No RTL Yet? No Problem
UVM Testing a SystemVerilog Fabric Model

Rich Edelman
Mentor Graphics, Fremont, CA
rich_edelman@mentor.com
510-354-7436

***Abstract*-SystemVerilog is a powerful language which can be used to build models of RTL in order to facilitate early testbench testing. The early RTL model uses higher level abstractions like SystemVerilog threads, queues, dynamic arrays and associative arrays. Using high level abstractions allows a functional model to be created with little effort. A simple fabric model is created implementing AXI-like READY/VALID channels.**

I. INTRODUCTION

Building a UVM [2] testbench is a hard job, made harder when operational RTL is not yet available to test. SystemVerilog [1] is a powerful modeling language that can be used to build a high level model of hardware before RTL is available. This model is fast to write, and can be as functionally complete as needed. This paper will describe the creation and use of a fabric model to build and bring up a testbench. When the RTL is available it can be plugged into the testbench model with little change required.

The main contributions of this paper are: showing a fully functional model of a medium complexity communication fabric; writing the model using SystemVerilog; and building a reusable testbench that can support block testing as well as support system level tests. The testbench implementation is not discussed in this paper. Contact the author for more information.

The Fabric

The fabric is a simple two port switch with buffering. Each communication pathway contains 5 channels. Each channel is a Ready/Valid channel with similarities to an AMBA® AXI™ [4] channel. The fabric supports multiple outstanding transactions, pipelining and large burst data transfers. It also supports traffic priority (quality of service). This is the model. It is written using SystemVerilog. Associative arrays are used to manage out-of-order transactions. Queues and dynamic arrays are used for managing lists. Classes are used to hold the transferred information - a class each for RA (Read Address), WA (Write Address), RD (Read Data), WD (Write Data) and B (Write Response), respectively. Packed structs are used for managing the tag bit fields.

The Testbench

The testbench is built to test the features supported by the fabric. It is a basic block testbench, testing transfer, buffering limits and other edge conditions in the RTL. This testbench will also be reusable to the system tests. The testbench is a basic UVM testbench with transfer sequences and background traffic sequences available.

The DUT

The actual device under test will be two fabrics connected together, as in Figure 2

II. BACKGROUND

The RTL implements an AXI-like fabric. It is a simple fabric (simpler than AMBA AXI), but has support for quality of service, out-of-order completion, and other high end features. The verification team wishes to get an early start on verification of this fabric. But early, functional RTL is not available.

In order for the verification team to get an early start on the verification environment, having an early model of the DUT is desirable.

In this paper, a theoretical verification team has built a UVM testbench, and needs to make sure that the testbench can check and verify the hardware. The tests are quite simple – generating streams of READs and WRITEs. This theoretical verification team is testing a simple AXI-like fabric. (It is not an AXI system). The fabric is a simple two port switch as seen in Figure 1. The system under test connects two of these fabrics.

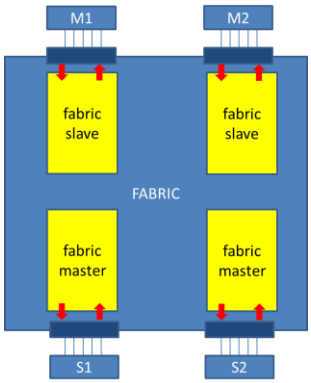

Figure 1 - Simple two port switch

The DUT and fabric models will be used instead of the real RTL and have enough functionality to provide a useful test environment for early testbench bring up. The model is not a completely functioning version of the RTL. Modeling the complete functionality for the RTL would be too large a task for early verification, and offer limited extra verification points.

The fabric has two ports on top and two ports on the bottom. The top ports can be routed to either bottom port. The DUT is a combination of two of these fabrics, as in Figure 2. A request is issued at either Master 1 (M1) or Master 2 (M2). It flows out to either Slave 1 (S1) or Slave 2 (S2) depending on many factors including QOS and Address Maps ranges defined.

