Matchlib SOC Verification and Debugging Tutorial

Stuart Swan

Platform Architect

Mentor, A Siemens Business

12 Jan 2022

Introduction

Some of the most challenging verification and debugging work occurs as blocks in SOCs are integrated together: it is only then that various interactions between different blocks may cause bugs to be exhibited. Traditionally such bugs are often caught in HW emulation when entire SOC subsystems are integrated and tested. This tutorial shows how many such integration bugs can be caught and analyzed in SystemC Matchlib models, before HLS is even run. Debugging these types of bugs in the SystemC model still can be challenging, but it is significantly easier than traditional HW emulation based approaches or RTL simulation approaches.

The intent of this tutorial is to demonstrate the various techniques and strategies that are useful for SOC debug in SystemC models. This tutorial uses capabilities that are available in all EDA HDL simulators and debuggers, and is not specific to a particular EDA toolset.

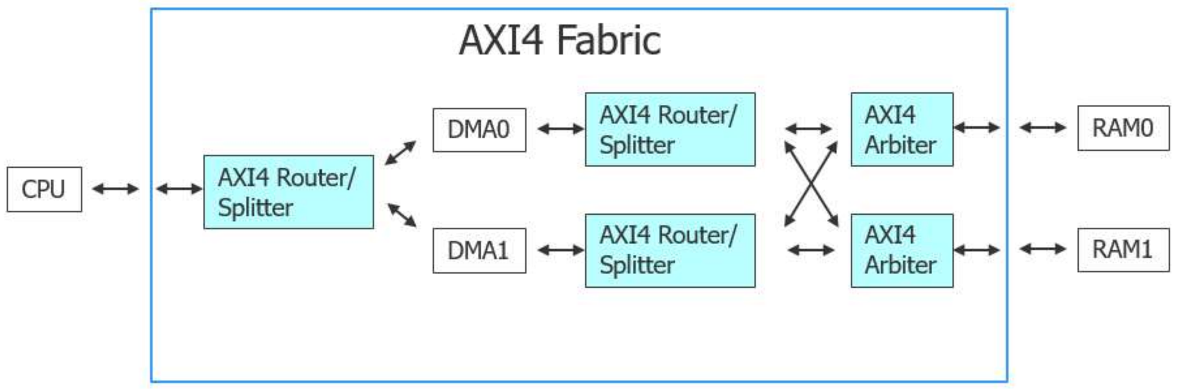
Description of Design

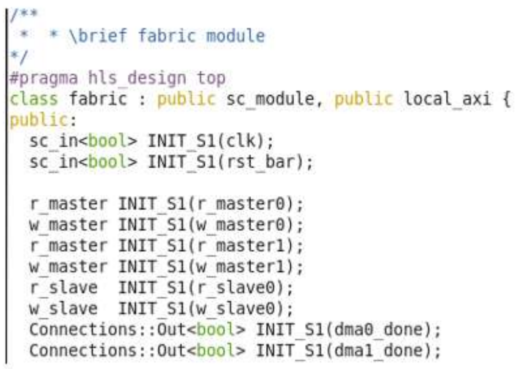
For this tutorial we use a simple AXI4 SOC fabric example to demonstrate the debug process. It is helpful if you have a basic understanding of bus protocols such as AXI4 for this tutorial. There is an introduction to this example here:

https://www.mentor.com/hls-lp/events/early-axi4-soc-performance-verification-using-nvidia-matchlib-and-catapult-systemc-hls

The complete source code and build scripts for this example are included in the Mentor Catapult 10.5 release (and later versions) by looking at Help->Examples->Matchlib. Export the examples and change directory to examples/60_rand_stall.

The DUT ("device under test") is the large blue block in the diagram below. The fabric has two DMA instances which can read and write to either of two RAM instances. The CPU can program the two DMA instances to perform various types of transfers. The AXI4 bus data is 64 bits wide and the RAMs have words that are 64 bits wide.





```
/**
 *  * \brief fabric module
*/
#pragma hls_design top
class fabric : public sc_module, public local_axi {
public:
  sc_in<bool> INIT_S1(clk);
  sc_in<bool> INIT_S1(rst_bar);

  r_master INIT_S1(r_master0);
  w_master INIT_S1(w_master0);
  r_master INIT_S1(r_master1);
  w_master INIT_S1(w_master1);
  r_slave  INIT_S1(r_slave0);
  w_slave  INIT_S1(w_slave0);
  Connections::Out<bool> INIT_S1(dma0_done);
  Connections::Out<bool> INIT_S1(dma1_done);
```

© Mentor Graphics Corp.   Company Con

Description of Test

For this tutorial we will have the DUT perform a single specific test. As per AXI convention, all addresses are byte addresses. The test is as follows:

At startup every 64 bit word in the RAMs is initialized with its byte address within that specific RAM. For example, the second word in each RAM will be initialized with the value eight.

The CPU will program DMA0 to perform a copy operation that reads from RAM0 address 0x0000 and writes to RAM0 address 0x2000. The copy operation is comprised of 16 AXI4 beats, where the burst length has been limited to 2 beats per burst.

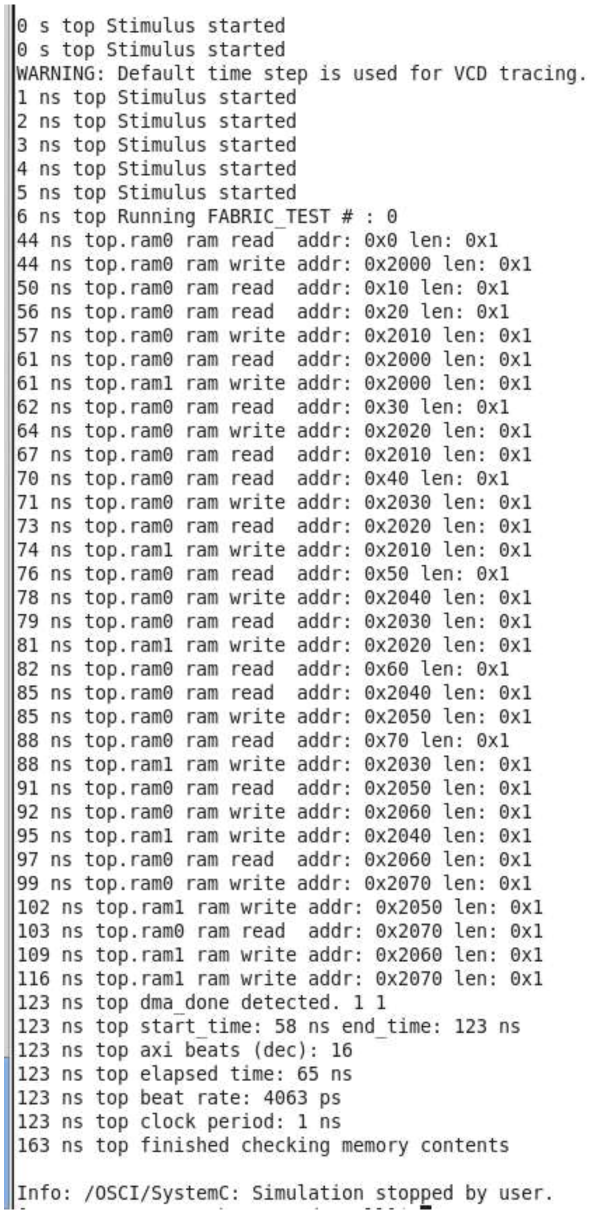
After the above copy operation, the CPU will program DMA1 to perform a similar copy operation from RAM0 address 0x2000 to RAM1 address 0x2000.

At the completion of the test we should see that 16 words at RAM0 address 0x0000 have been copied to RAM0 address 0x2000 (by DMA0), and we should also see those same 16 words copied to RAM1 address 0x2000 (by DMA1).   The testbench is self-checking and at the completion of the simulation it flags an error if the memory contents are not correct.

You can build and run the SystemC simulation executable by typing:

```
make no_stall
```

You will see that the test runs and the self-check at the end completes successfully:

