Catapult Matchlib Memory Logs and Debug Methodology

Stuart Swan

Platform Architect

Siemens EDA

31 January 2023

Introduction

This document describes the Matchlib memory logging feature and related memory debug methodology. Memories are commonly used in HLS designs, and design and verification engineers often need visibility into what is occurring in the memories in order to be able to debug issues both in pre-HLS scenarios, as well as debugging memory-related problems that may occur when comparing pre-HLS versus post-HLS results.

The memory logging and debug methodology described here is complimentary to the Matchlib channel logs and debug methodology. The channel logs are described in detail in the document titled matchlib_soc_debug_tutorial.pdf which is in the Catapult Matchlib examples toolkit in the "doc" directory. It is recommended that you read that document in addition to this document to understand the overall logging and debug methodology.

The memory logs and debug methodology described in this document are not tightly coupled to either Matchlib or SystemC designs, and the same approach can be used in Catapult C++ models.

Background

Memories appear as normal C/C++ arrays (or similar classes such as std::array<>) within HLS models, and are mapped to RAMs during HLS. Pointers may appear in HLS models, and pointer accesses are resolved to array accesses during HLS. Array accesses are coded in models using the indexing operator, for example:

```
  uint16 my_mem[128];
  my_mem[x] = y;
  x = my_mem[y];
```

There are several aspects of arrays which complicate debugging when using HLS:

1. HLS often adds latency when it synthesizes the design, so an array access which occurs at a particular simulation time in the pre-HLS model most likely will not occur at the same time in the post-HLS model.
2. Array accesses may be re-ordered in the pre-HLS simulation, and in some cases that is OK, and in other cases it can cause a bug. For example, when a single pre-HLS model operates with and without random stall injection, two processes which are writing to a shared RAM may cause the writes to the RAM to be seen in different orders in the two scenarios. If the two processes always write to independent addresses, there never is a bug. However, if the two processes

write to the same addresses, then a "WRITE/WRITE" conflict is possible and different overall simulation results may be seen in the two scenarios.

3. Array accesses may be re-ordered by HLS. For example, HLS may reorder READ operations (which is typically safe to do), and may reorder WRITE operations in certain situations. Certain HLS directives such as "ignore_memory_precedences" explicitly allow HLS to do reordering of accesses.

The memory logs and debug methodology described in this document are designed to address the above issues.

Memory Logs

The memory log feature is provided by a few classes that add the logging functionality. The most commonly used class is extended_array:

```
#include <array>

template <typename T, unsigned N>
class extended_array : public std::array<T,N> { .. }
```

This class extends std::array<>, which itself is a simple wrapper around a normal C array, i.e.

```
T my_array[N];
```

The extended_array template class overloads the indexing operators "[]" to gain visibility into when array reads and writes are performed in the pre-HLS model. It then generates log files to record the array accesses. For each array instance, a separate log file is generated for all the read operations, and another log is generated for the write operations. All log files are in ASCII, and all numbers are represented in hex format, and each read/write operation takes exactly one line. The format of the log files is:

```
Address  write_count  value
```

For example, in a "read" log file, the following entry:

```
0000040e 2 0000000000000070
```

indicates that address 0x40e was read and the read value was 0x70, and that read value represents what was written on the second write operation to that location in the RAM.

Note that both "read" logs and "write" logs have write counts in them, there is no corresponding "read count".

As we will see, the "write count" is a key tool to enable effective debugging strategies even in the presence of the complicating factors mentioned above.

The constructor for extended_array is:

```
extended_array(std::string _nm = "", bool _add_time_stamp=0) { ...}
```

If the first argument is omitted or an empty string, then no log will be produced, else the argument is used to name the read and write log files for that array instance. If the second argument is true, then the simulation time will be appended to each line in the log files. This can be helpful in certain debugging situations, however by default this option is not enabled because it causes "diff" and related comparison utilities to flag harmless differences.

The extended_array class is not the only way to generate memory logs. In example 80* in the Matchlib toolkit, as we will see below, we can also generate these logs with the same format from RTL models. It is also possible to add this logging format to customer-specific memory models. Once that is done, the methodology described in this document can be easily used.

Example #1

Let's first see how memory logs are used in a simple example, which is example 12* in the Matchlib examples toolkit. This example is a simple "ping-pong" memory that is written by a producer process and read by a consumer process. The two processes share a memory that has 16 entries. The producer writes to the first set of eight entries in ascending address order, then hands ownership of that half over to the consumer process, which reads the entries in ascending address order. Once the producer has handed ownership of a half of memory over, it proceeds to write to the other half of the memory. The ping-ping handoff sequence then repeats.

