cādence®

Dynamic Priority Arbiter Implementation

March 2014

Contents

1

Introduction

Arbiters are a critical component in systems containing shared resources. For example, a design containing a memory controller, where multiple memory agent clients share a common memory bus, requires an arbitration scheme to prevent more than one agent from accessing the bus at the same time. Similarly, for a bridge design where more than one input port shares a common output port, arbitration is needed to prevent dropping or corrupting data transported through the bridge.

Note: If the arbitration scheme is in error, then service to a high-priority client might be ignored, which could result in lost data or even a system crash.

Figure 1-1 Arbiters and Shared Resources
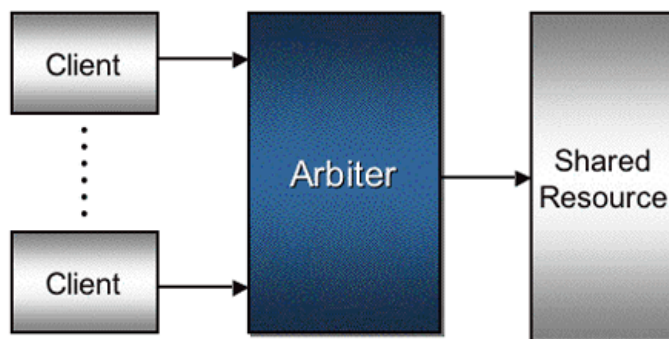
There are many different arbitration schemes ranging from the unfair priority scheme to the fair round-robin scheme, or a combination of priority with fairness. Although arbiters are used in multiple applications, their basic requirements are generally straightforward.

This application note describes how to verify a dynamic priority arbiter.

2

Overview

In this application note you will learn about dynamic priority arbiters. The information in this Overview is useful for understanding the various terms and phrases we use throughout the remainder of this application note.

Fairness, strict priority, and weighted priority arbiters are considered static, which means once they are configured, they are not reconfigured regularly. However, in many cases, the priority of any given port can be raised for some defined situation such as the infrequency of sending requests, a FIFO that holds the data is getting full, and so forth.

Consider the following example:

Figure 2-1 Dynamic Allocated Priority Arbitration
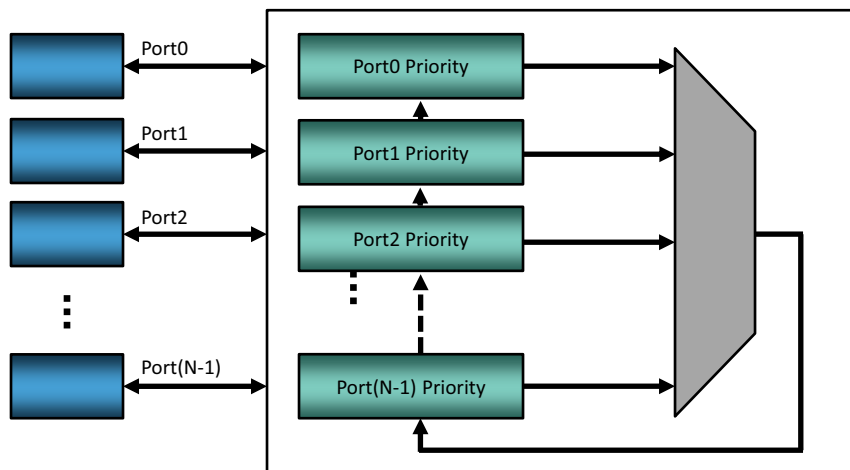
All ports are initially put into the normal-priority bracket and arbitrated using the round-robin scheme. When the input FIFO for `Port0` reaches the high-water mark, it sends a signal to elevate `Port0` to the high-priority bracket. The ports in the high-priority bracket have strict priority over the ones in the normal-priority bracket. After `Port0` gets a grant, it will be placed back in the normal-priority bracket. The following are the requirements for this arbiter:

■ If there is only one port with high priority, it will always get the grant.

■ If there is more than one port with high priority, those ports will follow the round-robin fairness requirement.

■ If there is no port with high priority, all normal-priority ports will follow the round-robin fairness requirement.

All these requirements are verified with stopats on the priority levels of each port. After that we verify that the priority levels are generated correctly:

■ If a FIFO reaches the high-water mark, the priority level will be increased to high.

■ The priority level will decrease if the FIFO drops below the high-water mark.

Reviewing the Weighted Priority Arbiter example reproduced below in Figure 2-2 on page 8, recall that the weight or credit for a particular port is decreased whenever a grant is given.

Figure 2-2 Credit-Based Weighted Round-Robin
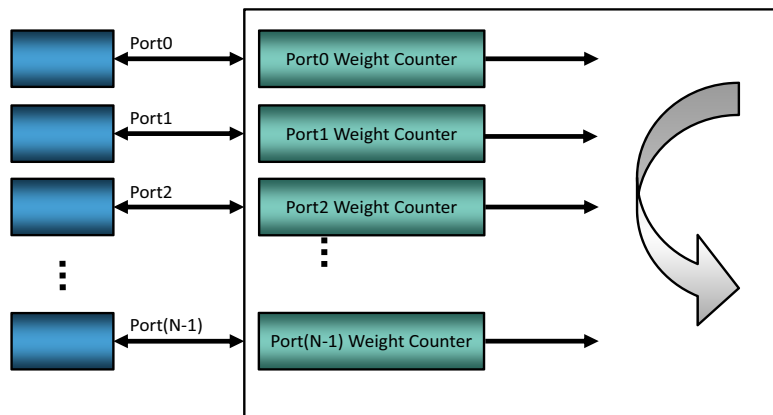
The weight is increased whenever one of the following conditions arise:

■ All ports reach zero

■ All non-zero credit ports do not have a request and

■ Credit refresh occurs (if applicable)

In some designs, the first two conditions refresh the credits.

Consider the following example:

```
wire [`NUM_PORT-1:0] credit_remains;
wire gnt_no_credit = ~credit_remains[p1] & gnt[p1];
wire gnt_fairness = req[p1] & (~p1p2_gnt | ~credit_remain[p2]) & gnt[p2];
wire groupB_gnt = gnt[4] | gnt[5] | gnt[6] | gnt[7];
```

where `credit_remains` is constrained to match the following behavior:

■ If `credit_remains` is high, it should not be de-asserted unless the port gets a grant.

■ If `credit_remains` is low, it should not be asserted unless one of the refresh conditions occurs.

The arbiter "remembers" which port got the last grant. For example, a pair of ports, `p1` and `p2`, has been issuing requests and getting grants. The relative priority between `p1` and `p2` is based on which port has the most recent grant. If `p1` sends a request and `p2` has the most recent grant between the two, and yet it gets the grant, this scenario violates the fairness requirement.

After verifying all the arbitration requirements, we verify that the credit/weight handling is correct:

■ If credit is available, each grant will decrement the credit by one.

■ If a credit refill condition occurs, the credit will be initialized.

■ If neither credit refill nor grant occurs, the number of credits for that port should remain constant.

In the following example, `refill_condition` is the condition where credit is being refilled. The signal `credit_init` is the initial weight/credit assigned to each port. To verify all possibilities, do not assign constants to `credit_init`, but rather keep it pseudo-constant (can take any value but does not change).

