Catapult Matchlib Memory Modeling Methodology

Stuart Swan
Platform Architect
Siemens EDA
24 March 2023

Introduction

This document introduces some common scenarios and useful guidelines for modeling memories when using Matchlib and Catapult HLS.

In pre-HLS models, memories appear as normal C/C++ arrays, and are often mapped to RAMs during HLS. To meet performance and area goals, RAMs and their associated logic must often be carefully constructed. For example, to meet design throughput requirements, RAMs may often need to have multiple banks so that multiple accesses to the banks can occur concurrently.

The discussion in this document assumes that the RAMs used in the examples support a maximum of 1 read and 1 write operation per clock cycle, which is a typical limit. Note that it is possible to use RAMs with more ports in Catapult (e.g. 2r2w), but there is a significant area cost for such RAMs so they are typically avoided unless required.

The Catapult Matchlib memory modeling methodology described in this document can be used in both the Catapult SystemC and C++ flows. The Catapult SystemC flow examples which accompany this document are examples 40*, 41*, and 42* in the Matchlib examples kit. The Catapult C++ flow examples which accompany this document are examples 43* and 44* in the Matchlib examples kit.

Terms Used in this Document

implicit memory: an implicit memory appears as a normal C/C++ array in the pre-HLS model and is mapped to a RAM during HLS.

explicit memory: an explicit memory appears as an instantiation of a particular RAM module (sc_module) within a SystemC design.

shared memory: a shared memory is a memory that is directly accessed by more than one process. The processes may be in the DUT and/or within the testbench.

banked memory: a banked memory is a group of related memories (banks) that are constructed to operate like a single larger memory. Each bank has its own dedicated RAM ports, so multiple IO requests may proceed concurrently to separate banks. Done properly, banking increases memory IO thruput while adding minimal or no area or performance penalty compared to implementing the same RAM storage without banking.

interleave: interleave is a banked memory organization where the low order bits of the addresses are used to select specific banks. Catapult has the INTERLEAVE directive which can automatically create banked memories with this organization.

block_size: block_size is used to specify a banked memory organization where the high order bits of the addresses are used to select specific banks. Catapult has the BLOCK_SIZE directive which can automatically create banked memories with this organization.

word_width: Catapult has the WORD_WIDTH directive which is used to widen the width of the data that is read or written on each RAM IO operation, compared to the width of the array accesses in the pre-HLS model. Sometimes this can increase design throughput.

Scope of this Document

Shared memories will be covered in future update to this document.

- Note that you can always wrap a non-shared memory in a process and then share it, this is demonstrated in Catapult Matchlib example 35*.

Usage of explicit memories is now deprecated, and explicit memories should not be used in new designs.

- There is an example of an explicit memory in Catapult Matchlib example 12*.

Goals of the Matchlib Memory Modeling Methodology

1. Meet PPA (power, performance, area) goals without any compromises.
2. Discover and resolve all functional and performance issues related to memories in the pre-HLS model, not in the post-HLS model.
3. The pre-HLS model code should be clean and easy to understand.

In a sense, the third goal is somewhat in conflict with the first two goals, since in general an appropriate coding style must be followed to achieve goals #1 and #2. This coding style is easy to learn and follow, however, so in general the third goal can be met too.

It is important to understand that memories play a key role in most designs, and that functional and performance issues related to memory organization or surrounding logic is a common problem. This makes goal #2 very important, since finding and fixing problems in the post-HLS RTL can be very time-consuming because the RTL is machine-generated.