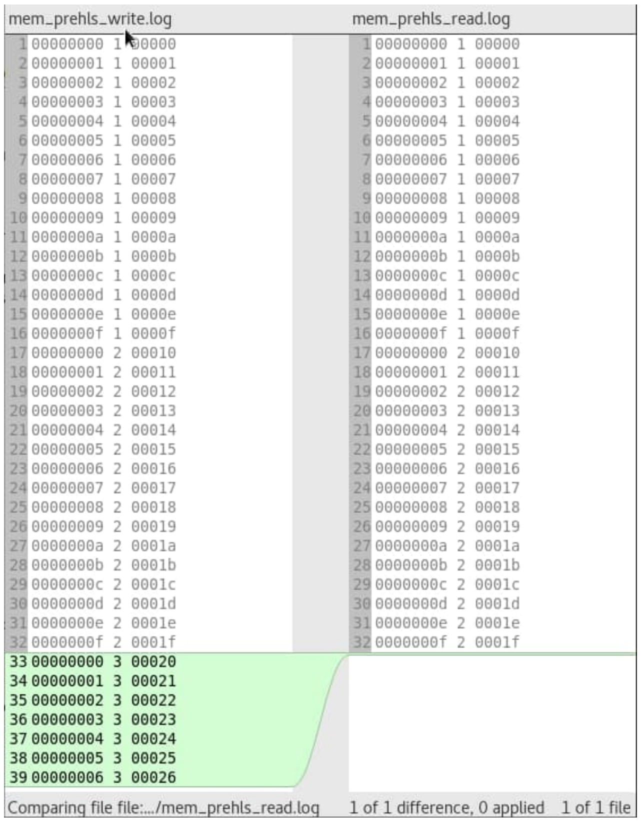
You can build and run the example by doing:

```
cd 12_ping_pong_mem
make build
./sim_sc
```

This will produce the files:

```
mem_prehls_write.log
mem_prehls_read.log
```

In this case the "write.log" is what the producer process has written, and the "read.log" is what the consumer process has read. If we use the "kompare" utility to compare the logs, we will see:



```
mem_prehls_write.log          mem_prehls_read.log
00000000 1 00000              00000000 1 00000
00000001 1 00001              00000001 1 00001
00000002 1 00002              00000002 1 00002
00000003 1 00003              00000003 1 00003
00000004 1 00004              00000004 1 00004
00000005 1 00005              00000005 1 00005
00000006 1 00006              00000006 1 00006
00000007 1 00007              00000007 1 00007
00000008 1 00008              00000008 1 00008
00000009 1 00009              00000009 1 00009
0000000a 1 0000a              0000000a 1 0000a
0000000b 1 0000b              0000000b 1 0000b
0000000c 1 0000c              0000000c 1 0000c
0000000d 1 0000d              0000000d 1 0000d
0000000e 1 0000e              0000000e 1 0000e
0000000f 1 0000f              0000000f 1 0000f
00000000 2 00010              00000000 2 00010
00000001 2 00011              00000001 2 00011
00000002 2 00012              00000002 2 00012
00000003 2 00013              00000003 2 00013
00000004 2 00014              00000004 2 00014
00000005 2 00015              00000005 2 00015
00000006 2 00016              00000006 2 00016
00000007 2 00017              00000007 2 00017
00000008 2 00018              00000008 2 00018
00000009 2 00019              00000009 2 00019
0000000a 2 0001a              0000000a 2 0001a
0000000b 2 0001b              0000000b 2 0001b
0000000c 2 0001c              0000000c 2 0001c
0000000d 2 0001d              0000000d 2 0001d
0000000e 2 0001e              0000000e 2 0001e
0000000f 2 0001f              0000000f 2 0001f
00000000 3 00020
00000001 3 00021
00000002 3 00022
00000003 3 00023
00000004 3 00024
00000005 3 00025
00000006 3 00026
Comparing file file:.../mem_prehls_read.log    1 of 1 difference, 0 applied    1 of 1 file
```

First, we can note the extra entries at the bottom of the write log. These represent write operations to the memory that were never read by the consumer process. The reason the consumer never read them is because the testbench ended the test before the consumer process had sufficient time to read these items. In this case this result is intended by the author of the testbench.

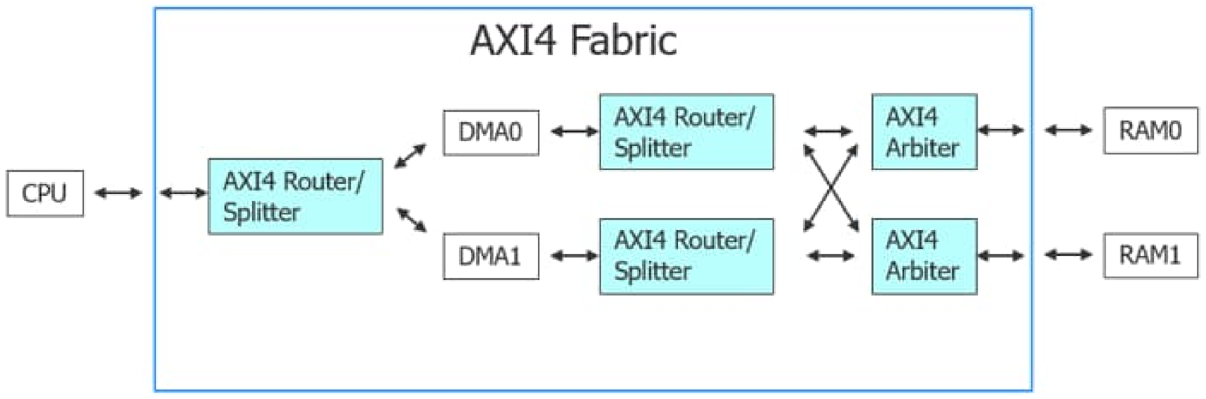
The second item to notice is the write_count fields. Note that when the producer writes to the same addresses for the second and third times, the write_count field clearly indicates that this occurred. Note that the identical write_count values appear in the read log.

