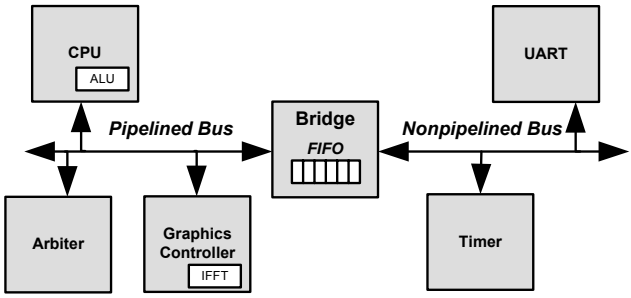
CHAPTER

8

DATAPATH

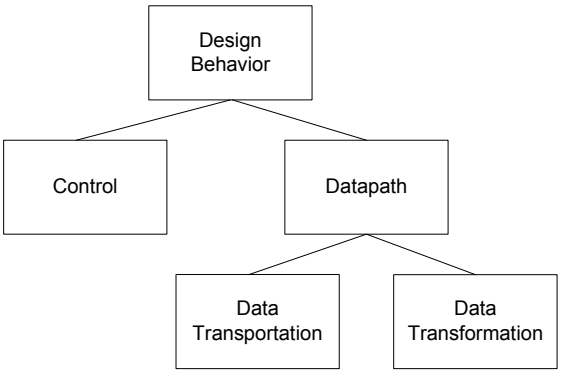
Design components are often classified as either datapath- or control-oriented. In the previous chapters, we demonstrated our assertion-based IP creation process for control-oriented design components. In this chapter, we focus on various datapath components that are likely to be found in today's platform-based SoC designs. Figure 8-1 illustrates our platform-based SoC example with various datapath components highlighted.

Figure 8-1. Platform-based SoC design with datapath components



Most people classify datapath as some form of design behavior that performs data processing. In Chattterjee [2006] and Foster *et al*. [2006], the authors propose refining the classification of datapath into either *data transport* or *data transform*, as illustrated in Figure 8-2.

Figure 8-2. Design behavior classification



A data transport design component essentially transports unchanged data (for example, packets) from multiple input sources to multiple output sources (for example, a PCI Express Data Link Layer design component). On the other hand, a data transform design component performs a mathematical computation (an algorithm) over different data input values (for example, an IFFT convolution block). Foster *et al.* [2006] argue that data transport blocks are amenable to formal verification (that is, model checking) due to the independence of the bits in the datapath, which makes the formal verification independent of the width of the datapath. However, this symmetry reduction technique (a formal verification abstraction) cannot be applied to data transform design components.

As we discussed in the introduction, our philosophy toward creating assertion-based IP in this book is to first focus on capturing the intent of the design in a set of assertions suitable for simulation, and then optimize any of the problematic assertions you encounter during formal verification only if you experience problems while attempting proof—and only if there is a clear return on effort to achieve a complete proof. Hence, in this chapter we have not limited our discussion to written assertions that are only amenable to formal verification, and we demonstrate our assertion-based IP creation process on both data transportation and data transform design components.

You will note that the data integrity class of properties we demonstrate in this chapter become progressively more difficult to express from one section to the next, even with the added expressive power of SVA local variables. In fact, we argue in Section 8.4, "Data compression" and Section 8.5, "Data decompression" that using assertion-based IP to check data integrity in simulation for this particular class of design components is not necessarily the best technique. Alternative solutions, such as scoreboarding, might be a better solution to the problem. Finally, even though this chapter focuses on datapath components, you will note that there are still many control oriented properties associated with these components that need to be expressed.

8.1 Multiport register file

A register file is a data transportation design component example that consist of an array of registers found in many of today's devices, such as a central processing unit (CPU) or a network router. These modern devices will almost always define a set of registers that are used to stage data that is to be transported between memory and various other connected functional units.

In this section, we introduce a simple multiport register file design component to demonstrate our assertion-based IP creation process. While reviewing this chapter, you might notice some similarities in our simple multiport register file to a simple network router. Hence, many of the concepts can be extended to more complicated switch type designs.

8.1.1 Block diagram interface description

Figure 8-3 illustrates a block diagram for a simple multiport register file design component. For our example, all signal transitions relate only to the rising edge of the clock (clk).

Table 8-1 provides a summary of the interface signals for our simple packet data compressor design example.

Figure 8-3 A simple multiport register file block diagram

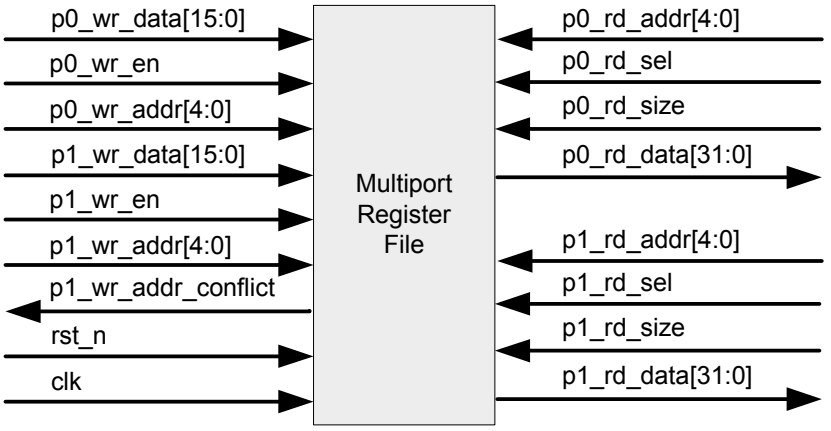


Table 8-1 Interface description for the multiport register file

Name	Description
p0_wr_data[15:0]	Write port 0 write data
p0_wr_en	Write port 0 write enable
p0_wr_addr[4:0]	Write port 0 address
p1_wr_data[15:0]	Write port 1 write data
p1_wr_en	Write port 1 write enable
p1_wr_addr[4:0]	Write port 1 address
p1_wr_addr_conflict	Write address conflict (p0 takes priority over p1)
p0_rd_addr[4:0]	Read port 0 address
p0_rd_sel	Read port 0 read select
p0_rd_size	Read port 0 read size (16-bit=0, 32-bit=1)
p0_rd_data[31:0]	Read port 0 register data
p1_rd_addr[4:0]	Read port 1 address
p1_rd_sel	Read port 1 read select
p1_rd_size	Read port 1 read size (16-bit=0, 32-bit=1)
p1_rd_data[31:0]	Read port 1 register data
clk	Synchronous clock
rst_n	Asynchronous reset

8.1.2 Overview description

Our simple multiport register file consists of 32 16-bit registers. There are two 16-bit input write ports and two 32-bit output read ports. Each write port can individually address any register within the register file. However, port p0 has higher write priority over p1 when both ports are attempting to write to the same register file address. A memory address write conflict results in p0 completing its write and the register file then asserts `p1_wr_addr_conflict` to indicate that the p1 write port attempt was unsuccessful.

The read ports can access either a 16-bit or 32-bit value. That is, if `p0_rd_size` is set to zero, then a 16-bit read on port p0 occurs. However, if `p0_rd_size` is set to one, then a 32-bit read of port p0 occurs. Note that a 16-bit read access can address any register within the register file. However, for a 32-bit read, the lower returned bits will always contain the value of an even address register while the upper bits will contain the value of an odd register. This means that if the `p0_rd_addr` (or `p1_rd_addr`) is set to an odd value (for example, five) then the read value for the lower 16 bits will consist of the value obtained from register four, while the read value for the upper 16 bits will consist of the value obtained from register five.

Basic register operation

The waveforms in Figure 8-4 illustrate a write operation for our simple multiport register file. The write operation begins with an address of `5'd0` and some arbitrary data value (represented by `'DA` in Figure 8-4) placed on port p0's input controls. If `p0_wr_en` is asserted at clock one then the input data is written into register zero. At clock two, the data value `'DB` is written into register address `5'd1` from port p0, and simultaneously, the data value `'DX` is written into register address `5'd4` from port p1. At clock three, the data value `'DY` is written into register address `5'd3` from port p1. Finally, at clock five, both ports p0 and p1 are trying to write data into register address `5'd2`. This creates a conflict that is resolved by writing the higher priority port p0's input data value of `'DC` into register address `5'd2` and asserting

p1_wr_addr_conflict. Hence, the input data value from port p1 `DZ is dropped at clock five.

Figure 8-4 Register file 16-bit write operation

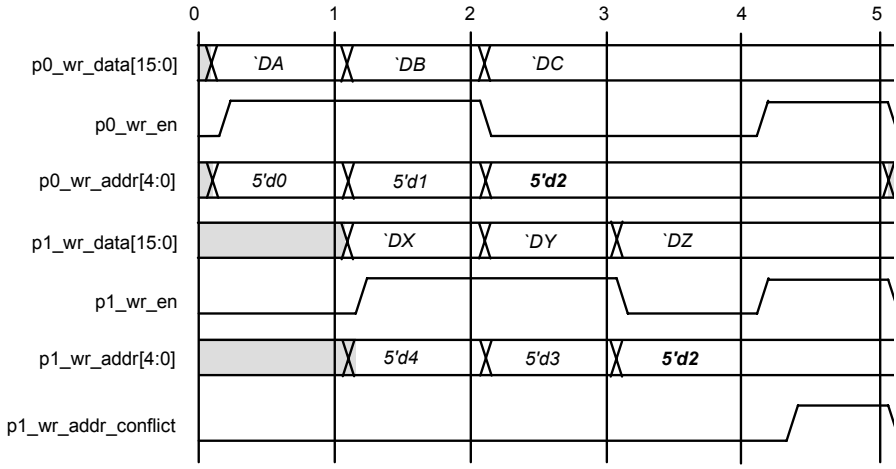
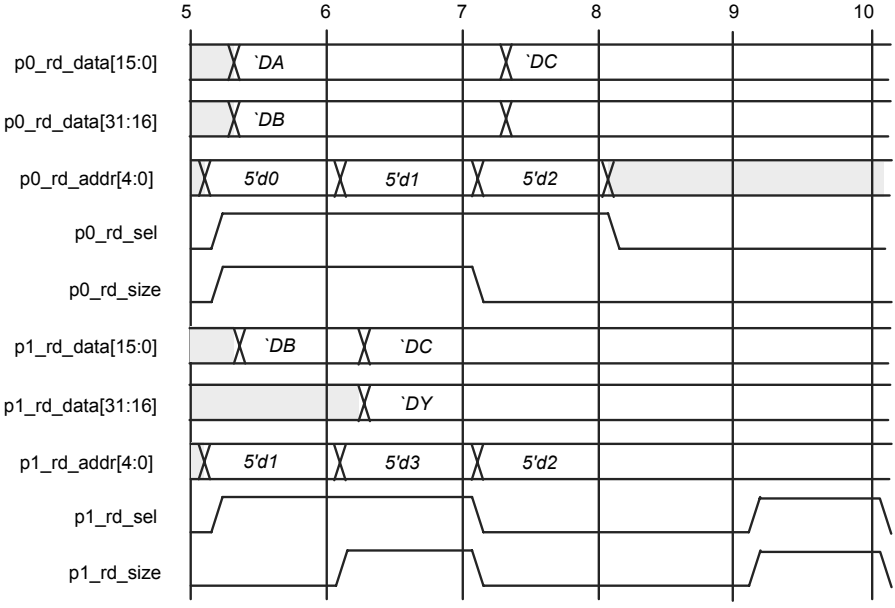


Figure 8-5 Register file 16-bit and 32-bit read operation



The waveforms in Figure 8-5 illustrate a write operation for our simple register file.

This figure illustrates the case where a 32-bit read from either an odd or even address yields the same result. That is, the upper 16 bits of a 32-bit read will always align with an odd address, while the lower 16 bits align with an even address (that is, odd address minus one). Hence, a 32-bit read to either 5'd1 or 5'd0 yields the same result as illustrated prior to clocks six and seven in our example.

8.1.3 Natural language properties

Prior to creating a set of SystemVerilog assertions for our simple multiport register file, we must identify a comprehensive list of natural language properties, as shown in Table 8-2. Although the set of properties found in this table is not necessarily comprehensive, it is representative of a real set of properties and sufficient to demonstrate our process.

Table 8-2 Simple multiport register file properties

Assertion name	Summary
a_reset_state_inactive	Initial value of registers after reset is 0
a_wr_rd_data_integrity	A write followed by a read on any port to the same register will return the last highest priority written data
a_reg_after_reset	Reads before any writes will read 0
a_rd_rd_data_integrity	A pair of reads from a register, without an intervening write on any port, will each return the same data.
a_rd_stable	Output read data without a read select signal will not change from its prior value

8.1.4 Assertions

To create a set of SystemVerilog assertions for our simple multiport register file, we begin defining the connection between our assertion-based monitor and other potential components within the testbench (such as a driver transactor and the DUV). We accomplish this goal by creating a SystemVerilog interface, as Figure 8-15 illustrates.

Example 8-7 on page 219 demonstrated an interface for our simple multiport register file example. For this case, our assertion-based monitor references the interface signals via the direction defined by the `monitor_mp` named modport.

Example 8-1 Encapsulate signals inside an interface

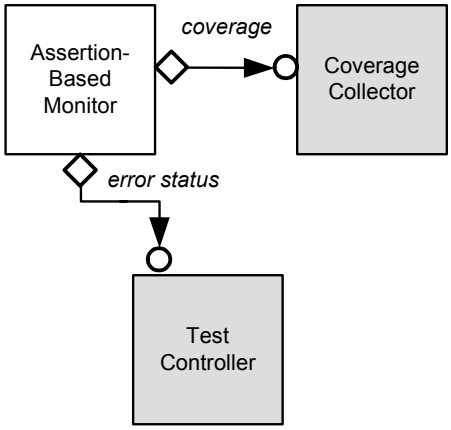
```
interface tb_register_file_if( input clk , input rst );

  bit [15:0] p0_wr_data;
  bit        p0_wr_en;
  bit  [4:0] p0_wr_addr;
  bit [15:0] p1_wr_data;
  bit        p1_wr_en;
  bit  [4:0] p1_wr_addr;
  bit        p1_wr_addr_conflict;
  bit  [4:0] p0_rd_addr;
  bit        p0_rd_sel;
  bit        p0_rd_size;
  bit [31:0] p0_rd_data;
  bit  [4:0] p1_rd_addr;
  bik        p1_rd_sel;
  bit        p1_rd_size;
  bit [31:0] p1_rd_data;

  modport dut (
    input clk , rst ,
    . . .
  );

  . . .

  modport monitor_mp (
    input clk , rst ,
    input p0_wr_data,
    input p0_wr_en,
    input p0_wr_addr,
    input p1_wr_data,

// Continued on next page
```

Example 8-1 Encapsulate signals inside an interface

```
    input p1_wr_en,
    input p1_wr_addr,
    input p1_wr_addr_conflict,
    input p0_rd_addr,
    input p0_rd_sel,
    input p0_rd_size,
    input p0_rd_data,
    input p1_rd_addr,
    input p1_rd_sel,
    input p1_rd_size,
    input p1_rd_data  );

endinterface
```

In addition to pin-level interfaces, we need a means for our simple multiport register file assertion-based monitor to communicate with various analysis verification components within the testbench. Hence, we introduce an error status analysis port within our monitor, as Figure 8-6 illustrates.

