CHAPTER

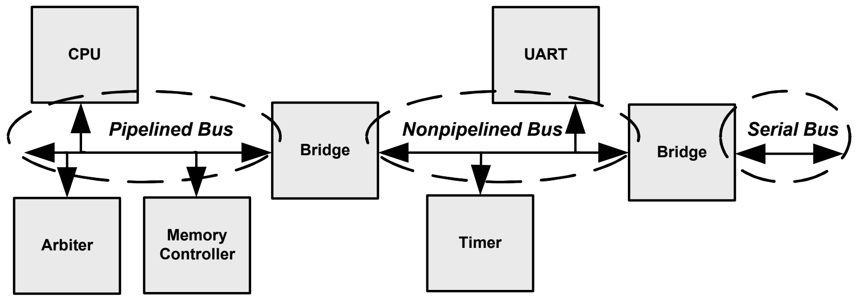
5

INTERFACES

On-chip busses and standard interfaces serve as the framework for platform-based SoC designs, effectively providing the mortar that binds IP blocks together. In fact, on-chip bus protocols such as the ARM AMBA Advanced High-performance Bus (AHB) [AMBA 1999] protocol and the Open Core Protocol (OCP) [OCP 2003] form the foundation for many of today's design reuse strategies. This chapter demonstrates our *process* of creating assertion-based IP for the three generic bus interfaces illustrated in Figure 5-1:

- Simple serial bus interface
- Simple nonpipelined bus interface
- Simple pipelined bus interface

Figure 5-1 Standard interfaces for three common bus protocols



In this chapter, you will note that we are following a standard development pattern previously defined in Chapter 3. Each section within this chapter was defined to stand on its on. Hence, you might noticed repetitive text in portions of the chapter.

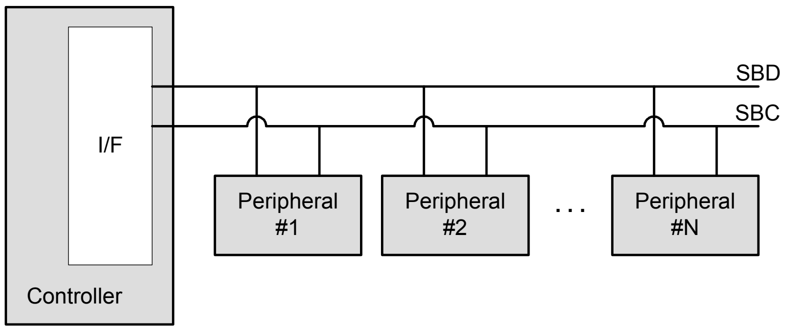
5.1 Simple generic serial bus interface

This section introduces a simple generic serial bus interface example, which is loosely based on a subset of the Inter-IC Bus (I^2C) protocol [I2C 2000]. We ask you to focus on the techniques for creating interface assertions. By using our generic, simple serial bus protocol design example, we hope that you will be able to set aside the details of particular standards 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 other proprietary and real standard serial bus interfaces.

5.1.1 Block diagram interface description

Figure 5-2 illustrates a simple serial bus-based design. The simple serial bus protocol we present provides good support for communication with multiple peripheral components that are accessed intermittently, while being extremely modest in their hardware resource needs. It is a low-bandwidth protocol that allows linking multiple devices through a simple built-in addressing scheme.

Figure 5-2 A simple serial bus design



As illustrated in Figure 5-2, our serial bus-based design consists of a two-bit serial bus—a serial data (SBD) bit and serial clock enable (SBC) bit (see Table 5-1). Together, these signals support a serial protocol transmission of eight-bit bytes of data, seven-bit device addresses, and control bits over the two-bit serial bus. There is no need for a bus select signal or arbitration logic for our serial bus protocol, making it inexpensive and simple to implement in hardware

Table 5-1 Serial bus signal description.

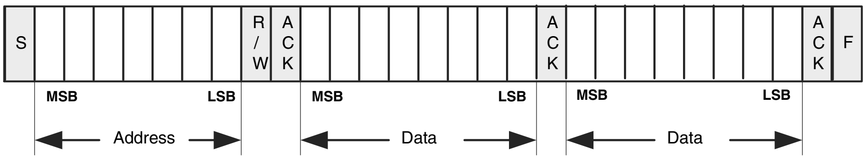
Name	Description
`SBD`	Serial bus data signal
`SBC`	Serial bus clock signal

5.1.2 Overview description

The design component that initiates a bus transaction is the master, which normally controls the clock signal. The design component addressed by the master is a slave. During a transaction, the slave component can pause the master by using a technique known as clock stretching (that is, the slave forces SBC low until it is ready to continue).

Each serial bus slave component is assigned a predefined bus address. The master transmits the address of the slave it intends to communicate with onto the bus at the beginning of every transaction. Each slave monitors the bus and responds only to its own address.

Figure 5-3 A simple serial bus transaction



Simple serial bus transaction

Figure 5-3 illustrates a typical transaction for our simple serial bus protocol. The master begins the transaction by issuing the start command (S), along with a unique seven-bit slave component address.[1] Then, the bit transmitted after the address specifies the type of transaction (that is, R/W). When the master sets R/W to zero, data will be written to the slave. When the master sets R/W to one, data will be read from the slave. Next, the selected slave transmits an ACK, indicating the receipt of the previous byte. The transmitter (either the slave or master depending on the type of transaction) then starts the transfer of a byte of data a bit at a time, beginning with the MSB. After completing the data transfer, the receiver issues an ACK. This data transmit pattern can be repeated, if required, to transmit a contiguous stream of data, without needing to transmit a new address before each data byte. After a transaction completes, the master issues a finish command (F).[2]

1. Our simple serial bus commands start (S), ACK, and finish (F) are defined by a unique rising or falling edge phase relationship between the SBD when SBC is high (similar to I^2C). The details are not important for our discussion. We can assume that our bus interface contains the required circuitry to detect these unique phase relationships, and then decode an appropriate bus command.
2. Our serial bus differs from a real standard serial bus protocol, such as the I^2C, in that out simple example does not support NAK, restart, global call, and other features. Our goal is to present a simple example to illustrate the assertion-based IP development process.

5.1.3 Natural language properties

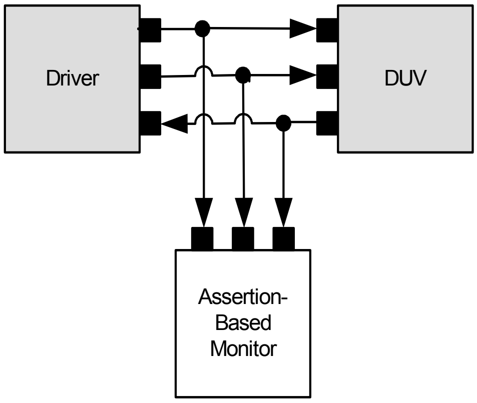
Our first task when creating a set of interface assertions for our simple serial bus is to identify a comprehensive list of natural language properties. You might ask the question *where do the properties come from?* In general, properties can be extracted from your proprietary or standard serial bus specification. Table 5-2, although certainly not a comprehensive list of properties for a real serial bus, is a representative list of properties that we will use to demonstrate our process. You might note that the data integrity class of properties is missing from this list. This class of properties is discussed separately in Chapter 8.

Table 5-2 Simple serial bus properties

Assertion name	Summary
Handshaking properties	
a_bus_reset	After a reset, the `SBD` and `SBC` signals must be driven high
a_no_double_start	A new `start` command cannot be issued until a `finish` command is issued[1]
a_no_finish_before_start	A single `finish` command can only be issued after a `start`.
Valid serial transaction property	
a_valid_transfer_size	A serial transaction consists of a `start`, 7-bits of address, `R/W` command, `ACK`, followed by a sequence of 8-bit transfers with an `ACK`, and a `finish`

1. Our simple example, unlike the I^2C serial bus, does not support a restart. Hence, we have added this property to illustrate how to write an assertion to check for illegal back-to-back events.

Figure 5-4 SystemVerilog interface



5.1.4 Assertions

To create a set of SystemVerilog assertions for our simple nonpipelined bus, 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). To accomplish this task, we create a SystemVerilog interface, as illustrated in Figure 5-4 (see page 68).

Example 5-1 demonstrates an interface for our serial bus example. For this case, our assertion-based monitor references the interface signals in the direction defined by the `monitor_mp` named modport.

