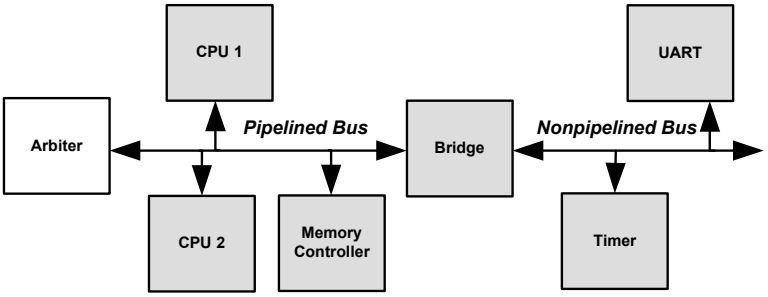
CHAPTER

6

ARBITERS

Arbiters are probably one of the most widely studied components in software and hardware design and verification. In fact, you will find that an arbiter has traditionally served as the primary design example in many published technical papers and books [Kariniemi and Nurmi 2005] [Dasgupta 2006]. Hence, this chapter presents very little new information on the topic of specifying assertions for various arbitration schemes. Yet, arbiters are a fundamental component in systems containing shared resources, such as our bus-based design example illustrated in Figure 6-1 that must arbitrate a shared bus between multiple masters to prevent dropping or corrupting data transported through the bus. Thus, understanding how to create assertion-based IP for an arbiter, which easily integrates with other testbench components, is an essential topic of discussion.

Figure 6-1. Bus-based design with arbiter



This chapter begins with a review of various common arbitration schemes and associated properties. It then demonstrates the process of creating assertion-based IP for an arbiter component.

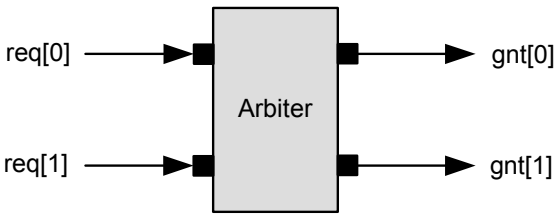
In this chapter, you will note that we are following a standard development pattern previously defined in Chapter 3. Each section within this chapter was defined to stand on its on. Hence, you might noticed repetitive text in portions of the chapter.

6.1 Arbitrations schemes

Arbitration schemes range from the unfair, fixed-priority scheme to the fair, round-robin scheme, as well as a combination of variable priority and fairness schemes. (See [Kariniemi and Nurmi 2005] for a comprehensive discussion on a range of arbitration schemes.) Although arbiters are used in multiple applications, their basic properties are generally straightforward and are easily expressed as SystemVerilog assertions.

Consider the requirement for the arbiter illustrated in Figure 6-2, which states that a requesting client will eventually be served. In other words, we want to ensure that a client cannot be starved from access to a shared resource.

Figure 6-2. A simple two client arbiter



As Assertion 6-1 shows, we can use SystemVerilog to express an assertion for client 0. This assertion states that

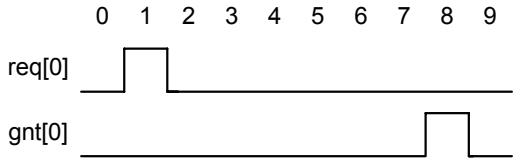
every request (assertion of `req[0]`) must be followed at some time by a grant (assertion of `gnt[0]`).

Assertion 6-1 A requesting client will eventually be served

```
property p_req_gnt;
  @(posedge clk) disable iff (rst)
    req[0] |-> ##[0:$] gnt[0];
endproperty
a_req_gnt: assert property (p_req_gnt);
```

Figure 6-3 illustrates a trace on which Assertion 6-1 holds.

Figure 6-3. Trace on which Assertion 6-1 holds



The next section refines our arbiter assertion to make it more precise and ensure that an arbiter servicing multiple clients is fair.

6.1.1 Fair arbitration

To begin the discussion of how to create assertion-based IP for fair arbitration, we examine the simple two-client static fair arbiter that Figure 6-2 illustrates. For this simple example to be fair, a pending request for a particular client should never have to wait more than two arbitration cycles before it is serviced. Otherwise, the alternate client must have been unfairly issued a grant multiple times. This method of verifying fairness is based on the concept of finitary fairness introduced by [Alur and Henzinger 1998].

For client 0, which asserts a request `req[0]`, we can use a forbidden sequence to express the fair requirement, as

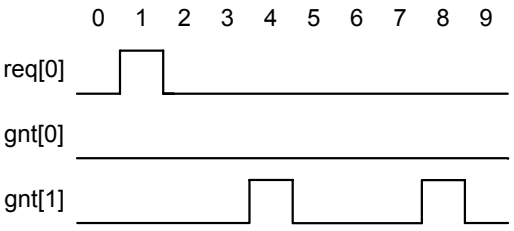
Assertion 6-2 shows. This assertion states that we should never see a sequence where client 0 asserts a request (`req[0]`), and then no grant (`gnt[0]`) is issued within two arbitration cycles. For our example, we use the goto repetition operator `[->2]` to express two back-to-back arbitration cycles for client 1, and the `throughout` operator for the sequence where no grants are issued to client 0 during the two arbitration cycles for client 1.

