Catapult HLS User View Scheduling Rules

Stuart Swan, Platform Architect, Siemens EDA

10 December 2025

Introduction

In a high-level synthesis (HLS) design methodology, much of the benefit is lost if design verification and debugging must still be performed at the RTL level. In manual RTL design flows, RTL verification is typically one of the most costly and time-consuming phases of the overall flow. In HLS flows, RTL-level verification is even more challenging, because the RTL is machine-generated rather than hand-crafted.

This document presents a set of HLS scheduling rules, a modeling methodology, and a collection of formal proofs that together allow almost all design verification and debug for full systems to be carried out on the pre-HLS SystemC model. The methodology and proofs are organized around a latency-insensitive design style, while still fully supporting real-world techniques such as:

- latency-sensitive signal-level protocols
- latency-sensitive portions of systems
- both sequential and combinational logic
- pipelined designs
- shared memories between sequential processes
- reordering of RAM accesses to optimize hardware pipelines
- removal of pipeline registers for stable signal inputs to hardware pipelines

These techniques capture the requirements gathered from hundreds of Catapult HLS customer tape-outs and are needed to meet the demanding quality-of-results (QOR) targets traditionally achieved with manual RTL design.

The methodology presented here builds on established verification practices such as SystemVerilog UVM, constrained-random stimulus generation, and functional and code coverage. The system under test is modeled and verified in SystemC, and formal proofs establish behavioral equivalence for the synthesized RTL implementation.

Although the focus of this document is HLS, the design methodology and formal proofs can also be applied when HLS is not used, enabling much more efficient verification and debug at a higher level of abstraction than RTL.

A document abstract is provided in Appendix M.

Document Goals

This document presents HLS tool scheduling rules and modeling methodology rules. The goals are:

1. To provide easy-to-understand rules to HLS users.
2. To ensure precise and consistent rules in both SystemC and C++.
3. To offer rules that are effectively compatible with how Catapult currently operates.

4. To enable the best possible *quality of results* (QOR) in Catapult synthesis.
5. To cover all known user requirements and scenarios.
6. To serve as a suitable starting point for a standardization proposal (e.g., in Accellera SWG).

To illustrate the goals of this document more specifically, consider what an engineer writing a testbench for an HLS model needs to understand about how HLS tools operate. This engineer may be using SystemVerilog UVM or may be writing a testbench in C++/SystemC. They likely are not an expert in any specific HLS tool (and may not want to be), but their testbench needs to work for both the pre-HLS model as well as the post-HLS model. Thus, it is crucial for the DV engineer 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 giving the HLS tool ample freedom to optimize the design.

Background

HLS tools generate RTL from C++ models. Broadly speaking, this conversion takes a sequential C++ model and turns it into concurrent hardware that maintains the same 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 each process is specified by the control and dataflow behavior of the C++ code within the process. However, the external communication that each 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. These rules are specified with respect to an individual process, but the intent of the rules is to enable reliable and verifiable behavior of large sets of interacting processes operating as a system in real-world designs. (Appendix G provides formal guarantees regarding the equivalence of the pre-HLS and post-HLS systems.)

By default, HLS tools can insert additional clock cycles (or latency) anywhere within a process -- for example, when pipelining a loop or to enable HW resource sharing. The overall approach used in this document is to make the entire design and testbench *latency insensitive* to the maximum extent possible, while still fully enabling key HW optimizations.

In some cases, designs or testbenches may use protocols which are latency sensitive. These situations can be handled by isolating the latency sensitive portions to small, self-contained parts of the design or

testbench, and then keeping the rest of the design and testbench latency insensitive. See Appendix D for more information.

In many cases the overall design will need to satisfy end-to-end latency requirements. For these designs it is still highly advantageous to use a latency-insensitive modeling approach and verify in the post-HLS model that the overall design latency requirements have been satisfied, since this is typically easy to do.

This document only covers sequential HW processes. Combinational HW processes are omitted since their synthesis is straightforward. All the examples and discussion in this document are for SystemC processes that are sensitive only to a single rising clock edge. (Appendix O discusses support for multiple clock domains.)

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 certain cases cause small deviations from the conceptual model in either simulation or HLS.

A simple real-world example that illustrates the motivation for the scheduling rules is provided in appendix A of this document.

The examples referred to in this document are available here:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/tree/master

The most up-to-date version of this document is available here:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/doc/catapult_user_view_scheduling_rules.pdf

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 <= #some_delay new_val;
```

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 or error for the code above like "Simulation and synthesis results are likely to mismatch because the delay in model is greater than clock period." Some RTL synthesis tools might outright reject a model containing such delays.

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 easily managed.

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

Next let's clarify some terminology related to signal IO. Say you have a Verilog model like:

```
forever begin
  @(posedge clk);  // wait for 1st rising clock edge
  output1 <= input1 + 10;
  @(posedge clk);  // wait for 2nd rising clock edge
end
```

In this example we say that the read of the input1 signal occurs at the first wait statement, and the write of the output1 occurs at the second wait statement, since that is what you would observe in Verilog simulation when you ran this model. To generalize this, in Verilog designs within implicit state machines like this model which are only sensitive to a clock edge, signal reads occur at the closest preceding wait statement, and signal writes occur at the closest succeeding wait statement.

Catapult HLS Status Concerning Rules in this Document

A separate document provides a list of clarifications related to support within Catapult HLS for the rules described in this document. The items in the list are named *Cat#*, so that each item has a unique number.

This document annotates certain rules with *Cat#* to refer to the items in the separate document.

Terms Used in this Document

Latency-Insensitive: In digital hardware design, *latency-insensitive* refers to a system that operates correctly despite variable communication delays between components. This is achieved by using mechanisms to decouple computation from communication timing. Such designs improve scalability and reliability in complex systems with unpredictable or variable latencies. ARM's AXI4 and APB are examples of latency-insensitive protocols. ARM's AMBA 5 CHI credit-based NOC protocol is also an example of a latency-insensitive protocol.

Process: In Verilog, a process is an *always block* and its equivalent constructs. In SystemC, a process is an instance of an SC_THREAD or SC_METHOD.

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
Message-passing read interfaces are always separate from message-passing write interfaces – there are no bidirectional message-passing interfaces.

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.

One-way / two-way handshake protocols: A one-way handshake protocol only has one signal between a sender and receiver to synchronize communication. A two-way handshake protocol has two signals (one in each direction) between a sender and receiver to synchronize communication.

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 in 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 reads and writes, also C++ model *direct inputs*)

All the operations above are referred to as *IO operations*.

Basic Conceptual Model

The basic conceptual model encompasses processes that have no loop pipelining but may have preserved loops. If a process has a preserved loop, then the user may place a wait statement in the loop, or 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 wait statements* in this document. Wait statements explicitly placed in the model are called *explicit wait statements* in this document.

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

The basic conceptual model 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 on separate interfaces which appear in sequence in the model may be issued either in the same sequence or in parallel in simulation and in synthesis, but they shall not be issued in the reverse sequence. (Cat2)

Some explanation for the very last point: While pure message-passing systems with unbounded FIFOs are immune to reordering concerns, real-world hardware implementations have finite buffer sizes. Reversing the order of message-passing calls in a system with bounded FIFOs can introduce deadlocks that were not present in the original model. The last rule ensures that HLS tools cannot introduce such deadlock cases.

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.

The user may place wait statements in the body of a pipelined 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 pipeline stage wait statements*, and the latter *implicit pipeline stage wait statements*.

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

The scheduling rules for pipelined loops are the same as the rules given in the *basic conceptual model*, with the addition of these pipeline stage wait statements into the set of calls to synchronization interfaces. However, one rule from the conceptual model is relaxed when loops are pipelined: During loop pipelining, message-passing read operations from later loop iterations may be scheduled before or in parallel with message-passing write operations from the current loop iteration, and SystemC wait() statements and ac_wait statements, if present in the loop body, will not preclude this from occurring. Potential deadlocks are avoided by having the pipeline automatically flush. See Appendix B for further discussion.

When HLS pipelines a loop, multiple iterations of the loop are overlapped and execute at the same time. During loop pipelining, for all IO operations, HLS shall ensure that an access to a specific 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.

When HLS pipelining occurs, any signal reads and writes and associated synchronization interface calls become embedded in specific pipeline stages. With pipelining, the post-HLS model may begin execution of the next loop iteration before the current iteration has completed. Considering the entire set of signals read or written by the process, if HLS pipelining would cause the order of signal IO to differ between the pre-HLS and post-HLS models, or if any two signal IO operations that occur on the same clock edge in the pre-HLS model would not occur on the same clock edge in the post-HLS model, then the HLS tool shall detect such a situation and require the user to explicitly indicate in the model source (e.g., via a pragma) that HLS pipelining can still be used. (Cat7) Another way to state this requirement is the following: If a process communicates with other processes using only latency-insensitive protocols, then HLS pipelining by default shall not break any of the protocols.

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 process, or if it pipelines the process, 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 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 process.

The simplest case is if such external signals are held stable after the process 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. Note that with this approach it is allowable to have *dynamic resets*, i.e. activation of block resets and associated resetting of direct input signals as part of the normal operation of the HW.

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. 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> SC_NAMED(clk);
  sc_in<bool> SC_NAMED(rst_bar);

  Connections::Out<uint32_t>  SC_NAMED(out1);
  Connections::In<uint32_t>   sample_in[num_samples];
  Connections::SyncIn         SC_NAMED(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 unrolled in cat .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 shall only occur at a precise point. The TB shall first wait for the `rdy` signal for the sync to be asserted by the DUT, and then the TB shall 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 inputs 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.

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.

It is recommended that this relaxed option only be used when the user wants to see what order the HLS tool prefers to schedule message-passing interface calls (e.g., to achieve best QOR). Once the user knows the preferred order, it is recommended that the user modify the pre-HLS source code to reflect the preferred order, and then return to the use of the default scheduling modes. This methodology ensures that HLS cannot introduce any new deadlocks into the system as described earlier.

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 the process, and thus their scheduling is also not visible externally.
   - All the scheduling rules described elsewhere in this document remain unaffected in this case.

2. Array instantiation in the HW is external to the 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.
- In all cases array accesses shall remain constrained by any explicit synchronization interface calls present in the process in the source model. Precisely speaking, all array reads or writes before a synchronization call shall commit before or in the same cycle at which the synchronization call commits, and all array reads or writes after a synchronization call shall commit in a cycle after the synchronization call commits.
- 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 account for 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.

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 directive 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 mismatches between pre-HLS and post-HLS simulations 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 provided in Appendix L. Note that the scheduling rules described previously in this document fully specify how HLS tools are required to schedule such operations.

Unidirectional message-passing between two processes should not be relied on to achieve synchronization between the two processes, since in general the message latency and storage capacity between the processes may be variable. Also, it is possible that the prefetch behavior of the message reader may vary. Bidirectional message-passing can be relied upon for synchronization between two processes. (An example of this is the AXI4 ar and r, and aw and b, channels). Note that such synchronization is weaker than explicit synchronization like SyncChannel or signal IO that uses a two-way handshake.

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.

When signal IO operations are synchronized with a wait statement, there generally should be a proper two-way handshake associated with the wait statement so that the signal IO is latency insensitive. (Note that this statement does not apply to the signal synchronization approaches described in the Direct Inputs section.)

For example, here's a simple two-way handshake protocol when writing the signal `out_dat`:

```
out_dat = value;
out_vld = 1;
do {
  wait();
} while (out_rdy != 1);
out_vld = 0;
```

And here's a two-way handshake example when reading signal `in_dat`:

```
in_rdy = 1;
do {
  wait();
} while (in_vld != 1);
value = in_dat;
in_rdy = 0;
```

Some signal-level protocols have different two-way handshaking approaches (e.g. ARM APB), but they are still latency insensitive.

If signal IO operations are associated with a wait statement and that wait statement does not have a proper two-way handshake, then the signal IO is likely to be latency sensitive and may result in pre-HLS versus post-HLS simulation mismatches. In some systems a one-way signal handshake is sufficient for reliable system operation. See Appendix E for further discussion.

The scheduling rules state that signal IO operations occur at either SystemC wait statements or SyncChannel calls (sync_in and sync_out). In the remainder of 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)

If a pre-HLS model adheres to RULE 1 and RULE 2, then all signal IO in the post-HLS model shall occur only at clock cycles that correspond to explicit wait statements or explicit synchronization statements. User designs that do not adhere to both RULE 1 and RULE 2 are *ill-formed*.

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 <= #some_delay new_val;
```

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 like "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, provided the pipelines flush automatically. Even if the design or testbench are sensitive to changes in latency, or if 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. (Appendix G provides formal guarantees regarding the equivalence of the pre-HLS and post-HLS systems.)

- 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 can check that every signal IO operation is tightly coupled with exactly one wait statement / synchronization operation in the pre-HLS model.

Appendix A – Factory Analogy

The scheduling rules described in this document apply to pre-HLS and post-HLS HW models. Although the rules may seem abstract and perhaps even arbitrary, they are shaped by the need to model systems in the real world that are required to have predictable behavior.

To understand the motivation behind the rules, it might be helpful to draw a simple real-world analogy and its correspondence to the rules described earlier.

Consider a factory that produces various types of wooden furniture:

The factory consists of people (processes) stationed at workbenches with various tools.

Each person is given written instructions about the specific tasks they are to perform (C++ code within a process).

People are instructed to send or receive objects (messages) to or from other people in the factory.

Sending or receiving objects may be blocking or non-blocking from the perspective of a person.

There is a clock with a second hand on the factory wall that everyone can see. (HW clock).

People have colored flags they can raise or lower to communicate with other people (signals).

If someone raises or lowers a flag, this is only seen by others the next time the clock second hand is at the top of the clock. (propagation delay of signals between sequential processes)

A person can choose to pipeline the tasks that they were assigned by hiring subordinates (pipeline stages) and having each one do a subtask. In general, this will improve the throughput for the tasks that the person was assigned.

It is possible for two people to explicitly synchronize their work by communicating via a synchronization protocol such as a barrier (synchronization).

It is possible to restart the work of some or all the people by raising a reset flag (HW resets).

Assume:

1. That the time that people take to complete their various tasks is in general variable, and similarly that the time that objects (messages) take to pass between people is in general variable.
2. That each person in general wants to complete their tasks as quickly and efficiently as possible.
3. That the factory needs to be able to make multiple types of furniture at the same time. For example, it might make chairs that need to be different colors or have different styles.

To reliably produce output, each person in the factory will need to adhere to rules like those described in this document.

Here's a sketch of a specific way the above example relates to the scheduling rules:

Person 1 sends chair seats and chair backs to Person 2. This is done with blocking operations, and there is no storage capacity between the people when the objects are sent. (This means that a Push operation cannot complete until the corresponding Pop operation is performed.) Assume that the fastest an object can be sent between people is 1 minute.

The written instructions that person 1 is given are:

```
while (1) {
 // internal processing code for seats and backs ...
 seats.Push(seat_object);
 backs.Push(back_object);
}
```

The written instructions that person 2 is given are:

```
while (1) {
 seat_object = seats.Pop();
 back_object = backs.Pop();
 // internal processing code for seats and backs ...
}
```

If both person 1 and person 2 choose to perform their tasks sequentially as written, then objects will be passed over time as:

```
            Person 1            Person 2
Minute 1:   seats.Push(seat1);
Minute 2:   backs.Push(back1);  seat1 = seats.Pop();
Minute 3:   seats.Push(seat2);  back1 = backs.Pop();
Minute 4:   backs.Push(back2);  seat2 = seats.Pop();
Minute 5:                       back2 = backs.Pop();
```

If both person 1 and person 2 choose to perform their IO operations in parallel, then objects will be passed over time as:

```
            Person 1                                Person 2
Minute 1:   seats.Push(seat1); backs.Push(back1);
Minute 2:   seats.Push(seat2); backs.Push(back2);   seat1 = seats.Pop(); back1 = backs.Pop();
Minute 3:                                           seat2 = seats.Pop(); back2 = backs.Pop();
```

If person 1 chooses to perform his IO operations in parallel, and person 2 chooses to perform his IO operations sequentially as written, then objects will be passed over time as:

```
            Person 1                                Person 2
Minute 1:   seats.Push(seat1); backs.Push(back1);
Minute 2:                                           seat1 = seats.Pop();
Minute 3:   seats.Push(seat2); backs.Push(back2);   back1 = backs.Pop();
Minute 4:                                           seat2 = seats.Pop();
Minute 5:                                           back2 = backs.Pop();
```

If person 1 chooses to perform his IO operations sequentially as written, but person 2 chooses to perform his IO operations sequentially in the reverse order as it was written, then objects will be passed over time as:

```
            Person 1            Person 2
Minute 1:   seats.Push(seat1);
Minute 2:   backs.Push(back1);  back1 = backs.Pop();
Minute 3:                       seat1 = seats.Pop();
```

In this case the person 1 seats.Push(seat1) operation at minute 1 will not complete at the start of minute 2. This means that the person 1 backs.Push(back1) operation will never start, and thus the Person 2 back1 backs.Pop() operation will never complete. So, the system will be in deadlock.

This example directly corresponds to the ordering rules related to message-passing operations in the *basic conceptual model* within this document, and in particular the specific rule that disallows reversing the order of message-passing operations.

Appendix B – Pipelined Loops and Message-passing

The scheduling rules state that during loop pipelining message-passing read operations from later loop iterations may be scheduled before or in parallel with message-passing write operations from the current loop iteration. This might seem to be in contradiction with the rules for message-passing operations described in the *basic conceptual model*. However, it is important to remember that each message-passing channel is *self-synchronized*. If a process has a pipelined loop that automatically flushes in the post-HLS model, then from an external perspective the only effect of allowing the message-passing read operations to occur earlier is like a *pre-fetch* operation. If the incoming message is not available, the pipelined process in the post-HLS model will flush, and its behavior from the external perspective will appear exactly as if it is not pipelined.

Appendix C – Verification of Designs that are Partially Latency Sensitive

This content has been moved to Appendix L.

Appendix D – Modeling Latency-Sensitive Protocols

In some cases, designs or testbenches may use protocols which are latency sensitive. These situations can be handled by isolating the latency-sensitive portions to small, self-contained parts, and then keeping the rest of the design and testbench latency insensitive.

Consider a case where a signal-level protocol is latency sensitive. The protocol will have specific timing behavior that it must meet. To handle this, we create a transactor that has two sides: the first side interacts with the signals and handles the detailed timing requirements, and the second side sends and receives messages with the rest of the system. The message-passing side is latency insensitive.

To properly model the detailed timing behavior, the transactor is modeled at the cycle-accurate level in SystemC. There is an example of such a transactor model in the following document and example:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/examples/53_transactor_modeling/transactor_modeling.pdf

Appendix E – One-Way Handshake Protocols

In some systems a one-way signal handshake is sufficient for reliable system operation. When using a one-way handshake, the implicit assumption is that the "missing" handshake isn't required since it is always true.

For example, in time-domain signal processing hardware designs, signal processing HW blocks may input new samples at a fixed rate. The HW blocks are always ready to receive new samples on each clock, but they need to know if the samples are valid. These types of designs can be modeled with the one-way handshake dat/vld protocol demonstrated in this example:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/tree/master/matchlib_examples/examples/32_dat_vld

Appendix F – Scheduling Rules: Modeling Guidelines Summary

1) Prefer to use Connections::In/Out + SyncChannel over signal IO and wait().

2) Prefer to use Pop()/Push() over PopNB()/PushNB().

3) When pipelining a loop, prefer to use `hls_stall_mode flush`.

When using signal IO:

4) If modeling a cycle-accurate process, use `disable_spawn` and follow example 53_transactor_modeling style and do not use Push/Pop in the process.

5) If modeling a *direct input*, use `hls_direct_input` and possibly `hls_direct_input_sync`, and only change the signal at allowed times.

6) If combining signal IO with Connections::In/Out in same process, use proper signal IO handshake that does not rely on In/Out ports for process synchronization. Place signal IO operations very close to their wait() statements.

7) Unless you are modeling a cycle-accurate process, you should expect that latency will change between the pre-HLS and post-HLS models.

Appendix G – Equivalence Rules

This document informally states that a primary goal is to present "no surprises" to a DV engineer using the same testbench to verify the pre-HLS and post-HLS models.

Somewhat more formally, we can show how the scheduling rules within this document establish a set of equivalence rules between the pre-HLS and post-HLS models that provide strong guarantees about their behaviors.