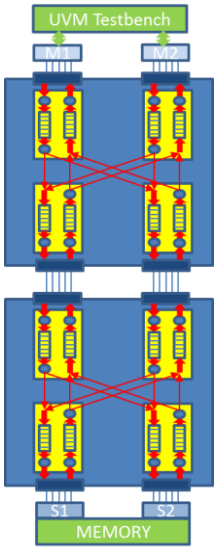

Figure 2 - DUT, Slave Memory and Testbench

A connection is virtual – the data transfer occurs in hops from master to slave. Additionally, the connections are managed as independent channels. The channels are WRITE ADDRESS (WA), WRITE DATA (WD), WRITE

RESPONSE (B), READ ADDRESS (RA) and READ DATA (RD). Each channel operates independently, with a few basic rules. For example, WRITE DATA must start after the corresponding WRITE ADDRESS. READ DATA must start after the corresponding READ ADDRESS. Data transfers from different transactions can be interleaved. A QUALITY OF SERVICE (qos) property exists for each transfer that can be used to prioritize traffic. There are many other features of the fabric, but these basic features are sufficient for the testbench to be built and rudimentary testing to begin.

III. THE CHANNEL

The channel is the most basic bus. It is a collection of signals, organized as the RA, RD, WA, WD and B sub-channels. It is implemented as a SystemVerilog interface.

The Bus Pins

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

  // Read Data
  logic  RD_ready;
  logic  RD_valid;
  tag_t  RD_tag;
  int    RD_beat_count;
  data_t RD_data;
  int    RD_qos;

  // Write Address
  logic  WA_ready;
  logic  WA_valid;
  tag_t  WA_tag;
  addr_t WA_addr;
  int    WA_qos;

  // Write Data
  logic  WD_ready;
  logic  WD_valid;
  tag_t  WD_tag;
  int    WD_beat_count;
  data_t WD_data;
  int    WD_qos;

  // Write Response
  logic  B_ready;
  logic  B_valid;
  tag_t  B_tag;
  int    B_qos;
  ...
endinterface
```

Figure 3 - The Channel (The Bus)

The channel represents the pins of the bus.

The Fabric "Payload"

In addition to the pins, each transfer in the fabric is represented by a simple class, each containing the same transfer information as might appear on the bus. Using these classes to represent the transfers on the fabric makes the fabric quite simple, and powerful. (See Appendix XVI for the definition of the types).

```
  class read_address_c;
    tag_t  RA_tag;
    addr_t RA_addr;
    int    RA_beat_count;
    int    RA_qos;
  endclass

  class read_data_c;
    tag_t  RD_tag;
    int    RD_beat_count;
    data_t RD_data;
    int    RD_qos;
  endclass

  class write_address_c;
    tag_t  WA_tag;
    addr_t WA_addr;
    int    WA_qos;
  endclass

  class write_data_c;
    tag_t  WD_tag;
    int    WD_beat_count;
    data_t WD_data;
    int    WD_qos;
  endclass

  class write_response_c;
    tag_t  B_tag;
    int    B_qos;
  endclass
```

Figure 4 - The Fabric Communication Packets

IV. THE TESTBENCH

The testbench is a "regular" UVM testbench, with an environment, test, agent, driver, sequencers, sequences and transactions (sequence items). The basic sequence issues a write of semi-random data to an address, then issues a read of the same address, and compares the data read with the data written. In this way, it is self-checking. Other sequences, corresponding to traffic types (such as video streaming, audio streaming or push notifications) are beyond the scope of this paper, but quite easy to create as a collection of memory accesses with the characteristic bandwidth, latency and length.

The sequence

The sequences issue reads and writes. The built-in sequence issues a write then a read from the same address. It then compares the read and written data. Each sequence is assigned a memory range in which it reads and writes.

The transaction

The transaction is either a READ or a WRITE. It has an address and a data payload. Additionally, there is a Quality of Service request field (qos). The tag field is not used by the testbench, but is set by the synthetic fabric we are building.