Assertion 6-2 Two-client fair arbitration assertion for client 0

```
property p_no_req_0_two_gnt_1;
  @(posedge clk) disable iff (rst)
    req[0] ##1 (!gnt[0] throughout (gnt[1])[->2]) |-> 0;
endproperty
a_no_req_0_two_gnt_1: assert property (p_no_req_0_two_gnt_1);
```

Figure 6-4 illustrates behavior on which Assertion 6-2 does not hold.

Figure 6-4. Trace on which Assertion 6-2 does not hold



Assertion 6-3 shows a fair arbitration requirement for the case where client 1 asserts a request (`req[1]`).

Assertion 6-3 Two-client fair arbitration assertion for client 1

```
property p_no_req_1_two_gnt_0;
  @(posedge clk) disable iff (rst)
    req[1] ##1 (!gnt[1] throughout (gnt[0])[->2]) |-> 0;
endproperty
a_no_req_1_two_gnt_0: assert property (p_no_req_1_two_gnt_0);
```

For a large fair arbiter containing many clients (for example, an eight-client round-robin arbiter), we can simplify the process of writing a large set of assertions for each request-and-grant pair by using the SystemVerilog `generate` construct (see Assertion 6-4).

Assertion 6-4 Eight-client fair arbitration assertion

```
property p_no_req_i_two_gnt_j (i,j);
  @(posedge clk) disable iff (rst)
    req[i] ##1 (!gnt[i] throughout (gnt[j])[->2]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
      for (j = 0; j<=7; j++) begin
        assert property (p_no_req_i_two_gnt_j(i,j));
      end
    end
  end
endgenerate
```

6.1.2 Specific arbiter properties

In addition to the general fair arbitration assertion we previously discussed, there are often additional specific arbiter assertion properties that you will want to consider when creating your set of arbiter assertions [Dasgupta 2006]. For example:

- Is there a minimum time interval after a request when a grant must not occur due to latency considerations within the arbiter design?
- Does the interface support queuing multiple requests on the same port while waiting for a grant?

This section illustrates a few specific arbiter interface properties through examples. Many of the properties we discuss in this section apply to all arbiter designs and

arbitration schemes (for example, mutually exclusive grants), while other properties are unique and specific to a particular arbiter implementation (for example, interface handshake properties). Although your specific arbiter properties may differ from the examples in this section, our intent is to provide a starting point with multiple examples that you can modify to fit your specific needs.

Mutually exclusive grants

For our simple two-client fair arbiter (illustrated in Figure 6-2), it is a requirement that the arbiter's grants remain mutually exclusive to prevent resource conflicts when accessing a shared resource. In other words, the arbiter should never assert `gnt[0]` and `gnt[1]` at the same time. Assertion 6-5 shows a SystemVerilog assertion for this requirement.

Assertion 6-5 Mutually exclusive grants

```
property p_mutex_gnt;
  @(posedge clk) disable iff (rst)
    !(gnt[0] & gnt[1]);
endproperty
a_mutex_gnt: assert property (p_mutex_gnt);
```

Assertion 6-6 shows an alternative way to express this assertion using the SystemVerilog system function `$onehot0()`. The `$onehot0()` built-in function checks that the bits of `gnt` (`gnt[0]` and `gnt[1]`) are mutually exclusive and returns true when there is at most one bit with the value one. Otherwise, it returns false. In fact, for an arbiter that supports a large number of clients, using the `$onehot0()` system function simplifies the process of writing a mutually exclusive assertion.

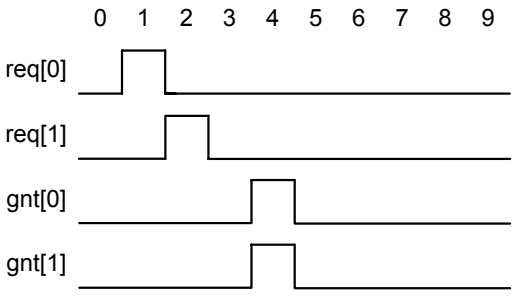
Assertion 6-6 Built-in function to check mutually exclusive grants

```
property p_mutex_gnt;
  @(posedge clk) disable iff (rst)
    $onehot0(gnt);
endproperty
a_mutex_gnt: assert property (p_mutex_gnt);
```

Trace on which Assertion 6-7 does not hold

Figure 6-5 illustrates behavior on which Assertion 6-6 does not hold.

Figure 6-5 Trace on which Assertion 6-6 does not hold



Minimum latency

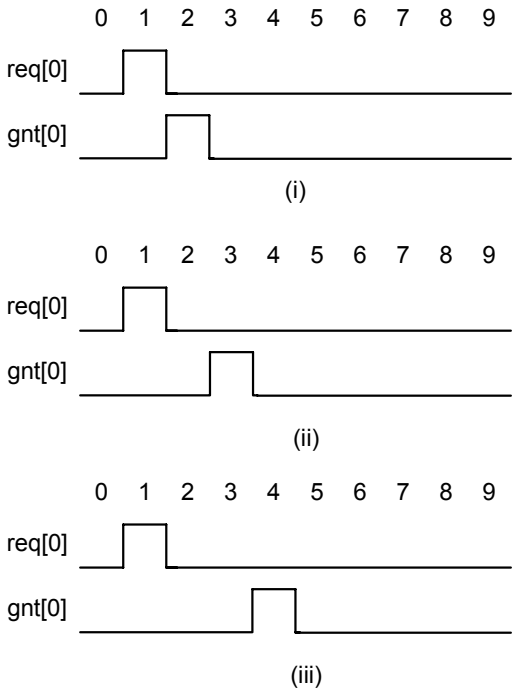
For many arbiter designs, there is a minimum time interval between the point that a request is asserted and a grant is issued. This interval occurs due to latency considerations within the arbiter design. Obviously, specifying the minimum time requirement between a request and grant in SystemVerilog is design specific. For instance, we will assume for our simple example that there is a minimum latency of three cycles between a request and a grant. Hence, a request is identified by a rising transition of the `req[0]` signal. Assertion 6-7 expresses an assertion for this requirement.

