameter           WIDTH = 1;
  input  [WIDTH-1:0] i0, i1, i2, i3;  // input data
  input  [3:0]       sel;             // select signal
  output [WIDTH-1:0] o;               // output

  always @ (i0 or i1 or i2 or i3 or sel) begin
    casez (1'b1) // synopsys parallel_case
      sel[0] : o = i0;
      sel[1] : o = i1;
      sel[2] : o = i2;
      sel[3] : o = i3;
      default: $display ("No active select line on dmux4 %m");
    endcase
  end
endmodule
```

Solution. You can check for a *zero or one-hot* condition on a multi-bit variable *expr* by writing the following Verilog expression:

```
(expr & (expr - 1)) == 1'b0
```

Hence, detecting a pure one-hot condition can be expressed as:

```
(expr != 0) && ((expr & (expr - 1)) == 1'b0)
```

OVL one-hot check

Using the OVL assert_one_hot monitor, the four-bit select line `sel` in Example 6-8 can be validated as shown in Example 6-9.

Example 6-9 ***OVL*** **one-hot check**

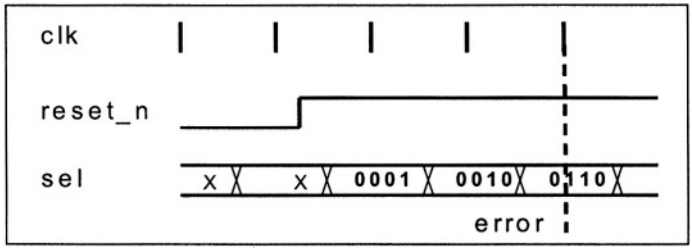
```
assert_one_hot  #(0,4) dmux4_one_hot ('TOP.clk, 'TOP. reset_n, sel);
```

methodology guideline

Notice the multiplexer circuit shown in Example 6-8 is unclocked. To prevent false firings due to the transient behavior of events in simulation, we have chosen to use a global clock defined in the top module of the design to sample the select signal (that is, `'TOP.clk`, where the `TOP` Verilog macro can be redefined to any appropriate top module in the heirarchy). Some designs define a special assertion clock and reset signal in the top module, which can be tuned to prevent false firings of assertions during simulation. This works well for simulation, but can create problems when synthesizing assertions into actual hardware checkers [Nacif et al. 2003] or formal tools.

Figure 6-3 demonstrates the one-hot violation that the OVL assertion in Example 6-9 would identify.

Figure 6-3 **dmux4 's' variable select one-hot violation**



Example 6-10 ***SystemVerilog*** **one-hot multiplexer check**

```
module dmux4 (o, sel, i0, i1, i2, i3);
  parameter           WIDTH = 1;
  input  [WIDTH-1:0]  i0, i1, i2, i3; // input data
  input  [3:0]        sel;            // select signal
  output [WIDTH-1:0]  o;              // output

  always @(i0 or i1 or i2 or i3 or sel) begin

    // procedural clocked assertion
    if ('TOP.reset_n)
      assert property (@(posedge 'TOP. clk) ($countones (sel) ==1))
        else $error ("dmux4 select line one-hot violation at %m");

    casez (1'b1) // synopsys parallel_case
      sel[0] : o = i0;
      sel[1] : o = i1;
      sel[2] : o = i2;
      sel[3] : o = i3;
    endcase
  end
endmodule
```

SystemVerilog $countones system task

Alternatively, we could code this assertion in SystemVerilog as shown in Example 6-10. This example uses the new $countones system task, which returns an integer count for the number of bits set to one.

The $countones system task evaluates a bit-vector expression as its input argument and returns an integer value that represents the total number of bits that were identified as a one in the expression.

Considerations. In addition to validating one-hot signals, such as a mux control set of signals or one-hot encoding of state machines, the *one-hot pattern* is useful for validating mutually exclusive events. For instance, assume we have a design that contains three separate memory controller finite state machines (FSMs). If only a single memory controller is permitted to be in a write mode state at a time, we could check this condition using the OVL assert_zero_one_hot monitor as follows:

Example 6-11 *OVL* assert_zero_one_hot 3-bit mutual exclusive event check

```
assert_zero_one_hot  #(0,3) mutex_wr_state (clk, reset_n,
      {FSM_1==`WR_STATE, FSM_2==`WR_STATE, FSM_3==`WR_STATE});
```

Example 6-11 parameterizes assert_zero_one_hot monitor to a width of three bits to check the mutual exclusivity of the three FSMs. Notice that a Verilog concatenation expression is created, with each bit representing one of the mutually exclusive events (that is, a current write mode status for each of the three FSMs).

6.2.4 Gray-code pattern

ensure proper gray-code encoding for queue pointers

Pattern name. Gray-code

Problem. Ensure that only a single bit changes value between clock transitions.

Motivation. To ensure data integrity when transferring queue pointers between different clock domains, often the pointer value is encoded with a gray-code. Hence, only a single bit of the gray-code encoded pointer is permitted to change per clock transition. This encoding helps identify possible skew issues between the multiple bits during an asynchronous transfer.

Context. There are a number of problems to be addressed when passing information across clocking domains. These are: metastability, fast to slow transfer (and vice versa), and timing skew between multiple bits. Two of these problems are solved by use of proper synchronization logic. The third problem can be

minimized (and identified) through gray-code encoding of the transferred multi-bit value.

Solution. To ensure proper gray-code transitions, we can specify an assertion as shown in Example 6-12.

Example 6-12 *SystemVerilog* assertion for gray-code encoding

```
property legal_graycode (code);
  @(posedge clk)  ($countones($past(code) ^ code)<=1));
endproperty
assert legal_graycode(async_ptr);
```

Note that the exclusive OR function computes the changing bits between the previous and current value of `code`.

6.3 Set patterns

In this section, we define assertion patterns that evaluate a group of signals or a bit-vector expression. The evaluation (that is, value) of this group of signals must be contained within a set of valid possible choices. Examples of *set patterns* include valid bus tags, valid opcodes, and any valid encoding defined within an RTL model.

6.3.1 Valid opcode pattern

Pattern name. Valid opcode

Problem. Ensure that an RTL bit-vector signal (expression) evaluates to a value that is contained within a set of possible legal values.

an ALU opcode must evaluate to a valid values

Motivation. In RTL design, multiple signals are often grouped by type, which permits information encoding. If the signal group evaluates to a value outside the set of legal values, then an error can occur within the design. Consider an opcode sent to an ALU. The ALU decodes the opcode to control its operations. If an invalid opcode value is decoded, it could result in an error condition or unpredictable behavior in the design (that is, if the ALU hardware doesn't trap the illegal case).

Context. The *valid opcode pattern* is useful in control or other circuits that contain a finite set of encoded commands or opcodes.

Solution. Ensure that an opcode evaluates to a valid set of values by writing an explicit expression to check each legal value. For example, assume a three-bit opcode was encoded with the following commands:

```
`define ADD=1
`define SUB=2
`define RD=3
`define WR=7
```

Hence, the values in the set (0, 4, 5, 6) are not valid. For this simple case, write a Verilog expression to validate the opcode values as follows:

```
(opcode==`ADD) || (opcode==`RD) ||
(opcode==`SUB) || (opcode == `WR)
```

OVL set check

Using the OVL assert_always monitor, you can validate the opcode encoding for this simple case as shown in Example 6-13:

Example 6-13 *OVL* valid opcode check

```
assert_always valid_opcode (clk,reset_n,
   (opcode==`ADD) || (opcode==`RD) || (opcode== `SUB) || (opcode==`WR));
```

SystemVerilog **inside** operator

Alternatively, check that a signal evaluates to a value contained within a set with the newly defined SystemVerilog inside operator. This operator compares a Verilog *expression* as its left side operator against a comma-separated list of expressions (*or constants.*) The task compares the value of the expression with the list of constants. If the expression evaluates to one of the arguments, then a 1 is returned. However, if the expression does not evaluate to one of the arguments, then a 0 is returned. Example 6-14 demonstrates a SystemVerilog assertion you can use to check for valid values of an opcode:

Example 6-14 *SystemVerilog* valid opcode check with inside

```
// concurrent assertion

valid_op: assert property (@(posedge clk) disable iff (reset_n)
              opcode inside {`ADD, `RD, `SUB, `WR} ) else
            $error("CTL sent illegal opcode (%0h) to ALU.", opcode);
```

Considerations. While the *valid opcode* pattern is useful for simple opcodes, there are times when a larger set of signals can produce an incongruous value, which would be hard to enumerate using the solution recommended by this pattern. The *valid signal combination* pattern demonstrates another technique for validating that a group of signals evaluates to a valid set of values.

6.3.2 Valid signal combination pattern

Pattern name. Valid signal combination

Problem. Ensure that a combination of signals evaluates to only legal (acceptable) values defined within a set.

Motivation. RTL designers frequently group multiple signals to convey a specific piece of information. In other words, the relationship between the set of signals must be consistent to convey a proper meaning. Incorrect combinations of signals can lead to inaccurate and unintended operations by the receiver. Consider a processor-to-memory interface with signals that describe `read`,`write`,`burst` (that is, cache line operation), `size` (that is, byte, halfword, word), and *write through* `wt` (that is, writing directly through to memory). For this set of signals, specific combinations are illegal, or only a few combinations may be legal.

Context. The *valid signal combination* pattern is useful when the combination of individual signals convey a particular meaning, which are then assembled and sent to another unit. Examples include bus interfaces, control interfaces, and status buses.

Example 6-15 *OVL* valid signal combination check

