HLS Transactor Modeling Guide

Stuart Swan

Platform Architect

Siemens EDA

12 Mar 2025

Introduction

This document walks through three different approaches to modeling transactors for custom protocols for use in HLS models. We use a very simple but representative custom protocol and discuss some of the pros and cons of the three different approaches.

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/53_transactor_modeling

Protocol Used In this Tutorial

The protocol used here is very simple but intended to illustrate common issues in modeling real-world protocols that are not simple message passing protocols. (Simple message passing protocols are easily layered on top of the existing Matchlib Connections classes).

The protocol is called "hilo". The structural models and transactors are all defined in the hilo.h file.

The "hilo" protocol is the same as the basic rdy/vld/dat protcol with the following differences:

- The "dat" signal is an sc_uint<8>
- When rdy & vld are both true, the high byte is transferred.
- In the very next clock cycle, rdy and vld are driven low and the low byte is transferred.
- Each call to the transactor thus transfers a sc_uint<16> message.

The hilo protocol channels and ports, which are common to all the different transactor modeling approaches, are:

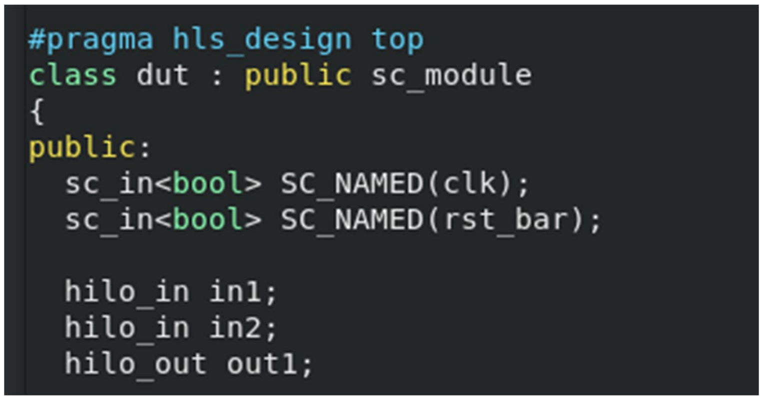
```
// The hilo protocol channel. This channel is the same for all transactor styles

struct hilo_chan : public sc_channel {
  hilo_chan(sc_module_name nm) : sc_channel(nm) {}

  sc_signal<bool> SC_NAMED(rdy);
  sc_signal<bool> SC_NAMED(vld);
  sc_signal<sc_uint<8>> SC_NAMED(dat);
};
```

```
// The hilo protocol input port. This port is the same for all transactor styles

struct hilo_in {
  hilo_in(const char* nm = "") {}

  sc_in<bool> vld;
  sc_out<bool> rdy;
  sc_in<sc_uint<8>> dat;

  template <class C>
  void operator()(C& c) {
     vld(c.vld);
     dat(c.dat);
     rdy(c.rdy);
  }
};

// The hilo protocol output port. This port is the same for all transactor styles

struct hilo_out {
  hilo_out(const char* nm = "") {}

  sc_out<bool> vld;
  sc_in<bool> rdy;
  sc_out<sc_uint<8>> dat;

  template <class C>
  void operator()(C& c) {
     vld(c.vld);
     dat(c.dat);
     rdy(c.rdy);
  }
};
```

Because the hilo transaction payload is 16 bits, but the channel only has an 8 bit dat signal, it takes exactly two clock cycles to transfer a 16 bit transaction payload.

The DUT (Design Under Test)

The DUT interface is:

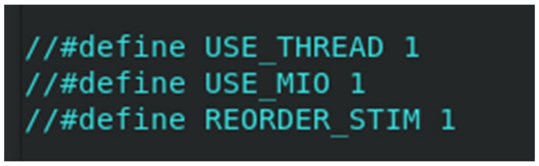


```
#pragma hls_design top
class dut : public sc_module
{
public:
  sc_in<bool> SC_NAMED(clk);
  sc_in<bool> SC_NAMED(rst_bar);

  hilo_in in1;
  hilo_in in2;
  hilo_out out1;
```

The DUT uses the 3 different transactor modeling styles to read the 2 16 bit hilo transaction payloads, add them together, and push out the single 16 bit hilo transaction payload.

