Goldilocks and System Performance Modeling

A SystemVerilog Adaptive Rate Control (ARC) Stimulus Generation Methodology

Rich Edelman
Mentor Graphics
Fremont, CA

Shashi Bhutada
Mentor Graphics
Los Angeles, CA

Abstract- This paper focuses on the process of verification of a System-on-a-Chip (SoC) consisting of multiple ARM AMBA® AXI™ bus fabrics with mix of RTL IP and verification IP master and/or slave blocks. Specific focus in this paper is adjusting stimulus generation in order to maintain consistent bandwidth loads. Adjusting loads is done two ways - by adjusting the arrival rate, and adjusting the transaction delays.

Keywords - SystemVerilog UVM, constrained random stimulus generation, constant rate stimulus

I. Background

The verification system in question is testing the fabric and slave response under load. The load needs to be kept within a certain range – neither too heavily loaded, nor too lightly loaded. The bus matrices and slaves should be kept busy, but not so busy as to become saturated. Attention must be paid to traffic generation and slave response pattern in order to maintain the load of the overall system or specific leg of the system with constant or varying bandwidth.

The idealized example SoC consists of three masters, two slaves, and two bus matrix fabrics. This is a simplified version of a typical SOC using AXI [1] interfaces, but sufficiently describes the problems faced while verifying a typical SoC.

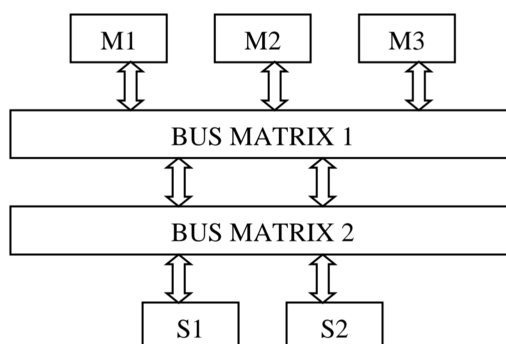

Figure 1: Ideal Bus Matrix Fabric

The main contribution from this paper is a simplified example demonstrating Adaptive Rate Control (ARC) algorithms and techniques using SystemVerilog [2] UVM [3] Sequences with constrained randomization timing

control of stimulus generation to maintain desired loads. Additionally, the simplified example is itself useful to build a system with a programmable number of masters.

If the SoC contained real RTL code, we could substitute the real RTL for a slave or master model in which we could control delays. Due to proprietary IP concerns we adapted our SoC to only contain slave or master models. The models provide channel based transactions roughly similar to AXI. These models are sufficient to demonstrate the underlying concepts of ARC.

The stimulus generation for this testbench requires that under load the traffic bandwidth at the slave remain constant or adapt to certain data rate - *it must not be too high nor too low* - the "background" stimulus or traffic must adapt to the actual load in the system posed by the current test.

II. EXPERIMENTAL SETUP – THE STRUCTURE

In order to avoid disclosing any proprietary information the examples are simplified from the diagram above. AMBA AXI-like protocol is used without a specific verification IP. No actual RTL is used. These adjustments to the experimental test harness do not change the basic algorithm or technique. They simply remove all proprietary information.

The experiment performed is to hold the bandwidth constant at a slave. For example, in the diagram below, an arbitrarily complex fabric is modelled with a programmable number of masters and a single slave. Using this system we will create N masters each supplying requests to the slave through a “fabric”. The slave bandwidth will be monitored. The arrival rate and transaction delay characteristics of the master will be adjusted to keep the desired slave bandwidth.

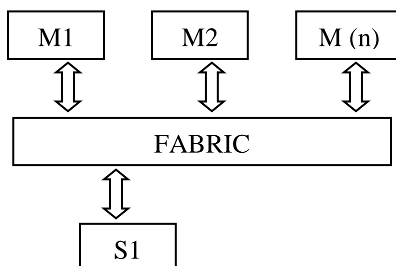

Figure 2: Experimental Bus Matrix Fabric

The fabric is modeled as a simple first-in-first-out system. The fabric simply matches up responses with requests and processes the FIFOs.

