CHAPTER

7

ASSERTION COOKBOOK

Like a culinary cookbook that offers new ways to prepare traditional dishes, this chapter offers a set of recipes for cooking up assertions and functional coverage for many common design structures found in today's RTL designs. Just as a good cookbook offers more than lists of ingredients, the goal of our cookbook is to offer examples of design assertions and functional coverage coding techniques that combine the right ingredients and methods to achieve successful coverage. As you consider applying our examples to your own designs, we recommend that you initially write assertions that specify the design intent prior to coding the RTL implementation. Capture additional assertions during the process of coding the RTL implementation as design details develop. Also, specify implementation functional coverage points during RTL coding to ensure that key features of the lower-level implementation are adequately tested—along with the higher-level functionality and functional coverage defined by the verification team.

In this chapter, we explore a typical set of assertions and functional coverage points for queues, stacks, finite state machines, encoders, decoders, multiplexers, state table structures, memory, and arbiters. Certainly our list of common structures is not exhaustive. Nonetheless, if you combine these common examples with the patterns examples discussed in Chapter 6, you can extend these ideas to cover your own unique designs.

add functional coverage to RTL only when it makes sense

Many of our examples demonstrate how to specify functional coverage for various design structures. However, we advise you to selectively add RTL implementation functional coverage. It is only for portions of your RTL implementation that concerns you—specifically related to adequate testing (such as boundary conditions on queue pointers, which can be hard to observe or forgotten, in a chip-level verification environment). Some of our examples, such as functional coverage on multiplexer select lines

or counter values and controls, could generate enough data to overwhelm the engineer analyzing the results. In addition, for some of the lower-level functional coverage examples (again, for example the multiplexer select lines), the coverage could be measured using a commercial code coverage tool. However, our goal in this chapter is to demonstrate different forms of specification using the structures described in this section, you must decide what makes sense specific to your testing goals when specifying functional coverage in your own RTL implementation. See section 5.4.2 "Correct coverage density" on page 139 and section 5.4.3 "Incorrect coverage density" on page 141 for detailed discussions of appropriate specification of functional coverage. Incidentally, some functional coverage tools have an upper limit on the reporting of a particular functional coverage point. This is useful to minimize the noise that can occur during verification when incorrectly specifying functional coverage on high occurrence events.

user-defined error messages

For many of the examples in this section, we have omitted *user-defined error messages* and property names for simplification. However, we recommend that you always code user-defined error messages for your assertions (assuming your assertion language permits reporting user error messages), since doing so provides a context for other users who analyze simulation failures and do not know the specific details of the design (or assertion). Refer to section 2.2.2 "Best practices" on page 33.

clocks and resets

For simplicity, many of our PSL examples are coded under the assumption that the engineer had previously defined a default clock in PSL. For example:

```
default clock = (posedge clk);
```

Furthermore, many of our examples, for simplicity, do not take into account a reset condition. We recommend that you augment our examples with either the PSL abort:

```
assert always (a^b) @(posedge clk) abort rst_n;
```

or SystemVerilog disable iff operator to handle your specific reset requirements:

```
assert property (@(posedge clk) disable iff (rst_n) (a^b));
```

7.1 Queue—FIFO

Context. The *queue* (that is, FIFO), one of the most easily distinguished elements in a design, is primarily used to buffer data between multiple processing elements—typically operating at dissimilar rates. Unfortunately, there are multiple implementations of queues containing many unique features, even within the same design. However, there are several common features in all queue structures that warrant assertions. By specifying assertions associated with common features, you can validate that data is neither lost nor used incorrectly. Specifically, the control circuits designed to prevent overflow or underflow conditions are ideal queue features that assertions should check.

Example 7-1 and Example 7-2 are common implementations of a queue. In Example 7-1, we implement a queue with an index counter (which represents the current depth of the queue). Alternatively, Example 7-2 implements a queue using a unique read and write pointer.

In the following sections, we demonstrate a number of common assertions associated with queues. These assertions detect the error conditions such as: *overflow, underflow, invalid status, data corruption, invalid flush,* and *invalid state.* In addition, we recommend a number of likely features you might wish to specify as functional coverage.

Example 7-1 *Verilog* fragment for FIFO with index counter

```
// use a counter to represent current depth of the FIFO
parameter FIFO_depth=16;
parameter logDEPTH=4;
reg [logDEPTH-1:0] cnt;

always @ (posedge clk) begin
  case ({push, pop})
    2'b10: begin
             cnt <= cnt + 1;
             ...
    2'b01: begin
             cnt <= cnt - 1;
             ...
```

Example 7-2 *Verilog* fragment for FIFO using read and write pointers

```
// use two decoded pointers starting with value 1
// when pointers are equal (and full is not set) FIFO is empty
parameter FIFO_depth=16;
reg [FIFO_depth-1:0] rdptr, wrptr;
reg                  full;
reg                  empty;

always @ (posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    rdptr <= 1;
    wrptr <= 1;
    full  <= 0;
    ...
  end
  case ({push, pop})
    2'b10 : // WRITE
     begin
  // rotate the write pointer, set full if update will match rdptr
       wrptr <= {wrptr[FIFO_depth-2:0], wrptr[FIFO_depth-1]};
       full  <= rdptr == {wrptr [FIFO_depth-2:0], wrptr[FIFO_depth-1]};
        ...
    2'b01 : // READ
     begin
  // rotate read pointer, clear full since we can't
       rdptr <= {rdptr[FIFO depth-2:0], rdptr[FIFO depth-1]};
       full <= 1'b0;
        ...
```

7.1.1 Assertions

7.1.1.1 FIFO overflow assertion

detecting overflow

Pattern name. FIFO overflow

Problem. Overflowing a FIFO causes data loss.

Solution. The assertions in Example 7-3 specify that it is not possible to write a new element into the FIFO when the FIFO is full. Note that these assertions are a variation on the *invalid signal combination pattern* discussed in section 6.3.3 on page 177.

Example 7-3 *PSL* FIFO overflow assertion

```
assert never (cnt == FIFO_depth & push & ! pop); // for Example 7-1
assert never (full & push & !pop);               // for Example 7-2
```

Regardless of the two implementations, both assertions share the same pattern. That is, it is invalid to write into the FIFO while the FIFO is full.

7.1.1.2 FIFO underflow assertion

detecting underflow

Pattern name. FIFO underflow

Problem. Invalid data read from an empty FIFO results in unpredictable behavior, which can be difficult to debug and isolate at the chip-level during verification.

Motivation. Similar to the overflow problem, attempting to remove too much data creates unpredictable results (that is, an error in the design). When you specify an assertion for an underflow situation, you must take into account the architecture of the FIFO. For example, for a feed forward FIFO, the input data port is directly routed to the output data port when the FIFO is in an empty state.

Solution. Example 7-4 specifies that it is not possible for the design's control logic to read from an empty FIFO. Hence, this assertion identifies underflow conditions during verification.

Example 7-4 *PSL* FIFO underflow with feed forward design

```
assert never ((cnt == 0) & !push & pop);              // Example 7-1
assert never ((wrptr == rdptr) & !full & !push & pop);  // Example 7-2
```

Example 7-5 demonstrates how to specify an underflow assertion for a non-feed forward FIFO. That is, the FIFO does not allow direct routing of the input data to an output data port. Hence, the push signal is not part of the assertion.

Example 7-5 *PSL* FIFO underflow without feed forward

```
assert never ((cnt == 0) & pop);             // for Example 7-1
assert never ((wptr == rdptr) & !full & pop);  // for Example 7-2
```

7.1.1.3 FIFO non-corrupt data assertion

ensure that outgoing data matches original incoming data

Pattern name. FIFO non-corrupt data

Problem. If the data leaving the FIFO doesn't match the original data entering the FIFO (with the appropriate latency), then the read data is corrupt and can result in unpredictable behavior, which can be difficult to debug and isolate at the chip-level during verification.

Motivation. Detecting errors within a FIFO design is easiest when you identify them close to the source and time of occurrence. Often you must track bus transaction errors back to a specific FIFO because the data is not correct. By using FIFO

non-corrupt data assertions, you will be able to accurately identify problems, such as FIFO flush, complicated push/pop operations, and general data corruption.

Solution. Example 7-6 uses a module to ensure that the final data read out of the FIFO is not corrupted with respect to the original data written into the FIFO. This module makes use of the new SystemVerilog *dynamic variable* data facilities associated with properties and sequences, which enables us to store incoming FIFO data to compare with the output data of the FIFO at a future point in time. For a more complete definition of the pipeline_reqack module, see Example 6-44, "SystemVerilog pipelined_reqack module" on page 201.

Consideration. The pipeline_reqack module instantiation in Example 7-6 is connected to the FIFO control signals `push` and `pop`. The incoming data (that is, `data_in`) is stored in the FIFO for comparison when a `push` occurs. Later, the stored data is compared with the `dataout` value when a `pop` occurs.

Example 7-6 *SystemVerilog* module to ensure FIFO is not corrupted

```
// see details section Example 6-44 on page 201
 pipeline_reqack FIFO_check
           (.pipedepth  (8),         // depth of FIFO
            .latency    (20),        // latency to acknowledge
            .req        (push),
            .req_datain (data_in),   // data put into FIFO
            .ack        (pop),
            .dataout    (data_out), // data exiting the FIFO
            .clk        (clk));
```

7.1.1.4 FIFO flush assertion

ensure that a FIFO flush operates correctly

Pattern name. FIFO flush

Problem. If FIFO data is not invalidated during a FIFO flush, and is read at some future point in time, then unpredictable behavior can occur, which can be difficult to debug and isolate at the chip-level during verification.

Motivation. One source of FIFO design errors is associated with FIFO flushing (or invalidation) logic. Hence, we must ensure that all data is invalidated after a flush operation.

Solution. Consider the requirement for a FIFO containing a `flush` signal that when active, invalidates the FIFO data on the cycle immediately after the active flush. For the FIFO in Example 7-7, we could write an OVL assertion to ensure that all counters and read and write pointers have been properly reset after a flush.

Example 7-7 *OVL* valid flush operation

```
// for Example 7-1
assert_next flush_7_1 (clk, rst_n, flush, (cnt == 0));
// for Example 7-2
assert_next flush_7_2 (clk, rst_n, flush, (!full && rdptr == wrptr));
```

7.1.1.5 FIFO contiguous data assertion

ensure legal pointer state

Pattern name. FIFO contiguous data

Problem. If the FIFO valid bits, representing queued data, are not in a legal state, unpredictable behavior can occur, which can be difficult to debug and isolate at the chip-level during verification.

Motivation. Some FIFO implementations use a valid bit structure (as compared with a head/tail pointer implementation shown in Example 7-2) to track the element status. That is, one valid bit for each element in the FIFO indicates that the FIFO element is not empty. Under proper operation, the state of the FIFO must contain contiguous valid entries, which represent no holes in the FIFO data.