Lastly, you should note that in this simple test case the order of write addresses and read addresses are identical. That may not always be the case for other designs, and if the order is different it may not necessarily represent a bug, as we will see.

Example #2

The next example using memory logs is example 60* in the Catapult Matchlib toolkit. This example is extensively described in the document titled matchlib_soc_debug_tutorial.pdf which is in the Catapult Matchlib examples toolkit in the "doc" directory. We will be looking at the same tests described in that document, so we will only briefly describe the test here.

The design is an AXI4 fabric consisting of two DMAs and two RAMs. The memory logs are generated from each of the RAMs:



I/**

For this test scenario the DUT performs 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 the design and run the "no_stall" and "stall" simulations with the following commands:

```
cd 60_rand_stall
make all
```

This will produce the following memory log files:

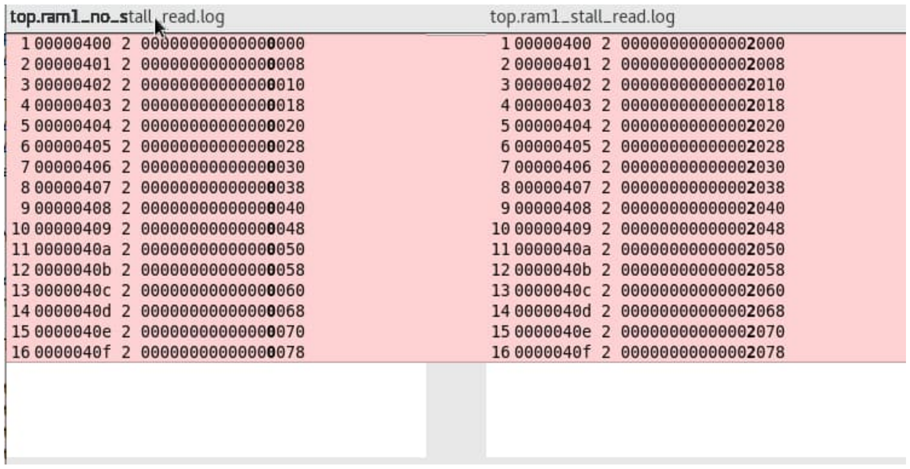
```
top.ram0_no_stall_read.log
top.ram0_no_stall_write.log
top.ram0_stall_read.log
top.ram0_stall_write.log
top.ram1_no_stall_read.log
top.ram1_no_stall_write.log
top.ram1_stall_read.log
top.ram1_stall_write.log
```

The "no_stall" files represent the desired system behavior, and the "stall" files represent the incorrect system behavior that is described in detail in the matchlib_soc_debug_tutorial.pdf document. Briefly, the incorrect behavior arises in the "stall" scenario because the second copy operation starts too early, so it reads incorrect data from RAM0.

All the addresses in the log files are indexes into the RAM arrays, which contain 8 bytes per element. Since AXI4 addresses are always byte addresses, we need to keep in mind the mapping between AXI4 byte addresses and the array indexes. For example, the byte address 0x2000 needs to be divided by 8 to arrive at the array index address of 0x400.

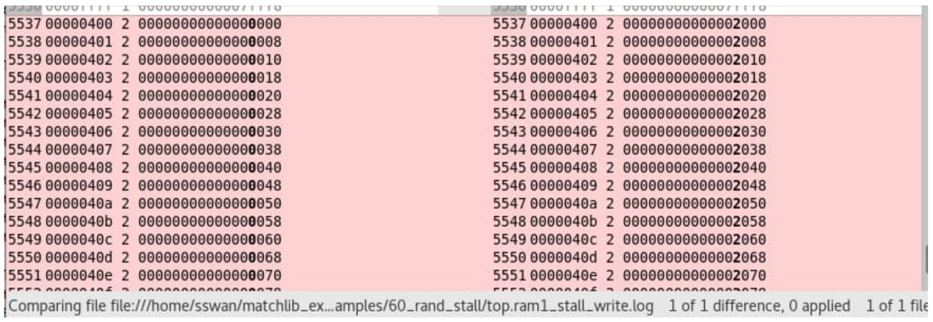
The result of the incorrect behavior described in the matchlib_soc_debug_tutorial.pdf document is easily seen when comparing the read and write operations on RAM1:

```
    kompare *ram1*read*
```



```
top.ram1_no_stall_read.log                     top.ram1_stall_read.log
 1 00000400  2  0000000000000000                1 00000400  2  0000000000002000
 2 00000401  2  0000000000000008                2 00000401  2  0000000000002008
 3 00000402  2  0000000000000010                3 00000402  2  0000000000002010
 4 00000403  2  0000000000000018                4 00000403  2  0000000000002018
 5 00000404  2  0000000000000020                5 00000404  2  0000000000002020
 6 00000405  2  0000000000000028                6 00000405  2  0000000000002028
 7 00000406  2  0000000000000030                7 00000406  2  0000000000002030
 8 00000407  2  0000000000000038                8 00000407  2  0000000000002038
 9 00000408  2  0000000000000040                9 00000408  2  0000000000002040
10 00000409  2  0000000000000048               10 00000409  2  0000000000002048
11 0000040a  2  0000000000000050               11 0000040a  2  0000000000002050
12 0000040b  2  0000000000000058               12 0000040b  2  0000000000002058
13 0000040c  2  0000000000000060               13 0000040c  2  0000000000002060
14 0000040d  2  0000000000000068               14 0000040d  2  0000000000002068
15 0000040e  2  0000000000000070               15 0000040e  2  0000000000002070
16 0000040f  2  0000000000000078               16 0000040f  2  0000000000002078
```

```
    kompare *ram1*write*
```