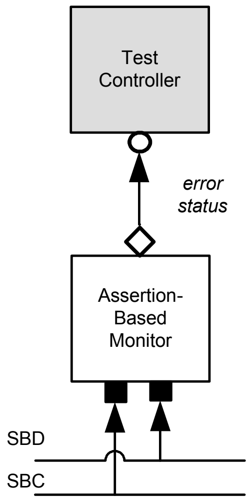
Example 5-1 Encapsulate bus signals inside a SV interface

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

  bit       SBC;
  bit       SBD;
  bit       start;
  bit       finish;
  bit       ack;

  modport driver_mp (
    . . .
  );

  modport duv_mp (
    . . .
  );

  modport monitor_mp (
    input       sclk , rst ,
    input       SBC , SBD ,
    input       start , finish , ack
  );

endinterface
```

In addition to pin-level interfaces (which monitor the bus) we need to define the analysis port interface, as illustrated in Figure 5-5, which broadcasts an error status transaction upon detecting a serial bus error. Example 5-2 defines the error status transaction class, which is an extension on an `ovm_transaction` base class. There are a few standard utility methods that must be defined for the `ovm_transaction`, which include a transaction copy and compare method (see Appendix B, "Complete OVM/AVM Testbench Example" for additional details).

The `tb_status_sb` class in Example 5-2 includes an `enum` that identifies the various types of serial bus errors and a set of methods to uniquely identify the specific error it detected.

Figure 5-5 A simple serial bus transaction



minimize an assertion-based monitor's knowledge of a particular simulation architecture

By introducing an error status transaction object into our assertion-based monitor, which is broadcast through an analysis port upon detecting an error, our monitor does not need to know any details on how a particular simulation architecture represents an error condition within its environment. This makes the assertion-based monitor truly reusable across many different simulation architectures.

clocking and reset

We are now ready to write our assertions for the serial bus properties defined by Table 5-2 (see page 67). For our examples, we introduce the signals `sclk` and `rst`, which are not part of the serial bus definition. However, to simplify our example, we assume that these signals (or similar signals) exist within the bus interface for the various design components attached to the bus.

Example 5-2 Error status transaction class

```
class tb_status_sb extends ovm_transaction;

  typedef enum { ERR_BUS_RESET ,
                 ERR_NO_DOUBLE_START ,
                 ERR_NO_FINISH_BEFORE_START ,
                 ERR_VALID_DATA_SIZE } bus_status_t;

  bus_status_t bus_status;

// Standard OVM transaction methods here, not important for our
// discussion. Reference Section B.1.7 (see page 287) for more
// details on these standard methods

   . . .

// Error status transaction methods

  function void set_err_bus_reset;
    bus_status = ERR_BUS_RESET;
  endfunction

  function void set_err_no_double_start;
    bus_status = ERR_NO_DOUBLE_START;
  endfunction

  function void set_err_no_finish_before_start;
    bus_status = ERR_NO_FINISH_BEFORE_START;
  endfunction

  function void set_err_valid_transfer_size;
    bus_status = ERR_VALID_TRANSFER_SIZE;
  endfunction

endclass
```

serial bus reset assertion

Our first assertion states that after a reset, the serial bus signals `SBD` and `SBC` must be driven high. Hence, we can express this property in SVA as shown in Assertion 5-1. If an error is detected, an error status transaction object (previously declared within the body of the assertion-based monitor) is constructed within the action block (that is, `else` clause) of the assertion. Then the `set_err_bus_reset` method is called to identify the specific serial bus error. Finally, the error `status` object is broadcast out through an analysis port (using the `write()` method) to any verification component that had previously subscribed to the assertion-based monitor’s analysis port (see section 3.3.3 "Analysis ports" on page 50 for details).

Assertion 5-1 Serial bus reset condition

```
. . .
tb_status_sb status; // see Example 5-3 on page 75 for more details
. . .
property p_bus_reset;
  @(posedge monitor_mp.sclk)
    $fell(monitor_mp.rst) |->
                 (monitor_mp.SBD==1'b1 & monitor_mp.SBC==1'b1);
endproperty
a_bus_reset: assert property (p_bus_reset) else begin
  status = new();
  status.set_err_bus_reset();
  status_ap.write(status);
end
```

On the surface, you might argue that all this extra coding in the action block is unnecessary complexity! If this was an assertion embedded in the design, we would agree with you. However, our goal is to create a reusable verification component that must interact with other verification components within a testbench. Hence, adding an error status analysis port to our monitor and then having the assertions report errors through this analysis port helps us achieve our goal of reuse.

Table 5-2 defines our second assertion (`a_no_double_start`), which is shown in Assertion 5-2. Although the `start` signal is not defined as part of our serial bus protocol, to simplify our example, we assume it exists inside the serial interface and is asserted when the interface logic decodes a serial bus start command (`S`). Likewise, the `finish` signal asserts when the logic decodes the serial bus `finish` command (`F`) and a serial bus acknowledge (ACK).

Assertion 5-2 No double start property

```
property p_no_double_start;
  @(posedge monitor_mp.sclk) disable iff (monitor_mp.rst)
    monitor_mp.start |=>
        (~monitor_mp.start throughout monitor_mp.finish[->1]);
endproperty
a_no_double_start: assert property (p_no_double_start)else begin
  status = new();
  status.set_err_no_double_start();
  status_ap.write(status);
end
```

For our third property, *no finish before start*, we must handle a boundary condition that ensures `finish` is not asserted until the first occurrence of `start` after a reset. Hence, in Assertion 5-3 we have added the extra `a_no_finish_until_start_init` property to handle this boundary condition.

Assertion 5-3 No finish before start property

```
// Boundary condition

property p_no_finish_until_start_init;
  @(posedge monitor_mp.sclk)
    $fell(monitor_mp.rst) |->
         ~monitor_mp.finish throughout monitor_mp.start[->1];
endproperty
a_no_finish_until_start_init:
  assert property (p_no_finish_until_start_init) else begin
    status = new();
    status.set_err_no_finish_until_start();
    status_ap.write(status);
  end

// Normal condition

property p_no_finish_before_start;
  @(posedge monitor_mp.sclk) disable iff (monitor_mp.rst)
    monitor_mp.finish |=>
         (~monitor_mp.finish throughout monitor_mp.start[->1]);
endproperty
a_no_finish_before_start:
  assert property (p_no_finish_before_start) else begin
    status = new();
    status.set_err_no_finish_until_start();
    status_ap.write(status);
  end
```

Since we are really checking the same property, we did not introduce a separate error condition as part of the error status transaction enum (although we could have). Refer to Example 5-2 on page 71.

Table 5-2 defines our fourth assertion (`a_valid_transfer_size`). We can express a SystemVerilog assertion as shown in Assertion 5-4.

Assertion 5-4 Valid transfer size property

```
property p_valid_transfer_size;
 @(posedge monitor_mp.sclk) disable iff (monitor_mp.rst)
   monitor_mp.start |=>
// repeat minimum of two times for address followed by data phase
     ($rose(monitor_mp.SBC)[=8] ##1 monitor_mp.ack)[*2:$]
 ##1 (monitor_mp.finish);
endproperty
a_valid_data_size: assert property (p_valid_transfer_size) else
begin
  status = new();
  status.set_err_valid_transfer_size();
  status_ap.write(status);
end
```

As previously discussed, the serial bus interface `start` and `finish` signals are not defined by the serial bus protocol. They exist inside the serial interface and are asserted when the interface logic decodes a serial bus start command (`S`) or a finish command (`F`). Likewise, the serial bus interface `ack` signal is asserted when the interface logic decodes a serial bus ack command (`ACK`). In addition, for our SystemVerilog example in Example 5-4, the serial bus interface contains a posedge clock (`sclk`) and an active high reset (`rst`). The master (and slave) interface's internal data sample signal is asserted upon synchronizing the interface's internal clock (`sclk`) with the bus's SBC signal, which represents a valid time to sample the bus data.

5.1.5 Encapsulate properties

In this step, we turn our set of related serial bus interface assertions (and any defined coverage properties, not demonstrated in this example) into a reusable assertion-based monitor verification component. We add analysis ports to our monitor for communication with other simulation verification components as demonstrated in Chapter 3, “The Process.”

Example 5-3 Encapsulate properties inside a module or interface

