SVA for statistical analysis of a weighted work-conserving prioritized round-robin arbiter

Ben Cohen, ben@SystemVerilog.us 4/27/2022

1.1 OVERVIEW

In the 70's I worked on a project that demonstrated the value of statical analysis of computer systems. The language used was ECSS, the *Extendable Computer System Simulator*; it was developed by the Rand Corporation for NASA with continuing work sponsored by the Air Force under Project RAND. As a superset language of SIMSCRIPT II, ECSS offered a powerful English-like syntax for describing computer hardware characteristics and the operations performed by software work-loads.

Interestingly enough, SystemVerilog can perform statistical analysis of computer systems, though it is rarely used in this manner. I strongly believe that kind of performance analysis is very important because it reaffirms the needed requirements. To demonstrate those statistical performance concepts by example, I used a weighted work-conserving prioritized round-robin arbiter when exposed to random requests with various configurations and input profiles. The term "work-conserving" refers to the ability of a round-robin arbiter to skip requestors that do not have any data to process and move directly to the next requestor who has valid data to process. This method of arbitration is more efficient as it does not "waste" time on idle requestors

Readers of this paper may appreciate the design approach angle for the arbiter, the usage of assertion coverage along with SystemVerilog `covergroup` and `coverpoint`.

1.2 REQUIREMENTS

Since this document is not an IP and the goal of this paper is to demonstrate performance evaluation, I am providing a loose set of requirements along with a high-level view of the architecture. I am not writing a formal requirement document.

1.2.1 Summary

Arbiters play an important role when multiple requests are sent to access a single resource. The thesis paper by Aung Toe *Design and Verification of a Round-Robin Arbiter* address the various types of arbiters[i].

An initial and vague set of requirements for this design came from a post at the VerificationAcademy. The user needed help with the design of a 16-bit round-robin arbiter where the priority level changes every four clocks. I modified those requirements by adding the notion of weighted work-conserving prioritized round-robin arbitration. After an initial statistical simulation of such an approach, it became clear that having *the priority level change every four clocks* made little sense because it gave little advantage to a prioritized port particularly when requests are interspersed in intervals. The modeling for changing the next-to-be-served priority level every *n* clocks was more like Russian roulette. It became clear that this was a poor requirement, but having a priority level change in a weighted manner after a request is served (i.e., granted the resource) made more sense.

This arbiter has a 16-bit request (`req`) bus, a priority configuration in groups of 4 (`pgrp15_12, pgrp11_8, pgrp7_4, pgrp3_0`), and a 16-bit grant (`grant`) bus; After any active request, the requester with the highest priority is given a grant. Any priority slot that has no requests is skipped for the granting process, but not for the determination of the next priority level. In other words, if a high-priority requestor is not requesting access to the resource, the next active high-priority requestor will be given the grant. Following a grant, the highest priority level is sustained at that level for up to 4 total consecutive grants, depending upon the priority configuration. When that count times out, the priority level is decremented to indicate the next requestor who would have the highest priority.

The group configuration for requests is defined by inputs “`pgrp15_12=3; pgrp11_8=0;  pgrp7_4=0; pgrp3_0=0;`”, as they address a priority weight for ranges of requests. A value of 3 for a group means that the priority level stays at that state for 4 consecutive grants; a value of 2 is for 3 consecutive grants, and so forth. For example, if the group configuration for requests is “`pgrp15_12=3; pgrp11_8=0;  pgrp7_4=0; pgrp3_0=0;`” then after every grant, the priority level will cycle as follows:

```
15, 15, 15, 15, 14, 14, 14, 14, 13, 13, 13, 13, 12, 12, 12, 12,
11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0,
15, 15, 15, 15, 14, 14, ….
```

The priority delineates two bounds for the request: an upper bound from bit 15 up to the current priority, and a lower bound to be computed first from the current level down to bit zero. For example, if the priority level is at bit 10, then if there is a request in bits 10 through 0 (i.e., `req[10:0] > 0`) then the request with the highest weight within that bound gets the grant. However, if there is no request in the lower bound then the request with the highest weight within the upper bound (i.e., `req[15:11]`) gets the grant.

This approach allows a request with higher weight more chances to acquire the resource. However, as previously stated, if a higher priority requestor fails to make a request in its time slot and a lower priority requestor has a request in that same slot, that lower priority requestor will be granted the resource, and the countdown for its time to stay at that level drops by one.