Example Design #1

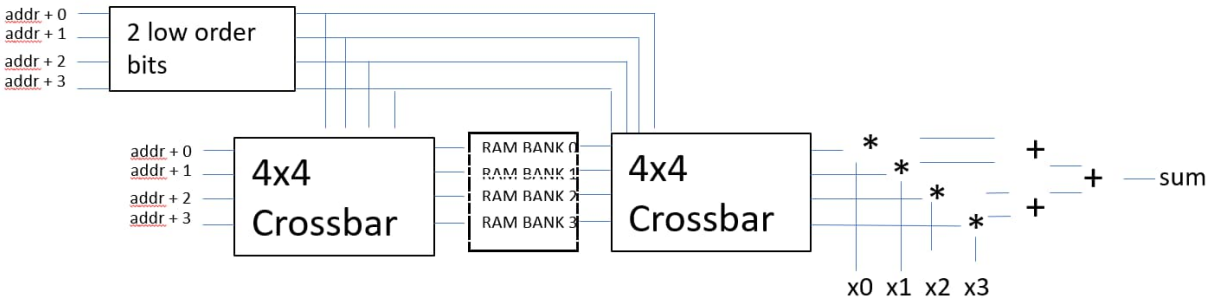
```
uint16 coeffs[1024];

 void main() {
    wait();

#pragma hls_pipeline_init_interval 1
#pragma pipeline_stall_mode flush
  while (1) {
    uint16 addr = addr_in.Pop();
    uint16 x0 = in0.Pop();
    uint16 x1 = in1.Pop();
    uint16 x2 = in2.Pop();
    uint16 x3 = in3.Pop();
    uint16 sum = (x0 * coeffs[addr + 0]) +
                 (x1 * coeffs[addr + 1]) +
                 (x2 * coeffs[addr + 2]) +
                 (x3 * coeffs[addr + 3]);
    out1.Push(sum);
  }
 }
```

This is a sketch of a simple multiply accumulate (or MAC) design to be synthesized thru HLS. This design is similar to example 41* in the Catapult Matchlib examples. The design goal is an II=1, so a new "sum" output will be produced on every clock cycle. For this to be achieved the coeffs array must support 4 uint16 reads per clock cycle. We can meet this goal by using a banked memory architecture where the bank selection is the two low order address bits. It is easy to analyze the indexes of the array accesses and prove that there will be no bank conflicts on each cycle. This is true even if "addr" is not evenly divisible by 4, which is a case that the HW for this design would need to support.

The HW that must be generated for this design will be similar to the following:



The crossbars shown above are designed under the assumption that there are never any conflicts in routing inputs to outputs. This enables the HW to be simple, and also means that there is no need for arbitration or queueing of any of the memory read requests. All the read requests have a fixed delay to produce their outputs. None of the HW in the diagram above ever applies backpressure on upstream requests.

Example Design #2

```
  uint16 coeffs[1024];
```

```
  void main() {
    wait();

#pragma hls_pipeline_init_interval 1
#pragma pipeline_stall_mode flush
  while (1) {
    uint16 addr = addr_in.Pop();
    uint16 x0 = in0.Pop();
    uint16 x1 = in1.Pop();
    uint16 x2 = in2.Pop();
    uint16 x3 = in3.Pop();
    uint16 sum = (x0 * coeffs[func0(addr)]) +
                 (x1 * coeffs[func1(addr)]) +
                 (x2 * coeffs[func2(addr)]) +
                 (x3 * coeffs[func3(addr)]);
    out1.Push(sum);
  }
 }
```

This design is the same as Example #1, except that the coeffs indexes now depend on func0, func1, etc. If it can be proved that the low order address bits of these functions will never result in any bank conflicts, then the same HW as shown in example #1 can be used. However, if it cannot be proved that there will not be any bank conflicts, then more complex HW will need to be created. This HW will look similar to the HW in Example #1, but it will need to insert arbiters to handle competing requests to the same RAM bank. Because there may be bank contention, bank requests may need to be delayed, which means that the coeffs read operations now need to apply backpressure to the calling process when contention occurs. This possible backpressure needs to be handled carefully so that the overall design can still continue executing and making progress, otherwise the whole design would deadlock.

When there is bank contention, clearly a new "sum" output will no longer be guaranteed to be produced on every clock edge.