Since there are many possibilities in terms of the weight assigned to each port, you can choose to abstract the counter that keeps track of the weight addition/subtraction.

```
// The previous (registered) value of credit

reg [3:0] credit0_d1;

// The previous value of grant

reg [`NUM_PORTS-1:0] gnt_d;
```

```verilog
wire credit_decrement_0 = gnt_d1[0] & (credit[0]!=credit0_d1-1);
wire credit_refill = refill_condition & (credit[0]!=credit_init);
wire credit_constant = refill_condition & !gnt_d1[0] &
                                          (credit[0]!=credit0_d1);
```

3

Interface

In this section you will learn about various interface characteristics you will need to consider when customizing our example requirements for your specific design. In this section, we illustrate through example a few common arbiter interface requirements. Although your specific arbiter interface requirements might differ from the examples we provide, our intent is to provide a starting point that you can modify to fit your specific needs.

This section includes the following topics:

- ■ Interface Description

- ■ Steps for Customizing Your Interface Requirements

- ■ Interface Names

- ■ Examples

Interface Description

In addition to the dynamic priority requirements previously described in the Overview (see "Overview" on page 7), often there are other temporal interface requirements for the relationship of the client's request and the arbiter's grant signals. For example:

- ■ Is there a minimum time interval after a request when a grant must not occur due to latency considerations within the design?

- ■ Does the interface support queuing multiple requests on the same port while waiting for a grant?

Steps for Customizing Your Interface Requirements

Use the following steps to define your specific arbiter's interface requirement.

1. Generally describe the operational sequence of events for the interface.

2. Graphically illustrate the interface behavior.

3. Using the operational description and graphical waveform, define the detailed temporal requirements for the interface.

Interface Names

We use the following names for interface signals during our discussion and in the modeling requirements:

`req [n]`	Request issued by a client to the arbiter
`gnt [n]`	Grant issued by the arbiter to a client
`xfer [n]`	Start of a transfer issued by a client
`p1`	Specific input request port under test
`p2`	Specific input request port under test
`req2gnt_latency`	Required latency time between req and gnt

Examples

Refer to the following interface examples:

■ Example 1. Grant Remains Active Until Transfer

■ Example 2. Multiple Pending Requests on the Same Port

■ Example 3. Latency Considerations for Request and Grant Signals

Example 1. Grant Remains Active Until Transfer

Step 1: Generally describe the operational sequence of events for the interface.

    a. A request (Req) is asserted, indicating that the client is ready to transfer data on a bus.

    b. The request (Req) stays asserted until a grant is provided, after which the request is de-asserted.

    c. The arbiter eventually asserts a grant (Gnt).

   d. The grant (Gnt) must remain asserted until up to four cycles or until a start of transfer occurs (Xfer).

   e. The grant (Gnt) issued to any client must remain de-asserted from the time that a transfer begins (Xfer) until the transfer ends.

Step 2: Graphically illustrate the interface behavior.

Figure 3-1 Request/Grant/Transfer Relationships for Example 1
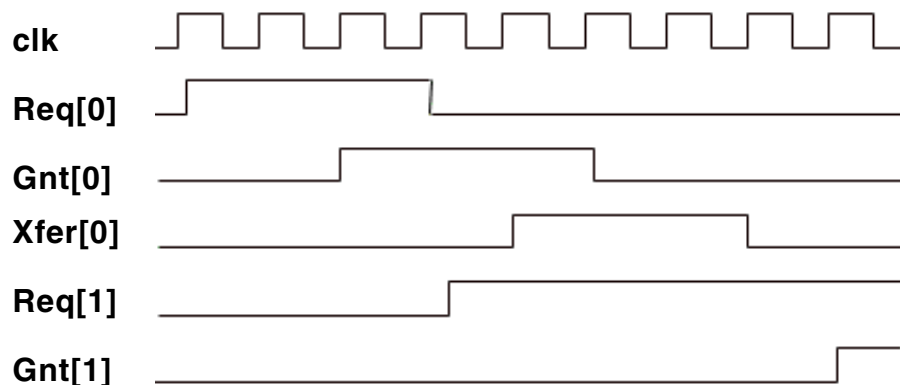
Step 3: Using our operational description and graphical waveform, define the detailed temporal requirements for the interface.

   a. Request should be asserted until grant is asserted, after which request should be de-asserted.

   b. Transfer should not be asserted until grant is provided (constraint).

   c. Grant should be asserted until either four cycles have elapsed or after a transfer is asserted.

   d. Grant should not be asserted until the bus is idle (transfer is done).

   e. Only one grant can be issued at one time.

   f. Grant should be asserted if there is any request pending and the bus is idle.

   g. Grant should not be asserted if there is no request pending for any port.

Example 2. Multiple Pending Requests on the Same Port

Step 1: Generally describe the operational sequence of events for the interface.

    a. Request is asserted for one cycle, indicating that the client is ready to transfer data.

    b. Request can be asserted again. Up to four pending requests can be issued.

    c. Arbiter asserts grant for one cycle to allow up to four cycles of transfer.

    d. Arbiter does not assert another grant until the bus is idle.

Step 2: Graphically illustrate the interface behavior.

Figure 3-2 Request/Grant/Transfer Relationship for Example 2
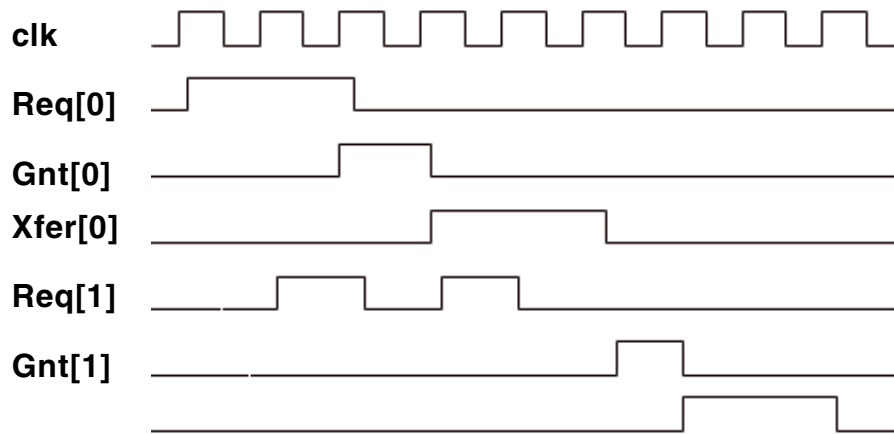
Step 3: Using our operational description and graphical waveform, define the detailed temporal requirements for the interface.

    a. Grant should not be asserted until the bus is idle (transfer is done).

    b. Only one grant can be issued at one time.

    c. Grant should be asserted if there is any request pending and the bus is idle.

    d. Grant should not be asserted if there is no request for that port (need to keep track of the number of pending requests).

Example 3. Latency Considerations for Request and Grant Signals

Although most arbiter interface requirements are relatively simple, there is often latency between the requests at the input to the actual arbiter. Similarly, there is usually some latency after the grant is issued before it is visible at the interface. To model and verify the proper behavior, we must register the request and/or grants to take into account the latency. This is especially apparent in the following example.