```
class transaction extends uvm_sequence_item;
  ...
  tag_t            tag; // Set in the initiator (master_if.sv)
```

```
  rw_t              rw;
  bit [31:0]        addr;
  array_of_bytes_t  data;
  rand int          qos;
  ...
endclass
```

Figure 5 - transaction.svh

The Test

The test is a simple test, but is flexible to provide many parallel threads. It uses two arrays to hold the interesting interfaces – the master (initiators) and the monitor interface. These are set from above using the uvm_config_db mechanism.

The test has two associative arrays that hold the agent and sequence handles. The number of these created is controlled by the command line option +threads=N. The number of agents is the number of threads. The number of sequences is four times the number of agents. During the run_phase, the number of transactions that each sequence should create is set from the command line using the +transactions=M option. Each of the created sequences is started, and simulation ends when each sequence has completed.

```
class test1 extends uvm_test;
  `uvm_component_utils(test1)

  virtual master_interface          vif [1:2];
  virtual monitor_interface monitor_vif [1:2];

  agent     agent_h [int];
  sequenceA   seq_h [int];

  // The controls.
  int parallel_threads = 16;
  int number_of_transactions = 100;
...
    master_count = 0;
    for (int i = 0; i < parallel_threads; i++) begin
      for (int j = 1; j <= 2; j++) begin
        agent_h[master_count] = agent::type_id::create(
          $sformatf("agent-%0d-%0d", j, master_count), this);
        agent_h[master_count].vif = vif[j];
        master_count++;
      end
    end
  endfunction

`define N 4

  task run_phase(uvm_phase phase);
    int sequence_count;
    phase.raise_objection(this);

    sequence_count = 0;
    foreach (agent_h[i]) begin
      for (int j = 0; j < `N; j++) begin
        // N sequences on each sequencer
        seq_h[sequence_count] = sequenceA::type_id::create($sformatf("seq%0d",
          sequence_count));
        seq_h[sequence_count].sequence_id = sequence_count;
        seq_h[sequence_count].base_address = (sequence_count+1) * 2048;
        seq_h[sequence_count].number_of_transactions = number_of_transactions;
        sequence_count++;
```

```
      end
    end

    foreach (seq_h[i])
      fork
        automatic int j = i;
        #(j*1000) seq_h[j].start(agent_h[j/`N].sqr);
      join_none
    wait fork;

    phase.drop_objection(this);
  endtask
...
endclass
```

Figure 6 - test.svh

V. THE FABRIC BASICS

The fabric connects to the pins of the AXI-like interfaces, but instead of pins, the DUT will use SystemVerilog interfaces. Those interfaces were defined above as 'interface channel' buses.

There are two input channels, i0 and i1, and two output channels, o0 and o1. There are 4 instances to manage the interfaces and queues, two each of slaves and masters (fabric_slave1, fabric_slave2, fabric_master1 and fabric_master2).

```
import uvm_pkg::*;
import types_pkg::*;

module fabric #(parameter type FABRIC_TAG_T) (input wire clk,
  channel i0, channel i1,
  channel o0, channel o1);

  fabric_slave_interface    fabric_slave1(i0);
  fabric_slave_interface    fabric_slave2(i1);

  fabric_master_interface   fabric_master1(o0);
  fabric_master_interface   fabric_master2(o1);
```

Figure 7 - fabric.sv

When the fabric is instantiated, it is parameterized with a type (FABRIC_TAG_T); the type of the tag which is appropriate for this instance. For example, in the DUT there are two fabrics instantiated, one with 'fabric_tag_t' and one with 'fabric2_tag_t' as in Figure 8