Figure 8-6 Error status and coverage analysis ports



In addition to assertion error status, coverage events are an important piece of analysis data that require their own analysis ports. We discuss coverage with respect to creating assertion-based IP separately in Section 5.4.

Upon detecting a register file error, the analysis port broadcasts the error condition (using an error status transaction object) to other verification components.

The *error status* transaction class is defined in Example 8-2, which is an extension of avm_transaction base class. The `tb_reg_file_status` class includes an `enum` that identifies the various types of register file errors and a set of methods to uniquely identify the specific error it detected.

Example 8-2 Error status class

```
class tb_reg_file status extends avm_transaction;

  typedef enum { ERR_RESET_STATE_INACTIVE,
                 ERR_WR_RD_DATA_INTEGRITY,
                 ERR_REG_AFTER_RESET,
                 ERR_RD_RD_DATA_INTEGRITY,
                 ERR_RD_STABLE } reg_file_err_status_t;

  reg_file_err_status_t   reg_file_err_status;
  bit [5:0] err_reg_addr;

  function void set_err_reset_state_inactive;
    reg_file_err_status = ERR_RESET_STATE_INACTIVE ;
  endfunction

  function void set_err_wr_rd_data_integrity (input bit [5:0] i);
    reg_file_err_status = ERR_WR_RD_DATA_INTEGRITY ;
    err_reg_addr = i;
  endfunction

  function void set_err_reg_after_reset (input bit [5:0] i);
    reg_file_err_status = ERR_REG_AFTER_RESET ;
    err_reg_addr = i;
  endfunction

  function void set_err_rd_rd_data_integrity (input bit [5:0] i);
    reg_file_err_status = ERR_RD_RD_DATA_INTEGRITY ;
    err_reg_addr = i;
  endfunction

  function void set_err_rd_stable(input bit [5:0] i);
    reg_file_err_status = ERR_RD_STABLE;
    err_port_num = i;
  endfunction
  . . .
endclass
```

Assertion 8-1, “Register file output inactive after reset” demonstrates our first property.

Assertion 8-1 Register file output inactive after reset

```
property p_reset_state_inactive;
  @(posedge monitor_mp.clk)
   $rose(monitor_mp.rst_n) |->
            (monitor_mp.p0_rd_data==32'd0) &
            (monitor_mp.p1_rd_data==32'd0) &
            (monitor_mp.p1_wr_addr_conflict==1'b0) ;
endproperty
a_reset_state_inactive :
  assert property (p_reset_state_inactive) else begin
    status = new();
    status.set_err_reset_state_inactive();
    status_ap.write(status);
  end
```

Assertion 8-2, "Register file data integrity" demonstrates our next property.

Assertion 8-2 Register file data integrity

```
// Parameterized property to check 16-bit data integrity
// Address A is a constant from a generate statement!

property p_wr_rd_data_integrity(A, we0, wa0, wdata0,
                                   we1, wa1, wdata1,
                                   re,  ra,  rdata, rdsize);
  logic [16:0] lv_data;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)

// (a) If either a 16-bit write to port 0 address A or to port 1
// (that is, the lower priority port) address A. . .
    (we0 & wa0==A, lv_data=wdata0) or
    (!(we0 & wa0==A) & (we1 & wa1==A), lv_data=wdata1)

// (b) followed by a sequence in which no writes to address A from
// either port, and eventually a read to address A. . .
##1 !(we0 & wa0==A | we1 & wa1==A) throughout (re & ra==A)[->1]

// (c) new read data is same as original read data, selecting
//    appropriate upper or lower 16-bit slice from 32-bit read value
|-> (lv_data==(rdsize & (A & 5b1)) ? rdata[31:16] : rdata[15:0]);
endproperty

generate
  begin
   genvar i;
    for (i = 0; i<=31; i++) begin

// Continued on next page
```

Assertion 8-2 Register file data integrity

```
      a_wr_rd_p0:
       assert property ( p_wr_rd_data_integrity( i,
                    monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
                    monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
                    monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
                    monitor_mp.p0_rd_sel,  monitor_mp.p0_rd_addr,
                    monitor_mp.p0_rd_data, monitor_mp.p0_rd_size)
                       ) else begin
            status = new();
            status.set_err_wr_rd_data_integrity(i);
            status_ap.write(status);
          end
      a_wr_rd_p1:
       assert property (p_wr_rd_data_integrity( i,
                    monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
                    monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
                    monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
                    monitor_mp.p1_rd_sel,  monitor_mp.p1_rd_addr,
                    monitor_mp.p1_rd_data, monitor_mp.p1_rd_size)
                       ) else begin
            status = new();
            status.set_err_wr_rd_data_integrity(i);
            status_ap.write(status);
          end
      end // for
    end // generate
endgenerate
```

Steps for creating data integrity assertions

When attempting to create a data integrity assertion for a register file, you might be overwhelmed initially by the complexity of the problem. However, we recommend that you approach this class of problems in the following manner:

- First, only consider a single output port when creating your assertion—we will consider the other output ports separately
- Next, only consider a single register address value to check for data integrity as part of the assertion—we will consider the other register addresses separately
- Finally, for simulation assertion-based IP, you must consider all possible *valid* input behaviors
- Then, either implicitly or explicitly create a separate assertion for each register address related to a specific

port. For example, an assertion could be written using a local variable that holds the last written address, which implicitly can handle all possible addresses.

Implicit versus explicit addresses

This implicit addressing approach has the advantage of handling large memories, yet the assertion can be more complicated to code or understand.

Alternatively, a set of explicit address assertion could be generated by creating a SystemVerilog generate statement that explicitly specifies all possible address values as a constant. This approach often has the advantage of being simpler to code, which we demonstrate in our next assertion. However, for large memories, the explicit addressing technique can grow the number of generated assertions rapidly and potentially create a performance problem.

In this section, we demonstrate both explicit and implicit addressing approaches. You might note that both approaches (for our example) result in about the same amount of code.

Our Assertion 8-2 begins by looking for a write on the higher priority port 0, or a write on the lower priority port 1 with no write on port 0 (see label (a) in our example), to a specific reference address of interest. Note that our reference address `A` in this example is actually a constant created by a generate statement, so it is unnecessary to store the address into a local variable.

Next, in the step labeled (b), our property looks for a sequence of no writes from either of the input ports to the specific address of interest (in this case, `A`) followed by a read to address `A`. Then, the property checks in the step labeled (c) that the read data is the same as the last write data. Notice that if a new write occurs prior to the read occurring, the antecedent on the currently evaluating property does not hold. This effectively ends the previous check and restarts the evaluation of the property for the new write to address `A`. Keep in mind that we are only interested in the last write to address `A`.

For Assertion 8-2, we are monitoring a 16-bit write to a specific memory address (A). Hence, for the 16-bit read, we can simply compare the previously written value (held in the lv_data local variable) with the lower 16 bits of the read data. However, for a 32-bit read, we need to account for a 16-bit write to an odd address, which we detect by the equation (A & 5'b1), and then compare the upper 16 bits of the 32-bit read value to the previously written value for the case when the address is odd. For the case when the address is even, we simply check the lower 16 read bits to the previously written value.

Assertion 8-3, "Register file data cleared after reset" demonstrates our next property. The technique we use for the coding problem demonstrated in Assertion 8-2 is also applied here.

Assertion 8-3 Register file data cleared after reset

```
// Parameterized property to check a single reg (16-bit data)
// Address A is a constant from a generate statement!

property p_reg_after_reset(A, we0, wa0, wdata0, we1, wa1, wdata1,
                                    re,  ra,  rdata, rdsize);
  @(posedge monitor_mp.clk)

// (a) After a reset
    $rose(monitor_mp.rst_n)

// (b) and a sequence in which no writes to reg address A from
// either port, and eventually a read to reg address A
##0 !(we0 & wa0==A | we1 & wa1==A) throughout (re & ra==A)[->1]

// (c) then the read data must be 0 (accounting for either a 16- or
// 32-bit read and that we are only monitoring a single reg addr A)
|-> (rdsize & (A & 5'b1)) ? rdata[31:16]=16'b0 :
                            rdata[15:0]==16'b0;
endproperty

generate
  begin
    genvar i;
    for (i = 0; i<=31; i++) begin

// Continued on next page
```

Assertion 8-3 Register file data cleared after reset

```
      a_rd_p0:
       // reads from port 0
       assert property ( p_reg_after_reset(i,
                 monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
                 monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
                 monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
                 monitor_mp.p0_rd_sel,  monitor_mp.p0_rd_addr,
                 monitor_mp.p0_rd_data, monitor_mp.p0_rd_size)) else
         begin
            status = new();
            status.set_err_init_rd_is_0(i);
            status_ap.write(status);
         end
      a_rd_p1:
       // reads from port 1
       assert property (p_reg_after_reset( i,
                 monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
                 monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
                 monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
                 monitor_mp.p1_rd_sel,  monitor_mp.p1_rd_addr,
                 monitor_mp.p1_rd_data, monitor_mp.p1_rd_size)) else
         begin
            status = new();
            status.set_err_data_integrity(i);
            status_ap.write(status);
         end
      end // for
    end
  endgenerate
```

The property begins whenever reset completes (see step (a) in example), which is detected by looking for a rising edge of the active low reset. Once this occurs, in step (b) we look for a sequence in which no writes occur on either input port, followed by a read to a specific memory address of interest (in our case, that is `A`). Once this occurs, in step (c) we check that the read value is zero. Note that we are monitoring 16-bit writes to a specific address. Hence, a 32-bit read must account for either an odd address (and check the upper 16 bits of the read data value) or it must account for an even address (and check the lower 16 bits of the read data value).

Read followed by another read to same address

Concerning data integrity, it is insufficient to simply check that the data is valid for the case of a read following a write to the same address. We also need to ensure that a read followed by another read to the same address (with no

writes to the address we are monitoring) results in the same value. Hence, property `a_rd_rd_data_integrity` defined in Table 8-2, "Simple multiport register file properties" checks this situation.

To simplify understanding this property, we have broken the problem down into four cases that you must consider:

1. A 16-bit read to a specific address, followed by another 16-bit read to the same address, and no writes have occurred to the address we are monitoring
2. A 16-bit read to a specific address, followed by another 32-bit read to the same address, and no writes have occurred to the address we are monitoring
3. A 32-bit read to a specific address, followed by another 16-bit read to the same address, and no writes have occurred to the address we are monitoring
4. A 32-bit read to a specific address, followed by another 32-bit read to the same address, and no writes have occurred to the address we are monitoring

Although there are similarities between each case—there are sufficient differences that warrant partitioning these behaviors into separate properties (at least initially, during the development phase, to simplify the complexity that can occur when the properties are combined into a single property).

Assertion 8-4, "16-bit read followed by 16-bit read data integrity" is our first case. Our property begins by looking for the occurrence of a 16-bit read, which is labeled as (a) in our example. Once a 16-bit read occurs, the data and address is stored in a local variable for future reference.

The next step, which is labeled as (b) in our example, is to look for a sequence of an eventual read to the same address (stored in the local variable `lv_A`), and throughout this sequence there are no new writes to the address that we are monitoring. If this sequence is detected, then the new 16-bit read data, which is the lower 16-bits of the 32-bit read port, must match the original 16-bit read data, which were stored

in local variable `lv_data`. This step is labeled as (c) in our example.

Assertion 8-4 16-bit read followed by 16-bit read data integrity

```
property p_16_rd_16_rd_integrity(we0, wa0, wdata0,
                                 we1, wa1, wdata1,
                                 re,  ra,  rdata, rdsize);
  logic [15:0] lv_data;
  logic [5:0]  lv_A;

  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
// (a) 16-bit read occurs, grab data and address
      (re & !rdsize, lv_data = rdata[15:0], lv_A = ra)
// (b) another 16-bit read to same address and no writes
  ##1 !((we0 & wa0==lv_A) | (we1 & wa1==lv_A)) throughout
                                 (re & ra==lv_A & !rdsize)[->1] |->
// (c) new 16-bit read data same as original 16-bit read data
                                          ( lv_data == rdata[15:0] );
endproperty
a_16_rd_16_rd_integrity_p0: // port 0 read
  assert property ( p_16_rd_16_rd_integrity(
          monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
          monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
          monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
          monitor_mp.p0_rd_sel,  monitor_mp.p0_rd_addr,
          monitor_mp.p0_rd_data, monitor_mp.p0_rd_size)) else
        begin
          status = new();
          status.set_err_rd_rd_integrity(monitor_mp.p0_rd_addr);
          status_ap.write(status);
        end
a_16_rd_16_rd_integrity_p1: // port 1 read
  assert property ( p_16_rd_16_rd_integrity(
          monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
          monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
          monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
          monitor_mp.p1_rd_sel,  monitor_mp.p1_rd_addr,
          monitor_mp.p1_rd_data, monitor_mp.p1_rd_size)) else
        begin
          status = new();
          status.set_err_rd_rd_integrity(monitor_mp.p1_rd_addr);
          status_ap.write(status);
        end
```

Assertion 8-5, “16-bit read followed by 32-bit read data integrity” demonstrates our second case. Our property begins exactly like our previous example. However, in step (b) we are now looking for a 32-bit read. Once this sequence is detected, the appropriate 16-bit slice of the new 32-bit

read data (based on an odd or even read address) is compared to the original 16-bit read data, as demonstrated in step (c) in our example.