The choice of which transactor style to use is controlled by #ifdef statements in dut.h and testbench.cpp. *It should be noted that all three styles are very similar in these files and almost the same number of lines of code.*

These ifdefs are specified at the top of the dut.h file:



```
//#define USE_THREAD 1
//#define USE_MIO 1
//#define REORDER_STIM 1
```

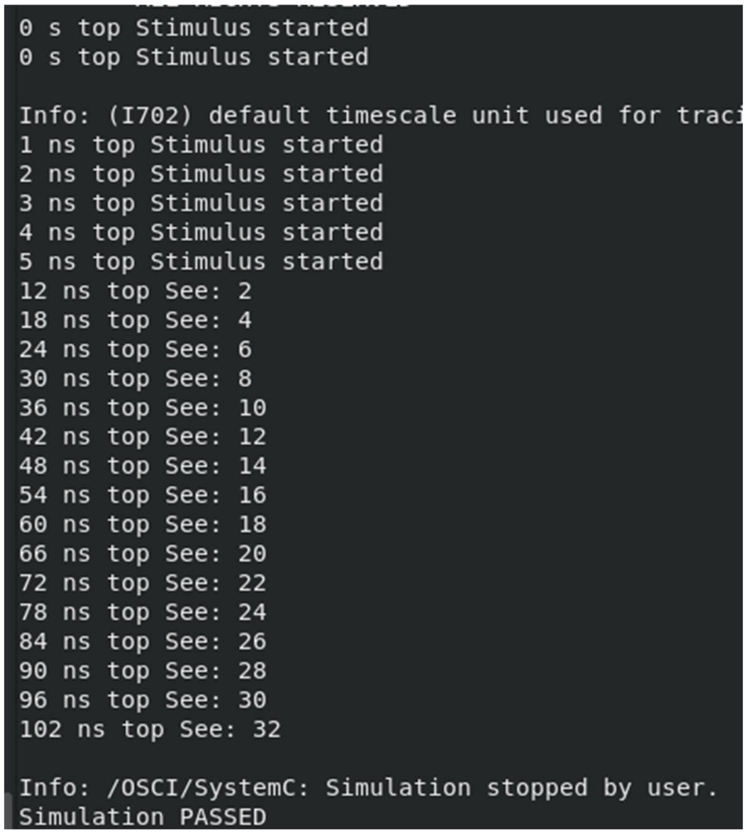
When none of the defines are specified, the DUT and Testbench use simple function transactors (style 1).

Transactor Style 1 – Simple Functions Inlined into Calling Process

The first transactor style uses simple functions that are inlined into the calling process during HLS. Here is a what the code looks like for the transactors:

```
template <class C>
void hilo_reset_read(C& chan) {
  chan.rdy = 0;
}

template <class C>
sc_uint<16> hilo_pop(C& chan) {

  chan.rdy = 1;
  do {
   wait();
  } while (chan.vld.read() == 0);
  chan.rdy = 0;

  sc_uint<8> hi_byte = chan.dat.read();
  wait();
  sc_uint<8> lo_byte = chan.dat.read();
  return ((hi_byte << 8) | lo_byte);
}

template <class C>
void hilo_reset_write(C& chan) {
  chan.vld = 0;
  chan.dat = 0;
}

template <class C>
void hilo_push(C& chan, sc_uint<16> v) {

  chan.vld = 1;
  chan.dat = v >> 8;
  do {
    wait();
  } while (chan.rdy.read() == 0);
  chan.vld = 0;

  chan.dat = v;
  wait();
}
```

