cadence®

Fair Arbiter Implementation

March 2014

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Product JasperGold Apps incorporates software developed by others and redistributed according to license agreement. For further details, see doc/third_party_readme.txt.

Contents

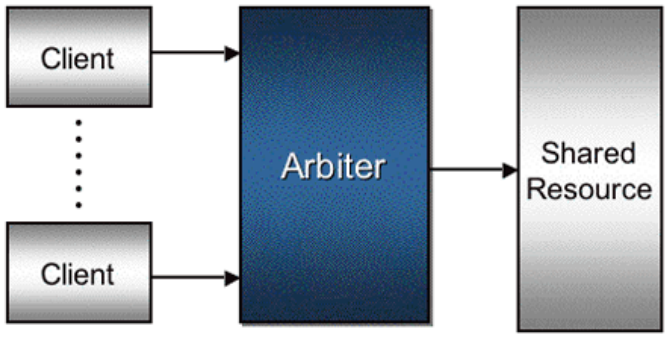
1

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 fair arbiters.

2

Overview

In this application note you will learn about different implementations of fair arbiters. This Overview is useful for understanding the various terms and phrases we use throughout the remainder of this application note.

Fair arbiters are generally implemented by circulating the designated highest-priority port to a neighboring port after each arbitration cycle. This form of arbitration, which is often referred to as *round-robin* arbitration, ensures that every port has an equal opportunity to access a shared resource. Or more importantly, no port is *starved* (that is, no port should go through an extended period of time without being serviced).

Click on the fair arbiters listed below for detailed discussions:

- Pointer-Based Round-Robin
- Group Variation on Pointer-Based Round-Robin
- Register-Based Round-Robin
- Time Division Multiplexed

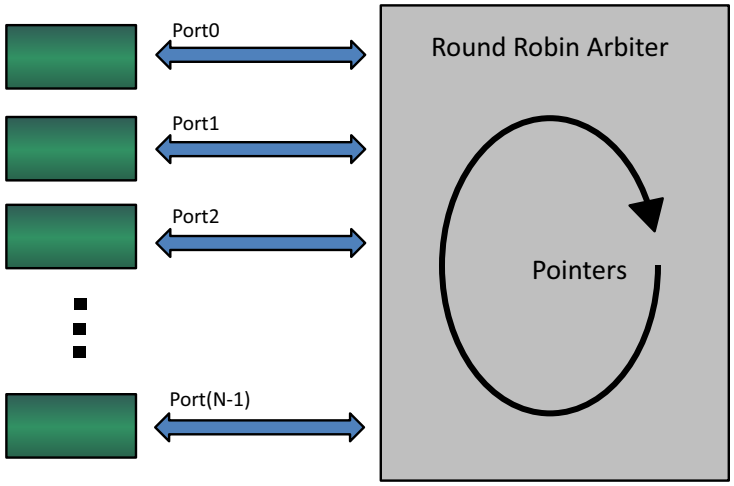
Pointer-Based Round-Robin

Pointer-based round-robin arbitration is based on a circular-moving pointer that identifies the specific port with the highest priority. For example, if port 0 was given a grant on the previous arbitration cycle, then port 1 will be designated the highest priority port for the next arbitration cycle. However, if port 1 has no pending request during the current arbitration cycle, yet port 2 has a pending request, then port 2 will be issued the grant, which means that port 3 will be designated the highest-priority port and port 2 the lowest-priority port on the next arbitration cycle.

The advantage of pointer-based round-robin arbitration over other implementations is its simplicity. That is, after a grant is issued for a specific port, it is only necessary to change the pointer to identify the neighboring port as having the highest priority. The disadvantage of a pointer-based round-robin implementation is that if port 1 is a high-traffic port, port 0 will end up having the lowest priority most of the time. This is due to having continuous pending

requests on port 1, while other ports have infrequent requests, which moves the pointer to the next highest priority (port 2) and designates port 0 as a lower priority port. This situation is usually not an issue, since at the point in time when port 0 issues a request, the fair arbitration scheme guarantees that it will receive a grant within N arbitration cycles, where N represents the number of requesting ports (that is, clients). For example, if the arbiter has eight clients, a pending request for a particular port should never have to wait more than seven arbitration cycles before it is serviced. Otherwise, a different port must have been unfairly granted the bus multiple times.

Figure 2-1 Pointer-Based Round-Robin Arbiter



The round-robin fairness property can be described as follows:

- For any port request service, that particular port should receive a grant within `N` arbitration cycles, where `N` represents the number of requesting ports (that is, clients).

Alternatively, the round-robin fairness property can be specified using a pair of ports (for example, `p1` and `p2`) as follows:

- If port `p1` has a pending request, then port `p2` should never receive two grants prior to port `p1` receiving a grant.

Although `p1` and `p2` represent two fixed ports, we can use the technique from the Strategy section (see "Strategy" on page 31) to cover all pairs of ports.

Group Variation on Pointer-Based Round-Robin

One variation of the pointer-based round-robin arbiter is to group the ports into a set of ports. The arbitration scheme is then modified to first arbitrate in a round-robin fashion between the groups, followed by arbitrating in a round-robin fashion between particular ports contained within a single group. For example, if we designate that ports 0 to 3 belong to group A, ports 4 to 7 belong to group B, ports 8 to 11 belong to group C, and ports 12 to 15 belong to group D, then if port 0 is issued a grant, port 1 will be designated as the highest priority port within group A. However, since group A just received the grant, group B will be designated the highest priority for evaluation during the next arbitration cycle. Hence, group priority is considered prior to port priority.

To verify this special type of round-robin arbiter, use the normal requirements among all ports in the same group. Then consider all requests from the same group as one request ("OR-ing" all the requests). Finally, perform the same round-robin requirements among groups.

