Catapult HLS User View Scheduling Rules
Stuart Swan, Platform Architect, Siemens EDA
13 Jan 2022

This document describes the user view scheduling rules for C++ and SystemC models that are synthesized through Catapult HLS. Some of the goals for this document are:

- fairly easy to explain to HLS users
- precise and consistent rules in SystemC and C++
- in practice, not different from how Catapult works today
- QOR as good or better than today
- covers all known user requirements/scenarios
- if needed, provide a suitable starting point for a standardization proposal (e.g. in Accellera SWG)

To more specifically illustrate the goals of this document, consider what an engineer writing a testbench for an HLS model needs to understand about how HLS tools operate. This engineer may be using SV UVM, or may be writing a testbench in C++/SystemC. He likely is not an expert in any particular HLS tool (and may not want to be), but his testbench needs to work for both the pre-HLS model as well as the post-HLS model. Thus, the DV engineer needs to have a precise understanding of how the HLS tool will transform the design while still enabling it to be fully verified. This document describes what transformations the HLS tool is allowed to perform so that the pre-HLS and post-HLS models can be effectively verified with the same testbench. The overarching philosophy of the scheduling rules is to present “no surprises” to such a DV engineer, while still allowing the HLS tool sufficient freedom to optimize the design.

Background

HLS tools generate RTL from C++ models. In a general sense, this involves taking a sequential C++ model and transforming it into concurrent hardware with equivalent behavior. HLS tools identify concurrent processes within the C++/SystemC model and then independently synthesize each process. Briefly, some of the techniques that HLS tools use to achieve good HW QOR when synthesizing each process include:

- Optimized scheduling based on the selected silicon target technology
- Automatic HW pipeline construction according to the user’s specifications
- Automatic HW resource sharing
- Automatic scheduling of memory accesses

The internal behavior of the process is specified by the control and dataflow behavior of the C++ code within the process. However, the external communication that the process has with other processes and HW blocks is specified via IO operations that are coded within the model. To enable a reliable, scalable, and verifiable HLS flow that generates high quality hardware, the scheduling behavior of these IO operations needs to be precisely handled at all steps of the flow. This document specifies the rules that govern the scheduling behavior for these IO operations within HLS models.

This document only covers sequential HW processes. Combinational HW processes are not covered since their synthesis is straightforward.

This document distinguishes between the following:

1. The "conceptual model" for the scheduling rules

2. The simulation behavior of a model using the rules in C++ or SystemC

3. The synthesis of a C++/SystemC model using the rules in an HLS tool such as Catapult

The goal is to align each of these three cases as closely as possible, so that the user has easy to understand rules, while simulation and synthesis work without surprises. However, as we will see, there are practical considerations which may in some cases cause small deviations from the conceptual model in certain cases in either simulation or HLS.

An Analogy from RTL Synthesis

To better understand the specific purpose of this document, let's consider how RTL synthesis works. Say you have a sequential block that you are modeling in Verilog RTL, and it has an output port coded like:

```
Out1 <= new_val # some_delay;
```

In Verilog simulation, if some_delay is less than the clock period of the block, then it will probably not affect the overall cycle level behavior of the system during simulation. However, if some_delay is more than the clock period, it probably will.

During RTL synthesis, all RTL synthesis tools will ignore all delays in the input model, in this case even if some_delay is greater than the clock period. Some RTL synthesis tools might give a warning for the code above similar to "Simulation and synthesis results are likely to mismatch because delay in model is greater than clock period."

One might argue that RTL synthesis tools should always match the Verilog simulation behavior of the input model. But, the overall approach works well because RTL is a good and simple "conceptual model" that users and tool vendors can align around. The slight differences between the "simulation model" and the "conceptual model" used by RTL synthesis tools can be fairly easily managed.

We'll return to this example later in this document.

Catapult HLS Status Concerning Rules in this Document

At the end of this document there is a numbered list of clarifications on the current Catapult HLS status concerning the rules outlined in this document. Within this document rules which have associated clarifications in that list are annotated with "Cat#", where the number refers to the item in the numbered list at the end of the document.

