Custom Memory Protocol Modeling Guide

Stuart Swan
Platform Architect
Siemens EDA
17 September 2025

Introduction

This document shows how to model complex memory protocols that may include stall signals driven by the memories, while targeting high throughput and best possible QOR in Catapult HLS.

The complete source code for this tutorial is available here:
https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/tree/master/matchlib_examples/examples/54_mem_with_stall

Design Example Used in this Guide

The design used in this document has specific design requirements that influence the way the memory protocol transactor is modeled. However, the intent of this document is to show a set of general techniques which can be used to model many specific memories and bus interfaces. In many cases the code used in this example could be used as a starting point and tailored to meet different memory and bus interface protocols.

The DUT in this design has two memory master interfaces. The two memories are each 1rw memories, and the memories can assert a stall signal to indicate when they cannot accept new requests. Since they are 1rw memories, they can only accept either one read request or one write request on each clock cycle.

The data width of these memories is assumed to be very wide (e.g. 100s of bits).

```
// memory controller side of interface
template <typename T_data, typename T_addr>
class mem_with_stall_out {
 public:
  mem_with_stall_out(const char* name = "")
      : stall(SYNTH_NAME(name, "_stall"))
      , read_en(SYNTH_NAME(name, "_read_en"))
      , write_en(SYNTH_NAME(name, "_write_en"))
      , address(SYNTH_NAME(name, "_address"))
      , write_data(SYNTH_NAME(name, "_write_data"))
      , read_data(SYNTH_NAME(name, "_read_data")) {}

  sc_in<bool> stall;
  sc_out<bool> read_en;
  sc_out<bool> write_en;
  sc_out<T_addr> address;
  sc_out<T_data> write_data;
  sc_in<T_data> read_data;
```

The DUT declaration is:

```
11 class dut : public sc_module {
12  public:
13   static const unsigned int READ_ADDR1_OFFSET = 0;
14   static const unsigned int READ_ADDR2_OFFSET = 8;
15   static const unsigned int WRITE_ADDR_OFFSET = 16;
16   static const unsigned int MEM_SIZE = 1024;
17   using T_data = int;
18   using T_addr = unsigned int;
19   sc_in_clk SC_NAMED(clk);
20   sc_in<bool> SC_NAMED(rst_n);
21
22   sc_in<bool> SC_NAMED(start);
23
24   mem_with_stall_out<T_data, T_addr> SC_NAMED(mem1);
25   mem_with_stall_out<T_data, T_addr> SC_NAMED(mem2);
26   sc_out<bool> SC_NAMED(done);
27
28   // abstract the memory interface to use array subscript
29   mem_with_stall_out_xact<T_data, T_addr> SC_NAMED(mem1_xact);
30   mem_with_stall_out_xact<T_data, T_addr> SC_NAMED(mem2_xact);
31
32
33   SC_HAS_PROCESS(dut);
34
35   dut(sc_module_name name) {
36     SC_CTHREAD(ProcessingThread, clk.pos());
37     async_reset_signal_is(rst_n, 0);
38     mem1_xact.bind(clk, rst_n, mem1, mem2.stall); // swapped stall intentional
39     mem2_xact.bind(clk, rst_n, mem2, mem1.stall); // swapped stall intentional
40   }
41
42   void ProcessingThread();
43 };
```

On lines 24 and 25 we see the memory master interfaces. On lines 29 and 30 we instantiate transactors to enable indexing operators to be used to access the memories.