The model assumes that once a grant is provided, the acquired resource or the resource will flag a `busy` status, thus preventing another grant until the `busy` flag is released. Note that this `busy` feature could block the system and bring it to a deadlock. For this model, the resource sevice time is constrained to a service time from 1 to 7 (`servtime inside {[1:7]}`). Further analysis can modify that constraint as needed by the system.

1.2.2 Arbiter interface

Signal	Description	
req[15:0]	**Input:** 16 bits, The request bus, prioritized from high to low, with bit 15 as the highest. Priority is weighted and is a round-robin	arb_scoreboard req busy sc_grnt reset_n Clk Prty_level Grant Logic Priority level pgrp15_12 pgrp11_8 pgrp7_4, pgrp3_0 Highest priority logic
busy	**Input:** 1 bit, When busy is active (1’b1), no grant shall be provided	
sc_grnt[15:0]	**Output:** 16 bits, The grant. It is one-hot or zero.	
reset_n	**Input:** 1 bit, Active low reset	
clk	**Input:** 1 bit, 50% duty cycle clock	
Prty_level	**Output:** 4 bits, priority level used for statistical analysis. It identifies the bit with the current highest priority.	
pgrp15_12, pgrp11_8, pgrp7_4, pgrp3_0,	**Input**: 2 bits for each group. There are 4 groups. It identifies the number of active cycles a priority should hold at that level after a grant. Once that number is exhausted, the priority level can change. Each group is for 4 req bits. For example, pgrp15_12 addresses the holding priority for requests for bits 15 through 12. The values indicate the following: 00 Priority level held for 1 grant occurrence 01 Priority level held for 2 grant occurrences 10 Priority level held for 3 grant occurrences 11 Priority level held for 4 grant occurrences	

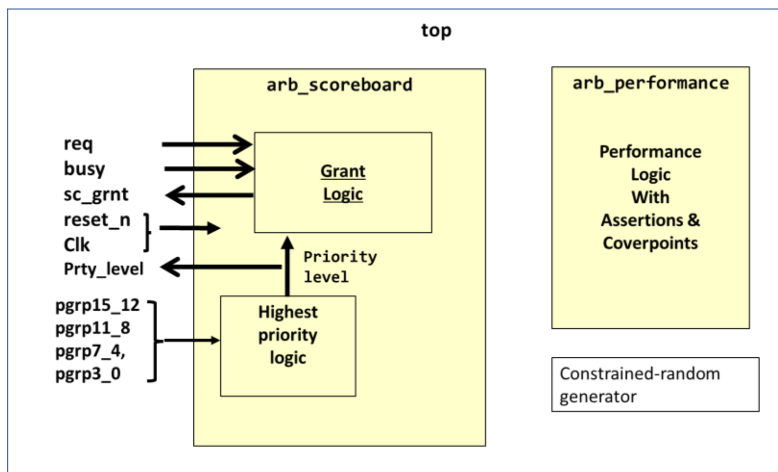
1.3 ARBITER DESIGN

This arbiter is designed as a scoreboard element to be used for the analysis of its performance when exposed to various scenarios. The architecture consists of the following sub-blocks:

- Highest priority logic
- The grant logic. That is supported by two functions:
    - The `grant_lower_bound` that addresses the winner of a request in the range between the current priority level and zero,
    - The `grant_upper_bound` that addresses the winner of a request in the range between 15 and the current priority level plus one,

1.4 THE TESTBENCH

The testbench includes the scoreboard arbiter, a performance logic, and a constrained-random generator.



1.4.1 Performance logic

The performance logic makes use of SVA to create endpoints or calls to functions to trigger the `covergroup sample` command. Below is a property to compute the delay between `req(j)` and `sc_grnt(j)`. Those delays are stored in the variable array

```
int req2grant[15:0]; // For each bit, stores delays from req to grant
```