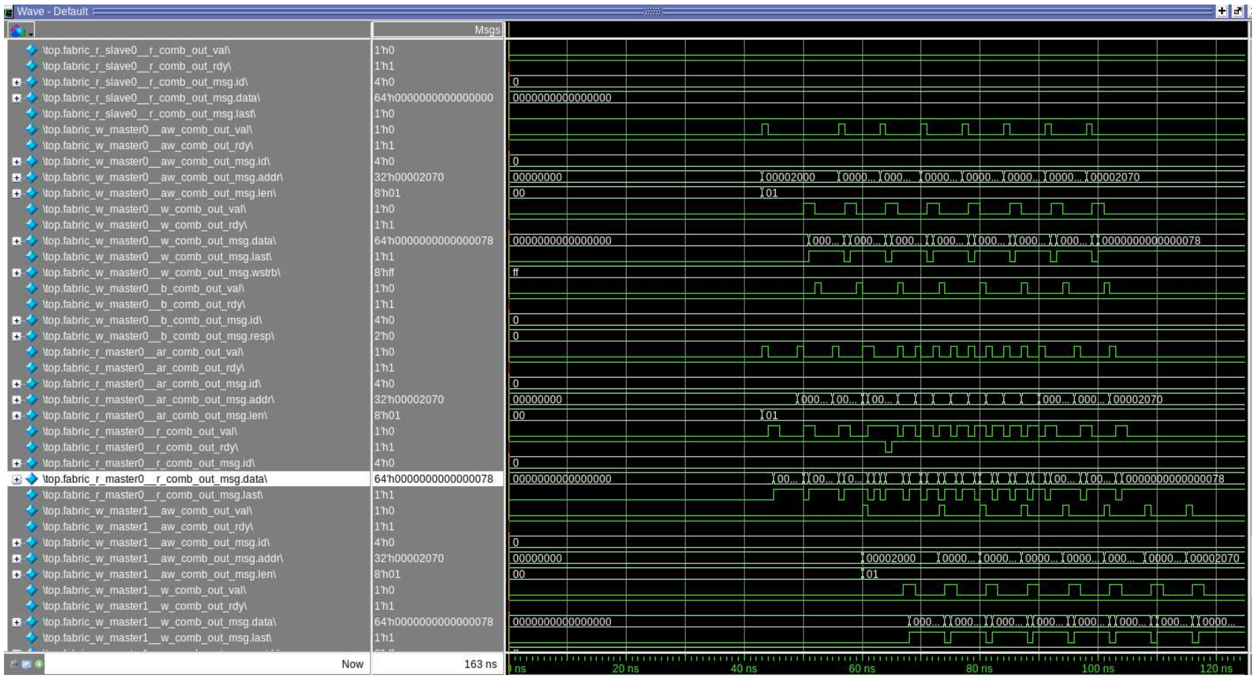


```
0 s top Stimulus started
0 s top Stimulus started
WARNING: Default time step is used for VCD tracing.
1 ns top Stimulus started
2 ns top Stimulus started
3 ns top Stimulus started
4 ns top Stimulus started
5 ns top Stimulus started
6 ns top Running FABRIC_TEST # : 0
44 ns top.ram0 ram read  addr: 0x0 len: 0x1
44 ns top.ram0 ram write addr: 0x2000 len: 0x1
50 ns top.ram0 ram read  addr: 0x10 len: 0x1
56 ns top.ram0 ram read  addr: 0x20 len: 0x1
57 ns top.ram0 ram write addr: 0x2010 len: 0x1
61 ns top.ram0 ram read  addr: 0x2000 len: 0x1
61 ns top.ram1 ram write addr: 0x2000 len: 0x1
62 ns top.ram0 ram read  addr: 0x30 len: 0x1
64 ns top.ram0 ram write addr: 0x2020 len: 0x1
67 ns top.ram0 ram read  addr: 0x2010 len: 0x1
70 ns top.ram0 ram read  addr: 0x40 len: 0x1
71 ns top.ram0 ram write addr: 0x2030 len: 0x1
73 ns top.ram0 ram read  addr: 0x2020 len: 0x1
74 ns top.ram1 ram write addr: 0x2010 len: 0x1
76 ns top.ram0 ram read  addr: 0x50 len: 0x1
78 ns top.ram0 ram write addr: 0x2040 len: 0x1
79 ns top.ram0 ram read  addr: 0x2030 len: 0x1
81 ns top.ram1 ram write addr: 0x2020 len: 0x1
82 ns top.ram0 ram read  addr: 0x60 len: 0x1
85 ns top.ram0 ram read  addr: 0x2040 len: 0x1
85 ns top.ram0 ram write addr: 0x2050 len: 0x1
88 ns top.ram0 ram read  addr: 0x70 len: 0x1
88 ns top.ram1 ram write addr: 0x2030 len: 0x1
91 ns top.ram0 ram read  addr: 0x2050 len: 0x1
92 ns top.ram0 ram write addr: 0x2060 len: 0x1
95 ns top.ram1 ram write addr: 0x2040 len: 0x1
97 ns top.ram0 ram read  addr: 0x2060 len: 0x1
99 ns top.ram0 ram write addr: 0x2070 len: 0x1
102 ns top.ram1 ram write addr: 0x2050 len: 0x1
103 ns top.ram0 ram read  addr: 0x2070 len: 0x1
109 ns top.ram1 ram write addr: 0x2060 len: 0x1
116 ns top.ram1 ram write addr: 0x2070 len: 0x1
123 ns top dma_done detected. 1 1
123 ns top start_time: 58 ns end_time: 123 ns
123 ns top axi beats (dec): 16
123 ns top elapsed time: 65 ns
123 ns top beat rate: 4063 ps
123 ns top clock period: 1 ns
163 ns top finished checking memory contents

Info: /OSCI/SystemC: Simulation stopped by user.
```

You can look at the VCD waveforms produced by the SystemC simulation by typing:

    make no_stall_wave

You will see:



	Msgs
\top.fabric_r_slave0__r_comb_out_val\	1'h0
\top.fabric_r_slave0__r_comb_out_rdy\	1'h1
\top.fabric_r_slave0__r_comb_out_msg.id\	4'h0
\top.fabric_r_slave0__r_comb_out_msg.data\	64'h0000000000000000
\top.fabric_r_slave0__r_comb_out_msg.last\	1'h0
\top.fabric_w_master0__aw_comb_out_val\	1'h0
\top.fabric_w_master0__aw_comb_out_rdy\	1'h1
\top.fabric_w_master0__aw_comb_out_msg.id\	4'h0
\top.fabric_w_master0__aw_comb_out_msg.addr\	32'h00002070
\top.fabric_w_master0__aw_comb_out_msg.len\	8'h01
\top.fabric_w_master0__w_comb_out_val\	1'h0
\top.fabric_w_master0__w_comb_out_rdy\	1'h1
\top.fabric_w_master0__w_comb_out_msg.data\	64'h0000000000000078
\top.fabric_w_master0__w_comb_out_msg.last\	1'h1
\top.fabric_w_master0__w_comb_out_msg.wstrb\	8'hff
\top.fabric_w_master0__b_comb_out_val\	1'h0
\top.fabric_w_master0__b_comb_out_rdy\	1'h1
\top.fabric_w_master0__b_comb_out_msg.id\	4'h0
\top.fabric_w_master0__b_comb_out_msg.resp\	2'h0
\top.fabric_r_master0__ar_comb_out_val\	1'h0
\top.fabric_r_master0__ar_comb_out_rdy\	1'h1
\top.fabric_r_master0__ar_comb_out_msg.id\	4'h0
\top.fabric_r_master0__ar_comb_out_msg.addr\	32'h00002070
\top.fabric_r_master0__ar_comb_out_msg.len\	8'h01
\top.fabric_r_master0__r_comb_out_val\	1'h0
\top.fabric_r_master0__r_comb_out_rdy\	1'h1
\top.fabric_r_master0__r_comb_out_msg.id\	4'h0
\top.fabric_r_master0__r_comb_out_msg.data\	64'h0000000000000078
\top.fabric_r_master0__r_comb_out_msg.last\	1'h1
\top.fabric_w_master1__aw_comb_out_val\	1'h0
\top.fabric_w_master1__aw_comb_out_rdy\	1'h1
\top.fabric_w_master1__aw_comb_out_msg.id\	4'h0
\top.fabric_w_master1__aw_comb_out_msg.addr\	32'h00002070
\top.fabric_w_master1__aw_comb_out_msg.len\	8'h01
\top.fabric_w_master1__w_comb_out_val\	1'h0
\top.fabric_w_master1__w_comb_out_rdy\	1'h1
\top.fabric_w_master1__w_comb_out_msg.data\	64'h0000000000000078
\top.fabric_w_master1__w_comb_out_msg.last\	1'h1

Introduction to Random Stall Injection

Matchlib provides the ability to perform random stall injection on all channels within your SystemC model. When random stall injection is enabled, transactions passing through channels are delayed by a random amount of time (however this random amount is repeatable across simulation runs). This tends to be very effective in uncovering bugs in systems, for example cases where synchronization between various blocks has not been implemented correctly or needs to be added.

You can build and run the SystemC simulation executable that uses random stall injection by typing:

```
make stall
```

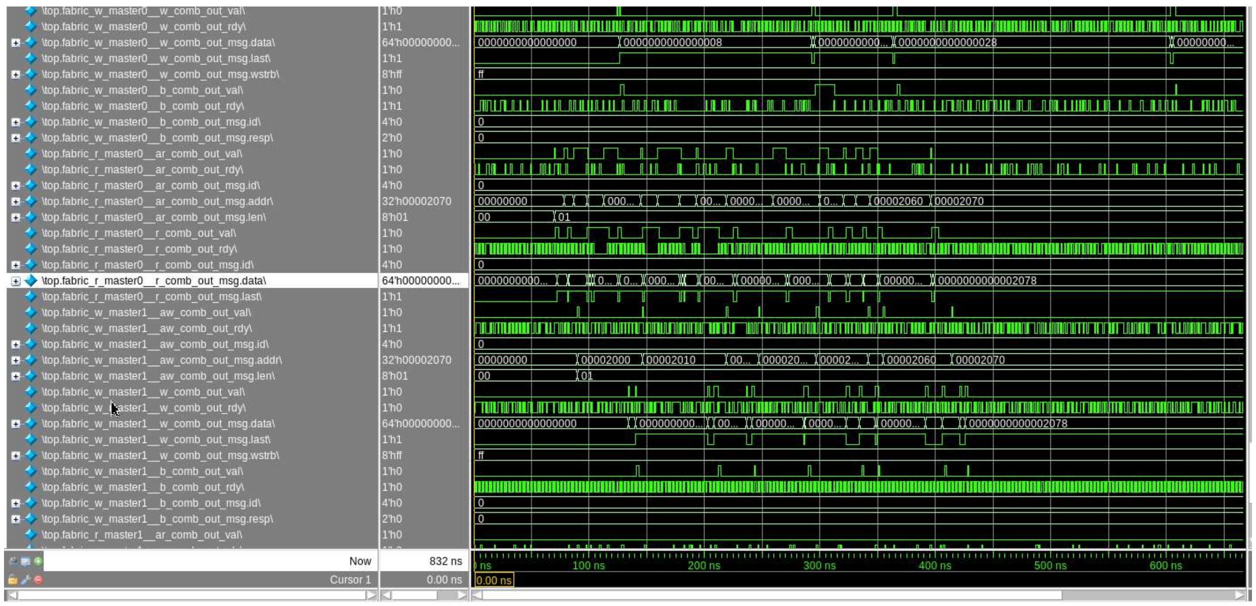
You will see that the test runs and the self-check at the end fails with an error:

```
792 ns top dma_done detected. 1 1
792 ns top start_time: 87 ns end_time: 792 ns
792 ns top axi beats (dec): 16
792 ns top elapsed time: 705 ns
792 ns top beat rate: 44063 ps
792 ns top clock period: 1 ns
832 ns top source and target data mismatch! DMA#: 1 Beat#: 0  s:0 t: 2000
832 ns top source and target data mismatch! DMA#: 1 Beat#: 1  s:8 t: 2008
832 ns top source and target data mismatch! DMA#: 1 Beat#: 2  s:10 t: 2010
832 ns top source and target data mismatch! DMA#: 1 Beat#: 3  s:18 t: 2018
832 ns top source and target data mismatch! DMA#: 1 Beat#: 4  s:20 t: 2020
832 ns top source and target data mismatch! DMA#: 1 Beat#: 5  s:28 t: 2028
832 ns top source and target data mismatch! DMA#: 1 Beat#: 6  s:30 t: 2030
832 ns top source and target data mismatch! DMA#: 1 Beat#: 7  s:38 t: 2038
832 ns top source and target data mismatch! DMA#: 1 Beat#: 8  s:40 t: 2040
832 ns top source and target data mismatch! DMA#: 1 Beat#: 9  s:48 t: 2048
832 ns top source and target data mismatch! DMA#: 1 Beat#: a  s:50 t: 2050
832 ns top source and target data mismatch! DMA#: 1 Beat#: b  s:58 t: 2058
832 ns top source and target data mismatch! DMA#: 1 Beat#: c  s:60 t: 2060
832 ns top source and target data mismatch! DMA#: 1 Beat#: d  s:68 t: 2068
832 ns top source and target data mismatch! DMA#: 1 Beat#: e  s:70 t: 2070
832 ns top source and target data mismatch! DMA#: 1 Beat#: f  s:78 t: 2078
832 ns top finished checking memory contents

Info: /OSCI/SystemC: Simulation stopped by user.
```