Solution. In Example 7-8 we check legal combinations of the *valid bit* structure using the new SystemVerilog inside operator to ensure that data isn't lost.

Example 7-8 *SystemVerilog* legal internal state for valid bits

```
// Property for 6 deep FIFO to ensure legal state values
property legal_valid_states;
  @ (posedge clk) (valid inside {6'b0, 6'b1, 6'b11, 6'b111, 6'b1111,
                                 6'b1_1111, 6'b11_1111});
endproperty
assert property (legal_valid_states);
```

In an alternate implementation shown in Example 7-9, we use a one-hot encoding for the read and write pointers. Each time a read or write occurs, the state rotates to the next available bit. We specify that the one-hot encoding is correct for all valid states. We also specify that all state transitions follow a valid rotation of encoding, which represents the contiguous content of the FIFO.

Example 7-9 *SystemVerilog* valid pointer behavior

```
parameter N=FIFO_depth;

property legal_ptr(ptr);
  @ (posedge clk) ($countones(ptr)==1);
endproperty

property good_ptr_update (ptr);
  @ (posedge clk)
     ptr inside {0,                                    // reset
                 $past(ptr),                           // stay same
                 $past ({ptr[N-2:0], ptr[N-1]}) };     // rotate
endproperty

assert property (legal_ptr(wrptr));
assert property (legal_ptr(rdprt));
assert property (good_ptr_update(wrptr));
assert property (good_ptr_update(rdptr));
```

In an alternate implementation shown in Example 7-9, we use a one-hot encoding for the read and write pointers. Each time a read or write occurs, the state rotates to the next available bit. We specify that the one-hot encoding is correct for all valid states. We also specify that all state transitions follow a valid rotation of encoding, which represents the contiguous content of the FIFO.

7.1.2 FIFO functional coverage

ensure proper coverage for FIFO full and empty cases

Pattern name. FIFO boundary condition

Problem. Without checking full and empty boundary conditions, key functionality could remain unverified.

Motivation. We want to ensure that the FIFO is sufficiently exercised. We could write functional coverage for each entry within the FIFO. Alternatively, we could focus on the end points (that is, FIFO full and empty conditions).

Solution. We use the full and empty status flags to determine the status of the FIFO. In Example 7-10 we show a functional coverage point using a PSL cover construct to report whenever the FIFO is full. We also report situations when the FIFO is empty.

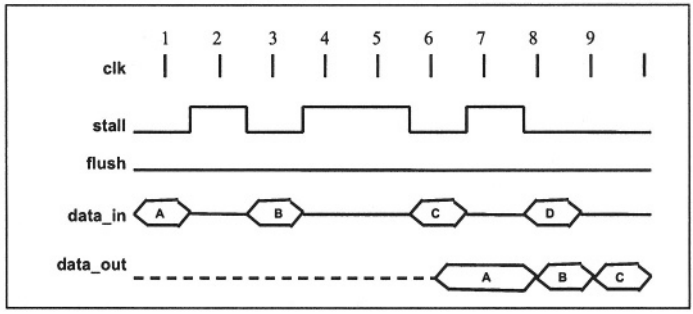
Example 7-10 *PSL* functional coverage for FIFO boundaries

```
// for Example 7-2
default clock = (posedge clk);
cover { !full; full };
cover { !empty; empty };
```

7.2 Fixed depth pipeline register

Context. In addition to a FIFO operating as a buffer, FIFO pipeline structures are common in communication protocols across asynchronous clock domains. Pipeline registers often have stalling or flushing capabilities, which is typically where errors occur during implementation.

Figure 7-1 Behavior of pipeline register of depth three with stall



Example 7-11 demonstrates a simple pipeline register of depth three, with flushing and stalling capabilities. Figure 7-1 illustrates the behavior of the pipeline register. Data is read into the pipe when the `stall` command is inactive (and a `flush` is inactive). The data appears at the output of the pipe after two additional non-stalled cycles later. In addition, the current output is held constant during a `stall`.

Example 7-11 *Verilog* fragment for Simple pipeline register

```
...
// width is the width of the data being piped
// depth is the number of pipe stages
parameter WIDTH=4;
parameter DEPTH=3;
parameter N = WIDTH*DEPTH; // size of register * # stages
reg [N-1:0] pipeline;

wire [WIDTH-1:0] data_in;
// output upper bits (WIDTH) of the pipe
wire [WIDTH-1:0] data_out = pipeline[N-1:N-WIDTH];

always @(posedge clk) begin
    if (flush)
      pipeline <= {N{1'b0}};
    else // shift value in pipe
      if (!stall)
        pipeline <= {pipeline[N-1-WIDTH:0], data_in[WIDTH-1:0]};
    end
end
```

7.2.1 Assertions

7.2.1.1 Pipeline data consistency assertion

ensure data is not corrupted through pipe

Pattern name. Pipeline data consistency

Problem. If a fixed depth pipeline register structure does not maintain consistency between the input data and the output data (accounting for the latency of the pipeline depth), then unpredictable behavior can occur, which can be difficult to debug and isolate during verification.

Motivation. The pipeline implementation may be a simple set of registers or it may be a more complex FIFO model with tags and multiplexers. For complex implementations where tags may be incorrectly flushed at the output of the pipeline, incorrect data can be returned due to an implementation error. Hence, it is critical to ensure that the pipeline register's data is consistent under various stalling and flushing scenarios.

Solution. In Example 7-12 we ensure that `data_in` entering the pipeline exits correctly to the `data_out` port. The SystemVerilog assertion uses the *dynamic variable* `thedata` to capture the data entering the pipe (that is, whenever a `stall` is not occurring). On the cycle immediately after the end of the sequence, where the number of *un-stalled* cycles within the sequence is equal to the depth of the pipe, the captured data `thedata` is compared with the

data_out. Note that our implementation of the pipeline register is fairly straight forward. One could argue that the assertion we wrote is not useful for such a simple implementation—and we would agree. However, in many cases the pipeline register is not as simple as our example. Hence, our solution would apply to any implementation with stalling and flushing capabilities.

Example 7-12 *SystemVerilog* pipeline data in—data out consistency check

```
// related to Example 7-11

property good_pipedata;
    reg [WIDTH-1:0] x;          // to save the incoming data
        // define a sequence that captures data for comparison
     @(posedge clk) disable if (flush)
       !stall, x=data_in   |=>  // first capture the data
         !stall [-> `DEPTH-1]   // valid transfer through block
              ##0 x==data_out;  // now check against the output
endproperty

assert property (good_pipedata);
```

7.2.1.2 Pipeline flush assertion

ensure pipeline data is flushed correctly

Pattern name. Pipeline flush

Problem. Failure to flush data in a pipeline can cause events (for example, a request) to occur earlier than expected, and the data itself will be out of sync with correct execution.

Motivation. Flushing pipeline registers may only require flushing valid signals, but in some cases data may also need to be flushed. The data that comes out incorrectly after a flush may cause incorrect control operations, or be data that should have been ignored.

Solution. The assertion we use in Example 7-13 ensures a correct flushing operation. Example 7-14 shows an assertion that checks that the data_out is flushed until valid data is clocked through the depth of the pipe.

Example 7-13 *SystemVerilog* pipeline data is flushed

```
// related to Example 7-11
property flushed_pipe;
  @(posedge clk)($rose (flush) |=> pipeline=={N {1'b0}});
endproperty

assert property (flushed_pipe);
```

Example 7-14 *SystemVerilog* pipeline data is zero until pipeline is flushed

```
// related to Example 7-11
property flushed_depth;
  @(posedge clk) ($fell (flush) |=>
    !data_out throughout !stall [->DEPTH]);
endproperty

assert property (flushed_depth);
```

7.2.2 Functional coverage

ensure flushes occur during verification

Pattern name. Pipeline flush coverage

Problem. Without verifying that a flush occurs after writing into a pipeline register, key functionality could go unverified.

Motivation. Depending on the pipeline usage, coverage may include reporting the various data types sent through the pipe. For a general data pipeline, coverage of a proper flush operation is necessary.

Solution. As shown in Example 7-15, we report a special situation, which is all occurrences of a `flush` operation (and no `stall`) within two cycles of valid `data` appearing on the pipe output.

Example 7-15 *SystemVerilog* functional coverage of flushed data

```
// report occurrence of no stall and valid data followed
// within 1 or 2 cycles a flush
property cov_flushed;
  @ (posedge clk)
   (!stall & |data ## [1:2] flush);
endproperty
cover (cov_flushed);
```

7.3 Stack—LIFO

Context. A stack is another common structure used for storing and retrieving data in a *last in first out* fashion, which is then used by a processing element. A stack is similar in complexity to a queue; and thus, so are the assertions you will write. Example 7-16 demonstrates a simple coding for a stack. In this example, we shift data into the variable stack_data whenever a `push` occurs, and we shift data out whenever a `pop` occurs. We maintain a set of

valid bits for each stack entry to determine when the stack is full. In Section 7.1 "Queue—FIFO", we defined a set of assertions that are common to stacks as well as queues. These assertions include:

- Example 7-3, “PSL FIFO overflow assertion”
- Example 7-5, “PSL FIFO underflow without feed forward”
- Example 7-8, “SystemVerilog legal internal state for valid bits”
- Example 7-8, “SystemVerilog legal internal state for valid bits”

In the following section, we define an additional assertion (unlike assertions for queues) for the case when both a stack push and pop occur, requiring a special sequence order of operations to replace the data as the first entry of the stack.

Example 7-16 *Verilog* fragment for simple stack

```
// use a valid vector to indicate entries that are filled (valid)
// as data is pushed in, we shift the stack data and add the new data.
parameter DEPTH   = 16; // 16 entry stack.
parameter WIDTH   = 8; // 8 bits wide.
parameter STKWIDTH = WIDTH*DEPTH;
reg   [DEPTH-1:0]     valid;
wire  [WIDTH-1:0]      data_in;
reg   [STKWIDTH-1:0]  stack_data;
wire  [WIDTH-1:0]      top_of_stack = stack_data [WIDTH-1: 0] ;

always @ (posedge clk or negedge rst_n) begin
  case ((push, pop, flush})
    3'b100 : begin
      valid <= {valid[DEPTH-2:0], 1'b1}; // shift in another entry.
      stack_data <= {stack_data[STKWIDTH-WIDTH-1: 0], data_in};
          ...
    3'b001: begin
      valid <= {DEPTH{1'b0}};
      stack_data <= {STKWIDTH{1'b0}};
          ...
    3'b010: begin
      valid <= {1'b0, valid[DEPTH-1: 1]} ; // pop it out.
      stack_data <= {{WIDTH{1'b0}}, stack_data [STKWIDTH-1 :WIDTH]};
          ...
  endcase
```

7.3.1 Assertion

Pattern name. Stack push with pop

Problem. A simultaneous push and pop stack operation may incorrectly replace the top element of the stack.