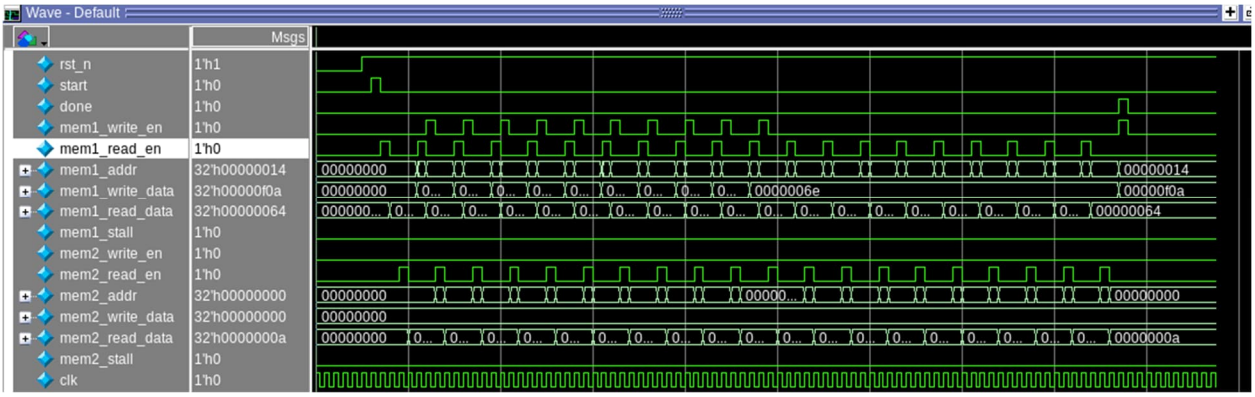
The DUT ProcessingThread is:

```
 6 void dut::ProcessingThread() {
 7   done.write(false);
 8   mem1_xact.Reset();
 9   mem2_xact.Reset();
10   CCS_LOG("Processing Thread resetting");
11   wait();
12
13   CCS_LOG("Processing Thread reset");
14 update_loop:
15   while (true) {
16     do {
17       wait();
18     } while (!start);
19
20 #pragma hls_pipeline_init_interval 2
21 #pragma pipeline_stall_mode stall
22     for (unsigned int idx = 0; idx < 10; ++idx) {
23       mem1_xact[idx + 10] = mem1_xact[idx] + mem2_xact[idx];
24       CCS_LOG("Doing vector add");
25     }
26
27     CCS_LOG("Done with vector add");
28     T_data accum = 0;
29 #pragma hls_pipeline_init_interval 1
30 #pragma pipeline_stall_mode stall
31     for (unsigned int idx = 0; idx < 10; ++idx) {
32       accum += mem1_xact[idx] * mem2_xact[idx];
33     }
34
35     mem1_xact[20] = accum;
36
37     CCS_LOG("Done with dot product");
38     done.write(true);
39     wait();
40     done.write(false);
41   }
42 }
```

On line 22 we have a loop that is pipelined with an II=2 and which reads both memories and writes one of them on each iteration. On line 31 we have a loop that is pipelined with an II=1 and which reads both memories (but performs no writes) on each iteration.