You can look at the VCD waveforms produced for this simulation by typing:

    make stall_wave

It will display the following:



```
\top.fabric_w_master0__w_comb_out_val\          1'h0
\top.fabric_w_master0__w_comb_out_rdy\          1'h1
\top.fabric_w_master0__w_comb_out_msg.data\     64'h00000000...
\top.fabric_w_master0__w_comb_out_msg.last\     1'h1
\top.fabric_w_master0__w_comb_out_msg.wstrb\    8'hff
\top.fabric_w_master0__b_comb_out_val\          1'h0
\top.fabric_w_master0__b_comb_out_rdy\          1'h1
\top.fabric_w_master0__b_comb_out_msg.id\       4'h0
\top.fabric_w_master0__b_comb_out_msg.resp\     2'h0
\top.fabric_r_master0__ar_comb_out_val\         1'h0
\top.fabric_r_master0__ar_comb_out_rdy\         1'h0
\top.fabric_r_master0__ar_comb_out_msg.id\      4'h0
\top.fabric_r_master0__ar_comb_out_msg.addr\    32'h00002070
\top.fabric_r_master0__ar_comb_out_msg.len\     8'h01
\top.fabric_r_master0__r_comb_out_val\          1'h0
\top.fabric_r_master0__r_comb_out_rdy\          1'h0
\top.fabric_r_master0__r_comb_out_msg.id\       4'h0
\top.fabric_r_master0__r_comb_out_msg.data\     64'h00000000...
\top.fabric_r_master0__r_comb_out_msg.last\     1'h1
\top.fabric_w_master1__aw_comb_out_val\         1'h0
\top.fabric_w_master1__aw_comb_out_rdy\         1'h1
\top.fabric_w_master1__aw_comb_out_msg.id\      4'h0
\top.fabric_w_master1__aw_comb_out_msg.addr\    32'h00002070
\top.fabric_w_master1__aw_comb_out_msg.len\     8'h01
\top.fabric_w_master1__w_comb_out_val\          1'h0
\top.fabric_w_master1__w_comb_out_rdy\          1'h0
\top.fabric_w_master1__w_comb_out_msg.data\     64'h00000000...
\top.fabric_w_master1__w_comb_out_msg.last\     1'h1
\top.fabric_w_master1__w_comb_out_msg.wstrb\    8'hff
\top.fabric_w_master1__b_comb_out_val\          1'h0
\top.fabric_w_master1__b_comb_out_rdy\          1'h0
\top.fabric_w_master1__b_comb_out_msg.id\       4'h0
\top.fabric_w_master1__b_comb_out_msg.resp\     2'h0
\top.fabric_r_master1__ar_comb_out_val\         1'h0
                                        Now     832 ns
                                   Cursor 1     0.00 ns
```

These are the same signals that were shown in the previous waveforms. You will note several key differences between the “no stall” and “stall” waveforms:

- The stall test case takes significantly longer to complete (800 ns vs 120 ns)
- The stall waveforms are much more irregular – this is because the “ready” signals on the channels are being randomly held low for varying amounts of time.

You will probably also conclude that using waveforms to do the initial debug of the failing (stall) scenario by comparing against the passing (no stall) scenario would be fairly difficult. This would only become even more difficult for larger, more complex SOCs. In practice, waveforms are generally not particularly effective for the early stages of SOC debug efforts, but they can be a useful debug tool for the final stages of the SOC debug process.

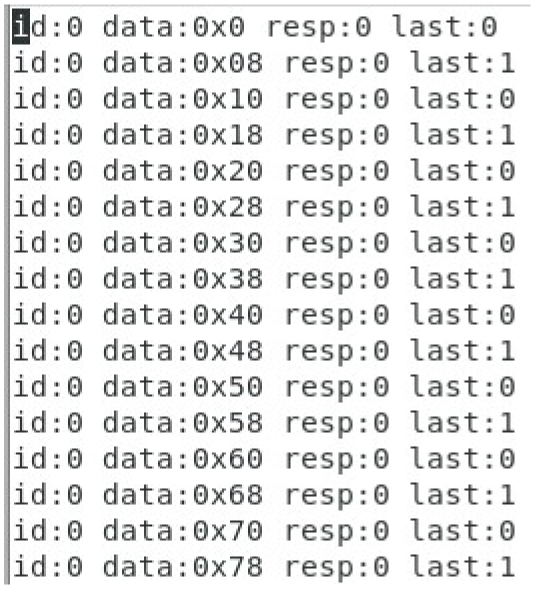
Let's now look at some more effective techniques to debug these kinds of scenarios.

Introduction to Matchlib Channel Logs

Matchlib provides to create a log file for each channel instance in your system. Each such log file is a text file that has the contents of a single transaction on each line. The name of each file is the hierarchical pathname to the channel instance in your system. An example channel log file can be seen by typing:

```
view no_stall_log/top.fabric1.dma0_r_master0_out__r_comb_out_msg_0.txt
```

This will show:



```
id:0 data:0x0 resp:0 last:0
id:0 data:0x08 resp:0 last:1
id:0 data:0x10 resp:0 last:0
id:0 data:0x18 resp:0 last:1
id:0 data:0x20 resp:0 last:0
id:0 data:0x28 resp:0 last:1
id:0 data:0x30 resp:0 last:0
id:0 data:0x38 resp:0 last:1
id:0 data:0x40 resp:0 last:0
id:0 data:0x48 resp:0 last:1
id:0 data:0x50 resp:0 last:0
id:0 data:0x58 resp:0 last:1
id:0 data:0x60 resp:0 last:0
id:0 data:0x68 resp:0 last:1
id:0 data:0x70 resp:0 last:0
id:0 data:0x78 resp:0 last:1
```

This channel is the r_master0 AXI4 interface at the top of the bus fabric, and the transactions are the AXI4 read channel transactions being returned from RAM0. The second field is the actual read data being returned.

For our simple AXI4 SOC example, there are about 60 separate channel logs produced for each unique channel instance in the design. For real world SOCs, there may be hundreds or even thousands of such channel logs produced for a single simulation run.

In most SOCs, most channel instances will have the property that the sequence of transactions should not change even if the timing behavior of the system changes. Such channel instances are called "latency insensitive".

There may be some channel instances on SOCs where timing differences can cause the transaction streams to vary, but typically not in completely arbitrary ways. For example, an arbiter might produce different outputs as the timing of its input requests varies, but over the entire simulation every requesting transaction must successfully pass through the arbiter.

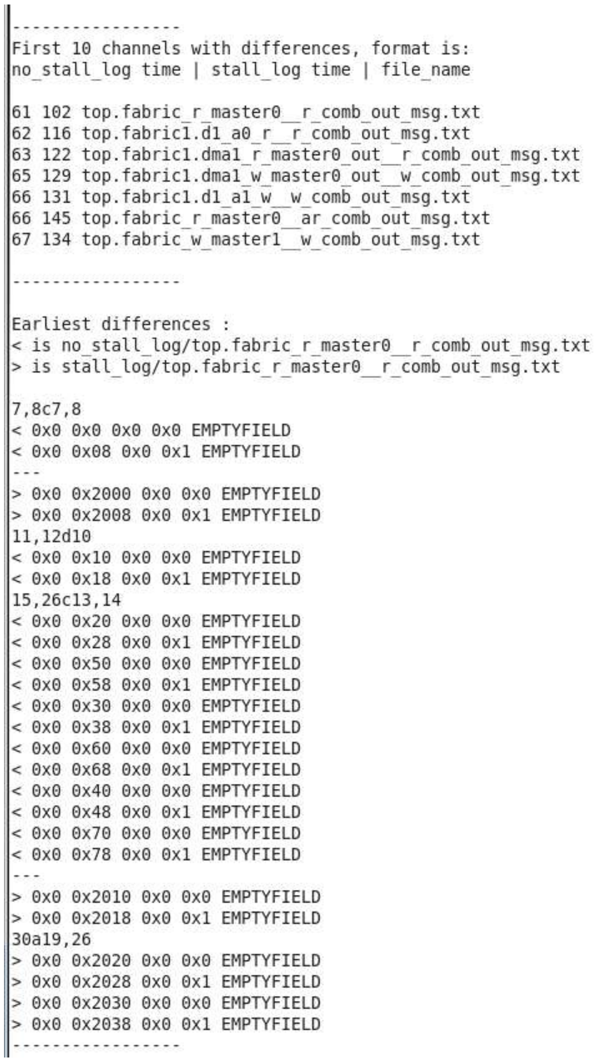
Debugging with Matchlib Channel Logs

Channel logs can be a very effective way to check and debug Matchlib designs. Since every channel instance in a Matchlib system can be enabled to generate a channel log, all of the transactions passing though these instances become observable. Even if you use self-checking testbenches, the vastly increased observability provided by the channel logs can be very valuable. For example, you may have a set of tests that are all passing at one point in time. You can generate channel logs for all of the tests and treat them as “golden” files, and use them to check that the same set of tests still produce matching logs as your models evolve over time.