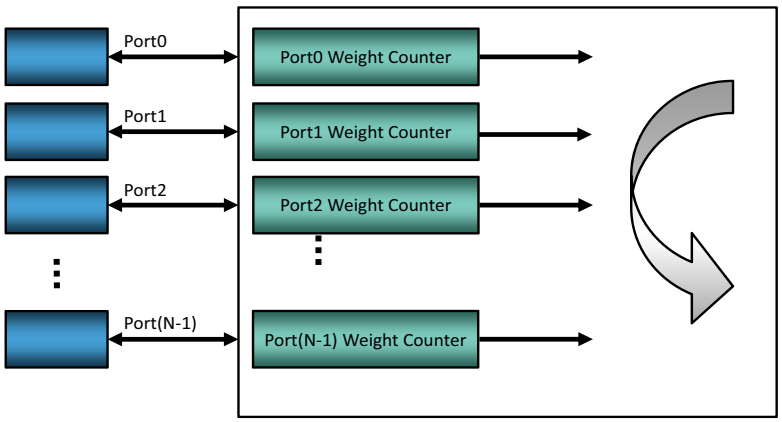
```
wire groupA_req = req[0] | req[1] | req[2] | req[3];
wire groupA_gnt = gnt[0] | gnt[1] | gnt[2] | gnt[3];
wire groupB_req = req[4] | req[5] | req[6] | req[7];
wire groupB_gnt = gnt[4] | gnt[5] | gnt[6] | gnt[7];
.
.
.
```

Perform the same requirements using the request/grant from each group.

Register-Based Round-Robin

The second class of round-robin arbiters is based on the grant provided. Instead of moving the pointers, this scheme moves the priority of the port that has just been granted to the bottom priority without affecting the priority of other ports. Each port has a priority attached to it. If port 0 gets a grant, its priority will be moved to the lowest priority. All ports that had lower priority than port 0 will have their priorities moved up by one. The main advantage for this implementation is that it is not affected by any port that has a higher than normal amount of traffic because a port does not "lose" priority when it does not issue a request. The main disadvantage for this type of implementation is that it requires registers to keep track of the priority levels of each port, which uses more logic gates.

Figure 2-2 Register-Based Weighted Round-Robin



The fairness requirement is very similar to that of pointer-based round-robin. There are minor differences because the arbiter “remembers” which port got the last grant. For example, consider a pair of ports, `p1` and `p2`. Each 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.

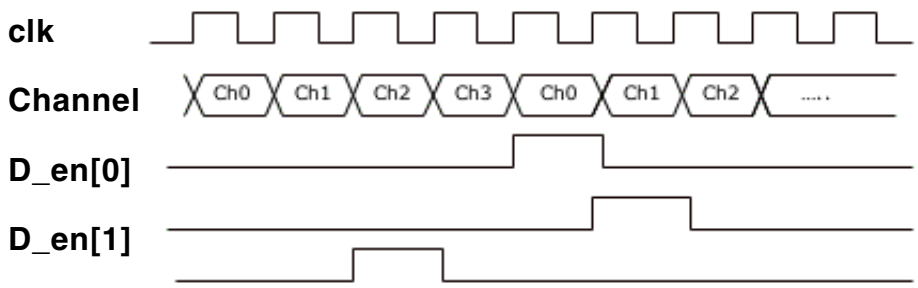
```
reg [1:0]  p1p2_gnt;

always @(posedge clk) begin

// Depending on which port has the
// higher initial priority level

  if (rst) p1p2_gnt <= 2 'b00;
  else if (gnt[p1]) p1p2_gnt <= 2'b01;
  else if (gnt[p2]) p1p2_gnt <= 2'b10;

// p1p2_gnt == 2'b01 indicates that p1 has
// a more recent grant so p2 now has the priority
// p1p2_gnt == 2'b10 indicates that p2 has
// a more recent grant so p1 now has the priority

end
```

```
wire err_fairness_p1 = p1p2_gnt==2'b10 & req[p1] & gnt[p2]; wire err_fairness_p2
= p1p2_gnt==2'b01 & req[p2] & gnt[p1];
```

Time Division Multiplexed

The third class of fair arbiters uses *time-division multiplexing* (TDM) to provide access to each client. In this case, only one port is served at a given time. The active port rotates after a predefined interval of time. The main advantages include its simple design and easily configured bandwidth for each port. To configure, increase the time-interval for the specified port. The main disadvantage is that it is rather inefficient because bandwidth is wasted when one port has no data to be sent and another port has a considerable amount of data pending.

Figure 2-3 Time-Division Multiplexing of a Four-Channel Arbiter



TDM-based arbitration is the simplest to verify because it keeps the sequence of channels (grants) constant. Therefore, the main fairness requirement is to ensure that the channel rotation is as expected.

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. The examples in this section illustrate a few common arbiter interface requirements. Although your specific arbiter interface requirements might differ from the examples in this section, they are a starting point that you can modify to fit your specific needs.

This section includes the following topics:

Interface Description

In addition to the fairness requirements previously described in the Overview (see “Overview” on page 9), 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.

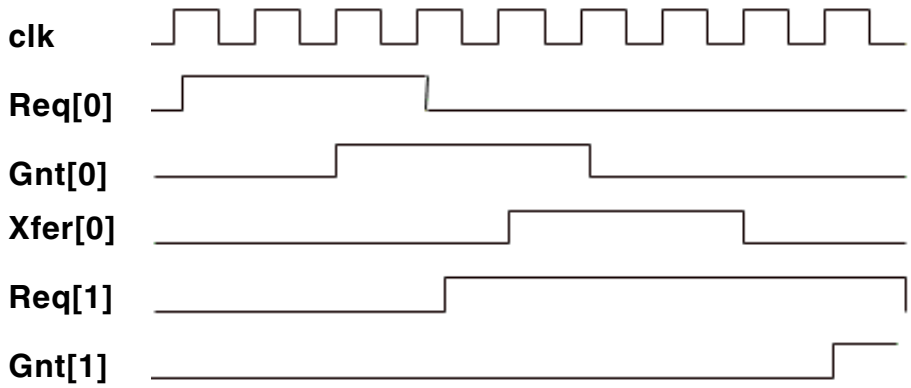
c. The arbiter eventually asserts a grant (Gnt).

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.