Terms Used in this Document:

Message Passing Interface: A message passing interface reliably delivers messages (or transactions) from one process to another. This document uses this term to denote the type of communication found in Kahn Process Networks. See https://en.wikipedia.org/wiki/Kahn_process_networks

Synchronization Interface: A synchronization interface synchronizes one process with another and/or with a global clock. For an example of a synchronization interface, see https://en.wikipedia.org/wiki/Barrier_(computer_science)

Signal IO: In digital HW design, signals are the fundamental communication mechanism. Signals enable communication between two HW blocks/processes, but communication with signals in real HW always incurs at least some delay because communication cannot be faster than the speed of light.
In HDLs and in SystemC, signal delays are modeled with the "delayed update" semantics.

"blocking" / "non-blocking": A blocking message passing interface suspends the execution of the calling process until the message is either sent or received. A non-blocking message passing interface never suspends execution of the calling process: instead, a return code is provided to indicate whether the operation completed or not.

"**shall**": This term indicates that a compliant tool or flow is required to follow the indicated rule.

"**may**": This term indicates that a compliant tool or flow is allowed to follow the indicated rule, but is not required to do so.

Classes of operations involved scheduling rules:

There are three classes of operations involved in the scheduling rules:

1. Calls to message passing interfaces (which are all ac_channel methods, all SystemC MIO calls except calls to SyncChannel)
2. Calls to synchronization interfaces (which are calls to ac_sync and Matchlib SyncChannel, also ac_wait and SystemC wait)
3. Signal IO (which are SystemC signal read and writes, also C++ model "direct inputs")

Basic Conceptual Model:

The basic conceptual model encompasses models that have no loop pipelining, but may have preserved loops. If a model has a preserved loop, then the user may place a wait statement in the loop, or else the HLS tool may automatically add one into the body of the loop. For a preserved loop the HLS tool may also add a wait statement at the loop termination. Such automatically added wait statements are called "implicit user wait statements" in this document. Wait statements explicitly placed in the model are called "explicit user wait statements" in this document.

Note that both "implicit user wait statements" and "explicit user wait statements" are classified as calls to a "synchronization interface".

The "conceptual model" scheduling rules are:

1. Synchronization interface calls within a process always remain in the source code order.

2. Signal read operations occur at the closest preceding call to a synchronization interface. (Cat1)

3. Signal write operations occur at the closest succeeding call to a synchronization interface.

4. Message passing operations are free to be reordered subject to the following constraints:
- All message passing operations before a call to a synchronization interface shall be completed when the call to the synchronization interface has completed.
- All message passing operations after a call to a synchronization interface shall not start until the call to the synchronization interface has completed.
- Two message passing operations which appear in sequence in the model may be executed either in the same sequence or in parallel in simulation and in synthesis, but they shall not be executed in the reverse sequence. (Cat2)

Some explanation for the very last point: in pure message passing models (ie KPN systems), all fifo sizes are unbounded, and calls to message passing interfaces can be reordered and the overall system behavior will not change (ie it will still be deterministic). However, real HW systems cannot have unbounded fifo sizes, and in practice these fifo sizes need to be kept as small as possible to save area. When fifo sizes are bounded and message passing interface call sequences are reordered by HLS tools, then it is possible that system deadlock cases may be introduced. The last rule insures that HLS tools cannot introduce such deadlock cases.

Conceptual Model for Pipelined Loops

When a loop is pipelined in HLS, the body of the loop is split into pipeline stages. HLS may start the next iteration of the loop before the current iteration has completed.

When a loop is pipelined, the user may place wait statements in the body of the loop to manually separate operations into their respective pipeline stages. Alternatively the HLS tool may implicitly add these wait statements into the model to separate the pipeline stages. We call the former "explicit user pipeline stage wait statements, and the latter "implicit user pipeline stage wait statements".

Both explicit and implicit user pipeline stage wait statements are classified as calls to a "synchronization interface".

The scheduling rules for pipelined loops are the same as the scheduling rules given in the "basic conceptual model", with the addition of these pipeline wait statements into the set of calls to synchronization interfaces.

When a loop is pipelined, multiple iterations of the loop are overlapped and execute at the same time. During loop pipelining, for all IO operations, HLS shall insure that an access to a message passing interface, signal, or synchronization interface shall not be moved over or in parallel with an access to the same interface from a different loop iteration.