Assertion 6-7 Forbidden occurrences of grants due to latency

```
property p_minimum_latency
  @(posedge clk) disable iff (rst)
    $rose(req[0]) ##[0:2] gnt[0] |-> 0;
endproperty
a_minimum_latency:
  assert property (p_minimum_latency);
```

Assertion 6-7 specifies three arbiter forbidden sequences in which the grant must not be issued sooner than the latency through the arbiter (for this example, less than three cycles). Figure 6-6 illustrates traces where Assertion 6-7 does not hold.

Figure 6-6. Traces on which Assertion 6-7 does not hold



Grant without pending request

One fundamental requirement of an arbiter is that it should never issue a grant to any client that has not asserted a request. For this requirement, we need to consider the following cases:

1. A client has never asserted a request in the past (that is, an initial condition after reset).
2. An arbitration cycle associated with a client has completed in the past.

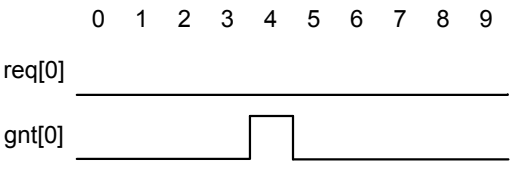
We can express a SystemVerilog assertion for the initial condition case as shown in Assertion 6-8, which specifies that a `!gnt[0]` holds up to and including the cycle in which the first `req[0]` holds—since we are assuming for our example that there is at minimum a one-cycle latency between the time in which a request is asserted and a grant is issued.

Assertion 6-8 Initial condition case for no grant prior to request

```
property p_init_no_gnt_before_req;
  @(posedge clk)
   $fell(rst) |-> (~gnt[0] throughout (req[0])[->1]);
endproperty
a_init_no_gnt_before_req:
  assert property (p_init_no_gnt_before_req);
```

Figure 6-7 illustrates behavior on which Assertion 6-8 does not hold.

Figure 6-7. Trace on which Assertion 6-8 does not hold



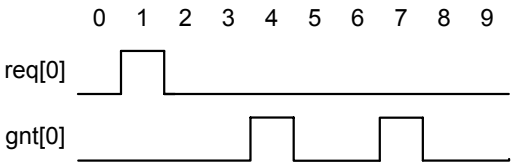
Assertion 6-9 shows how to write a SystemVerilog assertion for the general case in which an arbitration cycle associated with client 0 has completed in the past.

Assertion 6-9 General case for no grant prior to a request

```
property p_no_gnt_before_req;
  @(posedge clk) disable iff (rst)
   gnt[0] |=> (~gnt[0] throughout (req[0])[->1]);
endproperty
a_no_gnt_before_req:
  assert property (p_no_gnt_before_req);
```

Figure 6-8 illustrates behavior on which Assertion 6-9 does not hold. Notice the situation where a grant is issued at clock four, which ends the arbitration cycle started at clock one, and another grant is issued at clock seven without a pending request.

Figure 6-8. Trace on which Assertion 6-9 does not hold



6.1.3 Fixed priority

Section 6.1.1 described how to express assertions associated with fair arbiters. Fairness, however, is only one example in the class of static arbitration schemes commonly used in today's designs. This section describes how to express assertions for *fixed priority*, which is one of the most common static arbitration schemes. An arbiter that follows a fixed priority scheme always issues a grant to a requesting, higher-priority client before it issues a grant to a lower-priority client.

Figure 6-2 illustrated a simple two-client arbiter. For our discussion, we will assume that client 0 has a higher priority over client 1, and there is a minimum one-cycle latency through the arbiter.

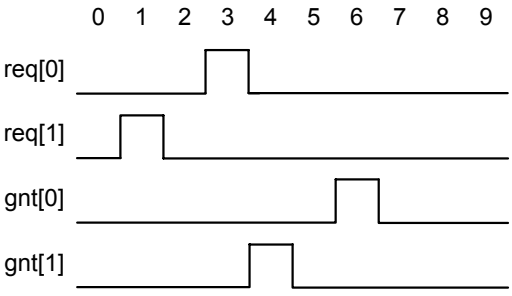
Assertion 6-10 Client 0 has fixed priority over client 1

```
//  req[0] higher priority over req[1], gnt[0] before gnt[1]

property p_fixed_priority_client0;
 @(posedge clk) disable iff (rst)
  $rose(req[0]) ##1 (~gnt[0] throughout (gnt[1])[->1]) |-> 0;
endproperty

a_fixed_priority_client0:
  assert property (p_fixed_priority_client0);
```

Assertion 6-10 expresses a fixed priority assertion for our simple example *assuming a latency of one cycle*.