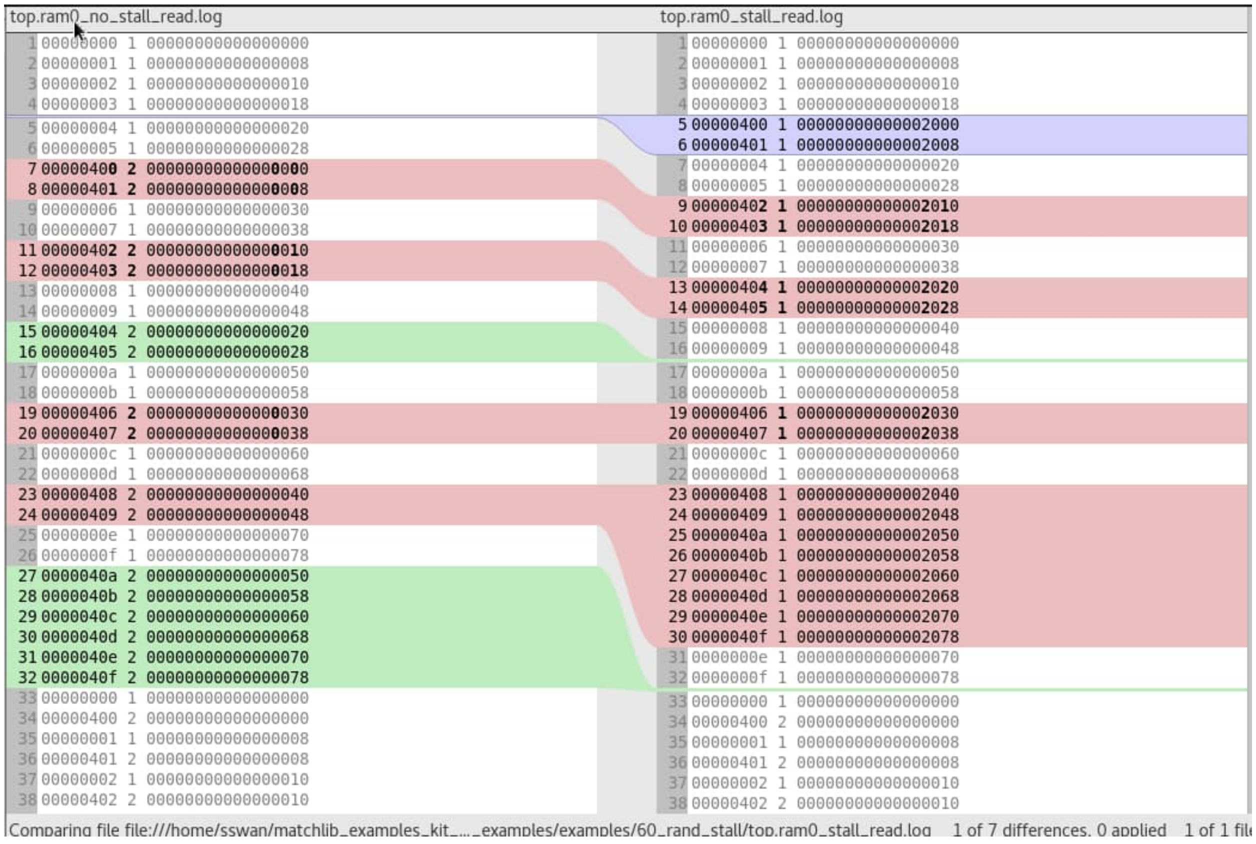
```
5537 00000400  2  0000000000000000              5537 00000400  2  0000000000002000
5538 00000401  2  0000000000000008              5538 00000401  2  0000000000002008
5539 00000402  2  0000000000000010              5539 00000402  2  0000000000002010
5540 00000403  2  0000000000000018              5540 00000403  2  0000000000002018
5541 00000404  2  0000000000000020              5541 00000404  2  0000000000002020
5542 00000405  2  0000000000000028              5542 00000405  2  0000000000002028
5543 00000406  2  0000000000000030              5543 00000406  2  0000000000002030
5544 00000407  2  0000000000000038              5544 00000407  2  0000000000002038
5545 00000408  2  0000000000000040              5545 00000408  2  0000000000002040
5546 00000409  2  0000000000000048              5546 00000409  2  0000000000002048
5547 0000040a  2  0000000000000050              5547 0000040a  2  0000000000002050
5548 0000040b  2  0000000000000058              5548 0000040b  2  0000000000002058
5549 0000040c  2  0000000000000060              5549 0000040c  2  0000000000002060
5550 0000040d  2  0000000000000068              5550 0000040d  2  0000000000002068
5551 0000040e  2  0000000000000070              5551 0000040e  2  0000000000002070
Comparing file file:///home/sswan/matchlib_ex...amples/60_rand_stall/top.ram1_stall_write.log   1 of 1 difference, 0 applied   1 of 1 file
```

When we do "`diff *ram0*write*`" we see that there are no differences at all in the two scenarios in terms of what was written to RAM0.