The *basic conceptual model* establishes the base behavior for a single process:

1) Synchronization interface calls always remain in source code order.
2) Signal reads occur at closest preceding synchronization interface call.
3) Signal writes occur at closest succeeding synchronization interface call.
4) All message-passing calls after a synchronization interface will not start until the synchronization interface call has completed.
5) All message-passing calls before a synchronization interface shall be completed when the synchronization interface call has completed.
6) Two message-passing operations on separate interfaces which appear in sequence in the model may be issued either in the same sequence or in parallel in simulation and in synthesis, but they shall not be issued in the reverse sequence.
7) 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.
8) The STRICT_IO_SCHEDULING=relaxed option is not used.
9) In the pre-HLS model, the storage capacity of message channels is zero (rendezvous-style communication).

Conceptually, systems are composed of many such processes which follow the *basic conceptual model,* and which communicate using only *synchronization interface calls* and *latency-insensitive* signal level protocols and *blocking message-passing* calls, and no *non-blocking message-passing* interface calls. Here we call this system of processes the *conceptual system*.

Practically, the modeling approach described above is too restrictive for real-world systems with optimized HW, so the scheduling rules in this document allow for key HW optimizations. But this is done in a carefully controlled manner, such that the HW optimizations do not break equivalent behavior with the *conceptual system*.

Here is a summary of the HW optimizations that the scheduling rules allow, and a brief description of how we maintain equivalent behavior with the *conceptual system*:

1. Pipelined loops: Message-passing read operations from later loop iterations can execute before message write operations from the current loop iteration. Assuming the pipeline automatically flushes, this behavior change can be viewed as a *prefetch* operation. If the message to be read is unavailable, the pipeline will flush, and the process will appear exactly as if it is not pipelined.

2. Pipelined loops: HLS by default will not break any latency-insensitive signal IO protocols when pipelining loops.

3. Direct inputs: Signals that do not change after a process comes out of reset can be read as late as possible using *hls_direct_input*, saving register area.

4. Direct inputs: When *hls_direct_input_sync* is used, signals read by a process are updated at the precise point at which the HW pipeline is guaranteed to be ramped down, so the signal values do not need to be saved within pipeline registers, saving register area.

5. Memory accesses: HLS can reorder memory accesses within a process only if it can prove that the reordering is conflict-free.

6. Shared memories: Memories shared between processes are modeled as simple C arrays but always include explicit synchronization between processes in both the pre-HLS and post-HLS models.

7. Non-blocking message-passing interfaces: If latency differences are externally visible to the DUT, equivalent behavior between pre-HLS and post-HLS systems can be maintained by snooping the post-HLS system and using this to delay pre-HLS message arrival.

8. Latency-sensitive global signal IO: If latency differences are externally visible to the DUT, equivalent behavior between pre-HLS and post-HLS systems can be maintained by snooping the post-HLS system and using this to delay pre-HLS signals.

9. Latency-sensitive local signal IO: Isolate local latency-sensitive protocols to small, dedicated transactors, and use latency-insensitive signal IO or message-passing to communicate with the rest of the system.

10. One-way signal handshake protocols: Only use in cases where backpressure is not possible and embed assertions into the model to check that this is always true.

11. Combinational processes: If the pre-HLS model contains combinational processes, they will have identical behavior in the post-HLS model and thus they cannot introduce any differences between the models.

Taken as a whole, these rules allow full system verification to be performed on the pre-HLS system model. Full system verification does not need to be repeated on the post-HLS RTL system, provided the equivalence rules described above are followed.

Formalization of the Equivalence Rules in Appendix G

The following notation is used in this section.

Symbol Meaning

P	A process (thread or method) in the design
Σ	The alphabet of observable IO actions

τ_P	A (finite or infinite) trace of observable actions executed by process P, ordered by global clock cycles
S	The subset of Σ that are synchronization calls (explicit wait, SyncChannel, START_P/FINISH_P indicating process start/finish)
R	The subset of Σ that are signal reads
W	The subset of Σ that are signal writes
M	The subset of Σ that are message-passing calls (Push, Pop, etc.)

A *system trace* is the tuple
τ = (τ_P)_P, one component per process.
For any action a ∈ Σ, let clk(a) be the clock cycle at which a is *committed*.
For any message operation *m*, issue_pre(m)/issue_post(m) is the clock cycle in which the pre-HLS or post-HLS process first asserts the ready/valid handshake for *m*, independent of the later commit cycle clk(m). Because issue timing is under sole control of the issuing process, it can be used to express per-process ordering constraints.

---

1. Conceptual Model for a Single Process

For every process P the pre-HLS trace τ_P must satisfy the following *partial-order constraints*:
R1 (Program order of S).
∀ s1, s2 ∈ S : s1 <_src s2 ⇒ clk(s1) < clk(s2)
R2 (Location of R/W).
∀ r ∈ R : clk(r) = clk(pred_S(r))
∀ w ∈ W : clk(w) = clk(succ_S(w))
where pred_S(r) (resp. succ_S(w)) is the closest synchronization call that precedes (resp. follows) the statement in source order.
If no lexical predecessor (successor) exists, a virtual synchronization operation at time 0 (resp. ∞) is assumed.
R3 (Isolation around S).
Let pref_S(s) (resp. suff_S(s)) be the set of message calls lexically before (resp. after) synchronization call s. Then
∀ m ∈ pref_S(s) : clk(m) ≤ clk(s) and ∀ m ∈ suff_S(s) : clk(s) < clk(m).
R4 (Safe message issue ordering).
For every pair m1,m2 ∈ M on distinct channels,
m1 <_src m2 ⇒ issue_pre(m1) ≤ issue_pre(m2).
R5 (Rendezvous channel capacity in pre-HLS model).
For every pre-HLS channel c, at any clock cycle t, the number of unmatched Push_c actions committed is zero, and no Push_c shall commit unless a matching Pop_c also commits at t.
The pre-HLS *conceptual* semantics for a process is thus a labeled partial order
(Σ, ≤_P), where ≤_P is generated by R1–R5.

---

2. Equivalence Relation

Let τ_pre and τ_post be the traces of the same process before and after HLS.
We say τ_pre ≈ τ_post iff all of the following hold:
E1 (Synchronization-order preservation).
∀ s1,s2 ∈ S : clk_pre(s1) < clk_pre(s2) ⇒ clk_post(s1) < clk_post(s2)

E2 (Signal-visibility preservation).
$\forall r \in R$: clk_post(r) = clk_post(pred_S(r))
$\forall w \in W$: clk_post(w) = clk_post(succ_S(w))
E3 (Safe message issue ordering).
For every pair m1,m2 ∈ M on distinct channels within a single process,
m1 <_src m2 ⇒ issue_post(m1) ≤ issue_post(m2).
E4 (Per-channel FIFO semantics).
For every channel c, the sequence of Push_c and Pop_c actions in τ_post is a legal FIFO schedule (no dropped or duplicated messages).
E5 (Messages cannot cross syncs).
For every message m and synchronization call s,
clk_pre(m) ≤ clk_pre(s) ⇒ clk_post(m) ≤ clk_post(s)
clk_pre(m) > clk_pre(s) ⇒ clk_post(m) > clk_post(s)

See *Appendix I – Per Process Trace Equivalence Proof* for formal proof of the following:
For any single pre-HLS process P that obeys the conceptual R rules and B rules and communicates over channels of finite capacity $B(c) \geq 0$, every finite observable trace produced by P has a matching post-HLS RTL trace that satisfies the equivalence properties E1–E5.

---

3. Allowed Transformations and Why They Preserve ≈

Transformation permitted by the rules	Formal justification
Loop pipelining (overlaps iterations)	Creates additional S actions (implicit stage waits). Rules R1–R5 are applied with the larger S set. E1–E5 hold because actions of different iterations that refer to the same channel or signal remain separated by at least one S.
Re-ordering of message reads vs. later writes inside a pipeline	Allowed only when the two actions are on *different* channels and the pipeline auto-flushes. This preserves FIFO property per channel (E4) and does not violate E3 and E5.
#pragma hls_direct_input (late sampling of static signals)	This pragma signals a designer-guaranteed exception to Rule E2, where functional equivalence is preserved due to the asserted stability of the signal's value after reset.
#pragma hls_direct_input_sync (controlled updates)	This pragma establishes a specific timing exception to Rule E2 for the controlled input signals. The update of these signals is explicitly bound to the specific, preceding synchronization call s*, creating a new, localized rule where clk(w_update) = clk(s*). Functional equivalence is preserved because this explicit timing contract replaces the default rule and is applied consistently to both the pre-HLS and post-HLS models.
Memory-access reordering proven conflict-free	Let addr(a) be the symbolic address of access a. If the tool proves $addr(a_1) \neq addr(a_2)$ for the overlapped window, then the commutation of a_1 and a_2 is observationally silent; no external signal/message depends on the internal order.

Transformation permitted by the rules	Formal justification
Implicit FSM states / added latency	Extra states introduce *silent* ε-steps between observable actions; the partial order on Σ is unchanged, so E1–E5 are unaffected.
Shared-memory arrays with explicit synchronization	When a memory is accessed by multiple processes, the *designer* explicitly coordinates access with a sync operation modeled as an S-action. Because these S-actions appear in both traces, the read/write order is fixed by E1–E2, port-level FIFO legality is preserved by E4, and the overall equivalence relation ≈ still holds.
Non-blocking message-passing (PushNB/PopNB) verified by post-HLS snooping	If the latency differences are externally visible to the DUT, a verification wrapper delays the *pre-HLS* arrival of the nb-message until the identical event is observed on the *post-HLS* channel. This wrapper is itself purely synchronising (S), so it cannot violate R1–R5. Once the wrapper is assumed, τ_pre and τ_post coincide on all M-actions, so E3–E5 are preserved. This is not a transformation that inherently preserves ≈, but rather a verification technique to enforce equivalence for inherently latency-sensitive operations.
Latency-sensitive *global* signal IO matched by snooping	If the latency differences are externally visible to the DUT, the same "mirror-and-delay" wrapper used for NB channels is applied to any globally-visible signal whose timing matters. Because the wrapper inserts only S-actions, the partial-order constraints are unchanged. This is not a transformation that inherently preserves ≈, but rather a verification technique to enforce equivalence for inherently latency-sensitive operations.
Latency-sensitive *local* signal IO isolated in cycle-accurate transactors	Local timing-exact protocols are confined to dedicated transactor processes that expose only latency-insensitive M or S operations to the rest of the design. Since the transactor boundary is now the observable interface, R1–R5 apply *outside* the timing-sensitive region, so system-level ≈ still holds.
One-way signal-handshake protocols with run-time assertions	For single-direction vld or rdy signals, an assertion proves that back-pressure can *never* occur. That assertion establishes a refinement in which the missing handshake is equivalent to a permanent logical '1', so the protocol is observationally identical to a two-way handshake that trivially satisfies E2.
Combinational processes	These have no S, R, W, or M actions, so their τ_P is the empty trace. The equivalence relation therefore holds vacuously.

