Arbiters: Design Ideas and Coding Styles

Matt Weber

Silicon Logic Engineering, Inc.

matt@siliconlogic.com

Matthew.D.Weber@ieee.org

ABSTRACT

Arbiters exist in nearly every logic design. This paper will present several design ideas for effectively interfacing to an arbiter and investigate coding styles for some common arbitration schemes.

1.0 Introduction

Many systems exist in which a large number of requesters must access a common resource. The common resource may be a shared memory, a networking switch fabric, a specialized state machine, or a complex computational element. An arbiter is required to determine how the resource is shared amongst the many requesters. When putting an arbiter into a design, many factors must be considered. The interface between the requesters and the arbiter must be appropriate for the size and speed of the arbiter. Also, the coding style used will usually impact the synthesis results.

2.0 Interfacing to an Arbiter

Interfacing to an arbiter can appear very straight forward at first. The requester sends a request (req) signal, and the arbiter returns a grant. However, as the timing margin of the design is tightened, some modifications to this interface may be necessary.

2.1 Example Requesters

2.1.1 State Machine Requester

Requests to an arbiter are generally driven by either a FIFO queue or a state machine. A state machine requester is commonly used when the arbiter is used in a memory controller. If a portion of the memory is used for variable storage, a state machine may need to periodically read and/or write those variables. For a state machine requester, one of the states of the state machine will generally be dedicated to handling the request to grant interface as shown in Listing 1. The req signal in this case is a simple decode of the state vector.

Listing 1

```
case (state_reg)
   `STATE_REQ: begin
      req <=  1'b1;
      mem_read <= 1'b1;
      if (grant) begin
         next_state  <= `STATE_AFTER_REQ;
         data_reg_in <= read_data;
      end else begin
         next_state  <= `STATE_REQ;
         data_reg_in <= data_reg;
      end
   `STATE_AFTER_REQ : begin
endcase
```

2.1.2 FIFO Requester

When the data which is destined for the shared resource is being stored in a FIFO, the request to the arbiter can generally come right from the FIFO. Often, the request can simply be the inverse of the FIFO's empty signal. A convenient way to generate the fifo_empty and fifo_full signals is to compare the read and write pointers of the FIFO. When the pointers are equal, the FIFO is either full or empty. By adding an extra bit to the pointers, we can distinguish between full and empty as shown in Listing 2.

Listing 2

```
// For a 32-position FIFO, five pointer bits are used to access
// the storage array. The sixth bit is used to distinquish between full
// and empty

assign fifo_ptrs_match = (fifo_rdptr[4:0] == fifo_wrptr[4:0]);
assign fifo_full  = fifo_ptrs_match && (fifo_rdptr[5] != fifo_wrptr[5]);
assign fifo_empty = fifo_ptrs_match && (fifo_rdptr[5] == fifo_wrptr[5]);
assign req        = ~fifo_empty;
```

2.2 Basic Request - Grant

2.2.1 Simple Interface

Figure 1 shows a FIFO requester interfacing to an arbiter as described in the previous section. The paths from req to grant through the block labeled “Arbiter” are assumed to be combinational. As we add registers to the request and grant signals in later examples, those registers will be placed in the “Arbitration Logic” block. The “Arbiter” block will continue to represent just the core logic with a combinational path from req to grant.

The timing of the interface is shown in Figure 2. When a new request is pushed onto the FIFO, the FIFO's write pointer gets incremented and the req signal is generated. When the arbitration logic finally grants the request, the FIFO's read pointer gets incremented. If no new pushes have been received, the FIFO's read and write pointers are now equal again and the request is dropped. This type of requester to arbiter interface works well for many arbiters and is very common in smaller, well contained designs.

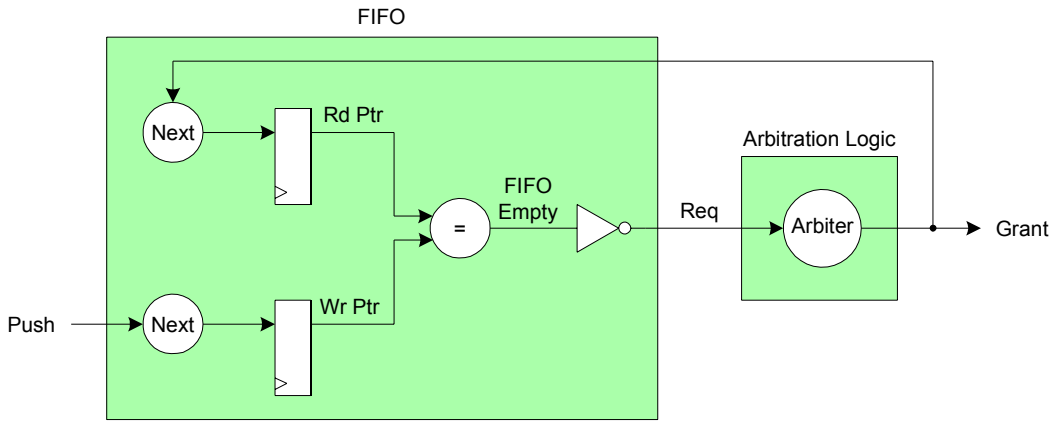

Figure 1. Simple Interface

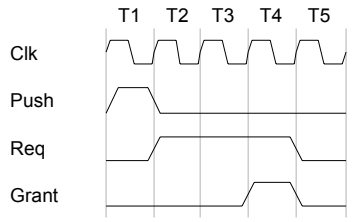

Figure 2. Interface Timing

2.2.2 Register the Request

The interface in Figure 1 shows the grant signal being used to pop an element from the FIFO by incrementing the FIFO's read pointer. The grant signal is typically needed for many other tasks as well, such as multiplexing a datapath, updating the pointer in a round-robin arbiter, or starting a state machine. One disadvantage of the interface shown in Figure 1 is that the timing path for the grant signal begins back at the FIFO's read and write pointers. The timing on the grant signal can be improved by registering the req signal as shown in Figure 3. Instead of comparing the outputs of the read and write pointer registers, the logic compares the inputs to the pointer registers and then registers the result of this compare.