```
  fabric #(fabric_tag_t)  fA(clk, i0, i1, m0, m1);
  fabric #(fabric2_tag_t) fB(clk, m0, m1, o0, o1);
```

Figure 8 - Using FABRIC_TAG_T to parameterize the fabric

The fabric contains queues and arrays to manage the transfers. It also has one big job, deciding which transfer goes out which output port (The big blue oval below). The various algorithms for quality of service and priority are beyond the scope of this paper, but would be quite easy to explore using this framework.

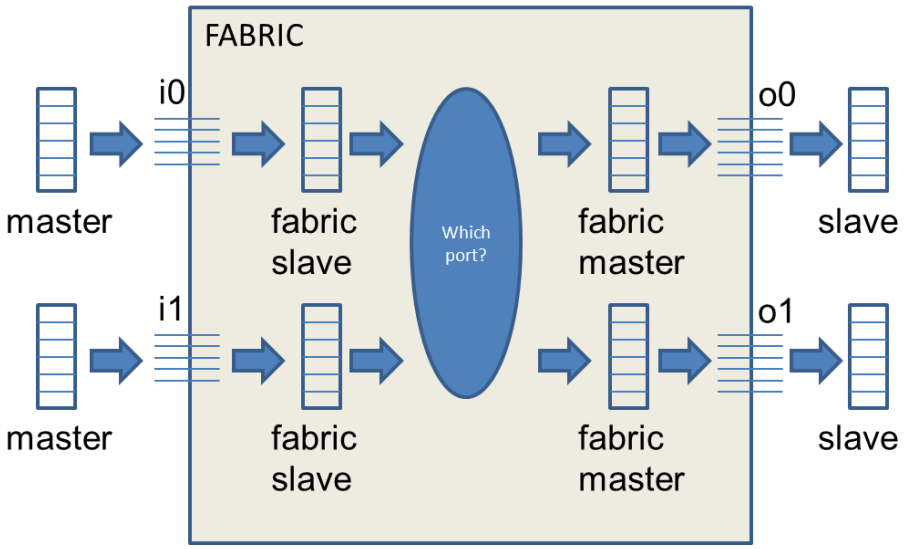

Figure 9 – Request flow: queue to bus to queue to queue to bus to queue

Figure 9 is a block diagram of the major structures in the fabric. The left-side connections are i0 and i1. The right-side connections are o0 and o1. Inside each fabric, at the inputs and outputs are two slaves and two masters respectively. For example the fabric i0 port is connected outside to a master interface. On the inside, fabric i0 port is connected to a fabric slave.

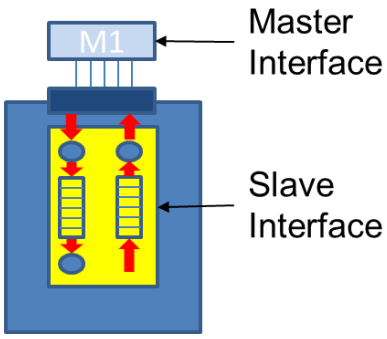

Figure 10 - Master and Slave

Tracing the Read Address (RA) and Read Data (RD) channels can help explain the connectivity. (See Figure 1 and Figure 9). The master issues a Read Address (RA). That RA is sent across the i0 channel by wiggling the pins on the i0 channel. The fabric slave connected to i0 recognizes the RA and creates a read_address_c packet. That read_address_c packet contains all the information from the RA transfer. It is placed into a "work queue" for one of the fabric masters. A thread in the fabric master detects that something has arrived in its work queue, and processes it. The read_address_c packet from the work queue is turned into pin wiggles and sent out the connected interface (either o0 or o1). Outside of the fabric another slave will receive this transfer.

In Figure 11, a Read Address (RA) request on the top of the diagram (light blue arrows) traverses across the structures of the fabric, resulting in a Read Address (RA) request executing on the slave memory. The Read Data (RD) response (deep green arrows) traverses the reverse path from the slave memory back to the originating master requester.

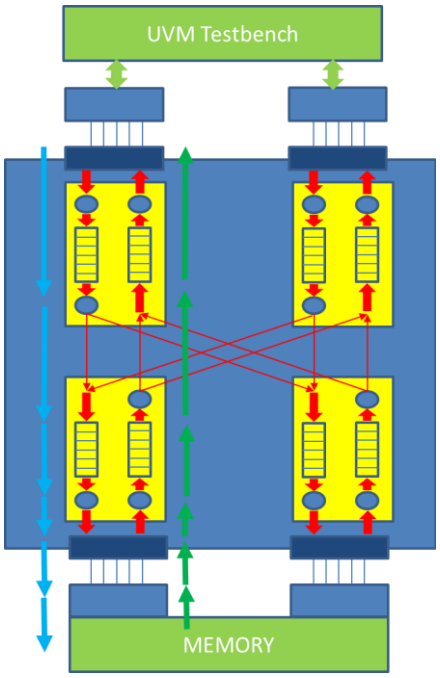

Figure 11 - Tracing RA request and RD response

VI. FABRIC SLAVE

The fabric slave implementation is quite simple. The slave waits for VALID and READY to both be high on a positive clock edge. When his happens, a transfer occurs. The transfer in this case copies the bus values into a class container that represents the transfer (i.e. read_address_c). The read_address_c packet is pushed into the ra_q that is managed in the slave. Some other thread will decide what to do with the new member of the queue.