Motivation. It is easy to overlook the case when push and pop commands occur simultaneously such that the old data is removed

from the stack while the new data is added to the top of the stack and valid data depth of the stack does not change.

Solution. In Example 7-17, we check for stack push with pop using a SystemVerilog property. Note that this assertion is actually independent of any specific stack implementation. Hence, not only does it work for our simple stack example, the assertion can be used for a more complex implementation. Note for this example, we are using the new SystemVerilog $stable system function, which returns true if the previous value of its argument is the same as the current value. For additional details, see Appendix C.

Example 7-17 *SystemVerilog* correct push and pop operation of stack

```
// for a push and a pop, then on the next cycle the top of the
// stack must equal what was previously pushed onto the stack
// and the stack depth is the same
property push_and_pop_good;
  @(posedge clk)
     (push & pop |=>$past(data_in) == top_of_stack &&$stable(valid));
endproperty
assert property (push_and_pop_good);
```

7.3.2 Functional coverage

7.3.2.1 Stack depth coverage

Pattern name. Stack depth

Problem. Without tracking coverage measurement for various levels of valid content contained within a stack, we do not know if we have tested boundary conditions for the stack, and we have not tested the architectural performance.

Motivation. We must ensure that boundary conditions for the stack have been adequately tested. If we have not used all elements, we have either over designed the stack (the stack depth is too large), under designed the stack (not large enough, which affects performance), or we have not fully tested our design.

Solution. In Example 7-18., we report the filling for each possible level of the stack to determine the maximum depth achieved during verification. This is useful if we want to measure architectural performance.

functional coverage caution

As previously discussed in the introduction to this chapter, be cautious when specifying this level of functional coverage. We recommend that you only do this if it is critical to determine the architectural performance of a specific design implementation.

Example 7-18 *SystemVerilog* functional coverage usage of each element

```
genvar i;
generate for(i=0;i<=16;i= i + 1)
   cover property @(posedge clk) ($rose (valid[i]));
endgenerate
```

7.3.2.2 Stack flushes

Pattern name. Stack flushes

Problem. If the stack flush control circuit is not designed properly, invalid data could be popped out of the stack.

Motivation. It can be difficult to debug unexpected behavior during simulation. Without testing flush operations, we cannot ensure that flushed content on the stack does not affect subsequent behavior. However, if we are monitoring the occurrence of critical behavior during simulation, we can associate the failure with other events that are occurring about the same time as the failure.

Solution. Example 7-19 reports the occurrence of a flush for various levels of content within the stack.

Example 7-19 *SystemVerilog* functionl coverage of flushing of each valid entry

```
genvar i;
generate for(i=0;i<=7;i= i + 1)
   cover property @ (posedge clk) (flush & valid[i]));
endgenerate
```

7.4 Caches—direct mapped

Context. Caches are used to reduce the required high memory bandwidth and data latency to a processor's main memory. They are a critical design element required to achieve a desired system performance, particularly in multi-processing systems. Assertions combined with functional coverage ensure that a system design makes full use of its cache elements (that is, achieves the desired

perfomance) and that integrity of the data is correct within the system (that is, *coherent*).

Example 7-20 *Verilog* fragment for high-level direct mapped cache

```
parameter ADDRW    = 32;        // number of address bits
parameter NLINES   = 8;         // number of lines
parameter LINE     = 9;         // bytes in a line
parameter logNLINS = 3;         // log based 2 of N lines
parameter TAG      = ADDRW-3;   // address minus 3 index minus 3 offset

reg  [8*LINE-1:0]   cache_line  [NLINES-1 :0] ;
reg  [8*TAG-1:0]    cache_tag;  // tag set
reg  [NLINES-1:0]   valid, n_valid, match;
reg  [NLINES-1:0]   cache_we;   // write data [line]
reg  [logNLINS-1:0] line_sel;   // hit line selector
reg                 hit;        // request hit in cache
wire                cache_fill; // fill line from mem, mark valid
wire                write;      // store request
wire                request;    // request to the cache
wire  [ADDRW-1:0]   addr;       // request address
wire                invalidate; // invalidate the cache

always @(*) begin
  case ({write, request, cache_fill, invalidate})
     4'b1100,
     4'b0100: begin             // write or read request
  // index is a function returns index from the addr to select a line
         line_sel  = index(addr);            // extract index
  // hit_detect is a function returns vector for tag match
         match     = hit_detect (addr, cache_tag) & valid;
         hit       = |match;                 // compute hit of match
         write_hit = {NLINES{write}} & match; // compute write enable
        ...
     4'b0010: begin                          // fill (from memory)
         n_valid = valid | cache_we;         // setting a new valid
         ...
     4'b0001: begin                          // invalidate
         n_valid = {NLINES{1'b0} } ;         // clear all valids
         ...
```

Example 7-20 is a fragment of a *direct mapped cache* controller. This simple cache example manages data from a specific memory region in segments of eight bytes, which is referred to as a cache *line*. The cache supports both *write* and *read* requests (corresponding to stores and loads). It also supports cache data *invalidate*, as well as updates of the cache from a specific region of memory (referred to as cache *fills*).

A high-level description of cache operations follows. When a processor (requestor) makes a request to the cache, the cache looks up the address in its cache *tag set*. If the requested address matches a *valid* line (referred to as a cache *hit*), the cache returns data from the line to the requesting processor. When there is no hit, the cache makes a request to memory. When this request

completes, the cache line is filled with the data from the specific region of memory (a *cache fill*) and returns the data to the requestor. This data is now available in the cache if a similar address (that is, an address within regions of the address for the valid cache line) is requested in the future.

7.4.1 Assertions

7.4.1.1 Cache line fill

Pattern name. Cache line fill

Problem. Performance and coherency problems within a cache system occur if a cache *fill* does not result in a *valid* cache line in the absence of an *invalidate* command.

Motivation. We must ensure that the cache is updated from memory when a *cache fill* request occurs. Checking to ensure that a *fill* produces a *valid* line reduces further request latencies.

Solution. Example 7-21 shows an assertion for eight cache lines using a PSL forall construct. A *fill* request for a specific line must produce a *valid* line the next cycle (unless an *invalidate* occurs).

Example 7.21 *PSL* fill (write) implies valid line

```
// for eight cache lines
assert
  forall N in (0:7) :
    always ({cache_fill && cache_we[N] & !invalidate} |=> (valid[N]))
      @(posedge clk);
```

7.4.1.2 No hit on invalid cache lines

Pattern name. No hit on invalid cache lines

Problem. If an invalid line receives a *hit* (that is, address match), and thus returns data, unpredictable behavior can occur, which can be difficult to debug and isolate during verification.

Motivation. Calculation of the cache *hit* result may be distributed among multiple equations where it is easy to overlook required terms for a specific equation. If a the logic calculating a hit is in error, invalid data could be returned from the cache, resulting in an error.

Solution. The cache line selection is determined from an address. However, the valid bit must be included as part of an

address match. Example 7-22 shows an assertion that specifies that an inactive valid bit will never result in a hit.

Example 7-22 *SystemVerilog* not valid line implies no hit

```
property no_invalid_hit;
   @(posedge clk) (!valid(line_sel] |-> !hit);
endproperty
assert property  (no_invalid_hit);
```

7.4.1.3 Invalidating a cache

Pattern name. Invalidating a cache

Problem. Stale data (that is, data that should have been invalidated) within a cache causes coherency problems in a system.

Motivation. Cache invalidation is used to clear a cache of data that is no longer valid with respect to the main memory store, thus ensuring coherency in a system. Use of stale data causes a system crash, algorithm failure, and difficult to diagnose problem.

Solution. Example 7-23 ensures that once invalidate is issued, all cache lines are *invalid.*

Example 7-23 *PSL* invalidate implies empty cache

```
assert always  ({rose (invalidate)} |=> (!valid});
```

7.4.1.4 Write request update

Pattern name. Write request update

Problem. If an invalid cache line is accessed after a write request update, unpredictable behavior can occur, which can be difficult to debug and isolate during chip-level verification.

Motivation. For writes that *hit* a cache line, depending on the design, the cache may choose to either update the cache data—or may invalidate the cache line (for example, for a write-thru cache design or a hit of a shred line). Reading stale data after an invalidate is a coherency problem.

Solution. Example 7-24 shows a write request producing a *hit* that must cause an update to the cache data—or an *invalidate* to the cache line on the following cycle.

Example 7-24 PSL write hit creates new data or invalid line

```
// For all eight cache line.
assert
  forall L in {0:7}:
    always({rose(write_hit[L])} |=>
       {prev(cache_line[L]) != cache_line[L] || !valid[L]});
```

7.4.2 Functional coverage

Functional coverage for a cache serves two purposes. It ensures the logic is operating correctly and that system performance is not being lost (that is, cache performance relies on full use of the cache structure to provide data with minimal latency).

7.4.2.1 Cache line fill

Pattern name. Cache line fill

Problem. Cache lines that are not filled do not contribute to the desired system performance.

Motivation. The performance of a cache system is based on the proper handling of existing data contained within the cache during a request. To ensure good performance, it is critical that we verify that all cache lines have gone from invalid to valid for a new address during a write request.

Solution. Example 7-25 shows a coverage event for a cache *fill* (write) request to a specific cache line (`cache_we[i]`). We have specified functional coverage, which will report each time a requested line is filled during verification.

Example 7-25 *SystemVerilog* functional coverage of each valid line

```
genvar i;
generate for(i=0;i<=7;i= i + 1)
   cover property @(posedge clk) (cache_fill & cache_we[i]);
endgenerate
```

7.4.2.2 Cache line usage

Pattern name. Cache line usage

Problem. If we do not receive a *hit* on a given cache line, then the cache line does not contribute positively to the cache system performance (that is, it is impossible to get a hit on the line due to

a design error or we have not achieved adequate testing of the line).

Motivation. Once a cache line is filled, future requested addresses should be able to *hit* in the cache, ensuring good system performance. Failure to *fill* a line or *hit* a line during verification is cause for further analysis to determine if an error exists in the cache system design.

Solution. Example 7-26 creates functional coverage, which reports the hit for a given cache line. With this and the previous example, we can see if lines are being brought into the cache (that is, was previously filled) and are subsequently reused by the processor.

Example 7-26 *SystemVerilog* functional coverage of each cache line hit

```
genvar i;
generate for(i=0;i<=7; i= i + 1)
   cover property @(posedge clk) (L == line_sel && valid[i] & hit);
endgenerate
```

7.4.2.3 Write hit to cache line

Pattern name. Write hit to cache line

Problem. If we do not receive a *write* hit on a given cache line, then the cache line has not been adequately tested, or the cache line might have been corrupted.

