

# Matchlib SOC Verification and Debugging Tutorial

Stuart Swan

Platform Architect

Siemens EDA

1 May 2024

## Introduction

Some of the most challenging verification and debugging work occurs as blocks in SOC's 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:

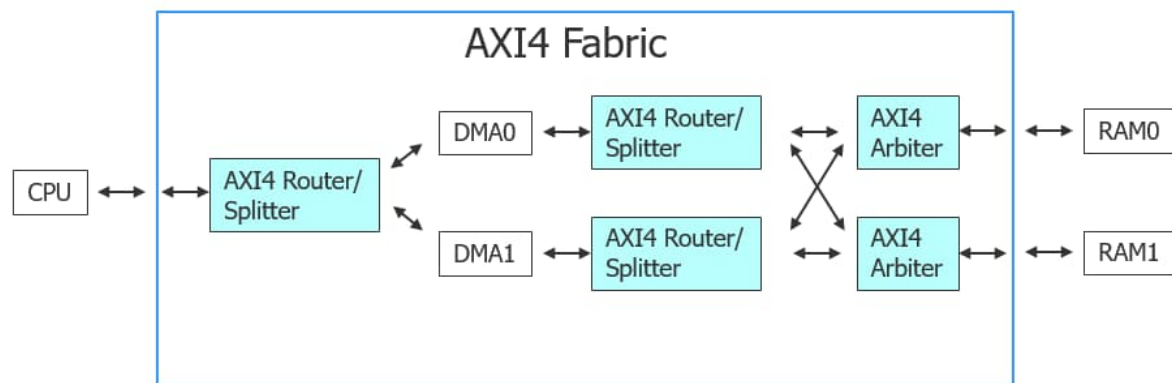
<https://webinars.sw.siemens.com/en-US/early-axi-soc-performance>

The complete source code and build scripts for this example are open source and do not require any EDA tools at all. The code is available here:

<https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG>

The example in this tutorial is in the `matchlib_examples/examples/60_rand_stall` directory.

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 bursts, 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:

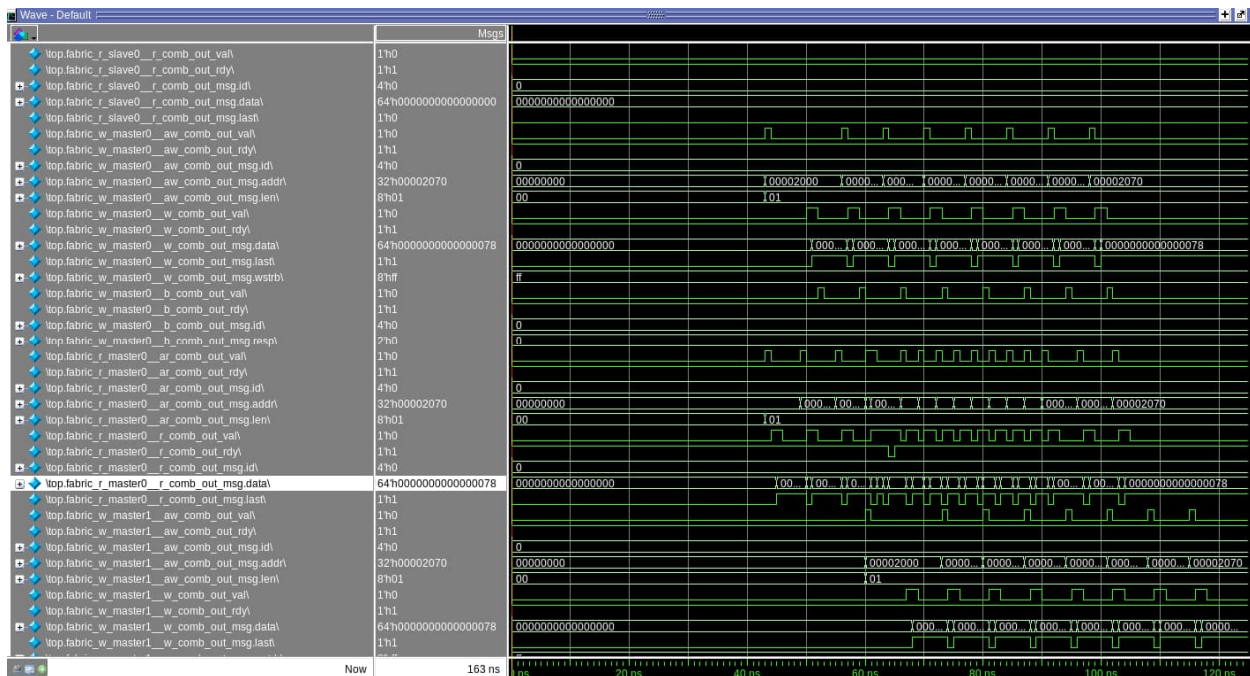
```
0 s top Stimulus started
0 s top Stimulus started
WARNING: Default time step is used for VCD tracing.
1 ns top Stimulus started
2 ns top Stimulus started
3 ns top Stimulus started
4 ns top Stimulus started
5 ns top Stimulus started
6 ns top Running FABRIC_TEST # : 0
44 ns top.ram0 ram read addr: 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:



## 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
```