```
`ifdef ASSERT_ON

reg trans_ok;

always @(read or write or burst or size or wt)
  casez({read, write, burst, size, wt})
    6'b1_0_1_00_?, // cache (burst) read.
    6'b1_0_0_00_?, // single byte read
    6'b1_0_0_01_?, // halfword read
    6'b1_0_0_11_?, // word read
    6'b0_1_1_00_0, // cache (burst) write
    6'b0_1_0_00_0, // single byte write
    6'b0_1_0_01_0, // halfword write
    6'b0_1_0_11_0, // word write
    6'b0_1_0_00_1, // single byte writethru
    6'b0_1_0_01_1, // halfword writethru
    6'b0_1_0_11_1, // word writethru
    6'b0_0_0_00_0: // nothing,
                   trans_ok = 1'b1;
    default:       trans_ok = 1'b0;
  endcase

  // OVL assertion
  assert_always illegal_mem_req (clk, reset_n, trans_ok);

`ifdef ASSERT_OFF
```

Solution. To specify the valid values associated with the grouping of a set of signals, you should create a table for specifying the legal combinations of signal values. This table can be expressed using a Verilog case/casez statement or the SystemVerilog inside operator. The case/casez statement assigns a variable, which is then checked by an OVL monitor to identify any illegal combinations, as shown in Example 6-15.

Example 6-16 shows a SystemVerilog concurrent assertion that is used to check a processor-to-memory interface for a valid combination of signal values.

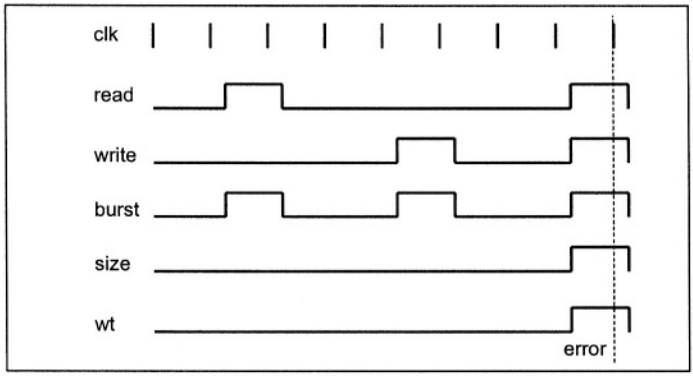
Note that on some lines, the casez *case_item* alternative contains a don't care matching character (? representing Z), which enables us to express the legal combinations in a compact form. This makes the tables regular and improves readability. In Example 6-16, the SystemVerilog inside operator enables us to achieve the same type of check as the casez structure demonstrated in Example 6-15.

Figure 6-4 illustrates the usefulness of checking illegal signal combinations on a CPU bus. For this example, a cache `read burst` precedes a cache `write burst`, which precedes an illegal active `read` and `write` signal. To reduce debug time, it is best to isolate illegal combinations close to the source of the error, as opposed to depending on the effect of the illegal combination to propagate to an observable point (for example, an output port).

Example 6-16 *SystemVerilog* legal signal combination check

```
//declarative assertion

assert property ( @(posedge clk)
  {read, write, burst, size, wt} inside
         {6'b1_0_1_00_?,   // cache (burst) read.
          6'b1_0_0_00_?,   // single byte read
          6'b1_0_0_01_?,   // halfword read
          6'b1_0_0_11_?,   // word read
          6'b0_1_1_00_0,   // cache (burst) write
          6'b0_1_0_00_0,   // single byte write
          6'b0_1_0_01_0,   // halfword write
          6'b0_1_0_11_0,   // word write
          6'b0_1_0_00_1,   // single byte writethru
          6'b0_1_0_01_1,   // half word writethru
          6'b0_1_0_11_1,   // word writethru
          6'b0_0_0_00_0})  // nothing.
  else $error ("Illegal memory request (read,write,burst,size,wt}=%0h",
                {read, write, burst, size, wt});
```

Figure 6-4 **Illegal read/write burst**



Considerations. This pattern is useful in many situations in your design to validate what you consider a legal state, illegal states, or valid input requests. The combination signals may be from dissimilar sources, which combine to produce an incoherent request. Sometimes, the legal combinations outnumber the illegal combinations, which means that an evaluation contained within the illegal might be simpler to express, as demonstrated by the *invalid signal combination* pattern.

6.3.3 Invalid signal combination pattern

Pattern name. Invalid signal combination

Problem. Ensure that a combination of signal values do not evaluate to values specified within a set of values.

Motivation. It is common to see combinations of two or more signals that should never be active at the same time. It is also common to have a grouping or set of signals, which should not evaluate to a combination of values. Defining these illegal relationships not only aids other readers by documenting expected behavior, it also reminds the designer of important characteristics of the design that must be considered (and not ignored) during future logic optimization or bug fixes.

Context. The *invalid signal combination* pattern is useful when applied to a grouping of individual signals, which are then assembled and sent on to another unit to convey a particular

meaning or control. Examples include bus interfaces, control interfaces, and status buses.

Example 6-17 *SystemVerilog* illegal signal combination check

```
// declarative assertion

assert property ( @(posedge clk) disable iff (reset_n)
   not ({read, write, burst, size[1:0], wt} inside
     {6'b1_0_1_01_?, 6'b1_0_1_10_?, // wrong size
      6'b1_0_1_11_?,
      6'b0_1_1_01_0, 6'b0_1_1_10_0, // wrong size
      6'b0_1_1_11_0,
      6'b0_1_1_00_1, 6'b0_1_1_01_1, // burst writethru
      6'b0_1_1_10_1, 6'b0_1_1_11_1,
      6'b1_0_0_10_?, 6'b0_1_010 _0})}// wrong size.
  else $error("Illegal memory request {read,write,burst,size,wt}=%0h ",
              {read, write, burst, size, wt});
```

Solution. Check for illegal signal value combinations by writing an expression that represents the combinations that should not occur. A table approach (like the previous *valid signal combination* pattern) is useful for larger sets of signals. Example 6-15 shows a valid CPU bus check that has been modified in Example 6-17 to check for the illegal signal value combinations.

Note that in this example the number of elements in the illegal set is the same as that expressed in the legal set. However, in many cases the illegal set is easier to express than the valid set.

For another simpler case of the *illegal signal set* pattern, consider an assertion for a design that contains a cache, as shown in Example 6-18. This cache has a read, write, invalidate, and a flush port. For this particular design, it is illegal for an active invalidate and a flush signal to occur at the same time:

Example 6-18 *SystemVerilog* illegal cache invalidate and flush request

```
// declarative assertion

assert property ( @(posedge clk) disable iff (reset_n)
    not (invalidate & flush) )
  else $error("Cache received illegal invalidate and flush request.");
```

This design may also forbid writing during invalidation or flushing (or both). The equation is easily extended to include more cases of illegal behavior.

Considerations. Additional illegal patterns may occur when considering combination of signals spanning across multiple cycles (for example, a previous cycle or a future cycle). See section 6.5 on page 185 to detect these types of illegal operations.

6.4 Conditional patterns

Conditioned logic is common in RTL design. For example, signals with names like `reset, enable, valid, ready, request, ack` and `done` often trigger conditions for process activation. In this section we demonstrate *conditional expression* patterns that utilize a controlling signal (or signals) to define precisely when a specific requirement must be checked. In addition, we demonstrate *sequence implication* patterns where the conditional event is a sequence of Boolean expression.

6.4.1 Conditional expression pattern

Pattern name. Conditional expression

Problem. Many designs use a `valid` or `enable` signal to indicate the proper time when information is available for processing. Hence, validating the correctness of the received data is dependant on the status of the conditional expression.

Motivation. Returning to the examples in Section 6.3.1 "Valid opcode pattern", we extend the valid opcode pattern to use a signal *valid* to indicate when the opcode is to be use to analyze the data coming to an ALU block.

Context. Apply the *conditional expression* pattern to any design containing enabled or conditional logic.

Solution. Check enabled or conditional logic by using the new SystemVerilog implication operator as shown in Example 6-19. For this example, legal opcodes are checked only when the *enable* signal is active.

Example 6-19 *System Verilog* conditional checking of valid opcode