Assertion 8-5 16-bit read followed by 32-bit read data integrity

```
property p_16_rd_32_rd_integrity(we0, wa0, wdata0,
                                 we1, wa1, wdata1,
                                 re,  ra,  rdata, rdsize);
  logic [15:0] lv_data;
  logic [5:0]  lv_A;

  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
// (a) 16-bit read occurs, grab data and address
      (re & !rdsize, lv_data = rdata[15:0], lv_A = ra)
// (b) another 32-bit read to same address and no writes
  ##1 !((we0 & wa0==lv_A) | (we1 & wa1==lv_A)) throughout
                                 (re & ra==lv_A & rdsize)[->1] |->
// (c) new read data is same as original read data, selecting
//    appropriate upper or lower 16-bit slice from 32-bit read value
                          ( lv_data == (lv_A & 5'b1) ? rdata[31:16] :
                                                       rdata[15:0] );
endproperty
a_16_rd_32_rd_integrity_p0: // port 0 read
  assert property ( p_16_rd_32_rd_integrity(
          monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
          monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
          monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
          monitor_mp.p0_rd_sel,  monitor_mp.p0_rd_addr,
          monitor_mp.p0_rd_data, monitor_mp.p0_rd_size)) else
        begin
          status = new();
          status.set_err_rd_rd_integrity(monitor_mp.p0_rd_addr);
          status_ap.write(status);
        end
a_16_rd_32_rd_integrity_p1: // port 1 read
  assert property ( p_16_rd_32_rd_integrity(
          monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
          monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
          monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
          monitor_mp.p1_rd_sel,  monitor_mp.p1_rd_addr,
          monitor_mp.p1_rd_data, monitor_mp.p1_rd_size)) else
        begin
          status = new();
          status.set_err_rd_rd_integrity(monitor_mp.p1_rd_addr);
          status_ap.write(status);
        end
```

Assertion 8-6, "32-bit read followed by 16-bit read data integrity" demonstrates our third case. Our property begins

similar to our first example, except the appropriate upper or lower 16-bit read data is stored for future comparison based on the reference to either an odd or even address (see step (a) in our example). In step (c) from our example, the new 16-bit read data, which comprises the lower bits of the read port, is compared to the original 16-bit read data, which is stored in local variable `lv_data`.

Assertion 8-6 32-bit read followed by 16-bit read data integrity

```
property p_32_rd_16_rd_integrity(we0, wa0, wdata0,
                                 we1, wa1, wdata1,
                                 re,  ra,  rdata, rdsize);
  logic [15:0] lv_data;
  logic [5:0]  lv_A;

  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
// (a) 32-bit read occurs, grab appropriate 16-bit data and address
    (re & rdsize, lv_data = (ra&5'b1) ? rdata[31:16] : rdata[15:0],
                                                       lv_A = ra)
// (b) another 16-bit read to same address and no writes
##1 !((we0 & wa0==lv_A) | (we1 & wa1==lv_A)) throughout
                            (re & ra==lv_A & !rdsize)[->1] |->
// (c) new 16-bit read data same as original read data
                                        ( rdata[15:0] == lv_data );
endproperty
a_32_rd_16_rd_integrity_p0: // port 0 read
  assert property ( p_32_rd_16_rd_integrity(
          monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
          monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
          monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
          monitor_mp.p0_rd_sel,  monitor_mp.p0_rd_addr,
          monitor_mp.p0_rd_data, monitor_mp.p0_rd_size)) else
        begin
          status = new();
          status.set_err_rd_rd_integrity(monitor_mp.p0_rd_addr);
          status_ap.write(status);
        end
a_32_rd_16_rd_integrity_p1: // port 1 read
  assert property ( p_32_rd_16_rd_integrity(
          monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
          monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
          monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
          monitor_mp.p1_rd_sel,  monitor_mp.p1_rd_addr,
          monitor_mp.p1_rd_data, monitor_mp.p1_rd_size)) else
        begin
          status = new();
          status.set_err_rd_rd_integrity(monitor_mp.p1_rd_addr);
          status_ap.write(status);
        end
```

Assertion 8-7, “32-bit read followed by 32-bit read data integrity” demonstrates our final case. Our property begins by looking for the occurrence of a 32-bit read, which is labeled as (a) in our example. When this event is detected, the data and address are stored in a local variable for future reference. The next step, which is labeled (b) in our example, is to look for a sequence of an eventual read to the same address (stored in the local variable `lv_A`), and throughout this read sequence there are no new 16-bit writes to either the even (lower) or odd (upper) address that we are monitoring. To accomplish this, in step (b) we force the write address to an odd value with the equation:

```
((wa0|5'b1)==(A|5'b1))
```

By doing this, we are able to detect the occurrence of a 16-bit write to either the even or odd address for the read address we are monitoring. Note that it is not necessary to know specifically which even or odd address was written to abort our check. Finally, in step (c), we compare the new 32-bit read value to the previously stored 32-bit read value.

Assertion 8-7 32-bit read followed by 32-bit read data integrity

```
property p_32_rd_32_rd_integrity(we0, wa0, wdata0,
                                 we1, wa1, wdata1,
                                 re,  ra,  rdata, rdsize);
  logic [31:0] lv_data;
  logic [5:0]  lv_A;

  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
// (a) 32-bit read occurs, grab appropriate 16-bit data and address
    (re & rdsize, lv_data = rdata, lv_A = ra)
// (b) another 32-bit read to same address and no 16-bit writes to
//     either the upper or lower word the original read address
##1 !(we0 & ((wa0|5'b1)==(A|5'b1)) | we1 & ((wa1|5'b1)==(A|5'b1)))
                  throughout (re & ra==lv_A & rdsize)[->1] |->
// (c) new 32-bit read data same as original read data
                                                ( rdata == lv_data );
endproperty
// Continued on next page
```

Assertion 8-7 32-bit read followed by 32-bit read data integrity

```
a_32_rd_32_rd_integrity_p0: // port 0 read
  assert property ( p_32_rd_32_rd_integrity(
          monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
          monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
          monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
          monitor_mp.p0_rd_sel,  monitor_mp.p0_rd_addr,
          monitor_mp.p0_rd_data, monitor_mp.p0_rd_size)) else
        begin
          status = new();
          status.set_err_rd_rd_integrity(monitor_mp.p0_rd_addr);
          status_ap.write(status);
        end
a_32_rd_32_rd_integrity_p1: // port 1 read
  assert property ( p_32_rd_32_rd_integrity(
          monitor_mp.p0_wr_en,   monitor_mp.p0_wr_addr,
          monitor_mp.p0_wr_data, monitor_mp.p1_wr_en ,
          monitor_mp.p1_wr_addr, monitor_mp.p1_wr_data,
          monitor_mp.p1_rd_sel,  monitor_mp.p1_rd_addr,
          monitor_mp.p1_rd_data, monitor_mp.p1_rd_size)) else
        begin
          status = new();
          status.set_err_rd_rd_integrity(monitor_mp.p1_rd_addr);
          status_ap.write(status);
        end
```

Assertion 8-8, “Register file output stable when not selected” demonstrates our next property.

Assertion 8-8 Register file output stable when not selected

```
property p_rd_stable (rd_sel, rd_data);
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
        (!rd_sel) |-> (rd_data == $past(rd_data));
endproperty
a_rd_stable_p0 :
  assert property (p_rd_stable(monitor_mp.p0_rd_sel,
                               monitor_mp.p0_rd_data)) else begin
    status = new();
    status.set_err_rd_stable(0);
    status_ap.write(status);
  end
a_rd_stable_p1 :
  assert property (p_rd_stable(monitor_mp.p1_rd_sel,
                               monitor_mp.p1_rd_data)) else begin
    status = new();
    status.set_err_rd_stable(1);
    status_ap.write(status);
  end
```

8.1.5 Encapsulate properties

In this step, we turn our set of related multiport register file assertions (and any coverage properties) into a reusable assertion-based monitor verification component. We add analysis ports to our monitor for communication with other simulation verification components, as Chapter 3 “The Process” demonstrates.

Example 8-3 Encapsulate properties inside a module

```
import avm_pkg::*;
import tb_tr_pkg::*; // tb_reg_file_status class definition

module tb_register_file_mon (
  interface.monitor_mp monitor_mp
);

  avm_analysis_port #(tb_reg_file_status)
                        status_ap = new("status_ap", null);
  avm_analysis_port #(tb_reg_file_coverage)
                        coverage_ap = new("coverage_ap", null);

  tb_reg_file_status status;
  . . .
 // Any required modeling code here (to model environment)
  . . .
 // All assertions and coverage properties here
  . . .
endmodule
```

8.2 Data queue

A queue is a buffer data structure type of design component that provides data transportation services to its various entities, such as data or control commands that are stored unmodified for later processing. Probably the most well-known form of a queue is the first-in-first-out (FIFO) queue. In a FIFO queue, the first element placed in the queue will be the first one out. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element

can be accessed. There are also non-FIFO queue data structures, such as a priority queue.

Queues are typically verified separately from other design components to ensure their proper operation. Their interfaces are generally simple and can be easily isolated from their driving logic. Queues generally are fixed in depth, though they can be parameterized to support various depths. Dynamically resized queues, such as an elasticity buffer, have recently become popular—providing a means to *smooth out* or balance the flow from the burst of data within a system.

Our goal in this section is to demonstrate the *process* of creating an assertion-based IP verification component versus teaching you all the details about various complex queues. We ask you to focus on the techniques for creating the assertions and verification IP. By using our generic, simple, eight-entry queue, we hope that you will be able to set aside the details of a particular larger complex queue that could otherwise overwhelm you and distract you from the learning objectives. After understanding the process we present, you should be able to extend these ideas to create assertion-based verification IP for more complex queues.

8.2.1 Block diagram interface description

Figure 8-7 illustrates a block diagram for a simple eight-entry queue design. For our example, all signal transitions relate only to the rising edge of the clock (clk).

Table 8-3 provides a summary of the interface signals for our simple eight-entry queue design example.

Figure 8-7 A simple eight-entry queue block diagram

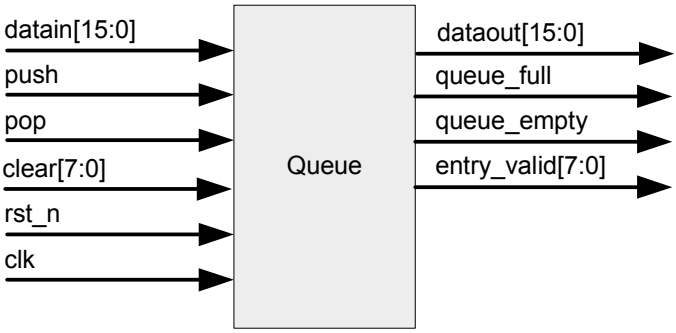


Table 8-3 Interface signals for our simple eight-entry queue

Name	Description
datain[15:0]	Input data to the queue
push	Command to enqueue an entry into the queue
pop	Command to dequeue entry from queue
clear[7:0]	Setting bit active high clears a queue entry (younger entries shift to occupy cleared entries)
dataout[15:0]	Data value for dequeued entry
queue_full	All entries in the queue are full
queue_empty	All entries in the queue are empty
entry_valid[7:0]	Indicates valid entries (rhs bit represents oldest entry while lhs bits represent youngest entries)
rst_n	Asynchronous active low reset
clk	Synchronous clock

8.2.2 Overview description

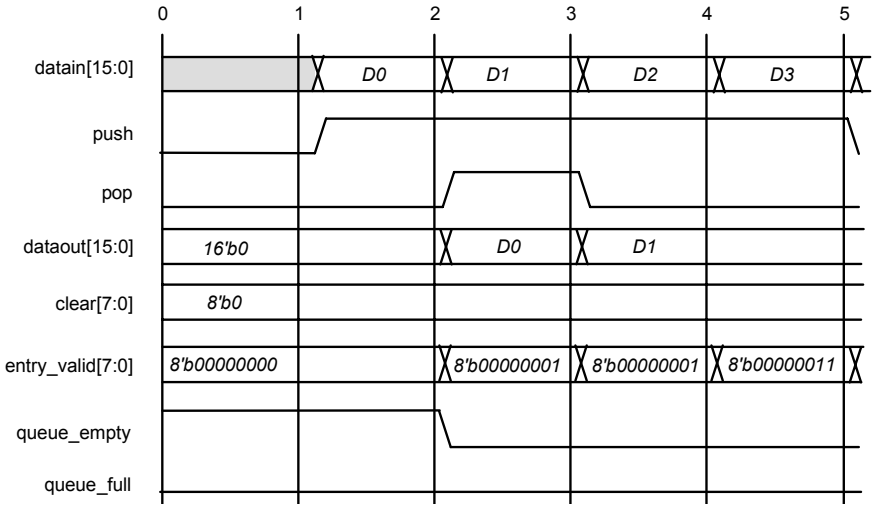
Our simple eight-entry queue consists of a 16-bit input data port with an enqueue (`push`) signal, an indicator for when the queue is full, a 16-bit output port with a dequeue (`pop`) signal, and an indicator for when the queue is empty. One unique aspect of our queue is the ability to remove an entry from the middle of the queue using the `clear` entry command.

Basic enqueue and dequeue operation

The waveforms in Figure 8-8 illustrate a typical enqueue (push) and dequeue (pop) operation for our simple queue.

Our simple eight-entry queue is initially empty, which is indicated by both the active `queue_empty` signal and the `entry_valid` signals set to zero and the `dataout` value is set to zero. Prior to clock two, a `D0` value is placed on the `datain` bus, and the `push` signal is asserted. This illustrates an *enqueue* operation. The first (right-hand side) bit of `entry_valid` is set (after clock two) to indicate that there is a single item in the queue, and the `dataout` value now represents the first item placed in the queue.

Figure 8-8 Enqueue and dequeue operation



Prior to clock three, a `D1` value is placed on the `datain` bus, and the `push` signal is asserted, which again illustrates an enqueue operation. At the same time, a `pop` signal is asserted, which illustrates a simultaneous *dequeue* operation. After clock three, the end item in the queue (`D1`) appears on the `dataout` bus and remains stable until some time in the future when either it is cleared or another queue entry is dequeued. Notice after clock three, since we just performed a simultaneous enqueue and dequeue operation, the `entry_valid` signals indicate only one item is in the

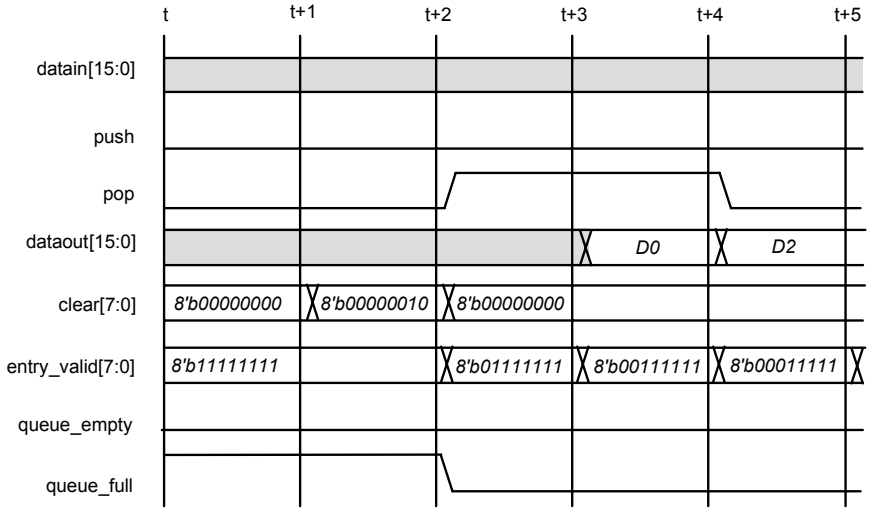
queue (which represents the D1 value that was just enqueued)

Prior to clock four, a D2 value is placed on the datain bus, and the push signal is asserted to perform an enqueue operation. The entry_valid signals after clock four now indicate that there are two valid items in the queue (since each bit represents a valid entry item for our simple queue). As new enqueue operations occur, additional entry_valid bits will be set in a contiguous fashion from right to left. That is, the oldest enqueued entry will be represented by the right-hand-most set bit while the youngest entry is represented by the left-hand-most set bit.