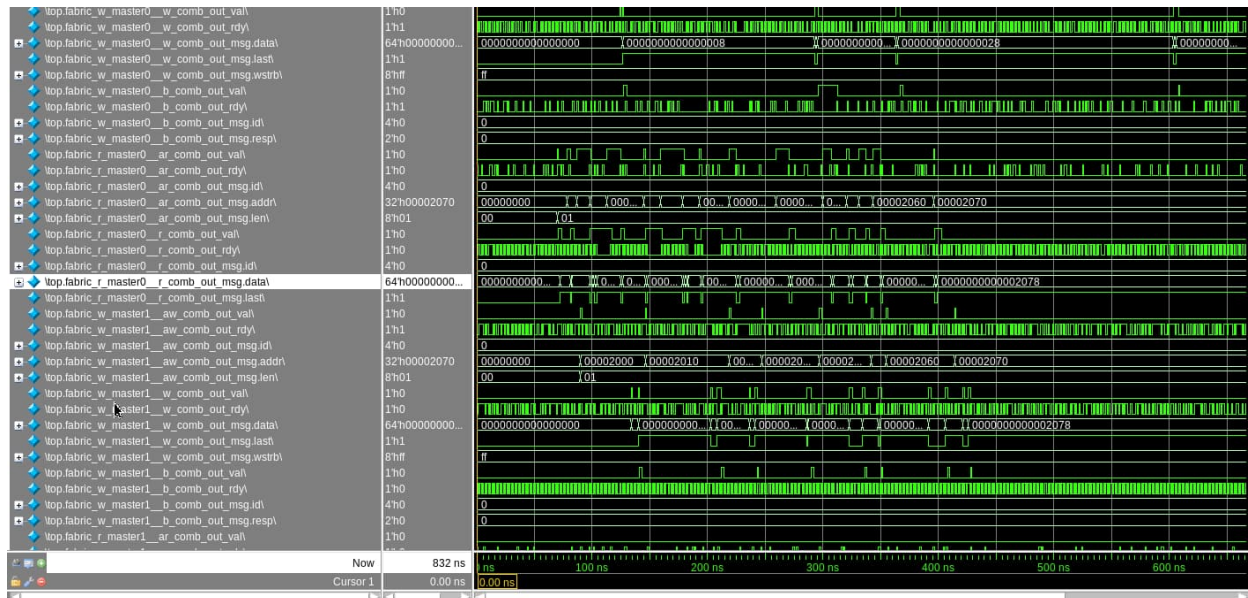
You will see that the test runs and the self-check at the end fails with an error:

```
1091 ns top elapsed time: 1010 ns
1091 ns top beat rate: 63125 ps
1091 ns top clock period: 1 ns
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 0 s:0 t: 2000
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 1 s:8 t: 2008
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 2 s:10 t: 2010
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 3 s:18 t: 2018
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 4 s:20 t: 2020
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 5 s:28 t: 2028
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 6 s:30 t: 2030
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 7 s:38 t: 2038
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 8 s:40 t: 2040
1131 ns top source and target data mismatch! DMA#: 1 Beat#: 9 s:48 t: 2048
1131 ns top source and target data mismatch! DMA#: 1 Beat#: a s:50 t: 2050
1131 ns top source and target data mismatch! DMA#: 1 Beat#: b s:58 t: 2058
1131 ns top source and target data mismatch! DMA#: 1 Beat#: c s:60 t: 2060
1131 ns top source and target data mismatch! DMA#: 1 Beat#: d s:68 t: 2068
1131 ns top source and target data mismatch! DMA#: 1 Beat#: e s:70 t: 2070
1131 ns top source and target data mismatch! DMA#: 1 Beat#: f s:78 t: 2078
1131 ns top finished checking memory contents
```

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

```
make stall_wave
```

It will display the following:



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 (1091 ns vs 123 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 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.dma1_r_master0_out__r_comb_out_dat.txt
```

This will show:



```

id{0x0} data{0x0} resp{0x0} last{0x0} ruser{}
id{0x0} data{0x08} resp{0x0} last{0x1} ruser{}
id{0x0} data{0x10} resp{0x0} last{0x0} ruser{}
id{0x0} data{0x18} resp{0x0} last{0x1} ruser{}
id{0x0} data{0x20} resp{0x0} last{0x0} ruser{}
id{0x0} data{0x28} resp{0x0} last{0x1} ruser{}
id{0x0} data{0x30} resp{0x0} last{0x0} ruser{}
id{0x0} data{0x38} resp{0x0} last{0x1} ruser{}
id{0x0} data{0x40} resp{0x0} last{0x0} ruser{}
id{0x0} data{0x48} resp{0x0} last{0x1} ruser{}
id{0x0} data{0x50} resp{0x0} last{0x0} ruser{}
id{0x0} data{0x58} resp{0x0} last{0x1} ruser{}
id{0x0} data{0x60} resp{0x0} last{0x0} ruser{}
id{0x0} data{0x68} resp{0x0} last{0x1} ruser{}
id{0x0} data{0x70} resp{0x0} last{0x0} ruser{}
id{0x0} data{0x78} resp{0x0} last{0x1} ruser{}

```

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 SOC's, there may be hundreds or even thousands of such channel logs produced for a single simulation run.

In most SOC's, 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 SOC's 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 the transactions passing through 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 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.

The Matchlib toolkit 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

55 98 top.fabric1.axi_arbiter0.comb_ba_0.txt
55 98 top.fabric_r_master0__ar_comb_out_dat.txt
56 102 top.fabric_r_master0__r_comb_out_dat.txt
62 113 top.fabric1.d1_a0_r_r_comb_out_dat.txt
63 114 top.fabric1.dma1_r_master0_out__r_comb_out_dat.txt
64 120 top.fabric1.dma1.w_segment0_w_chan_comb_BA.txt
65 121 top.fabric1.dma1_w_master0_out__w_comb_out_dat.txt
66 123 top.fabric1.d1_a1_w__w_comb_out_dat.txt
67 128 top.fabric_w_master1__w_comb_out_dat.txt