```
import ovm_pkg::*;
import tb_tr_pkg::*; // tb_status_sb class definition

module serial_bus_mon(
  interface.monitor_mp monitor_mp
);

 ovm_analysis_port #(tb_status_sb) status_ap =
                                 new("status_ap", null);
  . . .
 tb_status_sb status; // error status object
  . . .
 // Any required modeling code here (to model environment)
  . . .
 // All assertions and coverage properties here
  . . .
endmodule
```

5.2 Simple generic nonpipelined bus interface

This section introduces a simple, generic nonpipelined bus interface example, which is loosely based on a subset of the AMBA Advanced Peripheral Bus (APB) protocol [AMBA 1999]. 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 a particular industry protocol standard. We ask you to focus on the techniques for creating interface assertions. By using our generic, simple nonpipelined bus protocol design, we hope that you will be able to set aside the details of particular standards 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 other proprietary and standard nonpipelined bus interfaces. If you are interested in seeing specific property examples for the AMBA protocol (verses our generic examples), there have been numerous papers and books published that provide these details (a few examples include [Marschner 2002] [Ruah et al., 2005] [Susantol and Melham 2003]). In addition, there are commercial AMBA VIP solutions available for purchase.

5.2.1 Block diagram interface description

Figure 5-6 illustrates a block diagram for a simple nonpipelined bus design. For our example, all signal transitions relate only to the rising edge of the bus clock (`clk`). Table 5-3 provides a summary of the bus signals for our simple nonpipelined bus interface example.

Figure 5-6 Simple nonpipelined bus design

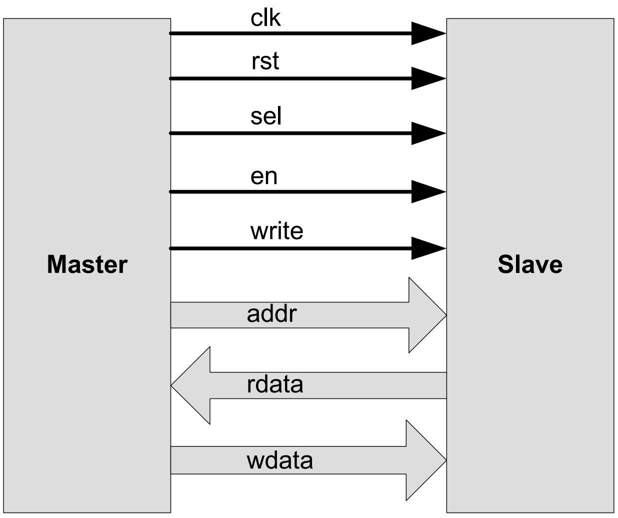
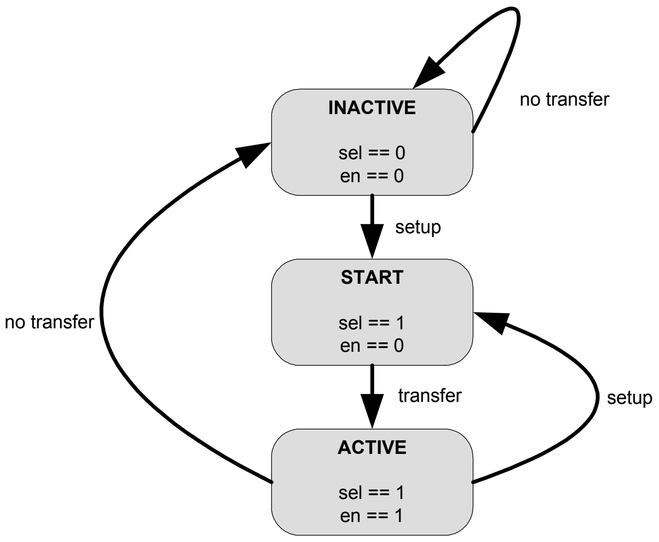


Table 5-3 Simple nonpipelined bus signal description

Name	Description
`clk`	All bus transfers occur on the rising edge of `clk`
`rst`	An active high bus reset
`sel`	These signals indicate that a slave has been selected. Each slave has its own select (for example, `sel[0]` for slave 0). However, for our simple example, we assume a single slave
`en`	Strobe for active phase of bus
`write`	When high, write access When low, read access
`addr[7:0]`	Address bus
`rdata[7:0]`	Read data bus driven when `write` is low
`wdata[7:0]`	Write data bus driven when `write` is high

Figure 5-7 Simple nonpipelined bus conceptual states



5.2.2 Overview description

We use a conceptual state-machine, illustrated in Figure 5-7, to describe the operation of the nonpipelined bus (for a single slave). Essentially, our state-machine maps bus control values to conceptual states.

After a reset (that is, `rst==1'b1`), our simple nonpipelined bus is initialized to its default INACTIVE state, which means both `sel` and `en` are de-asserted. To initiate a transfer, the bus controls moves into the START state, where the master asserts a slave select signal, `sel`, selecting a single slave component. To further simplify our example, we assume that there is only a single slave (`sel`). The bus only remains in the START state for one clock cycle, and will then move to the ACTIVE state on the next rising edge of the clock. The ACTIVE state lasts a single clock cycle for the data transfer. Then, the bus will move back to the START state if another transfer is required, which is indicated when the selection signal remains asserted. Alternatively, if no additional transfers are required, the bus moves back to the INACTIVE state when the master de-asserts the slave's select and bus enable signals.

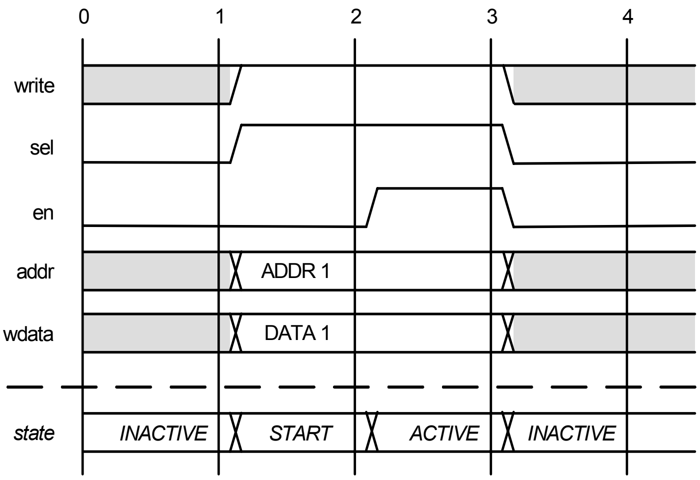
The address (`addr[7:0]`), write control (`write`), and transfer enable (`en`) signals are required to remain stable during the transition from START to ACTIVE states. However, it is not required that these signals remain stable during the transition from ACTIVE back to the START state.

Basic write operation

Figure 5-8 illustrates a basic write operation for our simple nonpipelined bus interface involving a bus master and a single slave.

At clock one, since both the slave select (`sel`) and bus enable (`en`) signals are de-asserted, our bus controls are in an INACTIVE state, as we previously defined in our conceptual state-machine (see Figure 5-6) and illustrated in Figure 5-8. The `state` variable in Figure 5-8 is actually a conceptual state of the bus with respect to the bus controls, not a physical state implemented in the design.

Figure 5-8 Non-burst write transaction



The first cycle of the transfer is called the START cycle, which the master initiates by asserting one of the slave select lines. For our example, the master asserts `sel`, and this event is detected by the rising edge of clock two. During the START cycle, the master places a valid address on the bus and in the next cycle, places valid data on the bus. This data will be written to the currently selected slave component.

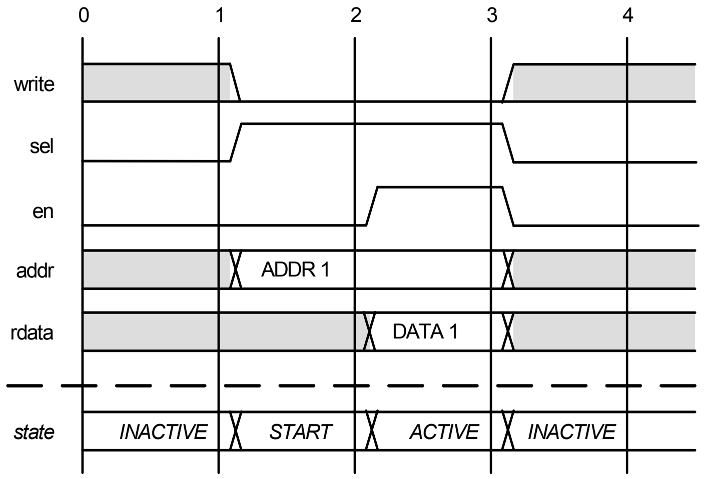
The data transfer (referred to as the ACTIVE cycle) actually occurs when the master asserts the bus enable signal. In our case, it is detected on the rising edge of clock three. The address, data, and control signals all remain valid throughout the ACTIVE cycle.

When the ACTIVE cycle completes, the bus master de-asserts the bus enable signal (`en`), and thus completes the current single-cycle write operation. If the master has finished transferring all data to the slave (that is, there are no more write operations), then the master de-asserts the slave select signal (that is, `sel`). Otherwise, the slave select signal remains asserted, and the bus returns to the START cycle to initiate another write operation. It is not necessary for the address data values to remain valid during the transition from the ACTIVE cycle back to the START cycle.

Basic read operation

Figure 5-9 illustrates a basic read operation for our simple bus interface involving a bus master and slave zero (`sel`).

Figure 5-9 Non-burst read transaction



Just like the write operation, since both the slave select (`sel`) and bus enable (`en`) signals are de-asserted at clock one, our bus is in an INACTIVE state, as we previously defined in our conceptual state machine (see Figure 5-6). The timing of the address, write, select, and enable signals are all the same for the read operation as they were for the write operation. In the case of a read, the slave must place the data on the bus for the master to access during the ACTIVE cycle, which Figure 5-9 illustrates at clock three. Like the write operation, back-to-back read operations are permitted from a previously selected slave. However, the bus must always return to the START cycle after each ACTIVE cycle completes.

5.2.3 Natural language properties

Prior to creating a set of SystemVerilog interface assertions for our simple nonpipelined bus, we must identify a comprehensive list of natural language properties. We begin by classifying the properties into categories, as shown in Table 5-4. The first category relates to properties associated with bus state transitions, while the remaining categories relate to properties associated with specific bus interface signals. 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.

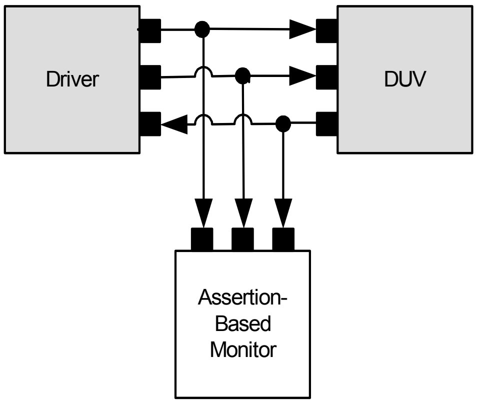
Table 5-4 Simple nonpipelined bus interface properties

Assertion Name	Summary
Bus legal state	
a_state_reset_inactive	INACTIVE is the initial state after reset
a_valid_inactive_transition	INACTIVE or START follows INACTIVE
a_valid_start_transition	ACTIVE state follows START
a_valid_active_transition	INACTIVE or START follows ACTIVE
a_no_error_state	Bus `sel` and `en` must be in a valid state
Bus select	
a_sel_stable	From START to ACTIVE, `sel` is stable
Bus address	
a_addr_stable	From START to ACTIVE, `addr` is stable
Bus write control	
a_write_stable	From START to ACTIVE, `write` is stable
Bus data	
a_wdata_stable	From START to ACTIVE, `wdata` is stable

5.2.4 Assertions

To create a set of SystemVerilog assertions for our simple nonpipelined bus, 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). To accomplish this task, we create a SystemVerilog interface, as illustrated in Figure 5-10.

Figure 5-10 SystemVerilog interface



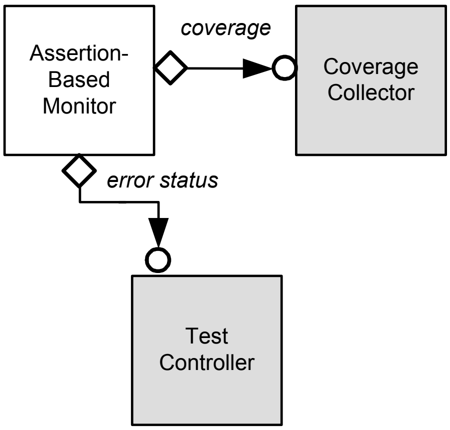
Example 5-4 Encapsulate bus signals inside a SV interface

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

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

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

  modport driver_mp (
    . . .
  );

  modport duv_mp (
    . . .
  );

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

endinterface
```