Basic clear operation

The waveform in Figure 8-9 illustrates the removal of a previous enqueued entry from the middle of the queue.

Figure 8-9 Clear operation



For our example, we are assuming that the queue has been filled with a sequence of values (D0, D1, D2,...,D7), which is indicated by the active queue_full signal and all bits of

entry_valid being set prior to clock t. The first item in the queue (D0) appears on the dataout port.

Prior to clock t+2, the clear command is set to 8'b00000010, indicating that the second oldest item in the queue is to be cleared (that is, the previous enqueued value of D1 has been cleared). Notice after clock t+2 that the queue_full signal is no longer active, and the entry_valid indicator signals have shifted right to occupy the previously cleared entry. You can see that there is now one unoccupied entry in the queue.

Prior to clock t+3, a dequeue command occurs, and the oldest item (represented as D0) in the queue appears on dataout after clock t+3. The entry_valid value now shifts to the right to indicate that there are two available queue entries.

Prior to clock t+4, another dequeue command occurs. This time, and the oldest item in the queue appears on dataout. For our example, we illustrate this item as D2, since D1 was previously cleared on clock t+2. Notice that entry_valid now indicates that there are three available queue entries after the previous dequeue command on clock t+4.

8.2.3 Natural language properties

Prior to creating a set of SystemVerilog assertions for our simple queue design, we must identify a comprehensive list of natural language properties, as shown in Table 8-4. Although the set of properties found in this table is not necessarily comprehensive, it is representative of a real set of properties and sufficient to demonstrate our process.

Table 8-4 Simple queue design properties

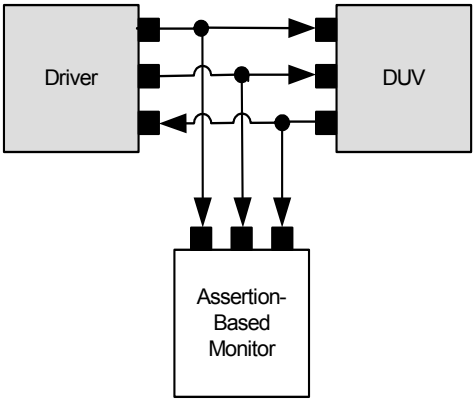
Assertion name	Summary
a_reset_state	Initial state after reset is empty and no entries are valid

Assertion name	Summary
a_no_overflow	A push cannot occur when the queue is full
a_queue_full	When all bits of entry_queue are set, then queue_full must be asserted
a_no_underflow	A pop cannot occur for an empty queue
a_queue_empty	Either `entry_queue` has at least one bit set, or `queue_empty` is asserted
a_push_pop_clear_entry_valid	A push, pop, and clear operation should set entry_valid to the correct number of entries in the queue
a_contiguous_entry_valid	The entry_valid bits are always contiguous and shifted to the right
a_no_invalid_entry_clear	A clear bit should not be asserted for an invalid entry
a_stable_dataout	The output data shall not change in the next cycle when pop is not asserted
a_data_integrity	A datum pushed into the queue and not cleared should exit the queue after the prior entries are popped or cleared

8.2.4 Assertions

To create a set of SystemVerilog assertions for our simple queue design, we begin defining the connection between our assertion-based monitor and other potential components within the testbench (such as a driver transactor and the DUV). We accomplish this goal by creating a SystemVerilog interface, as Figure 8-10 illustrates.

Figure 8-10 SystemVerilog interface



Example 8-4 on page 205 demonstrated an interface for our queue example. For this case, our assertion-based monitor references the interface signals with the direction defined by the `monitor_mp` named modport.

Example 8-4 Encapsulate signals inside an interface

```
interface tb_queue_if( input clk , input rst );

  parameter int DATA_SIZE  = 16;
  parameter int ENTRY_SIZE = 8;

  bit [DATA_SIZE-1:0]   datain;
  bit                   push;
  bit                   queue_full;
  bit                   queue_empty;
  bit [ENTRY_SIZE-1:0]  entry_valid;
  bit [DATA_SIZE-1:0]   dataout;
  bit                   pop;
  bit [ENTRY_SIZE-1:0]  clear;

  modport queue_mp (
    . . .
  );

  . . .

  modport monitor_mp (
    input        clk , rst_n ,
    input        datain ,
    input        push ,
    input        queue_full ,
    input        queue_empty ,
    input        entry_valid ,

// Continued on next page
```

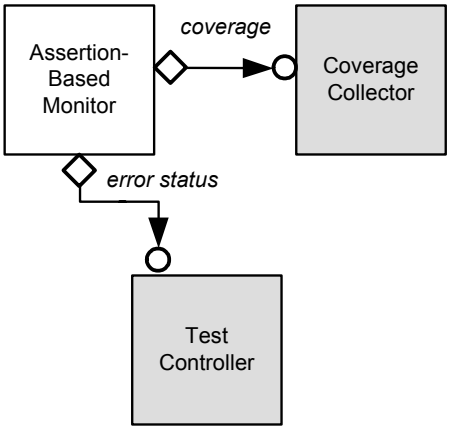
Example 8-4 Encapsulate signals inside an interface

```
    input        dataout ,
    input        pop ,
    input        clear
  );

endinterface
```

In addition to pin-level interfaces, we need a means for our simple queue design assertion-based monitor to communicate with various analysis verification components within the testbench. Hence, we introduce an error status analysis port within our monitor, as Figure 8-11 illustrates.

In addition to assertion error status, coverage events are often an important piece of analysis data that requires its own analysis port, as illustrated in Figure 8-11.

Figure 8-11 Error status and coverage analysis ports



Upon detecting an error, the analysis port broadcasts the error condition (using an error status transaction object) to other verification components.

The *error status* transaction class is defined in Example 8-5, which is an extension of avm_transaction base class. The `tb_queue_status` class includes an `enum` that identifies the various types of queue errors and a set of methods to uniquely identify the specific error it detected.

Example 8-5 Error status class

```
class tb_queue_status extends avm_transaction;

  typedef enum { ERR_RESET_STATE,
                 ERR_NO_OVERFLOW,
                 ERR_QUEUE_FULL,
                 ERR_NO_UNDERFLOW,
                 ERR_QUEUE_EMPTY,
                 ERR_PUSH_POP_CLEAR_ENTRY_VALID,
                 ERR_CONTIGUOUS_ENTRY_VALID,
                 ERR_NO_INVALID_ENTRY_CLEAR,
                 ERR_STABLE_DATAOUT,
                 ERR_DATA_INTEGRITY } queue_status_t;

  queue_status_t   queue_status;

  function void set_err_reset_state;
    queue_status = ERR_RESET_STATE ;
  endfunction

  function void set_err_no_overflow;
    queue_status = ERR_NO_OVERFLOW ;
  endfunction

  function void set_err_queue_full;
    queue_status = ERR_QUEUE_FULL;
  endfunction

  function void set_err_no_underflow;
    queue_status = ERR_NO_UNDERFLOW ;
  endfunction

  function void set_err_queue_empty;
    queue_status = ERR_QUEUE_EMPTY;
  endfunction

  function void set_err_push_pop_clear_entry_valid;
    queue_status = ERR_PUSH_POP_CLEAR_ENTRY_VALID ;
  endfunction

  function void set_err_contiguous_entry_valid;
    queue_status = ERR_CONTIGUOUS_ENTRY_VALID ;
  endfunction

  function void set_err_no_invalid_entry_clear;
    queue_status = ERR_NO_INVALID_ENTRY_CLEAR ;
  endfunction
  function void set_err_stable_dataout;
    queue_status = ERR_STABLE_DATAOUT ;
  endfunction

  function void set_err_data_integrity;
    queue_status = ERR_DATA_INTEGRITY ;
  endfunction
. . .

endclass
```

We are now ready to write assertions for our simple queue design properties defined in Table 8-4 on page 203.

Assertion 8-9, “Queue inactive after reset” demonstrates our first property.

Assertion 8-9 Queue inactive after reset

```
property p_reset_state;
  @(posedge monitor_mp.clk)
    $rose(monitor_mp.rst_n) |->
           (monitor_mp.entry_valid==8'b0) &
           (monitor_mp.queue_full==1'b0) &
           (monitor_mp.queue_empty==1'b1);
endproperty
a_reset_state :
  assert property (p_reset_state) else begin
    status = new();
    status.set_err_reset_state();
    status_ap.write(status);
  end
```

Assertion 8-10, “Queue full” demonstrates our next two properties associated with a full queue.

Assertion 8-10 Queue full

```
property p_no_overflow;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
    monitor_mp.queue_full |-> !monitor_mp.push;
endproperty
a_no_overflow :
  assert property (p_no_overflow) else begin
    status = new();
    status.set_err_no_overflow();
    status_ap.write(status);
  end

property p_queue_full;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
    (monitor_mp.queue_full == &(monitor_mp.entry_valid));
endproperty
a_queue_full :
  assert property (p_queue_full) else begin
    status = new();
    status.set_err_queue_full();
    status_ap.write(status);
  end
```

Assertion 8-11, “Queue empty” demonstrates our next two properties associated with an empty queue.

Assertion 8-11 Queue empty

```
property p_no_underflow;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
    monitor_mp.queue_empty |-> !monitor_mp.pop;
endproperty
a_no_underflow :
  assert property (p_no_underflow) else begin
    status = new();
    status.set_err_no_underflow();
    status_ap.write(status);
  end

property p_queue_empty;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
    ( |(monitor_mp.entry_valid) != monitor_mp.queue_empty);
endproperty
a_queue_full :
  assert property (p_queue_full) else begin
    status = new();
    status.set_err_queue_full();
    status_ap.write(status);
  end
```

Assertion 8-12, “Correct entry_valid after push, pop, and clear” demonstrates our next two properties. The first property specifies that the `entry_valid` signal must have the correct number of bits set as a result of a combination of push, pop, and clear commands at any given cycle. The second property specifies that the set `entry_valid` bits must be contiguous, and shifted to the right.

Assertion 8-12 Correct entry_valid after push, pop, and clear

```
property p_push_pop_clear_entry_valid;
  bit [3:0] lv_cnt; // count valid entries after push, pop, clear
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
     (1, lv_cnt =
      ($countones(monitor_mp.entry_valid) + monitor_mp.push -
       $countones(monitor_mp.clear | {7'b0,monitor_mp.pop}) ) |=>
                                  $countones(entry_valid)==lv_cnt);
endproperty

// Continued on next page
```

Assertion 8-12 Correct entry_valid after push, pop, and clear

```
a_push_pop_clear_entry_valid :
  assert property (p_ppush_pop_clear_entry_valid) else begin
    status = new();
    status.set_err_push_pop_clear_entry_valid();
    status_ap.write(status);
  end

property p_contiguous_entry_valid;
  bit [3:0] lv_cnt;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
     (1, lv_cnt =
      ($countones(monitor_mp.entry_valid) + monitor_mp.push -
       $countones(monitor_mp.clear | {7'b0,monitor_mp.pop}) ) |=>
          &((8'b11111111<lv_cnt)|entry_valid);
endproperty
a_contiguous_entry_valid :
  assert property (p_contiguous_entry_valid) else begin
    status = new();
    status.set_err_contiguous_entry_valid();
    status_ap.write(status);
  end
```

Assertion 8-13, “No cleared invalid entry” demonstrates our next property.

Assertion 8-13 No cleared invalid entry

```
property p_no_invalid_entry_clear;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
     |(monitor_mp.clear) |->
             |(monitor_mp.entry_valid & monitor_mp.clear));
endproperty
a_no_invalid_entry_clear :
  assert property (p_no_invalid_entry_clear) else begin
    status = new();
    status.set_err_no_invalid_entry_clear();
    status_ap.write(status);
  end
```

Assertion 8-14, “When no dequeue operation dataout is stable” demonstrates our next property.

Assertion 8-14 When no dequeue operation dataout is stable

```
property p_stable_dataout;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
     !monitor_mp.pop |=> $stable(monitor_mp.dataout));
endproperty
a_stable_dataout :
  assert property (p_stable_dataout) else begin
    status = new();
    status.set_err_stable_dataout();
    status_ap.write(status);
  end
```

So far, our data queue properties have focused more on the control behavior details for our simple queue. From an end-to-end (or black-box) perspective, probably the most important queue properties are related to data integrity. That is, data entering the queue, which has never been cleared, will eventually exit the queue uncorrupted in the correct order.

When are assertions not appropriate?

Up to this point, we have expressed most of our properties purely through the set of assertion constructs found in SystemVerilog. However, not all design behaviors are easily expressed by only using assertion constructs. Some complex behavior often requires a combination of additional modeling code that is then combined with various SVA constructs to properly specify it. In addition, there are some design behaviors that are best specified only using modeling code. To illustrate what we are talking about, let's examine the following simplified example. For this case, we will express a data integrity property on our simple queue. However, to simplify the problem further, we assume that our queue does not support the entry clear capabilities (that is, there is no `clear[7:0]` control and all items enqueued will eventually be dequeued). With this simplification, we write Assertion 8-15 to verify that data entering the queue always exits without corruption.