Direct Inputs

Normal SystemC signal read operations occur at the closest preceding synchronization interface call (e.g. wait statement) in both the pre-HLS and post-HLS models. (Cat1). If the HLS tool adds states to the

design, or if it pipelines the design, then this implies that the HLS tool must add registers for each such read operation such that the read occurs where specified in the pre-HLS model, and the value is stored until the point where it is consumed in the post-HLS model. The area cost of such registers may be fairly high if there are a lot of signals, and in some cases it may be unneeded area since the value of the signals may not actually need to be stored internally to the design.

The simplest case is if such external signals are held stable after the design comes out of reset. In this case, HLS may assume that it is free to read the signal values as late as possible, with no need for register storage. This case is handled with the following pragma on SystemC signals and ports (Cat6):

#pragma hls_direct_input

To ensure that there are no pre-HLS versus post-HLS simulation mismatches, the environment that drives the signal shall hold it stable after all receiving processes that use this signal with this pragma come out of reset.

A related but somewhat more complex case is where input signals to the HW block may only be changed at “agreed upon” times, typically while the portion of the HW block that relies on them is temporarily idle. For example, a block may process 2D images. At the start of each new image, it may be desirable for the TB or external environment to update the control signals for how the block will process the next image. This needs to be done precisely since typically HLS designs are pipelined, and the HW pipeline for the current iteration must be fully “ramped down” before the input signals can be updated to affect the next iteration. In this case we can use the SystemC “SyncChannel” or C++ “ac_sync” primitives to precisely synchronize the DUT with the TB/environment to enable the input signals to be updated at the correct time. The #pragma hls_direct_input_sync directive shown below associates the sync operation with the direct inputs that it controls. (Cat7) The precise synchronization scheme shown here ensures that there are no pre-HLS versus post-HLS simulation mismatches even though we are using direct inputs and also changing their values while the design is executing.

```
  // This is example 61* in Catapult Matchlib examples
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst_bar);

  Connections::Out<uint32>  CCS_INIT_S1(out1);
  Connections::In <uint32>  sample_in[num_samples];
  Connections::SyncIn       CCS_INIT_S1(sync_in);
#pragma hls_direct_input
  sc_in<uint32_t> direct_inputs[num_direct_inputs];


  void main() {
    out1.Reset();
    sync_in.Reset();

#pragma hls_unroll yes
    for (int i=0; i < num_samples; i++) {
      sample_in[i].Reset();
    }

    wait();  // reset state

    while (1) {
#pragma hls_direct_input_sync all
     sync_in.sync_in();
```

```
#pragma hls_pipeline_init_interval 1
#pragma hls_stall_mode flush
     for (uint32_t x=0; x < direct_inputs[0]; x++) {
       for (uint32_t y=0; y < direct_inputs[1]; y++) {
        uint32_t sum = 0;
#pragma hls_unroll yes
        for (uint32_t s=0; s < num_samples; s++) {
           sum += sample_in[s].Pop() * direct_inputs[2 + s];
        }
        ac_int<32, false> ac_sum = sum;
        ac_int<32, false> sqrt = 0;
        ac_math::ac_sqrt(ac_sum, sqrt);  // internal loop is unrolled in catapult .tcl
file
        if (sqrt > direct_inputs[7])
          out1.Push(sqrt);
       }
     }
    }
  }
};
```

From the perspective of the testbench or the environment, the updating of the signals controlled by the hls_direct_input_sync directive needs to occur at a precise point. The TB needs to first wait for the “rdy” signal for the sync to be asserted by the DUT, and then the TB must update all the input signals it wishes to change while simultaneously driving the sync “vld” signal high for one cycle.

It is important to note that the only safe operation to use to synchronize the updating of direct inputs is the “sync” operation as shown above. Other operations such as Push/Pop or ac_channel operations should not be used for this.

In the example above the DUT block that is being synthesized determines when to call sync, and thus it determines when the direct inputs will be updated. In some cases it may be necessary for the environment around the DUT to determine when the direct updates should be updated. In this case the same approach as shown above should be used, however a separate input from the environment to the DUT (either using a signal or a message passing interface) should request that the DUT call sync as soon as feasible. This will ensure that the DUT has properly ramped down its pipeline and is ready to receive the newly updated direct inputs as per the overall synchronization scheme described above.