```
// declarative assertion

assert property ( @(posedge clk) disable iff (reset_n)
    enable |-> opcode inside {1, 2, 3, 7})
  else $error("CTL sent illegal opcode (%0h) to ALU.", opcode);
```

In math, the implication operator consist of an *antecedent* that implies a *consequence* (for example, `A -> C,` which reads `A` implies `C`). If the antecedent is true, then the consequence must be true for the implication to pass. If the antecedent is false, then the implication passes regardless of the value of the consequence. Similarly, the SystemVerilog implication operator `|->` allows you

to state a Boolean expression or a prerequisite sequence as an antecedent. When the Boolean expression or prerequisite sequence is satisfied, then this implies that the consequence Boolean expression or suffix sequence must be satisfied.

Figure 6-5 Illegal opcode 5 to ALU

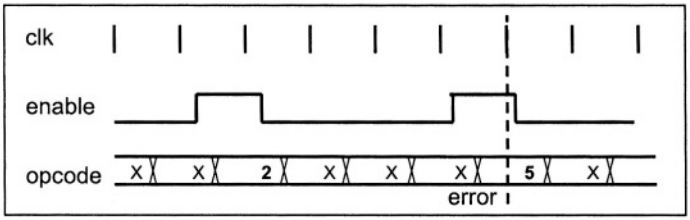


Figure 6-5 demonstrates an error case when an illegal opcode of 5 occurs at the same time an `enable` signal is active. This case could be caught with the *conditional expression* pattern assertion described in Example 6-19.

As another example, consider the case for the four-bit register (`reg_select`) that is expected to evaluate to a one-hot value whenever the `valid_read` signal is active, as shown in Example 6-20.

Example 6-20 *SystemVerilog* legal selection check during valid operation

```
// declarative assertion

assert property ( @(posedge clk) disable iff (reset_n)
     (valid_read |-> $countones (reg_select) ==1))
  else $error("REG block performed illegal register selection (%b)", select);
```

As another example of a *conditional expression* pattern, we re-code Example 6-15 to check for a valid transaction on a legal combination of signals using the implication operation, as shown in Example 6-21.

Finally, the X detection patterns presented in Section 6.2.1 might be coded with an implication if the check is dependant on a conditional expression as shown in Example 6-22.

Considerations. These examples demonstrate how to use the interaction of conditional expression within your design to validate inputs, outputs, and internal states. However, for some assertions, combinatorial interactions may be insufficient to specify the enabling condition. In Section 6.4.2 "Sequence implication pattern", we discuss assertions that consider sequence triggering conditions for implication.

Example 6-21 *SystemVerilog* check for valid transaction

```
// declarative assertion

assert property ( @(posedge clk) disable iff (reset_n)
   trans_start |-> {read, write, burst, size, wt} inside
         {6'b1_0_1_00_?,   // cache (burst) read.
          6'b1_0_0_00_?,   // single byte read
          6'b1_0_0_01_?,   // halfword read
          6'b1_0_0_11_?,   // word read
          6'b0_1_1_00_0,   // cache (burst) write
          6'b0_1_0_00_0,   // single byte write
          6'b0_1_0_01_0,   // halfword write
          6'b0_1_0_11_0,   // word write
          6'b0_1_0_00_1,   // single byte writethru
          6'b0_1_0_01_1,   // halfword writethru
          6'b0_1_0_11_1,   // word writethru
          6'b0_0_0_00_0}) // nothing.
  else $error ("Illegal request {read, write, burst, size, wt}=%0h",
               {read, write, burst, size, wt});
```

Example 6-22 *SystemVerilog* check for undriven data when valid

```
// declarative assertion

assert property ( @(posedge clk) disable iff (reset_n)
     (data_valid |-> !$isunknown(data[31:0]))
  else $error ("Undriven data bus (%h) during data return",
               data[31:0]);
```

6.4.2 Sequence implication pattern

Pattern name. Sequence implication

Problem. A combinational condition within a design (for example, a simple active `enable` or `valid` signal) may be insufficient to describe the triggering event required for a conditional pattern. A *prerequisite sequence* might be required as a triggering event for an assertion check.

Motivation. Bus protocols generally contain some kind of arbitration scheme where an active `grant` is generated (giving permission to use the bus) after an active `request`. Consider a system where a bus client generates a `request` signal, then eventually receives a `grant` signal, and finally is expected to activate a `start` signal to begin a transaction in the cycle immediately after the `grant`. In this system, the prerequisite sequence condition "*a request followed by a grant*" implies an

active `start` signal in the next cycle. The *sequence implication* pattern addresses this type of system.

Context. Apply the sequence implication pattern to complex protocols between a set of blocks, where a block's specified input sequence (that is, a prerequisite sequence) generates an expected result (that is, a suffix sequence). The *sequence implication* pattern may also be applied to inter-block communications where a sequence of internal states triggers a check for an expected result.

Solution. A complex event, such as a prerequisite sequence, can be used to define a triggering event for a design. When the triggering event occurs, then some other condition within the design must be valid. For example, we can write an assertion that specifies the following *sequence implication* pattern:

> Whenever an active *grant* is received—then on the following cycle, an active *start* must occur—provided that the initial *request* was never removed. In addition, the *grant* must be received within three cycles after the initial request.

Figure 6-6 illustrates the legal case where a bus request was made, as shown by the active `req` signal. The active `grant` occurs on the third cycle after the initial `req`. However, the bus request was removed prior to the start of the transaction, which means that an active `start` signal must not occur on the cycle immediately after the `grant`.

Figure 6-6 **Legal sequence implication pattern**

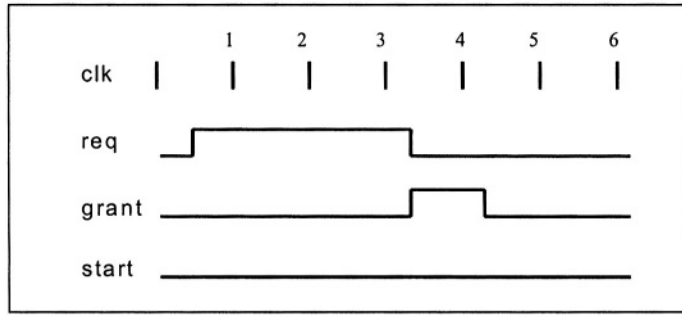
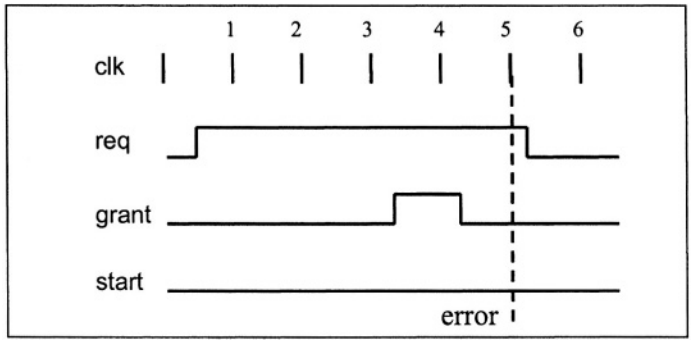


Figure 6-7 illustrates an illegal case where a bus transaction request was made as shown by the active `req` signal. The active `grant` is generated on the third cycle after the initial `req`. (that is, clock tick 4). However, since the initial `req` is still active, an error occurs at clock tick 5 since an active `start` did not occur on the immediate cycle after an active `grant`.

Figure 6-7 Illegal sequence implication pattern



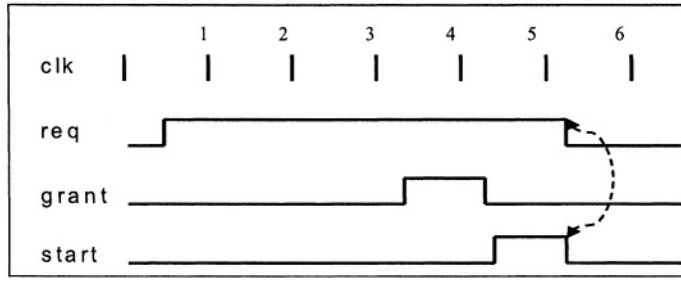
Example 6-23 demonstrates how to write a PSL assertion to specify the legal behavior described in the previous example.

Example 6-23 *PSL* `grant` to transaction request assertion

```
// declarative assertion

// A start can only occur after a grant for an active request

assert always ({{req[*1:4]}:{grant}; req} |-> {start}) @ (posedge clk);
```

For this example, we assert that if a prerequisite sequence (involving the `req` and `grant` signals) is satisfied, then an active `start` signal is required. To specify a time limit where an active `grant` occurs within four cycles after an initial `req`, we use the PSL sequence *fusion operator* (**:**). The fusion operator enables us to describe a sequence in which the ending event for the operator's left-hand sequence overlaps with the starting event for its right-hand sequence. Hence, for our example, we are specifying that an active `grant` overlaps with the last active `req` in a set of sequences of one, two, three, or four consecutive `reqs`. Figure 6-8 demonstrates an example where a `grant` overlaps the final cycle in a sequence consisting of four consecutive `reqs`, at clock tick 4.

Figure 6-8 Ending event in sequence overlaps with consequence



Note that Example 6-23 uses the PSL *weak suffix implication operator* (|->), which does not advance time between the *prerequisite* and *suffix sequences* (that is, there is no new clock tick). This enables us to specify that the ending cycle in the prerequisite sequence (that is, the final active `req` signal) must overlap with the starting cycle of the suffix sequence (that is, an active `start` signal). This situation is demonstrated in Figure 6-8 at clock tick 5.

See Appendix B for additional details on the PSL *weak suffix implication operator.*

Note that for the previous example, to assert that `grant` must occur within four cycles of an initial `req` (regardless of whether the `req` is removed), we could write:

Example 6-24 *PSL* `grant` timeout check

```
// declarative assertion

// assert grant is recieved within 3 clocks after an initial req

assert always ({rose(req)} |-> {{[*1:4]}:{grant}}) @(posedge clk);
```

The PSL sequence fusion operator in Example 6-24 enables us to specify a *set* of suffix sequences where a `grant` occurs on the first, second, third, or fourth cycle of one of the sequences.

For example, the fusion of the sequences `{{1[*1:4]}:{grant}}` expands into the following set of sequences:

```
(a)   {grant}
(b)   {1;grant}
(c)   {1;1;grant}
(d)   {1;1;1;grant}
```

Where the "1" represents *true,* which allows us to match a signal or Boolean expression and advance time to the next cycle. For case (a), where the `grant` occurs on the first cycle of the suffix sequence, the `grant` will actually overlap the initial `req` of the prerequisite sequence shown in Example 6-24. For case (d), where the `grant` occurs on the fourth cycle of the suffix sequence, the actual `grant` occurs on the third cycle immediately after the initial `req`.

Considerations. To apply sequence implication patterns to your design using the OVL, consider using the assert_next and assert_cycle_sequence monitors.

In addition to specifying bus protocol assertions, the *sequence implication* pattern is useful for specifying various control logic

assertions where a prerequisite sequence is represented as a valid progression of valid states and input values.

6.5 Past and future event patterns

Generally, a previous event within a system places requirements on the system's current state. Similarly, the current state in a system places a requirement on future events. The choice of specifying the relationship between a past or future event to a current state in the system is a matter of convenience. In this section, we introduce *past event* and *future event* patterns to describe these relationships.

6.5.1 Past event pattern

Pattern name. Past event

Problem. Detecting incorrect behavior for the current state of a system often depends on a previous event within the system.

Motivation. Cache protocols often have an *invalidate* command, which is used to mark the cache data as invalid for use (that is, due to a memory update somewhere in the system, the local cache data is no longer valid). If a cache read (that is, *hit*) occurs after an *invalidate,* then an error occurs (that is, there was an attempt to read invalid data). The *past event* pattern is useful for identifying invalid protocol errors.

Context. When designing an FSM, control circuit, or bus interface, a previous event of the system often influences the current state. These relationships must be validated.

Solution. Detect incorrect dependencies between a past event and the current state by referencing a previous combination of signal values within the design. System Verilog provides a means to access a previous combination of signal values within the verification environment through the use of the $past system function:

$past (*bit_vector_expr* [, *number_of_ticks*])

The *number_of_ticks* argument specifies the number clock ticks used to retrieve the previous value of *bit_vector_expr.* If *number_of_ticks* is not specified, then it defaults to one.

Example 6-25 demonstrates a *past event* pattern for the case where a cache `hit` must never occur when an `invalidate` occurred in the previous cycle.

Example 6-25 *System Verilog* past event pattern for illegal cache transaction

```
// declarative assertion

assert property ( @(posedge clk) disable iff (reset_n)
    not ($past(invalidate) & hit))
  else $error ("Cache hit occurred while previous invalidate active");
```

Example 6-26 demonstrates another *past event* pattern. For this example, the bus interface starts a memory transaction by activating a request (for example, a single pulse of a `req_valid` signal). Once this occurs, the address bus (`addr`) must not change its value until the new request occurs. The PSL code in Example 6-26 demonstrates how to apply this pattern to ensure that a memory address bus is stable:

Example 6-26 *PSL* past event pattern *to* check for stable signal

```
// declarative assertion

// assert that address will not change values after a request is made.

assert always (!req_valid -> prev(addr)==addr) @ (posedge clk);
```

Note that the assertion in Example 6-26 uses the PSL prev built-in-function to retrieve the value of `addr` from the previous clock cycle (defined by `@(posedge clk)`) in the same way as System Verilog $past system task demonstrated in Example 6-25 was used to retrieve the previous value of the `invalidate` signal.

Figure 6-9 Address changed without a request

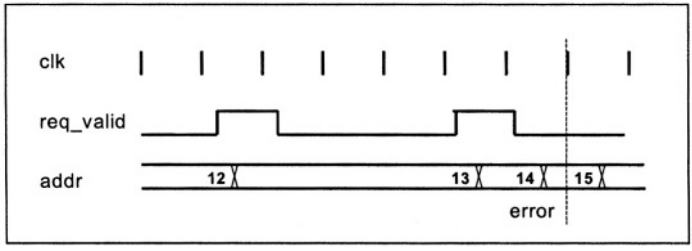


Figure 6-9 illustrates a failure for the assertion specified in Example 6-26.

methodology guideline

Considerations. Take care when coding assertions that use either the PSL prev built-in-function or System Verilog $past

system task to ensure that the past referenced value is valid at the current verification point in time (for example, you should not reference values prior to the start of simulation). Apply a *conditional* pattern to ensure that the past referenced time is a valid value.

6.5.2 Future event pattern

Pattern name. Future event

Problem. The current state of a system often places obligations or expectations that must be validated when some future event within the system occurs.

Motivation. Validating single-cycle and multi-cycle pulse widths on bus interfaces is critical for proper protocol behavior. For example, a protocol might place a requirement on a bus interface that a *request* signal must never be active for more than a single cycle.

Context. Bus interface protocols present many opportunities to apply the *future event* pattern. In addition, internal cycle-based timing relationships between multiple signals within the design (for example, `valid`, `flush`, and `restart`) are excellent *future event* pattern candidates.

Solution. The *future event* pattern example we present in this section is an extension of the *conditional* pattern discussed in Section 6.4. PSL, OVL, and SystemVerilog all provide a convenient means of specifying a requirement for a future event, which is dependant on the current state of the system. For example, use the PSL next operator to specify that the `req_valid` is always inactive during the cycle immediately after it was activated, as shown in Example 6-27.

Example 6-27 *PSL* future event pattern check for a single cycle pulse

```
// declarative assertion
// assert req_valid will never be active high for more than 1 cycle

assert always (req_valid -> next !req_valid) @(posedge clk)
              abort !reset_n;
```

The PSL assertion in Example 6-27 can be coded with an OVL assertion as shown in Example 6-28:

Example 6-28 *OVL* future event pattern check for a single cycle pulse

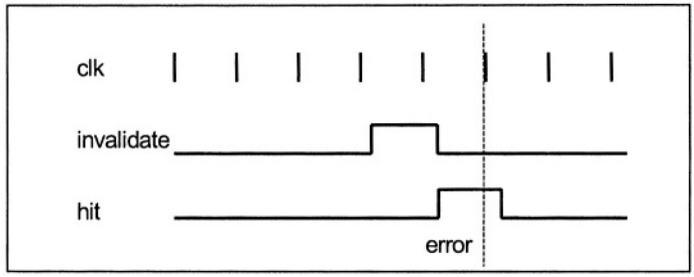
```
// assert req_valid will never be active high for more than 1 cycle

assert_next one_req (clk, reset_n, req_valid, !req_valid);
```

Example 6-29 and Figure 6-10 are examples of a *future event* pattern related to a cache protocol. For this example, whenever an active `invalidate` occurs, then a cache `hit` must never occur on the next cycle.

Example 6-29 *PSL* cache `invalidate/hit` check using the next operator

```
// declarative assertion
// assert that a cache hit never occurs after an invalidate

assert never ({invalidate; hit}) @(posedge clk);
```

Figure 6-10 Illegal cache `hit` after `invalidate`



Considerations. Validating a future event is not limited to specifying events on subsequent cycles. For example, we might have a requirement: the cache must never return a `hit` within four cycles after an `invalidate`. Example 6-30 demonstrates this multi-cycle future event pattern using a PSL repetition operator for the `hit` signal (that is, [*4]).

Example 6-30 *PSL* future event pattern check for 4 cycles of no `hit`

```
// declarative assertion
// assert cache will not return a hit within 4 cycles after invalidate

assert always ({invalidate} |=> {!hit [*4]}) @(posedge clk);
```

The PSL assertion in Example 6-30 and the OVL in Example 6-31 achieve the same purpose. That is, they specify a requirement for four inactive `hit` cycles after `invalidate`.

Example 6-31 *OVL* future event pattern check for 4 cycles of no `hit`

```
// declarative assertion
// assert cache will not return a hit within 4 cycles after invalidate

assert_cycle_sequence #(0,4) inv_hit
              (clk, 'TRUE, invalidate, (4{!hit}));
```

Example 6-32 also specifies a requirement for four inactive `hit` cycles after `invalidate`.

Example 6-32 *SystemVerilog* future event pattern check for 4 cycles of no `hit`

```
// declarative assertion
// assert cache will not return a hit within 4 cycle after invalidate
assert property (@ (posedge clk) disable iff (reset_n)
   (invalidate |=> !hit [*4]));
```

6.6 Window patterns

In this section, we define a set of patterns related to bounded events that reciprocally affect or influence each other. We refer to a bounded requirement on a transaction as a window, since the specification window bounds are defined by an initial starting event and conclude with either a specified time limit (that is, number of cycles) or an ending event. For example, within a window of time after an initial starting event, assert that a control signal must change its value. We discuss this type of assertion in the section for *time-bounded window* patterns. Alternatively, specify a range of time or window that terminates with an ending event. We discuss this type of assertion in the section for *event-bounded window* patterns.

6.6.1 Time-bounded window patterns

Pattern name. Time-bounded window

Problem. Ensure that logic in a design reacts to a transaction within a specified number of cycles.

Motivation. For performance reasons, or to satisfy a specified protocol requirement, often logic must be designed to react to a

transaction within a specified limit of time. For example, many protocols are initiated with a *request* and conclude with an *acknowledge* within a specified number of cycles. To facilitate rapid debug for these protocols, check for a maximum timeout condition on the *acknowledge* event. These assertions help isolate problems such as a stalling bus transaction caused by a deadlock.

Context. Apply the *time-bounded window* pattern in control circuits to ensure proper synchronization of events. Common usage includes:

- verify multi-cycle data operations with an enabling condition
- verify single-cycle operations with data loaded on different cycles
- verify synchronizing conditions that require stable data after a specified initial triggering event

Solution. Specify a time limit for a transaction by defining a sequence (or set of sequences) that limits the response recognition to a fixed number of cycles. For example, if a bus interface requires that an `ack` must occur within 100 cycles after a `req`, then the PSL assertion in Example 6-33 will validate that the response does not occur outside the time window.

Example 6-33 *PSL* time limit sequence check