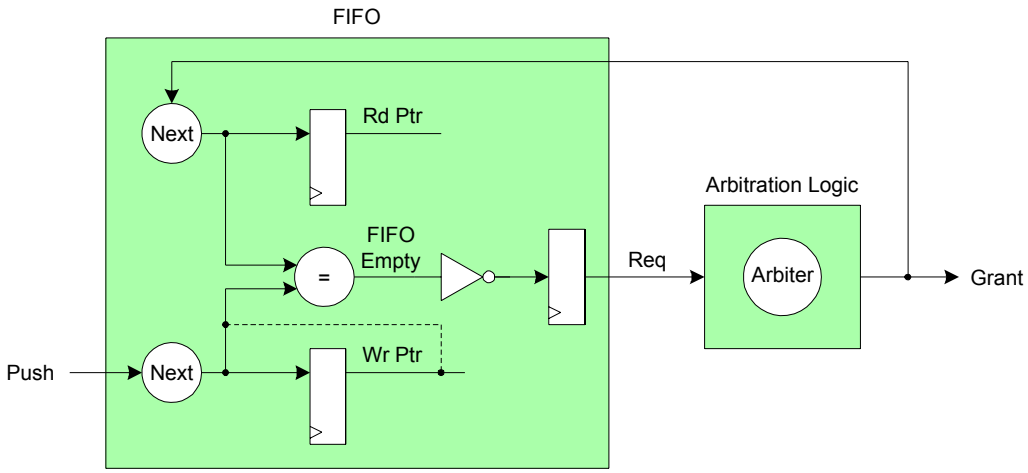

Figure 3. FIFO to Arb Interface with Registered Request

In Figure 1, there is a timing path from RdPtr, through the compare, Arbiter, next pointer logic, and back to RdPtr. Registering the request signal has not really helped this timing path. The timing path has just been moved. In Figure 3, the equivalent timing path is from req, through the Arbiter, next pointer logic, compare, and back to req. The timing of these paths should be roughly equivalent. What has changed is that the pointer compare logic has been removed from the grant signal. This can be important in designs where the grant signal has a large fanout or must travel some distance across the chip. Another change is that the push input now goes through that compare logic on it's way to the req register. If this path has timing problems, the compare can be done against the output of the write pointer register rather than the input as shown by the dashed line in Figure 3. This is an easy change to make and the extra clock cycle of latency that it adds is often not a significant impact on system performance.

2.3 Registering the Request and the Grant

2.3.1 The Problem with Registering the Grant Signal

When timing on the grant signal becomes troublesome, it may be necessary to register the grant signal. The problem with simply registering the grant signal, as shown in Figure 4, is that an extra grant could be given at the end of the request. The timing diagram in Figure 5 shows that because the grant is now returning one cycle later, the request gets dropped one cycle later, and the arbiter may issue a grant for this “late” request.

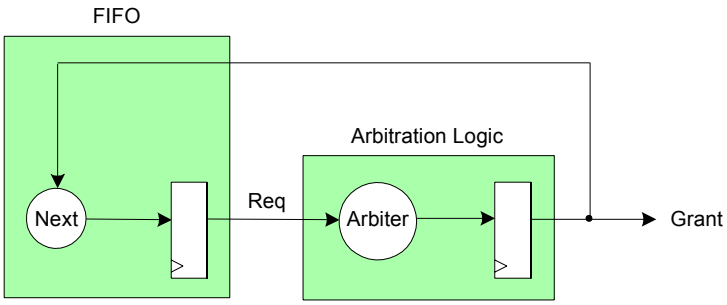

Figure 4. Registered Grant - Bad Design

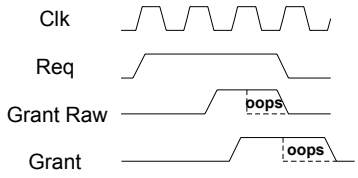

Figure 5. Timing Error of Registered Grant

2.3.2 One Simple Fix

The first way to fix this problem is shown in Figure 6. During the cycle when the grant signal is high, the request is prevented from reaching the arbiter. While this prevents the extra grant from occurring, it also prevents the requester from getting back-to-back grants if the FIFO has several items in it. At first glance, limiting access to alternating clock cycles may appear to be a huge performance penalty. However, if other requesters are active, an arbiter such as a round-robin will generally choose a different requester anyway, so the performance penalty might be negligible.

One case where this solution would not be appropriate is when one or two requesters dominate the bandwidth of the system. For example, a SCSI Ultra320 hard disk controller may have 500 MBytes/sec of bandwidth to its DRAM buffer, but the SCSI interface alone requires 320 MBytes/sec of this bandwidth. Supporting this requires that the arbiter be able to supply back-to-back grants to the FIFO in the SCSI interface logic.

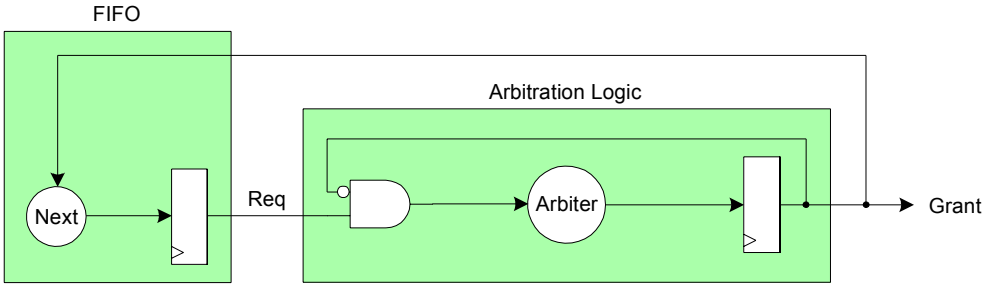

Figure 6. Registered Grant - Okay Design

2.3.3 Regain Performance with the last_req Signal

Some of the performance lost by registering the grant signal can be regained if the requester can see into the future. A last_req signal has been added to the circuit in Figure 7. This signal tells the arbiter if the request will drop after the next grant is received. This circuit will allow the requester to participate in arbitration until it's last request is granted.

In the case of a FIFO requester, the last request occurs when there is only one entry left in the FIFO. With one entry left in the FIFO, the read pointer plus one equals the write pointer . While this could be implemented with combinational logic, it is usually easier to add a “read pointer plus one” register as shown in Figure 7. This “read pointer plus one” register is identical to the original read pointer register except on reset it initializes to one instead of initializing to zero.