```
import types_pkg::*;
import delay_pkg::*;

interface fabric_slave_interface(channel bus);

  read_address_c  ra_q[$];
  read_data_c     rd_q[$];

  write_address_c wa_q[$];
  write_data_c    wd_q[$];
  write_response_c b_q[$];

  always @(posedge bus.clk) begin: RA_Channel
    if ((bus.RA_ready == 1) && (bus.RA_valid == 1)) begin
      read_address_c ra;
      ra = new();
      ra.RA_tag = bus.RA_tag;
      ra.RA_qos = bus.RA_qos;
      ra.RA_addr = bus.RA_addr;;
      ra.RA_beat_count = bus.RA_beat_count;
      ra_q.push_front(ra);

      @(negedge bus.clk);
      bus.RA_ready = 0;
    end
  end
```

Figure 12 - Fabric Slave Interface

VII. FABRIC RA SLAVE SERVICE

In the fabric, a thread (the RA_Channel1 thread) is waiting for something in the ra_q from the fabric_slave1. When something arrives, it is removed from the queue and a new tag is created. This new tag is the key to tracing the return values.

```
always begin: RA_Channel1
  wait (fabric_slave1.ra_q.size() != 0);
  ...
  while (fabric_slave1.ra_q.size() > 0) begin
    read_address_c ra;
    ra = fabric_slave1.ra_q.pop_back();

    setup_fabric_tag_R(0, ra, ra.RA_tag, fabric_tag);

    if ( fabric_tag.slave_o_port == 0 )
      fabric_master1.ra_q.push_front(ra);
    else
      fabric_master2.ra_q.push_front(ra);
  end
end
```

Figure 13 - Fabric Slave - RA Channel

setup_fabric_tag_R – The new tag.

The original transfer contained a 'tag' or 'id'. That tag was created by the master interface, and is a simple counter to be able to match up and organize parallel transfers. Without the tag, the master would not know which outstanding READ address went with which READ data. This function will determine which output slave port the RA request goes to.

The tag_matcher_R associative array is used to keep track of in-flight transfers and to do error checking.

```
function automatic void setup_fabric_tag_R(input master_port,
                      read_address_c ra,
                      ref tag_t              tag,
                      ref FABRIC_TAG_T fabric_tag);
  fabric_tag.tag = tag;
  fabric_tag.master_i_port = master_port;
  fabric_tag.slave_o_port = fabric_tag.tag & 1'b1;
  tag = fabric_tag;

  if (tag_matcher_R.exists(tag)) begin
    $display("tag_matcher_R already has tag=%p", tag);
    $finish(2);
  end
  tag_matcher_R[tag] = ra;
  if (verbose_fabric)
    $display("FABRIC %t POP/PUSH %m RA tag=%0d %p", $time, tag, ra);
endfunction
```

Figure 14 - Managing fabric tags

VIII. FABRIC TAGS

Tags are used within this AXI-like system to connect parallel requests to each other. Tags serve as ids for transactions. As a transaction crosses this fabric, it chooses a path to take. We must ensure that the response ends up back in the right place, so the tag is used to keep track of where the transaction has been, and where it is going.
In the fabric, the master could either be 0 or 1, and the slave can be 0 or 1. Two bits each are reserved in the fabric tag.

The fabric tag consists of the original tag along with the slave and master information. As the transaction traverses the second fabric (this is a two fabric DUT), the tag is extended again, this time the tag is 16 bits – the previous "extended tag" simply looks like a "regular tag" to the second fabric.