For the purposes of the formal analysis, we take as an axiom that any global latency-sensitive signals and any latency-sensitive non-blocking message-passing operations in which the latency differences can be

seen externally to the DUT can be perfectly synchronized between the pre-HLS and post-HLS simulations via the snooping technique. With this axiom we can treat non-blocking message-passing operations in an identical manner to blocking message-passing operations within the overall formal analysis.

Appendix L provides detailed guidance to enable this perfect alignment to be achieved in practice.

---

4. Proof Structure

The proofs proceed in three stages:
First, in Appendix I we establish per-process equivalence by showing that, under a *live environment*, every RTL process trace corresponds to a source-level trace and vice versa. Here, *live environment* does not mean assuming the entire system is deadlock-free; it refers specifically to the standard conditions of weak fairness, finite pipeline depth, and the local progress invariants (e.g., P_push(c),P_pop(c)). These are scheduling and channel-usage side-conditions, not the global liveness property that will be proved later.
Second, in Appendix J we compose these per-process results into a system-level equivalence theorem. This stage relies on channel-ordering and occupancy lemmas but does not assume system-level liveness.
Finally, Appendix K discharges the earlier “live environment” assumption by proving that system-level liveness is preserved through HLS. Specifically, if the rendezvous model is live under weak fairness, then the buffered post-HLS system is also live. This step ensures that the overall proof is non-circular: the liveness property invoked in the first stage is rigorously established only at the end.

---

5. Causal Dependency Between Processes

To prove system-level equivalence, we must distinguish between incidental timing and functionally significant ordering.

Formal Definition of Causal Dependency

To formalize the notion of functionally significant ordering and resolve ambiguity, we define the happens-before relation, denoted by ->, on the set of all observable IO actions (Σ) across all processes in the system. This relation establishes a strict partial order of events, where a -> b is read as "a happens before b".
The *happens-before* relation is the smallest relation satisfying the three conditions below:

1. Intra-Process Order: If actions a and b occur within the same process P and a precedes b in the program's execution trace, then a -> b. This directly reflects the sequential nature of the code within a single thread.

2. Inter-Process Communication: The relation is established by the direct exchange of information or synchronization between processes.

    - Message-passing: If a is a blocking Push or non-blocking PushNB action on a channel in process P1 that sends a message, and b is the corresponding Pop or PopNB action in process P2 that receives that same message, then a -> b.

- Signal Handshake: If a is a signal write action w(sig) in P1 and b is a signal read action r(sig) in P2 that reads the value written by a as part of an explicit handshake protocol, then a -> b. A complete two-way handshake (e.g., vld/rdy) creates a causal chain, such as w(vld)@P1 -> r(vld)@P2 -> w(rdy)@P2 -> r(rdy)@P1.
- Explicit Synchronization: If a set of actions {s_1, s_2, ... s_n} across different processes participates in a single, atomic synchronization event (e.g., a SyncChannel barrier), then the completion of that event for the group happens-before any subsequent, dependent action in any of the participating processes.

3. Transitivity: The relation is transitive. If a -> b and b -> c, then a -> c.

Using this relation, we now provide a formal definition for causal dependency between synchronization events, which are the fundamental ordering anchors in the scheduling model.

Definition: Causal Dependency
A synchronization event s2 in process P2 is causally dependent on a synchronization event s1 in process P1 if and only if s1 -> s2.
If neither s1 -> s2 nor s2 -> s1 holds true, the events s1 and s2 are considered causally independent (or concurrent). Any change in the observed temporal ordering of causally independent events between the pre-HLS and post-HLS simulations is considered an incidental, functionally insignificant variation in latency. A well-formed design, under this methodology, shall not rely on a specific ordering of causally independent events for functional correctness.

To be clear, the shall not rely requirement is a design rule. To adhere to this design rule, designs must use message passing, SyncChannel operations, and signal handshakes to properly order operations across the system. Designs which violate this design rule are outside of the formal guarantees.

---

6. System-Level Theorem

Theorem (Compositional Equivalence).
For every process P in the design, let the HLS compiler apply only transformations listed in Section 3. Then
$\forall P : \tau\_pre,P \approx \tau\_post,P \Rightarrow$ the entire system trace satisfies
$\tau\_pre,system \approx \tau\_post,system$.
Proof sketch. (Full proof is in Appendix J – System-Level Trace Equivalence Proof)
Equivalence is defined per observable channel or signal.
By E1–E5 each process preserves:

1. ordering of synchronizing actions visible to other processes,
2. FIFO legality of every channel,
3. atomic association between signal IO and its bounding sync.
4. the *side-of-sync* predicate for all message actions (E5).

Because the composition of partial-order–preserving relationships is itself preserving, and because channels/signals are the only inter-process observables, system-level behavior is bisimilar under the relation ≈.

---

7. Practical Implication (“No Surprises” Guarantee)

If a verification environment exercises only the alphabet Σ (signals, message ports, synchronization calls) and does not test internal latency, then any testbench that passes on the pre-HLS model is *provably*

guaranteed to pass on the post-HLS RTL, provided the design obeys the scheduling rules. This justifies single-testbench methodologies.

Appendix G thus establishes that the scheduling rules create a *trace-equivalence* relation between the high-level model and the synthesized RTL: every legal compiler optimization is a morphism of the labeled partial-order structure defined by R1–R5, ensuring functional indistinguishability at all observable interfaces.

Appendix H – Possible Criticisms

This section highlights several potential challenges in the verification approach of Appendix G.

1. Designer-Managed Shared-Memory Synchronization

Appendix G requires that any memory shared between processes use explicit synchronization primitives inserted by the designer. The HLS tool does not perform static verification of those primitives. In practice, we mitigate this risk by encapsulating shared memories within parameterized library classes (for example, see examples 12_ping_pong_mem and 15_native_ram_fifo) that are pre-verified for correct memory access coordination in both the pre-HLS and post-HLS models. Typically such classes will be parameterized for aspects such as element type and memory size.

2. Latency-Sensitive Signal Alignment (“Snooping”)

See Appendix L for detailed discussion on snooping.

3. Matchlib Connections by Default Model a Skidbuffer inside In<> Ports

(Note that this overall document and specifically Appendix G are not intended to be strictly tied to Matchlib. Instead, this document is intended to be applicable to any pre-HLS model written in SystemC using signals, message-passing channels, and synchronization calls.)

Matchlib Connections supports throughput accurate modeling in the pre-HLS simulation. Currently the throughput accurate modeling mechanism relies on the Pre() and the Post() methods using a 1-place buffer to store transactions that are in flight on Connections::In<> ports. This means that by default In<> ports function like a skidbuffer. Connections::Out<> ports do not introduce any additional storage capacity into the pre-HLS simulation.

By default, Catapult adds skidbuffers on input ports into the post-HLS design, so the formal equivalence relationship described in this document still holds. In other words, the default pre-HLS and post-HLS designs both still model rendezvous semantics, since both explicitly include a structural instantiation of a skidbuffer on input ports.

It is possible to remove some or all the skidbuffers during Catapult HLS. In this case, to make the formal equivalence relationship hold, the pre-HLS simulation must use the
`-DFORCE_AUTO_PORT=Connections::SYN_PORT`
compile flag. This will remove all the skidbuffers from the pre-HLS model. If some of the skidbuffers are still inserted during HLS, the formal equivalence relationship will still hold.

In this case the recommended methodology is to use both approaches:

- Use the default throughput accurate mode in Matchlib for most analysis and debug, and for pre-HLS performance verification.

- To enable the pre-HLS simulation to strictly conform to the formal model presented in Appendix G, then use -DFORCE_AUTO_PORT=Connections::SYN_PORT.

Common Formal Definitions for Appendices I-K

Symbol / Rule	Definition — concise but complete
Σ	Set of *observable* actions: • channel operations Push_c, Pop_c • synchronization events wait(), Sync, START_P, FINISH_P • single-cycle signal Write/Read actions.
START_P	First observable action emitted by process P after reset. Marks the moment P begins executing user code. A new START_P is emitted each time P exits reset.
FINISH_P	Last observable action of process P. If P executes an unbounded loop, FINISH_P never occurs, and the trace is treated as infinite.
ε-step	An *internal*, unobservable RTL state transition (pipeline advance, FSM micro-state, etc.).
B(c)	Compile-time *capacity* of channel *c* chosen by the HLS tool. Finite, B(c) **≥** 0. • B(c)=0 ⇒ rendezvous (capacity-zero) channel. • B(c)>0 ⇒ bounded FIFO.
occ_c(t)	*Occupancy* of channel *c* after cycle *t*: number of committed Push_c minus committed Pop_c.
P_push(c)	*Finite-progress invariant (producer)*: if Push_c is continuously enabled while occ_c(t)=B(c), then some future cycle t′>t commits a Pop_c.
P_pop(c)	*Finite-progress invariant (consumer)*: if Pop_c is continuously enabled while occ_c(t)=0, then some future cycle t′>t commits a Push_c.
Equivalence rules (E1–E5)	*E1 Synchronization-order preservation*: pre- and post-HLS traces share the same order of synchronization events. *E2 Signal visibility*: every Write/Read pair appears in the same relative order in both traces. *E3 Safe message issue ordering* For every pair of message operations m1, m2 on distinct channels, if m1 <_src m2 then issue_post(m1) ≤ issue_post(m2)

Symbol / Rule	Definition — concise but complete
	FIFO order is always preserved on same channel: $\forall m_1,m_2 \in M$ on the same channel: clk_pre(m_1) < clk_pre(m_2) ⇒ clk_post(m_1) < clk_post(m_2) *E4 FIFO legality*: for each channel *c*, the post-HLS (Push_c, Pop_c) history is a legal bounded-capacity execution consistent with the pre-HLS rendezvous history. *E5 Side-of-Sync guarantee*: a message action never moves across its bounding synchronization event (START_P, FINISH_P, explicit wait/Sync).

Additional Semantic Assumptions (B-rules)

The following B rules are process assumptions used within the formal proofs. These B rules are in addition to the R rules presented in Appendix G.