Motivation. Once a cache line is filled, future addresses should be able to *hit* in the cache. Not receiving a write hit during verification is cause for further analysis to determine if an error exists in the cache system design.

Solution. In Example 7-27, the functional coverage specification ensures sufficient write *updates* have occurred to all cache lines during verification. Sufficient updates, combined with other cache operations to a cache line ensures that the cache data is consistent on future operations. Note how this complements the assertion in Example 7-24, by ensuring that a `write_hit` occurs (that is, we cannot check the assertion in Example 7-24 if the `write_hit` never occurs).

Example 7-27 *SystemVerilog* functional coverage of each cache line written (store - hit)

```
genvar i;
generate for(i=0;i<=7;i= i + 1)
   cover property @ (posedge clk) (!cache_fill & write_hit[i]);
endgenerate
```

7.4.2.4 Cache event timing

Pattern name. Cache event timing

Problem. Without adequate testing of specific timing interaction of cache events, design errors can go undetected.

Motivation. Testing with specific timing relationships between multiple events in a cache can uncover potential corner cases where other unexpected independent events coincide. By reporting the occurrence for specific timing relationships between independent events (for example, a *fill* and a *hit*) you ensure that specific timing of events is not overlooked, which could cause a failure if not exercised.

Solution. Example 7-28 shows functional coverage for a cache *fill* request related to a load request hit of the cache. By checking that these events occurred during verification, you ensure your logic properly handles the specific timing relationships.

Example 7-28 *PSL* functional coverage of timing of events

```
// ensure timing between fill and hit of same line is covered (-2..2)
// request is used in the last two coverage specifications, because we
// can't hit when the fill is not done yet
cover {cache_fill; 1; hit && prev(fill_sel, 2) == line_sel};  // 2cyc1e
cover {cache_fill; hit && prev(fill_sel) == line_sel};        // 1cyc1e
cover {cache_fill && hit && fill_sel == line_sel};            // 0cyc1e
cover {request; cache_fill && prev(line_sel) == fill_sel};    // 1cyc1e
cover {request; 1; cache_fill &&
                          prev(line_sel, 2) == fill_sel}; // 2cyc1e
```

7.5 Cache—set associative

Context. A set associative cache, unlike a direct mapped cache, allows for more than one region of memory to be stored in a cache line at a time. Hence, an address can reside in a *set* of locations within the single line (versus a single location for a direct mapped

cache). A line now contains multiple address sets, where each set contains data for specific addresses that map into a single line. Thus a set associative cache can store multiple sequences of code that would otherwise conflict for the cache space of a direct mapped cache. This allows for a specific software routine to complete its execution while still referencing data in the cache, even though some other concurrent executing code may be accessing data from a different address space loaded into the same cache line.

The overall operation for a set associative cache is the same as the direct mapped cache described in Section 7.4 "Caches—direct mapped". A requestor provides an address to the cache that is compared to the tags in the cache, which represents the data from a specific region of memory currently loaded into the cache line. In the set associative cache, multiple tags are compared to the specified address, and only one set from the cache line can match (and *hit*). When a hit does occur, the data from that particular set is returned. When a miss occurs, a request to a region of memory is made and the data from memory is loaded into a specific set (chosen by the cache line replacement policy).

A set associative cache, in general, has challenges similar to a direct mapped cache. These problems include efficient usage of the entire cache and correctness of *fills, hits* and *invalidates* of the cache. (We demonstrated assertion and coverage techniques for these problems in the previous section.) In this section, we focus on specifying assertions and coverage specifically related to set associative caches.

Example 7-29 *Verilog* fragment for high level set-associative cache

```
parameter ADDRW     = 32;       // number of address bits
parameter NLINES    = 8;        // number of lines
parameter LINE      = 8;        // bytes in a line
parameter logNLINS  = 3;        // log based 2 of N lines
parameter TAG       = ADDRW-6;  // address minus 3 index minus 3 offset
parameter NSETS     = 4;        // number of sets in a line

reg [8*LINE-1:0]    cache_line[NSETS*NLINES-1:0];
reg [NSETS*TAG-1:0] cache_tag[NLINES-1:0];
reg [NSETS-1:0]     valid[NLINES-1:0], n_valid [NLINES-1:0],
                    set_match, // aset matched
                    set_valid, // valid bits of selected line
                    write_set; // write a set
reg [NLINES-1:0]    cache_we;  // write a line
reg [logNLINS-1:0]  line_sel;  // hit line selector
reg                 hit;       // request hit in cache
wire                cache_fill; // fill line from mem, mark valid
reg [logNLINS-1:0]  fill_sel;  // line to fill
wire                write;     // store request
wire                request;   // request to the cache
wire [ADDRW-1:0]    addr;      // request address
wire                invalidate; // invalidate the cache

always @ (*) begin
  for(i=0;i<NLINES; i=i+1) n_valid[i] = valid[i];
  case ({write, request, cache_fill, invalidate})
     4'b1100,
     4'b0100: begin             // write or read request
  // index is a function returns index from addr for line selection
        line_sel = index(addr);         // extract index
        set_valid = valid[line_sel];
  // set_hit_detect is a function returns vector for a tag match
        set_match = set_hit_detect(addr, cache_tag[line_sel])
                  & set_valid;
        hit       = |set_match;          // compute hit of match
        write_hit = {WIDTH{write}} & set_match; // compute enable
       ...
     4'b0010: begin                     // fill (from memory)
        n_valid   = valid | cache_we;    // setting a new valid
        ...
     4'b0001: begin                     // invalidate all valid bits
       for(i=0;i<NLINES;i=i+1) n_valid[i] = {NSETS{1'b0}};
        ...
```

7.5.1 Assertion

Pattern name. Match a single set within a line

Problem. If a single region of memory are mapped into multiple sets within the same cache line (that is, a request address matches multiple tags within the line), coherency in the system cannot be achieved.

Motivation. A line in a set associative cache contains several addresses. These addresses must be unique for correctness (and maximum performance). Hence, there must not be more than one (if any) matches for a given requested address to a tag within the cache line.

Solution. Example 7-30 specifies an assertion for the *set match signal* (that is, `set_match`) to ensure that it has matched the requested address to no more than a single tag from a set within the line.

Example 7-30 *SystemVerilog* at most only a single set match allowed

```
property onehot_match;
  @(posedge clk) ($countones(set_match[3:0])<=1);
endproperty
assert property (onehot_match);
```

7.5.2 Functional coverage

7.5.2.1 Cache set fill

Pattern name. Cache set fill

Problem. Cache sets that are not filled do not contribute to the desired system performance.

Motivation. The performance of a cache system is based on the proper handling of existing data within the cache during a request. To ensure good performance, it is critical to verify that all cache lines have gone from invalid to valid for a new address during a write request.

Solution. Example 7-31 shows a coverage event for a cache *fill* (write) request to a specific set within a cache line (`write_set[S]`). We have specified functional coverage that will report each time a requested set is filled during verification.

Example 7-31 *SystemVerilog* functional coverage for each set written

```
genvar L, S;
generate for(L=0;L<=7;L= L+ 1)
  generate for (S=0;S<=3;S= S+ 1)
   cover property @ (posedge clk)
     (cache_we[L] & write_set[S] & cache_fill);
  endgenerate
endgenerate
```

7.5.2.2 Usable cache set

Pattern name. Usable cache set

Problem. If we do not receive a *hit* on a given set within a cache line, then the set does not contribute positively to the cache system performance (that is, it is impossible to get a hit on the set due to a design error or we have not achieved adequate testing of the set).

Motivation. Once a set within cache line is filled, future requested addresses should be able to *hit* in the cache, ensuring good system performance. Failure to *fill* a set or *hit* a set during verification is cause for further analysis to determine if a error exist in the cache system design.

Solution. Example 7-32 creates functional coverage, which reports a hit for a given set within a cache line.

Example 7-32 *SystemVerilog* functional coverage for each set hit

```
genvar L, S;
generate for(L=0;L<=7;L= L+ 1)
  generate for (S=0;S<=3;S= S+ 1)
   cover property @(posedge clk)
     (L == line_sel & set_match[S] & hit);
  endgenerate
endgenerate
```

7.5.2.3 Write hit to set

Pattern name. Write hit to set

Problem. If we do not receive a *write* hit to a given set within a cache line, then the set has not been adequately tested, or the cache set might have been corrupted.

Motivation. Once a set within a cache line is filled, future addresses should be able to *hit* in the cache. Not receiving a write hit during verification is cause for further analysis to determine if a error exist in the cache system design.

Solution. These events in Example 7-33 ensure that sufficient use of the set within a cache lines (updated when already *valid*) is done to ensure that the cache works more than once and is robust.

Example 7-33 *SystemVerilog* functional coverage for each valid set (of a line) written

```
genvar L, S;
generate for(L=0;L<=7;L= L+ 1)
  generate for(S=0;S<=3;S= S+ 1)
   cover property @(posedge clk)
      (!cache_fill & cache_we[L] & write_set[S] & set_valid[S]);
  endgenerate
endgenerate
```

7.5.2.4 Reuse of existing set

Pattern name. Reuse of existing set

Problem. To ensure good performance, each set within a cache line must either go from *invalid* to *valid* for the new requested address—or from a different *valid* address currently contained within the cache to the new requested address not contained within the cache.

Motivation. Ensuring the cache line's *set replacement policy* is operating correctly is another performance aspect that is difficult to diagnose. By adding functional coverage, we can ensure that the this functionality is being tested.

Solution. In Example 7-34, the functional coverage specification ensures sufficient write *updates* have occurred to all cache lines during verification. Sufficient updates, combined with other cache operations, to a cache line ensures that the cache data is consistent on future operations.

Example 7-34 *SystemVerilog* functional coverage for each valid set filled

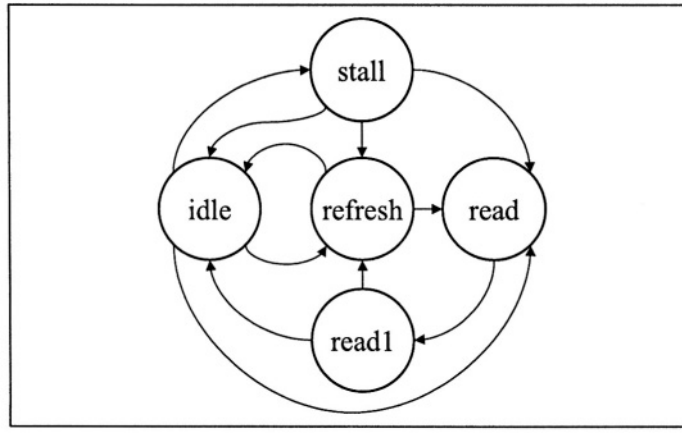
```
genvar L, S;
generate for (L=0;L<=7;L= L+ 1)
  generate for (S=0;S<=3;S= S+ 1)
   cover property @ (posedge clk)
      (cache_fill & cache_we[L] & write_set[S] & set_valid[S]);
  endgenerate
endgenerate
```

7.6 FSM

Context. Finite state machines offer many situations that are ideal for specifying correct operation using assertions and functional coverage. Some of the functional coverage examples

shown in this section may be more effectively measured using commercial coverage tools. However, complex interactions between state machines is generally difficult to measure with commercial tools and are better suited to functional coverage specification.