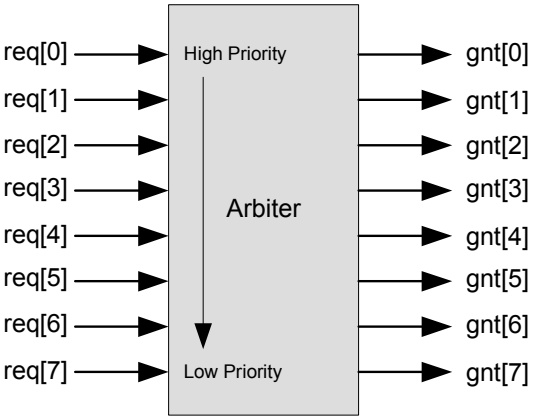
This assertion states that whenever client 0 asserts a request (`req[0]`), then a grant (`gnt[0]`) must be issued to the higher-priority client 0 prior to issuing a grant (`gnt[1]`) to the lower-priority client 1. Figure 6-9 illustrates a behavior on which Assertion 6-10 does not hold.

Figure 6-9. Trace on which Assertion 6-10 does not hold



In Figure 6-9, there is a pending request from client 1 that has yet to be serviced when a higher priority request comes from client 0. Hence, we should not see a grant issued for client 1 before a grant is issued for client 0 (assuming a one-cycle latency through the arbiter for this example).

Figure 6-10. Eight-client fixed priority arbiter



Let us assume we have a larger eight-client arbiter, where the lowest numbered port has the highest priority, and the priority decreases as the port number increases, as illustrated in Figure 6-10.

We can simplify the process of writing a large set of assertions for each request and grant pair (as we did in Assertion 6-4) by using the SystemVerilog `generate` construct as shown in Assertion 6-11.

Assertion 6-11 Eight-client fixed priority assertion

```
// Assume client i is higher priority to client j, when value j > i
// E.g., client i=0 higher priority than client j=1, for value j > i

property p_fixed_priority_i_j(i,j);
 @(posedge clk) disable iff (rst)
       (j > i) && $rose(req[i])
  ##1 (~gnt[i] throughout (gnt[j])[->1]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
      for (j = 0; j<=7; j++) begin
        assert property (p_fixed_priority_i_j(i,j));
      end
    end
  end
endgenerate
```

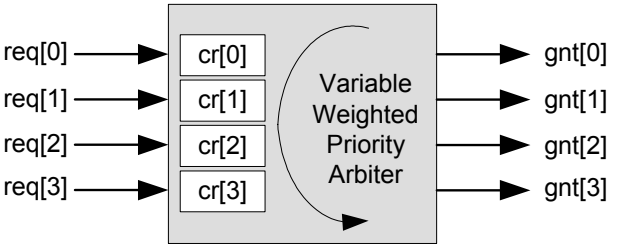
6.1.4 Credit-based weighted priority

In the previous sections, we introduced two common static arbitration schemes (fair arbitration and fixed-priority arbitration). A fixed priority scheme suffers from the possibility of creating a starvation condition for a particular client. In contrast, a fair arbitration scheme's problem area is in failing to use a shared resource by a high bandwidth client in an optimal manner. In this section, we introduce our first variable arbitration scheme that is based on assigning a "weight" to each client, which indicates its expected bandwidth when accessing a shared resource. The weight value is used as a bias during the arbitration cycle (where higher-biased clients ultimately obtain greater access to a shared resource).

There are a number of ways to implement a weighted-priority scheme within a flow-control design [Kariniemi and Nurmi 2005]. One popular weighted priority arbitration implementation in a contemporary flow-control SoC design is to allocate a set of credits to each client, and then process the pending arbiter client's request in a classic round-robin fashion. Credit registers associated with each client are initialized with a value that represents the number of credits available to that particular client. Weighting is accomplished by allocating more credits to a higher-priority client and fewer credits to a lower-priority client. When a client asserts a request to the arbiter, it can only be issued a grant if it has available credit. After a grant is issued, the credit register is decremented. When no credits remain (all clients' credit registers are equal to zero), the credit registers are re-initialized and the arbitration process repeats.

Figure 6-11 illustrates a simple four-client, weighted-priority arbiter. Each arbiter request input port has a credit register associated with it (for example, `cr[0]` for client 0). The arbiter evaluates the combination of pending requests and credit registers in a round-robin fashion (as indicated by the arrow). For our discussion, we assume there is a minimum latency through the arbiter of one cycle.

Figure 6-11. Four-client variable weighted priority arbiter



In general, a weighted priority arbiter has a requirement that a grant should never be issued for a client lacking sufficient credits (that is, the client's credit register is equal to zero). Assertion 6-12 expresses this requirement for our four-client weighted priority arbiter.

Assertion 6-12 No grant without credit

```
property p_no_gnt_without_credit(i);
 @(posedge ck) disable iff (rst)
    gnt[i] & (cr[i]==0) |-> 0;
endproperty

generate
  begin
    genvar i;
    for (i = 0; i<=7; i++) begin
      assert property (p_no_gnt_without_credit(i));
    end
  end
endgenerate
```

Figure 6-12. Trace on which Assertion 6-12 holds

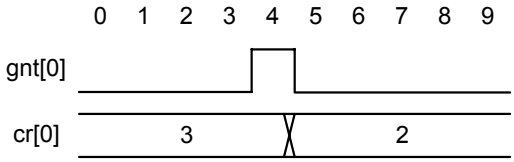


Figure 6-12 illustrates behavior on which Assertion 6-12 holds for client 0. Notice that client 0 has three credits available when a grant is issued.

