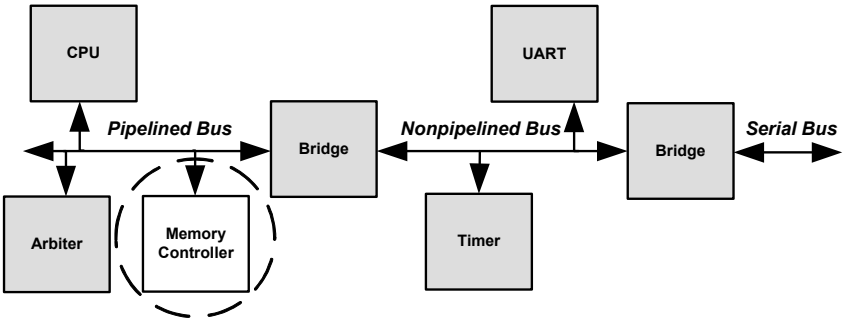
CHAPTER

7

CONTROLLERS

While busses serve as the framework for today's platform-based SoC designs, it is controllers that provide the operational brains. Controllers span the spectrum of design, from lower-level control units embedded in complex design components (such as a simple one-hot encoded) to very complex controllers (such as the memory controller illustrated in the bus-based design example in Figure 7-1).

Figure 7-1. Controller example within a bus-based design



This chapter demonstrates our *process* of creating assertion-based IP for a simple memory controller. We ask you to focus on the techniques for creating the assertion-based IP. By using our generic, simple memory-controller design, we hope that you will be able to set aside the details of a specific type of memory controller design that could

otherwise overwhelm you and distract you from the learning objectives. After learning the process, you should be able to extend these ideas to create assertion-based verification IP for your own specific memory controller.

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.

7.1 Simple generic memory controller

Specifications for commercially available memories, such as an SDR or DDR SDRAM, generally describe the required behavior for the memory's controller in terms of a conceptual state-machine that defines the legal sequences of memory commands. In Chapter 5, "Interfaces," we demonstrated the process of creating a conceptual state-machine to describe legal bus operations. Developing a conceptual state-machine for a memory controller follows a similar process and is fairly straight forward. This process consists of the following steps:

1. Define the legal memory controller interface commands
2. Map conceptual states of the memory controller to specific output commands
3. Define the legal command sequences (that is, transitions for the conceptual state-machine)

This section introduces a simple memory controller to demonstrate the process of creating assertion-based IP.

7.1.1 Block diagram interface description

Figure 7-2 illustrates a block diagram for a system containing our simple memory controller.

Our simple memory controller is designed to support all transactions for a generic SDRAM, with the goal of minimizing memory wait states.

Figure 7-2 Block diagram for a memory controller system

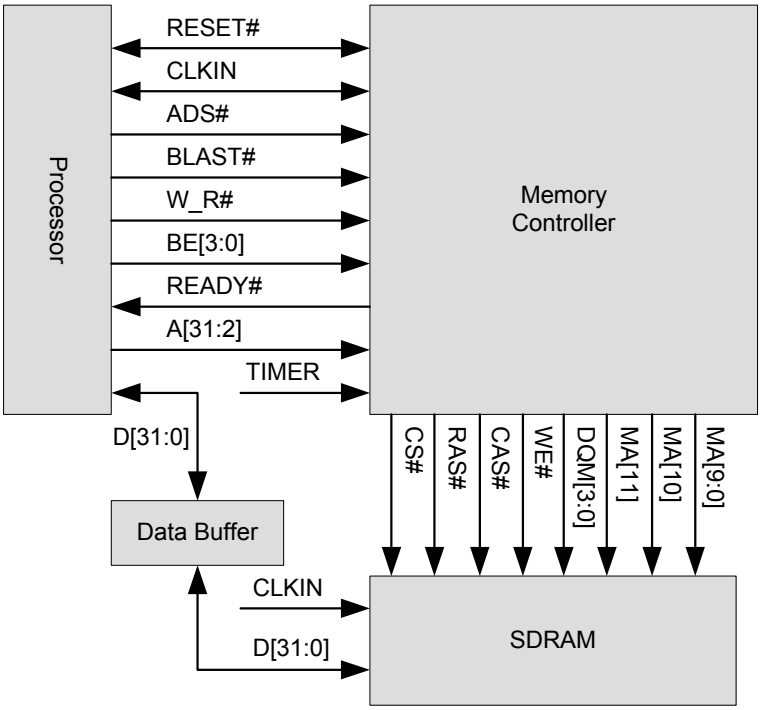


Table 7-1 provides a summary of the signals for the memory controller example.

Table 7-1 Memory controller signal description

Name	Connection	Description
CLKIN	Processor to Controller	External clock
A[31:2]	Processor to Controller	Address
W_R#	Processor to Controller	Write/read strobe

Name	Connection	Description
BLAST#	Processor to Controller	Last transfer of a burst
ADS#	Processor to Controller	Address strobe
BE(3:0)	Processor to Controller	Byte enables decoded from A[2:0]
RESET#	Processor to Controller	System reset
TIMER#	Timer to Controller	Refresh timeout
READY#	Controller to Processor	Data valid
RAS#	Controller to SDRAM	Row address strobe
CAS#	Controller to SDRAM	Column address strobe
WE#	Controller to SDRAM	Write enable
CS#	Controller to SDRAM	Chip select
MA[11:0]	Controller to SDRAM	Multiplexed row/col address lines
DQM[3:0]#	Controller to SDRAM	Data output mask lines
D[31:0]	SDRAM to/from Processor	Data

7.1.2 Overview description

We begin by introducing the basic operations supported by our simple memory controller, which consists of the following transactions:

- Single Read Cycle
- Burst Read Cycle
- Single Write Cycle
- Burst Write Cycle

Most vendors provide an SDRAM specification that contains an interface command truth table and state transition table, which describes the normal operation of the SDRAM. Assume, for our example, that the SDRAM interface command truth table is defined as shown in

Table 7-2, which is a mapping of output controls to command names.[1] To simplify the example, and without loss of generality, we have abstracted the details for the DQM lines, the power on initialization sequence, the mode register set (MRS) sequence, and other potential states generally used to describe specific details related to memory access timing. After understanding this example, you should be able to extend the process ideas we present in this section to your design to include additional features.