Figure 7-2 **FSM example used in the following sections**



7.6.1 Assertions

7.6.1.1 Legal FSM states

Pattern name. Legal FSM state

Problem. If an FSM enters an illegal state, unpredictable behavior can occur, which can be difficult to debug and isolate at the chip-level during verification.

Motivation. Often, the designer focuses on only the legal states of an FSM during the design process—or makes assumptions that illegal states are not possible. Without trapping situations that cause an FSM to enter an illegal state, a significant amount of debug effort is generally required to isolate (and map) a chip-level failure down to the offending FSM.

Solution. In Example 7-35, we apply a version of the *valid opcode pattern* to specify valid states for the FSM illustrated in Figure 7-2.

Example 7-35 *SystemVerilog* check for only legal states

```
property legal_states;
  @(posedge clk) disable iff !rst_n
     state inside {`IDLE, `READ, `READ1, `REFRESH, `STALL};
endproperty
assert property (legal_states);
```

7.6.1.2 Inputs for specific states

Pattern name. Inputs for specific states

Problem. An illegal input value for a specific FSM state can cause unpredictable behavior, which can be difficult to debug and isolate at the chip-level during verification.

Motivation. Generally, FSMs only allow certain combinations of input values when the state machine is in a specific state. For example, assume the FSM shown in Figure 7-2 does not allow either a new `req` or `abort` to occur when the FSM is in a `READ` or `READ1` state. If a new `req` or `abort` occurs, then the FSM might enter into an illegal state—or make an unexpected transition to a different legal state.

Solution. Example 7-36 demonstrates how to code an assertion that ensures a `req` or `abort` is not received as an input while the state FSM is in either a `READ` or `READ1` state.

Example 7-36 *SystemVerilog* check for specific inputs only in specific states

```
property illegal_input; // wrong time for input
  @ (posedge clk) not ((state inside {`READ, `READ1}) & (req | abort));
endproperty
assert property (illegal_input);
```

7.6.1.3 State transition performance limits

Pattern name. State transition performance limits

Problem. Failure to satisfy a state transition timing requirement may indicate an error situation (for example, a deadlock).

Motivation. Often, interfaces are designed under the assumption that they will meet a specific performance requirement. For example, a protocol controller might require that

an acknowledge is received within a fixed number of specified clocks after a request—or a memory controller is expected to return to an idle state within a specified limited range of cycles after the start of a refresh cycle. Failure to satisfy the timing requirements often indicates a forward progress or deadlock situation.

Solution. In Example 7-37, we demonstrate how to specify a performance limit for a state transition.

Example 7-37 *SystemVerilog* check for operation to complete in fixed time periods

```
property trans_toidle;
  @ (posedge clk) ($rose(state==`REFRESH) |-> ## [5:15] state==`IDLE);
endproperty
assert property (trans_toidle);
```

7.6.2 Functional coverage

7.6.2.1 FSM state coverage

Pattern name. FSM state coverage.

Problem. Without visiting all states of an FSM during the course of verification, potential bugs may go undetected.

Motivation. It is important during the course of verification to visit all legal states within a given FSM. Otherwise major functionality could go untested prior to tapeout.

Solution. In Example 7-38 we demonstrate how to write a functional coverage specification, which is used to report the visits to legal states within a given FSM. Note for simple FSMs, that follow a well defined coding style, specifying state coverage is unnecessary if you use a commercial tool that provides this form of coverage. However, complicated state machines or unconventional coding styles typically require explicit functional coverage specification.

Example 7-38 *PSL* functional coverage each state entered

```
cover {state == `IDLE};
cover {state == `READ};
cover {state == `READ1};
cover {state == `REFRESH};
cover {state == `STALL};
```

7.6.2.2 State sequence coverage

Pattern name. State sequence coverage

Problem. Without specifying expected state sequences (for example, transactions), major functionality within the design could go unverified.

Motivation. The occurrence of specific state sequences or transactions may be difficult to identify in a simulation run. By specifying expected sequences, we can be assured that sufficient verification has occurred and identify functionality within the design that has not been adequately tested.

Solution. Example 7-39 reports the occurrence for any transition from a READ state to a REFRESH state three cycles later. Note that this would potentially be equivalent to going through the sequence: READ -> READ1 -> IDLE -> REFRESH, shown in Figure 7-2.

Example 7-39 *PSL* ensure proper state sequence

```
cover {(state == 'READ); [*2]; state == 'REFRESH};
```

Example 7-40 demonstrates how to enumerate the functional coverage specification for specific state transitions. These transitions are related to Figure 7-2.

Example 7-40 *PSL* functional coverage for each state transition

```
cover {state == 'IDLE ; state == 'READ);
cover {state == 'READ1 ; state == 'STALL); // and so on.
```

7.7 Counters

Context. Counters are generally thought of as minor structures within a design. However, counters often have well defined properties (for example, no overflow or no underflow) that, if violated, can cause unpredictable behavior. Assertions enable us to check for the correctness for these properties during verification. In addition, often a specific counter value relates to the occurrence of a specific event within a design. Hence, it is often desirable to specify functional coverage on certain counter values.

The following code fragment shows an up-down counter with a parallel load option.

Example 7-41 *Verilog* fragment up-down counter with parallel load

```
parameter max=5'd26;
reg [4:0] count;
always @ (posedge clk or negedge rst_n) begin
  if (!rst_n)
    count <= 5'b0;
  else
    casez ({inc, dec, load})
      3'b??1: count <= count_in; // load takes priority.
      3'b100: count <= count + 1'b1;
      3'b010: count <= count - 1'b1;
      //0, 6 - The default case keeps the same counter value.
    endcase
end
```

7.7.1 Assertions

7.7.1.1 Maximum counter value

Pattern name. Maximum counter value

Problem. A counter that exceeds its maximum value (for example, wraps around to a lower value), can cause lost data if used as a data pointer—or, in general, cause unpredictable behavior, which can be difficult to debug or isolate at the chip-level during verification.

Motivation. Note in Example 7-41 (above) that no maximum value for `count` is enforced. Hence, the counter will wrap (that is, roll over from its maximum value) if incremented on its maximum value. This might be the desired behavior of the counter (for example, if we have implemented a counter for a circular queue). However, if this is not the desired behavior, then we should add an assertion to ensure that the circuitry preventing the illegal increment is functioning correctly.

Solution. Example 7-42 demonstrates the coding of an assertion in SystemVerilog that ensures that a counter does not exceed a specified maximum value. In addition, this example demonstrates how to code an assertion to check for an overflow condition.

Example 7-42 *SystemVerilog* counter limits

```
property exceed_max;
  @ (posedge clk) not (count > max);
endproperty

property no_overflow;
  @ (posedge clk) not (count == 0 && $past(inc & !load, 1));
endproperty

assert property (exceed_max);
assert property (no_overflow);
```

7.7.1.2 Counter underflow

Pattern name. Counter underflow

Problem. A counter that underflows its minimum value (for example, wraps around to a higher value), can cause lost data if used as a data pointer—or, in general, cause unpredictable behavior, which can be difficult to debug or isolate at the chip-level during verification.

Motivation. Note in Example 7-41 (above) that no minimum value for count is enforced. This might be the desired behavior of the counter (for example, if we have implemented a counter for a circular queue). However, if this is not the desired behavior, then we should add an assertion to ensure that the circuitry preventing illegal decrement is functioning correctly.

Solution. Example 7-43 demonstrates the coding of an assertion in PSL that ensures that a counter does not underflow.

Example 7-43 *PSL* no wraparound (underflow)

```
assert never (count == 0 & dec & !load);
```

7.7.2 Functional coverage

In some designs, specific values within a counter are associated with critical events. Hence, functional coverage for specific values helps identify the occurrence of these critical events during verification.

7.7.2.1 Counter limits

Pattern name. Counter limits

Problem. If the boundary conditions for a counter are not adequately verified, unpredictable behavior can occur, which can be difficult to debug or isolate at the chip-level during verification.

Motivation. In many designs, counters are used as pointers into other structures, such as memory addressing, FIFO, and stack pointers. Testing the boundary conditions for these counters is critical during verification to flush out comer case errors.

Solution. In Example 7-44 we demonstrate how to specify functional coverage for the boundary conditions on a counter.

Example 7-44 *PSL* functional coverage for boundary condition

```
cover {count==max-2; count==max-1};
cover {count==1; count==0};
```

7.7.2.2 Counter control

Pattern name. Counter control

Problem. If all control combinations for a counter are not exercised, then a corner case error might be missed during verification.

Motivation. For non-trivial counters, or counters associated with complex control logic, we might wish to know what combinations of controls for the counter have been exercised. Hence, the counter control could give us a level of confidence in testing the complex circuit.

Solution. Example 7-45 demonstrate how to specify functional coverage on the controls related to our counter example. Note that for simple circuits, this would probably be too much functional coverage data, as we pointed out in the introduction to this chapter. Yet for key portions of a design, we might want to determine exactly what combination of events are occurring during verification.

Example 7-45 *SystemVerilog* functional coverage for increment/decrement conditions

```
genvar i;
generate for(i=0;i<=7;i= i + 1)
   cover property @ (posedge clk) ({inc, dec, load} == i);
endgenerate
```

7.8 Multiplexers

Context. Multiplexers are key structures used to conditionally control the flow of data through a design. In this section, we discuss three common types of multiplexers, whose implementation differs based on the design of the data selection circuit (for example, *encoded, decoded,* or *prioritized*).

7.8.1 Encoded multiplexer

In Example 7-46, we demonstrate the RTL for an encoded multiplexer. We use this example during our following discussion.

Example 7-46 *Verilog* fragment for encoded multiplexer code

```
always @(select or data0 or data1 or data2 or data3)
  case (select)
    2'd0: outdata = data0;
    2'd1: outdata = data1;
    2'd2: outdata = data2;
    2'd3: outdata = data3;
  endcase
```

7.8.1.1 Functional coverage

Pattern name. Encoded selector coverage

Problem. Without exercising all critical control paths within a design during verification, corner cases can go undetected.

add functional coverage to RTL only when it makes sense

Motivation. For critical control paths, it is necessary to ensure all possible paths are exercised. Code coverage, is ideal for many of these lower-lever structures, like encoded multiplexers. However, if a particular multiplexer structure is deemed sufficiently critical (particularly related to a complex design with corner case concerns), functional coverage for a particular encoded multiplexer could be specified.