Step 1: Generally describe the operational sequence of events for the interface.

    a. Request is asserted until grant is provided. Otherwise, if request is de-asserted before the grant, it means the client is abandoning the request.

    b. Requests take three cycles to propagate to the internal arbiter, hence there is a minimum three-cycle latency before the request is processed.

    c. Arbiter asserts grant for one cycle.

    d. Transfer signal to indicate bus is busy has one-cycle latency to the arbiter.

Step 2: Graphically illustrate the interface behavior.

Figure 3-3 Request with Latency of Three
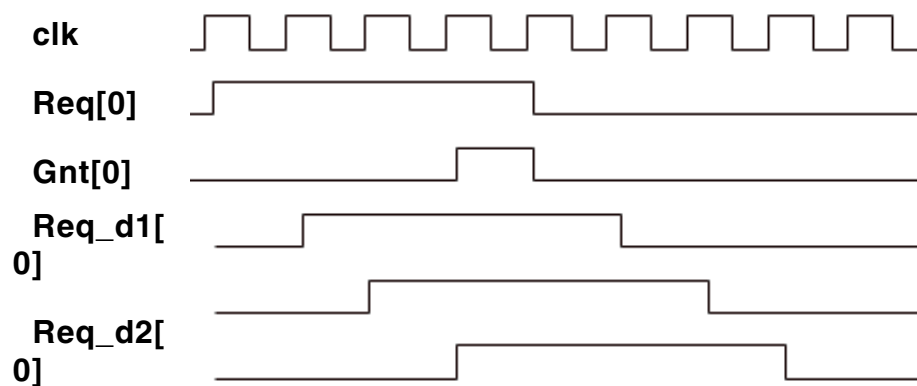
Step 3: Using our operational description and graphical waveform, define the detailed temporal requirements for the interface.

The requirements are similar to the ones described in the previous example. However, there are latency indications. Let us consider the following requirements.

    a. Grant should not be asserted without a request.

    b. Grant should be given to a request within three cycles if the bus is idle.

Consider requirement 1: *Grant should not be asserted without a request*. According to Figure 3-3 on page 15, the earliest grant can be asserted is three cycles after request. Asserting the grant earlier contributes to a violation, even though the request is asserted due to internal latency. To properly capture the requirement, we should register the request three times (denoted by `req_d1`, `req_d2`, and `req_d3`) and use `req_d3` as the actual request that is supposed to trigger a grant. If a grant is asserted even one cycle earlier, it is a violation because the internal arbiter will not have seen the request yet. We model this behavior as:

```
wire err_gnt_wo_req = gnt[0] & ~req_d3[0];
```

Now we consider requirement 2: *Grant should be given to a request within three cycles if the bus is idle*. The waveform shown in Figure 3-4 on page 16 includes the transfer signal that indicates the bus is busy.

Figure 3-4 Request with Latency of Three and Transfer with Latency of One
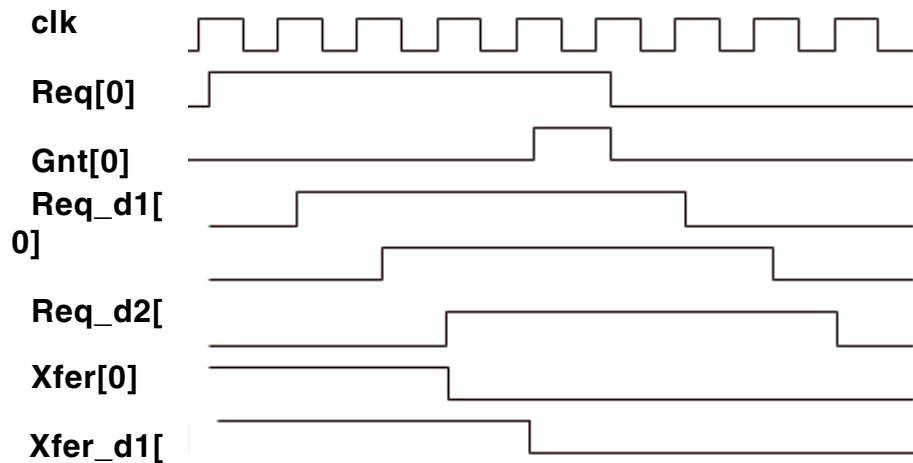
Here the signal xfer is de-asserted (indicates that the bus is idle) after the first cycle. However, since there is a one-cycle latency between transfer and grant, we must use the registered version of xfer to verify the grant condition. It is also possible that even though the bus is idle, gnt[0] is not asserted because a different port is getting the grant. Therefore, the requirement must include conditions that no other grant is asserted. The following is a sample requirement for a 16-port arbiter.

```
wire err_gnt_not_assert = ~xfer_d1[0] & req_d3[0] & gnt[15:1]==15'b0 & ~gnt[0];
```

4

Testplan

This section presents the high-level requirements that should be formally verified on your dynamic priority arbiter.

Refer to the following tables:

■ [Requirements Set 1 – General Arbiter Requirements](#)

■ [Requirements Set 2 – Dynamic Priority Requirements](#)

Note: You will create an additional table that lists interface requirements (see [Table 4-3, "Requirements Set 3 – Arbiter Interface Requirements"](#)).

Table 4-1 Requirements Set 1 – General Arbiter Requirements

Requirement Name	Summary
err_req_no_gnt	When a client generates a request, and no other client has generated a request, then the next grant must be issued to the requesting agent.
err_gnt_wo_req	No grant is issued without a request.
err_multiple_gnt	No more than one grant can be issued at one time.

Table 4-2 Requirements Set 2 – Dynamic Priority Requirements

Summary
If there is only one port with high priority, it will always get the grant.
If there is more than one port with high priority, it will follow the round-robin fairness requirements.

Table 4-2 Requirements Set 2 – Dynamic Priority Requirements

Summary

If there is no port with high priority, all normal priority ports will follow the round-robin fairness requirements.

Note: If your arbiter has multiple arbitration schemes, simply combine the requirements from the different arbiter formal testplan sections.

Table 4-3 Requirements Set 3 – Arbiter Interface Requirements

This list includes temporal interface requirements for your specific arbiter. Since this list is very specific to your design, you will create it. Refer to the Interface section (see "Interface" on page 11) for examples of various request/grant interface requirements.

5

Requirements

In this section we use the interface signals (see "Interface Names" on page 12) and the set of requirements identified in the Testplan (see Table 4-2 on page 17) to create the high-level requirements models that monitor design block input and output signals.

This section contains the Verilog® source code for various dynamic priority arbiter high-level requirements models. A common set of constraints used to prove dynamic priority arbiters is provided.

In addition, high-level requirements are modeled for FIFO-based and credit-based priority generation arbiters.

The example requirement and constraint models provided in this section include:

■ Dynamic Priority Arbiter Requirements Model

■ Dynamic Priority Arbiter Constraint Model

■ FIFO-Based Priority Generation Requirements Model

■ Credit-Based Priority Generation Requirements Model

Dynamic Priority Arbiter Requirements Model

For dynamic priority, the priority levels change from time to time based on various factors. Here we illustrate two examples, the first shows that priority can be increased; hence, we increase priority from one (normal) to two and three. In the second scenario, the priority decreases to zero, meaning the port will not have access to the bus.