The pair of And gates shown in Figure 7 could be included in either the arbitration logic or the FIFO logic, depending on synthesis and physical considerations. Including them in the FIFO logic preserves the traditional req-grant interface between the FIFO and arbiter. Including them in the arbitration logic as shown provides registered inputs to the arbitration logic which may simplify synthesis scripting. Also, the grant signal goes through the And gates, then returns to the arbiter. Therefore, having the And gates in the arbitration logic may help timing if the FIFO and arbiter are widely separated on the chip.

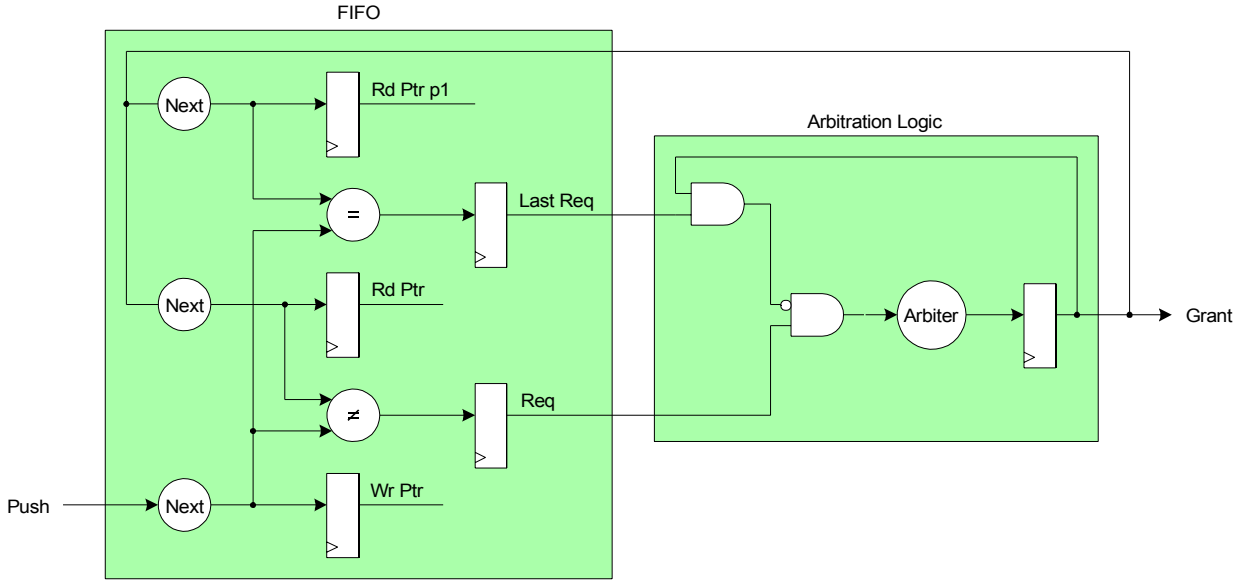

Figure 7. Using a last_req Signal

2.3.4 Add a Small Queue to the Arbitration Logic

An alternative to adding the last_req signal described above is to add a small queue to the arbitration logic for each requester. This is shown in Figure 8. Access to the shared resource is now a three step process. First, the FIFO sends the request. Second, the request is acknowledged when the arbitration logic accepts it into its queue. Third, the grant signal is sent when the arbiter selects this requester.

The FIFO read pointer is still incremented by the grant signal, but it no longer is used in the generation of the req signal. Instead, a new pointer called rdptr_sent is added. This pointer is incremented when the FIFO makes a request and the queue in the arbitration logic is not full.

This approach becomes useful if the requesting FIFOs and the arbitration logic are separated from each other on the chip. With the last_req approach shown in Figure 7, req and last_req are top level signals and will have some delay traveling across the chip. Then these signals still need to get through the arbiter before being captured at the grant register. With the design in Figure 8, the arbiter is getting its inputs from the nearby queue, rather than from the far-away FIFO.

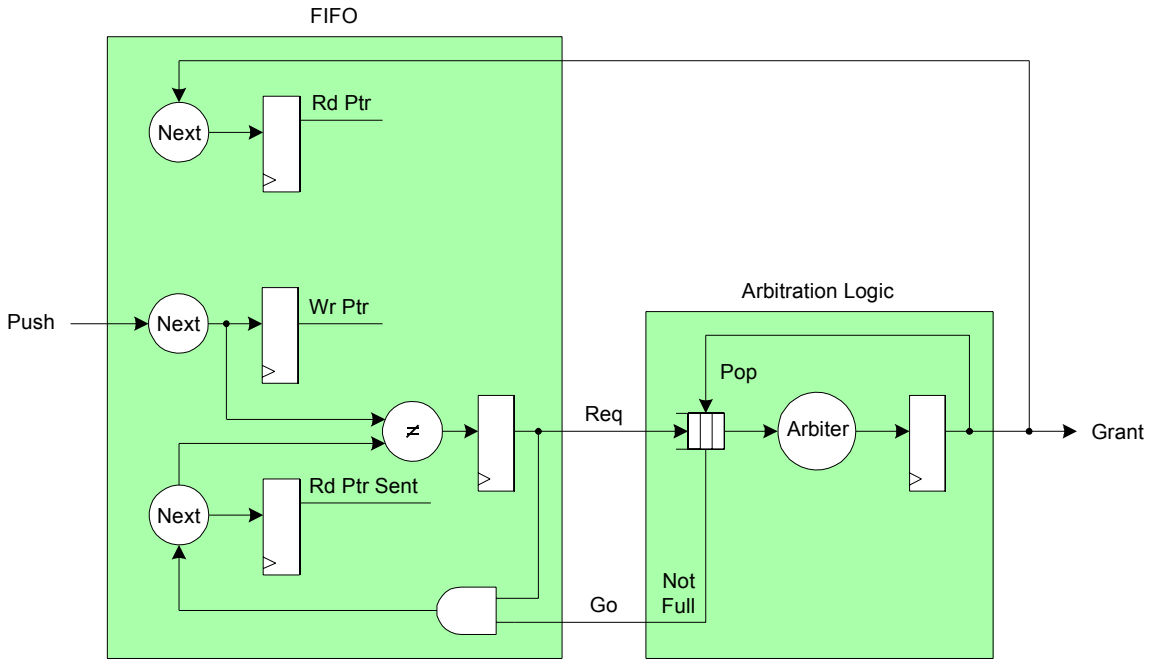

Figure 8. Adding a Queue to the Arbitration Logic

2.4 Big Chips - Registered Repeaters on Req and Grant

One disadvantage of the previous two designs is that an extra signal has been added to the FIFO to Arbiter interface. By adding one more pointer to the FIFO, we can return to a two signal req-grant interface. The new pointer is called rdptr_limit and is shown in Figure 9. At reset, while the other pointers are initialized to zero, this pointer is initialized to the depth of the queue that has been added to the arbitration logic. To prevent an overrun of the queue in the arbitration logic, the rdptr_sent counter is not allowed to proceed beyond the rdptr_limit value. As grants are received from the arbitration logic, rdptr_limit is incremented allowing more requests to be sent.