In Figure 6-12, the credit register (`cr[0]`) decrements after a grant is issued. This is a requirement of our implementation. Hence, we could express an assertion for this requirement as shown in Assertion 6-13.

Assertion 6-13 Credit register decrements after grant

```
property p_credit_reg_dec(i);
 @(posedge ck) disable iff (rst)
    gnt[i] |=> (cr[i] == ($past(cr[i])-1));
endproperty

generate
  begin
    genvar i;
    for (i = 0; i<=7; i++) begin
      assert property (p_credit_reg_dec(i));
    end
  end
endgenerate
```

Let us assume we have an eight-client arbiter. We can use Assertion 6-14 to express that a client with sufficient credit is not starved.

Assertion 6-14 Eight-client weighted priority starvation assertion

```
property p_no_starvation(i,j);
 @(posedge ck) disable iff (rst)
   req[i] & (cr[i]!=0) ##1 (~gnt[i] throughout (gnt[j])[->2]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
      for (j = 0; j<=7; j++) begin
        assert property (p_no_starvation(i,j));
      end
    end
  end
endgenerate
```

Since our variable weighted priority arbiter is implemented using a round-robin scheme, you might have noticed that

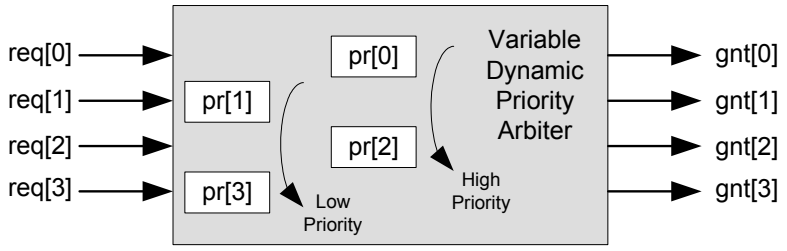
Assertion 6-14 is similar to Assertion 6-4, which checks for fair arbitration.

6.1.5 Dynamic priority

Conventional static arbitration schemes, such as fixed priority and round-robin arbitration, exhibit a number of undesirable behaviors, such as starvation and low client-to-resource bandwidth due to the impartiality (or latency) of the static arbitration scheme. In Section 6.1.4, "Credit-based weighted priority" we examined a variable arbitration scheme that was designed to address some of the undesirable behaviors of a static arbitration scheme. While credit-based weighted priority addresses the starvation problem, poor system performance may still occur for situations where the bandwidth requirement for a particular client changes from a default credit-based configuration. To solve this problem, some designs require the clients to adjust their priority dynamically based on the urgency of servicing a critical event (for example, a full FIFO buffering input transaction). Consequently, the priority of a particular client is increased dynamically to prevent data loss or corruption. In this section, we examine another variable priority scheme known as dynamic priority.

There are a number of ways to implement a dynamic-priority scheme. One technique is to associate a single-bit priority register with each client, as Figure 6-13 illustrates.

Figure 6-13. Four-client variable priority



These registers are initialized to zero (low priority). When a critical event occurs within the design, the client can set the priority register to one (high priority), and then the arbiter processes the pending client's request in a classic round-robin fashion. For example, if client 0 and client 2 decide to increase their priority from low to high (which we have illustrated by shifting their priority registers to the right in Figure 6-13), then these higher-priority requests must be serviced prior to servicing any of the lower-priority requests.

In general, a variable-priority arbiter requires that a lower-priority grant should never be issued to a client when a high-priority request is pending. We can express this requirement for our four-client, weighted-priority arbiter with Assertion 6-15.

Assertion 6-15 Four-client dynamic priority

```
property p_no_lower_priority(i,j);
 @(posedge ck) disable iff (rst)
    (req[i] & (pr[i]==1) & req[j] & (pr[j]==0))
##1 ((~gnt[i] & (pr[i]==1) & (pr[j]=0))
                                  throughout (gnt[j])[->1]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
      for (j = 0; j<=7; j++) begin
        assert property (p_no_lower_priority(i,j));
      end
    end
  end
endgenerate
```

Figure 6-14. Trace on which Assertion 6-15 does not holds

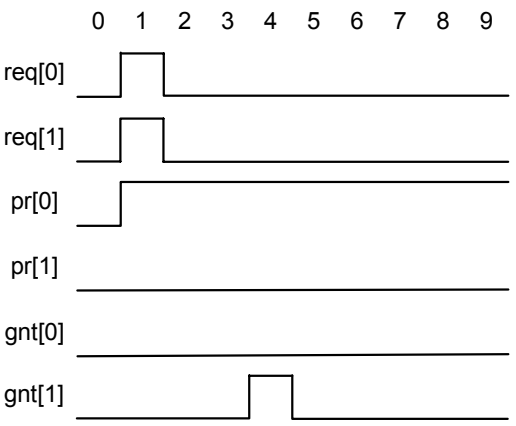


Figure 6-14 illustrates behavior on which Assertion 6-15 does not hold for client 0. Notice that client 0 has a higher priority when a grant is issued to a lower-priority client.

We can write an assertion (similar to Assertion 6-4) that specifies that the arbiter services the high-priority request in a fair fashion by using the SystemVerilog `generate` construct as shown in Assertion 6-16.