Assertion 8-15 Data integrity example with additional modeling

```
// Modeling code to identify specific push and pops

reg [2:0] r_pop_tag, r_push_tag;

always @(posedge monitor_mp.clk or negedge monitor_mp.rst_n) begin
   if (~monitor_mp.rst_n) begin
     r_pop_tag <= 0;
     r_push_tag <= 0;
   end
   else begin
     if (pop) r_pop_tag <= r_pop_tag+1;
     if (push) r_push_tag <= r_push_tag+1;
   end
end

// Example property for simple queue that doesn't support clear!

property p_data_integrity;
  bit [15:0] lv_datain;
  bit [2:0] lv_tag;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
     (monitor_mp.push, lv_datain=datain, lv_tag=r_push_tag) |->
                      (monitor_mp.pop && r_pop_tag == lv_tag)[->1])
                  ##1 (monitor_mp.dataout == lv_datain);
endproperty
```

Assertion 8-15, data queue integrity, is an example of a property that is difficult to express with SVA constructs alone. For example, to express this property, we need a means to uniquely (and globally) identify entries as they are being enqueued (`push`) so we can identify these entries when they are dequeued (`pop`). Hence, we must introduce additional modeling code that assigns a tag to each enqueued entry. We accomplish this unique identification by globally keeping track of push and pop occurrences. However, as demonstrated in Assertion 8-15, this requires modeling outside our property definition.

For our example, when an entry is enqueued, the current value of the modeling code push tag (`r_push_tag`) is assigned to the local variable (`lv_tag`) within a specific property thread of evaluation. In addition, the queue data entry value (`datain`) is sampled into a local variable (`lv_datain`), which is used for a future dequeue comparison. Once an entry is dequeued, and the value of the current modeling code pop tag (`r_pop_tag`) for the dequeue

operation matches the thread's local variable (`lv_tag`), then the queue output data (`dataout`) is checked against the expected value previously captured in the `lv_datain` local variable.

This simple example illustrates how additional modeling code is required in some circumstances to express a complex data integrity class of properties. Additional, and more complex modeling code would be required to express the same property if we expanded our example to include support for the queue clearing capabilities (that is, `clear[7:0]`), like our original queue example. This illustrates the point that assertions are not always the most efficient means of specifying and verifying data integrity properties in simulation. Alternative solutions involving verification components, such as a scoreboard, often provide a more effective approach to verifying data integrity properties. For an excellent overview of scoreboard verification component use, see [Glasser et al., 2007].

Explicit versus implicit queue

In this section, we demonstrated the process of creating assertion-based IP for an explicit queue. However, when verifying end-to-end behavior of a block with our property set, we must often describe an implicit queue behavior of the block—without knowing any specific details about the queue implementation. That is, the block might perform a specific function (not necessarily an explicit queue block), and from an end-to-end perspective the behavior possesses certain queuing properties (that is, an implicit queue). Properties describing a block's implicit queue behavior are generally intended to verify data integrity. Hence, assertions are not always the most efficient means of specifying and verifying these implicit queue behaviors for simulation.

8.2.5 Encapsulate properties

In this step, we turn our set of related queue assertions (and any coverage properties) into a reusable assertion-based monitor verification component, as demonstrated in

Example 8-6 (see page 214). We add analysis ports to our monitor for communication with other simulation verification components, as Chapter 3 “The Process” demonstrates.

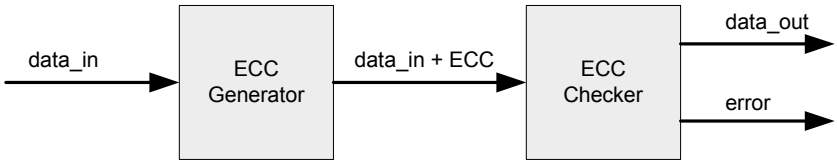
Example 8-6 Encapsulate properties inside a module

```
import avm_pkg::*;
import tb_tr_pkg::*; // tb_queue_status class definition

module tb_queue_mon (
  interface.monitor_mp monitor_mp
);

  avm_analysis_port #(tb_queue_status)
                        status_ap = new("status_ap", null);
  avm_analysis_port #(tb_coverage)
                        coverage_ap = new("coverage_ap", null);

  tb_queue_status status;
  . . .
 // Any required modeling code here (to model environment)
  . . .
 // All assertions and coverage properties here
  . . .
endmodule
```

8.3 Data error correction

Error correction is the process of detecting bit errors during data transfer—and we can correct these bits in either software or hardware. Figure 8-12 illustrates a high-level error correcting datapath flow that includes an error correcting code (ECC) generator and checker.

Figure 8-12 High-level error correcting datapath flow



For high data rates, error correction must be done in special-purpose hardware because software is too slow. The ECC bits are generally computed by an algorithm that is implemented as a set of exclusive OR trees in hardware. For our discussion, an ECC generator block computes the ECC. Each data bit contributes to more than one ECC bit. Hence, by a careful selection of data bits in the algorithm that directly contribute to the calculation for a specific ECC bit, it is not only possible to detect a single-bit error, but actually identify which bit is in error (including the ECC bits). In fact, the computed ECC is usually designed so that all single-bit errors are corrected, while all double-bit errors are detected (but not corrected).

For our discussion, an ECC checker recomputes the ECC from the transmitted data bits. The output of the recomputed ECC network is called a *syndrome*. If the syndrome is zero, no error occurred. If the syndrome is non-zero, it can be used to index into a look-up table[1] to determine exactly which bit is in error and then correct it. For a multi-bit error, no match occurs in the lookup table, and the error output bit is set in our high-level datapath flow illustrated in Figure 8-12.

ECC falls under the classification of data transformation, and in general, they pose difficulties for formal verification. However, Richards [2003] demonstrates a clever way of using assumptions and restrictions to obtain a proof across an ECC generator and checker block. Since our focus in creating assertion-based IP is to capture the intent of the design in a set of assertions suitable for simulation, and then only optimize the assertions for formal verification later if necessary, we have not limited our discussion to written assertions that are only amenable to formal verification. Therefore, in this section we demonstrate our assertion-based IP creation process on a single ECC checker block. You will note in this section that we treat the ECC checker as a black-box (that is, no detailed knowledge of the ECC checker implementation is required).

1. The lookup table could be implemented in hardware, firmware, or software.

8.3.1 Block diagram interface description

Figure 8-13 illustrates a block diagram for a simple ECC checker design component. For our example, all signal transitions relate only to the rising edge of the clock (`clk`). Table 8-5 provides a summary of the interface signals for our simple ECC checker design example.

Figure 8-13. ECC checker block

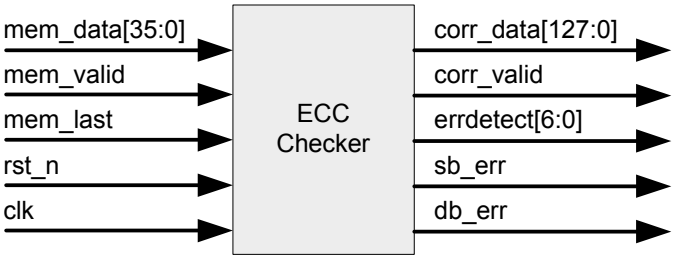


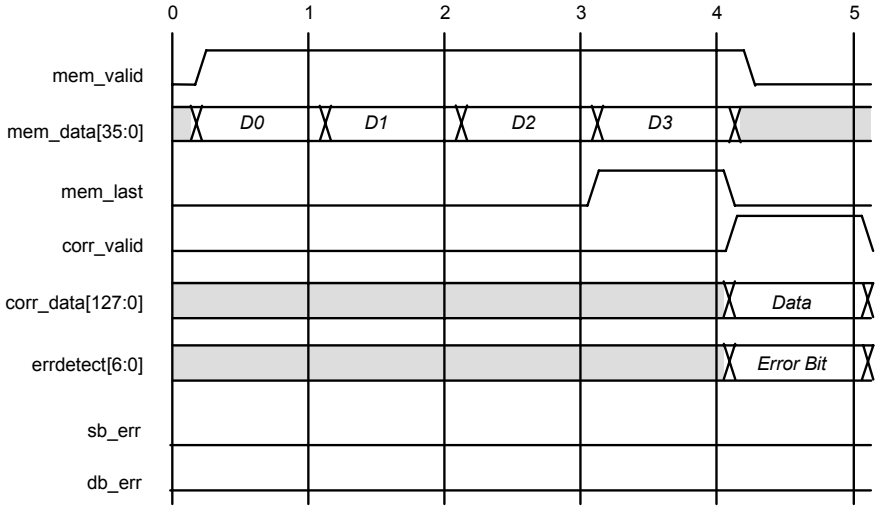
Table 8-5 Interface signals of the ECC checker block.

Name	Description
mem_data[35:0]	Memory data input beat
mem_valid	Active high memory data is valid
mem_last	Active high last (of 4) memory data beats
corr_data[127:0]	Corrected (possible) data from memory
corr_valid	Active high valid correct data
errdetect[6:0]	Bit error location identifier
sb_err	Active high single bit err
db_err	Active high double bit err
rst_n	Active low asynchronous reset
clk	Rising edge synchronous clock

8.3.2 Overview description

Our simple ECC checker is designed to accept a 128-bit input (that is, 128-bit data and 16-bit ECC), which is time multiplexed transferred over four beats of 36-bit slices. Figure 8-14 illustrates a four-beat time multiplexed transfer waveform for our simple ECC checker.

Figure 8-14 Four-beat multiplexed transfer to ECC checker



Our ECC checker accepts a 36-bit slice of input data from `mem_data` when the `mem_valid` signal is asserted. The last beat of the four-beat transfer occurs when `mem_last` input signal is asserted. The ECC checker will then reconstruct the 128-bit data word and 16-bit ECC on the last transfer from the four-beat, 36-bit slices. Then, the ECC checker will detect and correct any single-bit errors, or detect (and not correct) any double-bit errors.

When the data is ready for output, the ECC checker asserts the `corr_valid` output signal. If a single-bit error was detected, then the ECC checker asserts the `sb_err` output signal and identifies the appropriate failing bit on the `errdetect` bus. However, if a double-bit error occurred during the data transfer, the `db_err` output signal is asserted.

Our simple ECC checker has a requirement that the output `corr_data` and `errdetect` buses must retain their previous values (that is, remain stable) when `corr_valid` is inactive.

8.3.3 Natural language properties

Prior to creating a set of SystemVerilog assertions for our simple ECC checker, we must identify a comprehensive list of natural language properties, as shown in Table 8-6. Although the set of properties found in this table is not necessarily comprehensive, it is representative of a real set of properties and sufficient to demonstrate our process.

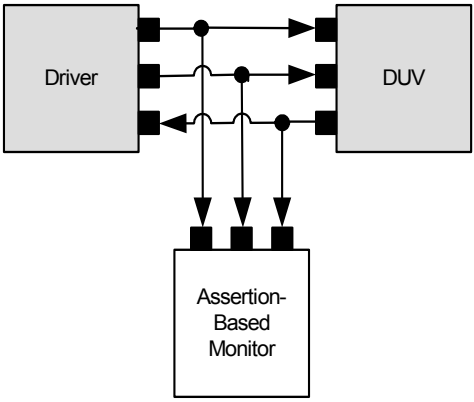
Table 8-6 Simple ECC checker properties

Assertion name	Summary
a_state_reset_inactive	The `corr_valid`, `sb_err`, and `db_err` are inactive after reset
a_legal_last_valid	The last indicator is only asserted every 4 valid input transfers
a_corr_data_stable	Corrected data does not change values after `corr_valid` is deasserted until next `corr_valid`
a_errdetect_stable	`errdetect` does not change after `corr_valid` is deasserted until next `corr_valid`.
a_data_correct	If not double-bit error, then the output data must not be corrupted

8.3.4 Assertions

To create a set of SystemVerilog assertions for our simple ECC checker, we begin defining the connection between our assertion-based monitor and other potential components within the testbench (such as a driver transactor and the DUV). We accomplish this goal by creating a SystemVerilog interface, as Figure 8-15 illustrates.

Figure 8-15 SystemVerilog interface



Example 8-7 demonstrates an interface for our example. For this case, our assertion-based monitor references the interface signals via the direction defined by the `monitor_mp` named modport.

Example 8-7 Encapsulate signals inside an interface

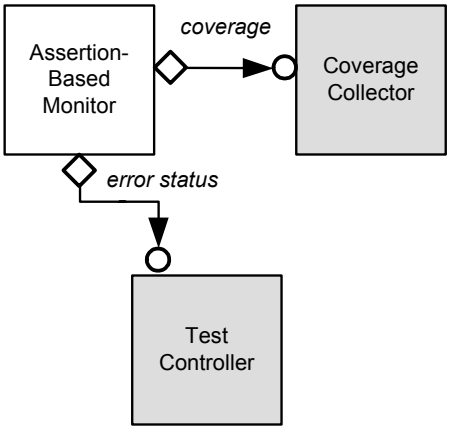
```
interface tb_ecc_if( input clk , input rst_n );

  parameter int BEAT_SIZE = 36;
  parameter int DATA_SIZE = 128;
  parameter int ERR_SIZE = 7;

  bit [BEAT_SIZE-1:0] mem_data;
  bit                 mem_valid;
  bit                 mem_last;
  bit [DATA_SIZE-1:0] corr_data;
  bit                 corr_valid;
  bit [ERR_SIZE-1:0]  errdetect;
  bit                 sb_err;
  bit                 db_err;

  modport ecc_gen_mp (
    . . . .
  );

  modport ecc_ck_mp (
    . . . .
  );

// Coninuted on next page
```

Example 8-7 Encapsulate signals inside an interface

```
  modport monitor_mp (
    input        clk , rst_n ,
    input        mem_data ,
    input        mem_valid
    input        mem_last ,
    input        corr_data ,
    input        corr_valid ,
    input        errdetect ,
    input        sb_err ,
    input        db_err
  );

endinterface
```

In addition to pin-level interfaces, we need a means for our ECC checker assertion-based monitor to communicate with various analysis verification components within the testbench. Hence, we introduce an error status analysis port within our monitor, as Figure 8-16 illustrates.

Figure 8-16 Error status and coverage analysis ports



In addition to assertion error status, coverage events are generally an important piece of analysis data that requires its own analysis port if your assertion monitor contains coverage detection.

Upon detecting an ECC checker error, the analysis port broadcasts the error condition (using an error status transaction object) to other verification components.

The *error status* transaction class is defined in Example 8-8, which is an extension of an `avm_transaction` base class. The `tb_ecc_status` class includes an `enum` that identifies the various types of ECC checker errors, and a set of methods to uniquely identify the specific error it detected.

Example 8-8 Error status class

```
class tb_ecc_status extends avm_transaction;

  typedef enum { ERR_STATE_RESET_INACTIVE,
                 ERR_LEGAL_LAST_VALID,
                 ERR_CORR_DATA_STABLE,
                 ERR_ERRDETECT_STABLE,
                 ERR_DATA_CORRECT } ecc_status_t;

  ecc_status_t   ecc_status;

  function void set_err_state_reset_inactive;
    ecc_status = ERR_STATE_RESET_INACTIVE ;
  endfunction

  function void set_err_legal_last_valid;
    ecc_status = ERR_LEGAL_LAST_VALID ;
  endfunction

  function void set_err_corr_data_stable;
    ecc_status = ERR_CORR_DATA_STABLE;
  endfunction

  function void set_err_errdetect_stable;
    ecc_status = ERR_ERRDETECT_STABLE;
  endfunction

  function void set_err_data_correct;
    ecc_status = ERR_DATA_CORRECT;
  endfunction

. . .

endclass
```

We are now ready to write assertions for our simple ECC checker properties defined in Table 8-6. Assertion 8-16, “ECC checker inactive after reset” demonstrates our first property.