Mentor provides scripts to help automate checking and debugging using channel logs. Let’s use the “earliest_diff.sh” script to debug the failing simulation by comparing it to the passing simulation:

```
make diff
```

This will produce the following:



```
-----------------
First 10 channels with differences, format is:
no_stall_log time | stall_log time | file_name

61 102 top.fabric_r_master0__r_comb_out_msg.txt
62 116 top.fabric1.d1_a0_r__r_comb_out_msg.txt
63 122 top.fabric1.dma1_r_master0_out__r_comb_out_msg.txt
65 129 top.fabric1.dma1_w_master0_out__w_comb_out_msg.txt
66 131 top.fabric1.d1_a1_w__w_comb_out_msg.txt
66 145 top.fabric_r_master0__ar_comb_out_msg.txt
67 134 top.fabric_w_master1__w_comb_out_msg.txt

-----------------

Earliest differences :
< is no_stall_log/top.fabric_r_master0__r_comb_out_msg.txt
> is stall_log/top.fabric_r_master0__r_comb_out_msg.txt

7,8c7,8
< 0x0 0x0 0x0 0x0 EMPTYFIELD
< 0x0 0x08 0x0 0x1 EMPTYFIELD
---
> 0x0 0x2000 0x0 0x0 EMPTYFIELD
> 0x0 0x2008 0x0 0x1 EMPTYFIELD
11,12d10
< 0x0 0x10 0x0 0x0 EMPTYFIELD
< 0x0 0x18 0x0 0x1 EMPTYFIELD
15,26c13,14
< 0x0 0x20 0x0 0x0 EMPTYFIELD
< 0x0 0x28 0x0 0x1 EMPTYFIELD
< 0x0 0x50 0x0 0x0 EMPTYFIELD
< 0x0 0x58 0x0 0x1 EMPTYFIELD
< 0x0 0x30 0x0 0x0 EMPTYFIELD
< 0x0 0x38 0x0 0x1 EMPTYFIELD
< 0x0 0x60 0x0 0x0 EMPTYFIELD
< 0x0 0x68 0x0 0x1 EMPTYFIELD
< 0x0 0x40 0x0 0x0 EMPTYFIELD
< 0x0 0x48 0x0 0x1 EMPTYFIELD
< 0x0 0x70 0x0 0x0 EMPTYFIELD
< 0x0 0x78 0x0 0x1 EMPTYFIELD
---
> 0x0 0x2010 0x0 0x0 EMPTYFIELD
> 0x0 0x2018 0x0 0x1 EMPTYFIELD
30a19,26
> 0x0 0x2020 0x0 0x0 EMPTYFIELD
> 0x0 0x2028 0x0 0x1 EMPTYFIELD
> 0x0 0x2030 0x0 0x0 EMPTYFIELD
> 0x0 0x2038 0x0 0x1 EMPTYFIELD
-----------------
```

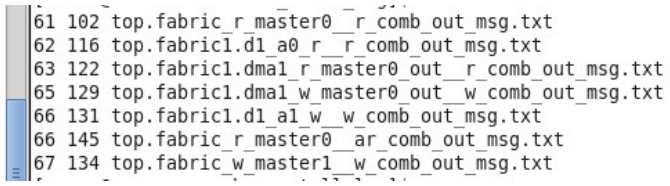
Let’s look at the channel logs in more detail. First, change directory to the no_stall_log

```
cd no_stall_log
```

Next, do:

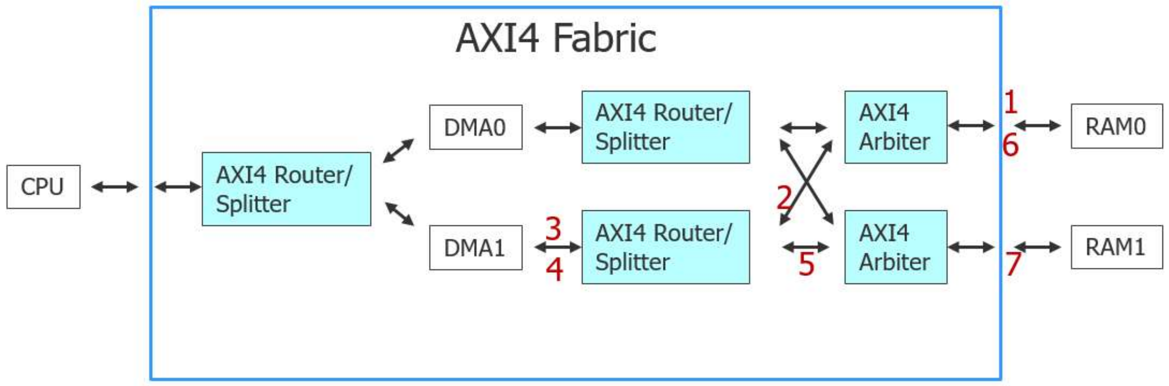
```
cat earliest.sorted
```

You will see:



```
61 102 top.fabric_r_master0__r_comb_out_msg.txt
62 116 top.fabric1.d1_a0_r__r_comb_out_msg.txt
63 122 top.fabric1.dma1_r_master0_out__r_comb_out_msg.txt
65 129 top.fabric1.dma1_w_master0_out__w_comb_out_msg.txt
66 131 top.fabric1.d1_a1_w__w_comb_out_msg.txt
66 145 top.fabric_r_master0__ar_comb_out_msg.txt
67 134 top.fabric_w_master1__w_comb_out_msg.txt
```

This is a sorted list of channel instances that differ between the no_stall simulation and the stall simulation. The first number is the simulation time when the first difference occurred on that channel, where the time is with respect to the no_stall simulation. The second number is the simulation time with respect to the stall simulation (keep in mind that times vary between the two simulations, and that that is intentional). The last item is the name of the channel instance. There are seven channel instances listed. Let’s label them 1-7. Their names are the hierarchal paths to the channel instances, and are labeled on the block diagram here:

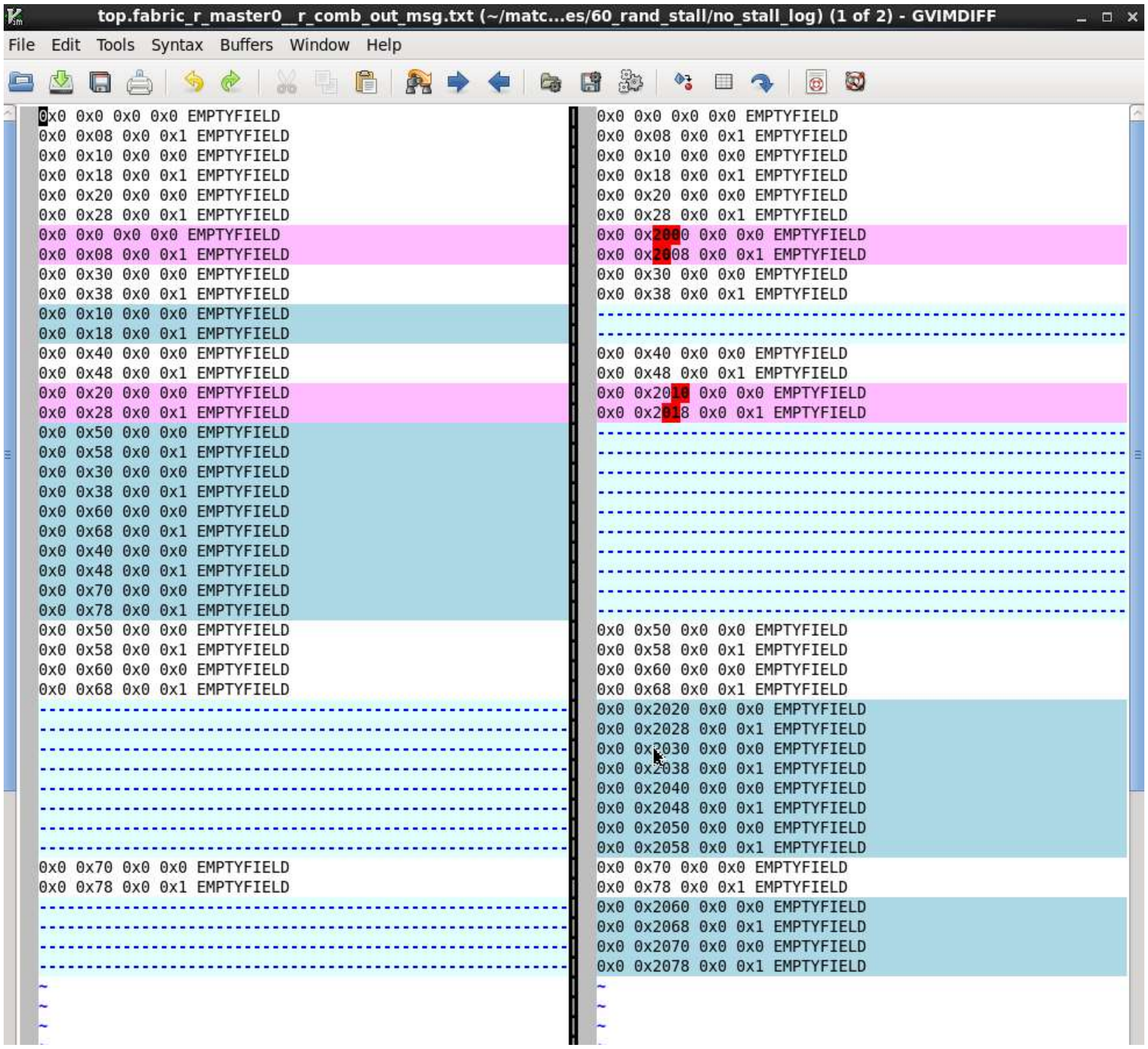


I/**

Let’s do a side by side comparison of the log that has the earliest differences by typing:

```
gvimdiff top.fabric_r_master0__r_comb_out_msg.txt ../stall_log
```

This will show:


top.fabric_r_master0__r_comb_out_msg.txt (~/matc...es/60_rand_stall/no_stall_log) (1 of 2) - GVIMDIFF
File Edit Tools Syntax Buffers Window Help

```
0x0 0x0 0x0 0x0 EMPTYFIELD
0x0 0x08 0x0 0x1 EMPTYFIELD
0x0 0x10 0x0 0x0 EMPTYFIELD
0x0 0x18 0x0 0x1 EMPTYFIELD
0x0 0x20 0x0 0x0 EMPTYFIELD
0x0 0x28 0x0 0x1 EMPTYFIELD
0x0 0x0 0x0 0x0 EMPTYFIELD
0x0 0x08 0x0 0x1 EMPTYFIELD
0x0 0x30 0x0 0x0 EMPTYFIELD
0x0 0x38 0x0 0x1 EMPTYFIELD
0x0 0x10 0x0 0x0 EMPTYFIELD
0x0 0x18 0x0 0x1 EMPTYFIELD
0x0 0x40 0x0 0x0 EMPTYFIELD
0x0 0x48 0x0 0x1 EMPTYFIELD
0x0 0x20 0x0 0x0 EMPTYFIELD
0x0 0x28 0x0 0x1 EMPTYFIELD
0x0 0x50 0x0 0x0 EMPTYFIELD
0x0 0x58 0x0 0x1 EMPTYFIELD
0x0 0x30 0x0 0x0 EMPTYFIELD
0x0 0x38 0x0 0x1 EMPTYFIELD
0x0 0x60 0x0 0x0 EMPTYFIELD
0x0 0x68 0x0 0x1 EMPTYFIELD
0x0 0x40 0x0 0x0 EMPTYFIELD
0x0 0x48 0x0 0x1 EMPTYFIELD
0x0 0x70 0x0 0x0 EMPTYFIELD
0x0 0x78 0x0 0x1 EMPTYFIELD
0x0 0x50 0x0 0x0 EMPTYFIELD
0x0 0x58 0x0 0x1 EMPTYFIELD
0x0 0x60 0x0 0x0 EMPTYFIELD
0x0 0x68 0x0 0x1 EMPTYFIELD
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
0x0 0x70 0x0 0x0 EMPTYFIELD
0x0 0x78 0x0 0x1 EMPTYFIELD
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
~
~
~
```