```

```

-----
Earliest differences :
< is no_stall_log/top.fabric1.axi_arbiter0.comb_ba_0.txt
> is stall_log/top.fabric1.axi_arbiter0.comb_ba_0.txt

```

```

3d2
< 0x0
10d8
< 0x1
16a15,16
> 0x1
> 0x0
-----

```

```

Finished difference summary

```

Let's look at the channel logs in more detail. Type:

```
cat no_stall_log/earliest.sorted
```

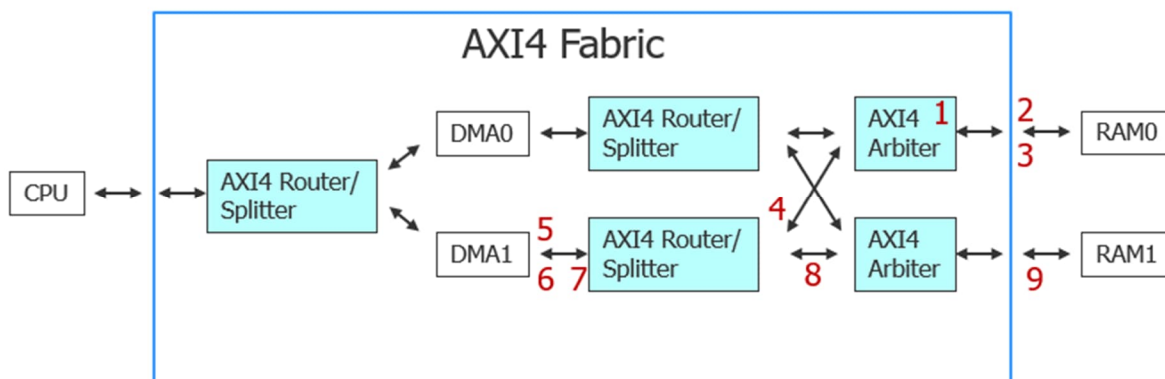
You will see:

```

[sswan@orw-sswan7-rh 60_rand_stall]$ cat no_stall_log/earliest.sorted
55 98 top.fabric1.axi_arbiter0.comb_ba_0.txt
55 98 top.fabric_r_master0__ar_comb_out_dat.txt
56 102 top.fabric_r_master0__r_comb_out_dat.txt
62 113 top.fabric1.d1_a0_r_r_comb_out_dat.txt
63 114 top.fabric1.dma1_r_master0_out__r_comb_out_dat.txt
64 120 top.fabric1.dma1.w_segment0_w_chan_comb_BA.txt
65 121 top.fabric1.dma1_w_master0_out__w_comb_out_dat.txt
66 123 top.fabric1.d1_a1_w__w_comb_out_dat.txt
67 128 top.fabric_w_master1__w_comb_out_dat.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 nine channel instances listed. Let's label them 1-9. Their names are the hierarchal paths to the channel instances, and are labeled on the block diagram here:



Make sure you are in the 60\_rand\_stall directory, and then do a side-by-side comparison of the log that has the earliest differences by typing:

```
kompare {no_stall_log,stall_log}/top.fabric1.axi_arbiter0.comb_ba_0.txt
```

This will show:

top.fabric1.axi_arbiter0.comb_ba_0.txt	top.fabric1.axi_arbiter0.comb_ba_0.txt
1 0x0	1 0x0
2 0x0	2 0x0
3 0x0	3 0x1
4 0x1	4 0x0
5 0x0	5 0x1
6 0x1	6 0x0
7 0x0	7 0x1
8 0x1	8 0x0
9 0x0	9 0x0
10 0x1	10 0x1
11 0x0	11 0x0
12 0x1	12 0x1
13 0x0	13 0x1
14 0x1	14 0x1
15 0x1	15 0x1
16 0x1	16 0x0

These are the outputs of arbiter0. Because of the timing differences between the two simulations, we should expect the outputs to differ in order, but we can also see that overall, the set of outputs is the same (i.e., ignoring the order). So, this does not look like the source of the simulation mismatch.

Let's look at the second channel in the difference list:

```
kompare {no_stall_log,stall_log}/top.fabric_r_master0__ar_comb_out_dat.txt
```

This will show:



top.fabric_r_master0__ar_comb_out_dat.txt	top.fabric_r_master0__ar_comb_out_dat.txt
1 id{0x0} addr{0x0} burst{} len{0x1} size{} cache{} auser{}	1 id{0x0} addr{0x0} burst{} len{0x1} size{} cache{} auser{}
2 id{0x0} addr{0x10} burst{} len{0x1} size{} cache{} auser{}	2 id{0x0} addr{0x10} burst{} len{0x1} size{} cache{} auser{}
3 id{0x0} addr{0x20} burst{} len{0x1} size{} cache{} auser{}	3 id{0x0} addr{0x2000} burst{} len{0x1} size{} cache{} auser{}
4 id{0x0} addr{0x2000} burst{} len{0x1} size{} cache{} auser{}	4 id{0x0} addr{0x20} burst{} len{0x1} size{} cache{} auser{}
5 id{0x0} addr{0x30} burst{} len{0x1} size{} cache{} auser{}	5 id{0x0} addr{0x2010} burst{} len{0x1} size{} cache{} auser{}
6 id{0x0} addr{0x2010} burst{} len{0x1} size{} cache{} auser{}	6 id{0x0} addr{0x30} burst{} len{0x1} size{} cache{} auser{}
7 id{0x0} addr{0x40} burst{} len{0x1} size{} cache{} auser{}	7 id{0x0} addr{0x2020} burst{} len{0x1} size{} cache{} auser{}
8 id{0x0} addr{0x2020} burst{} len{0x1} size{} cache{} auser{}	8 id{0x0} addr{0x40} burst{} len{0x1} size{} cache{} auser{}
9 id{0x0} addr{0x50} burst{} len{0x1} size{} cache{} auser{}	9 id{0x0} addr{0x50} burst{} len{0x1} size{} cache{} auser{}
10 id{0x0} addr{0x2030} burst{} len{0x1} size{} cache{} auser{}	10 id{0x0} addr{0x2030} burst{} len{0x1} size{} cache{} auser{}
11 id{0x0} addr{0x60} burst{} len{0x1} size{} cache{} auser{}	11 id{0x0} addr{0x60} burst{} len{0x1} size{} cache{} auser{}
12 id{0x0} addr{0x2040} burst{} len{0x1} size{} cache{} auser{}	12 id{0x0} addr{0x2040} burst{} len{0x1} size{} cache{} auser{}
13 id{0x0} addr{0x70} burst{} len{0x1} size{} cache{} auser{}	13 id{0x0} addr{0x2050} burst{} len{0x1} size{} cache{} auser{}
14 id{0x0} addr{0x2050} burst{} len{0x1} size{} cache{} auser{}	14 id{0x0} addr{0x2060} burst{} len{0x1} size{} cache{} auser{}
15 id{0x0} addr{0x2060} burst{} len{0x1} size{} cache{} auser{}	15 id{0x0} addr{0x2070} burst{} len{0x1} size{} cache{} auser{}
16 id{0x0} addr{0x2070} burst{} len{0x1} size{} cache{} auser{}	16 id{0x0} addr{0x70} burst{} len{0x1} size{} cache{} auser{}

These are the AXI4 “address read” (or AR) requests to RAM0. Again, because of the timing differences that result in arbitration differences in the two simulations, we should expect some ordering differences. If you look carefully, you can see that the set of AR transactions between the left side and the right side are equivalent. We can confirm this by explicitly comparing:

```
[sswan@orw-sswan7-rh 60_rand_stall]$ sort no_stall_log/top.fabric_r_master0__ar_comb_out_dat.txt > left
[sswan@orw-sswan7-rh 60_rand_stall]$ sort stall_log/top.fabric_r_master0__ar_comb_out_dat.txt > right
[sswan@orw-sswan7-rh 60_rand_stall]$ diff left right
[sswan@orw-sswan7-rh 60_rand_stall]$
```

Let’s look at the third channel in the list:

kompare {no\_stall\_log,stall\_log}/top.fabric\_r\_master0\_\_r\_comb\_out\_dat.txt

This will show:

top.fabric_r_master0__r_comb_out_dat.txt	top.fabric_r_master0__r_comb_out_dat.txt
1 id{0x0} data{0x0} resp{0x0} last{0x0} ruser{}	1 id{0x0} data{0x0} resp{0x0} last{0x0} ruser{}
2 id{0x0} data{0x08} resp{0x0} last{0x1} ruser{}	2 id{0x0} data{0x08} resp{0x0} last{0x1} ruser{}
3 id{0x0} data{0x10} resp{0x0} last{0x0} ruser{}	3 id{0x0} data{0x10} resp{0x0} last{0x0} ruser{}
4 id{0x0} data{0x18} resp{0x0} last{0x1} ruser{}	4 id{0x0} data{0x18} resp{0x0} last{0x1} ruser{}
5 id{0x0} data{0x20} resp{0x0} last{0x0} ruser{}	5 id{0x0} data{0x2000} resp{0x0} last{0x0} ruser{}
6 id{0x0} data{0x28} resp{0x0} last{0x1} ruser{}	6 id{0x0} data{0x2008} resp{0x0} last{0x1} ruser{}
7 id{0x0} data{0x0} resp{0x0} last{0x0} ruser{}	7 id{0x0} data{0x20} resp{0x0} last{0x0} ruser{}
8 id{0x0} data{0x08} resp{0x0} last{0x1} ruser{}	8 id{0x0} data{0x28} resp{0x0} last{0x1} ruser{}
9 id{0x0} data{0x30} resp{0x0} last{0x0} ruser{}	9 id{0x0} data{0x2010} resp{0x0} last{0x0} ruser{}
10 id{0x0} data{0x38} resp{0x0} last{0x1} ruser{}	10 id{0x0} data{0x2018} resp{0x0} last{0x1} ruser{}
11 id{0x0} data{0x10} resp{0x0} last{0x0} ruser{}	11 id{0x0} data{0x30} resp{0x0} last{0x0} ruser{}
12 id{0x0} data{0x18} resp{0x0} last{0x1} ruser{}	12 id{0x0} data{0x38} resp{0x0} last{0x1} ruser{}
13 id{0x0} data{0x40} resp{0x0} last{0x0} ruser{}	13 id{0x0} data{0x2020} resp{0x0} last{0x0} ruser{}
14 id{0x0} data{0x48} resp{0x0} last{0x1} ruser{}	14 id{0x0} data{0x2028} resp{0x0} last{0x1} ruser{}
15 id{0x0} data{0x20} resp{0x0} last{0x0} ruser{}	15 id{0x0} data{0x40} resp{0x0} last{0x0} ruser{}
16 id{0x0} data{0x28} resp{0x0} last{0x1} ruser{}	16 id{0x0} data{0x48} resp{0x0} last{0x1} ruser{}
17 id{0x0} data{0x50} resp{0x0} last{0x0} ruser{}	17 id{0x0} data{0x50} resp{0x0} last{0x0} ruser{}
18 id{0x0} data{0x58} resp{0x0} last{0x1} ruser{}	18 id{0x0} data{0x58} resp{0x0} last{0x1} ruser{}
19 id{0x0} data{0x30} resp{0x0} last{0x0} ruser{}	19 id{0x0} data{0x2030} resp{0x0} last{0x0} ruser{}
20 id{0x0} data{0x38} resp{0x0} last{0x1} ruser{}	20 id{0x0} data{0x2038} resp{0x0} last{0x1} ruser{}
21 id{0x0} data{0x60} resp{0x0} last{0x0} ruser{}	21 id{0x0} data{0x60} resp{0x0} last{0x0} ruser{}
22 id{0x0} data{0x60} resp{0x0} last{0x1} ruser{}	22 id{0x0} data{0x60} resp{0x0} last{0x1} ruser{}
23 id{0x0} data{0x40} resp{0x0} last{0x0} ruser{}	23 id{0x0} data{0x2040} resp{0x0} last{0x0} ruser{}
24 id{0x0} data{0x48} resp{0x0} last{0x1} ruser{}	24 id{0x0} data{0x2048} resp{0x0} last{0x1} ruser{}
25 id{0x0} data{0x70} resp{0x0} last{0x0} ruser{}	25 id{0x0} data{0x2050} resp{0x0} last{0x0} ruser{}
26 id{0x0} data{0x78} resp{0x0} last{0x1} ruser{}	26 id{0x0} data{0x2058} resp{0x0} last{0x1} ruser{}
27 id{0x0} data{0x50} resp{0x0} last{0x0} ruser{}	27 id{0x0} data{0x2060} resp{0x0} last{0x0} ruser{}
28 id{0x0} data{0x58} resp{0x0} last{0x1} ruser{}	28 id{0x0} data{0x2068} resp{0x0} last{0x1} ruser{}
29 id{0x0} data{0x60} resp{0x0} last{0x0} ruser{}	29 id{0x0} data{0x2070} resp{0x0} last{0x0} ruser{}
30 id{0x0} data{0x68} resp{0x0} last{0x1} ruser{}	30 id{0x0} data{0x2078} resp{0x0} last{0x1} ruser{}
31 id{0x0} data{0x70} resp{0x0} last{0x0} ruser{}	31 id{0x0} data{0x70} resp{0x0} last{0x0} ruser{}
32 id{0x0} data{0x78} resp{0x0} last{0x1} ruser{}	32 id{0x0} data{0x78} resp{0x0} last{0x1} ruser{}

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 fourth channel log in the list:

kompare {no\_stall\_log,stall\_log}/top.fabric1.d1\_a0\_r\_\_r\_comb\_out\_dat.txt

This will show:

top.fabric1.d1_a0_r__r_comb_out_dat.txt	top.fabric1.d1_a0_r__r_comb_out_dat.txt
1 id{0x0} data{0x0} resp{0x0} last{0x0} ruser{}	1 id{0x0} data{0x2000} resp{0x0} last{0x0} ruser{}
2 id{0x0} data{0x08} resp{0x0} last{0x1} ruser{}	2 id{0x0} data{0x2008} resp{0x0} last{0x1} ruser{}
3 id{0x0} data{0x10} resp{0x0} last{0x0} ruser{}	3 id{0x0} data{0x2010} resp{0x0} last{0x0} ruser{}
4 id{0x0} data{0x18} resp{0x0} last{0x1} ruser{}	4 id{0x0} data{0x2018} resp{0x0} last{0x1} ruser{}
5 id{0x0} data{0x20} resp{0x0} last{0x0} ruser{}	5 id{0x0} data{0x2020} resp{0x0} last{0x0} ruser{}
6 id{0x0} data{0x28} resp{0x0} last{0x1} ruser{}	6 id{0x0} data{0x2028} resp{0x0} last{0x1} ruser{}
7 id{0x0} data{0x30} resp{0x0} last{0x0} ruser{}	7 id{0x0} data{0x2030} resp{0x0} last{0x0} ruser{}
8 id{0x0} data{0x38} resp{0x0} last{0x1} ruser{}	8 id{0x0} data{0x2038} resp{0x0} last{0x1} ruser{}
9 id{0x0} data{0x40} resp{0x0} last{0x0} ruser{}	9 id{0x0} data{0x2040} resp{0x0} last{0x0} ruser{}
10 id{0x0} data{0x48} resp{0x0} last{0x1} ruser{}	10 id{0x0} data{0x2048} resp{0x0} last{0x1} ruser{}
11 id{0x0} data{0x50} resp{0x0} last{0x0} ruser{}	11 id{0x0} data{0x2050} resp{0x0} last{0x0} ruser{}
12 id{0x0} data{0x58} resp{0x0} last{0x1} ruser{}	12 id{0x0} data{0x2058} resp{0x0} last{0x1} ruser{}
13 id{0x0} data{0x60} resp{0x0} last{0x0} ruser{}	13 id{0x0} data{0x2060} resp{0x0} last{0x0} ruser{}
14 id{0x0} data{0x68} resp{0x0} last{0x1} ruser{}	14 id{0x0} data{0x2068} resp{0x0} last{0x1} ruser{}
15 id{0x0} data{0x70} resp{0x0} last{0x0} ruser{}	15 id{0x0} data{0x2070} resp{0x0} last{0x0} ruser{}
16 id{0x0} data{0x78} resp{0x0} last{0x1} ruser{}	16 id{0x0} data{0x2078} resp{0x0} last{0x1} ruser{}

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 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 5, 6, 7, 8, and 9 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 9, which got it from channel 8, then from 7, 6, 5, 4 and all the way back to channel 3 starting with bad read data at simulation time 56 as reported by the earliest\_diff.sh script. To be precise, the earliest\_diff.sh script reports that first significant difference is at time 56 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.

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 for channels 1 and 2. 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 102 (in the stall simulation) 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 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. One technique we use simplify usage of the debugger is to only set breakpoints on file line numbers, for example:

```
br ram.h:95
```

If you are executing these steps in your environment, you may need to adjust the line numbers for your version of the source files, since code locations may have slight changes. To do this adjustment, find corresponding source code line in your version of the files compared to what is shown in this document.

### Debugging the “No Stall” Simulation

Let’s debug the no stall simulation first:

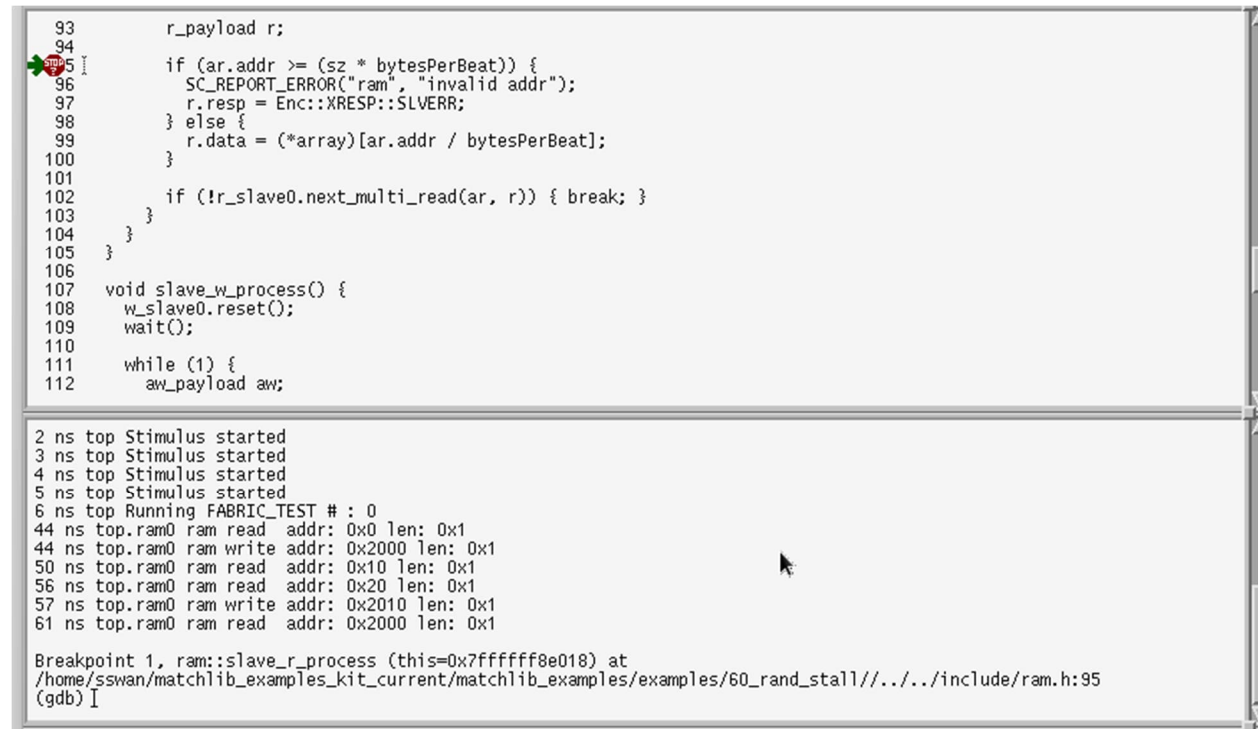
```
ddd sim_no_stall &
```

Once it starts, let’s set a breakpoint for reads in RAM0 at time at address 0x2000, since we know from the investigation of the channel logs that the bad data in the stall scenario occurs at that address:

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

```
run
```

This will then show:



The screenshot shows a debugger window with two panes. The top pane displays C++ source code from a file named `ram.h`. The code includes a function `r_payload` and a `while` loop. A red arrow points to line 95, which is the location of the breakpoint. The bottom pane shows a console log with simulation output. The log includes several lines of memory access (reads and writes) and a final line indicating that a breakpoint has been hit at line 95 of `ram.h`.