The index "j" for the property below originates in a `generate` loop to address each bit if the `req` and `sc_grant`. A `cover` statement along with a `covergroup` is used to create the necessary performance statistics.

```
    property p_req2grant_delay; // Used to count the cycle delays
        // from each req to each corresponding grant
        // At the grant, the function call provides the sampling of the covergroup
        int delay;
        @(posedge clk) ($rose(req[j]), delay = 1)  |->  ##1
           (!sc_grnt[j], delay++)[*0:$] ##1
           (sc_grnt[j], setdelay(delay, j));
     endproperty

   function automatic void setdelay(int delay, k);
        req2grant[k]=delay;
        cg_inst.sample();   // for covergroup
    endfunction

   covergroup cg_req2grant_length(); // @ (s_req2grant_delay.triggered);
```

```
        req2grant_cp: coverpoint req2grant[j]
        {bins Delay1_4 =   {[1 : 4]}; // counts number of occurrences with this delay range
         bins Delay5_8 =   {[5 : 8]};
         bins Delay9_12 =  {[9 : 12]};
         bins Delay13_16 = {[13 : 16]};
         bins Delay17_30 = {[17 : 30]};
         bins Delay31_50 = {[31 : 60]};
          }
          prty_level_cp: coverpoint prty_level; // counts the priority level when ec_grnt
          sc_grnt_cp:  coverpoint sc_grnt[j];  // counts number of sc_grant for req[j]
    endgroup
    cg_req2grant_length cg_inst = new();
```

Note: The sample function to trigger the covergroup could have been originated by an endpoint of a sequence.
For example:

```
sequence s_req2grant_delay;
  @(posedge clk) $rose(req[j]) ##1 sc_grnt[j][->1];
endsequence
covergroup cg_req2grant_length() @ (s_req2grant_delay.triggered);
```

For this model I chose to initiate the sample of the covergroup from the function setdelay. That function is called from within the sequence_match_item in the property. This is because the property is needed anyway to calculate the delays between requests and grants.

1.4.2 Top-level and constrained-random generator

The top-level instantiates the modules and emulates the environment in terms of how requests are asserted, and emulates a busy for the usage of resources once a grant is provided. For example, this code makes sure that each request is held until it is granted. Once granted, the request is deasserted immediately and then triggers a service time to create the busy signal.

```
generate for (genvar i=0; i<16; ++i) begin: the_gen
    always_ff @(posedge clk  or sc_grnt[i]) begin
        if(r_req[i]) req[i] <= 1'b1;
        if(sc_grnt[i])   begin
           req[i] = 1'b0;
           service(i, servtime);
        end
    end
```

The creation of the requests (req), the number of clocks between (v_space), and service time by the resource is defined below with the following constraints. Note that by using a distribution profile for each bit of the req it is possible to create various kinds of probability distributions. In the case below, bit 15 of the request occurs more frequently than the other bits. Specifically, bit 15 is active 12.5% of the time, whereas bits 11 through 0 are each active 3.1%, and bits 14 through 12 are inactive.

```
if (!randomize(v_space, servtime) with {
        v_space < 10; // between requests
        servtime inside {[1:7]};
      }) `uvm_error("MYERR", "This is a randomize error");
```

```
       if (!randomize(v_req) with {
         v_req[15] dist {1'b1 := 4, 1'b0 := 32};
         v_req[14] dist {1'b1 := 0, 1'b0 := 32};
         v_req[13] dist {1'b1 := 0, 1'b0 := 32};
         v_req[12] dist {1'b1 := 0, 1'b0 := 32};
         v_req[11] dist {1'b1 := 1, 1'b0 := 32};
         v_req[10] dist {1'b1 := 1, 1'b0 := 32};
         v_req[9] dist {1'b1 := 1, 1'b0 := 32};
         v_req[8] dist {1'b1 := 1, 1'b0 := 32};
         v_req[7] dist {1'b1 := 1, 1'b0 := 32};
         v_req[6] dist {1'b1 := 1, 1'b0 := 32};
         v_req[5] dist {1'b1 := 1, 1'b0 := 32};
         v_req[1] dist {1'b1 := 1, 1'b0 := 32};
         v_req[3] dist {1'b1 := 1, 1'b0 := 32};
         v_req[2] dist {1'b1 := 1, 1'b0 := 32};
         v_req[1] dist {1'b1 := 1, 1'b0 := 32};
         v_req[0] dist {1'b1 := 1, 1'b0 := 32};
       }) `uvm_error("MYERR", "This is a randomize error");
```

1.5 RESULTS

The following two simulation runs provided interesting performance statistics.
Using the following configuration and statistical patterns in 300,000 cycles:

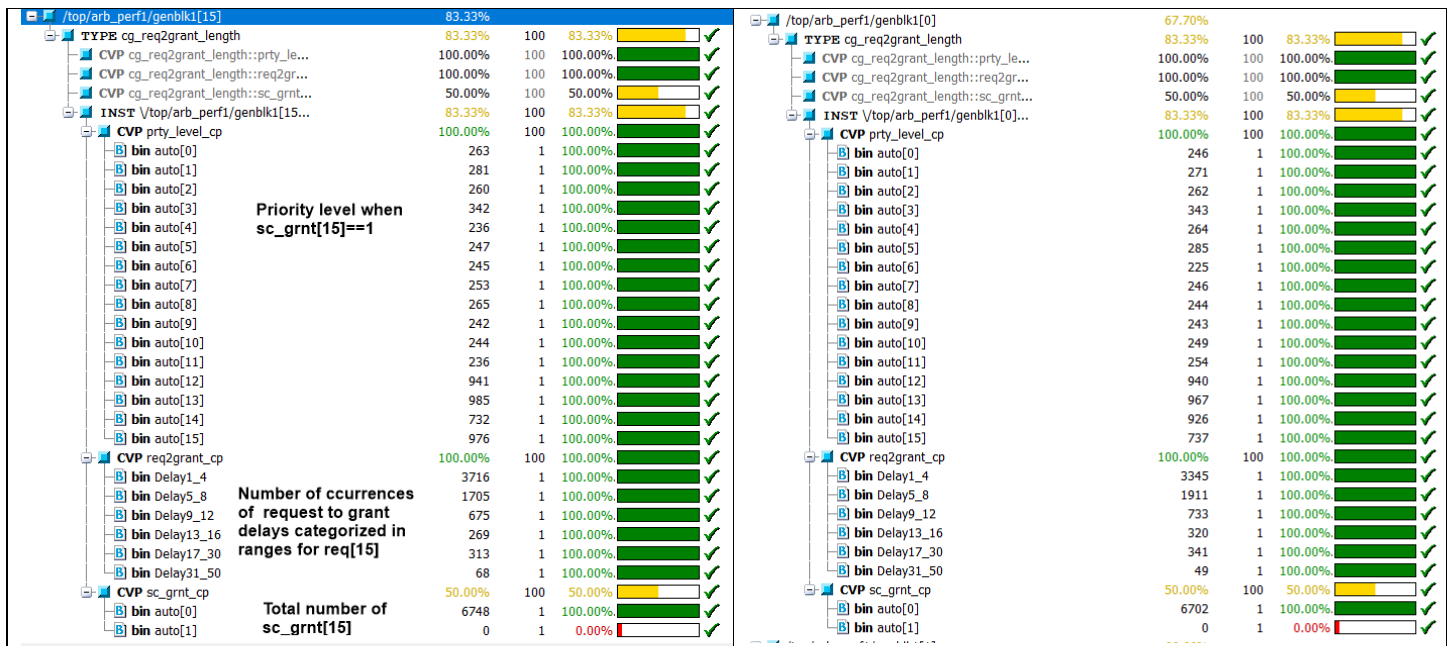
```
pgrp15_12=3; pgrp11_8=0;  pgrp7_4=0; pgrp3_0=0;
```

// priority dwells at level 15 for 4 grants, then switches to level 14 for four more grants
// priority dwells at level 13 for 4 grants, then switches to level 12 for four more grants
//priority dwells for 1 grant for the other levels.

```
 if (!randomize(v_space, servtime) with {
        v_space < 10; // between requests
        servtime inside {[1:7]};
      }) `uvm_error("MYERR", "This is a randomize error");
// Following discribution is evenly distributed with the exception that req[14:12]
// are never activated.
      if (!randomize(v_req) with {
        v_req[15] dist {1'b1 := 1, 1'b0 := 32};
        v_req[14] dist {1'b1 := 0, 1'b0 := 32};
        v_req[13] dist {1'b1 := 0, 1'b0 := 32};
        v_req[12] dist {1'b1 := 0, 1'b0 := 32};
        v_req[11] dist {1'b1 := 1, 1'b0 := 32};
        v_req[10] dist {1'b1 := 1, 1'b0 := 32};
        v_req[9]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[8]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[7]  dist {1'b1 := 2, 1'b0 := 32};
        v_req[6]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[5]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[1]  dist {1'b1 := 2, 1'b0 := 32};
        v_req[3]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[2]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[1]  dist {1'b1 := 2, 1'b0 := 32};
        v_req[0]  dist {1'b1 := 1, 1'b0 := 32};
```

The table below provides information for bit 15 and bit 0 for the following statistics: Using bit 15 as an example, we see the following results:

1) The number of times a grant was provided for `req[15]` when the round-robin priority was at a priority level.
   For example, we see that grant was given 976 times when the priority level was 15, but only 263 times when the priority level was 0. This is expected since more time slots were provided for group pgrp15_12.
2) The delay ranges from `req[15]` to `sc_grant[15]`: We can see that most of the delays for a grant were within 4 cycles, but some extended more than 31 cycles.
3) The total number of grants serviced for bit 15.