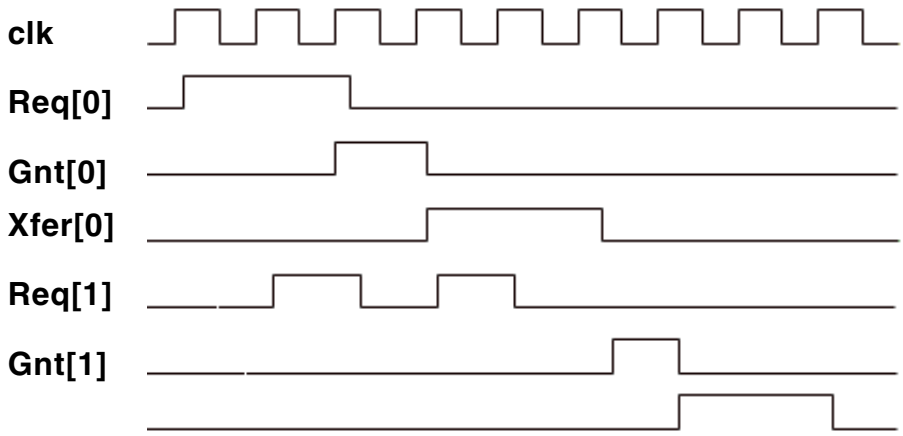
b. Request can be asserted again. Up to four pending requests can be issued.

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. The

need for registration is especially apparent in the fairness requirements discussed in this 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.

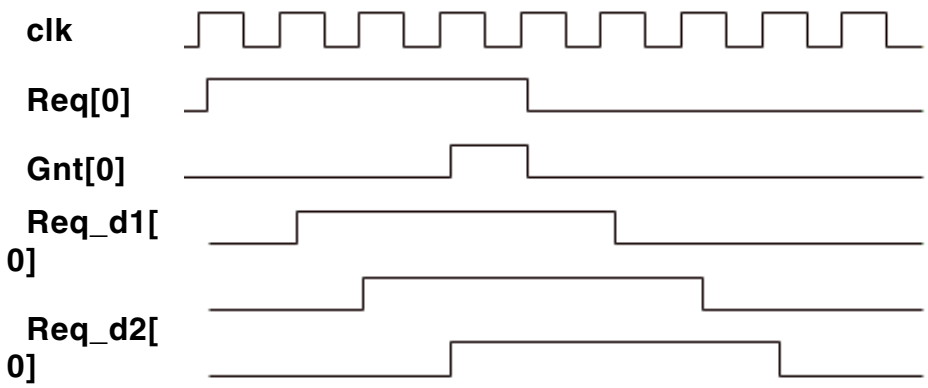
b. Requests take three cycles to propagate to the internal arbiter; hence, there is a minimum three-cycle latency before the request is processed.

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.

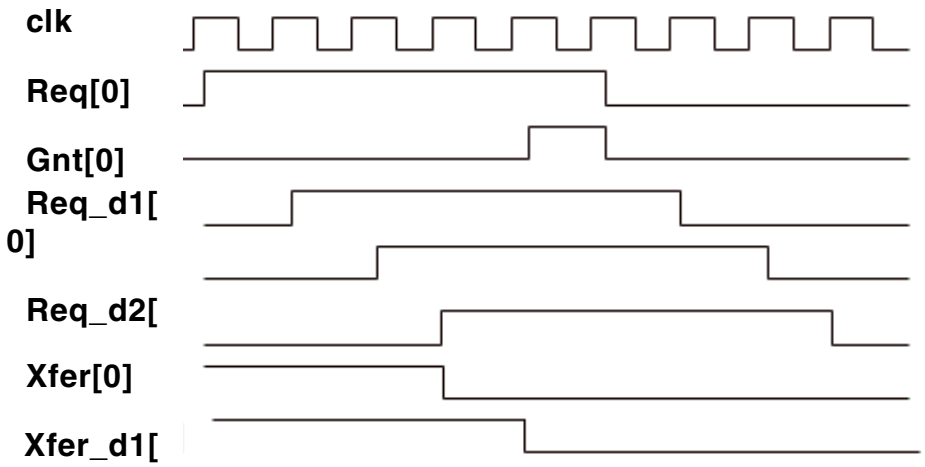
Consider requirement 1: *Grant should not be asserted without a request*. According to Figure 3-3 on page 19, the earliest grant can be asserted is three cycles after a request. Asserting the grant earlier contributes to a violation, even though the request is asserted due to internal latency. To properly 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 20 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];
```

Common Arbiter Checks

To review a requirements model for the three examples in the previous section, see Appendix A, “General Arbiter Requirements.”. This appendix shows how an arbiter interface might behave. Keep in mind that your model will be highly specific to your implementation, but you can use the model we provide as a jump-start to understanding. Review this appendix to learn the concepts you need to understand and modify it for your own implementation.

4

Testplan

This section presents the high-level requirements on your fair arbiter. A formal testplan includes a high-level requirements checklist, which is the list of requirements you plan to formally verify. Requirements tables list the specific signal names coded in the high-level requirements model and a description of what the requirements check.

Refer to the following tables:

- Requirements Set 1 – Common Arbiter Requirements
- Requirements Set 2 – Fairness Requirements

Note: You will create an additional table that lists interface requirements (see "Requirements Set 3 – Arbiter Interface Requirements" on page 22).

Common Arbiter Requirements

Because your requirements will be highly specific to your implementation, it is not possible to offer a requirements table template. However, you can refer to the following table to learn about common arbiter requirements, and to learn how we modeled the requirements in our interface example, see Appendix A, "General Arbiter Requirements,".

Table 4-1 Requirements Set 1 – Common 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.

Fairness Requirements

The following table lists fairness requirements. Click on the links to view the Verilog® model.

Table 4-2 Requirements Set 2 – Fairness Requirements

Requirement Name	Summary
`err_fairness_1`	When a client generates a request, it should receive a grant within `N` arbitration cycles (that is, within `N` issued grants). **Note:** Use `err_fairness_1` if the number of ports is greater than eight. See "Requirements" on page 23 to review the Verilog requirements for `err_fairness_1`. For requirement models in other supported languages, see Appendix B, "VHDL, PSL, and SVA Requirements,"
`err_fairness_2`	When a client generates a request, it should receive a grant within `N` arbitration cycles (that is, within `N` issued grants). **Note:** Use `err_fairness_2` if the number of ports is low (`<=8`). See "Requirements" on page 23 to review the Verilog requirements for `err_fairness_2`, and see Appendix B, "VHDL, PSL, and SVA Requirements," for requirement models in other supported languages.

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

Arbiter Interface Requirements

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 15) for examples of various request/grant interface requirements.

5

Requirements

In this section we use the interface signals (see “Interface Names” on page 16) and the set of requirements identified in the Testplan (see Table 4-2 on page 22) to create the high-level requirements models that monitor design block input and output signals.

This section contains the Verilog® source code for various fair arbiter requirements models.

- Pointer-Based Round-Robin Verilog Requirements
- Register-Based Round-Robin
- Time Division Multiplexed

Note: For requirement models in other supported languages, see Appendix B, “VHDL, PSL, and SVA Requirements,”.

Pointer-Based Round-Robin Verilog Requirements

For a detailed discussion of this type of fair arbiters, see “Pointer-Based Round-Robin” on page 9.