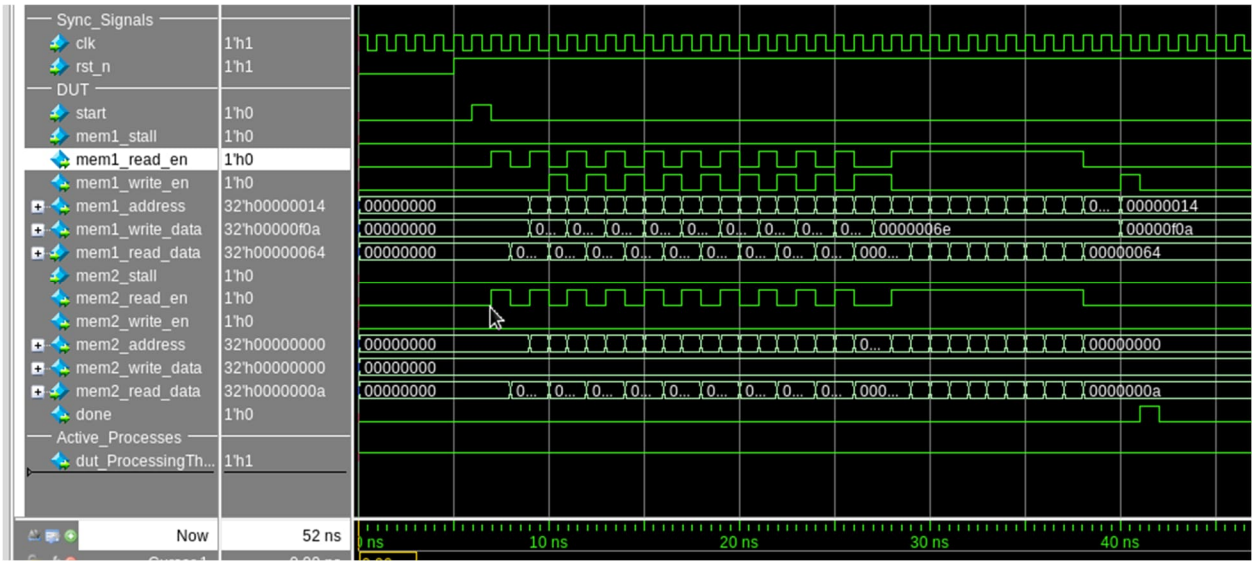
Running the Design

We first run the pre-HLS design with memory stalling disabled. The waveforms for this case are:



We see that the highlighted mem1_read_en signal is only high every third clock cycle. The reason for this is that in the pre-HLS sim, the design is stuttering. (The pre-HLS model is not pipelined, so the delay between the read requests and the read responses makes the design stutter).

When we run the post-HLS simulation, we see:

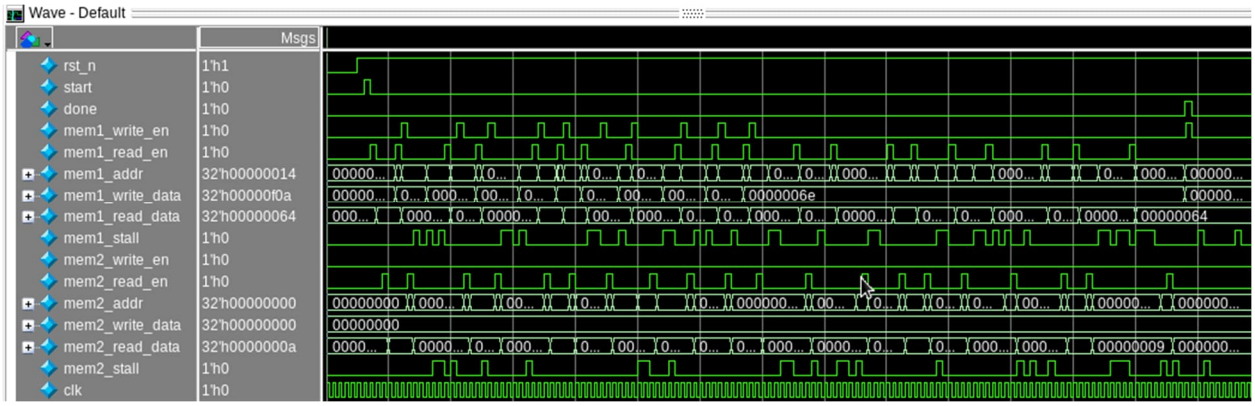


Now the design is pipelined. For the first loop, we see that mem1_read_en and mem1_write_en are enabled every other clock cycle. This is optimal throughput (II=2). For the second loop, we see mem1_read_en is enabled on every clock cycle, showing optimal throughput with II=1.