Table 7-2 Memory controller to SDRAM command table

Name	CS#	RAS#	CAS#	WE#	Operation
O_DSEL	1	X	X	X	General deselect
O_IDLE	1	1	1	1	Idle
O_NOP	0	1	1	1	No operation
O_READ	0	1	0	1	Read operation
O_WRITE	0	1	0	0	Write operation
O_ACT	0	0	1	1	Row activating
O_PRE	0	0	1	0	Precharge
O_REF	0	0	0	1	Refresh

In Table 7-3 we list the processor to memory controller commands (that is, a mapping of input controls to command names).

Table 7-3 Processor to memory controller command table

Name	ADS#	W_R#	Operation
I_RD_INIT	0	0	Initiate read
I_WR_INIT	0	1	Initiate write
I_READ	X	0	Continue read
I_WRITE	X	1	Continue write
I_IDLE	1	X	Idle

1. Each vendor's SDRAM specification is unique. However, you will find that each vendor provides a truth table describing the SDRAM interface commands, similar to our generic example.

Contained within our simple memory controller is an eleven-bit page detection tag register, which is used to determine a memory address page hit for either a read or write transaction. For this example, upon detecting a page hit, we assume that the controller's internal HIT# signal is asserted. In addition, we assume that our design has an internal signal DONE#, which indicates the completion of the memory transaction. DONE# is asserted low in the same cycle as BLAST#, and it remains asserted low until the cycle after READY# is asserted low. Finally, when the refresh timer times out (TIMER asserted for a single clock cycle), the controller's internal TIM# is asserted low until the refresh is serviced (an O_REF command is issued to initiate a refresh). In general, you might be tempted to exactly replicate this logic in your verification IP (to eliminate the need to reference any internal state of the memory controller). However, this approach misses an important point of functional verification: the source of design intent must be independently referenced and implemented by the verification environment to serve as a golden reference. An alternative approach would be to interpret the spec and separately implement logic that identifies a hit, done, blast, and timeout condition. Due to space limitations, we have decided to reference these three internal signals to simplify our example.

Table 7-4 defines our memory controller legal state transitions.

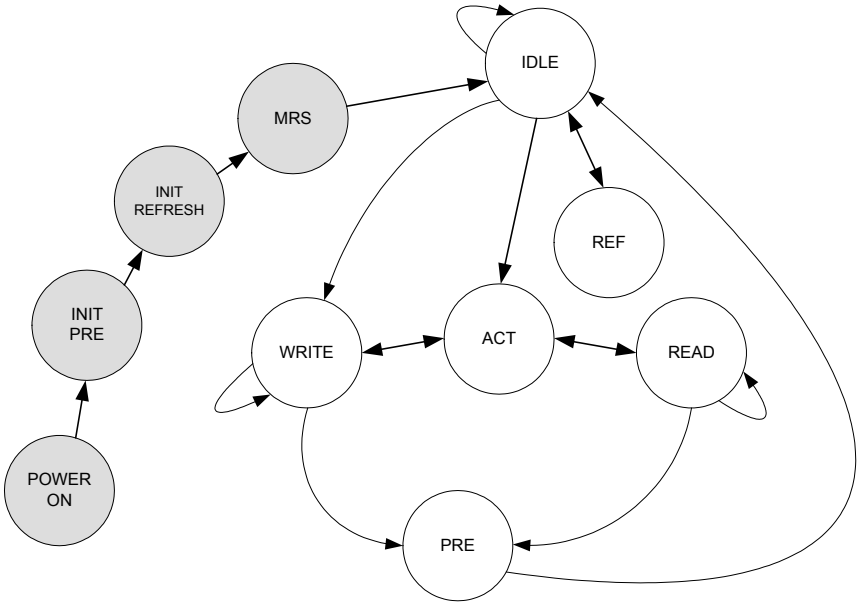
Table 7-4 Memory controller conceptual state transitions

Current State	Input Memory Controller Command (see Table 7-3)	HIT#	DONE#	TIM#	Next State	READY#	Output SDRAM Command (see Table 7-2)
IDLE	X	X	X	0	REF	1	O_NOP
IDLE	I_IDLE	X	X	1	IDLE	0	O_IDLE
IDLE	I_WR_INIT	0	1	X	WRITE	0	O_WRITE
IDLE	I_WR_INIT	1	X	X	ACT	1	O_ACT

Current State	Input Memory Controller Command (see Table 7-3)	HIT#	DONE#	TIM#	Next State	READY#	Output SDRAM Command (see Table 7-2)
IDLE	I_RD_INIT	0	1	X	READ	0	O_READ
IDLE	I_RD_INIT	1	X	X	ACT	1	O_ACT
WRITE	I_WRITE	0	1	X	WRITE	0	O_WRITE
WRITE	I_WRITE	1	X	X	ACT	1	O_ACT
WRITE	I_WRITE	0	0	X	PRE	1	O_PRE
READ	I_READ	0	1	X	READ	0	O_READ
READ	I_READ	1	X	X	ACT	0	O_ACT
READ	I_READ	0	0	X	PRE	1	O_PRE
REF	X	X	X	X	IDLE	1	O_DSEL
PRE	I_IDLE	X	1	X	IDLE	0	O_IDLE
ACT	I_WRITE	0	X	X	WRITE	0	O_WRITE
ACT	I_READ	0	X	X	READ	0	O_READ

The state diagram for our SDRAM memory sequences is shown in Figure 7-3 on page 152. All memory sequences are initiated when ADS# is asserted low, which is sampled on the rising edge of CLKIN. When the RESET# signal is asserted low (during power-on), the processor is in its reset state. The SDRAM controller issues a precharge command (O_PRE) to all banks, eight auto-refresh commands (O_REF), and a mode register set command. When RESET# is released, the controller is in its IDLE state waiting for the processor to initiate a memory transaction.