The stimulus is a simple WRITE to an address and a READ from the same address. The read-data is compared with the write-data. In a more general system, the stimulus would model real known traffic envelopes. For example, CPU memory access for algorithm behavior, video traffic, background WIFI data transfer or any other kind of traffic the SoC may contain.

III. AXI-LIKE PROTOCOL

The AMBA AXI Protocol is a complex protocol with many powerful capabilities. The complete power is not required for our experiments. We'll issue simple READs and WRITEs with a simplified transaction definition.

A Read occurs when the master issues a call to READ(). The READ is supplied an address and a number of bytes to read, along with the delays for this transfer.

The READ() task used in this model is below.

```
task READ(addr_t addr, inout array_of_bytes_t data, transaction_delay_t transaction_delays);
  ...
  RA(tag, addr, number_of_beats);
  RD(tag, beats);
endtask
```

A Write occurs when the master issues a call to WRITE(). The WRITE is supplied an address and an array of bytes to write, along with the delays for this transfer.

The WRITE() task used in this model is below.

```
task WRITE(addr_t addr, input array_of_bytes_t data, transaction_delay_t transaction_delays);
  ...
  WA(tag, addr);
  WD(tag, beats);
  B(tag);
endtask
```

The AXI-like transaction we use is defined below.

```
class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

  event really_done;

  transaction_delay_t transaction_delays;

  rw_t rw;
  bit [31:0] addr;
  array_of_bytes_t data;
endclass
```

It has a simple READ/WRITE bit, with an address and an array of bytes. The address field is assigned a value within a range. Each master is modeled with a specific address map on the slave in order to simplify the scoreboarding. The data field is filled in with "well-known" data, so that it can be tracked across the system. The transaction data - an array of bytes - will be converted to beats for transfer over the bus. A write is issued and then a read. The only randomized values are the delays. Each transaction carries its delays in the 'transaction_delays' field.

The transaction delays are randomized, keeping the size of the read and write beats the same. The call to randomize the transaction delays is below.

```
if (!t.transaction_delays.randomize() with {
    delay_WD_valid_to_WD_valid.size() == delay_RD_valid_to_RD_valid.size();
    delay_WD_valid_to_WD_valid.size() == delay_RD_valid_to_RD_ready.size();
    delay_WD_valid_to_WD_valid.size() == delay_WD_valid_to_WD_ready.size();
    })
  `uvm_fatal("sequenceA", "Delay Randomization Failed")
```

Using $display ("DELAY DEBUG: %p", t.transaction_delays); we can examine the generated delays. In this case, there will be 9 beats.

The generated delays for a single randomization call are listed below.

```
"WA_valid_to_WA_ready":1,
"WA_valid_to_WD_valid":2,
"WD_valid_to_WA_valid":1,
"WD_valid_to_WD_ready":'{0:2, 1: 1, 2:  1, 3:62, 4:117, 5:11, 6:5, 7: 1, 8:1, ...},
"WD_valid_to_WD_valid":'{0:1, 1: 1, 2:120, 3: 1, 4:  2, 5: 1, 6:8, 7:11, 8:5, ...},
"B_valid_to_B_ready"  :1,

"RA_ready_to_RD_valid":1,
"RA_valid_to_RA_ready":1,
"RD_valid_to_RD_ready":'{0:1, 1:56, 2:  1, 3:19, 4:  1, 5:14, 6:1, 7: 1, 8:1, ...},
"RD_valid_to_RD_valid":'{0:1, 1: 1, 2:  1, 3: 1, 4:  2, 5: 1, 6:1, 7: 1, 8:1, ...},
```

IV. THE AXI-LIKE SIGNALS – THE CHANNEL

The AXI-like bus contains 5 channels: a Read Address channel, a Read Data channel, a Write Address channel, a Write Data channel and a Write Response channel. These are modelled after the AMBA AXI channels, but do not adhere to the AMBA AXI protocol – these are experimental channels used to create a simplified example for discussing ARC.