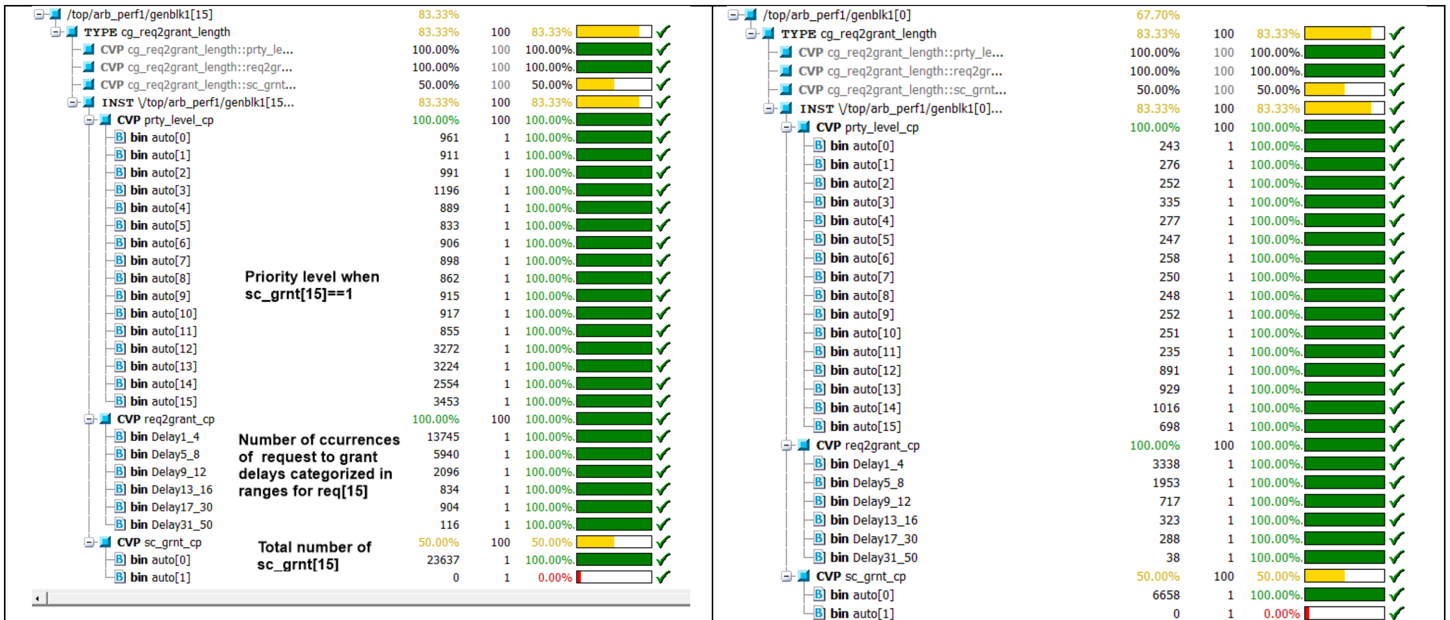


The following set of results uses a different profile where more requests are made from bit 15.

pgrp15_12=3; pgrp11_8=0; pgrp7_4=0; pgrp3_0=0;