Note: See "Overview" on page 7 for a review of relevant terms and phrases.

```
/* ------------------------------------------------------------------------ */
/* -                       JASPER_arb_dyn_reqm                         - */
/* -                                                                   - */
/* ------------------------------------------------------------------------ */
```

```verilog
`define NUM_PORT 8

module dynamic_arb (clk, rstN, req, gnt, p1, p2, req2gnt_latency,
                    priority_0, priority_1, priority_2, priority_3,
                    priority_4, priority_5, priority_6, priority_7);

    input               clk;
    input               rstN;
    input [`NUM_PORT-1:0] req;
    input [`NUM_PORT-1:0] gnt;
    input [1:0]         priority_0, priority_1, priority_2, priority_3,
                         priority_4, priority_5, priority_6, priority_7;

    // Select two arbitrary ports for verification

    input [2:0]         p1, p2;
    input [2:0]         req2gnt_latency;
    wire                req_p1;
    reg                 req_d1_p1, req_d2_p1,req_d3_p1,req_d4_p1,
                         req_d5_p1, req_d6_p1, req_d7_p1;

    // Selecting the request that corresponds to the grant

    always @(req_d1_p1 or req_d2_p1 or req_d3_p1 or req_d4_p1 or
             req_d5_p1 or req_d6_p1 or req_d7_p1 or req or p1 or
             req2gnt_latency) begin
      case (req2gnt_latency)
       0: req_p1 = req[p1];
       1: req_p1 = req_d1_p1;
       2: req_p1 = req_d2_p1;
       3: req_p1 = req_d3_p1;
       4: req_p1 = req_d4_p1;
       5: req_p1 = req_d5_p1;
       6: req_p1 = req_d6_p1;
       7: req_p1 = req_d7_p1;
      endcase
    end

    // Registering the request to reflect the proper request-to-grant latency
```

```verilog
always @(posedge clk) begin
  if (!rstN) begin
     req_d1_p1 <= 1'b0;
     req_d2_p1 <= 1'b0;
     req_d3_p1 <= 1'b0;
     req_d4_p1 <= 1'b0;
     req_d5_p1 <= 1'b0;
     req_d6_p1 <= 1'b0;
     req_d7_p1 <= 1'b0;
  end
  else  begin
     req_d1_p1 <= req[p1];
     req_d2_p1 <= req_d1_p1;
     req_d3_p1 <= req_d2_p1;
     req_d4_p1 <= req_d3_p1;
     req_d5_p1 <= req_d4_p1;
     req_d6_p1 <= req_d5_p1;
     req_d7_p1 <= req_d6_p1;
  end
end


//   The priority of ports p1 and p2

reg [2:0] priority_p1, priority_p2;

always @(priority_0 or priority_1 or priority_2 or priority_3 or
         priority_4 or priority_5 or priority_6 or priority_7 or
         p1 or p2) begin

  case(p1)
   0: priority_p1 = priority_0;
   1: priority_p1 = priority_1;
   2: priority_p1 = priority_2;
   3: priority_p1 = priority_3;
   4: priority_p1 = priority_4;
   5: priority_p1 = priority_5;
   6: priority_p1 = priority_6;
   7: priority_p1 = priority_7;
  endcase

  case(p2)
```

```verilog
      0: priority_p2 = priority_0;
      1: priority_p2 = priority_1;
      2: priority_p2 = priority_2;
      3: priority_p2 = priority_3;
      4: priority_p2 = priority_4;
      5: priority_p2 = priority_5;
      6: priority_p2 = priority_6;
      7: priority_p2 = priority_7;
    endcase
  end


  //  Pointer-based round-robin algorithm approach
  //  #1 recommended if the number of ports is high (>8).
  //  Monitor two ports at a time, verify one port at a time.
  //  Allow activity in all ports.


  reg other_gnt;

  always @(posedge clk) begin
    if (req_p1) begin
       if (gnt[p2]) other_gnt <= 1'b1;
       else if (gnt[p1]) other_gnt <= 1'b0;
    end
  end


  //  For dynamically-assigned priority, we must monitor priority
  //  levels; otherwise, it is the same as a regular round-robin

  wire err_dyn_pri_1 = (priority_p2==priority_p1) & req_p1 &
                                        other_gnt & gnt[p2];


  //  For priority 0, gnt should not be issued

  wire err_dyn_pri_2 = priority_p1==2'b0 & gnt[p1];


  //  If the dynamic priority of p1 is higher than that of p2,
  //  gnt should not be issued to p2

  wire err_dyn_pri_3 = (priority_p2<priority_p1) & req_p1 & gnt[p2];

endmodule
```

Dynamic Priority Arbiter Constraint Model

The following code provides the modeling for some common constraints used to prove dynamic priority arbitration.

Note: See "Overview" on page 7 for a review of relevant terms and phrases.

```
/* ---------------------------------------------------------------------- */
/* -                      JASPER_arb_dyn_constraint                  - */
/* -                                                                  - */
/* ---------------------------------------------------------------------- */


`define NUM_PORT 8


module dynamic_constraint (clk, rstN, gnt, priority_p1, priority_p2, p1, p2);

    input                 clk;
    input                 rstN;
    input [`NUM_PORT-1:0] gnt;
    input [1:0]           priority_p1, priority_p2;
    input [2:0]           p1, p2;


    reg [1:0]             prev_priority_p1, prev_priority_p2;


    always @(posedge clk) begin
      if (!rstN) begin
         prev_priority_p1 <= 2'b0;
         prev_priority_p2 <= 2'b0;
      end
      else begin
        prev_priority_p1 <= priority_p1;
        prev_priority_p2 <= priority_p2;
      end
    end


    reg prev_gnt_p1, prev_gnt_p2;


    always @(posedge clk) begin
      if (!rstN) begin
         prev_gnt_p1 <= 1'b0;
         prev_gnt_p2 <= 1'b0;
```

```
            end
        else begin
            prev_gnt_p1 <= gnt[p1];
            prev_gnt_p2 <= gnt[p2];
        end
    end


    //  In a normal case, priority levels are incremented when some
    //  event occurs in that port, such as FIFO is getting full,
    //  or the port has been waiting for the bus for a while.
    //  Priority should not be increased if the port just got a grant.

    wire forbid_1 = prev_gnt_p1 & (prev_priority_p1 < priority_p1);


    //  Priority should not be decreased if the port did not get a grant.

    wire forbid_2 = ~prev_gnt_p1 & (prev_priority_p1 > priority_p1);


endmodule
```

FIFO-Based Priority Generation Requirements Model

The next section of code is used to check that the priority generation is correct. Combining this with the arbitration requirement completes the set of high-level requirements necessary for the proof. In this example, the FIFO for each port has a high watermark, upon reaching the watermark, the priority level increases.