```
// declarative assertion
// assert that ack must occur within 100 cycles after a req.

assert always (req -> next {{[*1:100]}:{ack}}) @(posedge clk);
```

Note that Example 6-33 uses the PSL *fusion* operator **(:)**, which allows us to specify a single-cycle overlap between the ending-cycle of the left-hand sequence and the starting-cycle of the right-hand sequence. Our objective is to specify a set of `ack` sequences ranging from a length of one cycle up to a limit of 100 cycles, which would satisfy the `req` implication. The first sequence regular expression (that is, `{[*1:100]}`), is a shortcut representation of this range:

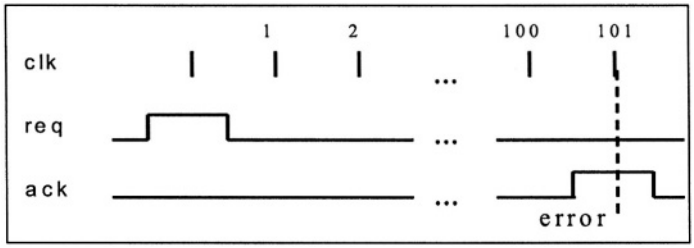
```
(1), {1;1}, {1;1;1}, {1;1;1;1}, {1;1;1;1},..
```

In this sequence, “1” followed by a semicolon allows us to advance the clock tick without any obligation (that is, we do not need to satisfy any particular Boolean expression when matching sequences at that particular point in time). Hence, fusing `{[*1:100]}` with `{ack}` enables us to match the following sequences:

```
{ack}, {1,ack}, {1,1,ack}, {1,1,1,ack}, . . .
```

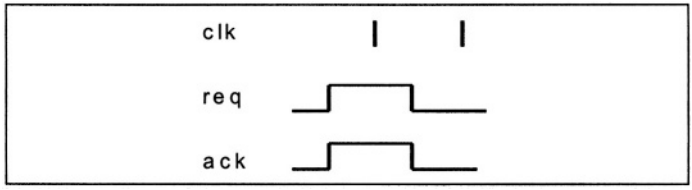
Figure 6-11 illustrates an error that would be detected by the PSL assertion specified in Example 6-24.

Figure 6-11 Time limit sequence error



For cases when the acknowledge can occur in the same cycle as the request, as shown in Figure 6-12, the previous assertion can be re-coded, as shown in Example 6-34, with the PSL *weak suffix implication operator* (`|->`). This operator permits an overlap of the ending cycle of the prerequisite sequence (that is, `req`) with the first cycle in the suffix sequence. Hence, we must define a suffix sequence of length 101, since the first cycle in the sequence potentially overlaps with the `req`, followed by up to 100 additional cycles in which `ack` could occur.

Figure 6-12 Overlapping `req` and `ack`



Example 6-34 *PSL* time limit sequence check with single-cycle overlap

```
// declarative assertion
// assert that ack must occur within 100 cycles after a req
// the ack can overlap with the req

assert always ({req} |-> {{[*0:100]}:{ack}}) @(posedge clk);
```

Use the OVL assert_frame monitor in Example 6-35 to validate a timeout condition, as previously specified in Example 6-34.

The assert_frame specifies a time window (that is, a frame), which is used to limited the response. See Appendix A for additional details for the OVL assert_frame.

Example 6-35 *OVL* time limit sequence check

```
// OVL assertion
// assert that ack must occur within 100 cycles after a req.

assert_frame #(0,0,100) req_ack (clk, 1, req, ack);
```

A SystemVerilog version of this assertion is shown in Example 6-34.

Example 6-36 SystemVerilog time limit sequence check

```
// declarative assertion

assert property ( @(posedge clk) disable iff (reset_n)
    (req |-> ##[1:100] ack)) else
  $error ("ack did not occur within 100 cycles after req");
```

Considerations. To apply time-bounded window patterns to your design using the OVL, consider using the following monitors: assert_change, assert_unchange, assert_frame, assert_width, assert_next, and assert_time.

methodology guideline

In the previous example, we specified a window to limit completion of an acknowledge for a transaction. This timeout window may be somewhat arbitrary, but is helpful for identifying situations with unfilled requests. By parameterizing (or macro-defining) the timeout associated with your assertion, you can tune the assertion to accurately diagnose where a given request times out and investigate this region.

6.6.2 Event-bounded window patterns

Pattern name. Event-bounded window

Problem. Ensure that logic in a design behaves correctly within an arbitrary window of time, which is bounded by a specified *starting event* and *ending event.*

Motivation. Consider a simple interface protocol example between two blocks in a design, where a *single* pulse of req initiates a transaction, followed eventually by an active ack. That is, one block pulses a request signal to initiate a transaction. The other block, after receiving the request, eventually returns an acknowledge pulse to complete the transaction. In our simple example, a protocol requirement is that the first block cannot send another request until the first transaction has completed. Apply the

event-bounded window pattern to this example to ensure proper behavior of the request signal.

Context. The *event-bounded window* pattern applies to protocol transactions verification or control logic involving data stability requirements, where the window of time for the transaction or data-stability is not explicitly stated, but is bounded by events within the design.

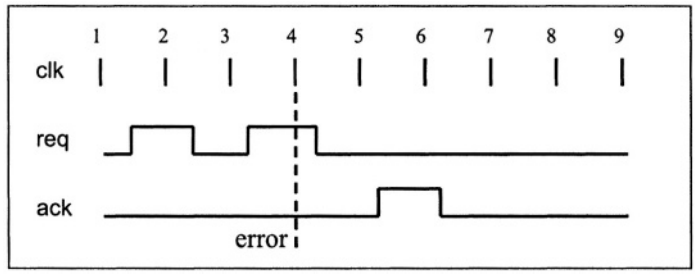
Solution. For protocols in which only a single transaction can be processed at a time, state a requirement that only a single pulse of `req` can occur prior to and including an `ack`. In other words, the `req` pulse defines an initial *starting event,* while the `ack` pulse defines the final *ending event.* These events bound an arbitrary window of time during which another active `req` signal must never occur. Example 6-37 demonstrates a PSL assertion for this *event-bounded window* pattern.

Example 6-37 *PSL* event-bounded window pattern

```
// declarative assertion
// a new request cannot start before the first one completes

assert always(req->next!req until_ack) @ (posedge clk);
```

Figure 6-13 demonstrates a failing case that the PSL assertion specified in Example 6-37 would catch. The SystemVerilog version of this assertion is demonstrated in Example 6-38.

Figure 6-13 Invalid multiple request.



Example 6-38 *SystemVerilog* event-bounded window pattern

```
// declarative assertion
// a new request cannot start before the first one completes

assert property (@ (posedge clk) disable iff (reset_n)
   (req |=> (!req throughout ack [-> 1])));
```

In general, the SystemVerilog *nonconsecutive exact repetition (goto repetition)* operator [-> 1] is intended to cover the semantic which matches the nth occurrence of an event (for example, [->2] would match (a, true, a), see Chapter 3 and Appendix C for additional details on the SystemVerilog repetition operators).

Considerations. To apply event-bounded window patterns to your design using the OVL, consider using assert_win_change assert_win_unchange, and assert_window . Also note that the *event-bounded window* pattern works best with non-overlapping starting events. It is problematic to associate a unique ending event with multiple unique starting events, unless each event has a unique ID or tag associated with it. In section 6.4.2 "Sequence implication pattern" on page 181 and section 6.7.4 "Pipelined protocol pattern" on page 199 we demonstrate how to handle overlapping transactions.

6.7 Sequence patterns

This section presents patterns involving sequences of Boolean expressions that span across multiple cycles. These patterns typically describe protocols (implied or expressed) or FSM transitions. They include *for bidden sequence, comparing captured data,* and *tagged transaction* patterns.

6.7.1 Forbidden sequence patterns

Pattern name. Forbidden sequence

Problem. Simple two-cycle (previous and current or current and next) combinations of signals should not occur within a design. Often there are multi-cycle combinations of events that can not occur.

Motivation. Consider a cache protocol with the following requirement: whenever an active `invalidate` occurs, then another cache `invalidate` or `hit` must never occur within the next four cycles. Often, it is easier to specify a forbidden sequence than specify all legal sequences.

Context. The following are example situations that provide opportunities to detect incompatible conditions that may lead to incorrect operations:

- Control logic that creates next cycle inputs that are illegal in specific states (flushing, invalidating, stalls, and so forth)
- Protocols that note illegal combinations of signals across a number of cycles
- State machines that have inputs that are illegal for a given relationship between current and next states

Solution. Use a forbidden sequence to identify the illegal occurrence of a `hit` and `invalidate` in the quiescent period.

Forbidden sequences allow a variable or fixed-width specification to be applied to expressions. The variable width specification allows us to use one sequence to account for the four cycles where a *hit* is not expected:

Example 6-39 *PSL* forbidden sequence check