```
typedef struct packed {
  bit [ 1:0] slave_o_port;
  bit [ 1:0] master_i_port;
  bit [11:0] tag;
} fabric_tag_t;

typedef struct packed {
  bit [ 1:0] slave_o_port;
  bit [ 1:0] master_i_port;
  bit [15:0] tag;
} fabric2_tag_t;
```

Figure 15 - Fabric Tag Definition

In the first level fabric, the slave and master occupy two bits. The tag is 12 bits long, for a total fabric tag length of 16 bits. In the second level fabric, the slave and master occupy two bits, but this time the tag is 16 bits. The tag in the second level fabric is the first level fabric tag. (All 16 bits).

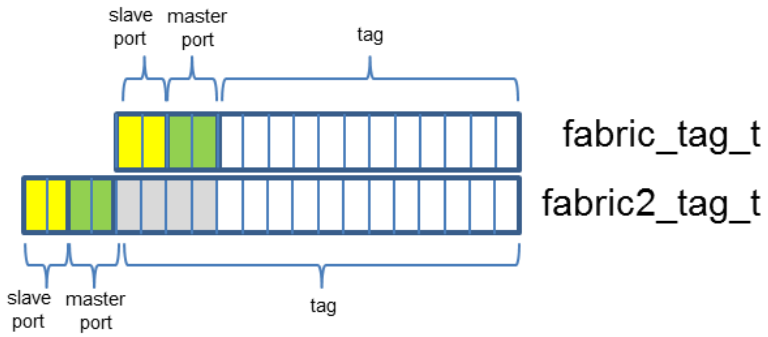

Figure 16 - Fabric Tag Bit Vector

For example, when a tag is traversing from the first level fabric to the second level, the fabric level 2 tag is assigned as:

```
fabric2_tag_t fabric2_tag;
fabric_tag_t  fabric_tag;

fabric2_tag.tag = fabric_tag;
```

IX. FABRIC MASTER INTERFACE

The fabric master RA_Channel thread wakes up when it finds something in the ra_q queue. That new arrival is popped off the queue, and turned into a pin wiggle.

```
import types_pkg::*;
import delay_pkg::*;
import tb_pkg::*;
import util_pkg::*;

interface fabric_master_interface(channel bus);

  read_address_c  ra_q[$];
```

```
  read_data_c     rd_q[$];

  write_address_c wa_q[$];
  write_data_c    wd_q[$];
  write_response_c b_q[$];

…

  always begin: RA_Channel
    wait (ra_q.size() != 0);
    while (ra_q.size() > 0) begin
      read_address_c ra;

      ra = ra_q.pop_back();

      bus.RA_tag = ra.RA_tag;
      bus.RA_qos = ra.RA_qos;
      bus.RA_addr = ra.RA_addr;
      bus.RA_beat_count = ra.RA_beat_count;

      bus.RA_valid = 1;
      while(1) begin
        @(posedge bus.clk);
        if ((bus.RA_ready == 1) && (bus.RA_valid == 1))
          break;
      end
      @(negedge bus.clk);
      bus.RA_valid = 0;
    end
  end
```

Figure 17 - Fabric Master - RA Channel Thread

X. THE SLAVE

The slave is quite simple, sharing a memory handle with the other slave. There is one physical memory (mem_interface), but two slave interfaces using it. The RA Channel recognizes a read request and pushes the tag onto the rd_work queue. The rd_work_queue is serviced by the RD_Channel thread. The RD_Channel thread wakes up and performs the requested number of reads, creating a RD response for each one.