Solution. In Example 7-47, we demonstrate how to specify functional coverage for all encoded multiplexer `select` lines.

Hence, we can easily determine during verification if a particular critical path has not been exercised.

Example 7-47 *SystemVerilog* functional coverage for a four-to-one encoded mux

```
genvar i;
generate for (i=0;i<=3;i= i + 1)
   cover property @(posedge clk) (select == i);
endgenerate
```

7.8.2 Decoded one-hot multiplexer

Context. Decoded multiplexers uses a single select bit (from a multi-bit value) as an enable to transfer the selected input data to an output. Example 7-48 demonstrates the RTL for a simple decoded multiplexer, which is used in the following related discussion on assertions and functional coverage.

Example 7-48 *Verilog* fragment for decoded multiplexer example

```
always @ (select or data0 or data1 or data2 or data3)
  case(1'b1)
    select[0]: outdata = data0;
    select[1]: outdata = data1;
    select[2]: outdata = data2;
    select[3]: outdata = data3;
  endcase
```

7.8.2.1 Assertion

Pattern name. Decoded valid selector value

Problem. If more than one selector value is active for a one-hot multiplexer, then potentially the data could be corrupted—or the multiplexer circuit could be damaged (for example, a tri-state implementation).

Motivation. A decoded multiplexer are often used in a design to achieve a required timing performance, in contrast to a slower encoded multiplexer. However, for proper operation, it is critical that only a single select line is enabled at a time for the decoded multiplexer. Otherwise, the multiplexed data might be corrupted—or the multiplexer circuit is might be damaged.

Solution. In Example 7-49, we demonstrate how to specify an assertion to validate the decoded multiplexer's one-hot select line

requirement. For this example, we use the SystemVerilog $onehot system function.

Example 7-49 *SystemVerilog* assertion to validate one-hot selector

```
assert property ($onehot(select));
```

7.8.2.2 Functional Coverage

Pattern name. Decoded input selector values

Problem. Without exercising all critical control paths within a design during verification, corner cases can go undetected.

add functional coverage to RTL only when it makes sense

Motivation. For critical control paths, it is necessary to ensure all possible paths are exercised. Code coverage, is ideal for many of these lower-lever structures, like decoded multiplexers. However, if a particular multiplexer structure is deemed sufficiently critical (particularly related to a complex design with corner case concerns), functional coverage for a particular decoded multiplexer could be specified.

Solution. In Example 7-47, we demonstrate how to specify functional coverage for all decoded multiplexer `select` lines. Hence, we can easily determine during verification if a particular critical path has not been exercised.

Example 7-50 *PSL* functional coverage for each input selected

```
cover {select[2'b00]};
cover {select[2'b01]};
cover {select[2'b10]};
cover {select[2'b11]};
```

7.8.3 Priority multiplexer

Context. In many designs, certain events and interrupts are categorized such that higher-priority events execute prior to lower-priority events. In these types of designs, priority multiplexers are useful for routing the appropriate event in a prioritized order. In Example 7-51, we demonstrate a simple priority multiplexer, which is used in the following related discussion on assertions and functional coverage.

Example 7-51 *Verilog* fragment for priority multiplexer

```
always @ (select or data0 or data1 or data2 or data3)
  casez (select)  // synopsys full_case parallel_case
    4'b???1: outdata = data0;
    4'b??10: outdata = data1;
    4'b?100: outdata = data2;
    4'b1000: outdata = data3;
    default: assert property (@(posedge `TOP.clk) (1'b0));
            // Not legal select value.
  endcase
```

7.8.3.1 Assertion

Pattern name. Priority legal selection values

Problem. If a zero multiplexer selector value is driven from some controlling logic, then potentially the data could be corrupted, or unpredictable behavior, which can be difficult to debug or isolate at the chip-level during verification.

Motivation. A priority multiplexers are useful for routing the appropriate event in a prioritized order. However, for proper operation, it is critical that at least one of the select bits is enabled.

Solution. In Example 7-51, we demonstrated the RTL for a priority multiplexer, with a procedural assertion. In Example 7-52, we demonstrate a declarative form for this assertion.

Example 7-52 *SystemVerilog* ensure at least one select is active

```
property at leastone;
  @(posedge clk) not (select == 4'b0);
endproperty

assert property (atleastone);
```

7.8.3.2 Functional Coverage

Pattern name. Priority input select values

Problem. Without exercising all critical control paths within a design during verification, corner cases can go undetected.

add functional coverage to RTL only when it makes sense

Motivation. For critical control paths, it is necessary to ensure all possible paths are exercised. Code coverage, is ideal for many of these lower-lever structures, like priority multiplexers. However, if a particular multiplexer structure is deemed sufficiently critical (particularly related to a complex design with

corner case concerns), functional coverage for a particular priority multiplexer could be specified.

Solution. In Example 7-53, we demonstrate how to specify functional coverage for all priority multiplexer `select` lines. Hence, we can easily determine during verification if a particular critical path has not been exercised.

Example 7-53 ***SystemVerilog*** **functional coverage for all combinations of selects**

```
genvar  i;
generate for(i=1;i<=15;i= i + 1)
   cover property @ (posedge clk)select == i);
endgenerate
```

7.8.4 Complex multiplexer

Context. Multiplexers in many of today's designs are usually not as simple as the previous examples. Often, the design requires special needs to handle complex situations when routing control or data. Multiplexing structures, such as a Verilog `case` statement, are typical used to describe these complex routing needs. In Example 7-54, we demonstrate a complex multiplexer, which uses a Verilog `casez` construct.

Example 7-54 ***Verilog-2001*** **fragment for complex multiplexer**

```
always @(*) begin

  legal_state_size: assert property
    (@(posedge clk) disable iff (reset_n)
      ($onehot0 (state_size1 [2 : 0] )))
    else $error ("illegal state_sizel value %0b.", state size1 [2 :0]);

  casez ({size_sel, state sizel[2:0]})  // synopsys parallel_case full_case
    4'b1_??1: next_size = case_a_route;
    4'b0_??1: next_size = case_b_route;
    4'b?_1??: next_size = case_c_route;
    4'b?_?1?: next_size  = case_d_route;
    4'b0_000: next_size = case_e_route;
  endcase
end
```

7.8.4.1 Assertion

Pattern name. Complex multiplexer legal selection

Problem. If design assumptions are invalid, control logic can experience unexpected behavior during verification.

Motivation. Use of casez (or casex) allows specification of don't care values on signals. This is done to reduce the selection logic. Often, assumptions are made about legal and select possibilities during the initial design. However, these assumptions need to be validated. Furthermore, as the design is modified in the future, the original assumptions should be re-verified.

Solution. The assertion shown in Example 7-54 specifies that no more than one bit of the `state_size1` variable is set to one. This example uses the SystemVerilog $onehot0 system task.

7.9 Encoder

Context. An encoder generally converts a sparse or uncompressed bit representation into a more dense or compressed format, which is useful when transferring control information between multiple components in a design. For example, a four-bit one-hot state machine encoding can be converted into a more compressed two-bit decimal encoding for communication use, as shown in Example 7-55.

Example7-55 *Verilog* fragment for a two-bit encoder

```
always @ (control_state)
 // Convert the decoded input to encoded form.
 case(1'b1)
   control_state[0]: encode2 = 2'd0;
   control_state[1]: encode2 = 2'd1;
   control_state[2]: encode2 = 2'd2;
   control_state[3]: encode2 = 2'd3;
 endcase
```

7.9.1 Assertion

Pattern name. Encoder legal input value

Problem. If the value being encoded is in error, then the component receiving the encoded value can exhibit unpredictable behavior, which can be difficult to debug or isolate at the chip-level during verification.

Motivation. Encoder logic within the RTL design often makes assumptions that the input data is correct. If there is an unexpected error with the input encoding, then this can result in unpredictable behavior.

Solution. In Example 7-55, our RTL encoder assumes that its `control_state` variable has a one-hot value. Hence, in Example 7-56 we demonstrate how to write a SystemVerilog assertion to validate this requirement.

Example 7-56 *SystemVerilog* validate encoder legal input values

```
valid_encoder_input: assert property
    ( @(posedge clk) $onehot(control_state));
```

7.9.2 Functional coverage

Pattern name. Encoder input values

Problem. Without exercising all encoder input values within a design during verification, corner cases can go undetected.

add functional coverage to RTL only when it makes sense

Motivation. For complex encoders (unlike our simple encoder), it might be necessary to identify which values have not been exercised. For simpler encoders, this should be avoided, since code coverage tools generally provide enough feedback on the quality of their testing. However, if a particular encoder structure is deemed sufficiently critical or complex, then functional coverage for these special cases could be specified.

Solution. In Example 7-57, we demonstrate how to specify functional coverage for all encoded inputs. Hence, we can easily determine during verification if a particular critical path has not been exercised.

Example 7-57 *SystemVerilog* functional coverage for each value received

```
genvar N;
generate for (N=0;N<=3;N= N+ 1)
   cover property @ (posedge clk) (encode2 == N);
endgenerate
```

7.10 Priority encoder

Context. A priority encoder converts a prioritized sparse or uncompressed bit representation into a more dense or compressed format, which is useful when transferring control information between multiple components in a design. For example, a combination of interrupts could be grouped into a prioritized order, and then encoded into a request, which is then transferred to other components. Note that using the priority encoder in this fashion is actually a simple form of an arbiter.

In Example 7-55, we demonstrated a priority encoder, which is used in the following related discussion on functional coverage.

Example 7-58 *Verilog* fragment for a priority two-bit encoder

```
always @(int1 or int2 or int3 or int4)
  casez({int1, int2, int3, int4}) // sysnopsys full_case
    4'b???1: int_req2 = 2'd0;
    4'b??10: int_req2 = 2'd1;
    4'b?100: int_req2 = 2'd2;
    4'b1000: int_req2 = 2'd3;
  endcase
```

7.10.1 Functional coverage

Pattern name. Priority encoder input values

Problem. Without exercising all priority encoder input values within a design during verification, corner cases can go undetected.

add functional coverage to RTL only when it makes sense

Motivation. For complex priority encoder (unlike our simple priority encoder), it might be necessary to identify which values have not been exercised. For simpler encoders, this should be avoided, since code coverage tools generally provide enough feedback on the quality of their testing. However, if a particular priority encoder structure is deemed sufficiently critical or complex, then functional coverage for these special cases could be specified.

Solution. In Example 7-59, we demonstrate how to specify functional coverage for all priority encoded inputs. Hence, we can easily determine during verification if a particular critical path has not been exercised.