ID	Basic-process property (informal statement)	Why it is needed / how it is used
B1 Deterministic Trace Property	For any fixed test stimulus and initial state, the sequence of *observable* actions (S, R/W, M) emitted by the post-HLS design is unique. Internal ε-steps may differ between runs, but the externally visible trace cannot.	Ensures the per-process and system-level equivalence proofs (App. I & J) can match *one* post-HLS trace to *one* pre-HLS trace without branching on scheduler nondeterminism.
B2 Weak Fairness of the Scheduler (WF)	If an action's enabling predicate remains continuously true from cycle *t* onward, the scheduler must eventually select that action (within a finite, but unspecified, number of cycles). Applies to Push, Pop, and Sync operations.	Required for all progress arguments, especially the liveness lemmas in Appendix K and the channel-progress corollaries.
B3 Channel Progress Invariants	For every channel *c* with capacity B(c): • P_push(c): If $Push_t(c)$ stays enabled while the channel is full *($occ_t(c) = B(c)$)* then some $Pop_t(c)$ must eventually occur. • P_pop(c): If $Pop_t(c)$ stays enabled while the channel is empty *($occ_t(c) = 0$)* then some $Push_t(c)$ must eventually occur.	Explicitly assumed in Appendix J to rule out permanent back-pressure loops that are *logically* independent of the cycle-by-cycle R-rules.
B4 Finite Stutter Bound	Every process *P* has a finite constant depth D_P such that between any two successive observable actions generated by *P* there are at most D_P clock cycles containing only ε-steps (micro-states).	Guarantees the ε-stutter closure used when constructing witness mappings in Appendix I; lets the proofs quantify "within ≤ D_P cycles" instead of "eventually".
B5 System Quiescence Closure	If, at some cycle t_o, no observable actions are enabled in any process, then within at most max_P D_P further cycles the system reaches a fixed point and executes no additional ε-steps.	Needed to finish the starvation-vs-deadlock analysis in Appendix K: ensures an all-disabled state cannot hide behind an infinite tail of unobservable activity.

Appendix I – Per Process Trace Equivalence Proof

I.1 Overview

For any single pre-HLS process P that obeys the basic R rules and B rules and communicates over channels of finite capacity $B(c) \geq 0$, every finite observable trace produced by P has a matching post-HLS RTL trace that satisfies the equivalence properties E1–E5. The construction is explicit and does not rely on unbounded buffers; rendezvous channels ($B(c)=0$) are handled directly.

I.2 Formal Preconditions

- Observable alphabet Σ = { Push_c, Pop_c, Sync, wait, START_P, FINISH_P, Write, Read }.
- Silent step Any internal RTL micro-state transition is written ε.
- Finite-pipeline-depth premise (FPD) There exists a finite constant
  pipe_depth(P) = $D\_P < \infty$
  such that between any two consecutive observable actions of P the scheduler inserts at most D_P clock cycles of ε-steps. (Commercial HLS tools emit finite scheduling tables, making this premise true in practice.)
- Weak-fairness premise (WF) If a micro-operation remains continuously enabled, the scheduler eventually issues it. Clock-synchronous RTL execution with either rendezvous channels or bounded FIFOs satisfies this property. (Weak fairness is an assumed guarantee of the execution/scheduling environment; as long as the designer avoids constructs that would defeat that guarantee, the burden of actually enforcing fairness rests with the environment, not with the model.)
- Finite-progress invariants For every channel c, producer and consumer obligations $P\_push(c)$ and $P\_pop(c)$ hold, guaranteeing that data (or space) eventually becomes available.
- No ill-formed signal IO Every signal read/write has a unique bounding synchronization operation in the source program; otherwise, the design is ill-formed. (From RULE 1 and RULE 2).

I.3 Auxiliary Lemmas

Lemma I.0 (Source-order issue discipline, from R4). For any two message operations m_1, m_2 issued by the same process on distinct channels, if $m_1 <\_src\ m_2$ then

issue_post(m_1) ≤ issue_post(m_2).

Proof. The scheduling rule R4 dictates that for messages on distinct channels, the issue of m1 must precede or be concurrent with the issue of m2 if m1 appears before m2 in the pre-HLS source code. HLS is required to honor this constraint in the post-HLS model. ■

Lemma I.1 (Bounded **ε**-chain). Starting from any state of P, the next observable action appears after at most D_P clock cycles. *Proof.* Direct from FPD. ■

Lemma I.2 (Eventual space / data). Assume P is blocked on • Push_c with occ_c = $B(c)$,
or • Pop_c with occ_c = 0. Then a complementary Pop_c (respectively Push_c) commits within finite time.

Proof. In both situations the guard of the complementary action is continuously true. By the invariants P_push / P_pop and weak fairness, that action must eventually fire, unblocking P.

I.4 Inductive Construction for Finite Traces

Let

$\tau\_pre = \tau\_pre[0..n-1] \circ e$ (where $|\tau\_pre| = n + 1$)

be the next pre-HLS prefix. Assume by induction that we already have a matching post-HLS prefix $\tau\_post[0..n-1]$ satisfying E1–E5. We extend it with a finite ε-chain (written ε^*) followed by an observable action e' so that

$\tau\_post \circ \varepsilon^* \circ e'$ matches τ_pre.

Kind of event *e*	Why ε^* is finite
Sync, wait, START_P, FINISH_P, Write, Read	Scheduler may idle for ≤ *D_P* cycles (Lemma I.1).
Push_c	• issue_post(e′) occurs in the cycle the guard becomes true; Lemma I.0 guarantees source-order issue discipline.• If *B(c) > 0* and occ_c < *B(c)*: the commit fires within ≤ *D_P* cycles.• If *B(c) > 0* and occ_c = *B(c)*: Lemma I.2 frees space.• If *B(c) = 0*: rendezvous fires when both sides are enabled; Lemma I.2 guarantees the peer.
Pop_c	Symmetric to Push_c.

Each case ensures ε^* terminates, so the extension preserves E1–E5; in particular, the issue-order clause of E3 is satisfied via Lemma I.0. ■

I.5 Extension to Infinite Traces

Because every finite prefix of a pre-HLS trace can be extended in bounded time and the construction is prefix-consistent, König's lemma on finitely branching trees yields an infinite post-HLS trace whose every finite prefix matches the corresponding pre-HLS prefix. Hence

$\tau\_pre \approx \tau\_post$ (for countably infinite traces as well).

Appendix J – System-Level Trace Equivalence Proof

Theorem J.1 (Compositional Equivalence)

Let every process P in the system satisfy $\tau\_pre,P \approx \tau\_post,P$ by Appendix I, and let every channel c have finite capacity $B(c) \geq 0$. Assume that for every channel c, the progress invariants P_push(c) and P_pop(c) hold, and that the system satisfies weak fairness (WF). Then the aggregate traces of the whole design are equivalent: $\tau\_pre,system \approx \tau\_post,system$ under rules E1-E5, given the R rules and B rules.

Notation: Let <_src denote the textual order of operations within a single process in the source code.

Proof

We first establish key invariants, then prove each equivalence property.

Lemma J.1 (Channel Occupancy Invariant)

For every channel c and at every clock cycle t in the post-HLS execution, the occupancy occ_post(c,t) satisfies $0 \leq occ\_post(c,t) \leq B(c)$.

Proof of Lemma J.1: RTL never de-queues from an empty FIFO and never en-queues into a full FIFO. A Push_c can only commit when occ(c) < B(c), enforced by the ready/valid handshake protocol. Similarly, Pop_c can only commit when occ(c) > 0. Since occ(c,0) = 0 and each operation atomically increments

(Push) or decrements (Pop) the occupancy by 1, the invariant $0 \le occ(c,t) \le B(c)$ is maintained inductively at every clock edge. □

Lemma J.2 (Inter-Process Link Ordering)

If Push_c in process P happens-before the corresponding Pop_c in process Q in the pre-HLS model, then clk_post(Push_c) ≤ clk_post(Pop_c) in the post-HLS model.

Proof of Lemma J.2:

- For B(c) = 0 (rendezvous): Push_c and Pop_c must occur simultaneously in both pre-HLS and post-HLS models, so clk_post(Push_c) = clk_post(Pop_c).
- For B(c) > 0: The FIFO semantics ensure that a Pop_c can only retrieve data that was previously pushed. By Lemma J.1, the channel respects its capacity bound at all times. Since the ready/valid handshake prevents Push_c from committing until space is available, and prevents Pop_c from committing until data is available, we have clk_post(Push_c) ≤ clk_post(Pop_c). □

We now prove each equivalence property:

E1 (Synchronization-order preservation):

Let s1 and s2 be any two synchronization events in the system with clk_pre(s1) < clk_pre(s2).

Case 1: If s1 and s2 are in the same process P, then by Appendix I's per-process equivalence, clk_post(s1) < clk_post(s2).

Case 2: If s1 ∈ P and s2 ∈ Q with P ≠ Q, and they are causally related through the happens-before relation, then there exists a chain of inter-process communications: s1 → a1 → a2 → ... → an → s2.

For each link in this chain:

- If the link is a message-passing operation (Push_c → Pop_c), Lemma J.2 guarantees temporal ordering is preserved
- If the link is a signal handshake (Write → Read), the signal propagates at clock edges per HDL semantics
- If the link is an explicit synchronization, all participating processes coordinate atomically

Since each link preserves temporal ordering and clock comparison is transitive, we obtain clk_post(s1) < clk_post(s2).

Case 3: If s1 and s2 are causally independent, E1 does not constrain their relative ordering.

E2 (Signal-visibility preservation):

For any signal operation r ∈ R or w ∈ W in any process P:

- By Appendix I, within process P: clk_post(r) = clk_post(pred_S(r)) and clk_post(w) = clk_post(succ_S(w))
- For inter-process signal visibility: In the pre-HLS model, a signal write occurs at clk_pre(succ_S(w)), and a signal read occurs at clk_pre(pred_S(r)). The post-HLS RTL preserves these semantics since R2 and E2 guarantee that a signal write occurs at clk_post(succ_S(w)), and a signal read occurs at clk_post(pred_S(r), so signal values propagate between processes exactly as in the pre-HLS model.

E3 (Safe message issue ordering on distinct channels):

For any two message operations m1 and m2 on distinct channels c1 ≠ c2 where m1 <_src m2 in the same process, we must show issue_post(m1) ≤ issue_post(m2).

This follows directly from Appendix I's per-process guarantee for message operations on distinct channels.