```
93     r_payload r;  
94  
95     if (ar.addr >= (sz * bytesPerBeat)) {  
96         SC_REPORT_ERROR("ram", "invalid addr");  
97         r.resp = Enc::XRESP::SLVERR;  
98     } else {  
99         r.data = (*array)[ar.addr / bytesPerBeat];  
100     }  
101  
102     if (!r_slave0.next_multi_read(ar, r)) { break; }  
103  
104 }  
105 }  
106  
107 void slave_w_process() {  
108     w_slave0.reset();  
109     wait();  
110  
111     while (1) {  
112         aw_payload aw;
```

```
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=0x7ffffff8e018) at  
/home/sswan/matchlib_examples_kit_current/matchlib_examples/examples/60_rand_stall/../../include/ram.h:95  
(gdb) I
```

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)
```

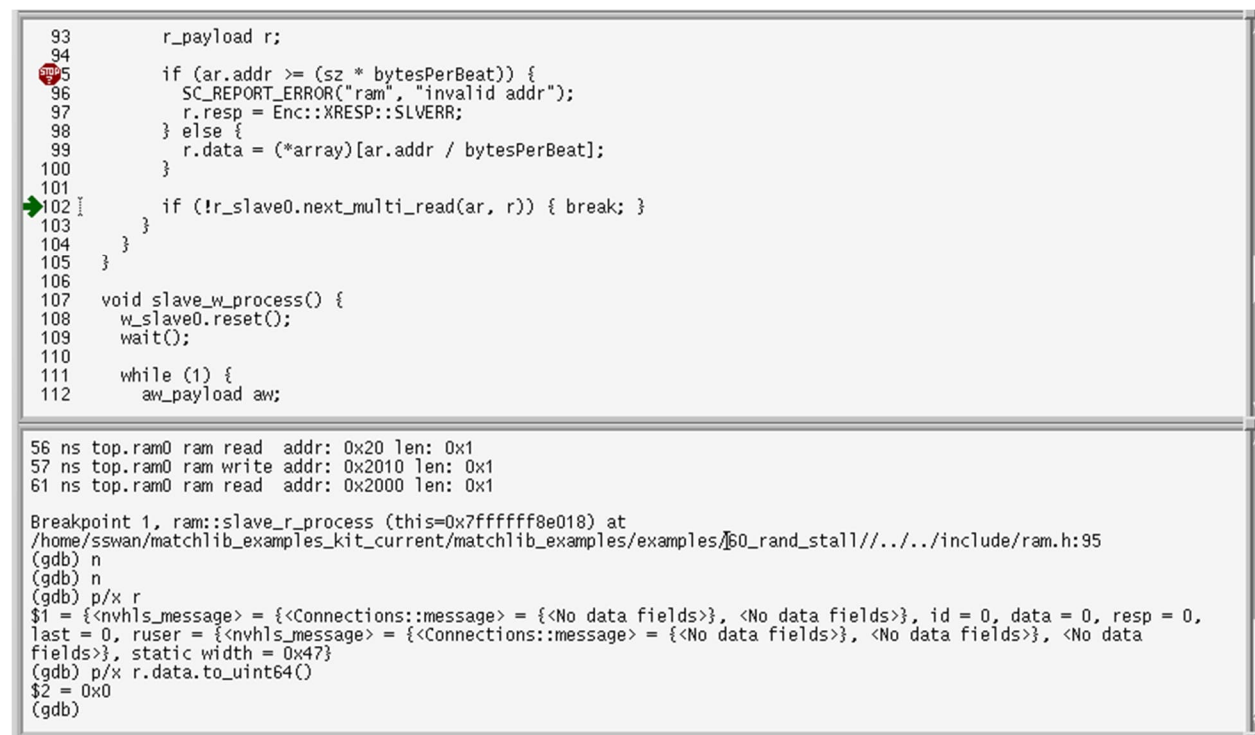
You can check the simulation time by typing:

```
p ac_dbg_time_ns()
```

Everything looks correct. Now let’s check the data that will be read. Use the “n” command to step forward to line 102, then use the print command to print the read data value that will be returned:

```
p/x r.data.to_uint64()
```

You will see:



The screenshot shows a debugger window with two panes. The top pane displays C code from a file named `ram.h`, with line numbers 93 to 112. Line 102 is highlighted with a green arrow, indicating the current execution point. The code defines a function `r_payload` and a `while` loop that calls `next_multi_read`. The bottom pane shows GDB output, including memory reads from `top.ram0` at addresses `0x20`, `0x2010`, and `0x2000`. It also shows a breakpoint hit at line 95 of `ram.h`. The GDB prompt shows the command `p/x r` being entered, and the output shows the value of `r` as `{<nvhl_message> = {<Connections::message> = {<No data fields>}, <No data fields>}, id = 0, data = 0, resp = 0, last = 0, ruser = {<nvhl_message> = {<Connections::message> = {<No data fields>}, <No data fields>}, <No data fields>}, static width = 0x47}}`. The command `p/x r.data.to_uint64()` is also shown, with the output `$2 = 0x0`.