```
// declarative assertion
// Once an invalidate occurs, neither hit or invalidate may be
// asserted for the next 4 cycles.

assert never ({invalidate; {invalidate || hit} [*1:4] })
   @ (posedge clk) ;
```

Considerations. Forbidden sequences can be specified using the OVL assert_cycle_sequence. For example, to specify that the sequence `A` followed by `B`, which is then followed by `C` should never occur, we can specify the forbidden sequence as a Verilog concatenation as follows:

```
{A, B, C, 1'b0}
```

If the prefix sequence is match (that is, `{A, B, C}`),then the assert_cycle_sequence will flag a failure since the last element in the sequence is defined as `1'b0`.

6.7.2 Buffered data validity pattern

Pattern name. Buffered data validity

Problem. Data can be dropped (that is, lost) or corrupted when attempting to transfer data across interfaces that involve latency.

Motivation. When transferring data between blocks on a shared bus, there is a risk of dropping data during a transfer. Hence, it is necessary to ensure that the received data matches the transferred data.

Context. This pattern is useful for protocols that transfer information (such as, data and address). It allows for assertions that describe correct operation without requiring additional RTL to capture the data.

Solution. The assertion in Example 6-40 captures the transfer data at the beginning of the transaction and compares the captured data with the received data at the end of the transfer.

Example 6-40 *SystemVerilog* captured data check

```
// declarative assertion
// Capture data (into tdata) and compare it when you see the new
// transaction (new_trans).

property capture_check;
  reg [31:0] tdata;
  @ (posedge clk) (new req, tdata=data |-> ##[1:100] new_trans
      ##0 tdata == trans_cmd);
endproperty

assert property capture_check)
  else $error ("Transaction (%0d)not started within 100 cycles, or trans_cmd (%0d) wrong.",
           new_trans, trans_cmd);
```

Considerations. This pattern applies to various control and datapath sections of logic. However, for some protocols, this pattern is insufficient. This is the case for protocols implementing pipelined transactions. See also Section 6.7.3 "Tagged transaction pattern" and Section 6.7.4 "Pipelined protocol pattern", which discuss techniques to handle overlapping transactions.

6.7.3 Tagged transaction pattern

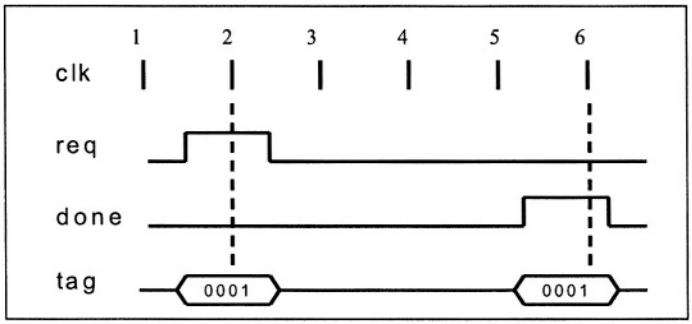
Pattern name. Tagged transaction

Problem. Ensure that transactions consisting of multiple out-of-order responses match the appropriate initial request.

Motivation. To improve throughput, many protocols allow multiple transactions to complete in an out-of-order fashion. For example, a tag (that is, unique ID) is often associated with a transaction's initial request, while another tag is associated with the transaction's ending response. To ensure that no data is lost while processing the transaction, it is necessary to validate that for any given tag associated with an initial request, there eventually exists an ending tag with the same ID value.

Consider the interface protocol illustrated in Figure 6-14. This interface can initiate a transaction with an active `req` and an accompanying four-bit request `tag` that contains a unique transaction ID. The transaction completes when `done` is asserted and the done `tag` equals the same value as the request `tag`.

Figure 6-14 Tagged transaction.



Context. The *tagged transaction* pattern is useful in bus protocols when there is a potential for different latencies to exist between multiple bus components, which results in an out-of-order response.

Solution. Validate the proper completion of a tagged transaction by creating a set of assertions that meet the following requirements:

- Ensure that a transaction is never lost (that is, completes within 100 cycles).
- Ensure that a unique tag is never reused (that is, multiple *requests* or *completions* for the same tagged transaction are not allowed).
- Ensure that there is one completion for each request.

Example 6-41 demonstrates how to code these assertions in SystemVerilog.

For an out-of-order transaction, a request tag must not be reused until after the original request (with the same tag) completes. Thus, our second property (`reqtag_once`) in Example 6-41 specifies that another request will not occur with the same tag before the acknowledge (`done`) is returned for the original request with the same tag. It is also required that only a single acknowledge is returned for a given tag. Hence, we use this same technique to check this condition. That is, when `done` is received, we specify that an additional `done` responses cannot be received for the same requested tag.

Example 6-41 *SystemVerilog* tagged `req/ack` protocol

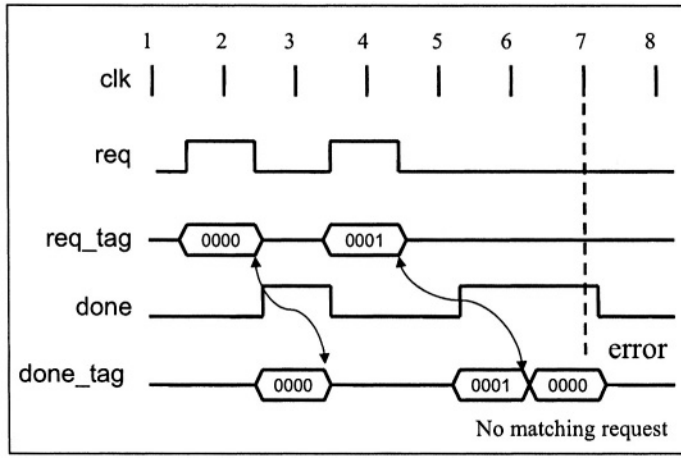
```
 . . .

// Property definitions.
// assert request 1 completes within 100 cycles.
property req2done;
   int rtag;
   @(posedge clk) (req, rtag=req_tag |->
      ##[1:100] done && done_tag == rtag);
endproperty

property reqtag_once;
   int rtag;
   // Once a request (with a specific tag) is made,
   // there must be a done with that same tag, before
   // another request with the same tag is issued.
   @ (posedge clk) not (req, rtag=req_tag
      ##1 !(done && done_tag == rtag) [* 1:$]
      ##0 req && req tag == rtag) ;
endproperty

property donetag_once;
   int dtag ;
   // Once a done (with a specific tag) is issued,
   // there must be a request with that same tag, before
   // another done with the same tag is issued.
    @(posedge clk) not (done, dtag=done tag
      ##1 !(req && req_tag == dtag) [* 1:$]
      ##0 done && done_tag == dtag);
endproperty

// Concurrent asertion statements.
assert property (req2done)
     else $error("Request tag didn't complete within 100 cycles." ) ;

assert property (reqtag_once)
     else $error("Request tag re-used before done received." ) ;

assert  property  (donetag once )
     else $error("Two acknowledges received for request tag.") ;
```

replicating a set of assertions

The forall construct in PSL conveniently enables us to replicate a set of assertions related to a unique tag as shown in Example 6-42.

In other words, the forall construct in Example 6-42 creates eight unique assertions associated with each tag, which are then used to validate the completion of the transaction within 100 cycles, and to validate that a given request tag will not be reused within 100 cycles.

Figure 6-15 **Done tag 0000 transaction error**



Example 6-42 ***PSL*** **multiple tagged** req/ack **protocol**

```
// declarative assertion
// assert any request 'n' completes (receives its done) within 100
// cycles.

assert
  forall T in {0:7} :
   always (req && tag==T -> next {[*1:99]; done && done_tag==T})
      @(posedge clk) ;

// Note this assertion does not ensure that there is only 1 done
// corresponding to the original request.
```

Considerations. The tagged transaction pattern is not appropriate for all protocols. For inorder protocols, where a given transaction must complete prior to the completion of the next transaction, apply the pipelined protocol pattern discussed in Section 6.7.4 "Pipelined protocol pattern".

6.7.4 Pipelined protocol pattern

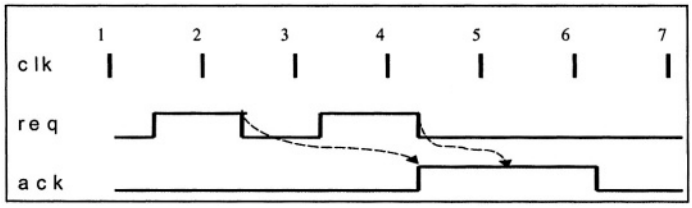
Pattern name. Pipelined protocols.

Problem. Due to the overlapping nature of inorder transactions, and the implicit ordering, it is difficult for assertions alone to associate a specific transaction completion event with the corresponding request. Hence, more than one request can falsely match a single transaction completion event.

Motivation. Pipelined protocols allow for greater throughput on an interface by allowing a limited number of additional transactions to begin before previous transactions complete. Yet, matching the appropriate transaction completion event with an appropriate request is problematic.

Consider a simple inorder handshake protocol for servicing a request. A `request` signal starts a transaction. An `acknowledge` signal completes the transaction. In general, there is a restriction on the maximum number of outstanding requests that are supported by the protocol. For this protocol, we require that each acknowledge completes the oldest initiated transaction. This strict ordering of transactions must be maintained, as illustrated in Figure 6-16.

Figure 6-16 REQ/ACK with FIFO in order semantics



Context. Pipelined (inorder) protocols are common on design interfaces as well as within control logic. Queue (FIFO) modules implicitly have this protocol, since they store and output (*push* and *pop*) data in the same order it comes in. The queue depth represents the maximum number of outstanding requests that the protocol can support.

Example 6-43 *PSL* and Verilog pipelined req/ack handshake protocol