```
/* ------------------------------------------------------------------------ */
/* -                          jasper_pointer_arb                          - */
/* -                                                                      - */
/* ------------------------------------------------------------------------ */

`define NUM_PORT 32

module pointer_arb (clk, rstN, req, gnt, p1, p2, req2gnt_latency);

  input clk;
  input rstN;
  input [`NUM_PORT-1:0] req;
  input [`NUM_PORT-1:0] gnt;
```

```
// Select two ports for verification (arbitrary).

input [4:0]           p1, p2;

// The latency from request asserted to grant asserted

input [2:0]           req2gnt_latency;

reg                   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
        )
  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

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

always @(posedge clk, negedge rstN)
  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

//  Pointer-based round-robin algorithm

//  Approach #1 is recommended if the number of ports is high (>8).
//  Monitor two ports at a time, verify one port at a time,
//  allow activities in all ports.

reg other_gnt;

//  Value of "other_gnt" is not set during reset. This ensures that no
//  assumptions are made about the initial internal priority between p1
//  and p2.

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

//  ************************************************************
//  Formal Testplan: Requirements Set 2: Fairness Requirements
//  APPROACH #1
//  ************************************************************
//  If port p1 sent a request, after which another port got two
//  grants before p1 got a grant, then the fairness requirement
//  for p1 is violated. other_gnt is asserted if port p2 got a

//  grant after req_p1 is asserted. req_p1 refers to the request
//  from p1 delayed by request/grant latency.
//  ************************************************************
```

```
  wire err_fairness_1 = req_p1 & other_gnt & gnt[p2];

  //  ***********************************************************
  //  Formal Testplan: Requirements Set 2: Fairness Requirements
  //  APPROACH #2
  //  ***********************************************************
  //  Approach #2 is recommended if the number of ports is low (<=8).
  //  Monitor all ports, verify one port at a time, allow activities
  //  in all ports. num_gnt refers to the number of non-p1 grants
  //  since req_p1 is asserted. req_p1 refers to the request from
  //  p1 delayed by request/grant latency
  //  ***********************************************************

  reg[4:0] num_gnt;

  always @(posedge clk, negedge rstN)
    if (!rstN) num_gnt <= 5'b0;
    else if (gnt[p1]) num_gnt <= 5'b0;
    else if (req_p1 & gnt!=`NUM_PORT'b0 & ~gnt[p1])
      num_gnt <= num_gnt + 1'b1;

  //  We count the number of non-p1 grants that have elapsed.
  //  If this number is equal to NUM_PORT-1 and the next
  //  grant is still not for p1, there is a fairness violation.


  wire err_fairness_2 = req_p1 & num_gnt==5'b11111 & gnt[p2];

endmodule
```

Register-Based Round-Robin

For a detailed discussion of this type of fair arbiters, see "Register-Based Round-Robin" on page 11.

```
/* ------------------------------------------------------------------------ */
/* -                       jasper_reg_round_robin                         - */
/* -                                                                      - */
/* ------------------------------------------------------------------------ */


`define NUM_PORT 32

module register_arb (clk, rstN, req, gnt, p1, p2, req2gnt_latency);

input clk;
input rstN;
input [`NUM_PORT-1:0] req;
input [`NUM_PORT-1:0] gnt;

//  Select two ports for verification (arbitrary).

input [4:0] p1, p2;

//  The latency from request asserted to grant asserted

input [2:0] req2gnt_latency;

//  wire req_p1;
reg 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)

  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

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

always @(posedge clk)
  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

// Register-based round-robin algorithm

// Monitor two ports at a time. Verify one port at a time.
// Allow activities in all ports.

reg p1p2_gnt;

always @(posedge clk)
    if (!rstN) p1p2_gnt <= 1'b1;
```

```
//  Assume p1 has a higher initial priority.
//  Otherwise, reset it to 0.

      else if (gnt[p2]) p1p2_gnt <= 1'b1;
      else if (gnt[p1]) p1p2_gnt <= 1'b0;

//  ************************************************************
//  Formal Testplan: Requirements Set 2: Fairness Requirements
//  ************************************************************
//  p1p2_gnt refers to the monitored port that got the most
//  recent request. 1 signifies port p2, 0 signifies port p1.
//  req_p1 refers to the request from p1 delayed by request/grant latency.


  wire err_fairness_1 = req_p1 & p1p2_gnt & gnt[p2];

endmodule
```

Time Division Multiplexed

For a detailed discussion of this type of fair arbiters, see "Time Division Multiplexed" on page 13.

```
/* ------------------------------------------------------------------------ */
/* -                           jasper_tdm_arb                             - */
/* -                                                                      - */
/* ------------------------------------------------------------------------ */


`define NUM_PORT 32

module tdm_arb (clk, rstN, gnt, p1, p2, time_slot_en);

  input clk;
  input rstN;
  input [`NUM_PORT-1:0] gnt;

  //  Select two ports for verification (arbitrary).
```

```
  input [4:0] p1, p2;

  //  Enable to move to a new port

  input time_slot_en;

  reg [4:0] channel_num;

  always @(posedge clk)
    if (!rstN) channel_num <= 5'b0;
    else if (time_slot_en)
              channel_num <= channel_num + 5'b00001;

//  ************************************************************
//  Formal Testplan: Requirements Set 2: Fairness Requirements
//  ************************************************************
//  If the channel_num is supposed to be assigned to p1, it should
//  give p1 the grant.


wire err_fairness_1 = channel_num==p1 & gnt[p2];

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, and only if you are unable to complete your proof due to a performance problem.