The AXI-like bus is modeled below. It consists of five channels.

```
interface channel(input wire clk);
  logic  RA_ready;
  logic  RA_valid;
  tag_t  RA_tag;
  addr_t RA_addr;
  int    RA_beat_count;

  logic  RD_ready;
  logic  RD_valid;
  tag_t  RD_tag;
  int    RD_beat_count;
  data_t RD_data;

  logic  WA_ready;
  logic  WA_valid;
  tag_t  WA_tag;
  addr_t WA_addr;

  logic  WD_ready;
  logic  WD_valid;
  tag_t  WD_tag;
  int    WD_beat_count;
  data_t WD_data;

  logic  B_ready;
  logic  B_valid;
  tag_t  B_tag;
endinterface
```

For example, here is a signaling protocol in the slave model.

- when WA_valid occurs, wait a defined number of clocks, then set WA_ready.
- when WD_valid occurs, wait a defined number of clocks, then set WD_ready.

Both of the always blocks below use the 'delay_table' to retrieve the delay, either for a specific tag, or for a specific tag and a specific beat.

The always block in the slave which controls WA_ready.

```
always @(posedge bus.WA_valid) begin: WA_Channel_ready
  repeat(delay_table.get_delay("WA_valid_to_WA_ready", bus.WA_tag))
    @(posedge bus.clk);
  @(negedge bus.clk);
  bus.WA_ready = 1;
end
```

The always block in the slave which controls WD_ready. (The delay control is per tag per beat).

```
always @(posedge bus.WD_valid) begin: WD_Channel_ready
  repeat(delay_table.get_per_beat_delay(
    "WD_valid_to_WD_ready", bus.WD_tag, bus.WD_beat_count))
    @(posedge bus.clk);
  @(negedge bus.clk);
  bus.WD_ready = 1;
end
```

These are examples of the simple modeling for this experimental setup. This experiment allows for multiple outstanding transactions and out-of-order completion. It does not consider special addressing modes, and other true AXI specifications.

For example, in the waveform screenshot below, if we are tracking the WRITE transaction with ID 118 (highlighted in blue), we can see the delay from WA to WD, and then B. Meanwhile many other transactions have occurred.