TODO: hls_direct_input_sync example in C++ flow

Additional Options for Scheduling Message Passing Interfaces

The following option may be added during HLS (Cat2):
STRICT_IO_SCHEDULING=relaxed

when this is specified, the HLS tool is allowed to reorder message passing interface calls freely. However it is still not allowed to move these calls across synchronization interface calls.

Scheduling of Array Accesses

Arrays may appear in HLS models, and they may be preserved through synthesis and mapped to RAMs.

Pointers may also appear in HLS models, and pointer dereferences are resolved to array accesses during HLS.

There are two cases to consider for arrays for the purposes of the scheduling rules:

1. Array instantiation in the HW is internal to the process
   - The array accesses are not visible external to process, and thus their scheduling is also not visible externally.
   - All of the scheduling rules described elsewhere in this document remain unaffected in this case.

2. Array instantiation in the HW is external to process
   - In this case the user model shall indicate how array accesses are mapped onto IO operations that are external to the process.
   - We call this the "array access mapping layer". The array access mapping layer maps array accesses onto IO operations described above (signal IO, message passing interface calls, and synchronization calls).
   - The user model may indicate that it is allowable for HLS to transform array accesses, for example, to cache, merge, split, or reorder array accesses (e.g. to improve QOR). These transformed operations, if allowed, are an outcome from the use of the "array access mapping layer".
   - In all cases the scheduling rules described elsewhere in this document for the core IO operations (signal IO, message passing calls, synchronization calls) remain unaffected.
   - Note that if transformed operations occur and array accesses are visible externally in both pre-HLS and post-HLS model, then comparison of pre-HLS and post-HLS behaviors may need to take into account the transformed operations.

When a loop is pipelined, multiple iterations of the loop are overlapped and execute at the same time. During loop pipelining, an access to a memory interface (or array) may be moved over or in parallel with an access to the same memory if the HLS tool can prove the reordering is conflict free. If the array/memory is external to the process, the “array access mapping layer” shall indicate that such reordering is allowable if such reordering is to occur during loop pipelining.

CCORES and MODULARIO

A user may add the following pragmas or directives to a function:

#pragma hls_design ccore

Or

#pragma design modulario

When these directives are added to a function, the HLS tool shall transform the function as follows:

1. The function shall become an independent sequential process, running in an infinite loop and scheduled according to all the rules outlined in this document.

2. The input arguments to the function shall be transformed into message passing interfaces, and shall be received from the calling function.

3. The output arguments and return value of the function shall be transformed into message passing interfaces, and shall be sent to the calling function.

Additional States added by HLS Synthesis

By default HLS synthesis tools may add additional states to processes (e.g. add latency to enable resource sharing), which may introduce latency differences in the interface behavior between the pre-HLS and post-HLS models. These additional states are never included in the set of "synchronization interface calls" as described above.

When the direct IMPLICIT_FSM=true is set on a process, the HLS synthesis tool shall ensure that the cycle level behavior of the interfaces of the pre-HLS and post-HLS models shall be identical. With this option, the internal state machines of the pre-HLS and post-HLS models will be the same.

When the directive IO_MODE=FIXED is set on a process, the HLS synthesis tool shall ensure that the cycle level behavior of the interfaces of the pre-HLS and post-HLS models shall be identical. With this option, it is still possible that the state machine internal to the process is different between the pre-HLS and post-HLS models (e.g. the post-HLS model may choose to use a pipelined multiplier where the pre-HLS model did not.)

Avoiding Pre-HLS and Post-HLS Simulation Mismatches

The scheduling rules described in this document are designed to be easy to understand, while providing good QOR via HLS and generally avoiding any mismatches between the pre-HLS and post-HLS simulation behaviors.