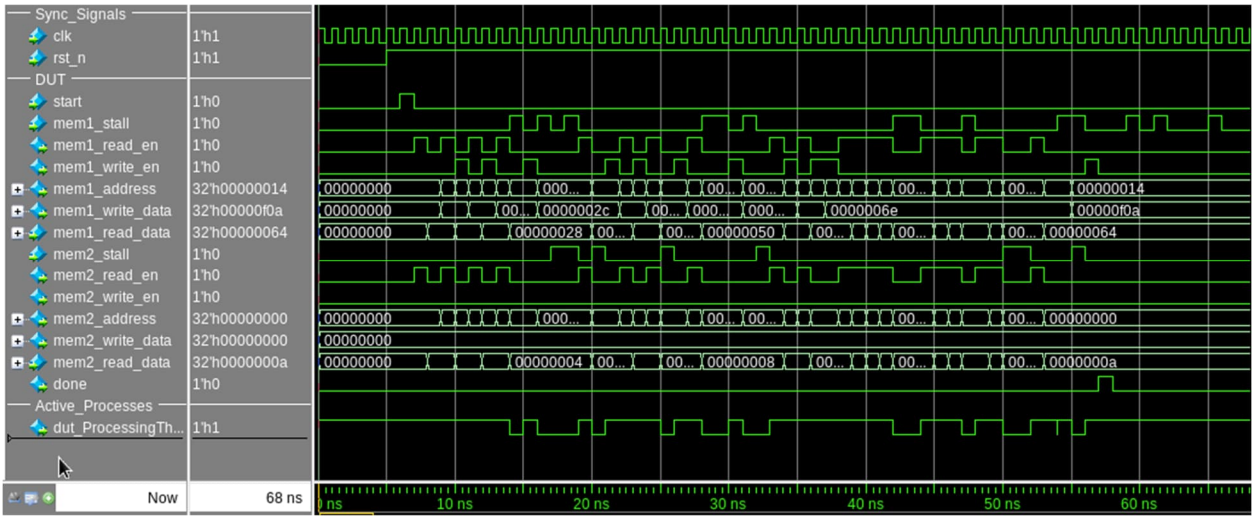
We can enable stalling by setting the do_stall flags in testbench.cpp:

```
64
65      mem1.debug = 1;
66      mem1.do_stall = 1;  // Set to 1 to enable stall
67      mem2.debug = 1;
68      mem2.do_stall = 1;  // Set to 1 to enable stall
69      mem1.clk(clk);
```

When we then run the pre-HLS simulation we can see the effects of the stalling, but functionally the design behaves identically to the no stall case:



When we run the post-HLS design with stalling enabled, we see:



We can see the effects of the stalling, but functionally the design results are the same as the no stall case.

Memory Transactor Implementation Approach

Our approach for implementing the memory transactor is to convert array accesses into Push/Pop operations which represent read and write requests and read responses. The `mem_with_stall_out_xact` class overloads the index operators so that it can be used like an array with the DUT:

```
479   elem_proxy<T_data, T_addr> operator[](T_addr idx) {
480     return elem_proxy<T_data, T_addr>(*this, idx);
481   }
482
483   const elem_proxy<T_data, T_addr> operator[](T_addr idx) const {
484     return elem_proxy<T_data, T_addr>(*this, idx);
485   }
486 };
```

The `elem_proxy` class converts array accesses into Push/Pop operations:

```
442     operator T_element_data() {
443       mem_with_stall_req<T_element_data, T_element_addr> req;
444       req.write_en = false;
445       req.read_en = true;
446       req.address = idx;
447       xact.rd_req_chan.Push(req);
448       auto rsp = xact.rsp_chan.Pop();
449       return rsp.read_data;
450     }
451
452     void operator=(const T_element_data& val) {
453       mem_with_stall_req<T_element_data, T_element_addr> req;
454       req.write_en = true;
455       req.read_en = false;
456       req.address = idx;
457       req.write_data = val;
458       xact.wr_req_chan.Push(req);
459     }
460   };
```

It is important to note that HLS does not know that the DUT is accessing a memory. All it sees after the design is compiled are the various Push/Pop operations that represent the memory traffic. A key reason for resolving the memory accesses to Push/Pop operations is that they have full flow control with rdy/vld signaling, which enables us to easily and efficiently implement the stall functionality.

Transactor Scheduling Considerations

For a given Push/Pop channel, HLS always keeps transactions in order. If we were to have a single combined read and write request channel, then the order of read and write requests in the pre-hls model would need to be preserved in the post-hls model. For the first loop, this approach would prevent pipelining, since the first loop pipeline needs to be ramped up first with multiple sequential read requests before the first write request is performed. Thus, it is essential for pipelining the first loop to have separate read and write request channels, as shown in the code directly above.