Note: See "Overview" on page 7 for a review of relevant terms and phrases.

```
/* ------------------------------------------------------------------------ */
/* -                     JASPER_arb_dyn_fifo_based                      - */
/* -                                                                    - */
/* ------------------------------------------------------------------------ */



module priority_gen_1 (clk, rstN, priority, high_watermark, gnt, transfer);


    input    clk, rstN;
```

```verilog
//   0 for normal, 1 for high


input   priority;
input   high_watermark;
input   gnt;
input   transfer;              //   Transfer in process

reg     busy;                  //   The duration after each grant and
                               //    before transfer is completed
reg     prev_transfer;         //   Delayed version of transfer
reg     prev_gnt;              //   Delayed version of gnt
reg     prev_high_watermark;   //   Delayed version of high_watermark
reg     prev_priority;         //   Delayed version of priority

always @(posedge clk) begin
  if (!rstN) busy <= 1'b0;
  else if (gnt) busy <= 1'b1;
  else if (prev_transfer & ~transfer) busy <= 1'b0;
end

always @(posedge clk) begin
  if (!rstN) begin
     prev_gnt <= 1'b0;
     prev_transfer <= 1'b0;
     prev_high_watermark <= 1'b0;
     prev_priority <= 1'b0;
  end
  else begin
     prev_gnt <= gnt;
     prev_transfer <= transfer;
     prev_high_watermark <= high_watermark;
     prev_priority <= priority;
  end
end


//   If high_watermark is reached, priority should be increased


wire err_priority_0 = prev_high_watermark & ~busy & ~priority;


//   If gnt is provided and high_watermark is removed,
//   priority should be at normal again.
```

```
//   Here, transfer just finished and high_watermark is at zero.
//   Priority should go back to zero.
```

wire err_priority_1 = ~prev_high_watermark & ~busy &
                                        prev_transfer & priority;

```
//   Priority level should not change if there is no triggering event.
```

wire err_priority_2 = (prev_high_watermark==high_watermark) &
              ~prev_gnt & ~prev_transfer & (prev_priority!=priority);

endmodule

Credit-Based Priority Generation Requirements Model

The next file shows some of the code that checks that the priority generation is correct. Combining this with the arbitration requirement completes the proof. In this example, each port is given a fixed amount of credit. When the credit is used, the priority will be zero until the credit is refreshed. Note that priority one is the normal priority.

Note: See for a review of relevant terms and phrases.

```
/* ------------------------------------------------------------------------- */
/* -                       JASPER_arb_dyn_credit                        - */
/* -                                                                    - */
/* ------------------------------------------------------------------------- */
```

module priority_gen_2 (clk, rstN, priority, gnt, credit, credit_refr, max_credit);

input	clk, rstN;	
input	priority;	// *1 for normal, 0 for no credit*
input	gnt;	
input [3:0]	credit;	// *Credit remaining*
input [3:0]	max_credit;	// *Maximum credit upon refresh*
input	credit_refr;	// *Credit refresh*
reg [3:0]	prev_credit;	// *Delayed version of credit*
reg	prev_gnt;	// *Delayed version of grant*
reg	prev_credit_refr;	// *Delayed version of credit refresh*

```verilog
always @(posedge clk) begin
  if (!rstN) begin
    prev_credit <= 1'b0;
    prev_gnt <= 1'b0;
    prev_credit_refr <= 1'b0;
  end
  else begin
    prev_credit <= credit;
    prev_gnt <= gnt;
    prev_credit_refr <= credit_refr;
  end
end


// If gnt is given, credit should be decremented

wire err_credit_1 = prev_gnt & (credit!=(prev_credit-1'b1));

// If credit_refr is asserted, credit should be refreshed

wire err_credit_2 = prev_credit_refr & (credit!=max_credit);

// When credit is 0, priority level should be zero

wire err_credit_3 = prev_credit==4'b0 & priority;

// When credit is not 0, priority level should be one

wire err_credit_4 = prev_credit!=4'b0 & ~priority;

// When credit is 0, grant should not be provided

wire err_gnt_1 = prev_credit==4'b0 & gnt;

endmodule
```

6

Strategy

This section shows techniques and strategies that address potential arbiter performance issues. Several advanced techniques are generally used to formally verify arbiters. We recommend you review all the suggestions in this section. However, it is not critical to understand all the details of the various techniques prior to starting your proof. You may choose to apply a few of our suggestions to proactively prevent performance issues during your proof. Alternatively, you might choose to wait and apply a few of our suggestions later only if you are unable to complete your proof due to a performance problem.

This section includes the following topics:

- Verification Plan

- What Ports Should You Monitor as Part of Your Requirements?

- Speeding Up Your Proof

Verification Plan

The Verification Plan contains a set of optional "Restriction Definitions" and the recommended "Verification Steps." The combination of restrictions and steps forms an effective methodology.

Restriction Definitions

For an immature design, you might need restrictions. As the design matures, you might choose to reduce the restrictions or skip them altogether. Restrictions allow you to verify basic functionality and mainstream problems before complicated corner-case bugs. They also allow you to verify a design before it is complete.

For example, a designer is working on a 64-port arbiter with round-robin arbitration and three levels of strict priority. To simplify, you might implement a one-priority, round-robin arbiter with two ports before expanding to more ports and more fixed priorities. Therefore, it is probably beneficial to first set the restrictions to arbitrate only between two ports.

1. Only one port has an active request.

2. Only two ports have active grants.

3. Only one arbitration scheme is active at a time (for arbiters containing multiple arbitration schemes).

Verification Steps

The following lists recommended steps for your proof.

1. Prove [Requirements Set 1](#) with [Restriction Definition 1](#), [Restriction Definition 2](#), and [Restriction Definition 3](#).

2. Prove [Requirements Set 3](#) with [Restriction Definition 1](#), [Restriction Definition 2](#), and [Restriction Definition 3](#).

3. Prove [Requirements Set 1](#) with [Restriction Definition 2](#) and [Restriction Definition 3](#).

4. Prove [Requirements Set 3](#) with [Restriction Definition 2](#) and [Restriction Definition 3](#).

5. Prove [Requirements Set 2](#) with [Restriction Definition 2](#) and [Restriction Definition 3](#).

6. Repeat step 5 for all different arbitration schemes if applicable.

7. Prove [Requirements Set 1](#) with [Restriction Definition 3](#).

8. Prove [Requirements Set 3](#) with [Restriction Definition 3](#).

9. Prove [Requirements Set 2](#) with [Restriction Definition 3](#) (for all arbitration schemes if applicable).

10. Prove [Requirements Set 1](#) with no restrictions.

11. Prove [Requirements Set 3](#) with no restrictions.

12. Prove [Requirements Set 2](#) with no restrictions.

What Ports Should You Monitor as Part of Your Requirements?

When creating an arbiter requirements model, we often take advantage of the following property:

If each individual port of an arbiter does not violate any of its requirements, then an arbiter that consists of multiple ports will never violate any of the requirements.

That is, for many arbiter requirements, it is sufficient to prove the requirement with respect to a single port separately versus attempting to prove the requirement with respect to all the ports at once. Furthermore, writing a requirement for a single port is usually easier than writing a requirement that attempts to capture the behavior of all the ports.

Consider the following example:

■ **Specification**

A grant should not be asserted for any particular port when there was not a corresponding request issued for that same port.

■ **Requirements Model**

```
wire err_gnt_wo_req_1 = (gnt != `NUM_PORT'b0) & ((gnt & req) == `NUM_PORT'b0);
```

In the previous example, we are modeling the required behavior using a Boolean expression, which is monitoring the occurrence of grant (that is, the n-bit `gnt` variable is non-zero). When a grant occurs, the expression checks to see if a corresponding request is missing (that is, the n-bit `req` variable is zero). We model this behavior by "AND-ing" n-bit `req` (request) and `gnt` (grant) variables while making the assumption that grant will be one-hot (which you should prove as a separate requirement).

As you can see, trying to specify all ports at once makes a simple requirement more complicated than necessary. Furthermore, specifying all ports as part of the requirement can increase the proof run time since exploring a larger state space might be required to complete the proof.