An important side benefit of this design is that it is tolerant of any additional latency that might be added to the request or grant signals. In process technologies of 0.18 micron and below, and clock speeds in excess of 200 MHz, signals might reach less than halfway across the chip before needing to be registered. Figure 9 shows a single registered repeater on both the req and the grant signals. While these registered repeaters would “break” any of the previous designs discussed, this design will continue to function regardless of the number of repeaters required.

To allow the FIFO to stream requests properly, the queue in the arbitration logic needs to be at least as large as the total number of registers on the path from the output of the req resister, through the arbiter, and back to the req register's input. For the example shown in Figure 9, the queue depth should be at least five (Req repeater + Arbiter Queue + Grant register + Grant repeater + Read pointer limit register).

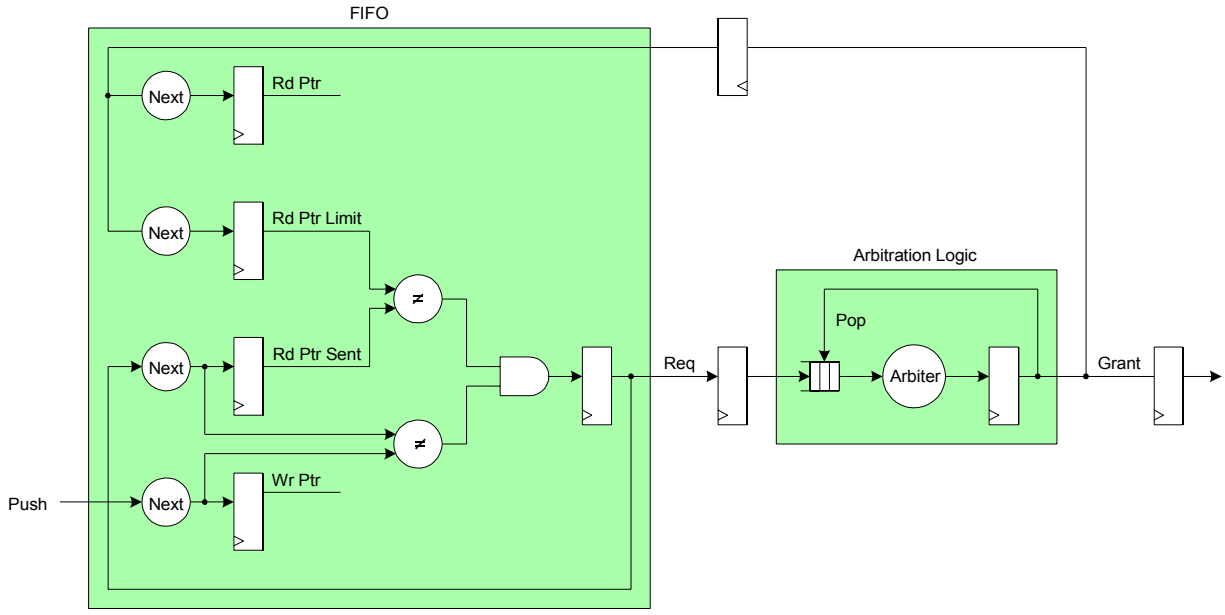

Figure 9. Latency Independent Design

2.4.1 A Few Words about FIFO Pointers

The functionality that was added to the FIFOs in Figures 7, 8, and 9 could have been implemented with up/down counters instead of the pointers that were shown. For example, rdptr_plus1 and rdptr_sent could have been replaced with a count of how many elements were in the FIFO. This counter would increment on a push and decrement on grant (see Figure 7), go (see Figure 8), or req (see Figure 9). The last_req register in Figure 7 would then simply get loaded with next_count = 1. In Figure 8, a req could be sent as long as the count was not equal to zero. Similarly, the rdptr_limit pointer in Figure 9 could be replaced by a "Number of Reqs Available" counter which resets to the arbitration logic's queue depth, decremented on req, and incremented on grant.

While using up/down counters like these would be functionally correct, it is generally less efficient for both timing and area. A compare versus a constant (e.g., count = = 1) will be smaller and faster than a compare between two variables (e.g., next_wrptr != next_rdptr_sent). However, this benefit is usually more than offset by the increased area and path delay of an up/down counter versus a simple incrementing counter.

Another choice for implementing this functionality is to code the counters as one-hot shift registers. This type of implementation can be very fast. However, for deep FIFO's, the area required may be undesirable.

3.0 Simple Priority Arbiter

3.1 Description

One common arbitration scheme is the simple priority arbiter. Each requester is assigned a fixed priority, and the grant is given to the active requester with the highest priority. For example, if the request vector into the arbiter is req[N-1:0], req[0] is typically declared the highest priority. If req[0] is active, it gets the grant. If not, and req[1] is active, grant[1] is asserted, and so on. Simple priority arbiters are very common when choosing between just a few requesters. For example, a maintenance port may always be lower priority than the functional port. ECC corrections may always be higher priority than all other requests.

Priority arbiters are also often used as the basis for other types of arbiters. A more complex arbiter may reorder the incoming requests into the desired priority, run these scrambled requests through a simple priority arbiter, then unscramble the grants which come out. Several examples of this can be seen in the round-robin arbiters as discussed in Section 4.0.

3.2 Coding Styles

3.2.1 Coding a Priority Arbiter with a Case Statement

The most common way of coding a priority arbiter is with a case statement as shown in Listing 3.