Figure 7-3 Memory controller conceptual state-machine

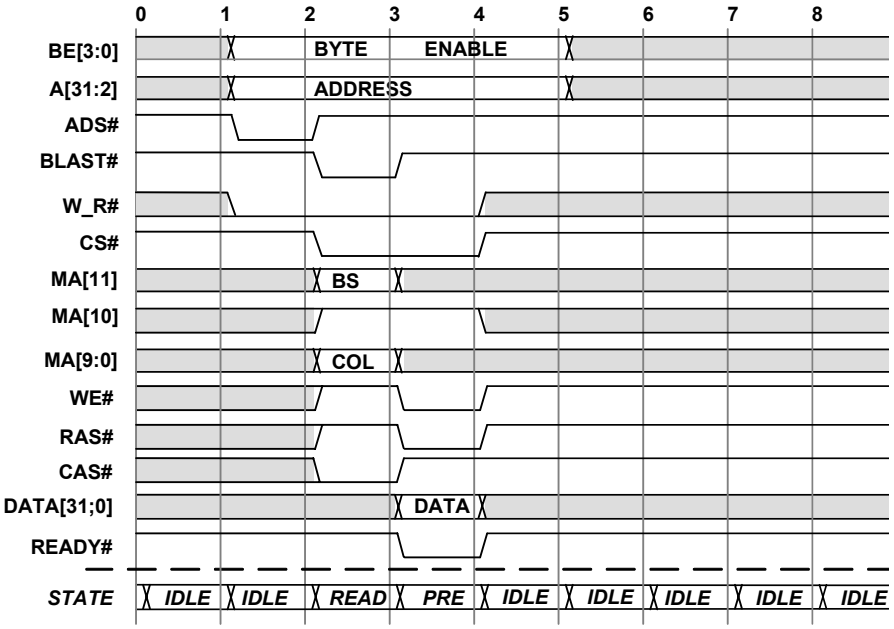


To simplify the discussion, we have decided to ignore the details for the power-on sequence (which is illustrated by the grey circles in Figure 7-3), as well as the configuration details of the memory controller mode register. In addition, we ignore all timing and latency considerations and assume all transitions complete in a cycle. For a real memory controller, you will need to consider these additional details, which are unique to a particular vendor's SDRAM, when creating your memory controller conceptual state-machine.

Our example SDRAM control sequences are defined as follows:

- **Refresh sequence.** The refresh sequence is initiated when the external timer pulses high every 32 ms. Any memory cycles that are in progress will be completed prior to beginning the refresh sequence.
- **Idle state.** When no read, write, or refresh transaction is in progress, the memory controller remains in an idle state.

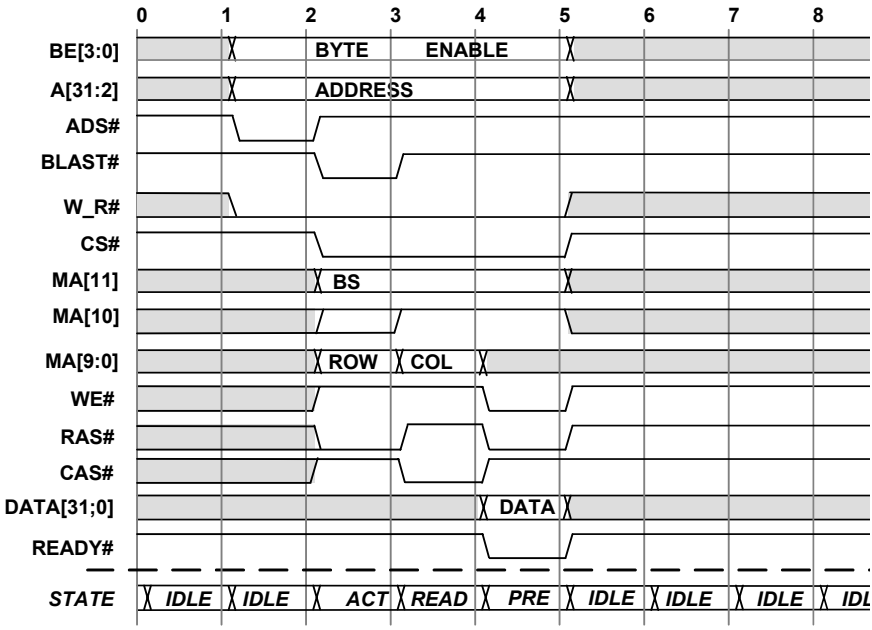
Figure 7-4 Single read hit sequence



- **Single read hit sequence.** When the processor issues an I_RD_INIT command (see Table 7-3), and HIT# is asserted low (the internal page tag register for the bank currently accessed matches the requested row address), a single read hit sequence begins as shown in Figure 7-4. First, an O_READ command (see Table 7-2) is issued from the memory controller to the SDRAM. On the following cycle, the SDRAM places the data on the bus, and the memory controller alerts the processor by asserting READY#. The memory controller terminates the transaction on the following cycle by issuing a O_PRE precharge command to the SDRAM.
- **Single read miss sequence.** When the processor issues an I_RD_INIT command (see Table 7-3), and HIT# is de-asserted (the internal page tag register for the bank currently accessed does not match the requested row address), a single read miss sequence begins as shown in Figure 7-5. The memory controller de-asserts the READY# signal to pause the processor until the data is valid. Then the memory controller issues an O_ACT

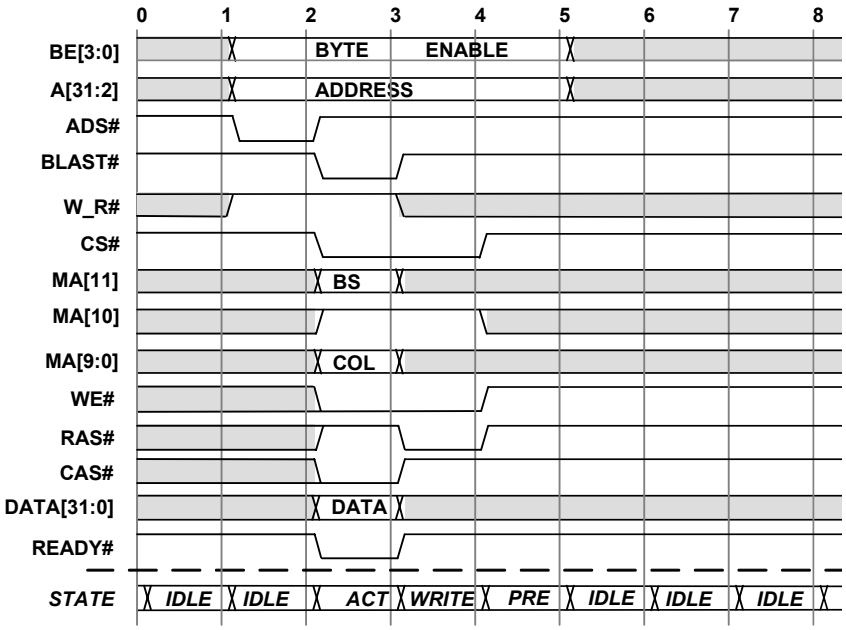
command (see Table 7-2) to the SDRAM. Next, an O_READ command is issued from the memory controller to the SDRAM. On the following cycle, the SDRAM places the data on the bus, and the memory controller alerts the processor by asserting READY# low. The memory controller terminates the transaction on the following cycle by issuing an O_PRE precharge command to the SDRAM.

