Fixing Design Stuttering Problems in Catapult HLS

Stuart Swan
Platform Architect
Siemens EDA
8 June 2023

Introduction

This document describes Matchlib example 73_stutter_fix, which is an example of a "stuttering" issue that can arise when a process has multiple message-passing interfaces (or "MIOs") that have external communication paths between them that have latency. The term "stuttering" refers to the problem where a design is intended to process transactions at an optimal rate (e.g. II=1), but the actual design is unable to meet that rate and instead stutters.

Note that there is also an example and discussion of design stuttering in example 72* in the Matchlib examples kit. That example focuses on the use of the Matchlib connections back-annotation feature to enable the stuttering behavior seen in the post-HLS simulation to also be seen in the pre-HLS simulation. In contrast, the focus of this example is on resolving stuttering behavior in the RTL by appropriate use of HLS directives.

It should be noted that the general issues and the solutions presented here are all applicable if the design were instead created entirely in hand-written RTL. However, the discussion here is specific to the usage of HLS.

Description of Design

The example design DUT has three inputs and one output. All are 32 bit unsigned integers. The design computes:

```
out1 = square_root(in1 * in2) + mem[in3];
```

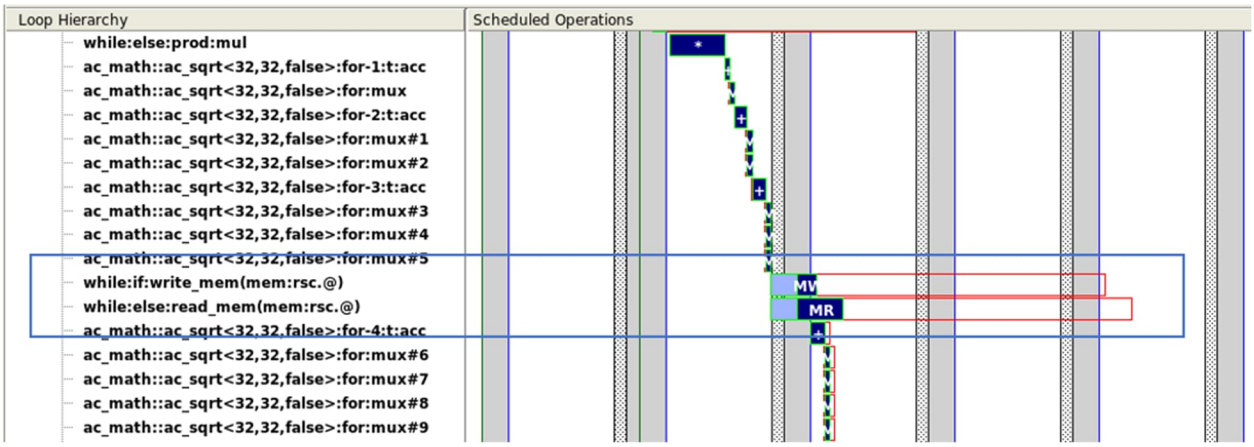
In the default case, "mem" is a C array that is mapped to a RAM that is local to the DUT. Using the setting USE_AXI_MEM, "mem" can be mapped to an external RAM that is instead accessed over an AXI4 master bus interface.

The testbench first writes 10 values to the first 10 locations of the memory, then it reads those locations and produces outputs from the DUT according to the equation above.

The DUT is pipelined with an II=1, and it should be noted that the square root operation necessarily has a latency of about 5 clock cycles. The overall design throughput goal is that when the design is producing outputs, a new output should be produced on every clock cycle.