```
// Setup two counters to tag the req's and ack's.
reg [3:0] req_cnt, ack_cnt;
initial {req_cnt, ack_cnt} = 8'b0;

always @ (posedge clk) begin
  // Increment counter each time event is seen.
  if (req) req_cnt <= req_cnt + 1;
  if (ack) ack_cnt <= ack_cnt + 1;
end

// declarative PSL assertion
// assert tagged req/ack sequence completes in 100 cycles

// For each tag.
assert
  forall C in {0:15} :
    always ({req && req_cnt == C} |=> {(*0:99]; ack && ack_cnt == C})
        @ (posedge clk) ;
```

Example 6.44 *SystemVerilog* pipelined_reqack module

```
module pipelined reqack // all arguments are inputs
  (         clk,
            req,
            ack,
             req_datain, //  Used to sample data at request time.
             dataout,    // Data exiting fifo to compare.
            latency,  // a constant expression
             pipedepth
   );

  // Setup two counters to tag the req' s and ack' s.
  reg [3:0] req_cnt = 4'b0,
            ack_cnt = 4'b0;

  // Update counters when req or ack event seen.
  always @ (posedge clk)
    begin
      if (req) req_cnt <= req_cnt + 1'b1;
      if  (ack)  ack_cnt  <=  ack_cnt  +  1'b1;
    end

  property req_no_ack;
     int cnt, req_data; // dynamic variables
     @ (posedge clk)
       (req, cnt=req_cnt, req_data=req_datain |->
          ## [1:latency] ack && ack_cnt==cnt
          ##0 req_data==dataout);
  endproperty

  property exceeded_depth;
    @ (posedge clk) (req_cnt - ack_cnt) < pipedepth);
  endproperty

  // We don't want to see an ack twice with the same tag without
  // a request with that tag between them.
  property no_extra_ack;
    reg [3:0] ackcnt; // dynamic variables
    @ (posedge clk) not (ack, ackcnt=ack_cnt
      ##1 !(req && req_cnt == ackcnt) [* 1:$]
      ##0 ack && ackcnt == ack_cnt);
  endproperty

  assert property (req_no_ack)
    else $error ("Ack not received within timeout limits %0d or ack_check_seq failed. " ,
         latency) ;
  assert property (exceeded depth)
    else $error("Exceeded pipedepth of interface.") ;
  assert  property  (no_extra_ack)
    else $error ("Additional ack not matching req." ) ;
endmodule
```

Solution. Apply the *pipelined protocol pattern* to a design to ensure the proper completion of `req-ack` pairs as demonstrated in Example 6-43. For this example, we create two counters `req_cnt` and `ack_cnt`, used to `'tag'` each request and acknowledge with a matching number. This enables us to support up to sixteen outstanding requests. At every `req` and `ack` event,

we increment the appropriate counter. The assertion we write specifies that for all active `req` (and the associated request count) there is an active `ack` whose acknowledge count matches the request count. Note that this example can support up to 16 outstanding requests.

Considerations. This *pipeline protocol pattern* can be extended to check transferred data. For example, if you are checking a FIFO, you can validate that data placed into a FIFO is read out in the correct order. In Example 6-44 we demonstrate how to code a SystemVerilog module, which can be instantiated in the SystemVerilog RTL to validate correct ordering of data.

This module is not limited to validating FIFOs. Due to its storage capabilities, it is also useful for validating a bus interface that permits overlapping transactions and can be used on blocks that buffer transfer data. This is useful for designs where the output transmission time is unpredictable due to the variety of events.

6.8 Applying patterns to a real example

The patterns we discussed in the previous sections provide multiple examples for effectively applying assertions. In this section, we explore an *assertion-based design* process with a real example, an SRAM controller. The design process begins with the engineer reviewing a design specification and refining the set of requirements into an RTL model that is ultimately synthesized into a gate-level implementation.

Let's consider the Verilog-2001 module interface shown in Example 6-45:

methodology guideline

The best process for adding assertions to a module is the following:

- Add assertions to each interface of a module. These assertions help to define the interface protocol, legal values, required sequencing, and so forth.
- Add assertions between interfaces of a module. These assertions help to define how the module operates on information from its interfaces and what it is supposed to do.
- Add assertions as you code structures within your module defining design intent (that is, acceptable operating conditions). This will identify simple mistakes due to incorrect internal operations.
- Add assertions to the control logic you implement that ties the interfaces, structures, and remaining logic.

Example 6-45 SRAM module Interface definition

```
module sramInterface
 #(
   parameter ADDRW = 16, // Number of address bits.
   parameter QueDepth = 6 // 2 <= QueDepth <= 16
  )
  (
  localparam IWB = ADDRW-2; // Upper address bit.
/*
  The abbreviated specification of this sram controller is the
  following:
  Provide a queued interface to a sram that accepts requests for
  reads/writes, but will not accept writes when previous reads are
  queued. Requests are issued to memory and completed when the memory
  sends memory done.
*/
input           clk;        // The clock
input           rst_n;      // The reset

// SM interface to Queue.
input           SMQueNew;   // New request on SM interface.
input   [IWB:1] SMQueAddr;  // Address of request
input           SMQueSpec;  // speculative request (only with reads).
input   [3:0 ]  SMQueWrEn;  // Write enables (0000,0011,1100,1111 valid)
input   [31:0]  SlvWrData;  // Data written with enables (each byte)

output  [IWB:2] QueBufAddr; // Address to be completed on ENG interface
output          QueBufValid;// Read request done.

// ENG interface from Queue to memory.
output          EngMemRd;    // Valid read request.
output          EngMemWr ;   // Valid write request.
output  [IWB:2] EngMemAddr;  // ENG address for operation.
output  [63:0]  EngMemData;  // ENG write data
output  [ 3: 0] EngMemWrEn;  // ENG write enables (each word).
input           MemEngDone;  // Request is complete from memory.

// Queue status bits.
output          QueAlmostFull;    // One more entry available.
output          QueFull;          // No more entries can be accepted.
output         ReadExistsInQue; // Writes not sent to Q until no reads.

// Queue management interface.
input          Flush;       // Hold flush to flush the Q. Accept no new
output         FlushAck;    // until FlushAck asserted-then clear Flush
input           SMQueStop;    // Hold Stop to empty Q.
output   reg    QueSMStopAck;// Ack will occur when Stop and Q empty.

endmodule
```

In Section 6.8.1 "Intra-interface assertions" we discuss and provide example assertions that apply *to* each of four interface types. Then in Section 6.8.2, “Inter-interface assertions”, we offer assertion examples that apply *across* interfaces to validate that the block is performing correctly. Each example includes the specific

pattern name we used above and offers an application that solves particular assertion requirements.

6.8.1 Intra-interface assertions

We begin by discussing intra-interface assertions. These apply to the following four interface types:

an interface is a set of signals that implement a protocol or transmit information

- New request interface
- SRAM interface
- Queue status interface.
- Queue management interface.

6.8.1.1 New request interface

The *new request interface* defines a valid signal (SMQueNew) and several data signals that define the request. Write the following assertions for this purpose:

Example 6-46 *SystemVerilog* During a request, all signals must evaluate to 1 or 0

```
// X detection pattern

assert property (@ (posedge clk) disable iff ( rst_n )
    (SMQueNew | ->
        !$isunknown(SMQueAddr, SmQueSpec, SMQueWrEn, SlvWrData)))
  else $error ("Unknown signal value when asserting a new request.");
```

Example 6-47 *SystemVerilog* SMQueSpec must not be asserted during a write request

```
// invalid signal combination pattern

assert property (@ (posedge clk) disable iff ( rst_n )
    not (SMQueNew & SmQueWrEn & SmQueSpec))
  else $error ( "Received illegal request for speculative (SMQueSpec) write." ) ;
```

Example 6-48 *SysttemVerilog* Only certain enable patterns; allowed during write request

```
// conditional pattern

assert property (@(posedge clk) disable iff ( rst_n )
   (SMQueNew |->
      SMQueWrEn[3:0] inside {4'b0000, 4'b001l, 4'bll00, 4'bllll))
  else $error ("SmQueWrEn has illegal write enable pattern %b.",
            SmQueWrEn);
```

6.8.1.2 SRAM interface

The SRAM interface also specifies two signals (`EngMemRd` and `EngMemWr`) that validate the request. Assertions for this interface are:

Example 6-49 *SystemVerilog* Unknown signals not allowed during a valid `EngMemRd` or `EngMemWr` request