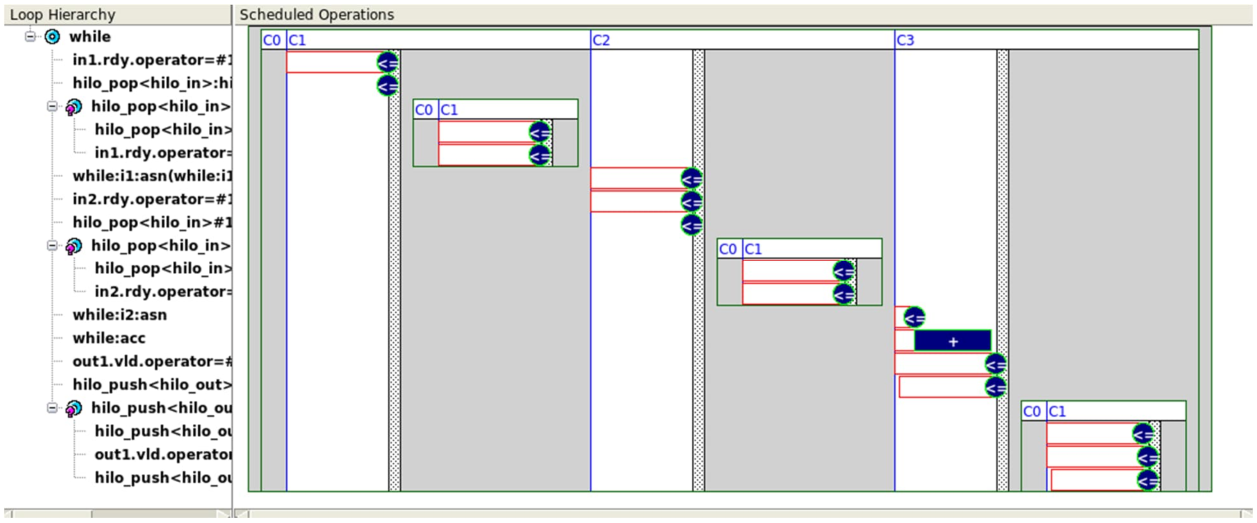
The testbench has a stimulus thread that counts up from 0, and pushes the count to each input of the DUT. When the design is run, it prints:



```
0 s top Stimulus started
0 s top Stimulus started

Info: (I702) default timescale unit used for traci
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
12 ns top See: 2
18 ns top See: 4
24 ns top See: 6
30 ns top See: 8
36 ns top See: 10
42 ns top See: 12
48 ns top See: 14
54 ns top See: 16
60 ns top See: 18
66 ns top See: 20
72 ns top See: 22
78 ns top See: 24
84 ns top See: 26
90 ns top See: 28
96 ns top See: 30
102 ns top See: 32

Info: /OSCI/SystemC: Simulation stopped by user.
Simulation PASSED
```

The transactors used in style 1 cannot be pipelined with an II=1 in Catapult, so any pipeline directives must be disabled for HLS to succeed.

When we synthesize and look at the schedule, we see:



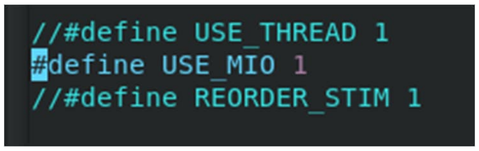
We can see that the 2 pop operations and the 1 push operation require their own states and loops. This is because in both the pre-HLS simulation, and in the post-HLS model (and simulation), the sequential order of the wait statements in the DUT process is being strictly preserved.
When we run the post-HLS model we see:

```
# 1 ns sc_main/top Stimulus started
# 2 ns sc_main/top Stimulus started
# 3 ns sc_main/top Stimulus started
# 4 ns sc_main/top Stimulus started
# 5 ns sc_main/top Stimulus started
# 12 ns sc_main/top See: 2
# 18 ns sc_main/top See: 4
# 24 ns sc_main/top See: 6
# 30 ns sc_main/top See: 8
# 36 ns sc_main/top See: 10
# 42 ns sc_main/top See: 12
# 48 ns sc_main/top See: 14
# 54 ns sc_main/top See: 16
# 60 ns sc_main/top See: 18
# 66 ns sc_main/top See: 20
# 72 ns sc_main/top See: 22
# 78 ns sc_main/top See: 24
# 84 ns sc_main/top See: 26
# 90 ns sc_main/top See: 28
# 96 ns sc_main/top See: 30
# 102 ns sc_main/top See: 32
```

Note that the log exactly matches the pre-HLS log, and that both have a gap of 6 cycles between each output.