Example 5-4 (see page 82) demonstrates an interface for our nonpipelined bus example. For this case, our assertion-based monitor references the interface signals in the direction defined by the `monitor_mp` named modport.

In addition to pin-level interfaces (that monitor the bus) we need a means for our nonpipelined bus assertion-based monitor to communicate with various analysis verification components within the testbench. Hence, we introduce an error status analysis port within the monitor, as Figure 5-11 illustrates. In addition to assertion error status, coverage events provide 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 5-11 Error status and coverage analysis ports



Upon detecting a bus error, the analysis port broadcasts the error condition (using an error status transaction object) to other verification components. Example 5-5 defines the *error status* transaction class, which is an extension on an `ovm_transaction` base class. The `tb_status_nb` class includes an `enum` that identifies the various types of nonpipelined bus errors and a set of methods to uniquely identify the specific error it detected.

Example 5-5 Error status class

```
class tb_status_nb extends ovm_transaction;

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

  bus_status_t   bus_status;

 // Standard OVM transaction methods here, not important for our
 // discussion. Reference Section B.1.7 (see page 287) for more
 // details on these standard methods
    . . .
 // Error status transaction methods

  function void set_err_trans_reset;
    bus_status = ERR_TRANS_RESET;
  endfunction

  function void set_err_trans_inactive;
    bus_status = ERR_TRANS_INACTIVE;
  endfunction

  function void set_err_trans_start;
    bus_status = ERR_TRANS_START;
  endfunction

  function void set_err_trans_active;
    bus_status = ERR_TRANS_ACTIVE;
  endfunction

  function void set_err_trans_error;
    bus_status = ERR_TRANS_ERROR;
  endfunction

. . .

endclass
```

To simplify writing our assertion (and to increase the clarity), we create some modeling code (see Example 5-6) to map the `sel` and `en` bus control signals to the conceptual bus states (see Figure 5-7). We then write a set of assertions to detect protocol violations by monitoring illegal bus state transitions related to these conceptual states.

Example 5-6 Modeling to map control signals to conceptual states

```
module tb_nonpipelined_mon(
  interface.monitor_mp monitor_mp
);

  ovm_analysis_port #(tb_status_nb) status_ap =
                                       new("status_ap", null);
  . . .
  parameter DATA_SIZE  = 8;
  parameter ADDR_SIZE  = 8;

  tb_status_nb status;

// Used to decode bus control signals

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

  bit                  bus_write;
  bit                  bus_sel;
  bit                  bus_en;
  bit                  bus_reset;

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

// Identify conceptual states from bus control signals

  always @(posedge monitor_mp.clk) begin

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

    if (monitor_mp.rst) begin
      bus_inactive = 1;
      bus_start    = 0;
      bus_active   = 0;
      bus_error    = 0;
    end
    else begin // decode sel and en to conceptual states
      bus_inactive = ~monitor_mp.sel & ~monitor_mp.en;
      bus_start    =  monitor_mp.sel & ~monitor_mp.en;
      bus_active   =  monitor_mp.sel &  monitor_mp.en;
      bus_error    = ~monitor_mp.sel &  monitor_mp.en;
    end
  end

  . . .

 endmodule
```

We are now ready to write assertions for the bus interface properties that Table 5-4 (see page 81) defines. The first property states that after a reset, the bus must be initialized to an INACTIVE state (which means the `sel` and `en` signals are de-asserted). Assertion 5-5 demonstrates a SystemVerilog assertion for this property.

Assertion 5-5 Bus must reset to INACTIVE state property

```
property p_state_reset_inactive;
  @(posedge monitor_mp.clk)
    $fell(bus_reset) |-> (bus_inactive);
endproperty
a_state_reset_inactive:
  assert property (p_state_reset_inactive) else begin
    status = new();
    status.set_err_trans_reset();
    status_ap.write(status);
  end
```

Once again, you might argue that all this extra coding in the action block is unnecessary complexity! If this was an embedded assertion within the design, we would agree with you. However, our goal is to create a reusable verification component that must interact with other verification components within a testbench. Hence, adding an error status analysis port to our monitor, and then having the assertions report errors through this analysis port helps us achieve our goal of reuse.

We can write assertions for all the *bus legal state* properties that Table 5-4 defines, as shown in Assertion 5-6 and defined by our conceptual state-machine.

Assertion `a_valid_inactive_transition` specifies that if the bus is currently in an INACTIVE state, then the next state of the bus must be either INACTIVE again or a START state.

Assertion `a_valid_start_transition` specifies that if the bus is currently in a START state, then on the next clock cycle, the bus must be in an ACTIVE state.