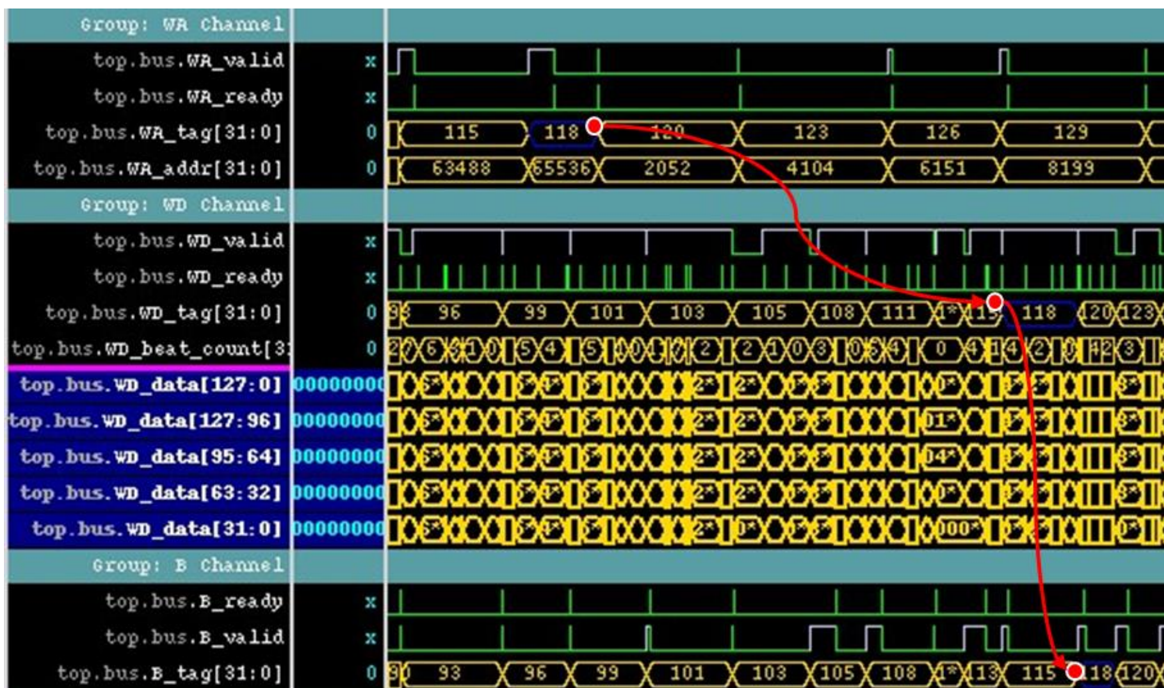

Figure 3: Waveform showing a Write Transaction with WA, WD and B channels

V. THE DELAY STRUCTURE

The entire SystemVerilog constraint modeling master and slave delay timing is about 100 lines long. The final sum which we wish to calculate – the total read delay and the total write delay is below. The partial code snippets below belong together as the controls for the generation of the transaction delays.

The read delay constraint.

```
constraint sum_read_delay {

  // Master
  delay_RD_valid_to_RD_ready.sum() +

  // Slave
  delay_RA_valid_to_RA_ready       +
  delay_RA_ready_to_RD_valid       +
  delay_RD_valid_to_RD_valid.sum()

  < read_cycles;
}
```

The write delay constraint.

```
constraint sum_write_delay {

  // Master
  delay_WD_valid_to_WD_valid.sum() +
  delay_WA_valid_to_WD_valid       +
  delay_WD_valid_to_WA_valid       +

  // Slave
  delay_WD_valid_to_WD_ready.sum() +
  delay_B_valid_to_B_ready         +
  delay_WA_valid_to_WA_ready

  < write_cycles;
}
```

Some values in the constraint system are tightly controlled – the number of beats is to be between 0 and 16. The value range constraint.

```
constraint size_range {
  delay_WD_valid_to_WD_valid.size() >  0;
  delay_WD_valid_to_WD_valid.size() < 16;

  delay_RD_valid_to_RD_ready.size() >  0;
  delay_RD_valid_to_RD_ready.size() < 16;

  delay_WD_valid_to_WD_ready.size() >  0;
  delay_WD_valid_to_WD_ready.size() < 16;

  delay_RD_valid_to_RD_valid.size() >  0;
  delay_RD_valid_to_RD_valid.size() < 16;

}
```

The total number of cycles is limited between 256 and 1000 clocks.

```
   read_cycles > 256;  read_cycles < 1000;
  write_cycles > 256; write_cycles < 1000;
```

Some values must be below a maximum setting and above zero. The value limit constaint.

```
constraint value_range {

  delay_WA_valid_to_WD_valid < max_clock_delay;
  delay_WD_valid_to_WA_valid < max_clock_delay;
  delay_B_valid_to_B_ready   < max_clock_delay;
  delay_WA_valid_to_WA_ready < max_clock_delay;
```

```
    delay_RA_valid_to_RA_ready < max_clock_delay;
    delay_RA_ready_to_RD_valid < max_clock_delay;

    foreach (delay_WD_valid_to_WD_valid[x])
      delay_WD_valid_to_WD_valid[x] < max_clock_delay;
    foreach (delay_RD_valid_to_RD_ready[x])
      delay_RD_valid_to_RD_ready[x] < max_clock_delay;
    foreach (delay_WD_valid_to_WD_ready[x])
      delay_WD_valid_to_WD_ready[x] < max_clock_delay;
    foreach (delay_RD_valid_to_RD_valid[x])
      delay_RD_valid_to_RD_valid[x] < max_clock_delay;
  }
```

VI. PROTOCOL DELAYS MODELED

Write Channel

The AXI-like bus issues writes by using the Write Address Channel, the Write Data Channel and the Write Response Channel. There are associated delays which can be configured to produce different kinds of traffic. Those delays are illustrated below.