E4 (Per-channel FIFO semantics):

For each channel c, we must show:

1. The sequence of Push_c and Pop_c operations in τ_post forms a legal FIFO schedule
2. No messages are dropped or duplicated
3. FIFO order is preserved: if clk_pre(m1) < clk_pre(m2) for operations on the same channel, then clk_post(m1) < clk_post(m2)

From Appendix I, each process preserves the order of its operations on each channel. Combined with Lemma J.1 (occupancy bounds) and Lemma J.2 (Push happens-before Pop), the system-level execution maintains:

- One-to-one correspondence between pre-HLS and post-HLS Push/Pop operations
- FIFO ordering of all operations on each channel
- Respect for capacity constraints at every clock cycle

Therefore, the post-HLS execution represents a legal bounded-FIFO schedule.
E5 (Messages cannot cross syncs):
For any message operation m and synchronization call s in the same process P:

- If clk_pre(m) ≤ clk_pre(s), then clk_post(m) ≤ clk_post(s)
- If clk_pre(m) > clk_pre(s), then clk_post(m) > clk_post(s)

This property is guaranteed per-process by Appendix I and requires no inter-process reasoning, so it lifts directly to the system level.
Conclusion:
All five equivalence properties hold at the system level. The composition is valid because:

- Intra-process properties are preserved by Appendix I
- Inter-process communication respects capacity bounds (Lemma J.1) and ordering (Lemma J.2)
- The progress invariants and weak fairness ensure all operations eventually complete

Therefore, $\tau_{pre,system} \approx \tau_{post,system}$. □

Appendix K – Liveness Preservation for Buffered Implementations

---

K.1 Scope, Notation, and Definition of Deadlock
We work with a fixed design:

- System S: The collection of pre-HLS processes obeying the R-rules and B-rules. All message-passing channels in S are rendezvous channels (capacity B(c) = 0).
- Post-HLS implementation RTL(S): The hardware implementation in which each channel c has a finite capacity B(c) >= 0.

A global state sigma of either system consists of:

1. For each process P: a control location (program point) and local state.
2. For each channel c: its current occupancy occ(c) and contents (for buffered channels).
3. Any additional architectural state (pipeline registers, arbitration state, etc.).

We adopt the usual Wait-For Graph (WFG) abstraction:

- Vertices are processes.
- Edges: There is a directed edge P -> Q via channel c if, in state sigma, the next observable action of P is a blocking operation on channel c, and completion of that operation requires some action by Q on the same channel (e.g., producer waiting for consumer, consumer waiting for producer).

Non-blocking interfaces and purely internal actions do not contribute edges to the WFG; they are modeled as internal epsilon/tau steps.
We say that a process P is blocked on a channel operation in state sigma if:

1. The next operation in P's program order is a Push_c or Pop_c,
2. Its local guard is true, but
3. That operation cannot make progress because its channel-side enabling condition (data or space availability) is false.

A global deadlock is a reachable state sigma in which there exists a non-empty set of processes D such that:

- Every process P in D is blocked on some channel operation.
- The vertices in D, together with the channels they access, form a strongly connected component in the WFG that has no outgoing edges (a closed cycle): processes in D can only wait on each other.

Intuitively, D is a set of processes that are waiting only on each other and can never be unblocked by activity elsewhere in the system.

Our notion of liveness in this appendix is: Liveness = absence of global deadlock as defined above. *(Starvation of a single process in an otherwise live system is ruled out separately by weak fairness.)*

SyncChannels and other synchronization interfaces.

In this appendix we build the wait-for graph only from blocking message-passing operations (Push_c, Pop_c). Calls to synchronization interfaces (SystemC wait, ac_sync, Matchlib SyncChannel) are treated as pure synchronization events: under the weak-fairness and protocol-progress assumptions from Appendix I, once all participating processes have reached a synchronization call, the handshake completes in finite time. Any global deadlock that appears at a synchronization point must therefore arise because some participant is itself blocked earlier on a message-passing channel; in that case, the deadlock is already represented as a cycle of blocked Push/Pop operations in the WFG. For this reason, synchronization interfaces do not need separate nodes or edges in the WFG and do not alter the liveness argument in this appendix.

K.2 Assumptions and Imported Results

We assume throughout:

1. R-rules and B-rules: All processes and channels in both S and RTL(S) obey the semantic constraints specified earlier.
2. Finite Pipeline Depth (FPD): There is a global bound on the length of any chain of purely internal steps between observable actions.
3. Weak Fairness (WF, B2): If the guard for an enabled action remains continuously true, the scheduler eventually selects that action.
4. Channel Progress (B3): On each channel, if there is continuous demand for data/space and the complementary actions remain enabled, the protocol eventually completes a handshake.
5. Pre-HLS Liveness Assumption: The pre-HLS rendezvous system S (where B(c) = 0) is deadlock-free under the above assumptions.
6. Monotonicity of Blocking: If a process is blocked on a channel operation, it remains blocked until the channel-side enabling condition (data or space) changes.
7. Trace Equivalence Theorem (Appendix J): We rely on the corollary that for every finite post-HLS execution tau_post reaching state sigma_post, there exists a matching pre-HLS execution tau_pre reaching state sigma_pre such that:
    - Each process P is at the same abstract program point in both states.
    - The history of operations on every channel is identical in sequence.
    - We write sigma_pre == sigma_post to denote this correspondence.

K.3 Lemma K.1 — Characterization of Buffered Deadlock

Statement: Let sigma_post be a reachable global state of the buffered post-HLS system RTL(S). Suppose there is a non-empty set of processes D that is globally deadlocked. Then:

1. Every process P in D is blocked on a blocking channel operation (Push or Pop), not on an internal step.
2. If P is blocked on Push_c, then occ(c) = B(c) (channel is full).
3. If P is blocked on Pop_c, then occ(c) = 0 (channel is empty).

4. The processes in D form a closed strongly connected component in the WFG.

Proof:

- *Internal Steps:* If a process were blocked on an internal step with a true guard, Weak Fairness and Finite Pipeline Depth guarantee it would proceed. Thus, deadlocked processes must be blocked on channel operations.
- *Blocked Push:* If occ(c) < B(c), a Push with a true guard would succeed under Channel Progress (B3). Thus, for a Push to be blocked indefinitely, the channel must be full.
- *Blocked Pop:* If occ(c) > 0, a Pop with a true guard would succeed under B3. Thus, for a Pop to be blocked indefinitely, the channel must be empty.
- *Closed Cycle:* By definition of global deadlock, processes in D cannot wait on processes outside D, or else an external action could unblock them.

K.4 Lemma K.2 — Dependency Refinement

We now relate the blocking dependencies in the buffered system to those in the rendezvous system.

Statement: For any pair of corresponding states (sigma_pre, sigma_post) where sigma_pre == sigma_post: Every wait-for edge in the buffered WFG is also present in the rendezvous WFG. *(The rendezvous system may have additional edges, but it strictly includes all edges found in the buffered system.)*

Proof: Since sigma_pre == sigma_post, every process P is at the same program point (attempting the same next operation).

1. Consumer (Blocked Pop): Suppose P waits for Q via Pop_c in the buffered system. By Lemma K.1, the buffer is empty (occ = 0). P must wait for Q to Push. In the rendezvous system (B=0), a Pop also requires a simultaneous Push. Thus, P still waits for Q in the rendezvous system.
2. Producer (Blocked Push): Suppose P waits for Q via Push_c in the buffered system. By Lemma K.1, the buffer is full (occ = B). P must wait for Q to Pop to make space. In the rendezvous system (B=0), a Push *always* requires a simultaneous Pop (zero capacity is logically equivalent to a "full" buffer of size 0). Thus, P still waits for Q in the rendezvous system.

Since all blocking dependencies in the buffered system map to dependencies in the rendezvous system, the set of edges in WFG_post is a subset of WFG_pre.

K.5 Theorem K.1 — Liveness Preservation

Statement: If the pre-HLS rendezvous system S is deadlock-free, then the post-HLS implementation RTL(S) is also deadlock-free.

Proof (by Contradiction):

1. Assume RTL(S) reaches a deadlocked state sigma_post.
2. Extract Cycle: By Lemma K.1, there exists a set of processes D forming a closed cycle in WFG_post.
3. Map to Rendezvous: By Theorem J.1, there exists a corresponding rendezvous state sigma_pre.
4. Transfer Cycle: By Lemma K.2, every edge in the cycle in WFG_post also exists in WFG_pre. Therefore, the same set of processes D forms a closed wait-for cycle in WFG_pre.
5. Contradiction: This implies S is deadlocked in state sigma_pre. This contradicts the Pre-HLS Liveness Assumption.
6. Conclusion: RTL(S) cannot deadlock.

K.6 Summary and Interpretation

This appendix proves that under the standard fair scheduling assumptions:

1. Any deadlock in the buffered hardware must manifest as a cycle of processes waiting for full or empty channels (Lemma K.1).
2. Buffering relaxes constraints: it allows producers to run ahead, but it never *creates* a dependency that didn't exist in the unbuffered model (Lemma K.2).

3. Therefore, if the unbuffered (Pre-HLS) model is deadlock-free, the buffered (Post-HLS) hardware is guaranteed to be deadlock-free (Theorem K.1).

Implementation Note

Because the Appendix I, J, and K proofs above are parametric in $B(c) \geq 0$, designers are free to instantiate *any* finite depth—zero included—on every channel without jeopardizing functional correctness or liveness, provided the environment upholds the weak fairness assumption.

Appendix L – Snooping Introduction, Implementation and Concerns

Snooping Introduction

If designs are completely latency insensitive, verification of the pre-HLS versus post-HLS models is straightforward. However, if the design contains some components such as arbiters which have latency-sensitive behavior, and if that latency-sensitive behavior is in some cases externally visible to the DUT, then verification becomes somewhat more complex. Techniques can be applied to simplify verification.

Consider a design which has an arbiter like Matchlib toolkit example 09*. The arbiter uses non-blocking PopNB() on its inputs, and blocking Push() on its output to emit the winner of the arbitration. Since the latency between the pre-HLS and post-HLS designs will differ, the order of transactions presented to the arbiter in the two scenarios will differ, and thus the order of the winners will differ. This may result in verification mismatches between the two models if the order of the winners is externally visible to the DUT.

To force the pre-HLS and post-HLS simulations to match, we can run the two designs side-by-side. We can snoop the inputs to the arbiter in the post-HLS model, and only allow the inputs to the pre-HLS arbiter to proceed when the corresponding inputs are seen in the post-HLS model. This will force the order of the inputs to be equivalent between the pre-HLS and post-HLS models, and thus both arbiters will pick the same winners. When this technique is used, the pre-HLS simulation will be throttled by the post-HLS simulation, but the overall verification will still work properly.