Non-blocking message passing operations (PushNB/PopNB in SystemC, C++ ac_channel nb_read/nb_write) are a potential source of pre-HLS versus post-HLS simulation mismatches since their behavior is inherently dependent on the latency within the model, which often changes during HLS. Because of this, non-blocking message passing interfaces should only be used when no alternative approach is possible. For example, non-blocking message passing interfaces are required to model time-based arbitration of multiple message streams which access a shared resource. A full discussion of recommended guidelines on the use and verification of non-blocking message passing interfaces is beyond the scope of this document. Note that the scheduling rules described previously in this document fully specify how HLS tools are required to schedule such operations.

SystemC signal IO operations are a potential source of pre-HLS versus post-HLS simulation mismatches since timing behaviors may change between the two models. The following section provides guidance and rules to help avoid potential mismatches due to signal IO.

The scheduling rules state that signal IO operations occur at either SystemC wait statements or SyncChannel calls (sync_in and sync_out). In this section we will use “wait” statement to refer to both.

RULE 1: It is always best coding style to group signal write operations just before their corresponding wait statement, and signal read operations just after their corresponding wait statement. (Cat3). An example is below:

```
 sc_in<int> i1;
 sc_in<bool> go;
 sc_out<int> o1;
 void my_thread {
   int new_val=0;
   while (1) {
     o1.write(new_val);
     do {
       wait();
     } while (!go.read());
     new_val = i1.read();
     new_val = some_function(new_val); // function has no internal IO
   }
}
```

By placing the signal IO operations as close as possible to their corresponding wait statement, the HW intent is very clear. And, there is no benefit either in terms of simulation performance or HLS QOR if they are placed further away from their corresponding wait statement.

Let's look at another similar example, which now also uses a Matchlib Connections blocking Pop operation:

```
 sc_in<int> i1;
 sc_in<bool> go;
 sc_out<int> o1;
 Connections::In<int> pop1;
 void my_thread {
   int new_val=0;
   while (1) {
     o1.write(new_val);
     do {
       wait();
     } while (!go.read());
     int pop_val = pop1.Pop();
     new_val = i1.read();
     new_val = some_function(new_val + pop_val); // function has no internal IO
   }
}
```

According to the "Conceptual Model scheduling rules" part of this document, the Pop operation does not affect the scheduling of the i1.read() operation. However, it is possible that in the pre-HLS SystemC simulation, the Pop operation may "block" for a clock cycle or more (only if no items are available to Pop). This means that it is possible in the pre-HLS simulation that the value of the signal i1 may change between the time before the Pop operation starts and the time it completes. If this occurs, there may be a pre-HLS and post-HLS simulation mismatch if the HLS tool schedules the i1.read() operation at the wait statement. The proper fix to this issue (to reiterate) is to move the i1.read() operation as close as possible to its corresponding wait statement. This will make the potential simulation mismatch disappear, and it will not affect QOR or simulation performance.

To automatically avoid all such potential pre-HLS versus post-HLS simulation mismatches, HLS tools may provide error or warning messages in cases where models have the pattern shown above. Precisely

speaking: if a blocking message passing operation separates a signal read or write operation from its corresponding synchronization interface call, then the HLS tool may emit an error or warning indicating that reordering the signal IO operation and the message passing operation in the source text is advisable.

Another scenario in which RULE 1 applies is shown below:

```
 sc_in<bool> go;
 sc_out<int> o1;
 void my_thread {

   while (1) {
     wait();              WAIT 1
     o1.write(some_value);
     if (go.read()) {
       some_value = some_function();
     }
     else {
       wait();            WAIT 2
     }
     some_other_function();
     wait();              WAIT 3
   }
}
```

The signal read of “go” is clearly and uniquely associated with WAIT 1. However the signal write of “o1” associates with WAIT 2 if “go” is false and WAIT 3 if it is not. This is a violation of RULE 1 and should be flagged as an error during HLS. The fix, as before, is to move the signal IO operation as close as possible to its intended wait statement so that the association is unconditional.

Next, let’s consider rolled (or “preserved”) loops that perform signal IO within the loop body. Consider the following example:

```
 sc_in<int> i1;
 sc_out<int> o1;
 void my_thread {
     wait(); // reset state
     while (1) {
       wait(); // start of while loop
       #pragma hls_unroll no
       for (int i=0; i < 10; i++) {
         o1.write(i1.read() * i);
       }
   }
}
```

Note that the i1.read() operation is located inside the “for” loop, so presumably the user’s intent is that it should be read as the loop iterates. *If that is not the user’s intent, then he simply should move the i1.read() operation before the loop start.*

In the post-HLS simulation, each iteration of the loop will consume at least one clock cycle, and a new value for i1 will be read (and a new value for o1 written) on each iteration. Again, this is the user’s intent as per the code.