This section includes the following topics:

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 straightforward. 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 a fairness arbiter requirement, 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. See "Pointer-Based Round-Robin Verilog Requirements" on page 23 for an example of this technique.

Speeding Up Your Proof

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

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

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.
- Coverage on any functions that are separate from fairness requirements.

A

General Arbiter Requirements

```
/* ------------------------------------------------------------------------ */
/* -                          JASPER_request_grant                        - */
/* -                                                                      - */
/* ------------------------------------------------------------------------ */

`define NUM_PORT 8

module req_grant (clk, rstN, req, gnt, xfer);

      input clk, rstN;
      input [`NUM_PORT-1:0] req;
      input [`NUM_PORT-1:0] gnt;
      input [`NUM_PORT-1:0] xfer;

    //  Modeling logic for section2

    //  Example 1

    reg [`NUM_PORT-1:0] prev_req;

    always @(posedge clk)
      if (!rstN) prev_req <= 8'b0;
      else prev_req <= req;

    reg [`NUM_PORT-1:0] prev_gnt;

    always @(posedge clk)
      if (!rstN) prev_gnt <= 8'b0;
      else prev_gnt <= gnt;

    reg [`NUM_PORT-1:0] prev_xfer;
```

```
always @(posedge clk)
  if (!rstN) prev_xfer <= 8'b0;
  else prev_xfer <= xfer;

//  Approach 2

wire [2:0] p1, p2;     //  Assuming up to 16 ports.
                       //  p1 and p2 are any two ports.

//  Count the number of cycles since the signal grant was
//  asserted.

reg [1:0] gnt_ctr;

always @(posedge clk)
  if (!rstN) gnt_ctr <= 2'b11;
  else if (gnt!=`NUM_PORT'b0 & prev_gnt!=`NUM_PORT'b0) gnt_ctr <= 2'b1;
  else if (xfer!=`NUM_PORT'b0) gnt_ctr <= 2'b0;
  else if (gnt_ctr!=2'b0) gnt_ctr <= gnt_ctr + 2'b1;

//  Example 2

//  Number of request_queued using Approach 2.

reg [2:0] req_queued_p1;

always @(posedge clk)
  if (!rstN) begin
    req_queued_p1 <= 3'b0;
  end
  else begin
    if (req[p1]) req_queued_p1 <= req_queued_p1 + 1'b1;
    else if (gnt[p1]) req_queued_p1 <= req_queued_p1 - 1'b1;
  end

//  Example 3

reg req_d1, req_d2, req_d3;     //  Delayed version of req
reg xfer_d1;                    //  Delayed version of xfer
```

```
always @(posedge clk)
  if (!rstN) begin
    req_d1 <= 1'b0;
    req_d2 <= 1'b0;
    req_d3 <= 1'b0;
    xfer_d1 <= 1'b0;
  end
  else begin
    req_d1 <= req[p1];
    req_d2 <= req_d1;
    req_d3 <= req_d2;
    xfer_d1 <= xfer!=`NUM_PORT'b0;
  end


// Constraints section2, Example 1
// Request should be asserted until grant is issued.

wire forbid_req_deassert = ((~prev_gnt & prev_req & ~req)!=`NUM_PORT'b0);

// Transfer should not be asserted until grant is provided.

wire forbid_xfer_assert = ((~prev_gnt & ~prev_xfer & xfer)!=`NUM_PORT'b0);

// Constraints section2, Example 2
// Request should not be asserted if there are four pending requests.

wire forbid_req_limit = req_queued_p1==3'd4 & req[p1];

// Transfer should not be asserted until grant is provided.

wire forbid_xfer_assert = ((~prev_gnt & ~prev_xfer & xfer)!=`NUM_PORT'b0);

// Requirements section2, Example 1, Approach 1
// Grant should be asserted until either four cycles have elapsed or
// after transfer is asserted.

wire err_gnt_deassert = (gnt_ctr!=2'b00 | prev_xfer!=`NUM_PORT'b0) &
prev_gnt!=`NUM_PORT'b0 & gnt==`NUM_PORT'b0;

// Grant should not be asserted until the bus is idle.
```

```
wire err_gnt_assert = prev_xfer!=`NUM_PORT'b0 & prev_gnt==`NUM_PORT'b0 &
gnt!=`NUM_PORT'b0;

// Only one grant can be asserted at one time.

wire err_gnt_one_hot = ((gnt-1'b1 & gnt)!=`NUM_PORT'b0) & gnt!=`NUM_PORT'b0;

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

wire err_gnt_assert_idle = prev_xfer==`NUM_PORT'b0 & prev_req!=`NUM_PORT'b0 &
gnt==`NUM_PORT'b0;

// Grant should not be asserted if there is no request pending to any port.

wire err_gnt_without_req = prev_req==`NUM_PORT'b0 & prev_gnt==`NUM_PORT'b0 &
gnt!=`NUM_PORT'b0;

// Requirements section2, Example 1, Approach 2
// Grant should be asserted until either four cycles have elapsed or
// after transfer is asserted.

wire err_gnt_deassert_2 = (gnt_ctr!=2'b00 | prev_xfer[p1]) &
prev_gnt[p1] & ~gnt[p1];

// Grant should not be asserted until the bus is idle.

wire err_gnt_assert_2 = (prev_xfer[p2]|prev_xfer[p1]) & ~prev_gnt[p1] &
gnt[p1];

// Only one grant can be asserted at one time.

wire err_gnt_one_hot_1 = ((gnt-1'b1 & gnt)!=`NUM_PORT'b0) & gnt!=`NUM_PORT'b0;

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

wire err_gnt_assert_idle_1 = prev_xfer==`NUM_PORT'b0 & prev_req[p1] &
gnt==`NUM_PORT'b0;
```

```
//  Grant should not be asserted if there is no request pending to any port.

wire err_gnt_without_req_1 = ~prev_req[p1] & ~prev_gnt[p1] &
gnt[p1];

//  Requirements section2, Example 2, Approach 2
//  Grant should be asserted for only one cycle at a time.

wire err_gnt_assert_one_cycle = prev_gnt[p1] & gnt[p1];

//  Grant should not be asserted until the bus is idle.

wire err_gnt_assert_2a = (prev_xfer[p2]|prev_xfer[p1]) & ~prev_gnt[p1] &
gnt[p1];

//  Only one grant can be asserted at one time.

wire err_gnt_one_hot_2 = ((gnt-1'b1 & gnt)!=`NUM_PORT'b0) & gnt!=`NUM_PORT'b0;

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

wire err_gnt_assert_idle_2 = prev_xfer==`NUM_PORT'b0 & req_queued_p1!=4'b0 &
gnt==`NUM_PORT'b0;

//  Grant should not be asserted if there is no request pending to any port.

wire err_gnt_without_req_2 = req_queued_p1==4'b0 & ~prev_gnt[p1] &
gnt[p1];

//  Requirements section2, Example 3
//  Grant should not be asserted if there is no request.
//  NOTE: req_d3 indicates the request observed by the arbiter.
//  We don't need to use the assertion of gnt (~prev_gnt & gnt) because
//  grant should only be asserted for one cycle.

wire err_gnt_without_req_3 = ~req_d3 & gnt[p1];

//  Grant should be given to a request within four cycles if bus is idle.

wire err_gnt_bus_idle = req_d3 & ~xfer_d1 & gnt==`NUM_PORT'b0;
```

```
    // Grant is one hot. Use the one from Example 2
    // Grant is only asserted for one cycle.

    wire err_gnt_one_cycle = prev_gnt[p1] & gnt[p1];

endmodule
```