Figure 7-5 Single read miss sequence



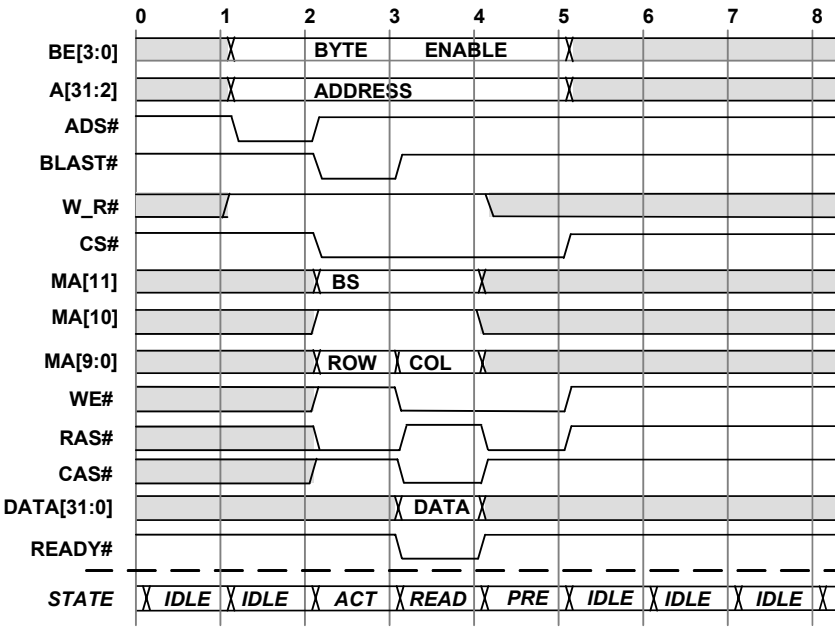
- **Single write hit sequence.** When the processor issues an I_WR_INIT command (see Table 7-3), and HIT# is asserted low (the internal page tag register for the bank currently accessed matches the requested row address), a single write hit sequence begins as shown in Figure 7-6. An O_WRITE command (see Table 7-2) is issued from the memory controller to the SDRAM, while the processor places the data on the bus. The memory controller asserts READY# low, indicating that the SDRAM is ready. Then, the memory controller terminates the transaction on the following cycle by issuing an O_PRE precharge command to the SDRAM.

Figure 7-6 Single write hit sequence



- **Single write miss sequence.** When the processor issues an `I_WR_INIT` command (see Table 7-3), and `HIT#` is de-asserted (the internal page tag register for the bank being accessed does not match the requested row address), a single write miss sequence begins as shown in Figure 7-7. The memory controller de-asserts the `READY#` signal to pause the processor until the data is valid. Then the memory controller issues an `O_ACT` command (see Table 7-2) to the SDRAM. Next, an `O_WRITE` command is issued from the memory controller to the SDRAM, while the processor places the data on the bus. The memory controller asserts `READY#` low indicating that the SDRAM is ready. Then, the memory controller terminates the transaction on the following cycle by issuing a `O_PRE` precharge command to the SDRAM.

Figure 7-7 Single write miss sequence

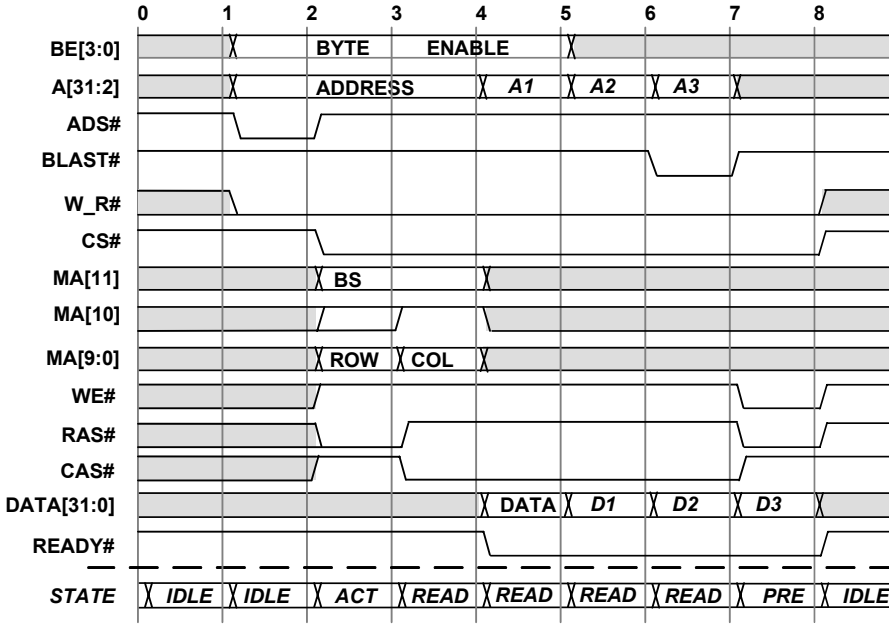


- **Burst read sequence.** When the processor asserts an `I_RD_INIT` command (see Table 7-3), a burst read sequence begins as shown in Figure 7-8.[2] The memory controller de-asserts the `READY#` signal to pause the processor until the data is valid. Then, the memory controller issues an `O_ACT` command (see Table 7-2) to the SDRAM. Next, an `O_READ` command is issued from the memory controller to the SDRAM. On the following cycle, the SDRAM places the first beat of data on the bus, and the memory controller alerts the processor by asserting `READY#` low. The last beat of data is identified by the processor asserting `BLAST#` low with a requesting address to the memory controller. After the final beat of data is read, memory controller terminates the

2. We have simplified the read and write burst sequences for our memory controller example by always forcing row-activating command (O_ACT) prior to starting the memory burst. If required, it is straightforward to modify our example to handle a page hit or miss case, like we did for a single cycle read or write.