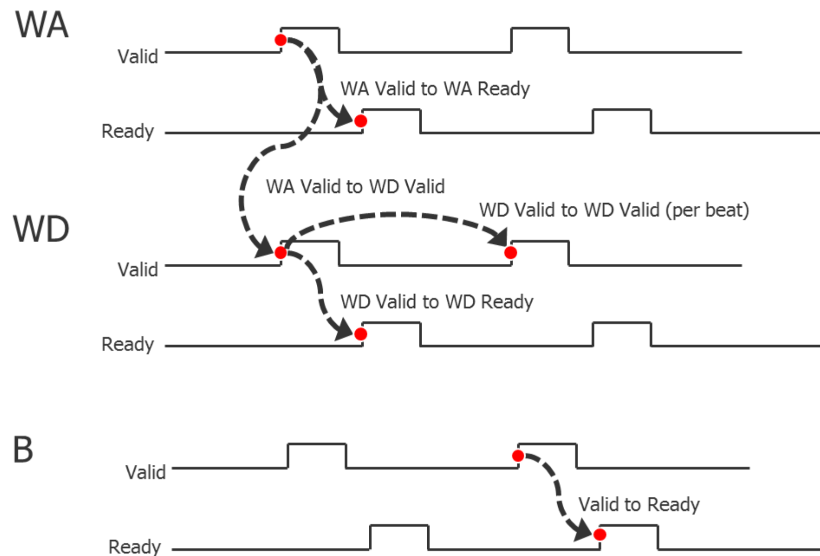

Figure 4: Write Delays

Slow Slave WA: The WA valid signal is driven by the master. The WA Ready signal is driven by the slave. The delay WA_valid_to_WA_ready models at what speed the **slave** signals ready, once a valid is received.

Slow Slave WD: The WD valid signal is driver by the master. The WD ready signal is driven by the slave. The delay WD_valid_to_WD_ready models at what speed the **slave** signals ready, once a valid is received.

Slow Master B: The B valid signal is driven by the slave. The B ready signal is driven by the master. The delay B_valid_to_B_ready models at what speed the **master** signals ready, once a valid is received.

How fast does the Master supply WD after WA: The delay WA_valid_to_WD_valid models at what speed the **master** supplies data follows address (WD follows WA). There is an additional delay to model data changing first, which we ignore for this paper.

How fast does the Master supply the write data beats: The delay WD_valid_to_WD_valid models at what speed the **master** supplies the data beats.

Read Channel

The AXI-like bus issues writes by using the Read Address Channel and the Read Data Channel There are associated delays which can be configured to produce different kinds of traffic. Those delays are illustrated below.

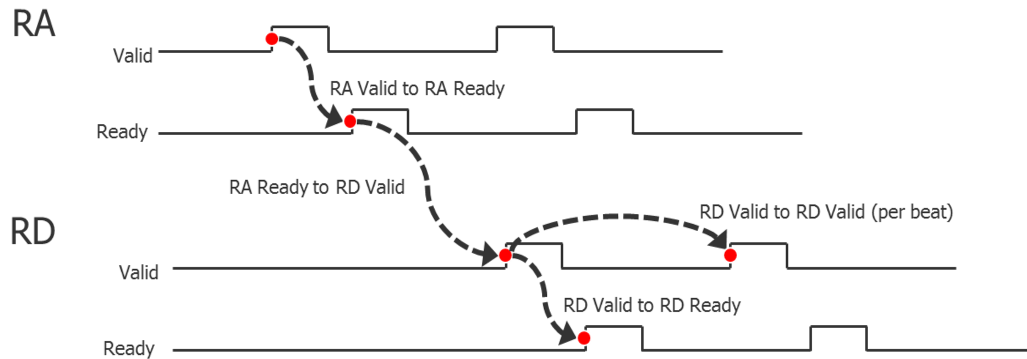

Figure 5: Read Delays

Slow Slave RA: The RA valid signal is driven by the master. The RA ready signal is driven by the slave. The delay RA_valid_to_RA_ready models at what speed the **slave** signals ready, once a valid is received.