B

VHDL, PSL, and SVA Requirements

This appendix contains the VHDL, PSL, and SVA source code for Pointer-Based Round-Robin fair arbiter requirements. It includes the following sections:

- Pointer-Based Round-Robin VHDL Requirements
- Pointer-Based Round-Robin PSL-Verilog Requirements
- Pointer-Based Round-Robin PSL-VHDL Requirements
- Pointer-Based Round-Robin SVA Requirements

Pointer-Based Round-Robin VHDL Requirements

For a detailed discussion of this type of fair arbiters, see “Pointer-Based Round-Robin” on page 9.

```
--  ------------------------------------------------------------------------ --
--  -                          jasper_pointer_arb                          - --
--  -                                                                      - --
--  ------------------------------------------------------------------------ --

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity pointer_arb is

  generic (
    NUM_PORT : natural := 32
  );

  port (
    clk  : in std_logic;
```

```
    rstN : in std_logic;
    req  : in std_logic_vector(NUM_PORT-1 downto 0);
    gnt  : in std_logic_vector(NUM_PORT-1 downto 0);

    -- Select two ports for verification (arbitrary).
    p1   : in std_logic_vector(4 downto 0);
    p2   : in std_logic_vector(4 downto 0);

    -- The latency from request asserted to grant asserted
    req2gnt_latency : in std_logic_vector(2 downto 0)
  );

end pointer_arb;

architecture req of pointer_arb is

  signal req_p1 : std_logic;

  signal req_d1_p1, req_d2_p1, req_d3_p1, req_d4_p1 : std_logic;
  signal req_d5_p1, req_d6_p1, req_d7_p1 : std_logic;

  signal other_gnt : std_logic;

  signal num_gnt : unsigned(4 downto 0);

  signal err_fairness_1 : boolean;
  signal err_fairness_2 : boolean;

begin  -- req

  -- Selecting the request that corresponds to the grant
  process( req_d1_p1, req_d2_p1, req_d3_p1, req_d4_p1
         , req_d5_p1, req_d6_p1, req_d7_p1
         , req, p1, req2gnt_latency
         )
  begin  -- process

    case (req2gnt_latency) is
      when "000" => req_p1 <= req(conv_integer(unsigned(p1)));
      when "001" => req_p1 <= req_d1_p1;
      when "010" => req_p1 <= req_d2_p1;
```

```
    when "011" => req_p1 <= req_d3_p1;
    when "100" => req_p1 <= req_d4_p1;
    when "101" => req_p1 <= req_d5_p1;
    when "110" => req_p1 <= req_d6_p1;
    when "111" => req_p1 <= req_d7_p1;
  end case;

end process;

-- Registering request to reflect the proper request-to-grant latency
process (clk, rstN)
begin  -- process
  if rstN = '0' then                    -- asynchronous reset (active low)
    req_d1_p1 <= '0';
    req_d2_p1 <= '0';
    req_d3_p1 <= '0';
    req_d4_p1 <= '0';
    req_d5_p1 <= '0';
    req_d6_p1 <= '0';
    req_d7_p1 <= '0';
  elsif clk'event and clk = '1' then  -- rising clock edge
    req_d1_p1 <= req(conv_integer(unsigned(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 if;
end process;

-- Pointer-based round-robin algorithm

-- Approach #1 is recommended if the number of ports is high (>8).
-- Monitor two ports at a time, verify one port at a time,
-- allow activities in all ports.

process (clk)
begin  -- process
  if clk'event and clk = '1' then  -- rising clock edge
    if (req_p1 = '1') then
```

```
        if (gnt(conv_integer(unsigned(p2))) = '1') then
          other_gnt <= '1';
        elsif (gnt(conv_integer(unsigned(p1))) = '1') then
          other_gnt <= '0';
        end if;
      end if;
    end if;
  end process;

  --  ************************************************************
  --  Formal Testplan: Requirements Set 2: Fairness Requirements
  -- APPROACH #1
  --  ************************************************************
  --  If port p1 sent a request, after which another port got two
  -- grants before p1 got a grant, then the fairness requirement
  -- for p1 is violated. other_gnt is asserted if port p2 got a
  -- grant after req_p1 is asserted. req_p1 refers to the request
  -- from p1 delayed by request/grant latency.
  --  ************************************************************

  err_fairness_1 <= req_p1 = '1' and other_gnt = '1' and
  gnt(conv_integer(unsigned(p2))) = '1';

  --  ************************************************************
  --  Formal Testplan: Requirements Set 2: Fairness Requirements
  -- APPROACH #2
  --  ************************************************************
  --  Approach #2 is recommended if the number of ports is low (<=8).
  -- Monitor all ports, verify one port at a time, allow activities
  -- in all ports. num_gnt refers to the number of non-p1 grants
  -- since req_p1 is asserted. req_p1 refers to the request from
  -- p1 delayed by request/grant latency
  --  ************************************************************

  process (clk, rstN)
  begin  -- process
    if rstN = '0' then                  -- asynchronous reset (active low)
      num_gnt <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if (gnt(conv_integer(unsigned(p1))) = '1') then
        num_gnt <= (others => '0');
```

```
      elsif ( req_p1 = '1' and gnt /= (gnt'range => '0')
              and (gnt(conv_integer(unsigned(p1))) = '0')
            ) then
        num_gnt <= num_gnt + 1;
      end if;
    end if;
  end process;

  --  We count the number of non-p1 grants that have elapsed.
 -- If this number is equal to NUM_PORT-1 and the next
 -- grant is still not for p1, there is a fairness violation.

  err_fairness_2 <= req_p1 = '1' and num_gnt = conv_unsigned(31,5) and
  gnt(conv_integer(unsigned(p2))) = '1';

end req;
```

Pointer-Based Round-Robin PSL-Verilog Requirements

For a detailed discussion of this type of fair arbiters, see "Pointer-Based Round-Robin" on page 9.