Example 7-59 ***SystemVerilog*** **functional coverage for each select value received**

```
genvar N;
generate for (N=1;N<=15;N= N+ 1)
   cover property @ (posedge clk) ({int1,int2,int3,int4 } == N);
endgenerate
```

7.11 Simple single request protocol

Context. A simple single request protocol allows synchronization between multiple components. In the following sections, we demonstrate how to specify assertions and functional coverage related to simple single request protocols

7.11.1 Assertions

7.11.1.1 Simple handshake

Pattern name. Simple handshake

Problem. A request-acknowledge handshake protocol violation can cause data corruption or data lost, which can be difficult to debug or isolate at the chip-level during verification.

Motivation. The controlling protocol circuits a component communicating other system components can be complex, and error prone. Problems typically encountered include lack of a returned acknowledgement within an expected time limit, dropped acknowledgements during communication, or unexpected multiple request. These problems can result in data corruption or data lost, which can be difficult to debug or isolate at the chip-level during verification.

Solution. We use the pipelined_reqack module, previously defined in Example 6-44 on page 201, to monitor the correct pipeline handshake protocol behavior. For our example, we only use the request (`req`) and acknowledge (`ack`) ports. We specify a maximum latency of 100 cycles, and the maximum number of possible outstanding request as 1 (that is, the depth of the pipelining). This monitor will report violations for the following conditions:

- multiple request without a matching acknowledge

- multiple acknowledges for a single request
- acknowledge not received in the specified maximum limit.

Example 7-60 *Verilog* module to validate req-to-done transaction

```
// See Example 6-44 on page 201 for details of pipelined_reqack
// module
pipelined_reqack
   sendReadReq(.req(req), // The handshakes, request
               .ack(done),     // and done (completion)
               .req_datain(1'b1), .dataout(1'b1), // not used
              .clk(elk),
               .latency(100), // Maximum latency for return of done.
                .pipedepth(1)); // Only 1 request at a time.
```

7.11.1.2 Legal interface instruction

Pattern name. Legal interface instruction

Problem. An illegal instructions transferred with a protocol request can result in unexpected behavior, which can be difficult to debug or isolate at the chip-level during verification.

Motivation. Some protocols transfers information (such as an instruction) during a protocol request. With complex FIFO designs, it is possible that an illegal instructions is transferred with a request.

Solution. In Example 7-61, we ensure the instruction (`cmd`) in a protocol request is contained within a legal set of instructions. We use the SystemVerilog inside construct to specify the legal instructions with *don't care* bits in their code. This can shorten the list of legal values required during specification.

Example 7-61 *SystemVerilog* check for legal information sent

```
property legal_cmds;
  @(posedge clk) disable iff (reset_n)
    req |-> cmd inside {'READ, 'WRITE, 'INTA, 'EIEIO);
endproperty
assert property (legal_cmds);
```

7.11.2 Functional Coverage

Pattern name. Timing between single request

Problem. If the starting of a new transaction can occur within a specified a range, then if the various range of possibilities are not explored, corner cases will not be flushed out.

Motivation. Often, corner case bugs are related to complex combination of events, which are unexpected. The completion of one transaction may affect the next transaction after a specific period. Hence, it is important to cover various ranges of possibilities during verification to ensure a particular period is not overlooked.

Solution. In Example 7-62, we demonstrate how to specify functional coverage for the case where a new request can occur anywhere between zero and twenty cycles after the completion of a previous cycle.

Example 7-62 *System Verilog* functional coverag for done-to-req timing (end to being)

```
genvar N;
generate for (N=0;N<=20;N= N+ 1)
   cover property @ (poaedge clk) (done ##N req);
endgenerate
```

7.12 In-order multiple request protocol

Context. In-order multiple request protocols allow for greater throughput on an interface by allowing a limited number of transactions to start prior to the completion of a previous transaction. Yet, matching the appropriate transaction completion event with an appropriate request is problematic. In Section 6.7.4, “Pipelined protocol pattern” on page 199 we demonstrated how to code assertions for an in-order multiple request protocol. In this section we demonstrate how to specify functional coverage.

7.12.1 Functional Coverage

7.12.1.1 Multiple request transaction timing

Problem. If the ending of a specific transaction can occur within a specified a range, then if the various range of possibilities are not explored, corner cases will not be flushed out—particularly related to multiple in-order overlapping transactions.

Motivation. Often, corner case bugs are related to complex combination of events, which are unexpected. The completion of one transaction may affect the next transaction after a specific period. Hence, it is important to cover various ranges and overlapping of possibilities during verification to ensure a particular period is not overlooked.

Example 7-63 *SystemVerilog* functional coverage for req-to-done timing

```
// extra modeling to simplify expressing coverage property
reg [3:0] req_cnt, ack_cnt;
initial {req_cnt, ack_cnt} =8'b0;
always @(posedge clk) begin
  // Increment counter each time a req or ack occurs.
  if (req) req_cnt <= req_cnt + 1;
  if (ack) ack_cnt <= ack_cnt + 1;
end

genvar C, T;
generate for (C=0;C<=15;C= C+ 1)
  generate for (T=0;T<=9;T= T+ 1)
    cover property @ (posedge clk)
    (req && req_cnt == C; ##T ack && ack_cnt == C);
  endgenerate
endgenerate
```

Solution. In Example 7-63, we specify functional coverage, which enables us to identify the range of occurrences of an acknowledge response for up to sixteen in-order overlapping transactions.

7.12.1.2 Timing of outstanding request

Pattern name. Timing of outstanding request

Problem. For an in-order multiple request protocol, it is critical that multiple outstanding request occur in varying timing relationships to each other during verification to detect potential design errors.

Motivation. Uncovering errors within a design often depends on the generation of stimulus with unexpected alignment of complex events. By reporting the timing relationship from one request to another for an in-order multiple request protocol, we can determine which cycle relationships have not been explored—thus enabling us to tune our verification environment to improve coverage.

Solution. In Example 7-64, we demonstrate how to specify functional coverage used to report various timing occurrences (between zero and ten cycles) for multiple outstanding request.

We use a counter as part of our functional coverage model to determine when an outstanding request is present.

Example 7-64 *SystemVerilog* functional coverage for outstanding request timing

```
// extra modeling to simplify expressing coverage property
reg [3:0] out_cnt;
always @(posedge clk) begin // Track outstanding req's with counter.
  if (reset_n) begin
    out_cnt <= 0;
  end
  else begin
    if (req & !done) out_cnt <= out_cnt + 1;
    if (done & ! reg) out_cnt <= out_cnt - 1;
  end
end

genvar N;
  generate for (N=0;N<=9;N= N+ 1)
    cover property @(posedge clk)
      (req; (!done || out_cnt>1) ##N req);
endgenerate
```

7.12.1.3 Maximum outstanding in-order transactions

Pattern name. Maximum outstanding in-order transactions

Problem. If we do not check the boundary condition, when the protocol's maximum supported outstanding request has occur, then a corner case could go undetected,.

Motivation. Determining if we utilize the full depth of the queues within a protocol's interface is important—not only for verification, but for performance reason—to determine if the queues are too deep or not deep enough.

Solution. In Example 7-65, specify functional coverage that will report each time the situation when maximum number of supported request has occurred during verification. Our goal is to ensure that full system (the limit of outstanding transactions) has occurred to ensure all operations are functioning as expected.

Example 7-65 *PSL* functional coverage for max outstandinig req's

```
reg [3:0] req_cnt;
always @(posedge clk) begin
  if (reset_n) begin
    out_cnt <= 0;
  end
  else begin
// Increment counter each time event is seen.
    if (req &   !ack)   req_cnt   <= req_cnt + 1;
    if (ack &   !req)   req_cnt   <= req_cnt - 1;
  end
end

// PSL cover {rose(req_cnt) == MAX_OUTSTANDING);
```

7.12.1.4 Timing between multiple requests

Problem. If the starting of a new transaction can occur within a specified a range of cycles, then if the various range are not explored, the a corner case could go undetected.

Motivation. Often, corner case bugs are related to complex combination of events, which are unexpected. The completion of one transaction may affect the next transaction after a specific period. Hence, it is important to cover various ranges of possibilities during verification to ensure a particular period is not overlooked.

Solution. In Example 7-66, we demonstrate how to specify functional coverage for the case where a new request can occur anywhere between one and ten cycles after the completion of a previous cycle.

Example 7-66 *SystemVerilog* functional coverage for done-to-next-req timing

```
genvar N;
generate for(N=1;N<=10;N= N+ 1)
   cover property @ (posedge clk) (done ##N req);
endgenerate
```

7.13 Out-of-order request protocol

Context. 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. In our pattern discussion in Section 6.7.3, "Tagged transaction pattern" on page 196, we demonstrate how to write assertions for out-of-order protocols. We now demonstrate how to specify functional coverage for these protocols.

7.13.1 Functional coverage

7.13.1.1 Maximum outstanding out-of-order transactions

Pattern name. Maximum outstanding out-of-order transactions

Problem. If all tags have not been used (that is, observed) during verification, then either the out-of-order interface has not been thoroughly test, or the queue depths within the interface were over designed, and the maximum outstanding limit cannot be achieved

Motivation. Determining if we utilize the full depth of the queues within a protocol's interface is import to determine—not only for verification, but for performance reason to determine if the queues or too deep or not deep enough.

Solution. In Example 7-67, we demonstrate how to specify functional coverage to report the occurrence of any of eight possible tags observed during an out-of-order transaction. Many commercial functional coverage tools limit the number of times it reports a specific occurrence of a functional coverage point in simulation. If your tool does not offer this feature, the functional coverage specification could be modified in such a way where you could add in your own limit mechanism.

Example 7-67 ***SystemVerilog*** **functional coverage for each tag used**

```
genvar T;
generate for (T=0;T<=7;T= T + 1)
   coverproperty @(posedge clk)(req&&reqtag==T);
endgenerate
```

7.13.1.2 Timing between multiple requests

Problem. If the completion of a transaction can occur within a specified a range, then if the various range of possibilities are not explored, corner cases will not be flushed out.

Motivation. Often, corner case bugs are related to complex combination of events, which are unexpected. The completion of one transaction may affect the next transaction after a specific period. Hence, it is important to cover various ranges of possibilities during verification to ensure a particular period is not overlooked.

Solution. Example 7-68 we demonstrate how to specify functional coverage for the case where an out-of-order acknowledge can occur anywhere between one and ten cycles after the its initial request.

Example 7-68 *SystemVerilog* funcional coverage for reg-to-done timing

```
genvar C, TAG;
generate for (C=1;C<=16;C= C+ 1)
  generate for(TAG=0;TAG<=9;TAG= TAG + 1)
   cover property @ (posedge clk)
      (req && req_tag == TAG ##C ack && ack_tag == TAG);
  endgenerate
endgenerate
```

7.14 Memories

Context. Memories, a fundamental component within a system, are used to store the instructions and data for processing components. Interfaces to memory consist of control, data, and address buses. The control signals are used to select a specific memory component to service a system read or write request. Assertions added to the interface of memory components helps isolate modeling problems quickly; such as illegal control signal combinations—or illegal control, address, and data signal values (such as X). Example 7-69 demonstrates a Verilog fragment for a

memory module interface, which we use as a reference for the assertions specified in this section.