Slow Master RD: The RD valid signal is driven by the slave. The RD ready signal is driven by the master. The delay RD_valid_to_RD_ready models at what speed the **master** signals ready, once a valid is received.

How fast does the Slave supply RD after RA: The delay RA_ready_to_RD_valid models at what speed the **slave** supplies data once the master has signaled ready.

How fast does the Slave supply the READ data beats: The delay RD_valid_to_RD_valid models at what speed the **slave** supplies the data beats.

VII. THE MASTER DELAYS

The master READ and WRITE performance can be controlled with the delays below.

For Master READ, the RD_valid_to_RD_ready models how fast the Master READ Data Channel is ready, once a RD valid occurs. By making this delay larger, a sluggish master data channel can be created.

For Master WRITE, there are a number of controls.

WA_valid_to_WD_valid models the delay between WRITE Address and WRITE Data transactions. In the highest speed system, this delay would be zero – sending both transactions at the same time. By making this delay larger, the master can allow more time for the slave to accept the data.

WD_valid_to_WD_valid models the delay between WRITE Data beats. The master supplies the beats for the transaction at this rate. The beat delay can be changed for each beat. This delay is the overriding delay for a WRITE transaction to complete – since it is possible to have many beats.

The B_valid_to_B_ready models how fast the Master WRITE Data Channel is ready, once a B valid occurs. By making this delay larger a sluggish master write channel is created.

VIII. The Slave Delays

The slave READ and WRITE performance can be controlled with the delays below.

For Slave READ, the RA_valid_to_RA_ready models how fast the Slave READ Address Channel is ready, once a RA valid occurs. By making this delay larger, a sluggish read address channel can be created.

RA_ready_to_RD_valid models the delay between READ Address and READ Data. By making this delay larger, a sluggish slave can be modeled.

RD_valid_to_RD_valid models the delay between READ Data beats. The slave supplies the beats for the transaction at this rate. The beat delay can be changed in each beat. This delay is the overriding delay for a READ transaction to complete – since it is possible to have many beats.

For Slave WRITE, The WA_valid_to_WA_ready models how fast the slave signals ready once a valid is seen.

WD_valid_to_WD_ready models the delay between each valid/ready pair for the write data. This is the overriding delay for a WRITE transaction to complete – since it is possible to have many beats.

IX. Stimulus Generation

The Arrival Rate

The easiest way to control the bandwidth is by changing the arrival rate of the generated transactions. By lowering the arrival rate, there are fewer transactions submitted, and the overall bandwidth requirements will decrease.

The transaction delay

The transaction delay for this experimental system has a large impact on bandwidth requirements. In this experiment once a transaction (transaction ID) is initiated, it cannot be interrupted, nor can other transactions be interleaved amongst the data beats. Because of this behavior, a "long-slow" transaction can occupy the bus while other "short-fast" transactions are waiting. This effectively causes the "short-fast" transactions to become "longer-slower" transactions.

Once this happens, the "long-slow" transaction rate can become the dominating factor for down-stream bandwidth. In order to compensate for the slowdown, the "long-slow" transactions need to be made into "shorter-faster" transactions. When the bandwidth desired drops too low, the transactions can be made shorter. This will limit the time they control the bus, and will allow other fast transactions to complete within their desired specification.

X. Conclusion

The experimental system was able to be used to test out ideas about controlling bandwidth, and providing constant rate bandwidth at a slave. A future experiment will expand the example to multiple slaves, and allow for interleaved data beats, to more closely match AXI behavior and provide better throughput.

XI. References

[1] ARM AMBA® AXI™ Protocol, ARM IHI 002D.

[2] SystemVerilog LRM, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[3] UVM 1.1d Reference implementation code, http://www.accellera.org/downloads/standards/uvm/uvm-1.1d.tar.gz