```
import types_pkg::*;
import delay_pkg::*;

interface slave_interface(channel bus);

  virtual mem_interface mem;

  addr_t  ra[tag_t];
  int     ra_beat_count[tag_t];

  beats_t rd[tag_t];
  addr_t  wa[tag_t];
  beats_t wd[tag_t];
  bit     wd_complete[tag_t];
  tag_t    b[tag_t];

  tag_t rd_work_queue[$];

  always @(posedge bus.clk) begin: RA_Channel
    if ((bus.RA_ready == 1) && (bus.RA_valid == 1)) begin
      ra[bus.RA_tag] = bus.RA_addr;
```

```
      ra_beat_count[bus.RA_tag] = bus.RA_beat_count;
      rd_work_queue.push_front(bus.RA_tag);

      @(negedge bus.clk);
      bus.RA_ready = 0;
    end
  end

  always begin: RD_Channel
    int unsigned addr;
    beats_t beats;
    tag_t tag;

    wait (rd_work_queue.size() != 0);
    while (rd_work_queue.size() > 0) begin
      tag = rd_work_queue.pop_back();
      for (int i = 0; i < ra_beat_count[tag]; i++) begin
        addr = (i + ra[tag]) & 24'hffffff;
        beats[i] = mem.read(addr);
      end
      rd[tag] = beats;
      beats.delete();
      RD_internal(tag);
    end
  end
```

Figure 18 - Slave Interface - RA Channel and RD Channel

XI. THE TEST TOP

The test top instantiates 6 channels. Two channels for the input and two channels for the output and two channels for the intermediate connections between the fabrics. (See Figure 2). It instantiates two master interfaces, each connected to the input channels. It instantiates the two fabrics, each connected to the proper channels. Finally, it instantiates the two slaves and the two monitors and the actual memory.

```
module top;
  channel i0(clk);
  channel i1(clk);
  channel m0(clk);
  channel m1(clk);
  channel o0(clk);
  channel o1(clk);

  master_interface initiator0(i0);
  master_interface initiator1(i1);

  fabric #(fabric_tag_t)  fA(clk, i0, i1, m0, m1);
  fabric #(fabric2_tag_t) fB(clk, m0, m1, o0, o1);

  slave_interface  target0(o0);
  slave_interface  target1(o1);

  monitor_interface moni0(i0);
  monitor_interface moni1(i1);

  mem_interface mem();

  initial begin
    uvm_config_db#(virtual master_interface )::set( null, "", "m0",       initiator0);
    uvm_config_db#(virtual master_interface )::set( null, "", "m1",       initiator1);
    uvm_config_db#(virtual monitor_interface)::set( null, "", "monitor0", moni0     );
    uvm_config_db#(virtual monitor_interface)::set( null, "", "monitor1", moni1     );
```

```
    target0.mem = mem; // Each target gets a handle to the memory
    target1.mem = mem;

    run_test("test1");
  end
endmodule
```

Figure 19 - t.sv - The Top

XII. THE SIMPLE MEMORY

The simple memory (mem_interface) has two functions: read and write. They provide an easy way to encapsulate the memory access. Each read and write is echoed to standard out, and if a location is read which has never been written, an error is generated and simulation stops immediately. The memory uses an associative array – it is a sparse array implementation.

```
import types_pkg::*;

interface mem_interface();
  data_t mem[bit[31:0]]; // Associative Array

  function data_t read(int unsigned addr);
    data_t beat;
    if (!mem.exists(addr)) begin
      $display("@%t: %m READ  mem[%d] NON-EXISTENT Address", $time, addr);
      $finish(2);
    end
    beat = mem[addr];
    $display("@%t: %m READ  mem[%d] => %x", $time, addr, beat);
    return beat;
  endfunction