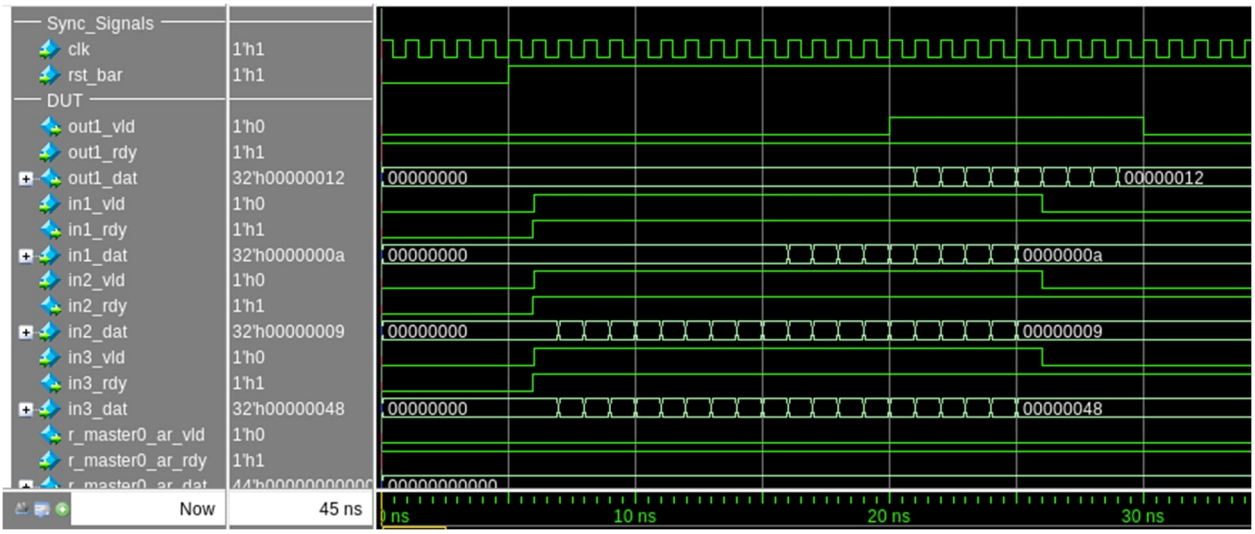
Catapult Run #1

For the first run of Catapult, we map the mem array to a local RAM. This is done by setting use_axi_mem and use_cycle_set to zero in the "go_hls.tcl" script. When we view the RAM read and write operations in the Catapult schedule view, we see:



The blue MR box represents the memory read operation, and the outlined red box represents the scheduling mobility of that operation. In this case, since the operation is known to be a RAM read operation, Catapult knows the exact throughput and fixed latency of the operation. The latency in this case is 1 clock cycle, but even if the RAM was pipelined and had a latency of more than 1 clock cycle, Catapult would be able to schedule the RAM read and write operations and still achieve an II=1 for the overall design since it knows the fixed latency and throughput of the individual RAM operations.

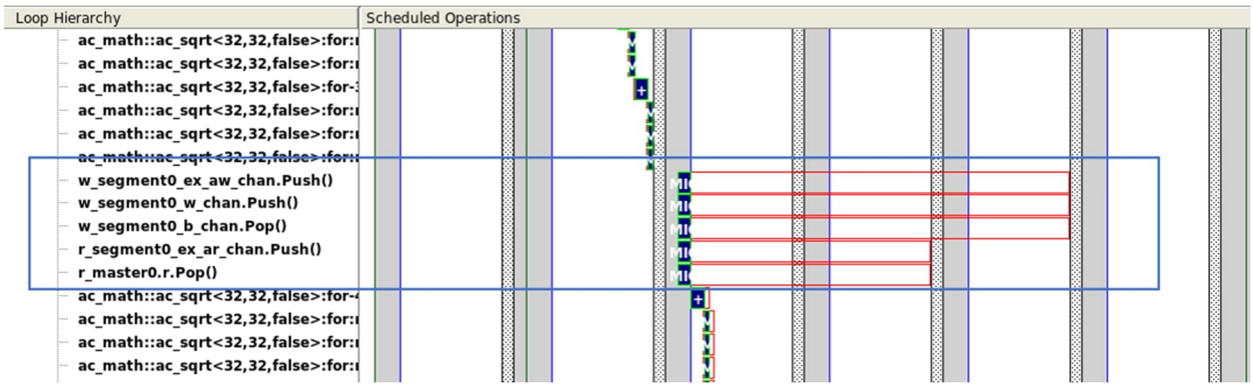
When we look at the RTL waveforms, we see that optimal throughput (II=1) is achieved for both the write operations to the design (to initialize the RAM), and for the read operations from the RAM (which are used to produce design outputs):