However, when we do:

```
    kompare *ram0*read*
```

We see:



top.ram0_no_stall_read.log

```
00000000 1 0000000000000000
00000001 1 0000000000000008
00000002 1 0000000000000010
00000003 1 0000000000000018
00000004 1 0000000000000020
00000005 1 0000000000000028
00000400 2 0000000000000000
00000401 2 0000000000000008
00000006 1 0000000000000030
00000007 1 0000000000000038
00000402 2 0000000000000010
00000403 2 0000000000000018
00000008 1 0000000000000040
00000009 1 0000000000000048
00000404 2 0000000000000020
00000405 2 0000000000000028
0000000a 1 0000000000000050
0000000b 1 0000000000000058
00000406 2 0000000000000030
00000407 2 0000000000000038
0000000c 1 0000000000000060
0000000d 1 0000000000000068
00000408 2 0000000000000040
00000409 2 0000000000000048
0000000e 1 0000000000000070
0000000f 1 0000000000000078
0000040a 2 0000000000000050
0000040b 2 0000000000000058
0000040c 2 0000000000000060
0000040d 2 0000000000000068
0000040e 2 0000000000000070
0000040f 2 0000000000000078
00000000 1 0000000000000000
00000400 2 0000000000000000
00000001 1 0000000000000008
00000401 2 0000000000000008
00000002 1 0000000000000010
00000402 2 0000000000000010
```

top.ram0_stall_read.log

```
00000000 1 0000000000000000
00000001 1 0000000000000008
00000002 1 0000000000000010
00000003 1 0000000000000018
00000400 1 0000000000002000
00000401 1 0000000000002008
00000004 1 0000000000000020
00000005 1 0000000000000028
00000402 1 0000000000002010
00000403 1 0000000000002018
00000006 1 0000000000000030
00000007 1 0000000000000038
00000404 1 0000000000002020
00000405 1 0000000000002028
00000008 1 0000000000000040
00000009 1 0000000000000048
0000000a 1 0000000000000050
0000000b 1 0000000000000058
00000406 1 0000000000002030
00000407 1 0000000000002038
0000000c 1 0000000000000060
0000000d 1 0000000000000068
00000408 1 0000000000002040
00000409 1 0000000000002048
0000040a 1 0000000000002050
0000040b 1 0000000000002058
0000040c 1 0000000000002060
0000040d 1 0000000000002068
0000040e 1 0000000000002070
0000040f 1 0000000000002078
0000000e 1 0000000000000070
0000000f 1 0000000000000078
00000000 1 0000000000000000
00000400 2 0000000000000000
00000001 1 0000000000000008
00000401 2 0000000000000008
00000002 1 0000000000000010
00000402 2 0000000000000010
```

Comparing file file:///home/sswan/matchlib_examples_kit_..._examples/examples/60_rand_stall/top.ram0_stall_read.log 1 of 7 differences, 0 applied 1 of 1 fil

The left side is the correct behavior, and the right side is the "stall" incorrect behavior.

All the reads starting at indexes 0x400 and higher represent the second copy operation in the test scenario. We see on the left that those read operations all read data with "write_counts" that are consistently 2. However, on the right side, we see the incorrect behavior where those reads are occurring where some write_counts are still 1. This is the root cause of the bug - it is a clear indication that the reads from the second copy operation are occurring too soon in the stall scenario.

Example #3

The third example is example 80* in the Matchlib toolkit. In this example we generate memory logs for the pre-HLS and post-HLS model and compare the results. This is the code in the DUT:

```
        typedef ac_int<32, false> elem_type;
        static const int mem_size = 128;

       #ifdef USE_EXT_ARRAY
         extended_array<elem_type, mem_size> mem{"mem_prehls"};
       #else
         elem_type mem[mem_size];
       #endif

         Connections::In <elem_type> CCS_INIT_S1(raddr_in);
         Connections::In <elem_type> CCS_INIT_S1(waddr_in);
         Connections::In <elem_type> CCS_INIT_S1(data_in);
         Connections::Out<elem_type> CCS_INIT_S1(data_out);
```

```
    void main() {
      // ...
      wait();  // Reset state

  #pragma hls_pipeline_init_interval 1
  #pragma pipeline_stall_mode flush
      while (1) {
        mem[waddr_in.Pop()] = data_in.Pop();
        elem_type t = mem[raddr_in.Pop()];
        data_out.Push(t);
      }
    }
```

In this design we are mapping "mem" to a RAM with 1 read and 1 write port, and our goal is to pipeline with an II=1. In the pre-HLS model source code, the mem write operation occurs strictly before the memory read operation, and the required behavior is clear even if waddr and raddr are the same. In this specific case the required behavior would be that whatever is read from data_in would be written to data_out. Catapult by default will want to preserve this strict ordering, and this will prevent it from pipelining this design with an II=1 unless we explicitly tell it to ignore the scheduling precedence for the memory operations. We do this by using the following Catapult directive:

```
ignore_memory_precedences -from *write_mem* -to *read_mem*
```

This then allows Catapult to schedule this design with an II=1, but the user is effectively pledging to never allow waddr and raddr to be the same in the same iteration, or else the pre-HLS behavior and the post-HLS behavior will mismatch (since the ordering relationship in the pre-HLS model is no longer preserved in the post-HLS model).