Assertion 8-16 ECC checker inactive after reset

```
property p_state_reset_inactive;
  @(posedge monitor_mp.clk)
    $rose(monitor_mp.rst_n) |->
                (monitor_mp.db_err==1'b0) &
                (monitor_mp.sb_err==1'b0) &
                (monitor_mp.corr_valid==1'b0) &
                (monitor_mp.errdetect==7'b0) &
                (monitor_mp.corr_data[127:0]==128'b0);
endproperty
a_state_reset_inactive :
  assert property (p_state_reset_inactive) else begin
    status = new();
    status.set_err_state_reset_inactive();
    status_ap.write(status);
  end
```

Assertion 8-17, "ECC checker legal mem_last behavior" demonstrates our second property. We have partitioned this property into two assertions:

- Check the boundary condition after reset to ensure no spurious occurrence of `mem_last`
- Check normal operation to ensure proper sequencing of `mem_last`

Assertion 8-17 ECC checker legal mem_last behavior

```
// Forbid a mem_last after reset and before a four-beat transfer

property p_legal_last_valid_init;
  @(posedge monitor_mp.clk)
    $rose(monitor_mp.rst_n) |=>
          (~monitor_mp.mem_last thoughout monitor_mp.mem_valid[=3])
      ##1 (monitor_mp.mem_last & monitor_mp.mem_valid) ;
endproperty
a_legal_last_valid_init :
  assert property (p_legal_last_valid_init) else begin
    status = new();
    status.set_err_legal_last_valid();
    status_ap.write(status);
  end

// Continued on next page
```

Assertion 8-17 ECC checker legal mem_last behavior

```
// Forbid a mem_last between four-beat transfer

property p_legal_last_valid;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
    monitor_mp.mem_last |=>
          (~monitor_mp.mem_last thoughout monitor_mp.mem_valid[=3])
      ##1 (monitor_mp.mem_last & monitor_mp.mem_valid) ;
endproperty
a_legal_last_valid :
  assert property (p_legal_last_valid) else begin
    status = new();
    status.set_err_legal_last_valid();
    status_ap.write(status);
  end
```

Assertion 8-18, “ECC checker stability” demonstrates our next set of properties described in Table 8-6.

Assertion 8-18 ECC checker stability

```
property p_corr_data_stable;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
    !monitor_mp.corr_valid |-> $stable(monitor_mp.corr_data);
endproperty
a_corr_data_stable :
  assert property (p_corr_data_stable) else begin
    status = new();
    status.set_err_corr_data_stable();
    status_ap.write(status);
  end

property p_errdetect_stable;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
    !monitor_mp.corr_valid |-> $stable(monitor_mp.errdetect);
endproperty
a_errdetect_stable :
  assert property (p_errdetect_stable) else begin
    status = new();
    status.set_err_errdetect_stable();
    status_ap.write(status);
  end
```

Assertion 8-19, “ECC checker data integrity” demonstrates our final set of properties described in Table 8-6.

Assertion 8-19 ECC checker data integrity

```
property p_data_correct;
logic [143:0] lv_indata;
@(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
       ($rose(~monitor_mp.rst_n) | monitor_mp.mem_last)
   ##1 (monitor_mp.mem_valid[->1],
          lv_indata[143:108] = monitor_mp.mem_data)
   ##1 (monitor_mp.mem_valid[->1],
          lv_indata[107:72] = monitor_mp.mem_data)
   ##1 (monitor_mp.mem_valid[->1],
          lv_indata[71:36] = monitor_mp.mem_data)
   ##1 (monitor_mp.mem_valid[->1],
          lv_indata[35:00] = monitor_mp.mem_data)
   ##1 monitor_mp.corr_valid[->1] |->
 (monitor_mp.db_err | ( f_ecc(lv_indata) == monitor_mp.corr_data));
endproperty;
a_data_correct:
  assert property (p_data_correct) else begin
    status = new();
    status.set_err_data_correct();
    status_ap.write(status);
  end
```

To verify the data integrity for our ECC Checker, it becomes necessary to create a function (`f_ecc`) as part of our assertion (shown in Assertion 8-19). This function accepts a 144-bit input data value and outputs a 128-bit data value (with single bit correction if necessary). The draw back to this approach is that the function `f_ecc` replicates the algorithm (that is, code) we are checking. A better approach that is totally independent of any specific ECC algorithm would be to write a data integrity assertion across both the ECC generator and the ECC checker design components, as illustrated in Figure 8-12 (see page 214). However, the disadvantage of this approach is that the assertion-based IP would no longer be dedicated to checking a single design components (since two design components would be required).

8.3.5 Encapsulate properties

In this step, we turn our set of related ECC checker assertions into a reusable assertion-based monitor verification component, as demonstrated in Example 8-9 on page 225. We add analysis ports to our monitor for communication with other simulation verification components, as Chapter 3 “The Process” demonstrates.

Example 8-9 Encapsulate properties inside a module

```
import avm_pkg::*;
import tb_tr_pkg::*; // tb_ecc_status class definition

module tb_ecc_checker_mon (
  interface.monitor_mp monitor_mp
);

  avm_analysis_port #(tb_ecc_status)
                          status_ap = new("status_ap", null);
  avm_analysis_port #(tb_ecc_coverage)
                          coverage_ap = new("coverage_ap", null);

  tb_ecc_status status;
  . . .
 // Any required modeling code here (to model environment)
  . . .
 // All assertions and coverage properties here
  . . .
endmodule
```

8.4 Data compression

Data compression, an important application in the areas of data transmission and data storage, is the process of encoding information using fewer bits. Compressing data that is to be stored or transmitted reduces storage and communication costs. In fact, when the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel. Similarly, when a file is compressed to half of its original size, the effect is equivalent to doubling the capacity of the storage medium.

In this section, we introduce a simple packet data compressor design component to demonstrate our assertion-based IP creation process. Our simple packet data compressor provides another example of a data transformation datapath component.

8.4.1 Block diagram interface description

Figure 8-17 illustrates a block diagram for a simple packet data compressor design. For our example, all signal transitions relate only to the rising edge of the clock (clk). Table 8-7 provides a summary of the interface signals for our simple packet data compressor design example.

Figure 8-17 A simple packet data compressor block diagram

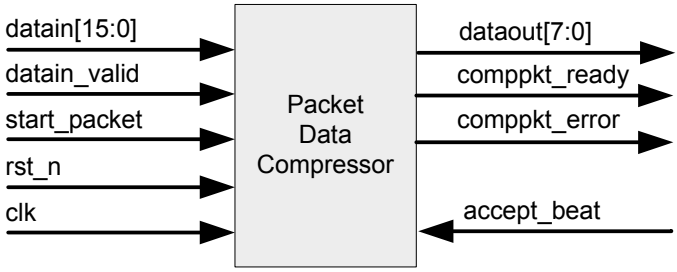


Table 8-7 Interface description for packet data compressor

Name	Description
datain[15:0]	Input data to the compressor
datain_valid	Enter input data into compressor
start_packet	First entry of a set to be compressed
comppkt_ready	Compressed packet is ready to be read out
comppkt_error	Error signal asserted when an uncompressed packet is not fully read out before another uncompressed packet is ready to be transmitted The comppkt_error signal will be asserted coincident with a subsequent comppkt_ready
dataout[7:0]	Data sent out of compressor
accept_beat	Output data has been accepted

Name	Description
clk	Synchronous clock
rst_n	Asynchronous reset

8.4.2 Overview description

Our simple data compressor accepts a packet of eight 16-bit data beats and sends out the packet compressed into four 8-bit data beats. The compression operates on a 128-bit packet taking advantage of the limited ranges of the data fields within the packet. The details of the encoding algorithm are not important for our discussion and are not presented in this section. There are flow control signals that indicate when data is ready to be sent and when it has been picked up. In addition, the `start_packet` signal indicates the beginning transmission of an input packet that is to be compressed

Basic packet data compression operation

The waveforms in Figure 8-18 and Figure 8-19 illustrate the typical operation for our simple packet data compressor.

An input packet is composed of eight beats of data, where each beat (that is, 16-bit data transfer) is identified by asserting datain_valid. Once eight beats of data have been received, the compression operation occurs on the entire set, and the data is ready to be read after a minimum and maximum latency of two clock for our simple example. The signal comppkt_ready is then asserted to indicate that a compressed data packet is ready. The four compressed data packet beats are removed a single beat at a time when the accept_beat signal is asserted.

Figure 8-18 Packet data compression input operation

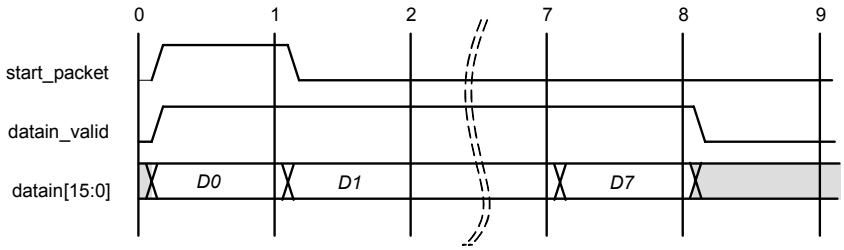
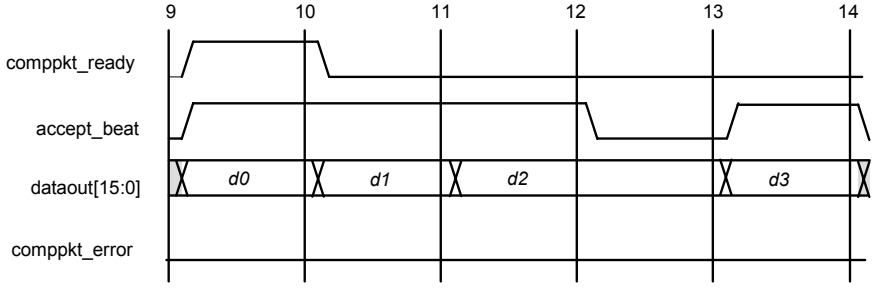


Figure 8-19 Packet data compression output operation



Error transfer operation

Our simple packet data compressor supports slow output devices by allowing a previously compressed data packet to be dropped when a complete incoming packet has arrived prior to completing the transmission of a previously compressed packet. When this error condition occurs, the signal `comppkt_error` is asserted for a single cycle, which indicates that the receiving device should flush all partially received packet beats.

8.4.3 Natural language properties

Prior to creating a set of SystemVerilog assertions for our simple packet data compressor design, we must identify a comprehensive list of natural language properties, as shown

in Table 8-8. Although the set of properties found in this table is not necessarily comprehensive, it is representative of a real set of properties and sufficient to demonstrate our process.

Table 8-8 Simple packet data compressor design properties

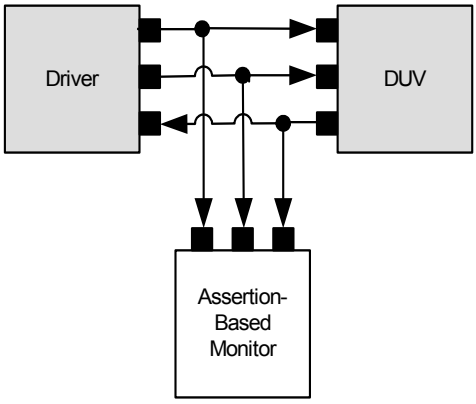
Assertion name	Summary
a_reset_state	Initial state after reset is no compressed packets
a_8_input_pkts_after_start	Eight and only eight uncompressed packets are received only after a `start_packet` No new start_packet occurs until all input packets received
a_4_output_pkts_after_ready	Four and only four compressed packets are sent only after a `comppkt_ready` is asserted No new `comppkt_ready` are asserted until all output packets are sent, or a `comppkt_error` is asserted
a_input_to_output_or_error	After input packet of eight beats of uncompressed data is received, then either a compressed output packet will eventually be sent or a `comppkt_error` occurs
a_valid_pkt_error	The signal `comppkt_error` shall be asserted only when `comppkt_ready` rises (signaling new transaction)
a_valid_pkt_error_pulse	The signal `comppkt_error` shall be a one-cycle pulse

Assertion name	Summary
a_valid_fast_packet	A subsequent compressed packet that is ready prior to completing the transmission of an existing compressed packet will force an abort on the currently transmitted compressed compacket, and comppkt_error will then be asserted
a_dataout_stable	The `dataout` bus must be stable when `accept_beat` is not asserted
a_compressed_data_integrity	The data from the input packet, once compressed, should be the same value as the output packet.

8.4.4 Assertions

To create a set of SystemVerilog assertions for our simple packet data compressor design, we begin defining the connection between our assertion-based monitor and other potential components within the testbench (such as a driver transactor and the DUV). We accomplish this goal by creating a SystemVerilog interface, as Figure 8-20 illustrates.

Figure 8-20 SystemVerilog interface



Example 8-10 Encapsulate signals inside an interface

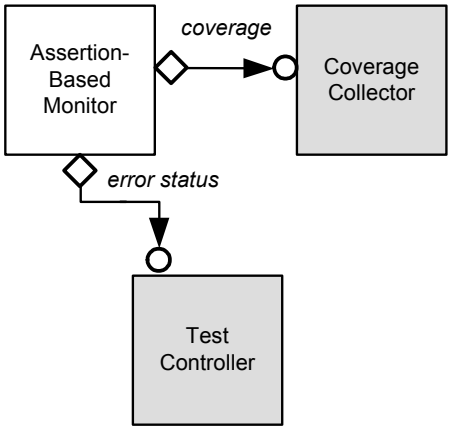
```
interface tb_data_compressor_if( input clk , input rst_n );
  bit [15:0] datain;
  bit        datain_valid;
  bit        start_packet;
  bit        comppkt_ready;
  bit        comppkt_error;
  bit [7:0]  dataout;
  bit        accept_beat;

  modport driver_mp (
    input        clk , rst ,
    . . . .
  );

  . . .

  modport duv_mp (
    input        clk , rst ,
    . . . .
  );

modport monitor_mp (
    input        clk , rst ,
    input        datain,
    input        datain_valid,
    input        start_packet,
    input        comppkt_ready,
    input        comppkt_error,
    input        dataout,
    input        accept_beat
  );
endinterface
```

Example 8-10 on page 231 demonstrated an interface for our simple packet data compressor example. For this case, our assertion-based monitor would reference the interface signals via the direction defined by the `monitor_mp` named modport.