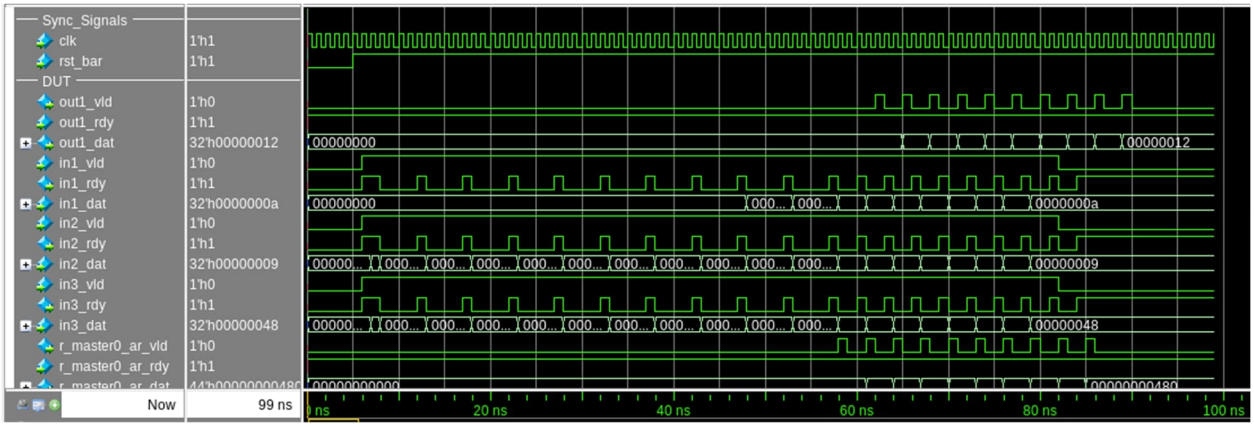
Catapult Run #2

In this run of Catapult, we map the memory to an external RAM that is accessed by the DUT over an AXI4 master bus interface. This is done by setting use_axi_mem to one and use_cycle_set to zero in the "go_hls.tcl" script. AXI4 has 5 channels (aw,w,b,ar,r), each of which is either written or read using Matchlib Push/Pop operations. These operations are message passing operations (or "MIOs") and are independently scheduled by Catapult. In this run of Catapult we do not specify any scheduling directives

related to these MIOs, and when we run Catapult we see in the schedule view that they are all scheduled in the same clock cycle:



When we run the post-HLS RTL we see the following waveforms:



Our focus is on the output of the DUT, which is the out1_dat signal. We can see that a new output is only being produced every three clock cycles, in contrast to the previous Catapult run, where a new output was produced every clock cycle. This is because the design is stuttering. There is a two cycle latency between when the AXI4 address read (or "ar") transaction is pushed by the DUT, and when the corresponding read (or "r") transaction is popped by the DUT. This latency exists because it takes a few clock cycles for the "ar" read request to propagate thru the AXI4 transactor and be serviced by the external RAM model.

The stuttering arises because the MIO operations for the "ar" and "r" transactions were scheduled by Catapult in the same clock cycle, since Catapult by default has no knowledge of their latency relationships arising due to HW outside of the current process. So, the state machine in the RTL must stall the pipeline waiting for the "r" transaction to be received, because it was constructed on the assumption that the "r" transaction could be popped in the same cycle that the "ar" transaction was pushed. In constrast, in Catapult run #1 above, the array was mapped to a local RAM and Catapult knew that the read data would always be ready exactly one cycle after the read address was sent.

(Note that usage of AXI4 interfaces in this example is done in a way to highlight the stuttering problem. In this example all write and read requests only send a single data item, or "beat" in AXI4 terminology. In

typical designs using AXI4, burst read and write transactions would be used, and this would also mostly resolve stuttering problems.)

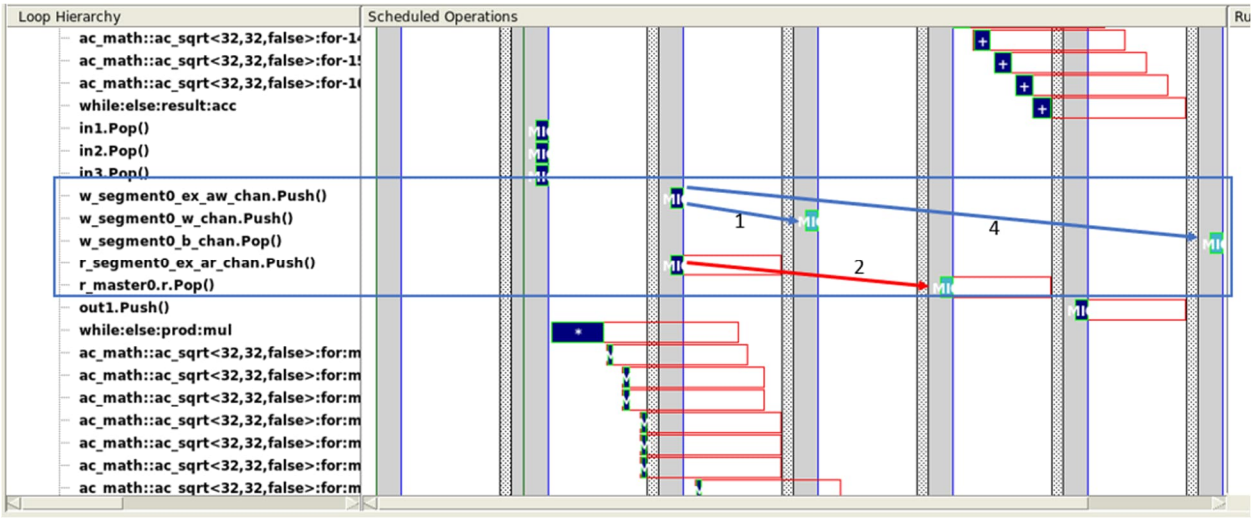
Catapult Run #3

In this run of Catapult, we again map the memory to an external RAM that is accessed by the DUT over an AXI4 master bus interface, and we use Catapult directives to schedule the AXI4 message passing operations more optimally so that the stuttering problem is resolved. This is done by setting use_axi_mem to one and use_cycle_set to one in the "go_hls.tcl" script.