  function void write(int unsigned addr, data_t beat);
    $display("@%t: %m WRITE mem[%d] <= %x", $time, addr, beat);
    mem[addr] = beat;
  endfunction
endinterface
```

Figure 20 - mem_if.sv

XIII. CONCLUSION

This paper has described the implementation of a simple AXI-like skeleton fabric that was used to test the early development of a UVM testbench. It used high level SystemVerilog features such as threads, dynamic arrays, associative arrays, queues and simple classes to manage the complexity of implementing a real fabric.

The current system is flexible and could be used as a test vehicle for new quality of service algorithms, or other address map schemes to control transfer flow. The code implementing the fabric is less than 1000 lines of code, and was written during a 5 day period, with another 5 days to debug by one person, as a part-time exercise. The model is easy to read and easy to extend with new functionality or capabilities. It is not a model that is suited for current synthesis tools, but rather is a model used to get complex functionality implemented early in the design phase; enabling early testing and verification.

In this example, the DUT modeled was a fabric, but the power of SystemVerilog would allow any model to be written. SystemVerilog is a powerful, general purpose programming language.

XIV. REFERENCES

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

[2] SystemVerilog UVM 1.1d, http://accellera.org/images/downloads/standards/uvm/uvm-1.1d.tar.gz
[3] Sparse Arrays, https://en.wikipedia.org/wiki/Sparse_array
[4] ARM AMBA AXI, "AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite", https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR500-DA-10008-r2p1-00rel0/AR500-DA-10008-r2p1-00rel0.tgz

XV. APPENDIX: TRANSFERS ON THE TWO INITIATORS

This screenshot is of the two masters (initiators). There are four streams shown. Each stream represents an independent communication channel. For example the first stream is the initiator 1 write channel. Then the initiator 1 read channel, initiator 2 write channel and initiator 2 read channel. Even with this limited set of channels and a limited simulation run time, it is easy to see that there are large quantities of data to check and analyze, even for this simple circuit.

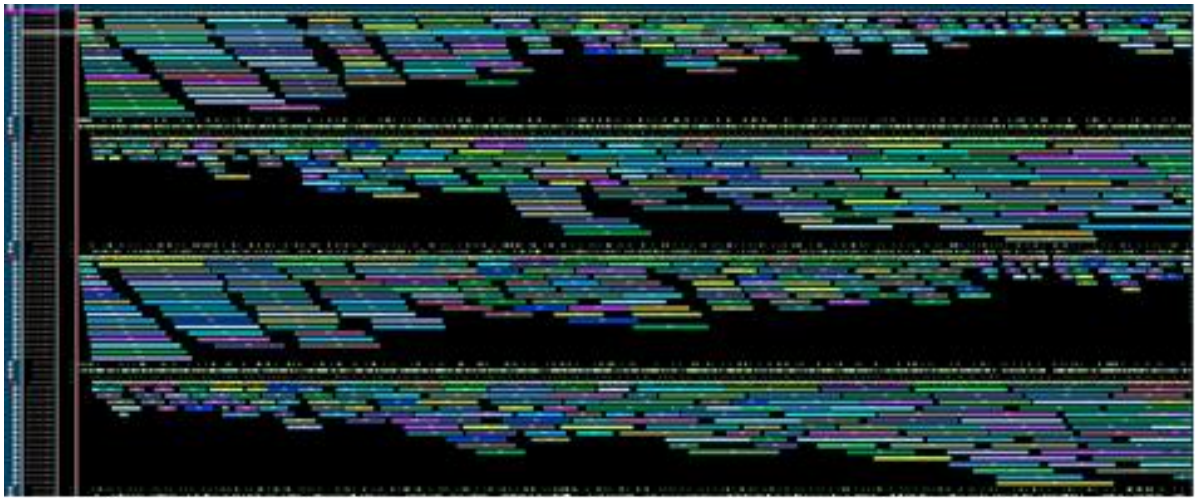

Figure 21 - Initiator 1 and 2 Write and Read Channels

XVI. APPENDIX: TYPES.SVH

Miscellaneous types used throughout the testbench and DUT.

```
typedef bit [ 31:0]  tag_t;     // Maximum size
typedef bit [ 31:0] addr_t;
typedef bit [127:0] data_t;
typedef bit [  7:0] byte_t;
typedef int        delay_t;

typedef enum bit [  1:0] {WRITE, READ, NOTUSED} rw_t;

typedef  byte_t array_of_bytes_t[   ]; // Dynamic Array
typedef  data_t          beats_t[int]; // Associative Array
typedef delay_t    beats_delay_t[int]; // Associative Array
typedef delay_t   rbeats_delay_t[   ]; // Dynamic Array
```