Assertion 6-16 Eight-client dynamic high-priority assertion

```
property p_fair_high_priority(i,j);
  @(posedge ck) disable iff (rst)
     (req[i] & (pr[i]==1) & (pr[j]==1))
##1  (~gnt[i] & (pr[i]==1) & (pr[j]==1))
                                  throughout (gnt[j])[->2]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
      for (j = 0; j<=7; j++) begin
        assert property (p_fair_high_priority(i,j));
      end
    end
  end
endgenerate
```

Assertion 6-17 Eight-client dynamic low-priority assertion

```
property p_fair_low_priority(i,j);
  @(posedge ck) disable iff (rst)
    (req[i] & pr==8'b0)
##1 ((~gnt[i] & pr==8'b0) throughout (gnt[j])[->2]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
      for (j = 0; j<=7; j++) begin
        assert property (p_fair_low_priority(i,j));
      end
    end
  end
endgenerate
```

Similarly, Assertion 6-17 specifies that the lower-priority requests must be treated fairly.

complex combined priority schemes

In fact, you might have noticed that we have partitioned the behaviors we are interested in by checking the following:

1. Higher-priority requests are serviced before lower-priority requests.
2. Requests within the same grouping of priority are serviced in a fair fashion.

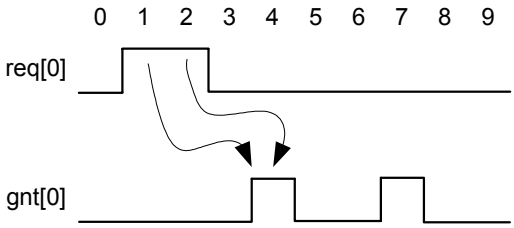
Many of today's complex arbiters actually combine a number of different arbitration schemes. To simplify the task of writing assertions for these complex arbiters, we recommend that you partition the behaviors in a fashion similar to what we did for our dynamic arbiter (for example, separate the priority properties from the fairness properties and do not attempt to write a single complex property to check both schemes).

6.1.6 Interleaving request

Thus far, we have only considered arbiters that support non-interleaving requests. That is, a requesting client must wait until it completes a pending arbitration cycle before it can queue a new request. However, a number of designs, such as pipelined protocols, are optimized for greater throughput by allowing a limited number of outstanding transactions to begin prior to the completion of a previous transaction. Hence, supporting interleaving requests is necessary.

Let us re-consider our simple, two-client, fixed-priority arbiter and assume that our arbiter is able to buffer up to four pending requests per client. If we attempt to use our original Assertion 6-10 for an arbiter supporting interleaving requests, we run into a situation where multiple requests are satisfied by a single grant, as illustrated in Figure 6-15, which is not what we want.

Figure 6-15. A trace on which Assertion 6-10 holds erroneously



As you can see, if we use Assertion 6-10, then we run into a problematic situation where the grant issued at clock five matches both the clock two and clock three requests.

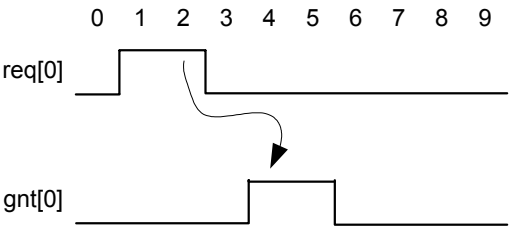
To solve this problem, we introduce a variable in the modeling layer to uniquely tag the appropriate request-grant pairs, as Assertion 6-18 demonstrates.

Assertion 6-18 fixed priority assertion for pipelined request

```
// Modeling code to keep track of req-gnt pairs

// Set up variables to tag req's & gnt's.

 reg [0:1] req_0_cnt, gnt_0_cnt;
 initial {req_0_cnt, gnt_0_cnt} = 4'b0;

 always @(posedge clk) begin
   // Increment counter when event is seen.
     if (req[0]) req_0_cnt <= req_0_cnt + 1;
     if (gnt[0]) gnt_0_cnt <= gnt_0_cnt + 1;
 end

// Fixed priority for 4 pending requests
// For each uniquely tagged request
// minimum arbiter request latency of 3

property p_pipelined_req_gnt;
  reg [0:1] v;
  @(posedge clk) disable iff (rst)
    (req[0], v=req_0_cnt)
##3 (~(gnt[0] & gnt_0_cnt == v) throughout (gnt[1])[->1]) |-> 0;
endproperty

a_pipelined_req_gnt:
  assert property (p_pipelined_req_gnt);
```

For pipelined request-grant pairs, it becomes necessary to consider the arbiter's latency when writing your assertions. For example, consider the trace for a level-sensitive design shown in Figure 6-16, where the lone `req[0]` is asserted at clock one, and the corresponding `gnt[0]` is issued at clock four due to a latency of three cycles through the arbiter. Without considering arbiter latency, it would appear that the fixed priority for the second `req[0]` asserted at clock two is violated at clock four. If we account for the arbiter's three-cycle latency, then this is a valid trace. We have specified this behavior in the assertion of Figure 6-16 by adding a three-cycle delay between the occurrence of `req[0]` and the sequence in which `gnt[0]` occurs.

Figure 6-16. Minimum latency needs to be considered



6.2 Creating an arbiter assertion-based IP

This section demonstrates how to create an arbiter assertion-based IP verification component for a non-interleaving, eight-client fair arbiter with a latency of three cycles by:

- Defining interfaces
- Grouping all the required fair arbiter assertions into a module
- Adding an error status analysis port

6.2.1 Block diagram interface description

Figure 6-17 illustrates a block diagram for a simple eight-client fair arbiter design. For our example, all signal transitions relate only to the rising edge of the bus clock (`clk`). Table 6-1 provides a summary of the bus signals for our simple arbiter example.