The scheduling directives that are used are:

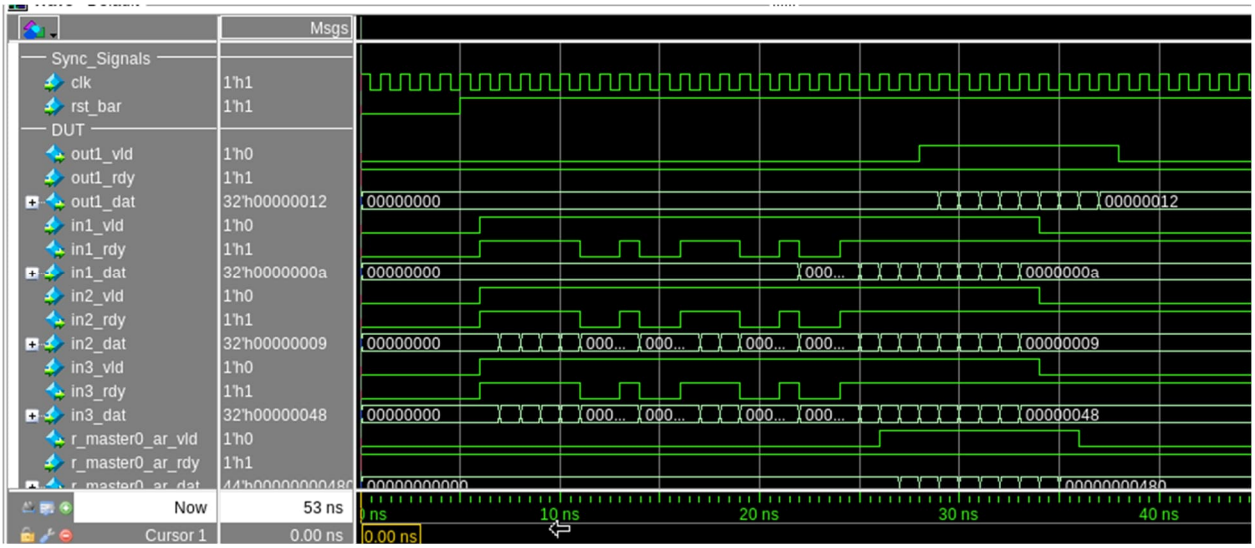
```
cycle set r_master0.r.Pop() -from r_segment0_ex_ar_chan.Push() -equal 2
cycle set w_segment0_w_chan.Push() -from w_segment0_ex_aw_chan.Push() -equal 1
cycle set w_segment0_b_chan.Pop() -from w_segment0_ex_aw_chan.Push() -equal 4
```

When we view the schedule in Catapult we see:



Note in particular the red arrow that shows the r.Pop() operation now is scheduled 2 cycles later than the ar.Push() operation.

When we view the RTL waveforms, we now see:



Our focus remains on the output of the DUT, which is the out1_dat signal. We now can see that a new output is produced every clock cycle, and that the stuttering problem we saw in run #2 is resolved. By scheduling the r.Pop() operation two cycles after the ar.Push() operation, the pipeline that Catapult constructs is able to never stall waiting for the the r.Pop() to complete, and full throughput is seen in the post-HLS RTL.

Conclusion

Stuttering problems can arise in pre-HLS and post-HLS models when there are reconverging communication paths with unbalanced latencies. The latencies for these paths can be composed of latency internal to processes (e.g. due to loop pipelining during HLS), and also latency external to the process (e.g. due to cycles needed for other processes to execute and respond).

Stuttering problems will typically not result in functional mismatches between the pre-HLS and post-HLS simulations, but it will be apparent that the post-HLS RTL simulation is not meeting its performance targets. By understanding the root cause of stuttering issues, and understanding the techniques used to resolve them, you should be able to resolve stuttering problems efficiently.

The general issues and solutions related to stuttering are common both to designs that use HLS as well as hand-written RTL designs. But resolving stuttering problems when using HLS should be easier than in the hand-written RTL case, since identification of the issues and implementation of the solutions is easier.