Listing 3

```
casez (req[3:0])
   4'b???1 : grant <= 4'b0001;
   4'b??10 : grant <= 4'b0010;
   4'b?100 : grant <= 4'b0100;
   4'b1000 : grant <= 4'b1000;
   4'b0000 : grant <= 4'b0000;
endcase
```

This coding style works fine in small cases. However, when a larger priority encoder is needed, generally as part of a more complex arbitration scheme, it gets to be a lot of typing and mistakes are easy to make.

3.2.2 Coding a Priority Arbiter with Three Assign Statements

For a much simpler method of coding a priority arbiter, first look at the logic that is required as shown in Listing 4.

Listing 4

```
grant[0] =  req[0];
grant[1] = ~req[0] &  req[1];
grant[2] = ~req[0] & ~req[1] & req[2];
...etc...
```

A bit in the grant signal will be active if its corresponding request signal is active, and there are no higher priority requests. By introducing an intermediate term called higher_priority_reqs, the arbitration can be coded with three assign statements. If desired, it can even be parameterized as shown in Listing 5.

Listing 5

```
parameter N = 16;  //  Number of requesters

// For example, higher_pri_reqs[3] = higher_pri_reqs[2] | req[2];
assign higher_pri_reqs[N-1:1] = higher_pri_reqs[N-2:0] | req[N-2:0];
assign higher_pri_reqs[0]     = 1'b0;
assign grant[N-1:0]           = req[N-1:0] & ~higher_pri_reqs[N-1:0];
```

3.2.3 Using Design Ware

Designers with a license for the Synopsys DesignWare Foundation library have another choice for building a priority arbiter. DW_arbiter_sp is a parameterized arbiter with a fixed priority scheme. This component includes several features which may be useful at times, or they can be removed by Design Compiler. The mask feature prevents some requesters from participating in the arbitration. The lock feature allows the granted requester to lock the arbiter and continue to receive grants, regardless of the state of other requesters, until the lock input is released. The park feature, which can be disabled by setting the appropriate instantiation parameter, sets a default grant if there are no active requesters. The DesignWare arbiter component includes registering of the grant signal, so using the park feature can save a clock cycle if the first requester to come on after an idle period is the “parked” requester. One potential disadvantage of the DesignWare component is that its registers are implemented with asynchronous resets, which sometimes cause difficulty in static timing and testability.

3.3 Synthesis Setup

The coding styles investigated were generally implemented for three different sizes of arbiters: four requesters, sixteen requesters, and sixty-four requesters. The DesignWare component supports a maximum of thirty-two requesters, so it was only implemented for arbiters of four and sixteen requesters. Because the DesignWare component includes a register on the grant signal, the FIFO to Arbiter interface from Figure 7 was used. The designs were synthesized using a 0.18 micron technology. Listings 6 and 7 show the major portions of the build and constraint scripts used with Design Compiler v2000.11.

Listing 6 - compile.tcl

```
# Read in FIFO code
foreach SUB_DESIGN $SUB_DESIGNS {
   set V_SRC [format "%s%s" $SUB_DESIGN ".v"]
   analyze -format verilog $V_SRC
}
# Read in Arbiter code
set V_SRC [format "%s%s" $TOP_DESIGN ".v"]
read_verilog $V_SRC
current_design $TOP_DESIGN
source $CON_DIR/$CON_FILE
link
uniquify
compile -map medium
# Get rid of any design ware and sub designs
ungroup -all -flatten
compile -incremental
# Separate FIFO timing paths from Arbiter timing paths
group-path -name FIFO -to [req_fifo*req*/*D*]
```

Listing 7 - constrain.tcl

```
# Use 0.3ns for flop->output. Although unrealistic, add 0ns
# for top level nets so critical path stays in arbiter
# instead of moving to FIFO.
set IN_DELAY 0.3
set OUT_DELAY [expr $CLOCK_PERIOD - 0.3]
set CLOCK_PORT [get_ports clk]
set RESET_PORT [get_ports rst]
# Assume reset will have buffer tree built for it later
set_wire_load_model -name NONE $RESET_PORT
set_drive 0 $RESET_PORT
# Constrain flop -> flop paths
create_clock -period $CLOCK_PERIOD -name CLK $CLOCK_PORT
set_dont_touch_network [get_clocks CLK]
# Constrain input -> flop paths
create_clock -period $CLOCK_PERIOD -name IO_VIRTUAL_CLK
set input_list [remove_from_collection [remove_from_collection [all_inputs]
$CLOCK_PORT] $RESET_PORT]
set_input_delay -max $IN_DELAY -clock IO_VIRTUAL_CLK $input_list
group_path -name INPUT -from $input_list
# Constrain flop -> output paths
set_output_delay -max $OUT_DELAY -clock IO_VIRTUAL_CLK [all_outputs]
```

Since the primary objective was to see how fast the designs would run, synthesis was run iteratively, each time with a faster clock speed until the “fastest” design was achieved for each size of arbiter. This was then used as the target frequency for the final synthesis run which gave the results shown in Tables 1 through 3.

3.4 Synthesis Results

Tables 1, 2, and 3 show the timing, area, and compile time required for each of the designs. While the design which used the DesignWare component was slower, the “case” and “assign” coding styles were generally comparable. Because it is easier to code, the “assign” coding style is the preferred choice for priority arbiters.

Table 1: Timing Results for Priority Arbiters
(Delay of Longest Path in ns)

Arbiter Size	64	16	4
DesignWare	- - -	2.26	1.51
Case	1.60	1.51	0.97
Assign	1.59	1.50	1.08

Table 2: Area Results for Priority Arbiters

Arbiter Size	64	16	4
DesignWare	- - -	8250	2035
Case	32778	8101	2014
Assign	34260	8576	2166

Table 3: Compile Time Results for Priority Arbiters

Arbiter Size	64	16	4
DesignWare	- - -	7 min	2 min
Case	27 min	7 min	2 min
Assign	21 min	8 min	3 min

4.0 Round-Robin Arbiter

4.1 Description

The key shortcoming of priority arbiters is that, in very busy systems, there is no limit to how long a lower priority request may need to wait until it receives a grant. A round-robin arbiter on the other hand allows every requester to take a turn in order. A pointer register is maintained which points to the requester who is next. If that requester is active, it gets the grant. If not, the next active requester gets the grant. The pointer is then moved to the next requester. In this way, the maximum amount of time that a requester will wait is limited by the number of requesters.