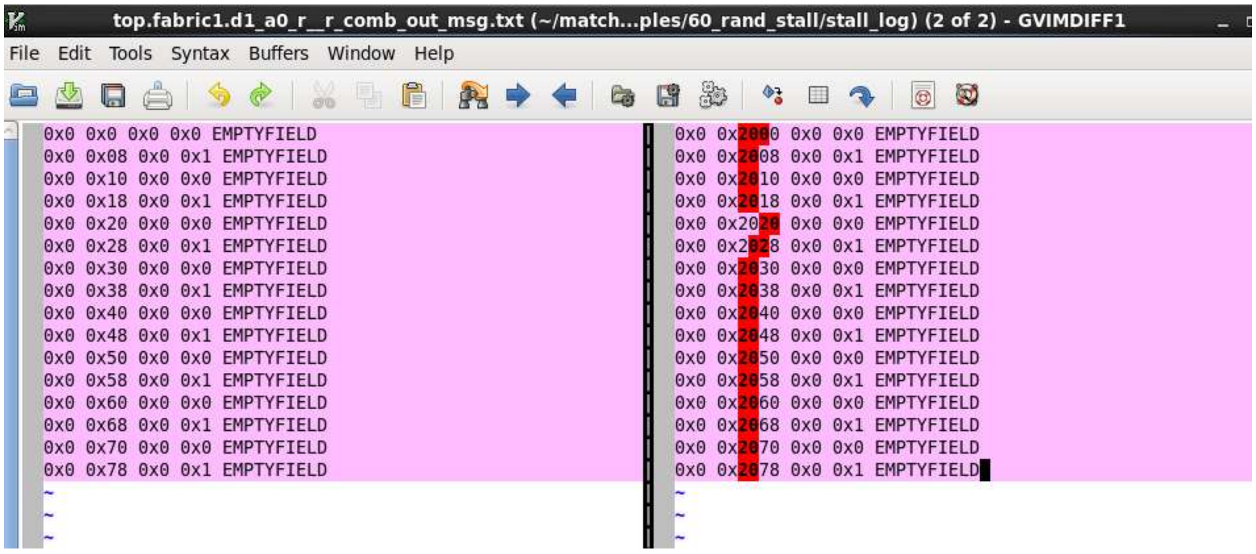
```
0x0 0x0 0x0 0x0 EMPTYFIELD
0x0 0x08 0x0 0x1 EMPTYFIELD
0x0 0x10 0x0 0x0 EMPTYFIELD
0x0 0x18 0x0 0x1 EMPTYFIELD
0x0 0x20 0x0 0x0 EMPTYFIELD
0x0 0x28 0x0 0x1 EMPTYFIELD
0x0 0x2000 0x0 0x0 EMPTYFIELD
0x0 0x2008 0x0 0x1 EMPTYFIELD
0x0 0x30 0x0 0x0 EMPTYFIELD
0x0 0x38 0x0 0x1 EMPTYFIELD
---------------------------------------------------------
---------------------------------------------------------
0x0 0x40 0x0 0x0 EMPTYFIELD
0x0 0x48 0x0 0x1 EMPTYFIELD
0x0 0x2010 0x0 0x0 EMPTYFIELD
0x0 0x2018 0x0 0x1 EMPTYFIELD
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
---------------------------------------------------------
0x0 0x50 0x0 0x0 EMPTYFIELD
0x0 0x58 0x0 0x1 EMPTYFIELD
0x0 0x60 0x0 0x0 EMPTYFIELD
0x0 0x68 0x0 0x1 EMPTYFIELD
0x0 0x2020 0x0 0x0 EMPTYFIELD
0x0 0x2028 0x0 0x1 EMPTYFIELD
0x0 0x2030 0x0 0x0 EMPTYFIELD
0x0 0x2038 0x0 0x1 EMPTYFIELD
0x0 0x2040 0x0 0x0 EMPTYFIELD
0x0 0x2048 0x0 0x1 EMPTYFIELD
0x0 0x2050 0x0 0x0 EMPTYFIELD
0x0 0x2058 0x0 0x1 EMPTYFIELD
0x0 0x70 0x0 0x0 EMPTYFIELD
0x0 0x78 0x0 0x1 EMPTYFIELD
0x0 0x2060 0x0 0x0 EMPTYFIELD
0x0 0x2068 0x0 0x1 EMPTYFIELD
0x0 0x2070 0x0 0x0 EMPTYFIELD
0x0 0x2078 0x0 0x1 EMPTYFIELD
~
~
~
```

These are the AXI4 read data beats returning from RAM0. The no_stall transactions are on the left, the stall transactions are on the right. Because both DMA0 and DMA1 may be concurrently reading from RAM0, we may expect to see some ordering differences between the two simulations, but here we see on the right some read data values starting with 0x2000 that we do not see on the left. That looks suspicious.

Let’s look at the next channel log in the list:

```
gvimdiff top.fabric1.d1_a0_r__r_comb_out_msg.txt ../stall_log
```

This will show:


top.fabric1.d1_a0_r__r_comb_out_msg.txt (~/match...ples/60_rand_stall/stall_log) (2 of 2) - GVIMDIFF1
File Edit Tools Syntax Buffers Window Help

```
0x0 0x0 0x0 0x0 EMPTYFIELD            0x0 0x2000 0x0 0x0 EMPTYFIELD
0x0 0x08 0x0 0x1 EMPTYFIELD           0x0 0x2008 0x0 0x1 EMPTYFIELD
0x0 0x10 0x0 0x0 EMPTYFIELD           0x0 0x2010 0x0 0x0 EMPTYFIELD
0x0 0x18 0x0 0x1 EMPTYFIELD           0x0 0x2018 0x0 0x1 EMPTYFIELD
0x0 0x20 0x0 0x0 EMPTYFIELD           0x0 0x2020 0x0 0x0 EMPTYFIELD
0x0 0x28 0x0 0x1 EMPTYFIELD           0x0 0x2028 0x0 0x1 EMPTYFIELD
0x0 0x30 0x0 0x0 EMPTYFIELD           0x0 0x2030 0x0 0x0 EMPTYFIELD
0x0 0x38 0x0 0x1 EMPTYFIELD           0x0 0x2038 0x0 0x1 EMPTYFIELD
0x0 0x40 0x0 0x0 EMPTYFIELD           0x0 0x2040 0x0 0x0 EMPTYFIELD
0x0 0x48 0x0 0x1 EMPTYFIELD           0x0 0x2048 0x0 0x1 EMPTYFIELD
0x0 0x50 0x0 0x0 EMPTYFIELD           0x0 0x2050 0x0 0x0 EMPTYFIELD
0x0 0x58 0x0 0x1 EMPTYFIELD           0x0 0x2058 0x0 0x1 EMPTYFIELD
0x0 0x60 0x0 0x0 EMPTYFIELD           0x0 0x2060 0x0 0x0 EMPTYFIELD
0x0 0x68 0x0 0x1 EMPTYFIELD           0x0 0x2068 0x0 0x1 EMPTYFIELD
0x0 0x70 0x0 0x0 EMPTYFIELD           0x0 0x2070 0x0 0x0 EMPTYFIELD
0x0 0x78 0x0 0x1 EMPTYFIELD           0x0 0x2078 0x0 0x1 EMPTYFIELD
~                                     ~
~                                     ~
~                                     ~
```

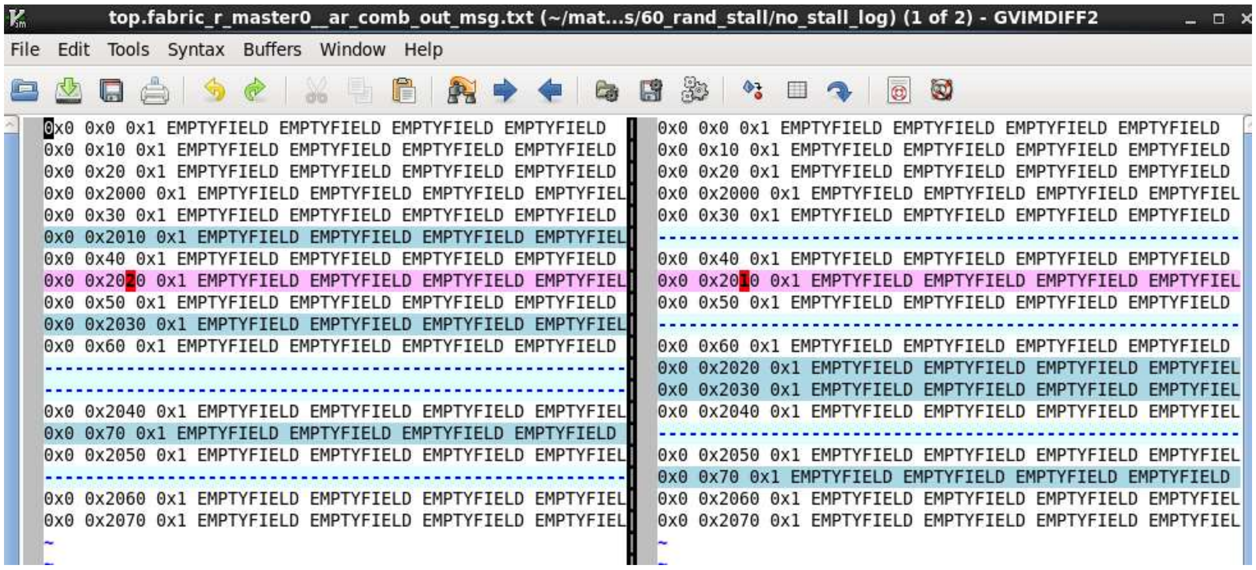
This is the stream of read data transactions that DMA1 is receiving. Every single transaction is different, and off by exactly 0x2000. Since the data in the stall sim at this point is all bad, all of the rest of the channel instances that receive this data will have the same bad stream of data. If you look at the differences for channels 3, 4, 5, and 7 you will see the same stream of data. They show the bad data being written to RAM1, which then causes the testbench self-check to fail.