Example 7-69 Verilog fragment for a memory module interface

```
module memory device (ce_n, rd_n, we_n, addr, wdata, data, clk);
  input          ce_n,   // chip select ( -- _n == active low )
                 rd_n,   // read enable
                 we_n,   // write enable
                 clk;    // the clock
  input  [11:0]  addr;   // the address for the device
  input  [7:0]   wdata; // write data for writes
  output [7:0]   data;   // read data being returned.
```

7.14.1 Assertions

7.14.1.1 Unknown controls

Pattern name. Unknown controls

Problem. If a neighboring block within an RTL model drives X values as controls into a memory component, unpredictable behavior can occur, which could be difficult to isolate if the X value is not visible at the chip-level boundary.

we encourage using lint to identify unconnected ports

Motivation. When an engineer instantiates a memory model, it is possible that the controls signals were inadvertently left unconnected, which would result in X values driven into the memory model during simulation. Furthermore, neighboring components could source an X value into the memory model due to an internal error, which might create unpredictable behavior. Hence, trapping illegal values on major block interfaces reduces debug time by isolating problems closer to their source.

Solution. Example 7-70 specifies an assertion that checks for unknown control signals due to unconnected ports or neighboring components sourcing an illegal X value.

Example 7-70 *SystemVerilog* check for illegal control signals.

```
assert property (@(posedge clk) not $isunknown(ce_n, rd_n, we_n))
   else $error ("Unknown control signals present (%0b).",
                {ce_n, rd_n, we_n));
```

7.14.1.2 Unknown address

Problem. If a neighboring block within an RTL model drives X values as address into a memory component, unpredictable behavior can occur, which could be difficult to isolate if the X value is not visible at the chip-level boundary.

we encourage using lint to identify unconnected ports

Motivation. When an engineer instantiates a memory model, it is possible that address signals were inadvertently left unconnected, which would result in X values driven into the memory model during simulation. Furthermore, neighboring components could source an X value into the memory model due to an internal error, which might create unpredictable behavior. Hence, trapping illegal values on major block interfaces reduces debug time by isolating problems closer to their source.

Solution. Example 7-71 specifies an assertion that checks for unknown address signals due to unconnected ports or neighboring components sourcing an illegal X value.

Example 7-71 *SystemVerilog* forbidden sequence

```
property forbidden_sequence;
  @ (posedge clk)
  not (~ce_n & (~rd_n | ~we_n ) ##0 $isunknown(addr));
endproperty
assert property (forbidden sequence)
   else $error ("Unknown address during memory request.");
```

7.14.1.3 Unknown store data

Problem. If a neighboring block within an RTL model drives X values as data into a memory component, unpredictable behavior can occur, which could be difficult to isolate if the X value is not visible at the chip-level boundary.

we encourage using lint to identify unconnected ports

Motivation. When an engineer instantiates a memory model, it is possible that data signals were inadvertently left unconnected, which would result in X values driven into the memory model during simulation. Furthermore, neighboring components could source an X value into the memory model due to an internal error, which might create unpredictable behavior. Hence, trapping illegal values on major block interfaces reduces debug time by isolating problems closer to their source.

Solution. Example 7-72 specifies an assertion that checks for unknown data signals due to unconnected ports or neighboring components sourcing an illegal X value.

Example 7-72 *SystemVerilog* forbidden sequence for store data unknown

```
property not storedata unknown;
  @ (posedge clk) not (~ce_n & ~we_n ##0 $isunknown(wdata));
endproperty
assert property (not_storedata_unknown)
    else $error ("Unknown write data presented for operation.");
```

7.14.1.4 Unknown read data

Pattern name. Unknown read data

Problem. If a neighboring block within an RTL model reads an X value as data from a memory component, unpredictable behavior can occur, which could be difficult to isolate if the X value is not visible at the chip-level boundary.

we encourage using lint to identify unconnected ports

Motivation. If a memory component was not initialized properly, or incorrectly addressed, a neighboring components could read an X value, which might create unpredictable behavior. Hence, trapping illegal values on major block interfaces reduces debug time by isolating problems closer to their source.

Solution. Example 7-72 specifies an assertion that checks for unknown data read from a memory.

Example 7-73 *SystemVerilog* forbidden sequence for read data unknown

```
property not readdata unknown;
  @ (posedge clk) not (~ce_n & ~rd_n ##0 $isunknown(data) ) ;
endproperty
assert property  (not_readdata_unknown)
   else $error ("Unknown data returned for read operation.");
```

7.15 Arbiter

Context. Arbiters are a critical component in systems containing shared resources. For example, a system containing multiple processors that share a as a common memory bus would require an arbitration scheme to prevent multiple processors accessing the bus at the same time. The are a number of different arbitration schemes, such as the unfair priority scheme or the fair round-robin scheme, which are not discussed in this book. In this section, we demonstrate a few common assertions, which are useful at identifying problems in the arbiter's implementation.

Example 7-74 demonstrates a simple arbiter interface, which is used in the following related discussion on assertions.

This arbiter chooses between four requestors. A requestor corresponds to a single bit in each signal (the same bit position.) A requestor may make a request by asserting its bit in the vector (for example `req[0]`). When it receives the corresponding grant (for example, gnt[0]) it is allowed access to the bus. The `hipri` signal is used by the requestor to signal a high priority request. It asserts both its request bit and its high priority bit (for example req[0], hipri[0]) to the arbiter. To simplify our discussion, it is expected that only one high priority request is asserted, if any.

Example 7-74 *Verilog* fragment for simple arbiter interface

```
module simple_arb(
  input [3:0] req;   // Request vector.
  input  [3:0] hipri; // High priority flags.
  output [3:0] gnt;  //  Grant vector.

  // Grant is asserted two cycles (minimum)
  // after request is asserted.
   ...
```

7.15.1 Assertions

7.15.1.1 Request timeout

Pattern name. Request timeout

Problem. Failure to generate a grant to a requestor in a reasonable period of time often indicates a a deadlock situation.

Motivation. Forward progress problems, related to bus transaction, can be difficult to identify during simulation. By adding performance restrictions on the servicing of a request to an arbiter, we can isolate deadlock situations closer in time (and location) to the error in the design.

Solution. Example 7-75 specifies a performance limit for the service of a request, which states that a `grant` must be received within 50 cycles of a `req`.

Example 7-75 *System Verilog* req receives a grant within N cycles

```
assert property (@(posedge clk) $rose(req) |-> ##[2:50] grant)
  else $error ("Request did not receive grant within timout limit.");
```

7.15.1.2 High priority request

Pattern name. High priority request

Problem. If a priority arbitration scheme is violated, a critical system event might not be serviced—causing unpredictable behavior that can be difficult to debug or isolate at the chip-level during verification.

Motivation. High priority arbitration schemes must ensure that the highest priority requestor immediately receives the next grant after the current transaction completes, otherwise a critical system event could go unserviced.

Solution. Example 7-76 demonstrates how to specify an assertion for a priority arbitration scheme. Note that we use the Verilog-2001 generate construct. The generate construct is similar to the PSL forall operator, which enables us to iterate through the various priority levels for a grant. This example asserts that for all request associated with a priority level, then if the request was the highest priority, a grant associated with that request must be generated within 50 cycles—and no other lower priority grant can be generated before the highest priority grant (which is checked by the $stable system functional).

Example 7-76 *SystemVerilog* high priority req's receive grant next

```
// Property to detect a high priority request and then expect
// the next grant change to be for this requestor.
property hipri_grant(N)
    @(posedge clk) ($rose(req[N] && hipri[N])
      |=> ($stable(grant) [*1:50] ##1 grant[N]))
endproperty

// Generate an assertion for each requestor.
genvar i;
generate for (i=0; i<4;i = i + 1)
  assert property (hipri_grant(i))
    else $error("Grant [%d] for hi priority not next or timedout.", i);
endgenerate
```

7.15.1.3 Round-robin arbitration

Pattern name. Round-robin arbitration

Problem. If a round-robin arbitration scheme is violated, than a particular requester could go un-serviced.

Motivation. Round-robin arbitration schemes are fair, in that a requester cannot receive a grant if any other request has been received, The arbiter will generate a grant in a circular or rotation

fashion to a neighboring request from the list. If the arbitration scheme were in error and skipped a requestor, then potentially *starvation* could occur, which means that the requestor would never be serviced.

Solution. In Example 7-77 we demonstrate how to specify assertions for a round-robin arbiter, to ensure that the scheme is fair (that is, no single requestor could be serviced multiple times if another request has been made—and the next request is the immediate neighboring request from a list). This assertion is written for consecutive pairs of requests (N and N+1.) It can be instantiated for each pair ((0, 1) (1, 2) (2, 3)) of requests. A different assertion could be written for nonconsecutive pairs.

Example 7-77 *SystemVerilog* round-robin arbitration assertion

```
// Request N+1 needs to be asserted 2 cycles before end of current
// transaction.

// Property to expect the next requestor is granted if they are
// requesting at the appropriate time before the transaction end.

property arb_rotation(N);
  @(posedge clk) (req[N+1] ##1
    req[N+1] & req[N] & gnt[N] & end_of_trans |=> gnt[N+1]);
endproperty

assert property (arb rotation(0))
    else $error("Grant did not switch to next request at end of transaction.");

// Next combination of request(1, 2) and current grant...
assert property (arb_rotation(1))
    else $error ("Grant did not switch to next request at end of transaction. ");
```

7.15.2 Functional coverage

7.15.2.1 Grant transition

Pattern name. Grant transition

Problem. If we do not adequately explore all possible combination of grants generated by an arbiter, then potentially a corner case error will go untested.

Motivation. Often, corner case bugs are related to complex combination of events, which are unexpected. The completion of one transaction may affect the next transaction after a specific period. Hence, it is important to identify which combination of grants generated by an arbiter have been explored.

Solution. Example 7-78 reports the occurrence of any pairing of back-to-back grants by an arbiter, which helps us determine which pairs have not been explored.

Example 7-78 *SystemVerilog* grant transfers between all pairs of requestors

```
genvar R1, R2;
generate  for(R1=0;R1=3;R1= R1+ 1)
  generate for(R2=0;R2<=3;R2= R2 + 1)
   cover property @ (posedg clk) (gnt[R1]; gnt[R2]);
  endgenerate
endgenerate
```

7.16 Summary

In this chapter, we explored a typical set of assertions and functional coverage points for queues, stacks, finite state machines, encoders, decoders, multiplexers, state table structures, memory, and arbiters. From this base set of assertions, we expect you would create additional assertions to cover your specific needs. Our experience has been during the process of specifying assertions and functional coverage, errors are detected and corrected. Furthermore, during verification the time spent debugging problems within your designs is significantly reduced