The following PSL example uses non-determinism to arbitrarily select a request during the proof. Since it is arbitrary, all requests are verified.

The non-deterministic behavior is best modeled by creating a wrapper around your DUV and adding two additional ports, `p1` and `p2`, that are not assigned (which means that JasperGold® Apps can non-deterministically assign any value to these input ports). The ports should be named as follows:

```
    // Used to arbitrarily select two request ports for verification.
     input [4:0] p1, p2;

vunit v_pointer_arb( /* arbiter module name here */ ) {

  default clock = (posedge clk);

  //  Select two ports for verification (arbitrary).

  wire [4:0] p1, p2;
```

```
// p1 and p2 are pseudo constants.

restrict {p1 == prev(p1)[*]};
restrict {p2 == prev(p2)[*]};


// ************************************************************
// Formal Testplan: Requirements Set 2: Fairness Requirements
// APPROACH #1
// ************************************************************
// If port p1 sent a request, after which another port got two
// grants before p1 got a grant, then the fairness requirement
// for p1 is violated. req_p1 refers to the request from p1
// delayed by request/grant latency.
// ************************************************************

endpoint e_req_p1 =
  { req[p1] [3] };
      // change the 3 to request-grant latency of your arbiter

endpoint e_gnt_p1 =
  { gnt[p1] };
endpoint e_gnt_p2 =
  { gnt[p2] };

// If a request is issued on p1, then it must never be the case
// that two grants are issued on p2 before a grant is issued
// on p1.


property p_fairness_1 =
  never { e_req_p1;
          { { e_gnt_p2[=2] } &&
            {(!e_gnt_p1)[*]}
          }
        };
assert p_fairness_1;


// ************************************************************
// Formal Testplan: Requirements Set 2: Fairness Requirements
```

```
  //  APPROACH #2
  //  ***********************************************************
  //  Approach #2 is recommended if the number of ports is low (<=8).
  //  Monitor all ports, verify one port at a time, and allow
  //  activities in all ports.
  //  ***********************************************************

  endpoint e_some_gnt =
    { gnt != 32'b0 };

  //  A sequence of n grants that are not for p1.

  sequence s_no_p1_gnt(const n) =
    { { e_some_gnt[->n] } &&
      { (!e_gnt_p1)[*] }
    };

  sequence s_next_grant_is_p1 =
    { !e_some_gnt[*]; e_gnt_p1 };

  //  If a request is issued on p1 and followed by NUM_PORT-1 grants
  //  that are not for p1, then the next grant must be for p1.


  property p_fairness_2 =
    always { e_req_p1; s_no_p1_gnt( /* NUM_PORT - 1 */ ) } |=>
           { s_next_grant_is_p1 };
  assert p_fairness_2;
}
```

Pointer-Based Round-Robin PSL-VHDL Requirements

For a detailed discussion of this type of fair arbiters, see “Pointer-Based Round-Robin” on page 9.

The following PSL example uses non-determinism to arbitrarily select a request during the proof. Since it is arbitrary, all requests are verified.

The non-deterministic behavior is best modeled by creating a wrapper around your DUV and adding two additional ports, `p1` and `p2`, that are not assigned (which means that JasperGold®

Apps can non-deterministically assign any value to these input ports). The ports should be named as follows.

```
    -- Used to arbitrarily select two ports for verification.
      signal p1, p2 : std_logic_vector(4 downto 0);

vunit v_pointer_arb() {

  default clock is clk'event and clk = '1';

  -- Select two ports for verification (arbitrary).

  signal p1, p2 : std_logic_vector(4 downto 0);


  -- p1 and p2 are pseudo constants


  restrict {p1 = prev(p1)[*]};
  restrict {p2 = prev(p2)[*]};

  -- ************************************************************
  -- Formal Testplan: Requirements Set 2: Fairness Requirements
  -- APPROACH #1
  -- ************************************************************
  -- If port p1 sent a request, after which another port got two
-- grants before p1 got a grant, then the fairness requirement
-- for p1 is violated. req_p1 refers to the request from p1
-- delayed by request/grant latency.
  -- ************************************************************

  endpoint e_req_p1 is
    { req(conv_integer(unsigned(p1))); [*] };

  endpoint e_gnt_p1 is
    { gnt(conv_integer(unsigned(p1))) };
  endpoint e_gnt_p2 is
    { gnt(conv_integer(unsigned(p2))) };

  -- If a request is issued on p1, then it must never be the case
```

```
-- that two grants are issued on p2 before a grant is issued
-- on p1.


property p_fairness_1 is
  never { e_req_p1;
          { { e_gnt_p2[=2] } &&
            {(not e_gnt_p1)[*]}
          }
        };
assert p_fairness_1;

--  ************************************************************
--  Formal Testplan: Requirements Set 2: Fairness Requirements
-- APPROACH #2
--  ************************************************************
--  Approach #2 is recommended if the number of ports is low (<=8).
-- Monitor all ports, verify one port at a time, and allow
-- activities in all ports.
--  ************************************************************

endpoint e_some_gnt is
  { gnt /= (gnt'range => '0') };

--  A sequence of n grants that are not for p1.

sequence s_no_p1_gnt(const n) is
  { { e_some_gnt[->n] } &&
    { (not e_gnt_p1)[*] }
  };

sequence s_next_grant_is_p1 is
  { not e_some_gnt[*]; e_gnt_p1 };

--  If a request is issued on p1 and followed by NUM_PORT-1 grants
-- that are not for p1, then the next grant must be for p1.


property p_fairness_2 is
  always { e_req_p1; s_no_p1_gnt() } |=>
         { s_next_grant_is_p1 };
```

```
  assert p_fairness_2;
}
```

Pointer-Based Round-Robin SVA Requirements

For a detailed discussion of this type of fair arbiters, see "Pointer-Based Round-Robin" on page 9.