You can run the pre-HLS simulation by doing:

```
cd 80_mem_log_pre_post_hls
make build
./sim_sc
```

You can run HLS by doing:

```
catapult -f go_hls.tcl &
```

Launch Questa by typing:

```
dofile scverify.tcl
```

In Questa type:

```
run -all
wave zoom full
```

Look at the waveforms and then properly quit from the simulator. Properly quitting will cause the post-HLS memory logs to be created.

At this point the following memory logs will exist:

```
mem_posthls_read.log
mem_posthls_write.log
mem_prehls_read.log
mem_prehls_write.log
```

There is a memory initialization loop in the reset state of the DUT:

```
  void main() {
    for (unsigned u=0; u < mem_size; u++)
      mem[u] = 0;
    wait();  // Reset state
```

In the pre-HLS model, the memory is initialized by writing locations in ascending order, as per the code. Interestingly, in the post-HLS model, the memory is initialized in the reverse order. This behavior is not incorrect, it is just different. However, it does complicate the comparison of memory logs a bit. We need to sort the logs by address to see the real differences:

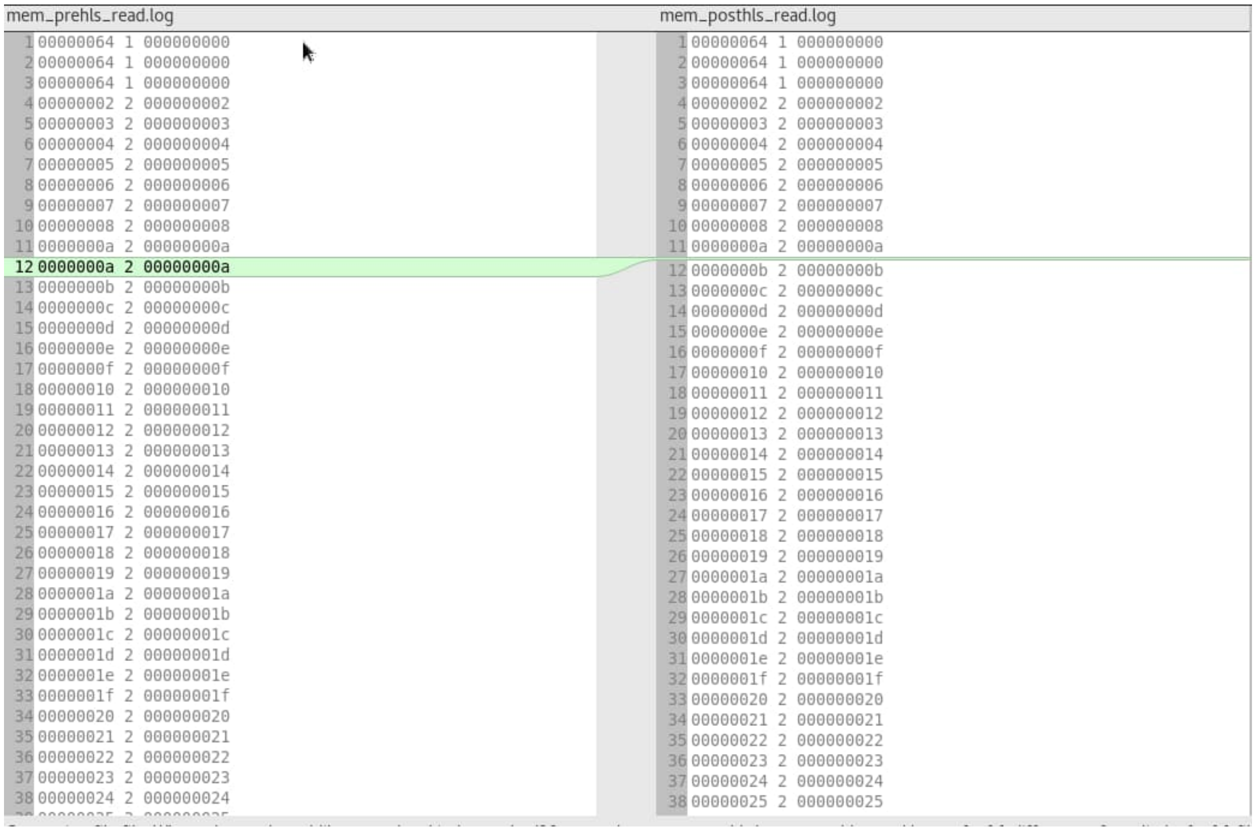
```
sort -k1 mem_posthls_write.log > mem_posthls_write_sorted.log
sort -k1 mem_prehls_write.log > mem_prehls_write_sorted.log
diff mem_prehls_write_sorted.log mem_posthls_write_sorted.log
```

This will show there are no real differences in what is written to the RAM in the pre-HLS versus post-HLS models.