Now consider the following alternative:

```
wire err_gnt_wo_req_2 = gnt[p1] & ~req[p1];
```

where the `p1` variable (which is part of our requirement model) is used to select any arbitrary arbiter port from the n-bit `gnt` and `req` signals. You can see that the new requirement is simpler and straight forward. To ensure that all ports are proven correctly, we recommend that you use one of the two following techniques.

■ **Technique 1:** Use the `assume -constant` command.

```
assume -constant p1
```

The `assume -constant` command ensures that the port number is held to an arbitrary constant value throughout the evaluation of the proof. That is, all arbitrary values are explored, but for any particular single trace (path) the value will not change, which means that the formal tool will not return a false failure due to changing a port selection for a request or grant in the middle of its evaluation.

The advantage of using this constraint to specify an arbitrary value is that we are able to verify our requirement for all ports without the need to verify each port separately (since JasperGold® Apps will try all possible combinations). Alternatively, you can use Technique 2 to set `p1` to a constant and go through all ports sequentially.

■ **Technique 2:** Create a script to prove each port separately.

```
# Prove port 0
assume -name assume_port_0 p1==3'd0
prove ~err_gnt_wo_req_1
assume –remove assume_port_0
#
# Prove port 1
assume -name assume_port_1 p==3'd1
prove ~err_gnt_wo_req_1
assume –remove assume_port_1
.
.
.
```

The advantage of the second technique is that each proof is simpler compared to the first technique's `assume -constant` command. However, it comes at a cost of less automation since the second technique requires you to manually create the script to check each port. Therefore, we recommend that you initially try Technique 1.

Note: When proving arbiter requirements, it is often easier to prove an arbitrary pair of ports that is specified by two variables `p1` and `p2` (similar to Technique 1) than attempt to prove all ports as part of a single requirement.

Speeding Up Your Proof

For a general discussion on techniques to speed up your proof, refer to the "Appendix: Techniques and Strategies" on page 37. To go directly to a specific topic of interest, click on one of the following cross-reference links:

■ Taking Advantage of Design Symmetry on page 37

7

Coverage

In this section, we define a set of coverage goals you can use to confirm that your coverage objectives were met during the proof.

After your requirements prove true, you should always perform a sanity trace and cover on all major functionality. It is important to:

■   Ensure that you are proving what you think you are proving

■   Cover the function and exit from that function

■   Ensure that specific states or conditions in your high-level requirements model have not been over-constrained by any assumptions

For example, we can get the coverage on a request being asserted. However, the proper coverage includes a request being asserted *and* de-asserted. Similarly, we should check the coverage of the priority going from one level to another, rather than just one priority level at one time. To support these objectives, this section lists coverage goals you will check after a requirement proves true.

Note: Refer to the command reference for details on the `visualize`, `cover`, and `prove` commands.

Coverage Set 1

■   Coverage on all requests, grants, and transfers (if applicable).

■   Coverage on all priority levels, all credit/weight levels.

Coverage Set 2

■   Coverage on all dynamic allocation processes: refresh, out-of-credit, and so forth.

Appendix: Techniques and Strategies

In some cases, your proof can benefit from an advanced methodology technique to overcome complexity problems during a proof. For example, various abstraction techniques are required when the number of arbiter clients is high. Abstraction is one technique that can simplify many complex structures without compromising the integrity of the proof. In this appendix, we introduce a few advanced techniques that can be useful for speeding up your proof. We suggest you review all the suggestions in this appendix; however, it is not critical to understand all the details of the various techniques prior to starting your proof. You might choose to apply a few of our suggestions to proactively prevent performance issues during your proof. Or you might choose to wait and apply our suggestions later if you are unable to complete your proof due to a performance problem.

This appendix includes the following techniques for speeding up your proofs:

■   Taking Advantage of Design Symmetry

■   Multiplexer Abstraction

■   FIFO Abstraction

■   Induction

■   Counter Abstraction

■   Separating Decoding Logic

Taking Advantage of Design Symmetry

Many types of designs (such as pointer-based, round-robin arbiters) have a symmetrical implementation. That is, the logic associated with each port has the exact same RTL structure. Generally, the only exception to a symmetrical arbiter implementation is the initial priority value for each port (that is, you will give one port the highest priority).

Symmetry allows us to efficiently prove requirements on a pair of symmetrical paths (for example, ports of an arbiter) while giving us confidence that the requirement is true on all ports due to the symmetrical implementation. If you are confident about the symmetry among ports in the implementation of your design, then we recommend you take advantage of this symmetry as part of your proof.

Arbiter Example

Consider the following Verilog® code fragment for an arbiter that has a symmetrical implementation per port with the exception of the initial priority value:

```verilog
// pointer to determine round-robin

reg [3:0] priority_pointer;

// priority port

reg [7:0] grant;

always @(posedge clk) begin
  if (reset) begin
    priority_pointer <= 4'd0;
    grant <= 8'd0;
  end
  else if (grant !=8'd0) grant <= 8'd0;
  else if (request[priority_pointer]) begin
    priority _pointer <= priority_pointer + 5'd1;
    grant[priority _pointer] <= 1'b1;
  end
  else if (request[priority_pointer+5'd1]) begin
    priority_pointer<=priority_pointer + 5'd2;
    grant[priority _pointer + 5'd1] <= 1'b1;
  end
  else if (request[priority_pointer+5'd2]) begin
    priority_pointer<=priority_pointer + 5'd3;
    grant[priority _pointer + 5'd2] <= 1'b1;
  end
  .
  .
  .
end
```

In the previous example, you can see that the logic associated with each port is symmetrical since each n-bit variable assignment follows the same RTL structural path. The only part of this implementation that is not symmetrical is the reset value of priority_pointer, which points to port0 after reset (that is, it has the highest priority after reset). To allow true

symmetry for our proof, we can use the `abstract -init_value` command to ignore the initial value.

> **abstract -init_value** `priority_pointer`

> This command permits all possible initialization values for the signal `priority_pointer` to be explored, which restores the true symmetry to the circuit. In the induction section (below), you will see that this command is useful for proving other aspects of an arbiter.

Multiplexer Abstraction

Multiplexers are frequently used to implement arbitration schemes that involve multiple ports. For the case where the port count is high, multiplexers increase the complexity of the proof and present a problem. However, most arbiter requirements can be represented in terms of `p1` and/or `p2` instead of specifying all ports or a fixed port number directly. Refer to any of the arbiter application notes (for example, "Fairness Arbiter Implementations") to learn about a technique for specifying a few arbiter requirements by using a fixed (but arbitrary) port `p1` as part of your specification.

Since our objective is to identify a bug in the control logic for our arbiter, we can take advantage of the technique we present in this section to reduce the complexity of the multiplexer requirements model (as well as the design itself during the proof).