So, based on the above analysis we can work backwards towards the root cause of the bug: RAM1 is receiving bad data from channel 7, which got it from channel 5, 4, 3, 2, and all the way back to channel 1 starting with bad read data at simulation time 61 as reported by the earliest_diff.sh script. To be precise, the earliest_diff.sh script reports that first difference is at time 61 with respect to the no_stall simulation and at time 102 with respect to the stall simulation. We will use these two times in later debugging steps.

Channel 6 is not involved in this communication path, but it does have a difference reported for it. Let's look at it:

```
gvimdiff  top.fabric_r_master0__ar_comb_out_msg.txt ../stall_log
```

This will show:


top.fabric_r_master0__ar_comb_out_msg.txt (~/mat...s/60_rand_stall/no_stall_log) (1 of 2) - GVIMDIFF2
File Edit Tools Syntax Buffers Window Help

```
0x0 0x0 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x10 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x20 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2000 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x30 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2010 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x40 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2020 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x50 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2030 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x60 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
---------------------------------------------------------
---------------------------------------------------------
0x0 0x2040 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x70 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2050 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
---------------------------------------------------------
0x0 0x2060 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x2070 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
~
~
```

```
0x0 0x0 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x10 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x20 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2000 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x30 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
---------------------------------------------------------
0x0 0x40 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2010 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x50 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
---------------------------------------------------------
0x0 0x60 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2020 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x2030 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x2040 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
---------------------------------------------------------
0x0 0x2050 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x70 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIELD
0x0 0x2060 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
0x0 0x2070 0x1 EMPTYFIELD EMPTYFIELD EMPTYFIELD EMPTYFIEL
~
~
```

These are the AXI4 Address Read requests to RAM0 (or "ar" requests). The second field is the read address. If you look carefully at the two logs, you will note that the lines are identical but the ordering is different. You can confirm that this is the case by sorting the transactions and then comparing them:

```
sort top.fabric_r_master0__ar_comb_out_msg.txt > t1

sort ../stall_log/top.fabric_r_master0__ar_comb_out_msg.txt > t2

diff t1 t2
```

This is a case where the differences are acceptable – both DMA0 and DMA1 are making read requests to RAM0, and because of the timing variation between the two simulations we should not expect them to arrive at RAM0 in exactly the same order. So, the channel 6 logs do not seem to be involved in the bug.

Typically in SOC development efforts there will be a subset of channel instances that require special handling for comparison between simulation runs, as we demonstrated above. These channels are identified as the project evolves and the comparison scripts are updated to either exclude the channels from the comparison process, compare only after the transactions have been sorted, etc.

Debugging Matchlib Models with the C++ Debugger

Now that we've narrowed down the problem to bad read data from RAM0 at time 61 using the channel logs, we will use the C++ debugger to find the exact cause. There are many debuggers available for debugging SystemC models – all of the major EDA HDL simulators have them built in, and open source debuggers such as gdb, Eclipse, and VSCode are widely used. Almost all of them are built on top of gdb. Because of this, the command syntax for all of them should be identical. For this tutorial we will use ddd, which is very lightweight graphical debugger built on top of gdb.

Regardless of which C++ debugger you use, C++ debugging can sometimes be frustrating due to some of the features of the language such as function inlining, function overloading, etc. Sometimes the C++

debuggers do not fully support these cases. In this tutorial we demonstrate very simple, very reliable usage of the gdb debugger which avoids its pitfalls.

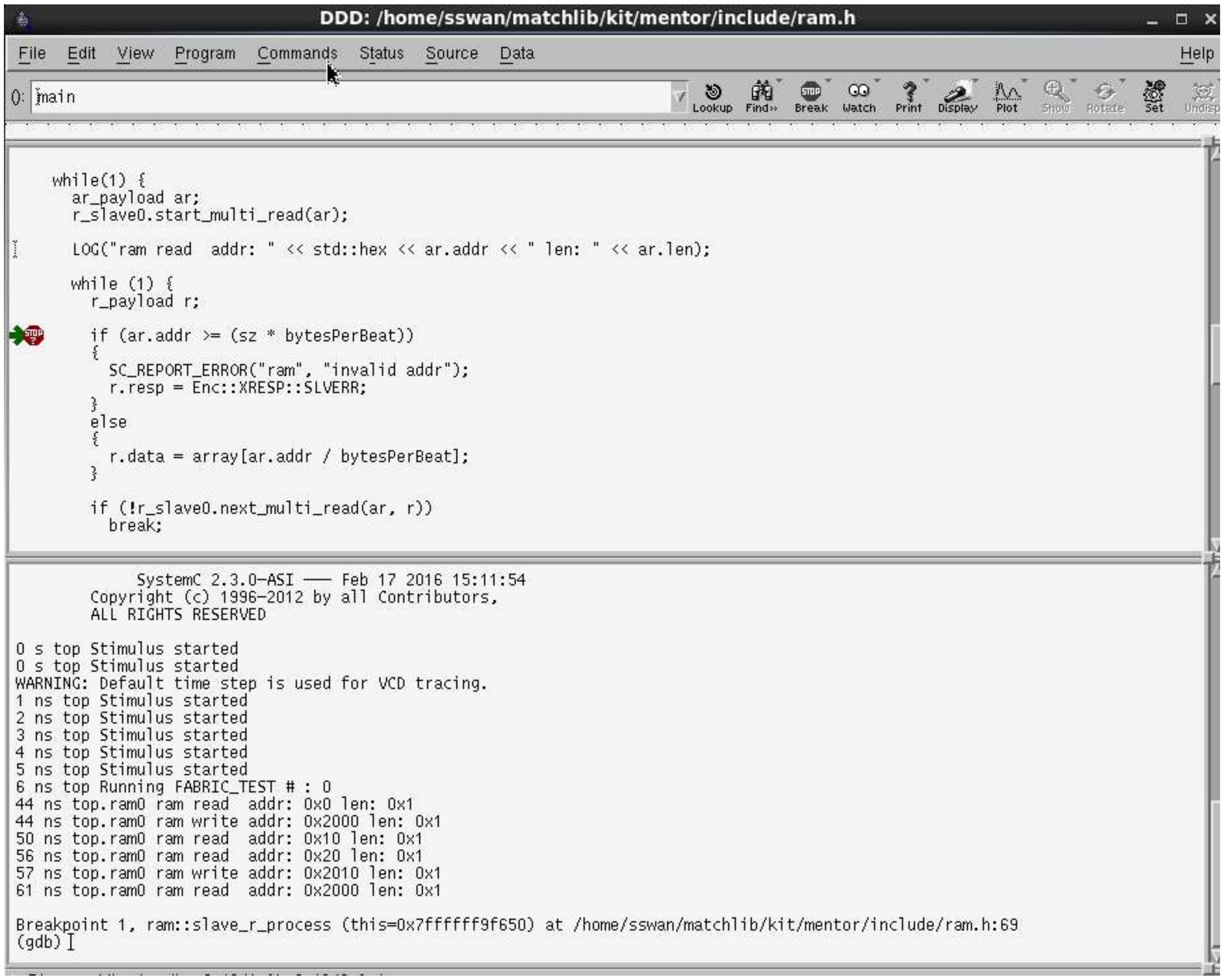
Let's debug the no_stall simulation first:

```
ddd sim_no_stall &
```

Once it starts, let's set a breakpoint for reads in RAM0 at time 61 (which is where we narrowed down the bug in the channel logs):

```
br ram.h:62 if ac_dbg_time_ns() == 61

run
```

This will then show:



```
    while(1) {
      ar_payload ar;
      r_slave0.start_multi_read(ar);

      LOG("ram read  addr: " << std::hex << ar.addr << " len: " << ar.len);

      while (1) {
        r_payload r;

        if (ar.addr >= (sz * bytesPerBeat))
        {
          SC_REPORT_ERROR("ram", "invalid addr");
          r.resp = Enc::XRESP::SLVERR;
        }
        else
        {
          r.data = array[ar.addr / bytesPerBeat];
        }

        if (!r_slave0.next_multi_read(ar, r))
          break;
```

```
             SystemC 2.3.0-ASI --- Feb 17 2016 15:11:54
        Copyright (c) 1996-2012 by all Contributors,
        ALL RIGHTS RESERVED

0 s top Stimulus started
0 s top Stimulus started
WARNING: Default time step is used for VCD tracing.
1 ns top Stimulus started
2 ns top Stimulus started
3 ns top Stimulus started
4 ns top Stimulus started
5 ns top Stimulus started
6 ns top Running FABRIC_TEST # : 0
44 ns top.ram0 ram read  addr: 0x0 len: 0x1
44 ns top.ram0 ram write addr: 0x2000 len: 0x1
50 ns top.ram0 ram read  addr: 0x10 len: 0x1
56 ns top.ram0 ram read  addr: 0x20 len: 0x1
57 ns top.ram0 ram write addr: 0x2010 len: 0x1
61 ns top.ram0 ram read  addr: 0x2000 len: 0x1

Breakpoint 1, ram::slave_r_process (this=0x7ffffff9f650) at /home/sswan/matchlib/kit/mentor/include/ram.h:69
(gdb) 
```

You can confirm that the current "ar" request is for address 0x2000 by typing:

```
p/x  ar.addr.to_uint64()
```

You can also confirm that we are in RAM0 (not RAM1) by typing:

```
p ac_dbg_name(this)
```

You can check the simulation time by typing:

```
p ac_dbg_time()
```

Everything looks correct. Now let's check the data that will be read:

```
p/x array[0x2000 / 8].to_uint64()
```

This reports 0x0, which is the expected data. (Remember that DMA0 has copied data from address 0x0000 to address 0x2000, so we now expect data at address 0x2000 to be zero).