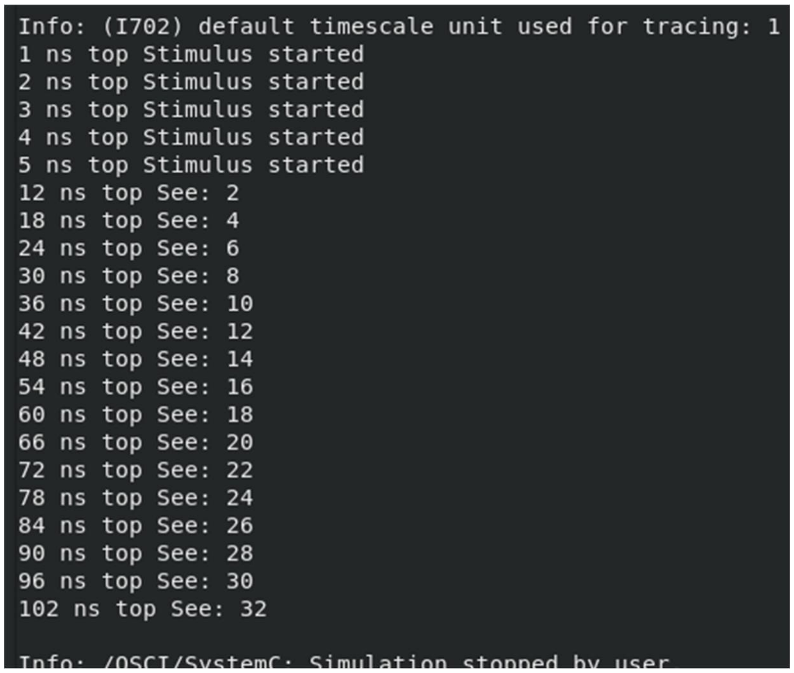
Transactor Style 2 – Transactors using "pragma modulario" functions

The second transactor style uses functions that use “pragma modulario”. This style is enabled with the following in dut.h:



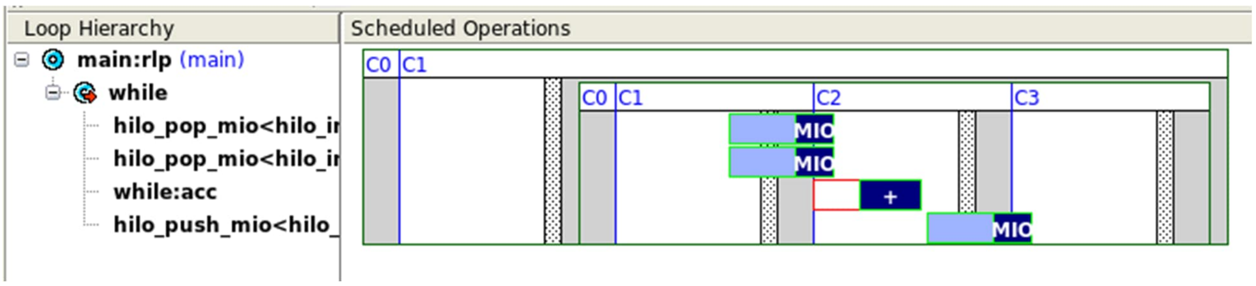
```
//#define USE_THREAD 1
#define USE_MIO 1
//#define REORDER_STIM 1
```

These functions are quite like style 1. In the pre-HLS simulation, they will behave identically to the style 1 transactors. The pre-HLS looks the same as before, with a 6-cycle gap between outputs:



```
Info: (I702) default timescale unit used for tracing: 1
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
12 ns top See: 2
18 ns top See: 4
24 ns top See: 6
30 ns top See: 8
36 ns top See: 10
42 ns top See: 12
48 ns top See: 14
54 ns top See: 16
60 ns top See: 18
66 ns top See: 20
72 ns top See: 22
78 ns top See: 24
84 ns top See: 26
90 ns top See: 28
96 ns top See: 30
102 ns top See: 32

Info: /OSCI/SystemC: Simulation stopped by user
```

During HLS, Catapult can schedule invocations of the push and pop operations into the same clock cycle. With this style, Catapult can pipeline the DUT inner loop with an II=1. (It is likely that this style is like the way "protocol regions" are scheduled.) When we view the schedule, we see:



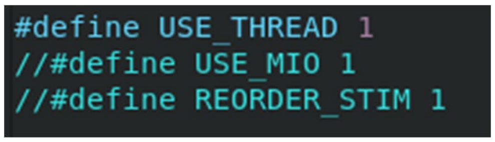
When we run the post-HLS simulation, we see:

```
# 1 ns sc_main/top Stimulus started
# 2 ns sc_main/top Stimulus started
# 3 ns sc_main/top Stimulus started
# 4 ns sc_main/top Stimulus started
# 5 ns sc_main/top Stimulus started
# 13 ns sc_main/top See: 2
# 17 ns sc_main/top See: 4
# 21 ns sc_main/top See: 6
# 25 ns sc_main/top See: 8
# 29 ns sc_main/top See: 10
# 33 ns sc_main/top See: 12
# 37 ns sc_main/top See: 14
# 41 ns sc_main/top See: 16
# 45 ns sc_main/top See: 18
# 49 ns sc_main/top See: 20
# 53 ns sc_main/top See: 22
# 57 ns sc_main/top See: 24
# 61 ns sc_main/top See: 26
# 65 ns sc_main/top See: 28
# 69 ns sc_main/top See: 30
# 73 ns sc_main/top See: 32
```

We see that a new transaction is output every 4 clock cycles. The DUT inputs are capable of being read in parallel (as seen in the schedule), however the testbench stimulus thread needs to push input one (which takes 2 cycles) and then push input two. So, the sequential nature of the transactor implementation for style 2 limits our ability to achieve a "throughput accurate" model of the system. In the pre-HLS simulation, there is a 6-cycle gap, in the post-HLS simulation there is a 4-cycle gap, and in a truly throughput accurate model there would need to be a 2-cycle gap (soley due to the limits of the hilo HW protocol).

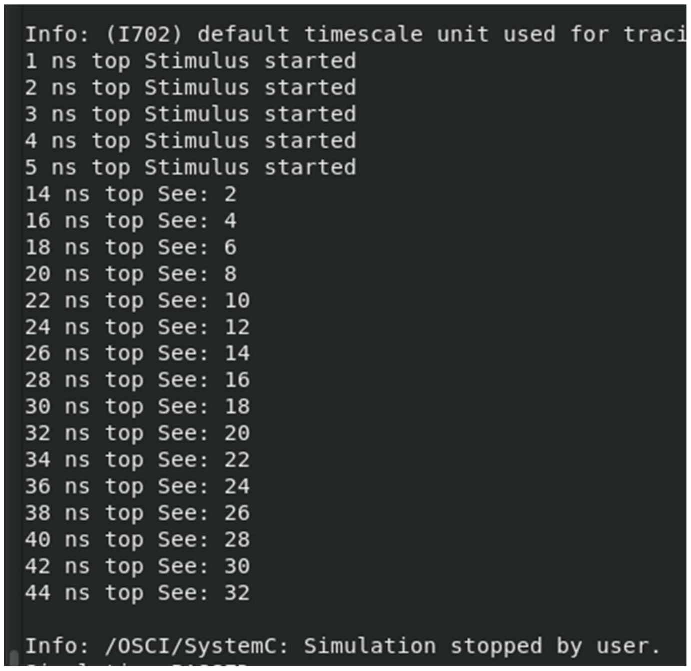
Transactor Style 3: Transactors modeled as separate sc_modules and SC_THREADs

The third transactor style uses transactors modeled as separate sc_modules with their own SC_THREAD. This style is enabled with following #defines at the top of dut.h:



```
#define USE_THREAD 1
//#define USE_MIO 1
//#define REORDER_STIM 1
```