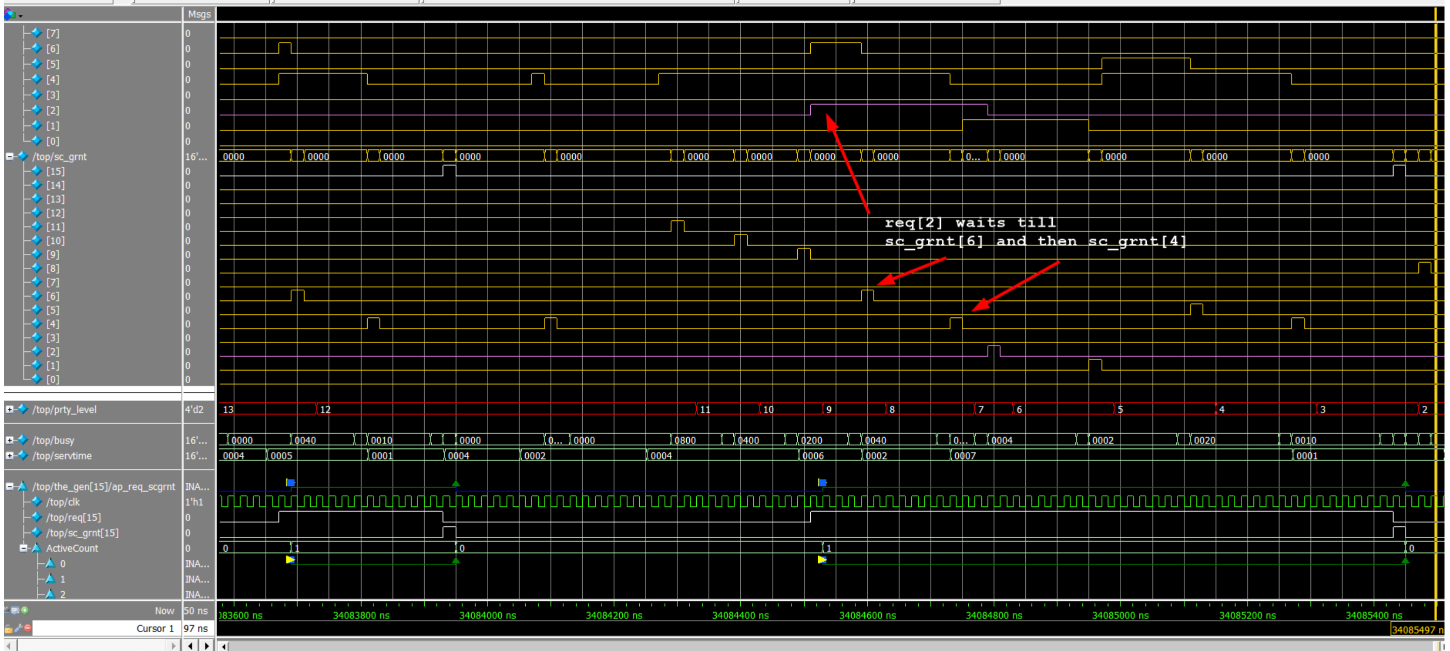
```
if (!randomize(v_space, servtime) with {
        v_space < 10; // between requests
        servtime inside {[1:7]};
      }) `uvm_error("MYERR", "This is a randomize error");

      if (!randomize(v_req) with {
        v_req[15] dist {1'b1 := 4, 1'b0 := 32}; // More occurrences of req[15]
        v_req[14] dist {1'b1 := 0, 1'b0 := 32};
        v_req[13] dist {1'b1 := 0, 1'b0 := 32};
        v_req[12] dist {1'b1 := 0, 1'b0 := 32};
        v_req[11] dist {1'b1 := 1, 1'b0 := 32};
        v_req[10] dist {1'b1 := 1, 1'b0 := 32};
        v_req[9]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[8]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[7]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[6]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[5]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[1]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[3]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[2]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[1]  dist {1'b1 := 1, 1'b0 := 32};
        v_req[0]  dist {1'b1 := 1, 1'b0 := 32};
```



Those results demonstrate that `req[15]` benefited from more time slots serviced on the highest priority port. Specifically, there were 13,745 request-to-grant delays that were less than 4 cycles, and 7,794 completed transactions that were greater than 4 cycles; this compute to a faster response in 64% of the time for this scenario. This compares to 50% for `req[0]` which had the lowest priority.

Below is a waveform than demonstrates th timing between requests and grants and priority levels.



1.6 CONCLUSIONS

SystemVerilog can perform statistical analysis of computer systems. Unlike other verification techniques like UVM that verify that the design meets the requirements, statistical analysis of the system can bring out more details about the system and can challenge the sanity of the requirements. For this arbiter design, the analysis demonstrated the weakness of changing the level of arbitration with a simple countdown rather than a countdown following a completed transaction, i.e., a request followed by a grant. The analysis also demonstrated the need to define the system load environment because that determines the system performance in terms of latencies, such as delays between a request and a grant. The greatest difficulty in doing statistical analysis is the definition of those statistical inputs, and it's the old GIGO issue.

---

[i] https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=10982&context=theses

Arbiter:
http://systemverilog.us/vf/arbiterv426a.sv
https://edaplayground.com/x/ZcAf