Assertion `a_no_active_to_active` specifies that if the bus is in an ACTIVE state, then on the next clock cycle, the bus must be in either an INACTIVE or START state.

Assertion 5-6 Assertions for bus legal state properties

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

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

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

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

Finally, `a_no_error_state` shown in Assertion 5-6 specifies that only valid combinations of `sel` and `en` are permitted on the bus.

Assertion 5-7 Stability properties

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

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

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

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

The remaining properties that Table 5-4 defines and Assertion 5-7 demonstrates, specify that the bus select, control, address, and data signals must remain stable between a bus `` `START `` state and the bus `` `ACTIVE `` state.

5.2.5 Encapsulate properties

In this step, we turn our set of related nonpipelined bus interface assertions (and any coverage properties, which are demonstrated at the end of this chapter in Section 5.4) 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 5-7 Encapsulate properties inside a module

```
import ovm_pkg::*;
import tb_tr_pkg::*; // tb_status_nb class definition

module tb_nonpipelined_mon (
  interface.monitor_mp monitor_mp
);

  ovm_analysis_port #(tb_status_nb)
                       status_ap = new("status_ap", null);
  ovm_analysis_port #(tb_coverage)
                       coverage_ap = new("coverage_ap", null);

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

5.3 Simple generic pipelined bus interface

In Section 5.2, we demonstrated how to create an assertion-based IP verification component for a bus interface supporting a simple, generic nonpipelined protocol. In this section, we introduce a more advanced pipelined bus that supports features such as a *split* transaction. For our discussion, we present a generic pipelined bus protocol, which is loosely based on a subset of the AMBA AHB protocol [AMBA 1999]. We ask you to focus on the techniques for creating interface assertions. By using our generic, simple pipelined bus protocol design, we hope that you will be able to set aside the details of particular standards 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 other proprietary and standard pipelined bus interfaces. If you are interested in seeing specific property examples for the AMBA protocol (versus our generic examples), there have been numerous papers and books published that provide these details (a few examples include [Marschner 2002] [Ruah et al., 2005] [Susantol and Melham 2003]). In addition, there are commercial AMBA VIP solutions available for purchase.

5.3.1 Block diagram interface definition

Figure 5-12 illustrates a generic pipelined bus design, and Table 5-5 provides descriptions of the bus signals. To simplify our example, we assume a single bus master. However, our example is easily extended to include multiple bus masters by introducing an arbiter into the pipelined bus design. Arbiters are discussed separately in Chapter 6, “Arbiters.”

Figure 5-12 Simple pipelined bus design

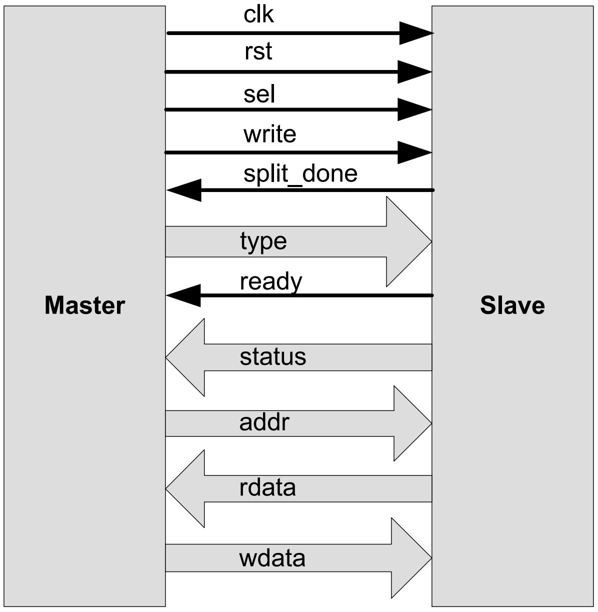


Table 5-5 Pipelined bus interface signal descriptions

Name	Description
`clk`	Bus clock. All transfers occur on rising edge of `clk`.
`rst`	Bus reset. An active high bus reset.
`sel`	Slave select. These signals indicate that a slave has been selected. Each slave has its own select (for example, `sel[0]` for slave 0). For our simple example, we assume a single slave.
`write`	Write control. When high, write transaction. When low, read.
`type[1:0]`	Transfer type. Set to `` `START `` for the first transfer of a burst. Set to `` `CONT `` for subsequent transfers after a `` `START ``. When set to `` `BUSY ``, this indicates that the bus master plans to continue with a burst, but not immediately. When set to `` `IDLE ``, no data transfer is required.

Name	Description
`status[1:0]`	Slave status response. Set to `` `OK`` indicates that the transfer has completed successfully. Set to `` `ERROR`` indicates an error has occurred; the transfer was unsuccessful. Set to `` `SPLIT`` indicates that the transfer has not yet completed successfully—the bus master must retry the transfer when it is next granted access to the bus, per a slave request.
`split_done`	This signal is low when the slave is performing a `'SPLIT` transaction (indicating not done). Otherwise, it is active high.
`ready`	Used by a slave to insert wait states.
`addr[31:0]`	Address bus.
`rdata[31:0]`	Read data driven when `write` is low.
`wdata[31:0]`	Write data driven when `write` is high.

5.3.2 Overview description

Pipelining bus transactions allows for higher transfer rates, often at a cost of an initial latency associated with the first transfer. That is, for certain types of transactions, the first access in a pipelined transfer requires several cycles, and subsequent transactions might only involve a single cycle.

Figure 5-13 illustrates a two-phase pipelined bus transfer, where during the first phase (also known as the *address phase*), a bus master asserts an address `A0` (and proper control signals) onto the bus. On the next clock cycle, the *data phase* begins. Here, as illustrated in Figure 5-13, either:

1 The master drives data `D0` onto the `wdata` bus (for a write transaction), which the selected slave samples at clock two.

Or

2 The selected slave drives data `D0` onto the `rdata` bus, which the master samples at clock two.

During the D0 data phase, the master asserts a new address A1 onto the bus to start another transaction. Hence, the pipelined bus's newest address phase will overlap its previous data phase, with the exception of the first and last clock cycles.

Figure 5-13 Two-phase pipelined bus transfer

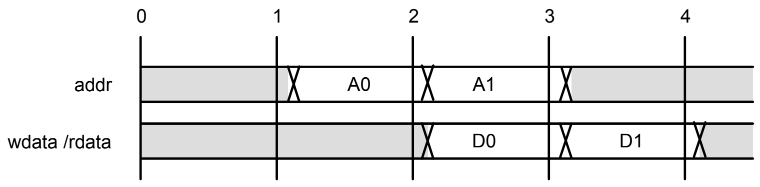
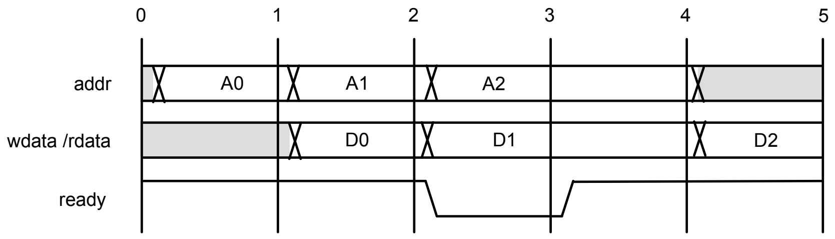


Figure 5-14 illustrates the same two-phase pipelined bus transfer illustrated in Figure 5-13, except in this case, the slave has inserted a wait state into the data transfer by de-asserting the `ready` signal. This effectively stretches the D1 data phase.

Figure 5-14 Pipelined bus transfer with wait state



Inserting a wait state into a pipelined bus cycle presents a number of design challenges, which we do not need to consider for this simple nonpipelined bus example (previously described in Section 5.2). For example, the master in a pipelined bus must stall its pending bus cycle to respect the slave's wait response. Once the slave is ready to proceed, which it indicates by asserting its bus `ready` signal, the master must continue the transfer from the correct address where it left off prior to the wait.

Master command type and slave status

For our simple pipelined bus example, the master sets the bus command `type` to `'IDLE` to indicate that the master does not wish to perform a data transfer. A slave provides a zero wait state `'OK status` response to a master's `'IDLE type` transfers and ignores the transfer.

When the master sets the bus command `type` to `` `START ``, it indicates the start of a burst bus transaction. When the master sets the bus command `type` to `` `CONT `` (after a `'START`), it indicates the continuation of a *burst* bus transfer.[3]

The `'BUSY` transfer `type` indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. Obviously, a bus command type of `'BUSY` (or `'CONT)` cannot occur after an `'IDLE`. When a master uses the `'BUSY` transfer `type`, the address and control signals must reflect the next transfer in the burst and remain stable on the cycle after the `'BUSY`. The slave must ignore the current transfer and provide a zero wait state `'OK` response, in the same way that it responds to `'IDLE` transfers.