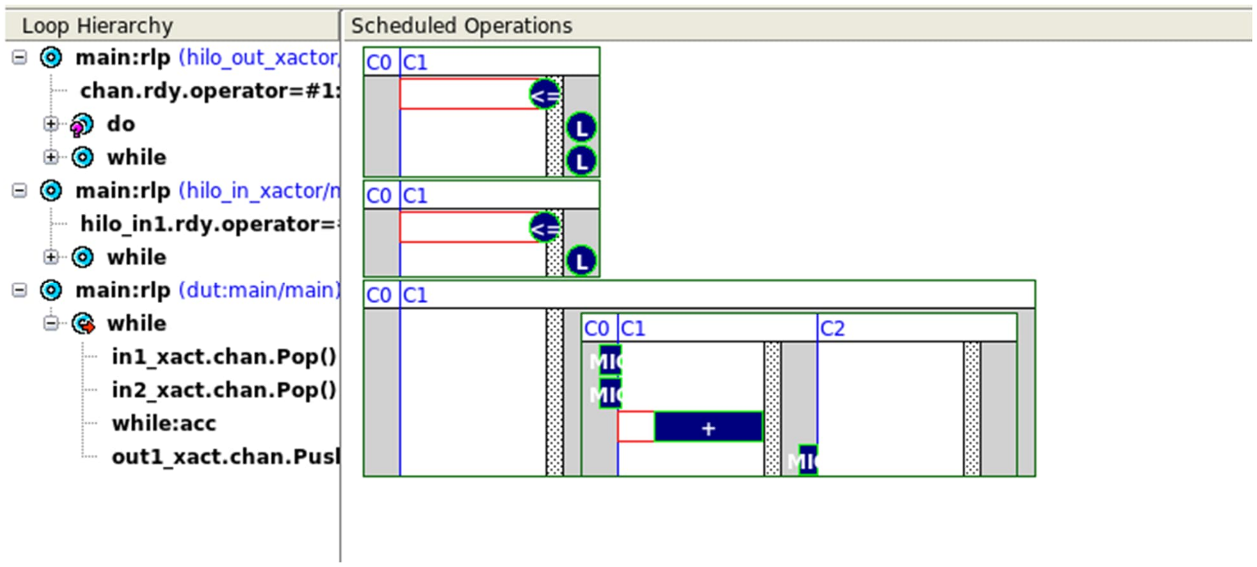
When this style is used the pre-HLS simulation log is:



```
Info: (I702) default timescale unit used for traci
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
14 ns top See: 2
16 ns top See: 4
18 ns top See: 6
20 ns top See: 8
22 ns top See: 10
24 ns top See: 12
26 ns top See: 14
28 ns top See: 16
30 ns top See: 18
32 ns top See: 20
34 ns top See: 22
36 ns top See: 24
38 ns top See: 26
40 ns top See: 28
42 ns top See: 30
44 ns top See: 32

Info: /OSCI/SystemC: Simulation stopped by user.
```

Note that there is a 2-cycle gap, and that the pre-HLS simulation is now throughput accurate – it matches what the actual HW can achieve in terms of performance.

When we look at the schedule in Catapult we see:



We see that Catapult can pipeline the loop with an II=1 and schedule the Pop and Push operations in parallel.

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

```
# 0 s sc_main/top Stimulus started
# 1 ns sc_main/top Stimulus started
# 2 ns sc_main/top Stimulus started
# 3 ns sc_main/top Stimulus started
# 4 ns sc_main/top Stimulus started
# 5 ns sc_main/top Stimulus started
# 14 ns sc_main/top See: 2
# 16 ns sc_main/top See: 4
# 18 ns sc_main/top See: 6
# 20 ns sc_main/top See: 8
# 22 ns sc_main/top See: 10
# 24 ns sc_main/top See: 12
# 26 ns sc_main/top See: 14
# 28 ns sc_main/top See: 16
# 30 ns sc_main/top See: 18
# 32 ns sc_main/top See: 20
# 34 ns sc_main/top See: 22
# 36 ns sc_main/top See: 24
# 38 ns sc_main/top See: 26
# 40 ns sc_main/top See: 28
# 42 ns sc_main/top See: 30
# 44 ns sc_main/top See: 32
# ** Note: (vsim-6574) SystemC simulation stopped by user.
```

Note that the post-HLS simulation is throughput accurate and matches what the actual HW can achieve in terms of performance. Remember that the testbench being used in this case is the same as the testbench being used in the post-HLS simulation for style 2. In the latter case, the style 2 transactor usage within the testbench made the post-HLS simulation inaccurate.

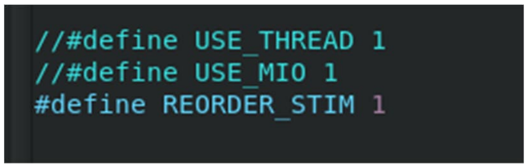
In summary, with the style 3 transactors, both the pre-HLS and post-HLS simulations are "throughput accurate", while with style 1 and style 2 transactors this never occurs.

Issues with Transaction Ordering

Realistic HW designs are highly concurrent and have complex flows of transactions between blocks. It is important to be able to accurately model and stress test pre-HLS models so that most issues can be identified early. Debugging of HLS-generated RTL models should generally be avoided if possible.