Figure 6-17. Eight-client fair arbiter

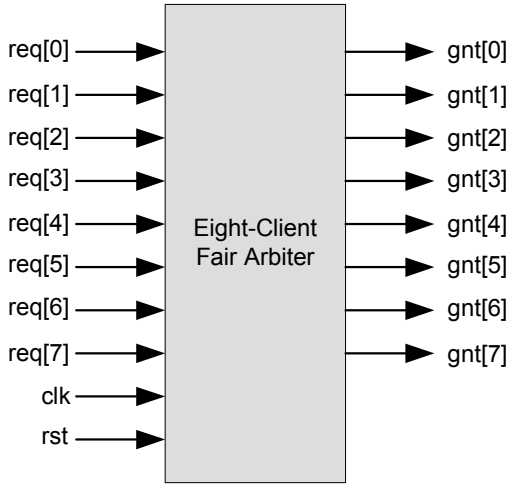


Table 6-1 Simple eight-client fair arbiter signal description

Name	Description
`clk`	All bus transfers occur on the rising edge of `clk`
`rst`	An active high bus reset
`req[0:7]`	Client request signals
`gnt[0:7]`	Arbiter grant signals

6.2.2 Overview description

For our simple arbiter example, to be fair, a pending request for a particular client should never have to wait more than eight arbitration cycles before it is serviced. Otherwise, the alternate client must have been unfairly issued a grant multiple times. There is a minimum time latency through the arbiter of three clocks between the point that a request is asserted and a grant is issued.

6.2.3 Natural language properties

Prior to creating a set of SystemVerilog interface assertions for our simple eight-client fair arbiter, we must first identify a comprehensive list of natural language properties, as shown in Table 6-2:

Table 6-2 Simple eight-client fair arbiter properties

Property name	Summary
Bus legal state	
`p_no_req_i_two_gnt_j`	Arbiter is fair
`p_mutex_gnt`	No more than one grant asserted at a time
`p_minimum_latency`	No grant within three cycles of a request
`p_no_gnt_before_req`	No grant before a request

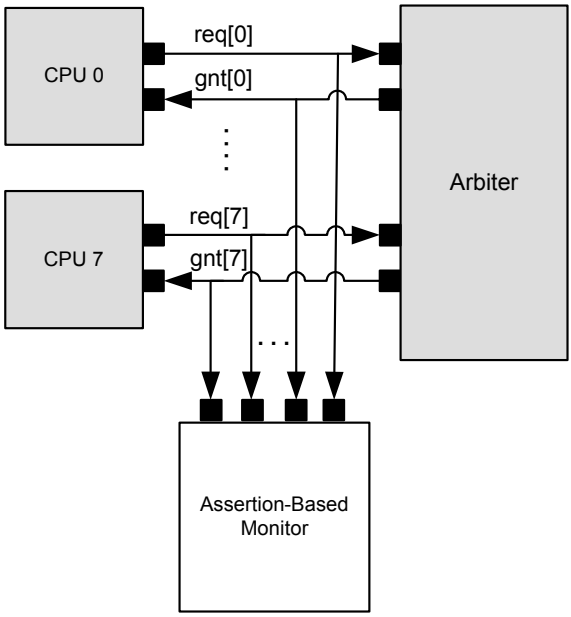
6.2.4 Assertions

To create a set of SystemVerilog assertions for our simple eight-client fair arbiter, we begin defining the connection between our assertion-based monitor and other potential components within the testbench (such as a driver transactor and the DUV). We accomplish this goal by creating a SystemVerilog interface, as Figure 6-18 illustrates.

Example 6-1 on page 137 demonstrated an interface for our arbiter example. For this case, our assertion-based monitor references the interface signals via the direction defined by the `monitor_mp` named modport.[1]

1. For the example in Figure 6-18, the parent module would pass in the appropriate SystemVerilog `modport` (versus the SystemVerilog `interface`) to each specific CPU instance to create the unique connection.

Figure 6-18 SystemVerilog interface

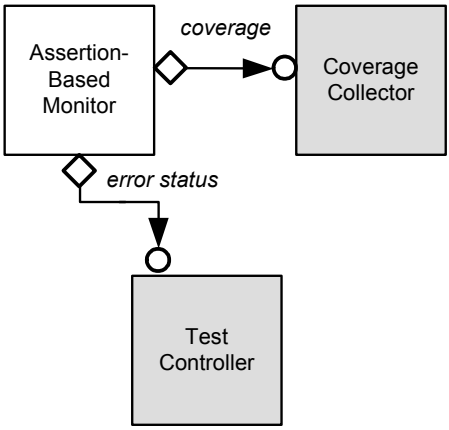


Example 6-1 Encapsulate bus signals inside an interface

```
interface tb_8_client_fair_arbiter_if( input clk , input rst );

  parameter int NUM_CLIENTS = 8;

  bit [0:NUM_CLIENTS-1] req;
  bit [0:NUM_CLIENTS-1] gnt;

  modport cpu_0_mp (
    input        clk , rst ,
    input        gnt[0] ,
    output       req[0]
  );
  . . .
  modport cpu_7_mp (
    input        clk , rst ,
    input        gnt[7] ,
    output       req[7]
  );

  modport monitor_mp (
    input        clk , rst ,
    input        gnt ,
    input        req
  );

endinterface
```

In addition to pin-level interfaces (which monitor the bus) we need a means for our eight-client fair arbiter assertion-based monitor to communicate with various analysis verification components within the testbench. Hence, we introduce an error status analysis port within our monitor, as Figure 6-19 illustrates.