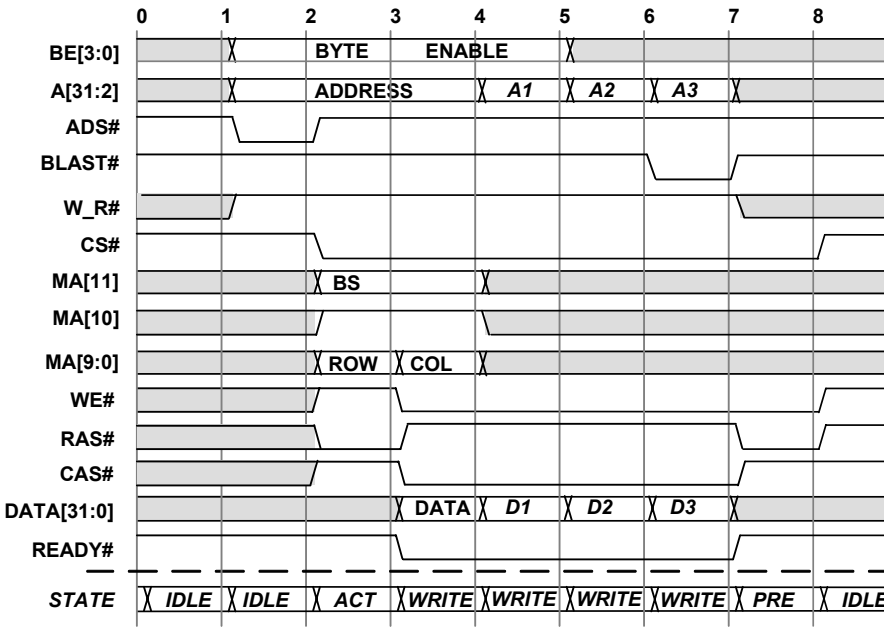
transaction by issuing an O_PRE precharge command to the SDRAM.

Figure 7-8 Burst read sequence



- **Burst write sequence.** When the processor asserts an I_WR_INIT command (see Table 7-3), a burst write sequence begins as shown in Figure 7-9. The memory controller de-asserts the READY# signal to pause the processor until the data is valid. Then, the memory controller issues an O_ACT command (see Table 7-2) to the SDRAM. Next, an O_READ command is issued from the memory controller to the SDRAM. The processor places the first beat of data on the bus, and the memory controller alerts the processor that the SDRAM is ready by asserting READY# low. The last beat of data is identified by the processor asserting BLAST# low with a requesting address to the memory controller. After the final beat of data is written, memory controller terminates the transaction by issuing an O_PRE precharge command to the SDRAM.

Figure 7-9 Burst write sequence



7.1.3 Natural language properties

Before we write our SystemVerilog assertion, our next step is to create a natural language list of properties for our simplified memory controller, as shown in Table 7-5. In terms of creating this organizing table, we recommend listing properties associated with conceptual state transitions followed by specific properties associated with individual output pins.

Table 7-5 Memory controller properties

Assertion Name	Summary
Valid reset transition	
a_reset_idle_trans	The initial state after reset is IDLE
Valid idle transitions	
a_idle_refresh_trans	REFRESH state follows IDLE when TIM# is asserted low
a_idle_idle_trans	IDLE state must follow IDLE when TIM# is de-asserted and no I_RD_INIT or I_WR_INIT command was received
a_idle_write_trans	WRITE state follows IDLE for an I_WR_INIT command, and a page hit occurs (HIT# asserted)
a_idle_read_trans	READ state follows IDLE for an I_RD_INIT command, and a page hit occurs (HIT# asserted low)
a_idle_act_trans	ACT state follow IDLE for either an I_RD_INIT or I_WR_INIT command and a page miss occurs (HIT# de-asserted high)
Valid refresh transition	
a_ refresh_idle_trans	IDLE state follows a REFRESH state
Valid write transitions	
a_write_write_trans	WRITE state follows WRITE state when DONE# is de-asserted, and a page hit occurs (HIT# asserted low)
a_write_act_trans	ACT state follows a WRITE state when DONE# is de-asserted, and a page miss occurs (HIT# asserted high)
a_write_pre_trans	PRE state follows a WRITE state when DONE# is asserted low
Valid read transitions	
a_read_read_trans	READ state follows READ state when DONE# is de-asserted, and a page hit occurs (HIT# asserted low)

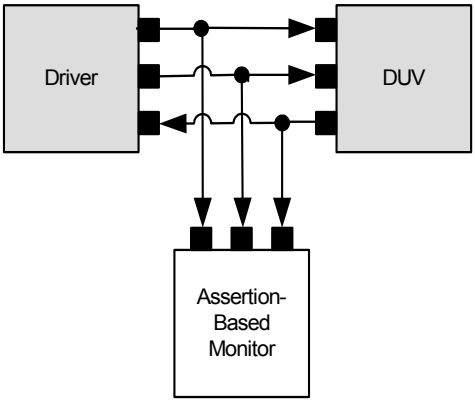
Assertion Name	Summary
a_read_act_trans	ACT state follows a READ state when DONE# is de-asserted, and a page miss occurs (HIT# asserted high)
a_read_pre_trans	PRE state follows a READ state when DONE# is asserted
Valid row activating transitions	
a_act_write_trans	WRITE state follows ACT when W_R# is de-asserted
a_act_read_trans	READ state follows ACT when W_R# is asserted
Valid precharge transition	
a_pre_idle_trans	IDLE state follows PRE state
Processor to memory controller address strobe	
a_ADS_valid	After ADS# is asserted for one cycle, it must not be asserted again until after the completion of the current memory transaction (that is, state IDLE)
Processor to memory controller last transfer strobe	
a_BLAST_valid	Once BLAST# is asserted low for one cycle, it cannot be asserted low again until after an ADS# is asserted
Processor to memory controller write / read signal	
a_W_R_stable	W_R# must remain stable throughout a memory transaction
Processor to memory controller address signals	
a_A_stable	Address must not change values during the ACT state
Processor to memory controller byte enable signals	
a_BE_stable	Byte enable signals must not change values during the ACT state
Memory controller to processor ready signal	
a_READY_low	READY# must only be asserted low during a READ or WRITE state
a_READY_high	READY# must be asserted high when not in a READ or WRITE state

Assertion Name	Summary
Memory controller to SDRAM valid commands	
a_valid_commands	Only valid commands issued from controller

7.1.4 Assertions

To create a set of SystemVerilog assertions for our simple memory controller, we begin by defining the connection between the assertion-based monitor and other potential components in the testbench (such as a driver transactor and the DUV). To accomplish this task, we create a SystemVerilog interface, as illustrated in Figure 7-10.