Example Design #3

```
  uint16 coeffs[1024];

  void main() {
    wait();

#pragma hls_pipeline_init_interval 1
#pragma pipeline_stall_mode flush
  while (1) {
    uint16 addr0 = addr_in0.Pop();
    uint16 addr1 = addr_in1.Pop();
    uint16 addr2 = addr_in2.Pop();
    uint16 addr3 = addr_in3.Pop();
    uint16 x0 = in0.Pop();
    uint16 x1 = in1.Pop();
    uint16 x2 = in2.Pop();
```

```
    uint16 x3 = in3.Pop();
    uint16 sum = (x0 * coeffs[addr0]) +
                 (x1 * coeffs[addr1]) +
                 (x2 * coeffs[addr2]) +
                 (x3 * coeffs[addr3]);
    out1.Push(sum);
  }
 }
```

This design is similar to Example #1 and Example #2, but now all of the indexes into the coeffs array are addresses that are read from an input channel. Thus there is no way for an HLS tool to prove that there will never be any bank contention on the memory read operations. It is possible that the designer may know that the externally provided addresses will never result in bank contention. If this is the case, then the simpler and smaller HW described in Example #1 perhaps could still be used, rather than using the arbitration logic described in Example #2.

Catapult Memory Banking Directives

Catapult supports the INTERLEAVE and BLOCK_SIZE directives to automatically build banked memories. With the directive "-INTERLEAVE 4" set on the coeffs memory shown in Example #1, Catapult will do the index analysis and prove there are no bank conflicts and build HW similar to what is described in Example #1.

For Example #2, with the same INTERLEAVE directive, Catapult may or may not succeed in proving there are no bank conflicts. If it cannot prove there are no bank conflicts, it will not build a banked memory with arbitration, instead it will effectively give up on building a banked memory altogether. This may not result in any error or warning messages, but it will result in the II=1 goal not being met.

For Example #3, with the same INTERLEAVE directive, Catapult definitely will not succeed in proving there are no bank conflicts. It will not build a banked memory, and the II=1 goal will not be met.

So, we see that use of the Catapult INTERLEAVE directive for Examples #2 and #3 do not meet our top two goals for the memory modeling methodology:

1. Meet PPA (power, performance, area) goals without any compromises.

2. Discover and resolve all functional and performance issues related to memories in the pre-HLS model, not in the post-HLS model.

Because of these problems, we recommend that Catapult Matchlib users never directly use BLOCK_SIZE, INTERLEAVE, and WORD_WIDTH directives in their models. (BLOCK_SIZE and WORD_WIDTH have very similar issues as those described directly above).

Recommended Matchlib Memory Modeling Methodology

The recommended Matchlib memory modeling methodology is to use memory models in the pre-HLS design that explicitly specify the desired memory architecture. This guarantees that HLS will build exactly what the user wants, and it also enables all performance and functional problems related to the memory architecture to be easily observed in the pre-HLS simulation.

Scratchpad

The most commonly-used Matchlib memory model is called Scratchpad, and the model is currently located in the Catapult Matchlib examples toolkit include directory, called ScratchpadClass.h.

This model implements the simple crossbar structure described in the Example #1 scenario above. It has no arbitration or queueing of requests, thus all bank requests must be free of conflicts. In the pre-HLS simulation, if there are any bank conflicts, it will emit an error message at the time of the conflict. Because the model emits errors when there are bank conflicts, it is safe to use for the scenarios described in Example #2 and Example #3. In the latter case, if the user knows that the externally-provided addresses are free of bank conflicts, then the optimized HW can still be used.