In addition to pin-level interfaces, we need a means for our simple packet data compressor assertion-based monitor to communicate with various analysis verification components within the testbench. Hence, we introduce an error status analysis port within our monitor, as Figure 8-21 illustrates.

In addition to assertion error status, coverage events are an important piece of analysis data that requires its own analysis port. We discuss coverage with respect to creating assertion-based IP separately in Section 5.4.

Figure 8-21 Error status and coverage analysis ports



Upon detecting a data compression error, the analysis port broadcasts the error condition (using an error status transaction object) to other verification components.

Example 8-11 Error status class

```
class tb_cmp_status extends avm_transaction;

  typedef enum { ERR_RESET_STATE,
                 ERR_8_INPUT_PKTS_AFTER_START,
                 ERR_4_OUTPUT_PKTS_AFTER_READY,
                 ERR_INPUT_TO_OUTPUT_OR_ERROR,
                 ERR_VALID_PKT_ERROR,
                 ERR_VALID_PKT_ERROR_PULSE,
                 ERR_VALID_FAST_PACKET,
                 ERR_DATAIN_STABLE
} cmp_status_t;

  cmp_status_t   cmp_status;
  int err_client_num;

  function void set_err_reset_state;
    cmp_status = ERR_RESET_STATE ;
  endfunction

  function void set_err_8_input_pkts_after_start;
    cmp_status = ERR_8_INPUT_PKTS_AFTER_START ;
  endfunction

// Continued on next page
```

Example 8-11 Error status class

```
  function void set_err_4_output_pkts_after_ready;
    cmp_status = ERR_4_OUTPUT_PKTS_AFTER_READY ;
  endfunction

  function void set_err_input_to_output_or_error;
    cmp_status = ERR_INPUT_TO_OUTPUT_OR_ERROR ;
  endfunction

  function void set_err_valid_pkt_error;
    cmp_status = ERR_VALID_PKT_ERROR ;
  endfunction
  function void set_err_valid_pkt_error_pulse;
    cmp_status = ERR_VALID_PKT_ERROR_PULSE ;
  endfunction

  function void set_err_valid_fast_packet;
    cmp_status = ERR_VALID_FAST_PACKET ;
  endfunction

  function void set_err_datain_stable;
    cmp_status = ERR_DATAIN_STABLE ;
  endfunction
. . .
endclass
```

The *error status* transaction class is defined in Example 8-11, which is an extension of avm_transaction base class. The `tb_cmp_status` class includes an `enum` that identifies the various types of data compressor errors and a set of methods to uniquely identify the specific error it detected.

We are now ready to write assertions for our simple packet data compressor design properties defined in Table 8-8 (see page 229).

Assertion 8-20, “Packet data compressor inactive after reset” on page 234 demonstrates our first property.

Assertion 8-20 Packet data compressor inactive after reset

```
property p_reset_state;
  @(posedge monitor_mp.clk)
    $rose(monitor_mp.rst_n) |->
                    monitor_mp.comppkt_ready==1'b0) &
                    (monitor_mp.comppkt_error==1'b0) &
                    (monitor_mp.dataout==8'b0);
endproperty
a_reset_state :
  assert property (p_reset_state) else begin
    status = new();
    status.set_err_reset_state();
    status_ap.write(status);
  end
```

Assertion 8-21, "Valid uncompressed input packet sequence" demonstrates our next property. Note that we wrote a separate assertion to handle the boundary case after reset. However, we chose to use the same error status method as the normal case to identify the violation.

Assertion 8-21 Valid uncompressed input packet sequence

```
// Boundary case after reset
property p_8_input_pkts_after_start_init;
  @(posedge monitor_mp.clk)
    $rose(monitor_mp.rst_n) |->
          (!monitor_mp.datain_valid throughout
                                    monitor_mp.start_packet[->1]);
endproperty

a_8_input_pkts_after_start_init :
  assert property (p_8_input_pkts_after_start_init) else begin
    status = new();
    status.set_err_8_input_pkts_after_start();
    status_ap.write(status);
  end

// Normal case
property p_8_input_pkts_after_start;
  @(posedge monitor_mp.clk)
    monitor_mp.start_packet |->
          monitor_mp.data_valid
      ##1 (~monitor_mp.start_packet throughout
                                   monitor_mp.datain_valid[=7])
      ##1 (!monitor_mp.datain_valid & monitor_mp.start_packet);
endproperty

// Continued on next page
```

Assertion 8-21 Valid uncompressed input packet sequence

```
a_8_input_pkts_after_start :
  assert property (p_8_input_pkts_after_start) else begin
    status = new();
    status.set_err_8_input_pkts_after_start();
    status_ap.write(status);
  end
```

Assertion 8-22, “Valid compressed output packet sequence” demonstrates our next property.

Assertion 8-22 Valid compressed output packet sequence

```
// Boundary case after reset
property p_4_output_pkts_after_ready_init;
  @(posedge monitor_mp.clk)
   $rose(monitor_mp.rst_n) |->
      !(amonitor_mp.accept_beat | monitor_mp.comppkt_error)
                          throughout monitor_mp.comppkt_ready[->1];
endproperty
a_4_output_pkts_after_ready_init :
  assert property (p_4_output_pkts_after_ready_init) else begin
    status = new();
    status.set_err_4_output_pkts_after_ready();
    status_ap.write(status);
  end

// Normal case
property p_4_output_pkts_after_ready;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
   $rose(monitor_mp.comppkt_ready) |->
      ( (monitor_mp.comppkt_ready throughout
                                  monitor_mp.accept_beat[->1])
    ##1 (!monitor_mp.comppkt_ready throughtout
                                  monitor_mp.accept_beat[=3])
    ##1 (!monitor_mp.accept_beat &
                                  monitor_mp.comppkt_ready) )
     or (monitor_mp.comppkt_error[->1] intersect
                                  monitor_mp.accept_beat[=0:4]);
endproperty
a_4_output_pkts_after_ready :
  assert property (p_4_output_pkts_after_ready) else begin
    status = new();
    status.set_err_4_output_pkts_after_ready();
    status_ap.write(status);
  end
```

Note that we wrote a separate assertion to handle the boundary case after reset. However, we chose to use the

same error status method as the normal case to identify the violation. For the normal case, the assertion checks that once `comppkt_ready` is asserted, it remains asserted until either the first occurrence of `accept_beat` or when `comppkt_error` is asserted. In addition, this assertion checks that four and only four output packets are transferred (indicated by an active `accept_beat`), unless an error occurs (indicated by an active `comppkt_error`).

Assertion 8-23, "Valid input to output control behavior" demonstrates our next property. This assertion accounts for the minimum and maximum latency of two clocks after the eight uncompressed input beats have been received and compressed output beats are ready.

Assertion 8-23 Valid input to output control behavior

```
property p_input_to_output_or_error;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
   monitor_mp.start_packet ##0 monitor_mp.datain_valid[->8] |->
                                      ##2 monitor_mp.comppkt_ready;
endproperty
a_input_to_output_or_error :
  assert property (p_input_to_output_or_error) else begin
    status = new();
    status.set_err_input_to_output_or_error();
    status_ap.write(status);
  end
```

Assertion 8-24, "Valid packet error behavior" demonstrates our next property.

Assertion 8-24 Valid packet error behavior

```
property p_valid_pkt_error;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
   monitor_mp.comppkt_error |-> $rose(monitor_mp.comppkt_ready);
endproperty
a_valid_pkt_error :
  assert property (p_valid_pkt_error) else begin
    status = new();
    status.set_err_valid_pkt_error();
    status_ap.write(status);
  end
```

Assertion 8-25, “Valid packet error pulse” demonstrates our next property.

Assertion 8-25 Valid packet error pulse

```
property p_valid_pkt_error_pulse;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
   monitor_mp.comppkt_error |=> !monitor_mp.comppkt_error;
endproperty
a_valid_pkt_error_pulse :
  assert property (p_valid_pkt_error_pulse) else begin
    status = new();
    status.set_err_valid_pkt_error_pulse();
    status_ap.write(status);
  end
```

Assertion 8-26, “Valid fast packet behavior” demonstrates our next property.

Assertion 8-26 Valid fast packet behavior

```
property p_valid_fast_packet;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
     ((monitor_mp.comppkt_ready[->2] intersect
                                    monitor_mp.accept_beat[=0:3])
  or (monitor_mp.comppkt_ready[->2] intersect
                                    monitor_mp.accept_beat[->4]))
 |->  monitor_mp.comppkt_error;
endproperty
a_valid_fast_packet :
  assert property (p_valid_fast_packet) else begin
    status = new();
    status.set_err_valid_fast_packet();
    status_ap.write(status);
  end
```

Assertion 8-27, “Packet data compressor stable output data” demonstrates our next property.

Assertion 8-27 Packet data compressor stable output data

```
property p_dataout_stable;
  @(posedge monitor_mp.clk)
    !monitor_mp.accept_beat |-> $stable(monitor_mp.dataout))
endproperty
a_dataout_stable :
  assert property (p_dataout_stable) else begin
    status = new();
    status.set_err_dataout_stable();
    status_ap.write(status);
  end
```

Data integrity property

Our final property (`a_compressed_data_integrity`) involves data compression integrity. This is an example of a property that is generally too difficult to express using SVA constructs alone. We could introduce additional modeling code that assembles the eight beats of uncompressed data. Then, our modeling code would implement the compression algorithm to create a compressed output packet for comparison. We would then write a set of assertions that compare the four beats of compressed data read out of the compressor to the values calculated by the modeling code. This illustrates the point that assertions are not always the most efficient means of specifying and verifying data integrity properties in simulation. Alternative solutions involving verification components, such as a scoreboard, often provide a more effective approach to verifying data integrity properties. For an excellent overview of scoreboard verification component use, see [Glasser et al., 2007].

8.4.5 Encapsulate properties

In this step, we turn our set of related packet data compressor assertions (and any coverage properties) into a reusable assertion-based monitor verification component. We add analysis ports to our monitor for communication

with other simulation verification components as Chapter 3 “The Process” demonstrates.

Example 8-12 Encapsulate properties inside a module

```
import avm_pkg::*;
import tb_tr_pkg::*; // tb_cmp_status class definition

module tb_data_compressor_mon (
  interface.monitor_mp monitor_mp
);

  avm_analysis_port #(tb_cmp_status)
                        status_ap = new(“status_ap”, null);
  avm_analysis_port #(tb_cmp_coverage)
                        coverage_ap = new(“coverage_ap”, null);

  tb_cmp_status status;
  . . .
 // Any required modeling code here (to model environment)
  . . .
 // All assertions and coverage properties here
  . . .
endmodule
```

8.5 Data decompression

Data decompression is the inverse application of data compression. Compressed input data is expanded (decompressed) and restored to its original form. Data being expanded may be either a single word or a group of incoming words within a packet. Similarly, the expanded data may be sent out as a single word or as multiple words within a packet.

In this section, we introduce a simple packet data decompressor design component to demonstrate our assertion-based IP creation process.

8.5.1 Block diagram interface description

Figure 8-22 illustrates a block diagram for a simple packet data decompressor design. For our example, all signal transitions relate only to the rising edge of the clock (clk). Table 8-9 provides a summary of the interface signals for our simple packet data decompressor design example.

Figure 8-22 A simple packet data decompressor block diagram

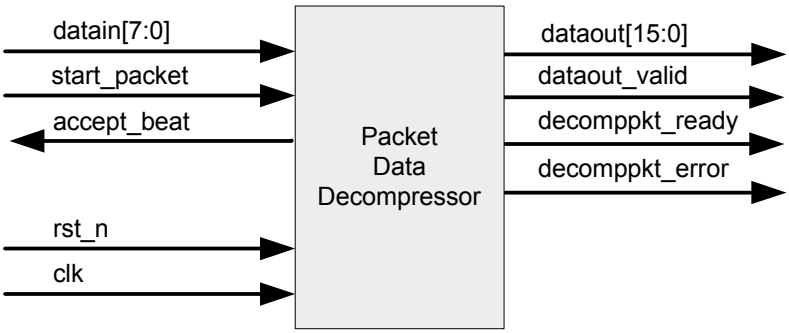


Table 8-9 Interface description for packet data decompressor

Name	Description
start_packet	First packet beat of compressed data
datain[7:0]	Compressed input data
accept_beat	Input packet data beat has been accepted
decomppkt_ready	Decompressed packet is ready for input
decomppkt_error	Error signal asserted when a packet is not fully received or is not fully read out before another packet is started
dataout_valid	Output data valid from decompressor
dataout[15:0]	Expanded data out
clk	Synchronous clock
rst_n	Asynchronous reset

8.5.2 Overview description

Our simple data decompressor accepts a packet of four eight-bit compressed data beats and sends out the packet

uncompressed into eight 16-bit data beats. The details of the decoding algorithm are not important for our discussion and are not presented in this section. There are flow control signals that indicate when data is ready to be sent and when it has been received. In addition, the `comppkt_ready` signal indicates the beginning transmission of an input packet, which is to be uncompressed.

Basic packet data decompression operation

The waveforms in Figure 8-23 and Figure 8-24 illustrate the typical operation for our simple packet data decompressor.

Figure 8-23 Packet data decompressor input operation

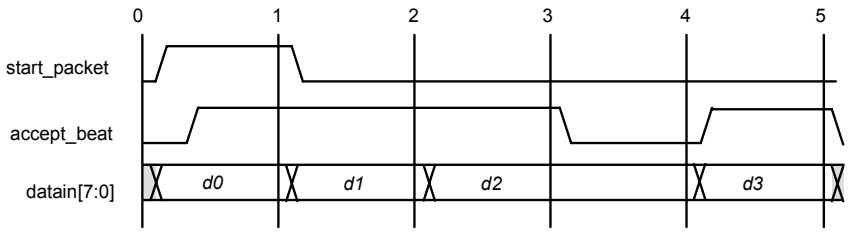
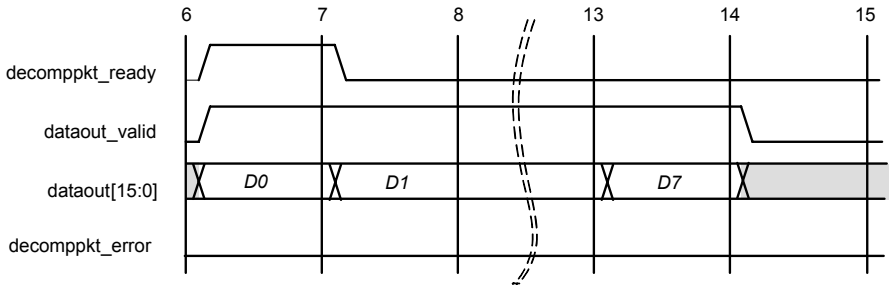


Figure 8-24 Packet data decompressor output operation



Once the four beats of compressed and data have been received, the packet is expanded and is then sent out as a packet of eight beats of uncompressed data. For our simple example, we assume a minimum and maximum latency of two clocks between the time the last beat of a compressed packet is received and the first beat of an expanded packet is transmitted.