4.2 Coding Styles

There are many different ways that a round-robin arbiter can be coded. Contrary to the results seen with priority arbiters, the coding style used to implement a round-robin arbiter can have a significant effect on the synthesis results obtained. Several possibilities for coding a round-robin arbiter are explored.

4.2.1 Coding a Big Blob

The first method of coding a round-robin arbiter is shown in Listing 8. The arbiter is implemented with nested case statements. I actually taped out a chip with an arbiter coded like this....once. There are a few side effects of this coding style. The most noticeable side effect is called carpel-tunnel-vision syndrome. This condition is characterized by sore fingers and wrists caused by failing to look around enough to see a better way of coding your design.

Listing 8

```
always @ ( /*AUTOSENSE*/pointer_reg or req) begin
   case (pointer_reg) // synopsys full_case parallel_case
      2'b00 :
         if      (req[0])  grant = 4'b0001;
         else if (req[1])  grant = 4'b0010;
         else if (req[2])  grant = 4'b0100;
         else if (req[3])  grant = 4'b1000;
         else grant = 4'b0000;
      2'b01 :
         if      (req[1])  grant = 4'b0010;
         else if (req[2])  grant = 4'b0100;
         else if (req[3])  grant = 4'b1000;
         else if (req[0])  grant = 4'b0001;
         else grant = 4'b0000;
      2'b10 :
         if      (req[2])  grant = 4'b0100;
         else if (req[3])  grant = 4'b1000;
         else if (req[0])  grant = 4'b0001;
         else if (req[1])  grant = 4'b0010;
         else  grant = 4'b0000;
      2'b11 :
         if      (req[3])  grant = 4'b1000;
         else if (req[0])  grant = 4'b0001;
         else if (req[1])  grant = 4'b0010;
         else if (req[2])  grant = 4'b0100;
         else  grant = 4'b0000;
   endcase // case(req)
end
```

4.2.2 Rotate + Priority + Rotate

Perhaps the most common method of coding a round-robin arbiter is built on top of a simple priority arbiter. The requester that is pointed to by the round-robin pointer is shifted to the highest priority position, and the other requests are rotated in behind it. This rotated request vector is then sent through a simple priority arbiter. The grant vector from the priority arbiter is then "unrotated" to come up with the round-robin arbiter's final grant signal. This is shown in the block diagram of Figure 10.

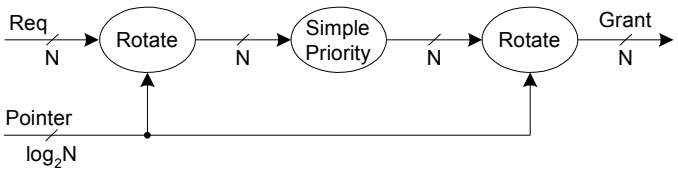

Figure 10. Round-Robin Arbiter: Rotate + Priority + Rotate

The rotate can be coded with a case statement as shown in Listing 9. A more compact method using Verilog's shift operator is shown in Listing 10. The synthesis results reported in Section 4.3 used the shift operater.

Listing 9

```
always @ ( /*AUTOSENSE*/pointer_reg or req) begin
   case (pointer_reg) // synopsys full_case parallel_case
      2'b00 : req_shifted[3:0] = req[3:0];
      2'b01 : req_shifted[3:0] = {req[0],req[3:1]};
      2'b10 : req_shifted[3:0] = {req[1:0],req[3:2]};
      2'b11 : req_shifted[3:0] = {req[2:0],req[3]};
   endcase // case(pointer_reg)
end // always @ (...
```

Listing 10

```
//  The shift operator fills in vacated bits
//  with zeros. We would like it filled in with
//  the bits that were pushed out. This is implemented
//  by concatenating req onto itself, doing a shift,
//  then taking the rightmost bits.
assign req_shifted_double[31:0] = {req[15:0],req[15:0]} >> pointer_reg;
assign req_shifted[15:0] = req_shifted_double[15:0];
```

4.2.3 Muxed Parallel Priority Arbs: Fast, but Big

Another method of constructing a round-robin arbiter for N requesters using N simple priority arbiters arranged in parallel is shown in Figure 11. The “rotate i” blocks which precede the simple priority arbiters rotate the req to the right by “i” positions. These blocks contain no logic; they are wires only. The inputs to the mux are then the grant vectors for every possible value of the round-robin pointer. The pointer controls the mux which selects which of the intermediate grant vectors will actually be used. Both the mux in this example and the rotate in the previous design can be implemented with a 2:1 mux tree with lgN stages. Therefore, this design, with only one mux tree in the critical path, is expected to be faster than the previous design. Unfortunately, for arbiters with a large number of requesters, the area of this design is expected to blow up due to the large number of priority arbiters it contains.

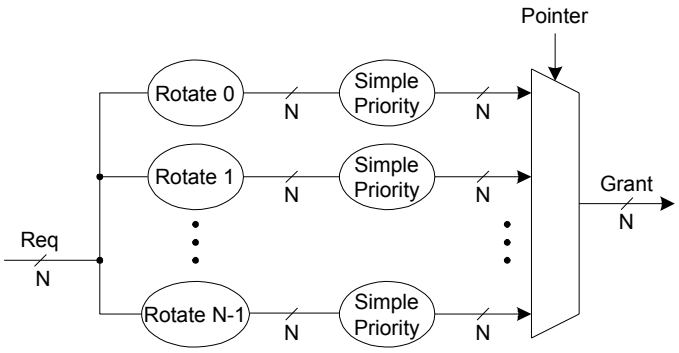

Figure 11. Round-Robin Arbiter: Parallel Priority Arbiters

4.2.4 Using Two Simple Priority Arbiters with a Mask

A round-robin design which generally gives good results for both area and timing is shown in Figure 12. This design uses two priority arbiters. One of the priority arbiters is fed with the entire request signal, while the other one first masks out any requests which come before the one selected by the round-robin pointer. The mask is built from the round-robin pointer as shown in Listing 11. If a grant is selected by the upper arbiter, the mux chooses that grant for the final result, otherwise the result from the lower arbiter is used. Since NoMask = 0 implies that MaskGrant = 0, the mux can be simplified somewhat as shown in the lower portion of Figure 12.

The path from req to grant has now been reduced to the simple priority arbiter, two And gates, and an Or gate. Therefore, if the req to grant path is the critical path, this design is expected to have the best performance of the designs presented here.