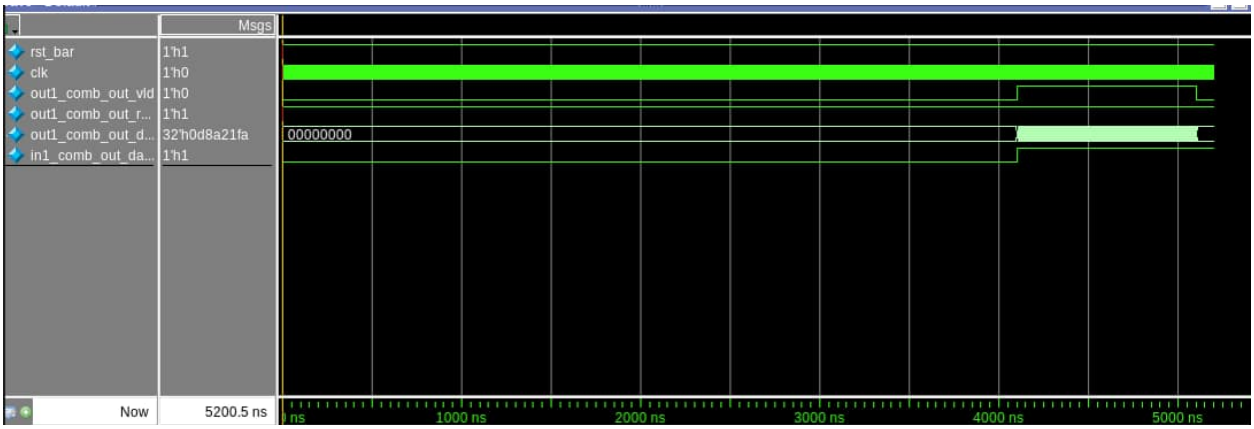
The Scratchpad has these characteristics:

- The template parameters are the word_type (ie the element type that the overall array stores), the number of banks, and the total capacity of the banked memory in words.
- Each request to the banked memory includes multiple inputs. The number of inputs is equal to the number of banks. Each input is on its own "lane".
- Similarly, each response includes multiple outputs.
- One new request is consumed on each invocation.
- One new response is produced on each invocation if the requesting operation is a load (i.e. a read).
- The model can be pipelined with II=1 or any other desired II.
- There are no variations in delay since there is no arbitration or backpressure.
- The scratchpad can be instantiated in your model as a basic class (ScratchpadClass<>) which is inlined into the calling process, or it can be instantiated as a separate SC module (Scratchpad<>) which has its own process.

The following code for the DUT in example 41* shows its usage, this usage is very similar to Design #1 above:

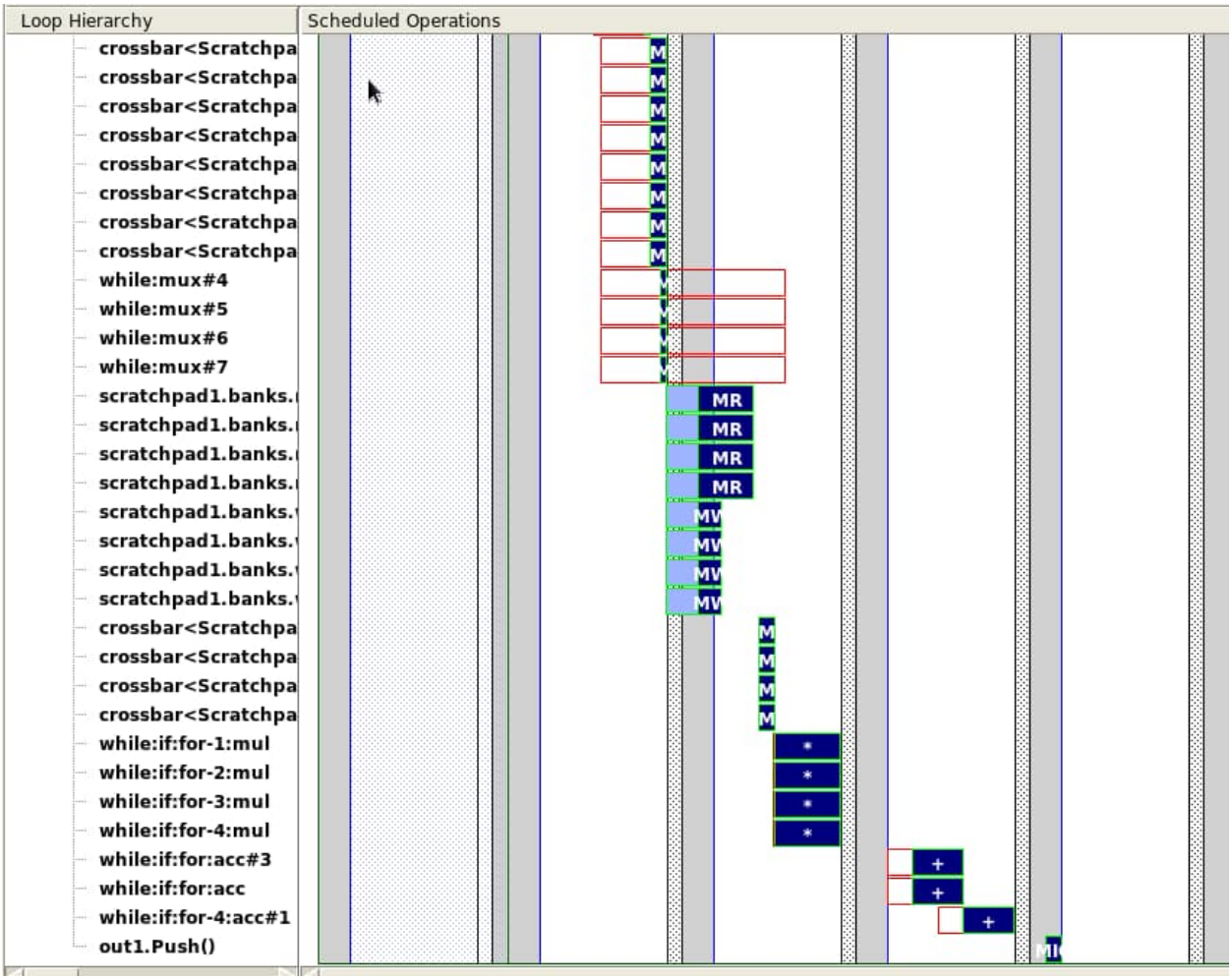
```
54   void run() {
55     in1.Reset();
56     out1.Reset();
57     wait();
58
59 #pragma hls_pipeline_init_interval 1
60 #pragma pipeline_stall_mode flush
61     while (1) {
62       // get the input request from the testbench
63       dut_in_t req1 = in1.Pop();
64
65       local_mem::scratchpad_req_t sp_req;  // local scratchpad request type
66
67       // copy incoming request to scratchpad request
68 #pragma hls_unroll yes
69       for (int i=0 ; i < local_mem::num_inputs; i++)
70         sp_req.set(i, req1.addr + i, req1.data[i]);
71
72       if (req1.is_load)
73       {
74         // if it is a load (i.e. read) operation, get the read data from the RAM
75         local_mem::base_rsp_t rsp = scratchpad1.load(sp_req);
76
77         // compute MAC
78         local_mem::word_type sum=0;
79 #pragma hls_unroll yes
80         for (int i=0; i < local_mem::num_inputs; i++) {
81           sum += rsp.data[i] * req1.data[i];
82         }
83
84         // Push out the sum
85         out1.Push(sum);
86       }
87       else
88       {
89         // if it is a store (i.e. write) operation, write the data to the RAM
90         scratchpad1.store(sp_req);
91       }
92     }
93   }
94 };
```

The following are the pre-HLS waveforms for example 41*:

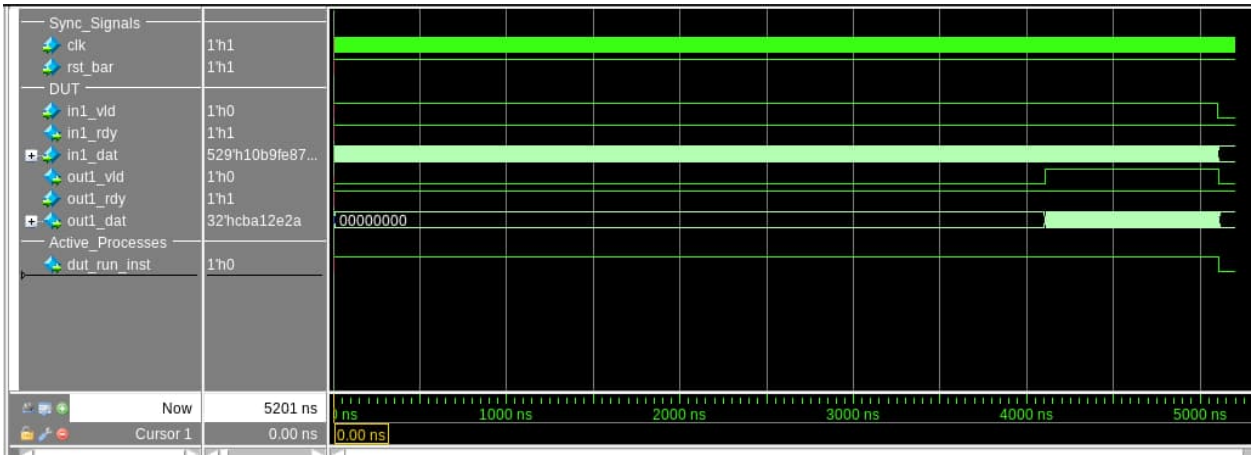


During the first 4000 ns 4000 write requests are done to the banked RAM, where each request has 16 separate lanes. During the last 1000 ns, 1000 read requests are done to the banked RAM, where each request has 16 separate lanes.

You can observe that the RAM is fully (and optimally) utilized since all the rdy/vld signals are high when they can be. Note that this means that Catapult has successfully pipelined this design with an II=1. There is a non-trivial amount of logic within the pipeline due to the two crossbar instantiations. To see the actual logic generated by Catapult for scratchpad for example 41*, we can reduce the number of banks from the original 16 down to 4 to make it more easily visible. This design then closely corresponds to Example #1. The schedule view is shown below, and you should note its close correspondence to the schematic diagram shown for Example #1.



The following are the post-HLS waveforms for example 41*. They are basically the same as the pre-HLS waveforms.



As a final comment on ScratchpadClass, if some of the bank addresses are constant (or have some bits which are constants), the HW for the crossbar will likely by optimized during HLS and RTL synthesis, so likely the HW cost will be lower than what it would be for a full crossbar.