Another related example involves interrupt request signals feeding into an interrupt controller within a CPU model. Typically, each interrupt request signal is a single bit sc_signal<>, indicating that an interrupt request is pending. If the requests originate from accelerator blocks that are being synthesized through HLS, then because of the latency differences between the pre-HLS and post-HLS models, the order of interrupt requests arriving at the interrupt controller will differ between the two models, and this will likely result in verification mismatches if the differences are externally visible to the DUT. To force the order of the requests to match, we snoop the requests arriving at the controller in the post-HLS model, and only then allow the requests to be seen in the pre-HLS model.

Simplified Snooping Approaches

In the snooping example directly above in which the arbiter is snooped, the entire post-HLS RTL DUT is simulated alongside the pre-HLS model to force the arbiter requests to be aligned in time. This is the most general case and assumes the input delays to the arbiter are difficult to determine via analysis.

Sometimes there are simpler cases where the input delays to the arbiter in the RTL DUT are easier to determine. For example, each input to the arbiter might arrive a fixed number of clock cycles after one of the primary inputs to the RTL DUT is pushed by the testbench. In this case, a much simpler model can be used to align the pre-HLS model with the post-HLS model. We can simply monitor the primary inputs to the pre-HLS model and then apply the fixed cycle delays to the pre-HLS arbiter inputs.

The two cases above illustrate two ends of a spectrum of possible approaches for extracting the delays from the RTL DUT. Between these two points there exist other possible approaches.

Snooping Implementation

To enable perfect matching between the pre-HLS and post-HLS system simulations, latency-sensitive global signals and latency-sensitive non-blocking message-passing operations need to be synchronized between the two simulations if their latency differences are externally visible to the DUT.

As explained earlier in this appendix, in the general case the entire post-HLS DUT RTL must be run alongside the pre-HLS system to achieve alignment. Examples of signals that need to be synchronized include global interrupt request signals (as discussed earlier in this appendix) and rdy/vld signal pairs for non-blocking operations on message-passing channels. All such synchronization signals and their corresponding handshake signals must be level-stable:

- A latency-sensitive signal s is level-stable if, once s becomes 1 in cycle t, it must remain 1 until the corresponding handshake signal h is 1 in some cycle $t' \geq t$.

Once the set of signals that need to be aligned between the two simulations is identified, the general rule to implement snooping is simple:

- For each of the pre-HLS and post-HLS simulations, make all readers of the signals see the logical AND of the values of each signal being driven in each separate simulation.

Often the post-HLS simulation will never run ahead of the pre-HLS simulation, because HLS typically only adds (rather than subtracts) latency within each process. This means that often the only signals that need to be delayed are on the pre-HLS side, and therefore the logical AND of the nets may not need to be driven on the post-HLS side. If this optimization technique is used, the logical AND of the nets should be compared with the actual value driven on the post-HLS side, and if they do not perfectly match then the optimization technique must not be used.

Snooping Concerns

The snooping technique is used to align pre-HLS and post-HLS latency-sensitive global signals and latency-sensitive non-blocking message-passing operations in cases where their latency differences are externally visible at the DUT boundary. When such a wrapper is applied, the pre-HLS model is throttled to follow the latency choices made by the post-HLS RTL, and the resulting traces at the observable interfaces are forced to agree.

Readers with a formal background may be concerned by this technique, because it appears to use the post-HLS RTL implementation to modify the behavior of the pre-HLS specification. From a formal point of view, the important observation is that the pre-HLS model is intentionally non-deterministic with respect to latency: subject to the R-rules, B-rules, weak fairness, and the happens-before relation on Σ, it admits a set of legal traces. Snooping does not change this set. Instead, it selects, for verification purposes, a particular legal pre-HLS trace whose latency-sensitive events align with those observed in the RTL. In other words, the implementation is used to pick one admissible execution of the specification, not to redefine what executions are allowed.

Experienced SoC architects tend to view this slightly differently, in terms of design tradeoffs and verification cost.

First, it is usually possible at the architectural level to avoid latency-sensitive behaviors entirely, for example by insisting that all communication be latency-insensitive and fully synchronized. However, the quality-of-results (QOR) costs of such designs may be unacceptable. Introducing latency-sensitive behavior—non-blocking arbiters, latency-sensitive global interrupt signals, and so on—is a deliberate choice made by the designer to meet performance, power, or area goals.

Second, in many practical systems, latency-sensitive behavior is not functionally observable at the DUT boundary under the equivalence relation ≈ defined in Appendix G. As an example, consider a DUT that contains a single-port RAM used to exchange data between two internal processes. An arbiter is required to control access to that RAM port, and the arbiter uses latency-sensitive non-blocking Pop operations on its request channels. During HLS, internal latencies will change, so the order in which the arbiter sees pending requests and chooses winners may differ between the pre-HLS and post-HLS models.

In this example:

- The individual RAM read/write operations and the internal arbitration decisions are not directly visible at the DUT interface.
- Only the higher-level IO of the two processes (for example, their message-passing interfaces to the rest of the system) is externally visible.

One might initially conclude that it is necessary to snoop the arbiter's inputs to make the pre-HLS and post-HLS traces match. However, the modeling rules impose two additional constraints:

1. Weak fairness for the arbiter. The arbiter must satisfy the weak fairness assumptions, so that any request that remains pending is eventually granted in both the pre-HLS and post-HLS models.
2. Happens-before discipline on shared memory. The system must enforce a happens-before relation on shared-memory accesses, ensuring that sufficient synchronization exists between the two processes so that there are no read/write races through the RAM in either model.

Under these conditions, the different arbitration orders merely change the relative timing of internal operations and of causally independent external events. They do not change the functional ordering of causally dependent observable actions at the DUT boundary. In the terminology of Appendix G, the differences are incidental variations in latency rather than changes to the happens-before partial order, and therefore they are not considered observable differences in behavior. In such cases, snooping is not required. (See also Note 1 below).

Experienced SoC architects therefore try to avoid designs in which latency-sensitive internal behaviors leak directly into the observable behavior of the system under test, since this both complicates verification and makes RTL behavior less predictable. When they do introduce such behaviors—examples include non-blocking arbiters whose winners affect externally visible ordering, global interrupt request signals at the DUT boundary—they typically know exactly where those latency-sensitive interfaces are and can isolate them.

A useful analogy is a subsystem whose clock frequency is adjusted dynamically based on on-die temperature. As temperature changes, the externally visible behavior of the SoC can change (for example, in terms of throughput or timing of events), and designers may choose such a temperature-dependent scheme to meet overall system requirements. To verify such a system, we may wish to compare a concrete RTL trace captured at a specific temperature profile with a higher-level reference model. To do so, the reference model must be driven with the same temperature evolution as the RTL saw. If that temperature profile is not otherwise under direct control, the simplest way to obtain it is to "snoop" the temperature as observed in the RTL trace and replay it into the reference model.
In this analogy, temperature is an external parameter of the environment rather than a fundamental part of the functional specification, but it still influences observable behavior. Snooping simply provides a mechanism to recover that parameter from the implementation when it cannot be easily prescribed a priori. Similarly, snooping of latency-sensitive IO provides a mechanism to supply implementation-specific latency choices—subject to the fairness and happens-before constraints—back to the pre-HLS specification so that the two models can be compared under a common environment.

Note 1: Non-blocking interfaces whose latency-sensitive behavior is not observable at the DUT boundary (in the sense of Appendix L) influence only the internal scheduling of observable actions. Under the fairness and progress assumptions B2–B3, such interfaces cannot introduce new deadlocks beyond those captured by the Appendix K Wait-For Graph over blocking Push/Pop operations, so the liveness preservation result of Appendix K applies unchanged whether or not these interfaces are snooped.
From a trace-equivalence standpoint, the same reasoning applies. Because these non-blocking operations affect only the relative timing and interleaving of causally independent actions, they do not change the happens-before relation on the observable action set Σ or on the subset of actions visible at the DUT boundary under the equivalence $\approx$ of Appendix G. In the terminology of that appendix, any differences are incidental variations in latency rather than changes to the partial order of causally dependent events, and therefore they are not considered observable differences in behavior. Consequently, the per-process and system-level trace equivalence theorems of Appendices I and J (E1–E5) continue to hold unchanged whether or not such non-blocking interfaces are snooped.

Stress Testing Pre-HLS Models for Latency Robustness

The formal equivalence results in Appendices G–K establish that, under the scheduling rules and fairness assumptions, the post-HLS RTL is trace-equivalent to the pre-HLS model at all observable interfaces. These results implicitly assume that the pre-HLS specification is *well-formed*: it must not rely on incidental, implementation-specific timing alignments for functional correctness. Designs that do rely on such incidental alignments fall outside the formal guarantees.

Designs that use non-blocking message-passing operations (e.g., PushNB/PopNB, ac_channel nb_read/nb_write) or latency-sensitive global signals are particularly vulnerable to this kind of fragility. For such designs, it is strongly recommended to subject the pre-HLS model to aggressive *latency stress tests* in which message and handshake latencies are varied across executions. The intent is to validate that the design behaves correctly across the range of weakly fair schedules permitted by the modeling rules, not just under one convenient execution order.

More concretely, verification should check that:

- Causal dependencies are explicit. All functionally significant "happens-before" relationships are enforced via message-passing, SyncChannel operations, or explicit handshake protocols, rather than by relying on the incidental execution order of concurrent processes. In the terminology of Appendix G, the design shall not depend on the ordering of causally independent events for correctness.
- Weak fairness does not hide latent deadlocks or starvation. The design continues to make progress under adversarial but *weakly fair* scheduling—i.e., enabled actions may be delayed arbitrarily long but not forever—consistent with the B2/B3 obligations on the scheduler and environment summarized in Appendix N.

If the pre-HLS model fails under such randomized-latency stress scenarios, then the specification itself is outside the formal model of this document: it violates the design rule that well-formed systems must not rely on the relative ordering of causally independent events. In that situation, the equivalence theorems still apply to well-formed traces, but they no longer guarantee that the "golden" specification is robust under the full range of latency variations allowed by the environment.

The Matchlib library provides practical mechanisms to automate these stress tests in pre-HLS simulation, including:

- Random stall injection on message-passing channels, to simulate variable communication delays while preserving rendezvous semantics.
- Latency and capacity back-annotation, to model specific per-channel buffer depths and delays, including the capacities B(c) chosen during HLS.