pipelined bus wait cycle

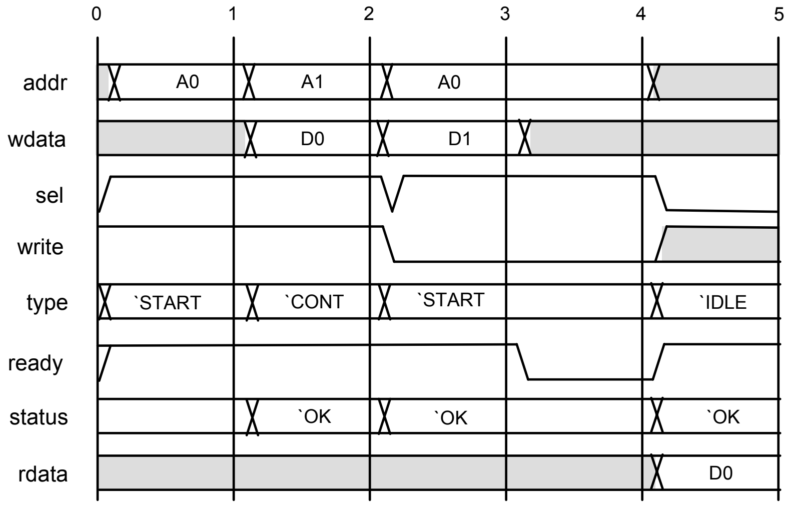
Figure 5-15 (see page 95) illustrates a two-beat write burst followed by a non-burst read. A wait cycle was inserted into the second beat of the write burst at clock four (that is, `ready` is de-asserted and `type` is not equal to `'IDLE`). Note that a slave must always set its status response to `'OK` when inserting a wait cycle into a transfer.

For our simple example, the slave can issue various responses to a master's bus command by setting its `status` to `` `OK ``, `` `ERROR ``, or `` `SPLIT ``. If `` `SPLIT `` is asserted, the current master suspends the bus transfer. For our simple pipelined bus example, we assume that there is only a single master and a single slave and no arbitration is required. (Arbitration is discussed in Chapter 6.) The master then uses the `split_done` as an indication to continue the bus transfer (similar to a grant). However, for a more complex pipelined

3. Unlike the AMBA AHB, our simple pipelined bus example only supports undefined-length burst.

bus protocol that supports multiple masters (such as the actual AMBA AHB), after a `SPLIT, the bus might become available for a different master's bus transfer. The suspended bus master would then need to re-arbitrate for access to the bus when the slave indicates that it is ready to continue the transfer by asserting a signal similar to our simple example's split_done.

Figure 5-15 Pipelined bus burst and non-burst with wait cycle



Two-cycle response

Only an `OK response can be given in a single cycle. The `ERROR and `SPLIT responses require at least two cycles. A two-cycle response consists of the following events:

1 In the first cycle of the two-cycle response, the slave drives type to indicate `ERROR or `SPLIT while de-asserting ready to extend the transfer for an extra cycle.

2 In the second cycle of the two-cycle response, ready is asserted to end the transfer, while type remains driven to indicate `ERROR or `SPLIT.

If the slave needs more than two cycles to provide the `'ERROR` or `'SPLIT` response, then additional wait states may be inserted at the start of the transfer. During this time the `ready` signal is deasserted and the response must be set to `'OK`.

The two-cycle response is required because of the pipelined nature of the bus. By the time a slave starts to issue either an `'ERROR` or `'SPLIT` response, the address for the following transfer has already been broadcast onto the bus. The two-cycle response allows sufficient time for the master to cancel this address and drive `type` to `'IDLE` before the start of the next transfer.

For the `'SPLIT` response, the following transfer must be cancelled because it must not take place before the current transfer has completed. However, for the `'ERROR` response, where the current transfer is not repeated, completing the following transfer is optional.

5.3.3 Natural language properties

Our next step is to create a natural language list of properties for our pipelined bus, as shown in Table 5-6. It is generally helpful to organize the interface properties into separate sets. For example, either organize by functionality or unique interface signal, as shown in Table 5-6. 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.[4]

4. Our simple pipelined bus interface example does not support all the features found in a real pipelined bus protocol, such as sequential burst (SEQ), protection, bus lock, and address wrapping, that are supported by the AMBA AHB protocol.

Table 5-6 Pipelined bus interface properties

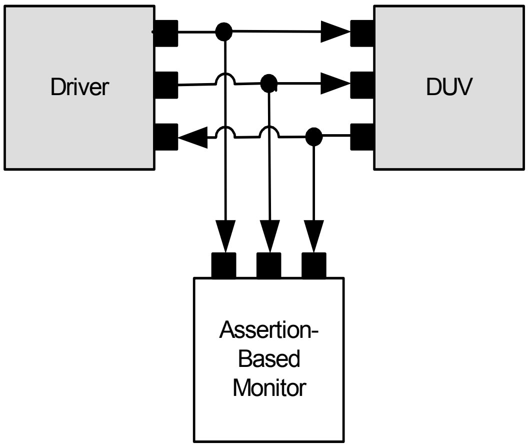
Assertion name	Summary
Bus master type control	
a_type_reset	`type` must be `` `IDLE `` after a reset.
a_type_idle	`type` of `` `BUSY `` or `` `CONT `` cannot be asserted following a `type` of `` `IDLE ``.
a_type_error	For a slave `status` response of `` `ERROR ``, the master must assert `` `IDLE `` or continue the transfer.
a_type_split_done	After a slave `status` response of `` `SPLIT ``, the master cannot set `type` to `` `CONT `` until after the slave asserts `split_done`.
Bus master address	
a_addr_stable_wait	`addr` remains stable after the slave inserts a wait state.
a_addr_stable_busy	`addr` remains stable after a transfer type of `` `BUSY ``.
Bus master write control	
a_write_stable_wait	`write` remains stable after the slave inserts a wait state (that is, de-asserts `ready`).
a_write_stable_burst	`write` remains stable during a burst transaction (`type` = `` `CONT ``).
Bus master wdata	
a_wdata_stable_wait	Master must ensure `wdata` remains stable after slave inserts a wait state.
Bus slave ready	
a_ready_reset	Slave must assert `ready` after reset.
a_ready_idle_wait	Slave must assert `ready` when master sets `type` to `` `IDLE ``.
a_ready_busy_wait	Slave must assert `ready` when master sets `type` to `` `BUSY ``.
a_ready_not_selected	Slave must assert `ready` when not selected.
a_ready_error_cycle	Slave must assert `ready` low on first cycle of error response (that is, when `status` set to `` `ERROR) ``, followed by asserting ready high on the second cycle.

Assertion name	Summary
a_ready_max_wait	Slave must not insert more than 16 consecutive wait states (that is, active low `ready`).
Bus slave status (response)	
a_status_reset	Slave must set `status` response to `` `OK `` after a reset.
a_status_idle_busy_sel	Slave must set `status` response to `` `OK `` when type is set to `` `IDLE ``, type is set to `'BUSY`, or slave not selected.
Bus slave split_done	
a_split_done_reset	Slave must assertion `split_done` after reset.
a_split_done_valid	Split done must only be de-asserted after a status of `'SPLIT`.

5.3.4 Assertions

To create a set of SystemVerilog assertions for our simple pipelined bus, we begin defining the connection between the assertion-based monitor and other components (such as a driver transactor and the DUV), as Figure 5-16 illustrates.

Figure 5-16 SystemVerilog interface



For instance, in Example 5-4 we demonstrate an interface for our pipelined bus example. For this case, our assertion-based monitor references the interface signals in the direction defined by the `monitor_mp` named modport.

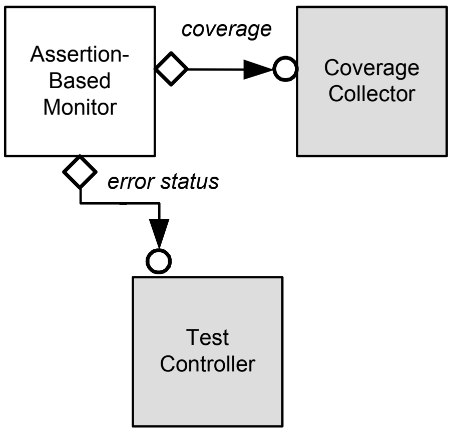
Example 5-8 Encapsulate bus signals inside SV interface

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

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

  bit                   sel;
  bit [1:0]             type;
  bit                   write;
  bit [1:0]             status;
  bit                   split_done;
  bit                   ready;
  bit [DATA_SIZE-1:0]   wdata;
  bit [DATA_SIZE-1:0]   rdata;
  bit [ADDR_SIZE-1:0]   addr;

  modport driver_mp (
    input        clk , rst ,
    output       sel , type, write , addr ,
    output       wdata ,
    input        rdata , status , ready , split_done
  );

  modport duv_mp (
    input        clk , rst ,
    input        sel , type, write , addr ,
    input        wdata ,
    output       rdata , status , ready , split_done
  );

  modport monitor_mp (
    input        clk , rst ,
    input        sel , type, write , addr ,
    input        wdata ,
    input        rdata , status , ready , split_done
  );

endinterface
```

In addition to pin-level interfaces (that monitor the bus) we need a means for our pipelined bus 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 illustrated in

Section . In addition to assertion error status, coverage events provide 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 5-17 Error status and coverage analysis ports



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

Example 5-9 defines the error status transaction class, which is an extension on an `ovm_transaction` base class. The `tb_status_pb` class includes an `enum` that identifies the various types of pipelined bus errors and a set of methods to uniquely identify the specific error it detected.