The Catapult directives include the following:

```
directive set /dut/dut:ProcessingThread/ProcessingThread -STRICT_MIO_SCHEDULING false
go architect
directive set /dut/dut:ProcessingThread/ProcessingThread/mem1_xact.rsp_chan.Pop() -CSTEPS_FROM {{mem1_xact.rd_req_chan.Push() == 1}}
directive set /dut/dut:ProcessingThread/ProcessingThread/mem2_xact.rsp_chan.Pop()#1 -CSTEPS_FROM {{mem2_xact.rd_req_chan.Push()#1 == 1}}
directive set /dut/dut:ProcessingThread/ProcessingThread/mem1_xact.rsp_chan.Pop()#2 -CSTEPS_FROM {{mem1_xact.rd_req_chan.Push()#2 == 1}}
directive set /dut/dut:ProcessingThread/ProcessingThread/mem2_xact.rsp_chan.Pop()#3 -CSTEPS_FROM {{mem2_xact.rd_req_chan.Push()#3 == 1}}
```

The STRICT_MIO_SCHEDULING=false directive enables Catapult to freely reorder Push/Pop operations on different channels. This is required to enable the first loop to be pipelined.

The four CSTEPS_FROM directives tell Catapult to schedule the read response Pop operations 1 cycle after their corresponding read request Push operations. This is needed to enable the pipelined design to perform at optimal throughput. In effect, this directive informs Catapult about the 1 cycle latency of the memory for read operations.

Finally, the pipelined loops use stall_mode=stall, since we do not need the pipelines to flush when a stall occurs (we want the pipelines to stall):

```
20 #pragma hls_pipeline_init_interval 2
21 #pragma pipeline_stall_mode stall
22     for (unsigned int idx = 0; idx < 10; ++idx) {
23       mem1_xact[idx + 10] = mem1_xact[idx] + mem2_xact[idx];
24       CCS_LOG("Doing vector add");
25     }
26
27     CCS_LOG("Done with vector add");
28     T_data accum = 0;
29 #pragma hls_pipeline_init_interval 1
30 #pragma pipeline_stall_mode stall
31     for (unsigned int idx = 0; idx < 10; ++idx) {
32       accum += mem1_xact[idx] * mem2_xact[idx];
33     }
```

Transactor Implementation Considerations

As mentioned earlier, the design requirements are that the memory data width is very wide (100s of bits), and the transactor implementation must have best possible QOR – matching what could be achieved in fully optimized handwritten Verilog RTL. Specifically, the transactor must:

- Add minimal latency between the DUT and the memories.
- Provide highest possible throughput.
- Have minimal area costs (in particular, it must not store either read or write data due to its width).

All the transactor code is implemented within the `mem_with_stall_out_xactor` class.

To achieve these goals, we use the "RTL in SystemC" modeling approach, which enables us to create precisely crafted logic. For the transactor, this logic will include only the following:

- Two 1-bit delay lines to track rd_req/wr_req state. (`rd_start` and `wr_req_delayed`)
- Several combinational gates to coordinate rdy/vld signals to the DUT, and control signals to the memories.
- The "`disable_spawn()`" functions are used within the transactor to inform the Matchlib library that we are directly accessing the rdy/vld/dat signals on Push/Pop channels.

When either memory asserts its stall signal, the transactors:

- Apply backpressure to all rdy/vld signals to the DUT
- Freeze the state of all sequential processes (the two delay lines) within the transactor

If both a read request and a write request are presented to the transactor on the same clock cycle, reads take priority, and the writes are deferred. This is because typically reads occur at the beginning of the pipeline, and we want to keep the pipeline from stalling. Writes occur at the end of the pipeline, so if they are deferred the pipeline will not stall. When a write request is deferred, the next clock cycle must have no new requests received from the DUT, so that the deferred write can be executed. This condition is checked via an HLS_ASSERTION within the transactor. When a deferred write occurs, we rely on the fact that the write request data and address will be held stable from the previous clock cycle when the

write request was consumed by the transactor. This optimization enables us to avoid storing the write request data internally to the transactor, saving area.

Conclusion

This modeling guide has shown some of the techniques to create highly optimized transactors for complex memory and bus protocols. The code within this example can be used as a starting point for other memories and bus protocols.

In most typical cases, the modeling approach shown in this document does not need to be used:

- In general, memories which do not drive stall signals, and which have fixed latency, can be easily modeled within the Catapult memory generator tool, and then used directly within Catapult.
- Most modern bus protocols such as AXI4 use rdy/vld/dat signaling and can be easily layered on top Matchlib connections.

However, complex memories with stall signals, and legacy protocols such as AHB which do not use rdy/vld signaling would use a modeling approach very similar to the one shown in this document.