Figure 6-19 Error status and coverage analysis ports



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

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

The *error status* transaction class is defined in Example 6-2, which is an extension on an `ovm_transaction` base class. The `tb_status_arb` class includes an `enum` that identifies the various types of arbiter errors, and a set of methods to uniquely identify the specific error it detected.

Example 6-2 Error status class

```
class tb_status_arb extends ovm_transaction;

  typedef enum { ERR_FAIR ,
                 ERR_MUTEX_GNT ,
                 ERR_MINIMUM_LATENCY ,
                 ERR_INIT_NO_GNT_REQ ,
                 ERR_NO_GNT_BEFORE_REQ } arb_status_t;

 arb_status_t   arb_status;
  int err_client_num;

  function void set_err_fair (input int i);
    arb_status = ERR_FAIR ;
    err_client_num = i;
  endfunction

  function void set_err_mutex_gnt;
    arb_status = ERR_MUTEX_GNT ;
  endfunction

  function void set_err_minimum_latency(input int i);
    arb_status = ERR_MINIMUM_LATENCY;
    err_client_num = i;
  endfunction

  function void set_err_init_no_gnt_req(input int i);
    arb_status = ERR_INIT_NO_GNT_REQ;
    err_client_num = i;
  endfunction


  function void set_err_no_gnt_before_req(input int i);
    arb_status = ERR_NO_GNT_BEFORE_REQ;
    err_client_num = i;
  endfunction

. . .

endclass
```

We are now ready to write assertions for our bus interface properties defined in Table 6-2.

Example 6-3 Modeling to map control signals to conceptual states

```
 bit [0:7] gnt = monitor_mp.gnt;
 bit [0:7] req = monitor_mp.req;

// Fair arbiter (minimum 3 cycle latency between req and gnt)

property p_no_req_i_two_gnt_j (i,j);
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    req[i] ##3 (~gnt[i] throughout (gnt[j])[->2]) |-> 0;
endproperty

generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
      for (j = 0; j<=7; j++) begin
         assert property (p_no_req_i_two_gnt_j(i,j)) else begin
           status = new();
           status.set_err_fair(i,j);
           status_ap.write(status);
         end
       end
     end
  end
endgenerate

// mutually exclusive grants

property p_mutex_gnt;
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    $onehot0(gnt);
endproperty
a_mutex_gnt:
  assert property (p_mutex_gnt) else begin
    status = new();
    status.set_err_mutex_gnt();
    status_ap.write(status);
  end

// minimal latency

property p_minimum_latency(i);
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    $rose(req[i]) ##[0:2] gnt[i] |-> 0;
endproperty

// Continued on next page
```

Example 6-3 Modeling to map control signals to conceptual states

```
generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
        assert property (p_minimum_latency(i)) else begin
          status = new();
          status.set_err_minimum_latency(i);
          status_ap.write(status);
        end
     end
  end
endgenerate

// grant without pending request

property p_init_no_gnt_before_req(i);
  @(posedge monitor_mp.clk)
    $fell(monitor_mp.rst) |-> (~gnt[i] throughout (req[i])[->1]);
endproperty

property p_no_gnt_before_req(i);
  @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
    gnt[0] |=> (~gnt[i] throughout (req[i])[->1]);
endproperty

generate
  begin
    genvar i, j;
    for (i = 0; i<=7; i++) begin
        assert property (p_init_no_gnt_before_req(i)) else begin
          status = new();
          status.set_err_init_no_gnt_before_req(i);
          status_ap.write(status);
        end;
        assert property (p_no_gnt_before_req(i)) else begin
          status = new();
          status.set_err_no_gnt_before_req(i);
          status_ap.write(status);
        end
      end
  end
endgenerate
```

6.2.5 Encapsulate properties

In this step, we turn our set of related arbiter 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 6-4 Encapsulate properties inside a module

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

module tb_8_client_fair_arbiter_mon (
  interface.monitor_mp monitor_mp
);

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

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

6.3 Summary

Understanding how to create assertion-based IP for an arbiter, which easily integrates with other testbench components, is an essential topic of discussion. In this chapter, we reviewed various common arbitration schemes and their associated properties. We then demonstrated the process of creating assertion-based IP for a simple eight-client fair arbiter.

A common mistake assertion-based IP creators make when attempting to write properties for various arbitration schemes is that they try to specify the behavior of all inputs with respect to all outputs at the same time. This is obviously overwhelming and complex. In this chapter, we demonstrated that formal properties can be written for

various arbitration schemes by only considering the behavior of pairs of clients.

Another common mistake assertion-based IP creators make when attempting to write a formal property for complex nested arbitration schemes is that they try to write a single assertion that captures the behavior for the combined nested scheme. A better approach is to consider partitioning the nested arbitration schemes and writing a simpler set of assertions for each separate arbitration scheme.

Finally, when creating parameterized assertion-based IP for arbiters, it is critical to accommodate many different arbiter interface, latency, and size requirements.