Consider the following example:

Figure A-1 Multiplexer Abstraction
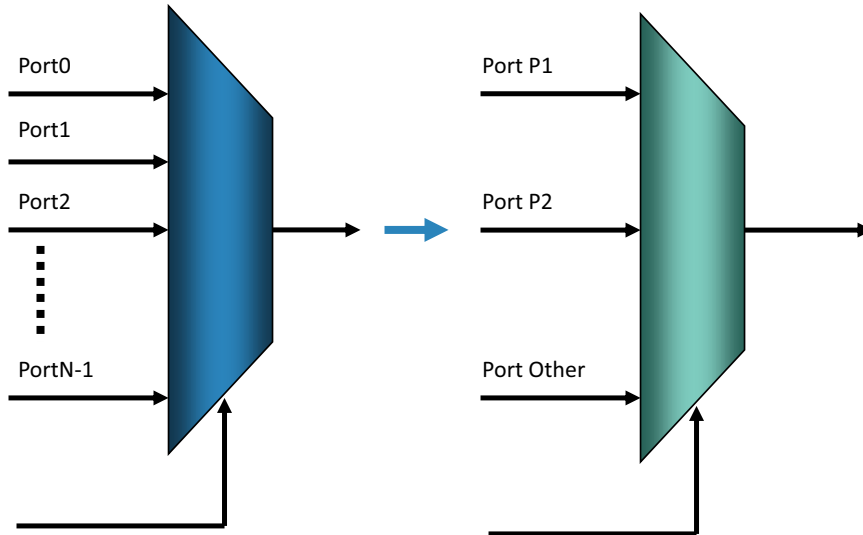
In Figure A-1 on page 40, we illustrate the concept of reducing the complexity of the multiplexer. For this example, we have taken an N-to-1 multiplexer (shown on the left) and reduced it to a 3-to-1 multiplexer (shown on the right). Our abstraction is safe since, for our proof, we only need to distinguish two arbitrary ports `p1` and `p2` (represented as "Port P1" and "Port P2" in the figure). Since we are not distinguishing the behavior of all other ports as part of our requirement, we can represent the behavior of all other ports using a single port we have labeled "Port Other" (refer to Figure A-1 on page 40). JasperGold Apps explores the behavior of the ports we must distinguish (that is, `p1` and `p2`) in the context of the behavior for all the other ports that we do not need to distinguish.

For example, consider the original model:

```
always @(port[0] or port[1] or port[2] or port[3] or port[4] or port[5] or
port[6] or port[7] or round_robin_ptr) begin
  case(round_robin_ptr)
    0: port_out = port[0];
    1: port_out = port[1];
     .
     .
     .
    15: port_out = port[15];
  endcase
end
```

There are two abstraction techniques we can use to simplify this code when proving the requirements in the context of two arbitrary ports, `p1` and `p2`.

■   **Technique 1**: RTL Code Version of Abstracted Multiplier Model

```
wire [2:0] port_other;


always @(port[0] or port[1] or port[2] or port[3] or port[4] or port[5] or
port[6] or port[7] or round_robin_ptr or p1 or p2) begin
   if (round_robin_ptr==p1) port_out=port[p1];
   else if (round_robin_ptr==p2) port_out=port[p2];
   else port_out = port_other;
end
```

Using the above model, you should add the following Tcl commands to your script:

```
assume {port_other!=port[p1]}
assume {port_other!=port[p2]}
```

■   **Technique 2:** Tcl Script Version of Abstracted Multiplexer Model

```
stopat {port_out}
assume {round_robin==p1 => port_out==port[p1]}
assume {round_robin!=p1 => port_out!=port[p1]}
assume {round_robin==p2 => port_out==port[p2]}
assume {round_robin!=p2 => port_out!=port[p2]}
```

FIFO Abstraction

Although in general, FIFOs are more likely to be found in the datapath portion of a design rather than the actual arbiter, there are some arbiter implementations that take advantage of a FIFO structure. Consider a round-robin arbiter that uses registers to store the priority for each port. Every time a grant is provided, the priority changes, which affects at least one of the priority registers, and potentially affects all of the priority registers. This priority scheme using multiple registers, in a way, behaves similar to a FIFO (except it may not always behave in a first-in first-out fashion). Since the set of priority registers can be large, we might need to replace this set of large registers with an abstracted model. The following example demonstrates this technique.

Example

Consider a simple 32-location, circular FIFO-type structure that is keeping track of some type of information (for example, its priority) related to each port of our arbiter. Information enters our FIFO at location `31` and is shifted as new information arrives until it exits at location `0`. Essentially, for an arbiter design, the information indicating high priority at location `0` will move to location `31` using this circular FIFO structure. By using the basic ideas related to `p1` and `p2`, that is, if each individual port of an arbiter does not violate any of its requirements, then an arbiter that consists of multiple ports will never violate any of the requirements, we are able to dramatically simplify the FIFO and reduce the complexity of the design for our proof.[1] Instead of keeping track of the content for every FIFO location, we only need to keep track of the information for `p1` and `p2`, along with their locations within the FIFO. You will see that this reduces our large 32-location FIFO down to four smaller registers and enables us to prove complex properties related to the FIFO controller.

The following Verilog RTL code fragment demonstrates the original RTL code for the FIFO.

```
reg [15:0] fifo0;
reg [15:0] fifo1;
reg [15:0] fifo2;
.
.
.
reg [15:0] fifo31;

always @ (posedge clk) begin
  if (reset) begin
        fifo0 <= 16'h0;
        fifo1 <= 16'h0;

        .
        .
        .
        fifo31 <= 16'h0;
  end
  else if (fifo_en) begin
        fifo31 <= new_info;
        fifo30 <= fifo31;
        fifo29 <= fifo30;
```

---

1. Refer to <u>"What Ports Should You Monitor as Part of Your Requirements?"</u> on page 30 for a discussion of this concept.

```verilog
            .
            .
            .
            fifo0 <= fifo1;
            info_out <= fifo0;
      end
   end
```

The following Verilog RTL code fragment demonstrates our FIFO abstraction technique.

```verilog
   reg [15:0] p1_info;
   reg [15:0] p2_info;
   reg [4:0]  p1_loc;
   reg [4:0]  p2_loc;

   always @(posedge clk) begin
     if (reset) begin
            p1_info <= 16'h0;
            p2_info <= 16'h0;

            //initial position of p1 within the fifo

            p1_loc <= p1_init;

            //initial position of p2 within the fifo

            p2_loc <= p2_init;
      end
      else if (fifo_en) begin
            p1_loc <= p1_loc - 5'h1;
            p2_loc <= p2_loc = 5'h1;
            if (p1_loc <= 5'h0) begin
               p1_info <= new_info;
               info_out <= p1_info;
            end
            else if (p2_loc <= 5'h0) begin
               p2_info <= new_info;
               info_out <= p2_info;
            end
      end
   end
```

Induction

One of the main advantages of using JasperGold Apps is that it provides the trace that demonstrates the requirement violations with a minimum number of clock cycles. This feature makes the full verification a lot faster and a lot easier to debug. However, certain structures can unnecessarily increase the number of cycles required to demonstrate the bug. These situations are usually caused by structures where the function being evaluated follows this type of relationship:

```
f(x+1) = g[f(x)]
```

where `f[0]` is constant. One of the most common examples is a counter. Refer to the "Overview" section of the application note "Fair Arbiter Implementations" to see examples of functions that follow this relationship.

For example, the logic that generates the priority pointer values for a round-robin arbiter follows this relationship. Similarly, in the FIFO abstraction example shown above, the logic that generates location pointer values for `p1` and `p2` follows this relationship. If we can ignore the initial value for these pointer examples, thus allowing all possible initialization conditions, we can shorten the number of cycles required to verify the requirement by using induction.