Example 5-9 Error status transaction class

```
class tb_status_pb extends ovm_transaction;
   typedef enum { ERR_TYPE_RESET ,
                  ERR_TYPE_IDLE ,
                  ERR_TYPE_ERROR ,
                  ERR_TYPE_SPLIT_RETRY ,
                  ERR_TYPE_SPLIT_DONE ,
                  . . . . ,
                  ERR_STATUS_BUSY ,
                  ERR_STATUS_SELECT ,
                  ERR_SPLIT_DONE_RESET ,
                  ERR_SPLIT_DONE_VALID
                } bus_status_t;

   bus_status_t bus_status;

// Standard OVM transaction methods here, not important for our
// discussion. Reference Section B.1.7 (see page 287) for more
// details on these standard methods.
   . . .
// Error status transaction methods

  function void set_err_type_reset;
    bus_status = ERR_TYPE_RESET;
  endfunction

  function void set_err_type_idle;
    bus_status = ERR_TYPE_IDLE;
  endfunction

  function void set_err_type_error;
    bus_status = ERR_TYPE_ERROR;
  endfunction
  . . . .

  function void set_err_split_done_valid;
    bus_status = ERR_SPLIT_DONE_VALID;
  endfunction
  . . .

endclass
```

The first set of properties listed in Table 5-6 (see page 97) specifies the expected behavior of the *bus transaction type command* (`type`) with respect to a reset condition, idle state, wait state, and split transaction. We have grouped these related properties and demonstrated how to express them as SystemVerilog assertions in Assertion 5-8.

Assertion 5-8 Bus transaction type command properties

```
property p_type_reset;
  @(posedge monitor_mp.clk)
    $fell(monitor_mp.rst) |-> (monitor_mp.type==`IDLE);
endproperty
a_type_reset:
  assert property (p_type_reset) else begin
    status = new();
    status.set_err_type_reset();
    status_ap.write(status);
  end

property p_type_idle;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (monitor_mp.type==`IDLE) |=>
              (monitor_mp.type!=`BUSY && monitor_mp.type!=`CONT);
endproperty
a_type_idle:
  assert property (p_type_idle) else begin
    status = new();
    status.set_err_type_idle();
    status_ap.write(status);
  end

property p_type_error;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    ($rose(monitor_mp.ready) && (monitor_mp.status==`ERROR)) |=>
             (monitor_mp.type==`IDLE || $stable(monitor_mp.type));
endproperty
a_type_error:
  assert property (p_type_error) else begin
    status = new();
    status.set_err_type_error();
    status_ap.write(status);
  end

property p_type_split_done;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (monitor_mp.status==`SPLIT) |=> (monitor_mp.type=='IDLE)
throughout split_done[->1];
endproperty
a_type_split_done:
  assert property (p_type_split_done) else begin
    status = new();
    status.set_err_type_split_done();
    status_ap.write(status);
  end
```

On the surface, you might argue that all this extra coding in the action block is unnecessary complexity! If this was an embedded assertion within the design, we would agree with you. However, our goal is to create a reusable verification component that must interact with other verification components within a testbench. Hence, adding an error status analysis port to our monitor, and then having the assertions report errors through this analysis port helps us achieve our goal of reuse.

For the *bus master address* properties that Table 5-6 (see page 97) defines, we can write a set of SystemVerilog assertions as demonstrated in Assertion 5-9. Note that, unlike the real AMBA AHB protocol, our simple pipelined bus example does not support a sequential burst (for example, an AMBA AHB transfer type of SEQ).

Assertion 5-9 Bus master address properties

```
property p_addr_stable_wait;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (~monitor_mp.ready & monitor_mp.status=='OK &
      monitor_mp.type!='IDLE) |=>  $stable(monitor_mp.addr);
endproperty
a_addr_stable:
  assert property (p_addr_stable_wait) else begin
    status = new();
    status.set_err_addr_stable_wait();
    status_ap.write(status);
  end

property p_addr_stable_busy;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (monitor_mp.type=='BUSY) |=>  $stable(monitor_mp.addr);
endproperty
a_addr_stable:
  assert property (p_addr_stable_busy) else begin
    status = new();
    status.set_err_addr_stable_busy();
    status_ap.write(status);
  end
```

Assertion 5-10 demonstrates a SystemVerilog assertion for the *bus master write control* property that Table 5-6 defines, which specifies a stable bus write control condition for our bus interface.

Assertion 5-10 Bus master write command properties

```
property p_write_stable_wait;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (~monitor_mp.ready & monitor_mp.status=='OK &
      monitor_mp.type!='IDLE) |=> $stable(monitor_mp.write);
endproperty
a_write_stable_wait:
  assert property (p_write_stable_wait) else begin
    status = new();
    status.set_err_write_stable_wait();
    status_ap.write(status);
  end

property p_write_stable_burst;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (monitor_mp.type==`CONT) |-> $stable(monitor_mp.write);
endproperty
a_write_stable_burst:
  assert property (p_write_stable_burst) else begin
    status = new();
    status.set_err_write_stable_burst();
    status_ap.write(status);
  end
```

Assertion 5-11 demonstrates a SystemVerilog assertion for the *bus wdata* property that Table 5-6 defines.

Assertion 5-11 Bus wdata properties

```
property p_wdata_stable_wait;
  @(posedge monitor_mp.clk)
    (~monitor_mp.ready & monitor_mp.status=='OK &
      monitor_mp.type!='IDLE) |-> $stable(monitor_mp.wdata);
endproperty
a_data_stable_wait:
  assert property (p_data_stable_wait) else begin
    status = new();
    status.set_err_data_stable_wait();
    status_ap.write(status);
  end
```

Assertion 5-12 demonstrates a SystemVerilog assertion for the *bus slave ready* property that Table 5-6 defines.

Assertion 5-12 Bus slave ready assertions

```
property p_ready_reset
  @(posedge monitor_mp.clk)
    $fell(monitor_mp.rst) |-> (monitor_mp.ready);
endproperty
a_ready_reset:
  assert property (p_ready_reset) else begin
    status = new();
    status.set_err_ready_reset();
    status_ap.write(status);
  end

property p_ready_idle_wait;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (monitor_mp.type==‘IDLE) |=> monitor_mp.ready;
endproperty
a_ready_idle_wait:
  assert property (p_ready_idle_wait) else begin
    status = new();
    status.set_err_ready_idle_wait();
    status_ap.write(status);
  end

property p_ready_busy_wait;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (monitor_mp.type==‘BUSY) |=> monitor_mp.ready;
endproperty
a_ready_busy_wait:
  assert property (p_ready_busy_wait) else begin
    status = new();
    status.set_err_ready_busy_wait();
    status_ap.write(status);
  end

property p_ready_not_selected;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    (monitor_mp.sel==1’b0) |-> monitor_mp.ready;
endproperty
a_ready_not_selected:
  assert property (p_ready_not_selected) else begin
    status = new();
    status.set_err_ready_not_selected();
    status_ap.write(status);
  end

// Continued on next page
```

Assertion 5-12 Bus slave ready assertions

```
property p_ready_error_cycle;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    $rose(monitor_mp.status=='ERROR) |-> ~monitor_mp.ready ##1
                  monitor_mp.ready && (monitor_mp.status=='ERROR);
endproperty
a_ready_error_cycle:
  assert property (p_ready_error_cycle) else begin
    status = new();
    status.set_err_ready_error_cycle();
    status_ap.write(status);
  end

property p_ready_max_wait;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    $fell(monitor_mp.ready) |=> (
              ~monitor_mp.ready[*0:15] ##1 monitor_mp.ready);
endproperty
a_ready_max_wait:
  assert property (p_ready_max_wait) else begin
    status = new();
    status.set_err_ready_max_wait();
    status_ap.write(status);
  end
```

Assertion 5-13 demonstrates a SystemVerilog assertion for the *bus slave status response* properties that Table 5-6 defines.