The following SVA example uses non-determinism to arbitrarily select a request during the proof. Since it is arbitrary, all requests are verified.

The non-deterministic behavior is best modeled by creating a wrapper around your DUV and adding two additional ports, `p1` and `p2`, that are not assigned (which means that JasperGold® Apps can non-deterministically assign any value to these input ports). The ports should be named as follows:

```
    // Used to arbitrarily select two request ports for verification.
     input [4:0] p1, p2;

module pointer_arb_sva (clk, rstN, req, gnt, p1, p2);

  parameter P_req2gnt_latency=1;    //  request latency
  parameter P_num_ports=32;         //  num requesting ports

  input clk;
  input rstN;
  input [P_num_ports-1:0] req;
  input [P_num_ports-1:0] gnt;

  //  Select two ports for verification (arbitrary).

  input [4:0]           p1, p2;     //  pseudo constants


  //  ***********************************************************
  //  Formal Testplan: Requirements Set 2: Fairness Requirements
  //  APPROACH #1
  //  ***********************************************************
  //  If port p1 sent a request, after which another port got two
  //  grants before p1 got a grant, then the fairness requirement
  //  for p1 is violated.
```

```
//  ************************************************************

sequence req_latency;
  //   @(posedge clk) req[p1] ##[P_req2gnt_latency] 1'b1;
  @(posedge clk)( req[p1] ##3 1'b1);
endsequence

sequence s_2_p2;
  @(posedge clk) (gnt[p2])[=2];
endsequence

sequence s_2_p2_gnt_no_p1;
  @(posedge clk)
    ( req_latency ##1 ( !gnt[p1] throughout s_2_p2) );
endsequence

//   If a request is issued on p1, then it must never be the case
//   that two grants are issued on p2 before a grant is issued
//   on p1.


property p_fairness_1;
  @(posedge clk) not s_2_p2_gnt_no_p1;
endproperty

assert property (p_fairness_1);


//  ************************************************************
//   Formal Testplan: Requirements Set 2: Fairness Requirements
//   APPROACH #2
//  ************************************************************
//   Approach #2 is recommended if the number of ports is low (<=8).
//   Monitor all ports, verify one port at a time, and allow
//   activities in all ports.
//  ************************************************************

sequence s_some_gnt;
  @(posedge clk) (gnt != 32'b0);
endsequence
```

```
  //  A sequence of n grants that are not for p1.

  sequence s_no_p1_gnt;
    @(posedge clk)
      (!gnt[p1] throughout (gnt != 32'b0) [->P_num_ports-1]);
  endsequence

  sequence s_next_grant_is_p1;
    @(posedge clk) ((gnt == 32'b0)[*0:$] ##1 gnt[p1]);
  endsequence

  //  If a request is issued on p1 and followed by NUM_PORT-1 grants
  //  that are not for p1, then the next grant must be for p1.

  property p_fairness_2;
    @(posedge clk) req_latency ##1 s_no_p1_gnt |=>
           s_next_grant_is_p1;
  endproperty

  assert property (p_fairness_2);

  //  ************************************************************
  //  Formal Testplan: General assumption
  //  ************************************************************
  //  req on p1 must remain active until a grant gnt on p1
  //  ************************************************************
  //  NOTE: Depending on your implementation, you might need
  //  to add similar assumptions on other ports.
  //  ************************************************************

  property req_p1_until_gnt_p1;
    @(posedge clk)
      ( $rose(req[p1]) |-> (req[p1])[*0:$] ##1 gnt[p1]);
  endproperty

  assume property (req_p1_until_gnt_p1);

endmodule
```

C

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
- Mulitplexer 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:

```
// 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.

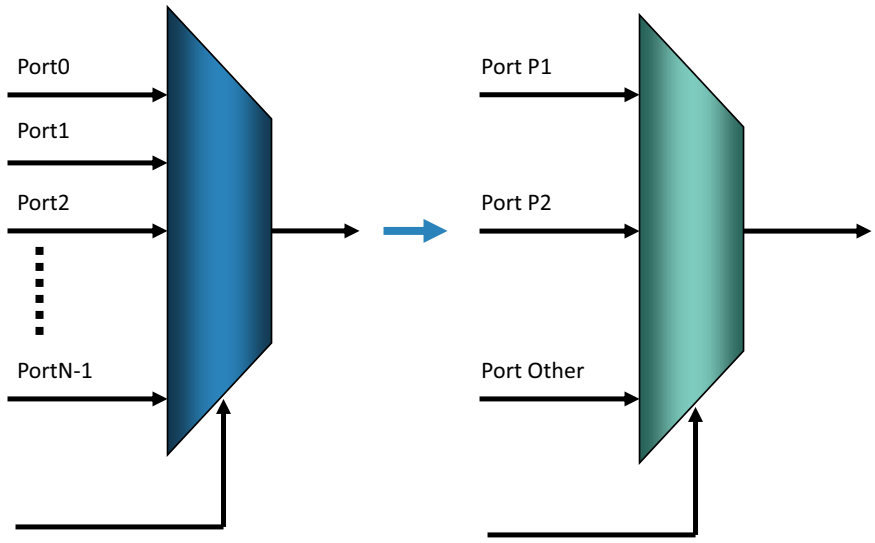
Mulitplexer Abstraction

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 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 C-1 Multiplexer Abstraction



In Figure C-1 on page 60, 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 C-1 on page 60). Jasper tools explore 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 "What Ports Should You Monitor as Part of Your Requirements?" on page 32 for a discussion of this concept.

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

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

```
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 Jasper tools is that they provide 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 "Overview" on page 9 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. (See "Pointer-Based Round-Robin" on page 9.) 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:

```
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:

```
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:

```
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 Jasper tools 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 counterexample. 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)}
.
.
.
```