Example

The round-robin pointers are initialized to zero. To prove the requirements, the design must go through the states containing every pointer value. If the pointer goes from 0 to 31, it will take at least 32 transactions to complete the transfer. However, if we free up the initial value of the pointers to allow any initial values, it only takes one transaction to reach all pointer values from 0 to 31. Consequently, the number of cycles needed to prove the requirements is significantly smaller.

Being able to prove the requirements without needing the initial value also implies that the design is symmetrical. That is, if the design is truly symmetrical, we only need to prove the interface requirements by monitoring one port and fairness requirements by monitoring one pair of ports.

The next section demonstrates how to use induction to simplify a proof that involves a large counter.

Counter Abstraction

Counters are one of the most common components used in a design. If a counter is large, it can be a challenge for any formal verification tool because it can cause the number of evaluation iterations to be very high.

In an arbiter, there are several places that require counters. Dynamic arbiters, for example, typically use a counter to keep track of the arbitration change events. Fortunately, it is usually possible to abstract these counters to reduce the reachable states without compromising the proof.

There are two main counter abstractions: *Counter Induction* and *Counter Reduction*.

Counter Induction

Counter induction uses *induction* to abstract the counter. For example, consider a dynamic priority arbiter that requires us to prove that the credit/weight is incremented, reset, and decremented correctly. If we follow the actual logic where the credit is reset to a fixed value, we will need to go through many cycles to explore all possible credit values. Alternatively, we can partition the requirement into two parts:

- The credit register is initialized correctly.

- The credit register is incremented and decremented correctly.

The first requirement can be modeled as follows:

```
// "credit_init" is the expected initialization value of credit and
// "prev_rst" is the registered version of reset, hence indicate the value
// of credit during the very first cycle

    wire err_credit_reset = prev_rst & (credit!=credit_init);
```

The second requirement can be modeled as follows:

```
    wire credit_increment = credit_inc & (credit!=prev_credit+1);
    wire credit_decrement = credit_dec & (credit!=prev_credit-1);
```

In addition, for the second requirement you must let the initial credit value be free (that is, it is uninitialized and allowed to take on any value). Add the following command to your JasperGold Tcl script to let the initial credit value be free:

```
    abstract –init_value {credit}
```

Counter Reduction

Use the second type of counter abstraction when a specific counter value triggers a specific event (for example, a timeout counter). Since most of the counting values are uninteresting, and are only used as a sequence to generate the next interesting counter value, we can perform an abstraction to reduce the number of states the formal engine needs to evaluate.

For example, assume our design has a counter that calculates the elapsed time before the design must refill its credit for a credit-based arbiter. For this type of design, there are only two counter values that are interesting (that is, zero and the timeout count to generate the credit refill). All other counter values have no influence or impact on the rest of the design. Hence, we can use an abstraction technique that will *fast forward* the counter values that are uninteresting and thus reduce the state space needed to prove the design's high-level requirements.

Original counter model:

```verilog
reg [15:0] count;

always @(posedge clk) begin
  if (rst) count <= 16'h0;
  else if (count == credit_refresh_cnt) count <= 16'h0;
  else count <= count + 16'h1;
end
```

Abstracted counter model:

```verilog
reg abstract_cnt;
reg [15:0] nxt_count;

always @(posedge clk) begin
  if (rst) abstract_cnt <= 1'b0;
  else if (count==credit_refresh_cnt) abstract_cnt <= 1'b0;
  else abstract_cnt <= 1'b1;
end
```

For the abstracted counter, add the following to your JasperGold Tcl script:

```tcl
stopat count

# count can assume any value less than or equal to credit_refresh_cnt
```

```
    assume {abstract_cnt => (count<=credit_refresh_cnt)}

    # count will be set back to zero

    assume {~abstract_cnt => (count == 16'h0)}
```

In our previous example, during reset, the value of `abstract_cnt` is set to zero, which in turn sets `count` to zero with the Tcl assumption. After that, `abstract_cnt` is set to one, which allows `count` to assume any (and all) values between zero and the `credit_refresh_cnt` with the Tcl assumption. If there are no meaningful states for JasperGold Apps to analyze between zero and `credit_refresh_cnt`, then `count` will not arbitrarily assume a value that is less than `credit_refresh_cnt` for any time longer than necessary. Hence, count will assume `credit_refresh_cnt` (an interesting state) sooner without having to sequence through all the counter values. When `count` assumes the value of `credit_refresh_cnt`, `abstract_cnt` resets back to zero, which in turn resets `count` back to zero.

One advantage for this type of counter abstraction is that if we miss some meaningful counts (counts that impact the rest of the design) when we are creating our abstract model, our proof returns a false with a counter example. Thus, this form of abstraction might give you a false negative, but it does not give you a false positive. If there are more counter values needed in our abstracted model, we can then declare more states and map these states onto states in our abstracted model. This type of counter abstraction is also useful when counting the number of remaining credits in the credit-based dynamic arbiter.

Separating Decoding Logic

Separating the decoding logic from the logic used to ensure fairness is especially useful for arbiters that involve dynamic prioritizations and fixed priority arbiters. The essential idea is that for many arbiters there is decoding logic that is responsible for generating the various priority levels based on multiple factors.

For example, assume we are designing an arbiter where the priority of a specific port is increased from normal to high whenever the input FIFO for that port reaches the high watermark. The priority increases further from high to highest when the FIFO is full. All ports with the same priority levels (normal, high, and highest) are arbitrated using a round-robin scheme. However, each priority level has strict priority over the lower priority levels.

To write a high-level requirement for this combined round-robin and priority scheme can be complicated. In addition, proving this combined scheme can be complicated for a formal tool. However, if we separate the verification problem into two parts, we can simplify the coding of our high-level requirement and simplify the proof. The requirement can be partitioned as follows:

■ Verify that the arbitration requirement is met.

■ Verify that the generation of the priority is correct.

For example, to separate the logic that generates the priority from the logic that implements the arbitration scheme, add the following command to your JasperGold Tcl script when you prove that the arbitration requirement is correct:

```
stopat {priority[15:0]}
```

Then, in your requirements model, code the following abstract priority model:

```
// If there is at least one port with high priority, the effective
// requests are the ports with request and high priority.
// Otherwise, it is the same as a request
// if there is no high priority request.

reg [15:0] priority_req = (priority==16'b0) ? (priority&req) : req;

// Use priority_req in place of regular request for any
// requirements that are applicable
```

There are also a few assumptions that might be needed to model the behavior of the abstracted priority model correctly. For example, whenever a priority is high, then the priority should not change to low priority prior to receiving a grant. This assumption requires additional modeling in the requirement as shown:

```
// registered previous value of priority

reg [15:0] prev_priority;

// registered previous value of grant

reg [15:0] prev_gnt;

always @(posedge clk) begin
  if (rst) begin
    prev_priority <= `NUM_PORT'h0;
    prev_gnt <= `NUM_PORT'h0;
  end
  else begin
    prev_priority <= priority;
```

```
        prev_gnt <= gnt;
    end
end
```

Then add the following assumptions to your Tcl script:

```
assume {(~prev_gnt[0] & prev_priority[0]) => (priority[0]==1'b1)}
assume {(~prev_gnt[1] & prev_priority[1]) => (priority[1]==1'b1)}
.
.
.
```