Assertion 5-13 Bus slaves status response properties

```
property p_status_reset;
  @(posedge monitor_mp.clk)
    $fell(monitor_mp.rst) |-> (monitor_mp.status==`OK);
endproperty
a_status_reset:
  assert property (p_status_reset) else begin
    status = new();
    status.set_err_status_reset();
    status_ap.write(status);
  end

// Continued on next page
```

Assertion 5-13 Bus slaves status response properties

```
property p_status_idle_busy_no_sel;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    ((monitor_mp.sel==1'b0)   ||
     (monitor_mp.type==`IDLE) ||
     (monitor_mp.type==`BUSY))   |-> (monitor_mp.status==‘OK);
endproperty
a_status_idle_busy_no_sel:
  assert property (p_status_idle_busy_no_sel) else begin
    status = new();
    status.set_err_status_idle_busy_no_sel();
    status_ap.write(status);
  end
```

Assertion 5-14 demonstrates a SystemVerilog assertion for the *bus split_done* property that Table 5-6 defines.

Assertion 5-14 Bus split_done properties

```
property p_split_done_reset;
  @(posedge monitor_mp.clk)
    $fell(monitor_mp.rst) |-> monitor_mp.split_done;
endproperty
a_split_done_reset:
  assert property (p_split_done_reset) else begin
    status = new();
    status.set_err_data_stable_wait();
    status_ap.write(status);
  end

property p_split_done_valid;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    $fell(monitor_mp.split_done) |->
                        $past(monitor_mp.type==‘SPLIT);
endproperty
a_split_done_valid:
  assert property (p_split_done_valid) else begin
    status = new();
    status.set_err_data_stable_wait();
    status_ap.write(status);
  end
```

5.3.5 Encapsulate properties

In this step, we turn our set of related pipelined bus interface 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 5-10 Encapsulate properties inside a module

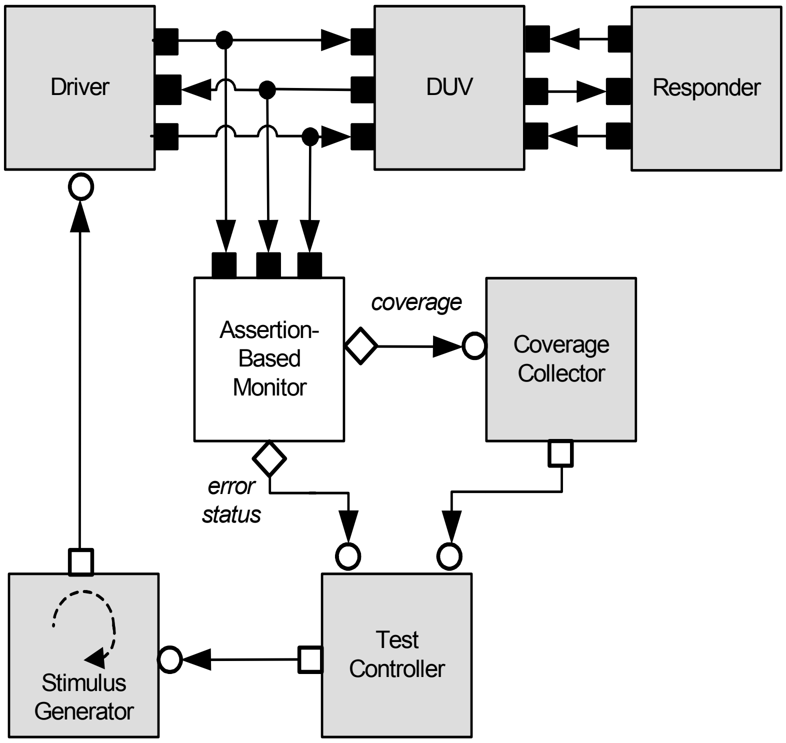
```
import ovm_pkg::*;
import tb_tr_pkg::*; // tb_status_pb class definition

module tb_pipelined_mon (
  interface.monitor_mp monitor_mp
);

  ovm_analysis_port #(tb_status_pb)
                       status_ap = new("status_ap", null);
  ovm_analysis_port #(tb_coverage)
                       coverage_ap = new("coverage_ap", null);

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

5.4 Interface monitor coverage example

Thus far in this chapter, we have demonstrated how to create a set of assertion to detect bus interface errors. In addition to assertions for error detection, assertion-based IP generally contains coverage properties that implement associated coverage models. Each model defines DUV behavior that must be observed to determine completeness. The recorded coverage data is analyzed during verification to gauge verification progress.

In this section, we return to our nonpipelined bus interface example (previously discussed in Section 5.2) to demonstrate the process of adding coverage properties to assertion-based IP. The coverage property we demonstrate tracks the size of a bus read or write burst transaction for our nonpipelined bus interface.

Figure 5-18 Assertion-based IP coverage analysis port



Like our pervious assertion examples, we recommend following the philosophy of separating coverage detection from action. This requires adding an analysis port similar to our assertion-based monitor, which is used to communicate between other testbench verification components (as Figure 5-18 illustrates).

The coverage analysis port broadcasts a *coverage* transaction class with an extension on an `ovm_transaction` base class, as Figure 5-18 illustrates. The `tb_coverage` class, which Example 5-11 defines, includes an `enum` that identifies the various types of nonpipelined bus errors, and a set of methods to set the appropriate transaction type and capture the address and data values for analysis.

Example 5-11 Coverage transaction class

```
class tb_coverage extends ovm_transaction;

  typedef enum {IDLE, WRITE, READ} bus_trans_t;

  rand bus_trans_t  bus_trans_type;
  . . . .

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

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

  . . .

endclass
```

Example 5-12 demonstrates our modified nonpipelined assertion-based monitor that includes the coverage analysis port and a coverage property to measure burst sizes.

The `psize` local variable, defined within the sequence of the coverage property, increments each time a new data word is transferred for a given bus transaction burst. The burst *size* and *type* (for example, read or write), is passed to the `build_tr` function, which is responsible for broadcasting the burst size and type coverage information through the assertion-based monitor's coverage analysis port.

Example 5-12 Encapsulate coverage properties inside a module

```
import ovm_pkg::*;
import tb_tr_pkg::*; // tb_status_nb class definition

module tb_nonpipelined_mon (
  interface.monitor_mp monitor_mp
);

  ovm_analysis_port #(tb_status_nb) status_ap =
                                   new("status_ap", null);
  ovm_analysis_port #(tb_coverage) coverage_ap =
                                   new("coverage_ap", null);
  tb_status_nb status;
  . . .
 // Add modeling code here as described in Section 5.2.4
  . . .

 // All assertions and coverage properties here

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

  cover property (p_burst_size);

  function void build_tr(int psize);
    tb_transaction tr;
    tr = new();
    if (bus_write) begin
      tr.set_write();
      tr.data = bus_wdata;
    end
    else begin
      tr.set_read();
      tr.data = bus_rdata;
    end
    tr.burst_count = psize;
    tr.addr        = bus_addr;
    coverage_ap.write(tr);
  endfunction
endmodule
```

5.5 Summary

In this chapter, we demonstrated our *process* of creating assertion-based IP for the three generic bus interfaces Figure 5-1 illustrates:

- Simple serial bus interface
- Simple nonpipelined bus interface
- Simple pipelined bus interface

In addition, we demonstrated how to include coverage properties as part of your overall assertion-based IP development strategy.

When creating assertion-based IP, we recommend that you separate the process of error (or coverage) *detection* from the process of taking *action*. For example, the decision about taking action when an assertion fires is not made within our module-based assertion monitor. When an assertion fires, it calls a method from the transaction error `status` class within the failing assertion's action block to uniquely identify the error. Error status is then passed to the simulation controller via the module-based monitor's analysis port. The simulation controller is responsible for taking an appropriate action or ignoring the error if it chooses. This clear separation between detection and action facilitates reuse for the module-based monitor and enables us to support advanced testbench features (such as error injection) without modifying the monitor.