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:122 if aw.addr.to_uint64() == 0x2000
```

```
run
```



```
120     w_payload w = w_slave0.w.Pop();
121
122     if (aw.addr >= (sz * bytesPerBeat)) {
123         SC_REPORT_ERROR("ram", "invalid addr");
124         b.resp = Enc::XRESP::SLVERR;
125     } else {
126         decltype(w.wstrb) all_on{~0};
127
128         if (w.wstrb == all_on) {
129             (*array)[aw.addr / bytesPerBeat] = w.data.to_uint64();
130         } else {
131             CCS_LOG("write strobe enabled");
132             arr_t orig = (*array)[aw.addr / bytesPerBeat];
133             arr_t wdata = w.data.to_uint64();
134
135             #pragma hls_unroll
136             for (int i=0; i<WSTRB_WIDTH; i++)
137                 if (w.wstrb[i]) {
138                     orig = nvhls::set_slc(orig, nvhls::get_slc<8>(wdata, (i*8)), (i*8));
139                 }
140         }
141     }
142 }

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=0x7ffffff8e018) at
/home/sswan/matchlib_examples_kit_current/matchlib_examples/examples/60_rand_stall/../../include/ram.h:122
(gdb) p ac_dbg_time_ns()
$1 = 51
(gdb) p ac_dbg_name(this)
top.ram0
$2 = void
(gdb) I

A Top ram0
```

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

```
p ac_dbg_time_ns()
```

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)
```

Everything looks correct. Let's quit the debugger:

```
quit
```

## Debugging the "Stall" Simulation

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 address 0x2000. Let's set a breakpoint for that read:

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

Then let's step forward to line 102 as we did above to see what the r.data value is being returned:



```

93     r_payload r;
94
95     if (ar.addr >= (sz * bytesPerBeat)) {
96         SC_REPORT_ERROR("ram", "invalid addr");
97         r.resp = Enc::XRESP::SLVERR;
98     } else {
99         r.data = (*array)[ar.addr / bytesPerBeat];
100     }
101
102     if (!r_slave0.next_multi_read(ar, r)) { break; }
103 }
104 }
105 }
106
107 void slave_w_process() {
108     w_slave0.reset();
109     wait();
110
111     while (1) {
112         aw_payload aw;

```

---

```

98 ns top.ram0 ram read  addr: 0x10 len: 0x1
100 ns top.ram0 ram read  addr: 0x2000 len: 0x1

Breakpoint 1, ram::slave_r_process (this=0x7ffff8e018) at
/home/sswan/matchlib_examples_kit_current/matchlib_examples/examples/60_rand_stall/../../include/ram.h:95
(gdb) n
(gdb)
(gdb) p/x r.data.to_uint64()
$1 = 0x2000
(gdb) p ac_dbg_time_ns()
$2 = 100
(gdb) p ac_dbg_name(this)
top.ram0
$3 = void
(gdb) l

```

Top.ram0

We can see that the bad data value 0x2000 is being returned, in contrast to the correct data value 0x0 we saw in the "no\_stall" simulation. Note that the simulation time is 100 ns.

Quit DDD and restart it again on the "stall" simulation:

```
ddd sim_stall &
```

Now let's see when that address in RAM0 is being written. 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:122 if (aw.addr.to_uint64() == 0x2000) && ac_dbg_name_is(this, "top.ram0")
```

Now let's run the simulation:

```
run
```

We will see:

```

120     w_payload w = w_slave0.w.Pop();
121
122     if (aw.addr >= (sz * bytesPerBeat)) {
123         SC_REPORT_ERROR("ram", "invalid addr");
124         b.resp = Enc::XRESP::SLVERR;
125     } else {
126         decltype(w.wstrb) all_on{~0};
127
128         if (w.wstrb == all_on) {
129             (*array)[aw.addr / bytesPerBeat] = w.data.to_uint64();
130         } else {
131             CCS_LOG("write strobe enabled");
132             arr_t orig = (*array)[aw.addr / bytesPerBeat];
133             arr_t wdata = w.data.to_uint64();
134
135             #pragma hls_unroll
136             for (int i=0; i<WSTRB_WIDTH; i++)
137                 if (w.wstrb[i]) {
138                     orig = nvhls::set_slc(orig, nvhls::get_slc<8>(wdata, (i*8)), (i*8));
139                 }
140
141         }
142     }
143 }
144
145 //-----
146
147 100 ns top.ram0 ram read   addr: 0x2000 len: 0x1
148 125 ns top.ram0 ram read   addr: 0x20 len: 0x1
149 127 ns top.ram0 ram read   addr: 0x2010 len: 0x1
150 136 ns top.ram1 ram write  addr: 0x2010 len: 0x1
151 146 ns top.ram0 ram read   addr: 0x30 len: 0x1
152 174 ns top.ram1 ram write  addr: 0x2020 len: 0x1
153 178 ns top.ram0 ram read   addr: 0x2020 len: 0x1
154 180 ns top.ram0 ram read   addr: 0x40 len: 0x1
155 194 ns top.ram0 ram read   addr: 0x50 len: 0x1
156
157 Breakpoint 1, ram::slave_w_process (this=0x7ffff8e018) at
158 /home/sswan/matchlib_examples_kit_current/matchlib_examples/examples/60_rand_stall/../../include/ram.h:122
159 (gdb) p ac_dbg_time_ns()
160 $1 = 215
161 (gdb)

```

▲ \$1 = 215

This is bad. We are seeing that the write occurs at time 215, which is after the read occurred at time 100. This is the source of the bad read data in the "stall" simulation. The write was supposed to occur first (as in the no\_stall simulation), but it did not due to the delays introduced into the bus fabric via the random stall injection.

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 (i.e. ensure 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 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 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 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://verificationacademy.com/topics/uvm-universal-verification-methodology/uvmc/>

### A Note about Memory Logs

Matchlib has a memory logging feature which can also be used to effectively debug the example presented in this paper. To keep the scope of this paper more contained, we did not discuss this feature here. Memory logs are discussed in detail in the document titled "memory\_logging\_and\_debug.pdf" within the Catapult Matchlib Examples toolkit documentation directory.

### 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.