```
// X detection pattern

assert property (@(posedge clk) disable iff ( rst_n )
     (EngMemRd | EngMemWr |->
           !$isunknown(EngMemAddr, EngMemData, EngMemWrEn)))
  else $error("Sram interface contains unknown values.");
```

Example 6-50 *SystemVerilog* `EngMemRd` and `EngMemWr` are mutually exclusive

```
// valid signal combination pattern

assert property (@ (posedge clk) disable iff ( rst_n )
    not (EngMemRd & EngMemWr)) else
  $error("Sram interface asserts illegally read and write together.");
```

Example 6-51 *SystemVerilog* For a vaild write, the `EngMemWrEn` may only have one of seven legal values

```
// valid combination of signals pattern

assert property (@(posedge clk) disable iff ( rst_n )
     (EngMemWr |-> EngMemWrEn inside {0,1,2,3,4,8,12})) else
  $error("Sram interface asserts illegal write en (%0b)", EngMemWrEn);
```

Example 6-52 System Verilog For a request, the completion signal MemEngDone must eventually be returned

```
// pipeline protocol pattern
// See pipelined_reqack templete definition in Example 6-44

// Make sure queued read is completed by memory (with MemEngDone).
// Atmost 6 requests (pipedepth) may be queued.
// Limit latency for return to 100 cycles.

pipelined_reqack
   sendReadReq(.req(EngMemRd) , .req_datain(),
               .ack(MemEngDone),
               .dataout(),
               .clk(clk),.latency(100),.pipedepth(6));
```

The assertion in Example 6-52 helps define the interface by specifying that the module can send a new request during the same cycle that MemEngDone is returned (to decrease latency). By using the assertion to show that the overlap is possible, designers get more of the type of information that is necessary to complete the design.

6.8.1.3 Queue status interface

The queue status interface relays the state of the queue to the interfacing modules. Assertions for this interface are:

Example 6-53 *SystemVerilog* QueFull and QueAlmostFull are mutually exclusive

```
// valid signal combination pattern

assert property (@ (posedge clk) disable iff ( rst_n )
     not (QueFull & QueAlmostFull))
   else $error ("Queue interface illegal state Quefull and QueAlmostFull.");
```

Example 6-54 *SystemVerilog* The queue status must not contain unknown values

```
// X detection pattern

assert property (@ (posedge clk) disable iff ( rst_n )
     $isunknown(QueFull, QueAlmostFull, ReadExistsInQue))
   else $error("Queue status interface has unknown values.");
```

6.8.1.4 Queue management interface

The queue management interface gives control to the other blocks to direct the shutdown of the interfaces. Assertions for this interface are:

Example 6-55 ***SystemVerilog*** **A** `Flush` **must remain active until** `FlushAck`

```
// event-bounded window pattern

assert property (@ (posedge clk) disable iff ( rst_n )
     ($rose(Flush) |-> (Flush throughout (##[0:100] FlushAck)))) else
  $error("Flush must be held until Ack asserted.");
```

Example 6-56 ***SystemVerilog*** `FlushAck` **must be asserted for no more than one cycle**

```
// forbidden sequence pattern

assert property (@ (posedge clk) disable iff ( rst_n )
    not (FlushAck ##1 FlushAck)) else
  $error ("FlushAck must be a single pulse.");
```

Example 6-57 ***SystemVerilog*** **A** `SMQueStop` **must remain active until** `StopAck` **is asserted**

```
// event-bounded window pattern

assert property (@ (posedge clk) disable iff ( rst_n )
      ($rose(SMQueStop) |->
          (SmQueStop throughout ##[0:100] StopAck)))
   else $error("Stop must be held until ack is asserted.");
```

Example 6-58 ***SystemVerilog*** `StopAck` **must be asserted for no more than one cycle**

```
// forbidden sequence

assert property (@ (posedge clk) disable iff ( rst_n )
    not (StopAck ##1 StopAck)) else
  $error ("StopAck must be a single pulse.");
```

Example 6-59 ***SystemVerilog*** `FlushAck` **and !**`Flush` **are mutually exclusive**

```
// valid signal combination pattern

assert property (@ (posedge clk) disable iff ( rst_n )
    not (FlushAck & !Flush)) else
  $error("FlushAck must not assert without Flush asserted.");
```

Example 6-60 ***SystemVerilog*** **When a `Flush` Is requested, the `Flush` must be acknowledged within 100 cycles**

```
// time-bounded combination pattern

assert property (@(posedge clk) disable iff ( rst_n )
    ($rose(Flush) |-> ##[1:100] IntFlushAck))
  else $error("Flush took too long (>100 cycles) to complete.");
```

The assertions shown above are applied to individual interfaces to a block. Next, we'll look at assertions that are written *across* interfaces.

6.8.2 Inter-interface assertions

Since an interface is typically driven from a single block, errors on the interface can be directly attributed to this one source. However, the assertions in this section comprise a set of inter-interface assertions. They interact with several interfaces to further check for correct operation or check for proper operation of the block given the data received. The assertions are:

Example 6-61 ***SystemVerilog*** **When there Is a new valid read request, the address must occur on the SRAM Interface within 100 cycles, and the SRAM address must match the request address**

```
// pipelined-protocol pattern

// See pipelined_reqack templete definition in Example 6-44

pipelined_reqack
   sendReadReq(.req(SMQueNew & !SMQueWrEn), .valin(SmQueAddr),
               .ack(EngMemRd),
               .dataout(EngMemAddr),
               .clk(clk), .latency(100), .pipedepth(6));
```

Example 6-62 ***SystemVerilog*** **When there Is a new valid read request, there must be a completion of this read `QueBufValid`**

```
// pipelined-protocol pattern

// See pipelined_reqack templete definition in Example 6-44

pipelined_reqack
   receiveData (.req(SmQueNew & !SMQueWrEn), .req_datain(),
                .ack(QueBufValid),
                .dataout(),
                .clk(clk), .latency(100), .pipedepth(6));
```

Example 6-63 ***SystemVerilog*** **When there is a new valid write request, the address and data must occur on the SRAM Interface within 100 cycles, and the SRAM address and data must match the request**

```
// pipelined-protocol pattern

// See pipelined_reqack module definition in Example 6-44

pipelined_reqack
   sendWriteReq(.req (SMQueNew & |SMQueWrEn),
                .req_datain ({SlaveWrData, SmQueAddr}),
                .ack(EngMemWr),
                .dataout ({EngMemData, EndMemAddr}),
                .clk(clk), .latency(100), .pipedepth(6));
```

Example 6-64 ***SystemVerilog*** **Check that when a new valid read occurs, `ReadExistsInQue` is asserted the next cycle**

```
// conditional pattern

assert property (@ (posedge clk) disable iff ( rst_n )
      (SmQueNew & !SmQueWrEn |-> ReadExists InQue)) else
  $error ("ReadExistsInQue not asserted for valid read.");
```

Example 6-65 ***SystemVerilog*** **If asserting `SMQueStop` or `Flush` (internal flush state), no new requests can be received**

```
// conditional pattern

assert property (@(posedge clk) disable iff ( rst_n )
    ((Flush | SMQueStop) |-> SMQueNew))
  else $error ("Illegal SMQueNew while asserting Stop or Flush.");
```

Example 6-66 ***SystemVerilog*** **A new valid write must not be asserted while `ReadExistsInQue`**

```
// valid signal combination pattern

assert property (@(posedge clk) disable iff ( rst_n )
    not (SmQueNew & | SMQueWrEn & ReadExistsInQue)) else
  $error("%s\n%s", "A write is being enqueued while there is a valid read in the queue.",
    "This must be avoided so that the read can be invalidated by the write.");
```

Example 6-67 ***SystemVerilog*** **If `QueAlmostFull` and a new valid read request and no completion of a request, then `QueFull` must be asserted the next cycle**

```
// valid signal combination pattern

assert property (@ (posedge clk) disable iff ( rst_n )
    (QueAlmostFull & !MemEngDone & SMQueNew |-> QueFull)) else
  $error ("Queue did not fill when one more request entered.");
```

Example 6-68 ***SystemVerilog*** `SmQueNew` **and** `QueFull` **are mutually exclusive**

```
// valid signal combination pattern

assert property (@ (posedge clk) disable iff ( rst_n )
    not (SmQueNew & QueFull)) else
  $error ("Overflow of SMQue block.");
```

Together, the inter- and intra-interface assertions provide a valuable net for trapping incorrect behavior for a block. They not only detect illegal operation of the block, which is valuable for allowing quick corrections, they also detect any illegal stimulus from verification testcases, testbench stimulus generators, or the actual companion system block. The complete process includes adding assertions to the implementation of this block, including the necessary structures, and the derived logic to the interfaces of the structures.

6.9 Summary

In this chapter, we introduced the concept of an *assertion pattern* as a convenient medium used to document and communicate design insight for assertions that recur. The goal in creating an *assertion pattern* is to present two interdependent components (that is, *property structure* and *rationale*) about a particular characteristic or verification concern associated with a particular design.

The patterns we introduced in this chapter include:

- Signal patterns
- Set patterns
- Conditional patterns
- Past and future event patterns
- Window patterns
- Sequence pattern