Let's restart the simulation and check that the above statement is true:

```
kill

br ram.h:95 if aw.addr.to_uint64() == 0x2000

run
```

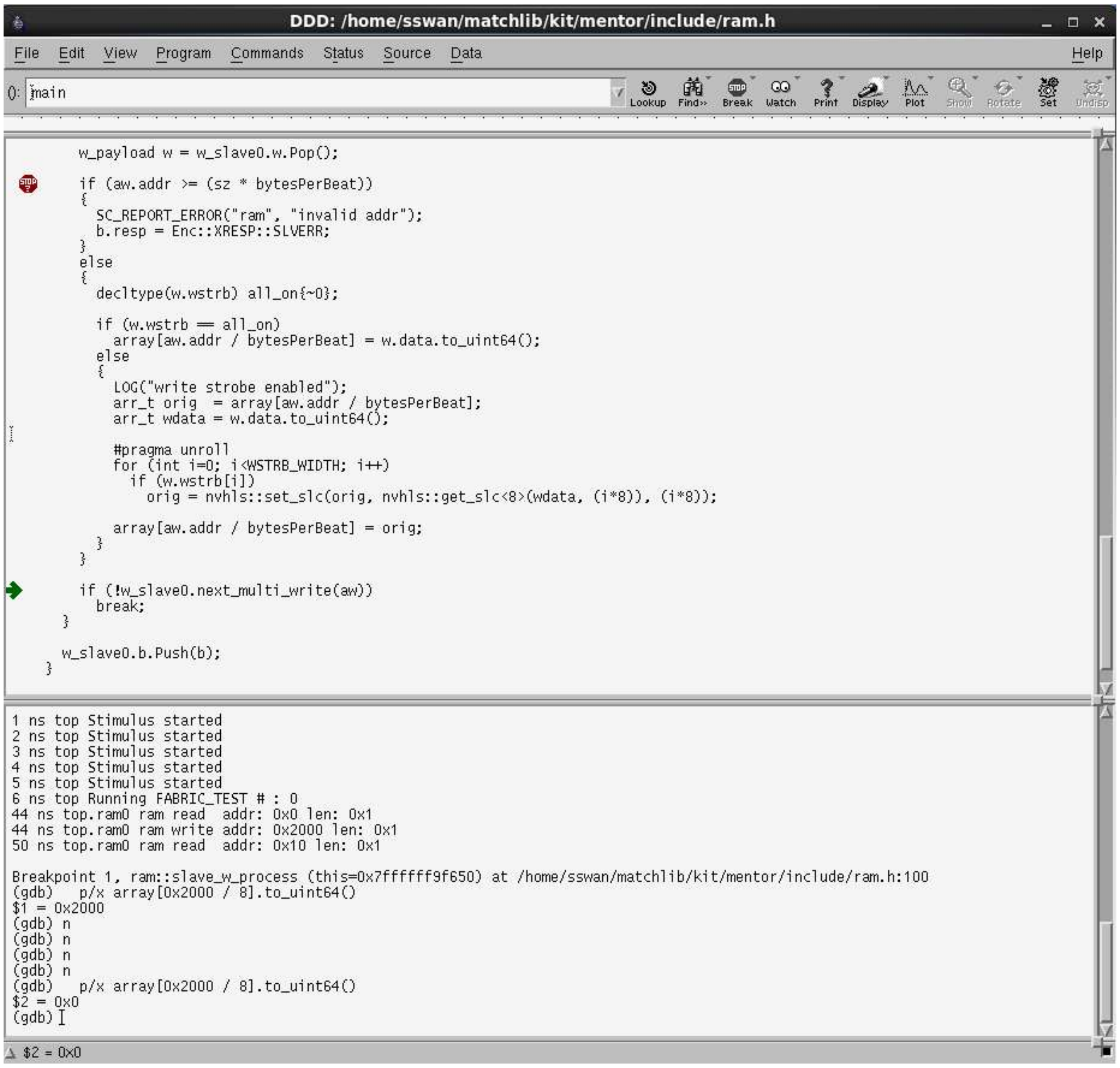
When the breakpoint is hit we can check what simulation time it is:

```
p ac_dbg_time()
```

The time reported is 51 ns, which is before 61 ns, so we can see that location 0x2000 is being properly written to by DMA0 before it is read by DMA1. We should also confirm that this write is happening to RAM0 and not RAM1:

```
p ac_dbg_name(this)
```

The snapshot below shows that we can print out the value at address 0x2000 in the RAM0 data array before the write occurs (which is 0x2000 as expected), step thru as the write is completed, and then print out the updated value (which is 0x0, as expected).


DDD: /home/sswan/matchlib/kit/mentor/include/ram.h
File Edit View Program Commands Status Source Data Help
(): main
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp

```
      w_payload w = w_slave0.w.Pop();

      if (aw.addr >= (sz * bytesPerBeat))
      {
        SC_REPORT_ERROR("ram", "invalid addr");
        b.resp = Enc::XRESP::SLVERR;
      }
      else
      {
        decltype(w.wstrb) all_on{~0};

        if (w.wstrb == all_on)
          array[aw.addr / bytesPerBeat] = w.data.to_uint64();
        else
        {
          LOG("write strobe enabled");
          arr_t orig  = array[aw.addr / bytesPerBeat];
          arr_t wdata = w.data.to_uint64();

          #pragma unroll
          for (int i=0; i<WSTRB_WIDTH; i++)
            if (w.wstrb[i])
              orig = nvhls::set_slc(orig, nvhls::get_slc<8>(wdata, (i*8)), (i*8));

          array[aw.addr / bytesPerBeat] = orig;
        }
      }

      if (!w_slave0.next_multi_write(aw))
        break;
    }

    w_slave0.b.Push(b);
  }
```

```
1 ns top Stimulus started
2 ns top Stimulus started
3 ns top Stimulus started
4 ns top Stimulus started
5 ns top Stimulus started
6 ns top Running FABRIC_TEST # : 0
44 ns top.ram0 ram read  addr: 0x0 len: 0x1
44 ns top.ram0 ram write addr: 0x2000 len: 0x1
50 ns top.ram0 ram read  addr: 0x10 len: 0x1

Breakpoint 1, ram::slave_w_process (this=0x7ffffff9f650) at /home/sswan/matchlib/kit/mentor/include/ram.h:100
(gdb)   p/x array[0x2000 / 8].to_uint64()
$1 = 0x2000
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb)   p/x array[0x2000 / 8].to_uint64()
$2 = 0x0
(gdb) 
```

```
$2 = 0x0
```

Everything looks correct. Let’s quit the debugger:

quit

Let’s now debug the failing “stall” simulation.

ddd sim_stall &

The channel log debugging told us that the bad read data occurred from RAM0 at time 102. Let’s set a breakpoint for that read:

br ram.h:64 if ac_dbg_time_ns() == 102

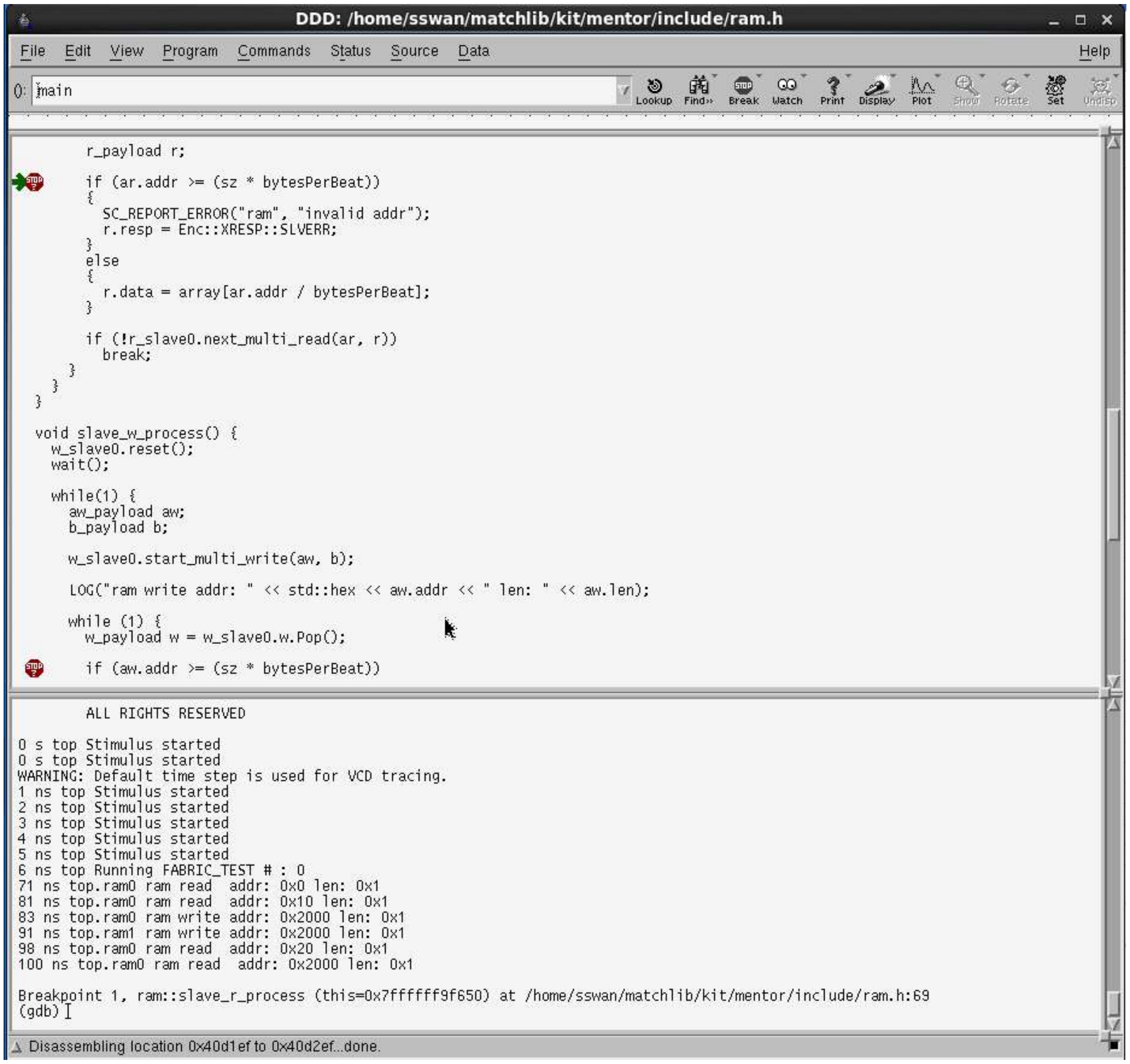
Let’s also set a breakpoint for the write transaction that should occur before it. We don’t know when it occurs, but we know that it is to address 0x2000 in RAM0 (note that there are also writes to RAM1 at this same address, so we need to specify the breakpoint more carefully here):

```
br ram.h:95 if (aw.addr.to_uint64() == 0x2000) && ac_dbg_name_is(this,  "top.ram0")
```

Now let's run the simulation:

```
run
```

We will see:


DDD: /home/sswan/matchlib/kit/mentor/include/ram.h
File Edit View Program Commands Status Source Data Help
(): main
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp

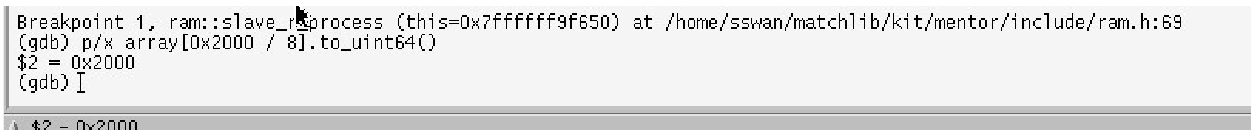
```
      r_payload r;

      if (ar.addr >= (sz * bytesPerBeat))
      {
        SC_REPORT_ERROR("ram", "invalid addr");
        r.resp = Enc::XRESP::SLVERR;
      }
      else
      {
        r.data = array[ar.addr / bytesPerBeat];
      }

      if (!r_slave0.next_multi_read(ar, r))
        break;
    }
  }
}

void slave_w_process() {
  w_slave0.reset();
  wait();

  while(1) {
    aw_payload aw;
    b_payload b;

    w_slave0.start_multi_write(aw, b);

    LOG("ram write addr: " << std::hex << aw.addr << " len: " << aw.len);

    while (1) {
      w_payload w = w_slave0.w.Pop();

      if (aw.addr >= (sz * bytesPerBeat))
```

```
        ALL RIGHTS RESERVED

0 s top Stimulus started
0 s top Stimulus started
WARNING: Default time step is used for VCD tracing.
1 ns top Stimulus started
2 ns top Stimulus started
3 ns top Stimulus started
4 ns top Stimulus started
5 ns top Stimulus started
6 ns top Running FABRIC_TEST # : 0
71 ns top.ram0 ram read  addr: 0x0 len: 0x1
81 ns top.ram0 ram read  addr: 0x10 len: 0x1
83 ns top.ram0 ram write addr: 0x2000 len: 0x1
91 ns top.ram1 ram write addr: 0x2000 len: 0x1
98 ns top.ram0 ram read  addr: 0x20 len: 0x1
100 ns top.ram0 ram read  addr: 0x2000 len: 0x1

Breakpoint 1, ram::slave_r_process (this=0x7ffffff9f650) at /home/sswan/matchlib/kit/mentor/include/ram.h:69
(gdb) 
```

Disassembling location 0x40d1ef to 0x40d2ef...done.

This is bad. We hit the breakpoint in the read first, and for correct results we needed to hit the breakpoint in the write first (as in the passing simulation).

We can print out the value in the RAM0 data array at address 0x2000 and see that it is not what we want it to be:

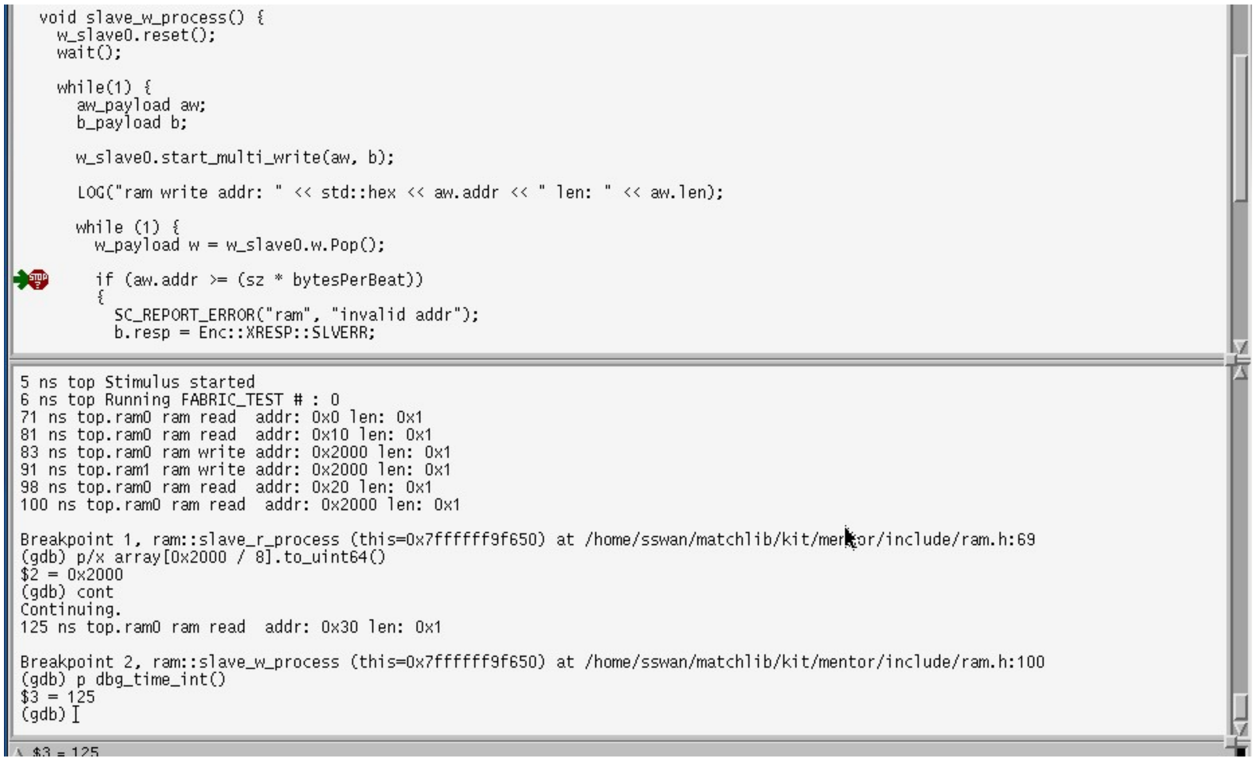


```
Breakpoint 1, ram::slave_r_process (this=0x7ffffff9f650) at /home/sswan/matchlib/kit/mentor/include/ram.h:69
(gdb) p/x array[0x2000 / 8].to_uint64()
$2 = 0x2000
(gdb)
```

Let's let the simulation continue by typing:

    cont

We will see:



```
  void slave_w_process() {
    w_slave0.reset();
    wait();

    while(1) {
      aw_payload aw;
      b_payload b;

      w_slave0.start_multi_write(aw, b);

      LOG("ram write addr: " << std::hex << aw.addr << " len: " << aw.len);

      while (1) {
        w_payload w = w_slave0.w.Pop();

        if (aw.addr >= (sz * bytesPerBeat))
        {
          SC_REPORT_ERROR("ram", "invalid addr");
          b.resp = Enc::XRESP::SLVERR;
```

```
5 ns top Stimulus started
6 ns top Running FABRIC_TEST # : 0
71 ns top.ram0 ram read  addr: 0x0 len: 0x1
81 ns top.ram0 ram read  addr: 0x10 len: 0x1
83 ns top.ram0 ram write addr: 0x2000 len: 0x1
91 ns top.ram1 ram write addr: 0x2000 len: 0x1
98 ns top.ram0 ram read  addr: 0x20 len: 0x1
100 ns top.ram0 ram read  addr: 0x2000 len: 0x1

Breakpoint 1, ram::slave_r_process (this=0x7ffffff9f650) at /home/sswan/matchlib/kit/mentor/include/ram.h:69
(gdb) p/x array[0x2000 / 8].to_uint64()
$2 = 0x2000
(gdb) cont
Continuing.
125 ns top.ram0 ram read  addr: 0x30 len: 0x1

Breakpoint 2, ram::slave_w_process (this=0x7ffffff9f650) at /home/sswan/matchlib/kit/mentor/include/ram.h:100
(gdb) p dbg_time_int()
$3 = 125
(gdb)
```

We see that the write is occurring at time 125, which is **after** time 102 when the read occurred. This is the reason for the bad data that is read. In the passing simulation, the write occurs at time 51 and the read occurs at time 61.

So we have located the root cause of the simulation failure for the stall simulation: the reason is that the two DMA operations potentially interfere with each other because they operate on overlapping regions of memory. The fix for the bug could either be made in the testbench (ie insure that the TB never issued concurrent DMA transactions that might interfere with each other) or in the DUT (add HW functionality to the DMAs to delay any new DMA operations that would operate on memory regions that already have active DMA transfers in process).

To see a simple fix, edit the testbench.cpp and find the line which reads:

```
wait(12); // allow DMA0 to do transfers before DMA1 starts
```

Change the value 12 to 1000. Then type:

```
make clean

make stall

make no_stall

make diff
```

You will see that by delaying the start of the DMA1 transfers the two simulations now match. (A more robust solution in the testbench would be to delay DMA1 until the dma0_done signal is asserted).

Leveraging Channel Logs for RTL Verification

Matchlib channel logs can also be leveraged for RTL verification. For example, channel logs generated from a SOC level SystemC simulation can be used as stimulus and checking data for RTL designs at the unit, block, or top level. In this case it doesn't matter if the particular RTL blocks being verified were generated via HLS or are handwritten RTL. A benefit of this approach is that complex scenarios which may be difficult to create within SV testbenches, but which can be fairly easily generated in the SystemC SOC model can still be applied to RTL designs.

Representing the transaction streams as text files has benefits for design verification engineers:

- The text files are versatile, are easy to use, and it is easy to identify differences when there are mismatches.
- The text files are also easy to manage within source code control systems (e.g. golden log files can be checked in and updated as needed).

However a potential downside is that the size of the text logs for large verification efforts may be an issue, and also in some cases it may be desirable to have the SystemC models which are generating the transaction data execute together with the RTL models, so that problems can be precisely debugged in both the TB and the DUT. In this case users may want to leverage a mixed language simulation (i.e. Matchlib SystemC SOC model as the TB, Verilog RTL as the DUT) in a mixed language simulator. In this scenario the transaction streams are never written out as text files, instead the transaction streams are passed "on the fly" from the SystemC TB to the RTL DUT and applied as stimulus and used as checking data. Mixed language interfaces such as the open source "UVM Connect" package can be used for such "on the fly" transaction passing between SystemC and Verilog. For more information see:

https://www.mentor.com/products/fv/multimedia/uvm-connect

Conclusion

This tutorial has demonstrated some of the most effective approaches for verifying and debugging SOC models using a simple SOC example. As we noted at the beginning of this tutorial, SOC verification and debug in SystemC Matchlib models can still be challenging. However by applying the right techniques and strategies it can be much more effective than traditional RTL or HW emulation based approaches.