ArbitratedScratchpad

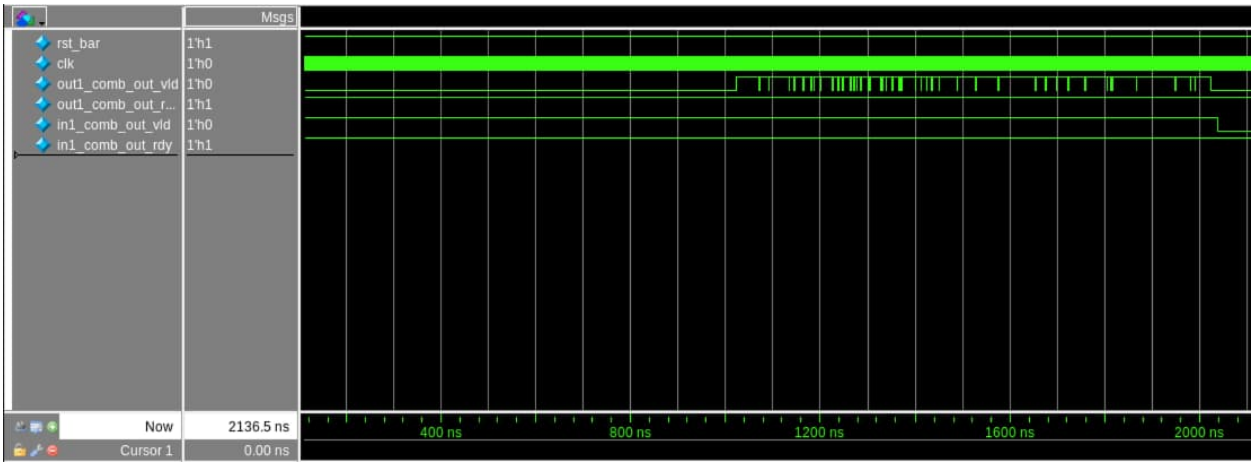
If your design needs a banked memory but there may be conflicts among bank request inputs, then a more complex model is needed. This model is ArbitratedScratchpad. It is very similar in concept to Scratchpad, but it has arbitration and queueing of request inputs since there may be conflicts among the banks.

ArbitratedScratchpad usage is shown in Catapult Matchlib example 42*. This is example is very similar to example 41* described above.

The following code for the DUT in example 42* shows its usage.

```
 64   void main() {
 65     out1.Reset();
 66     in1.Reset();
 67
 68     bool was_consumed[local_mem::num_inputs];
 69     bool all_consumed = 1;
 70
 71     wait();
 72
 73     local_mem::req_t req;
 74     local_mem::rsp_t rsp;
 75
 76 #pragma hls_pipeline_init_interval 1
 77 #pragma pipeline_stall_mode flush
 78     while (1) {
 79       // if there are remaining requests in some lanes that were not yet consumed
 80       // then dont Pop a new input request
 81       if (all_consumed)
 82         req = in1.Pop();
 83
 84       mem.load_store(req, rsp, was_consumed);
 85
 86       bool any_valid = 0;
 87       all_consumed = 1;
 88 #pragma hls_unroll yes
 89       for (unsigned i=0; i < local_mem::num_inputs; i++) {
 90         // check if some of the requests were not consumed..
 91         if (was_consumed[i] == 0)
 92           all_consumed = 0;
 93
 94         // if a particular request was consumed, then it is no longer valid
 95         if (was_consumed[i] == 1)
 96           req.valids[i] = 0;
 97
 98         // check if we have any read (aka "load") responses
 99         if (rsp.valids[i] == 1)
100           any_valid = 1;
101       }
102
103       if (any_valid)
104         out1.Push(rsp);
105
106       wait();
107     }
108   }
```

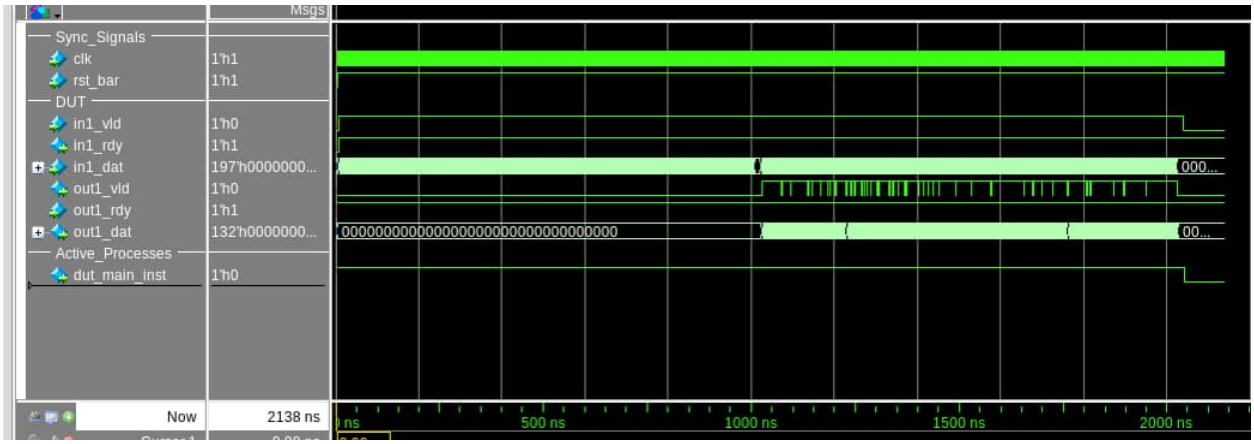
The following are the pre-HLS waveforms for example 42*:



During the first 1000 ns 1000 write requests are done to the banked RAM, where each request has 16 separate input lanes. During the last 1000 ns, 1000 read requests are done to the banked RAM, where each request and response has 16 separate lanes.

You can observe that the RAM is almost fully utilized since all the rdy/vld signals are almost always high when they can be. The rdy/vld signals are occasionally not high when they can be because of the arbitration and backpressure, and because the testbench occasionally sends empty requests.
Note that Catapult has successfully pipelined this design with an II=1. There is a non-trivial amount of of logic within the pipeline due to the two crossbar instantiations as well as the arbitration and queuing logic.

The following are the post-HLS waveforms for example 42*. They are basically the same as the pre-HLS waveforms.



The ArbitratedScratchpad has these characteristics:

- Generally, any desired thruput can be achieved simply by increasing the number of banks in the memory and the number of inputs in each request
- Generally, you want the number of banks to be greater (e.g. 4-16x) than number of inputs in each request. This reduces possible bank contention at minimal additional HW cost.
- Bank contention is OK, so no error or warning message is emitted when it occurs.

- Note the "was_completed" flags in dut.h. These indicate if requests were accepted in previous call, if they were not completed they need to be presented again to the memory.

You should be aware of these ordering concerns when using ArbitratedScratchpad:
- Write requests to the same address on different lanes may not occur in the order that they are presented to the model due to the arbitration and queuing inside the model.
- Read and write requests to the same address on different lanes may not occur in the order that they are presented to the model due to the arbitration and queuing inside the model.
- In the two scenarios above, in all cases, the pre-HLS and post-HLS ArbitratedScratchpad designs will always behave identically (i.e. there will not be any differences).

These ordering issues are inherent to banked memories like this which support arbitration and queueing. They can be properly handled by appropriate synchronization and coordination at higher levels in the design. In example 42*, the testbench performs all writes to independent addresses, then flushes the writes (by sending several empty write requests), then proceeds to only issue read requests. This is one possible strategy for insuring expected behavior with this model.

In summary, ArbitratedScratchpad is a very powerful and capable model, however it does add additional design considerations and additional HW beyond Scratchpad. When considering using ArbitratedScratchpad, you should always also consider if your design requirements can be changed such that Scratchpad can be used instead.

What about the Catapult BLOCK_SIZE directive?

Scratchpad and ArbitratedScratchpad effectively implement the Catapult INTERLEAVE directive.
If you instead want the higher order address bits to determine bank selection, simply use ac_int::get_slc<> and ac_int::set_slc<> to form a new address with the bank selection bits located as the low order bits in a transformed address before presenting this transformed to Scratchpad and ArbitratedScratchpad.

What about the Catapult WORD_WIDTH directive?

We do not recommend using the WORD_WIDTH directive within Catapult Matchlib models. Instead, use the ScratchpadClass model described above. In this case, the Scratchpad word_type should be the same as the data type used for the original array accesses in the pre-HLS model. The Scratchpad num_banks parameter should be: WORD_WIDTH / number_of_bits(word_type) , where WORD_WIDTH is the number of bits that would have been used in the Catapult WORD_WIDTH directive.