To stress test a pre-HLS model, the same techniques used to stress test RTL models are used, including random stimulus, functional and code coverage, etc. A common stress test to apply is to randomly delay or change the order of arrival of transactions to the DUT, since this is typically something that designs must tolerate while still operating correctly.

At the top of the dut.h file, there is the REORDER_STIM flag, which we now enable:



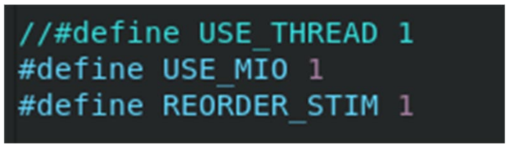
```
//#define USE_THREAD 1
//#define USE_MIO 1
#define REORDER_STIM 1
```

When this #define is set, the testbench on the 7th transaction will reverse the order of the two Push calls that it performs to push the 2 inputs to the DUT.

For the Style 1 transactors, with the #define flags set as above, the pre-HLS simulation will now hang when the 7th transaction occurs. The reason the simulation hangs is because the DUT is modeled with sequential transactor code that tries to Pop input 1 first. Since the TB is pushing input 2 first (on the 7th transaction), no progress in either the DUT or TB can be made, and the simulation simply hangs.

When we run the post-HLS simulation for Style 1 the same hang will occur for the same reason.

Now we try the same thing with the Style 2 transactors:

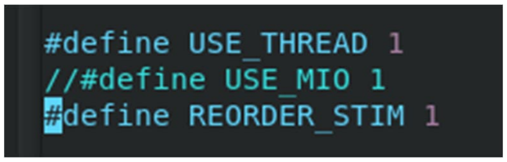


```
//#define USE_THREAD 1
#define USE_MIO 1
#define REORDER_STIM 1
```

When we run the pre-HLS simulation, the simulation hangs as before, for the same reasons.

When we run the post-HLS simulation, the simulation does not hang. This is because even if the TB changes the order of the 2 inputs to the DUT, the DUT RTL can accept them in a concurrent manner, in whatever order they arrive.

Now we try the same thing with the Style 3 transactors:



```
#define USE_THREAD 1
//#define USE_MIO 1
#define REORDER_STIM 1
```

When we run the pre-HLS simulation, the simulation does not hang. Everything works properly. This is because the transactors used for the 2 Push operations in the testbench support full concurrency.

When we run the post-HLS simulation, the simulation does not hang. This is because even if the TB changes the order of the 2 inputs to the DUT, the DUT RTL can accept them in a concurrent manner, in whatever order they arrive.

In summary, with the style 3 transactors, transactions can be reordered to reflect what is allowable in the real HW, and neither the pre-HLS nor post-HLS simulation will hang. However, with the style 1 and style 2 transactors, simulation hangs may occur.

Keep in mind that this is a trivial design and testbench, so it may not be too difficult to debug why the simulation hangs occur when using the style 1 or style 2 transactors. In larger, more complex systems with complex transaction flows between blocks, such simulation hangs can be much more difficult to debug.

Transactor Modeling Guidance

Style 3 is the preferred approach for modeling HLS transactors, since it enables throughput accurate modeling in the pre-HLS and post-HLS simulations, and completely avoids any simulation hang issues.

Style 1 is an acceptable approach for modeling transactors if concurrent invocation of the transactor is never needed by a process in either the DUT or testbench. With Style 1, each transactor invocation is strictly sequential in the pre-HLS and post-HLS models, and the user should have no expectation of concurrent execution.

Style 2 (“pragma modulario”) is a deprecated modeling approach in Catapult. It does not enable throughput accurate modeling, and while it enables concurrent invocation of transactor calls in the RTL, it does not enable it in the pre-HLS model, leading to the simulation hang problem.

Regarding QOR differences between the various styles:

Style 1 isn’t comparable in performance with the other styles, but the area difference it has with the other approaches is very small.

To accurately compare Style 2 and Style 3, the Style 3 transactors must be used in the testbench even when the Style 2 transactors are used in the DUT. When this is done, the Style 2 overall DUT and TB achieves “throughput accuracy” and full performance (like full use of Style 3) in the post-HLS RTL, but still not in the pre-HLS model. In terms of area, the differences between the Style 2 DUT and the Style 3 DUT are very small, and any differences would be even smaller on realistic designs.