Figure 7-10 SystemVerilog interface



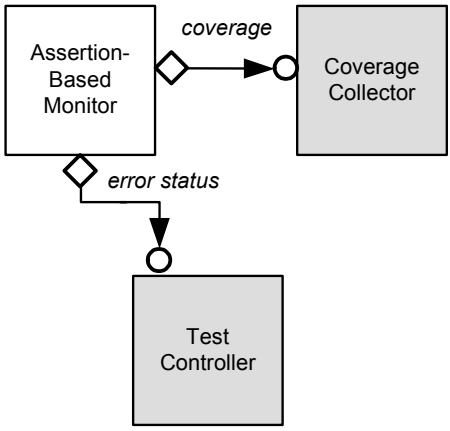
Example 7-1 demonstrates an interface for our simple memory controller example. For this case, the assertion-based monitor references the interface signals in the direction defined by the `monitor_mp` named modport.

Example 7-1 Encapsulate bus signals inside a SV interface

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

  parameter int DATA_SIZE = 32;
  parameter int ADDR_SIZE = 32;
  parameter int BE_SIZE = 4;
  parameter int MA_SIZE = 12;
  parameter int DQM_SIZE = 4;

  bit [ADDR_SIZE-1:2] a;      // A
  bit w_r_n;                  // W_R#
  bit blast_n;                // BLAST#
  bit ads_n;                  // ADS#
  bit [BE_SIZE-1:0] be;       // BE
  bit ready_n;                // READY#

  bit tim_n;                  // TIM#
  bit hit_n;                  // HIT#
  bit done_n;                 // DONE#

  bit ras_n;                  // RAS#
  bit cas_n;                  // CAS#
  bit we_n;                   // WE#
  bit cs_n;                   // CS#
  bit [MA_SIZE-1:0] ma;       // MA
  bit [DQM_SIZE-1:0] dqm_n;   // DQM#
  bit [DATA_SIZE-1:0] d;      // D

  modport driver_mp (
    . . . .
  );

  modport duv_mp (
    . . . .
  );

  modport monitor_mp (
    input       clk , rst ,
    input       a , w_r_n, blast_n , ads_n , be , ready_n ,
    input       tim_n , hit_n , done_n ,
    input       ras_n , cas_n , we_n , cs_n , ma , dqm_n , d
  );

endinterface
```

In addition to pin-level interfaces, we need a means for the memory controller assertion-based monitor to communicate with various analysis verification components within the testbench. Hence, we introduce error status and coverage analysis ports within the monitor, as Figure 7-11 illustrates.

Figure 7-11 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 7-2 defines the *error status* transaction class, which is an extension on an `ovm_transaction` base class. The `tb_status` class includes an `enum` that identifies the various types of memory controller errors and a set of methods to uniquely identify the specific error it detected.

Example 7-2 Error status class

```
class tb_status_mc extends ovm_transaction;

  typedef enum { ERR_RESET_IDLE_TRANS ,
                 ERR_IDLE_REFRESH_TRANS ,
                 ERR_IDLE_IDLE_TRANS ,
                 ERR_IDLE_WRITE_TRANS ,
                 ERR_IDLE_READ_TRANS ,
                 ERR_IDLE_ACT_TRANS ,
                 ERR_ REFRESH_IDLE_TRANS ,
                 ERR_WRITE_WRITE_TRANS ,
                 ERR_WRITE_ACT_TRANS ,
                 ERR_WRITE_PRE_TRANS ,
                 ERR_READ_READ_TRANS ,
                 ERR_READ_ACT_TRANS ,
                 ERR_READ_PRE_TRANS ,
                 ERR_ACT_WRITE_TRANS ,
                 ERR_ACT_READ_TRANS ,
                 ERR_PRE_IDLE_TRANS ,

// Continued on next page
```

Example 7-2 Error status class

```
                ERR_ADS_VALID ,
                ERR_BLAST_VALID ,
                ERR_W_R_STABLE ,
                ERR_ERR_STABLE ,
                ERR_BE_STABLE ,
                ERR_READY_LOW ,
                ERR_READY_HIGH ,
                ERR_VALID_COMMANDS } mc_status_t;

mc_status_t   mc_status;

  function void set_err_reset_idle_trans;
    mc_status = ERR_RESET_IDLE_TRANS;
  endfunction

  function void set_err_idle_refresh_trans;
    mc_status = ERR_IDLE_REFRESH_TRANS;
  endfunction

  function void set_err_idle_idle_trans;
    mc_status = ERR_IDLE_IDLE_TRANS;
  endfunction

  function void set_err_idle_write_trans;
    mc_status = ERR_IDLE_WRITE_TRANS;
  endfunction

 . . . .

  function void set_err_valid_commands;
    mc_status = ERR_VALID_COMMANDS;
  endfunction

. . .

endclass
```

To simplify writing the assertion (and to increase the clarity), we create some modeling code (see Example 7-3) to map the interface signals defined in Table 7-2 and Table 7-3 (see page 149) into SDRAM and memory controller commands.

Example 7-3 Modeling to map control signals to conceptual states

```
module tb_mem_cntrl_mon(
  interface.monitor_mp monitor_mp
);
   ovm_analysis_port #(tb_status) status_ap = new("status_ap", null);

 // Continued on next page
```

Example 7-3 Modeling to map control signals to conceptual states