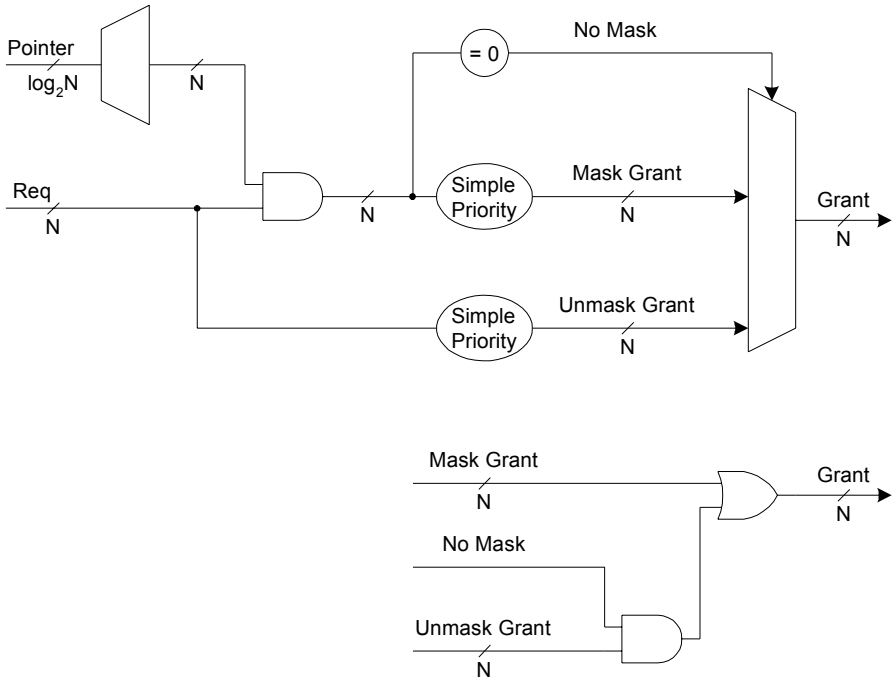

Figure 12. Round-Robin Arbiter: "Mask" Method

Listing 11

```
// Build the mask
always @ ( /*AUTOSENSE*/pointer_reg) begin
   case (pointer_reg) // synopsys full_case parallel_case
      2'b00: req_mask <= 4'b1111;
      2'b01: req_mask <= 4'b1110;
      2'b10: req_mask <= 4'b1100;
      2'b11: req_mask <= 4'b1000;
   endcase
end
```

4.2.5 DesignWare

The Synopsys DesignWare Foundation library also includes an arbiter with a dynamic priority scheme called DW_arbiter_dp. In this arbiter, each of the N requesters sends not only a request, but also a ceil($\log_2$N) bit priority to the arbiter. By properly controlling the priority inputs from each requester, this arbiter could be used to build a round-robin arbiter. However, the extra flexibility makes comparisons to the other round-robin designs unfair so the DesignWare part was not included in this study.

4.3 Synthesis Results

Tables 4, 5, and 6 show the timing, area, and compile time required for each of the designs.

Table 4: Timing Results for Round-Robin Arbiters

Arbiter Size	64	16	4
Mux	3.20	2.09	1.60
Mask	2.90	2.03	1.57
Shift	4.33	2.84	1.74
Blob	- - -	1.97	1.58

Table 5: Area Results for Round-Robin Arbiters

Arbiter Size	64	16	4
Mux	35786	8739	2184
Mask	29588	7736	2226
Shift	37076	8991	2187
Blob	- - -	7758	2175

Table 6: Compile Time Results for Round-Robin Arbiters

Arbiter Size	64	16	4
Mux	67 min	10 min	3 min
Mask	26 min	7 min	2 min
Shift	90 min	24 min	2 min
Blob	- - -	7 min	2 min

As expected, the mask and mux coding styles were the fastest implementations. Because the sixty-four requester version of the mux design starts with sixty-four priority arbiters in parallel, the area of this design was expected to be much higher than the mask design. The results show the mux design with about 20% greater area than the mask design for an arbiter with sixty-four requesters. Also, Design Compiler worked for about two and one-half times as long on the mux design as it did on the mask design.

Design Compiler was able to optimize the “blob” coding style to similar speed and area results as the mask design for four and sixteen requesters. However, a sixty-four requester version of this coding style was not implemented due to the amount of tedious typing it would have required.

The surprising result was the poor results seen with the shift coding style. With sixty-four requesters, its performance was 50% slower than the other coding styles. It also had the largest area and the longest compile time.

4.4 Common Variations

4.4.1 Round-Robin Pointer Implementations

For some architectures, there may be an advantage to having the round-robin pointer implemented as a one-hot vector. Synthesis results for using a one-hot pointer are shown in Tables 7, 8, and 9. Also, the mask coding style could benefit from having the mask value stored as the pointer, rather than needing to calculate it from the pointer. This is also shown in Tables 7, 8, and 9 as the mask_expand design.

Table 7: Timing Results for Round-Robin Arbiters

Arbiter Size	64
mask_onehot	2.72
mask_expand	2.70
mux_onehot	2.70

Table 8: Area Results for Round-Robin Arbiters

Arbiter Size	64
mask_onehot	27684
mask_expand	27430
mux_onehot	34192

Table 9: Compile Time Results for Round-Robin Arbiters

Arbiter Size	64
mask_onehot	34 min
mask_expand	13 min
mux_onehot	49 min

For both the mask and mux designs, one hot encoding of the round-robin pointer was beneficial for both area and timing. The mask_expand design, however is the fastest and smallest of all the round-robin designs investigated. It is also the fastest to compile and requires the fewest number of lines to code. This is because the pointer update equations can be built almost entirely from existing logic. If the two priority arbiters which are used in the mask design are coded with the "three assign" technique shown in Listing 5, the next value of the pointer register is the higher_pri_reqs signal from either the masked or unmasked arbiter. The final code for this round-robin arbiter is shown in Listing 12.