In the pre-HLS simulation, the for loop body will execute in zero time, and only the last write to o1 will have any effect. The solution to avoid this mismatch is to manually place a "wait()" statement within the for loop body so that the signal IO synchronization is explicit in the pre-HLS simulation.

RULE 2: If you have signal IO operations within rolled (or "preserved" loops), manually place a wait statement within the body of the loop to avoid pre-HLS versus post-HLS simulation mismatches, and while doing so also follow **RULE 1**.

To automatically prevent these types of pre-HLS versus post-HLS simulation mismatches, HLS tools may emit warning or error messages if they encounter a rolled loop which has signal IO operations within the loop body, and the loop body does not have a wait statement included within the loop body. (Cat4)

Returning to the Analogy from RTL Synthesis

At the beginning of this document we presented the example of a Verilog sequential block with an output coded like:

```
Out1 <= new_val # some_delay;
```

Recall that in Verilog simulation, if some_delay is less than the clock period of the block, then it will probably not affect the overall cycle level behavior of the system during simulation. However, if some_delay is more than the clock period, it probably will.

During RTL synthesis, all RTL synthesis tools will ignore all delays in the input model, in this case even if some_delay is greater than the clock period. Some RTL synthesis tools might give a warning for the code above similar to "Simulation and synthesis results are likely to mismatch because delay in model is greater than clock period."

HLS tools that choose to adhere very closely to the "conceptual model" presented in this document should automatically provide errors or warnings for violations of **RULE 1** and **RULE 2** as described in the section above. This is analogous to the error message that the RTL synthesis tool would provide in the example directly above.

However, it is possible also that HLS synthesis tools may choose to adhere in these cases more closely to the pre-HLS SystemC simulation behavior. In this case such HLS tools might not provide any errors or warnings for violations of **RULE 1** and **RULE 2**. This is analogous to an RTL synthesis tool being "very smart" (maybe even too smart) about synthesizing matching HW based on the actual value of some_delay in the example directly above.

Summary

At the beginning of this document we said that the intent was to present "no surprises" to a DV engineer who is using a single testbench to verify both the pre-HLS and post-HLS models. The key aspects of the document which support this are:

- Three groups of IO operations are defined (message passing, signal IO, and synchronization calls) and each is treated uniformly. These IO operations are easy for verification engineers to understand because they are already using them in their testbenches.

- The document specifically avoids complex constructs such as “protocol regions” used in some HLS tools.

- The document preserves the ability of the pre-HLS SystemC model to be “throughput accurate” by using a library such as Matchlib.

- Synchronization calls can affect the scheduling of signal IO operations, and synchronization calls can affect the scheduling of message passing calls, but message passing calls cannot affect the scheduling of signal IO operations and vice-versa.

- HLS cannot by default reverse the order of message passing calls, so it cannot introduce new deadlocks into the post-HLS model.

- HLS pipelining is largely a “don’t care” from the perspective of the verification engineer. If the design and the testbench are insensitive to changes in latency, and if external array accesses are not reordered or rearranged during loop pipelining, then the possible use of HLS pipelining will not affect verification. Even if the design or testbench is sensitive to changes in latency, or they are sensitive to reordering or rearranging of external memory accesses due to the use of HLS loop pipelining, then the behavior of the DUT will only change in expected (rather than unexpected) ways.

- Signal IO operations in the post-HLS model by default always occur exactly at synchronization points (e.g. wait statements) that are either explicit in the pre-HLS model or easily deducible based on the use of loop roll/unroll or loop pipelining directives. The HLS tool is able to check that every signal IO operation is tightly coupled with exactly one wait statement / synchronization operation in the pre-HLS model.

Catapult HLS Status Concerning Rules in this Document

This section clarifies the status of Catapult HLS regarding the rules in this document. It is updated with release-specific information in each Catapult release, so please check the current version of this document in your specific Catapult release for the latest update.