```
  . . .
  bit i_wr_init  = ~monitor_mp.ads_n &  monitor_mp.w_r_n;
  bit i_rd_init  = ~monitor_mp.ads_n & ~monitor_mp.w_r_n;
  bit i_read     = ~monitor_mp.w_r_n;
  bit i_write    =  monitor_mp.w_r_n
  bit i_idle     =  monitor_mp.ads_n;

  bit o_dsel     =  monitor_mp.cs_n;
  bit o_idle     =  monitor_mp.cs_n  &  monitor_mp.ras_n &
                    monitor_mp.cas_n &  monitor_mp.we_n;
  bit o_nop      = ~monitor_mp.cs_n  &  monitor_mp.ras_n &
                    monitor_mp.cas_n &  monitor_mp.we_n;
  bit o_read     = ~monitor_mp.cs_n  &  monitor_mp.ras_n &
                   ~monitor_mp.cas_n &  monitor_mp.we_n;
  bit o_write    = ~monitor_mp.cs_n  &  monitor_mp.ras_n &
                   ~monitor_mp.cas_n & ~monitor_mp.we_n;
  bit o_act      = ~monitor_mp.cs_n  & ~monitor_mp.ras_n &
                    monitor_mp.cas_n &  monitor_mp.we_n;
  bit o_pre      = ~monitor_mp.cs_n  & ~monitor_mp.ras_n &
                    monitor_mp.cas_n & ~monitor_mp.we_n;
  bit o_ref      = ~monitor_mp.cs_n  & ~monitor_mp.ras_n &
                   ~monitor_mp.cas_n &  monitor_mp.we_n;

  bit o_error    = ~monitor_mp.cs_n  & ~monitor_mp.ras_n &
                   ~monitor_mp.cas_n & ~monitor_mp.we_n
              ||   ~monitor_mp.cs_n  &  monitor_mp.ras_n &
                    monitor_mp.cas_n & ~monitor_mp.we_n;

  . . .
endmodule
```

In our simple memory controller example, we followed the signal naming convention you would encounter when reading a typical SDRAM specification. That is, active low signals in the specification are denoted with a "#" character at the end of the signal name. During the implementation process, engineers typically convert to an implementation naming convention when denoting an active low signal. In our SystemVerilog examples, we follow an implementation naming convention in which the "_n" characters are added to the end of all active low signal names.

We are now ready to write assertions for the memory controller properties defined in Table 7-5 (see page 159). Our first property demonstrated in Assertion 7-1 specifies the valid reset transition.

Assertion 7-1 Memory controller reset transition

```
property p_reset_idle_trans;
  @(posedge monitor_mp.clk)
    $fell(monitor_mp.rst) |-> o_idle;
endproperty
a_reset_idle_trans:
  assert property (p_reset_idle_trans) else begin
    status = new();
    status.set_err_reset_idle_trans();
    status_ap.write(status);
  end
```

Assertion 7-2 specify the valid idle state transitions, as defined by our conceptual state-machine.

Assertion 7-2 Memory controller idle transitions

```
property p_idle_refresh_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_idle & ~tim_n |=> o_refresh;
endproperty
a_reset_idle_trans:
  assert property (p_idle_refresh_trans) else begin
    status = new();
    status.set_err_idle_refresh_trans();
    status_ap.write(status);
  end

property p_idle_idle_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_idle & tim_n & i_idle |=> o_idle;
endproperty
a_idle_idle_trans:
  assert property (p_idle_idle_trans) else begin
    status = new();
    status.set_err_idle_idle_trans();
    status_ap.write(status);
  end

property p_idle_write_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_idle & tim_n & i_wr_init & ~hit_n |=> o_write;
endproperty
a_idle_write_trans:
  assert property (p_idle_write_trans) else begin
    status = new();
    status.set_err_idle_write_trans();
    status_ap.write(status);
  end

// Continuned on next page
```

Assertion 7-2 Memory controller idle transitions

```
property p_idle_read_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_idle & tim_n & i_rd_init & ~hit_n |=> o_read;
endproperty
a_idle_read_trans:
  assert property (p_idle_read_trans) else begin
    status = new();
    status.set_err_idle_read_trans();
    status_ap.write(status);
  end

property p_idle_act_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_idle & tim_n & ~i_idle & hit_n |=> o_act;
endproperty

a_idle_act_trans:
  assert property (p_idle_act_trans) else begin
    status = new();
    status.set_err_idle_act_trans();
    status_ap.write(status);
  end
```

Assertion 7-3 specifies the valid refresh state transition, as defined by our conceptual state-machine.

Assertion 7-3 Memory controller refresh transitions

```
property p_refresh_idle_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_refresh |=> o_idle;
endproperty
a_refresh_idle_trans:
  assert property (p_refresh_idle_trans) else begin
    status = new();
    status.set_err_refresh_idle_trans();
    status_ap.write(status);
  end
```

Assertion 7-4 specify the valid write state transitions, as defined by our conceptual state-machine.

Assertion 7-4 Memory controller write transitions

```
property p_write_write_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_write & ~hit_n & done_n |=> o_write;
endproperty
a_write_write_trans:
  assert property (p_write_write_trans) else begin
    status = new();
    status.set_err_write_write_trans();
    status_ap.write(status);
  end

property p_write_act_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_write & hit_n & done_n |=> o_act;
endproperty
a_write_act_trans:
  assert property (p_write_act_trans) else begin
    status = new();
    status.set_err_write_act_trans();
    status_ap.write(status);
  end

property p_write_pre_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_write & ~done_n |=> o_pre;
endproperty
a_write_pre_trans:
  assert property (p_write_pre_trans) else begin
    status = new();
    status.set_err_write_pre_trans();
    status_ap.write(status);
  end
```

Assertion 7-5 specify the valid read state transitions, as defined by our conceptual state-machine.

Assertion 7-5 Memory controller read transitions

```
property p_read_read_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_read & ~hit_n & done_n |=> o_read;
endproperty
a_read_read_trans:
  assert property (p_read_read_trans) else begin
    status = new();
    status.set_err_read_read_trans()
    status_ap.write(status);
  end

property p_read_act_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_read & hit_n & done_n |=> o_act;
endproperty
a_read_act_trans:
  assert property (p_read_act_trans) else begin
    status = new();
    status.set_err_read_act_trans();
    status_ap.write(status);
  end

property p_read_pre_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_read & ~done_n |=> o_pre;
endproperty
a_read_pre_trans:
  assert property (p_read_pre_trans) else begin
    status = new();
    status.set_err_read_pre_trans();
    status_ap.write(status);
  end
```

Assertion 7-6 specify the valid row-activating state transitions, as defined by our conceptual state-machine.

Assertion 7-6 Memory controller row-activating transitions