Error transfer operation

Our simple packet data decompressor supports fast input devices by allowing incoming compressed data packets to be dropped when the `comppkt_error` signal is asserted by the fast input device. Our simple packet data decompressor begins a new uncompressed operation on the incoming packet. When this error condition occurs, the signal `comppkt_error` is asserted for a single cycle, which indicates that the receiving device should flush all partially received packet beats.

8.5.3 Natural language properties

Prior to creating a set of SystemVerilog assertions for our simple packet data decompressor design, we must identify a comprehensive list of natural language properties, as shown in Table 8-10. Although the set of properties found in this table is not necessarily comprehensive, it is representative of a real set of properties and sufficient to demonstrate our process.

Table 8-10 Simple packet data decompressor design properties

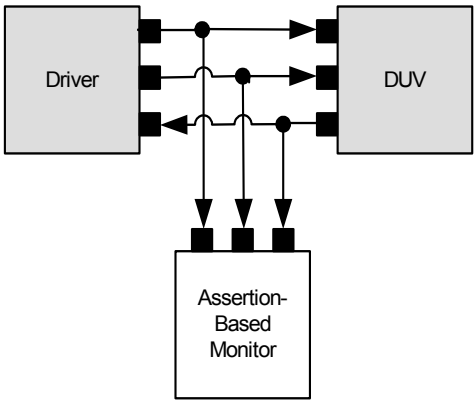
Assertion name	Summary
a_reset_state	Output controls after reset are inactive
a_8_output_pkts_after_start	Eight and only eight uncompressed packets are transmitted only after a `start_packet` No new start_packet(s) are asserted until all input packets are received, or a de`comppkt_error` is asserted

Assertion name	Summary
a_4_input_pkts_after_ready	Four and only four compressed packets are received only after a start_packet `is asserted` No new start_packet(s) are asserted until all input packets are received, or a de`comppkt_error` is asserted
a_datain_stable	The `datain` bus must be stable until `accept_beat` is asserted
a_in_to_out_or_err	After a `start_packet`, if there is no error while receiving four compressed packet beats, then `decomppkt_ready` must be asserted two cycles after last beat
a_uncompressed_data_integrity	The data from the input packet, once uncompressed, should be the same value as the output packet

8.5.4 Assertions

To create a set of SystemVerilog assertions for our simple packet data decompressor design, we begin defining the connection between our assertion-based monitor and other potential components within the testbench (such as a driver transactor and the DUV). We accomplish this goal by creating a SystemVerilog interface, as Figure 8-25 illustrates.

Figure 8-25 SystemVerilog interface



Example 8-13 demonstrated an interface for our simple packet data decompressor example. For this case, our assertion-based monitor would reference the interface signals via the direction defined by the `monitor_mp` named modport.

Example 8-13 Encapsulate signals inside an interface

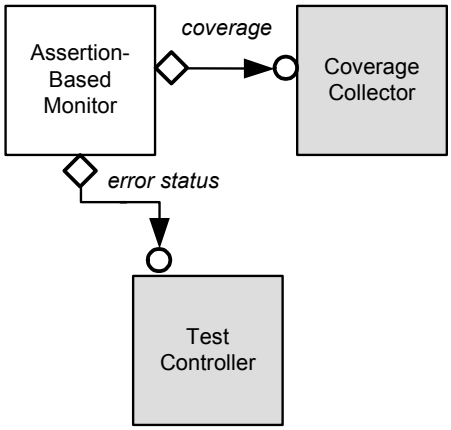
```
interface tb_data_decompressor_if( input clk , input rst_n );

  bit        start_packet;
  bit [7:0]  datain;
  bit        accept_beat;

  bit        decomppkt_error;
  bit        decomppkt_ready;
  bit [15:0] dataout;
  bit        dataout_valid;

modport driver_mp (
    input        clk, rst,
    . . . .
  );

  . . .

 modport duv_mp (
    input        clk, rst,
    . . . .
  );

// Continued on next page
```

Example 8-13 Encapsulate signals inside an interface

```
  modport monitor_mp (
    input        clk, rst,
    input        datain,
    input        start_packet,
    input        accept_beat,
    input        dataout_valid,
    input        decomppkt_ready,
    input        decomppkt_error,
    input        dataout
  );

endinterface
```

In addition to pin-level interfaces, we need a means for our simple packet data decompressor assertion-based monitor to communicate with various analysis verification components within the testbench. Hence, we introduce an error status analysis port within our monitor, as Figure 8-26 illustrates.

Figure 8-26 Error status and coverage analysis ports



In addition to assertion error status, coverage events are an important piece of analysis data that requires its own analysis port. We discuss coverage with respect to creating assertion-based IP separately in Section 5.4.

Upon detecting a data decompression error, the analysis port broadcasts the error condition (using an error status transaction object) to other verification components.

The *error status* transaction class is defined in Example 8-14, which is an extension of avm_transaction base class. The `tb_decmp_status` class includes an `enum` that identifies the various types of data decompressor errors and a set of methods to uniquely identify the specific error it detected.

Example 8-14 Error status class

```
class tb_decmp_status extends avm_transaction;

  typedef enum { ERR_RESET_STATE,
                 ERR_8_OUTPUT_PKTS_AFTER_START,
                 ERR_4_INPUT_PKTS_AFTER_READY,
                 ERR_DATAIN_STABLE,
                 ERR_IN_TO_OUT_OR_ERR,
                 ERR_UNCOMPRESSED_DATA_INTEGRITY
} decmp_status_t;

  decmp_status_t   decmp_status;
  int err_client_num;

  function void set_err_reset_state;
    decmp_status = ERR_RESET_STATE ;
  endfunction

  function void set_err_8_output_pkts_after_start;
    decmp_status = ERR_8_OUTPUT_PKTS_AFTER_START ;
  endfunction

  function void set_err_4_input_pkts_after_ready;
    decmp_status = ERR_4_INPUT_PKTS_AFTER_READY ;
  endfunction

  function void set_err_datain_stable;
    decmp_status = ERR_DATAIN_STABLE ;
  endfunction

  function void set_err_in_to_out_or_err;
    decmp_status = ERR_IN_TO_OUT_OR_ERR ;
  endfunction

  function void set_err_uncompressed_data_integrity;
    decmp_status = ERR_UNCOMPRESSED_DATA_INTEGRITY ;
  endfunction

. . .

endclass
```

We are now ready to write assertions for our simple packet data decompressor design properties defined in Table 8-10 (see page 242). Assertion 8-28, “Packet data decompressor inactive after reset” demonstrates our first property.

Assertion 8-28 Packet data decompressor inactive after reset

```
property p_reset_state;
  @(posedge monitor_mp.clk)
    $rose(monitor_mp.rst_n) |->
         (monitor_mp.accept_beat==1'b0) &
         (monitor_mp.decomppkt_error==1'b0) &
         (monitor_mp.decomppkt_ready==1'b0) &
         (monitor_mp.dataout==16'b0) &
         (monitor_mp.dataout_valid==1'b0);

endproperty
a_reset_state :
  assert property (p_reset_state) else begin
    status = new();
    status.set_err_reset_state();
    status_ap.write(status);
  end
```

Assertion 8-29, "Valid compressed input packet sequence" demonstrates our next property.

Assertion 8-29 Valid compressed input packet sequence

```
// Boundary case after reset
property p_4_input_pkts_after_ready_init;
 @(posedge monitor_mp.clk)
  $rose(monitor_mp.rst_n) |->
   !(monitor_mp.accept_decomppkt | monitor_mp.comppkt_error)
                            throughout monitor_mp.start_packet[->1];
endproperty
a_4_input_pkts_after_ready_init :
  assert property (p_4_input_pkts_after_ready_init) else begin
    status = new();
    status.set_err_4_input_pkts_after_ready();
    status_ap.write(status);
  end

// Normal case
property p_4_input_pkts_after_ready;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
   $rose(start_packet) |->
      ( (monitor_mp.start_packet throughout
                                  monitor_mp.accept_beat[->1])
    ##1 (!monitor_mp.start_packet thoughtout
                                  monitor_mp.accept_beat[=3])
    ##1 (!monitor_mp.accept_beat & monitor_mp.decomppkt_ready) )
     or (monitor_mp.comppkt_error[->1] intersect
                                  monitor_mp.accept_beat[=0:4]);
endproperty

// Continued on next page
```

Assertion 8-29 Valid compressed input packet sequence

```
a_4_input_pkts_after_ready :
  assert property (p_4_input_pkts_after_ready) else begin
    status = new();
    status.set_err_4_input_pkts_after_ready();
    status_ap.write(status);
  end
```

Note in Assertion 8-29 that we wrote a separate assertion to handle the boundary case after reset. However, we chose to use the same error status method as the normal case to identify the violation. For the normal case, the assertion checks that once `start_packet` is asserted, it remains asserted until either the first occurrence of `accept_beat` or when `comppkt_error` is asserted. In addition, this assertion checks that four and only four input packets are transferred (indicated by an active `accept_beat`), unless an error occurs (indicated by an active `comppkt_error`).

Assertion 8-30, "Valid uncompressed output packet sequence" demonstrates our next property. Note that we wrote a separate assertion to handle the boundary case after reset. However, we chose to use the same error status method as the normal case to identify the violation.

Assertion 8-30 Valid uncompressed output packet sequence

```
// Boundary case after reset
property p_8_output_pkts_after_start_init;
  @(posedge monitor_mp.clk)
    $rose(monitor_mp.rst_n) |->
                (!monitor_mp.datain_valid throughout
                              monitor_mp.decomppkt_ready[->1]);
endproperty
a_8_output_pkts_after_start_init :
  assert property (p_8_output_pkts_after_start_init) else begin
    status = new();
    status.set_err_8_output_pkts_after_start();
    status_ap.write(status);
  end

// Continued on next page
```

Assertion 8-30 Valid uncompressed output packet sequence

```
// Normal case
property p_8_output_pkts_after_start;
  @(posedge monitor_mp.clk)
    monitor_mp.decomppkt_ready |->
      ( (monitor_mp.dataout_valid)
    ##1 (~monitor_mp.decomppkt throughout
                                monitor_mp.dataout_valid[=7])
    ##1 (!monitor_mp.dataout_valid & monitor_mp.decomppkt_ready) );
endproperty
a_8_output_pkts_after_start :
  assert property (p_8_output_pkts_after_start) else begin
    status = new();
    status.set_err_8_output_pkts_after_start();
    status_ap.write(status);
  end
```

Assertion 8-31, “Packet data decompressor stable input data” demonstrates our next property.

Assertion 8-31 Packet data decompressor stable input data

```
property p_datain_stable;
  @(posedge monitor_mp.clk)
    !monitor_mp.accept_beat |-> $stable(monitor_mp.datain))
endproperty
a_datain_stable :
  assert property (p_datain_stable) else begin
    status = new();
    status.set_err_datain_stable();
    status_ap.write(status);
  end
```

Assertion 8-32, “Valid input to output control behavior” demonstrates our next property. This assertion accounts for the minimum and maximum latency of two clocks after the four compressed input beats have been received and decompressed output beats are ready.

Assertion 8-32 Valid input to output control behavior

```
property p_in_to_out_or_err;
  @(posedge monitor_mp.clk) disable iff (~monitor_mp.rst_n)
        (monitor_mp.start_packet)
    ##0 ( !monitor_mp.decomppkt_error throughout
                            (monitor_mp.accept_beat[->4]) ) |->
    ##2 (monitor_mp.decomppkt_ready);
endproperty
a_in_to_out_or_err :
  assert property (p_in_to_out_or_err) else begin
    status = new();
    status.set_err_in_to_out_or_err();
    status_ap.write(status);
  end
```

Data integrity property

Our final property (`a_expanded_data_integrity`) involves data decompression integrity. This is an example of a property that is generally too difficult to express using SVA constructs alone. We could introduce additional modeling code that assembles the four beats of compressed data. Then, our modeling code would implement the decompression algorithm to create an uncompressed output packet for comparison. We would then write a set of assertions that compare the eight beats of uncompressed data read out of the decompressor to the values calculated by the modeling code. This illustrates the point that assertions are not always the most efficient means of specifying and verifying data integrity properties in simulation. Alternative solutions involving verification components, such as a scoreboard, often provide a more effective approach to verifying data integrity properties. For an excellent overview of scoreboard verification component use, see [Glasser et al., 2007].

8.5.5 Encapsulate properties

In this step, we turn our set of related simple packet data decompressor assertions (and any coverage properties) into a reusable assertion-based monitor verification component. We add analysis ports to our monitor for communication with other simulation verification components as Chapter 3 "The Process" demonstrates.

Example 8-15 Encapsulate properties inside a module

```
import avm_pkg::*;
import tb_tr_pkg::*; // tb_decmp_status class definition

module tb_data_decompressor_mon (
  interface.monitor_mp monitor_mp
);

  avm_analysis_port #(tb_decmp_status)
                          status_ap = new("status_ap", null);
  avm_analysis_port #(tb_decmp_coverage)
                          coverage_ap = new("coverage_ap", null);
  tb_decmp_status status;
  . . .
 // Any required modeling code here (to model environment)
  . . .
 // All assertions and coverage properties here
  . . .
endmodule
```

8.6 Summary

In this chapter, our focus was on demonstrating our assertion-based IP creation process on various datapath components. One message we hope you take away from this chapter is that not all behaviors are easily expressed using temporal constructs from an assertion language alone. In fact, in this chapter we demonstrated the increasing difficulty of writing data integrity properties as the sections progress. While in Section 8.1, "Multiport register file," after a fair amount of work, we were able to create a set of data integrity assertions by taking advantage of SVA local variables, in Section 8.2, "Data queue" it became necessary

to add modeling code to simplify coding our data integrity assertions. The design component examples in Section 8.4, “Data compression” and Section 8.5, “Data decompression” are actually easier to check in simulation using a scoreboard than trying to write a set of complex data integrity assertions.

A key point in this section is that, when creating verification IP, choose the form that is easiest to express. Do not be a purist! You do not have to capture all behaviors of the design using SVA constructs alone. Indeed, doing so is often not possible.