Listing 12 - mask_expand round-robin arbiter

```
// Simple priority arbitration for masked portion
assign req_masked = req & pointer_reg;
assign mask_higher_pri_reqs[N-1:1] = mask_higher_pri_reqs[N-2: 0] |
req_masked[N-2:0];
assign mask_higher_pri_reqs[0] = 1'b0;
assign grant_masked[N-1:0] = req_masked[N-1:0] & ~mask_higher_pri_reqs[N-1:0];

// Simple priority arbitration for unmasked portion
assign unmask_higher_pri_reqs[N-1:1] = unmask_higher_pri_reqs[N-2:0] | req[N-2:0];
assign unmask_higher_pri_reqs[0] = 1'b0;
assign grant_unmasked[N-1:0] = req[N-1:0] & ~unmask_higher_pri_reqs[N-1:0];

// Use grant_masked if there is any there, otherwise use grant_unmasked.
assign no_req_masked = ~(|req_masked);
assign grant = ({N{no_req_masked}} & grant_unmasked) | grant_masked;

// Pointer update
always @ (posedge clk) begin
   if (rst) begin
      pointer_reg <= {N{1'b1}};
   end else begin
      if (|req_masked) begin // Which arbiter was used?
         pointer_reg <= mask_higher_pri_reqs;
      end else begin
         if (|req) begin // Only update if there's a req
            pointer_reg <= unmask_higher_pri_reqs;
         end else begin
            pointer_reg <= pointer_reg ;
         end
      end
   end
```

```
end
```

4.4.2 Round-Robin Pointer Updating

There are three basic methods for updating the round-robin pointer:

1. After a grant, increment the pointer.
2. After a grant, move the pointer to the requester after the one which just received the grant.
3. After a grant, move the pointer to the first Active requester after the one which just received the grant.

The first method is sometimes considered unfair because a single requester can receive back-to-back grants, even if other requesters are active. For example, if requesters 7, 9, and 10 are active, and the pointer is on requester 2, requester 7 will receive six grants in a row before the pointer is beyond it and requesters 9 and10 will be allowed access.

This fairness issue is fixed in the second method. Advancing the pointer to the location after the grant automatically puts the recently granted signal as the lowest priority request on the next cycle. This method was used for the results reported in Tables 4 through 9.

Some of the arbiter's performance statistics can be improved slightly by using the third method. Continuing with the example above, after the grant is given to requester 7, method two will update the pointer to requester 8, while method three will update the pointer to requester 9. If requester 8 becomes active in this cycle, method two will give it an immediate grant, while method three will continue to grant requesters 9 and 10 before coming back around to requester 8. The average latency of all requesters will be the same in either case. However, the latency variance will be less with method three, and the max latency may be less with method three. Whether or not this translates into improved system performance depends on the system's requirements.

One of the difficulties with implementing the third pointer update method is meeting timing on the pointer update. A common method essentially requires going through two arbitrations to determine the next pointer value. First, the traditional arbitration is run to determine the grant vector. Then, the req from the winning requester is negated and the second arbiter is run to determine which of the remaining valid requesters (if any) is the first one after the granted requester. Because this path runs through two arbiters, it often becomes the critical path.

An alternate method which achieves the same results as method 3 is shown in Figure 13. In this design, the pointer updates like method 2, but some logic has been added so that the arbiter will give the same results as a method 3 update. This is achieved by running two arbiters in parallel. The lower arbiter uses the current request vector as its inputs. The upper arbiter uses as its inputs the request vector from the previous cycle, except for the requester which was granted and any requests which have been dropped. If there are any results from the upper arbiter, its results are used, otherwise the standard arbiter results are used. This design adds only a 2:1 mux and a couple of And gates to the timing path. This should be much easier to close timing than the design which had two arbiters in series.

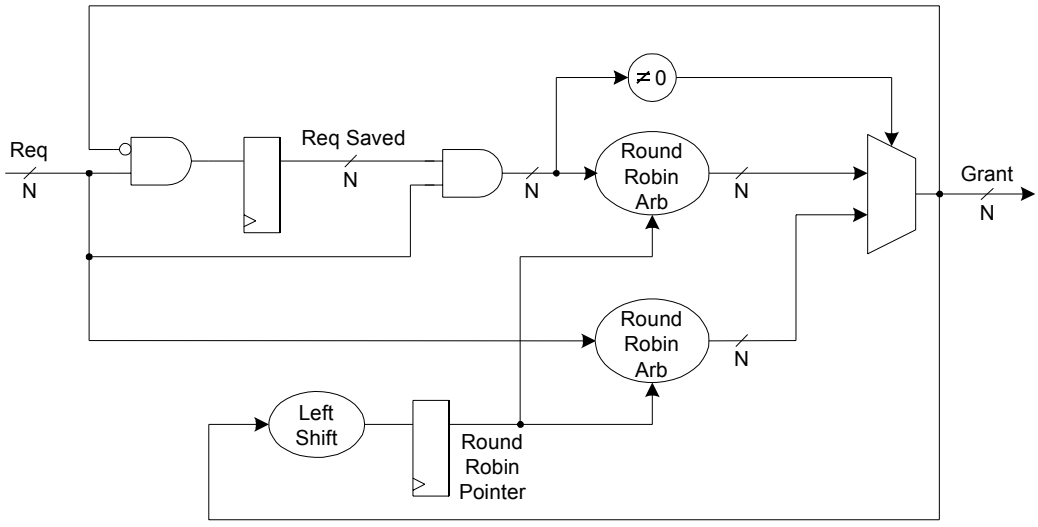

Figure 13. Alternate Use of Round-Robin Pointer

4.0 Conclusions

This paper presented several different methods for interfacing to an arbiter. Which method is the best one to use for a particular design depends on the size and speed of the chip that is being built. Options were presented which cover a wide range of potential applications.

Two common arbitration schemes were then investigated. For priority arbiters, it was shown that the primary consideration for coding style is ease of writing, reading, and maintaining the code. A simple three-line implementation was shown to accomplish these goals. For round-robin arbiters, the coding style can greatly influence the quality of synthesis results. For all metrics, the “mask” coding style with the round-robin pointer stored as the mask, was shown to be superior to other common coding styles.

References

[1] Pankaj Gupta, “On the Design of Fast Arbiters”, Oct 2, 1997.

[2] Synopsys, Inc., “DesignWare Foundation Library Databook, Volume 2”, v2000.11, pp 111-126.

[3] Jonathan Chao, “Saturn: A Terabit Packet Switch Using Dual Round-Robin”, IEEE Communications Magazine, vol 38, no 12, December 2000, pp 78-84.