Now you can compare the read logs:

```
kompare *pre*read* *post*read*
```

This will show:



mem_prehls_read.log

```
00000064 1 000000000
00000064 1 000000000
00000064 1 000000000
00000002 2 000000002
00000003 2 000000003
00000004 2 000000004
00000005 2 000000005
00000006 2 000000006
00000007 2 000000007
00000008 2 000000008
0000000a 2 00000000a
0000000a 2 00000000a
0000000b 2 00000000b
0000000c 2 00000000c
0000000d 2 00000000d
0000000e 2 00000000e
0000000f 2 00000000f
00000010 2 000000010
00000011 2 000000011
00000012 2 000000012
00000013 2 000000013
00000014 2 000000014
00000015 2 000000015
00000016 2 000000016
00000017 2 000000017
00000018 2 000000018
00000019 2 000000019
0000001a 2 00000001a
0000001b 2 00000001b
0000001c 2 00000001c
0000001d 2 00000001d
0000001e 2 00000001e
0000001f 2 00000001f
00000020 2 000000020
00000021 2 000000021
00000022 2 000000022
00000023 2 000000023
00000024 2 000000024
```

mem_posthls_read.log

```
00000064 1 000000000
00000064 1 000000000
00000064 1 000000000
00000002 2 000000002
00000003 2 000000003
00000004 2 000000004
00000005 2 000000005
00000006 2 000000006
00000007 2 000000007
00000008 2 000000008
0000000a 2 00000000a
0000000b 2 00000000b
0000000c 2 00000000c
0000000d 2 00000000d
0000000e 2 00000000e
0000000f 2 00000000f
00000010 2 000000010
00000011 2 000000011
00000012 2 000000012
00000013 2 000000013
00000014 2 000000014
00000015 2 000000015
00000016 2 000000016
00000017 2 000000017
00000018 2 000000018
00000019 2 000000019
0000001a 2 00000001a
0000001b 2 00000001b
0000001c 2 00000001c
0000001d 2 00000001d
0000001e 2 00000001e
0000001f 2 00000001f
00000020 2 000000020
00000021 2 000000021
00000022 2 000000022
00000023 2 000000023
00000024 2 000000024
00000025 2 000000025
```

In this case we see that the read at address 0xa in the pre-HLS simulation is missing in the post-HLS simulation model. Looking at the testbench.cpp file, we see that this is where we are intentionally violating the "`ignore_memory_precedence`" pledge discussed at the beginning of this section. Specifically, we are intentionally reading and writing to the same address in the same cycle. The post-HLS RAM model simply omits the read operation in this error case, so it shows up in the memory logs as a missing read operation.

To conclude the discussion on this example, it should be clear that the memory log feature is a useful tool for comparing pre-HLS and post-HLS simulation results, and it is also useful for detecting and debugging incorrect usage of directives such as "`ignore_memory_precedence`".

Example #4

The fourth example is Matchlib toolkit example `41*`. This example uses the `ScratchpadClass` to build a banked memory in the DUT, and is explained in detail in the matchlib_memory_modeling_methodology.pdf document within the Matchlib toolkit examples “doc” directory. The `ScratchpadClass` is instrumented to produce memory logs in the same manner as `extended_array`. To see the memory logs that are generated for this example, type:

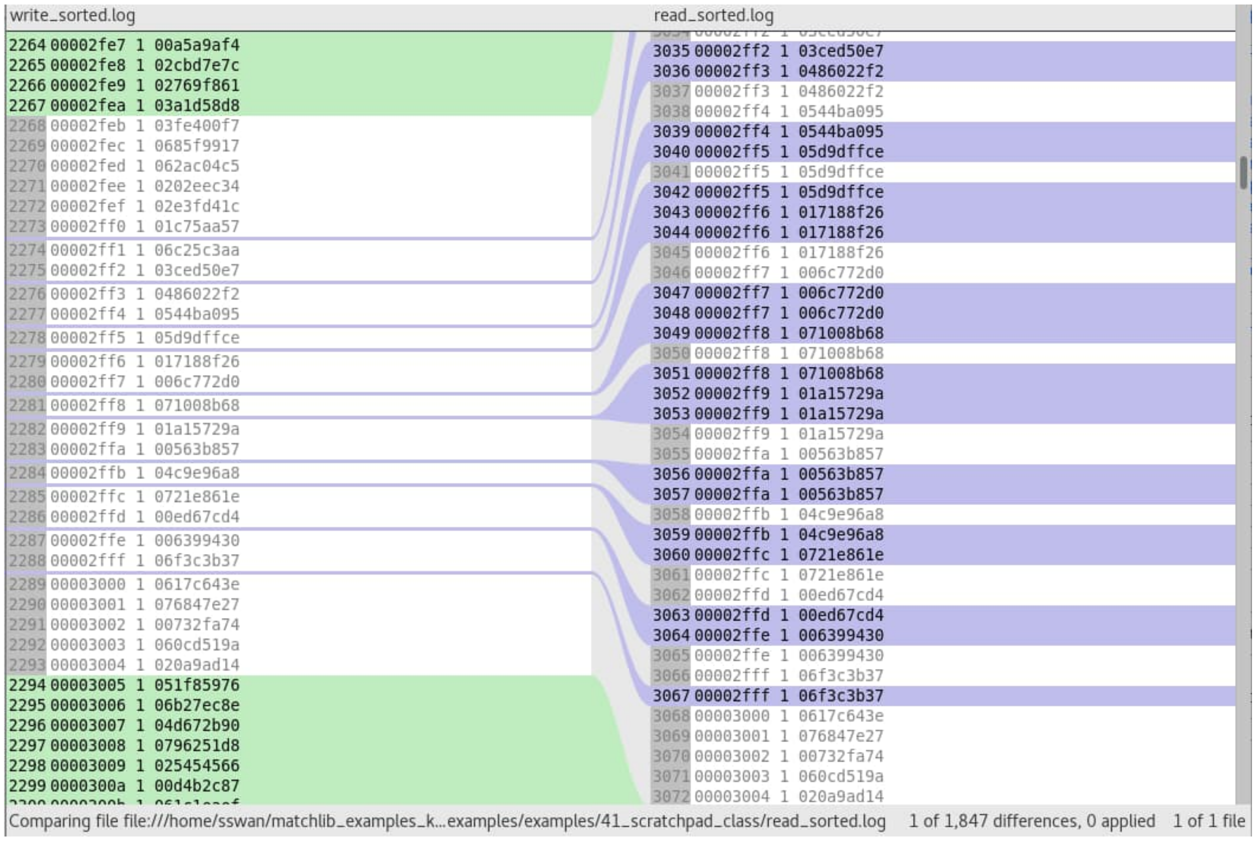
```
        cd 41_scratchpad_class
        make build
        ./sim_sc
```