Worked examples of this methodology are provided in Catapult Matchlib examples 60* and 72*, which illustrate how to configure randomized stalls and back-annotated latencies when validating that a design remains well-formed and latency-robust.

Appendix M – Document Abstract

This document defines a precise user-level scheduling model for high-level synthesis (HLS) that enables system-level verification and debug to be carried out almost entirely on the pre-HLS SystemC model, while still permitting aggressive RTL optimizations in the synthesized design. It introduces a small set of uniform rules governing three classes of I/O operations—message-passing channels, signal I/O, and explicit synchronization—and organizes them into a "basic conceptual model" that preserves source-order synchronization, pins signal reads and writes to their nearest synchronization points, and constrains the reordering of message-passing operations so that HLS cannot introduce new deadlocks.

These rules are then extended to pipelined loops, shared and external memories (via an explicit array-access mapping layer and conflict-free reordering), and "direct input" pragmas that allow stable or periodically synchronized signals to bypass unnecessary internal storage while maintaining equivalence between pre- and post-HLS behavior.

The methodology is latency-insensitive by construction but accommodates latency-sensitive islands such as cycle-accurate transactors, non-blocking arbiters, and one-way handshake protocols through encapsulation and carefully specified synchronization schemes. The document also provides concrete modeling guidelines (e.g., rules for placing signal reads/writes around wait statements, coding of rolled loops with signal I/O, and use of Matchlib Connections and SyncChannel) that allow digital verification engineers to write a single testbench in SystemVerilog UVM or SystemC/C++ and reuse it across both pre-HLS and post-HLS models with "no surprises."

A major contribution is the formalization, in Appendix G and subsequent appendices, of a trace-equivalence relation between the pre-HLS "rendezvous" model and the post-HLS RTL, expressed as partial-order constraints on observable I/O actions and encapsulated in equivalence rules E1–E5.

These proofs are compositional (per process and system-level), incorporate weak fairness and bounded-FIFO assumptions, and cover key HLS transformations including loop pipelining, added FSM states, memory-access reordering, and configurable channel buffering. Collectively, the scheduling rules, coding guidelines, and formal guarantees provide a tool-agnostic, Catapult-compatible foundation for industrial HLS use and a concrete starting point for standardization efforts within bodies such as the Accellera Synthesis Working Group.

Keywords:
High-level synthesis (HLS); SystemC; scheduling rules; latency-insensitive design; message-passing channels; signal I/O; loop pipelining; direct input pragmas; shared memory; trace equivalence; formal verification; bounded FIFOs; Catapult HLS; Matchlib; Accellera Synthesis Working Group.

Appendix N - Scheduler and Environment Fairness Assumptions

The formal results in Appendices I–K rely on several semantic assumptions about the post-HLS scheduler and its environment:

- B2 (Weak fairness of the scheduler). If the enabling predicate for an action (Push, Pop, Sync) remains continuously true from some cycle onward, the scheduler must eventually select that action within a finite, but unspecified, number of cycles.
- B3 (Channel progress invariants). For every channel c with capacity B(c), the producer and consumer satisfy:
    - P_push(c): If Push_c remains enabled while the channel is full (occ_c = B(c)), then some Pop_c eventually occurs.
    - P_pop(c): If Pop_c remains enabled while the channel is empty (occ_c = 0), then some Push_c eventually occurs.
      These obligations rule out "permanent back-pressure loops" that are logically independent of the local R-rules.

- B5 (System quiescence closure). If at some cycle no observable actions are enabled in any process, then within a bounded number of cycles the system reaches a fixed point and executes no further ε-steps. This prevents a dead, all-disabled state from being hidden behind an infinite tail of internal activity.

These B-rules are assumptions on the execution environment, not guarantees automatically provided by the HLS tool. In practice they place concrete requirements on arbiters, back-pressure, and external masters that interact with the post-HLS RTL.

Priority arbiter example: fair vs unfair priority

Consider a simple two-input priority arbiter inside the post-HLS RTL. Each input is driven by a message-passing channel; the arbiter issues Pop operations to select which request to serve in each cycle.

- Input 0: high-priority channel c_high
- Input 1: low-priority channel c_low

Both channels may have pending requests at the same time.

1. Fair priority arbiter (satisfies B2 / B3).

A fair implementation might still give c_high strict priority in *individual* decisions, but it ensures that a continuously pending c_low request cannot starve. In RTL terms, this can be realized by, for example:

- A round-robin or rotating-priority arbiter that periodically moves the "highest" priority to the next requester, or
- A bounded-burst priority scheme in which c_high may win at most N consecutive cycles while c_low is requesting; after that, the next grant is forced to c_low.

Under these implementations:

- If Pop_low's enable remains true indefinitely (the low-priority channel has a pending request and downstream space is available), the scheduler must eventually grant Pop_low. This satisfies B2 for that action.
- If the low-priority FIFO is full and keeps requesting service, the system ensures that some Pop_low eventually fires, discharging data and freeing space. This is consistent with P_push/P_pop in B3.

Intuitively, the arbiter is still "priority-based," but its micro-architecture ensures that every continuously enabled requester is eventually served.

2. Unfair priority arbiter (violates B2 / B3).

A more naive design might implement:

```
if (req_high) grant_high();
else if (req_low) grant_low();
```

combined with an environment in which req_high can remain asserted indefinitely. In this case, even if req_low is also continuously asserted:

- Pop_low's enabling predicate is true forever, but it is never chosen by the scheduler.
- Low-priority requests can starve indefinitely, even though they are logically ready to fire.

This behavior violates B2 (weak fairness of the scheduler). If c_low's FIFO is full and the only way to make space is to service it, permanent starvation also violates B3's progress expectations for that channel.

In such a design, the safety properties E1–E5 may still hold (trace-equivalence on actions that *do* occur), but the liveness/results that depend on B2/B3—especially the starvation-vs-deadlock arguments in Appendix K—no longer apply.

Patterns that typically satisfy the fairness assumptions

The following design patterns normally satisfy B2 and are compatible with B3/B5, provided the rest of the system is well-behaved:

- Round-robin or rotating-priority arbiters. Any requester that remains enabled will eventually be at the front of the rotation and will be granted.
- Age-based or credit-based arbiters. Requesters accumulate age or credits while waiting; the arbiter prefers older or more-starved requests, guaranteeing that long-pending actions eventually win.
- Bounded-burst fixed priority. Fixed priority is combined with a counter that limits how long a higher-priority requester can dominate while lower-priority requesters are pending. After the burst limit is reached, lower-priority requests are forced to win until the system is "caught up."
- Back-pressure with bounded stall. When ready/valid or similar handshake signals are used, the design (or its environment) must enforce that ready cannot remain low forever while valid remains high, and vice versa. For example:
    - Downstream consumers are required (by specification) to service queues at least once every N cycles when data is present.
    - External bus masters are configured such that they cannot indefinitely defer reading from full DUT FIFOs that are logically part of the interface contract.
- Clock-gating and power-management schemes that respect B5. Once no observable actions are enabled, the design either:
    - Quietly stabilizes (no further internal toggling), or
    - Explicitly enters a low-power state from which it only wakes when some observable action again becomes enabled.

In each case, the intent is the same: continuous enable implies eventual service, and once nothing is enabled, the system converges rather than oscillating internally.

Patterns that tend to violate the fairness assumptions

Conversely, the following patterns are likely to break B2/B3/B5 and therefore fall outside the scope of the liveness arguments in this document:

- Pure fixed-priority arbitration with unbounded high-priority traffic. A static priority tree with no rotation or aging, combined with workloads in which a high-priority master can keep its request asserted indefinitely, can starve lower-priority channels forever.
- "Best-effort" external masters with no progress guarantee. For example:
    - A software driver that reads from a DUT output FIFO only when it happens to poll, and that may be pre-empted indefinitely by higher-priority threads.
    - An external bus or DMA engine that is architecturally allowed to ignore some requesters forever if higher-priority traffic remains heavy.
      Such environments can violate P_push/P_pop by allowing FIFOs to remain full or empty indefinitely while the DUT is continuously requesting progress.
- Circular back-pressure loops with no escape. Two or more processes form a cycle in which each is waiting for the other's channel to change state, and no other process can break the cycle. Without an explicit design-level guarantee that some process in the loop will eventually act (for example, by dropping priority or issuing a compensating Pop/Push), B3 is not satisfied.
- Internal oscillation without quiescence. A scheduler or control FSM that can toggle internal ε-level state forever, even after all observable actions are disabled, violates B5. While such designs are uncommon in practice, patterns involving mis-configured clock gating, asynchronous feedback, or "keep-alive" timers that never expire can have this effect.

In all these cases, the trace-equivalence theorems still describe what happens when actions do occur, but the liveness claims that depend on B2/B3/B5 (for example, “a continuously enabled channel will not starve”) are no longer guaranteed. Designers and verification engineers should therefore either:

- Architect their arbiters and environments to satisfy these B-rules, or
- Treat the liveness guarantees in Appendix K as *out of scope* for those particular interfaces, and rely only on the safety-oriented parts of the model.

Appendix O - Support for Multiple Clock Domains

To lift the single-clock formalism of Appendices G–K to a multi-clock setting, we model each clock domain *d* as its own synchronous island with a local cycle counter *clk_d* and apply the existing R- and E-rules unchanged to processes within a fixed domain, interpreting *clk(.)* as *clk_d(.)* for all local synchronization, signal I/O, and message-passing events. All inter-domain communication is required to go through explicit clock-domain-crossing FIFOs; at the abstraction level of Appendix G, each such CDC FIFO is just another bounded channel with capacity *B(c)* and standard ready/valid semantics, so rendezvous pre-HLS channels and buffered post-HLS channels still satisfy FIFO legality (E4), occupancy invariants, and progress obligations *P_push(c)* and *P_pop(c)*, independent of the relative phasing of the two clocks. System behavior and liveness are then expressed in terms of the existing happens-before partial order over observable actions: intra-domain edges are ordered by the local *clk_d*, while each successful Push/Pop pair on a CDC FIFO induces a cross-domain happens-before edge. The weak-fairness and progress assumptions (B2/B3) are strengthened to require fairness of each local scheduler and of each CDC synchronizer, and Appendix K’s wait-for-graph and occupancy arguments are applied to this partial order rather than to a single total clock order, so the deadlock-preservation and trace-equivalence properties (E1–E5) continue to hold provided that no raw signals cross clock domains and every cross-domain path uses a CDC FIFO whose implementation is metastability-safe but otherwise abstracted as an ordinary bounded channel.