```
property p_act_write_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_act & i_write |=> o_write;
endproperty
a_act_write_trans:
  assert property (p_act_write_trans) else begin
    status = new();
    status.set_err_act_write_trans();
    status_ap.write(status);
  end

property p_act_read_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_act & i_read |=> o_read;
endproperty
a_act_read_trans:
  assert property (p_act_read_trans) else begin
    status = new();
    status.set_err_act_read_trans();
    status_ap.write(status);
  end
```

Assertion 7-7 specifies the valid precharge state transition, as defined by our conceptual state-machine.

Assertion 7-7 Memory controller precharge transitions

```
property p_pre_idle_trans;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    o_pre |=> o_idle;
endproperty
a_pre_idle_trans:
  assert property (p_pre_idle_trans) else begin
    status = new();
    status.set_err_pre_idle_trans();
    status_ap.write(status);
  end
```

Assertion 7-8 specifies the valid behavior for the processor to memory address strobe.

Assertion 7-8 Processor to memory controller address strobe

```
property p_ADS_valid;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    ~ads_n |=> (ads_n throughout o_pre[->1]);
endproperty
a_ADS_valid:
  assert property (p_ADS_valid) else begin
    status = new();
    status.set_err_ADS_valid();
    status_ap.write(status);
  end
```

Assertion 7-9 specifies the valid behavior for the processor to memory last transfer strobe.

Assertion 7-9 Processor to memory controller stable controls

```
property p_BLAST_initial_condition;
  @(posedge monitor_mp.clk)
    monitor_mp.rst |=> blast_n throughout ~ads_n[->1];
endproperty
a_BLAST_initial_condition:
  assert property (p_BLAST_initial_condition) else begin
    status = new();
    status.set_err_BLAST_initial_condition();
    status_ap.write(status);
  end

property p_BLAST_valid;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    ~blast_n |=> blast_n throughout ~ads_n[=1] ##1 ~blast_n;
endproperty
a_BLAST_valid:
  assert property (p_BLAST_valid) else begin
    status = new();
    status.set_err_BLAST_valid();
    status_ap.write(status);
  end
```

The set of properties shown in Assertion 7-10 specify stable behavior for the processor to memory controller `W_R#`

control (w_r_n), A address bits (a), and BE byte enable signals (be) during a memory transfer.

Assertion 7-10 Processor to memory controller last transfer strobe

```
property p_W_R_stable;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    ~i_idle |=> $stable(w_r_n) throughout i_pre[->1];
endproperty
a_W_R_stable:
  assert property (p_W_R_stable) else begin
    status = new();
    status.set_err_W_R_stable();
    status_ap.write(status);
  end

property p_A_stable;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    ~i_idle |=> $stable(a) throughout i_pre[->1];
endproperty
a_A_stable:
  assert property (p_A_stable) else begin
    status = new();
    status.set_err_A_stable();
    status_ap.write(status);
  end

property p_BE_stable;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    ~i_idle |=> $stable(be) throughout i_pre[->1];
endproperty
a_BE_stable:
  assert property (p_BE_stable) else begin
    status = new();
    status.set_err_BE_stable();
    status_ap.write(status);
 end
```

The set of properties shown in Assertion 7-11 specify legal behavior of the memory controller to process READY# signal.

Assertion 7-11 Processor to memory controller ready signal

```
property p_READY_low;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    $past(~ready_n) -> (o_read || o_write);
endproperty
a_READY_low:
  assert property (p_READY_low) else begin
    status = new();
    status.set_err_READY_low();
    status_ap.write(status);
  end

property p_READY_high;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    $past(ready_n) -> (~o_read && ~o_write);
endproperty
a_READY_high:
  assert property (p_READY_high) else begin
    status = new();
    status.set_err_READY_high();
    status_ap.write(status);
  end
```

The final property (shown in Assertion 7-12) specifies that only legal SDRAM commands are issued from the memory controller.

Assertion 7-12 Memory controller to SDRAM valid commands

```
property p_valid_commands;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    ~o_error;
endproperty
a_valid_commands:
  assert property (p_valid_commands) else begin
    status = new();
    status.set_err_valid_commands();
    status_ap.write(status);
  end
```

7.1.5 Encapsulate properties

In this step, we turn our set of related memory controller 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 7-4 Encapsulate properties inside a module

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

module tb_mem_controller_mon (
  interface.monitor_mp monitor_mp
);

  ovm_analysis_port #(tb_status_mc)
                        status_ap = new(“status_ap”, null);
  ovm_analysis_port #(tb_coverage)
                        coverage_ap = new(“coverage_ap”, null);

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

7.2 Summary

In this chapter, we demonstrated our process of creating assertion-based IP for a simple, yet non-trivial, memory controller. The general approach consists of the following steps:

1. Define the legal memory controller interface commands
2. Map conceptual states of the memory controller to specific output commands
3. Define the legal command sequences (that is, transitions for the conceptual state-machine)

Obviously, your particular conceptual state-machine is directly influenced by the memory vendor you have selected for your design.

Once you define the memory controller conceptual state-machine, properties are then easily identified by specifying legal transitions.

When creating assertion-based IP, some engineers prefer to explicitly model the conceptual state-machine as a monitor. An error state is introduced in this model, and a simple assertion is then written that states the conceptual state-machine must never enter the error state. This form of assertion-based IP does have a debugging advantage, since the implementor does not have to manage a large set of assertions normally required to implicitly describe legal conceptual state transitions. However, as the conceptual state-machine grows, or when it is necessary to describe concurrency or overlapping behaviors, a modeling approach to creating verification IP can quickly become overwhelming.

In this chapter, we did not explicitly model the conceptual state-machine in the verification IP. We created properties that specified each valid transition of the conceptual state-machine. Although the focus of this book is on simulation-based use of assertion-based IP, it is worth mentioning that implicit specification of the conceptual states using assertions tends to perform better in formal property checking than introducing a large state-machine as part of the specification. The negative aspect of creating an implicit conceptual state-machine through a large set of assertions is a concern of completeness, as well as potentially introducing contradictions contained within a large set of assertions. In either case, careful review is critical.