This will produce the memory logs:

```
mem_prehls_read.log
mem_prehls_write.log
```

The orders of writes and reads are random, so they should be sorted so they are understandable:

```
sort -k1 *write* > write_sorted.log
sort -k1 *read* > read_sorted.log
kompare w*sorted* r*sorted*
```



write_sorted.log

```
00002fe7 1 00a5a9af4
00002fe8 1 02cbd7e7c
00002fe9 1 02769f861
00002fea 1 03a1d58d8
00002feb 1 03fe400f7
00002fec 1 0685f9917
00002fed 1 062ac04c5
00002fee 1 0202eec34
00002fef 1 02e3fd41c
00002ff0 1 01c75aa57
00002ff1 1 06c25c3aa
00002ff2 1 03ced50e7
00002ff3 1 0486022f2
00002ff4 1 0544ba095
00002ff5 1 05d9dffce
00002ff6 1 017188f26
00002ff7 1 006c772d0
00002ff8 1 071008b68
00002ff9 1 01a15729a
00002ffa 1 00563b857
00002ffb 1 04c9e96a8
00002ffc 1 0721e861e
00002ffd 1 00ed67cd4
00002ffe 1 006399430
00002fff 1 06f3c3b37
00003000 1 0617c643e
00003001 1 076847e27
00003002 1 00732fa74
00003003 1 060cd519a
00003004 1 020a9ad14
00003005 1 051f85976
00003006 1 06b27ec8e
00003007 1 04d672b90
00003008 1 0796251d8
00003009 1 025454566
0000300a 1 00d4b2c87
```

read_sorted.log

```
00002ff2 1 03ced50e7
00002ff3 1 0486022f2
00002ff3 1 0486022f2
00002ff4 1 0544ba095
00002ff4 1 0544ba095
00002ff5 1 05d9dffce
00002ff5 1 05d9dffce
00002ff5 1 05d9dffce
00002ff6 1 017188f26
00002ff6 1 017188f26
00002ff6 1 017188f26
00002ff7 1 006c772d0
00002ff7 1 006c772d0
00002ff7 1 006c772d0
00002ff8 1 071008b68
00002ff8 1 071008b68
00002ff8 1 071008b68
00002ff9 1 01a15729a
00002ff9 1 01a15729a
00002ff9 1 01a15729a
00002ffa 1 00563b857
00002ffa 1 00563b857
00002ffa 1 00563b857
00002ffb 1 04c9e96a8
00002ffb 1 04c9e96a8
00002ffc 1 0721e861e
00002ffc 1 0721e861e
00002ffd 1 00ed67cd4
00002ffd 1 00ed67cd4
00002ffe 1 006399430
00002ffe 1 006399430
00002fff 1 06f3c3b37
00002fff 1 06f3c3b37
00003000 1 0617c643e
00003001 1 076847e27
00003002 1 00732fa74
00003003 1 060cd519a
00003004 1 020a9ad14
```

Comparing file file:///home/sswan/matchlib_examples_k...examples/examples/41_scratchpad_class/read_sorted.log   1 of 1,847 differences, 0 applied   1 of 1 file

Briefly, what this is showing is that all addresses are written exactly once (on the left), and that they are sometimes read multiple times (note duplicated addresses in purple on the right). Some addresses are never read. Even when there are duplicate reads, if you look carefully you can see that the values on the left and right always match. All of these are correct results, reflecting the code in the testbench and DUT.

General Tips for Using Memory Logs

1. Because all entries in memory write logs and read logs have write counts, log files can be sorted based on their address (`sort -k1`), and key information is not lost. This is valuable since memory access operations may be legitimately reordered in both pre-HLS scenarios (as discussed at the beginning of this document), and memory accesses may also be reordered during HLS.
2. In effect, memory logs treat each address in the memory as an independent communication channel, and the write_count indicates the sequence count for the transaction (or "message")

transmitted through that specific channel. The value field in the memory log indicates the value of the transaction written to or read from the channel.

3. The most common access pattern for memory usage is where every value written to a memory address is read exactly once. In this case, the write counts and values will be identical in the write logs versus the read logs.
4. Other patterns are possible, for example every value written to a memory address may always be read twice. This is analogous to an "upsample" operation in the DSP domain, and in terms of the memory logs you would see the read log showing read operations to addresses occurring twice with the same write count.
5. In most HW designs, the memory access patterns are simple and fixed, and should fit well with the framework described above.
6. When using memory logs to debug a problem, usually if the write_counts differ between the correct log and the incorrect log, this is likely to point to the root cause.